Fakulta informatiky
Masarykovy univerzity

Cvičení k předmětu IB002 Algoritmy a datové struktury I

Daliborek vzkazuje: Algoritmus je parciální funkce z domény tvořené kartézským součinem domén typů parametrů do domény typu výsledku, případně kartézského součinu těchto domén, která je vyčíslitelná v rámci stanoveného formalizmu.

poslední modifikace 15. května 2015

Tato sbírka byla vytvořena z příkladů k procvičení v předmětu *IB002 Algoritmy a datové struktury I.* K vytvoření sbírky přispívají cvičící předmětu IB002 od roku 2007, kdy jej ještě přednášel Libor Škarvada. Aktuální verzi spravují Ivana Černá, Karel Kubíček, Henrich Lauko a Vojtěch Řehák. Opravu faktických chyb provádí Filip Opálený, gramatické chyby hlásí Agáta Dařbujanová.

Obsah

Kapitola 1

Spojovaný seznam, fronta a zásobník

Datová struktura je náš vlastní datový typ, který je definován rozsahem hodnot, kterých může nabývat. Je nezávislý na konkrétní implementaci.

Statické datové struktury jsou datové struktury s pevnou velikostí. Příkladem je například uspořádaná k-tice a pole konstantní délky.

Dynamické datové struktury jsou datové struktury, jejichž velikost není před během programu známa, a tedy musí být alokována podle průběhu algoritmu.

Operace nad dynamickými datovými strukturami jsou operace, kterými můžeme modifikovat nebo jinak využívat obsah datové struktury.

1. SEARCH vyhledává v datové struktuře prvek s daným klíčem.
2. INSERT vkládá prvek do datové struktury.
3. DELETE maže prvek datové struktury.
4. MAXIMUM vrací ukazatel na maximální prvek z datové struktury.
5. MINIMUM vrací ukazatel na minimální prvek z datové struktury.
6. SUCCESOR vrací ukazatel na následující prvek (podle uspořádání).
7. PREDECESSOR vrací ukazatel na předcházející prvek (podle uspořádání).

Toto cvičení je zaměřeno na opakování znalostí z Úvodů do programování, které jsou v Algoritmech nutností. V roce 2014 nám hlavně Céčkaři naprosto nezvládali práci s pamětí. U Pythonistů zase byl problém s používáním tříd coby jednoduchých struktur.

Na cvičení se programují základní datové struktury – zásobník, fronta a seznam, které budou studenti v budoucnu potřebovat (a C je nemá ve standardní knihovně).

Cvičení bude výrazně specifické podle jazyka vaší skupiny. Jak dlouho se jednotlivým položkám budete věnovat, závisí na vaší skupině a studentech.

Také studentům řekněte, že další takové příklady k zamyšlení (které se můžou vyskytnout na implementačním testu a zkoušce) jsou ve sbírce na konci každé kapitoly. Jsou neřešené, kdyby měli dotazy, mohou se obracet na vás a nebo se zeptat na diskuzím fóru. Pokud máte dostatek času, můžete některé z „pokročilých příkladů“ probrat již na hodině.

V cvičení nedoporučujeme programovat skrz celé cvičení, ale jenom na konci až po probrání teorie.

Studenti najdou podklady ve studijních materiálech (dá se do nich dostat z interaktivní osnovy). Neočekává se, že studenti stihnou všechny struktury naprogramovat, je to pro ně práce na doma.

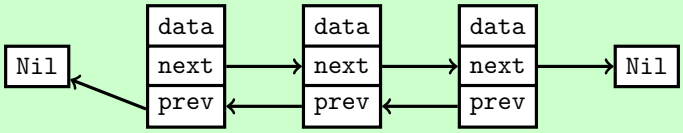
Toto cvičení je zaměřeno na opakování znalostí z kurzů IB001 a IB111. Protože zde opakujeme znalosti konkrétního jazyka, budeme místo pseudokódu psát kód v konkrétním jazyku. V budoucnu už budeme psát algoritmy pouze v pseudokódu (který byste měli bez problémů umět přepsat do vašeho jazyka). Kód v programovacím jazyce od pseudokódu odlišujeme pomocí neproporcionálního písma.

1.1 Procvičte si práci se zřetězeným seznamem:

Zřetězený seznam: ideálně nechte na studentech, aby si jej vymysleli. Takže tužka + papír (popřípadě někdo totéž na tabuli).

Z minulých let se osvědčilo vysvětlovat implementaci postupně nejprve pomocí čtverečků a šipek na tabuli a až pak částečný pseudokód (nedoporučujeme psát úplný pseudokód – zabere to hodně času, lepší jenom v náznacích nastínit postup).

Velmi nápomocné při práci s dynamickými datovými strukturami jsou obrázky. Takto si můžete představit oboustranně zřetězený seznam. Doporučujeme si jej překreslit nejlépe tužkou a provést přidání a odebrání prvku.



a) Navrhněte strukturu oboustranně zřetězeného seznamu a prvku seznamu.

> Náš seznam typu *List* bude obsahovat ukazatele *first* a *last* ukazující na první a poslední prvek v seznamu. Pokud je seznam prázdný, jsou tyto ukazatele inicializovány na *nil*. V případě jediného prvku v seznamu jsou ukazatele na první a poslední prvek shodné.
>
> Každý prvek seznamu bude obsahovat klíč *key*, ukazatel na následující prvek *next* a ukazatel na předchozí prvek *prev*. Pokud je prvek poslední, pak hodnota v *next* odpovídá *nil*. Obdobně je pro první prvek $prev = nil$. Nový prvek seznamu vytvoříme pomocí New(*key*) a tato funkce nám vrátí ukazatel na nový prvek. Prvek seznamu může také obsahovat prvek *data*, které reprezentují data přiřazená k danému klíči. Pro jednoduchost implementace se daty nemusíme zabývat.

b) K dané struktuře seznamu navrhněte funkci vložení prvku – Insert(l, key). Výstupem bude ukazatel na nově přidaný prvek s klíčem *key*.

Při práci s dynamickými strukturami dávejte pozor, abyste na prvek struktury neztratili ukazatel.

Funkce Insert vytvoří z klíče nový prvek a správně aktualizuje ukazatele v seznamu.

Procedura Insert(L, key)

vstup: seznam L typu ň*mathitList*, vkládaný klíč *key*
výstup: ukazatel na nově přidaný prvek

```
1  new ←New (key) // vytvoří nový prvek seznamu
2  new.next ← nil // následující prvek není
3  new.prev ← L.last // předchozí prvek je bývalý poslední prvek
4  if L.first = nil then
5      L.first ← new // případ prázdného seznamu
6  else
7      L.last.next ← new
8  fi
9  L.last ← new // nový poslední prvek
10 return new
```

c) Navrhněte metodu odstranění prvku ze seznamu Delete ($L, node$). Odstraňovaný prvek je zadaný ukazatelem *node*. Proč je odstraňování efektivnější u spojovaného seznamu než u pole?

Funkce Delete ($L, node$) smaže prvek ze seznamu a upraví ukazatele sousedních prvků.

Procedura Delete($L, node$)

vstup: seznam L typu *List*, ukazatel *node* na prvek seznamu, který chceme odstranit

```
1  if node = nil then
2      return Chyba, byl zadán prázdný ukazatel
3  fi
4  if node.prev = nil then
5      L.first ← node.next // nemá předchůdce
6  else
7      node.prev.next ← node.next
8  fi
9  if node.next = nil then
10     L.last ← node.prev // nemá následníka
11 else
12     node.next.prev ← node.prev
13 fi
14 Release(node) // až po úpravě ukazatelů uvolňujeme prvek z paměti
```

Výhoda oproti obyčejnému poli je, že nám stačí upravit jenom sousední prvky, abychom zachovali strukturu seznamu. V poli bychom museli posunout v paměti všechny prvky následující za smazaným prvkem. Proto se v aplikacích, kde se často mění prostřední prvky dat, používají seznamy místo polí.

d) Navrhněte vyhledání prvku s konkrétním klíčem v seznamu – Search (l, key). Funkce vrací ukazatel na nalezený prvek, nebo *nil*, pokud se prvek v seznamu nenachází.

Funkce bude muset projít lineárně celý seznam, dokud nenajde hledaný prvek.

Procedura SEARCH(L, key)

vstup: seznam L typu *List*, hledaný klíč *key*

výstup: ukazatel na nalezený prvek s daným klíčem; *nil*, pokud neexistuje

```
1 node ← L.first
2 while node ≠ nil ∧ node.key ≠ key do
3     node ← node.next
4 od
5 return node
```

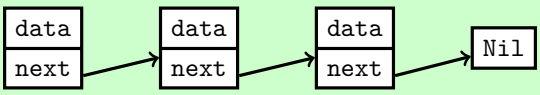
e) Porovnejte přístup k i-tému prvku ve spojovaném seznamu a v poli.

Jelikož prvky v poli jsou v paměti uspořádány za sebou, můžeme přímo přistupovat (s konstantní složitostí) ke konkrétnímu prvku. V seznamu to nelze, protože nevíme, kde se přesně i-tý prvek nachází. Proto přístup k i-tému prvku může být až lineární vzhledem k délce seznamu (musíme projít celý seznam, abychom prvek našli).

f) Jaké výhody nám poskytuje oboustranně spojovaný seznam oproti jednosměrně spojovanému seznamu?

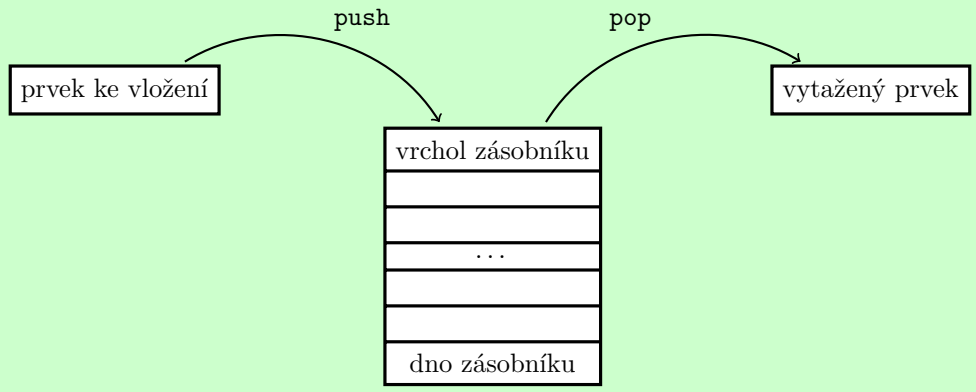
U některých aplikací se můžeme chtít odvolávat na předchozí prvky. Jak jsme už v předchozím příkladu zjistili, přístup k $(i-1)$-tému prvku by mohl mít až lineární složitost v jednosměrně spojovaném seznamu. V obousměrně spojovaném seznamu se však z i-tého prvku na předchozí prvek dostaneme v konstantním čase pomocí ukazatele *prev*. Toto vylepšení nám ovšem klade nároky na paměť pro udržování dalšího ukazatele.

Jednostranně zřetězený seznam si můžeme vizualizovat následovně:



1.2 Vytvořte zásobník na základě jednostranně spojovaného seznamu. Implementujte na něm operace PUSH pro vložení a POP pro odstranění vrcholu zásobníku.

Obrázek pro zásobník – LIFO (Last In First Out). Jaké kde mají být ukazatele si doplňte sami.



Pan Usměvavý dodává: Fronta na stojato je zásobník, kterému prasklo dno.

Mějme strukturu *Stack*, která obsahuje ukazatel na vrchol zásobníku *top*. Pokud je zásobník prázdný, je ukazatel *top* nastaven na *nil*.

Procedura Push(*stack*, *key*)

vstup: struktura *stack*, klíč *key*

```
1 added ←New (key)
2 added.below ← stack.top
3 stack.top ← added
```

Procedura Pop(*stack*)

vstup: struktura *stack*
výstup: hodnota odstraněného vrcholu

```
1 if stack je prázdný then
2     return zásobník je prázdný
3 fi
4 key ← stack.top.key
5 tmp ← stack.top // abychom neztratili ukazatele na prvek, který se má smazat
6 stack.top ← stack.top.below
7 Delete (tmp)
8 return key
```

1.3 Mějme zásobník a na něm sekvenci příkazů Pop a Push. Push vkládá popořadě hodnoty od 0 po 9, Pop vypíše odebranou hodnotu. Které z následujících sekvencí čísel nemůžou nastat?

Příkladu je mnoho, proto se studenty doporučujeme projít jenom pár podpříkladů.

a) 4 3 2 1 0 9 8 7 6 5

b) 4 6 8 7 5 3 2 9 0 1

c) 2 5 6 7 4 8 9 3 1 0

d) 4 3 2 1 0 5 6 7 8 9

e) 1 2 3 4 5 6 9 8 7 0

f) 0 4 6 5 3 8 1 7 2 9

g) 1 4 7 9 8 6 5 3 0 2

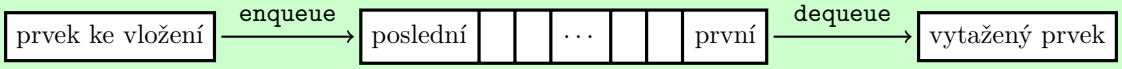
h) 2 1 4 3 6 5 8 7 9 0

Na rozdíl od fronty se mohou u zásobníku prvky přeházet. Prvky zachovávají vlastnost, že pokud již byly vloženy vyšší prvky, můžou být odebrány až po odebrání vyšších čísel. Nemůžou tedy nastat situace b) – problém je s poslední dvojicí čísel, f) – problém u 1 7 2 a g) – zase poslední dvojice.

Vzorově ještě rozeberme případ c). Posloupnost příkazů je Push(0), Push(1), Push(2), Pop 2, Push(3), Push(4), Push(5), Pop 5, Push(6), Pop 6, Push(7), Pop 7, Pop 4, Push(8), Pop 8, Push(9), Pop 9, Pop 3, Pop 1, Pop 0.

1.4 Vytvořte frontu na základě jednostranně spojovaného seznamu. Implementujte na ní operace Enqueue pro vkládání a Dequeue pro odstranění prvku.

Obrázek pro frontu – FIFO (First In First Out). Jaké kde mají být ukazatele si doplňte sami.



Pan Usměvavý dodává: Fronta jde zleva doprava, tak jak píšeme, to je přeci jasné. Ale když pak do fronty dáváme slova, tak jsou ve frontě uloženy v opačném pořadí a to nejde číst. Takže fronta musí jít zprava doleva. A ve frontě zprava doleva dáváme ukazatel `next` jako šipku směrem doleva. Ne, `next` musí jít směrem doprava.
Raději nadefinujeme frontu, která používá ukazatele `left`, tím pádem musí jít zleva doprava, tak jako píšeme. Nebo je to naopak?

Mějme strukturu *Queue*, která obsahuje ukazatel na první (*first*) a poslední (*last*) prvek fronty.

Procedura Enqueue(*queue*, *key*)

vstup: struktura *queue*, klíč *key*

```
1 added ←New (key)
2 added.left ← nil
3 if queue.last = nil then
4     queue.first ← added
5 else
6     queue.last.left ← added
7 fi
8 queue.last ← added
```

Procedura Dequeue(*queue*)

vstup: struktura *queue*
výstup: hodnota odstraněného prvku

```
1  if queue.first = nil then
2      return fronta je prázdná
3  fi
4  key ← queue.first.key
5  tmp ← queue.first // abychom neztratili ukazatele na prvek, který se má smazat
6  if queue.first = queue.last then
7      queue.first ← nil
8      queue.last ← nil
9  else
10     queue.first ← queue.first.left
11 fi
12 Delete (tmp)
13 return key
```

1.5 Mějme frontu a na ní sekvenci příkazů Dequeue a Enqueue. Enqueue vkládá popořadě hodnoty od 0 po 9, Dequeue vypíše odebranou hodnotu. Které z následujících sekvencí čísel nemůžou nastat?

a) 4 6 8 7 5 3 2 9 0 1

b) 0 1 2 3 4 5 6 7 8 9

c) 2 5 6 7 4 8 9 3 1 0

Jelikož je fronta datová struktura typu FIFO a prvky dáváme v pořadí od 0, pak jediný případ, který může nastat je, že tyto prvky budeme ve stejném pořadí i odebírat. Tedy případy a) a c) nikdy nenastanou.

1.6 Jak byste implementovali frontu jen za pomocí dvou zásobníků?

1.7 Naučte se pracovat s ukazateli a složenými typy:

Zopakování základních pojmů na cvičení nemusíte probírat, příklad slouží hlavně jako pomůcka při programování. Pokud budou vaši studenti ztraceni můžete zopakovat práci s pamětí: v C zopakovat pojmy ukazatel, statická a dynamická alokace paměti – malloc, struktura.

V Pythonu to bude trochu snazší, pojmy k zopakování jsou reference a třída. Třídu probírejte jen jako složený datový typ, tedy jako strukturu v C. OOP po studentech nechceme. Pokud studenti OOP umí, nebráníme jim, ale zbylé studenty neučíme OOP.

Cvičícím spíš doporučujeme ukázat konkrétní práci s implementačními zadáními a strukturami v nich.

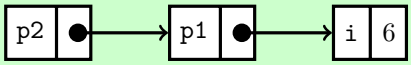
V jazyce C při práci s ukazateli používáme operátory reference a dereference.

Výraz v C	význam
`int* a`	proměnná `a` je ukazatelem, tj. adresou místa v paměti, kde se nachází proměnná/hodnota typu integer
`int b = *a`	do proměnné `b` ukládáme dereferenci `a`, tj. hodnotu uloženou na adrese `a`
`*a = b`	do dereference `a`, tj. na místo s adresou `a`, ukládáme hodnotu `b`
`a = &b`	do proměnné `a` ukládáme referenci na `b`, tj. adresu proměnné `b`

Všimněte si, že dereference lze číst i do ní ukládat, ale reference lze jen číst. Zamyslete se proč.

```
int i = 3;
int* p1 = &i;
int** p2 = &p1;

i = *p1 + i;
```



V C budeme pro složené typy používat struktury. Mějme proměnnou `a` typu `Struktura` a proměnnou `b`, která je typu ukazatel na místo v paměti s daty typu `Struktura`, tj. `Struktura a` a `Struktura* b`.

Výraz v C	Význam

`typedef struct Node {`
`  int key;`
`  struct Node* next;`
`} Node_t;`
	definice typu `struct Node`, který je strukturou s atributy `key` a `next`
	`key` je typu integer
	`next` je typu ukazatel na `struct Node`
	zde si zavádíme alias `Node_t` pro `struct Node`

`a.key`	přístup k atributu `key` proměnné `a`
`(*b).key` `b->key`	přístup k atributu `key` struktury referencované ukazatelem `b`, kromě dereference lze použít i přehlednější operátor `->`

V Pythonu místo struktur používáme třídy. Třída je zobecnění struktury, která navíc umožňuje vytvářet funkce uvnitř třídy (pak jim říkáme metody). V tomto předmětu se však neočekává znalost objektově orientovaného programování a třídy budeme v Pythonu potřebovat pouze ve smyslu složeného datového typu (avšak těm, co OOP ovládají, nebráníme metody používat).

Výraz v Pythonu	Význam

`class Node:`
`  key`
`  next`
	definice typu `Node`
	typ atributu `key` je určen dynamicky podle toho, co do něj přiřazujeme
	obdobně u atributu `next`, který chceme typu `Node`

`a.key`	přístup k atributu `key` proměnné `a`

a) Vytvořte vlastní složený datový typ, který reprezentuje osobu. K osobě je potřeba uchovávat jméno a věk.

Vhodným datovým typem může být například následující struktura:

Struktura Osoba
věk //celé kladné číslo *jméno //řetězec*

b) Přidejte do vašeho typu pole/seznam (C/Python) přátel dané osoby. Jakou formou je vhodné přátele ukládat?

Rozhodně není vhodné u každé osoby dělat kopie objektu pro každého přítele. Kdybychom totiž chtěli aktualizovat nějaké údaje, tak bychom aktualizaci museli provést nejen u originálního objektu, ale následně bychom museli vyhledat i všechny kopie, které by bylo třeba také aktualizovat. Dalším problémem je velká paměťová náročnost. Ideální je tedy na každého přítele nějak odkázat, k čemuž se hodí v C ukazatel a v Pythonu odkaz.
Naše struktura tedy výsledně vypadá takto:

Struktura Osoba
věk //celé kladné číslo *jméno //řetězec* *přátelé //seznam/pole odkazů/ukazatelů na proměnné typu* Osoba

c) Napište funkci Swap, která jako vstup přijímá 2 proměnné typu Osoba a prohodí jejich obsah. Jakého typu musí být proměnné, aby šel jejich obsah prohodit?

*Pokročilý úkol pro ty, kterým to přišlo jednoduché: zvládli byste napsat funkci Swap pomocí matematických operací tak, aby nebylo potřeba využít další paměť (na proměnnou *tmp* v C popřípadě dvojici v Pythonu)?

Funkci Swap by měli všichni znát, proto ji nedoporučujeme probírat hromadně.

Pro správné řešení tohoto příkladu potřebujeme funkci předat ukazatele (C), respektive odkaz (Python) na zadané proměnné (tedy informaci kde se proměnné nachází, ne přímo jejich hodnoty).
V Pythonu se vlastní datové typy předávají odkazem automaticky, odkazem se nepředávají jen primitivní datové typy (typicky čísla) – ty se předávají hodnotou. Řešení v Pythonu vypadá takto:

Procedura Swap(a, b)
vstup: odkazy na proměnné a a b typu Osoba
1 $a, b = b, a$ *// napravo vznikne uspořádaná dvojice (b, a), která se rozbalí do levé dvojice (a, b)*

Jelikož Python je dynamicky typovaný jazyk, tedy typ proměnné nepíšete do kódu, ale typ se odvozuje za běhu programu, tak tato verze funkce Swap funguje na libovolné 2 proměnné, které jsou funkci předávány odkazem.

Na primitivních datových typech by vám funkce nespadla, ale jelikož by se prohodil obsah lokálních proměnných, tak by neměla globální důsledek.

V C můžeme vytvořit ukazatel na jakoukoli proměnnou. Funkce Swap tedy bude vypadat v C takto:

Procedura Swap(Osoba* a, Osoba* b)
vstup: ukazatele na proměnné a a b typu Osoba*
1 Osoba $tmp = *a$ 2 $*a = *b$ 3 $*b = tmp$

Pro swap bez pomocné proměnné je možné použít operaci bitový *xor* – $\oplus$. Swap se provede následovně:

Procedura XorSwap(x, y)

vstup: bitové řetězce x a y

1 $x \leftarrow x \oplus y$
2 $y \leftarrow x \oplus y$
3 $x \leftarrow x \oplus y$

1.8 Naprogramujte si pořádně včetně všech potřebných operací seznam, frontu a zásobník. Budete je ještě v tomto předmětu potřebovat. Podklady pro C a Python najdete v studijních materiálech.

Pan Usměvavý dodává: Zkratky některých datových struktur:
LIFO (Last In, First Out) – také známo jako zásobník.
FIFO (First In, First Out) – také známo jako fronta.
FISH (First In, Still Here) – také známo jako „hung printer“.
FIGL (First In, Got Lost) – také známo jako byrokracie.
LIGL (Last In, Got Lost) – také známo jako „streams module“.
AIFO (All In, First Out) – také znám jako začátečník v pokeru.
FIGO (First In, Garbage Out) – také známo jako generátor náhodných čísel.
GHNW (Got Here, Now What?) – také známo jako „longjmp botch“.

Následující příklady jsou vhodné na domácí studium.

1.9 Mějme následující funkce:

- Head(n, A), která vrátí seznam prvních n prvků seznamu A.
- Tail(n, A), která vrátí seznam posledních n prvků seznamu A.

Pomocí těchto funkcí navrhněte funkci Interval(a, b, A), která vrátí seznam prvků ze seznamu A na pozicích v intervalu od a po b.

1.10 Implementujte funkci Delete, která odstraní první výskyt zadaného prvku:

- ze zásobníku s použitím pouze funkcí Push a Pop za použití pomocného zásobníku.
- z fronty s použitím pouze funkcí Enqueue a Dequeue za použití pomocné fronty.

Následující příklady jsou vytvořené ze starších implementačních testů. Mají sloužit jako bonusový materiál pro domácí studium nebo pro případ, že se již na cvičení všechno probralo.

Karlík varuje: Při řešení úloh s dynamickými datovými strukturami si dávejte pozor na přistupování a kontrolu k NULL/None prvkům.

1.11 Implementujte následující modifikace lineárního seznamu:

- Formulujte funkci, která obrátí pořadí prvků v jednostranně zřetězeném lineárním seznamu. Tedy začátek bude nový konec a konec bude nový začátek.
- Modifikujte oboustranně zřetězený lineární seznam tak, že ukazatele „dopředu“ budou ukazovat ob jeden prvek dál. Zpětné ukazatele zůstanou zachovány.

1.12 Mějme definovaný cyklický seznam jako seznam, jehož *začátek* = *konec*. Implementujte nad takovýmto seznamem následující funkce:

- IsCircular, která ověří, zdali je zadaný seznam opravdu cyklický,
- GetLength, která vrací počet prvků v seznamu,
- CalculateOpposite, která k jednotlivým prvkům seznamu najde protější prvek v kruhovém seznamu a vytvoří na něj ukazatel (řešte jen pro případ kruhového seznamu sudé délky).

1.13 Implementujte modifikovaný zásobník, jehož metody Push a Pop fungují následovně:

- Push vkládá na vrchol zásobníku v případě, že vkládaná hodnota je větší než aktuální vrchol zásobníku a vrchol aktualizuje. V opačném případě vloží nový prvek hned pod vrchol zásobníku a vrchol nemění.
- Pop odebírá prvek ze zásobníku z jeho vrcholu v případě, že vrchní prvek je větší než prvek pod ním. V opačném případě se odstraní prvek pod vrchním prvkem, tedy ten co má větší hodnotu.

Karlík varuje: Zkontrolujte si, jak se vaše implementace chová na prázdném a jednoprvkovém zásobníku.

1.14

1. Implementujte funkci, která bere na vstupu dva ukazatele na začátky zřetězených seznamů a určí, jestli jsou seznamy shodné. Seznamy považujeme za shodné, pokud obsahují stejný počet prvků se stejným obsahem ve správném pořadí.
2. Modifikujte předchozí porovnávací funkci na ověření shodnosti cyklických seznamů.
3. Jak byste postupovali, pokud byste měli určit, jestli seznamy obsahují stejné prvky (zapsané v libovolném pořadí)?

1.15 Mějme dva zřetězené seznamy, které obsahují prvek (bod shody), od něhož jsou dále stejné. Pro příklad mějme seznamy čísel $S_1 = [1, 2, 3, 5]$ a $S_2 = [1, 3, 5]$, jejichž bod shody je 3. Implementujte funkci, která najde bod shody zadaných dvou seznamů.

1.16 Mějme dva zřetězené seznamy, které obsahují seřazené posloupnosti čísel. Navrhněte funkci Merge, která spojí tyto dva zřetězené seznamy do jediného zřetězeného seznamu, který bude mít také všechny prvky seřazené. Při spojování nevytvářejte nový seznam ani nové prvky seznamu, pracujte jenom s ukazateli na následující prvky.

1.17 Implementujte 2D seznam, který splňuje následující požadavky. Každý prvek 2D seznamu má ukazatele *left*, *right*, *up* a *bottom*, které ukazují na příslušné okolní prvky (v případě, že daný prvek neexistuje, pak je hodnota ukazatele *nil*). Do seznamu lze vkládat pouze pomocí funkcí insertLeft a podobných, které berou ukazatel na prvek, vedle kterého se má nový prvek vložit, a vkládaný prvek. Vložení proběhne pouze v případě, že příslušný vedlejší prvek je před vložením *nil*. Při vkládání dejte pozor, abyste vytvořili příslušné ukazatele na všechny okolní prvky.

Pokud máte zájem, můžete implementovat i funkce pro odstranění okrajového prvku. Pro uvedení příkladu funkcionality uvažujme takovýto 2D seznam:

$$\begin{array}{ccc} & a & b \\ c & d & e \\ f & & h \end{array}$$

Prvek g (na pozici podle lexikografického uspořádání) lze vložit voláním insertRight(f, g), insertBottom(d, g), nebo insertLeft(h, g). Po vložení bude mít prvek g nastaven správně okolní ukazatele na f, d, h a ty budou mít zase zpětně nastaveny ukazatele na g.

Kapitola 2

Algoritmy a korektnost

Nejprve se studenty zopakujte definice těchto základních pojmů z přednášky:

Algoritmus je přesný a jednoznačný popis toho, jak máme postupovat, abychom po provedení tohoto postupu na vstupních hodnotách dostali kýžený výsledek.

Program je algoritmus zapsaný v jistém programovacím jazyce.

Následující pojmy jsou definovány pouze pro algoritmy. Obdobně se definují pro programy.

Vstupní podmínka ze všech možných vstupů do daného algoritmu vymezuje ty, pro které je algoritmus definován. Chování na ostatních vstupech nás nezajímá. Vstupní podmínka se značí symbolem φ.

Výstupní podmínka pro každý vstup daného algoritmu splňující vstupní podmínku určuje, jak má vypadat výsledek odpovídající danému vstupu. Značí se symbolem ψ.

Algoritmus je částečně (parciálně) korektní, pokud pro každý vstup, který splňuje vstupní podmínku a algoritmus na něm skončí, výstup splňuje výstupní podmínku (spolu s odpovídajícím vstupem).

Algoritmus je úplný (konvergentní), pokud pro každý vstup splňující vstupní podmínku výpočet skončí.

Totálně korektní algoritmus je parciálně korektní a konvergentní.

Invariant cyklu je každé takové tvrzení o algoritmu, které platí před vykonáním a po vykonání každé iterace cyklu.

Matematická indukce je běžný způsob dokazování korektnosti rekurzivního algoritmu. Nejdříve tvrzení musíme dokázat pro funkci bez zanoření, dále předpokládáme platnost pro obecné zanoření do hloubky m a indukčním krokem musíme potvrdit, že pokud platí pro m, pak platí i pro $m + 1$.

Cvičení na korektnost algoritmů nebývá jednoduché jak pro studenty, tak pro cvičící. Doporučuje se předem pořádně projít řešení, ať se u něčeho nezaseknete.

Úvodní příklad (GetCurrentYear) má být motivací ke studiu korektnosti. Zdůrazněte, že algoritmus je reálný kód z dílny Microsoftu, díky kterému 31. 12. 2008 cyklily přehrávače Zune. Neztrácejte

u příkladu moc času, má to být jen ukázka – najděte chybu, opravte ji, ale už neřešte korektnost opraveného algoritmu.

Jádrem cvičení jsou 4 příklady – 2 korektní (které zaberou víc času) a 2 různým způsobem nekorektní. Můžete změnit pořadí, ve kterém je chcete probírat. Komentáře k nim najdete níže.

Programování nabízí 2 možnosti, co dělat. Dokončit datové struktury z minulého týdne (případně seznam rozšířit na oboustranně zřetězený a cyklický). Druhá možnost je opravit chybové zdrojové kódy s typickými algoritmickými chybami. Očekává se, že na programovací část si necháte asi půl hodiny (což bude na dalších cvičeních standard, tak ať si na to studenti zvyknou).

Paní Bílá připomíná: Algoritmus je procedura proveditelná Turingovým strojem.

2.1 Je následující algoritmus, který jako vstup bere počet dnů od roku 1980 a jako výstup má vrátit aktuální rok, korektní?

Funkce GetCurrentYear(*days*)

vstup: *days* je počet dní od 1. 1. 1980
výstup: napočítaná hodnota *year* s aktuálním rokem

```
1  year ← 1980
2  while days > 365 do
3      if year je přestupný rok then
4          if days > 366 then
5              days ← days − 366
6              year ← year + 1
7          fi
8      else
9          days ← days − 365
10         year ← year + 1
11     fi
12 od
13 return year
```

Ne, protože 31. 12. 2008 (přestupný rok) došlo k zacyklení (hodnota *days* byla 10593 – můžete si to vyzkoušet sami). V iteraci, která měla být poslední, proměnná *days* byla 366. Byla splněna podmínka cyklu i první podmínka **if**, ale druhá podmínka **if** nebyla splněna a tedy nedošlo ke snížení hodnoty *days*. Řešením je přidat **else** větev pro tento případ, který cyklus zastaví.

S touto chybou se museli vypořádat programátoři Zune z Microsoftu.

Pan Usměvavý dodává: Jak informatik vaří vodu? Vezme konvici a podle toho, zda je prázdná, se rozhodne: Pokud je prázdná, pak do ní nalije vodu a dá ji vařit. Pokud není prázdná, pak z ní vylije vodu, čímž problém redukuje na předchozí případ. – toto je korektní algoritmus.

Zaměřeno na vstupní a výstupní podmínky, celkem na ně navádí podmínka cyklu. Motivace je naučit studenty pracovat se vstupními i výstupními podmínkami.

Podle vlastního uznání můžete studentům říci hned, že algoritmus má počítat x^2, nebo jim můžete hlavičku schovat, nechat je přijít na to, co algoritmus počítá a teprve potom říci, že má počítat x^2.

2.2 Mějme následující algoritmus, jehož vstupem i výstupem je reálné číslo.

Funkce SQUARE(x)

vstup: x je číslo
výstup: x^2

```
1 i ← x
2 z ← 0
3 while ⌊i⌋ ≠ 0 do
4     z ← z + x
5     i ← i − 1
6 od
7 return z
```

a) Vzhledem ke kterým z následujících vstupních podmínek je algoritmus konvergentní? Přirozená čísla uvažujeme včetně nuly.

a) $\varphi(x) \equiv x \in \mathbb{R}$ b) $\varphi(x) \equiv x \in \mathbb{R}^+$
c) $\varphi(x) \equiv x \in \mathbb{Z}$ d) $\varphi(x) \equiv x \in \mathbb{N}$

Algoritmus cyklí právě tehdy, když je hodnota x na počátku menší než 0. Proto algoritmus není konvergentní vzhledem k podmínkám a) a c). Vzhledem ke zbylým podmínkám je konvergentní.

b) Vzhledem ke kterým z uvedených vstupních podmínek a výstupní podmínce

$$\psi(x, z) \equiv z = x^2$$

je algoritmus parciálně korektní?

Algoritmus pro nezáporná čísla spočítá hodnotu $z = x\lfloor x \rfloor$, což se rovná x^2 právě pro celá čísla x. Proto je parciálně korektní vzhledem ke vstupní podmínce c) a také k d). Algoritmus je proto totálně korektní vzhledem k uvedené výstupní podmínce a vstupní podmínce d).

c) Jak se změní vstupní podmínka, pokud cyklus zapíšeme bez zaokrouhlení i dolů, tedy $i \neq 0$? Pro jaké vstupy je nyní algoritmus korektní pro výstupní podmínku z části b)?

Vstupní podmínka musí vstup omezit i o reálná a racionální čísla, takže zůstane pouze množina přirozených čísel, tedy $\varphi(x) \equiv x \in \mathbb{N}$.

Chyba v podmínce cyklu. Motivací je studentům dostat do podvědomí opatrnost na chyby v takovýchto místech. Na konci algoritmus jednoduše opravte nahrazením $\neq$ za $<$ a korektnost opraveného řešení jim nechte na doma.

2.3

a) Rozhodněte, zda následující algoritmus korektně testuje, zdali je vstupní řetězec S palindrom. Pro potřeby příkladu ještě zadefinujme, že přístup mimo meze řetězce vždy vrátí prázdný znak „–“.

Funkce PalindromCheck(S, n)

vstup: řetězec S délky n
výstup: $true$ pokud je S palindrom, jinak $false$

```
1  i ← 1
2  j ← n
3  while i ≠ j do
4      if S[i] ≠ S[j] then
5          return false
6      else
7          i ← i + 1
8          j ← j − 1
9  od
10 return true
```

Algoritmus není totálně korektní, jelikož není pro všechny vstupy konečný.

b) Pokud je algoritmus totálně korektní, dokažte to pomocí invariantu cyklu. Pokud není, uveďte příklad vstupní posloupnosti a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní. Algoritmus se v takovém případě pokuste opravit.

Vstupy, pro které algoritmus nefunguje, jsou všechny vstupy sudé délky. Na těchto vstupech totiž algoritmus není konečný.

Algoritmus můžeme opravit tak, že na řádku 3 zapíšeme podmínku $i < j$. Pokud máme extra smysl pro humor, můžeme opravu provést nastavením vstupní podmínky na řetězec S s lichou délkou. Takové řešení je totálně korektní, ale není to to, co bychom od algoritmu očekávali.

Správný, jednoduchý a praktický algoritmus, ale vstupní a výstupní podmínky a invariant ve formální podobě vypadají hrozivě. Stačí tedy ze studentů dostat slovní verzi, která je jednodušší (ale pozor na malé chyby). Invariant cyklu používáme pro důkaz korektnosti iterativního algoritmu, pro rekurzivní budeme používat indukci (až ve 4. cvičení).

Alternativně můžete změnit pořadí příkladů a na toto místo zařadit příklad 6. Tam je důkaz korektnosti snazší. Důvod, proč je ve sbírce toto pořadí je, že tento příklad provedete společně se studenty na tabuli, zatímco příklad 6 už necháte samostatně na studentech na papír.

2.4 Funkce Soucet pro vstupní posloupnost $A = (a_1, a_2, \ldots)$ celých čísel vypočte součet prvních i prvků posloupnosti, kde i je nejmenší index takový, že $a_i = 0$.

Funkce Soucet(A)

vstup: A je pole celých čísel indexováno od 1
výstup: součet posloupnosti A

```
1 i ← 1
2 s ← 0
3 while A[i] ≠ 0 do
      // místo pro invariant
4     s ← s + A[i]
5     i ← i + 1
6 od
7 return s
```

Ze zadání formulujte vstupní a výstupní podmínku, následně nalezněte invariant **while** cyklu na řádcích **3** až **6** a dokažte pomocí něj korektnost algoritmu.

Využijte případných špatně zformulovaných podmínek pro důkaz korektnosti, který je sice korektní, ale neplatí pro podmínku ze zadání. Tedy algoritmus počítá něco jiného, než po něm chceme.

Význam různých indexů u tohoto důkazu: s a i jsou proměnné používané v algoritmu, na následujících řádcích tedy odkazují na hodnoty těchto proměnných v algoritmu. Index k odpovídá pozici posledního sčítaného prvku posloupnosti A. Index j bude používán ve spojení s nějakým kvantifikátorem. Abychom v sumách nepřetěžovali index i, budeme používat index h.

Vstupní podmínka:

$$\varphi(A) \equiv \forall j.A[j] \in \mathbb{Z} \ \wedge \ \exists k \geq 1.A[k] = 0 \ \wedge \ (\forall j.1 \leq j < k \Rightarrow A[j] \neq 0)$$

Přepis vstupní podmínky do češtiny: všechna čísla v posloupnosti jsou celá a existuje jedno číslo, které je nula a všechny hodnoty s menším indexem, než je index této nuly, jsou nenulové.

Výstupní podmínka:

$$\psi(A, s) \equiv \exists k \geq 1. \left(A[k] = 0 \ \wedge \ (\forall j.1 \leq j < k \Rightarrow A[j] \neq 0) \ \wedge \ s = \sum_{h=1}^{k} A[h] \right)$$

Přepis výstupní podmínky do češtiny: výstupem algoritmu je suma čísel po první výskyt nuly v posloupnosti.

Invariant while cyklu:

V textu dále budeme používat k k označení indexu první nuly. Chcete-li vidět další zápisy matematicky přesně, představte si na začátku každé formule $\exists k \geq 1.A[k] = 0 \ \wedge \ \forall j.1 \leq j < k \Rightarrow A[j] \neq 0 \wedge$.

$$i \leq k \ \wedge \ s = \sum_{h=1}^{i-1} A[h]$$

Před vstupem do cyklu je $i = 1$ a $s = 0$. Víme, že $k \geq 1$ a $\sum_{h=1}^{0} A[h]$ je prázdná suma, která se tradičně klade rovna 0. Invariant tedy před cyklem platí.

Nyní ukážeme, že když provedeme průchod cyklem, invariant zůstane v platnosti (tj. platí po průchodu, pokud platil před). Podíváme se nejdříve na $i \leq k$. Při každém průchodu cyklem se nám i zvětší o 1. Nemůže nám vyrůst nad k? Nemůže, protože při $i = k$ je $A[i] = 0$ a další průchod cyklem už se neprovede. Teď se podívejme na s. Označme si s' a i' hodnoty po průchodu, ať je odlišíme od hodnot před průchodem. Víme, že $i < k$, $i' = i + 1$ a předpokládáme $s = \sum_{h=1}^{i-1} A[h]$. Invariant platí, protože

$$s' = s + A[i] = \sum_{h=1}^{i-1} A[h] + A[i] = \sum_{h=1}^{i} A[h] = \sum_{h=1}^{i'-1} A[h].$$

Korektnost algoritmu:

Oproti vstupní podmínce je ve výstupní podmínce navíc návratová hodnota $s = \sum_{h=1}^{k} A[h]$. Jakou hodnotu algoritmus vrací? Hodnotu s po posledním průchodu cyklem, tj. v situaci, kdy $A[i] = 0$, a tedy $i = k$. Invariant cyklu platí i po posledním průchodu, proto $s = \sum_{h=1}^{i-1} A[h]$, a tedy $\sum_{h=1}^{k-1} A[h]$. Protože $A[k] = 0$, je tato suma rovna $\sum_{h=1}^{k} A[h]$. Tím jsme dokázali parciální korektnost. **Totální korektnost** plyne z toho, že dle vstupní podmínky k existuje. Na začátku cyklu je $i \leq k$, i se zvedá vždy jen o jedna a při $i = k$ je cyklus ukončen.

Chyba v hlídání mezí. Znovu se snažíme ve studentech vybudovat podvědomou opatrnost k nejčastějším chybám.

2.5

a) Rozhodněte, zda následující algoritmus správně vypíše horní trojúhelník hodnot matice M. Horní trojúhelník zahrnuje všechny pozice nad hlavní diagonálou (hlavní diagonála odpovídá pozicím, kde $x = y$) včetně diagonály.

Funkce TriangleMatrixPrint(M, x, y)

vstup: matice M s rozměry y řádků a x sloupců

```
1  i ← 1
2  while i ≤ y do
3      j ← 1
4      while j ≤ x do
5          if j < i then
6              j ← i // přeskoč k diagonále
7          fi
8          Print(M[i, j])
9          j ← j + 1
10     od
11     i ← i + 1
12 od
```

Algoritmus není totálně korektní, jelikož u obdélníkové matice, která má více řádků než sloupců (větší rozměr na ose y), nekorektně přeskakuje k diagonále a následně vypisuje hodnotu mimo rozměry matice.

b) Pokud je algoritmus korektní, dokažte to. Jestliže není, uveďte příklad vstupní posloupnosti a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní a pokuste se algoritmus opravit.

Vstupem může být například $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$. Výstupem by mělo být 1 2 4, ale na 3. řádku není chování algoritmu definováno (bude záležet na konkrétním jazyce a implementaci v něm – v C by pravděpodobně došlo k přístupu mimo paměť příslušející matici, Python kontroluje přístup mimo paměť listu, takže by vyhodil výjimku „list index out of range“).

Opravu lze provést změnou řádku **3** na $j \leftarrow i$ a smazáním řádků **5** až **7**.

Poslední cvičení na invariant cyklu. Není nutné probrat, pokud nemáte dostatek času na programování.

2.6 Následující algoritmus SelectSort (A, n) seřadí pole čísel $A = [A_1, \ldots, A_n]$. Na konci výpočtu budou prvky v poli $A' = [A'_1, \ldots, A'_n]$ permutací prvků pole A tak, že budou seřazené vzestupně.

Pro vyhledání maxima se použije následující algoritmus, který vrací index v poli, na kterém se nachází maximální prvek.

Funkce Maximum(A, n)

vstup: (A, n) // *A je pole a n je počet prvků v poli*
výstup: index maxima z pole A

```
1 maxIndex = 1
2 for i ← 2 to n do
3     if A[i] > A[maxIndex] then
4         maxIndex ← i
5     fi
6 od
7 return maxIndex
```

Dokazování korektnosti algoritmu Maximum si necháme na příklad 2.15, zatím můžete předpokládat jeho korektnost.

Funkce SelectSort(A, n)

vstup: (A, n) // *A je pole a n je počet prvků v poli*
výstup: vzestupně seřazená posloupnost A

```
1 for i ← n downto 2 do
2     j ← Maximum (A, i)
3     Swap (A[i], A[j]) // přehodí prvky
4 od
5 return A
```

Ze zadání zformulujte vstupní a výstupní podmínky a na základě invariantu **for** cyklu dokažte korektnost algoritmu vzhledem k těmto podmínkám.

Daliborek vzkazuje: Abychom dokázali invariant, musíme dokázat invariant.

Podmínky jsou následující:

$$\varphi(A, n) \equiv n = |A| \wedge n \geq 1$$

$$\psi(A, n, A') \equiv A'_1 \leq \cdots \leq A'_n \wedge A' \text{ je permutací } A.$$

Jako invariant **for** cyklu zvolme tvrzení, že posledních $n - i$ prvků posloupnosti je vzestupně uspořádaných a zároveň jsou větší, než všechny zbylé prvky posloupnosti. Nerovnost v invariantu je neostrá, stejně jako v důkazu níže (v posloupnosti samozřejmě může být více stejných prvků).

Před prvním průchodem cyklem platí $i = n$, a proto je tvrzení triviálně splněno.

Zachování invariantu. Podíváme se, jestli nám průchod cyklem nepokazí invariant. Před průchodem předpokládáme platnost invariantu, tedy posledních $n - i$ prvků je vzestupně seřazených a zároveň jsou větší než zbytek posloupnosti. Zbytek posloupnosti je prvních i prvků. V cyklu se v tomto zbytku provede hledání maxima a prvek s indexem maxima je následně prohozen s prvkem na pozici i. Tím je prodloužen blok posledních seřazených a zachována platnost invariantu pro další průchod cyklem se sníženým i.

Korektnost algoritmu. Algoritmus vrací pole A hned, jak cyklus skončí. Dle invariantu je po posledním průchodu posledních $n - 1$ prvků seřazených a zbývající první prvek je jim menší nebo roven. Pole A je tedy celé seřazené. Tím jsem dokázali parciální korektnost. **Totální korektnost** plyne z toho, že běh **for** cyklu je vždy konečný (pokud nezasahujeme do proměnné i).

2.7 Na programovací část jsme pro vás připravili zdrojové kódy v C a Python se spoustou typických chyb. Vaším úkolem je jich najít a opravit co nejvíce. Zkoušejte různé vstupy, hledejte na místech, kde se chyby dělají nejčastěji.

Následující příklady jsou vhodné na domácí studium.

2.8 Které algoritmy jsou vzhledem k dané vstupní a výstupní podmínce parciálně korektní právě tehdy, když jsou totálně korektní?

Pro každý algoritmus platí, že když je totálně korektní, tak je také parciálně korektní – to je důsledek definice korektnosti. Naopak z parciální korektnosti vyplývá totální korektnost jen u konvergentních algoritmů. Celkem tedy, parciální korektnost je ekvivalentní totální korektnosti právě pro algoritmy, které jsou konvergentní (tedy zastaví na každém vstupu splňujícím vstupní podmínku).

2.9 Který algoritmus je parciálně korektní vzhledem k libovolným vstupním a výstupním podmínkám?

Jelikož můžeme vytvořit výstupní podmínku typu $\psi \equiv false$, zdánlivě neexistuje žádná vstupní podmínka, pro kterou by byl algoritmus korektní. Navíc náš algoritmus musí být parciálně korektní pro libovolnou vstupní podmínku, tedy i pro *true*. To zní vražedně. Avšak vzhledem k tomu, že vyžadujeme jen parciální korektnost, stačí mít jistotu, že program bude vždy cyklit a nikdy neskončí, např. **while** $1 = 1$ **do od**.

2.10 Předpokládejme, že číselné pole $A[1 \ldots n]$ obsahuje seřazenou posloupnost čísel (od nejmenšího po největší). Dále předpokládejme, že A **obsahuje číslo** x. Rozhodněte, zda volání funkce Search$(A, x, 1, n)$

vrátí hodnotu indexu l takového, že $A[l] = x$. Jestliže ano, dokažte korektnost algoritmu. Jestliže ne, uveďte příklad vstupní posloupnosti a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní.

Funkce SEARCH(A, x, l, r)

vstup: A je seřazené pole, x je hledaný prvek, l a r jsou indexy intervalu vyhledávaní
výstup: Index hledaného prvku

```
1  if l = r then
2      return l
3  else
4      m ← ⌊l + (r − l + 1)/2⌋
5      if x ≤ A[m] then
6          Search (A, x, l, m)
7      else
8          Search (A, x, m + 1, r)
9      fi
10 fi
```

2.11 Rozhodněte, zda následující algoritmus vrátí sumu všech prvků matice čísel. Jestliže ano, dokažte korektnost algoritmu. Jestliže ne, uveďte příklad vstupní posloupnosti a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní.

Funkce MATRIXSUM(A, n)

vstup: matice celých čísel A rozměrů $n \times n$, $n \geq 1$
výstup: suma všech prvků matice A

```
1  s ← 0
2  for i ← 1 to n do
3      for j ← 1 to n/2 do
4          s ← s + A[i, j]
5          s ← s + A[j, i];
6      od
7  od
8  return s
```

2.12 Rozhodněte, zda následující algoritmus zvětší vstupní argument o jedna. Jestliže ano, dokažte korektnost algoritmu. Jestliže ne, uveďte příklad vstupu a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní.

Funkce Increment(y)

vstup: $y \in \mathbb{N}$
výstup: $y + 1$

```
1  x ← 0, c ← 1, d ← 1
2  while (y > 0) ∨ (c > 0) do
3      a ← y mod 2
4      if a ⊕ c then
           // ⊕ je operace xor
5          x ← x + d
6      fi
7      c ← a ∧ c
8      d ← 2d
9      y ← ⌊y/2⌋
10 od
11 return x
```

2.13 Rozhodněte, zda následující algoritmus správně vypočítá sumu všech jedniček v binárním řetězci B. Jestliže ano, dokažte korektnost algoritmu. Jestliže ne, uveďte příklad vstupní posloupnosti a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní a pokuste se algoritmus opravit.

Funkce BitsSum(B)

vstup: pole B obsahující 0 a 1
výstup: počet jedniček v poli B

```
1  sum_1 ← 0
2  for i ← 1 to ⌈n/2⌉ do
3      if B[i] = 1 then
4          sum_1 ← sum_1 + 1
5      fi
6      if B[n − i] = 1 then
7          sum_1 ← sum_1 + 1
8      fi
9  od
10 return sum_1
```

2.14 Rozhodněte, zda následující algoritmus správně vypíše horní (nad diagonálou) trojúhelník hodnot matice M. Jestliže ano, dokažte korektnost algoritmu. Jestliže ne, uveďte příklad vstupní posloupnosti a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní a pokuste se algoritmus opravit.

Funkce TriangleMatrixPrint(M)

vstup: matice M rozměrů $x \times y$

```
1  i ← 1
2  j ← 1
3  while i ≤ x do
4      while j ≤ y do
5          if i = j then
6              while i > 0 do
7                  Print(M[i, j])
8                  i ← i − 1
9              od
10             j ← j + 1
11             i ← i + 1
12         fi
13     od
14 od
```

2.15 Dokažte parciální korektnost algoritmu pro seřazení prvků v dvouprvkovém poli A indexovaném od jedničky vzhledem k následujícím podmínkám.

$$\varphi(A) \equiv A \textit{ je dvouprvkové pole celých čísel}$$

$$\psi([x, y], [p, q]) \equiv p \leq q \wedge (p, q) \textit{ je permutací} (x, y)$$

Výstupní podmínka není zadaná korektně pro jiné velikosti polí. Můžeme předpokládat, že pro případy nepokryté uvedeným vzorem vrací $false$.

Funkce Sort(A)

vstup: A dvouprvkové pole
výstup: seřazené pole A

```
1  if A[1] > A[2] then
2      z ← A[1]
3      A[1] ← A[2]
4      A[2] ← z
5  fi
6  return A
```

Podmínky se doplní následovně (postupujte po krocích shora dolů).

Funkce SortCommented(A)

vstup: A dvouprvkové pole
výstup: seřazené pole A

```
1 if A[1] > A[2] then
      // A[1] = x, A[2] = y, x > y
2     z ← A[1]
      // A[1] = x, A[2] = y, z = x, x > y
3     A[1] ← A[2]
      // A[1] = y, A[2] = y, z = x, x > y
4     A[2] ← z
      // A[1] = y, A[2] = x, z = x, x > y
      // A[1] = x, A[2] = y, x ≤ y
5 fi
6 return A  // [p, q], p ≤ q ∧ ((p = x ∧ q = y) ∨ (p = y ∧ q = x))
```

2.16 Analýza algoritmu hledání maxima. Mějme následující algoritmus:

Funkce Maximum(A, n)

vstup: (A, n) // *A je pole a n je počet prvků v poli*
výstup: maximum z pole A

```
1 max = A[1]
2 for i ← 2 to n do
3     if A[i] > max then
4         max ← A[i]
5     fi
6 od
7 return max
```

Dokažte parciální korektnost algoritmu Maximum(A, n) vzhledem ke vstupní podmínce:

$$\varphi(A, n) \equiv A \text{ je neprázdné pole celých čísel délky } n$$

a výstupní podmínce

$$\psi(A, max) \equiv \; max \text{ leží v } A \text{ a pro všechna } q \text{ z pole } A \text{ platí } q \leq max.$$

Daliborek vzkazuje: Korektnost algoritmu je vhodné dokazovat pomocí nejslabší vstupní podmínky.

2.17 Následující algoritmus vzestupně seřadí číselnou posloupnost $a = (a_1, \ldots, a_n)$ uloženou v poli A. Tedy na konci výpočtu bude v poli A posloupnost $a' = (a'_1, \ldots, a'_n)$, která je permutací posloupnosti a a platí $a'_1 \leq \cdots \leq a'_n$.

Funkce Sort(A, n)

vstup: (A, n) // *A je pole a n je počet prvků v poli*
výstup: seřazená posloupnost A

```
1 for i ← 2 to n do
2     x ← A[i]
3     j ← i − 1
4     while (j > 0) ∧ (A[j] > x) do
5         A[j + 1] ← A[j]
6         j ← j − 1
7         A[j + 1] ← x
8     od
9 od
```

a) Formulujte vstupní a výstupní podmínky a dokažte korektnost algoritmu vzhledem k těmto podmínkám.

$\varphi(A, n) \equiv n = |A| \wedge n \geq 1$, $\psi(A, A', n) \equiv |A'| = n \wedge A'[1] \leq \cdots \leq A'[n] \wedge A'$ je permutací A.
Invariant vnějšího cyklu platící na jeho začátku: Úsek pole $A[1], \ldots, A[i-1]$ je permutací prvních $i-1$ prvků vstupní posloupnosti a tento úsek je neklesající, tj. $A[1] \leq \cdots \leq A[i-1]$.
Invariant vnitřního cyklu platící na jeho začátku: $\forall k.j \leq k \leq i \implies A[k] \leq x$.

b) Napište invariant vnějšího cyklu. Napište mezilehlé podmínky pro začátek a konec těla vnějšího cyklu a pomocí nich a invariantu vnějšího cyklu dokažte správnost algoritmu.

Invariant platící na *konci* těla vnějšího cyklu (ještě před dalším zvýšením proměnné i): $A[1] \leq A[2] \leq \cdots \leq A[i]$ a *současně* $\forall j.i < j \leq n \implies A[i] \leq A[j]$ a *současně posloupnost A je permutací vstupní posloupnosti A*.

Vztahuje-li se invariant ke *konci* těla cyklu, je pro důkaz korektnosti důležité, že tělo cyklu bude provedeno aspoň jednou. To je v našem případě zaručeno kladností n vyplývající ze vstupní podmínky.

Důkaz korektnosti je pak snadný. Po posledním průchodu tělem vnějšího cyklu je hodnota proměnné i rovna číslu n. Po dosazení n za i v invariantu dostaneme výstupní podmínku.

c) Formulujte invariant vnitřního cyklu. Pomocí něho a dříve napsaných mezilehlých podmínek ukažte správnost těla vnějšího cyklu vzhledem k těmto podmínkám.

Kapitola 3

Délka výpočtu, složitost

Studentům můžete na začátku připomenout několik základních pojmů.

Složitost vyjadřuje náročnost algoritmu na různé zdroje výpočtu: dobu výpočtu, velikost paměti, počet procesorů apod. Podle toho rozlišujeme různé míry složitosti.

Délka výpočtu konkrétního algoritmu na konkrétním vstupu je počet elementárních operací, ze kterých se tento výpočet skládá.

Časová složitost algoritmu je funkce f na množině přirozených čísel taková, že výpočet algoritmu pro každý vstup délky n má délku nejvýše $f(n)$. Složitost algoritmu obvykle vyjadřujeme asymptoticky.

Asymptotická notace: pro každou funkci $g : \mathbb{N} \to \mathbb{R}_0^+$ zavedeme následující množiny funkcí:

- $\mathcal{O}(g) = \{f \mid \exists c > 0, n_0 \in \mathbb{N} : \ \forall n \geq n_0 \ : \ f(n) \leq c \cdot g(n)\}$
  Množina funkcí rostoucích nejvýše tak rychle jako g.
- $\Omega(g) = \{f \mid \exists c > 0, n_0 \in \mathbb{N} : \ \forall n \geq n_0 \ : \ c \cdot g(n) \leq f(n)\}$
  Množina funkcí rostoucích alespoň tak rychle jako g.
- $\Theta(g) = \{f \mid \exists c_1 > 0, c_2 > 0, n_0 \in \mathbb{N} : \ \forall n \geq n_0 \ : \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} = \mathcal{O}(g) \cap \Omega(g)$
  Množina funkcí rostoucích stejně rychle jako g.

Cvičení na složitost bývá značně jednodušší, než korektnost. Matematika nebývá složitá, takže se nebojte ptát se na těžší otázky. Složitost se učí v Úvodech do programování (ale velmi stručně), takže ponětí o ní již studenti mají.

Úvodní příklady testují chápání principů asymptotické notace. Měly by být pro studenty jednoduché a zabrat jen pár minut.

Navazující příklad 3.3 je na delší dobu, ale ukáže studentům třídy asymptotické složitosti. Nechte u něj studenty spolupracovat, diskutovat, měli by to zvládnout vymyslet sami.

Příkladem 3.4 na matematickou indukci končí abstraktní část. Pokračují příklady na určení složitosti různých algoritmů. Případné komentáře najdete přímo u nich. Snaží se hlavně studentům ukázat, že rozdíl mezi různými algoritmy pro jeden problém může být zásadní, a proto by měli složitost chápat.

Programování je zaměřeno na vlastní měření doby běhu různých algoritmů (algoritmy pro umocňování a Fibonacciho posloupnost). Tradičně by vám měla na programování zbýt poslední půl hodina.

Daliborek vzkazuje: Logaritmické funkce rostou exponenciálně pomaleji než lineární funkce. Libovolná polylogaritmická funkce tedy roste pomaleji než libovolná polynomiální funkce. Exponenciální logaritmickou funkci a logaritmickou exponenciální funkci však lze převést na polynomiální či exponenciální funkci.

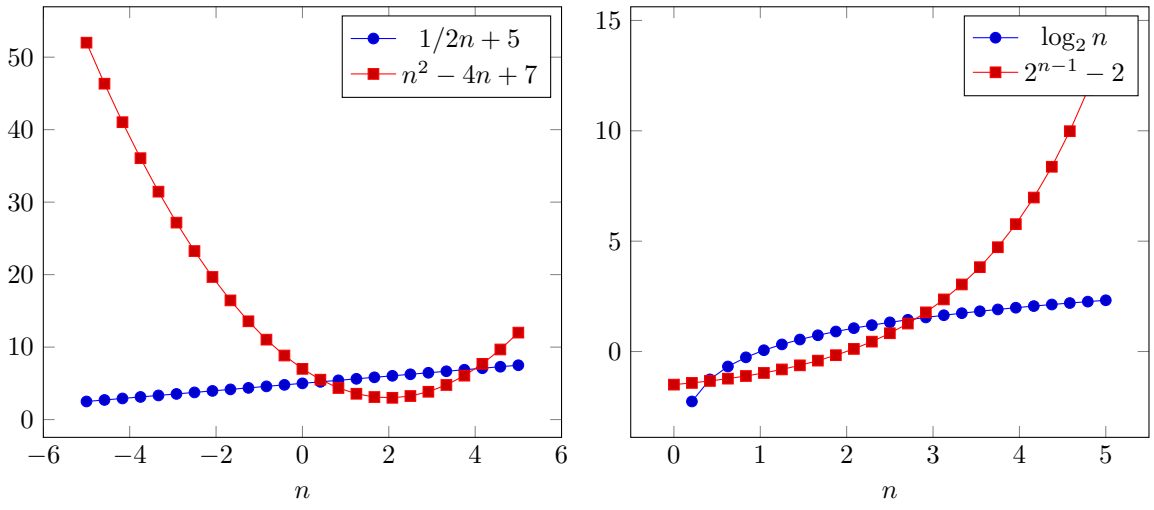
3.1 Rozhodněte a zdůvodněte, zda jsou následující tvrzení pravdivá:

1. $3n^5 - 16n + 2 \in \mathcal{O}(n^5)$
2. $3n^5 - 16n + 2 \in \mathcal{O}(n)$
3. $3n^5 - 16n + 2 \in \mathcal{O}(n^{17})$
4. $3n^5 - 16n + 2 \in \Omega(n^5)$
5. $3n^5 - 16n + 2 \in \Theta(n^5)$
6. $3n^5 - 16n + 2 \in \Theta(n)$

Příklad řešte pouze intuitivně, studenti by si hlavně měli odnést myšlenku, že důležitý je pouze řídící člen polynomu. Také by studenti měli zvládat intuitivně rozlišovat různé třídy složitosti, obdobný příklad bývá na zkoušce.

1. ano
2. ne (rozhodující člen je n^5 a ten je jistě větší než n)
3. ano (n^{17} je větší než n^5)
4. ano
5. ano (zřejmé z předchozích)
6. ne (n^5 je větší než n)

3.2 Pro následující funkce f a g určete dvojici konstant c a n_0, která je svědkem toho, že platí $f \in \mathcal{O}(g)$, resp. $g \in \Omega(f)$. Kolik existuje takových dvojic? Pro určení dvojice c a n_0 použijte následující grafy.



a) $f(n) = \frac{1}{2}n + 5$; $g(n) = n^2 - 4n + 7$

Graf funkce f má průsečíky s osami v bodech $[0, 5]$ a $[-10, 0]$, funkce g je konvexní parabola s vrcholem v bodě $[2, 3]$. Řešením je např. dvojice $c = 1$ a $n_0 = 5$.

b) $f(n) = \log_2 n$; $g(n) = 2^{n-1} - 2$

Graf funkce g je graf funkce 2^n s počátkem posunutým do bodu $[1, -2]$. Řešením je např. dvojice $c = 1$ a $n_0 = 3$.

Dvojic konstant c, n_0, které jsou svědky toho, že $f \in \mathcal{O}(g)$, resp. $g \in \Omega(f)$, je nekonečně mnoho.

Tento příklad podle možností můžete řešit nějakým konkrétním řadícím algoritmem. Osvědčil se QuickSort, který studenti v roce 2015 už znali z přednášky.

QuickSort pomůže rozdělit funkce do menších skupin, které už studenti lépe porovnají, než celou množinu funkcí. Pokud za pivota vyberete známé funkce (n, 2^n, $\log(n)$), pak poměrně dobře zvládnou i jednotlivé porovnání.

Některá srovnání nemusíte dokázat formálně. Například k porovnání n^n a 2^{2^n} stačí volba $n = 10$, pro kterou lze poměrně intuitivně rychle ukázat velký rozdíl hodnot – 10^{10} s 10 řády proti 2^{1024} s 309 řády.

3.3 Seřaďte následující funkce podle rychlosti jejich růstu:

$n^2 + \log n$	$7n^5 - n^3 + n$	n^2
$\log \log n$	2^n	$\log n$
$(\frac{3}{2})^n$	$n \cdot \log n$	$n!$
n	n^n	6
$\log(n!)$	$\log^{14} n$	$\sqrt{\log n}$
e^n	$2^{\log n}$	2^{2^n}

Výsledné seřazení je následující:

$$
\begin{aligned}
6 \\
\log\log n \\
\sqrt{\log n} \\
\log n \\
\log^{14} n \\
n = 2^{\log n} \\
\log(n!) = n\log n \\
n^2 = n^2 + \log n \\
7n^5 - n^3 + n \\
\left(\frac{3}{2}\right)^n \\
2^n \\
e^n \\
n! \\
n^n \\
2^{2^n}
\end{aligned}
$$

Detailní porovnání asymptotického růstu některých funkcí.

$\mathcal{O}(\log(n!)) = \mathcal{O}(n\log n)$.
Součin funkcí v logaritmu lze přepsat na součet logaritmů těchto funkcí. Rozepsáním tedy získáme $\log(n!) = \log 1 + \log 2 + \cdots + \log n$. To je ohraničeno shora funkcí $n\log n$. Je potřeba určit ještě dolní hranici. Pokud zanedbáme první polovinu sčítanců, zůstane pouze součet $\log(n!) \geq \log\frac{n}{2} + \log(\frac{n}{2}+1) + \cdots + \log n \geq \log\frac{n}{2} + \log\frac{n}{2} + \cdots + \log\frac{n}{2} = \frac{n}{2}\log\frac{n}{2}$.

Funkce $\log n!$ může být pouze v rozsahu funkcí $n\log n$ a $\frac{n}{2}\log\frac{n}{2}$, přičemž obě jsou jen konstantními násobky funkce $n\log n$.

Pokud však porovnáváme funkce n^n a $n!$ bez aplikace logaritmu, pak se funkce asymptoticky liší o $sqrt(n)$. Zvědavější z vás si mohou vyhledat Stirlingův vzorec (Stirling's approximation), jenž vyjadřuje přesnější odhad funkce faktoriál.

Proč je 2^{2^n} rychlejší něž n^n? Vezměme velké n. Jak se nám funkce změní pro $n+1$? $2^{2^{(n+1)}} = 2^{2^n\cdot 2} = (2^{2^n})^2$, tj. umocní se na druhou. Ukážeme, že n^n se o tolik nezmění, tj. $(n+1)^{(n+1)} < (n^n)^2$. $(n+1)^{(n+1)} = n^{(n+1)} + \binom{n}{1}n^n + \cdots + \binom{n}{i}n^{(n+1-i)} + \cdots + n^0$. Všimněme si, že $\binom{n}{i} < \frac{n!}{(n-i)!} = n\cdot(n-1)\cdots(n-(i-1)) < n^i$. Proto $(n+1)^{(n+1)} < (n+2)\cdot n^{(n+1)} = n\cdot(n+2)\cdot n^n < n^n\cdot n^n = (n^n)^2$.

3.4 Dokažte, že platí následující tvrzení:

$$2^{n+1} \in \mathcal{O}\left(\frac{3^n}{n}\right)$$

Uvědomte si vztah mezi $\mathcal{O}(2^n)$ a $2^{\mathcal{O}(n)}$.

Důkaz provedeme pomocí matematické indukce. Chceme ukázat, že pro všechna $n \geq 7$ platí

$$2^{n+1} \leq \frac{3^n}{n}$$

Pro $n = 7$: $2^{n+1} = 2^8 = 256$, $\frac{3^n}{n} = \frac{3^7}{7} = 312$. Tedy tvrzení platí.

Pro $n \geq 7$: Předpokládejme, že $2^{n+1} \leq \frac{3^n}{n}$ a chceme dokázat $2^{n+2} \leq \frac{3^{n+1}}{n+1}$. Z předpokladu dostáváme $2^{n+2} = 2 \cdot 2^{n+1} \leq 2 \cdot \frac{3^n}{n}$ a potřebujeme ještě ukázat, že $2 \cdot \frac{3^n}{n} \leq \frac{3^{n+1}}{n+1}$. Abychom to lépe viděli, nerovnost zjednodušíme, tj. vynásobíme obě strany $\frac{n(n+1)}{3^n}$. Dostaneme $2 \cdot (n+1) \leq 3n$, což je $2 \leq n$, a to pro $n \geq 7$ platí.

3.5 Určete časovou složitost následujících algoritmů.

a) Určete složitost algoritmu na základě délky pole A.

Procedura Printer1(A, n)

vstup: pole A délky n

```
1 for i ← 1 to 100000 do
2     if i < n then
3         print A[i]
4     fi
5 od
```

Složitost procedury Printer1 je v $\Theta(1)$, protože cyklus proběhne vždy 100000krát bez ohledu na velikost n.

b) Určete složitost algoritmu na základě délky pole A.

Procedura Printer2(A, n)

vstup: pole A délky n

```
1 for i ← 1 to n − 1 do
2     for j ← i to i + 1 do
3         print A[j]
4     od
5 od
```

Složitost procedury Printer2 je v $\Theta(n)$, protože vnitřní cyklus proběhne vždy pouze dvakrát.

c) Určete časovou složitost algoritmu Maximum.

Funkce Maximum(A, n)

vstup: pole A délky n
výstup: maximum z pole A

```
1 max ← A[1]
2 for i ← 2 to n do
3     if A[i] > max then
4         max ← A[i]
5     fi
6 od
7 return max
```

Složitost algoritmu Maximum je v $\mathcal{O}(n)$. Řádek 1 je proveden v čase $\mathcal{O}(1)$. Stejně tak řádky 3 a 4 jsou provedeny v čase $\mathcal{O}(1)$. Cyklus (řádky 2–4) se provede $(n-1)$-krát, tedy v čase $\mathcal{O}(n)$, protože algoritmus musí projít celé pole A.

d) Určete časovou složitost následujícího algoritmu, který vrátí součin dvou přirozených čísel y a z.

Funkce Multiply(y, z)

vstup: $y \in \mathbb{Z}$, $z \in \mathbb{Z}$
výstup: součin $y \cdot z$

```
1 x ← 0
2 while z > 0 do
3     if z is odd then
4         x ← x + y
5     fi
6     y ← 2 · y
7     z ← ⌊z/2⌋
8 od
9 return x
```

Jakou časovou složitost mají jednotlivé aritmetické operace?

Časová složitost se počítá vzhledem k délce vstupu. Pokud tedy sčítáme nebo odčítáme dvě n bitů dlouhá čísla, provedeme n sčítání, respektive odečítání, pro dvojice číslic stejného řádu. Časová složitost sčítání a odčítání je tedy lineární.

Násobení, které umíme ze základní školy, má kvadratickou složitost, jelikož každý řád prvního čísla musíme vynásobit všemi řády druhého čísla. Stejnou časovou složitost má i dělení.

Určit obecně časovou složitost tohoto algoritmu by bylo mnohem komplikovanější. Vystačíme-li si s čísly v rozsahu typu integer, pak se dá říci, že procesor provádí dané operace v konstantním čase. Pokud neuvedeme v příkladě jinak, předpokládáme časovou složitost matematických operací za konstantní.

Prvně rozeberme, kolikrát se které řádky volají.

Řádek č. 1 je volán jen jednou. V cyklu se v každém průchodu zmenšuje hodnota z na polovinu, klesá tedy logaritmicky. Podle počtu „instrukcí“ by tedy šlo usoudit, že algoritmus

pracuje v čase $\mathcal{O}(\log z)$. Uvědomte si ale, že ve skutečnosti aritmetické operace netrvají konstantní čas.

3.6 Určete časovou složitost různých verzí algoritmu Power.

a) Základní iterativní verze:

Funkce PowerIter($base, exp$)
vstup: $base \in \mathbb{R}$, $exp \in \mathbb{N}$
výstup: $base^{exp}$

```
1 output ← 1
2 for i ← 1 to exp do
3     output ← output · base
4 od
5 return output
```

Algoritmus vypočítá mocninu pomocí postupného přinásobování čísla *base* k průběžnému výsledku, což vede k lineární složitosti $\Theta(n)$.

b) Binární umocňování:

Funkce PowerBin($base, exp$)
vstup: $base \in \mathbb{R}$, $exp \in \mathbb{N}_0$
výstup: base^{exp}

```
1 output ← 1
2 while exp > 0 do
3     if exp mod 2 = 1 then
4         output ← output · base
5     fi
6     base ← base · base
7     exp ← exp/2
8 od
9 return output
```

Algoritmus má logaritmickou složitost $\Theta(\log n)$ díky půlení čísla *exp*. Nepřinásobujeme tedy postupně po jednom *base*, ale výsledek se po mocninách kumuluje v proměnné *output*.

c) Liší se nějak tato rekurzivní implementace (z hlediska složitosti) od 1. algoritmu?

Funkce PowerRecursive($base, exp$)
vstup: $base \in \mathbb{R}$, $exp \in \mathbb{N}_0$
výstup: base^{exp}

```
1 if exp = 0 then
2     return 1
3 else
4     return base · PowerRecursive(base, exp − 1)
5 fi
```

Rekurzivní implementace má stejnou asymptotickou časovou složitost. Liší se však doba výpočtu (alespoň ve většině programovacích jazyků). Procesor si totiž musí udržovat zásobník funkčních volání. Rekurzivní řešení bývají až na výjimky pomalejší (jen co se času týče, ne složitosti) než řešení bez rekurze. Výhodou bývá snazší (stručnější) zápis rekurze.

Změřte čas výpočtu volání různých implementací funkce Power. Jaké jsou vhodné hodnoty pro testování?

3.7 Následující 3 funkce vrací n-té číslo z Fibonacciho posloupnosti. Určete jejich časovou složitost. Časovou složitost základních matematických operací (sčítání, násobení...) pro zjednodušení uvažujte konstantní.

Fibonacciho posloupnost je nekonečná řada čísel, kde je každé číslo součtem dvou předchozích. 0. člen řady je 0, první je 1.

Prvních 10 členů je 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Pan Usměvavý dodává: Fibonacciho posloupnost můžeme také definovat pomocí populace králíčků. Předpokládáme, že první měsíc se narodí jediný pár, nově narozené páry jsou produktivní od druhého měsíce svého života, každý měsíc zplodí každý produktivní pár jeden další pár a pro reálnost králíci nikdy neumírají, nejsou nemocní atd. Pak n-té Fibonacciho číslo popisuje počet párů v n-tém měsíci.

a) Základní rekurzivní varianta výpočtu finonacciho čísel:

Funkce FibRecursive(n)

vstup: $n \in \mathbb{N}$

výstup: n-té číslo z Fibonacciho posloupnosti

```
1 if n ≤ 1 then
2     return n
3 else
4     return FibRecursive(n − 1) + FibRecursive(n − 2)
5 fi
```

Funkce FibRecursive patří do $2^{\mathcal{O}(n)}$. Důkaz:

Označme si časovou složitost funkce zavolané s parametrem n jako $T(n)$. Jelikož v cyklu provádíme 4 aritmetické operace (porovnávání, 2 odečítání a jedno sčítání), lze výslednou časovou složitost zapsat jako

$$T(n) = T(n-1) + T(n-2) + 4,$$

dále hodnoty $T(0) = T(1) = 1$.

Pro zjednodušení také počítejme dolní hranici, tím, že předpokládáme $T(n-1) \approx T(n-2)$.

Tedy $T(n) = 2T(n-2) + 4$, což můžeme zjednodušit na výraz

$$T(n) = 2^{\frac{n}{2}} \cdot T(0) + (2^{\frac{n}{2}} - 1) \cdot 4.$$

Zde už máme exponenciální člen, který je z asymptotického hlediska převládající.

b) Iterativní podoba:

Funkce FibIter(n)

vstup: $n \in \mathbb{N}$
výstup: n-té číslo z Fibonacciho posloupnosti

```
1 lower ← 0
2 higher ← 1
3 for i ← 1 to n do
4     tmp ← lower + higher
5     lower ← higher
6     higher ← tmp
7 od
8 return higher
```

Funkce FibIter patří do $\mathcal{O}(n)$. V každém průchodu for cyklu se rozdíl $n - i$ sníží o jedna a když dosáhne nuly, cyklus končí. Operace v cyklu dle zadání považujeme za konstantní.

c) Pomocí zlatého řezu:

Funkce FibConst(n)

vstup: $n \in \mathbb{N}$
výstup: n-té číslo z Fibonacciho posloupnosti

```
1 phi ← (1+√5)/2 // hodnota zlatého řezu, kterou si můžeme navíc předpočítat
2 return ⌊phi^n/√5 + 1/2⌋
```

Funkce FibConst žádný cyklus neobsahuje, pouze volá funkce Power a Sqrt při výpočtu mocniny a odmocniny, výsledná časová složitost tedy záleží na časové složitosti těchto funkcí. Pokud si algoritmus naimplementujete, možná budete překvapeni, že není výrazně rychlejší než předchozí iterativní algoritmus. To je dáno prací s typy pro reálná čísla, na nichž jsou operace výrazně pomalejší. Výpočet by mohl být rychlejší až na velmi vysokých hodnotách, pro které už je již kvůli zaokrouhlování mírně nepřesný.

Daliborek vzkazuje: Pojem Fibonacciho čísel lze zobecnit spoustou způsobů. Lze zadefinovat pro záporná, reálná a komplexní čísla, lze zadefinovat i nad vektorovým prostorem a Abelovou grupou. Zadefinováním jiných základů, popřípadě jiného sčítání, můžeme získat jiné posloupnosti, například Lucasova čísla. Pokud budeme místo dvojice předchozích čísel posloupnosti sčítat čísel více, získáme tribonacciho, tetranacciho... čísla.

3.8 Naměřte si dobu běhu různých verzí algoritmů Power a Fib. Programátorskou přípravu najdete ve studijních materiálech – C a Python. Vaším úkolem je naimplementovat různé podoby algoritmů, dobu běhu naměří připravená funkce main.

Následující příklady jsou vhodné na domácí studium.

3.9 Rozhodněte, zda jsou následující tvrzení pravdivá.

1. $3n^2 + 5n \in \mathcal{O}(n^2 \log n)$
2. $3n^2 + 5n \in \mathcal{O}(n)$
3. $3n^2 + 5n \in \Omega(n \log n)$
4. $3n^2 + 5n \in \Omega(0.05^n)$
5. $3n^2 + 5n \in \Theta(n^{2.05})$

3.10 Do tabulky doplňte symboly P a N podle toho, zda platí (P) nebo neplatí (N), že $3n^2 + 5n \in \Omega(f)$, resp. $3n^2 + 5n \in \mathcal{O}(f)$ a $3n^2 + 5n \in \Theta(f)$, pro funkci f zadanou v prvním sloupci tabulky.

f	$3n^2 + 5n \in \Omega(f)$	$3n^2 + 5n \in \mathcal{O}(f)$	$3n^2 + 5n \in \Theta(f)$
n			
n^2			
$n^2 \log n$			
n^3			
3^n			

3.11 Dokažte, že platí následující tvrzení:

$$\log(n) \in \mathcal{O}(n).$$

Pomocí matematické indukce ukážeme, že pro všechna $n \geq 1$ platí

$$\log n \leq n.$$

Pro $n = 1$: $\log n = 0$ a tedy tvrzení platí.
Pro $n \geq 1$: Předpokládejme, že $\log n \leq n$. Chceme dokázat, že $\log(n + 1) \leq n + 1$.

$$\log(n + 1) \leq \log 2n = \log 2 + \log n \leq 1 + n$$

3.12 Mějme algoritmus s časovou složitostí vyjádřenou funkcí $T(n)$. Rozhodněte, zda pro následující definice funkce $T(n)$ platí:

a)

$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{jinak} \end{cases}$$

Platí, že $T(n) = \mathcal{O}(n \log n)$?

b)

$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 2T(n-1) + n & \text{jinak} \end{cases}$$

Platí, že $T(n) = \mathcal{O}(n \log n)$?

c)

$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ T(\lfloor n/2 \rfloor) + 1 & \text{jinak} \end{cases}$$

Platí, že $T(n) = \mathcal{O}(\log n)$?

3.13

a) Mějme algoritmus se složitostí v $2^{\Theta(n)}$. Při jeho řešení na starém počítači trval výpočet pro $n = 36$ jeden den. Nyní máme k dispozici nový počítač, který je 1000krát rychlejší. Určete, pro jak velké n lze s novým počítačem algoritmus spustit, aby doba výpočtu nepřesáhla jeden den.

Nejprve odhadneme dobu jedné iterace.

$$i \cdot 2^{36} = 60 \cdot 60 \cdot 24 \ s$$

$$i = \frac{86400}{2^{36}} \ s$$

$$i \approx 1.3 \cdot 10^{-6} \ s$$

Nyní dobu jedné iterace vydělíme 1000, protože máme 1000krát rychlejší počítač.

$$i' = 1.3 \cdot 10^{-9} \ s$$

$$i' \cdot 2^n = 60 \cdot 60 \cdot 24 \ s$$

$$2^n = \frac{86400 \ s}{1.3 \cdot 10^{-9} \ s}$$

$$n = \left\lfloor \log_2 \frac{86400}{1.3 \cdot 10^{-9}} \right\rfloor$$

$$n = 45$$

b) Jaký bude rozdíl při $n = 10$ pro algoritmy se složitostí v $\Theta(n!)$ a v $\Theta(n^n)$? Výsledek odhadněte.

Pro algoritmus se složitostí v $\Theta(n!)$ bychom pro $n = 12$ zvládli výpočet provést za méně než 3,5 hodiny, zatímco pro $n = 13$ bychom potřebovali 1,716 dne.

Pro algoritmus se složitostí v $\Theta(n^n)$ bychom pro $n = 12$ zvládli výpočet provést za téměř 21,5 hodiny, zatímco pro $n = 13$ by výpočet trval více než 30 dní.

3.14

a) Určete časovou složitost následujícího algoritmu pro vyhledání klíče v zadaném poli (vstupní posloupnost je setříděná). Algoritmus binárního vyhledávání (vrátí index nalezeného prvku v poli D):

Funkce BinarySearch(D, l, r, k)

vstup: D je setříděné pole, l a r jsou levý a pravý konec pole, k je hledaný klíč
výstup: nalezený index výskytu čísla k, -1 s případě, že se k v poli nevyskytuje

```
1  if l > r then
2      return −1;
3  else
4      m ← (l+r)/2
5      if k < D[m] then
6          return BinarySearch (D, l, m − 1, k)
7      else if k > D[m] then
8          return BinarySearch (D, m + 1, r, k)
9      else
10         return m
11     fi
12 fi
```

Algoritmus na základě půlení intervalu nalezne daný prvek v čase $\mathcal{O}(\log n)$, kde n je délka prohledávaného pole.
Intuitivní důkaz: algoritmus končí, když je rozdíl indexů l a r roven nule. V každém průchodu cyklu se rozdíl dělí dvěma, v horším případě je vždy zaokrouhlen dolů. Počet těchto dělení vyjádříme pomocí funkce $\log_2 n$.

b) Naprogramujte si algoritmus vyhledávání pomocí binárního půlení v iterativní podobě.

Karlík varuje: Vyhledávání pomocí binárního půlení lze použít pouze na uspořádané pole. Nelze jej efektivně použít ani na zřetězený seznam, ani na neuspořádané pole.

Existuje i efektivnější vyhledávaní tzv. InterpolationSearch, které funguje na způsob hledání ve slovníku. Algoritmus interpoluje, kde přibližně by měl posloupnost rozdělit, na základě hraničních hodnot.

Průměrná časová složitost interpolačního vyhledávaní je $\log(\log(n))$, ale v nejhorším případě může být složitost až v $\mathcal{O}(n)$ a to v případě, že hodnoty rostou exponenciálně. Algoritmus vypadá následovně:

Funkce InterpolationSearch(D, k)

vstup: D je setříděné pole, k je hledaný klíč

výstup: nalezený index výskytu čísla k, -1 s případě, že se k v poli nevyskytuje

```
1  low ← 1
2  high ← |D|
3  while D[low] ≤ k ∧ D[high] ≥ k do
4      mid ← low + ((k − D[low]) · (high − low))/(D[high] − D[low])
5      if D[mid] < k then
6          low ← mid + 1
7      else if D[mid] > k then
8          high ← mid − 1
9      else
10         return mid
11 od
12 if D[low] = k then
13     return low
14 else
15     return −1;
```

Tabulka časů výpočtu algoritmů o složitostech $\log n$, n, n^2, 2^n a n^n pro vstup délky 10, 20, 50 a 1000. Předpokládejme, že jedna iterace algoritmu trvá 1 μs.

	10	20	50	1000
$\log n$	0,000001 s	0,000001 s	0,000002 s	0,000003 s
n	0,00001 s	0,00002 s	0,00005 s	0,001 s
n^2	0,0001 s	0,0004 s	0,0025 s	1 s
2^n	0,001024 s	1,048576 s	35,7 let	$3,4 \cdot 10^{287}$ let*
n^n	2,8 hod	$3 \cdot 10^{12}$ let*		

* Stáří vesmíru je odhadováno na $13,7 \cdot 10^9$ let.

Kapitola 4

Návrh algoritmů

Nejprve se studenty zopakujte definice těchto základních pojmů z přednášky:

Iterativní algoritmus je takový, který spočívá v opakování určité své části (bloku).

Rekurzivní algoritmus opakuje kód prostřednictvím volání sebe sama (obvykle na podproblémech menší velikosti). Každý rekurzivní algoritmus lze převést do iterativní podoby.

Rozděluj a panuj je přístup dělení problému na menší podproblémy. Sjednocením částečných řešení vyřešíme původní problém. Takové algoritmy se často volají rekurzivně.

Rekurzivní strom reprezentuje jednotlivá zanoření rekurze, kde vrcholy stromu vyjadřují složitost jednotlivých rekurzivních volání. Každý vrchol tedy reprezentuje součet složitostí všech hlubších zanoření rekurze a složitosti své. Sčítáním jednotlivých úrovní dostáváme výslednou složitost rekurze.

Paní Bílá připomíná: Dalším typem algoritmů jsou Greedy algorithms, které se opravdu nejmenují podle pana Greedyho. Tyto algoritmy hledají lokálně nejlepší řešení, které může být globálním řešením. Z toho pochází označení greedy, což do češtiny překládáme jako hladový, tedy hladové algoritmy.

4.1 Jakými algoritmy řešíte tyto problémy ze života a o jaký typ algoritmu se jedná?

Obhájit se dá většina přístupů, důležitá je tedy argumentace, proč to student zařadil do dané skupiny.

a) Vyhledávání telefonního čísla v seznamu (zlaté stránky).

> Zde se nabízí spousta možností.
>
> Pokud budeme seznamem listovat tak dlouho, dokud nebudeme na správné straně, pak můžeme říci, že problém řešíme iterativním algoritmem.
>
> Rychlejší postup by byl pomocí binárního vyhledávání, které se dá považovat za rekurzivní algoritmus.

b) Proces chůze do schodů.

Typicky jde o iterativní algoritmus – dokud nejsem nahoře, udělám krok.

c) Seřazování testů podle jména studentů ve 4 opravujících.

Vhodným řešením by bylo dát každému z opravujících menší hromádku, kterou by si každý seřadil samostatně. Spojení hromádek v jednu by proběhlo volením prvního jména vždy z vrcholu čtveřice hromádek. Takový algoritmus je typu rozděluj a panuj. Kdyby tak dělili i každou podhromádku, tak by se jednalo o spojení rekurze a rozděluj a panuj (merge sort).

d) Debugování programu.

Debugování je příkladem problému, který zahrnuje více postupů, a tedy i typů algoritmů.

Krokování programu je iterativní algoritmus.

Často je jedna chyba vyvolávaná jinou. Pokud tedy pro opravu chyby A potřebujete opravit chybu B a pro opravení B je nutné opravit C. . . , pak postupujete rekurzivně.

Pokud něco vyvíjíte ve více lidech, pak můžete hledat problém i pomocí rozděluj a panuj, kdy každý hledá problém ve své části kódu, dále dvojice programátorů hledají problém v komunikaci mezi jejich částmi kódu a větší skupinky hledají ve stále větším společném kódu.

Zadání příkladu není příliš konkrétní. Studenti vám tedy mohou navrhnout řešení s pomocným řetězcem, s takovým řešením se nespokojte.

Můžete se studenty diskutovat paměťovou složitost. „Má rekurzivní algoritmus lineární extrasekvenční složitost?“ V zásobníku bude $\lfloor \frac{length}{2} \rfloor$ volání. Ale při optimalizaci překladačem může být tail rekurze odstraněna a složitost redukována. Pokud chcete, je zde prostor pro vaše povídání.

4.2 Navrhněte algoritmus Reverse, který dostane na vstup řetězec znaků a obrátí v něm pořadí všech znaků.

Zkuste problém řešit pomocí různých technik návrhu algoritmů.

Iterativní řešení může vypadat například takto:

Procedura ReverseIterative($A, length$)

vstup: A je řetězec délky $length$ indexován od 1
výstup: obrácený řetězec

```
1 for i ← 1 to ⌊length/2⌋ do
2     Swap(A_i, A_{length−i+1})
3 od
4 return A
```

Rekurzivní řešení může vypadat například takto:

Procedura ReverseRecursive($A, from, to$)

vstup: A je řetězec v rozsahu $from$, to
výstup: obrácený řetězec

```
1 if from ≥ to − 1 then
2     return A
3 fi
4 return A_to + ReverseRecursive(A, from + 1, to − 1) + A_from
```

Pravděpodobně budete muset studentům zopakovat násobení matic. Vytvořte společně formuli, která algoritmus popisuje. Z této formule by studenti měli umět zkonstruovat algoritmus (tuto dovednost je chceme naučit).

Pak teprve pokračujte vstupními a výstupními podmínkami a složitostí, spolu s dolním odhadem složitosti v části b). V té můžete studenty vyzvat, že „máte algoritmus na násobení matic v $\mathcal{O}(n)$“, aby vám vyvrátili, že váš algoritmus nemůže fungovat obecně správně.

4.3

a) Navrhněte algoritmus pro násobení matic. Určete vstupní a výstupní podmínky algoritmu a určete jeho časovou složitost.

Karlík varuje: Pokud je vaše kladivo C++, všechno vypadá jako palec.

Intuitivní algoritmus násobení matic může vypadat následovně:

Procedura MultiplyMatrix(A, B, n)

vstup: A je matice $n \times n$, B je matice $n \times n$
výstup: matice C rozměrů $n \times n$

```
1 for i ← 1 to n do
2     for j ← 1 to n do
3         C[i, j] ← 0
4         for k ← 1 to n do
5             C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6         od
7     od
8 od
9 return C
```

Vstupní a výstupní podmínky jsou:

$\varphi(A, B, n) \equiv A$ je matice rozměrů $n \times n$ a B je matice rozměrů $n \times n$.
$\psi((A, B), C, n) \equiv C$ je matice rozměrů $n \times n$ rovnající se násobku matic A a B.

Časovou složitost algoritmu můžeme určit ze 3 zanořených cyklů. Jelikož každý z cyklů proběhne n-krát, ze zanoření cyklů vidíme, že každý běh prvního cyklu vynutí n běhů druhého cyklu a ten taky vynutí n běhů třetího cyklu. V součtu máme tedy $n \cdot n \cdot n$ běhů. Aritmetické operace v cyklech můžeme brát jako operace s konstantní časovou složitostí, což nám ve výsledku dává složitost je $\Theta(n^3)$.

b) V současnosti víme o algoritmu, který násobí matice s časovou složitostí $\Theta(n^{2.3727})$. Proč je násobení matic v $\Omega(n^2)$?

Paní Bílá připomíná: Jeden z komplikovanějších algoritmů násobení matic je Strassenův algoritmus, který provádí násobení asymptoticky rychleji než náš naivní algoritmus. Více se o tomto algoritmu můžete z vlastní iniciativy dozvědět v doporučené literatuře k tomuto předmětu.

Jelikož algoritmus pro násobení dvou matic velikosti $n \times n$ musí zpracovat alespoň $2 \times n^2$ vstupů, je jasné, že algoritmus nikdy nebude lepší než se složitostí n^2, tudíž násobení matic je v $\Omega(n^2)$.

Formální důkaz v řešení po studentech nechtějte. Soustřeďte se na hledání míst, kde lze udělat v algoritmu chybu jak z pohledu konvergence, tak z pohledu správného výpočtu.

4.4

a) Najde následující rozděluj a panuj algoritmus maximální prvek v poli A délky n? Pokud ano, dokažte korektnost algoritmu. Pro jednoduchost můžete předpokládat, že n je mocnina 2. Funkce $\textsc{Max}(a, b)$ vrací větší prvek z a a b.

Funkce $\textsc{Maximum}(x, y, A)$

vstup: x a y jsou indexy pole A o délce $y - x + 1 > 0$
výstup: maximum z pole A mezi indexy x a y

```
1 if y − x ≤ 1 then
2     return Max(A[x], A[y])
3 else
4     max_1 ← Maximum(x, ⌊(x + y)/2⌋, A)
5     max_2 ← Maximum(⌊(x + y)/2⌋ + 1, y, A)
6     return Max(max_1, max_2)
7 fi
```

Intuitivní řešení: potřebujeme zkontrolovat, zdali je algoritmus konvergentní, to znamená, že musíme ověřit, zdali dojde k zastavení rekurze. To je však zajištěno podmínkou na řádcích 1 a 2 a postupným snižováním argumentů, se kterým se funkce volá. Záleží tedy jen na tom, zdali kvůli zaokrouhlení nedojde k opakovanému volání se stejnými argumenty, k čemuž však dojde jedině v případě, že $x = y - 1$. V takovém případě se však splní podmínka, která rekurzivní zanořování ukončí.

Dalším krokem je ověření, zdali algoritmus počítá, co má. V našem případě by byl problém, kdyby algoritmus přeskočil některé hodnoty v poli. K tomu však nedojde, protože rekurze se volá na interval, který je rozdělen korektně.

Formálně: následující řešení platí i pro n, které není mocnina 2. Velikost problému je počet vstupů v poli, tedy $y - x + 1$. Dokážeme indukcí, že pro $n = y - x + 1$ vrátí $\textsc{Maximum}(x, y)$ maximální hodnotu z pole A. Algoritmus je zjevně korektní pro $n \leq 2$. Nyní předpokládejme $n > 2$ a to, že $\textsc{Maximum}(x, y)$ vrátí maximální hodnotu v $A[x, y]$ vždy, když $y - x + 1 < n$. Abychom mohli využít indukční předpoklad pro první rekurzivní volání, musíme dokázat $\lfloor (x + y)/2 \rfloor - x + 1 < n$. Existují dva případy. Když $y - x + 1$ je sudé, potom $y - x$ je liché, a tudíž $y + x$ je liché. Proto

$$\left\lfloor \frac{x+y}{2} \right\rfloor - x + 1 = \frac{x+y-1}{2} - x + 1 = \frac{y-x+1}{2} = \frac{n}{2} < n.$$

Nerovnost platí pro $n > 2$. Jestli je $y - x + 1$ liché, potom $y - x$ je sudé. Pak

$$\left\lfloor \frac{x+y}{2} \right\rfloor - x + 1 = \frac{x+y}{2} - x + 1 = \frac{y-x+2}{2} = \frac{n+1}{2} < n.$$

Abychom mohli použít indukční předpoklad na druhé rekurzivní volání, musíme dokázat, že $y - (\lfloor (x+y)/2 \rfloor + 1) + 1 < n$. Znovu nastávají dva případy. Když $y - x + 1$ je sudé, pak

$$y - \left(\left\lfloor \frac{x+y}{2} \right\rfloor + 1 \right) + 1 = y - \frac{x+y-1}{2} = \frac{y-x+1}{2} = \frac{n}{2} < n.$$

Pokud je $y - x + 1$ liché, pak

$$y - \left(\left\lfloor \frac{x+y}{2} \right\rfloor + 1 \right) + 1 = y - \frac{x+y}{2} = \frac{y-x+1}{2} - \frac{1}{2} = \frac{n}{2} - \frac{1}{2} < n.$$

Funkce Maximum rozděluje pole do dvou částí. Pomocí indukčního předpokladu rekurzivní volaní korektně najdou maximum v těchto částech. A jelikož funkce vrátí maximum ze dvou maxim, vrátí i korektní výsledek.

Po určení složitosti pomocí Master theorem doporučujeme analýzu dalších algoritmů. Můžete například analyzovat ReverseRecursive ze začátku hodiny a vyhledávání pomocí binárního půlení.

Pokud máte dostatek času, pak můžete určovat složitosti různých rekurentních funkcí z bonusových příkladů (10 a 11) a poté odhadovat, o jaký by se mohlo jednat algoritmus.

b) Zapište rekurentní rovnici pro nejhorší případ počtu porovnání při volání Maximum$(1, n)$. Následně tuto rovnici vyřešte a určete z ní časovou složitost vzhledem k délce vstupu. Opět můžete předpokládat, že n je mocnina 2.

Pro vyřešení rekurentní rovnice můžete použít master method z přednášky. Díky němu víme, že platí následující. Nechť $a \geq 1$, $b > 1$ a $d \geq 0$ jsou konstanty a $T(n)$ je definovaná na nezáporných číslech rekurentní rovnicí:

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ aT(n/b) + \mathcal{O}(n^d) & \text{jinak} \end{cases}$$

Pak platí:

$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{pokud } a < b^d \\ \mathcal{O}(n^d \log n) & \text{pokud } a = b^d \\ \mathcal{O}(n^{\log_b a}) & \text{pokud } a > b^d \end{cases}$$

Nechť $T(n)$ je počet porovnání ve funkci Maximum(x, y), kde $n = y - x + 1$ je zpracovávaná část pole. Pokud n je mocnina 2, $T(1) = 0$ a pro všechna $n > 1$ a mocniny 2 nám rekurentní rovnice vyjadřuje, že v každém zanoření rekurze 2krát provádím rekurzivní volání, které pokaždé rozdělí vstupní interval na půlku, z čehož máme část $2T(n/2)$. Zároveň musíme přičíst 1, která vyjadřuje konstantní složitost Max(a, b) (hledání maxima ze dvou prvků). Dohromady získáváme rovnici $T(n) = 2T(n/2) + 1$. Odkud $a = 2, b = 2, d = 0$. Tedy $a > b^d$ a master theorem nám říká, že složitost vyjádřená rekurentní rovnicí bude $\mathcal{O}(n^{\log_b a})$. Logaritmus v rovnici po dosazení nám dá $\log_2 2 = 1$. Vyřešením získáme výsledek $T(n) = \mathcal{O}(n)$. Časová složitost nám vlastně popisuje počet vykonaných porovnání za běhu algoritmu.

4.5 Napište rekurentní rovnici pro řešení problému batohu. V tomto problému je vaším úkolem vybrat objekty s nejvyšší možnou hodnotou tak, aby se vešly do batohu.

Vstupem je tedy n objektů, pro které zavedeme proměnné $(x_1, \ldots, x_n)$, které značí, zdali jsme objekt do batohu vložili. Dále je vstupem maximální váha V, tedy kolik se toho do batohu vleze. Každému objektu i je přiřazena váha v_i a cena c_i. Úkolem je vybrat objekty s nejvyšší možnou cenou tak, aby jejich součet vah nepřekročil limit V. Formálně definované:

$$\text{Maximalizujte} \sum_{i=1}^{n} c_i x_i \text{ přičemž platí } \sum_{i=1}^{n} v_i x_i \leq V,\ x_i \in \{0, 1\}.$$

Vaším úkolem je napsat rekurentní rovnici pro zmíněný problém a na základě ní vymyslete rekurzivní algoritmus pro řešení problému. Určete jeho složitost a následně ho naprogramujte.

Paní Bílá připomíná: Problém je všeobecně známý jako problém batohu (Knapsack problem). V navazujících předmětech IV003 a IA101 můžete poznat spoustu různých algoritmů řešících tento problém.

Problém můžeme rozdělit na menší podproblémy a následně z jejich řešení zkonstruovat původní řešení. Intuitivně:

1. Vezmeme prvek x_i a rozhodneme, jestli ho ještě můžeme přidat k vybraným prvkům.
2. Řešíme podproblém na prvcích $(x_1, \ldots, x_{i-1})$ pro oba případy, tedy že jsme x_i vybrali i nevybrali.
3. Vrátíme optimální řešení.

Postup můžeme popsat rekurentní rovnicí, která vrátí optimální řešení následovně:

$$\text{OPT}(i, V) = \begin{cases} 0 & \text{pokud } i = 0 \\ \text{OPT}(i-1, V) & \text{pokud } i \geq 1 \text{ a } V < v_i \\ \max\{c_i + OPT(i-1, V - v_i), OPT(i-1, V)\} & \text{pokud } i \geq 1 \text{ a } V \geq v_i \end{cases}$$

Což dokážeme algoritmicky popsat rekurzí:

Funkce ProblemBatohu(v, c, V, i)

vstup: pole v obsahující n vah objektů, pole c obsahující n cen objektů, V maximální váha, rozhodujeme se, zdali do batohu vložit objekt s indexem $i \in \mathbb{N}$

výstup: optimální cena objektů v batohu

```
1 if i = 0 then
2     return 0
3 else if i ≥ 1 ∧ V < v[i] then
4     return ProblemBatohu(v, c, V, i − 1)
5 else
6     return max(c[i] + ProblemBatohu(v, c, V − v[i], i − 1), ProblemBatohu(v, c, V, n − 1))
```

Prozkoumáním všech možných řešení dostáváme algoritmus o složitosti $2^{\mathcal{O}(n)}$.

Pomocí rekurze je možné psát kompaktní a elegantní programy, které ale mohou přestat fungovat za ne zcela zjevných okolností. Proto bychom si při psaní rekurze měli dávat pozor na následující běžné chyby:

- **Chybějící hraniční podmínka.** Následující funkce opakovaně volá sebe a nikdy nevrací hodnotu zpět.

  Funkce $f(n)$

  ```
  1 return f(n − 1) + 1/n;
  ```

- **Negarantovaná konvergence.** Další běžný problém je volat v rekurzivní funkci rekurzivní volání pro řešení podproblému, který není menší.

  Funkce $f(n)$

  ```
  1 if n = 1 then
  2     return 1
  3 fi
  4 return f(n) + 1/n;
  ```

  Rekurzivní funkce f se dostane do nekonečné smyčky, když je volaná s jinou hodnotou jinou než 1.

- **Překročení dostupné paměti.** Při volání rekurzivní funkce je potřebné si každé volání uložit na zásobník. Proto jsme při používaní rekurze omezeni velikostí zásobníku. I správně implementovaný program může spadnout jen proto, že mu došla dostupná paměť. Proto se při práci s většími daty preferují iterativní formy algoritmů.

- **Nadměrné počítání.** Při nepozorném psaní rekurze se může stát, že při pokusu o napsání čitelného algoritmu pro jednoduchý problém napíšeme až exponenciální algoritmus. Problém může nastat když při počítání opakovaně počítáme stejné hodnoty.

Například pro definici Fibonacciho čísel $f_n = f_{n-1} + f_{n-2}, n \geq 2$ s $f_0 = 0$ a $f_1 = 1$ může triviální implementace vypadat následovně:

Funkce $f(n)$

```
1 if n = 0 then
2     return 0
3 fi
4 if n = 1 then
5     return 1
6 fi
7 return f(n − 1) + f(n − 2);
```

Což je exponenciální algoritmus, protože některé větve rekurze počítáme opakovaně (pro $f(n)$ spočteme $f(n-2)$ a pro $f(n-1)$ opět počítáme $f(n-2)$ atd.).

Jako rozšiřující úlohy lze studentům ukázat Robotanika. Doporučujeme vytvořit studentům vlastní třídu a zadat jim vytvořenou sadu problémů na rekurzi.

Díky jednoduchému modelu není složité se naučit Robotanika ovládat, příklady jsou přínosné i do pozdější fáze semestru – problémy obsahují i průchody stromů.

Pokud chcete studentům zadat bonusové příklady, pak můžete probrat třeba příklad 12. Pěkně vede k zamyšlení o vyhledávání pomocí binárního půlení.

4.6 Jako programátorské cvičení implementujte vyhledávaní pomocí binárního půlení v rekurzivní podobě. Následně jej převeďte na nerekurzivní podobu. Jako druhé cvičení implementujte příklad na výpočet MIN a MAX. Podklady pro programování najdete už tradičně ve studijních materiálech: C a Python.

Následující příklady jsou vhodné na domácí studium.

4.7 Navrhněte a poté naprogramujte rekurzivní algoritmus, který vypočítá hodnotu kombinačního čísla. Využijte znalost rekurzivní formule:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ pro } n, k : 1 \leq k \leq n-1$$

s hraničními hodnotami:

$$\binom{n}{0} = \binom{n}{n} = 1 \text{ pro } n \geq 0.$$

4.8 Jaké znáte algoritmy pro nalezení největšího společného dělitele 2 čísel?

Jaký je rozdíl v časové složitosti „středoškolského“ algoritmu, který hledá společné prvočinitele, a Euklidova algoritmu?

Naprogramujte si Euklidův algoritmus (případně můžete implementovat oba pro možnost porovnání pomocí měření časů). O jaký typ algoritmu se jedná?

Zmíněný „středoškolský“ algoritmus, který rozkládá obě čísla na prvočinitele a z nich pak vybírá ty společné, má lineární složitost (počtem kroků vzhledem k $n = \max(a, b)$). Euklidův algoritmus patří do $\mathcal{O}(\log(n))$. Toto vyjádření složitosti vzhledem k hodnotě a ne délce vstupu je však velmi zjednodušené.

Pseudokód Euklidova algoritmu:

Funkce GCD(a,b)

vstup: $a, b \in \mathbb{N}$ jsou čísla, pro která hledáme největšího společného dělitele
výstup: největší společný dělitel a, b

```
1 while b ≠ 0 do
2     t ← b
3     b ← a mod b
4     a ← t
5 od
6 return a
```

Euklidův algoritmus patří mezi iterativní algoritmy.

Daliborek vzkazuje: Algoritmus jde napsat i hezčím způsobem (čti funkcionálním způsobem):

```
1 gcd a 0 = a
2 gcd a b = gcd b (a `mod` b)
```

4.9

Daliborek vzkazuje: Monáda je monoid v kategorii endofunktorů.

a) Navrhněte rekurzivní algoritmus pro počítání faktoriálu.

Algoritmus by šel zapsat například takto:

Procedura RecursiveFact(n)

vstup: $n \in \mathbb{N}$, jehož faktoriál hledáme
výstup: $n!$

```
1 if n ≤ 1 then
2     return 1
3 fi
4 return RecursiveFact (n − 1) · n
```

b) Převeďte předchozí rekurzivní algoritmus na iterativní.

Algoritmus by šel zapsat například takto:

Procedura IterativeFact(n)

vstup: $n \in \mathbb{N}$, jehož faktoriál hledáme
výstup: $n!$

```
1 output ← 1
2 for i ← 2 to n do
3     output ← output · i
4 od
5 return output
```

c) Porovnejte tato 2 řešení. Liší se nějak ve výkonu? Pokud ano, čím je to způsobeno?

V tomto případě velmi záleží na jazyce a překladači, který by neměl být náplní tohoto předmětu. Přesto většinou platí, že rekurzivní algoritmy jsou pomalejší, protože každé funkční volání znamená vytvoření nového rámce na zásobníku. To znamená zbytečnou režii s pamětí. Tomuto zpomalení často zabraňují překladače, které rekurzivní algoritmy přeloží do iterativní podoby (minimálně u jednodušších programů). Programátor by si toho však měl být vědom a neměl by se vždy na překladač spoléhat. Výhoda rekurzivního zápisu je často čitelnost a stručnost, má tedy smysl přemýšlet, který zápis má v dané situaci větší význam.

4.10 Vyřešte následující rekurentní rovnice. Určete těsnou asymptotickou hranici pro každou funkci ve formě $\Theta(f(n))$ pro rozpoznatelnou funkci $f(n)$. Jaké algoritmy by mohly rovnice popisovat?

1. $A(n) = A(n/2) + n$
2. $B(n) = 2B(n/2) + n$
3. $C(n) = 3C(n/2) + n$
4. $D(n) = \max_{n/3<k<2n/3}(D(k) + D(n-k) + n)$
5. $E(n) = \min_{0<k<n}(E(k) + E(n-k) + 1)$
6. $F(n) = 4F(\lfloor n/2 \rfloor + 5) + n$
7. $G(n) = G(n-1) + 1/n$
8. $H(n) = H(n/2) + H(n/4) + H(n/6) + H(n/12) + n$ [nápověda: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.]
9. $I(n) = 2I(n/2) + n/\log n$
10. $J(n) = \frac{J(n-1)}{J(n-2)}$

4.11 Vyřešte následující rekurentní funkce pro $T(1) = 1$. Předpokládejte, že n je mocnina dvou.

1. $T(n) = T(n/2) + 1$
2. $T(n) = 2T(n/2) + 1$
3. $T(n) = 2T(n/2) + n$
4. $T(n) = 4T(n/2) + 3$

4.12 Mějme pole $A[1..n]$ seřazených přirozených čísel, které bylo cyklicky posunuto o k pozic doprava. Například pole $[35, 42, 5, 15, 27, 29]$ je pole, které bylo cyklicky posunuto o $k = 2$ pozic, zatímco $[27, 29, 35, 42, 5, 15]$ bylo posunuto o $k = 4$ pozic.

Jednoduše dokážeme najít maximální prvek z pole A v čase $\mathcal{O}(n)$. Navrhněte algoritmus, který najde maximum v čase $\mathcal{O}(\log(n))$.

Navrhněte algoritmus, který vyhledá zadanou hodnotu v poli v čase $\mathcal{O}(\log(n))$.

4.13 Mějme dvoudimenzionální pole, které obsahuje čísla 0 a 1. Navrhněte efektivní rekurzivní algoritmus, který prohodí za 0 všechny 1, které sousedí pouze s 1. Prvky spolu sousedí, pokud jsou v x-ové, nebo y-ové ose vzdáleny právě o 1, tedy sousedství neplatí po úhlopříčkách.

4.14 Navrhněte rekurzivní algoritmus, který spočítá sumu všech čísel od 1 do n, kde n je daný argument algoritmu.

4.15 Navrhněte rekurzivní algoritmus, který najde a vrátí nejmenší prvek v poli A, kde A a jeho velikost jsou dané argumenty.

4.16 Navrhněte rekurzivní algoritmus, který vypočítá sumu všech prvků v poli A, kde A a jeho velikost jsou dané argumenty.

4.17 Navrhněte rekurzivní algoritmus, který rozhodne, jestli vstupní řetězec S je palindrom. S a jeho velikost jsou dané argumenty.

4.18 Mějme následující algoritmus:

Funkce RECURSION(a, b)

vstup: $a, b \in \mathbb{N}_0$

```
1 if b = 0 then
2     return 0
3 fi
4 if b mod 2 = 0 then
5     return RECURSION(a + a, b/2)
6 fi
7 return RECURSION(a + a, b/2) + a
```

- Určete, co je výstupem algoritmu.
- Jak se změní výstup algoritmu, když nahradíme součet za součin a *return 0* za *return 1*?

4.19 Napište program, který dostane na vstup číslo n, na základě kterého vypočítá a vypíše všech $n!$ permutací n písmen začínajících od a (předpokládejte, že n je menší než 26). Na pořadí výsledných permutací nezáleží. Pro $n = 3$ by jeden z možných výstupů programu byl následovný:

abc bac acb cab cba bca

Modifikujte předcházející program tak, aby vracel permutace délky k, tedy pro vstup n a k vrátí $\frac{n!}{(n-k)!}$ permutací. Výstup pro $n = 4$ a $k = 2$:

ab ac ad ba bc bd ca cb cd da db dc

Napište program, který vypíše všechny kombinace délky k. Vstupem programu jsou 2 argumenty n a k a výstupem je $\frac{n!}{k!\cdot(n-k)!}$ kombinací. Například pro vstup $n = 5$ a $k = 3$ program vypíše:

abc abd abe acd ace ade bcd bce bde cde

4.20 Navrhněte algoritmus, který pro hodnotu k a řetězec bitů b vrátí všechny řetězce, kterých Hammingova vzdálenost od b bude právě k. Hammingovu vzdálenost dvou stejně dlouhých řetězců definujeme jako počet bitů, ve kterých se řetězce liší. Pro $b = 0000$ a $k = 2$ budou výstupem programu binární řetězce:

0011 0101 0110 1001 1010 1100

4.21 Mějme zásobník n palačinek různých velikostí. Navrhněte algoritmus, který uspořádá palačinky podle velikosti vzestupně tak, že největší palačinka je vespod a nejmenší na vrcholu. Pro manipulaci s palačinkami jste schopni prohodit horních k palačinek tak, že obrátíte jejich pořadí. Vašim úkolem je utřídit palačinky v nejvíce $2n - 3$ krocích.

4.22 Mějme následující rekurzivní formuli pro aproximaci zlatého řezu:

$$f(n) = \begin{cases} 1 & \text{pokud } n = 0 \\ 1 + \frac{1}{f(n-1)} & \text{pokud } n > 0 \end{cases}$$

- Navrhněte a naprogramujte rekurzivní algoritmus počítající aproximaci zlatého řezu podle předchozí formule.
- Naprogramujte stejný algoritmus bez použití rekurze.

4.23 Mějme následující algoritmus. Určete, zdali je algoritmus konvergentní. Pokud není, určete dvojici čísel a, b, pro které algoritmus neskončí.

Funkce RECURSION(a, b)

vstup: $a, b \in \mathbb{N}$

```
1 if a ≠ b then
2     m ← (a+b)/2
3     RECURSION(a, m − 1)
4     RECURSION(m + 1, b)
5 fi
```

4.24 Mějme následující vzájemně rekurzivní funkce. Určete výstup volání $g(g(2))$.

Funkce $f(n)$

vstup: $n \in \mathbb{N}$

```
1 if n = 0 then
2     return 0
3 fi
4 return f(n − 1) + g(n − 1)
```

Funkce $g(n)$

vstup: $n \in \mathbb{N}$

```
1 if n = 0 then
2     return 0
3 fi
4 return g(n − 1) + f(n)
```

4.25 Mějme následující rekurzivní funkci. Určete výstup volání $f(0)$.

Funkce $f(n)$

vstup: $n \in \mathbb{N}$

```
1 if n > 100 then
2     return n − 4
3 fi
4 return f(f(n + 5))
```

4.26 Von Neumannova přirozená čísla jsou definována následovně: 0 je definována jako prázdna množina, každé další číslo $i > 0$ je definováno jako množina obsahující von Neumannova přirozená čísla od 0 po $i - 1$. Příklad:

$$0 = \{\} = \{\}$$
$$1 = \{0\} = \{\{\}\}$$
$$2 = \{0, 1\} = \{\{\}, \{\{\}\}\}$$
$$3 = \{0, 1, 2\} = \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$$

Napište program, který pro vstup $n \in \mathbb{N}$ vrátí řetězec reprezentující číslo ve von Neumannově zápisu.

4.27 Napište program, jehož vstupem je řetězec s a číslo k. Výstupem programu budou všechny podřetězce s délky k.

4.28 Mějme 2 řetězce různých znaků. Navrhněte algoritmus, který vrátí všechna možná proložení těchto 2 řetězců. Například mějme $s =$ "ab" a $t =$ "CD", pak výstupem algoritmu bude:

$$abCD \;\; CabD \;\; aCbD \;\; CaDb \;\; aCDb \; CDab$$

4.29 Napište program, který pro vstup $n \in \mathbb{N}$ vrátí všechna rozdělení na čísla, které v součtu dávají právě n. Program pro $n = 4$ vrátí:

4
3 1
2 2
2 1 1
1 1 1 1

Kapitola 5

Řadicí algoritmy

Nejprve se studenty zopakujte definice těchto základních pojmů z přednášky:

Řadicí algoritmus je algoritmus zajišťující seřazení daného souboru dat podle specifikovaného pořadí.

Uspořádání je dáno relací mezi prvky posloupnosti. Typicky řadíme podle hodnoty klíče vzestupně ($A_n \leq A_{n+1}$), nebo sestupně. Můžeme si však zvolit vlastní relaci pro uspořádání.

Stabilní řadicí algoritmus je takový algoritmus, který po seřazení zachovává vzájemné pořadí prvků se stejným klíčem.

Stabilní algoritmus se často hodí při řazení podle více klíčů. Chceme-li seřadit studenty podle skupin abecedně, stačí, když seřazenou posloupnost studentů podle abecedy seřadíme podle skupiny pomocí stabilního algoritmu.

Přirozený řadicí algoritmus rychleji zpracuje již částečně seřazenou posloupnost.

Prostorová složitost se definuje analogicky jako časová, přičemž mírou složitosti je namísto počtu kroků výpočtu množství paměti, která je v průběhu výpočtu obsazená.

In situ/in place je algoritmus, který na svůj běh potřebuje kromě samotných řazených dat konstantní množství paměti.

Online řadicí algoritmus dokáže řadit posloupnost, která v době spuštění algoritmu není kompletní. Dokáže tedy do (částečně) seřazené posloupnosti zařadit další prvky.

Cvičení na řadicí algoritmy je obsáhlé, je vhodné znát příklady, abyste v případě nedostatku času věděli, co přeskočit. Myšlenka cvičení není studenty řadicí algoritmy naučit, ty už mají znát, ale procvičit.

Motivací jsou příklady 5.1 až 5.6. Neprobírejte všechny, vyberte si ty zajímavé a ty se studenty řešte. Většina příkladů je podnětem k diskuzi. Dohromady by tato část cvičení neměla zabrat víc než 15 minut, spíše 10.

Probírání jednotlivých řadicích algoritmů zabere většinu cvičení. Nechceme opakovat přednášku, věnujeme se spíše příkladům k zamyšlení, modifikacím algoritmů a aplikaci na různých vstupech.

Tématem jsou tyto řadicí algoritmy: InsertSort, MergeSort, QuickSort a CountingSort. Podle času možná budete nuceni CountingSort nechat na doma, to nevadí, znají ho z přednášky, a

tedy to doma zvládnou. Konkrétnější informace k příkladům najdete až u nich.

Shrnutí v podobě příkladu 5.15 je vhodné jen když budete mít náhodou čas. Hodí se před programováním říci něco o knihovním řadicím algoritmu C a Pythonu (příklad 5.17).

Příprava na programování obsahuje všechny řadicí algoritmy ze sbírky, povinné jsou však jen InsertSort, MergeSort, QuickSort a CountingSort. Zbylé jsou jen bonus pro aktivní studenty. Na optimální řadicí algoritmy jsou připraveny testy i pro pomocné funkce Partition a Merge, proto je potřeba, aby studenti naprogramovali přesně tu verzi algoritmu, která je testována a popsána v zadání.

Daliborek vzkazuje: Řekneme, že uspořádání je úplné, pokud se jedná o trichotomickou relaci.

Toto je zde spíše symbolicky (na doma pro studenty, co nebyli na cviku), odpověď by měla zaznít do 5 sekund. Klidně přeskočte na následující příklad.

Alternativou je hra, kdy studentům připravíte 2 různý typy papírů plné náhodných čísel. Jedna verze však bude seřazená od nejmenšího čísla po největší, druhá bude náhodná permutace stejných čísel. Každému studentovi dejte během vypracovávání odpovědníku jeden z těchto papírů. Pak místo tohoto příkladu dejte úkol najít nějaké číslo, které se v posloupnosti nachází.

5.1 Proč se vůbec řazením zabýváme? Proč chceme posloupnosti dat řadit?

Smyslem je urychlit přístup k datům dané posloupnosti. Například pro možnost využití binárního vyhledávaní, které vyžaduje na vstupu seřazené pole.

Aplikace, které provádějí častá vyhledávání, mohou seřazením dat dramaticky urychlit svoji činnost. Je možné ukázat rozdíl ve složitosti (rychlosti) sekvenčního hledání a hledání např. půlením intervalů v seřazeném poli (viz příklad 3.14).

Pan Usměvavý dodává: Chuck Norris umí řadit v konstantním čase pomocí algoritmu ChuckSort.

5.2 S využitím principu binárního vyhledávání navrhněte algoritmus, který vrátí index prvního výskytu prvku k v seřazeném poli $A[1 \dots n]$. Jestli se prvek k v poli A nevyskytuje, pak algoritmus vrátí hodnotu -1.

Binárně nalezneme k. Od tohoto výskytu by šlo sekvenčně postupovat doleva, dokud nenarazíme na menší prvek. Takové řešení by však mělo lineární složitost (pro případ posloupnosti stejných čísel).

Správný postup je tedy použití binárního vyhledávání, ale nespokojíme se s pouhým nálezem k. Naše modifikované binární půlení se zastaví až v případě, že narazí na prvek k, který je buďto prvním prvkem pole, nebo se před ním nachází menší prvek. V případě, že narazíme na k, které toto pravidlo nesplňuje, pak pokračujeme ve hledání v intervalu nalevo od zkoumaného prvku.

Pokud předchozí algoritmus studenti nevyřešili za pár sekund, tak toto nechte na doma.

5.3 Modifikujte algoritmus z předchozího příkladu tak, aby vracel index prvního výskytu prvku většího než je dané číslo k. Pokud jsou všechny prvky menší než k, pak algoritmus vrátí -1.

Oproti předchozímu příkladu se řešení liší jen tím, že hledáme prvek na pravé straně od nalezeného k.

5.4 Pole A obsahuje nějaké náhodné prvky. Navrhněte algoritmus, který smaže duplicitní prvky v čase $\mathcal{O}(n \cdot \log(n))$.

Můžeme se opřít o to, že dokážeme řadit v čase $\mathcal{O}(n \cdot \log(n))$, tudíž vstupní pole nám stačí seřadit a pak v lineárním čase vyházet duplicity jedním během přes seřazené pole. Časová složitost takového algoritmu je $\mathcal{O}(n \cdot \log(n)) + \mathcal{O}(n) = \mathcal{O}(n \cdot \log(n))$. Pseudokód by mohl vypadat následovně:

Funkce EraseDuplicates(A, n)

vstup: Pole A délky n
výstup: Pole bez duplicitních prvků

```
1 sort(A) // seřadíme pole A řadicím algoritmem, který pracuje v čase O(n · log(n)), třeba
    mergesort
2 index ← 1
3 for i ← 2 to n do
4     if A[index] ≠ A[i] then
5         index ← index + 1
6         Swap(A[index], A[i])
7     fi
8 od
  // duplicity zůstanou v poli za indexem index
9 return A[1, ..., index]
```

5.5 Mějme databázi lidí, ke kterým uchováváme jména a místo bydliště. Na jedné adrese typicky bydlí více lidí. Chceme seřadit lidi primárně podle adres a na každé adrese podle jména, jak na to?

Potřebujeme stabilní řadicí algoritmus. Prvně seřadíme libovolným algoritmem podle jména a pak stabilním řadicím algoritmem podle bydliště. Celková složitost $\Theta(n \log(n))$.

Druhá možnost je seřadit lidi podle adresy a pak pro jednotlivé adresy ještě řadit podle jména. Řešení má stejnou asymptotickou složitost a je složitější na správné naprogramování, proto se v databázích běžně používá první přístup.

Diskuzní příklad na probuzení studentů, zabere maximálně 3 minuty.

5.6 Zkuste podrobně popsat vlastními slovy postup, jak seřadit:

Až na BubbleSort by všechny ostatní algoritmy měli studenti znát z přednášky. Odrážku d) je tedy vhodné přeskočit, ale po cvičení by ji studenti zvládnout mohli, proto není ze sbírky vyřazena.

Cílem příkladu je dát šanci kvalitně odpovědět i těm studentům, kterým matematika příliš nejde nebo jsou v programování úplní začátečníci. Nechte proto všechny chvíli přemýšlet, pak požádejte o jednotlivá řešení ty studenty, kteří se dosud moc nezapojovali nebo o kterých víte, že mívají u tabule problémy. Možná padnou i jiné návrhy, než jsou uvedeny v řešení. Zkuste o nich chvíli diskutovat a zapojit do diskuse co nejvíce studentů.

a) 10 hracích karet do ruky při rozdávání (po jedné),

InsertSort – vezmeme kartu a procházíme karty na ruce tak dlouho, dokud nenajdeme správnou pozici. Zařazením na danou pozici se posunou následující prvky.

b) nastoupenou fotbalovou jedenáctku v dresech, pokud má předstoupit o krok dopředu seřazená podle čísel na dresech,

SelectSort – trenér nechává postupně předstoupit hráče podle čísel na dresech.

c) 200 písemek podle abecedy, jsou-li k dispozici 4 pomocníci,

MergeSort – paralelizovatelný, každý uspořádá čtvrtinu a ty následně spojují.

d) lidi stojící ve frontě v úzké chodbě podle data narození (pokud každý může mluvit jenom se svým sousedem),

BubbleSort – ideálně paralelní a obousměrný, ale stačí princip porovnávání sousedních dvojic.

e) papírky s čísly v rozmezí 00-99, pokud si můžete vytvářet hromádky,

BucketSort

f) soubory v adresáři tak, že primárně chceme řadit podle dne změny a sekundárně podle jména souboru.

Prvně seřadíme soubory podle jména libovolným řadicím algoritmem, a potom je seřadíme stabilním řadicím algoritmem podle data změny. Na obojí lze použít MergeSort.

Na malé množství souborů by také šlo použít RadixSort.

Neprobírejte pseudokód, ten už mají znát z přednášky.

Při aplikaci řazení vkládáním po studentech vyžadujte přesné zdůvodnění, proč dělají jaký krok. První studenty ještě můžete nechat popisovat kroky algoritmu intuitivně, později však chtějte exaktní popis.

5.7

a) Použijte algoritmus InsertSort na tomto poli: 8 5 2 6 9 3.

b) Rozeberte, zdali je InsertSort stabilní a in situ, a poté určete jeho časovou složitost. Kde byste použili InsertSort vzhledem k jeho výhodám, jaké má nevýhody?

InsertSort rozdělí vstupní pole na seřazenou a neseřazenou část. V každém kroku vezmeme první prvek z neseřazené části a zařadíme ho na správné místo v první části posloupnosti. To může být uděláno například tak, že budeme seřazenou posloupnost procházet pozpátku (od nového prvku) a dokud je nový prvek větší, než prvek nalevo od něj, tak je prohodíme.

Ve zmíněné variantě je InsertSort stabilní.

InsertSort je in situ, jelikož potřebuje jen $\mathcal{O}(1)$ místa navíc (1 místo pro prohození prvků).

Oproti algoritmu SelectSort má InsertSort několik výhod. Pro seřazené pole pracuje v čase $\mathcal{O}(n)$, je tedy přirozený. Nejvýraznější výhodou je to, že umí řadit posloupnosti, které mu teprve během řazení přicházejí na vstup. Algoritmům s touto vlastností se říká online algoritmy. InsertSort může být vhodnou volbou například pro řazení paketů, které může řadit průběžně a nemusí čekat, až získá celou posloupnost.

Funkce InsertSort(A, n)

vstup: pole A délky n
výstup: vzestupně seřazené pole A původních prvků

```
1  for i ← 1 to n do
2      j ← i
3      tmp ← A[i]
4      while j > 1 ∧ A[j − 1] > tmp do
5          A[j] ← A[j − 1] // posune vetší prvek doprava
6          j ← j − 1
7      od
8      A[j] ← tmp // zařadí i-tý prvek na správné místo
9  od
10 return A
```

c) Jak funguje InsertSort na vzestupně a sestupně seřazené posloupnosti a na posloupnosti stejných čísel?

* na začátku příkladu značí pokročilý příklad nad rámec tohoto předmětu.

d) * Lze podmínka while cyklu v algoritmu InsertSort redukovat tak, aby používala jen jedno porovnání?

Ano, stačí přidat tzv. sentinel – hlídač, který je na pozici $A[0]$ (tedy před vstupním polem) a nabývá hodnoty, která je určitě menší, než všechny prvky posloupnosti. Pak lze z cyklu odstranit podmínku $j > 1$.

Postup u probírání algoritmu MergeSort. Prvně začněte funkcí Merge (nepište pseudokód). Na obrázku ilustrujte její chování, použijte pomocné pole. Pokud máte čas, tak prodiskutujte, zdali jde Merge udělat in place.

5.8

a) Navrhněte funkci, která spojí 2 seřazené posloupnosti tak, aby jejich spojením vznikla jedna výsledná posloupnost. Zamyslete se nad časovou i prostorovou náročností. Lze navrhnout asymptoticky optimálnější řešení? Je vaše funkce stabilní?

Funkce Merge používá pomocné pole *aux*. Do něj nejprve zkopíruje obě části pole A a poté z pomocného pole bere postupně nejmenší prvky a ukládá je do původního pole A. Pomocné pole může být pro snížení paměťové složitosti globální pro všechna volání funkce Merge. Povšimněte si, že funkce Merge zachovává pořadí prvků v poli, jde o předpoklad stability algoritmu MergeSort.

Funkce Merge($A, from, mid, to$)

vstup: Pole A, které obsahuje 2 seřazené posloupnosti. Jedna je v rozsahu $A[from..mid]$, druhá v $A[mid+1..to]$

výstup: seřazené pole A obsahující prvky z původního pole

```
1  aux ← A // vytvoříme kopii pole A
2  leftIndex ← from // index aktuálního porovnávaného prvku v levém poli
3  rightIndex ← mid + 1 // index aktuálního porovnávaného prvku v pravém poli
4  for k ← from to to do
5      if leftIndex > mid then
6          A[k] ← aux[rightIndex]
7          rightIndex ← rightIndex + 1
8      else if rightIndex > to then
9          A[k] ← aux[leftIndex]
10         leftIndex ← leftIndex + 1
11     else if aux[leftIndex] ≤ aux[rightIndex] then
12         A[k] ← aux[leftIndex]
13         leftIndex ← leftIndex + 1
14     else
15         A[k] ← aux[rightIndex]
16         rightIndex ← rightIndex + 1
17     fi
18 od
19 return A
```

b) Funguje vaše funkce i na neseřazené posloupnosti?

Ano, funkce umí pracovat i s neseřazenými posloupnosti, ale výsledek spojování neseřazených posloupností nemůže být seřazený. Pokud jsou v jedné posloupnosti 2 prohozené prvky, pak i ve výsledné posloupnosti musí být jejich pořadí špatně, jen mezi ně můžou přibýt další prvky.

c) Jak se změní složitost pro seřazení více posloupností $(3, 4, \ldots, k)$?

Asymptoticky se nezmění, avšak kvůli většímu množství porovnávání se zvětší konstanty funkce. Merge musí mít vždy alespoň lineární složitost.

Následuje použití Merge na posloupnosti. Zde je neoptimální řešení, máte analyzovat jeho složitost a navrhnout lepší. Kreslete stromečky, je to na nich hezky vidět.

5.9 Je dáno k seřazených posloupností $(p_1, p_2, \ldots, p_k)$, kde každá má n prvků. Vytváříme seřazenou postupnost následovně:

$$\textsc{Merge}(\ldots \textsc{Merge}(\textsc{Merge}(\textsc{Merge}(p_1, p_2), p_3), p_4), \ldots, p_k)$$

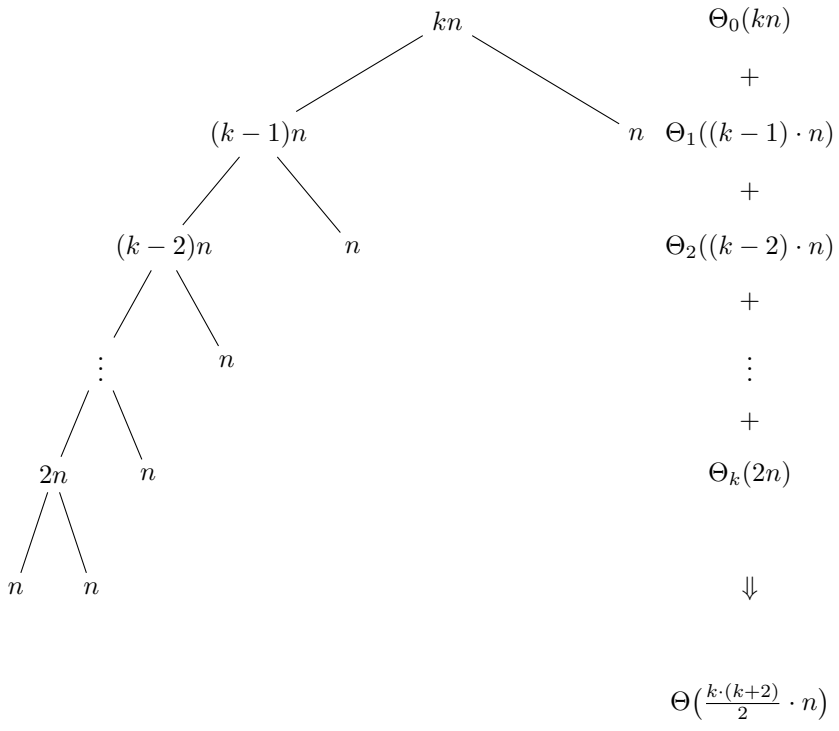
Složitost výpočtu je:

a) $\Theta(n \cdot k^2)$ b) $\Theta(n \cdot k)$

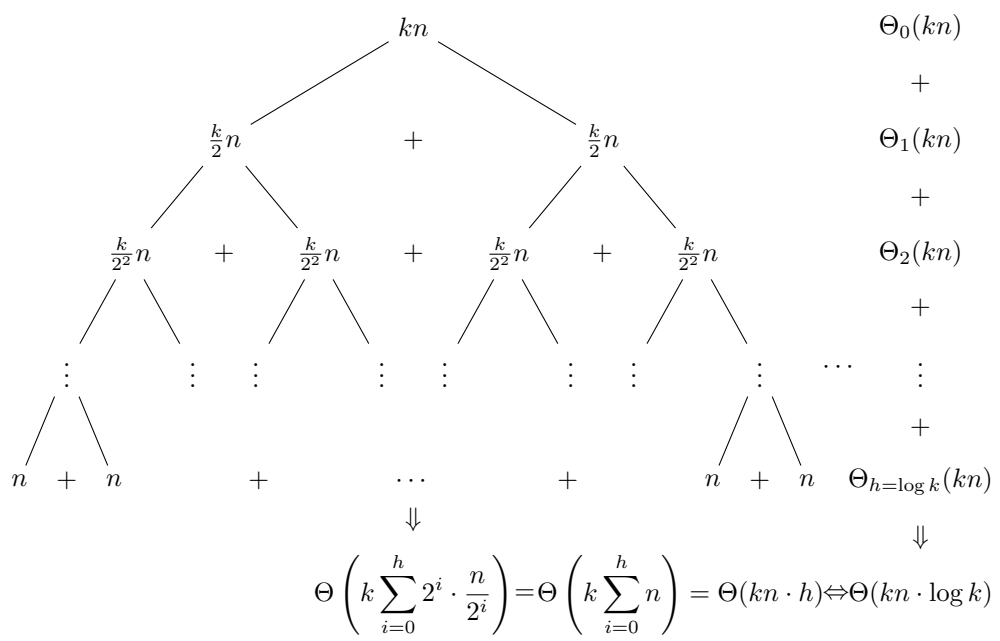
c) $\Theta(n^2 \cdot k)$ d) $\Theta(n \cdot \log(k))$

Navrhněte asymptoticky efektivnější řešení.

Složitost zadané posloupnosti volání Merge je $\Theta(n \cdot k^2)$. Důvodem je, že spojujeme po každém volání delší posloupnost, takže při posledních voláních spojujeme posloupnost délky $n \cdot (k-1)$ s posloupností délky n. Tyto poslední volání ve složitosti převažují. Strom rekurzivního volání se délkami posloupností v jednotlivých uzlech vypadá následovně:



Problém lze vyřešit i asymptoticky rychleji. Chceme volat funkci Merge na co nejmenších posloupnostech, proto chceme spojovat nejdříve dvojice posloupností délky n. Takto vzniklé posloupnosti chceme spojovat v druhém kroku. Postupně chceme spojovat dvojice stejně dlouhých posloupností. Počet volání Merge bude stále $k-1$, ale budeme častěji spojovat posloupnosti malé délky. Výsledná složitost je $\Theta(k \cdot n \cdot \log(k))$. Strom volání vypadá takto:



Na konci předchozího příkladu jste došli téměř k MergeSortu. Stačí říci, že posloupnosti nahradíte jednotlivými prvky pole. Nastává otázka, jak takový algoritmus volat (možná přepsání do pseudokódu). V minulém příkladu šlo o bottom-up přístup, rekurzivní algoritmus by byl top-down. Když máte Merge, tak je MergeSort jednoduchý. Proberte, jak funguje MergeSort na seřazených posloupnostech a posloupnosti stejných čísel.

5.10

a) Formulujte algoritmus MergeSort (řazení slučováním). Rozeberte, zdali je stabilní a in situ, a poté určete jeho časovou a prostorovou složitost.

MergeSort je navržený pomocí metody rozděluj a panuj, kde algoritmus rekurzivně vezme vstupní pole a rozdělí jej na dvě části. Když algoritmus rozebere vstupní pole na jednotlivé prvky, začne je slučovat a vzájemně seřazovat, na což využívá funkci Merge.

Když již máme funkci Merge, napsání řadicího algoritmu MergeSort je jednoduché. V každém zanoření rozdělíme pole na 2 poloviny (divide) a rekurzivně se na obou částech zavoláme. Z volání si vracíme seřazené části pole, které následně spojíme pomocí funkce Merge (conquer).

Procedura MergeSort($A, from, to$)

vstup: pole A obsahuje permutaci prvků posloupnosti $a = (a_{from}, \ldots, a_{to})$
výstup: seřazené pole A' původních prvků

```
1 if to ≤ from then
2     return A // jednoprvkové, nebo prázdné pole, konec zanořování (divide) rekurze
3 fi
4 mid ← from + ⌊(to−from)/2⌋
5 A[from..mid] ← MergeSort (A, from, mid) // divide
6 A[mid + 1..to] ← MergeSort (A, mid + 1, to) // divide
7 return Merge (A, from, mid, to) // conquer
```

Naše implementace algoritmu je stabilní, tudíž zachovává pořadí prvků se stejným klíčem. Existují i implementace, které stabilní nejsou. Jelikož používáme pomocné pole *aux* v Merge, tak není algoritmus in situ (ale in situ implementace existuje). Časovou složitost dokážeme spočítat pomocí rekurentní rovnice (jak bylo ukázáno na přednášce) a vychází $\Theta(n \cdot \log(n))$. Vzhledem ke spodní hranici porovnávacích algoritmů $\Omega(n \cdot \log(n))$ lepší asymptotické složitosti nedokážeme dosáhnout.

Karlík varuje: Doufám, že zvládnete víc než toto:

Procedura HalfHeartedMergeSort(*List*)

```
1 if |List| < 2 then
2     return List
3 fi
4 pivot ← ⌊|List|/2⌋
5 A ← HalfHeartedMergeSort(List[: pivot])
6 B ← HalfHeartedMergeSort(List[pivot :])
  // Co teď?
7 return [A, B] // víc nezvládnu
```

b) Jak funguje MergeSort na vzestupně a sestupně seřazené posloupnosti a na posloupnosti stejných čísel?

c) Použijte algoritmus MergeSort na tomto poli: 8 5 2 6 9 3.

Příklad, ve kterém mají vymyslet algoritmus Partition. Dá se k němu dostat postupně. Stačí vymyslet myšlenku, nemusíte psát pseudokód.

První nápad může být používat 2 výsledná pole – pole záporných čísel a pole kladných čísel. S tím se nespokojte.

Následně můžete chtít, aby řešení používalo jen jedno pomocné pole, do kterého ukládáte zepředu záporná a zezadu kladná čísla.

To nás ale vede k Partition, který používáme u QuickSortu s pivotem 0. Chceme tedy udělat řešení in place, s indexy zleva a zprava.

Studenti můžou zkoušet i jiné in place algoritmy (třeba takový, který byl na přednášce a je méně efektivní). S ním se nespokojte.

5.11 Navrhněte algoritmus, který přeuspořádá posloupnost n čísel tak, že všechna záporná čísla jsou před všemi kladnými čísly. Algoritmus by měl být časově i prostorově efektivní.

Nejjednodušší řešení je vytvořit 2 nová pole – pole záporných čísel a pole kladných čísel. Obě by museli mít délku n, protože rozdělení prvků předem neznáte. Použití 2 polí by šlo redukovat na použití jednoho pole. Menší hodnoty bychom ukládali zleva, větší zprava. Algoritmus však lze napsat i in situ. Jedním z možných řešení je například algoritmus Partition z přednášky, volaný s pivotem 0.

Domyslet QuickSort z Partition. Hlavní část příkladu má být diskuze na volbu pivota, spolu s ní zvládnete i diskuzi k seřazeným posloupnostem, posloupnostem stejných prvků...

5.12

a) Formulujte algoritmus QuickSort (řazení rozdělováním). Rozeberte, zdali je stabilní a in situ.

QuickSort je algoritmus typu rozděluj a panuj. V první fázi vybereme z posloupnosti pivota, podle kterého budeme pole dělit na část menších prvků než pivot a část větších prvků, než je pivot. Podrobný popis tohoto rozdělení najdete ve slidech z přednášky.

Po rozdělení následuje rekurzivní volání na podproblémy, tedy zvlášť na podpole menších hodnot než pivot a podpole větších hodnot než pivot.

Třetí část je spojení obou částí pole. Pokud pracujeme v rámci jednoho pole, tak tato fáze odpadá.

QuickSort ve variantě z přednášky není stabilní, jelikož při přehození prvků mezi částmi větší a menší než pivot může jeden ze stejných prvků přeskočit jiný.

Procedura QuickSort$(A, from, to)$

vstup: A je pole délky $|to - from + 1|$
výstup: A' je pole délky $|to - from| + 1$, pro které platí, že pro všechna $i \in \{from, \ldots, to\} : A'[i] \leq A'[i+1]$

```
1 if from < to then
2     pivotIndex ← Partition (A, from, to)
3     QuickSort (A, from, pivotIndex − 1)
4     QuickSort (A, pivotIndex + 1, to)
5 fi
6 return A
```

Funkce rozdělení pole může vypadat následovně:

Funkce Partition$(A, from, to)$

vstup: A je pole délky $|to - from + 1|$
výstup: pozice *index* a pole prvků A je přeskládané v úseku mezi indexy *from* a *to* tak, že pro všechna $i \in \{from, \ldots, index\}$ platí $A[i] \leq A[index]$ a pro všechna $j \in \{index, \ldots, to\}$ platí $A[index] \leq A[j]$

```
1 pivot ← A[to]
2 index ← from − 1
3 for i ← from to to do
4     if A[i] ≤ pivot then
          // prvky ≤ pivotu se dávají před pozici index
5         index ← index + 1
6         swap(A[index], A[i])
7     fi
8 od
9 return index
```

b) Jakou roli hraje u algoritmu QuickSort volba pivota? Určete časovou složitost pro náhodnou a uspořádanou posloupnost podle volby různého pivota.

Časová složitost algoritmu QuickSort se odvíjí od volby pivota. Pokud volíme za pivota medián, pak se vždy pole rozdělí na 2 poloviny a celkový počet rekurzivního volání bude $\log(n)$. Naopak v případě výběru okrajového prvku (minima nebo maxima) bude počet porovnání $\frac{n\cdot(n+1)}{2}$. QuickSort tedy patří do $\mathcal{O}(n^2)$, avšak u náhodně seřazeného pole je průměrná složitost $\Theta(n \cdot \log(n))$.

Algoritmus z přednášky volí za pivota poslední prvek posloupnosti. To je v pořádku u náhodně uspořádaných posloupností, kde se může medián posloupnosti vyskytovat se stejnou pravděpodobností v celé posloupnosti. Avšak v případě uspořádané (byť jen částečně) posloupnosti budeme vybírat velmi často maximum. To je pro volbu pivota naprosto nevhodné a časová složitost bude až kvadratická.

Vzhledem k tomu, že v praxi často pracujeme s částečně uspořádanou posloupností, je vhodné vybírat pivota jinak než vybráním posledního členu posloupnosti. Často dosáhneme velmi dobrého výsledku výběrem prvku ze středu posloupnosti.

Existuje algoritmus, který umí vybírat medián v logaritmickém čase. S jeho použitím pro volbu pivota se QuickSort stává algoritmem se složitostí $\mathcal{O}(n \cdot \log(n))$. Algoritmus se jmenuje MedianOfMedians a pro výběr pivota u algoritmu QuickSort se nepoužívá z důvodu velkých konstant u časové složitosti.

c) Jak funguje algoritmus z přednášky na posloupnosti stejných prvků? Lze jej nějak optimalizovat?

Algoritmus z přednášky pro pole stejných prvků pracuje v kvadratickém čase. První podpole bude obsahovat všechny prvky až na pivota, takže rekurze se bude volat jen na posloupnost o 1 menší. Opravit tento problém lze vytvářením 3 podpolí, podpole menších hodnot než medián, podpole hodnot stejných jako medián a podpole hodnot větších než medián.

d) Napadají vás další optimalizace ukázkového pseudokódu, které by mohly řazení urychlit?

Nejvýraznější optimalizace se týkají výběru pivota, což už jsme rozebrali výše. Další zrychlení lze získat přepsáním z rekurzivní na iterativní podobu. Poslední v praxi používanou optimalizací je uspořádání velmi malých celků jiným řadicím algoritmem – InsertSort. Ten má na malých posloupnostech výrazně rychlejší běh než QuickSort.

e) Použijte algoritmus QuickSort na tomto poli: 8 5 2 6 9 3.

Oslí můstek k algoritmu CountingSort.

5.13 Posloupnost n čísel obsahuje čísla z intervalu $\langle 1 \dots k \rangle$. Seřaďte ji v čase $\mathcal{O}(n \log(k))$.

Využijeme QuickSort. Jako pivot používáme postupně středy intervalu, tedy $k/2, k/4$ a $3k/4 \dots$.

Následující algoritmy patří mezi algoritmy, které řadí v lineární čase – CountingSort, RadixSort, BucketSort. Vzhledem na podobnou myšlenku všech algoritmů a podle dostupného času na cvičení doporučujeme odcvičit jenom CountingSort, ke kterému jsou i připravená programátorská zadání.

5.14

a) Formulujte algoritmus CountingSort (řazení počítáním). Rozeberte, zdali je stabilní a in situ, a poté určete jeho časovou složitost.

CountingSort předpokládá, že vstupní pole obsahuje čísla z intervalu $\langle 1, \ldots, k \rangle$ (pole indexujeme od 1). Algoritmus si spočítá počet prvků, které mají rozdílné klíče, a použitím aritmetiky rozhodne, kde který klíč bude mít svou pozici ve výsledném poli. Jeho časová složitost je lineární v závislosti na počtu prvků a rozdílu mezi nejmenším a největším klíčem. To nám ukazuje, že algoritmus je vhodný jenom v případech, kde variabilita v klíčích není výrazně větší než počet prvků vstupu. CountingSort je ale používán jako subrutina v jiných řadicích algoritmech jako například radix sort, který se umí vypořádat s většími klíči efektivněji. Složitost algoritmu je $\Theta(k + n)$ (nejde o řadicí algoritmus založený na porovnávání, tedy na něj neplatí dolní ohraničení složitosti $\Omega(n \cdot \log(n))$) jak bylo dokázáno na přednášce. Algoritmus je stabilní, jelikož zachovává relativní pozici objektů se stejnými klíči. Tato vlastnost je předpokladem i pro aplikaci algoritmu CountingSort v algoritmu RadixSort. Vzhledem k potřebě pomocného pole není CountingSort in situ.

Funkce CountingSort(A, n, k)

vstup: pole A délky n s hodnotami z intervalu $\langle 1, \ldots, k \rangle$
výstup: vzestupně seřazené pole A původních prvků

```
1  for i ← 1 to k do
2      count[i] ← 0;
3  od
4  for i ← 1 to n do
5      count[A[i]] ← count[A[i]] + 1
6  od
7  for i ← 2 to k do
8      count[i] ← count[i] + count[i − 1]
9  od
10 for i ← n downto 1 do
11     B[count[A[i]]] ← A[i]
12     count[A[i]] ← count[A[i]] − 1
13 od
14 return B
```

b) Použijte algoritmus CountingSort na tomto poli: 3 4 4 4 3 5 1 7 2 1.

Podle času lze probrat, není to na dlouho a je to shrnující příklad.

5.15 Který řadicí algoritmus je nejefektivnější pro řešení následujících problémů?

1. Malé pole celých čísel,
2. velké pole obsahující náhodná čísla,
3. velké pole obsahující téměř seřazenou posloupnost čísel,
4. velká množina čísel, které jsou z malého intervalu.

Řadit lze i jiné věci než pole, ale asi na to nebude čas.

5.16 Jaký řadicí algoritmus funguje nejlépe na seřazení spojovaného seznamu? Navrhněte různá řešení a porovnejte jejich výhody a nevýhody.

InsertSort funguje pro zřetězený seznam téměř stejně jako na poli. Navíc ušetří počet přesunů v paměti.

Ještě vhodnější je však MergeSort.

5.17 Znalost probraných řadicích algoritmů je důležitá, abyste věděli, jak s daty pracovat. V praxi však až na vzácné výjimky není programování vlastního řadicího algoritmu moudrá volba (připravené řadicí algoritmy jsou velmi precizně optimalizovány a navíc máte jistotu, že na rozdíl od vašeho kódu jsou určitě správně). Důležitá je tedy znalost řadicích algoritmů v standardní knihovně vašeho oblíbeného programovacího jazyka. Zkuste si tedy přečíst manuálovou stránku a poté seřadit pole pomocí knihovních funkcí.

Zajímavé je vyhledat, jaké řadicí algoritmy jsou používány v kterých knihovnách známých jazyků.

C má v knihovně stdlib.h hezky okomentovanou implementaci funkce qsort. Ta již podle jména napovídá, že využívá QuickSort. Pro volbu pivota využívá median-of-three, který určí medián z prvního, prostředního a posledního prvku. Samotný QuickSort je samozřejmě v iterativní podobě. Ve chvíli, kdy by měl v posloupnosti seřadit 4 a méně prvků, volá na uspořádání InsertSort. Pro některé vstupy (velmi speciálně uspořádané) však pracuje v kvadratickém čase.

C++ má v knihovně algorithm 3 různé řadicí algoritmy. Nestabilní, stabilní a částečné řazení. První z nich používá k řazení IntroSort, což je hybridní řadicí algoritmus. Ten prvně zkouší pole utřídit algoritmem QuickSort podobným qsortu ze stdlib, pokud se však posloupnost výrazně nezmenšuje, přepne se do řazení HeapSort. Pokud IntroSort neskončí do $2 \cdot \log_2(n)$ opakování, volá se na zbylou posloupnost InsertSort.

Stabilní verze za cenu snížené rychlosti nebo větší paměťové náročnosti využívá MergeSort.

Díky šablonám je verze v C++ výrazně rychlejší na primitivních datových typech (int), než verze z C.

Python používá TimSort, který je spojením algoritmů InsertSort a MergeSort. Pro posloupnost méně než 64 prvků se používá jen InsertSort. Na větších posloupnostech se prvně hledají již seřazené části. Na neseřazených částech se zavolá InsertSort. Poté se tyto různé části začnou spojovat pomocí algoritmu MergeSort. TimSort je velmi pěkně rozebrán na Wikipedii.

5.18 Naprogramujte si jednotlivé řadicí algoritmy. Ve studijních materiálech jsou k tomuto připravené zdrojové kódy: C a Python.

Následující příklady jsou vhodné na domácí studium.

5.19 Mějme pole velkých objektů A. Navrhněte efektivní způsob, jak toto pole seřadit s co nejmenším počtem přesunů a kopírování v paměti. Zamyslete se také nad tím, jak by šlo dané pole seřadit efektivně vzhledem k práci s pamětí.

Vzhledem k velikosti objektů v poli se budeme snažit minimalizovat práci s pamětí, tedy řadit objekty řadicím algoritmem, který používá co nejméně swapů a snaží se pracovat jenom ve vstupním poli, tudíž je in situ. Takovým řadicím algoritmem je SelectSort, který je in situ a vykoná právě n swapů.

Oproti předcházejícímu řešení však existuje vzhledem ke kopírování objektů i efektivnější způsob řazení. Nemusíme totiž přesouvat celé objekty v paměti, ale můžeme jednoduše seřadit jenom pole ukazatelů. Vzhledem k tomu, že v paměti už budeme pracovat pouze s ukazateli, které mají minimální velikost, je vhodné je řadit co nejrychlejším algoritmem, třeba pomocí algoritmu QuickSort.

5.20 Sekvenční vyhledávaní se dá se stejnou složitostí realizovat na poli i na zřetězeném seznamu. Platí to i pro binární vyhledávaní na seřazeném poli a seřazeném zřetězeném seznamu?

5.21 Mějme pole, ve kterém se opakují pouze znaky B, E, G. Navrhněte lineární in situ algoritmus, který přeuspořádá vstupní pole do tvaru $B^*E^*G^*$ (tedy všechny B se vyskytují před všemi E a ty se vyskytují před G).

Lze řazení provést v jednom průchodu?

5.22

a) Formulujte algoritmus BucketSort (přihrádkové řazení). Rozeberte, zdali je in situ, a poté určete jeho časovou složitost. Algoritmus předpokládá, že vstupní prvky jsou uniformě rozložené v intervalu $[0, 1)$.

BucketSort je řadicí algoritmus, který pro řazení využívá rozdělování vstupního pole klíčů do „přihrádek“. Každou přihrádku pak seřadíme individuálně pomocí jiného řadicího algoritmu, nebo rekurzivním aplikováním bucket sortu. Bucket sort můžeme rozdělit do 4 kroků:

1. inicializace přihrádek,
2. rozdělení vstupu do přihrádek,
3. seřazení každé neprázdné přihrádky,
4. spojení přihrádek v pořadí do jednoho výstupu.

Jelikož BucketSort není porovnávací řádící algoritmus, nemůžeme použít $\Omega(n \log n)$ jako spodní odhad složitosti. Složitost můžeme odhadnout pomocí počtu přihrádek, na

přednášce bylo dokázáno, že přihrádek je průměrně $\Theta(n)$. Vzhledem na paměť potřebnou pro přihrádky není algoritmus in situ.

Funkce BucketSort(A, n)

vstup: pole A délky n
výstup: vzestupně seřazené pole A původních prvků

```
1  bucket ← pole délky n // indexované od 0
2  for i ← 0 to n − 1 do
3      bucket[i] ← prázdný seznam
4  od
5  for i ← 1 to n do
6      vlož A[i] do seznamu bucket[⌊n · A[i]⌋]
7  od
8  for i ← 0 to n − 1 do
9      InsertSort (bucket[i])
10 od
11 spoj seznamy bucket[0], ..., bucket[n − 1] do jednoho seznamu
```

b) Použijte algoritmus BucketSort na tomto poli: 11 1 3 11 13 7 5 18 2.

5.23

a) Formulujte algoritmus RadixSort (číslicové řazení). Rozeberte, zdali je stabilní a in situ, a poté určete jeho časovou složitost.

RadixSort je vhodný algoritmus pro řazení podle různých klíčů a lexikální řazení řetězců. Položky posloupnosti se neporovnávají jako celek, nýbrž se rozdělí na části a řadí se postupně ve skupinách. Rozlišujeme 2 základní rozdílné přístupy, řazení zprava (most significant digit/bit = MSD) a zleva (least significant digit/bit = LSD). Řazení zprava se hodí pro lexikografické řazení (vezmi slova začínající na A a seřaď je rekurzivně, pak slova na B$\cdots$), zleva se jedná o stabilní řazení. Často se používá v kombinaci s jiným stabilním řadicím algoritmem, například CountingSort, alternativně se volá rekurzivně.

b) Použijte algoritmus RadixSort na tomto poli: 170 45 75 90 802 2 24 66

5.24

a) Formulujte algoritmus SelectSort (řazení výběrem). Rozeberte, zdali je stabilní a in situ, a poté určete jeho časovou složitost.

SelectSort řadí pole vybíráním maxima (popřípadě minima). Pole se projde n-krát, přičemž při každém průchodu vyhledáme maximum ze zatím neuspořádané části posloupnosti. Toto maximum prohodíme s posledním prvkem neuspořádané části.

SelectSort není stabilní, protože při prohození nalezeného maxima s aktuálním nejpravějším prvkem měníme pořadí tohoto prvku. Nicméně toto řešení je in situ. Za cenu větší prostorové složitosti však lze SelectSort napsat stabilně.

Časová složitost algoritmu SelectSort je kvadratická, odvození jste měli na druhém cvičení.

Procedura SelectSort(A, n)

vstup: pole A délky n
výstup: vzestupně seřazené pole A' původních prvků

```
1 for i ← n downto 2 do
2     j ← Maximum (A, i)
3     swap(A[i], A[j]) // přehodí prvky
4 od
5 return A
```

Funkce Maximum(A, n)

vstup: A je pole a n je počet prvků v poli
výstup: maximum z prvních n prvků pole A

```
1 max = A[1]
2 for i ← 2 to n do
3     if A[i] > max then
4         max ← A[i]
5     fi
6 od
7 return max
```

b) Použijte algoritmus SelectSort na tomto poli: 8 5 2 6 9 3.

5.25

a) Formulujte algoritmus BubbleSort. Rozeberte, zdali je stabilní a in situ, a poté určete jeho časovou složitost.

Algoritmus BubbleSort, neboli bublinkové řazení, je založen na kvadratickém počtu průchodů posloupností, přičemž při každém průchodu porovnává sousední prvky a pokud je prvek napravo menší než prvek nalevo, tak je prohodí. Tímto způsobem se v každém průchodu největší prvek posouvá napravo, zatímco menší se posunou o 1 doleva. Největší hodnoty tedy "probublávají" na konec posloupnosti, z čehož plyne jméno BubbleSort.
BubbleSort je stabilní a in situ a má kvadratickou časovou složitost. Oproti algoritmu InsertSort má však výrazně horší skutečnou časovou složitost, oproti algoritmu SelectSort zase výrazně více swapů. I jeho intuitivnost, pro kterou se někdy učí, je zpochybnitelná. Jedinou možnou výhodou zůstává paralelizovatelnost porovnávání sousedních prvků.

Nicméně pokud hledáme paralelizovatelné řadicí algoritmy, je snazší vybírat mezi algoritmy typu rozděluj a panuj.

Funkce BubbleSort(A, n)

vstup: pole A délky n
výstup: vzestupně seřazené pole A původních prvků

```
1 for i ← 0 to n do
2     for j ← 0 to n − i − 1 do
3         if A[j] > A[j + 1] then
4             swap(A[j], A[j + 1]) // přehodí prvky
5         fi
6     od
7 od
8 return A
```

b) Použijte algoritmus BubbleSort na tomto poli: 8 5 2 6 9 3.

Pan Usměvavý dodává: StupidSort (také nazývaný BogoSort) je řadicí algoritmus, jehož průměrná asymptotická složitost je $\mathcal{O}((n-1) \cdot n!)$.

Paní Bílá připomíná: Ovšem na kvantovém počítači by BogoSort běžel v lineárním čase.

5.26 Jsou dána seřazená pole A a B délky m a n. Hledáme algoritmus, který vrátí pole C obsahující všechny prvky, které se nacházejí v obou polích (bez duplicit). Jaký postup zvolíme, když:

- A a B mají přibližně stejnou velikost,
- A je výrazně větší než B.

Řešení seřazená postupně podle jednoduchosti:

- Triviální řešení je lineárně vyhledat každý prvek z A v poli B, tj. $\mathcal{O}(m \cdot n)$.
- Pole B je seřazené, a proto můžeme vyhledávat binárně, tj. $\mathcal{O}(n \cdot \log(m))$.
- Pokud uvážíme, které pole je kratší, složitost bude menší z hodnot $\mathcal{O}(n \cdot \log(m))$ a $\mathcal{O}(m \cdot \log(n))$.
- V případě přibližně stejně velkých polí můžeme použít Merge, tj. $\mathcal{O}(m + n)$.

5.27 Nech A je posloupnost n čísel, která obsahuje l inverzí (inverze je dvojice i, j taká, že $A[i] > A[j]$). Jaká bude složitost algoritmu InsertSort na posloupnosti A?

▌ Složitost algoritmu bude $\mathcal{O}(n + l)$.

5.28 Nech A je posloupnost n čísel, v které je každý prvek nanejvýš k pozicí od své pozice v seřazené postupnosti. Jaká bude složitost algoritmu InsertSort na posloupnosti A?

▌ Složitost algoritmu bude $\mathcal{O}(n \cdot k)$ – stačí spočítat počet inverzí.

Kapitola 6

Halda a prioritní fronta

Binární strom je grafová datová struktura, ve které každý uzel má maximálně dva potomky.

Kořen stromu je uzel, který nemá rodiče.

List stromu je uzel, který nemá potomky.

Skoro úplný binární strom je takový strom, jehož uzly na všech úrovních kromě posledních 2 mají právě dva následníky.

Binární halda je skoro úplný binární vyvážený strom (kromě spodní vrstvy stromu, která je zarovnaná doleva), ve kterém platí, že každý uzel splňuje vlastnost haldy.

Vlastnost maximové haldy je, že každý uzel je větší nebo roven všem svým potomkům.

Vlastnost minimové haldy je, že každý uzel je menší nebo roven všem svým potomkům.

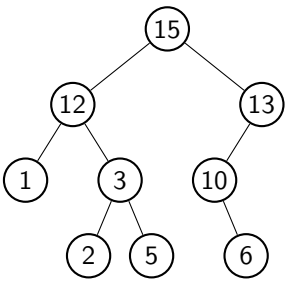
Délka větve stromu je počet uzlů od kořene daného stromu k listu.

Hloubka stromu je délka nejdelší větve.

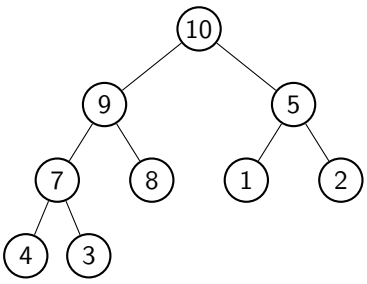
Vyvážený binární strom je takový, kde délka větví se od sebe liší maximálně o jedna.

6.1 Jsou následující stromy binární haldy? Odpověď zdůvodněte.

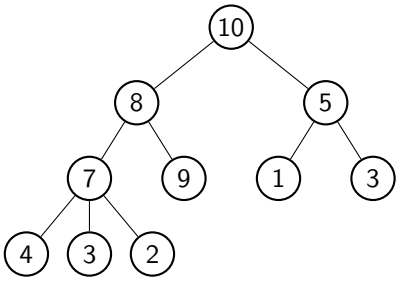
a)



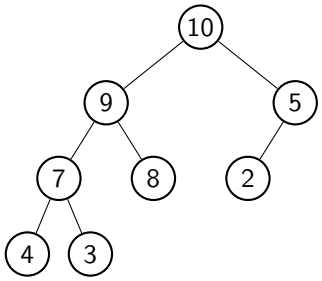
b)



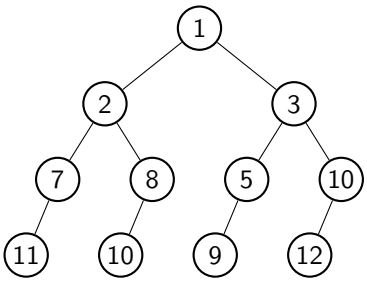
c)



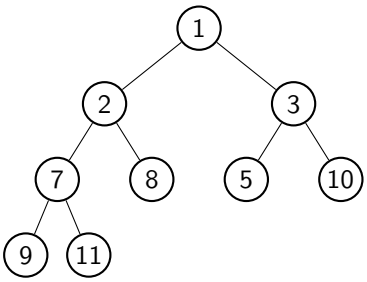
d)



e)



f)



a) není úplný strom, obsahuje neuspořádanou větev 15, 12, 3, 5 a není nalevo zarovnaná

b) je maximová halda

c) není, obsahuje neuspořádanou větev 10, 8, 9 a uzel 7 obsahuje 3 listy

d) není úplný strom

e) není nalevo zarovnaná

f) je minimová halda

Paní Bílá připomíná: V maximové haldě může existovat uzel, jehož vnučka je větší, než jeho strýc.

6.2 Nakreslete všechny možné maximové binární haldy, které mohou vzniknout z prvků 1, 2, 3, 4 a 5.

Možností je 8. Strom má stále stejnou strukturu, je to úplný binární strom (mimo spodní vrstvu s listy zarovnanými doleva). V kořeni je vždy 5. V levém synovi kořene je buď 4 nebo 3. Když 4, tak zbývající čísla mohou být libovoně (6 možností). Když 3, tak 4 musí být sourozenec (2 možnosti pro umístění 1 a 2).

6.3

a) Jaký je maximální počet prvků v binární haldě hloubky h?

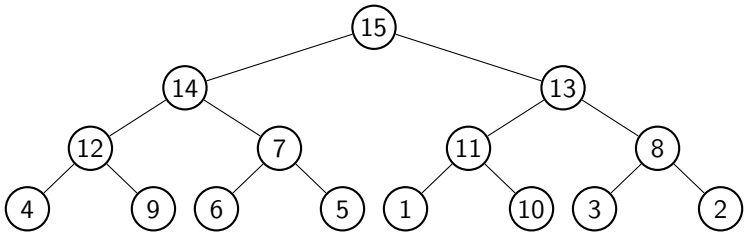
Bude to úplný strom hloubky h. Víme, že každé patro nám exponenciálně zvýší počet prvků a v první vrstvě se nachází jenom kořen, tudíž počet uzlů je $n_{max} = 2^h - 1$.

b) Jaký je minimální počet prvků v binární haldě hloubky h?

Aby měla halda hloubku h, musí mít h pater, přičemž z definice haldy víme, že poslední patro nemusí být plné. Aby byl počet prvků minimální, poslední patro bude obsahovat 1 prvek a zbylých $h-1$ pater budou plné. Z předchozího dostáváme, že $n_{min} = (2^{h-1}-1)+1 = 2^{h-1}$.

6.4

a) Přepište následující haldu do reprezentace polem.



$A = [15, 14, 13, 12, 7, 11, 8, 4, 9, 6, 5, 1, 10, 3, 2]$

Můžete se zeptat, jestli je to preorder, inorder, nebo postorder. (Odpověď je ani jedno). Ještě jsme to necvičili, ale na přednášce už to bylo.

b) Je každé vzestupně seřazené pole minimová halda?

Ano.

c) Je každá minimová halda vzestupně seřazené pole?

Ne.

6.5 Na jakých pozicích se může nacházet nejmenší prvek v maximové binární haldě?

Podle vlastnosti haldy víme, že si uchovává uspořádání na svých prvcích. Tudíž nejmenší prvky v maximové haldě se nacházejí v listech stromu.

6.6

a) Sestavte postupně minimovou haldu h z následujících prvků: 36 19 25 100 17 2 3 7 1.

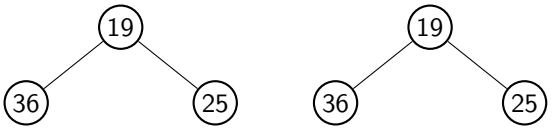
Řešení doporučujeme kreslit krok za krokem všechny změny co se na haldě provádí. Cvičení by mělo studentům předat intuici za operacemi na haldě, ne jenom mlhavou představu o provedení operace Insert nebo Delete.

1. Insert$(H, 36)$

2. Insert$(H, 19)$

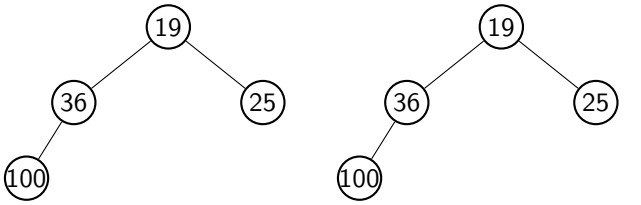
1. vložení 2. po kontrole zdola nahoru

3. Insert$(H, 25)$



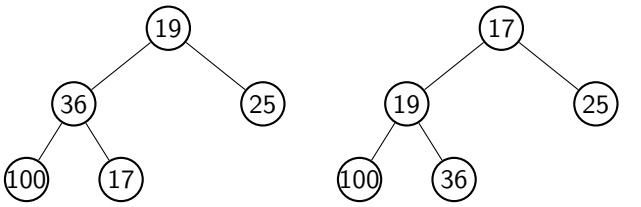
1. vložení 2. po kontrole zdola nahoru

4. Insert$(H, 100)$



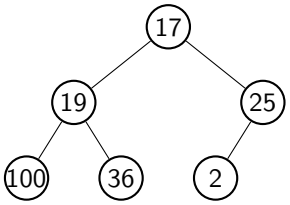
1. vložení 2. po kontrole zdola nahoru

5. Insert$(H, 17)$

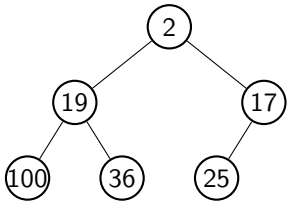


1. vložení 2. po kontrole zdola nahoru

6. Insert$(H, 2)$

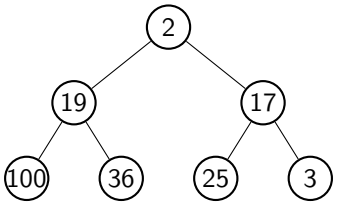


1. vložení

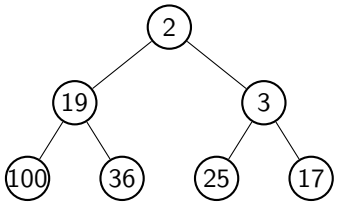


2. po kontrole zdola nahoru

7. Insert$(H, 3)$

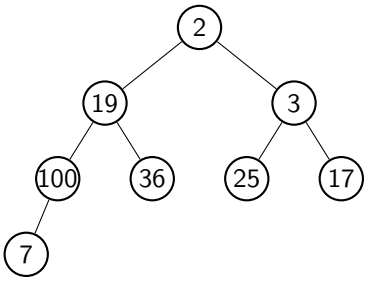


1. vložení

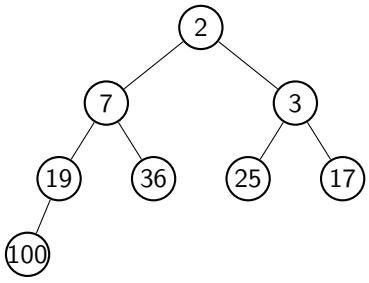


2. po kontrole zdola nahoru

8. Insert$(H, 7)$

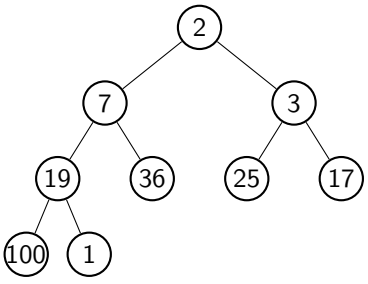


1. vložení

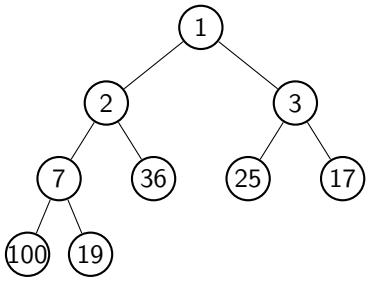


2. po kontrole zdola nahoru

9. Insert$(H, 1)$



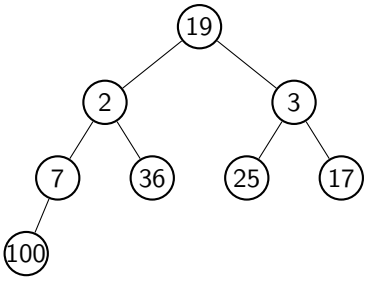
1. vložení



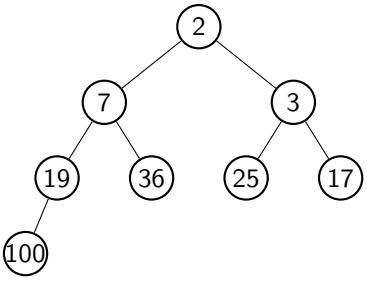
2. po kontrole zdola nahoru

b) Smažte z haldy 3 krát po sobě kořen. Použijte operaci ExtractMin(h).

1. ExtractMin(H)

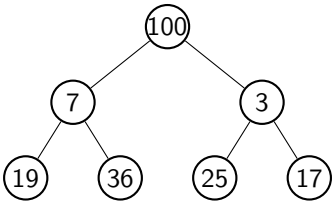


1. odstranění kořene

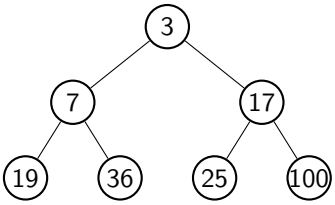


2. po kontrole shora dolu

2. ExtractMin(H)

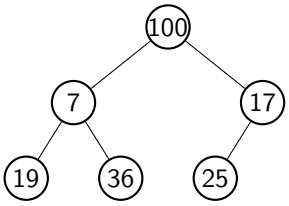


1. odstranění kořene

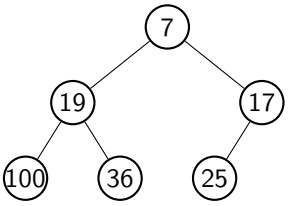


2. po kontrole shora dolu

3. ExtractMin(H)



1. odstranění kořene



2. po kontrole shora dolu

6.7 Mějme datové struktury – maximová halda, minimová halda, seřazené pole, seřazený seznam, neseřazené pole a neseřazený seznam využity pro prioritní frontu.

Vyplňování tabulky zabere více času. Proto doporučujeme si již předem nachystat tabulku na tabuli a postupně ji se studenty jenom vyplnit.

a) Určete časovou složitost operací *insert*, *remove maximum*, *remove* (s konkrétním ukazatelem na prvek ke smazání), *find maximum*, *change priority* (která změní prioritu zadaného prvku), *join* nad těmito datovými strukturami.

Efektivnost jednotlivých operací na prioritní frontě reprezentované různými datovými strukturami znázorňuje následující tabulka:

	insert	*remove*	*find maximum*	*remove maximum*	*change priority*	*join*
maximová halda	$\log(N)$	$\log(N)$	1	$\log(N)$	$\log(N)$	N
minimová halda	$\log(N)$	$\log(N)$	N	N	$\log(N)$	N
seřazené pole	N	N	1	1	N	N
seřazený seznam	N	1	1	1	N	N
neseřazené pole	1	N	N	N	1	N
neseřazený seznam	1	1	N	N	1	1

b) Mějme aplikaci, která provádí velký počet hledání maximálního prvku, ale málo vkládání a mazání maximálního prvku. Která datová struktura je nejefektivnější pro implementaci prioritní fronty?

Pro zadanou aplikaci stačí, aby byla prioritní fronta implementována datovou strukturou se složitostí FindMaximum v $\mathcal{O}(1)$. To jsou všechny seřazené posloupnosti a halda.

c) V jakým případě je efektivnější použít seřazený seznam, než seřazené pole?

Z porovnání datových struktur je seřazený seznam efektivnější v případě, že se ze struktury často odebírají prvky.

6.8 Navrhněte datovou strukturu, která v konstantním čase vrací minimum i maximum. Složitost vkládání i odstraňování je v čase $\mathcal{O}(\log(n))$.

Příklad může zabrat více času. Ve skupinách, kde nebudou schopni přijít na myšlenku, doporučujeme postupně dávat nápovědy k řešení. Ptejte se studentů na to, která datová struktura nám umožňuje dané operace provádět v kýženém čase.

Řešení může vypadat tak, že si udržujeme paralelně dvě haldy – jednu minimovou a druhou maximovou. Při vkládaní vložíme prvek do obou hald. Pro případ mazáni si musíme s každým prvkem pamatovat referenci na tentýž prvek v druhé haldě. Pak při mazání odstraníme prvek z obou hald za využití reference.

6.9

a) Navrhněte algoritmus, který v čase $\mathcal{O}(k \cdot \log(n))$ najde k-tý nejmenší prvek v minimové haldě obsahující n prvků.

Příklad by měl vést k intuici k HeapSort algoritmu, který nasleduje v dalších příkladech.

Prvek najdeme pomocí $(k-1)$-krát zavolané operace ExtractMin. Což nám přesune hledaný prvek na vrchol haldy. Odstraněné prvky můžeme dočasně uchovávat v jiné datové struktuře a po nalezení k-teho prvku je znovu do haldy vložit, co nám složitost nepokazí (provedeme k vložení). Celkova složitost tedy bude $\mathcal{O}(k \cdot \log(n))$.

b) * Navrhněte algoritmus, který problém řeší v čase $\mathcal{O}(k \cdot \log(k))$.

Možná je vhodné před algoritmem HeapSort probrat algoritmus BuildHeap. Není zrovna intuitivní a složitost v lineárním čase je pro studenty nepochopitelná. Jedná se o opakování přednášky, ale opakování asi nejtěžší části je opodstatněné.

Jinak studenti vytvoření haldy v algoritmu HeapSort řeší pomocí n vkládání do haldy.

6.10

a) Navrhněte algoritmus, který pomocí haldy seřadí pole.

Procedura HeapSort(A, n)

vstup: pole A délky n
výstup: vzestupně seřazené pole A původních prvků

```
1 BuildHeap(A, n)
2 A.heapsize ← n
3 for i ← n downto 2 do
4     Swap(A[1], A[i])
5     A.heapsize ← A.heapsize − 1
6     Heapify(A, 1)
7 od
```

b) Jaká je časová složitost řazení haldou? Je HeapSort in situ? Porovnejte ho s již známými algoritmy.

HeapSort má časovou složitost $n \cdot \log(n)$. V lineárním čase vybuduje maximovou haldu pomocí BuildHeap, pak vždy prohodí kořen s posledním uzlem haldy (kořen je maximum, takže se zařadí na správné místo) a následně v logaritmickém čase haldu opraví pomocí kontroly shora dolů. Tato časová složitost vede ke srovnání s algoritmami MergeSort a QuickSort.

HeapSort je narozdíl od algoritmu MergeSort in situ.

Díky nulové extrasekvenční složitosti se HeapSort hodí do slabších počítačů. Dříve fungoval rychleji než MergeSort, ale díky rychlejší paměťové hierarchii dnešních PC už převládá lepší časová složitost algoritmu MergeSort (MergeSort a HeapSort se liší jen v konstantách). MergeSort je navíc snadno paralelizovatelný.

c) Je HeapSort stabilní řadící algoritmus? Své tvrzení dokažte.

Není. Protipříklad můžeme být například pole obsahující dvě jedničky (jejich pořadí se změní).

6.11 Navrhněte následující operace nad minimovou binární haldou a určete jejich časovou složitost.

a) Na jakých pozicích bude při reprezentaci binární haldy polem rodič prvku, pravý a levý potomek?

Můžeme psát následující funkce pro získaní rodiče a potomků v haldě:

Funkce Parent(i)
vstup: index uzlu i
výstup: index rodiče uzlu s indexem i
1 **return** $\lfloor i/2 \rfloor$

Funkce Left(i)
vstup: index uzlu i
výstup: index levého potomka uzlu s indexem i
1 **return** $2i$

Funkce Right(i)
vstup: index uzlu i
výstup: index pravého potomka uzlu s indexem i
1 **return** $2i + 1$

b) Minimum najde minimum v haldě.

Procedura vrátí klíč v kořeni haldy. Tedy operace má konstantní časovou složitost.

Procedura Minimum(h)
vstup: halda reprezentovaná polem h
výstup: minimální hodnota v haldě h
1 **return** $h[1]$

c) Navrhněte algoritmus Heapify, který zkontroluje a opraví vlastnost haldy z daného uzlu i.

Možné řešení může vypadat následovně:

Procedura Heapify(h, i)
vstup: pole h reprezentující haldu a index i uzlu, kterého podstrom se má kontrolovat
1 **if** Left(i) $\leq h.size \wedge h[\text{Left}(i)] < h[i]$ **then**
2 $smallest \leftarrow$ Left(i)
3 **else**
4 $smallest \leftarrow i$
5 **fi**
6 **if** Right(i) $\leq h.size \wedge h[\text{Right}(i)] < h[smallest]$ **then**
7 $smallest \leftarrow$ Right(i)
8 **fi**
9 **if** $smallest \neq i$ **then**
10 Swap($h[i], h[smallest]$)
11 Heapify($h, smallest$)
12 **fi**

Procedura Heapify nám zajišťuje dodržení základní vlastnosti haldy. Musíme vybrat nejmenší z uzlů i, Left(i) a Right(i). Pokud je nejmenší uzel v některém z potomků, pak daného potomka s uzlem i prohodíme a následně provedeme kontrolu na potomku.

Toto rekurzivní volání nám zaručuje, že se opraví libovolná nově vzniklá dvojice porušující pravidlo haldy. Operace proběhne v logaritmickém čase, jelikož maximální počet kroků je délka dané větve. Této proceduře se někdy říká top-down check.

d) Pomocí algoritmu Heapify navrhněte způsob, jak z neseřazeného pole vytvořit haldu. Navrhněte tedy algoritmus BuildHeap.

Možné řešení může vypadat následovně:

Procedura BuildHeap(A, n)

vstup: pole A délky n

```
1 for i ← ⌊n/2⌋ downto 1 do
2     Heapify(A, i)
3 od
```

Intuitivní postup, jak z pole vytvořit haldu, by bylo postupné vkládání prvků od začátku pole a stavění haldy „shora“. Tento postup by měl časovou složitost $n \cdot \log(n)$, jelikož bychom museli n-krát vložit prvek do haldy se složitostí $\log(n)$.

Algoritmus BuildHeap buduje haldu „zdola nahoru“. Tento algoritmus prvně prohlásí dolních $\lceil n/2 \rceil$ uzlů za korektní jednoprvkové haldy. V dalším kroku je začne spojovat přes uzly v předposledním patře, přičemž vždy provede kontrolu procedurou Heapify.

Časová složitost je lineární, což se zdá možná trochu překvapivé (co se změnilo oproti intuitivnějšímu algoritmu?).

Nechť $n = 2^h - 1$, kde h je výška haldy, předpokládáme tedy plně zaplněnou haldu. Na nejspodnější patro se Heapify nevolá, cena je tedy 0. Na předposledním patře je cena Heapify pro jeden uzel 1, uzlů je 2^{h-1}, celková cena pro patro je 2^{h-1}, na vyšších patrech je cena vždy počet uzlů krát výška, tedy $2^{h-k} \cdot k$. Celková složitost je dána součtem složitostí jednotlivých pater, tedy

$$T(n) = \sum_{k=0}^{h} k \cdot 2^{h-k} = \sum_{k=0}^{h} k \cdot \frac{2^h}{2^k} = 2^h \sum_{k=0}^{h} \frac{k}{2^k}.$$

Součet sumy $\sum_{k=0}^{h} \frac{k}{2^k}$ je 2 (je to harmonická řada, kterou znáte z předášky), celková složitost tedy je $2^{h+1} = 2n + 2 \in \Theta(n)$.

e) DecreaseKey(H, i, key) sníží hodnotu prvku i na hodnotu key.

Možné řešení může vypadat následovně:

Procedura DecreaseKey(h, i, key)

vstup: pole h reprezentující haldu, key reprezentující vkládanou hodnotu na pozici i

```
1 if key > h[i] then
2     return nová hodnota je větší než vkládaná
3 fi
4 h[i] ← key
5 while i > 1 ∧ h[Parent(i)] > h[i] do
6     Swap(h[i], h[Parent(i)])
7     i ← Parent(i)
8 od
```

Pokud snižujeme hodnotu nějakého uzlu, musíme následně provést kontrolu od tohoto prvku směrem nahoru, abychom mu dali možnost „vybublat“ až na místo kořene, pokud byla uzlu přiřazena hodnota menší než kořen. Kontrolujeme, zdali není hodnota v uzlu menší než rodič uzlu. Pokud je, pak je prohodíme a rekurzivně voláme DecreaseKey (v pseudokódu výše je rekurze převedena na iteraci). Maximální počet volání funkce je $\log(n)$, což je maximální délka větve.

Této proceduře se také někdy říká bottom-up check.

f) Insert vloží prvek do haldy.

Možné řešení může vypadat následovně:

Procedura Insert(h, key)

vstup: pole h reprezentující haldu, key reprezentující vkládanou hodnotu

```
1 h.size ← h.size + 1
2 h[h.size] ← ∞
3 DecreaseKey(h, h.size, key)
```

Do haldy vždy vkládáme na první prázdnou pozici, tedy za poslední prvek v poli, kterým haldu reprezentujeme. Vložíme prvek a provedeme kontrolu od tohoto prvku směrem nahoru. Abychom využili už existující kód, tak můžeme vložit klíči ∞ a spustit na daný prvek proceduru DecreaseKey.

g) ExtractMin smaže minimální prvek.

Možné řešení může vypadat následovně:

Procedura ExtractMin(h)

vstup: pole h reprezentující haldu
výstup: minimum (tedy bývalý kořen) haldy

```
1 if h.size < 1 then
2     return prázdna halda
3 fi
4 min ← h[1]
5 h[1] ← h[h.size]
6 h.size ← h.size − 1
7 Heapify(h, 1)
8 return min
```

Pokud bychom měli odstranit nějaký uzel haldy, pak na jeho místo dáváme vždy poslední uzel, aby byla halda stále zarovnaná. Pokud odstraňujeme minimum (což potřebujeme například pro HeapSort), pak po vložení posledního prvku musíme provést kontrolu shora dolů, tedy Heapify.

6.12 Naprogramujte operace nad haldou, pro implementaci využijte připravené šablony ve studijních materiálech – C a Python.

Následující příklady jsou vhodné na domácí studium.

6.13 Dokažte pomocí indukce, že halda s n uzly má přesně $\lceil n/2 \rceil$ listů.

6.14 Navrhněte algoritmus, který ověří, jestli je pole korektní binární halda. Jaká je složitost vašeho algoritmu?

6.15 Mějme pole A o n prvcích, které mají být naráz vloženy do již existující haldy. Navrhněte algoritmus, který tuto operaci provede v čase $\mathcal{O}(n)$.

6.16 S použitím maximové haldy navrhněte algoritmus, který rozhodne, jestli k-tý největší prvek je větší nebo roven hodnotě x. Algoritmus bude mít složitost závislou na k.

6.17

Paní Bílá připomíná: Pro každou prababičku uzlu *já* v maximové haldě platí, že neteř strýce uzlu *já* je menší.

Největší prvek v maximové haldě se nachází na pozici 1. Druhý největší na pozici 2 nebo 3.

- Zamyslete se, kde se může v binární maximové haldě nacházet k-tý největší prvek.
- Určete konkrétní hodnoty pro $k = 2$, 3 a 4.

Můžete předpokládat, že hodnoty jsou vzájemně rozdílné.

6.18 Jaký typ prioritní fronty byste použili pro hledání 100 největších čísel v 10^6 náhodných číslech? Uveďte výhody a nevýhody jednotlivých typů.

6.19 Navrhněte algoritmus, který vytvoří z minimové haldy maximovou haldu. Jakou má algoritmus složitost?

> Změna haldy lze nejsnáze provést vytvořením nové haldy. Pomocí BuildHeap to lze udělat v lineárním čase (lépe to nejde).

6.20 Navrhněte efektivní reprezentaci n-árnej haldy pomocí pole. Popište detailně, jak se dostat z rodiče k jeho n dětem.

6.21 Navrhněte algoritmus HeapSort, který využívá n-árnou haldu.

6.22 Jaký řadící algoritmus je podobný algorimu HeapSort, který využívá n-árnou haldu, kde $n = 1$.

Kapitola 7

Binární vyhledávací stromy

Strom je souvislý neorientovaný graf bez cyklů, kde sousednost dvou uzlů vyjadřuje vztah rodič a potomek. Uzel může mít více potomků, ale pouze jednoho rodiče.

Větev je libovolná cesta mezi kořenem a listem.

Délka větve stromu je počet uzlů větve.

Hloubka stromu je délka nejdelší větve.

Binární vyhledávací strom je strom, kde každý uzel má 0 – 2 potomků, přičemž platí, že klíč každého uzlu je větší nebo roven než všechny klíče v jeho levém podstromě a menší nebo roven všem klíčům v jeho pravém podstromě.

Vyvážený binární strom má hloubku $\lceil \log_2(n+1) \rceil$, kde n je počet uzlů stromu.

Průchod inorder prochází prvně uzly v levém podstromu, následně kořen a nakonec uzly pravého podstromu.

Průchod preorder prochází prvně kořen stromu a následně uzly v levém a pravém podstromu.

Průchod postorder prochází uzly v levém a pravém podstromu, teprve potom kořen stromu.

Pan Usměvavý dodává: Pařez nemůže být strom, jelikož obsahuje kružnice.

Tento příklad nemá smysl cvičit, studenty na odpovědi nic moc nenavádí, jdete raději dál. Příklad začíná jako první pro domácí studium (kde může fungovat lépe), na cviku jej přeskočte.

7.1 Prodiskutujte odpovědi na následující otázky.

a) Kde se s datovou strukturou strom setkáváte v normálním světě? K čemu se hodí reprezentace stromovou strukturou?

Stromy reflektují strukturu vztahů jednotlivých uzlů. Mohou nám popisovat hierarchii dat. Pokud dáme mezi rodiče a potomka vazbu „větší než“, získáme maximovou haldu. Jiným seřazením můžeme získat vyhledávací strom.

Stromy využíváte například při řešení matematických rovnic, které lze reprezentovat jako stromy tak, že do listů dáte čísla a jejich rodiče budou operátory. Čím má operátor vyšší prioritu, tím níže ve větvi stromu je. Pokud jsou potomci uzlu listy, pak můžeme hodnotu tohoto uzlu vypočítat. Takto můžeme postupovat až ke kořeni a získáme korektní výsledek.

b) Jaké operace jsou u stromů (ne nutně vyhledávacích) výhodné?

Právě díky vazbám mezi uzly nám stromy dávají jednoduchou reprezentaci uspořádaní prvků, díky kterému dokážeme určit v jakém vztahu jsou vzájemně prvky v určité hierarchii.

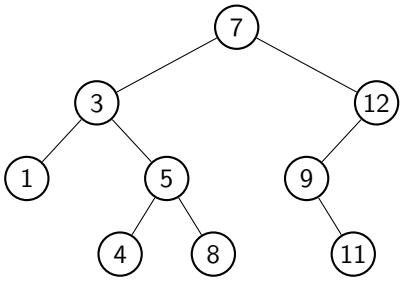
Další výhodou některých stromů může být jejich snadné spojování. Pokud nám nebrání specifické uspořádání, pak můžeme spojení stromů realizovat převěšením jednoho stromu na druhý v konstantním čase.

c) Jaké operace jsme na vyhledávacích stromech schopni realizovat a jakou mají složitost?

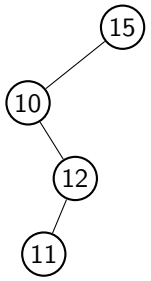
Typicky na BVS definujeme tyto operace: Search, Insert, Delete, Minimum a Maximum, které jsou v lineárním čase vzhledem k délce větve. Ještě mají smysl Predecessor a Succesor (v nejhorším případě se stejnou složitostí).

7.2 Rozhodněte, zdali jsou následující stromy BVS. Pokud ne, navrhněte, jak nejsnadněji stromy opravit:

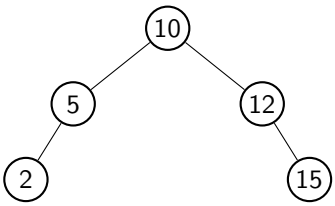
a)



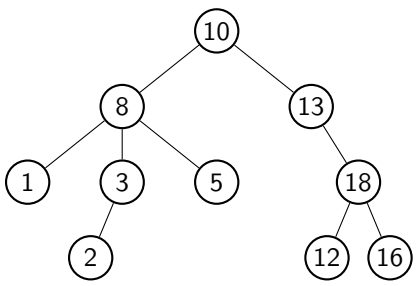
b)



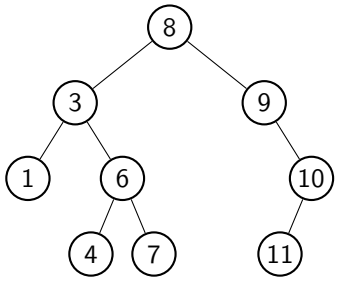
c)



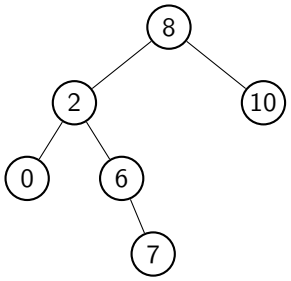
d)



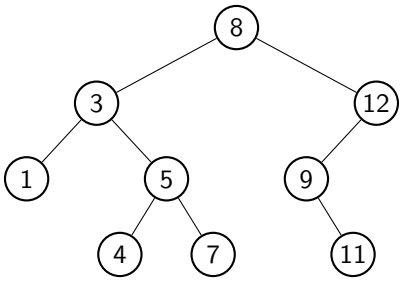
e)



f)



a) Tento binární strom obsahuje uzel s klíčem 8 na levé straně od kořenu s klíčem 7. Nejsnazší opravou je tedy prohození těchto dvou klíčů.

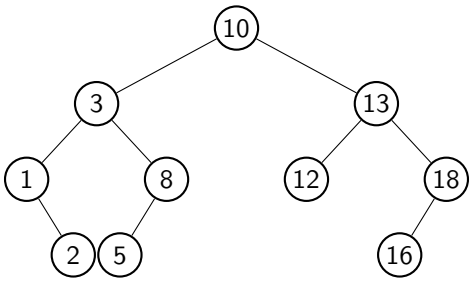


b) Jedná se o korektní BVS.

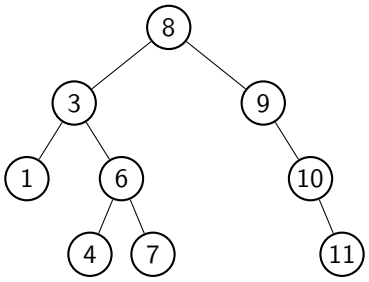
c) Jedná se o korektní BVS.

d) Tento strom porušuje celou řadu pravidel. Nejzřetelnější je, že se nejedná o binární, ale ternární strom. Další chybou je špatná pozice uzlů s klíči 5, 12 a 16. Díky více chybám existuje více možností opravení chyb.

V levém podstromu by bylo vhodné najít jiný kořen. Ideální je medián, aby byl strom alespoň trochu vyvážený. Proto zvolíme za kořen číslo 3. 2 zavěsíme za uzel 1, dvojici uzlů 8 - 5 můžeme nechat, ale musíme 5 zavěsit jako levého potomka. Tím je levý podstrom vyřešen. V pravém podstromu musíme 12 zavěsit doleva pod 13 a uzel 16 přesuneme na místo levého potomka pod 18.

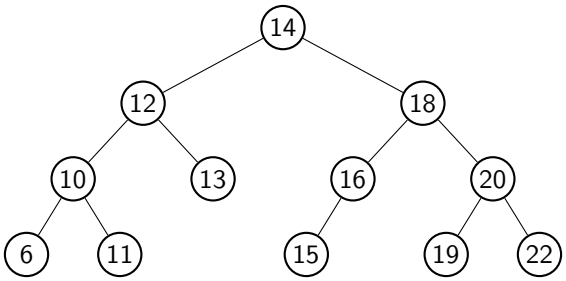


e) Zde je jen uzel s klíčem 11 zavěšen na špatné straně stromu. Stačí jej přesunout na pravou stranu a získáváme BVS.



f) Jedná se o korektní BVS.

7.3 Mějme následující BVS:



Každý uzel obsahuje 4 atributy: ukazatele na rodiče *node.parent*, ukazatel na levého *node.left* a pravého *node.right* syna a klíč *node.key*. Nechť je $Node(x)$ ukazatelem na uzel s klíčem x, tedy $Node(14)$ je ukazatel na kořen našeho stromu. Prázdný ukazatel odpovídá hodnotě *nil*.

Čemu budou odpovídat následující výrazy?

a) $Node(20).parent.left.left.key$

15

b) $Node(13).parent.parent.parent$

nil

c) $Node(14).left.left.right$

$Node(11)$

d) $Node(12).parent.right.right.left.key$

19

Průchody je rozhodně třeba probrat, studenti by je měli detailně chápat (nejen odkývat). Když je znají, tak lze mnohem lépe diskutovat další příklady. Hlavně z nich zkuste dostat odpověď, jaký průchod se k čemu hodí. Části za b) a c) nechte studentům na doma.

7.4

a) Jaké existují způsoby průchodu stromů? Porovnejte, jak se liší a k čemu se který hodí.

Průchod preorder může sloužit ke snadnému kopírování stromu. Výstupní sekvence obsahuje informaci o struktuře stromu („pohledem shora“). Stačí tedy výstupní sekvenci použít jako vstup nového stromu a vznikne stejný strom, jako byl původní procházený.

Inorder výpis můžeme použít například k výpisu seřazené posloupnosti, kterou ukládáme do binárního vyhledávacího stromu. Lze tedy použít ke kontrole validity stromu.

Postorder průchod je vhodný například pro mazání stromu. Jelikož se prochází strom od listů ke kořeni, máme zaručeno, že nepřijdeme o žádnou paměť ztrátou ukazatele.

Průchod stromem preorder v rekurzivní podobě:

Procedura PreorderRecursive(*root*)

vstup: *root* – kořen stromu/podstromu

```
1  if root = nil then
2      return
3  fi
4  vypiš root.key
5  PreorderRecursive(root.left)
6  PreorderRecursive(root.right)
```

Průchod stromem preorder v iterativní podobě:

Procedura PreorderIterative(*root*)

vstup: *root* – kořen stromu/podstromu

```
1  stack ← prázdný zásobník
2  Push(stack, root)
3  while stack není prázdný do
4      root ← Pop(stack)
5      vypiš root.key
6      if root.right ≠ nil then
7          Push(stack, root.right)
8      fi
9      if root.left ≠ nil then
10         Push(stack, root.left)
11     fi
12 od
```

Průchod stromem inorder v rekurzivní podobě:

Procedura InorderRecursive(*root*)

vstup: *root* – kořen stromu/podstromu

```
1  if root = nil then
2      return
3  fi
4  InorderRecursive(root.left)
5  vypiš root.key
6  InorderRecursive(root.right)
```

Průchod stromem postorder v rekurzivní podobě:

Procedura PostorderRecursive(*root*)

vstup: *root* – kořen stromu/podstromu

```
1 if root = nil then
2     return
3 fi
4 PostorderRecursive(root.left)
5 PostorderRecursive(root.right)
6 vypiš root.key
```

b) * Lze se pohybovat ve stromě bez ukazatele na rodiče? Můžete k tomu použít nějakou pomocnou strukturu, která by se na to hodila?

Ano, lze. Jen si v pomocné struktuře musíme pamatovat uzly, ve kterých jsme již byli. Tato struktura by měla sloužit pouze pro vkládání a odstraňování, na což se hodí třeba zásobník.

Pokud bychom použili místo zásobníku frontu, pak bychom se stromem pohybovali stejně, jak vypadá reprezentace haldy polem. Takovému průchodu se v kontextu grafů říká Breadth-first (při vyhledávání Breadth-first search = BFS). To se může hodit například při teoretickém nekonečném stromu. Při využití zásobníku se prvně zanoříme až na úroveň listů, což odpovídá Depth-first průchodu (vyhledávání DFS).

Pokud nám jde jen o vypsání, pak stačí samotná rekurze. Při ní si však sám procesor bude tvořit zásobník průchodu, rekurzivní řešení je tedy také Depth-first.

c) * Jak se změní předchozí iterativní algoritmy, pokud místo zásobníku použijete frontu? V jakém pořadí se budou vypisovat klíče?

Průchod stromem se provede pomocí prohledávání do šířky:

Procedura BreadthFirst(*root*)

vstup: *root* – kořen stromu/podstromu

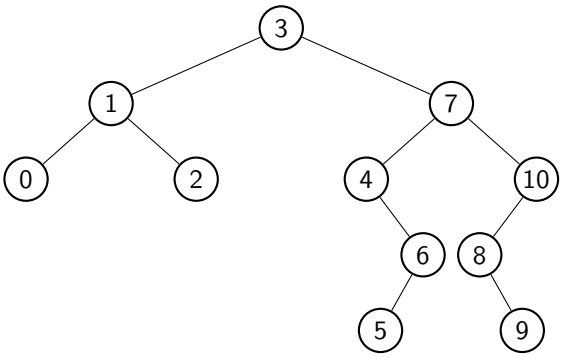
```
1  queue ← prázdná fronta
2  Enqueue(queue, root)
3  while queue není prázdná do
4      root ← Dequeue(queue)
5      vypiš root.key
6      if root.left ≠ nil then
7          Enqueue(queue, root.left)
8      if root.right ≠ nil then
9          Enqueue(queue, root.right)
10 od
```

7.5

Při řešení postupujte postupně jako algoritmus a ukažte, kde se co kontroluje. Nakreslený strom můžete využít k procvičení hledání konkrétního prvku. Diskutujte složitosti algoritmů, u 7.11 už na to asi nebudete mít prostor. Zato lineární složitost vzhledem k délce větví vám může pomoci navázat další příklad (na délky větví stromu).

a) Vytvořte BVS postupným vkládáním prvků 3, 1, 7, 10, 4, 8, 9, 5, 6, 0 a 2.

Pro ověření vašeho řešení můžete použít online nástroj. Výsledný strom vypadá následovně:



b) Nalezněte ve stromu minimum a maximum (algoritmicky).

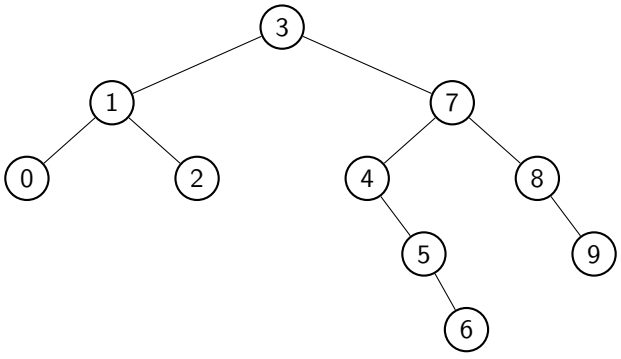
Minimum je vždy v nelevějším uzlu, musíme tedy postupovat tak dlouho doleva, dokud to jde, což nám dává uzel s klíčem 0. Obdobně funguje funkce maximum, která najde nejpravější uzel s klíčem 10.

c) Postupně odstraňte z tohoto stromu prvky 10, 3 a 4.

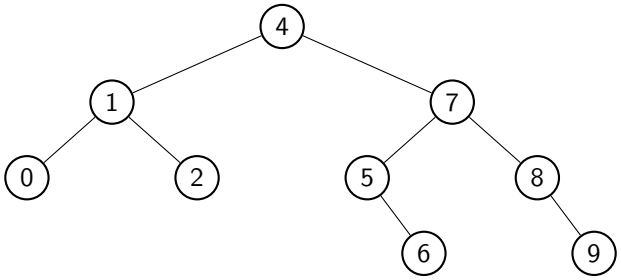
Pokud máte šikovnou skupinu, jako rozšiřující příklad můžete procvičit i příklad 2.20 který se ptá na komutativitu operace Delete.

Pro ověření vašeho řešení můžete použít online nástroj. (Pozor nástroj nefunguje přesně podle přednáškové implementace)

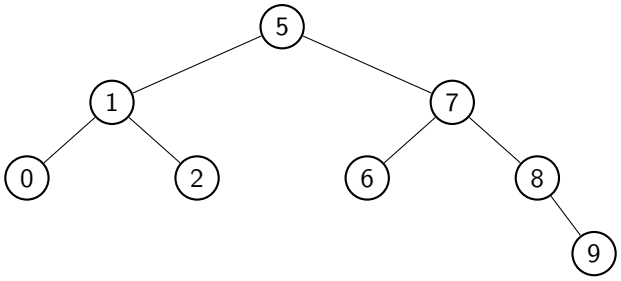
- Smazání hodnoty 10:



- Smazání hodnoty 3:



- Smazání hodnoty 4:



7.6 Navrhněte, v jakém pořadí musí být vkládány hodnoty 1, 2, 3, 4, 5, 6 a 7 tak, že výsledné stromy budou splňovat následující vlastnosti:

a) Strom odpovídá lineárnímu seznamu (všechny prvky jsou v jedné větvi).

Vkládaná posloupnost bude seřazená. Například posloupnost hodnot 1, 2, 3, 4, 5, 6 a 7.

b) Strom obsahuje větev délky právě 5.

Posloupnost si můžeme rozdělit v tom bodu, kde pravá strana bude obsahovat 5 prvků. Jako kořen zvolíme klíč 3 a ten vložíme jako první. Posloupnost tedy bude: 3, 1, 2, 4, 5, 6 a 7.

c) Strom je vyvážený. Kolik stromů může vzniknout?

Posloupnost hodnot musí být v takovém pořadí, že nejprve se vloží kořen výsledného stromu následně jeho děti atd. (vkládáme prvky postupně po patrech výsledného stromu). Posloupnost hodnot tedy bude 4, 2, 6, 1, 3, 5 a 7. Výsledný strom je však vždy stejný.

d) * Kolik různých posloupností je řešením předcházejícího případu?

32 možných posloupností.

7.7 Navrhněte algoritmus, který ověří, zdali je binární strom korektním binárním vyhledávacím stromem.

Se skupinou můžete nejprve vyjmenovat jednotlivá pravidla, která má strom splňovat. Podle vyjmenovaných pravidel pak navrhněte algoritmus, který je ověří. Studenti ověřování korektnosti stromu neměli na přednášce, proto mohou být ztraceni. Jelikož mají tento algoritmus naprogramovat v implementačním úkolu, můžete jim dát čas se s tím potrápit.

Alternativně můžete nejprve procvičit příklad 2.23, ve kterém je algoritmus pro ověření korektnosti BVS. Avšak zmíněný algoritmus je nekorektní, protože nekontroluje splnění pravidel v rámci pater.

Od kořene rekurzivně ověříme, zdali uzly splňují vlastnost binárního vyhledávacího stromu. Strom můžeme procházet například pomocí *inorder* průchodu.

Důležité je také správně předávat návratovou hodnotu. Povšimněte si tedy, že listy jsou korektní vyhledávací stromy, od nich se předává informace *true*. Ale jakmile se nalezne uzel, který nějak pravidla porušuje, je okamžitě propagována informace *false*, kterou již nelze změnit.

Funkce CheckTree($node, min, max$)

vstup: *node* – kořen binárního stromu, *min* a *max* jsou hranice možného intervalu pro klíče
výstup: *true* pokud je strom s kořenem *node* korektní binární vyhledávací strom, jinak *false*

```
1 if node = nil then
2     return true
3 else
4     if node.key < min ∨ node.key ≥ max then
5         return false
6     fi
7     return CheckTree(node.left, min, node.key) ∧ CheckTree(node.right, node.key, max)
8 fi
```

Pro kontrolu celého stromu voláme funkci s následujícími parametry CheckTree($root, -\infty, \infty$).

7.8 Předpokládejme, že máme čísla mezi 1 a 1000 v BVS a hledáme číslo 363. Které z následujících sekvencí nemohou být sekvencemi uzlů při hledání této hodnoty?

a) 2, 252, 401, 398, 330, 344, 397, 363

b) 925, 202, 911, 240, 912, 245, 363

c) 924, 220, 911, 244, 898, 258, 362, 363

d) 2, 399, 387, 219, 266, 382, 381, 278, 363

e) 935, 278, 347, 621, 299, 392, 358, 363

Daná sekvence uzlů představuje binární strom tvořený jednou větví. Cílem je rozhodnout, zda je tento strom BVS.

Sekvence uzlů b) netvoří BVS – uzel 912 leží v levém podstromu uzlu 911.
Stejně tak sekvence e) netvoří BVS – uzel 299 leží v pravém podstromu uzlu 347.

7.9 Navrhněte metodu Height, která vrátí výšku stromu. Určete její složitost.

Opět algoritmus, který neměli studenti na přednášce, ale měli by ho být všichni schopni zapsat. Studenti by se měli nad problémem zamyslet a napsat si rekurzivní funkci na papír, až pak je vhodné prodiskutovat správné řešení s celou skupinou.

Rekurzivní řešení může vypadat následovně:

Funkce Height(*node*)

vstup: *node* – kořen binárního stromu
výstup: výška stromu s kořenem *node*

```
1 if node = nil then
2     return 0
3 fi
4 return 1 + Max(Height(node.left), Height(node.right))
```

7.10 Lze efektivně (tedy lépe než v lineárním čase) spojit dva korektní binární vyhledávací stromy? Zamyslete se nad řešením a jeho složitostí.

Nejjednodušší algoritmus je postupně vkládat hodnoty z menšího stromu do většího. Složitost tohoto algoritmu je odpovídá součinu velikostí obou stromů ($\mathcal{O}(n \cdot m)$, kde n je velikost prvního stromu a m druhého).

Lepší algoritmus by postupoval takto. Z každého stromu bychom vytvořili uspořádaný seznam (k tomu nám stačí inorder průchod), to je v čase $\mathcal{O}(n+m)$. Následně bychom použili proceduru Merge, která nám spojí oba seznamy podobně jako spojení polí v Merge sortu (složitost znovu $\mathcal{O}(n+m)$). Ze seřazeného seznamu lze vytvořit binární vyhledávací strom v lineárním čase (už samotný seznam může být BST pokud si jej představíme jako jednu jedinou větev).

Algoritmus nelze provést lépe než v $\mathcal{O}(n+m)$. Musíme totiž projít všechny uzly stromu.

7.11 Navrhněte následující metody nad BVS, analyzujte jejich složitost a poté je naprogramujte.

Jednotlivé metody již měli studenti na přednášce, měli by je tedy teoreticky znát. Doporučujeme jim proto nechat raději více času na programování, kde se stejně s jednotlivými metodami setkají a můžete jim radit individuálně. Popřípadě se můžete zeptat, kdo neví jak jednotlivé metody implementovat. Tyto studenty si můžete přesunout blíž k sobě, kde s nimi projdete jednotlivé pseudokódy a metody podrobněji (ostatní necháte programovat).

a) Search vrátí uzel s daným klíčem. Pokud se daný klíč ve stromě nenachází, vrátí *nil*.

Procedůra Search může vypadat následovně:

Procedura Search(*node*, *key*)

vstup: prohledávaný podstrom s kořenem *node*, hledaný klíč *key*

```
1 if node = nil ∨ key = node.key then
2     return node
3 fi
4 if key < node.key then
5     return Search(node.left, key)
6 else
7     return Search(node.right, key)
8 fi
```

Nejsnadněji se algoritmus Search zapisuje rekurzivně. V každém průchodu porovnáme hledaný klíč s klíčem aktuálního uzlu, pokud je hledaný klíč menší, pokračujeme doleva, jinak pokračujeme doprava. Rekurzivní zarážkou je nalezení hledaného uzlu, nebo nalezení

nil. Časová složitost závisí na délce větve. Jelikož je délka větve až n, složitost hledání patří do $\mathcal{O}(n)$.

b) Insert vloží prvek do stromu.

Procedůra Insert může vypadat následovně:

Procedura Insert($tree, key$)

vstup: vkladaný klíč *key* do stromu *tree*

```
1  tmp ← nil
2  node ← New(key)
3  subroot ← tree.root
4  while subroot ≠ nil do
5      tmp ← subroot
6      if node.key < subroot.key then
7          subroot ← subroot.left
8      else
9          subroot ← subroot.right
10     fi
11 od
12 node.parent ← tmp
13 if tmp = nil then
14     tree.root ← node
15 else
16     if node.key < tmp.key then
17         tmp.left ← node
18     else
19         tmp.right ← node
20     fi
21 fi
```

Procedura Insert se skládá ze dvou částí. V první části musíme vyhledat pozici, na kterou daný prvek budeme vkládat, aby nebyla porušena vlastnost vyhledávacího stromu. To provedeme tak, že ve smyčce porovnáváme klíč vkládaného prvku s klíčem aktuálního uzlu. Pokud je klíč vkládaného prvku menší, postupujeme na levého syna, pokud je větší, postupujeme na pravého syna. Tento cyklus se zastaví s nalezením prázdného uzlu, což je místo, kam stačí uzel vložit (což už proběhne v konstantním čase). První část algoritmu má časovou složitost závislou na délce stromu. V případě nevyváženého BVS může být délka větve až n, takže složitost vkládání patří do $\mathcal{O}(n)$.

c) Minimum vrátí minimum v stromě pod daným uzlem.

Procedůra Minimum může vypadat následovně:

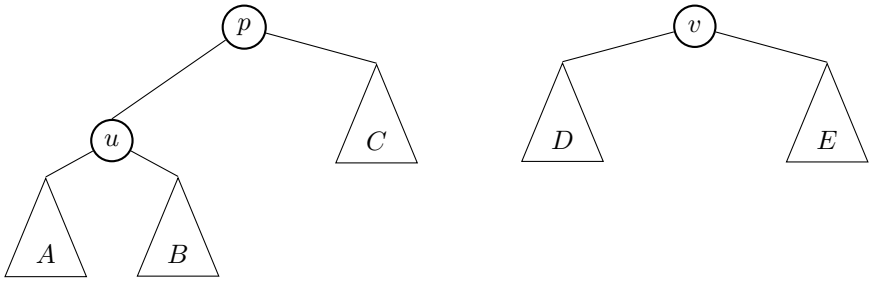
Procedura Minimum(*node*)

vstup: *node* je kořen podstromu od kterého hledáme minimum

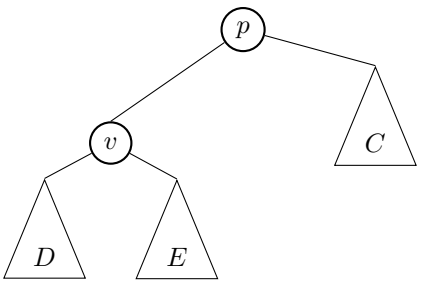
```
1 if node = nil then
2     return nil
3 fi
4 while node.left ≠ nil do
5     node ← node.left
6 od
7 return node
```

Minimum se nachází v nejlevějším uzlu každého stromu. Náš algoritmus se tedy musí zanořovat doleva a v případě, že nějaký uzel levého následníka nemá, algoritmus tento uzel vrátí. Časová složitost se zase odvíjí od délky největší větve, je tedy v $\mathcal{O}(n)$.

d) Transplant nahradí podstrom s kořenem u podstromem s kořenem v tak, že otcem vrcholu v se stane otec vrcholu u. Tuto proceduru budete potřebovat u procedury Delete, která musí nahradit vrchol u vrcholem v tak, aby se správně přepojily původní podstromy vrcholu u a pravý podstrom vrcholu v se musí napojit na správné místo v pravém podstromu vrcholu u. Operace Transplant z následující konfigurace stromů:



Vytvoří konfiguraci:



Procedůra Transplant může vypadat následovně:

Procedura Transplant($tree, u, v$)

vstup: strom *tree*, podstrom u a podstrom v

```
1  if u.parent = nil then
2      tree.root ← v
3  else if u = u.parent.left then
4      u.parent.left ← v
5  else
6      u.parent.right ← v
7  if v ≠ nil then
8      v.parent ← u.parent
9  fi
```

Popis této procedury je součástí popisu Delete.

e) Delete smaže uzel ze stromu. Zamyslete se nad využitím už vymyšlených metod.

Procedůra Delete může vypadat následovně:

Procedura Delete($tree, node$)

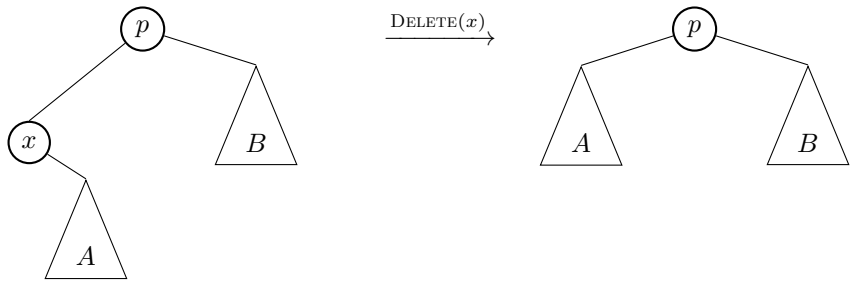
vstup: uzel *node* k odstranění ze stromu *tree*

```
1  if node.left = nil then
2      Transplant(tree, node, node.right)
3  else if node.right = nil then
4      Transplant(tree, node, node.left)
5  else
6      y ← Minimum(node.right)
7      if y.parent ≠ node then
8          Transplant(tree, y, y.right)
9          y.right ← node.right
10         node.right.parent ← y
11     fi
12     Transplant(tree, node, y)
13     y.left ← node.left
14     node.left.parent ← y
```

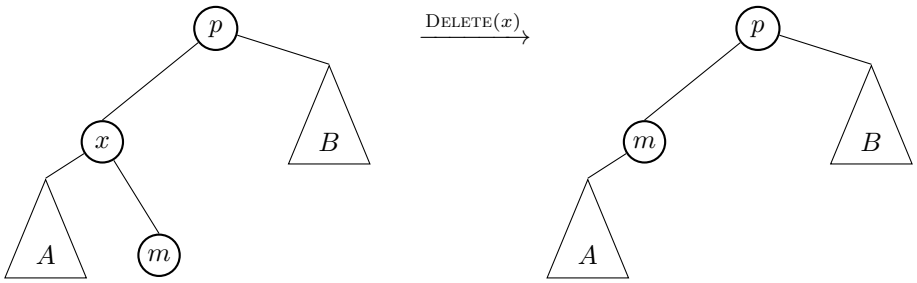
Procedura Delete je závislá na procedurách Transplant pro správné spojení stromů a Minimum pro nalezení náhradního uzlu za ten, který jsme smazali.

Procedura se dá podle situace rozdělit na 3 případy. Prvním je, že uzel nemá žádného potomka, druhým případem je uzel s právě jedním potomkem a třetím případem je uzel s oběma potomky. Náš pseudokód to však díky využití Transplant řeší trochu jinak. Rozdělení podle pseudokódu vypadá spíše takto:

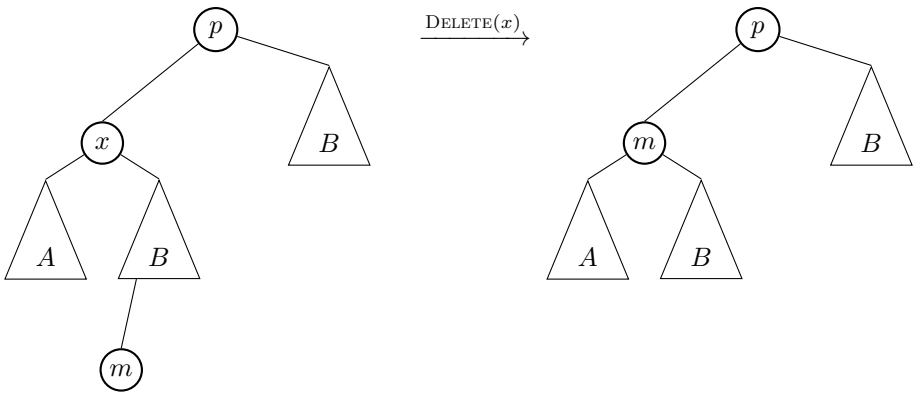
1. Pokud uzel, který chceme odstranit, nemá levého nebo pravého potomka, pak můžeme rovnou zavolat Transplant s parametry původní strom, uzel pro odstranění a druhá větev.



2. Pokud má uzel oba potomky, pak musíme v pravém potomkovi najít minimum. Pokud je toto minimum synem odstraňovaného uzlu, situace se zjednoduší v prosté posunutí minima na místo odstraněného potomka.



3. Pokud se minimum nachází někde jinde, musíme v Transplant minimum na místo odstraňovaného uzlu přesunout a pravý podstrom minima se přesune na původní místo minima.



Časová složitost je v $\mathcal{O}(n)$, což zabere vyhledání prvku a následné hledání minima od tohoto prvku.

7.12 Naprogramujte si operace nad binárním vyhledávacím stromem. Ve studijních materiálech jsou k tomuto připravené zdrojové kódy: C a Python.

Následující příklady jsou vhodné pro domácí studium.

7.13 Najděte 2 prohozené prvky v BST, který byl korektně vytvořen a následně mu byly 2 uzly prohozeny.

Předpokládejme, že v poli a máme inorder průchod stromu. Pak můžou nastat 2 případy.

1. Byly prohozeny sousední klíče. Pak platí, že existuje jenom jeden index p takový, že $a[p] > a[p+1]$.
2. Byly prohozeny klíče, které nejsou na sousedních pozicích. Pak existuje dvojice indexů p a q takové, že platí $a[p] > a[p+1]$ a $a[q-1] > a[q]$. Což znamená, že jsou prohozeny klíče $a[p]$ a $a[q]$.

7.14 Vytvořte alespoň 5 posloupností z následujících klíčů 1, 24, 3, 19, 5, 18 a 8 takové, že když je postupně budeme vkládat do binárního vyhledávacího stromu, vytvoří vyvážený binární vyhledávací strom.

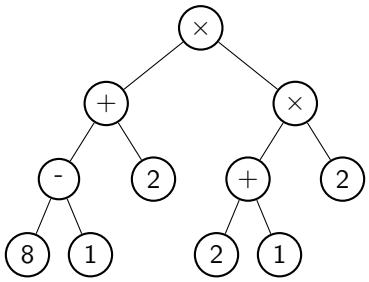
Řešením jsou všechny posloupnosti splňující nasledující vlastnosti:

1. začínají prvkem 8,
2. prvek 3 se v nich vyskytuje před oběma prvky 1 a 5,
3. prvek 19 se v nich vyskytuje před oběma prvky 18 a 24.

Všechny posloupnosti, které začínají číslem 8. Číslo 3 se musí vkládat dříve než čísla 1 a 5. Stejně tak číslo 19 musíme vložit dříve než 18 a 24.

7.15 Jak by se dal efektivně udržovat strom do kterého vkládáme vícero stejných klíčů?

7.16 Napište výraz následujícího stromu pomocí všech možných způsobů průchodů stromů. Jakou výhodu nám dává prefixový a postfixový zápis proti infixovému?



Infixově: $((8-1)+2) \times ((2+1) \times 2)$
Prefixově: $\times\ +\ -\ 8\ 1\ 2\ \times\ +\ 2\ 1\ 2$
Postfixově: $8\ 1\ -\ 2\ +\ 2\ 1\ +\ 2\ \times\ \times$
Jak lze vidět ze zápisů, infixový zápis je jediný, který je nutno závorkovat. Ačkoliv je tedy pro člověka možná nejintuitivnější (protože se jej učíme celý život), ostatní zápisy rovnici popisují jednodušeji.

7.17 Mějme daný strom *tree* a interval $[a, b]$. Navrhněte postup jak získáte všechny uzly stromu *tree* kterých hodnota klíče se nachází v intervale $[a, b]$.

7.18 Navrhněte efektivní způsob jak najít nejbližšího společného předka dvou uzlů v stromě. Algoritmus naprogramujte.

7.19 Mějme uzel *node*, navrhněte algoritmus, který vrací nejbližší (s nejmenší hloubkou) list v podstromě s kořenem *node*.

7.20 Je operace Delete komutativní – smazání uzlu x a pak y dává stejný výsledek jak smazání nejprve uzlu y a pak x?

7.21 Navrhněte reprezentaci binárního vyhledávacího stromu pomocí 3 polí. V prvním poli budou uloženy klíče, v druhém poli jsou ukazatele na uzly nalevo a ve třetím poli ukazatele na uzly napravo. Diskutujte o výhodách a nevýhodách této reprezentace.

7.22 Rozhodněte, zda následující algoritmus správně ověří, zdali je strom korektní binární vyhledávací strom. Pokud ano, dokažte korektnost algoritmu. Jinak uveďte příklad vstupní posloupnosti a vysvětlete, proč výpočet algoritmu pro daný vstup není korektní.

Funkce CheckTree(*node*)

vstup: *node* – uzel binárního stromu
výstup: *true* pokud je strom s kořenem *node* korektní binární vyhledávací strom, jinak *false*

```
1  if node = nil then
2      return true
3  fi
4  if node.left ≠ nil ∧ node.left.key > node.key then
5      return false
6  fi
7  if node.right ≠ nil ∧ node.right.key < node.key then
8      return false
9  fi
10 if not CheckTree(node.left) ∨ not CheckTree(node.right) then
11     return false
12 fi
13 return true
```

7.23 Navrhněte algoritmus, který pomocí BVS seřadí posloupnost čísel. Analyzujte váš algoritmus a porovnejte jej vůči známým řadícím algoritmům.

Výhody řazení stromem jsou: je online, správná implementace je stabilní, pro vyvážené stromy má optimální složitost $\mathcal{O}(n \log(n))$.

7.24 Daný je BVS,klíče x a y a ukazatel na uzel, který obsahuje klíč x. Najděte (co nejefektivněji) uzel, který obsahuje klíč y.

7.25 Navrhněte efektivní postup, který pro každý uzel stromu spočte kolik potomků se nachází jeho levém a pravém podstromu. Určete jeho časovou složitost.

Kapitola 8

Červeno-černé stromy

Strom je souvislý neorientovaný graf bez cyklů, kde sousednost dvou uzlů vyjadřuje vztah rodič a potomek. Uzel může mít více potomků, ale pouze jednoho rodiče.

Samovyvažující se vyhledávací strom je vyhledávací strom, který při vkládání a odstraňování udržuje maximální hloubku stromu v $\mathcal{O}(\log(n))$.

Červeno-černý strom je samovyvažující se binární vyhledávací strom, kde každý uzel je obarven červenou, nebo černou barvou a splňuje následující pravidla:

1. kořen a všechny uzly *nil* stromu jsou černé,
2. pokud je vrchol červený, jeho otec musí být černý a
3. každá jednoduchá cesta z libovolného vrcholu x do listu obsahuje stejný počet černých vrcholů.

Rotace jsou procedury sloužící k vyvažování vyhledávacích stromů. Musí tedy zachovávat vlastnost binárního vyhledávacího stromu.

Rank prvku odpovídá pořadí prvku ve vzestupné posloupnosti. Vyjadřuje tedy, kolik čísel je menších než prvek, kterého chceme znát rank. Pokud pro výpočet ranku použijeme červeno-černý strom (který už je vybudován), umíme rank prvku určit v logaritmickém čase.

Paní Bílá připomíná: Pojem červeno-černé stromy pochází z článku „Dichromatické struktury pro vyvážené stromy“ Roberta Sedgewicka a Leonida J. Guibase, 1978. Jejich červená barva byla zvolená, protože se nejlépe tiskla na laserové tiskárně, kterou autoři vlastnili.

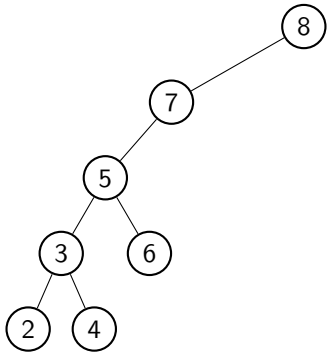
Tento příklad lze probrat intuitivně, bez rotací. Rotace jsou probrány v příkladu 8.3, který téměř vyřešíte, pokud se rozhodnete probrat detailně rotace.

Pokud chcete u příkladu 8.3 řešit hlavně přebarvování uzlů, pak můžete rotace detailně probrat už zde (například nakreslením obrázku z řešení 8.8 b)).

U rotací stojí za probrání, jak se mění ukazatele, to, že umí studenti překreslit obrázek je mnohem snazší, než napsat pseudokód.

8.1

a) Jak by šel vyvážit následující strom?



Pokud můžeme konstruovat stromy ručně, pak je nejvýhodnější jako kořen stromu volit medián všech hodnot, které se ve stromě nacházejí. To v našem případě odpovídá uzlu 5. Stejné pravidlo aplikujeme i na podstromy a získáme strom výšky 3.

b) * Mějme na vstupu nekonečnou rostoucí posloupnost čísel, které vkládáme do binárního vyhledávacího stromu. Jak byste modifikovali operaci vkládání, abyste udržovali maximální hloubku stromu v $\mathcal{O}(\log(n))$?

Nejjednodušším nápadem je po každém druhém vložení provést levou rotaci na úrovni kořene a jeho pravého syna. To nám však strom sníží pouze na polovinu (vzniknou 2 větvě), hloubka tedy stále zůstává v $\mathcal{O}(n)$. Rotace tedy musíme dělat i na nižších úrovních.

Nechť číslujeme patra stromu zdola, takže patro nejnižšího listu odpovídá 0. patru, nad ním je vrchol v 1. patře atd. až po kořen. Pak při vkládání do našeho stromu bychom vždy po každém druhém vložení provedli levou rotaci v 1. patře (v nejpravějším vrcholu – to je místo, kde tento strom roste), po každém 4. vložení levou rotaci v 2. patře a obecně po každém 2^i vložení levou rotaci v patře i. Strom, který by vznikal, by nebyl perfektně vyvážený, ale přesto by měl logaritmickou hloubku.

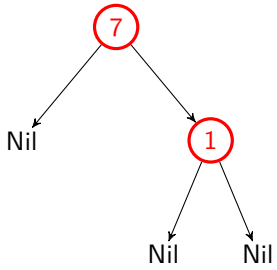
Tímto jsme vytvořili vlastní samovyvažující se binární vyhledávací strom, který však funguje jen pro vkládání stále větších a větších čísel. Mohli jsme samozřejmě použít jiný samovyvažující se strom, třeba červeno-černý strom.

Zde se hodí, aby vám studenti nadiktovali pravidla červeno-černých stromů. Pravidla si můžete vypsat bokem na tabuli tak, aby je studenti stále viděli.

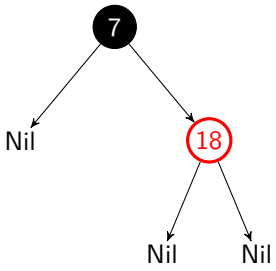
Stromy obsahují spoustu chyb, nechte si vždy vyjmenovat všechny chyby. Pozor, studenti často zapomínají na uspořádání klíčů. Stromy podle času můžete opravit.

8.2 Rozhodněte, zdali jsou následující grafy červeno-černé stromy:

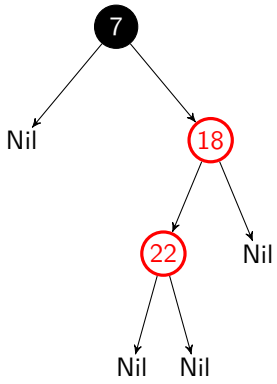
a)



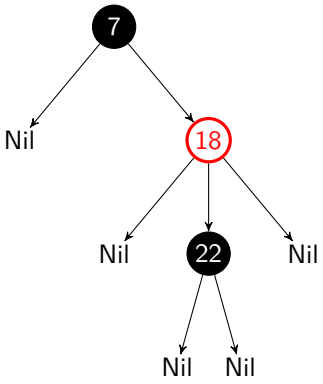
b)



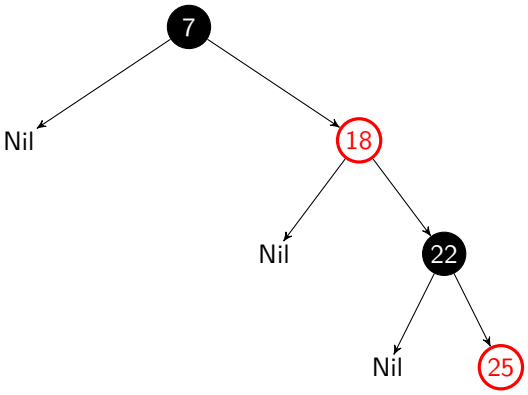
c)



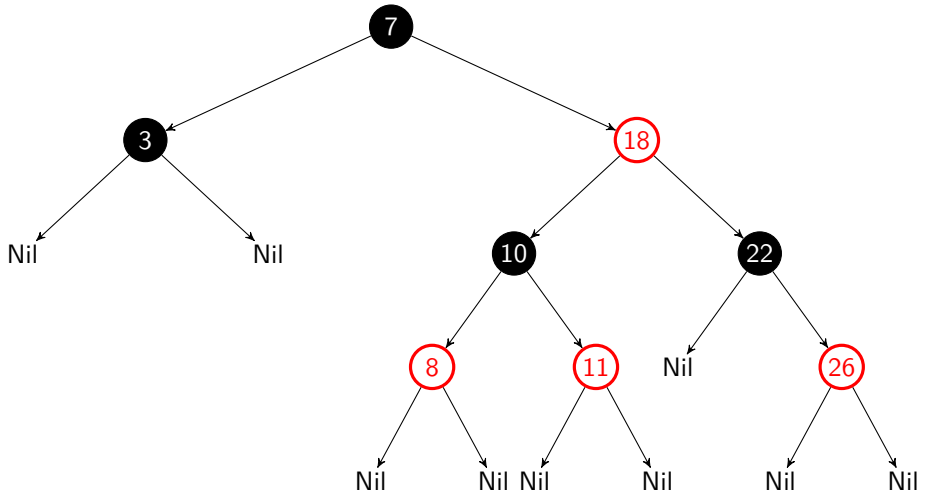
d)



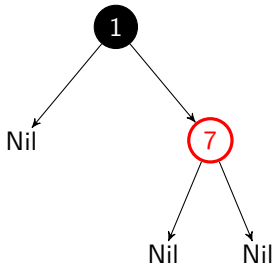
e)



f)

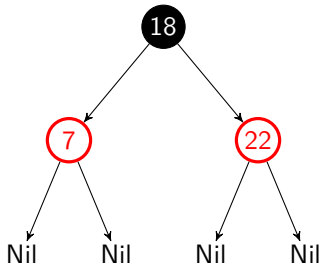


a) Tento strom porušuje pravidla 1 (kořen je červený), 2 (uzel 1 je červený a má červeného rodiče) a zároveň se kvůli špatnému pořadí nejedná ani o vyhledávací strom. Opravený strom může vypadat tedy takto:



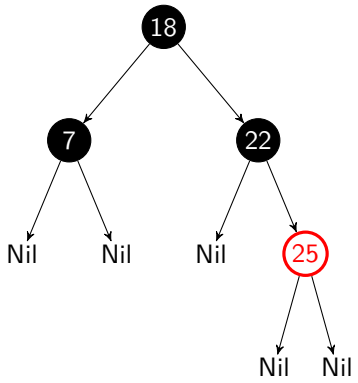
b) Jedná se o korektní červeno-černý strom.

c) Tento strom není BVS, jelikož uzel 22 je levým synem uzlu 18. Také jsou oba tyto uzly obarveny červeně, což porušuje pravidlo 2. Stačí provést obdobu levé rotace a přebarvení, formálně bychom však měli říci, že prvně musíme vyměnit klíče uzlů 18 a 22, pak provést pravou rotaci těchto dvou uzlů a nakonec provést levou rotaci, aby se 18 dostala do kořene. 18 bude mít tedy černou barvu, 7 a 22 budou mít stejnou barvu, přičemž je jedno, zdali budou červené nebo černé.



d) Toto není binární strom (je ternární). Dále je porušena černá hloubka jelikož uzlu 22 chybí synové *nil*. Opravou vznikne stejný strom jako je ten předchozí ze zadání c).

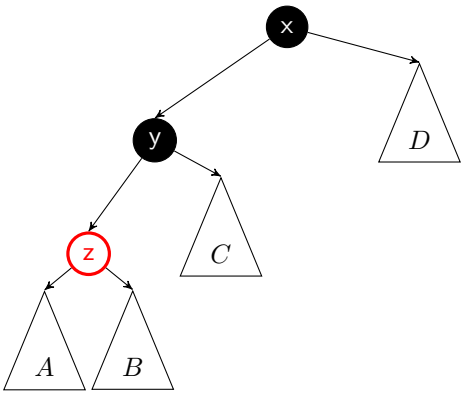
e) Malou chybou je, že uzel 25 nemá listy *nil*. Dále je porušeno pravidlo 3., černá hloubka se liší u levé a pravé větve od kořene. Je potřeba provést levou rotaci, aby se uzel 18 stal kořenem a následně jej přebarvíme na černo.



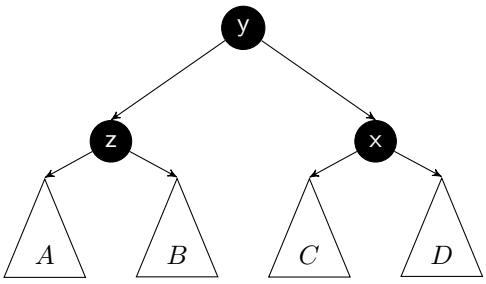
f) Jedná se o korektní červeno-černý strom.

8.3 Jak správně vyvážit stromy v následujících příkladech? Využijte rotaci doprava, doleva a správně obarvěte stromy. Předpokládejme, že všechny podstromy A, B, C, D, E, F, G jsou korektní červeno-černé stromy se stejnou černou hloubkou.

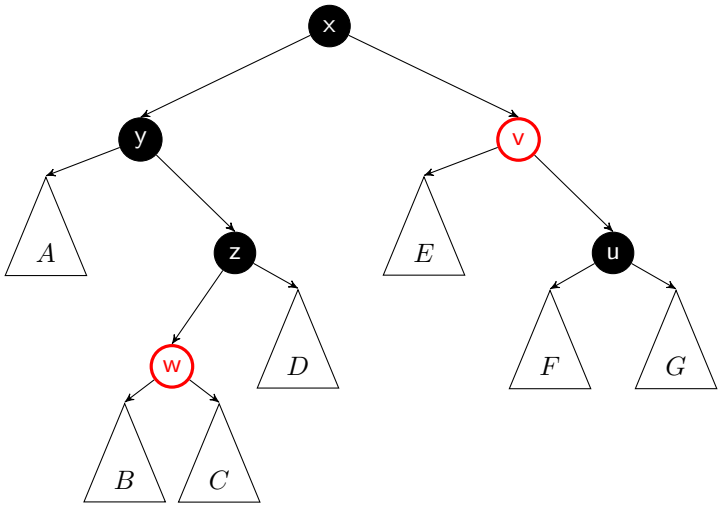
a) Nevyvážený strom:



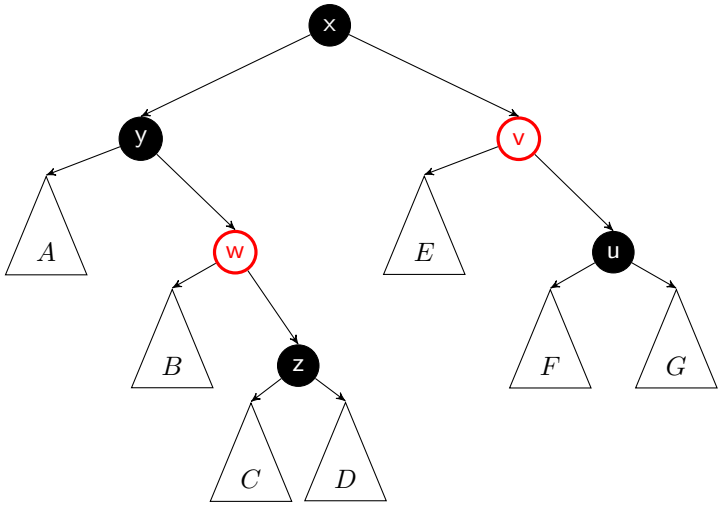
Provedeme rotaci doprava kolem uzlu x a abychom uchovali černou hloubku změníme barvu uzlu z na černou.



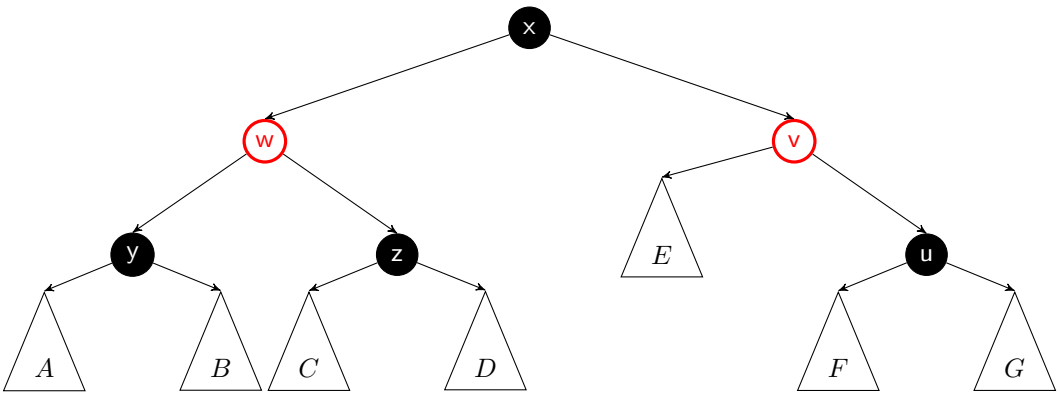
b) Nevyvážený strom:



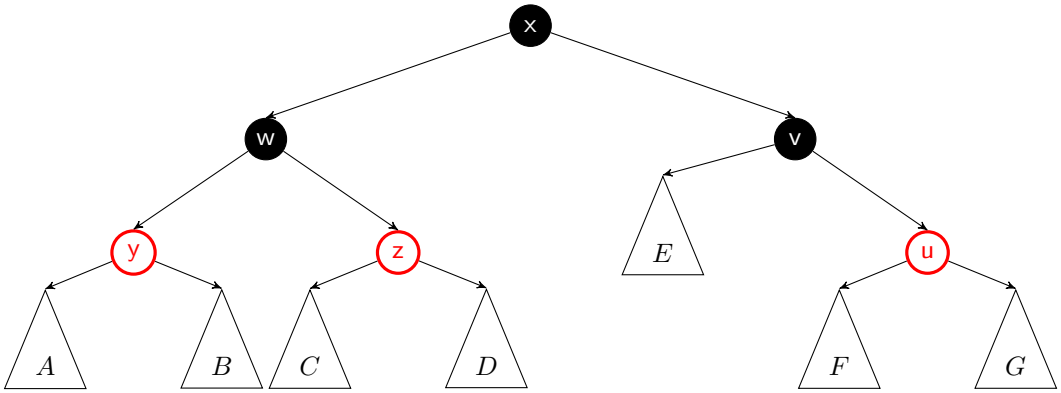
Nejprve orotujeme doprava okolo z:



Teď můžeme rotovat doleva okolo y:



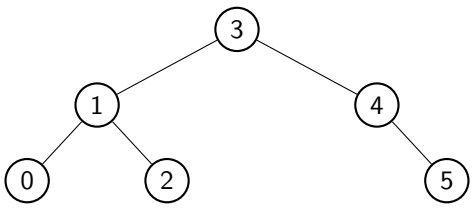
A upravíme obarvení tak, aby v každé větvi byly 2 černé uzly a 1 podstrom:



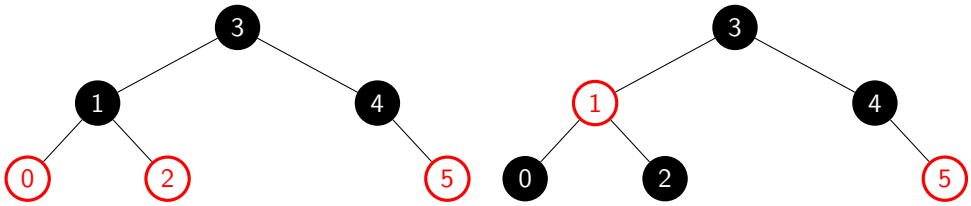
8.4

Pan Usměvavý dodává: Praktický a přehledný zápis červeno-černých stromů je čistě pomocí bílých uzlů (takzvaných bílo-bílých stromů). Dává to prostor k fantazii a rozšiřuje to paměťové schopnosti studentů.

Kolika způsoby je možné obarvit BVS z obrázku tak, aby zachovával pravidla červeno-černého stromu?



Existují 2 obarvení, viz obrázky níže.



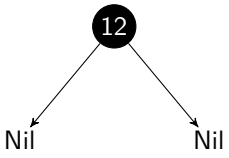
Obarvení se pokoušíme najít pomocí nejkratší větve, kterou obarvíme na černo. V tomto případě je tou větví: 3 – 4 – *nil*. Jelikož uzly 3 a 4 jsou také součástí větve 3 – 4 – 5 – *nil*, která je delší, víme, že vrchol 5 musí být červený, abychom neporušili pravidla červeno-černého stromu. Zbylé větve obarvíme podobně tak, abychom zachovali všude stejnou černou hloubku.

8.5 Zkonstruujte červeno-černý strom postupným vkládáním uzlů 12, 5, 9, 18, 2, 15, 13, 19 a 17. Poté postupně odstraňte uzly 9, 5, 15 a 12. Nakonec vyhledejte uzly 17 a 9.

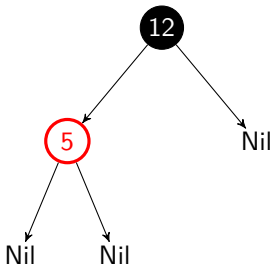
V tomhle příkladu je vhodné nejprve rozkreslit jednotlivé kolizní případy a až následně přecvičovat algoritmizaci na dané posloupnosti.

Jako bonusovou posloupnost na vkládaní můžete dát studentům: 41, 38, 31, 12, 19 a 8. Posloupnost přecvičí většinu kolizních případů a ověří jestli studenti postup pochopili.

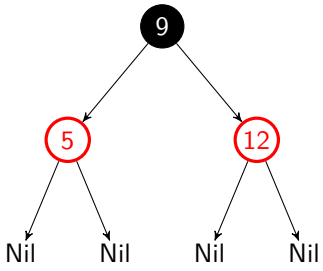
Insert(12)



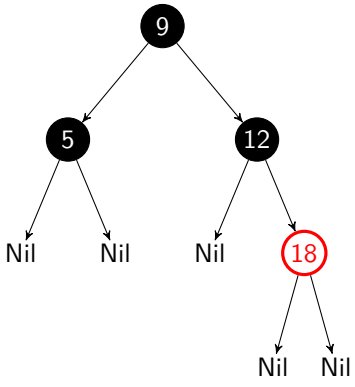
Insert(5)



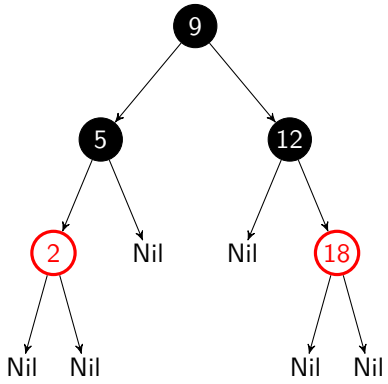
Insert(9)



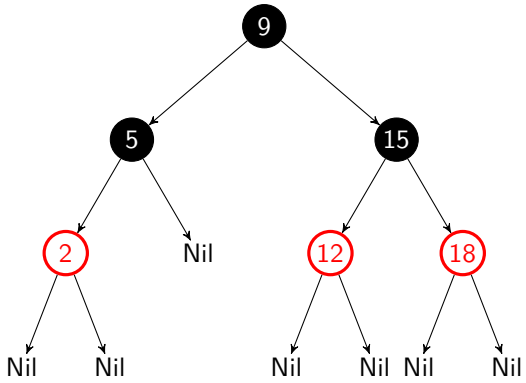
Insert(18)



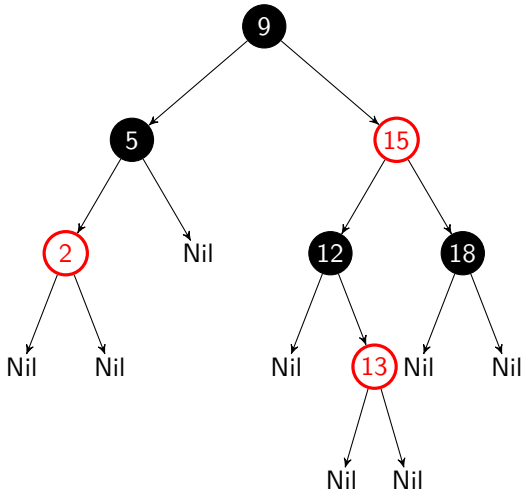
Insert(2)



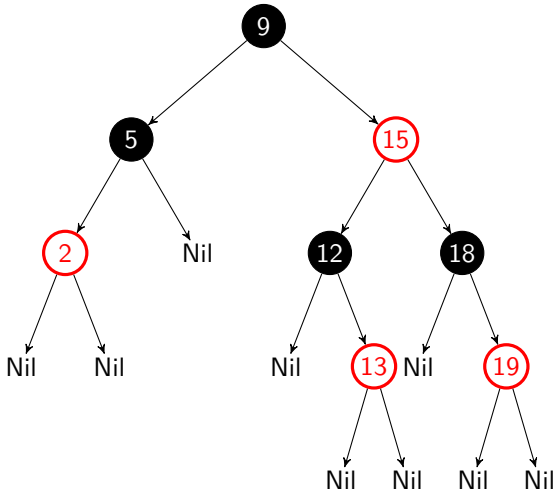
Insert(15)



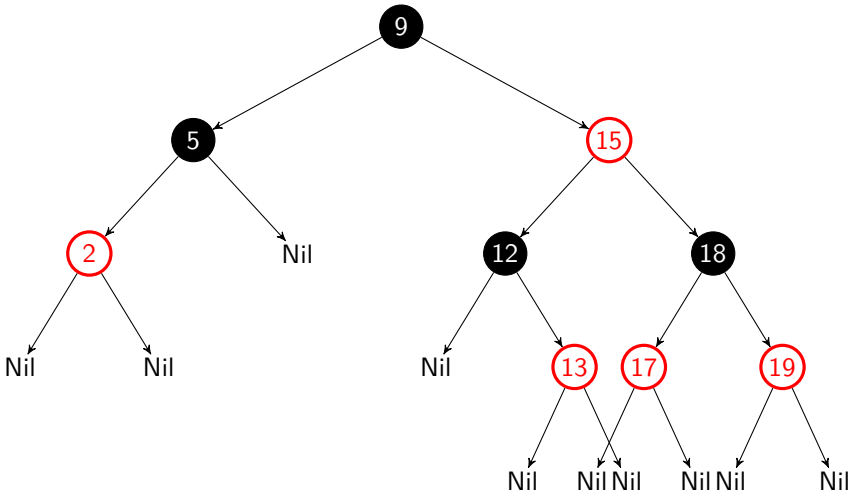
Insert(13)



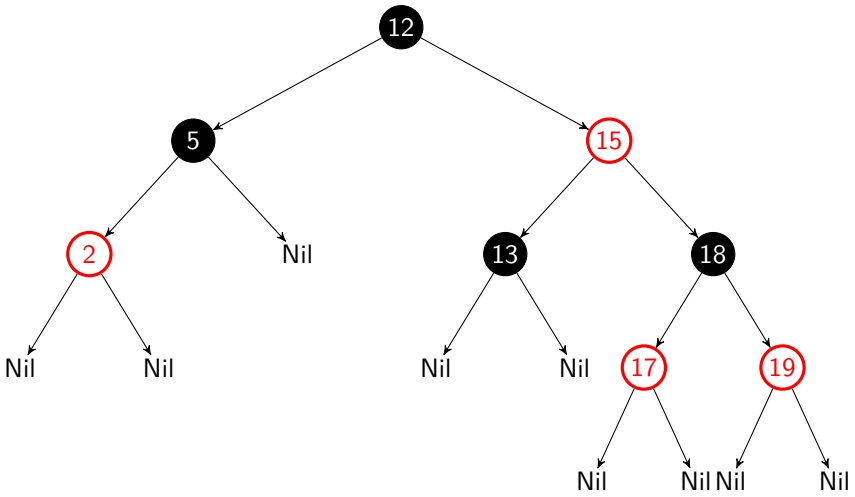
Insert(19)



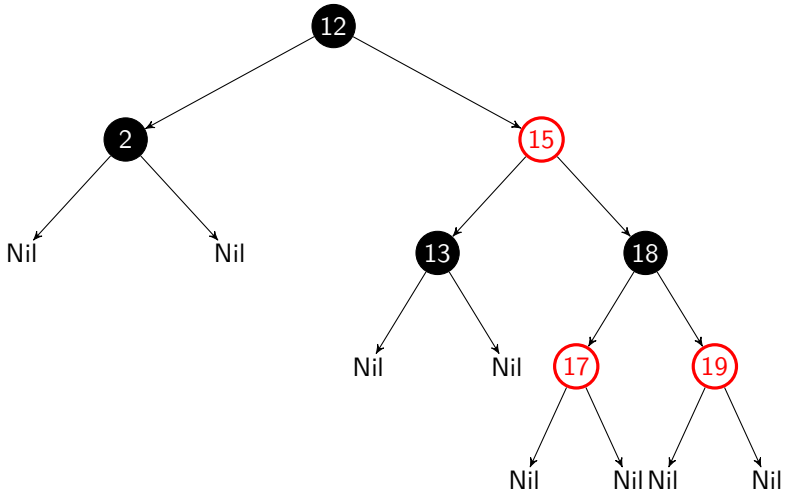
Insert(17)



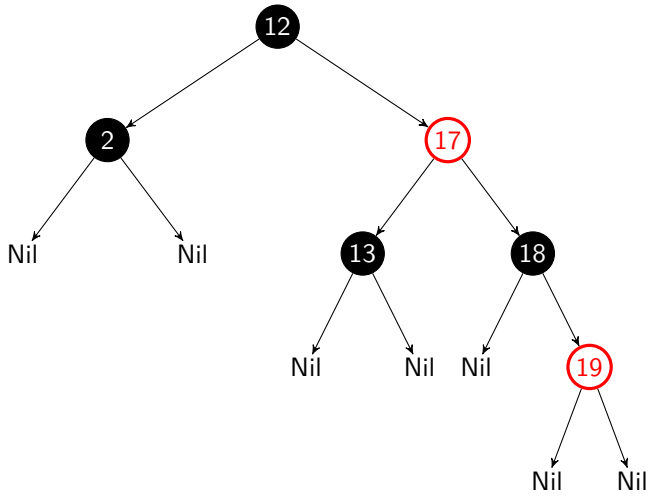
Delete(9)



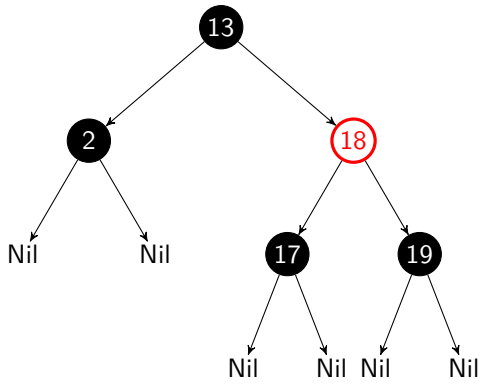
Delete(5)



Delete(15)



Delete(12)



8.6 Popište červeno-černý strom s n uzly takový, že:

a) operace Insert si vyžádá $\Omega(log(n))$ přebarvení uzlů.

Řešením je úplný binární strom se sudým počtem úrovní, kde uzly na liché úrovni jsou černé a na sudé úrovni červené. Vkládaný prvek se vloží jako červený list.

b) operace Delete si vyžádá $\Omega(log(n))$ přebarvení uzlů.

Řešením je úplný binární strom se sudým počtem úrovní, kde všechny uzly jsou černé. Odstraňuje se list.

8.7 Jaká je složitost jednotlivých operací na červeno-černých stromech?

Funkce Init má složitost opět ve třídě $\Theta(1)$. Ostatní funkce mají složitost $\mathcal{O}(\log n)$, protože hloubka červeno-černých stromů patří do třídy $\mathcal{O}(\log n)$.

8.8

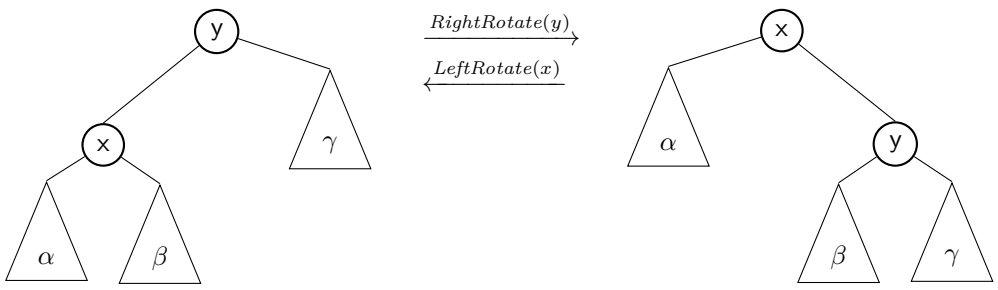
Pan Usměvavý dodává: Tužka a papír jsou nejlepší kamarádi programátora, když programuje dynamické datové struktury.

a) Které operace dokážeme implementovat stejně jako nad BVS? Které budou rozdílné, a proč?

Operace, které nemodifikují strom, můžeme implementovat stejně jako ve vyhledávacím binárním stromě. Rozdílné ale budou ty, které nějak mění strukturu stromu, tedy Delete a Insert. Implementace je rozdílná proto, že vkládáním a mazáním prvků dochází k porušování červeno-černé hierarchie, proto potřebujeme mít při operacích také opravu stromu, při které se také strom vyvažuje.

b) Navrhněte operaci LeftRotate nad červeno-černým stromem a určete její složitost.

Rotování vypadá následovně:



Algoritmus rotace vlevo:

Procedura LeftRotate($tree, x$)

vstup: strom $tree$ a jeho uzel x, který je kořenem rotace

```
1  y ← x.right
2  x.right ← y.left
3  if y.left ≠ nil then
4      y.left.parent ← x
5  fi
6  y.parent ← x.parent
7  if x.parent = nil then
8      tree.root ← y
9  else if x = x.parent.left then
10     x.parent.left ← y
11 else
12     x.parent.right ← y
13 y.left ← x
14 x.parent ← y
```

Časová složitost rotací je konstantní. Dojde maximálně ke změně 7 ukazatelů (+ nějaké v pomocných proměnných) a otestování 3 podmínek. Rotace se nijak dále ve stromě nepropagují.

c) Navrhněte operaci RightRotate červeno-černým stromem a určete její složitost.

Algoritmus rotace vpravo:

Procedura RightRotate($tree, x$)

vstup: strom $tree$ a jeho uzel x, který je kořenem rotace

```
1  y ← x.left
2  x.left ← y.right
3  if y.right ≠ nil then
4      y.right.parent ← x
5  fi
6  y.parent ← x.parent
7  if x.parent = nil then
8      tree.root ← y
9  else if x = x.parent.right then
10     x.parent.right ← y
11 else
12     x.parent.left ← y
13 y.right ← x
14 x.parent ← y
```

Časová složitost RightRotate je konstantní, podobně jak LeftRotate v podpříkladu b).

d) Navrhněte operaci Insert nad červeno-černým stromem a určete její složitost.

Prvně provedeme vložení pomocí Insert pro BVS. Následně obarvíme vložený uzel na červeno, aby se nezměnila barevná hloubka a poté provádíme kontrolu a úpravy tak, aby zůstaly zachovány základní pravidla červeno-černého stromu.

Procedura Insert($tree, node$)

vstup: strom $tree$ a vkládaný uzel $node$

```
1  y ← nil
2  x ← tree.root
3  while x ≠ nil do
4      y ← x
5      if node.key < x.key then
6          x ← x.left
7      else
8          x ← x.right
9      fi
10 od
11 node.parent ← y
12 if y = nil then
13     tree.root ← node
14 else if node.key < y.key then
15     y.left ← node
16 else
17     y.right ← node
18 node.left ← nil
19 node.right ← nil
20 node.color ← red
21 InsertFixUp(tree, node)
```

Bottom-up oprava vlastností BVS:

Procedura InsertFixUp($tree, node$)

vstup: strom $tree$ a opravovaný uzel $node$

```
1  while node ≠ tree.root ∧ node.parent.color = red do
2      if node.parent = node.parent.parent.left then
3          d ← node.parent.parent.right
4          if d.color = red then
               // případ 1
5              node.parent.color ← black
6              d.color ← black
7              node.parent.parent.color ← red
8              node ← node.parent.parent
9          else if node = node.parent.right then
               // případ 2
10             node ← node.parent
11             LeftRotate(tree, node)
12         else
               // případ 3
13             node.parent.color ← black
14             node.parent.parent.color ← red
15             RightRotate(tree, node.parent.parent)
16     else
           // podobně jako při then bloku, jenom prohodíme right za left
17 od
18 tree.root.color ← black
```

e) Navrhněte operaci Delete nad červeno-černým stromem a určete její složitost.

Procedura Delete(*tree*, *node*)

vstup: strom *tree* a smazávaný uzel *node*

```
1  y ← node
2  originalColor ← node.color
3  if node.left = nil then
4      x ← node.right
5      Transplant(tree, node, node.right)
6  else if node.right = nil then
7      x ← node.left
8      Transplant(tree, node, node.left)
9  else
10     y ← Minimum(node.right)
11     originalColor ← y.color
12     x ← y.right
13     if y.parent = node then
14         x.parent ← y
15     else
16         Transplant(tree, y, y.right)
17         y.right ← node.right
18         y.right.parent ← y
19     Transplant(tree, node, y)
20     y.left ← node.left
21     y.left.parent ← y
22     y.color ← node.color
23 if originalColor = black ∧ x ≠ nil then
24     DelteFixUp(tree, x)
25 fi
```

Procedura DeleteFixUp(*tree*, *node*)

vstup: strom *tree* a opravovaný uzel *node*

```
1  while node ≠ tree.root ∧ node.color = black do
2      if node = node.parent.left then
3          sibling ← node.parent.right
4          if sibling.color = red then
5              sibling.color ← black
6              node.parent.color ← red
7              LeftRotate(tree, node.parent)
8              sibling ← node.parent.right
9          if sibling.left.color = black ∧ sibling.right.color = black then
10             sibling.color ← red
11             node ← node.parent
12         else if sibling.right.color = black then
13             sibling.left.color ← black
14             sibling.color ← red
15             RightRotate(tree, sibling)
16             sibling ← node.parent.right
17         else
18             sibling.color ← node.parent.color
19             node.parent.color ← black
20             sibling.right.color ← black
21             LeftRotate(tree, node.parent)
22             node ← tree.root
23     else
           // podobně jako při then bloku, jenom prohodíme right za left
24     fi
25 od
26 node.color = black
```

8.9 Naprogramujte reprezentaci červeno-černého stromu a elementární operace nad ním. Ve studijních materiálech jsou k tomuto připravené zdrojové kódy: C a Python.

Díky nejhůře logaritmické složitosti vkládaní, odstraňování a vyhledávaní jsou červeno-černé stromy často využívány v real-time a jiných časově náročných aplikacích. Jsou využívány jako datové struktury pro výpočetní geometrii, také jsou integrovány do současných jader Linuxu, kde slouží jako datové struktury pro spravedlivý scheduler. Ve standardních knihovnách jazyků jako C++, Java a C# jsou díky jejich vlastnostem využívány pro implementaci multisetu a asociativních polí.

Všeobecně se také používají pro implementaci stálých datových struktur (stále datové struktury jsou takové, které si pamatují svůj předchozí stav když jsou modifikovány). Tedy mimo dobré časové složitosti poskytují při stálých strukturách i dobrou paměťovou složitost, která je v $\mathcal{O}(\log(n))$.

Následující příklady jsou vhodné na domácí studium.

8.10 Je možné transformovat libovolný BVS na libovolný jiný BVS se stejnou množinou klíčů jenom pomocí posloupnosti rotací doleva a doprava?

Přibližný postup je následující: orotujme nejmenší uzel prvního BVS až ke kořenu a postupně rotujme následující uzly tak, abychom dostali strom s hloubkou n, kde každý levý potomek je Nil. Udělejte totéž s druhým BVS. Z toho vidíme, že existuje posloupnost rotací která dokáže převést jeden BVS na druhý.

Není znám polynomiální algoritmus, který by určil minimální počet rotací na přeměnu jednoho BVS na druhý (i když „vzdálenost“ rotací je nanejvýš $2n - 6$ pro BVS s alespoň 11 uzly).

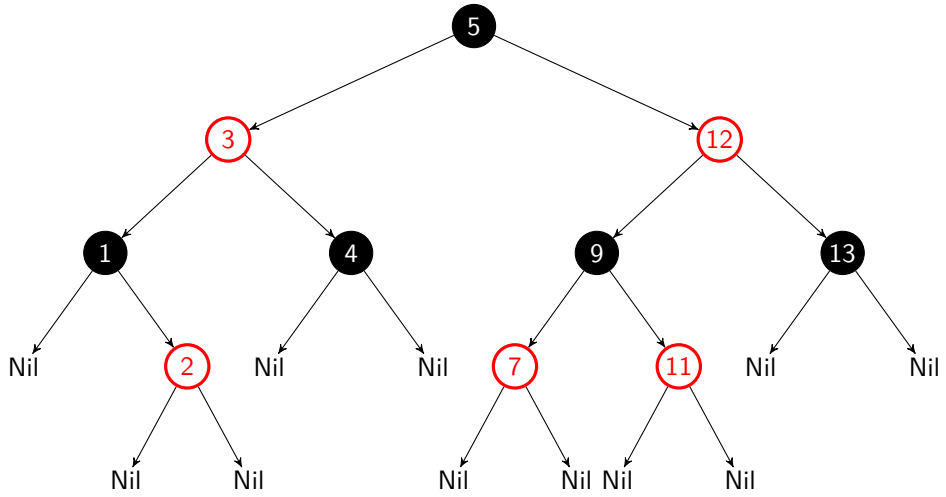
8.11

a) S jakou časovou složitostí dokážeme z libovolné posloupnosti n prvků vybudovat červeno-černý vyhledávací strom?

Z libovolné posloupnosti vytvoříme červeno-černý strom postupným vkládáním, tudíž jedno vložení nám zabere $\mathcal{O}(\log n)$ a těchto vkládaní bude n. Složitost vytvoření stromu je tedy $\mathcal{O}(n \log(n))$.

b) Mějme uspořádanou posloupnost klíčů. Popište, jak byste konstruovali červeno-černý strom tak, aby celková časová složitost vybudování stromu byla lineární.

8.12 Určete posloupnost operací Insert tak, že výsledný červeno-černý strom bude vypadat následovně:



8.13 Vkládaní seřazené posloupnosti:

- Nakreslete červeno-černý strom vzniklý vložením posloupnosti čísel $[1, \ldots, 11]$ do prázdného stromu.
- Jak bude strom vypadat, když posloupnost vložíme do stromu v opačném pořadí?
- Popište obecně, jak se bude tvořit červeno-černý strom z libovolné stoupající nebo klesající posloupnosti klíčů.

8.14 Navrhněte metodu, která ověří, že strom splňuje následující pravidla modro-zeleného stromu:

- Kořen stromu je vždy zelený.
- Každý uzel se sudým klíčem je modrý.
- V případě, že je rodič uzlu modrý, pak je uzel zelený.

8.15 Navrhněte metodu, která ověří, že strom splňuje následující pravidla červeno-modro-zeleného stromu:

- Pokud je rodič uzlu červený, pak je uzel modrý.
- Pokud je rodič uzlu modrý, pak je uzel zelený.
- Pokud je rodič uzlu zelený, pak je uzel červený.
- Pokud je počet listů sudý pak je kořen stromu červený, jinak je modrý.

8.16 Upravme u červeno-černých stromů pravidlo „pokud je vrchol červený, jeho otec musí být černý“ různými způsoby. Takto upravený strom pojmenujme „relaxovaný červeno-černý strom“. Jaké z následujících tvrzení jsou potom pravdivá?

1. Každý červeno-černý strom je i „relaxovaný červeno-černý strom“.
2. Existuje „relaxovaný červeno-černý strom“, který není korektní červeno-černý strom.
3. Výška každého „relaxovaného červeno-černého stromu“ s n uzly je v $\mathcal{O}(\log(n))$.
4. Každý BVS může být přetvořen na „relaxovaný červeno-černý strom“ (pomocí nějakého obarvení).

a) Zadané pravidlo změníme tak, že zakážeme pouze trojice následujících uzlů červené barvy (dvojice povolíme).

Nepravdivé je tvrzení d).

b) Zadané pravidlo zrušíme úplně.

Nepravdivé je tvrzení c).

Kapitola 9

B-stromy

B-strom je vyhledávací strom s následujícími vlastnostmi:

1. každý uzel obsahuje klíče v uspořádaném neklesajícím pořadí,
2. každý vnitřní uzel obsahuje ukazatele na všechny své syny,
3. klíče v uzlech oddělují intervaly možných hodnot v podstromech mezi nimi a
4. všechny listy mají stejnou hloubku.

Vnitřní uzel B-stromu s n klíči má $n + 1$ následníků.

Stupeň B-stromu $t \geq 2$ určuje následující vlastnosti B-stromu:

1. každý uzel kromě kořene obsahuje alespoň $t - 1$ klíčů a
2. každý uzel může obsahovat nanejvýš $2t - 1$ klíčů a $2t$ následníků.

Plný uzel je takový, který má přesně $2t$ následníků.

Arita stromu určuje kolik následníků daný uzel může maximálně mít. U B-stromu podle definice výše je arita $2t$.

9.1

Opakovaní přednášky, vše by již měli znát. Je na vás, zdali tento příklad chcete na cvičení řešit – pokud vám studenti diskutují u těchto diskuzních příkladů, tak jej klidně proberte, pokud nefungovaly už starší diskuzní příklady, tak jej přeskočte s komentářem, že se nad tím mohou zamyslet doma a odpovědi ve sbírce mají.

a) Jaké důsledky má zvýšení arity vyhledávacího stromu?

> Hloubka stromu se zmenší o konstantní faktor (z binárního na n-ární je to $\frac{\log(2)}{\log(n)} = \log_n 2$). Alternativně se dá říct, že do stromu stejné hloubky můžeme uložit více klíčů (překvapivě inverzní poměr oproti minulému).

b) Jaký bude poměr rychlosti vyhledávání v binárním a n-árním vyhledávacím stromu?

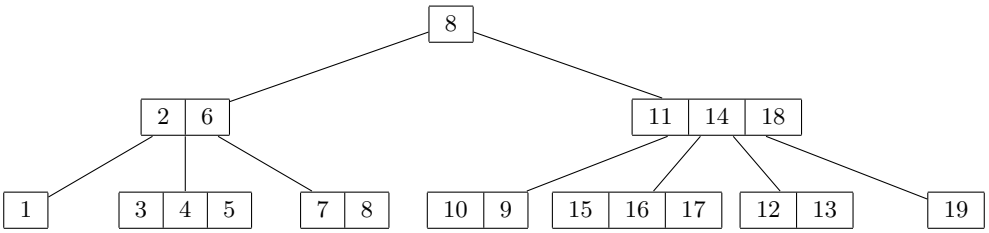
Hloubka stromu se sníží poměrem $\log_n 2$, ale při vyhledávání budeme muset projít všechny klíče v uzlu, což nám dává u binárního stromu jedno porovnání, u n-árního až $n-1$ porovnání. Výsledkem je tedy zpomalení v poměru $\log_n 2 \cdot n$.

c) Kterých operací musíme provádět méně díky zvýšení arity a jak nám to pomůže?

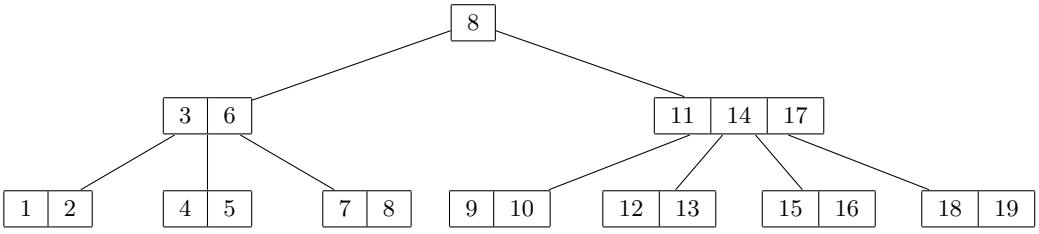
Snížíme počet čtení a zápisů celých uzlů. Pokud bychom vždy museli načíst znovu každý klíč v uzlu, pak by nám zvýšení arity moc nepomohlo, ale vhodnou implementaci nám zajistí menší počet I/O operací, což se nám hodí například při přístupu na disk.

9.2

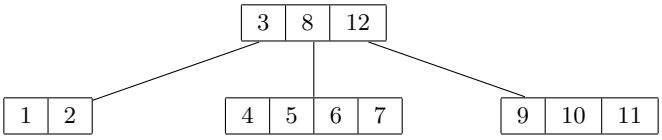
a) Je následující graf korektní B-strom stupně 3? Pokud ne, opravte jej.



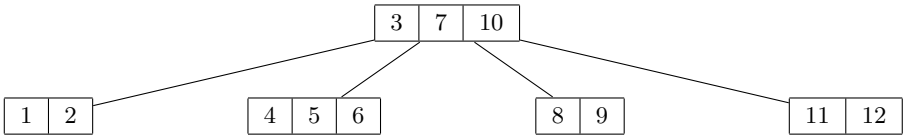
Není korektní. Podle definovaného B-stromu na přednášce musí mít každý uzel kromě kořene stupeň alespoň $t-1$ (uzly 1 a 19 jsou stupně jedna tedy je opravíme posunem prvků z vedlejších uzlů, ale tato úprava byla provedena až po následujících úpravách). Podle pravidel B-stromu jsou hodnoty v uzlech uspořádány. Je tedy nutné uspořádat hodnoty uzlu $10, 9$. Dále uzly s hodnotami $15, 16, 17$ a $12, 13$ jsou špatně zařazeny. Uzly je třeba prohodit tak, aby $12, 13$ byly mezi hodnotami 11 a 14 u rodiče a $15, 16, 17$ mezi hodnotami 14 a 18 u rodiče.



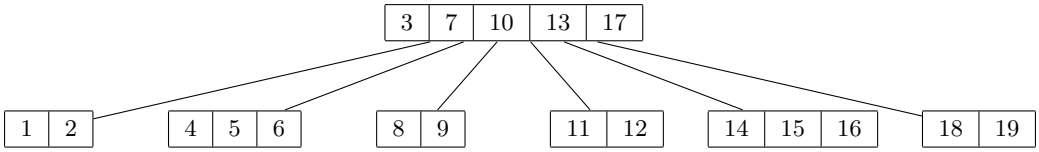
b) Je následující graf korektní B-strom stupně 3? Pokud ne, opravte jej.



Není korektní, protože kořen obsahuje 3 klíče, ale má jen 3 potomky (měl by mít 4). Je tedy potřeba vytvořit větev napravo od 12, k tomu potřebujeme přesunout prvky. Opravit strom lze třeba takto:

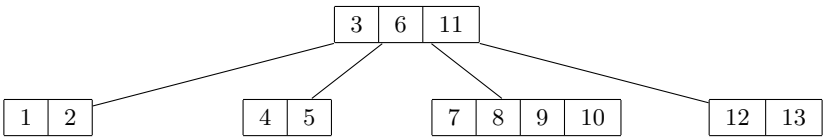


c) Je následující graf korektní B-strom stupně 3? Pokud ne, opravte jej.

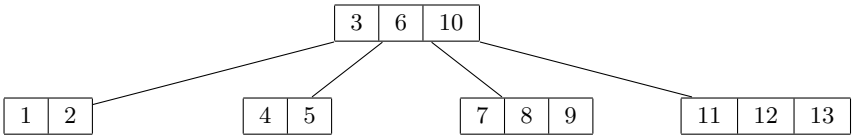


Je to korektní B-strom.

d) Je následující graf korektní B-strom stupně 2? Pokud ne, opravte jej.



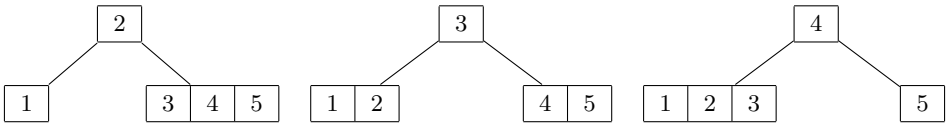
Není, B-strom stupně 2 nemůže obsahovat uzly s víc než 3 klíči. Proto uzel rozdělíme do sousedních uzlů.



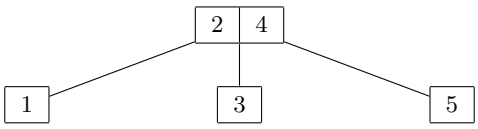
9.3 Ukažte všechny možnosti B-stromů se stupněm 2, které obsahují posloupnost 1, 2, 3, 4, 5.

Můžeme si jednotlivé případy rozdělit podle toho, kolik prvků bude v kořeni.

1. Kořen s jedním klíčem:



2. Kořen se dvěma klíči:

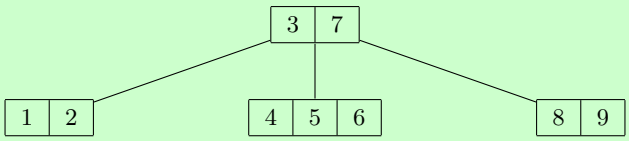


3. V kořeni nemůžeme mít tři klíče, jelikož bychom už neměli dost klíčů pro 4 potomky.

9.4 Určete z různých průchodů, jaký stupeň mají vypsané B-stromy a jak vypadají? Prodiskutujte, zdali existuje více možností.

Proberte tolik příkladů z tohoto úkolu, na kolik máte čas. Určete si například, že příkladu nebudete věnovat více než 5 minut a podle toho se řiďte.

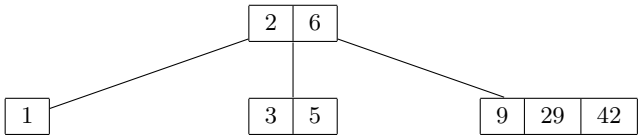
Průchody B-stromem definujeme následovně:



- Preorder: 3, 7, 1, 2, 4, 5, 6, 8, 9 (vypisuje se celý uzel)
- Inorder: 1, 2, 3, 4, 5, 6, 7, 8, 9
- Postorder: 1, 2, 4, 5, 6, 8, 9, 3, 7 (vypisuje se celý uzel)

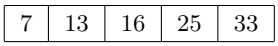
a) Preorder: 2, 6, 1, 3, 5, 9, 29, 42

Vypsaný B-strom musí být stupně $t = 2$ a vypadá následovně:



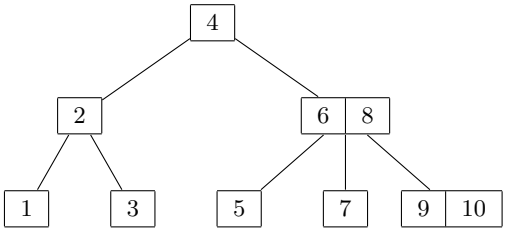
b) Preorder: 7, 13, 16, 25, 33

Vypsaný B-strom musí být stupně alespoň $t = 3$ a vypadá následovně:



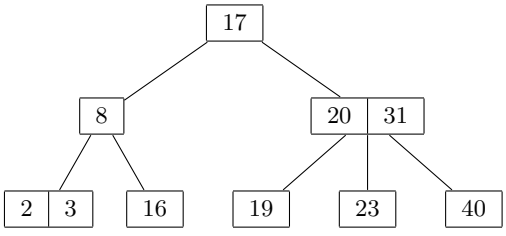
c) Preorder: 4, 2, 1, 3, 6, 8, 5, 7, 9, 10

Vypsaný B-strom musí mít stupeň právě $t = 2$ a vypadá následovně:



d) Postorder: 2, 3, 16, 8, 19, 23, 40, 20, 31, 17

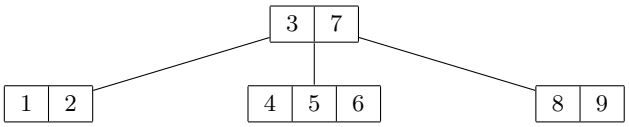
Vypsaný B-strom musí mít stupeň právě $t = 2$ a vypadá následovně:



e) Inorder: 1, 2, 3, 4, 5, 6, 7, 8, 9

Z inorder průchodu máme několik možností, jak může zadaný strom vypadat. Jedno z řešení je například B-strom se stupněm alespoň $t = 5$:

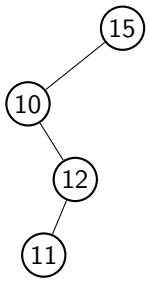
Nebo pro $t = 2$ například:



9.5 Určete z výpisu, o který typ n-arního vyhledávacího stromu se jedná.

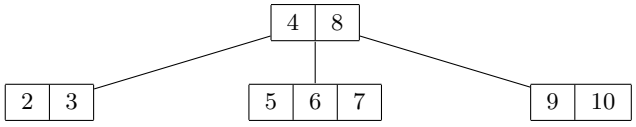
a) Preorder: 15, 10, 12, 11.

Strom musí mít hloubku 4. První klíč je větší než druhý, takže nejde o uzel s více klíči. Dvojice 10 a 12 by mohla tvořit uzel B-stromu, ale jelikož neexistuje žádný klíč větší než klíč v kořeni, nemůže být strom B-stromem, protože by kořen neměl druhého potomka. Stejné tvrzení, jen s černou hloubkou lze aplikovat na červeno-černý strom, jelikož vidíme, že tento strom nemůže být vyvážený. Jedná se tedy o obecný binární vyhledávací strom:



b) Preorder: 4, 8, 2, 3, 5, 6, 7, 9, 10.

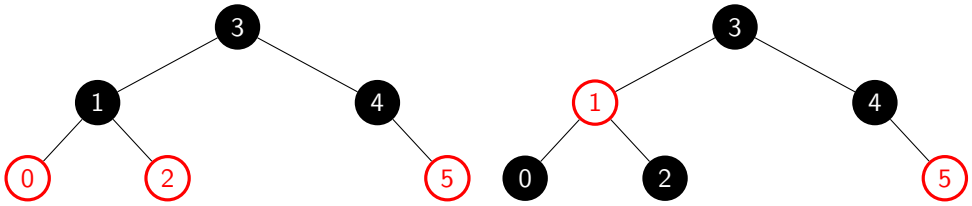
Trojice 4, 8 a 2 určuje, že nemůže jít o binární vyhledávací strom. Klíč 8 by musel být pravým potomkem klíče 3, ale následně nemáme kam zavěsit klíč 2, protože je menší než 4, ale už se musí nacházet v pravé větvi od 4. Jedná se tedy o následující B-strom, jehož stupěň je 3 nebo 4.



c) Postorder: 0, 2, 1, 5, 4, 3.

Zde budujeme strom zdola nahoru. Prvně musíme ověřit, zdali se může jednat o B-strom. Ten by musel mít v poslední n-tici čísel rostoucí posloupnost klíčů, jediná možnost je tedy kořen s jedním klíčem – 3. Aby měl strom všude stejnou hloubku, museli bychom zbytek posloupnosti rozdělit na 2 listy, což nelze, protože se v nich nenachází 2 rostoucí posloupnosti. Jedná se tedy o binární strom s klíčem 3 v kořeni. Pak lze zbylé klíče

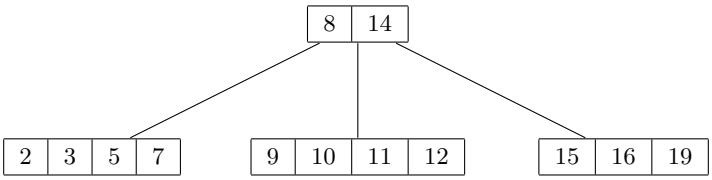
seřadit jen dvěma způsoby, z nichž jeden by porušoval pravidlo vyhledávacího stromu. Vzhledem k tomu, že výsledný strom je vyvážený, je možné jej obarvit tak, aby se jednalo o červeno-černý strom:



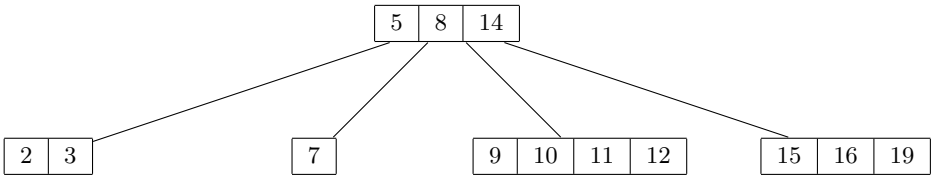
d) Lze rozeznat typ stromu z inorder výpisu?

Ne, všechny vyhledávací stromy se stejnou množinou klíčů mají inorder výpis stejný (vzestupnou množinu klíčů).

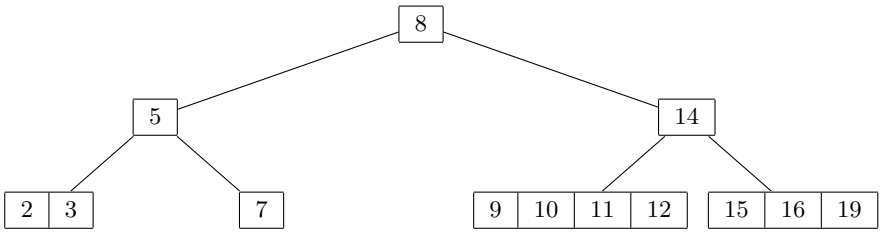
9.6 Přetvořte pomocí štěpení následující B-strom stupně 4 na B-strom stupně 2.



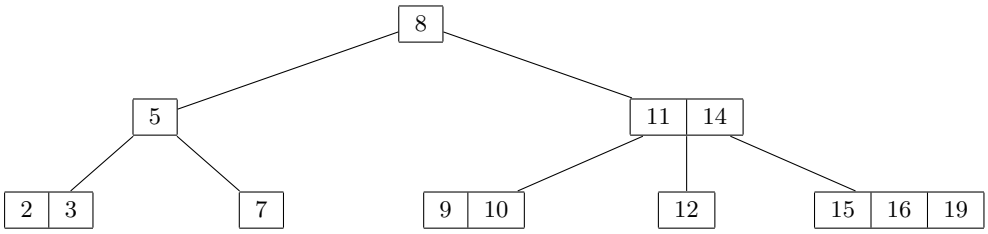
Pro řešení můžeme zvolit několik postupů. Pokud začneme štěpit uzly od kořene po patrech až do listů, garantuje nám algoritmus, že projdeme všechny uzly. Algoritmus si můžeme představit, jak kdyby jsme vkládali prvky do každého z listů a při průchodu plným uzlem ho rozštěpili. Výsledek v tomto případě vypadá následovně. Nejdříve rozštěpíme levý uzel:



Kořen je plný, musíme ho tudíž štěpit:



Následně můžeme rozštěpit poslední nekorektní uzel:



9.7

Začněte kreslit uprostřed tabule (nejen horizontálně, ale i vertikálně).

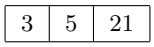
Ze začátku povolte studentům intuitivní postup. Později dbejte na dodržování algoritmu (k tomu se hodí kreslit si celé 3 políčka stromu, ačkoliv nejsou zaplněná). Procházejte klíče zprava, jak bylo ukázáno na přednášce. Preemptivně štěpte stejně jako na přednášce.

Pokud studenti nepochopí vkládání na příkladu, tak jej rozhodně nenaprogramují, takže klidně přidejte další čísla. Zajímavé případy klidně opakujte (smažte klíč a vložte obdobně jiné klíče).

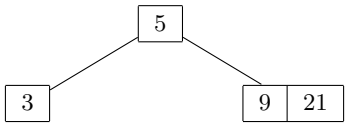
a) Vložte do B-stromu se stupněm 2 následující klíče: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12 a 4. Postupujte podle optimálního algoritmu, který prochází stromem jenom dolů.

V řešení jsou vysázeny pouze konfigurace, kdy bylo třeba nějaký uzel rozdělit a také výsledná konfigurace.

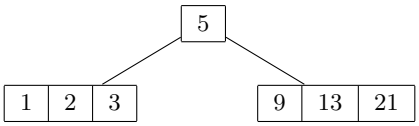
- Vytvoření stromu a vložení prvků 5, 3 a 21:



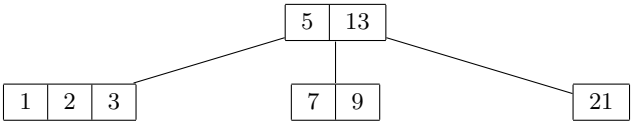
- Vložení 9:



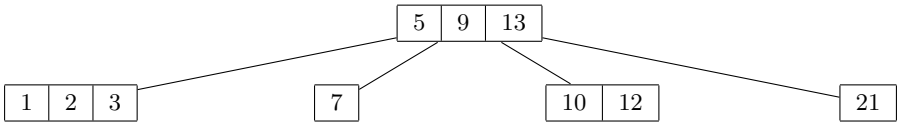
- Vložení 1, 13 a 2:



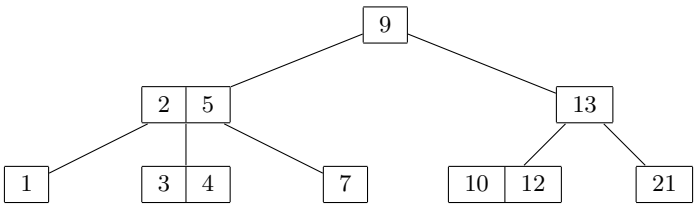
- Vložení 7:



- Vložení 10 a 12:

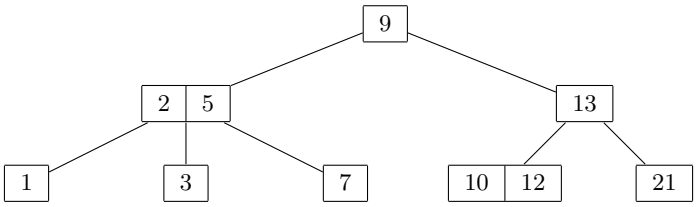


- Vložení 4:

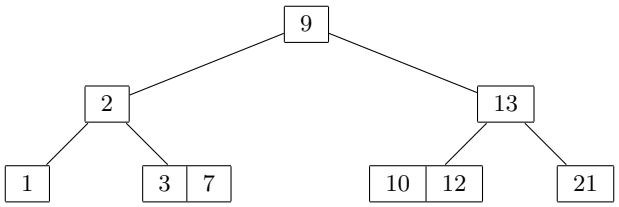


b) Z výsledného B-stromu smažte následující prvky: 4, 5, 1

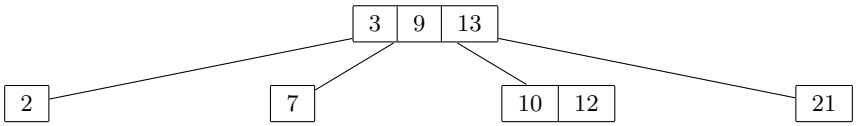
1. Smazání 4:



2. Smazání 5:

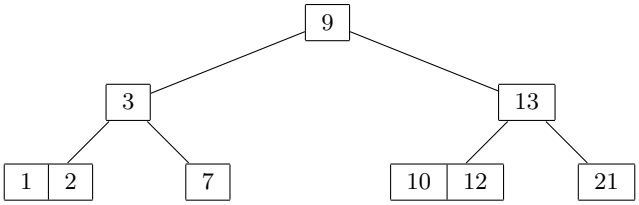


3. Smazání 1:

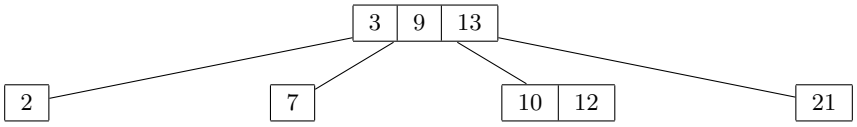


c) Opětovně vložte a smažte ze stromu hodnotu 1.

1. Vložení 1:



2. Smazání 1:

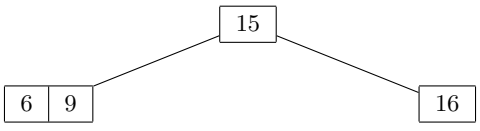


Pro testování operací na B-stromě můžete využít applet B-Tree, kde pro naši implementaci B-stromu stupně 2 zvolte Max Degree = 4 a preemptivní rozdělování.

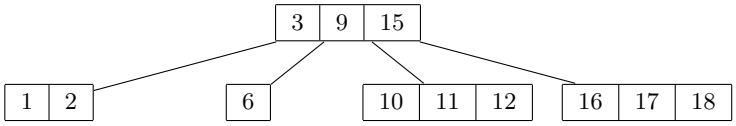
9.8 Navrhněte B-strom s $t = 2$ a hloubkou 4, který provede maximální počet štěpení uzlů při vkládání.

Strom bude mít plné uzly ve větvi, do které se vkládá nový prvek.

9.9 Kolik hodnot můžeme vložit do následujícího B-stromu bez toho, aby se štěpil kořen stromu?



Aby se uzly v stromu nerozštěpili můžeme vkládat hodnoty jenom dokud se kořen nezaplní.

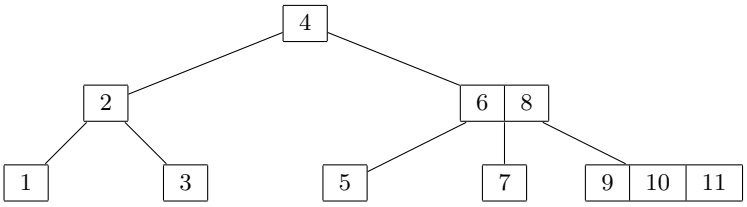


Maximálně jsme mohli doplnit 16 prvků.

9.10 Jak bude vypadat B-strom, do kterého je vkládána seřazená posloupnost čísel $[1, \ldots, n]$.

Uzly se budou plnit a štěpit pouze v pravé části stromu. Uzly v levé části stromu budou mít minimální aritu a budou se postupně „posouvat dolů“ (budou nad nimi vznikat stále výše nové kořeny).

Pro B-strom se stupněm 2 a posloupnost čísel $[1, \ldots, 11]$, vypadá strom následovně:



9.11 Mějme B-strom stupně $t = 32$ a výšky 4. Jaký bude nejmenší a největší počet uzlů (a klíčů) v tomto stromě?

Tento typ příkladu mohou studenti znát ze zkoušek jiných předmětů (organizace souborů a databáze). V těchto předmětech se probírá jiná implementace, ale v této situaci je odlišná jen použitím t pro určení stupně vrcholu (v jiných předmětech se o implementaci nemluví a používá se obecně pouze arita stromu).

Minimální B-strom bude takový, jehož každý uzel bude obsahovat minimální počet potomků. Tedy můžeme uvažovat v kořeni 2 potomky a ve zbylých uzlech 32 potomků. Tedy sčítáním postupně po vrstvách spočítáme celkový počet uzlů, tedy první vrstva má 1 uzel a ve druhé vrstvě máme 2 uzly. V další vrstvě je opět počet uzlů možné spočítat násobením počtu uzlů v předchozím patře větvícím faktorem (tedy stupeň stromu $t = 32$). Minimální počet uzlů je tedy $1 + 2 + 2 \cdot 32 + 2 \cdot 32 \cdot 32 = 2115$ uzlů. Zachyceno tabulkou pro počet uzlů a klíčů (celkový počet je suma sloupce):

Patro	Počet uzlů	Počet klíčů
1	1	1
2	2	$2 \cdot 31$
3	$2 \cdot 32$	$2 \cdot 32 \cdot 31$
4	$2 \cdot 32 \cdot 32$	$2 \cdot 32 \cdot 32 \cdot 31$

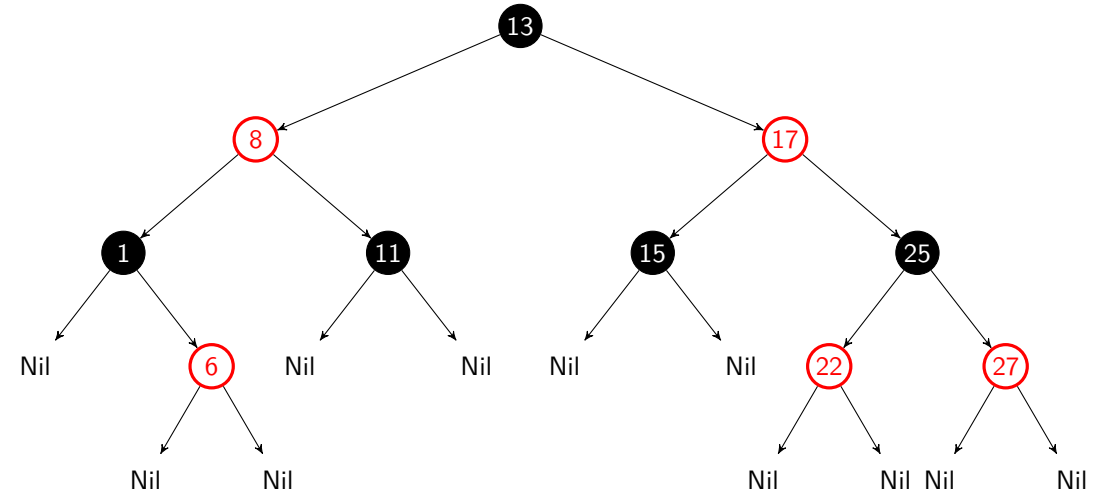
Podobnou úvahou můžeme spočítat maximální počet uzlů s tím, že budeme násobit maximálním větvícím faktorem, což je $2t$, tedy 64, což je $1 + 64 + 64 \cdot 64 + 64 \cdot 64 \cdot 64 = 266305$ uzlů. Zachyceno tabulkou pro počet uzlů a klíčů (celkový počet je suma sloupce):

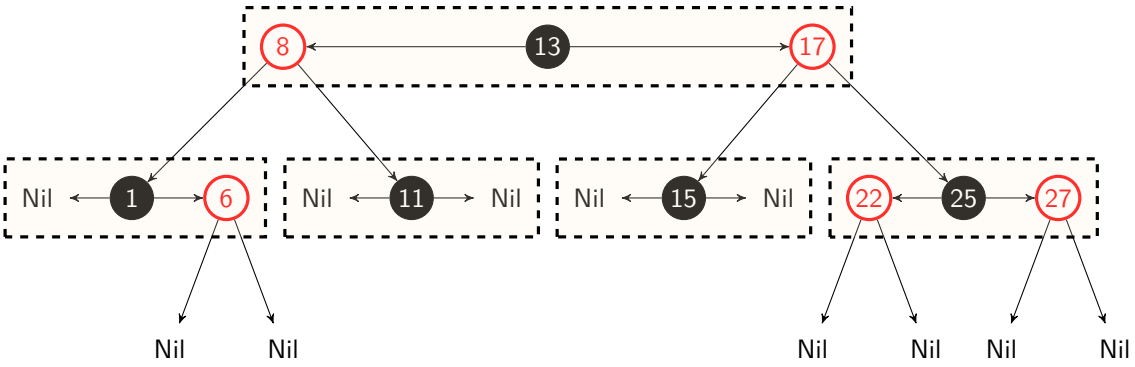
Patro	Počet uzlů	Počet klíčů
1	1	63
2	64	$64 \cdot 63$
3	$64 \cdot 64$	$64 \cdot 64 \cdot 63$
4	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 64 \cdot 63$

9.12

Tento příklad děláme pro nadanější studenty. Chceme jim ukázat, proč červeno-černé stromy mají vlastnosti, které jsme studentům dříve nadiktovali. Slabší studenti se budou možná ztrácet, proto je netrapte, pokud je téma nebude zajímat.

a) Červeno-černý strom lze převést na takzvaný 2,3,4-strom, což je 4-ární B-strom. Příkladem převodu jsou následující červeno-černý strom a jeho ekvivalent v podobě B-stromu. Výsledný strom má vždy černý uzel jako středový klíč, postranní klíče odpovídají červeným uzlům.





Zopakujte si pravidla červeno-černých stromů a podívejte se na ně v kontextu 2,3,4-stromu.

> Na 2,3,4-stromu je jasnější význam všech tří základních pravidel.
>
> Pravidlo „kořen je vždy černý“ odpovídá tomu, že prostřední klíč kořene je vždy černý.
>
> Do červeného uzlu (okrajový klíč) se dostaneme jedině z černého uzlu (středový klíč), proto musí mít každý červený uzel černého otce.
>
> Stejná černá hloubka všech větví znamená stejnou celkovou hloubku vzniklého B-stromu (všechny listy B-stromu jsou ve stejném patře).

b) Jaká je maximální hloubka červeno-černého stromu vzhledem k počtu klíčů (přesné číslo, ne jen v rámci $\mathcal{O}$ notace) a jak se to dá ukázat na 2,3,4-stromu?

Paní Bílá připomíná: Odpověď na tuto otázku znáte z 6. přednášky z důkazu věty 6.

Maximální hloubka červeno-černého stromu je $2 \cdot \log_2(n+1)$. U našeho 2,3,4-stromu s aritou 4 musíme logaritmus převést na základ 4, tedy $2 \cdot \log_4(n+1)$.

c) * Zamyslete se, čemu odpovídají rotace červeno-černých stromů v kontextu 2,3,4-stromu.

Rotace jsou důsledkem štěpení, spojování a změn pozic v uzlech 2,3,4-stromu.

9.13 Lze urychlit operace nad B-stromy pomocí binárního vyhledávání místo sekvenčního průchodu? Kde lze řešení použít, kde ne a proč se používá/nepoužívá?

Binární vyhledávání zrychlí vyhledávání klíče v uzlu, ale nelze použít v případě, že chceme do vkládat do listu, či z něj potřebujeme odstraňovat.

Zrychlení není vzhledem k velikosti stromu, ale vzhledem k aritě stromu.

9.14

a) Jak byste postupovali při hledání konkrétního prvku?

Při prohledávaní B-stromu na rozdíl od binárních stromů musíme procházet také všemi klíči v uzlu abychom našli správnou větev, kam se zanořit. Pseudokód může vypadat následovně:

Procedura SEARCH($node, k$)

vstup: uzel *node*, který má n potomků, pod kterým hledáme hodnotu k
výstup: nalezený uzel a index hledaného prvku, nebo *nil*, pokud neexistuje

```
1  i ← 1
2  while i ≤ node.n ∧ k > node.key_i do
       // hledání správné větve
3      i ← i + 1
4  od
5  if i ≤ node.n ∧ k = node.key_i then
       // klíč se nachází v uzlu node
6      return (node, i) // vrátí uzel a index hledaného prvku
7  fi
8  if node je list then
       // klíč se nenachází v node a node je list
9      return nil
10 else
       // klíč se nenachází v node
11     return SEARCH(node.child_i, k)
12 fi
```

b) Jak byste implementovali vložení prvku do B-stromu?

Při vkládaní je nutné dát si pozor, kam vkládáme. Může se nám stát, že bychom chtěli vložit klíč do už plného uzlu, proto je nutné jej rozdělit. Na přednášce byl představený optimalizovaný algoritmus, který při přechodu dolů preventivně rozděluje všechny plné uzly, aby po vložení do uzlu nemusel opět procházet stromem nahoru a opravovat jej, pokud vyvolal štěpení v dolním uzlu, které by se muselo propagovat nahoru. Pro jednoduchost kódu využijeme metodu SplitChild, která rozdělí uzel s $2t$ potomky na 2 uzly s t potomky a InsertNonfull, která vloží klíč do uzlu, který není plný. Algoritmus vkládaní je tedy následovný:

Procedura Insert(T, k)

vstup: strom T do kterého vkládáme klíč k

```
1  node ← T.root
2  if node.n = 2t − 1 then
       // kořen je plný a má 2t potomků
3      s ← NewNode()
4      T.root ← s// vytvoříme nový kořen
5      s.leaf ← false
6      s.n ← 0
7      s.child_1 ← node
8      SplitChild(s, 1) // rozdělíme prvního potomka
9      InsertNonfull(s, k)// vložíme prvek do nového uzlu
10 else
11     InsertNonfull(node, k)// vložíme prvek do node
12 fi
```

Pomocná procedura SplitChild:

Procedura SplitChild($node, i$)

vstup: uzel $node$ a pozice i, na které se nachází dítě k rozdělení

```
1  z ← NewNode()
2  y ← node.child_i
3  z.leaf ← y.leaf
4  z.n ← t − 1
5  for j ← 1 to t − 1 do
6      z.key_j ← y.key_{j+t}
7  od
8  if not y.leaf then
9      for j ← 1 to t do
10         z.child_j ← y.child_{j+t}
11     od
12 fi
13 y.n ← t − 1
14 for j ← node.n + 1 downto i + 1 do
15     node.child_{j+1} ← node.child_j
16 od
17 node.child_{i+1} ← z
18 for j ← node.n downto i do
19     node.key_{j+1} ← node.key_j
20 od
21 node.key_i ← y.key_t
22 node.n ← node.n + 1
```

Pomocná procedura InsertNonfull:

Procedura InsertNonfull($node, k$)

vstup: vloží hodnotu k do stromu s kořenem *node*, kde uzel *node* není plný

```
1  i ← node.n
2  if node.leaf then
3      while i ≥ 1 ∧ k < node.key_i do
4          node.key_{i+1} ← node.key_i
5          i ← i − 1
6      od
7      node.key_{i+1} ← k
8      node.n ← node.n + 1
9  else
10     while i ≥ 1 ∧ k < node.key_i do
11         i ← i − 1
12     od
13     i ← i + 1
14     if node.child_i.n = 2t − 1 then
15         SplitChild(node, i)
16         if k > node.key_i then
17             i ← i + 1
18         fi
19     fi
20     InsertNonfull(node.child_i, k)
21 fi
```

Preemptivní štěpení uzlů optimalizuje počet čtení z disku za cenu mírného zvýšení paměťových nároků. Preemptivní štěpení lze aplikovat jenom na B-stromy sudé arity. Sudou aritu máme zajištěnou tím, že B-strom definujeme pomocí stupně t a arita je $2t$.

c) Co všechno je nutné ošetřit při odstraňování prvku z B-stromu?

Při odstranění klíče z uzlu musíme dbát na dodržení pravidla o minimálním počtu klíčů ve stromě. Triviální implementace by postupovala tak, že najdeme klíč, ten smažeme a následně strom opravíme průchodem ke kořeni. Operaci opět dokážeme optimalizovat tím, že si strom při přechodu dolů budeme upravovat (stlačovat) abychom po mazání už nemuseli nic opravovat. Mazání klíče si můžeme rozdělit na vícero podpřípadů:

1. pokud klíč je v listu, odstraň klíč a
2. pokud klíč je ve vnitřním vrcholu:
   (a) pokud potomek v intervalu mezi klíčem k a $k-1$ má dostatek klíčů, tedy t, nahraď k jeho předchůdcem a rekurzivně opakuj odstraňování na původní pozici předchůdce,
   (b) pokud potomek má méně klíčů, tak podobně přezkoumej potomka, který je v intervalu k a $k+1$. Pokud podmínku splňuje, nahraď k za jeho následníka a následníka smaž a
   (c) pokud ani jeden z potomků nevyhovoval spoj oba potomky, přesuň k do jejich spojení a rekurzivně smaž k z nového uzlu.

9.15 Jak byste hledali minimum, předchůdce nebo následníka v B-stromě?

Všechny tři procedury jsou u B-stromů obdobné jako u binárního vyhledávacího stromu.

Procedura Minimum(*node*)

vstup: uzel *node* – kořen stromu, jehož minimum hledáme
výstup: ukazatel na uzel s minimálním klíčem

```
1 if node.leaf then
2     return node
3 fi
4 return Minimum (node.child_0)
```

Procedura Successor(*node*, *key*)

vstup: uzel *node* a jeho klíč *key*, jehož následníka hledáme
výstup: ukazatel na uzel s minimálním klíčem

```
1 i ← 1
2 while i ≤ node.n ∧ key > node.key_i do
      // hledání správné větve
3     i ← i + 1
4 od
5 return Minimum(node.child_{i+1})
```

Algoritmus k dohledání předchůdce je symetrický k následníkovi (tedy volá se maximum na levý podstrom, který je pod o 1 menším indexem).

9.16 Naprogramujte reprezentaci B–stromu a elementární operace nad ním. Ve studijních materiálech jsou k tomuto připravené zdrojové kódy: C a Python.

Následující příklady jsou vhodné pro domácí studium.

9.17 Pro použití B-stromů jako datové struktury pro ukládání dat v souborových systémech se většinou B-stromy optimalizují ještě tím, že data ukládají pouze do listů, přičemž uzly ve vyšších patrech se používají pouze k vyhledání a indexaci. V listech navíc máme ukazatele na předchozí a další list. Takto upraveným B-stromům se říká B+ stromy.

a) Jaké mají tyto změny důsledky? Popište, co se zlepší a co zhorší.

Mírně se zvedne paměťová složitost, jelikož klíče vnitřních uzlů jsou jen pomocné, data se v nich fyzicky neukládají.

Ukládání do listů nám však dá možnost rozlišit klíče od dat. Klíče nemusí být stejného typu jako data. Dále jednodušeji získáme předchůdce a následníka, což se hodí, pokud chceme číst data v nějakém intervalu, což odpovídá například čtení souboru, kde ukládaná data jsou jen částí (což může být třeba 4kB) souboru, takže pro přečtení celého souboru potřebujeme přečíst více listů.

b) Jak by probíhalo čtení bloku v B-stromě a B+ stromě?

> Zatímco v B+ stromě stačí najít počátek bloku a pak sekvenčně číst, dokud nejsme na konci, u B-stromu potřebujeme po každém přečtení celého listu vyjít do rodiče, do dat přidat daný klíč a jít do dalšího listu. Zatímco v B+ stromě se tedy čtení podobá čtení zřetězeného seznamu, v B-stromě se jedná o inorder výpis.

9.18 Porovnejte složitost operací *search*, *insert*, *delete* na binárním vyhledávacím stromě, červeno-černém stromě, B-stromě a poli při sekvenčním vyhledávaní a binárním vyhledávaní.

> Následující tabulka shrnuje složitost operací z vyhledávacích datových struktur.

datová struktura	*search*	*insert*	*delete*	*search* průměr
pole – sekvenční hledání	n	n	n	$n/2$
pole – binární hledání	$\log n$	n	n	$\log n$
binární vyhledávací strom	n	n	n	$1.39 \cdot \log n$
červeno-černý strom	$2 \cdot \log n$	$4 \cdot \log n$	$4 \cdot \log n$	$\log n$
B-strom	$2t \cdot \log_{2t} n$	$2t \cdot \log_{2t} n$	$2t \cdot \log_{2t} n$	$t \cdot \log_{2t} n$

Zamyslete se, jaká je vhodná kombinace operací pro testování efektivity stromů a jejich operací Insert, Search a Delete. Ve všeobecnosti efektivita závisí na aplikaci. Například rozhraní Java kolekcí je optimalizováno pro kombinaci 85% Search, 14% Insert a 1% Delete.

9.19 Vyhledávaní v překrývajících se intervalech: navrhněte datovou strukturu, ve které budou uložené číselné intervaly. Struktura bude schopná efektivně nalézt nejmenší interval, ve kterém se nějaká hodnota nachází.

9.20 Jakou minimální a maximální hloubku bude mít B-strom stupně 4 po vložení n prvků? Jakou konkrétní hloubku bude mít B-strom při vkládání seřazené posloupnosti $1, 2, \ldots n$?

9.21 Navrhněte a poté naprogramujte algoritmus, který ověří jestli jsou dva B-stromy identické. Snažte se o co nejefektivnější řešení.

9.22 Lze pomocí našeho algoritmu pro vkládání a odstraňování z B-stromu vytvořit libovolný zadaný B-strom?

Pokuste se například vytvořit plný B-strom stupně $t = 2$ s výškou 2. Nebo naopak strom s minimem uzlů a stejnými parametry.

Kapitola 10

Hašovací tabulka

Hašovací tabulka je dynamická datová struktura, která umožňuje efektivní provádění operací Insert, Delete a Search.

Hašovací funkce nám k zadanému prvku přiřadí hodnotu, kterou bereme jako index do hašovací tabulky.

Kolize je situace, když více prvkům odpovídá stejný index. Dochází k nim, jelikož hašovací funkce zobrazuje velkou množinu všech možných vstupů na menší množinu indexů. Kolize musíme v hašovací tabulce řešit (například pomocí seznamu hodnot se stejným indexem).

Lavinový efekt je vlastnost hašovací funkce, že malé změny vstupu znamenají velké změny výstupu. Je jedním z předpokladů pro uniformní hašovací funkci, tedy funkci, která nám minimalizuje počet kolizí.

Univerzální hašování je metoda hašování, při které se náhodně vybírá hašovací funkce ze skupiny hašovacích funkcí, které garantují nízký počet vzájemných kolizí.

10.1 Jaká datová struktura se hodí pro použití v následujících případech? Svoji volbu okomentujte.

a) Kontrola párovosti různých typů závorek v textu.

Zásobník. Otevírací závorka se vkládá na zásobník, uzavírající maže z vrcholu zásobníku.

b) Zpracování tiskových úloh na tiskárně.

Fronta. Chceme, aby byly dokumenty tisknuty v pořadí, v jakém do tiskárny přišly.

c) Rejstřík pojmů v knize.

Rejstřík je seřazené pole dvojic pojmu a čísel stránek s výskytem. Právě díky odkazu na příslušnou stránku se dá rejstřík považovat za hašovací tabulku, ačkoliv vyhledávání v rejstříku pro člověka není v konstantním čase (ale pro počítač může být, pokud řetězec převedeme na číslo).

d) Řetězec v textovém editoru, který může být modifikován v libovolné pozici.

Pro řetězce, u kterých je možná modifikace z libovolného místa, nelze efektivně použít pole (lineární čas pro vkládání i odstraňování uprostřed). Zřetězený seznam by zvládal tyto operace v konstantním čase, ale na druhou stranu by musel lineárně vyhledávat pozici v řetězci. V textových editorech se tedy používají stromy, kterým se říká lano (rope). Jsou to stromy s omezenou hloubkou a v listech se nenachází jednotlivé znaky, ale různě dlouhé části řetězce.

e) Hierarchii zaměstnanců velké korporace.

Strom, kde rodičem bude šéf a zaměstnanci budou potomky.

f) Seznam studentů, kteří mají zapsaný konkrétní předmět. V ideálním případě s konstantním přidáváním a odebíráním studentů z předmětu.

Pokuste se studenty nejdříve navést na použití pole, které má jako index UČO. Na takovém řešení můžete prodiskutovat nevýhody jeho použití. Zároveň se můžete ptát, kdy by bylo vhodné nebo nevhodné použít pole.

Pro odlehčení lze jako nevhodný příklad uvést nějaký menší předmět, například předmět IA081 Lambda calculus, který si zapisuje zpravidla méně studentů (3).

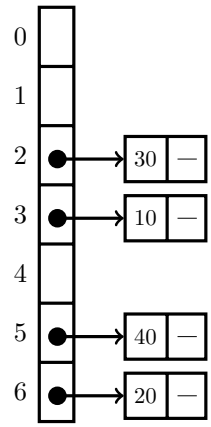
Příklad lze použít jako motivaci k hašovacím tabulkám – hašovací funkce nám umožňuje zmenšit paměťové nároky.

Pro zajištění konstantního přístupu by šlo použít pole všech studentů, kde by bylo indexem UČO. V poli bychom uchovávali informaci, zdali má student daný předmět zapsaný. Takové pole je vlastně hašovací tabulkou, kde hašovací funkcí je přímo UČO a velikost tabulky je počet studentů MUNI (450000 studentů, což jsou velké nároky na paměť). Proto je lepší použít hašovací tabulku, která bude uchovávat studenty na základě haše jejich identifikačního čísla.

10.2 Vybudujte hašovací tabulku, která využívá řetězení prvků. Do tabulky vložte hodnoty 10, 20, 30 a 40.

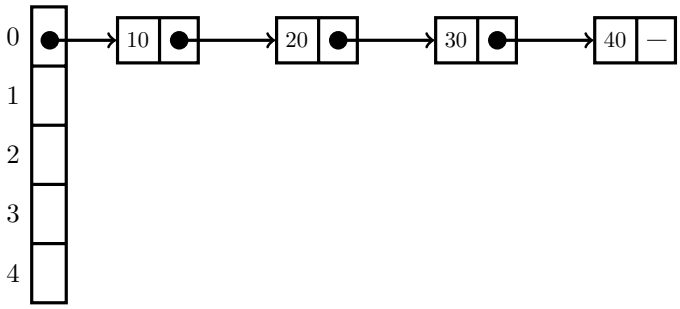
a) Jako hašovací funkci použijte $h(x) = x \bmod 7$.

Výsledná tabulka vypadá následovně:



b) Jako hašovací funkci použijte $h(x) = x \bmod 5$.

Výsledná tabulka vypadá následovně:



c) Jakou má složitost nalezení prvku ve vaší hašovací tabulce?

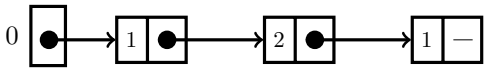
Složitost může být v nejhorším případě až lineární vzhledem k počtu prvků.

d) Jak by jste modifikovali hašovací tabulku, tak aby v ní byla složitost vyhledávaní v nejhorším případě v $\mathcal{O}(\log(n))$?

Můžeme namísto zřetězeného seznamu budovat vyvážený vyhledávací strom. Alternativou je použití vnořených tabulek pro políčka s hodně kolizemi (takové řešení však může být v nejhorším případě lineární). Vnořenou tabulku musíme hašovat jinou funkcí.

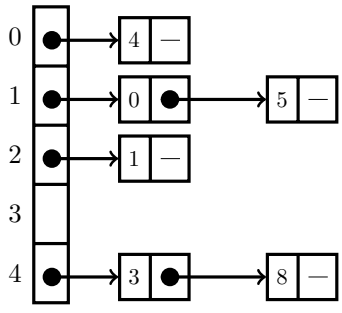
10.3 Jaká hašovací funkce mohla být použita u následujících hašovacích tabulek? Hodnoty v uzlech jsou použity jako klíče.

a) Hašovací tabulka, která řeší kolize pomocí zřetězených seznamů:



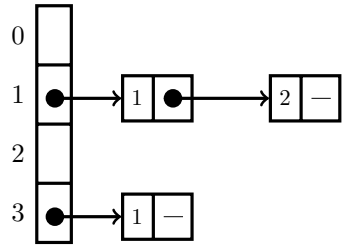
Libovolná hašovací funkce, která vrací pouze hodnotu 0.

b) Hašovací tabulka, která řeší kolize pomocí zřetězených seznamů:



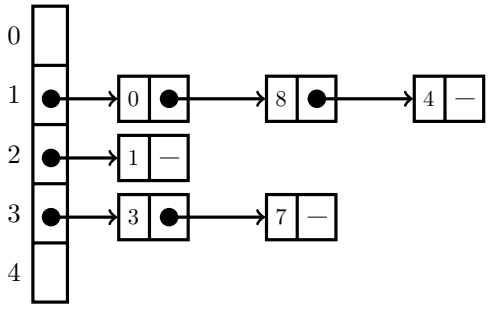
Řešením je například hašovací funkce $h(x) = (x + 1) \bmod 5$.

c) Hašovací tabulka, která řeší kolize pomocí zřetězených seznamů:



Nejedná se o hašovací tabulku, hašovací funkce v tomto případě není funkce, protože zobrazuje stejné vzory na 2 různé obrazy.

d) Hašovací tabulka, která řeší kolize pomocí zřetězených seznamů:



Řešením je například následující hašovací funkce:

$$h(x) = \begin{cases} 1 & \text{pokud } x \in \{0, 4, 8\} \\ 2 & \text{pokud } x \in \{1\} \\ 3 & \text{pokud } x \in \{3, 7\} \end{cases}$$

Což může být například $h(x) = (2^x) \bmod 5$. Taková funkce by mimochodem nikdy nevyužila pozici 0, protože žádná mocnina dvojky nekončí číslicí 0 ani 5.

10.4

a) Co nám hašovací tabulka nabízí oproti seznamu a poli? Jaké jsou výhody a jaké nevýhody?

V ideálním případě umožňuje hašovací tabulka konstantní přístup k prvku danému klíčem. Přístupem je zde myšleno vkládání, odstraňování a vyhledávání. Cenou je paměťová náročnost dána velikostí použité tabulky.

b) Kde byste hašovací tabulku použili vzhledem k jejím výhodám?

Hašovací tabulka se hodí v situacích, kdy potřebujeme rychlý přístup k prvku. Příklady použití tedy můžou být vyhledávání řetězce (rejstřík, slovník) a routovací tabulka.

c) Co je ovlivněno rozsahem výstupu hašovací funkce a jaký rozsah bychom tedy měli zvolit?

Rozsahem hodnot hašovací funkce určujeme, jak časté budou kolize proti paměťové složitosti. Velký rozsah hodnot znamená málo kolizí ale i spoustu volných slotů. Malý obor hodnot zase znamená menší paměťové nároky ale více kolizí a z toho pramenící větší časovou

složitost přístupu. Ideální je rozsah roven očekávanému počtu prvků, což však většinou nevíme předem.

Dobrým řešením, používaným v praxi, je proměnlivá velikost hašovací tabulky. Pokud počet prvků vzroste až na velikost tabulky, vytvoříme tabulku novou s větším rozsahem hodnot hašovací funkce a tabulku postupně znovu zaplníme. Toto přepočítání je drahá operace, která je provedena v lineárním čase, takže získáváme časovou složitost pro vložení $\mathcal{O}(n)$. Pokud budeme velikost tabulky vždy zdvojnásobovat, pak bude průměrná složitost stejná jako vložení normálního prvku.

10.5 Mějme následující hašovací funkce na hašování řetězců. Jaké jsou jejich výhody a nevýhody?

Můžete nejdříve dát studentům za úlohu vymyslet, jak by hašovali řetězce, a pak je nechat porovnávat jejich funkci s následujícími.

a) Hašovací funkce, která sečte znaky řetězce.

Tato funkce vrací pro různé řetězce stejné hodnoty – pouhá změna pořadí znaků v řetězci zachovává stále stejný haš.

Také je její obor hodnot příliš velký, není shora omezen.

b) Předchozí funkce, ale místo součtu znaků použijeme operaci *xor*.

Stále vrací pro různé permutace stejných znaků stejnou hodnotu. Obor hodnot je již omezen, a to na velikost abecedy (řekněme 256 hodnot). To může být velmi málo a vést k častým kolizím.

c) Součin znaků modulo zadanou velikostí tabulky (velikost může být libovolná).

Stále vrací pro různé permutace stejných znaků stejnou hodnotu. Obor hodnot je nastaven podle nás. Problémem však je, že pokud se v řetězci nachází znak s hodnotou 0, pak už výsledek stále zůstane 0. Stejně tak se k 0 lze dostat posloupností násobení, kde se v součinu prvočinitelů bude vyskytovat hodnota modula. Obor hodnot tedy není uniformně rozdělen.

d) Suma $znak \cdot pozice$ modulo zadanou velikostí tabulky (velikost může být libovolná).

Už téměř ideální řešení, problémem zůstává jen případ, kdy pozice znaku je násobkem velikosti tabulky, pak se hodnota znaku do řešení nezapočítá. Druhým problémem je, že pozice, které mají vysokého největšího společného dělitele s velikostí tabulky mají vysokou pravděpodobnost, že jejich znaky můžou být vyhodnoceny stejně pro různé symboly.

e) Suma $znak \cdot pozice$ modulo zadanou velikostí tabulky (velikost je prvočíslo).

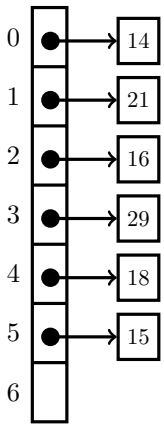
Napravuje 2. problém předchozího řešení.

10.6 Mějme hašovací tabulku a hašovací funkci $h(x) = x \bmod 7$. Řešte kolize v tabulce pomocí lineárního sondování.

Metoda lineárního sondování (otevřené adresování) funguje tak, že v případě kolize vložíme prvek do jiného volného slotu v tabulce. Pro hašovací funkci $h(x)$ by hašovací funkce lineárního sondování byla $h(x,i) = (h(x) + i) \bmod n)$, kde $h(x)$ je počáteční hodnota, i počet kolizí, které nám již při vkládání tohoto klíče nastaly.

a) Vložte do tabulky následující hodnoty: 14, 16, 21, 18, 29, 15.

Výsledná tabulka vypadá následovně:



b) Jak budeme postupovat při hledání prvku a jaká bude složitost hledání?

Při nepříznivých podmínkách dostáváme až lineární složitost, kdy musíme projít celou tabulku, abychom našli náš prvek.

c) Jak byste detekovali, že je už hašovací tabulka plná? Jak byste tuto tabulku předělali?

Všimněme si, že naše lineární sondování ukládá hodnotu při kolizi na první volné místo za pozici s kolizí. Pokud se při hledání volného místa vrátíme na počáteční místo v tabulce, pak prohlásíme, že je tabulka plná. Pro nápravu můžeme zvětšit n v hašovací funkci, čím se zvětší rozsah tabulky a můžeme překopírovat hodnoty z menší tabulky do větší (znovu za použití hašovací funkce, aby se hodnoty nacházely na správných pozicích).

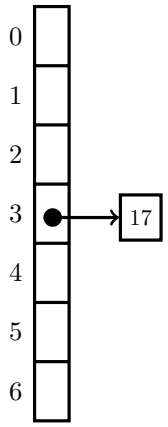
10.7 Mějme hašovací tabulku a hašovací funkci $h(x) = x \bmod 7$. Řešte kolize v tabulce pomocí kvadratického sondování, kde budou konstanty $c_1 = 2$ a $c_2 = 1$.

Metoda kvadratického sondování je rozšíření metody lineárního sondování, kde předpis funkce hašování je $h(x,i) = (h(x) + c_1 i + c_2 i^2) \bmod n)$, kde $c_2 \neq 0$. Pro $c_2 = 0$ by funkce degradovala na lineární sondování. Výhoda kvadratického sondováni proti lineárnímu je, že se lépe vyhýbá hromadění klíčů na jednom místě.

a) Vložte do tabulky následující hodnoty: 17, 24, 16, 13.

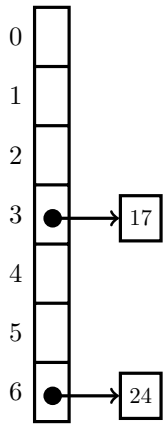
Postupně vkládáme hodnoty:

1. Vložíme 17: Tedy $h(x, 0) = h(x) + 0 = 3$.

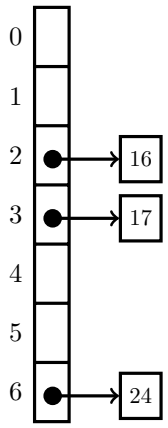


2. Vložíme 24: Jelikož je už pozice 3 zabraná zvětšíme krok na $i = 1$ pak

$$h(x, i) = (h(x) + 2i + i^2) \bmod 7 = 6$$



3. Vložíme 16: Tedy $h(x, 0) = h(x) + 0 = 2$.

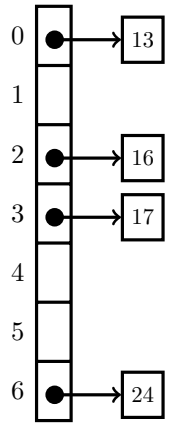


4. Vložíme 13: Jelikož je už pozice 6 zabraná zvětšíme krok na $i = 1$, pak

$$h(x, i) = (h(x) + 2i + i^2) \bmod 7 = 2$$

Což je ale opět zabraná pozice, proto musíme zvětšit krok na $i = 2$, pak

$$h(x, i) = (h(x) + 2i + i^2) \bmod 7 = (h(x) + 4 + 4) \bmod 7 = 0$$



b) Jakou bude mít složitost vyhledání a smazání prvku?

Při nepříznivých podmínkách opět dostáváme až lineární složitost, kdy musíme projít celou tabulku, abychom našli náš prvek.

Kvadratické sondování se využívá i pro hledání volných bloků paměti v adresových schématech souborových systémů. Jeho výhodou je skákání po velkých blocích a v případě plných bloků je přeskočí mnohem rychleji než lineární přístup.

V tomto okamžiku by už měli studenti znát všechny datové struktury, následující příklady procvičují již získané dovednosti z kapitol 5–10.

10.8 Navrhněte datovou strukturu pro reprezentaci množiny prvků. Datová struktura by měla mít všechny operace jako Insert, Delete, Search efektivní. Množina prvků nesmí obsahovat duplikáty.

Množina se v knihovnách programovacích jazyků reprezentuje pomocí vyváženého (obvykle červeno-černého) stromu. Alternativně ji lze reprezentovat hašovací tabulkou, ale ta se kvůli velké extrasekvenční paměťové složitosti a nejhůře lineární složitosti používá méně.

10.9 Navrhněte datovou strukturu, která se inicializuje v čase $\mathcal{O}(n)$ z n prvků. Struktura dále podporuje pouze operace Search a Delete v konstantním čase.

Pokud dokážeme vytvořit perfektní hašovací funkci, můžeme využít hašovací tabulku s vhodnou velikostí.

10.10 Navrhněte datovou strukturu, která podporuje následující operace:

- Insert v čase $\mathcal{O}(\log(n))$.
- Delete v čase $\mathcal{O}(\log(n))$.
- Nalezení většího a menšího prvku (hodnotou, ne časem vložení) v čase $\mathcal{O}(\log(n))$.
- Next(x) – nalezne první prvek, který byl vložený po prvku x a ještě se nachází v datové struktuře v čase $\mathcal{O}(1)$.

Vyvážený vyhledávací strom – například červeno-černý strom. Navíc však všechny uzly musíme spojit do obostranně spojovaného seznamu. K němu si budeme udržovat poslední vložený prvek a v případě vkládání na nový prvek odkážeme z bývalého posledního prvku. Při odstraňování budeme odstraňovat nejen ze stromu, ale musíme i napravit ukazatele seznamu.

Bylo loňským implementačním testem - v ISu je to jako TreeSet.

10.11 Navrhněte datovou strukturu, která reprezentuje hierarchii zaměstnanců. Datová struktura musí umět nejhůře v lineárním čase vzhledem k počtu zaměstnanců najít ke dvěma zaměstnancům jejich nejnižšího společného nadřízeného.

Obecný n-ární strom (ne vyhledávací). Byt podstromem je stejná relace jako být podřízeným. Nalezení nadřízeného pak znamená nalezení nejmenšího společného předka.

Ne vždy však jde o strom, jeden zaměstnanec může zastávat více rolí a tak mít různé nadřízené, nebo si sám může být nadřízeným. Obecným řešením je tedy orientovaný acyklický graf.

10.12 Navrhněte následující metody nad hašovací tabulkou, které využívá nějakou z předchozích hašovacích funkcí. Vhodně také řešte kolize, které mohou nastat.

a) Insert vloží objekt do hašovací tabulky.

Použití seznamu k řešení kolizí:

Procedura Insert(T,(k,v))

vstup: tabulka T, (k,v) je dvojice klíče a hodnoty

```
1 h ← Hash(k) mod |T|
2 vlož (k,v) do seznamu T[h]
```

Použití kvadratické sondovací metody pro řešení kolizí:

Procedura Insert(T,(k,v))

vstup: tabulka T, (k,v) je dvojice klíče a hodnoty

```
1  i ← 0
2  h ← Hash(k) mod |T|
3  while i < |T| do
4      if T[h] je prázdný slot then
5          T[h] ← (k,v)
6          return
7      fi
8      i ← i + 1
9      h ← (Hash(k) + i^2) mod |T|
10 od
11 return tabulka je plná
```

b) Search (k) metoda vrací index slotu obsahující klíč k nebo hodnotu Nil, pokud daný slot neexistuje.

Použití seznamu na řešení kolizí:

Procedura Search(T, k)

vstup: tabulka T, klíč k

```
1 h ← Hash(k) mod |T|
2 if (k, v) je v seznamu T[h] then
3     return v
4 else
5     return nil
6 fi
```

Použití kvadratické sondovací metody pro řešení kolizí:

Procedura Search(T, k)

vstup: tabulka T, klíč k

```
1  i ← 0
2  h ← Hash(k) mod |T|
3  while i < |T| do
4      if T[h] je plný a T[h].k = k then
5          return v
6      fi
7      i ← i + 1
8      h ← (Hash(k) + i^2) mod |T|
9  od
10 return nil
```

c) Delete smaže klíč z hašovací tabulky.

Použití seznamu na řešení kolizí:

Procedura Delete(T, k)

vstup: tabulka T, klíč k

```
1 h ← Hash(k))mod|T|
2 smaž (k, v) ze seznamu T[h]
```

Použití kvadratické sondovací metody pro řešení kolizí:

Procedura Delete(T, k)

vstup: tabulka T, klíč k

```
1 h ← Search(T, k) // upravený Search vrací index
2 if h ≠ nil then
3     smaž prvek v T[h]
4 fi
```

10.13 Naprogramujte si funkční hašovací tabulku, která bude řešit kolize pomocí seznamů. Pro seznamy můžete využít již naprogramované své nebo knihovní funkce. Ve studijních materiálech jsou k tomuto připravené zdrojové kódy: C a Python.

Následující příklady jsou vhodné na domácí studium.

10.14

a) Jaké jsou výhody a nevýhody metody dělení, kde $h(k) = k \mod (m)$? Jaké hodnoty m je vhodné používat?

Výhodou metody dělení je její rychlost, protože vyžaduje jenom jednu operaci dělení. Naopak zase nevýhodou je potřeba se vyhýbat některým konstantám m, pro které hašovací funkce není efektivní. To platí například pro m jako mocniny dvou, kdy $h(k)$ vrací spodní bity klíče k. Výhodné klíče jsou naopak prvočísla vzhledem k jejich nesoudělnosti. Abychom se vyhnuli problému s mocninou dvou, je dobré vybírat prvočísla dále od těchto mocnin.

b) Jak funguje multiplikativní metoda a jaké má výhody a nevýhody proti metodě dělením?

Multiplikativní metodu můžeme nadefinovat takto:

1. Zvolíme konstantu A v rozsahu $0 < A < 1$.
2. Vynásobíme klíč k konstantou A.
3. Necháme si desetinnou část z výsledku.
4. Tuhle část vynásobíme m.
5. Zaokrouhlíme dolů.

Ve zkratce $h(k) = \lfloor m(kA \mod 1) \rfloor$, kde $kA \mod 1 = kA - \lfloor kA \rfloor$ = desetinná část z kA. Nevýhodou multiplikativní metody je menší rychlost oproti metodě dělením. Ale naopak výhodou je, že hodnota m není kritická.

10.15

a) Co bychom chtěli po dobré hašovací funkci? Jak by měla zobrazovat zadané hodnoty?

Ideální hašovací funkce by měla zachovávat uniformnost hašování. Tedy pro náhodnou množinu vstupů by jednotlivé haše měly mít stejnou pravděpodobnost výskytu, čímž zajistíme minimální počet kolizí. Tato vlastnost také souvisí s lavinovým efektem, který říká, že malé změny vstupu znamenají velké změny výstupu.

V praxi tahle podmínka ale nemusí být postačující, vzhledem k tomu, že nevíme nic o pravděpodobnostním rozložení množiny klíčů. Pro lepší výsledky se v praxi využívají heuristické funkce, které vycházejí z vědomostí o doméně klíčů.

Dále od funkce očekáváme jednoduchost a rychlost výpočtu.

b) Jak byste hašovali složitější objekty, které obsahují více jednoduchých objektů?

Jedním z možných způsobů je spočítat si haše pro jednodušší podobjekty a následně xorovat, nebo sčítat haše daných podobjektů.

10.16 Mějme hodnoty 10, 13, 18, 3, 8, 40, 28, hašovací tabulku o velikosti 7 a hašovací funkci $h(x) = x \mod 7$. Zapište výsledek vkládaní hodnot pro každou z kolizních strategií.

Hašujeme vstupní klíče:
$10 \bmod 7 = 3; 13 \bmod 7 = 6; 18 \bmod 7 = 4; 3 \bmod 7 = 3; 8 \bmod 7 = 1; 40 \bmod 7 = 5; 28 \bmod 7 = 0$

Prvky tabulky	Metoda řetězení	Metoda lineárního sondování	Metoda kvadratického sondování
T[0]	[28]	[40]	[3]
T[1]	[8]	[8]	[8]
T[2]	[]	[28]	[28]
T[3]	[10, 3]	[10]	[10]
T[4]	[18]	[18]	[18]
T[5]	[40]	[3]	[40]
T[6]	[13]	[13]	[13]

10.17 Navrhněte datovou strukturu, která umožňuje řazení pomocí více klíčů. Podle primárního klíče však dochází k častým kolizím, takže potřebujete řadit i podle dalších klíčů. Vaše struktura umožňuje v nejhůře logaritmickém (vzhledem k počtu unikátních primárních klíčů) čase vrátit všechny hodnoty se stejným primárním klíčem, přičemž prvky jsou seřazeny podle sekundárního klíče.

Vyvážený binární vyhledávací strom řadící podle primárního klíče, který v uzlu kromě ukazatelů na potomky obsahuje i ukazatel na BVS řadící podle sekundárních klíčů, který obsahuje všechny hodnoty se zadaným primárním klíčem. Pro další klíče lze postupovat obdobně.

10.18 Navrhněte datovou strukturu (nebo modifikujte již známou), která podporuje vrácení hodnot z intervalu v čase $\mathcal{O}(\log(n))$ (struktura vrácených hodnot záleží na vás, lineární množtví hodnot nelze samozřejmě „vypsat“ v $\mathcal{O}(\log(n))$).

Vyvážený vyhledávací strom – například červeno-černý strom.

10.19 Navrhněte datovou strukturu, která umožňuje vkládaní prvku s klíčem v $\{0, 1\}$. Po struktuře dále chceme, aby uměla vrátit v konstantním čase všechny prvky, které mají pouze klíč 1 nebo prvky s klíčem pouze 0.

2 seznamy, jeden pro prvky s klíčem 0, druhý pro prvky s klíčem 1.

10.20 Navrhněte datovou strukturu pro množinu, ve které operace jsou Insert, Delete a Search v čase $\mathcal{O}(1)$. Můžete předpokládat, že prvky jsou hodnoty z intervalu $[1, 2, \ldots, n]$.

10.21 Nechť $S = \{s_1, s_2, \ldots, s_l\}$ a $T = \{t_1, t_2, \ldots, t_m\}$ jsou dvě množiny přirozených čísel takových, že $1 \leq s_i, t_j \leq n$ pro všechny $1 \leq i \leq l$ a $1 \leq j \leq m$. Navrhněte algoritmus, který rozhodne, zdali platí $S = T$ v čase $\mathcal{O}(l + m)$.

10.22 Navrhněte datovou strukturu, která poskytuje operace:

- Insert(x, D) vloží x do D.
- Delete(k, D) smaže k-tý nejmenší prvek z D.
- Member(x, D) vrátí *true* pokud $x \in D$.

Všechny operace musí být v čase $\mathcal{O}(\log(n))$.

10.23 Navrhněte datovou strukturu množiny, která poskytuje operace:

- Insert(x, D) vloží x do D.
- Delete(x, D) smaže x z D.
- Member(x, D) vrátí *true* pokud $x \in D$.
- Next(x, D) vrátí nejmenší prvek v D větší než x.
- Union(S, D) spojí struktury S a D.

Všechny operace musí být v čase $\mathcal{O}(\log(n))$, kromě operace Union, která má být v čase $\mathcal{O}(n)$.

10.24 Mějme datovou strukturu D a operaci $\oplus$, kde $D = \{d_1, \ldots, d_n\}$. Navrhněte datovou strukturu, která umožňuje sečíst $(d_i \oplus d_{i+1} \oplus \cdots \oplus d_j)$ pro libovolné $i \leq j$ s $\mathcal{O}(\log(n))$ operací $\oplus$.

10.25 Navrhněte datovou strukturu, která implementuje množinu uspořádaných dvojic (p, k), kde k je klíč a p je priorita. Vaše struktura musí umět následující operace v čase $\mathcal{O}(\log(n))$:

- Insert(p, k) vloží prvek s prioritou p a klíčem k.
- Member(k) vrátí prvek s nejmenší prioritou mezi prvky, které mají klíč menší nebo roven k.
- Delete(k) smaže všechny prvky s klíčem k.

10.26 Navrhněte datovou strukturu, která je tvořená z n hodnot $x_1, x_2, \ldots, x_n$. Struktura dokáže rychle vrátit nejmenší hodnotu z intervalu $x_i, \ldots, x_j$, pro $i \leq j$. Struktura splňuje následující podmínky.

1. Využívá $\mathcal{O}(n \cdot \log(n))$ prostoru a vrací nejmenší hodnotu z intervalu v čase $\mathcal{O}(\log(n))$.
2. Využívá $\mathcal{O}(n)$ prostoru a vrací nejmenší hodnotu z intervalu v čase $\mathcal{O}(\log(n))$.

10.27 Rozhodněte, jestli platí následující tvrzení:
Kolekce $H = \{h_1, h_2, h_3\}$ hašovacích funkcí je univerzální, pokud hašovací funkce mapují universum $\{A, B, C, D\}$ klíčů na rozsah hodnot $\{1, 2, 3\}$ vzhledem k následující tabulce:

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
A	1	0	2
B	0	1	2
C	0	0	0
D	1	1	0

▌ Tvrzení platí.

Kapitola 11

Grafy I.

Neorientovaná hrana je dvouprvková množina $\{u, v\}$, která značí, že vrcholy u a v spolu sousedí.

Orientovaná hrana je uspořádaná dvojice vrcholů (u, v), která značí, že z vrcholu u vychází hrana do vrcholu v.

Ohodnocená hrana má přiřazenou délku, což je cena přechodu této hrany (popřípadě šířka hrany). Pokud je délka celočíselná, pak lze graf s ohodnocenými hranami převést na graf s hranami neohodnocenými (tedy ohodnocenými jedničkou), hranu délky n nahradíme n po sobě jdoucími hranami délky 1.

V případě, že jednu dvojici vrcholů spojuje více hran, hovoříme o paralelních hranách (a multigrafu).

Stupeň vrcholu je počet hran, které z vrcholu vychází.

Graf je uspořádaná dvojice množiny vrcholů a množiny hran.

1. **Orientovaný graf** má orientované hrany.
2. **Neorientovaný** má neorientované hrany.

V bipartitní grafu lze vrcholy rozdělit do dvou skupin, přičemž hrany mohou existovat jen z jedné skupiny do druhé, ne v rámci jedné skupiny.

Reprezentace grafu v paměti počítače lze provést různými způsoby, které volíme podle typu grafu plánovaného využití.

1. **Seznam sousedů v neorientovaném grafu** – ke každému vrcholu u udržujeme seznam vrcholů v tak, že $\{u, v\}$ je hrana. Hodí se zejména pro řídké grafy, pro které má menší prostorovou složitost.
2. **Seznam následníků v orientovaném grafu** – ke každému vrcholu u udržujeme seznam následníků v tak, že (u, v) je hrana.
3. **Matice sousednosti v neohodnoceném grafu** je matice rozměrů $|V| \times |V|$, kde se přítomnost hrany reprezentujeme hodnotou $A_{uv} = 1$ v případě, že existuje hrana z u do v, a 0 v případě, že neexistuje.
4. **Matice vzdáleností v ohodnoceném grafu** je matice rozměrů $|V| \times |V|$, hranu (u, v) délky d reprezentujeme hodnotou $A_{uv} = d$ v případě, že mezi uzly u a v hrana není, pak klademe $A_{uv} = \infty$.

Transponovaný graf je graf obsahující hrany opačně orientované než graf původní. Má smysl jej definovat pouze pro orientovaný graf. Pro reprezentaci pomocí matice sousednosti je transponování grafu totožné s transponováním matice, která graf popisuje.

Průchody grafu

1. **BFS** (breadth-first search), tedy prohledávání do šířky. Používá datovou strukturu fronta (ve které uchováváme vrcholy čekající na zpracování) a hodí se pro hledání nejkratší cesty nebo testování, zdali je graf bipartitní.
2. **DFS** (depth-first search), tedy prohledávání do hloubky. Používá datovou strukturu zásobník (ve kterém ukládáme cestu). Používá se k hledání cyklů v grafu, nalezení topologického uspořádání grafů nebo rozdělení grafu na silně souvislé komponenty. K těmto aplikacím využívá časové známky, které popisují, kdy jsme vrchol objevili ($u.d$) a kdy jsme jej opustili ($u.f$).

Typy hran lze na základě DFS průchodu z konkrétního vrcholu klasifikovat na:

1. **Stromová hrana** (tree edge) je hrana (u, v), která odpovídá hraně v DFS stromě.
2. **Zpětná hrana** (back edge) je hrana (u, v), která spojuje vrchol s jeho předkem v DFS stromě.
3. **Dopředná hrana** (forward edge) je hrana (u, v), která spojuje uzel s některým z jeho potomků, ale není hranou výsledného DFS lesa (tím se liší od stromové hrany – vrchol který jsme s její pomocí objevili už byl objeven dříve).
4. **Příčná hrana** (cross edge) je hrana (u, v) je hrana, které neodpovídá žádná z jiných klasifikací.

Topologické uspořádání definujeme na orientovaném acyklickém grafu a jedná se o lineární uspořádání, ve kterém se může vrchol u vyskytovat před vrcholem v jedině pokud z v do u nevede hrana. Používá se pro vyjadřování závislostí (prerekvizity předmětů), řazení procesů...

Silně souvislá komponenta orientovaného grafu je maximální množina vrcholů taková, že z každého vrcholu této množiny se lze dostat do libovolného jiného vrcholu této množiny.

Motivační příklad, aby si studenti uvědomili, co vše může být grafem. Příkladů je hodně, zaberou hodně času, ale jsou zajímavé a je vhodné, aby si studenti udělali představu, jak data reprezentovat.

11.1 O jaký graf se jedná? Popište, co jsou v grafu hrany a co vrcholy:

a) dopravní síť,

Orientovaný ohodnocený graf (orientovaný, protože může obsahovat jednosměrky), kde uzel odpovídá křižovatce a hrana je silnice (ohodnocená délkou, nebo časem).

b) seznam kamarádů, kolegů, spolužáků,

V ideálním světě neorientovaný neohodnocený graf. Vhodná bude asi reprezentace seznamem následníků, matice by byla pro celý svět moc velká.

c) skládání Rubikovy kostky,

Neorientovaný neohodnocený graf. Jednotlivé uzly reprezentují možné stavy Rubikovy kostky, hrany pak naznačují, že existuje přechod mezi danými dvěma stavy. Pozor na to, ne všechny stavy musí být reálně existující (teoreticky jsme schopni přeskládat kostku tak, že ji rozebereme a náhodně seskládáme kostičky, což je ale nedovolený tah). V praxi je pak držení všech možných stavů v paměti počítače nereálné.

d) postup řešení her dvou hráčů (šachy, dáma),

Orientovaný neohodnocený graf. Co stav šachovnice, to uzel, hrany opět symbolizují přechody mezi stavy.

e) elektrický obvod na tištěném spoji,

Neorientovaný (ohodnocený) graf. Vrcholy jsou elektrické součástky v obvodě, hrany jsou spoje.

f) zdrojový kód programu,

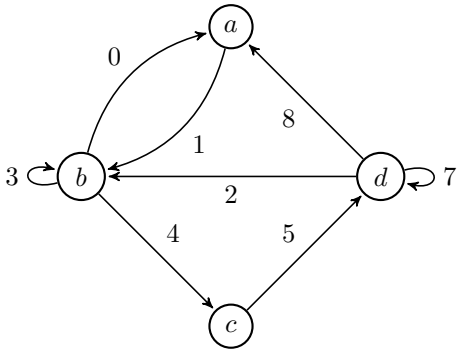
Máme několik možností.

1. Graf toku řízení je orientovaný neohodnocený graf. Vrcholy jsou bloky kódu, hrany značí větvení podle skoků v kódu.
2. Graf volání funkcí je orientovaný neohodnocený graf. Vrcholy jsou funkce, hrany vzájemné volání.
3. Syntaktický strom je orientovaný neohodnocený graf. Vrchol reprezentuje operátory, jeho potomci slouží jako operandy, na které je operátor aplikován.

g) projekt výroby auta.

Orientovaný graf. Vrcholy jsou úkoly, které je potřeba při výrobě splnit, hrany odpovídají závislostem mezi úkoly a jsou v topologickém uspořádání (abych mohl svařit karoserii, musím mít vyrobeny její části).

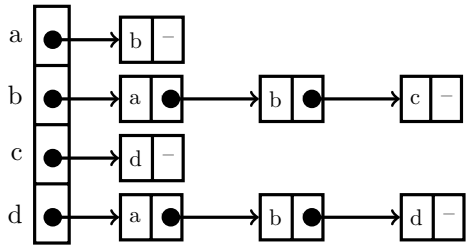
11.2 Zapište tento graf pomocí matice vzdáleností a seznamu následníků:



Matice pro tento graf vypadá následovně:

	a	b	c	d
a	∞	1	∞	∞
b	0	3	4	∞
c	∞	∞	∞	5
d	8	2	∞	7

Seznam následníků bude tedy:



11.3 Mějme orientovaný graf zadaný seznamem následníků.

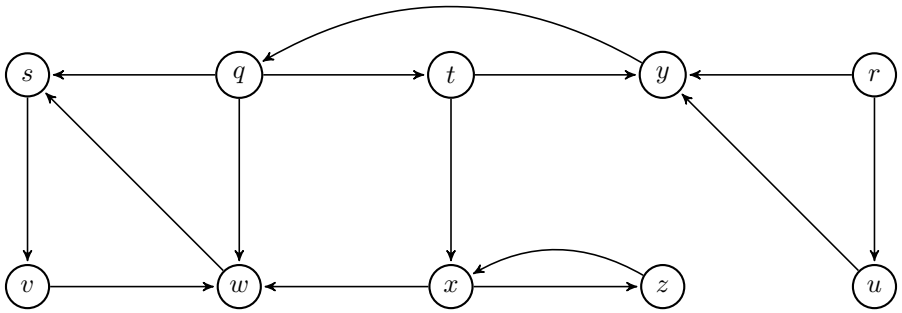
a) Jakou má složitost určení počtu výstupních hran ze zadaného vrcholu?

K zadanému vrcholu se dostaneme s konstantní složitostí, počet výstupních hran pak určíme jako velikost seznamu následujících vrcholů. To má buďto lineární složitost vzhledem k počtu hran ze zadaného vrcholu, nebo konstantní složitost, pokud si pamatujeme velikost seznamu.

b) Jaká bude složitost pro vstupní hrany?

Abychom nalezli všechny vstupní hrany do zadaného vrcholu, musíme projít všechny hrany grafu. Složitost je tedy lineární vzhledem k počtu hran celého grafu. V grafu reprezentovaném maticí by složitost vstupních i výstupních hrany byla lineární vzhledem k počtu vrcholů v grafu.

11.4 Graf, se kterým budete v tomto cvičení pracovat:

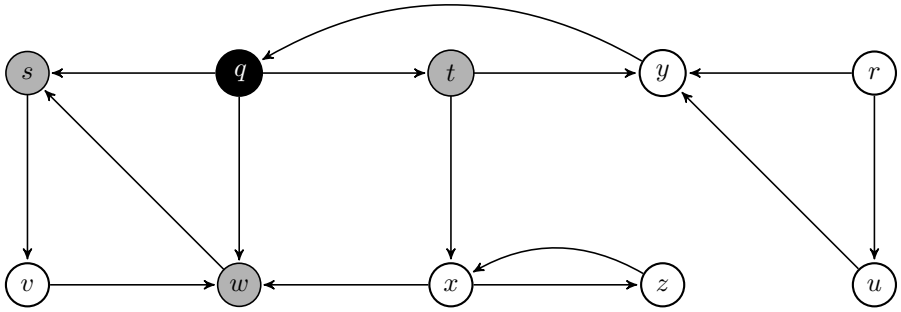


Pokud vám algoritmus umožní volit z více vrcholů, pak je berte podle abecedy.

a) Zjistěte pomocí BFS, zdali existuje cesta z q do y a určete její délku.

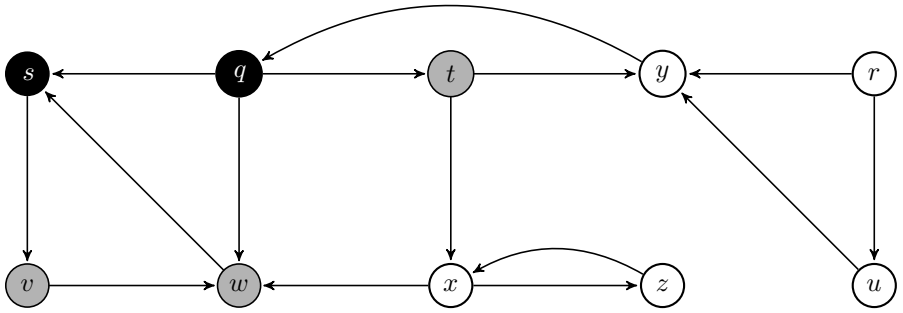
Provedeme BFS průchod. Bílou barvou značíme nenavštívené vrcholy, šedou vrcholy, které máme ve frontě a černou již zpracované vrcholy.

(a) Začneme prohledávat z vrcholu q:



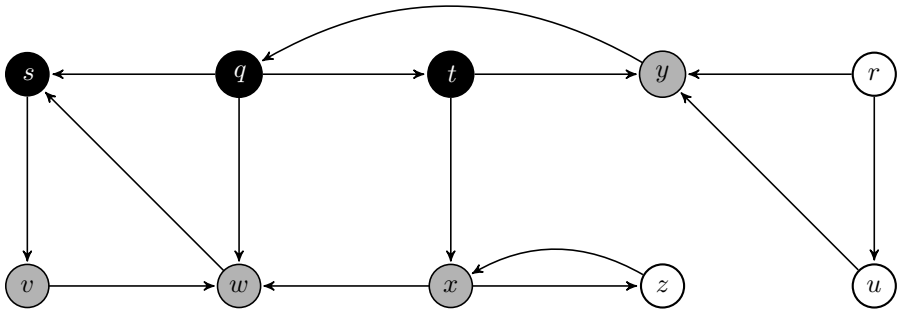
Fronta obsahuje s, t, w.

(b) Pokračujeme lexikograficky do s:



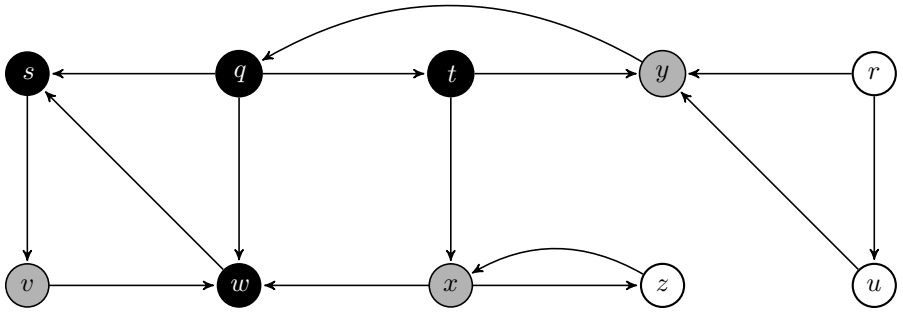
Fronta obsahuje t, w, v.

(c) Pokračujeme do t, který je prvním prvkem fronty:



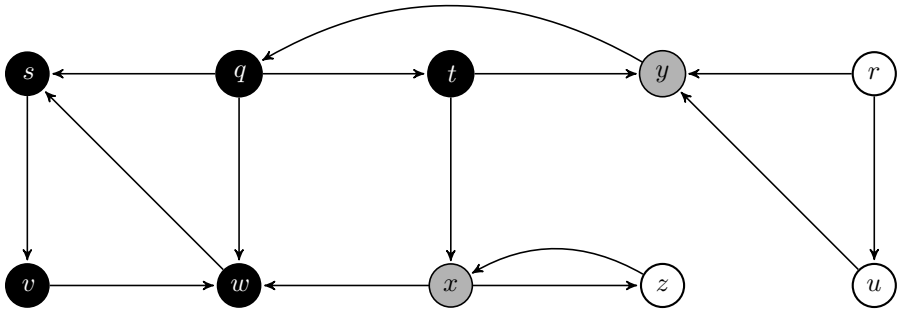
Fronta obsahuje w, v, x, y.

(d) Pokračujeme do w:



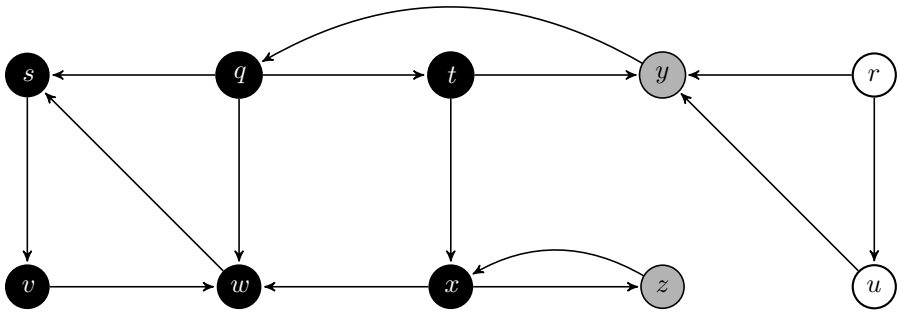
Fronta obsahuje v, x, y.

(e) Pokračujeme do v:



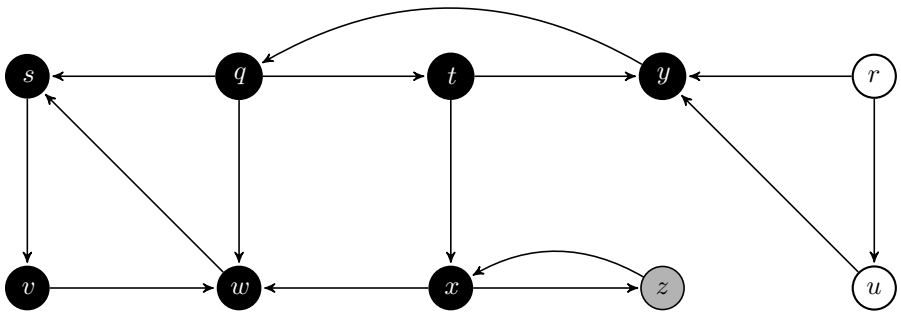
Fronta obsahuje x, y.

(f) Pokračujeme do x:



Fronta obsahuje y, z.

(g) Nalezneme y, ve vzdálenosti 2 zanoření z q:



b) Jaké vrcholy jste navštívili? Jaký vrchol musíte zvolit jako počáteční, abyste prošli celý graf?

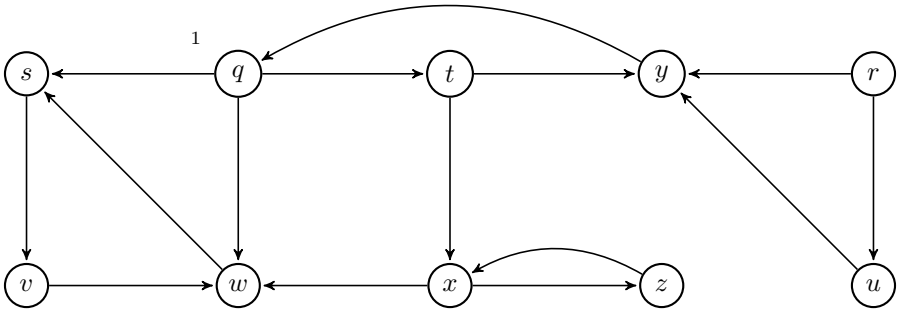
Všechny kromě vrcholů r a u. Právě z vrcholu r lze celý graf projít, jelikož se z r vede hrana do u a také se přes y lze dostat do q, ze kterého jsme nalezli zbytek grafu.

Sem lze zařadit příklad 11.8 na BFS strom, nebo si alespoň můžete připravit na tabuli bokem BFS strom, na kterém příklad 11.8 později ukážete.

c) Určete typ všech hran grafu pomocí průchodu DFS všech silně souvislých komponent. Pokud vám algoritmus umožní volit z více vrcholů, vybírejte je v abecedním pořadí (začněte tedy z vrcholu q).

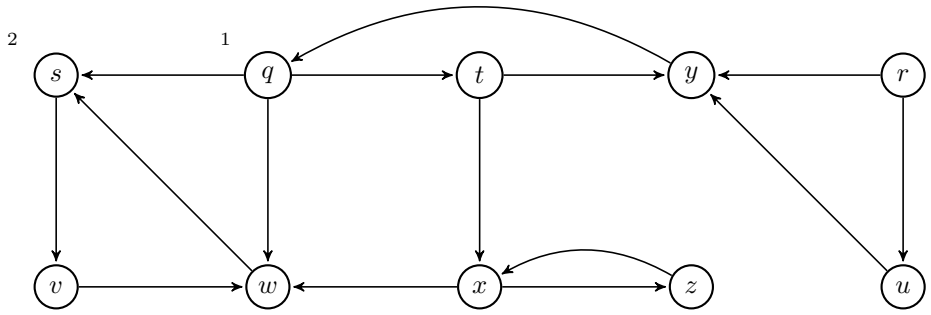
Provedeme DFS průchod, přičemž budeme značit časové známky.

(a) Začneme z vrcholu q:



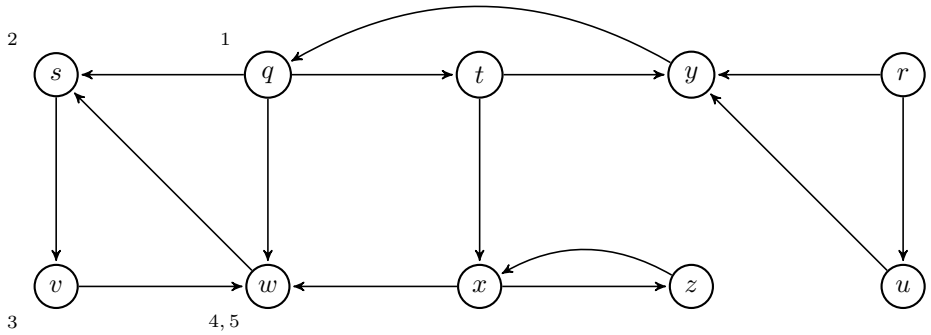
Zásobník obsahuje q.

(b) Pokračujeme do s:



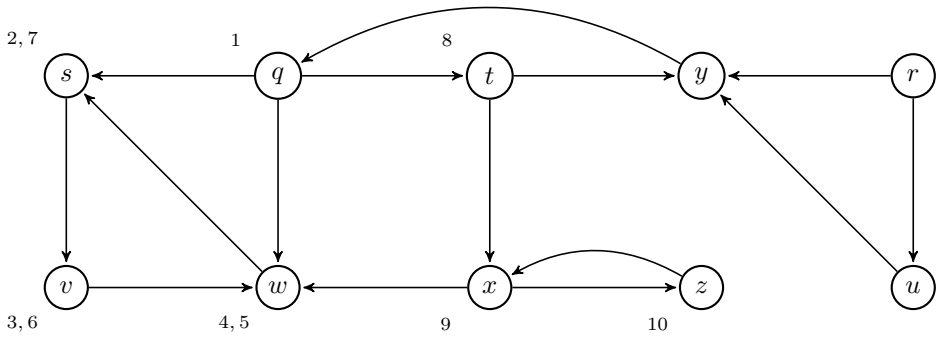
Zásobník obsahuje s, q (dno je vpravo).

(c) Pokračujeme do v (zásobník obsahuje v, s, q) a w (zásobník obsahuje w, v, s, q), pak w opouštíme:



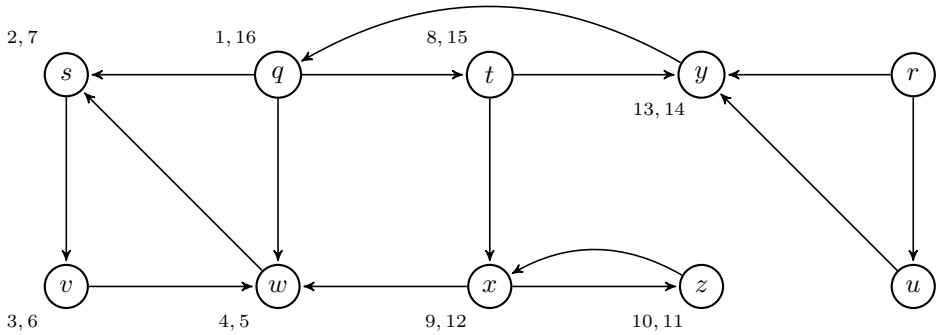
Zásobník obsahuje v, s, q (dno je vpravo).

(d) Pokračujeme do t, x a z:



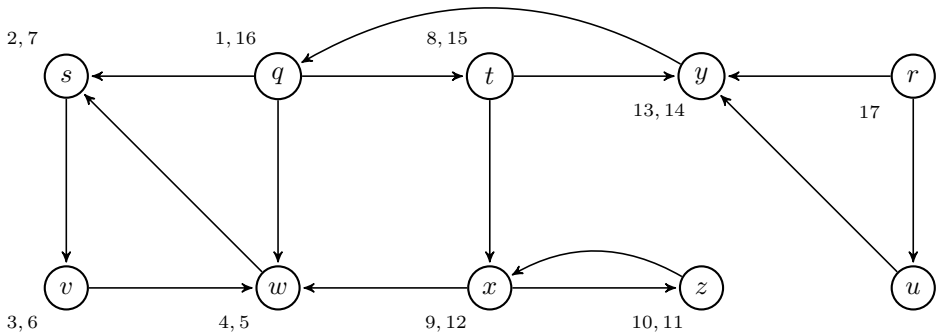
Zásobník obsahuje t, q (dno je vpravo).

(e) Pokračujeme do y (zásobník obsahuje y, t, q a pak jej vyprázdníme):



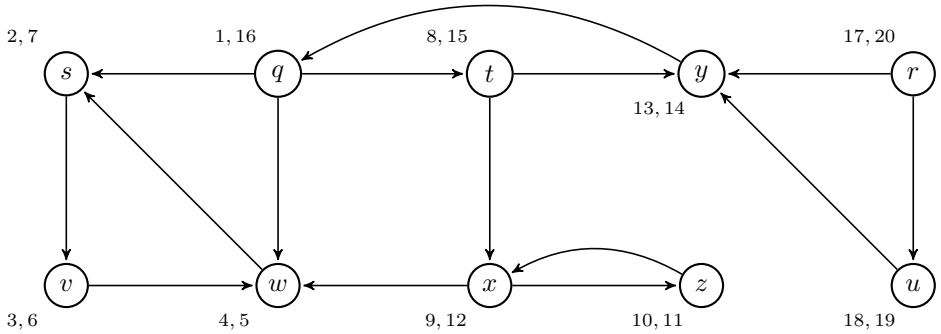
Zásobník je prázdný.

(f) Pokračujeme novým průchodem z r:

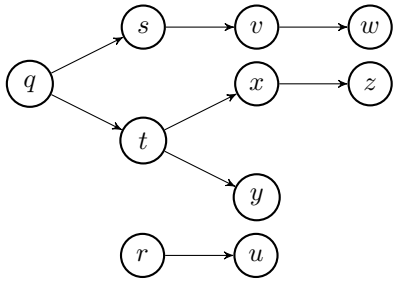


Zásobník obsahuje r.

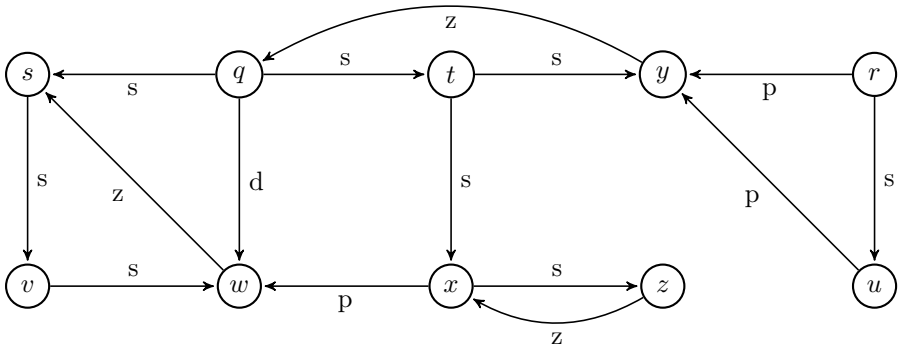
(g) Ukončujeme průchod z v, a jelikož mají všechny vrcholy časové známky, ukončujeme průchod:



DFS les vypadá takto:



Hrany klasifikujeme podle zvoleného lexikografického průchodu následovně:



d) Lze z časových známek získaných z průchodu v minulém příkladu zjistit, zda je z vrcholu u dosažitelný vrchol w? Jak obecně pouze z časových známek určíte, jaké vrcholy jsou dosažitelné ze zadaného vrcholu?

Z vrcholu u se lze dostat do vrcholu w, ale z časových známek to vyčíst nelze. Přesto můžeme z časových známek určit slabší tvrzení. Má-li vrchol a časové známky v intervalu časových známek vrcholu b, pak existuje cesta z b do a. Opačné tvrzení však neplatí.

Časové známky jsou závislé pouze na stromových hranách, jiné hrany v grafu být nemusí. Proto ze známek neurčíme podobu grafu.

Sem lze zařadit příklad 11.5 na pořadí průchodu uzlů v grafu pomocí DFS.

e) V kolika krocích navštívíte bod z z bodu q pomocí BFS a DFS? Jde se před započetím prohledávání rozhodnout, který typ prohledávání bude výhodnější?

BFS prozkoumá vrchol z jako 8. vrchol, DFS jako 7.

Obecně se před průchodem nezle rozhodnout, který průchod bude výhodnější. Můžete se rozhodovat jedině v případě, že máte nějaké další informace o grafu, který procházíte.

f) Porovnejte časovou i prostorovou složitost BFS a DFS. Jak se liší v závislosti na podobě grafu? Zadejte graf, ve kterém se složitosti liší.

Z pohledu časové složitosti oba algoritmy patří do $\mathcal{O}(|V| + |E|)$, protože musí prozkoumat všechny vrcholy a otestovat všechny hrany. Paměťová složitost se ale liší. U grafů s vrcholy s vysokým stupněm se nám hodně naplňuje fronta, grafy s dlouhými cestami jsou zase prostorově náročné na zásobník.

U prostorové složitosti se někdy zmiňuje, že DFS má lineární prostorovou složitost vůči délce nejdelší větve, zatímco BFS má exponenciální prostorovou složitost vůči větvení.

g) Jaký graf projdou průchody do šířky a do hloubky ve stejném pořadí?

Graf jednoduché cesty.

11.5

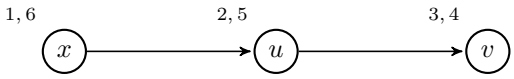
Časově náročné. Můžete ušetřit čas tím, že studentům řeknete, která tvrzení platí a která ne. Hledání protipříkladů je však kreativní.

Tvrzení je hodně, vyberte si ta, která vám přijdou zajímavá.

Mějme graf G, který obsahuje vrchol x, ze kterého jsou dostupné všechny vrcholy grafu G. Pro vrcholy u a v určíme časové známky DFS průchodem z vrcholu x. Která z následujících tvrzení jsou pravdivá, své rozhodnutí zdůvodněte?

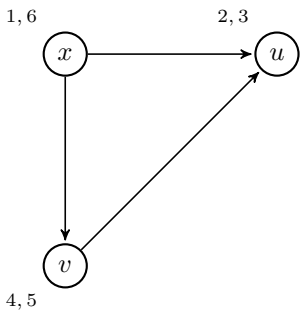
a) Pokud je $u.d < v.d$, pak neexistuje hrana z u do v.

Neplatí. Protipříklad:



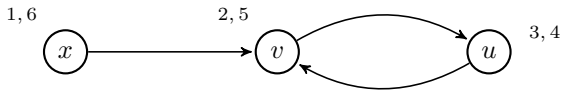
b) Pokud je $u.f < v.f$, pak neexistuje hrana z v do u.

Neplatí. Protipříklad:



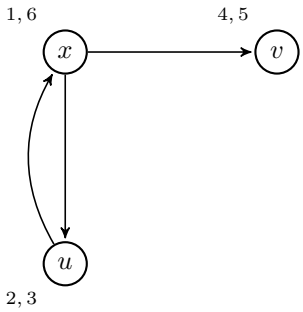
c) Pokud je $u.f < v.f$, pak neexistuje hrana z u do v.

Neplatí. Protipříklad:



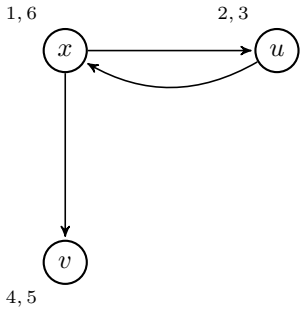
d) Pokud je $u.f < v.f \wedge u.d < v.d$, pak neexistuje cesta z u do v.

Neplatí. Protipříklad:



e) Pokud je $u.f < v.d$, pak neexistuje cesta z u do v.

Neplatí. Protipříklad:



f) Pokud je $u.f < v.d$, pak neexistuje hrana z u do v.

Platí. Nemohu ukončit procházení vrcholu u, pokud mám hranu do vrcholu v, který začnu procházet až někdy později.

11.6

Na počtu hran grafu záleží složitosti průchodů. Krátce se pobavte o možném počtu hran v grafu.

a) Jaký je maximální počet hran v orientovaném grafu s n vrcholy?

Graf s maximálním počtem hran obsahuje hranu mezi každými dvěma vrcholy. To odpovídá počtu kombinací dvojic vrcholů. Abychom vyjádřili počet pro orientovaný graf, musíme ještě počet vynásobit dvěma. Pak ještě musím přidat všechny smyčky nad vrcholy.

$$2 \cdot \binom{n}{2} + n = n^2$$

Alternativně na počet hran můžete přijít tak, že si zafixujete jeden vrchol. Z něj vedou hrany do všech. Pak je počet hran:

$$n \cdot n = n^2.$$

b) Jaký je minimální počet hran, pokud je celý orientovaný graf dosažitelný z jednoho vrcholu?

$n - 1$.

c) Kolik hran je třeba přidat, aby byl celý orientovaný graf silně souvislou komponentou?

Pokud hrany tvoří orientovanou cestu, pak stačí přidat 1 hrana pro uzavření cyklu. V nejhorším případě však všechny vrcholy mohou být ve vzdálenosti 1 od jednoho vrcholu. Pak musíme přidat ještě $n - 1$ hran.

Alternativně si můžete představit, že pomocí $n - 1$ hran můžete doplnit jednu hranu tak, aby vznikl cyklus.

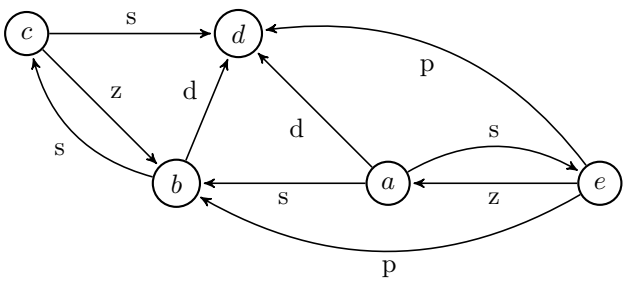
11.7

Nechte studenty soutěžit, kdo nejdřív na papír vymyslí graf splňující zadání. Kdo první graf vymyslí, ať ho nakreslí na tabuli a ostatní jej zkontrolují.

Navrhněte graf, který bude obsahovat po prohledání do hloubky:

- 2 dopředné hrany,
- 2 zpětné hrany,
- 2 příčné hrany a
- 4 stromové hrany.

Pořadí v rámci prohledávání je dáno abecedně.



11.8 Mějme strom minimálních vzdáleností z počátečního vrcholu, který vznikl průchodem BFS neorientovaného grafu (BFS strom). Co lze ze stromu určit o vzdálenosti 2 vrcholů, z nichž není ani jeden kořenem stromu (počátečním vrcholem)?

Vzdálenost zadaných vrcholů ve stromě nám dává horní odhad vzdálenosti vrcholů v grafu. Je tedy jisté, že vzdálenost v grafu je menší nebo rovna vzdálenosti v BFS stromě.

Rozdíl hloubky nám dává dolní odhad vzdáleností.

11.9

a) Kterým z algoritmů BFS nebo DFS byste hledali cyklus v orientovaném grafu a jak?

Nalezení cyklu odpovídá nalezení zpětné hrany. Můžeme tedy použít časové známky a použít je pouze k hledání zpětné hrany.

Jednodušší algoritmus je DFS s poznamenáváním aktuální procházené větve. Použijeme značku, která říká, že vrchol je zatím procházen. Když vrchol začneme procházet, nastavíme značku na *true*. Když končíme prozkoumávání, značku nastavíme zpět na *false*.

Graf obsahuje cyklus právě tehdy, když při prozkoumávání narazíme na vrchol, který má značku nastavenou na *true*.

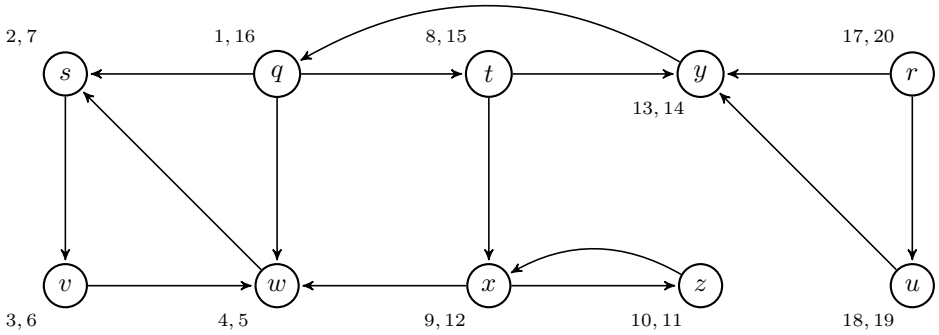
b) Navrhněte algoritmus, který určí délku nejkratšího cyklu v neohodnoceném orientovaném grafu. Pokud graf neobsahuje cykly, vrací ∞.

Dle předchozího najdu zpětné hrany (na každém cyklu nějaké je). Pro každou zpětnou hranu (u, v) už pak jen potřebuji najít nejkratší zbytek cyklu, tj. nejkratší cestu z v do u. K tomu mohu použít BFS z v.

11.10

Tento příklad udělejte algoritmicky, ne intuitivně. Studentům řekněte, ať si algoritmus udělají celý doma, zde je ušetřena práce s prvním průchodem na získání časových známek.

a) Určete silně souvislé komponenty v následujícím grafu:



Graf obsahuje následující silně souvislé komponenty (vypsány jsou v pořadí, v jakém je objeví algoritmus):

1. r
2. u
3. q, t, y
4. x, z
5. s, v, w

Část b) přeskočte, slouží jen jako přehled k domácímu studiu.

b) Jak byste hledali silně souvislé komponenty v grafu G?

Algoritmus můžeme provést v několika následujících krocích:

1. aplikuj DFS na G s uložením časových známek pro další využití,
2. vypočti transponovaný graf G^T,
3. aplikuj DFS na G^T tak, že v hlavním cyklu se vrcholy uvažují v pořadí od největší časové známky ukončení prozkoumávaní vrcholu z prvního kroku algoritmu a
4. vrcholy každého DFS stromu vypočteného při aplikaci DFS na G^T tvoří samostatné silně souvislé komponenty.

11.11

Tento příklad je hlavně pro nadanější studenty. Nečekejte příliš dlouho, než vám jej studenti vyřeší.

Dokažte, že z každého souvislého neorientovaného grafu, lze odebrat jeden vrchol tak, že nedojde k rozpojení grafu na samostatné části. Jak byste takový vrchol nalezli?

K řešení problému lze použít průchod DFS. Průchod nám vytvoří DFS strom, ze kterého lze zvolit libovolný list a bude o něm platit, že je vrcholem, který lze z původního grafu odstranit.

Obecněji lze odstranit libovolný list libovolné kostry grafu. DFS svým průchodem nalezne jednu z koster.

Vrchol můžeme z grafu odebrat právě tehdy, když po jeho odebrání existuje cesta mezi libovolnou dvojicí vrcholů. Jelikož jsme vzali vrchol, který je listem v kostře, musí každou dvojici vrcholů spojovat právě zbylá kostra.

U orientovaných grafů toto tvrzení neplatí, protipříkladem je orientovaná kružnice na 3 vrcholech, kde po odstranění lze oba vrcholy dosáhnout pouze z jednoho ze zbylých vrcholů.

11.12 U následujících algoritmů určete, zda se jedná o korektní DFS/BFS algoritmus. Pokud ne, popište proč a co dělá špatně. Předpokládejme, že všechny algoritmy voláme na neorientovaném grafu, který má u všech vrcholů nainicializovanou bílou barvu.

a) První algoritmus:

Procedura PruchodA(u)

vstup: vrchol u

```
1 u.color ← gray
2 for v ∈ u.successors do
3     if v.color = white then
4         PruchodA(v)
5     fi
6 od
7 u.color ← black
```

Jedná se o korektní DFS průchod, jen nepoznamenává časové známky (což není povinná funkcionalita DFS).

b) Druhý algoritmus:

Procedura PruchodB(u)

vstup: vrchol u

```
1 queue ← empty queue
2 Enqueue(queue, u)
3 while queue is not empty do
4     u ← Dequeue(queue)
5     for v ∈ u.successors do
6         Enqueue(queue, v)
7     od
8 od
```

Špatný algoritmus BFS. Nepoznamenává barvy vrcholů, takže pokud se v grafu nachází cyklus, tak se tento algoritmus zacyklí.

c) Třetí algoritmus:

Procedura PruchodC(u)

vstup: vrchol u

```
1 stack ← empty stack
2 Push(stack, u)
3 while stack is not empty do
4     u ← Pop(stack)
5     u.color ← gray
6     for v ∈ u.successors do
7         Push(stack, v)
8     od
9     u.color ← black
10 od
```

Algoritmus cyklí, jelikož opakovaně přidává již prozkoumané prvky do zásobníku. Je potřeba přidat kontrolu, že prvek, který chceme do zásobníku přidat již nebyl prohledáván. Výsledný

algoritmus je DFS bez časových známek, barvy slouží pouze k poznačení nalezených prvků.

Procedura PruchodC(u)

vstup: vrchol u

```
1  stack ← empty stack
2  Push(stack, u)
3  while stack is not empty do
4      u ← Pop(stack)
5      u.color ← black
6      for v ∈ u.successors do
7          if v.color = white then
8              Push(stack, v)
9          fi
10     od
11 od
```

d) Čtvrtý algoritmus:

Procedura PruchodD(G)

vstup: graf G

```
1  queue ← empty queue
2  Enqueue(queue, ∀u ∈ G)
3  while queue is not empty do
4      u ← Dequeue(queue)
5      u.color ← black
6      for v ∈ u.successors do
7          Enqueue(queue, v)
8      od
9  od
```

Algoritmus projde všechny vrcholy v pořadí, v jakém je při inicializaci vloží do fronty. Všechny další přidávání vrcholů do fronty už jsou zbytečné, vrcholy již budou v době Dequeue černé. Tímto způsobem tedy nelze provést DFS všech komponent grafu.

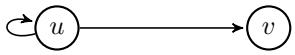
e) Pátý algoritmus:

Procedura PruchodE(u)

vstup: graf u

```
1  priorityQueue ← empty queue // uspořádaná podle abecedního pořadí vrcholů
2  Enqueue(priorityQueue, u)
3  while priorityQueue is not empty do
4      u ← Dequeue(priorityQueue)
5      u.color ← black
6      for v ∈ u.successors do
7          Enqueue(priorityQueue, v)
8      od
9  od
```

Použití prioritní fronty místo běžné fronty způsobí, že algoritmus není ani DFS ani BFS. Navíc algoritmus může cyklit, jednoduchým příkladem je graf:



11.13 Naprogramujte přidání hrany do matice sousednosti. Dále implementujte hledání nejkratší cesty z u do v pomocí BFS. Pomocí DFS otestujte, zdali graf obsahuje cykly. Také naprogramujte převod mezi reprezentacemi grafů. Můžete použít reprezentaci seznamů z předchozích cvičení. Postupujte podle pokynů v komentářích. Zdrojové kódy kódy jsou dostupné ve studijních materiálech: C a Python.

Následující příklady jsou vhodné na domácí studium.

11.14

Tento příklad byl jedním z implementačních testů z roku 2014.

Hraně neorientovaného grafu jejímž odstraněním rozdělíme graf na 2 nezávislé komponenty říkáme most. Navrhněte algoritmus, který nalezne most v grafu, nebo určí, že graf most neobsahuje.

Tento příklad byl zadáním jednoho z loňských implementačních testů. Naleznete jej ve studijních materiálech jako soubory se jménem `bridge`.

Tento příklad je opakováním přednášky. Detailněji popisuje pojmy z úvodu cvičení.

11.15 Jaké reprezentace grafů znáte? Jaké jsou jejich nevýhody a výhody? Kdy se která reprezentace hodí? Jak se vaše navržené reprezentace změní u ohodnoceného grafu?

Graf můžeme reprezentovat vícero způsoby:

1. Matice sousednosti je matice rozměrů $|V| \times |V|$, kde V je počet vrcholů v grafu. V matici M každá pozice M_{ij} vyjadřuje, zdali mezi i-tým a j-tým vrcholem existuje hrana (1 pokud hrana existuje, 0 jinak). Výhodou reprezentace maticí je konstantní časová složitost zjištění jestli jsou 2 vrcholy spojené hranou. Také se v některých algoritmech používá reprezentace maticí pro operace nad grafem (například násobení matic pro hledání nejkratších cest). Její nevýhodou je paměťová složitost, která je kvadratická vůči počtu vrcholů. Proto je výhodné matici používat jen na grafy s mnoha hranami. Další nevýhodou může být, že při počátku zpracovávání grafu nevíme, kolik vrcholů graf obsahuje.

   U ohodnoceného grafu si můžeme představit, že do matice budeme namísto 0 a 1 ukládat ceny konkrétních hran mezi vrcholy, pro reprezentaci, že hrana neexistuje můžeme zvolit nekonečno.

   Pro reprezentaci neorientovaného grafu nám stačí jenom trojúhelníková matice, protože bude symetrická kolem diagonály.

2. Další známá reprezentace je pomocí seznamu následníků. Reprezentujeme graf tak, že máme pole vrcholů a každému z nich přiřadíme provázaný seznam následníků (sousedních vrcholů).

Výhodou této reprezentace je, že ukládá do paměti jenom ty hrany, které v grafu existují, tedy má menší paměťové nároky než reprezentace maticí. Proto se využívá při reprezentací grafů, které mají menší počet hran. Nevýhodou je naopak zjišťování, zdali 2 vrcholy spolu sousedí, což lze provést v lineárním čase vůči počtu následníků daného vrcholu.

11.16 Mějme úplný binární strom hloubky 7 reprezentovaný pomocí seznamu následníků. Převeďte jej do reprezentace pomocí matice sousednosti.

11.17 Čtverec orientovaného grafu $G = (V, E)$ je graf $G^2 = (V, E^2)$ takový, že $(u, v) \in E^2$ právě tehdy, když G obsahuje cestu s maximálně dvěma hranami mezi u a v. Navrhněte efektivní algoritmus, který vytvoří graf G^2 z grafu G pro obě reprezentace – seznam následníků a matice sousednosti. Analyzujte složitost vašeho algoritmu.

11.18 Matice sousednosti orientovaného grafu $G = (V, E)$ bez smyček je $|V| \times |E|$ matice $B = b_{ue}$ taková, že:

$$b_{ij} = \begin{cases} -1 & \text{pokud hrana e směřuje z vrcholu u,} \\ 1 & \text{pokud hrana e směřuje do vrcholu u,} \\ 0 & \text{jinak.} \end{cases}$$

Popište, co bude reprezentovat výstupní matice produktu BB^T, kde B^T je transponovaná matice B.

11.19

a) Strom je jednoduchý souvislý graf, který neobsahuje kružnice. Popište souvislost pre/in/post order výpisu s DFS průchodem.

Výpisy pre/in/post order využívají průchodu stromu do hloubky. Liší se pouze v pozici výpisu klíče v uzlu. Pokud vypíšeme před zanořením, jedná se o preorder, pokud mezi zanořeními, pak se jedná o inorder a pokud až po obou zanořeních, pak se jedná o postorder.

Procedura PreorderDFS(u)

vstup: vrchol u – kořen stromu/podstromu

```
1 if u = nil then
2     return
3 fi
4 vypiš u.key
5 PreorderDFS(u.left)
6 PreorderDFS(u.right)
```

Zbylé průchody vypadají podobně, jen se liší v pozici příkazu vypiš $u.key$.

b) Lze preorder výpis použít na graf za předpokladu, že by každý uzel měl maximálně 2 následníky a to pod ukazateli LEFT a RIGHT?

Nelze. DFS algoritmus se od našeho průchodu stromem liší v tom, že testuje, zdali jsme již daný vrchol nenavštívili. To však u výpisu netestujeme, protože u stromů se nám to nemůže stát. Jakmile však graf obsahuje cykly, i náš výpis by se zacyklil. Museli bychom tedy přidat kontrolu, zdali vrchol, do kterého se chceme zanořit, již není v zásobníku (k takové kontrole však zásobník vůbec není vhodná datová struktura).

c) Jakým průchodem je vhodné procházet nekonečný (ale spočetný) strom konečné arity (strom, kde existuje alespoň jedna nekonečná (ale spočetně dlouhá) větev)?

Daliborek vzkazuje: Existenci nekonečně dlouhé větve v nekonečném grafu konečné arity máme zaručenu Königovým lemmatem (které však nelze dokázat v Zermelově-Fraenkelově teorii množin bez axiomu výběru).

Průchod do šířky. DFS by se na nekonečné větvi zaseklo, zatímco BFS alespoň projde všechny konečné větve. V praxi bychom však nastavili jak pro BFS, tak pro DFS maximální hloubku zanoření (omezením velikosti zásobníku/fronty, podle dostupné paměti).

d) ** Jakým průchodem je vhodné procházet nekonečný (ale spočetný) strom nekonečné (ale spočetné) arity?

Karlík varuje: Přemýšlení nad problémy reálného života pomocí DFS často nevede ke správnému výsledku: viz xkcd.

Algoritmus by iteroval hodnotu $i \in \mathbb{N}$. Podle aktuální hodnoty i by prozkoumal i-tou hranu prvního uzlu, v první větvi by se zanořil o i kroků. Obecně by pro každé i prozkoumal všechny hodnoty ve vzdálenosti i, kde vzdáleností máme na mysli součet délky cesty od kořene a počet hodnot, které jsme prozkoumávali ve všech uzlech cesty.

Navržený algoritmus je inspirován metodou „dove tailing“.

11.20 Navrhněte graf, který po libovolném prohledání do hloubky nebude obsahovat žádnou dopřednou hranu.

11.21 Kolik existuje různých grafů s n vrcholy a m neorientovanými hranami?

11.22 Použijme algoritmus BFS, pouze u něj nahraďme frontu zásobníkem. Nalezne algoritmus stále nejkratší cesty v grafu?

11.23 Excentricita vrcholu v je nejdelší vzdálenost z v do jiného vrcholu grafu. Průměr grafu je největší excentricita jeho vrcholů. Naopak nejmenší excentricita vrcholů je poloměr. Centrum je vrchol, jehož excentricita je rovna poloměru.

Navrhněte funkce, které určí excentricitu, průměr, poloměr a centrum.

11.24 Eulerovský tah je posloupnost neopakujících se hran, kterými procházíme zadaný graf. Hamiltonovský cyklus je cesta, která prochází přes všechny vrcholy.

Určete, kdy graf může mít Eulerovský tah a Hamiltonovský cyklus.

Jak byste problém řešili algoritmicky?

11.25 Dokažte, že graf obsahující kružnici liché délky nemůže být bipartitní. Platí také tvrzení, že každý graf, který obsahuje pouze cykly sudé délky je bipartitní?

11.26 Mějme obrázek, ve kterém chceme ze zadaného pixelu najít všechny pixely, které mají stejnou barvu a sousedí buďto s počátečním pixelem, nebo pixelem, který je již v množině sousedů.

Jak budete reprezentovat obrázek grafem a jaký grafový algoritmus se k problému hodí?

11.27 Jaká bude složitost algoritmu BFS, pokud použijeme pro reprezentaci grafu matici sousedů? Modifikujte algoritmus tak, aby byl schopen pracovat s maticí jako vstupem.

11.28 Existují dva typy wrestlerů: „babyfaces“ („good guys“) a „heels“ („bad guys“). Mezi každou dvojicí profesionálních wrestlerů je či není rivalita. Předpokládejme, že máme n profesionálních wrestlerů a máme seznam r párů rivalů. Navrhněte algoritmus, který v čase $\mathcal{O}(n + r)$ rozhodne, zdali je možné přiřadit wrestlerům typ „babyface“ a ostatním typ „heel“ tak, že všechny dvojice rivalů jsou dvojice „babyface“ a „heel“ wrestlerů. Pokud takové přiřazení existuje, váš algoritmus by měl toto přiřazení vrátit.

11.29 Nechť $G = (V, E)$ je souvislý, neorientovaný graf. Navrhněte algoritmus, který v čase $\mathcal{O}(V + E)$ spočítá cestu v G, která projde každou hranu právě jednou v každém směru. Představte si problém jako bludiště, jak byste našli cestu ven, pokud byste mohli použít neomezený počet mincí na značení cesty?

11.30 Mějme algoritmus BFS, který nepoužívá šedou barvu pro indikaci, že již je vrchol přidán do fronty. Jak se změní složitost algoritmu.

▌ Na frontu vložíme až $|E|$ vrcholů.

Kapitola 12

Grafy II.

Cesta z u do v je uspořádaná posloupnost neopakujících se vrcholů. Začíná v u a končí ve v, přičemž její délka je součet délek hran, které se na cestě nachází. Délku nejkratší z cest značíme $\delta(u, v)$.

Trojúhelníková nerovnost – pro každou hranu $(u, v) \in E$ a $\forall(s) \in V$ platí $\delta(s, v) \leq \delta(s, u) + w(u, v)$, kde $w(u, v)$ je ohodnocení hrany (u, v).

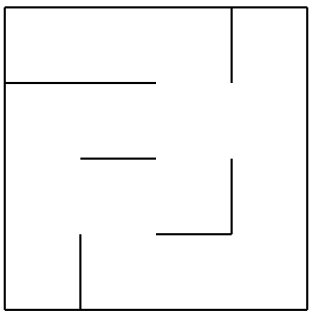
Bellmanův–Fordův algoritmus je algoritmus pro hledání nejkratších cest ze zadaného vrcholu do všech ostatních vrcholů. Je schopen pracovat s grafem s hranami záporné délky a v případě existence záporného cyklu jej umí detekovat.

Dijkstrův algoritmus je algoritmus pro hledání nejkratších cest ze zadaného vrcholu do všech ostatních vrcholů. Oproti Bellmanovu–Fordovu algoritmu neumí projít graf s hranami záporné délky. Je odvozen z průchodu do šířky. Výhodou oproti Bellmanovu–Fordovovu algoritmu je lepší časová složitost.

Relaxace cesty je procedura volaná na dvojici vrcholů, která v případě existence kratší cesty, než kterou zatím známe, aktualizuje vzdálenost vrcholu na novou kratší hodnotu.

Strom nejkratších cest grafu G definujeme jako strom, kde od fixního kořene v k libovolnému vrcholu u je cesta ve stromě nejkratší cestou v grafu G.

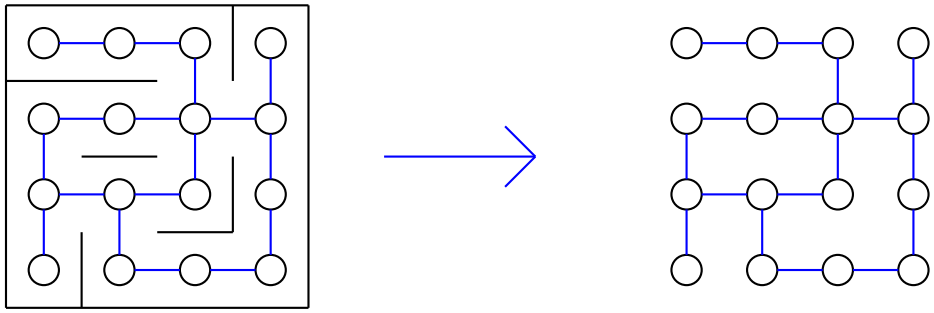
12.1 Mějme následující bludiště.



Můžete studenty nechat vytvořit i ohodnocený graf, potom však budou muset řešit části b) a c) jinak. Zamýšlené řešení by alespoň měli vidět.

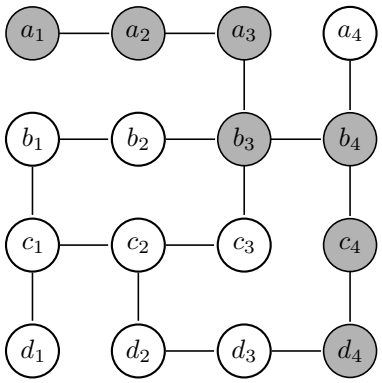
a) Převeďte bludiště na graf.

Graf vybudovaný z bludiště vypadá takto:



b) Navrhněte algoritmus pro nalezení nejkratší cesty v bludišti z levého horního rohu do pravého dolního rohu. Pokud cesta neexistuje, vrátí *false*. Algoritmus proveďte.

Na graf můžeme použít BFS. Cesta má délku 6 a vypadá následovně:

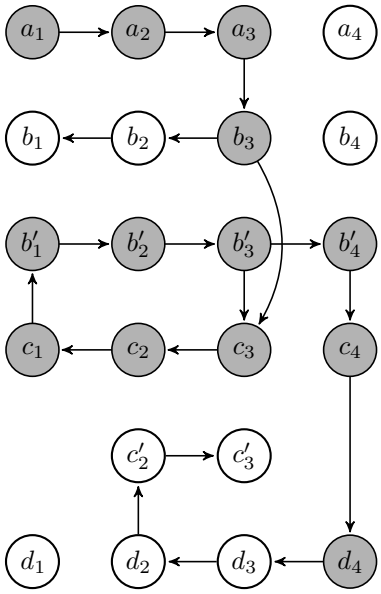


c) Upravte graf tak, aby v něm nešlo odbočovat doleva a vracet se. Co vrátí váš algoritmus po této úpravě?

Graf musíme předělat na orientovaný. Dále se nám některé uzly rozdělí, mají totiž jiné sousedy podle směru, ze kterého do nich přicházíme.

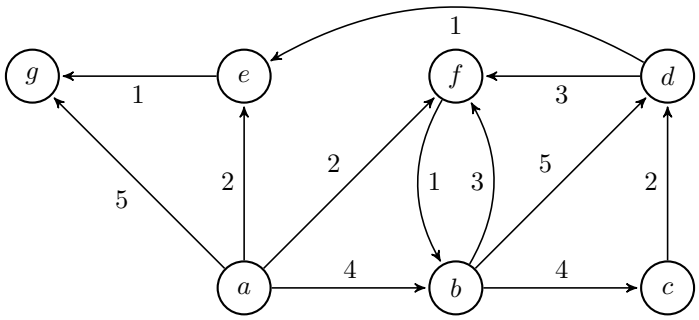
V následujícím grafu jsou rozděleny uzly řady b a c. Když se do nich dostáváme poprvé, mají původní popis. Když se opakují, používáme popis b' a c'.

Také si všimněte, že vrcholy a_4, b_4 a d_1 jsou v novém grafu nedostupné.

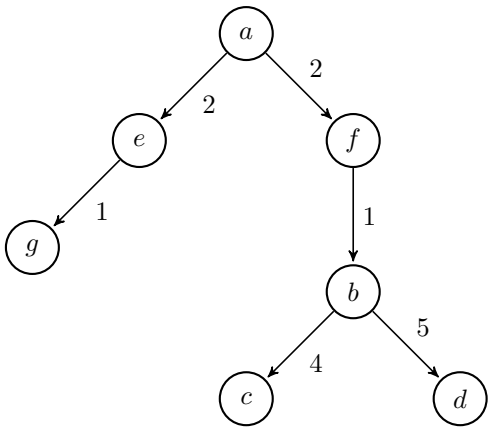


12.2

a) Zkonstruujte pomocí Bellmanova–Fordova algoritmu strom nejkratších cest z vrcholu a v následujícím grafu. Jaká je délka cesty z a do c?



Cesta má délku 7, strom nejkratších cest vypadá následovně:



b) Jaká je časová složitost Bellmanova–Fordova algoritmu?

Složitost je v $\mathcal{O}(|E| \cdot |V|)$, což lze určit ze 2 cyklů, které iterují nad vrcholy a hranami. Složitost Bellmanova–Fordova algoritmu je vyšší než u Dijkstrova algoritmu, ten však neumí zpracovávat graf se zápornými hranami. Další výhodou Bellmanova–Fordova algoritmu je poměrně snadná paralelizovatelnost (lze relaxovat naráz podle všech hran).

c) Jak pomocí Bellmanova–Fordova algoritmu určíte, že graf obsahuje cyklus záporné délky?

Pro všechny hrany grafu provedeme kontrolu, zda je nalezená vzdálenost jejich konce menší než vzdálenost jejich začátku plus délka hrany (pro hranu (u, v): $v.d \leq u.d + \delta(u, v)$).

d) Modifikujte algoritmus Bellman-Ford tak, že algoritmus vrátí $v.d = -\infty$ pro všechny vrcholy v, pro které existuje cyklus se zápornou délkou na cestě z počátečního vrcholu do v.

Použijeme detekci cyklů se zápornou délkou. Vždy, když algoritmus takový cyklus nalezne, projdeme všechny vrcholy, které lze dosáhnout od vrcholu na cyklu a vrcholům přiřadíme vzdálenost $v.d = -\infty$. Takto hledáme všechny výskyty cyklů.

12.3 V tabulce máme přehled nejkratších cest mezi dvojicemi českých vesnic. Navrhněte algoritmus, který nalezne v tabulce chyby. (Nápověda: graf reprezentující silniční síť musí splňovat trojúhelníkovou nerovnost).

	Peklo	Ráj	Hrob	Onen Svět	Záhrobí
Peklo		149	223	197	230
Ráj	150		84	129	139
Hrob	222	84		265	165
Onen Svět	197	129	264		41,4
Záhrobí	230	139	164	41,4	

Mohou vám jako chybné přijít údaje, že cesta $a \to b$ je jinak dlouhá, než $b \to a$. To je dáno tím, že silniční síť je orientovaný graf.

Chybná informace v tabulce je délka cest z Hrobu na Onen Svět. Trasa má mít 165 km (zpět 164 km). Chybu nalezneme relaxací zmíněné cesty. Pokud se rozhodneme cestovat přes Záhrobí, bude cesta Hrob – Záhrobí + Záhrobí – Onen Svět měřit pouze 165+41,4=206,4 km.

Algoritmickým řešením úkolu je spustit algoritmus Bellman-Ford ze všech vesnic. Při tom byste kontrolovali, zdali se některá ze vzdáleností nezměnila. Pokud se žádná nezmění, pak je tabulka v pořádku. Takové řešení odpovídá přístupu, kdy pro každou trojici vrcholů provedete relaxaci. Pokud by některá z relaxací změnila délku cesty, nalezli jste špatný údaj.

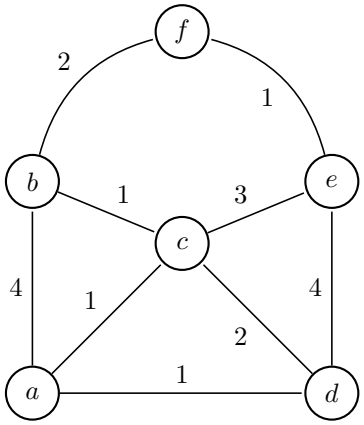
Tabulka bez chyb vypadá takto:

	Peklo	Ráj	Hrob	Onen Svět	Záhrobí
Peklo		149	223	197	230
Ráj	150		84	129	139
Hrob	222	84		206,4	165
Onen Svět	197	129	205,4		41,4
Záhrobí	230	139	164	41,4	

12.4

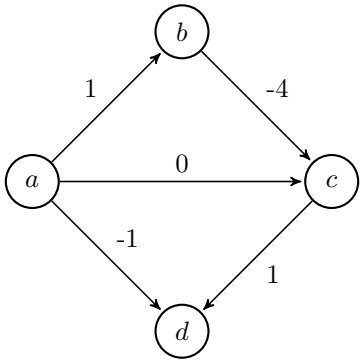
Pokud máte čas, můžete příklad řešit detailně s kreslením prioritní fronty v podobě haldy. Zopakujete studentům znalosti ze starší kapitoly.

a) Nalezněte pomocí Dijkstrova algoritmu nejkratší cestu z vrcholu a do vrcholu f v tomto neorientovaném grafu:



Nejkratší cesta vede přes vrcholy a, c, b a f a má délku 4.

b) Projděte následující graf z bodu a pomocí Dijkstrova algoritmu.

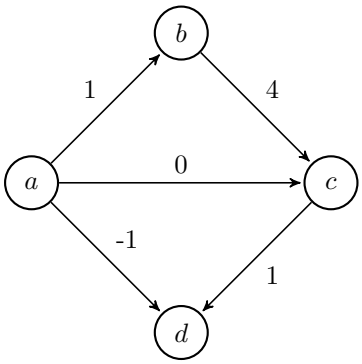


Dijkstrův algoritmus nefunguje na grafy se záporně ohodnocenými hranami. V grafu se záporně ohodnocenými hranami totiž může vzdálenost vrcholu od počátku klesat, což ale není v souladu s tím, že zpracováváme vrchol tehdy, kdy je jeho vzdálenost z počátku minimální.

V našem případě jako první po a uzavřený vrchol označíme d se vzdáleností -1. Pak stejně uzavřeme vrchol c se vzdáleností 0. Nakonec uzavřeme vrchol b se vzdáleností 1, ale pokud bychom z tohoto vrcholu pokračovali, tak objevíme vrchol c ve vzdálenosti -3 a vrchol d ve vzdálenosti -2, což jsou menší hodnoty, než se kterými jsme vrcholy uzavřeli.

Pro hledání nejkratší cesty se záporně ohodnocenými hranami se používá Bellmanův–Fordův algoritmus, cenou je však větší časová složitost.

c) Navrhněte graf s hranami záporné délky, které Dijkstrův algoritmus zpracuje, a přesto vrátí správný výsledek.



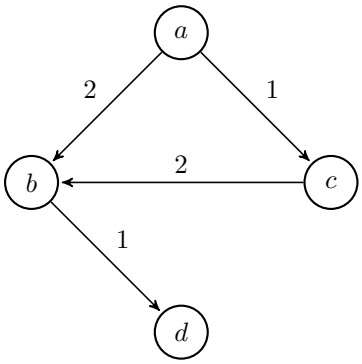
Obecně lze mít záporné hrany do vrcholů, ze kterých nevede žádná hrana. Také lze mít záporné hrany u vrcholů, které ještě nemají zpracován žádného následníka.

d) Co se stane s Dijkstrovým algoritmem, pokud místo prioritní fronty použijeme normální frontu? Na jakých grafech bude fungovat?

Algoritmus se nám změní v BFS (které zbytečně relaxuje hrany). Pokud je graf neohodnocený (ohodnocený konstantou), pak bude takto upravený Dijkstrův algoritmus fungovat.

12.5 Předpokládejme, že chceme vyřešit problém nejdelší cesty mezi dvěma vrcholy. Co dělá algoritmus Dijkstra, pokud zaměníme operaci *minimum* za *maximum*? Pokud bude korektně hledat délky nejdelších cest, pak to dokažte. Pokud ne, tak sestavte protipříklad.

Změna operace nevede ke korektnímu algoritmu pro hledání nejdelší cesty. Protipříklad (hledání cesty z a do d):



Takto upravený algoritmus by našel cestu $a - b - d$ s délkou 3, maximální cesta však je $a - c - b - d$ délky 4. Problém je v tom, že po vyjmutí vrcholu z prioritní fronty vrchol již dále nezpracováváme.

Pokud by algoritmus fungoval, pak by redukcí řešil problém Hamiltonovské kružnice, který je NP-těžký.

12.6 Jak byste řešili následující problémy hledání cest:

Část a) se řeší v celé počáteční části cvičení, můžete to jen zmínit a jít na část b).

a) Nejkratší cesta z jednoho vrcholu do všech ostatních vrcholů.

> Je zobecnění hledání nejkratší cesty mezi dvěma vrcholy. Pro tento účel se používají zase Dijkstrův nebo Bellmanův–Fordův algoritmus.

b) Nejkratší cesta, která přechází přes konkrétní vrcholy v daném pořadí.

> Problém si dokážeme rozbít na podproblémy. Třeba pokud hledáme cestu z a do b přes c, pak můžeme řešit nejkratší cestu z a do c (na grafu bez b), následně z c do b (na grafu bez a) a výsledky spojit.

c) Nejkratší cesty ze všech vrcholů do jednoho vrcholu.

> Problém můžeme redukovat na problém hledání nejkratších cest z jednoho vrcholu do všech ostatních tak, že náš cíl převedeme na počátek cesty a všechny orientace hran obrátíme, čímž můžeme využít řešení z příkladu a).

d) Identifikování vrcholů do zadané vzdálenosti (hledání měst v určitém okolí na mapě).

> Problém lze řešit pomocí Dijkstrova algoritmu. Neprovádíme jej do nějakého vrcholu, ale do té doby, dokud bude prioritní fronta obsahovat vrcholy v zadané vzdálenosti. Jakmile překročí vzdálenost, algoritmus ukončíme a nalezené vrcholy jsou v zadaném okolí.

e) Nalezení nejdelší cesty v acyklickém grafu.

> Stačí na to algoritmus pro hledání nejkratší cesty, který umí zpracovat graf se záporně ohodnocenými hranami. Záporné cykly nás neohrozí, protože máme acyklický graf.
>
> Když máme takový algoritmus, stačí nám ohodnocení hran převést na negaci (hranám přiřadíme jejich opačnou délku) a algoritmus nechat hledat nejkratší cestu. Výstup musíme zase znegovat zpět.

f) * Nejkratší cesty mezi všemi dvojicemi vrcholů.

> Lze provést V-krát hledání nejkratších cest do všech vrcholů z jednoho vrcholu. K tomu můžeme použít Bellmanův–Fordův algoritmus.Alternativou jsou algoritmy pro hledání všech cest v grafu – Floydův–Warshallův a nebo Jonsnův algoritmus.

g) * Nejkratší cyklus přes všechny vrcholy.

> Tento problém, také známý jako problém obchodního cestujícího, patří do třídy NP-těžkých problému, což znamená, že jej nedokážeme řešit v polynomiálním čase. Nejjednodušší řešení je vyzkoušet všechny cesty. Pro problém existují i různé heuristiky, které snižují čas výpočtu problému o nějaké konstanty.

12.7 Použijte Dijkstrův algoritmus k nalezení nejkratší cesty z více výchozích vrcholů do všech ostatních vrcholů v orientovaném grafu, který neobsahuje záporné hrany.

Do grafu přidáme nový vrchol, který bude iniciální vrchol k prohledávání a z něj vedeme hrany délky 0 do všech původně iniciálních vrcholů.

12.8 Navrhněte algoritmus k nalezení nejkratší cesty, která je rostoucí. Rostoucí cesta musí obsahovat hrany, jejichž délky tvoří rostoucí posloupnost.

Řešením je upravit graf tak, že si ve vrcholu pamatuji i příchozí hranu. Zvětším si takto množinu vrcholů podobně jako v příkladu o zákazu odbočování vlevo. Podle hodnoty příchozí hrany pak umažu výchozí hrany, jejichž ohodnocení je menší nebo rovno ohodnocení příchozí hrany.

12.9

Pseudokódy, tradičně doporučujeme vynechat a přejít na programování.

Příklad zaměřen na pseudokódy Bellmanova–Fordova a Dijkstrova algoritmu.

a) Jak provádíme inicializaci algoritmů, které hledají nejkratší cesty z jednoho zdroje?

Při hledání nejkratších cest si udržujeme pro každý vrchol informace o horním odhadu délky cesty ze zdroje až k danému vrcholu ($v.d$), při inicializaci nastavený na ∞. Dále udržujeme i informaci pro každý vrchol o jeho předchůdci v hledané cestě ($v.\pi$), pokud předchůdce neexistuje je nastavený na *nil*.

Procedura Initialize(V, s)

vstup: V seznam vrcholů, s iniciální vrchol

```
1 for v ∈ V do
2     v.d ← ∞
3     v.π ← nil
4 od
5 s.d ← 0
```

b) Relaxace hrany (u, v) slouží jako test, zdali existuje kratší cesta do vrcholu v přes vrchol u. Navrhněte metodu Relax s argumenty vrcholů u a v. Nezapomeňte na udržení informace o předchůdci.

Operace Relax vyhodnotí, zdali napočítaná cesta do v je delší než cesta do u + hrana (u, v).

Procedura Relax(u, v)

vstup: vrcholy u a v

```
1 if v.d > u.d + w(u, v) then
2     v.d ← u.d + w(u, v)
3     v.π ← u
4 fi
```

c) Pomocí předchozích metod zkonstruujte algoritmus Bellman–Ford. Tedy inicializujte algoritmus, použijte relaxaci hran a nezapomeňte ověřit, jestli graf obsahuje záporné cykly.

Řešení:

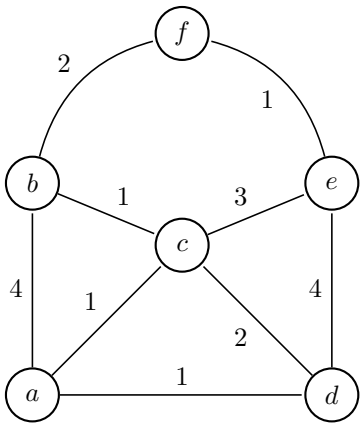
Procedura Bellman–Ford(G, s)

vstup: graf $G = (V, E)$ a počátek cesty s

```
1  Initialize (V, s)
2  for i ← 1 to |V| − 1 do
3      for každou hranu (u, v) ∈ E do
4          Relax(u, v)
5      od
6  od
7  for každou hranu (u, v) ∈ E do
8      if v.d > u.d + w(u, v) then
9          return false // obsahuje záporný cyklus
10     fi
11 od
12 return true
```

Dále se příklad věnuje Dijkstrovu algoritmu včetně úvah o jeho odvození.

d) Přepište hrany délky x na x hran přes pomocné vrcholy tak, aby zůstala délka cest zachována a na tomto grafu proveďte průchod do šířky. Ten ukončete při nalezení vrcholu f. Co jste zjistili?



BFS se zastavilo v hloubce 4 a nalezlo nejkratší cestu z a do f.

e) Kdy BFS prohledává původní vrcholy? Dá se vrcholům přiřadit nějaká metrika, podle které bychom mohli provést BFS na ohodnoceném grafu?

BFS prochází původní vrcholy právě ve chvíli, kde je zanořeno v hloubce odpovídající délce nejkratší cesty z vrcholu a do zkoumaného vrcholu. Vzdálenost se rovná právě hloubce zanoření. Pokud tedy během průchodu do šířky přiřazujeme při nalezení vrcholům aktuální hloubku BFS, vytváříme strom minimálních cest.

f) Kdy bude BFS na grafu s pomocnými vrcholy zpracovávat jaký vrchol?

Zpracováváme vrchol, který má nejmenší vzdálenost z počátku a není již zpracován.

g) Převod grafu s ohodnocenými hranami na graf s hranami délky 1 pomocí přidání vrcholů tedy umožňuje procházení (kladně) ohodnoceného grafu pomocí BFS. Je nalezená vzdálenost vrcholu finální po prvním nalezení vrcholu, nebo se může měnit? Zamyslete se nad tím v kontextu obarvování vrcholů.

Vzdálenost se nemůže měnit. Při prvním nalezení vrcholu musí být nutně nalezena nejkratší cesta k němu. Nicméně hrany, které jsou rozděleny pomocnými vrcholy můžeme začít prozkoumávat dříve, než k nim najdeme cestu přes jiné vrcholy. Příkladem je vrchol b, který nalezneme z vrcholu c a ještě se poté vracíme pomocí BFS kus hrany $\{a, b\}$ směrem do a.

h) Zkuste formulovat Dijkstrův algoritmus, pomocí úvah o BFS. Můžete k tomu použít funkce Initialize a Relax, které jste použili u Bellmanova–Fordova algoritmu.

Procedura Dijkstra(G, s)

vstup: graf $G = (V, E)$ a počátek cesty s

```
1  Initialize(V, s)
2  Q ← V // Q je množina vrcholů pro zpracování
3  while Q ≠ ∅ do
4      u ← ExtractMin(Q)
5      for každou hranu (u, v) ∈ E do
6          if v.d > u.d + w(u, v) then
7              v.d ← u.d + w(u, v)
8              v.π ← u
9              DecreaseKey(Q, v, v.d)
10         fi
11     od
12 od
```

i) Jaký je vhodný způsob výběru vrcholu, který máte zpracovat? Jaká datová struktura se k tomuto účelu hodí? Jaká bude složitost algoritmu podle zvolené datové struktury?

Minimová halda je datová struktura, která má dobrou rychlost operace DecreaseKey pro relaxaci a ExtractMin pro výběr vrcholu, který je v minimální vzdálenosti a můžeme jej zpracovat.

Oproti haldě, která operace realizuje v logaritmickém čase, seznam a pole mají lineární složitost alespoň jedné z operací.

Paní Bílá připomíná: Ještě vhodnější pomocnou strukturou pro Dijkstrův algoritmus je Fibonacciho halda, která podporuje DecreaseKey v konstantním amortizovaném čase.

Jelikož algoritmus projde všechny hrany a v každém průchodu while cyklu musí vybrat minimální vrchol, bude složitost algoritmu v $\mathcal{O}(|E|+|V|\cdot|V|)$ při použití pole nebo seznamu pro ExtractMin a $\mathcal{O}(|E| + |V| \cdot \log |V|)$ při požití haldy (prioritní fronty).

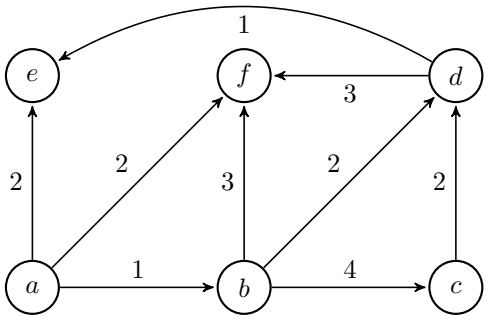
12.10 Naprogramujte hledání nejkratších cest v orientovaném ohodnoceném grafu pomocí Bellmanova–Fordova a Dijkstrova algoritmu. Pro Dijkstrův algoritmus použijte připravenou strukturu prioritní fronty. Postupujte podle pokynů v komentářích. Zdrojové kódy jsou dostupné ve studijních materiálech: C a Python.

Následující příklady jsou vhodné na domácí studium.

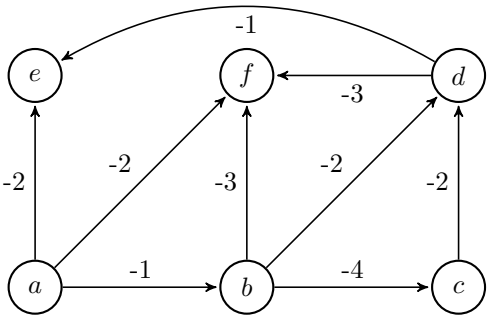
12.11

Pozor, algoritmus probraný v tomto příkladu funguje pouze pro orientované acyklické grafy. Obecně je hledání nejdelší cesty NP problém.

a) Určete nejdelší cestu z vrcholu a na tomto grafu:



Upravený graf:



Nejkratší cesta v tomto grafu je posloupností a, b, c, d a f s délkou -10. To znamená, že v původním grafu byla nejdelší cesta na stejné posloupnosti vrcholů a měla délku 10.

Pokud bychom však úkol jen trochu modifikovali tím, že bychom pracovali s neohodnoceným neorientovaným grafem a ptali bychom se, zdali obsahuje kružnici přes všechny vrcholy, jednalo by se o problém hamiltonovské kružnice, ke kterému neexistuje algoritmus řešící jej v polynomiálním čase.

Paní Bílá připomíná: Pro výpočet minimální kostry libovolného grafu je možné použít hladové algoritmy, protože pro graf můžeme sestrojit odpovídající matroid, jehož báze jsou tvořeny množinami hran koster tohoto grafu.

12.12 Mějme ohodnocený orientovaný graf $G = (V, E)$ s cyklem se zápornou délkou. Navrhněte efektivní algoritmus, který vypíše vrcholy téhož cyklu. Dokažte, že váš algoritmus je korektní.

12.13 Ukažte, že strom nejkratších cest z jednoho zdroje vytvořený pomocí algoritmu Dijkstra v neorientovaném grafu netvoří nutně minimální kostru.

12.14 Nacházíte se v n dimenzionální síti na pozici $(x_1, x_2, \ldots, x_n)$. Dimenze sítě jsou $(d_1, d_2, \ldots, d_n)$. V jednom kroku můžete udělat jeden krok dopředu nebo dozadu v kterékoliv z n dimenzí (tedy vždy existuje $2 \cdot n$ možných pohybů). Kolika způsoby můžete udělat m kroků tak, že nikdy neopustíte síť v žádném bodu? Síť opustíte pokud pro nějaké x_i platí, že $x_i \leq 0 \vee x_i > D_i$.

12.15 Dokažte, že v grafu s unikátními váhami hran existuje právě jedna minimální kostra. Kolik minimálních koster lze najít v grafu s hranami stejné vzdálenosti?

12.16 Jak se změní strom nejkratších cest do všech vrcholů z předem daného vrcholu, když obrátíme všechny hrany?

12.17 Jak byste našli průměr ohodnoceného orientovaného grafu?

12.18 Co se stane s Bellmanovým–Fordovým algoritmem, pokud je na cestě mezi zadanými dvěma vrcholy negativní cyklus? Bude některá hrana nekorektně relaxovaná?

12.19 Jaký algoritmus byste použili pro nalezení nejkratšího cyklu v grafu?

12.20 Mějme orientovaný graf bez záporných hran. Vrcholy tohoto grafu jsou rozděleny do dvou množin. Nalezněte nejkratší cestu mezi všemi dvojicemi vrcholů v různých množinách.

12.21 Navrhněte algoritmus k nalezení sousedů v určité vzdálenosti od zadaného vrcholu. Algoritmus by měl mít časovou složitost závislou na zadané vzdálenosti.

12.22 Navrhněte algoritmus k nalezení kritické hrany v orientovaném grafu bez záporných hran. Kritická hrana je hrana, jejíž odstranění maximálně zvětší délku nejkratší cesty mezi dvěma vrcholy.

12.23

a) Co bude počítat Bellmanův–Fordův algoritmus, pokud neprovedeme relaxaci pro každou hranu, ale pouze pro všechny hrany z výchozího vrcholu?

Algoritmus pouze nalezne všechny následníky počátečního vrcholu.

b) Co bude počítat v případě, že budeme relaxovat každou hranu pouze jednou, ne $|V| - 1$-krát?

Algoritmus dokáže korektně nahradit cesty délky 2 tranzitivní zkratkou, ale delší cesty nemusí být korektní.

12.24 Co se stane s Bellmanovým–Fordovým a Dijkstrovým algoritmem, pokud znegujeme podmínku v relaxaci? Počítá algoritmus délku nejdelší cesty?

Pan Usměvavý dodává: Takto modifikovaný algoritmus se nejmenuje Relax, ale Dřina.