1.sada - Zásobník, fronta, seznam

1. Vysvětlete co znamená, že zásobník představuje paměť typu LIFO.

Last-in, first-out. Prvek, který byl vložen poslední, je jako první ze zásobníku vyzvednut. Prvky vkládáme na tzv. vrchol zásobníku (stack pointer). Ukazatel zásobníku není nutně ukazatel ve smyslu práce s dynamicky alokovanou pamětí, haldou. Prvně vložený prvek se nazývá dno zásobníku (stack bottom).

2. Co je to vrchol zásobníku? Co je to dno zásobníku?

Ukazatel na aktuální prvek v zásobníku (posledně vloženy) se nazývá vrchol zásobníku (angl. stack pointer). Opakem je dno zásobníku.

3. Jaká je teoreticky kapacita zásobníku?

Zásobník má teoreticky neomezenou kapacitu. Pokud ji omezíme např. velikosti přidělené paměti, a nelze již přidat další prvek nastává opět chyba, tzv. přetečeni (angl. overflow).

4.Co je to přetečení zásobníku? Co je to podtečení zásobníku?

Pokud provedeme operaci *Pop*, na prázdném zásobníku nastává tzv. podtečení (stack underflow). Pokud není možné přidat další prvek, nastává tzv. přetečení (stack overflow).

5.Jak se typicky nazývá operace vložení prvku do zásobníku? Popište tuto operaci.

Push. Prvek je vložen na vrchol zásobníku.

6.Jak se typicky nazývá operace vyjmutí prvku ze zásobníku? Popište tuto operaci.

Pop. Prvek je vyjmut z vrcholu zásobníku (zkopírován a smazán).

7.Závisí složitost operací na počtu prvků v zásobníku? Ano či ne?

Ne, operace mají konstantní složitost.

8.Uveďte příklady použití zásobníku.

Volání funkcí (metod), vyhodnocování aritmetických výrazů, odstranění rekurze, zásobníkově orientované jazyky (PostScript, PDF), testování parity závorek, HTML/XML značek.

9. Do zásobníku vložím prvky v pořadí A, B, C, D. Potom je budu ze zásobníku vyjímat. Jakou dostanu posloupnost?

D, C, B, A

10.Vysvětlete, co znamená, že fronta představuje paměť typu FIFO.

First-in, first-out. Jako první je z fronty odebrán prvek, který byl do fronty první vložen. Jde tudíž o obdobu fronty, jak ji známe z každodenního života. (V tomto okamžiku neuvažujeme prvky, které se mohou „předbíhat“. Potom bychom hovořili o frontě s prioritou).

11. Co je to hlava fronty? Co je to ocas fronty?

Pro implementaci fronty jsou již potřeba dva ukazatele. Jeden ukazatel určuje hlavu (začátek) fronty (angl. head) tj. ukazuje na prvek, který je na řadě pro odebrání, druhým ukazatelem je ocas (konec) fronty (angl. tail). Tento ukazatel ukazuje na poslední prvek ve frontě.

12.Závisí složitost operací na počtu prvků ve frontě? Ano či ne?

Ne, operace mají konstantní složitost.

13.Co je to přetečení fronty? Co je to podtečení fronty?

Operace vložení prvku se tradičně nazývá *Put*, operace odebráni potom *Get*. Obdobně jako u zásobníku je definován dotaz *Empty*, který indikuje prázdnost fronty. Pokud provedeme operaci *Get* nad prázdnou frontou, nastane chyba podtečení. U velikostně omezené fronty může nastat i přetečeni, překročíme-li při vkládaní přiděleny prostor.

14.Do fronty vložím prvky v pořadí A, B, C, D. Potom je budu z fronty vyjímat. Jakou dostanu posloupnost?

A, B, C, D

15. Jak se typicky nazývá operace vložení prvku do fronty? Popište tuto operaci.

Put. Je to vložení prvku za ocas fronty.

16. Jak se typicky nazývá operace vyjmutí prvku ze fronty? Popište tuto operaci.

Get. Je to vyjmutí prvku z hlavy fronty.

17.Uveďte příklady použití fronty.

Tisková fronta u sdílené tiskárny, plánovač v operačním systému, obsluha uživatelů na serverech.

18.Popište hlavní problémy při implementaci fronty v poli. Jak je lze řešit pomocí kruhové reprezentace fronty?

Při odebírání prvku se nemohou všechny prvky přesunout -> jeden index hlava, jeden index ocas, při překročení prvků se index vrací na 0.

19.Jaký je rozdíl mezi frontou, zásobníkem a seznamem? Zaměřte se na operace vkládání a vyjímání prvků z těchto struktur.

Zásobník – LIFO, přístup jen k vrcholu; fronta – FIFO, přístup jen k ocasu; seznam – při vložení se vždy musí nastavit ukazatel (ukazatele) a při odstranění opět seznam „svázat“. Přístup je k prvnímu i poslednímu prvku.

20.Co je to hlava seznamu, ocas seznamu?

Obousměrný spojový seznam (angl. doubly linked list) je tvořen objekty (daty, prvky, záznamy) a dvěma ukazateli prev a next. Každý objekt pochopitelně může obsahovat další
data specifická pro danou aplikaci. Ukazatel prev ukazuje na předchůdce daného prvku seznamu, ukazatel next ukazuje na následníka daného prvku seznamu. Jestliže ukazatel prev
prvku x je roven hodnotě NULL, prvek x nemá tudíž předchůdce, je prvním prvkem seznamu a tvoří hlavu seznamu. Jestliže ukazatel next prvku x je roven NULL, daný prvek nemá následníka, je tedy poslední v seznamu a tvoří ocas seznamu. Položka m_head ukazuje na
první prvek seznamu. Jestliže je m_head rovna NULL, seznam je prázdný.

21.Rozdíl mezi jednosměrným, obousměrným a kruhovým seznamem.

Spojové seznamy se vyskytuji v mnoha variantách. Mohou byt jednosměrné nebo obousměrné, setříděné nebo nesetříděné, cyklické (kruhové) nebo acyklické. Jestliže v prvcích seznamu vynecháme ukazatel prev, dostaneme jednosměrný seznam. Seznam nazýváme setříděný, jestliže prvky seznamu jsou seřazeny. V opačném případě se seznam nazývá nesetříděný. V cyklickém seznamu ukazuje ukazatel prev hlavy seznamu na ocas seznamu a ukazatel next zase na hlavu seznamu. Seznam si lze představit jako prstenec z prvků.

- jednosměrný seznam - nejjednodušší varianta
- obousměrný seznam - položka obsahuje odkaz na předchůdce i následníka
- kruhový seznam - začátek a konec seznamu splývají

22. Popište algoritmus vyhledávání daného prvku v seznamu.

Stejný jako vyhledávání v neseřazeném poli. Sekvenční přístup, lineární složitost.

23.Lze využít algoritmus půlení intervalu pro vyhledávání v seznamu? Jaký je tu hlavní problém?

Nelze – prvky seznamu by musely být setříděny a pokud by byly, tak ani pak to nelze, jelikož seznam neumožňuje náhodný přístup, ale jen sekvenční.

24.Co je to iterátor?

- obecný mechanismus pro průchod datovou strukturou
- matematicky řečeno – pro všechny prvky datové struktury proveď
- *funkce void Reset* – nastaví iterátor na první prvek
- *funkce bool MoveNext* – posune iterátor na další prvek. Vrací true při úspěšném posunu, jinak vrací false
- *funkce T Current* – vrací data z aktuálního prvku.

25.Co je to garbage collector?

- mechanismus pro sběr nepoužívané paměti volného místa
- na první volnou pozici ukazuje index *Free*
- v nepoužívaných položkách seznamu je využita složka *Next*, která odkazuje na další volný prvek
- při alokaci položky si vezmeme první volnou pozici a *Free* posuneme přes *Next* na další volnou pozici
- při dealokaci vložíme uvolňovanou pozici k *Free*
- *GC* se tedy chová jako zásobník volného místa

2.sada – Pointery, dynamická alokace paměti

26. Co je to pointer? Jaký je význam dat obsažených v pointeru? Nápověda: zaměřte se na vysvětlení pojmu nepřímého adresování.

Ukazatel(angl. pointer), podobně jako index u datového typu pole, neobsahuje přímo data uložena v proměnné, ale určuje pouze polohu této proměnné v paměti. Rozdíl mezi ukazatelem a indexem spočívá v tom, že index určuje polohu proměnné v poli – i-tý prvek, zatímco ukazatel obsahuje přímo adresu buněk paměti počítače, kde je proměnná uložena. Ukazatel je proměnná; ukazatel obsahuje adresu nějakého místa v paměti.

27. Jak deklarujeme pointer p na typ int?

Int *p;

28. Jak se nazývá pointer ukazující „nikam“?

Prázdný ukazatel.

29. Co je to operátor reference? Uveďte příklad použití.

& reference, získání ukazatele na objekt (jeho adresy), p=&i;

Abychom mohli do ukazatelové proměnné přiřadit hodnotu (nějakou adresu), musíme ji nejprve někde získat. K tomu nám může posloužit referenční operátor '&'. Jeho zapsáním před identifikátor proměnné získáme adresu paměti, kde je tato proměnná uložena. Tuto adresu pak můžeme přiřadit nějaké ukazatelové proměnné.

30. Co je to operátor dereference? Uveďte příklad použití.

** dereference, získání hodnoty, na níž ukazuje ukazatel, *p=2;*

Použitím operátoru dereference '*' na ukazatel získáme objekt, na který ukazatel odkazuje, přičemž typ takto získaného objektu bude shodný s bázovým typem dereferovaného ukazatele. Navíc je tento objekt l-hodnotou, což znamená, že může stát na levé straně přiřazovacího příkazu.

31.Předpokládejme, že máme dánu proměnnou int a = 8; a pointer int*p;. Jakým způsobem zařídíte, aby pointer p ukazoval na proměnnou a?

Nastavíme ukazatel na adresu proměnné a.

p = &a;

32. Předpokládejme, že máme deklarovány dva pointery int*p; a int*q;. Co se stane po vyhodnocení výrazu p = q;?

Chyba. Pointery musí mít přiřazeny adresu = je potřeba definice. Samotná deklarace nestačí. Adresa proměnné na kterou ukazuje pointer q se přiřadí i do pointeru p, takže oba ukazatele budou ukazovat tam, kam ukazoval pointer q.

33. Předpokládejme, že máme deklarovány dva pointery int*p; a int*q;. Co se stane po vyhodnocení výrazu *p = *q;? Jaké zde můžou nastat problémy?

Hodnota proměnné, na kterou ukazuje pointer q se uloží do proměnné na kterou ukazuje pointer p. Problém je, pokud nejsou pointery definovány, nebo jsou definovány hodnotou NULL. Pak by nebylo kam uložit hodnotu, nebo by bylo kam, ale nebylo by co.

34. Deklarujte pointer d na typ double. Proveďte alokaci paměti a dealokaci paměti. Jaké operátory na to použijete?

```
double * d = new double();
delete d;
```

35. Máte dynamicky alokovat pole a s prvky typu int, které bude mít 20 prvků. Jak to provedete? Pomocí jakého operátoru?

```
int* a = new int[20];
```
Pomocí operátoru new.

36. Je dáno dynamicky alokované pole a s prvky typu int, které má 20 prvků. Máte toto pole dealokovat. Jak to provedete? Pomocí jakého operátoru?

```
delete [] a;
```
Pomocí operátoru delete[];

37. Máte za úkol dynamicky alokovat dvourozměrné pole a, které bude bude mít R řádků a S sloupců. Prvky budou typu int. Jak bude vypadat deklarace pointeru na takové pole? Jak se provede jeho alokace v paměti? Lze to provést jednorázovou alokací?

```
int **pole = new int*[S];
for(int i = 0; i <S; i++)
        pole[i] = new int[R];
```

Alokaci nelze provézt jednorázově, protože je potřeba nejprve vytvořit „pole polí“ – tedy pole ukazatelů, z nichž každý ukazuje na první prvek pole, které je třeba vždy naalokovat.

38. Máte za úkol uvolnit z paměti, dealokovat, dvourozměrné pole a, které má R řádků a S sloupců. Prvky jsou typu int. Jak to provedete? Lze to provést jednorázovou dealokací?

```
for(int i = 0; i <S; i++)
        delete [] pole[i];
delete pole;
```

Dealokaci nelze provést jednorázově, postupně odstraňujeme jednotlivá pole. Až v závěru uvolníme hlavní ukazatel.

39. Máte pointer daný následující deklarací int*p;. Co je na tomto pointeru konstantního a co proměnného?

Ukazatel i objekt, na který ukazuje, jsou proměnné. Konstantního zde není nic.

40. Máte pointer daný následující deklarací const int*p;. Co je na tomto pointeru konstantního a co proměnného?

Proměnný je ukazatel. Konstantní je objekt, na který ukazuje.

41. Máte pointer daný následující deklarací int* const p;. Co je na tomto pointeru konstantního a co proměnného?

Konstantní je ukazatel. Proměnný je objekt, na který ukazuje.

42. Máte pointer daný následující deklarací const int * const p;. Co je na tomto pointeru konstantního a co proměnného?

Ukazatel i objekt, na který ukazuje, jsou konstanty. Proměnného zde není nic.

43. Popište tři způsoby předávání parametru do funkce.

- předávání odkazem (hodnota změněná ve funkci se změní i mimo funkci)
- předávání hodnotou (hodnota změněná ve funkci se nevrací zpět)

44. Do funkce F předáváte pole pomocí pointeru. Ve funkci F na takto předané pole (na parametr z hlavičky funkce) aplikujete operátor sizeof? Jaký výsledek dostanete?

Výsledkem bude počet bajtů zabíraných pointerem (parametrem z hlavičky funkce).

45. Jaké operace jsou povoleny provádět s pointery v pointerové aritmetice? Za jakého předpokladu mají tyto operace smysl? Nápověda: Mají tyto operace smysl pokud pointery ukazují na libovolné místo v paměti? Nebo musí být s něčím svázány?

-, ==. !=, <, <=, >, >=, ++, --

(Aritmetika ukazatelů je omezena na dvě základní operace: odčítání a porovnání. Dále pak inkrementace a dekrementace). Pointerovou aritmetiku využíváme především při práci s poli - tedy s paměťovými bloky jenž jsou složeny z proměnných stejného typu ležících těsně za sebou.

46. Jak je možné chápat pole z pohledu pointeru? Co vlastně z tohoto pohledu představuje identifikátor pole?

Pole jsou paměťové bloky jenž jsou složeny z proměnných stejného typu a leží těsně za sebou. Název (identifikátor) pole je pointer, jenž obsahuje adresu prvního prvku pole.
(Neplatí obecně pro vícerozměrná dynamická pole.)

47. Máte dáno pole int a [10]; a ukazatel int*p = a;. Čemu odpovídá výraz *(p+3)?

a[3]

48. Máte dáno pole int a [10]; a ukazatel int *p = a;. Kam bude ukazovat pointer p po vyhodnocení výrazu p+=2;?

Bude ukazovat na třetí prvek v poli – tedy na a[2].

49. Máte dva pointery int*p, *q;. Jaký je smysl výrazu p == q?

Dva ukazatelé se rovnají, právě když ukazují na totéž místo v paměti.

50. Máte dáno pole int a [10]; a dva ukazatele int*p, *q;. Oba tyto ukazatele ukazují na některý prvek v poli a;. Jaký je smysl výrazů p–q a *p-*q?

p–q: počet prvků, které leží mezi ukazateli.
*p-*q: rozdíl hodnot prvků na které ukazatele ukazují.

3.sada – Dynamicky alokované datové struktury

51. Máme danou strukturu a čtyři proměnné:

```
Struct MyStruct
{ int Id ;
double Value;
} ;

MyStruct a, b;
MyStruct *p, *q;
```

Co se stane,pokud provedu a = b;? A co pokud provedu p = q;?

Chyba – proměnné a ukazatele nejsou definovány (inicializovány), pokud by byly, tak pro oba případy platí, že obsah struktury b (b.Id, b.Value) se překopíruje do struktury a

52. Jak alokuji dynamicky strukturu z předchozí otázky? Pomocí jakého operátoru?

MyStruct * a = new Mystruct;
Operátor new.

53. Jak uvolním z paměti dynamicky alokovanou strukturu, kterou jsem vytvořil v předchozí otázce?

delete a;
Operátor delete.

54. Mám deklarovanou proměnnou MyStruct a;. Jak se dostanu k položce Value v proměnné a?

a.Value;

55. Mám deklarovanou proměnnou MyStruct *p;. Jak se dostanu k položce Value v proměnné p pomocí jediného operátoru (bez dereference)?

(*p).Value; nebo p->Value;

56. Mám deklarovanou proměnnou MyStruct *p;. Jak se dostanu k položce Value v proměnné p pomocí operátoru dereference?

(*p).Value

57. Mám následující kód:

```
double *p = new double;
delete p;
*p = 15;
```

Co tento kód způsobí?

Chybu – ukazatel p je už dealokován, proto jej nelze znovu použít.

58. Mám následující kód:

```
double *p = new double;
p = NULL;
```

Co tento kód způsobí? K čemu v paměti dojde?

Tento kód způsobí alokaci ukazatel p na typ double a následně se inicializuje tak, že se z něj stane prázdný ukazatel – což je ukazatel, který „ukazuje nikam“.

59. Mám následující kód:

```
double *p = new double;
p = NULL;
*p = 15;
```

Co tento kód způsobí? K čemu v paměti dojde?

Chybu, snažíme se zapsat číslo 15 na místo které není definováno, jelikož ukazatel ukazuje nikam.

60. Mám seznam implementovaný pomocí pole. Musím tento seznam (především to pole) explicitně uvolňovat z paměti pomocí operátoru delete?

Pouze pokud je pole dynamicky alokováno.

61. Mám seznam implementovaný pomocí dynamicky alokovaných struktur. Musím tento seznam explicitně uvolňovat z paměti pomocí operátoru delete?

Ano (protože je seznam implementovaný pomocí dynamicky alokovaných struktur).

62. Co je to zarážka (angl. sentinel)? Co nahrazuje?

Zarážka je uzel zvláštního typu, umístěný na konec seznamu. Do zarážky vždy umístíme data, která vyhledáváme, čímž si zajistíme, že je také najdeme. Tímto trikem eliminujeme nutnost neustálé kontroly toho, jestli již nejsme na konci seznamu. Většinou se zarážka realizuje jako normální prvek seznamu, který nenese žádná data.

63. K čemu slouží zarážka?

Do zarážky vždy umístíme data, která vyhledáváme, čímž si zajistíme, že je také najdeme. Tímto trikem eliminujeme nutnost neustálé kontroly toho, jestli již nejsme na konci seznamu. Může urychlit běh programu jako celku, pokud například provádíme operaci se seznamem v cyklu s velkým počtem opakování.

64. Jak se změní algoritmus vkládání prvku na začátek obousměrného seznamu, pokud využijeme zarážku?

Ukazatel na předchozí prvek nebude na poslední v seznamu, ale na zarážku. Zarážka pak na následující prvek bude ukazovat na ten první, který vkládáme.

65. Jak se změní algoritmus smazání prvku z obousměrného seznamu, pokud využijeme zarážku?

Pokud se maže první prvek, tak jeho následovník bude ukazovat na zarážku – jako na svého předchůdce a zarážka na něj (prvek, který byl následovníkem smazaného prvku) jako na svého následovníka. Pokud se maže poslední prvek seznamu, pak jeho předchůdce bude ukazovat na svého následovníka – zarážku. Zarážka bude mít jako svého předchůdce prvek, který byl předchůdcem smazaného prvku. Na mazání ostatních prvků nemá zarážka vliv.

66. Kdy je vhodné využít zarážku? Zaměřte se na spotřebovanou paměť a počet prvků v uvažovaném seznamu.

Zarážky pochopitelně spotřebují paměť odpovídající jednomu prvku v seznamu navíc. Tento nárůst lze považovat za bezvýznamný, pokud zpracováváme seznamy s velkým počtem prvků.

67. Kdy je vhodné použít seznam implementovaný v poli a kdy pomocí dynamických struktur? Zaměřte se na rychlost operací vložení (s tím spojené alokování položky), operací mazání (s tím spojené dealokovaní položky). Dále se zaměřte na možnou změnu kapacity seznamu.

Jelikož algoritmus implementovaný v poli neumožňuje jeho dynamickou alokaci, ale rychlost mazání a vkládání prvků je rychlejší, neboť se místo neuvolňuje z paměti, ale posouvají se jenom indexy hlavy a ocasu. Je to vhodné použít pro zápis a mazání množiny dat, jejichž velikost dopředu známe. Seznam implementovaný pomocí dynamických struktur je vhodný použít tam, kde není přesně známá velikost množiny.

68. Jestliže budu k seznamu přistupovat pouze pomocí funkcí Insert, Delete nebo Search a změním implementaci seznamu. Musím měnit i okolní kód, kde se seznam využívá?

Ne, nemusím měnit tento kód, neboť přistupuji k seznamu pouze pomocí funkcí.

69. Pokud implementuji frontu pomocí dynamických struktur, potřebuji kruhový buffer a podobné „triky“ s polem?

Ne.

70. Zkuste navrhnout efektivní algoritmus pro třídění seznamu. Uvědomte si, že k položkám seznamu můžeme přistupovat pouze sekvenčně, jako například k datům v souboru. Jaký Vám známý algoritmus pro třídění lze na tento problém aplikovat?

BubbleSort/ ShakerSort, protože sekvenčně prochází prvky.

71. Zkuste navrhnout rekurzívní algoritmus pro vyhledání prvku v obousměrném seznamu.

Tento algoritmus je vhodné realizovat pomocí testování, zda je aktuální prvek hledaným prvkem, jestli ano, funkce vrátí prvek. Pokud ne, rekurzivně zavolá sama sebe s inkrementovaným ukazatelem.

72. Jak spočítáte počet položek v seznamu? Má na tento algoritmus vliv, zda je seznam obousměrný nebo jednosměrný?

Počet položek spočítáme sekvenčním průchodem celým seznamem. Nezávisle na tom v jakém směru, takže obousměrný na metodu spočítání nemá vliv.

73. Jak byste v obousměrném seznamu implementovali mazání k-té položky?

k-1. položka bude ukazovat na k+1. jako na svého následovníka, k+1. položka bude ukazovat na k-1. jako na svého předchůdce. Teď může být k. položka smazaná.

74.Popište algoritmus mazání položky z obousměrného seznamu. Položka určená ke smazání se musí nejprve vyhledat podle dat uložených v položce.

Sekvenčně se projde celý seznam a každá položka obsahující shodující se data se smaže jak je popsáno v 73. otázce.

75. Popište algoritmus vkládání na začátek obousměrného seznamu.

Prvek se uloží na začátek, ukazuje na hlavu jako na následovníka, hlava na něj jako na předchůdce.

4.sada – Grafy

76. Definujte pojem neorientovaný graf.

Neorientovaným grafem nazýváme dvojici G – (V, E), kde V je množina uzlů, E je množina jednoprvkových nebo dvouprvkových podmnožin V. Prvky množiny E se nazývají hrany grafu a prvky množiny V se nazývají uzly.

77. Čím se liší orientovaný graf od neorientovaného?

Orientovaný graf je takový graf, jehož hrany jsou uspořádané dvojice. Naproti tomu hrany neorientovaného grafu jsou dvouprvkové množiny.

78. Co je to stupeň uzlu?

Stupeň uzlu je počet všech potomků tohoto uzlu.

79. Co je to sled?

Libovolnou posloupnost uzlů a hran s nimi incidujících nazýváme sledem grafu.

80. Co je to cesta?

Cesta je sled, ve kterém se neopakuje žádný uzel. Existuje-li mezi dvěma vrcholy sled v grafu, existuje mezi nimi i cesta.

81. Co je to délka cesty?

Je to počet hran mezi dvěma vrcholy. Délka cesty z uzlu *u* do uzlu *v* je počet hran mezi uzly *u* a *v*.

82. Co je to uzavřený sled, uzavřená cesta? Jaké je alternativní pojmenování uzavřené cesty?

Uzavřený sled je takový sled, který končí ve stejném vrcholu, ve kterém začíná - tedy sled tvoří kružnici. Uzavřená cesta je tedy uzavřený sled, ve kterém je každá vrchol obsažen pouze jednou s vyjímkou prvního vrcholu, který je zároveň i posledním. Uzavřená cesta = kružnice grafu.

83. Co znamená, že graf je acyklický?

Znamená to, že v grafu neexistuje cesta, která by mě dostala zpět do vrcholu, ze kterého jsem vycházel, aniž bych přitom pro pohyb použil jednu hranu 2x. Lze také říci, že graf se nazývá acyklický, jestliže neobsahuje kružnici.

84. Co znamená, že graf je souvislý?

Znamená to, že z libovolného vrcholu dokážu najít cestu do jiného libovolného vrcholu. Jinak řečeno, graf se nazývá souvislý, jestliže mezi každými dvěma uzly existuje cesta.

85. Jak zní definice volného stromu?

Souvislý, acyklický, neorientovaný graf nazýváme volným stromem.

86. Kolik existuje mezi dvěma vrcholy volného stromu cest?

Existuje právě jedna cesta, protože volný strom je acyklický a souvislý.

87. Kolik hran musíme minimálně přidat do volného stromu, abychom dostali graf s kružnicí?

Stačí přidat jednu hranu a graf bude tvořit kružnici.

88. Kolik hran musíme z volného stromu nejméně odebrat, abychom dostali nesouvislý graf?

Libovolnou jednu hranu.

89. Má-li volný strom |V | vrcholů, kolik má hran?

Má o jednu hranu méně než je počet jeho vrcholů (tedy $|E| = |V| - 1$).

90. Co je to kořenový strom?

Kořenový strom je volný strom, který obsahuje jeden odlišný uzel. Tento odlišný uzel se nazývá kořen.

91. Vysvětlete pojem předchůdce uzlu.

Libovolný uzel y, na jednoznačné cestě od kořene r do uzlu x se nazývá předchůdce uzlu x. Každý uzel je pochopitelně předchůdcem a následovníkem sama sebe. Jestliže y je předchůdce x a zároveň x se nerovná y, potom y je vlastní předchůdce uzlu x.

92. Vysvětlete pojem následovník uzlu.

Jestliže y je předchůdce x, potom x se nazývá následovník uzlu y. Jestliže y je předchůdce x a zároveň x se nerovná y, potom x je vlastní následovník uzlu y.

93. Vysvětlete pojem rodič uzlu.

Jestliže poslední hrana na cestě z kořene r do uzlu x je hrana (y, x), potom se uzel y nazývá rodič uzlu x.

94. Vysvětlete pojem potomek uzlu.

Jestliže poslední hrana na cestě z kořene r do uzlu x je hrana (y, x), potom uzel x je potomek uzlu y.

95. Vysvětlete pojem sourozenec uzlu.

Dva uzly mající stejného rodiče se nazývají sourozenci.

96.Vysvětlete pojem list.

Uzel bez potomků se nazývá externí uzel neboli list. Je tedy je úplně na spodu stromu.

97. Vysvětlete pojem vnitřní uzel.

Nelistový uzel je vnitřním uzlem. Je to uzel, který není ani list, ani kořen. Tedy nenachází se ani úplně na vrcholu stromu (není to kořen), ani úplně dole (není to list), ale mezi nimi.

98. Jak se liší definice stupně uzlu pro obecný graf a kořenový strom?

V obecném grafu neexistuje pojem rodič a potomek. Existují jen sousední uzly.

99. Co je to hloubka uzlu.

Hloubka uzlu je vzdálenost uzlu od kořene, tedy počet hran mezi uzlem a kořenem.

100. Jak je definována výška stromu?

Je to délka nejdelší možné cesty od kořene k listu (tedy počet hran mezi kořenem a nejvzdálenějším listem).

101. Co je to seřazený strom?

Seřazený strom (angl. ordered tree) je kořenový strom, ve kterém jsou potomci každého uzlu seřazeni. Tudíž, pokud uzel má k potomků, lze určit prvního potomka, druhého potomka, až k-tého potomka.

102. Lze v seřazeném stromu u uzlu s jediným potomkem rozlišit, jestli je levý nebo pravý, první či druhý?

ANO, lze. Jestliže mám kořen 5 a potomek je 2, tak je levý. Kdyby ale byl potomek 7, tak je pravý.

5.sada – Průchody grafem

103. Matice sousednosti grafu. Jaký je řád této matice? Je čtvercová? Jaký je význam prvků v matici?

Matice je čtvercová. Počet prvků ve sloupci a řádku odpovídá počtu vrcholů v grafu. Čísla v matici představují váhy hran.

104. Matice sousednosti grafu. Jak poznám, že graf je neorientovaný? Jak poznám, že graf neobsahuje smyčky?

U neorientovaného grafu je matice sousednosti symetrická podle hlavní diagonály. Když graf neobsahuje smyčky, jsou na hlavní diagonále nuly.

105. Reprezentace grafu pomocí seznamu sousedních vrcholů. Jaké jsou výhody této implementace oproti matici sousednosti?

Matice sousednosti nám jako jediná reprezentace umožňuje v konstantním čase určit, zda-li je cesta „v“ a „u“ mezi vrcholem V a U hranou. Pro řídké grafy tímto testem zaplatíme vyšší paměťovou náročností, naopak pro husté grafy je to ideální reprezentace. jelikož v některých algoritmech procházíme často všechny sousedy některých vrcholů, testy, zda je „u“ a „v“ hranou nepotřebujeme. V takovém případě je nejlepší reprezentací seznam sousedů.

106. Co je cílem algoritmů průchodu grafem?

Průchod grafem řeší úlohu, kdy potřebujeme ve všech uzlech nad údaji provést nějakou operaci. Potřebujeme projít všechny uzly.

107. Jaké tři algoritmy průchodu grafem znáte?

Do hloubky, do šířky, značkování vrcholů

108.Popište obecný algoritmus značkování vrcholů.

Vrcholům přiřazuji značky, tzn.:
1. Inicializace: Označkujeme vrchol „s“, ostatní vrcholy jsou beze značek.
2. Výběr hrany: Vybereme libovolnou hranu „e“ jejíž jeden vrchol má značku a druhý vrchol nikoli. Pokračujeme podle kroku 3. Jestliže taková hrana neexistuje, algoritmus končí.
3. Značkování: Označkueme dosud neoznačkovaný vrchol hrany e a pokračujeme v algoritmu podle kroku 2.

109. V čem spočívá nevýhoda obecného značkovacího algoritmu? Co v jeho implementaci může zvyšovat časovou složitost?

Nevýhoda spočívá v opakování průchodu seznamem hran. V jeho implementaci zvyšuje náročnost potřeba procházet celý seznam hran.

110. Popište myšlenku algoritmu průchodu grafem do šířky.

Algoritmus můžeme představit tak, že se celým grafem rozlévá pomyslná tekutina, která postupně zaplavuje dostupnou část grafu. Každému vrcholu přiřadíme barvu, bílou, černou, šedou. Zprvu je každý vrchol bílý, poté šedý a nakonec jsou všechny černé.

111. V algoritmu BFS se používají pro označení vrcholů tři barvy – bílá, šedá a černá. Jaký význam mají tato tři označení?

Bílý vrchol ještě nebyl navštíven. Šedý vrchol byl již navštíven, může mít i bílé sousedy. Černý vrchol byl již navštíven, ale může mít černé, nebo šedé sousedy.

112. V jaké datové struktuře je při průchodu BFS udržována množina šedých vrcholů?

Množina šedých vrcholů je organizována jako fronta Q.

113. Jaký význam mají hodnoty d[u] a pí[u] z algoritmu BFS?

V tabulce d[u] je uložena vzdálenost od vrcholu. V tabulce pí[u] je uložen předchůdce vrcholu.

114. Jak detekuji konec BFS algoritmu? Jak poznám, že jsem skončil?

Již nemám co navštívit, fronta je prázdná.

115. Výstupem BFS je tzv. BF-strom. Jaké vrcholy tento strom obsahuje?

Obsahuje všechny vrcholy z původního grafu. (kostru původního grafu)

116. Mám dva vrcholy u a v v BF-stromu. Jaká je délka cesty mezi těmito dvěma vrcholy ze všech možných cest mezi těmito vrcholy v původním grafu? Nejkratší? Nejdelší? Průměrná?

Nejkratší.

117. Co je to průměr grafu?

Maximální délka nejkratší cesty pro libovolné dva vrcholy.

118. Jaká je složitost BFS algoritmu?

O(|V| + |E|).

119. Popište myšlenku algoritmu průchodu grafem do hloubky (DFS).

Je to jako prohledávání bludiště jedním člověkem. Nejprve označíme vrchol „s“, následně jeho následovníka a jeho následovníka atd.Až dojdeme nakonec, potom se vracíme postupně zpátky a značkujeme další ještě neoznačkované následovníky.

120. DFS algoritmus přiřazuje každému vrcholu dvě tzv. časová razítka. Jaký mají význam?

První časové razítko d[v] označuje okamžik, kdy byl vrchol navštíven, naopak f[v], okamžik, kdy byl ukončen průchod všemi sousedy vrcholu.

121. Jak mohu, pomocí časových razítek, určit pro dva vrcholy vztah rodič-potomek v DF- stromu?

Díky závorkové struktuře stromů -> Mějme vrchol „u“, pro všechny vrcholy v podstromu s kořenem „u“ platí, že d[u]< d[v] < f[v] <f[u]. Indexování XML dokumentů.

122. Jaká je složitost DFS algoritmu?

O(|V|+ |E|).

123. DFS lze implementovat i nerekurzivně? Pomocí jaké datové struktury mohu odstranit rekurzi?

Ano dá, pomocí zásobníku.

124. Co je to kostra grafu?

Acyklický podgraf obsahující všechny vrcholy původního grafu.

125. Co je to dopředná a zpětná hrana?

Dopředná hrana je součástí DF stromu, zpětná hrana je mimo něj.

126. Co je to topologické třídění?

Cílem topologického třídění je seřadit částečně uspořádané objekty do lineární posloupnosti.

127. Jak mohu detekovat v grafu cyklus?

Při průchodu grafem narazím na šedý vrchol (zpětná hrana).

6.sada – Binární vyhledávací stromy

128. Jak zní rekurzivní definice binárního stromu?

Binární strom je struktura definovaná nad konečnou množinou uzlů, která:

• neobsahuje žádný uzel
• je složena ze tří disjunktních množin uzlů: kořene, binárního stromu zvaného levý podstrom a binárního stromu tzv. pravého podstromu.

129. Je možné levého a pravého potomka daného uzlu v binárním stromu zaměňovat?

NE

130. Co je to úplný binární strom?

Je to strom, kde každý vnitřní uzel má stupeň právě dva – tedy 2 potomky.

131. Předpokládejme, že máme dán binární strom. Co musí platit pro klíče ve všech uzlech binárního stromu, abychom jej mohli považovat za binární vyhledávací strom?

Klíče v binárním vyhledávacím stromu jsou vždy uspořádány tak, že splňují vlastnost binárních vyhledávacích stromů: Jestliže y je z levého podstromu uzlu x, potom klíč[y] ≤ klíč[x]. Jestliže y je z pravého podstromu uzlu x, potom klíč[x] ≤ klíč[y].

132. Popište rekurzivní algoritmus vyhledání klíče x v binárním vyhledávacím stromu.

Mějme dánu množinu M reprezentovanou binárním stromem a jeho kořen r. Dále mějme dán prvek s klíčem k, který hledáme. Základem vyhledávací procedury je uspořádaní klíčů ve stromu. Hledání zahájíme v kořeni stromu r. Potom mohou nastat tyto možnosti: 1. Strom s kořenem r je prázdný (r = NULL), potom tento strom nemůže obsahovat prvek s klíčem k a hledání končí neúspěchem. Platí tedy k „nenáleží“ M. 2. V opačném případě srovnáme klíč k s klíčem kořene právě zkoumaného stromu resp. jeho podstromu r. V případě, že:

(a) k = klíč(r) strom obsahuje prvek s klíčem k. Platí k ∈ M. Hledání končí úspěšně;
(b) k < klíč(r) vzhledem k vlastnostem binárních vyhledávacích stromů, jsou všechny prvky s klíči menšími než je klíč r v jeho levém podstromu, pokračujeme rekurzivně v levém podstromu;
(c) k > klíč(r) na rozdíl od předchozího případu jsou všechny prvky s klíči většími než je klíč r v pravém podstromu, pokračujeme pravým podstromem.

Stručně lze tento algoritmus vyjádřit následujícím pseudokódem:

```
bool Search(CNode ∗ p, T k)
{
if (p == NULL) return false ;
if (k == p − >key) return true;
if (k < p − >key) Search(p − >left, k);
else Search(p − >right, k);
}
```

133. Z jaké vlastnosti (obecně) stromu plyne fakt, že jsme schopni v binárním vyhledávacím stromu nalézt všechny klíče, které do něj byly vloženy. Jinak řečeno, že lze navštívit všechny uzly ve stromu.

Z kořene existuje ke každému uzlu právě jedna cesta, která se řídí pravidlem, že pro každý uzel x jsou klíče v jeho levém podstromu menší nebo rovny klíči v x a klíče v jeho pravém podstromu jsou větší nebo rovny klíči v x.

134. Z jaké vlastnosti (obecně) stromu plyne fakt, že si nemusíme značkovat již navštívené uzly stromu? Například u průchodu grafem jsem si museli uzly označovat barvami – bílou, šedou a černou.

Protože graf je na 100% acyklický a tudíž nemá smyčky, takže se nemůže stát, že bychom se ve smyčce vrátili.

135. Popište nerekurzivní variantu vyhledání klíče x v binárním vyhledávacím stromu.

Opakuj, dokud není uzel prázdný:

- pokud je klíč hledaného záznamu menší než hodnota tohoto uzlu, pak se přesuň na levého potomka
- pokud je klíč hledaného záznamu větší než hodnota tohoto uzlu, pak se přesuň na pravého potomka
- pokud je klíč hledaného záznamu roven hodnotě tohoto uzlu, pak vrať true – nalezeno

Pokud je uzel prázdný, vrať false – nenalezeno.

136. Popište algoritmus jak lze v binárním vyhledávacím stromu najít minimum mezi všemi v něm uloženými klíči.

Prvek, jehož klíč je minimem z množiny reprezentované daným stromem, lze v binárním stromu velice lehce najít sledováním pointerů left od kořene až k uzlu, který nemá levého potomka.

137.Popište algoritmus jak lze v binárním vyhledávacím stromu najít maximum mezi všemi v něm uloženými klíči.

Prvek, jehož klíč je maximem z množiny reprezentované daným stromem, lze v binárním stromu velice lehce najít sledováním pointerů right od kořene až k uzlu, který nemá pravého potomka.

138. Popište algoritmus vložení klíče x do binárního vyhledávacího stromu.

Vkládání do binárního vyhledávacího stromu probíhá obdobně jako vyhledávání v takovém stromu. Nejprve je nutno určit kam vkládaný prvek přijde. Vzhledem k uspořádání klíčů ve stromu je takové místo určeno jednoznačně. Stejně jako při vyhledávání sestupujeme rekurzivně od kořene dolů směrem k listům. Vkládaný prvek x porovnáme s kořenem r zkoumaného podstromu.

139. Jakým způsobem souvisí algoritmus pro vkládání a vyhledávání v binárním vyhledávacím stromu?

Stejně jako při vyhledávání sestupujeme rekurzivně od kořene dolů směrem k listům. Při vkládání vlastně nejprve vyhledáme místo kam prvek patří a pokud se zde již nenachází, pak jej zde umístíme.

140. Jaká je nejhorší podoba binárního vyhledávacího stromu? V jakou jinou datovou strukturu strom v tomto případě zdegeneruje?

Každý klíč je při budování stromu připojen nalevo (resp. napravo) ke svému rodiči. Stane se z něj lineární seznam.

141. Jak vypadá posloupnost klíčů, jejímž vložením vznikne degenerovaný binární vyhledávací strom?

Všechny klíče jsou již setříděny. Budeme se snažit sestrojit takový strom, který je nějakým způsobem vyvážený.

142. Lze se nějak bránit vzniku degenerovaného binárního vyhledávacího stromu?

Např. použitím AVL stromu. AVL strom je vyvážený tehdy a jen tehdy, je-li rozdíl výšek každého uzlu nejvýše 1.

143. Máme daných n různých klíčů. Vložením těchto klíčů do binárního vyhledávacího stromu, vznikne degenerovaný binární vyhledávací strom. Jaká je výška takto zdegenerovaného binárního vyhledávacího stromu?

Výška je n, zdegeneruje to na seznam.

144. Máme dáno n různých klíčů. Tyto klíče vložíme do binárního vyhledávacího stromu, který není degenerovaný. Jaká je, v nejlepším případě, výška takto vzniklého binárního vyhledávacího stromu?

log2 n.

145. Předpokládejme, že máme binární vyhledávací strom s n uzly, který není degenerovaný, a hledáme klíč k. Kolik uzlů musíme v průměru projít, abychom zjistili, zda klíč k v binárním vyhledávacím stromu je nebo není?

$$a_n = \frac{1}{n}\sum_{i=1}^{n} p_i \quad +1$$

146.Předpokládejme, že máme binární vyhledávací strom s n uzly, který není degenerovaný a vkládáme klíč k . Kolik uzlů musíme v průměru projít, než můžeme klíč k do stromu vložit?

$$a_n = \frac{1}{n}\sum_{i=1}^{n} p_i \quad +1$$

147. Popište rekurzivní algoritmus, který zjistí počet uzlů ve stromu. Nápověda: zamyslete se, kolik mají uzlů stromy zmiňované v bodech 1 a 2 v definici binárního stromu.

```
int countNodes(Tree tree)
{
if (tree == NULL) { return 0; }
else if (tree->left == NULL && tree->right == NULL) { return 1; }
int left = countNodes(tree->left);
int right = countNodes(tree->right);
return 1 + left + right;
}
```

7.sada – Binární vyhledávací stromy

148. Máme za úkol odebrat klíč ze stromu, čili zrušit odpovídající uzel v binárním vyhledávacím stromu. Jak postupujeme, jestliže rušený uzel nemá žádné potomky?

Nejdříve je nutné rušený prvek nalézt ve stromu. Postupujeme obdobně jako v případě vyhledávání, tj. sestupujeme rekurzivně od kořene dolů, směrem k listům. Pokud rušený prvek ve stromu není, procedura končí bez jakékoliv činnosti. Pokud má x nula potomků – to znamená, že uzel x je list a lze jej snadno od stromu „odříznout. Prostě jej smažu a v uzlu který je jeho předchůdce nastavím pointer na rušený uzel jako NULL.

149. Máme za úkol odebrat klíč ze stromu, čili zrušit odpovídající uzel v binárním vyhledávacím stromu. Jak postupujeme, jestliže rušený uzel má jednoho potomka?

Pokud má x jednoho potomka - uzel x již nelze snadno od stromu oddělit, protože odříznutím uzlu x bychom ztratili i jeho potomka. Vezmeme tedy potomka uzlu x, přičemž je celkem lhostejné, zda je to potomek levý nebo potomek pravý a napojíme na něj ukazatel z rodiče uzlu x. Jinými slovy uzel x „obejdeme“, čímž se bezpečně vypojí ze stromu a můžeme jej uvolnit z paměti.

150. V binárním vyhledávacím stromu je dán klíč k. Jakým způsobem najdeme, či kde se v tomto stromu nachází nejbližší menší klíč než k?

Předchůdce uzlu x je nutno hledat v jeho levém podstromu, kde jsou uloženy všechny prvky s klíči menšími než je klíč x. Jelikož předchůdcem rozumíme nejbližší menší prvek než x, nejblíže prvku x bude právě maximum ze všech prvků v levém podstromu uzlu x. Maximum ve stromu se nalézá v jeho nejpravějším uzlu. Hledáme tedy nejpravější uzel z levého podstromu uzlu x. Tímto uzlem nahradíme uzel x. Platí pouze, pokud uzel s klíčem levý podstrom obsahuje. Jinak může být nejbližší menší klíč i rodič daného uzlu.

151. V binárním vyhledávacím stromu je dán klíč k. Jakým způsobem najdeme, či kde se v tomto stromu nachází nejbližší větší klíč než k?

Následovník uzlu x je nutno hledat v jeho pravém podstromu, kde jsou uloženy všechny prvky s klíči většími, než je klíč x. Jelikož následovníkem rozumíme nejbližší větší prvek než x, nejblíže prvku x bude právě minimum ze všech prvků v pravém podstromu uzlu x. Minimum ve stromu se nalézá v jeho nejlevějším uzlu – tedy hledáme nejlevější uzel z levého podstromu uzlu x a tímto uzlem nahradíme uzel x. Platí pouze, pokud uzel s klíčem pravý podstrom obsahuje. Jinak může být nejbližší větší klíč i rodič daného uzlu.

152. Máme za úkol odebrat klíč ze stromu, čili zrušit odpovídající uzel v binárním vyhledávacím stromu. Jak postupujeme, jestliže rušený uzel má dva potomky?

V tomto nejsložitějším případě musíme nahradit uzel x jeho předchůdcem resp. následovníkem, ve smyslu uspořádání klíčů uzlů. Předchůdce uzlu x je nutno hledat v jeho levém podstromu, kde jsou uloženy všechny prvky s klíči menšími než je klíč x. Následovník uzlu x je nutno hledat v jeho pravém podstromu, kde jsou uloženy všechny prvky s klíči většími, než je klíč x. Pokud má dva potomky, vydáme se doprava a poté jdeme stále doleva, popřípadě naopak, tedy že se vydáme doleva a poté jdeme stále doprava, dokud nenarazíme na list, který smažeme, ale jeho hodnotu přiřadíme původně mazanému vrcholu. Je doporučeno strany střídat, tedy nejít pořád doprava nebo pořád doleva.

153. Co je to průchod binárním stromem?

Návštěva každého uzlu stromu právě jedenkrát. Nad každým uzlem se provede obvykle nějaká operace.

154. Lze na binární strom aplikovat obecné algoritmy průchodu grafem? Proč?

Lze, protože strom je speciálním případem grafu.

155. Jakou datovou strukturu byste použili pro průchod stromem po „patrech“?

Frontu (průchod do šířky).

156. Popište přímý průchod binárním stromem (anglicky preorder).

R,A a B — nejprve byl navštíven kořen, pak jeho podstromy. Preorder lze vyjádřit i rekurzivně. R označuje kořen stromu, A a B jeho levý resp. pravý podstrom.

157. Popište vnitřní průchod binárním stromem (anglicky inorder).

A,R a B — nejprve levý podstrom, kořen a nakonec pravý podstrom. Inorder lze vyjádřit i rekurzivně.

158. Popište zpětný průchod binárním stromem (anglicky postorder).

A,B a R — kořen se navštíví až po podstromech. Postorder lze vyjádřit i rekurzivně.

159. Uveďte příklady využití jednotlivých způsobů průchodu binárním stromem.

Preorder je vhodný k vytvoření stromu. Inorder je vhodný pro výpis hodnot z binárního stromu tak aby byly vzestupně seřazeny. Postorder je vhodný například v destruktoru třídy při dealokaci celého stromu z paměti, kde je zcela evidentně potřeba dealokovat jednotlivé podstromy a teprve potom je možno zrušit daný uzel. Při jiném pořadí by se ukazatele na části stromu ztratily.

160. Uvažujme libovolný způsob průchodu stromem (preorder, inorder, postorder). Z jaké vlastnosti binárního stromu plyne fakt, že si nemusíme uzly ve stromu značkovat (černá, šedá a bílá barva), abychom věděli, který uzel jsme již navštívili a přitom algoritmus skončí, tj. neuvázne v nekonečném cyklu?

Strom je acyklický graf. Uzly jsou spojené tak, že ze žádného uzlu není možné se vrátit zpět, neexistuje smyčka.

161. Uvažujme libovolný způsob průchodu stromem (preorder, inorder, postorder). Z jaké vlastnosti binárního stromu plyne fakt, že si nemusíme uzly ve stromu značkovat (černá, šedá a bílá barva), abychom věděli, který uzel jsme již navštívili a přitom žádný uzel nevynecháme?

Strom je souvislý graf. Mezi libovolnými dvěma uzly stromu existuje v tomto případě právě jedna cesta.

162. Popište implementaci iterátoru v binárním vyhledávacím stromu. Jak se musíme vypořádat s faktem, že na každý uzel ve stromu mohou navazovat další dva potomci? Jakou musíme použít pomocnou datovou strukturu?

- Na začátku se posuneme úplně nalevo tzn. na nejmenší prvek ve stromu.
- Chceme se posunout na nejbližší větší prvek, ten se nachází úplně nalevo v sourozeneckém pravém podstromu. Takže se vrátíme ke svému kořenu, a přejdeme jednou doprava a po té zase úplně nalevo. Uzel neobsahuje žádný odkaz na svůj kořen, takže všechny kořeny si musíme uchovávat v zásobníku.

163. Jak najdeme v iterátoru další prvek pro iteraci? Předpokládejme, že iterujeme od nejmenšího prvku směrem k největšímu.

Nejbližší větší prvek je vždy úplně nalevo v sourozeneckém pravém podstromu. Postup:

1. Vytáhneme si ze zásobníku odkaz na svůj kořen.
2. Posuneme se do pravého podstromu.
3. V cyklu se budeme posouvat do levého podstromu a zapisovat si předchůdce.
4. Až narazíme na uzel,který nemá další levý podstrom - pak to bude další prvek.

164. Jak poznáme v iterátoru konec iterace?

Zásobník bude prázdný.

165. Výsledkem algoritmu průchodu grafu do hloubky (DFS) je kostra grafu. DFS algoritmus aplikujeme na binární vyhledávací strom. Jak bude vypadat kostra grafu pro tento druh grafu? Nápověda: Kostra grafu bude shodná s binárním stromem? Nebo bude nějaký uzel či hrana chybět?

Kostra grafu je strom, jelikož to aplikujeme na strom, tak se to nijak nezmění, uzly určitě chybět nebudou a hrany by asi taky neměly, protože jakmile se jakákoliv odebere, tak se to celé rozpadne.

166. Mohou existovat při DFS průchodu binárním vyhledávacím stromem zpětné hrany? Nebo jen dopředné? Vysvětlete.

Nemohou, jsou tam jenom dopředné hrany (protože je strom acyklický).

8.sada – Vyvážené a vícecestné stromy

167. Jak je definován dokonale vyvážený strom?

Strom se nazývá dokonale vyvážený, jestliže pro každý uzel stromu platí, že počet uzlů v jeho levém a pravém podstromu se liší nejvýše o jeden.

168. Jak lze snadno, pro pevně danou množinu klíčů, zkonstruovat dokonale vyvážený strom?

Pravidlo rovnoměrné distribuce známého počtu n uzlů se dá nejlépe formulovat rekurzivně takto:

1. Zvolíme jeden uzel za kořen stromu.
2. Vytvoříme levý podstrom s počtem uzlů nl = n div 2.
3. Vytvoříme pravý podstrom s počtem uzlů nr = n − nl − 1. Aplikací tohoto pravidla na posloupnost prvků dostaneme tzv. dokonale vyvážený binární strom.

169. Máme danou množinu klíčů. Pro tuto množinu sestrojíme odpovídající dokonale vyvážený strom, AVL strom, binární vyhledávací strom (základní nevyvážená varianta). Který z těchto stromů bude mít minimální výšku. Poznámka: v jistých případech se může výška všech tři stromů shodovat, ale vždy tím způsobem, že výška dvou stromů se shodou okolností bude rovnat výšce třetího.

Minimální výšku bude mít dokonale vyvážený strom. Druhý je AVL strom a třetí binární vyhledávací.

170. Jak je definováno kritérium vyváženosti AVL stromu?

Strom je vyvážený tehdy a jen tehdy, je-li rozdíl výšek každého uzlu nejvýše 1. *Stromy, které splňují toto kritérium, se často nazývají AVL–stromy.*

171. Kdy nazýváme binární vyhledávací strom červeno-černým stromem? Co musí vše takový strom splňovat (jaká je jeho definice)?

Binární vyhledávací strom je Red–Black strom, jestliže splňuje následující kritéria:

1. Každý uzel je buď černý nebo červený.
2. Každý list (NULL) je černý.
3. Jestliže je daný uzel červený, pak jeho potomci jsou černí.
4. Každá cesta z libovolného uzlu do listu (NULLu) obsahuje stejný počet černých uzlů.

172. Dokážete formulovat kritérium vyváženosti červeno-černého stromu?

Red–Black strom zajišťuje, že žádná cesta z kořene do libovolného listu stromu nebude dvakrát delší než kterákoli jiná, to znamená, že strom je přibližně vyvážený.

173. Jak se nazývají operace, které obnovují vyváženosti stromu jeho přetáčením tj. změnami vztahu rodič-potomek?

Rotace - rozeznáváme rotaci levou, pravou a jejich kombinace (LL, LR, RR, RL).

174. 2-3-4 strom obsahuje tři různé typy uzlů. Kolik obsahují jednotlivé typy uzlů klíčů?

2-uzel = 1 klíč, 3-uzel = 2 klíče, 4-uzel = 3 klíče.

175. 2-3-4 strom obsahuje tři různé typy uzlů. Kolik obsahují jednotlivé typy uzlů ukazatelů na potomky?

2-uzel – 2 ukazatele, 3-uzel – 3 ukazatele, 4-uzel – 4 ukazatele.

176. Co se stane s 4-uzlem jestliže je již zcela zaplněn a musíme do něj vložit další klíč?

Nejprve rozdělíme 4-uzel na dva 2-uzly a přesuneme jeden z klíčů do rodičovského uzlu.

177. Předpokládejme, že máme dán B-strom řádu n. Kolik každá stránka obsahuje nejvýše klíčů?

2n klíčů – položek.

178. Předpokládejme, že máme dán B-strom řádu n. Kolik musí každá stránka obsahovat nejméně klíčů?

n klíčů – položek.

179. Předpokládejme, že máme dán B-strom řádu n. Existuje výjimka z minimálního a maximálního počtu klíčů ve stránce?

Ano, kořen je výjimka z minimálního počtu klíčů ve stránce.

180. Předpokládejme, že máme dán B-strom řádu n. Jestliže stránka obsahuje m klíčů, kolik má tato stránky potomků?

Klíče jsou ve stránce v pořadí od nejmenšího po největší. M klíčů definuje m + 1 intervalů, kterým odpovídá m + 1 následovníků stránky p0 , p2 , až pm. Strana s m klíči má m + 1 potomků.

181. Předpokládejme, že máme dán B-strom řádu n. Jakou hloubku musí mít všechny listy ve stromu?

Všechny listy ve stromu musí mít stejnou hloubku.

182. Předpokládejme, že máme dán B-strom řádu n. Dále předpokládejme, že klíče ve stránce ukládáme v poli a klíče v tomto poli udržujeme setříděné. Jaký byste použili efektivní algoritmus pro nalezení klíče k v tomto poli?

Rekurzivní algoritmus, kdy si vezmeme prostřední prvek z pole a podle toho zda je menší nebo větší prohledáme jen jednu polovinu. Říká se tomu binární vyhledávání (metoda půlení intervalu).

183. Předpokládejme, že máme dán B-strom řádu n. Dále máme stránku S (S není kořen stromu), která je zcela zaplněna, tj. obsahuje 2n klíčů. Do této stránky chceme přidat další klíč. Popište, co se s touto stránkou stane a co se stane s jednotlivými klíči (skupinami klíčů).

Stránku S rozdělíme na 2 stránky: stránku S a R (R je nově vzniklá schránka). Všechny položky rovnoměrně rozdělíme mezi obě stránky. Zbyde nám jeden klíč, od kterého všechny menší položky jsou ve stránce S a větší ve stránce R. Tento klíč vložíme do jejich rodiče (do stránky předchůdce).

184. Předpokládejme, že máme dán B-strom řádu n. A dále máme kořenovou stránku B-stromu, která je zcela zaplněna, tj. obsahuje 2n klíčů. Do této stránky chceme přidat další klíč. Popište, co se s touto stránkou stane a co se stane s jednotlivými klíči (skupinami klíčů).

Kořenovou schránku rozdělíme na 2 stránky, např. S a R. Všechny položky rovnoměrně rozdělíme mezi obě stránky. Zbyde nám jeden klíč, od kterého všechny menší položky jsou ve stránce S a větší ve stránce R. Tento klíč bude novým kořenem (zvětší se výška stromu) a stránky S a R budou jeho potomky.

185. Předpokládejme, že máme dán B-strom řádu n. Ve stromu je celkem M klíčů. Jaká bude přibližně výška tohoto B-stromu?

Výška bude: $\log_n(M)$ (logaritmus M se základem n)

186. Je B-strom vyvážený nebo ne?

Ano, je vyvážený.

187. Co je to faktor využití paměti u B-stromu?

Faktorem využití paměti rozumíme poměr mezi obsazeným místem ve stránkách a celkovým místem.

188. Předpokládejme, že máme dán B-strom řádu n. Máme stránku, které obsahuje více než n klíčů, jeden z těchto klíčů chceme ze stránky odebrat. Jak to provedete?

V zásadě rozlišujeme dvě situace. Pokud je položka, kterou chceme odebrat, v listové stránce, můžeme ji rovnou odebrat. Pokud není v listové schránce, nahradíme ho nejbližším větším nebo nejbližším menším potomkem, tedy jedním ze dvou sousedních prvků (ten se bude nacházet v listové schránce). Je-li odebíraná položka umístěna ve stránce na pozici i zahájíme sestup i-tým odkazem. Dále sestupujeme ve směru nejlevějších (nejpravějších) odkazů stránky – nultých odkazů – až dosáhneme listové stránky P. Nahradíme položku, která se má odebrat, nejlevější (nejpravější) položkou stránky P a snížíme počet položek v P.

189. Předpokládejme, že máme dán B-strom řádu n. Máme stránku, které obsahuje přesně n klíčů, jeden z těchto klíčů chceme ze stránky odebrat. Jak to provedete?

Pokud má sousední stránka také n klíčů, můžeme svůj prvek odstranit a obě schránky společně s jejich prostředním klíčem z předchůdce spojit do jedné. V ostatních případech odebraný klíč nahradíme větším (resp. menším) klíčem z předchůdce a klíč tento klíč v předchůdci nahradíme nejbližším větším (resp. menším) klíčem z vedlejšího potomka. V každém případě musíme kontrolovat počet klíčů ve stránkách, ze kterých odebíráme, odebírání se může dostat až ke kořenu a strom pak sníží svou výšku.

190. Může B-strom snížit svoji výšku? Kdy k tomu dojde?

Může, dojde k tomu, když operace mazání dospěje k odstranění posledního zbývajícího klíče z kořene.

191. Může B-strom zvýšit svoji výšku? Kdy k tomu dojde?

Může, dojde k tomu, když bude kořenová stránka plná a operace vkládání bude vyžadovat rozdělení kořene.

9.sada – Hašování

192. Co je to univerzum klíčů?

Množina, ze které vybíráme prvky tříděné množiny = univerzum. Universum klíčů = množina všech možných klíčů.

193. Jaký bývá typický poměr velikostí mezi univerzem klíčů a množinou skutečně zpracovávaných klíčů, tj. těmi, které jsou skutečně uloženy v tabulce?

Množina skutečně zpracovávaných klíčů bývá řádově mnohem menší než univerzum klíčů.

194. Co je to kolize v hašovací tabulce?

Dva klíče se mohou hashovací funkcí zobrazit na tentýž slot.

195. Lze kolizím zcela zabránit?

Úplné odstranění kolizí není možné. Kvalitním návrhem hashovací tabulky a hashovací funkce lze výrazně zmenšit počet kolizí.

196. Jakým principem řeší separátní řetězení kolize?

Technika separátního řetězení řeší kolize velice jednoduše. V hashovaní tabulce je v každém slotu pointer na seznam a prvky se stejnou hodnotou hashovací funkce se vkládají do příslušného seznamu.

197. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. Popište, jak se do tabulky vloží klíč k.

1. Vypočteme hodnotu hashovací funkce pro klíč k: h(k).
2. Podle h(k) vyhledáme příslušný slot v hashovací tabulce.
3. Vložíme klíč do seznamu v příslušném slotu.

198. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. Popište, jak se v tabulce nalezne klíč k.

1. Vypočteme hodnotu hashovací funkce pro klíč k: h(k).
2. Podle h(k) vyhledáme příslušný slot v hashovací tabulce.
3. Hledáme klíč v seznamu v příslušném slotu.

199. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. Popište, jak se z tabulky odebere klíč k. Je to vůbec možné?

1. Vypočteme hodnotu hashovací funkce pro klíč k: h(k).
2. Podle h(k) vyhledáme příslušný slot v hashovací tabulce.
3. Hledáme klíč v seznamu v příslušném slotu.
4. Nalezený klíč smažeme ze seznamu.

200. Co je to faktor naplnění α? Jak se vypočítá?

Faktor naplnění je poměr počtu uložených prvků(n) a počtu slotů(m) v tabulce, tedy $\alpha = n/m$. Toto číslo zároveň udává průměrnou délku seznamu ve slotu. Číslo α může být menší než jedna, rovno jedné nebo větší než jedna.

201. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. V tabulce je uloženo celkem n klíčů. Kolik prvků v tabulce musíme prozkoumat, abychom rozhodli, zda se tam prvek nalézá nebo ne. Jak tento počet souvisí s faktorem naplnění α?

Hašovací funkce nám řekne, na který slot v tabulce se prvek umístil => prozkoumán 1 prvek. Dále prozkoumáváme prvky v seznamu, průměrná délka seznamu je dána faktorem naplnění α, a tedy průměrně pro nalezení prvku v tabulce prozkoumáme 1+α prvků.

202. Může být u hašovacích tabulek s kolizemi řešenými pomocí separátního řetězení faktor naplnění α větší než jedna, tj. $\alpha > 1$?

Ano.

203. Může být u hašovací tabulky, kde jsou kolize řešeny otevřeným adresováním faktor naplnění α větší než jedna, tj. $\alpha > 1$?

Na rozdíl od separátního řetězení nejsou ke slotům připojeny žádné seznamy, tabulka je jen průběžně plněna a z tohoto důvodu faktor naplnění nemůže nikdy překročit 1.

204. Co je to jednoduché uniformní hašování?

Hašování s předpokladem, že libovolný klíč je hashován do všech slotů stejně pravděpodobně, nezávisle na tom, kam se hashovaly ostatní klíče.

205. Co je to uniformní hašování?

Hashování, které tvrdí, že pro každý klíč jsou všechny permutace {0, 1, . . . ,m−1} posloupnosti pokusů stejně pravděpodobné. Uniformní hashování je zobecněním jednoduchého uniformního hashování, které pro libovolný klíč generovalo se stejnou pravděpodobností jedno číslo, kdežto uniformní hashování generuje celou posloupnost čísel. Dosáhnout v praxi uniformního hashování je obtížné, ale existují použitelné aproximace.

206. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny otevřeným adresováním, metodou lineárních pokusů. Popište, jak se do tabulky vloží klíč k.

Hashovací funkce nám řekne, kde v tabulce začít vkládat, pokud je slot volný, můžeme prvek vložit. Jinak ověřujeme následující sloty, až narazíme na nějaký volný a prvek vložíme tam.

207. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny otevřeným adresováním, metodou lineárních pokusů. Popište, jak se v tabulce nalezne klíč k.

Hashovací funkce nám řekne, kde v tabulce začít vkládat. Pokud se hledaný prvek v tomto slotu nenachází, pokračujeme v hledání v následujícím slotu, dokud nenajdeme nebo nenarazíme na prázdný slot/konec tabulky.

208. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny otevřeným adresováním, metodou lineárních pokusů. Popište, jak se z tabulky odebere klíč k. Je to vůbec možné?

Vyhledávací funkcí najdeme klíč v tabulce. Klíč nemůžeme přímo smazat, protože bychom mohli narušit posloupnost pokusů pro jiný testovaný klíč, a způsobit kolizi. Jediný způsob jak problém řešit je nastavit slotu příznak, že byl smazán. který by vyhledávací procedura interpretovala jako obsazený slot a naopak vkládací procedura jako volný slot. Takovým postupem už ale nebude záviset složitost vyhledávání jen na faktoru naplnění tabulky α. Proto, když se požaduje mazání prvků z hashovací tabulky, volí se obvykle metoda separátního řetězení.

209. Popište princip řešení kolizí pomocí lineárních pokusů u hašovací tabulky s otevřeným adresováním.

Metoda používá rozšířenou hashovací funkci: h(k, i) = (h′(k) + i) mod m, pro i = 0, 1, ..., m−1. Pro klíč k se nejprve prozkoumá slot T[h′(k)]. Dále se zkoumá slot T[h′(k) + 1]. Postupujeme až ke slotu T[m − 1]. V tomto okamžiku se indexy „přetočí“ a pokračujeme sloty T[0], T[1] až nakonec prozkoumáme slot T[h′(k) − 1]. Jelikož počáteční hodnota hashovací funkce určuje sekvenci pokusů, lze vygenerovat jen m pokusných sekvencí.

210. Popište princip řešení kolizí pomocí kvadratických pokusů u hašovací tabulky s otevřeným adresováním.

Metoda používá hashovací funkci tvaru h(k, i) = (h′(k) + c1i + c2i2) mod m, kde h′ je pomocná Metoda kvadratických pokusů, c1 ≠ 0 a c2 ≠ 0, jsou pomocné konstanty a i = 0, 1, ..., m−1. Prvně je otestován slot T[h′(k)]; další sloty jsou testovány v pořadí určeném funkcí h.

211. Popište princip řešení kolizí pomocí dvojitého hašování u hašovací tabulky s otevřeným adresováním.

Metoda používá hashovací funkci tvaru: h(k, i) = (h1(k) + ih2(k)) mod m, kde h1 a h2 jsou pomocné hashovací funkce. Na počátku je testován slot T[h1 (k)]; následující pokusy jsou určeny posunem o h2(k) pozic modulo m. Je jasné, že dvojité hashování umožňuje větší rozptyl pro výběr sekvencí pokusů, protože na klíči k závisí nejen počáteční pozice, ale i velikost kroku o který se v tabulce posunujeme.

212. Co je to primární shlukování?

Vzniká u metody lineárních pokusů. Prvky v tabulce tvoří souvislé úseky, které mají tendenci se shlukovat do řetězců v důsledku kolidování záznamů. Zvyšuje se tak počet pokusů potřebných k vyhledání prvku. Jestliže navíc budeme vkládat prvek, jehož hashovací hodnota, za jiných okolností nekolidující, padne dovnitř řetězce obsazených slotů a musíme i tento prvek zařadit na konec řetězce.

213. Co je to sekundární shlukování?

Vzniká u metody kvadratických pokusů. Prvky občas vytvoří souvislé úseky, které zvyšují počet pokusů potřebných k vyhledání prvku. Je to slabší obdoba primárního shlukování.

214. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny některou z metod otevřeného adresování. Kolik pokusů můžu maximálně provést při hledání klíče k?

Počet pokusů potřebných k nalezení klíče roste s faktorem naplnění. Teoreticky je-li hašovací tabulka plná, počet pokusů se blíží k nekonečnu. Proto musíme vhodně zvolit velikost hašovací tabulky.

215. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty. Kolize jsou řešeny separátním řetězením. Dále předpokládejme, že hašovací hodnota klíče k je h(k). Který slot (sloty) musím při hledání klíče k v tabulce prohledat?

Jen s indexem h(k), protože při separátním řetězení se klíče s kolizním hashem ukládají do seznamu, který je uložen právě v jednom slotu.

216. Jaká čísla jsou vhodnými kandidáty na velikost hašovací tabulky?

Prvočísla.

10.sada – Vyhledávání v textu (Pattern Matching)

217. Definujte pojem vyhledávání vzorku v textu. Kde se lze s těmito problémy setkat?

Vyhledávání v textu je operace, při které se zjišťuje, zda daný text obsahuje hledaná slova - vzorky. S těmito problémy se lze setkat:

* v textových editorech (pohyb v editovaném textu, záměna řetězců),
* v utilitách typu grep (OS Unix), které umožní najít všechny výskyty zadaných vzorků v množině textových souborů (což programátor ocení např. při hledání všech modulů, které se odkazují na danou globální proměnnou),
* v rešeršních systémech (výběr anotací podle klíčových slov),
* při studiu DNA,
* při analýze obrazu, zvuku apod.

218. Co je to předzpracování vzorku textu?

Předzpracování vzorku – metody - nejdříve pro daný vzorek vytvoří jistá pomocná data, která se následně využijí pro vyhledávání. Vytvoří se vyhledávácí stroj, který potom provádí vyhledávání.

Předzpracování textu – indexové metody - pro text, ve kterém se má vyhledávat, vytvoří index. Indexem zde rozumíme uspořádaný seznam slov s odkazy na jejich umístění v textu.

Předzprac. textu i vzorku - signaturové metody - jak pro daný vzorek tak pro daný text vytvoří řetězce bitů - signatury. Tyto signatury charakterizují jak vzorek, tak text. Vyhledávání se provádí porovnáním signatur.

219. Uveďte algoritmus, který pro svůj běh nepotřebuje předzpracovat ani text, ani vzorek.

BruteForce algoritmus – triviální alg. pomocí hrubé síly.

220. Uveďte příklad algoritmu/algoritmů, který předzpracovává vzorek, ale nepředzpracovává prohledávaný text.

Knuth-Morris-Prattův algoritmus, Boyer-Moorův algoritmus, Quick-Search

221. Uveďte příklad vyhledávací metody, která nepředzpracovává vzorek, ale předzpracovává vyhledávaný text.

Indexové metody.

222. Uveďte příklad vyhledávací metody, která předzpracovává vzorek i vyhledávací text.

Signaturové metody.

223. Co je to abeceda? Označíme ji například ∑.

Je to konečná neprázdná množina symbolů.

224. Co je to řetězec nad abecedou ∑? Jak je definována délka řetězce?

Konečná posloupnost symbolů ze ∑ se nazývá řetězec nad ∑. Délka řetězce x se značí |x| a rovná se počtu výskytů symbolů v něm obsažených.

225. Máte dánu abecedu ∑ a přirozená čísla m a n. Jak budou vypadat řetězce am a ambn, kde a,b ∈ ∑?

m a n udávají počet opakování slova v řetězci, tzn. a2 = aa, a5b4 = aaaaabbbb, a4b2cd3 = aaaabbcddd, atd.

226. Co je to prázdný řetězec? Jak jej značíme? Jaká je jeho délka?

Prázdná posloupnost se nazývá prázdný řetězec a budeme ji značit ∈ (malé epsilon??). Délka je nula.

227. Máte dánu abecedu ∑. Co označují množiny ∑+ a ∑*?

∑+ - je abeceda ∑ bez prázdného řetězce
∑* - je množina všech řetězců nad abecedou ∑

228. Co je to předpona (prefix) řetězce w?

Řetězec u se nazývá předponou (prefixem) řetězce w, jestliže existuje řetězec v (i prázdný) takový, že w = uv.

229. Co je to přípona (sufix) řetězce w?

Řetězec v se nazývá příponou (sufixem) řetězce w, jestliže existuje řetězec u (i prázdný) takový, že w = uv.

230. Co je to podřetězec (faktor) řetězce w?

Řetězec y se nazývá podřetězcem (faktorem) řetězce w, jestliže existují řetězce u a v (i prázdné) tak, že w = uzv.

231. Co je to hranice řetězce w?

Řetězec se nazývá hranicí řetězce w, jestliže existují dva řetězce u a v takové, že w = uz = zv. z je současně prefixem i sufixem w.

232. Popište princip elementárního algoritmu (BruteForce Algorithm).

Vzorek se postupně přikládá na všechny možné pozice v textu a testuje se, jestli nedošlo ke shodě. Prohledávaný text a zadaný vzorek se nijak nepředzpracovává.

233. Jaká je časová složitost elementárního algoritmu? Jaké operace nás zajímají u výpočtu složitosti vyhledávacích algoritmů?

O(mn). Zajímá nás očekávaný počet porovnání potřebných k vyhledání vzorku v textu délky n.

234. Jaká je očekávaná časová složitost elementárního algoritmu pro text a vzorek z přirozených jazyků?

$O(k_L n)$, kde k_L je konstanta závislá na jazyku.

235. Elementární algoritmus provádí jednu činnost se vzorkem velice neefektivně. Tato neefektivní trivia mu znemožňuje dosahovat menší časové složitosti při vyhledávání. O jakou činnost se vzorkem jde a jak tuto neefektivitu řeší ostatní algoritmy? Například Knuth-Morris-Prattův?

Elementární alg. při neshodě vzorku s prohledávaným textem posune vzorek pouze o jednu pozici dál a znovu jej od začátku porovnává. Knuth-Morris-Prattův (KMP) alg. oproti naivnímu přístupu eliminuje porovnávání již jednou zkontrolované části textu.

236. Popište princip Shift-Or algoritmu. Jaké dvě bitové operace se tu používají.

Používá bitové operace. Lze jej snadno přizpůsobit pro přibližné vyhledávání řetězců. Do pole ukládá informaci o shodě prefixů vzorku. Or - Bitový součet, Shift - bitový posun.

237. Jaká je časová složitost Shift-Or algoritmu?

O(n) - nezávisle na velikosti abecedy nebo délky vzorku.

238. Popište princip Karp-Rabinova algoritmu. K čemu slouží u tohoto algoritmu hašovací funkce (Pozn. Vzpomeňte si na kapitolu o hašování a pohled na hašovací funkci jako na "negativní filtr".)?

V textu budeme hledat pouze ty úseky, které mají shodnou hashovací hodnotu jako hledaný vzorek. Algoritmus si můžeme představit tak, že máme pomyslnou hashovací tabulku do které vkládáme části textu. Ty části, které se hashovaly do jiného slotu než vzorek nemusíme vůbec zkoumat. Zkoumat musíme jen ty části, které se hashovaly do téhož slotu jako vzorek tj. sledujeme jen kolidující části textu. Pokud objevíme kolidující část textu, nezbývá nic jiného, než tuto část znak po znaku porovnat se vzorkem.

239. Co je to souměrné a protisměrné vyhledávání?

Souměrné = porovnávání znaků ve vzorku ve stejném směru jako prohledávání textu, čili zleva doprava.

Protisměrné = vzorek je porovnáván zprava doleva, čili proti směru prohledávání textu.

240. Popište princip Boyer-Mooreova algoritmu. Soustřeďte se na využití protisměrného vyhledávání pro efektivní detekci (ne)výskytu vzorku v textu.

Algoritmus při hledání využívá dvě funkce (ve formě tabulek), které jsou vypočteny na základě vzorku během fáze předzpracování. Ve fázi vyhledávání je vzorek pomocí těchto funkcí posouván po textu doprava ať už v případě neshody znaku nebo shody celého vzorku. Jedna z funkcí posunuje vzorkem při nalezení vhodné přípony (angl. good-suffix shift), druhá při neshodě znaku ve vzorku a v textu (angl. bad-character shift).

241. Jaká je složitost Boyer-Mooreova algoritmu v nejhorším případě a jaká v nejlepším? Pro jakou kombinaci vzorku a textu dojde k nejhoršímu případu?

Průměrný případ O(m*n), v nejhorším případě 3n porovnávání znaků – platí pro neperiodický vzorek, v nejlepším případě O(n/m). m- délka vzorku, n- délka prohledávaného textu.