Algoritmy a datové struktury I

TIN060

Ondřej Čepek

Sylabus

1. Prostředky pro popis složitosti algoritmů
2. Základní grafové algoritmy
3. Extremální cesty v grafech
4. Minimální kostra grafu
5. Stromové datové struktury
6. Algoritmy typu „Rozděl a panuj“
7. Třídění
8. Hašování
9. Algoritmy lineární algebry

Jak porovnávat algoritmy?

časová složitost algoritmu
prostorová složitost algoritmu

$\}$ oboje závisí na „velikosti“ vstupních dat

Jak měřit velikost vstupních dat?

rigorózně: počet bitů nutných k zapsání vstupních dat

Příklad: vstupem jsou (přirozená) čísla $a_1, a_2, \dots, a_n$ která je třeba setřídit

velikost dat D v binárním zápisu je $|D| = \lceil \log_2 a_1 \rceil + \dots + \lceil \log_2 a_n \rceil$

časová složitost = funkce $f(|D|)$ udávající počet kroků algoritmu v závislosti na velikosti vstupních dat

intuitivně: není podstatný přesný tvar funkce f (multiplikativní a aditivní konstanty), ale pouze to, do jaké „třídy“ funkce f patří (lineární, kvadratická, exponenciální, …)

Příklad: $f(|D|) = a\,|D| + b$ lineární algoritmus

$f(|D|) = a\,|D|^2 + b\,|D| + c$ kvadratický algoritmus

$f(|D|) = k\,2^{|D|}$ exponenciální algoritmus

Co je to krok algoritmu?

rigorózně: operace daného abstraktního stroje (Turingův stroj, stroj RAM)

zjednodušení (budeme používat): krok algoritmu = operace proveditelná v konstantním (tj. na velikosti dat nezávislém) čase

- aritmetické operace (sčítání, odčítání, násobení, …)
- porovnání dvou hodnot (typicky čísel)
- přiřazení (pouze pro jednoduché datové typy, ne pro pole …)

→ tím se zjednoduší i měření velikosti dat (čísla mají pevnou maximální velikost)

Příklad: setřídit čísla $a_1, a_2, \dots, a_n$ → velikost dat je $|D| = n$

Toto zjednodušení (omezení číselných hodnot konstantou) většinou nevadí při porovnávání algoritmů, ale ovlivňuje zařazení řešených problémů do tříd složitosti

Proč měřit časovou složitost algoritmů?

stačí přeci mít dostatečně rychlý počítač …

Doba provádění f(n) operací (délka běhu algoritmu) pro vstupní data velikosti n za předpokladu že použitý hardware je schopen vykonat 1 milion operací za sekundu

	n						
f(n)	20	40	60	80	100	500	1000
n	20μs	40μs	60μs	80μs	0.1ms	0.5ms	1ms
n log n	86μs	0.2ms	0.35ms	0.5ms	0.7ms	4.5ms	10ms
n^2	0.4ms	1.6ms	3.6ms	6.4ms	10ms	0.25s	1s
n^3	8ms	64ms	0.22s	0.5s	1s	125s	17min
2^n	1s	11.7dní	36tis.let				
n!	77tis.let						

Růst rozsahu zpracovatelných úloh, tj. „zvládnutelné“ velikosti vstupních dat, díky zrychlení výpočtu (lepší hardware) za předpokladu, že na „stávajícím“ hardware lze řešit úlohy velikosti x

	Zrychlení výpočtu			
f(n)	původní	10 krát	100 krát	1000 krát
n	x	10x	100x	1000x
n log n	x	7.02x	53.56x	431.5x
n^2	x	3.16x	10x	31.62x
n^3	x	2.15x	4.64x	10x
2^n	x	x+3	x+6	x+9

Asymptotická (časová) složitost

Intuitivně: zkoumá „chování" algoritmu na „velkých" datech, tj. nebere v úvahu multiplikativní a aditivní konstanty, pouze zařazuje algoritmy do „kategorií" podle jejich skutečné časové složitosti

Rigorózně:

f(n) je asymptoticky menší nebo rovno g(n), značíme $f(n) \in O(g(n))$, pokud

$$\exists c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq f(n) \leq c\, g(n)$$

f(n) je asymptoticky větší nebo rovno g(n), značíme $f(n) \in \Omega(g(n))$, pokud

$$\exists c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n)$$

f(n) je asymptoticky stejné jako g(n), značíme $f(n) \in \Theta(g(n))$, pokud

$$\exists c>0\ \exists d>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n) \leq d\, g(n)$$

f(n) je asymptoticky ostře menší než g(n), značíme $f(n) \in o(g(n))$, pokud

$$\forall c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq f(n) \leq c\, g(n)$$

f(n) je asymptoticky ostře větší než g(n), značíme $f(n) \in \omega(g(n))$, pokud

$$\forall c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n)$$

Základní grafové algoritmy

Značení: graf $G=(V,E)$, V vrcholy, $|V|=n$, E hrany, $|E|=m$

neorientovaný graf: hrana = neuspořádaná dvojice vrcholů

orientovaný graf: hrana = uspořádaná dvojice vrcholů

Reprezentace grafů: matice sousednosti $\Theta(n^2)$

seznamy sousedů $\Theta(n+m)$

Prohledávání grafů

Prohledávání do šířky (BFS – breadth first search)

```
BFS(G,s)
for each u∈V do begin barva[u]:=bílá; d[u]:=Maxint; p[u]:=NIL end;
barva[s]:=šedá; d[s]:=0; Fronta:={s};
while Fronta neprázdná do
        u:=první ve Frontě;
        for each v je soused u do if barva[v]=bílá then
        begin barva[v]:=šedá; d[v]:=d[u]+1; p[v]:=u; v zařaď na konec Fronty end;
        barva[u]:=černá; vyhoď u z Fronty
```

Poznámky k BFS (opakování z přednášky Programování):

1. Prohledává graf po vrstvách podle vzdálenosti (měřeno počtem hran) od vrcholu s
2. Postupně navštíví všechny vrcholy dostupné z s a vytvoří strom nejkratších cest
3. Je základem složitějších algoritmů, např. Dijkstrova algoritmu (nejkratší cesty v grafu s nezápornými váhami na hranách) nebo Primova (Jarníkova) algoritmu (minimální kostra váženého grafu)
4. Funguje i na orientovaném grafu (beze změny)
5. Při zadání pomocí seznamů sousedů běží v čase $\Theta(n+m)$

Použití BFS: testování souvislosti neorientovaného grafu

- vybereme náhodně vrchol s a spustíme BFS z s
- pokud po ukončení BFS zůstane nějaký vrchol bílý: graf není souvislý
- spočítání počtu komponent souvislosti: opakované spouštění BFS z náhodně vybraného bílého vrcholu dokud nějaký bílý vrchol existuje
- opět běží v čase $\Theta(n+m)$

<u>Prohledávání do hloubky</u> (DFS – depth first search)

- neorientovaná verze viz přednáška z Programování – hlavní rozdíl proti BFS spočívá v tom, že aktivní (šedé) vrcholy nejsou ukládány do fronty ale do zásobníku, který je buď explicitně vytvářen algoritmem nebo implicitně rekurzivním voláním
- orientovaná verze: probereme podrobně, předpokládáme že graf je reprezentován pomocí seznamu sousedů

```
DFS(G)
begin   for i:=1 to n do barva[i]:=bílá;
        čas:=0;
        for i:=1 to n do if barva[i]=bílá then NAVŠTIV(i)
end;

NAVŠTIV(i) {jednoduchá verze}
begin   barva[i]:=šedá;
        čas:=čas+1;
        d[i]:=čas;
        for each j je soused i do if barva[j]=bílá then NAVŠTIV(j);
        barva[i]:=černá;
        čas:=čas+1;
        f[i]:=čas
end;
```

Klasifikace hran pro DFS na orientovaném grafu:

(i,j) je **stromová**	j byl objeven z i	při prohlížení (i,j) je j bílý
(i,j) je **zpáteční**	j je předchůdce i v DFS stromě	při prohlížení (i,j) je j šedý
(i,j) je **dopředná**	i je předchůdce j v DFS stromě (ale ne přímý rodič)	při prohlížení (i,j) je j černý a navíc $d(i) < d(j)$
(i,j) je **příčná**	nenastal ani jeden z předchozích tří případů	při prohlížení (i,j) je j černý a navíc $d(i) > d(j)$

Vlastnosti DFS

1. Stromové hrany tvoří orientovaný les (DFS les = množina DFS stromů)
2. Vrchol j je následníkem vrcholu i v DFS stromě $\Leftrightarrow$ v čase $d(i)$ existovala z i do j cesta sestávající výlučně z bílých vrcholů
3. Intervaly $[d(i), f(i)]$ tvoří „dobré uzávorkování“ tj. pro každé $i \neq j$ platí
   - buď $[d(j), f(j)] \cap [d(i), f(i)] = \emptyset$
   - nebo $[d(i), f(i)] \subset [d(j), f(j)]$ a i je následníkem j v DFS stromě
   - nebo $[d(j), f(j)] \subset [d(i), f(i)]$ a j je následníkem i v DFS stromě

   Důsledek: j je následníkem i v DFS stromě $\Leftrightarrow$ $[d(j), f(j)] \subset [d(i), f(i)]$

NAVŠTIV(i) {plná verze}

```
begin    barva[i]:=šedá;
         čas:=čas+1;
         d[i]:=čas;
         for each j je soused i do
         if barva[j]=bílá
                 then    begin   NAVŠTIV(j);
                                 označ (i,j) jako stromovou
                         end
                 else if barva[j]=šedá
                         then    begin   ohlas nalezení cyklu;
                                         označ (i,j) jako zpětnou
                                 end
                         else if d(i) < d(j)
                                 then označ (i,j) jako dopřednou
                                 else označ (i,j) jako příčnou
         barva[i]:=černá;
         čas:=čas+1;
         f[i]:=čas
end;
```

<u>Složitost</u>: stále lineární $\Theta(n+m)$

Topologické číslování vrcholů orientovaného grafu

Definice: Funkce $t : V \to \{1,2, \dots ,n\}$ je topologickým očíslováním množiny V pokud pro každou hranu $(i,j) \in E$ platí $t(i) < t(j)$.

Pozorování: topologické očíslování existuje pouze pro acyklické grafy

Hloupý algoritmus:

1. Najdi vrchol ze kterého nevede žádná hrana a přiřaď mu poslední volné číslo
2. Odstraň očíslovaný vrchol z grafu a pokud je graf neprázdný tak jdi na bod 1.

Složitost: $\Theta(n(n+m))$

Chytrý algoritmus: mírná modifikace DFS, běží v čase $\Theta(n+m)$

Lemma: G obsahuje cyklus $\Leftrightarrow$ DFS(G) najde zpětnou hranu

Věta: Očíslování vrcholů acyklického grafu G podle klesajících časů jejich opuštění (časy $f(i)$) je topologické.

Tranzitivní uzávěr orientovaného grafu

Definice: Orientovaný graf $G'=(V,E')$ je tranzitivním uzávěrem orientovaného grafu $G=(V,E)$ pokud pro každou dvojici vrcholů $i,j \in V$ takových, že $i \neq j$ platí

z i do j vede v G orientovaná cesta $\Rightarrow (i,j) \in E'$

Tranzitivní uzávěr G' reprezentovaný maticí sousednosti = matice dosažitelnosti grafu G

Matici dosažitelnosti lze získat v čase $\Theta(n(n+m))$ pomocí n použití DFS

Silně souvislé komponenty orientovaného grafu

Definice: Nechť $G=(V,E)$ je orientovaný graf. Množina vrcholů $K \subseteq V$ se nazývá silně souvislá komponenta grafu G pokud

- Pro každou dvojici vrcholů $i,j \in K$ takových, že $i \neq j$ existuje v grafu G orientovaná cesta z i do j a orientovaná cesta z j do i. (1)
- Neexistuje množina vrcholů L která by byla ostrou nadmnožinou K a splňovala (1).

Hloupý algoritmus: vytvoříme tranzitivní uzávěr (matici dosažitelnosti) a z něj v čase $\Theta(n^2)$ „přečteme“ jednotlivé SSK

Chytrý algoritmus:

Vstup: orientovaný graf $G=(V,E)$ zadaný pomocí seznamů sousedů

Fáze 1: DFS(G) doplněné o vytvoření spojového seznamu vrcholů podle klesajících časů jejich opuštění

Fáze 2: vytvoření transponovaného grafu G^T

Fáze 3: DFS(G^T) modifikované tak, že vrcholy jsou v hlavním cyklu zpracovávány v pořadí podle seznamu vytvořeného ve Fázi 1 (místo podle čísel vrcholů)

Výstup: DFS stromy z Fáze 3 = silně souvislé komponenty grafu G

Definice: Nechť $G=(V,E)$ je orientovaný graf. Graf $G^T=(V,E^T)$, kde

$$(i,j) \in E^T \Leftrightarrow (j,i) \in E$$

se nazývá transponovaný graf ke grafu G.

Platí: Transponovaný graf lze zkonstruovat v čase $\Theta(n+m)$ a tím pádem celý algoritmus běží v čase $\Theta(n+m)$.

Lemma: Nechť $G=(V,E)$ je orientovaný graf a K je SSK v G. Po provedení DFS(G) platí:

1. množina K je podmnožinou vrcholů jediného DFS stromu
2. v daném DFS stromě tvoří množina K podstrom

Extremální cesty v (orientovaných) grafech

extremální cesta = nejkratší (nejdelší) cesta (záleží na kontextu)

graf bez vah na hranách: délka cesty = počet hran na cestě (lze nalézt pomocí BFS)

graf s váhami na hranách: označme

$G = (V,E)$ orientovaný graf

$w : E \to R$ váhová funkce

pokud $p = (v_0, v_1, \dots ,v_k)$ je orientovaná cesta (povolujeme opakování vrcholů), tak

$$w(p) = w(v_0,v_1) + w(v_1,v_2) + \dots + w(v_{k-1},v_k)$$

Definice (váha nejkratší cesty z u do v)

$$\delta(u,v) = \begin{cases} \min \{ w(p) \mid p \text{ je cesta z } u \text{ do } v \} & \text{pokud } \exists \text{ cesta z } u \text{ do } v \\ \infty & \text{jinak} \end{cases}$$

Definice (nejkratší cesta z u do v)

Nejkratší cesta z u do v je libovolná cesta z u do v pro kterou platí $w(p) = \delta(u,v)$

Negativní cykly: negativní cyklus = orientovaný cyklus s celkovou negativní váhou

- Graf bez negativních cyklů: $\delta(u,v)$ definováno pro všechny dvojice vrcholů u a v a alespoň jedna nejkratší cesta je pro každou dvojici vrcholů prostá (bez cyklů)
- Graf s negativními cykly: pokud z u do v $\exists$ cesta obsahující negativní cyklus, tak dodefinujeme $\delta(u,v) = -\infty$

Nejkratší cesty z jednoho zdroje

Úloha: pro pevně zvolený vrchol $s \in V$ (zdroj) chceme

spočítat $\delta(s,v)$ pro všechna $v \in V \setminus \{s\}$

Co nás čeká:

1. acyklický graf (a jakékoli váhy) $\rightarrow$ algoritmus DAG (algoritmus kritické cesty)
2. nezáporné váhy (a jakýkoli graf) $\rightarrow$ Dijkstrův algoritmus
3. bez omezení (jakýkoli graf i váhy) $\rightarrow$ Bellman-Fordův algoritmus

Triviální pozorování

Vlastnost 1 Pokud $p=(v_0,v_1, \dots ,v_k)$ je nejkratší cesta z v_0 do v_k, pak $\forall i,j : 0 \le i \le j \le k$ platí, že (pod)cesta $p_{ij}=(v_i, \dots ,v_j)$ je nejkratší cestou z v_i do v_j.

Vlastnost 2 Pokud je p nejkratší cestou z s do v a poslední hrana na p je $(u,v)\in E$, pak $\delta(s,v) = \delta(s,u) + w(u,v)$

Vlastnost 3 Pokud je $(u,v)\in E$ hrana, tak $\delta(s,v) \le \delta(s,u) + w(u,v)$.

Zpřesňování horních odhadů pro nejkratší cesty

Pro každý $v\in V$ budeme držet hodnotu $d(v)$, pro kterou bude platit invariant $d(v) \ge \delta(s,v)$.

```
Inicializace (G,s);
 for each v∈V(G) do
        begin   d(v) := ∞ ;
                p(v) := NIL          {předchůdce na nejkratší cestě}
        end;
 d(s) := 0.
```

Po inicializaci se opakovaně (v nějakém pořadí) provádí přepočítávání odhadů:

```
Relax (u,v,w);
  if d(v) > d(u) + w(u,v) then
  begin    d(v) := d(u) + w(u,v);
           p(v) := u
  end.
```

Vlastnost 4 Pokud je $(u,v) \in E$ hrana, tak v okamžiku po provedení Relax (u,v,w) platí $d(v) \leq d(u) + w(u,v)$.

Vlastnost 5 Pokud byla provedena Inicializace (G,s), tak $\forall\ v \in V$ platí $d(v) \geq \delta(s,v)$ a tato nerovnost zůstane v platnosti po libovolné posloupnosti relaxačních kroků. Navíc pokud hodnota $d(v)$ klesne až na hodnotu $\delta(s,v)$, tak už se v dalším průběhu nezmění.

Vlastnost 6 Pokud z s do v nevede orientovaná cesta, tak od Inicializace (G,s) dál platí $d(v) = \delta(s,v) = \infty$.

Vlastnost 7 Nechť je p nejkratší cesta z s do v a poslední hrana na p je (u,v). Nechť je provedena Inicializace (G,s) a po ní posloupnost relaxačních kroků, která obsahuje volání Relax (u,v,w). Pak pokud $d(u) = \delta(s,u)$ platí v nějaký okamžik před zavoláním Relax (u,v,w), tak $d(v) = \delta(s,v)$ platí v jakémkoli okamžiku po zavolání Relax (u,v,w).

Algoritmus DAG (directed acyclic graph) = algoritmus kritické cesty

DAG (G,w,s);
  topologicky setřiď vrcholy grafu G;
  Inicializace (G,s);
  for each ($u \in V(G)$ v topologickém pořadí) do
      for each ($v \in V(G)$ takové že $(u,v) \in E(G)$) do Relax (u,v,w)

Věta: Nechť $G=(V,E)$ je acyklický vážený orientovaný graf a $s \in V(G)$ libovolný vrchol. Pak po ukončení procedury DAG (G,w,s) pro každý vrchol $v \in V(G)$ platí $d(v) = \delta(s,v)$.

Časová složitost: Celý algoritmus běží v $\Theta(n+m)$ protože

- topologické očíslování (setřídění) trvá $\Theta(n+m)$
- vlastní algoritmus trvá $\Theta(1)$ na vrchol a $\Theta(1)$ na hranu, tj. celkem také $\Theta(n+m)$

Aplikace: Acyklický graf, kde (hrany = činnosti) a (váhy = doby trvání činnosti). Graf vyjadřuje závislosti mezi činnostmi, každá orientovaná cesta odpovídá činnostem které musí být prováděny jedna po druhé. Snažíme se najít kritickou cestu, tzn. cestu v grafu s největším možným součtem (každé zpoždění činnosti na kritické cestě způsobí zpoždění celého projektu).

Řešení: V algoritmu DAG buď

- všem vahám otočíme znaménka nebo
- v Inicializace (G,s) zaměníme ∞ za $-\infty$ a v Relax (u,v,w) otočíme nerovnost

Dijkstrův algoritmus

- předpoklad: všechny váhy na hranách jsou nezáporné ($\forall$ $(u,v) \in E$ platí $w(u,v) \geq 0$)
- všechny vrcholy jsou během práce algoritmu rozděleny do dvou množin

a) vrchol v patří do S pokud je jeho nejkratší vzdálenost od zdroje s již spočítána, takže platí $d(v) = \delta(s,v)$ – na začátku (po Inicializace (G,s)) platí $S = \emptyset$

b) v opačném případě patří v patří do $Q = V \setminus S$ kde Q je implementována jako datová struktura podporující vyhledávání vrcholu v s minimálním $d(v)$

```
Dijkstra (G,w,s);
  Inicializace (G,s);
  S := ∅; Q := V(G);
  while (Q ≠ ∅) do
     u := Extract-Min (Q);
     S := S ∪ {u};
     for each (v∈V(G) takové že (u,v) ∈E(G)) do Relax (u,v,w)
```

<u>Věta</u>: Nechť $G=(V,E)$ je vážený orientovaný graf s nezápornými váhami na hranách a nechť $s \in V(G)$ je libovolný vrchol. Pak po ukončení procedury Dijkstra (G,w,s) pro každý vrchol $v \in V(G)$ platí $d(v) = \delta(s,v)$.

<u>Časová složitost:</u> $\Theta(n^2)$ pokud je Q implementováno jako pole

$\Theta((n+m)\log n)$ pokud je Q implementováno jako binární halda

Bellman-Fordův algoritmus

pomalejší než Dijkstra ale obecnější (umí pracovat i se zápornými váhami)

```
Bellman-Ford (G,w,s);
  Inicializace (G,s);
  for i:=1 to |V(G)|-1 do
    for each ((u,v) ∈E(G)) do Relax (u,v,w);
    {n-1 iterací, v každé iteraci zrelaxuje všechny hrany}
  for each ((u,v) ∈E(G)) do if d(v) > d(u) + w(u,v) then return FALSE;
  {hledání záporného cyklu dosažitelného z s}
  return TRUE
```

Časová složitost: $\Theta(nm)$ (každá iterace trvá $\Theta(m)$)

Lemma: Nechť $G=(V,E)$ je vážený orientovaný graf s váhovou funkcí w a zdrojovým vrcholem s. Předpokládejme, že G neobsahuje záporné cykly dostupné z vrcholu s. Pak v době ukončení práce algoritmu platí $d(v) = \delta(s,v)$ pro všechny vrcholy dosažitelné z s.

Věta: Nechť $G=(V,E)$ je vážený orientovaný graf s váhovou funkcí w a zdrojovým vrcholem s. Pak po ukončení procedury Bellman-Ford (G,w,s) platí:

- pokud G obsahuje negativní cyklus dosažitelný z s, tak algoritmus vrátil FALSE
- pokud G neobsahuje negativní cyklus dosažitelný z s, tak algoritmus vrátil TRUE a pro každý vrchol $v \in V(G)$ platí $d(v) = \delta(s,v)$.

Nejkratší cesty pro všechny páry vrcholů

<u>Úloha</u>: pro každou (uspořádanou) dvojici vrcholů u,v spočítat $\delta(u,v)$.

<u>Předpoklad</u>: graf G zadán maticí sousednosti W^G (pokud ne, tak ji ze seznamu sousedů v čase $\Theta(n^2)$ vytvoříme), která je definována předpisem

$$w_{uv} = \begin{cases} 0 & \text{pokud } u=v \\ w(u,v) & \text{pokud } u \neq v \text{ a } (u,v) \in E \\ \infty & \text{pokud } u \neq v \text{ a } (u,v) \notin E \end{cases}$$

<u>Zjednodušení</u>: předpokládáme, že v grafu nejsou záporné cykly (ale záporné hrany tam být mohou).

<u>Cíl</u>: Zkonstruovat matici D^G pro kterou $d_{uv} = \delta(u,v)$.

<u>Řešení</u> (pomocí předchozích algoritmů): n krát spustit algoritmus na hledání nejkratších cest z jednoho zdroje (každý běh spočítá jednu řádku matice D^G)

- G acyklický: n krát DAG $\rightarrow$ složitost $\Theta(n(n+m))$
- nezáporné váhy: n krát Dijkstra $\rightarrow$ složitost $\Theta(n^3)$ (pole) nebo $\Theta(n(n+m)\log n)$ (halda)
- bez omezení: n krát Bellman-Ford $\rightarrow$ složitost $\Theta(n^2m)$ což je $\Theta(n^4)$ pro husté grafy

Poslední hodnotu budeme zlepšovat (případ bez omezení).

Algoritmy „násobení matic“

Budeme postupovat indukcí podle počtu hran na nejkratší cestě. Definujme

$d_{uv}(k)$ = minimální váha cesty z u do v, která obsahuje nejvýše k hran

1. $k = 1$: $d_{uv}(1) = w_{uv}$ uspořádáme-li tato čísla do matice, tak $D^G(1) = W^G$
2. $k - 1 \rightarrow k$: $d_{uv}(k) = \min\{d_{uv}(k-1), \min_{1\leq i\leq n}\{d_{ui}(k-1) + w_{iv}\}\} = \min_{1\leq i\leq n}\{d_{ui}(k-1) + w_{iv}\}$

Poslední rovnost plyne z toho, že vezmeme-li $i = v$, tak $w_{v\,v} = 0$. Takže můžeme psát

$$D^G(k+1) = D^G(k) \otimes W^G$$

kde ovšem maticové násobení $\otimes$ používá skalární součin, v němž je:

- násobení nahrazeno sčítáním a
- sčítání nahrazeno vybráním minima.

Pokud v G nejsou záporné cykly, tak je každá nejkratší cesta jednoduchá (bez cyklů)

$\rightarrow$ každá nejkratší cesta má nejvýše $n - 1$ hran $\rightarrow$ $D^G(n-1) = D^G(n) = D^G(n+1) = \ldots = D^G$

Pomalá verze algoritmu: postupným násobením spočítáme $D^G(1), D^G(2), \ldots, D^G(n-1)$.

Složitost : $n-2$ násobení matic řádu $n \rightarrow (n-2)$ krát $\Theta(n^3) \rightarrow \Theta(n^4)$

Rychlá verze algoritmu: využijeme asociativitu operace $\otimes$ a počítáme jen mocniny.

Složitost : $\log_2 n$ násobení matic řádu $n \rightarrow \log_2 n$ krát $\Theta(n^3) \rightarrow \Theta(n^3 \log_2 n)$

Floyd-Warshallův algoritmus

- podobná idea jako maticové násobení (také „dynamické programování“, t.j postupné budování finálního výsledku z jednodušších výsledků)
- hlavní rozdíl: indukce ne podle počtu hran na nejkratších cestách, ale podle množiny vrcholů, které jsou „povoleny“ jako vnitřní vrcholy na nejkratších cestách

$d_{uv}(k)$ = minimální váha cesty z u do v s vnitřními vrcholy z množiny $\{1, \dots ,k\}$

1. $k=0$: $d_{uv}(0) = w_{uv}$ a tedy $D^G(0) = W^G$
2. $k-1 \rightarrow k$: $d_{uv}(k) = \min\{d_{uv}(k-1), d_{uk}(k-1) + d_{kv}(k-1)\}$

Příčina zlepšení: spočítání $d_{uv}(k)$ trvá $\Theta(1)$ a ne $\Theta(n)$ jako v předchozím případě

```
Floyd-Warshall (G,w);
D^G(0) := W^G;
for k:=1 to n do
  for u:=1 to n do
    for v:=1 to n do d_uv(k) = min{d_uv(k-1), d_uk(k-1) + d_kv(k-1)};
return D^G(n)
```

Složitost: $\Theta(n^3)$

Minimální kostra grafu

Vstup: Souvislý neorientovaný graf $G=(V,E)$ s váhovou funkcí $w : E \to R$.

Úloha: Nalezení minimální kostry grafu G, tj. souvislého acyklického podgrafu $G'=(V,T)$ kde $T \subseteq E$ s minimální možnou celkovou váhou $w(T)$.

Platí: $|T| = |V| - 1$ a proto lze BÚNO předpokládat, že $\forall e \in E: w(e) \geq 0$

Idea: Postupně přidáváme hrany do množiny A pro kterou v každém okamžiku platí, že je podmnožinou nějaké minimální kostry.

Definice: Nechť je množina hran A podmnožinou nějaké minimální kostry. Hrana $e \in E$ se nazývá bezpečná pro A pokud také $A \cup \{e\}$ je podmnožinou nějaké minimální kostry.

```
Min-kostra (G,w);
A := ∅;
for i := 1 to n -1 do
  najdi hranu  (u,v) ∈ E která je bezpečná pro A;
  A := A ∪ {(u,v)};
return A
```

Definice: Rozklad množiny vrcholů na dvě části $(S, V \setminus S)$ se nazývá řez. Hrana $(u,v) \in E$ kříží řez $(S, V \setminus S)$ pokud $|\{u,v\} \cap S| = 1$. Řez respektuje množinu hran A pokud žádná hrana z A nekříží daný řez. Hrana křížící řez se nazývá lehká hrana pro daný řez, pokud je její váha nejmenší ze všech hran které kříží daný řez.

Věta: Nechť $G=(V,E)$ je souvislý neorientovaný graf s váhovou funkcí $w : E \rightarrow R$, nechť $A \subseteq E$ je podmnožinou nějaké minimální kostry a nechť $(S,V \setminus S)$ je libovolný řez který respektuje A. Potom pokud $(u,v) \in E$ je lehká pro řez $(S,V \setminus S)$, tak je bezpečná pro A.

Důsledek: Nechť $G=(V,E)$ je souvislý neorientovaný graf s váhovou funkcí $w : E \rightarrow R$, nechť $A \subseteq E$ je podmnožinou nějaké minimální kostry a nechť C je souvislá komponenta (strom) podgrafu zadaného množinou A. Potom pokud $(u,v) \in E$ je hrana s minimální váhou mezi hranami spojujícími C s jinými komponentami podgrafu zadaného množinou A, tak je (u,v) bezpečná pro A.

Popíšeme dvě různé strategie vybírání bezpečných hran dle Důsledku:

- algoritmus Borůvka (1926) – Kruskal (1956)

  vždy vybírá hranu, která má nejmenší váhu ze všech hran, které vedou mezi stávajícími komponentami podgrafu zadaného množinou A

  v každém kroku sloučí nějaké dva stromy v A v jeden strom

  hrany v A tvoří v každém okamžiku les

- algoritmus Jarník (1930) – Prim (1957)

  hrany v A tvoří v každém okamžiku jediný strom

  vždy vybírá hranu, která má nejmenší váhu ze všech hran, které vedou mezi budovaným stromem a jeho okolím

```
Borůvka-Kruskal (G,w);
setřiď všechny hrany v E do neklesající posloupnosti podle jejich vah;
A := ∅;
for each v ∈ V do Make-Set (v) ;       {každý vrchol je v jednoprvkové množině}
for each (u,v) ∈ E v pořadí dle setříděných vah do
  if Find-Set (u) ≠ Find-Set (v) then  {vrcholy jsou v různých množinách}
        A := A ∪ {(u,v)};
        Union (u,v);                   {sjednocení obou množin}
return A
```

<u>Časová složitost:</u> $\Theta(m \log m)$ když množiny reprezentovány pomocí spojových seznamů

```
Jarník-Prim (G,w,r);                   {r je startovní vrchol – kořen budovaného stromu}
  Q := V(G);
  for each v∈V(G) do klíč(v) := ∞ ;
  klíč(r) := 0; p(r) := NIL;
  while (Q ≠ ∅) do
        u := Extract-Min (Q);
        for each (v∈V(G) takové že (u,v) ∈E(G)) do
          if (v ∈ Q) and  klíč(v) > w(u,v) then
                 klíč(v) := w(u,v);
                 p(v) := u
```

<u>Časová složitost:</u> $\Theta(n^2)$ pokud je Q implementováno jako pole

$\Theta(m \log n)$ pokud je Q implementováno jako binární halda

Dynamické množiny

dynamické – mění se v čase (velikost, obsah, …)

prvek dynamické množiny: je přístupný přes ukazatel (pointer) a obsahuje

- klíč (klíčovou položku) – typicky z nějaké lineárně uspořádané množiny
- ukazatel (nebo několik ukazatelů) na další prvek (nebo prvky)
- případně další data

Operace na dynamické množině

Nechť S je dynamická množina prvků, k hodnota klíče a x ukazatel na prvek :

Find(S,k)	vrací ukazatel na prvek s klíčem k v množině S nebo NIL
Insert(S,x)	do S vloží prvek na který ukazuje x
Delete(S,x)	z S odstraní prvek na který ukazuje x
Min(S)	vrací ukazatel na prvek v S s minimálním klíčem
Max(S)	vrací ukazatel na prvek v S s maximálním klíčem
Succ(S,x)	vrací ukazatel na v pořadí (podle lineárního pořadí klíčů) bezprostředně další prvek v S po prvku na který ukazuje x
Predec(S,x)	to samé pro bezprostředně předchozí prvek

Binární vyhledávací stromy (BVS)

dynamická datová struktura podporující všechny operace na dynamické množině

binární strom: každý prvek dynamické množiny obsahuje 3 ukazatele a to na

- levého syna (levý)
- pravého syna (pravý)
- rodiče (rodič)

binární vyhledávací strom:

pro každý prvek x platí : všechny prvky v levém podstromě prvku x mají menší klíč než x (rovnosti připouštíme) a všechny prvky v pravém podstromě prvku x mají větší klíč než x (POZOR – neplést si BVS a binární haldu)

```
Find(x,k)            {x je ukazatel na kořen BVS obsahujícího množinu S}
while (x<>NIL) and (k<>klíč(x)) do
   if (k< klíč(x)) then x := levý(x)
                   else  x := pravý(x)
return x
```

Složitost je O(h), kde h je výška BVS se kterým se pracuje

```
Min(x)                {x je ukazatel na kořen BVS obsahujícího množinu S}
while (levý(x)<>NIL) do x := levý(x)
return x

Max(x)                {x je ukazatel na kořen BVS obsahujícího množinu S}
while (pravý(x)<>NIL) do x := pravý(x)
return x

Succ(x)               {ukazatel na kořen BVS obsahujícího množinu S není potřeba}
if (pravý(x)<> NIL)
then return Min(pravý(x))    {x má P syna – následník je min v pravém podstromě}
else  {x nemá P syna – stoupáme vzhůru dokud nejdeme od L syna k rodiči}
begin     y := rodič(x)
          while (y<>NIL) and (x=pravý(y)) do
          begin     x := y
                    y := rodič(y)
          end
          return y
end

Preced(x)             {symetrické k  Succ(x)}
```

Složitost těchto <u>vyhledávacích</u> operací je opět O(h)

Zbývá popsat modifikující operace Insert a Delete :

```
Insert(x,z)                     {x je ukazatel na kořen BVS obsahujícího množinu S a z
                                je ukazatel na vkládaný prvek s levý(z) = pravý(z) = NIL}
y := NIL
w := x
while (w<>NIL) do                         {klesám stromem s dvojicí ukazatelů w (první)
begin     y := w                          a y (druhý), když w dojde na NIL, tak y ukazuje
          if (klíč(z) < klíč(w))          na prvek, pod který chceme z zavěsit}
            then w := levý(w)
            else w := pravý(w)
end
rodič(z) := y
if (y<>NIL)
then      if (klíč(z) < klíč(y))
            then levý(y) := z
            else pravý(y) := z
else      x := z                          {z se stává kořenem, strom byl prázdný}
```

Složitost je opět O(h).

Delete má 3 případy, podle počtu synů vyhazovaného prvku: 0,1, nebo 2 synové

```
Delete(x,z)                 {x je ukazatel na kořen BVS obsahujícího množinu S a z
                            je ukazatel na prvek vyhazovaný z BVS}
if (levý(z) = NIL) or (pravý(z) = NIL)
  then y := z
  else y := Succ(z)         {y nyní ukazuje na prvek, který budeme vyhazovat}
if (levý(y) <> NIL)
  then w := levý(y)         {w ukazuje na jediného syna prvku y (pokud existuje)
  else w := pravý(y)        nebo na NIL (pokud y nemá ani jednoho syna) }
if (w <> NIL)  then rodič(w) := rodič(y)        {navázání ukazatele nahoru}
if (rodič(y) = NIL)
  then x := w               {byl vyhozen kořen, w ukazuje na nový kořen}
  else if (y = levý(rodič(y)))  {y je levým synem svého rodiče}
          then levý(rodič(y)) := w              {navázání ukazatele dolů}
          else pravý(rodič(y)) := w
if (y <> z)  then klíč(z) := klíč(y)      {a případně také obsah(z) := obsah(y) tj.
                                          přesypání celého obsahu prvku y do prvku z
                                          (kromě ukazatelů na syny a rodiče)}
```

Složitost je opět O(h)

Červeno-černé stromy

Nevýhoda „obyčejných" BVS – všechny operace jsou O(h) což je O(log n) na „vyvážených" BVS (s n uzly) ale Ω(n) na „zdegenerovaných" BVS (strom může zdegenerovat na obyčejný spojový seznam délky n)

Cíl: chceme garantovat O(log n) pro všechny operace i v nejhorším případě

Červeno-černý strom je BVS, kde každý uzel má navíc dva atributy:

- barva, která může být a) červená nebo b) černá
- typ, který může být a) interní nebo b) externí

Interní uzly jsou všechny uzly v původním BVS, externí uzly jsou uměle přidané „nové listy", tj. na každém NIL ukazateli z interního uzlu do syna „visí" jeden externí uzel. Externí uzly nemají ani klíč ani obsah, jenom barvu a ukazatele na rodiče.

Požadované vlastnosti červeno-černých stromů (definice Č-Č stromů):

1. Každý uzel je buď červený nebo černý
2. Každý externí uzel je černý
3. Každý červený vrchol (který musí být díky 2. interní) má oba syny černé
4. Každá cesta od (libovolně pevně zvoleného) vrcholu x k listům v podstromě s kořenem x obsahuje **stejný počet** černých uzlů

Pozorování:

- každý interní uzel má právě dva syny
- na žádné větvi (od kořene k listu) nejsou dva červené vrcholy za sebou (viz 3.)
- každá větev (od kořene k listu) má stejně černých vrcholů (viz 4.)
- nejdelší větev je nejvýše dvakrát delší než nejkratší větev – strom je „vyvážený"

Definice:

- výška uzlu: $h(x)$ = počet uzlů (nepočítaje x) na nejdelší cestě z x do listu v podstromě s kořenem x
- černá výška uzlu: $bh(x)$ = počet černých uzlů (nepočítaje x) na nějaké cestě z x do listu v podstromě s kořenem x (definice je korektní díky vlastnosti 4.)

Lemma 1: Nechť x je libovolný uzel. Pak podstrom s kořenem x obsahuje alespoň $2^{bh(x)} - 1$ interních uzlů.

Lemma 2: Červeno-černý strom s n interními uzly má výšku nejvýše $2 \log_2(n+1)$.

Důsledek: Dotazovací operace (Find, Min, Max, Succ, Predec) pro BVS mají na Č-Č stromech garantovanou složitost $O(\log n)$ aniž by bylo potřeba je měnit (nemohou pokazit žádnou vlastnost Č-Č stromů, protože strom nemodifikují)

Rotace (levá a pravá)

pomocné operace potřebné pro implementaci operací Insert a Delete, splňující:

- zachovávají vlastnost BVS – pro každý uzel x platí, že klíče v levém podstromě jsou menší než klíč uzlu x a klíče v pravém podstromě jsou větší než klíč uzlu x
- pouze přesměrují konstantně mnoho ukazatelů a tím pádem běží v O(1)

Vkládání uzlu

Pozorování: pokud je kořen Č-Č stromu červený, tak ho lze přebarvit na černý, aniž by mohlo dojít k porušení některé vlastnosti Č-Č stromů. Můžeme tedy předpokládat, že před vkládáním uzlu je kořen stromu černý (a budeme tuto vlastnost i nadále udržovat v platnosti).

Preprocessing: uzel vložíme standardní procedurou na vkládání do BVS a vložený uzel obarvíme na červeno.

Která vlastnost Č-Č stromů může být po preprocessingu porušena? Jedině vlastnost 3. pokud vložený uzel x i jeho otec y jsou oba červené. Pokud je y červený, tak nemůže být kořenem stromu, takže musí mít ještě otce z (který je určitě černý). Nyní mohou nastat 3 případy:

1. Bratr uzlu y (strýc uzlu x) je červený.

Akce: uzly přebarvíme (y a bratra y na černo, z na červeno). Pokud má uzel z černého otce, tak končíme, pokud má červeného otce, tak jsme „chybu“ přesunuli výše a iterujeme (opět jsou tři možnosti). Pokud uzel z nemá otce (je to kořen), tak ho přebarvíme na černo a končíme.

2. Bratr uzlu y (strýc uzlu x) je černý a x je opačným synem y než je y synem z.

Akce: Pokud je x pravým synem y a y je levým synem z, tak Levá Rotace(y), v opačném případě Pravá Rotace(y). Tím je situace převedena na případ 3.

3. Bratr uzlu y (strýc uzlu x) je černý a x je stejným synem y jako je y synem z.

Akce: Pokud je x levým synem y a y je levým synem z, tak Pravá Rotace(z) a přebarvení y na černo a z na červeno. Tím jsou všechny vlastnosti Č-Č stromů splněny a končíme. Opačný případ je symetrický.

Složitost vkládání je O(log n) :

- preprocessing (obyčejný BVS insert) je O(log n)
- akce případu 1. je O(1) a provede se O(log n) krát
- akce případů 2. a 3. jsou obě O(1) a provedou se každá nejvýše jednou

Odstranění uzlu

Preprocessing: uzel odstraníme standardní procedurou na odstranění uzlu z BVS.

Pozorování: skutečně odstraňovaný uzel (nechť je to y) má nejvýše jednoho (interního) syna (nechť je to x) , pokud nemá žádné interní syny, tak označme jako x jednoho z externích synů uzlu y

Pokud je y červený, není nutné nic dělat, vlastnosti Č-Č stromů platí i po deletu y
Pokud je y černý, tak porušena vlastnost 4 (s výjimkou kdy y je kořen stromu), protože cesty původně vedoucí přes y ztratí jedničku ze své černé výšky, ale cesty k listům nevedoucí původně přes y mají stejnou černou výšku jako na začátku operace

Pokud je x červený, stačí přebarvit x na černo a všechny vlastnosti Č-Č stromů začnou platit. Tudíž jediný zajímavý případ je ten, když je x černý.

Myšlenka: uzel x uděláme „dvojitě černým" (což splní vlastnost 4) a tuto „černou barvu navíc" budeme posunovat vzhůru stromem dokud se jí nezbavíme

Pozorování: pokud je x kořen stromu, tak černou barvu navíc jednoduše odstraníme a černá výška **všech** vrcholů klesne o jedna, pokud x není kořen tak rodič(x) musí mít ještě jednoho interního syna (nechť je to w), jinak by cesty k listům nemohly splňovat vlastnost 4 (externí syn uzlu rodič(x) by měl menší černou výšku než x)

Budeme předpokládat, že x je levým synem uzlu rodič(x) (opačný případ je symetrický) a rozlišíme čtyři případy podle barvy w a jeho případných synů:

1. uzel w je červený (a tím pádem má dva černé syny)

Akce: vyměníme barvy w a jeho rodiče (což je také rodič uzlu x) a provedeme Levá Rotace(rodič(x)), čímž situaci převedeme na jeden z případů 2 nebo 3 nebo 4

2. uzel w je černý a má dva černé syny

Akce: odstraníme jednu černou z x a přebarvíme w na červeno. Jejich společného rodiče pokud je červený tak přebarvíme na černo (a končíme), pokud je černý tak přebarvíme na dvojitě černý. Tento postup dvojitě černého uzlu vzhůru se zastaví nejpozději v kořeni.

Poznámka: pokud následuje případ 2 po případu 1, tak proces končí (společný rodič uzlů x a w je po případu 1 červený a je v případu 2 přebarven na černo)

3. uzel w je černý, jeho levý syn je červený a pravý syn černý

Akce: vyměníme barvy w a jeho levého syna a provedeme Pravá Rotace(w), čímž situaci převedeme na případ 4

4. uzel w je černý a jeho pravý syn je červený

Akce: pravého syna uzlu w přebarvíme na černo a z x odebereme černou barvu navíc. Pokud byl rodič(x) červený, tak ho přebarvíme na černo a uzel w na červeno. Provedeme Levá Rotace(rodič(x)).

Složitost odstraňování je $O(\log n)$:

- preprocessing (obyčejný BVS delete) je $O(\log n)$
- akce případu 2 je $O(1)$ a provede se $O(\log n)$ krát
- akce případů 1, 3 a 4 jsou $O(1)$ a provedou se každá nejvýše jednou

AVL stromy

Definice (Adelson-Velskii, Landis) Binární vyhledávací strom je AVL strom (vyvážený strom) tehdy a jen tehdy, když pro každý uzel x ve stromě platí

|(výška levého podstromu uzlu x) - (výška pravého podstromu uzlu x)| $\leq$ 1

Věta Výška AVL stromu s n vrcholy je $O(\log n)$.

Důsledek Všechny dotazovací operace pro BVS (Find, Min, Max, Succ, Predec) které prohledávaný strom nemění mají na AVL stromu složitost $O(\log n)$ aniž by bylo potřeba je jakkoli upravovat.

Modifikující operace Insert a Delete pracují stejně jako na normálním BVS s případným dodatečným vyvážením pomocí rotací (které jsou podobné jako u červeno-černých stromů) – podrobnosti na cvičeních.

Algoritmy typu „Rozděl a panuj (Divide et impera)“

- metoda pro návrh algoritmů (ne dělení programu na samostatné celky)
- algoritmus typu „Rozděl a panuj“ má typicky 3 kroky

1. ROZDĚL úlohu na několik podúloh stejného typu ale menšího rozsahu
2. VYŘEŠ podúlohy, a to:
   a) rekurzivně dalším dělením pro podúlohy dostatečně velkého rozsahu
   b) přímo pro podúlohy malého rozsahu (často triviální)
3. SJEDNOŤ řešení podúloh do řešení původní úlohy

Příklady: MergeSort, Binární vyhledávání

Analýza časové složitosti

$T(n)$ doba zpracování úlohy velikosti n (předpoklad: pokud $n < k$ tak $T(n) = \Theta(1)$)

$D(n)$ doba na rozdělení úlohy velikosti n na a podúloh stejné velikosti n/c

$S(n)$ doba na sjednocení řešení podúloh do řešení původní úlohy velikosti n

$\Rightarrow$ rekurentní rovnice: $T(n) = D(n) + aT(n/c) + S(n)$ pro $n \geq k$

$T(n) = \Theta(1)$ pro $n < k$

Metody řešení rekurentních rovnic

1. substituční metoda
2. master theorem (řešení pomocí „kuchařky“)

V obou případech používáme následující zjednodušení:

- předpoklad $T(n) = \Theta(1)$ pro dostatečně malá n nepíšeme explicitně do rovnice
- zanedbáváme celočíselnost, tj. např. píšeme $n/2$ místo $\lceil n/2 \rceil$ nebo $\lfloor n/2 \rfloor$
- řešení nás většinou zajímá pouze asymptoticky (nehledíme na konkrétní hodnoty konstant) $\Rightarrow$ asymptotická notace používána už v zápisu rekurentní rovnice

Příklad: MergeSort $T(n) = 2T(n/2) + \Theta(n)$

BinSearch $T(n) = T(n/2) + \Theta(1)$

Substituční metoda

- uhodnout asymptoticky správné řešení
- indukcí ověřit správnost odhadu (zvlášť pro dolní a horní odhad)

Příklad: opět MergeSort

Master theorem

Nechť $a \geq 1$, $c > 1$, $d \geq 0$ jsou reálná čísla a nechť $T : \mathbf{N} \to \mathbf{N}$ je neklesající funkce taková, že pro všechna n ve tvaru c^k (kde $k \in \mathbf{N}$) platí

$$T(n) = aT(n/c) + F(n)$$

kde pro funkci $F : \mathbf{N} \to \mathbf{N}$ platí $F(n) = \Theta(n^d)$. Označme $x = \log_c a$. Potom

a) je-li $x < d$, potom platí $T(n) = \Theta(n^d)$,

b) je-li $x = d$, potom platí $T(n) = \Theta(n^d \log n) = \Theta(n^x \log n)$,

c) je-li $x > d$, potom platí $T(n) = \Theta(n^x)$.

Příklady:

- MergeSort $T(n) = 2T(n/2) + \Theta(n)$
- Binární vyhledávání $T(n) = T(n/2) + \Theta(1)$
- Rovnice $T(n) = 9T(n/3) + \Theta(n)$
- Rovnice $T(n) = 3T(n/4) + \Theta(n^2)$
- Rovnice $T(n) = 2T(n/2) + \Theta(n \log n)$

Násobení čtvercových matic

Vstup: matice A a B řádu $n \times n$

Výstup: matice $C = A \otimes B$ (také řádu $n \times n$)

Klasický algoritmus

```
begin for i:=1 to n do
    for j:=1 to n do
    begin C[i,j] := 0;
            for k:=1 to n do C[i,j] := C[i,j] + A[i,k] * B[k,j]
    end
end
```

Časová složitost: $T(n) = \Theta(n^3)$ (n^2 skalárních součinů délky n)

Nyní předpokládejme že n je mocnina čísla 2 ($n=2^k$), což umožňuje opakované dělení matice na 4 matice polovičního řádu až do matic řádu 1×1 a zkusme „rozděl a panuj“ (předpoklad $n=2^k$ později odstraníme)

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C_{11} = (A_{11} \otimes B_{11}) \oplus (A_{12} \otimes B_{21})$$

$$C_{12} = (A_{11} \otimes B_{12}) \oplus (A_{12} \otimes B_{22})$$

$$C_{21} = (A_{21} \otimes B_{11}) \oplus (A_{22} \otimes B_{21})$$

$$C_{22} = (A_{21} \otimes B_{12}) \oplus (A_{22} \otimes B_{22})$$

Každý skalární součin je „roztržen“ na dvě poloviny a „dokončen“ maticovým sčítáním.

Počet maticových operací na maticích řádu $n/2$: 8 násobení $\otimes$ a 4 sčítání $\oplus$

Počet sčítání (reálných čísel) v maticovém sčítání: $4(n/2)^2 = n^2$

<u>Časová složitost</u>: $T(n) = 8T(n/2) + \Theta(n^2)$

<u>Master theorem</u>: $a=8$, $c=2$, $\log_c a=3$, $d=2$ $\qquad T(n) = \Theta(n^3)$

(asymptoticky stejné jako klasický algoritmus)

Ke snížení složitosti je potřeba snížit $a=8$ a zachovat nebo jen mírně zvýšit $d=2$.

Strassenův algoritmus (1969)

Používá pouze 7 násobení submatic řádu n/2 (místo původních 8)

$$M_1 = (A_{12} \ominus A_{22}) \otimes (B_{21} \oplus B_{22})$$

$$M_2 = (A_{11} \oplus A_{22}) \otimes (B_{11} \oplus B_{22})$$

$$M_3 = (A_{11} \ominus A_{21}) \otimes (B_{11} \oplus B_{12})$$

$$M_4 = (A_{11} \oplus A_{12}) \otimes B_{22}$$

$$M_5 = A_{11} \otimes (B_{12} \ominus B_{22})$$

$$M_6 = A_{22} \otimes (B_{21} \ominus B_{11})$$

$$M_7 = (A_{21} \oplus A_{22}) \otimes B_{11}$$

Počet maticových operací řádu n/2: 7 násobení $\otimes$ a 10 sčítání $\oplus$ a odčítání $\ominus$

$$C_{11} = M_1 \oplus M_2 \ominus M_4 \oplus M_6$$

$$C_{12} = M_4 \oplus M_5$$

$$C_{21} = M_6 \oplus M_7$$

$$C_{22} = M_2 \ominus M_3 \oplus M_5 \ominus M_7$$

Počet maticových operací řádu n/2: 8 sčítání $\oplus$ a odčítání $\ominus$

<u>Časová složitost</u>: $T(n) = 7T(n/2) + \Theta(n^2)$

<u>Master theorem</u>: $a=7$, $c=2$, $\log_c a = \log_2 7 = x$, $d=2$ $\quad T(n) = \Theta(n^x) = \Theta(n^{2.81})$

Hledání k-tého z n prvků.

Vstup: neuspořádaná posloupnost n (různých) čísel

Výstup: k-té nejmenší číslo

Časovou složitost budeme pro jednoduchost měřit počtem porovnání.

Pro $k = 1$ (minimum) a $k = n$ (maximum) jde triviálně pomocí $n - 1$ porovnání.

Pro $k = n/2$ (medián) ?????? (ukážeme, že je to stejně těžké jako pro obecné k)

První nápad: setřídit posloupnost, potom vybrat k-tý $\Rightarrow$ časová složitost $\Omega(n \log n)$

Jde to lépe? $\Rightarrow$ zkusíme „Rozděl a panuj“

- z posloupnosti vybereme prvek (pivot) a podle něj roztřídíme posloupnost na tři části a to na m prvků menších než pivot, vybraného pivota a $(n-m-1)$ prvků větších než pivot
- na to je potřeba $n-1$ porovnání
- pokud $k>m+1$ tak zahodíme $m+1$ malých prvků a hledáme $(k-m-1)$-ní prvek mezi $(n-m-1)$ prvky většími než pivot
- pokud $k=m+1$ tak pivot je hledaný prvek a končíme
- pokud $k<m+1$ tak zahodíme $n-m$ velkých prvků a hledáme k-tý prvek mezi m prvky menšími než pivot

Tohle ovšem může špatně dopadnout … pokud nezajistíme „dobrý“ výběr pivota.

Algoritmus (Blum et al. 1972)

1. Rozděl posloupnost délky n na $\lceil n/5 \rceil$ pětic (poslední může být neúplná).
2. V každé pětici najdi její medián.
3. Rekurzivně najdi medián ze získané množiny $\lceil n/5 \rceil$ mediánů.
4. Použij medián mediánů jako pivot k rozdělení vstupní posloupnosti.
5. Pokud medián mediánů není hledaným prvkem, tak rekurzivně hledej v množině prvků menších než je on nebo v množině prvků větších než je on.

Jak „dobré“ je rozdělení podle pivota nelezeného výše uvedeným algoritmem?

Platí: Množina prvků menších než pivot i množina prvků větších než pivot každá obsahuje alespoň $3n/10$ prvků $\Rightarrow$ v bodě 5 iteruji s nejvýše $7n/10$ prvků

Nechť: $T(n)$ = počet porovnání použitý k nalezení k-tého z n prvků v nehorším případě

$$T(n) = 7n/5 + T(n/5) + (n-1) + T(7n/10)$$

mediány pětic (1.+2.) řešení podproblému (5)

medián mediánů (3.) dělení podle pivota (4.)

Tvrzení: $T(n) = O(n)$

Důkaz: substituční metodou (klíčový fakt: $1/5+7/10<1$, což při dělení na trojice nevyjde)

Analýza časové složitosti QuickSortu

1. vyber pivota (například nejlevější prvek v aktuálně tříděné podposloupnosti)
2. roztřiď posloupnost na tři skupiny podle pivota
3. rekurzivně setřiď množiny prvků menších než pivot a prvků větších než pivot

Časovou složitost opět měříme počtem porovnání

Nejlepší případ: pivot vždy přesně uprostřed

$$T(n) = 2T(n/2) + (n-1) \Rightarrow T(n) = \Theta(n \log n)$$

Nejhorší případ: pivot vždy úplně na kraji

$$T(n) = T(n-1) + (n-1) \Rightarrow T(n) = \Theta(n^2)$$

Průměrný případ: ???? (napřed zkusíme „intuitivně“)

Co když pivot vždy vybrán tak, že rozdělení posloupnosti je v poměru 99:1 nebo lepším?

$$T(n) = T(99n/100) + T(n/100) + (n-1) \leq T(99n/100) + T(n/100) + n$$

Řešení: $T(n) = \Theta(n \log n)$ jak lze zjistit analýzou stromu rekurzivních volání

Poznámka: takto to bude fungovat pro každý konstantní poměr, tj. poměr nezávislý na n

Pro přesnou analýzu udělejme následující předpoklady:

1. tříděná posloupnost je $\{1,2, \dots ,n\}$, což lze přepokládat BÚNO
2. každá z $n!$ vstupních permutací má stejnou pravděpodobnost výskytu
3. za pivota vždy vybírán nejlevější (první) prvek aktuálně tříděné podposloupnosti
4. dělení posloupnosti podle pivota zachovává náhodnost uspořádání v obou nově vzniklých podposloupnostech

pivot	pravděpodobnost	malé	velké
1	1/n	0	n-1
2	1/n	1	n-2
…	…	…	…
n-1	1/n	n-2	1
n	1/n	n-1	0

$$T(n) = 1/n\ \Sigma_{m+v=n-1}(T(m) + T(v)) + (n-1) = 1/n\ \Sigma_{i=0}^{n-1}(T(i) + T(n-1-i)) + (n-1)$$

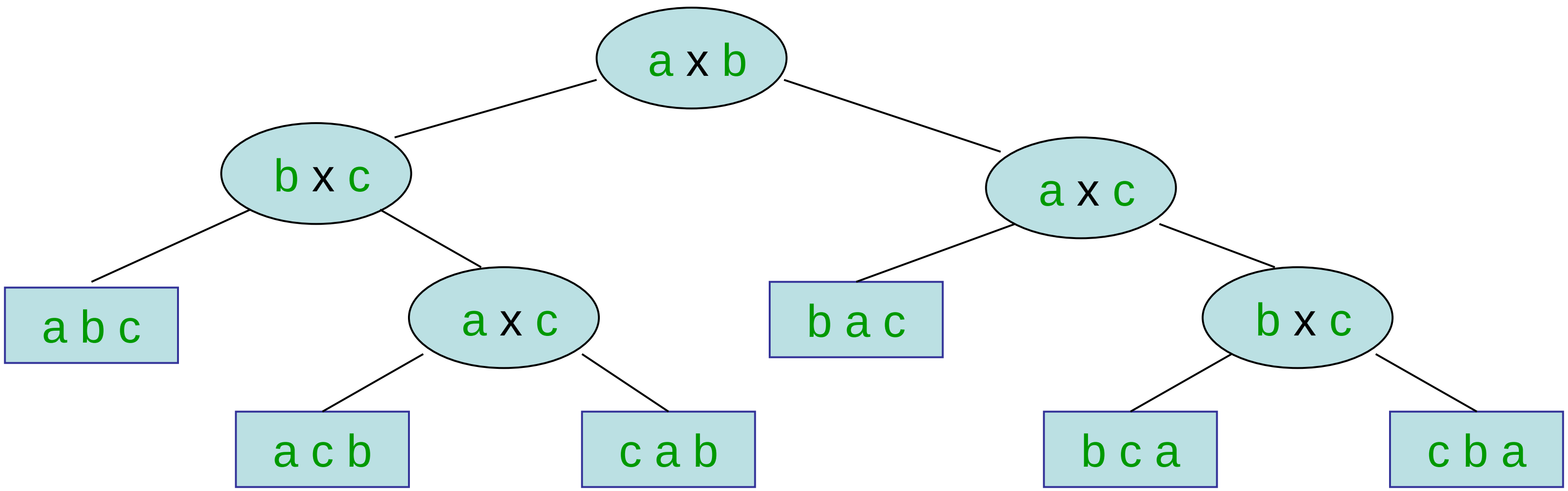
$$T(n) = 2/n\ \Sigma_{i=0}^{n-1} T(i) + (n-1)$$

s počáteční podmínkou $T(1) = 0$

Dolní odhad složitosti porovnávacích třídících algoritmů

Pozorování: každý (deterministický) třídící algoritmus založený na porovnávání (dvojic prvků) lze jednoznačně modelovat rozhodovacím stromem, což je binární strom, jehož vnitřní uzly odpovídají porovnáním a listy permutacím vstupní posloupnosti.

Příklad: rozhodovací strom pro Insertion-Sort a n=3 (označme vstup a,b,c)



levá větev vždy odpovídá výsledku “<“ a pravá větev “>” (BÚNO vstupy po dvou různé)

Rozhodovací strom modeluje korektní třídící algoritmus ⇒ musí obsahovat listy se všemi n! možnými pořadími (permutacemi) vstupní posloupnosti.

Počet porovnání v nejhorším případě = nejdelší větev od kořene k listu = výška stromu.

Věta: Binární strom s n! listy má výšku $\Omega(n \log n)$.

Třídění v lineárním čase

- nepoužívá porovnávání dvojic prvků (hodnot, čísel) v tříděné posloupnosti
- používá adresování (většinou v poli) pomocí tříděných hodnot (klíčů)

Counting Sort

Vstup: n čísel z nichž každé je z intervalu 1 až k (a budeme předpokládat k=O(n))

Datové struktury: A[1..n] vstupní pole

B[1..n] výstupní pole

C[1..k] pomocné pole

Algoritmus:

```
for i := 1 to k do C[i] := 0;                      {inicializace}
for j := 1 to n do C[A[j]] := C[A[j]] +1; {každé C[i] obsahuje # vstupních čísel rovných i}
for i := 2 to k do C[i] := C[i] + C[i-1];   {každé C[i] obsahuje # vstupních čísel ≤ než i}
for j := n downto 1 do begin B[C[A[j]]] := A[j]; C[A[j]] := C[A[j]] – 1 end;
```

Časová složitost: zjevně O(k+n) a tedy O(n) pokud k= O(n)

Doplňková vlastnost: stabilita = na výstupu zachováno takové pořadí stejných hodnot, jaké bylo na vstupu

Radix Sort

- historické použití: třídění děrných štítků na mechanické třídičce
- úloha: setřídit n štítků, každý má v posledních d sloupcích vyraženo d-místné číslo
- intuitivní algoritmus: roztřídit na hromady podle *nejvyššího* řádu, pak rekurzivně třídit jednotlivé hromady, na konci dát vše dohromady (náročné pro obsluhu třídičky)
- radix sort: roztřídit na hromady podle *nejnižšího* řádu, hromady dát ihned na sebe (ve správném pořadí) a dále obdobně třídit podle druhého nejnižšího řádu atd.
- podmínka fungování: každý z d průchodů je stabilní (dva štítky se stejnou cifrou v právě porovnávaném řádu musí být ve stejném pořadí na výstupu jako byly na vstupu)
- současné použití (softwarové verze radix sortu):

třídění dat s vícenásobnými hierarchicky uspořádanými klíči (např. rok, měsíc, den)

třídění alfanumerických klíčů (slov)

jako stabilní sort pro jednotlivé průchody lze použít counting sort

- časová složitost (při použití counting sortu jako podprocedury)

$O(d(n+k)) = O(n)$ pokud $k = O(n)$ a d je konstanta

poznámka: pokud na vstupu čísla s různým počtem cifer: doplníme *zleva* nulami

pokud na vstupu slova různých délek: doplníme *zprava* mezerami

Hašování

Hašovací tabulky jsou vhodné pro reprezentaci dynamických množin, které potřebují pouze operace Insert, Delete a Search

Přímo adresovatelná tabulka = Pole (triviální případ hašovací tabulky)

• velikost tabulky = počet všech **možných** klíčů bez ohledu na počet **použitých** klíčů

• předpoklady : žádné dvě položky nemají stejný klíč (= adresa v tabulce) a

universum možných klíčů je dostatečně malé

• v poli uložena: buď přímo příslušná data (pokud jsou dost malá) s daným klíčem

nebo odkaz (pointer) na příslušná data s daným klíčem (nebo NIL)

• Insert, Delete a Search mají všechny zjevně složitost $\Theta(1)$

Hašovací tabulka

• vhodná v situaci kdy je universum možných klíčů příliš velké, ať už absolutně (nesplnitelné paměťové nároky) nebo relativně vzhledem k počtu aktuálně uložených položek (neefektivní reprezentace, většina buněk v poli zůstane nevyužita).

• adresa v tabulce se nyní z klíče **počítá** pomocí hašovací funkce

$$h : U \rightarrow \{0,1, \ldots , m-1\} \quad \text{kde typicky} \quad |U| >> m$$

Problém: dva (či více) klíče se hašují do stejné hodnoty (adresy v tabulce) = **kolize**

Pozorování: kolizím se nelze vyhnout jakmile $|U| > m$

Metody řešení kolizí: řetězení prvků

otevřená adresace

Řešení kolizí řetězením prvků

- prvky nahašované do stejné adresy v tabulce jsou uloženy ve spojovém seznamu
- každé políčko hašovací tabulky obsahuje buď pointer na hlavu seznamu nebo NIL
- Insert(x) = spočítáme h(klíč(x)) a vložíme x na začátek příslušného seznamu - $\Theta(1)$
- Delete(x) = také $\Theta(1)$ pokud jsou spojové seznamy obousměrné a na x je pointer zvenku, jinak je složitost stejná jako u Search(x)

Analýza složitosti Search(x) při řetězení prvků

Předpoklady: - hodnota hašovací funkce se počítá v $\Theta(1)$
- jednoduché uniformní hašování = každý klíč se hašuje se stejnou pravděpodobností do každé z m adres v tabulce, nezávisle na jiných klíčích

Značení: Faktor naplnění tabulky $\alpha = n/m$
kde m je velikost tabulky a n je počet uložených prvků

Věta 1 V hašovací tabulce s řešením kolizí pomocí zřetězení trvá **neúspěšné** vyhledávání průměrně $\Theta(1 + \alpha)$ za předpokladu jednoduchého uniformního hašování

Věta 2 V hašovací tabulce s řešením kolizí pomocí zřetězení trvá **úspěšné** vyhledávání průměrně $\Theta(1 + \alpha)$ za předpokladu jednoduchého uniformního hašování

Důsledek Pokud $n = O(m)$ tak $\alpha = O(1)$ a tím pádem Search(x) trvá v průměru $\Theta(1)$

Hašovací funkce

tři nejběžnější metody jejich konstrukce:
- hašování dělením
- hašování násobením
- univerzální hašování

Poznámka 1: od hašovací funkce chceme aby klíče z universa klíčů rozdělovala do m adres v tabulce co nejrovnoměrněji

Poznámka 2: předpokládáme, že klíče jsou číselné (aby se na ně daly používat běžné aritmetické operace), pokud jsou klíče nečíselné (například znakové řetězce), tak je napřed musíme na číselné vhodným způsobem zkonvertovat.

Hašování dělením:

$$h(k) = k \bmod m$$ (zbytek po dělení číslem m)

Nevhodné pokud $m = 2^p, 10^p, 2^p - 1$ (např. $k \bmod 2^p$ je prostě posledních p bitů v binárním zápise čísla k, což není moc náhodné)
Vhodné když je m prvočíslo dostatečně vzdálené od mocnin dvojky

Hašování násobením:

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

kde $0<A<1$ je vhodně zvolené číslo (Knuth doporučuje $A \approx (\sqrt{5} - 1)/2 = 0.618033\ldots$) a pokud za m vezmeme mocninu dvojky, tak se dá $h(k)$ relativně snadno spočítat

Univerzální hašování:

Problém: pro každou deterministickou hašovací funkci lze vybrat n klíčů tak, že se všechny zobrazí do stejné adresy v hašovací tabulce (pokud $|U| > n^2$ a tedy $|U| / n > n$)

$\Rightarrow$randomizace

Idea: zvolíme hašovací funkci náhodně a nezávisle na klíčích, které budou hašovány (tj. na konkrétních n klíčích, které budou užity), z nějaké vhodně zvolené množiny funkcí

Platí: $\Rightarrow$ žádný konkrétní vstup (konkrétních n klíčů) není a priori špatný
$\Rightarrow$ opakované použití na stejný vstup volá (skoro jistě) různé hašovací funkce

Definice: Nechť H je konečná množina hašovacích funkcí z univerza klíčů U do množiny $\{0, \ldots ,m-1\}$. Množina H se nazývá univerzální, pokud pro každé dva různé klíče $x,y \in U$ je počet funkcí $h \in H$, pro které $h(x) = h(y)$, roven $|H| / m$.

Pozorování: Pro náhodně zvolenou funkci $h \in H$ je pravděpodobnost kolize dvou náhodně zvolených klíčů $x \neq y$ rovna $1 / m$, což je stejná pravděpodobnost jako když vybereme hodnoty $h(x)$ a $h(y)$ náhodně a nezávisle z množiny $\{0, \ldots ,m-1\}$.

Věta: Nechť h je náhodně vybrána z univerzální množiny hašovacích funkcí a nechť je použita k hašování n klíčů do tabulky velikosti m, kde $n \leq m$. Potom očekávaný počet kolizí, kterých se účastní náhodně vybraný konkrétní klíč x je menší než 1.

Poznámka: Předpoklad $n \leq m$ implikuje, že průměrná délka seznamu (klíčů nahašovaných do stejné adresy) je menší než 1.

Existuje vůbec univerzální množina hašovacích funkcí? A jestli ano, jak ji zkonstruovat?

Konstrukce (jedna z možností): zvolíme prvočíslo m a každý klíč x rozdělíme do $(r+1)$ částí (hodnota r závisí na délce klíčů). Píšeme

$$x = \langle x_0, x_1, \dots , x_r \rangle$$

a r je zvoleno tak, aby maximální hodnota každého x_i byla (ostře) menší než m. Nechť

$$a = \langle a_0, a_1, \dots , a_r \rangle$$

je posloupnost $(r+1)$ čísel náhodně a nezávisle vybraných z množiny $\{0, \dots ,m-1\}$. Nechť

$$h_a(x) = \left(\sum_{i=0}^{r} a_i x_i\right) \bmod m \quad \text{a} \quad H = \bigcup_a \{h_a\}$$

Platí: $|H| = m^{r+1}$ (počet různých vektorů a)

Věta: H je univerzální množina hašovacích funkcí.

Řešení kolizí otevřeným adresováním

• všechny prvky uloženy přímo v hašovací tabulce, faktor naplnění tabulky tedy musí splňovat $\alpha = n/m \leq 1$

• místo sledování ukazatelů v seznamu položek nahašovaných do stejného políčka (který je typicky uložen mimo vlastní tabulku) postupně počítáme adresy políček

• se stejnými paměťovými nároky je hašovací tabulka větší než při řešení kolizí řetězením (ušetří se místo pro pointery)

• posloupnost zkoušených políček závisí na zkoumaném klíči a pořadí pokusu:

$$h : U \times \{0,1, \dots ,m -1\} \rightarrow \{0,1, \dots , m-1\}$$

• pro daný klíč x prohledáváme políčka v hašovací tabulce v pořadí :

$$(h(x,0), h(x,1), \dots , h(x,m -1))$$

a toto pořadí musí tvořit permutaci prvků $\{0,1, \dots , m-1\}$, tj. postupně jsou pro každý klíč x odzkoušena všechna místa v hašovací tabulce

• otevřené adresování dobře podporuje operace Search a Insert, ale implementace operace Delete je složitá (pokud musí být Delete implementován, tak je lepší dát přednost řešení kolizí řetězením)

```
Hash-Search(T,k)              {hledání klíče k v tabulce T}
i := 0;
repeat   j := h(k,i);
         if T[j] = k then return j;        {a konec}
         i := i+1
until (T[j]=NIL) or (i=m);
return NIL
```

```
Hash-Insert(T,k)              {vkládání klíče k do tabulky T}
i := 0;
repeat   j := h(k,i);
         if T[j] = NIL then   T[j] := k;
                              return j;  {a konec}
                        else i:=i+1
until (i=m);
error  přetečení tabulky
```

Při implementaci funkce h se budeme snažit co nejvíce přiblížit uniformnímu hašování: náhodně vybraný klíč má se stejnou pravděpodobností jako svou „posloupnost zkoušek“ kteroukoli z $m!$ permutací prvků $\{0,1, \dots , m-1\}$

Nejčastější techniky pro počítání „posloupnosti zkoušek“ jsou lineární zkoušení

kvadratické zkoušení

dvojité hašování

(žádná z těchto technik nedosahuje vlastnosti uniformního hašování, ale postupně se tomuto předpokladu přibližují).

Lineární zkoušení

používá „obyčejnou“ hašovací funkci $h' : U \to \{0,1, \dots , m-1\}$ pomocí níž definuje

$$h(x,i) = (h'(x) + i) \bmod m$$

<u>Nevýhody:</u>

• pouze m různých posloupností zkoušek, každá posloupnost je jednoznačně definována svojí první pozicí

• vznikají dlouhé shluky (primární clustery) obsazených políček, což zvyšuje čas pro search

Kvadratické zkoušení

Hašovací funkce je $h(x,i) = (h'(x) + ci + di^2) \bmod m$ kde $c \neq 0$ a $d \neq 0$ (kde h' je jako výše).

Parametry c a d musí být zvoleny opatrně aby posloupnost zkoušek byla permutací hodnot z $\{0,1, \dots , m-1\}$. Opět jen m různých posloupností, ale nevznikají primární shluky, pouze sekundární shluky klíčů se stejnou „startovní pozicí“.

Dvojité hašování

Hašovací funkce je $h(x,i) = (h_1(x) + i\, h_2(x)) \bmod m$ kde h_1 a h_2 jsou pomocné hašovací funkce. Vlastnosti

- funkce h_2 musí být volena tak, aby $h_2(x)$ bylo nesoudělné s m (jinak posloupnost zkoušek nebude tvořit permutaci)
- počet různých posloupností zkoušek je zde m^2
- tato metoda je nejlepší a nejvíce se blíží uniformnímu hašování
- příklad (běžných) voleb:

1. $m = 2^p$ (mocnina dvojky) a $h_2(x)$ dává liché číslo (pro každé x), nebo
2. m je prvočíslo a $0 \leq h_2(x) < m$

Analýza hašování s otevřeným adresováním

Věta: V tabulce s otevřeným adresováním s faktorem naplnění $\alpha = n/m < 1$ je očekávaný počet zkoušených pozic při neúspěšném vyhledávání nejvýše roven $1/(1- \alpha)$ (za předpokladu uniformního hašování).

Věta: V tabulce s otevřeným adresováním s faktorem naplnění $\alpha = n/m < 1$ je očekávaný počet zkoušených pozic při úspěšném vyhledávání nejvýše $1/\alpha \ln(1/(1- \alpha)) + 1/\alpha$ (za předpokladu uniformního hašování a pokud je každý klíč vyhledáván se stejnou pravděpodobností).

Euklidův Algoritmus

Algoritmus na spočítání největšího společného dělitele dvou přirozených čísel

Definice: Největší společný dělitel přirozených čísel a,b je největší přirozené číslo, které dělí jak a tak b. Značíme ho $NSD(a,b)$.

Věta: Nechť a,b jsou přirozená čísla. Pak $NSD(a,b)$ je nejmenší kladný prvek množiny $L = \{ax + by \mid x,y \in \mathbf{Z}\}$

Důsledek: Nechť a,b jsou přirozená čísla. Pokud d je přirozené číslo, které dělí a i b, tak d dělí také $NSD(a,b)$.

Věta: Nechť a,b jsou přirozená čísla, kde $b>0$. Pak $NSD(a,b) = NSD(b, a \bmod b)$.

```
EUCLID(a,b)

if b=0 then Return(a)

        else Return(EUCLID(b, a mod b))
```

Lemma: Nechť $a > b \geq 0$ a EUCLID(a,b) udělá $k \geq 1$ rekurzivních kroků. Pak $a \geq F(k+2)$ a $b \geq F(k+1)$, kde $F(i)$ je i-té Fibonacciho číslo.

Důsledek (Lamého věta): Nechť $a > b \geq 0$ a $F(k) \leq b < F(k+1)$. Pak EUCLID(a,b) udělá nejvýše $k - 1$ rekurzivních kroků.

Věta (bez Dk – viz AVL stromy): $F(k) = \Theta(\varphi^k)$, kde $\varphi = (1+\sqrt{5})/2$ (což je tzv. „zlatý řez“).

Důsledek: Nechť $a > b \geq 0$. Pak EUCLID(a,b) udělá nejvýše $O(\log b)$ rekurzivních kroků.

Pozorování: Pokud a,b jsou dvě nejvýše t-bitová binární čísla, tak EUCLID(a,b) provede $O(t)$ rekurzivních kroků a v každém z nich $O(1)$ aritmetických operací na (nejvýše) t-bitových číslech, tj. $O(t^3)$ bitových operací, pokud předpokládáme, že každá aritmetická operace na t-bitových číslech potřebuje $O(t^2)$ bitových operací (což je snadné ukázat). EUCLID je tedy polynomiální algoritmus vzhledem k velikosti vstupu.