3. Stromy

Hašování je velmi efektivním nástrojem pro reprezentaci množin a řešení klasického slovníkového problému. V praxi se však velmi často setkáváme s univerzy, která jsou přirozeným způsobem totálně uspořádaná a při jejichž užití se často vyskytují operace založené na tomto uspořádání (hledání nejmenšího, resp. největšího prvku apod.). Hašování ale případné uspořádání univerza ignoruje a důsledkem toho je, že operace založené na uspořádání často nejdou realizovat. Existují sice modifikace hašovacích metod, které využívají uspořádání univerza a umožňují provádět operace na něm založené, ale obvykle za cenu ztráty jednoduchosti. Tato nevýhoda hašování spolu s faktem, že může být velmi neefektivní při nerovnoměrném rozložení vstupních dat, vedla v sedmdesátých letech minulého století k velkému zájmu o datové struktury založené na stromech. Zatímco hašování používá hlavně metody z teorie čísel, stromové struktury používají kombinatorické metody. Navíc umožňují rozšířit základní úlohu (tj. slovníkový problém) o několik operací založených na uspořádání univerza. Tuto rozšířenou úlohu budeme nazývat uspořádaný slovníkový problém a formálně ji popíšeme v úvodní sekci této kapitoly.

Druhá sekce bude věnována reprezentaci uspořádaným polem. Tato struktura je podobnou modifikací klasického pole jako je uspořádaný seznam modifikací obyčejného spojového seznamu. V uspořádaném poli lze efektivně realizovat operaci **MEMBER**, ale efektivní realizace operací **INSERT** a **DELETE** nejsou známy. Oproti tomu binární vyhledávací stromy, které jsou přirozeným zobecněním vyhledávání v uspořádaném poli, umožňují i tyto operace. Tyto stromy jsou rovinným znázorněním binárního vyhledávání v uspořádaném poli a splňují-li navíc podmínky tzv. vyváženosti, pak operace **MEMBER**, ale i **INSERT** a **DELETE** v nich vyžadují čas $O(\log |S|)$, kde S je reprezentovaná množina. Jak uvidíme, při reprezentaci uspořádaným polem existují pro operaci **MEMBER** prakticky výhodnější metody – interpolační vyhledávání nebo zobecněné kvadratické vyhledávání. Tyto metody mají očekávaný čas $O(\log \log |S|)$ a zobecnit je pro stromy je velmi komplikované. Navíc získat analogické výsledky se zatím nepovedlo. Stromy se budeme zabývat ve třetí sekci. Nejprve ukážeme algoritmy pro obecné binární vyhledávací stromy a pak se zaměříme na vyvážené binární vyhledávací stromy, jmenovitě na červeno-černé stromy a AVL-stromy. Poslední dvě sekce budou věnovány (a, b)-stromům (jejichž speciálním případem jsou B-stromy často používané při návrhu databází). Konkrétně ve čtvrté sekci se seznámíme se základními fakty o (a, b)-stromech a v poslední páté sekci s jejich použitím.

3.1 Uspořádaný slovníkový problém

Mějme totálně uspořádané univerzum $(U, \leq)$. To znamená, že $\leq$ je uspořádání na univerzu U takové, že pro každé dva různé prvky $u, v \in U$ platí buď $u < v$ nebo $v < u$. Řešit uspořádaný slovníkový problém znamená reprezentovat množinu $S \subseteq U$ a navrhnout algoritmy pro následující operace:

- (•) **MEMBER**(x, S) – zjistí, zda prvek $x \in U$ patří do reprezentované množiny S, případně vrátí pozici x v reprezentaci S;
- (•) **INSERT**(x, S) – vytvoří reprezentaci množiny $S \cup \{x\}$;
- (•) **DELETE**(x, S) – vytvoří reprezentaci množiny $S \setminus \{x\}$;
- (•) **MIN**(S) – nalezne nejmenší prvek v S;
- (•) **MAX**(S) – nalezne největší prvek v S;
- (•) **SPLIT**(S, x) – zkonstruuje reprezentace dvou množin $S_1 = \{s \in S \mid s < x\}$ a $S_2 = \{s \in S \mid s > x\}$ a oznámí, zda $x \in S$;
- (•) **JOIN** – používá se ve dvou verzích:
  - a) **JOIN2**(S_1, S_2) – pro dané reprezentace množin S_1 a S_2, které splňují $\max S_1 < \min S_2$, vytvoří reprezentaci množiny $S = S_1 \cup S_2$;
  - b) **JOIN3**(S_1, x, S_2) – pro dané reprezentace množin S_1 a S_2 a prvek $x \in U$, které splňují $\max S_1 < x < \min S_2$, vytvoří reprezentaci množiny $S = S_1 \cup \{x\} \cup S_2$.

Přitom se mlčky předpokládá, že výsledkem vyhledávacích operací je i adresa dat spojených s klíčem hledaného prvku a že aktualizační operace **INSERT** a **DELETE** vytvoří (resp. uvolní) prostor pro prvek, který je jejich argumentem, i pro data spojená s jeho klíčem. Operace **SPLIT** a **JOIN** vytvářejí nové stromy, původní stromy (argumenty těchto operací) přitom zanikají. Algoritmy realizující tyto operace navíc musí zajistit i správné fungování operací pro nově vytvořené stromy.

Je vidět, že operace **JOIN2** a **JOIN3** se dají pomocí operací **INSERT** a **DELETE** převést jedna na druhou. Operaci **JOIN3**(S_1, x, S_2) lze realizovat například tak, že nejprve vytvoříme $S =$ **JOIN2**(S_1, S_2) a potom provedeme **INSERT**(x, S). Naopak provedení operace **JOIN2**(S_1, S_2) se dá rozložit na posloupnost příkazů $x =$ **MIN**(S_2), **DELETE**(x, S_2), $S =$ **JOIN3**(S_1, x, S_2). Proto často budeme popisovat jen jednu z nich.

Při operacích **JOIN2**(S_1, S_2) (respektive **JOIN3**(S_1, x, S_2)) vždy budeme předpokládat, že $\max S_1 < \min S_2$ (respektive $\max S_1 < x < \min S_2$), tento předpoklad ale algoritmy nebudou ověřovat (to znamená, že by ho měl uživatel ověřit sám ještě před spuštěním těchto operací).

Občas se také používá operace

- (•) **ORD**(k) – pro $k \leq |S|$ nalezne k-tý nejmenší prvek v S.

Zřejmě operace **MIN** a **MAX** jsou speciálním případem operace **ORD**(k), konkrétně **MIN** je operace **ORD**(1) a **MAX** je operace **ORD**$(|S|)$.

Při návrhu datové struktury a při popisu algoritmů se, stejně jako při hašování, omezíme na způsob uložení klíče a na práci s ním.

3.2 Vyhledávání v uspořádaném poli

Jednou z nejzákladnějších datových struktur pro reprezentaci množiny je pole, které jsme popsali v podsekci 1.1.1. Základní verze této struktury nevyužívá dodatečné informace o univerzu, speciálně jeho uspořádání. Alternativou pro uspořádaná univerza je uspořádané pole. Je to velmi stará datová struktura, která se používala už v době před počítači – jako příklad lze uvést kartotéku se setříděnými záznamy. Uspořádané pole umožňuje aplikaci řady různých postupů. Zde se seznámíme se základním algoritmem, který má několik verzí lišících se v tom, jakou používají proceduru k určení dalšího testu. Právě tato procedura hraje zásadní roli v analýze jeho složitosti.

Mějme totálně uspořádané univerzum U a jeho podmnožinu $S \subseteq U$ velikosti n reprezentujme polem $P[1..n]$ tak, že $S = \{P(i) \mid i = 1, 2, \ldots, n\}$ a pro $i < j$ platí $P(i) < P(j)$. V takto reprezentované množině můžeme vyhledávat efektivnějším způsobem než je systematické prohledání hrubou silou, které bylo prezentováno v podsekci 1.1.1.

3.2.1 Metaalgoritmus

Algoritmus, který má zjistit, zda $x \in S$, nejprve testuje vztah mezi x a $P(1)$ a mezi x a $P(n)$. Pokud $x < P(1)$ nebo $P(n) < x$, pak x není prvkem S. Naopak, když $x = P(1)$ nebo $x = P(n)$, pak $x \in S$. Nenastane-li žádná z těchto možností, pak pro hodnoty $d = 1$ a $h = n$ platí $d < h$ a $P(d) < x < P(h)$. V tomto případě buď $n = 2$ a $x \notin S$ nebo $d + 1 < h$. Nyní algoritmus zavolá jako proceduru funkci **Next**(d, h), která vrátí celé číslo k takové, že $d + 1 \leq k \leq h - 1$, a porovná x a $P(k)$. Když $x = P(k)$, pak $x \in S$. Když $x < P(k)$, pak položí $h = k$, když $x > P(k)$, položí $d = k$. Po tomto kroku je tedy buď $x \in S$, nebo $d + 1 < h$ a $P(d) < x < P(h)$, nebo $d + 1 = h$. Zřejmě v posledním případě $x \notin S$, protože pro žádný prvek $s \in S$ neplatí $P(d) < s < P(h)$. Pokud $d + 1 < h$, pak se celý proces opakuje. Protože $h - d$ je vždy celé číslo, které je na počátku rovno $n - 1$, a protože tento postup vždy zmenší jeho hodnotu, tak algoritmus vždy po konečném počtu kroků skončí.

Formální zápis algoritmu

MEMBER(x):
if $x = P(1)$ **then**

```
   Výstup: x ∈ S, stop
else
   if x < P(1) then Výstup: x ∉ S, stop else d = 1 endif
endif
if x = P(|S|) then
   Výstup: x ∈ S, stop
else
   if x > P(|S|) then Výstup: x ∉ S, stop else h = |S| endif
endif
while d + 1 < h do
   k := Next(d, h)
   if x = P(k) then
      Výstup: x ∈ S, stop
   else
      if x < P(k) then
         h := k
      else
         d := k
      endif
   endif
enddo
Výstup: x ∉ S
```

Pro tento algoritmus je podstatné, že funkce **Next**(d, h) vrací celé číslo k takové, že $d < k < h$ (takové číslo k existuje, protože v okamžiku volání **Next** je $d+1 < h$). Jeho korektnost plyne z pozorování, že buď už našel správné řešení, nebo d a h jsou celá čísla taková, že $1 \leq d < h \leq n$ a $P(d) < x < P(h)$. Efektivita závisí na funkci **Next**. Zpracování dotazu, tj. porovnání hodnot x a $P(k)$, vyžaduje čas $O(1)$, a pokud i provedení funkce **Next** vyžaduje čas $O(1)$, pak čas celého algoritmu je úměrný počtu dotazů, a ten je roven počtu volání funkce **Next**.

Uvedeme čtyři nejpoužívanější verze procedury **Next** – první tři popíšeme zde, čtvrté věnujeme následující podsekci.

a) Unární vyhledávání. Zde **Next**$(d, h) = d+1$. Každý dotaz zvětší d o 1, a tedy maximální počet dotazů je $|S|$. Algoritmus v nejhorším případě vyžaduje čas $O(|S|)$ a protože očekávaný počet dotazů při rovnoměrném rozdělení vstupních dat je $\frac{|S|}{2}$, je i očekávaný čas $O(|S|)$.

b) Binární vyhledávání. Zde **Next**$(d, h) = \lceil \frac{d+h}{2} \rceil$. Abychom odhadli počet dotazů, tak si nejprve všimněme, že největší počet dotazů se položí, když $P(1) < x < P(n)$ a $x \notin S$. Dále, když po i dotazech (kde $i > 2$) platí $h - d = q$, pak po $i-1$ dotazech muselo platit $2q - 1 \leq h - d \leq 2q + 1$. Předpokládejme, že algoritmus skončil po m dotazech. Když algoritmus končí, pak platí $h - d = 1$, a tedy po předposledním dotazu muselo platit $h - d \geq 2 = 2^{m-(m-1)-1} + 1$. Z předchozí úvahy plyne, že když po i

dotazech platí $h - d \geq 2^{m-i-1} + 1$, pak po $i - 1$ dotazech muselo platit

$$h - d \geq 2(2^{m-i-1} + 1) - 1 = 2^{m-(i-1)-1} + 1.$$

Tedy indukcí pro $i = m, m-1, \ldots, 3, 2$ dostáváme, že po i dotazech platí $h - d \geq 2^{m-i-1} + 1$. Protože po 2 dotazech je $h - d = n - 1$, tak $2^{m-3} + 1 \leq n - 1$. Odtud po zlogaritmování dostaneme, že počet dotazů je nejvýše $3 + \log(|S| - 2)$. Algoritmus tedy v nejhorším případě vyžaduje čas $O(\log |S|)$ a jeho očekávaný čas při rovnoměrném rozložení vstupních dat je také $O(\log |S|)$.

c) Interpolační vyhledávání. V tomto případě předpokládáme číselné univerzum a používáme funkci $\mathbf{Next}(d, h) = d + \lceil \frac{x - P(d)}{P(h) - P(d)}(h - d) \rceil$. V nejhorším případě musíme položit více než $\frac{|S|}{2}$ dotazů, a proto čas v nejhorším případě je $O(|S|)$. Očekávaný čas při rovnoměrném rozložení vstupů je $O(\log \log |S|)$ (důkaz neuvádíme).

Poznámka. V unárním vyhledávání můžeme také postupovat duálně, to znamená, že $\mathbf{Next}(d, h) = h - 1$, a výsledky se nezmění. Obecně se mluví o unárním vyhledávání v případě, kdy existuje konstanta $c \geq 1$ taková, že buď $d < \mathbf{Next}(d, h) \leq d + c$ nebo $h - c \leq \mathbf{Next}(d, h) < h$. Pak se výsledky změní jen o multiplikativní konstantu c. Jak uvidíme později, toto zobecnění je výhodné použít při různých aplikacích. O konstantě c se pak mluví jako o délce kroku.

Interpolační vyhledávání je založeno na faktu, že hodnota **Next** závisí i na velikosti x. Když x je velké, je hodnota **Next** posunuta do větších hodnot, když x je malé, pak je posunuta do menších hodnot.

3.1 Poznámka. Když rozložení prvků není rovnoměrné, ale je známé, pak podle toho můžeme v interpolačním vyhledávání upravit funkci **Next** a v tom případě se odhad očekávaného času algoritmu nezmění. Podrobnosti vynecháváme.

3.2.2 Zobecněné kvadratické vyhledávání

Ukázat, že očekávaný čas interpolačního vyhledávání je $O(\log \log n)$, je dosti náročné. V této podsekci uvedeme jiné vyhledávání, které má podobné charakteristiky jako interpolační (a je také jen pro číselná univerza), ale k důkazu jeho vlastností jsou zapotřebí jen elementární znalosti (je založen na Čebyševově nerovnosti).

Pro zobecněné kvadratické vyhledávání je funkce $\mathbf{Next}(d, h)$ definována složitější procedurou, jejíž výsledek závisí i na předchozích dotazech a odpovědích na ně. Procedura pracuje v blocích a dotazy zadané v rámci téhož bloku jsou mezi sebou korelované. První dotaz v bloku je interpolační a procedura přitom zjistí, zda x je menší nebo větší než hodnota v dotazu, a stanoví velikost kroku pro další dotazy. Pak střídá unární a binární dotazy do té doby, než nalezne hledaný