Kapitola 1

Státnice - Dynamizace datových struktur

1.1 Úvod

Mnoho datových struktur podporuje velice efektivní operaci **MEMBER**, ale nemá operace **INSERT** a **DELETE**. Úkolem dynamizace je právě tyto operace do jakékoliv obecné struktury co nejefektivněji doplnit.

Zobecněný vyhledávací problém

Zobecněný vyhledávací problém je libovolná funkce $f : U_1 \times 2^{U_2} \to U_3$ – pro prvek a nějakou množinu vrací nějakou hodnotu.

Struktura $\mathcal{S}$ je statická struktura řešící vyhledávací problém (neboli S-struktura) f, je-li dán algoritmus, který pro $A \subseteq U_2$ zkonstruuje datovou strukturu $\mathcal{S}(A)$ a algoritmus, který pro $x \in U_1$ a $\mathcal{S}(A)$ spočte $f(x, A)$.

Struktura je semidynamická, pokud navíc existuje algoritmus, který pro $x \in U_2$ a $\mathcal{S}(A)$ zkonstruuje S-strukturu $\mathcal{S}(A \cup \{x\})$.

Struktura je dynamická, pokud navíc existuje algoritmus, který pro $x \in A$ a $\mathcal{S}(A)$ zkonstruuje S-strukturu $\mathcal{S}(A \backslash \{x\})$.

Ve všech následujících operacích budeme uvažovat jen rozložitelný vyhledávací problém:

- Vyhledávací problém je rozložitelný, pokud existuje binární operace $\circ$ na univerzu U_3 taková, že pro disjunktní $A, B \subseteq U_2$ a $x \in U_1$ platí $f(x, A) \circ f(x, B) = f(x, A \cup B)$.

To platí pro situaci, která nás nejvíc zajímá – pokud funkce f provádí operaci **MEMBER** nad nějakou datovou strukturou. Jiné problémy (patří x do konvexního obalu množiny?) ale rozložitelné nejsou.

Navíc budeme předpokládat, že funkce $\circ$ lze spočítat v konstantním čase. Označme

- $Q_{\mathcal{S}}(n)$ čas na vyčíslení $f(x, A)$, pokud $|A| = n$
- $S_{\mathcal{S}}(n)$ paměťový prostor potřebný k reprezentaci n-prvkové množiny
- $P_{\mathcal{S}}(n)$ čas na vytvoření struktury $\mathcal{S}(A)$ pro $|A| = n$

Pro asymptotické odhady budeme předpokládat, že funkce $Q_{\mathcal{S}}(n)$, $\frac{S_{\mathcal{S}}(n)}{n}$ a $\frac{P_{\mathcal{S}}(n)}{n}$ jsou neklesající.

1.2 Semidynamizace

Vytvoříme semidynamickou strukturu, jejíž čas operace **INSERT** bude velmi rychlý a navíc se nám moc nepokazí doba pro provedení operace **MEMBER**. Základní idea je podobná jako v binomiálních haldách – roztrháme reprezentovanou množinu A na sadu disjunktních podmnožin A_i o velikosti 2^i (takové množiny jsou neprázdné pro i odpovídající jedničkám v binárním zápisu čísla $|A|$.

Formálně budeme mít spojový seznam S-struktur $\mathcal{S}(A_i)$ odpovídajících množinám A_i, potom seznam spojových seznamů $s(A_i)$, které obsahují prvky jednotlivých množin, a nakonec pomocnou dynamickou strukturu T (např. binární vyhledávací strom). To všechno dohromady zjevně vyžaduje jen $O(S_{\mathcal{S}}(n))$ paměti (celkem reprezentujeme pořád stejně prvků a vyhledávací stromy i spoják jsou $O(n)$).

Struktura T se používá k "rychlému" testování před **INSERT**em, zda nechceme vkládat prvek, který už v množině máme, případně před **DELETEM**, jestli nechceme mazat neexistující prvek. Pro jednoduchost to v popisech operací vynecháme.

Operace MEMBER

Operace **MEMBER** potom projde všechny neprázdné množiny A_i a zavolá **MEMBER** na nich. Výsledek spojí za pomoci funkce $\circ$.

To celkem dává složitost **MEMBER**u $O(Q_{\mathcal{S}}(n)\log(n))$, protože hledáme v max. $\log_2$ dílčích množinách.

Protože jakékoliv mocniny rostou rychleji než logaritmus, platí, že pokud $Q_{\mathcal{S}} = n^{\epsilon}$ pro nějaké $\epsilon > 0$, pak je operace **MEMBER** $O(n^{\epsilon})$.

Operace INSERT

Operace **INSERT** najde nejmenší i takové, že $A_i = \emptyset$. Potom vezme všechny $A_j, j < i$ (ty jsou neprázdné) a spolu s vkládaným prvkem je slije do jedné množiny A_i (zahodí S-struktury pro všechna $A_j, j < i$, zkonkatenuje spojáky $s(A_j)$ a na jejich základě zkonstruuje novou S-strukturu $\mathcal{S}(A_i)$). Nová množina má správný počet prvků, protože $\sum_{j=0}^{i-1} 2^j + 1 = 2^i$.

V odhadu amortizované složitosti využijeme fakt, že S-struktura pro A_i se tvoří jen tehdy, když existují S-struktury pro všechny $A_j, j < i$. Tj. S-struktura o velikosti 2^i prvků se konstruuje znovu až po dalších $2^{i+1} - 1$ **INSERT**ech. Amortizovaně tak vychází :

$$\sum_{i=0}^{\log n} \frac{P_{\mathcal{S}}(2^i)}{2^i} \leq \sum_{i=0}^{\log n} \frac{P_{\mathcal{S}}(n)}{n} = \frac{P_{\mathcal{S}}(n)}{n} \log n = O(\frac{P_{\mathcal{S}}(n)}{n} \log n).$$

Podobně jako pro **MEMBER** platí, že pokud $P_{\mathcal{S}} = n^{\epsilon}$ pro nějaké $\epsilon > 1$, pak je složitost operace **INSERT** $O(n^{\epsilon-1})$.

Operace INSERT se zaručeným nejhorším časem

Protože někdy nestačí mít **INSERT** rychlý amortizovaně, byla vytvořena jeho varianta, která zaručuje rozumnou rychlost i v nejhorším případě. Dělá se to prostým rozložením potřebných operací do všech volání. Musíme ale povolit i požadavky na naši strukturu – místo max. jedné množiny pro každou velikost odpovídající mocnině dvou budeme mít max. 4 takové množiny $A_{i,0}, \ldots A_{i,3}$, z nichž poslední navíc bývá zkonstruovaná jen částečně ("rozpracovaná").

Označme počet množin A_i jako k_i, přičemž $k_i = -1$, pokud neexistuje žádná množina velikosti 2^i. Algoritmus potom vypadá následovně:

1. Vytvoří se jednoprvková $A_{0,k_0+1} = \{x\}$ a zvýší k_0 (tj. i odpovídající S-struktura)
2. Pro první souvislý úsek, kde $k_i \geq 0$:
   (a) Provedeme dalších $\frac{P_{\mathcal{S}}(2^i)}{2^i}$ konstrukce S-struktury $\mathcal{S}(A_i, k_i)$
   (b) Pokud jsme tím tuto S-strukturu dotvořili, vezmeme první dvě struktury "o patro níž" ($A_{i-1,0}$ a $A_{i-1,1}$) a připravíme je prvním krokem k sloučení. Zároveň je odtud odebereme a přečíslujeme zbylé. Jejich započaté sloučení přidáme na úroveň i.
3. Pokud jsme v posledním kroku (i, t.ž. $k_{i+1} = 0$) vytvořili druhou strukturu stejné velikosti na nejvyšším dosaženém "patře", připravíme první krok sloučení těchto dvou největších struktur, přemístíme jejich započaté sloučení "o patro výš" a odebereme původní struktury.

Díky tomu, že počet kroků volíme tak šikovně, vždycky dotvoříme strukturu právě včas na to, abychom mohli vytvářet další stejné velikosti. Navíc, protože se provede jen $O(\frac{P_{\mathcal{S}}(2^i)}{2^i})$ kroků pro každé i, odpovídá složitost v nejhorším případě amortizované složitosti původní operace **INSERT**.

1.3 Dynamizace

Při úplné dynamizaci navíc požadujeme efektivní algoritmus **DELETE**. Přidáme další předpoklad, a to, že na naší původní S-struktuře existuje operace **FAKE-DELETE** (ta spočívá v označení nějakého prvku jako "smazaného", aniž by se smazal skutečně – dál tedy zabírá místo a prodlužuje operace).

Základní ideou oproti semidynamizaci navíc je, že se budeme snažit za každou cenu zajistit, aby všechny naše pomocné S-struktury stále obsahovaly alespoň osminu "nesmazaných" prvků a přestavovat je až poté, co počet "smazaných" překročí tuto mez. Formálně naše S-struktury reprezentují vlastně množiny B_i, kde $B_i = A_i \backslash \{x_i; i = 1, 2 \ldots k\}$ (a $2^{i-3} < |A_i| \leq 2^j$).

Asymptoticky tak zaručíme, že množin (a pomocných S-struktur) bude stále jen logaritmicky mnoho a paměťové nároky taky zůstávají asymptoticky stejné. Struktura je tedy stejná jako pro (základní) dynamizaci. Navíc jsou nyní spojové seznamy prvků přímo provázané s S-strukturami (protože už nepoznáme podle velikosti seznamu, ke které S-struktuře vlastně patří).

Operace **MEMBER** pak pracuje úplně stejně, jen nakonec zkontroluje, zda nalezený prvek nebyl označen jako "smazaný" (a případně ho pak nevrátí).

Operace INSERT

Algoritmus operace **INSERT**(x) je následující:

1. Najdi nejmenší j takové, že $|\cup_{i\leq j} A_i| < 2^j$.
2. Vytvoříme $A_j = \cup_{i\leq j} A_i \cup \{x\}$ (včetně S-struktury a spojáku). To splňuje požadavky na velikost $2^{j-1} < |A_j| \leq 2^j$, jinak by pro j nebylo nejmenší.
3. Položíme $A_i = \emptyset$ pro $i < j$.

Díky zachovávané minimální velikosti vytvářené množiny máme složitost amortizovaně stejnou jako v případě semidynamizace.

Operace DELETE

Odebrání prvku x z naší dynamické struktury vypadá následovně:

1. Odstraníme x ze spojáku
2. Vyřešíme čtyři různé případy podle velikosti množiny A_i, která obsahuje prvek x:
   (a) Je-li A_i jen jednoprvková, prostě zahodím struktury s ní spojené.
   (b) Je-li $|A_i| > 2^{i-3}$ (tj. ještě mám víc než osminu prvků "platných"), provedeme pouze **FAKE-DELETE**.
   (c) Je-li A_{i-1} buď prázdná, nebo dost velká ($|A_{i-1}| > 2^{i-2}$), můžeme ji s A_i prohodit. Pro "nové" A_{i-1} pak vytvoříme novou S-strukturu.
   (d) Je-li A_{i-1} neprázdná, ale moc malá na prohození s A_i, potom je určitě můžeme sloučit a získáme novou množinu, která splňuje velikostní omezení pro A_i. Pro ni vyrobíme novou S-strukturu.

Amortizovaná složitost celé operace **DELETE** je $O(D_{\mathcal{S}}(n) + \log n + \frac{P_{\mathcal{S}}(n)}{n})$, kde $D_{\mathcal{S}}(n)$ je čas potřebný na operaci **FAKE-DELETE** a $\log n$ je potřeba na vyhledání prvku. Odhadneme, kolikrát se dělá skutečné přestavění S-struktur. Pokud $x \in A_i$, pak **DELETE** může nově vytvořit jen $S(A_i)$ nebo $S(A_{i-1})$. Z jejich podmínek velikosti (v algoritmu) vidíme, že mezi dvěma **DELETE** pro stejné A_j (přičemž $|A_j| \leq 2^{i-2}$) se musí provést aspoň 2^{j-3}-krát **FAKE-DELETE** (a libovolný počet **DELETE** na jiných množinách). Amortizovaná složitost vytváření S-struktur tak vychází:

$$\frac{P_{\mathcal{S}}(2^{j-2})}{2^{j-3}} = O(\frac{P_{\mathcal{S}}(n)}{n})$$

I se současně prováděnými operacemi **DELETE** se amortizovaná složitost operací **INSERT** zachovává. Aby **INSERT** vytvořil podruhé strukturu pro A_i, musí opět nastat $1 + \sum_{j\leq i} |A_j| > 2^{j-1}$. Protože **DELETE** nikdy nevytvoří strukturu s víc než polovinou skutečně zaplněnou, musí se do té doby provést aspoň 2^{i-2} **INSERT**ů, čímž získáme stejnou situaci jako u semidynamizace.