Povídání o datových strukturách (a algoritmech)

Dr. Vladan Majerech

Katedra teoretické informatiky a metematické logiky

Matematicko Fyzikální fakulta

Karlovy university

2006

1. Předmluva

Povídání o datových strukturách jsem chtěl napsat již dříve. Vzhledem k malému počtu posluchačů přednášek o dynamických datových strukturách a rychlému rozvoji oboru jsem od psaní skript o dynamických datových strukturách opustil. Nyní je obor poměrně stabilizován. Cvičení k přednášce algoritmy a datové struktury bylo rozhodujícím podnětem.

Povídání o datových strukturách je mnohem efektivnější v kontextu algoritmů. V tomto se obsah povídání bude prolínat s obsahem skript „Úvod do složitosti a NP-úplnosti“. Závěrečné kapitoly budou věnovány dynamickým datových strukturám. Toto povídání nejsou skripta k „Algoritmům a datovým strukturám“, ačkoli se tématicky s obsahem přednášky velice překrývají. Čtenář si brzy všimne, že v textu jsou „implementační detaily“ uváděny výjimečně. Úroveň abstrakce nad těmito detaily je v tomto povídání vyšší než obvykle. Obávám se proto, že povídání je třeba brát spíše jako rozcestník k detailnějším popisům načrtnutých datových struktur.

2. Úvod

Datové struktury jsou často popisovány způsobem připomínajícím biologii. Stromy mají takovýto tvar, listy mají takovouto barvu .... Můžeme je pěstovat tak, že vložíme list na nalezené místo, případně můžeme list utrhnout. Neříkám, že je to špatně, měli bychom získat i tyto informace, ale dle mého názoru je potřeba vědět především, k čemu se daná datová strukruta hodí.

Dobrý programátor nepracuje tak, že se rozhodne použít danou datovou strukturu a pak začne zjišťovat, k čemu se hodí. Programátor řeší nějaký problém, má pro něj abstraktní algoritmy (popsané matematicky) a snaží se zvolit vhodnou reprezentaci pro matematické objekty v algoritmech. To jakou reprezentaci (datovou strukturu) programátor zvolí závisí od toho, jaké operace s daným matematickým objektem v algoritmech provádíme.

Také závisí od toho, zda zpracovávaná data jsou krátkodobého charakteru (krátký jednorázový výpočet) nebo dlouhodobého (databáze klientů společnosti). V prvním případě nejrychlejší algoritmy, pracují v čase úměrném velikosti zpracovávaných dat, datové struktury vytváříme dočasně v krátkodobé paměti a na konci výpočtu je promažeme. Ve druhém případě naopak postupně budujeme dlouhodobě data(bázi) v dlouhodobé paměti (např. disková pole) a pravidelně používané algoritmy budou *Dotazy* zjišťující informace týkající se pouze malé části databáze. Čas nejrychlejší odpovědi může být úměrný velikosti odpovědi. Nově definované *Dotazy* nad takovouto databází mohou probíhat v čase úměrném velikosti celé databáze.

Pro dlouhodobé uchovávání dat je nejdůležitější dvojice operací **Write** resp. **Read**, vkládající nový prvek do databáze či přepisující starý resp. navracející dříve vložený prvek či informaci, že v databázi požadovaný prvek není. Abychom mohli jednoznačně identifikovat prvek v databázi, typicky má jednoznačný „klíč“. Klíč je určen volajícím algoritmem a předán operaci **Write** jako jeden z parametrů. Druhým parametrem je informace, která má být pod tímto klíčem dostupná. **Read** pak dostane jako parametr klíč a vrátí příslušnou hodnotu. Operace **Delete**, odstraňující prvek z databáze, již není nepostradatelná. *Dotazy* v případě databází většinou identifikují požadovaná data nikoli dle klíče, ale především dle obsahu. Hodnoty těchto identifikujících položek již mohou být v databázi zastoupeny vícekrát. Pokud vnější pravidlo garantuje neopakovatelnost identifikující položky, můžeme danou položku vnímat jako „sekundární klíč“, při porušení jednoznačnosti sekundárního klíče pak databáze bude hlásit chybu. Pro vícenásobně zastoupené identifikující položky je přirozené popužívat *Intervalové dotazy*. Tyto *Dotazy* vracejí všechny záznamy (či klíče záznamů) kde identifikující položka je v uvedeném rozsahu hodnot. Tento rozsah nemusí být jedna hodnota, ale interval hodnot daného lineárního uspořádání hodnot.

Já se poměrně zřídka setkávám s databázemi, kde by kromě *Intervalových dotazů* a operací **Read** byly potřeba jiné *Dotazy* na data. Přesto výzkum databází tohoto druhu stojí za pozornost viz „Dynamické datové struktury“. Příkladem užití takovýchto databází může být vyhledávání spojení v jízdních řádech či v mapách.

Tytéž požadavky na datové struktury jsou často kladeny i u jednorázových výpočtů, tam ale většinou neočekáváme tak rozsáhlá data. V takovém případě nemusejí být způsoby uložení dat a práce s nimi optimalizovány pro blokový (diskový) přístup. Mnohem častěji jsou používány operace **Delete** a operace spojující reprezentované množiny.

Jednou z nejrozšířenějších metod výpočtu je „Hladový algoritmus“. Ten pracuje tak, že definuje nějaké uspořádání, udržuje množinu prvků, které mají být zpracovány. Postupně vždy vybere minimální prvek v daném uspořádání, provede s tímto prvkem požadované akce a označí tento prvek za již zpracovaný. Požadované akce často zařazují další prvky do množiny pro zpracování, mnohdy začínáme s jednoprvkovou množinou. Hladový algoritmus pro svoji implementaci vyžaduje operace **Init**, **Insert**, **FindMin**, **DeleteMin**, **Destroy**. **Init** předpřipraví danou množinu prvků pro zpracování, **Insert** vloží hodnotou v uspořádání a (ukazatel na) reprezentovaný prvek, **FindMin** vrátí (ukazatel na) prvek ke zpracování s minimální hodnotou v uspořádání, **DeleteMin** odstraní (ukazatel na) minimální prvek z množiny prvků ke zpracování, **Destroy** odstraní množinu prvků (či ukazatelů na prvky) pro zpracování z paměti. Hladový algoritmus může mít varianty, kdy vznikne několik množin ke zpracování a ty jsou v průběhu práce sjednoceny. K tomu se hodí operace **Meld** spojující množiny prvků (či ukazatelů na prvky) pro zpracování. Některé varianty hladového algoritmu mohou zmenšovat hodnoty prvků které již byly určeny pro zpracování. K tomu se hodí operace **Decrement**, která sníží hodnotu určeného prvku. **Decrement** je operace vyžadující určit prvek, jehož hodnota má být snížena, proto při používání funkce **Decrement** musí volající algoritmus dostat od operací **Init**, **Insert** zpět klíče (ukazatele) určené

k identifikaci. Obdobný problém by nás čekal při implementaci operace **Delete**, která by se mohla hodit k odstranění libovolného prvku z množiny určené ke zpracování.

Při zpracovávání textů je často potřeba efektivně vyhledávat určitá klíčová (pod)slova (kompilátory, dolovače dat). Pro takovéto aplikace hraje rozklad textu na písmena abecedy zásadní význam a může být rozhodující při volbě datových struktur. Také je třeba zvážit, jak moc bude v průběhu zpracování měněna množina vyhledávaných slov.

Někdy nás zajímá celkový čas práce datové struktury, jindy optimalizujeme čas nejpomalejší operace (zdravotnictví). V případě, kdy nás zajímá celkový čas, je důležité zjistit, jak často budou vykonávány jednotlivé operace.

Příklad 1: Máme vyhodnotit shodu dvou posloupností celých čísel (od 0 do K) stejné délky N pravidly hry logik:

Jednorázový algoritmus, proto nás bude zajímat celkový čas. Potřebujeme spočítat počet shod na stejných pozicích a k tomu potřebujeme projít současně obě posloupnosti a hodnoty se stejnými indexy porovnat (rychleji než $O(N)$ to nepůjde). Dále potřebujeme spočítat počet shod na nesprávných pozicích (spočteme nezávisle na správnosti pozice a odečteme počet shod se shodou polohy). Potřebujeme porovnat dvě „množiny s četností“. K tomu nám stačí operace **Write** a **Read**, kde klíčem budou čísla posloupností a uložená hodnota bude o kolik víc hodnot daného klíče bylo dosud evidováno v první posloupnosi než v druhé. Nejprve projdeme první posloupnost a provedeme N dvojic **Read**, **Write**. Pak projdeme druhou posloupnost a opět provedeme N krát **Read** (a v případě nalezení nenulové hodnoty zaznamenáme shodu a hodnotu snížíme pomocí **Write**). Celkem provedeme $O(N)$ operací **Read** a **Write**. Při optimální implementaci s očekávanými časy $O(1)$ na jednotlivé operace získáváme algoritmus pracující v očekávaném čase $O(N)$ (viz kapitola „Reprezentace množiny“).

Příklad 2: Máme nalézt nejlacinější cestu z bodu s do bodu t v nezáporně ohodnoceném grafu s N vrcholy a M hranami:

Budeme postupovat hladovým algoritmem. Prvky ke zpracování budou ty, do nichž známe nějakou cestu z s. Hodnota prvku bude cena nejlacinější cesty tvořené dosud zpracovanými vrcholy a jedinou další hranou (i tuto hranu můžeme s prvkem evidovat). Algoritmus začne vložením (**Insert**) všech sousedů vrcholu s do struktury, pak v cyklu vždy nalezne (**FindMin**) a odstraní minimum m (**DeleteMin**) a přidá sousedy vrcholu m, které dosud nebyly určeny ke zpracování (**Insert**) a sníží hodnotu (**Decrement**) těch sousedů, kteří již byli určeni ke zpracování, ale cesta přes vrchol m je lacinější než dosud evidovaná cesta. Cyklus končí ve chvíli, kdy **FindMin** vrací hodnotu $m = t$. Pak již pouze uvolníme paměť (**Destroy**). Abychom mohli provádět **Decrement** a testovat, zda daný vrchol již je určen ke zpracování či již není určen ke zpracování, potřebujeme pro každý vrchol evidovat stav „nebyl a není“/„je“/„byl“ určen ke zpracování a klíč(pointer) do struktury pro ty které jsou určeny ke zpracování. Tuto evidenci zvládáme opravovat v čase $O(1)$ na operaci. V průběhu práce algoritmu (pro nejsložitější cesty) provedeme nejvýš $N - 1 \in O(N)$ operací **Insert**, nejvýš $N - 1 \in O(N)$ operací **FindMin** a **DeleteMin** a nejvýš $M - N \in O(M)$ operací **Decrement** a jednu operaci **Destroy**. Při optimální implementaci dostáváme časy $O(N \log N + M)$ (viz kapitola „H$\overleftrightarrow{\text{la}}$dové algoritmy“).

Příklad 3: Máme zjistit, zda v daném neorientovaném stromě z N vrcholů ohodnoceném celými čísly od 0 do K existuje dvojice vrcholů se vzdáleností právě d:

Strom budeme procházet prohledáváním do hloubky. Při návratu z podstromu pod vrcholem x vrátíme množinu M_x obsahující všechny vzdálenosti (stačí hodnoty nejvýš d) od vrcholu x, případně potvrzení, že v podstromu existuje hledaná dvojice vrcholů. Pokud dosud není známa odpověď, pak synové x_i vrcholu x postupně vracejí množiny M_{x_i} a na základě nich vytváříme M_x. Množinu M_x inicializujeme na $\{0\}$ (vzdálenost x od sebe sama), pak přidáme celou množinu M_{x_1} s hodnotami zvětšenými o vzdálenost $d(x, x_1)$ mezi x a x_1. Obdobně budeme přidávat množiny M_{x_i} zvětšené o vzdálenost $d(x, x_i)$ mezi x a x_i. Vždy před přidáním množiny M_{x_i} musíme zkontrolovat, zda neexistuje $m_{x_i} \in M_{x_i}$ a $m_x \in M_x$ tak, že $m_x + m_{x_i} = d - d(x, x_i)$ (v takovém případě by odpověď byla „existuje“).

Jaké operace provádíme s množinami M_x? Inicializujeme množinu na $\{0\}$, sjednocujeme dvojici množin, kde hodnoty v jedné z nich zvětšujeme o stejnou hodnotu h, a testujeme zda je možno nalézt po jednom zástupci z dvou daných množin tak, aby součet hodnot těchto zástupců byl dané číslo c.

Nenapadá mne jiná možnost jak provést uvedený test jinak než probrat postupně hodnoty m prvků jedné z množin a otestovat zda v druhé množině existuje prvek s hodnotou $c - m$. Budu-li probírat prvky

menší množiny, provedu tolik testů, jak je velká menší množina. V takové implementaci potřebuji znát velikosti množin (**Size**) M_x (potřebuji vědět, která je menší, přinejmenším tehdy, kdy je jedna výrazně větší), umět rychle otestovat, zda daná hodnota je v množině M_x (**Read**) a dále musím umět rychle vracet prvky množiny M_x (**List**) (stačí pro menší z dvojice množin). Sjednocení dvojice množin můžeme provést vložením (**Write**) každého prvku menší množiny do množiny větší. V takovém případě budeme muset umět do M_x přidat libovolný prvek. V případě, že menší z množin byla ta, jejíž hodnoty mají být o h zvětšeny, zvětšíme vkládané hodnoty. V opačném případě, pokud budeme vkládat hodnoty o h zmenšené, dostaneme množinu, jejíž všechny hodnoty jsou o h menší než by měly být. Tento posun celé množiny můžeme brát v úvahu, kdykoli přistupujeme k prvku množiny, zpomalí se tím práce s množinou pouze o konstantu.

Pro celkový čas algoritmu je důležité vědět, součet přes jednotlivé vrcholy, kolikrát byl vrchol v menší z množin M_x. Pokaždé, když byl vrchol v menší z množin M_x, pak množina vzniklá sjednocením alespoň zdvojnásobila velikost. Proto každý vrchol může být v menší množině nejvýš $\log N$ krát. Součet přes všechny vrcholy pak dává nejvýš $N \log N$. Počet vložení prvku (**Write**), testu existence hodnoty (**Read**) a součet velikostí menších množin je tedy $N \log N$. Očekávaný čas při optimálně zvolené datové struktuře je $O(N \log N)$. Dobrá implementace ale skrývá drobnou komplikaci, abychom nedosáhli veliké multiplikativní konstanty nevhodnou inicializací struktury pro M_x, je vhodné při inicializaci znát její očekávanou cílovou velikost. Spočítat tyto velikosti je možno přípravným průchodem stromu do hloubky. Při tomto průchodu je užitečné přečíslovat syny každého vrcholu x tak, aby byl x_1 vrchol s největším podstromem (zvládneme v čase $O(N)$).

S hodnotami v množinách M_x můžeme uchovávat i vrcholy, v nichž je hodnota nabývána, takže výsledkem v případě existující dvojice může být nalezená dvojice (viz kapitoly „Reprezentace množiny“, „Prohledávání“).

Příklad 4: Máme zjistit, zda v daném neorientovaném stromě z N vrcholů ohodnoceném celými čísly od 0 do K existuje dvojice vrcholů se vzdáleností mezi d_1 a d_2 (v $\langle d_1, d_2 \rangle$):

Označme $\ell = d_2 - d_1 + 1$ délku intervalu vyhovujících hodnot. Obdobně jako v předchozím příkladu budeme strom procházet prohledáváním do hloubky. Při návratu z podstromu pod vrcholem x vrátíme množinu M_x obsahující všechny vzdálenosti (stačí hodnoty nejvýš d_2) od vrcholu x, případně potvrzení, že v podstromu existuje hledaná dvojice vrcholů. Pokud dosud není známa odpověď, pak synové x_i vrcholu x postupně vracejí množiny M_{x_i} a na základě nich vytváříme M_x. Množinu M_x inicializujeme na $\{0\}$ (vzdálenost x od sebe sama), pak přidáme celou množinu M_{x_1} s hodnotami zvětšenými o vzdálenost $d(x, x_1)$ mezi x a x_1. Obdobně budeme přidávat množiny M_{x_i} zvětšené o vzdálenost $d(x, x_i)$ mezi x a x_i. Vždy před přidáním množiny M_{x_i} musíme zkontrolovat, zda neexistuje $m_{x_i} \in M_{x_i}$ a $m_x \in M_x$ tak, že $m_x + m_{x_i} \in \langle d_1 - d(x, x_i), d_1 + \ell - d(x, x_i) \rangle$ (v takovém případě by odpověď byla „existuje“).

Jaké operace provádíme s množinami M_x? Inicializujeme množinu na $\{0\}$, sjednocujeme dvojici množin, kde hodnoty v jedné z nich zvětšujeme o stejnou hodnotu h a testujeme zda je možno nalézt po jednom zástupci z dvou daných množin tak aby součet hodnot těchto zástupců byl v daném intervalu $\langle c_1, c_1 + \ell \rangle$.

Uvedený test je dotaz zda sjednocení intervalů $\langle c_1 - m, c_1 + \ell - m \rangle$ pro hodnoty prvků z jedné množiny protíná hodnoty prvků druhé množiny. Pokud se m od m' liší o nejvýš ℓ, pak sjednocením jejich intervalů je jediný interval. Místo prvků můžeme reprezentovat intervaly, čímž v nejhorším případě (disjunktní intervaly) velikost reprezentace zdvojnásobíme. Na první pohled se zdá že nejvhodnější je použití datové struktury umožňující dotazy na průnik libovolných intervalů, ale to bychom dosáhli času $O(N \log^2 N / \log \log N))$ (odhad získán řešením netriviální rekurence). Ve skutečnosti nám ale stačí průniky intervalů délky ℓ. Vše můžeme řešit obdobně jako předchozí příklad s tím, že do struktury ukládáme (pro nějaké pevné o) vrchol s klíčem $\lfloor (d - o)/\ell \rfloor$. Z důvodu nutnosti řešení „zaokrouhlovacích chyb“ musíme evidovat pro daný interval hodnot délky ℓ jak nejmenší, tak největší hodnotu (sjednocení intervalů).

Budu-li probírat intervaly menší množiny, provedu tolik testů, jak je velká menší množina. Pro každou hodnotu m menší množiny testuji, zda existuje m' tak aby $c_1 \leq m + m' < c_1 + \ell$ neboli $(c_1 - m - o)/\ell \leq (m' - o)/\ell < 1 + (c_1 - m - o)/\ell$, tedy připadají do úvahy pouze největší hodnota pod klíčem $\lfloor (c_1 - m - o)/\ell \rfloor$ a nejmenší hodnota pod klíčem $1 + \lfloor (c_1 - m - o)/\ell \rfloor$ (pokud tyto hodnoty neleží v požadovaném intervalu, nebudou tam ležet ani hodnoty ostatní). Zbytek analýzy již kopíruje analýzu příkladu 3. V takové implementaci potřebuji znát velikosti množin (**Size**) M_x (potřebuji vědět, která je menší přinejmenším tehdy, kdy je jedna výrazně větší), umět rychle nalézt nejmenší i největší hodnotu s daným klíčem v množině M_x (**Read**) a dále musím umět rychle vracet prvky množiny M_x (**List**) (stačí

pro menší z dvojice množin). Sjednocení dvojice množin můžeme provést vložením (**Write**) každého prvku menší množiny do množiny větší. V takovém případě budeme muset umět do M_x přidat libovolný prvek. V případě, že menší z množin byla ta, jejíž hodnoty mají být o h zvětšeny, zvětšíme vkládané hodnoty. V opačném případě, pokud budeme vkládat hodnoty o h zmenšené, dostaneme množinu, jejíž všechny hodnoty jsou o h menší než by měly být. Tento posun celé množiny můžeme brát v úvahu, kdykoli přistupujeme k prvku množiny, zpomalí se tím práce s množinou pouze o konstantu.

Pro celkový čas algoritmu je důležité vědět, součet přes jednotlivé vrcholy, kolikrát byl vrchol v menší z množin M_x. Pokaždé, když byl vrchol v menší z množin M_x, pak množina vzniklá sjednocením alespoň zdvojnásobila velikost. Proto každý vrchol může být v menší množině nejvýš $\log N$ krát. Součet přes všechny vrcholy pak dává nejvýš $N \log N$. Počet vložení prvku, testu průniku (náležení největšího či nejmenšího prvku s daným klíčem do intervalu) a součet velikostí menších množin je tedy $N \log N$. Očekávaný čas při optimálně zvolené datové struktuře je $O(N \log N)$, dobrá implementace ale skrývá drobnou komplikaci, abychom nedosáhli veliké multiplikativní konstanty nevhodnou inicializací struktury pro M_x, je vhodné při inicializaci znát její očekávanou cílovou velikost. Spočítat tyto velikosti je možno přípravným průchodem stromu do hloubky. Při tomto průchodu je užitečné přečíslovat syny každého vrcholu x tak, aby byl x_1 vrchol s největším podstromem (zvládneme v čase $O(N)$).

S hodnotami v množinách M_x můžeme uchovávat i vrcholy, v nichž je hodnota nabývána, takže výsledkem v případě existující dvojice může být nalezená dvojice. Pro pohodlnější výpočet klíčů můžeme místo $\lfloor (d-o)/\ell \rfloor$ používat například $\lfloor (d-o)/2^k \rfloor$ takové, že $2^k \leq \ell < 2^{k+1}$. V takovém případě ale budeme muset občas testovat 3 klíče (viz kapitoly „Reprezentace množiny“, „Prohledávání“, „Rekurence“).

Příklad 5: Máme dán neorientovaný strom z N vrcholů ohodnocený celými čísly od 0 do K. Budou nám pokládány dotazy, zda existuje dvojice vrcholů se vzdáleností mezi d_1^i a d_2^i (v $\langle d_1^i, d_2^i \rangle$):

Příklad je velmi podobný předchozí dvojici příkladů. Liší se ale tím, že cílem je vytvořit datovou strukturu (pokud možno co nejrychleji), která pak bude efektivně odpovídat na specifikovaný typ dotazů. Dotazy jsou intervalové (ačkoli je pro nás podstatná pouze neprázdnost výsledku), proto je přirozené vytvořit uspořádanou množinu všech vzdáleností ve stromě. K jejímu vytvoření je opět přirozené použít prohledávání stromu do hloubky a vracet množinu všech vzdáleností vrcholů od kořene podstromu. Konečné vzdálenosti vzešlé složením dvou větví podstromu rovnou ukládáme do množiny vzdáleností (očekáváme-li, že se některé vzdálenosti budou často opakovat (hrubé zaokrouhlování), pak je užitečné nejprve nejprve vytvořit množinu všech vzdáleností a teprve potom vytvářet uspořádanou množinu). Vzhledem k tomu, že evidujeme až $\Theta(N^2)$ vzdáleností, nemůžeme očekávat, že bychom množinu všech vzdáleností vytvořili rychleji. Vytvoření uspořádané množiny pak bude v čase $O(M \log M)$, kde M je velikost výsledné množiny. Dotazy pak budou odpovídány v čase $O(\log M)$. Obáváme-li se toho, že by počet položených dotazů neospravedlňoval čas předvýpočtu, můžeme vybudování struktury rozložit do dávek trvajících $\Theta(N)$, při nichž odpovídáme na položené dotazy použitím algoritmu z předchozího příkladu. Celkový čas tím zvětšíme pouze konstanta krát, ale při krátké posloupnosti dotazů tolik času neztratíme (viz kapitoly „Reprezentace množiny“, „Reprezentace uspořádání“, „Prohledávání“).

Příklad 6: V leasingové společnosti evidujeme databázi leasingových smluv. Informační systém má mimo jiné párovat příchozí platby s pravidelnými pohledávkami za klienty a pravidelně jednou za d_1 dní reportovat zpoždění plateb aspoň o d_2 dní:

Předpokládáme, že běžná smlouva je placena v měsíčních splátkách po dobu 4 let. Očekáváme tedy cca 50 krát více plateb než je smluv. Založení smlouvy (vyplnění požadovaných údajů) je časově náročné, a tento proces optimalizovat nebudeme (pro nás je smlouva zakládána jednou operací **Write**, případně může být načítán soubor smluv, což pro nás znamená sekvenci operací **Write** a při optimální implementaci načtení a uložení S smluv trvá $O(S)$).

Algoritmicky zajímavější jsou procesy párování plateb a reportování dlužníků, protože tyto procesy téměř nevyžadují interakci. Pokud je klient smluvně informován o neumísťování platby s nesprávně vyplněným variabilním symbolem, a leasingová společnost používá daný bankovní účet jen z důvodu inkasa splátek, je možné elektronicky získané bankovní výpisy zpracovávat tak, že variabilní symbol došlé platby bude klíčem do databáze leasingových smluv. Každá položka bankovního výpisu tedy vyžaduje jednu operaci **Read** (získáme informace o dosavadní historii placení dané smlouvy), informaci aktualizujeme na základě platby a jednou operaci **Write** novou informaci uložíme. V případě, že operace **Read** nevrací aktivní leasingovou smlouvu, je potřeba platbu reportovat jako neumístěnou (úroky či chyba klienta). Při optimální implementaci bude zpracování bankovního výpisu s B položkami trvat $O(B)$.

Pro reportování prodlení plateb je užitečné mít datum první nezaplacené splátky jako datovou položku p leasingové smlouvy. *Intervalový dotaz* který pravidelně pokládáme je „Vrať smlouvy kde $p < D$“. Položku p bychom mohli mít nastavenu na $+\infty$ pro smlouvy již splacené. Nebo pokud bychom nechtěli reportovat smlouvy, které jsme již reportovali, byl by „stav“ smlouvy datovou položkou s databáze, která by musela být zohledňována v pravidelných **Intervalových dotazech** „Vrať smlouvy s $p < D$ a $s = x$“. Při použití „reprezentace uspořádáné množiny s rychlým přístupem“ pro smlouvy ve stavu x (pro takové stavy x pro něž provádíme reporty) dostaneme časovou složitost reportu $O(L + \log N_x)$, kde N_x je počet smluv ve stavu x a L je počet reportovaných smluv. Každá aktualizace, kde se změnila jedna z položek p, s, si vyžádá čas $O(1 + \log N_s + \log N_{s'})$ na aktualizaci reprezentace, kde s je stav před a s' po aktualizaci a N_x je 1 pokud pro stav x dotazy nepokládáme a proto reprezentaci uspořádání neudržujeme. Aktualizace „reprezentace uspořádané množiny s rychlým přístupem“ nemusí být prováděna současně s aktualizací položky, stačí, když je provedena nejpozději na začátku spuštění reportu. Pokud bychom ale aktualizaci neprováděli ihned, musíme alespoň evidovat seznam záznamů, pro něž byla aktualizace odložena. Na počítači může běžet proces s nízkou prioritou, který kontroluje existenci odložených aktualizací, případně aktualizaci provede a odstraní ze seznamu. Tento proces nemá vliv na pořizování nových smluv ani párování plateb. V době reportování prodlení bude typicky odložena aktualizace nejvýše několika položek z posledního párování plateb. Pokud proces s nízkou prioritou stíhá v průběhu každého dne vyprázdnit seznam odložených aktualizací a víme, že všechny *Intervalové dotazy* „Vrať smlouvy s $p < D$ a $s = x$“ které v daném dni případají v úvahu použijí stejné D, x, pak můžeme seznam odložených transakcí rozdělit dle priorit dle toho, zda se mění platnost podmínky $p < D$ a $s = x$ nebo ne. Report prodlení plateb pak nemusí čekat na zpracování odložených transakcí neměnících platnost podmínky (pokud nám nevadí, že smlouvy nebudou uvedeny v pořadí dle doby prodlení). Typicky nereportujeme jednodenní prodlení, a v takovém případě nemusíme na zpracování odložených aktualizací čekat vůbec. Pokud by smlouva byla zahrnuta ve výběru pro report, ačkoli byla v párování plateb posledního dne uhrazena, byla by korektně reportována jako smlouva u níž bylo prodlení. Report by některé platby umístěné posledního dne nezohledňoval.

Vzhledem k tomu, že typicky reportujeme větší množství prodlení pro stejný den D, je při reprezentaci uspořádání vhodné využít toho, že klíče se opakují. V přesnějším časovém odhadu pak jsou N_x počty různých dní D smluv ve stavu x (viz kapitoly „Reprezentace množiny“, „Reprezentace uspořádání“).

3. Reprezentace množiny

Reprezentaci obecné množiny používáme tehdy, když potřebujeme ukládat informace identifikované „klíči“, aniž bychom využívali vzájemné vztahy mezi klíči. Používáme operace **Write**, **Read**, případně i **Delete**. Především jednorázové algoritmy vyžadují operaci **List** (**ListFirst**+**ListNext**) vracející postupně v libovolném pořadí všechny prvky množiny.

Optimální reprezentace využívající přímou adresaci pracuje v očekávaném čase $O(1)$ na operaci. Základní trik je ve výpočtu v čase $O(1)$ adresy záznamu pomocí tzv. *hashovací funkce*. Je to libovolná funkce, která adresuje náhodné klíče rovnoměrně v rozsahu svého oboru hodnot (typicky interval celých čísel).

Očekávané časy operací souvisejí s pravděpodobností kolizí spočtených adres a ta úzce souvisí s poměrem počtu evidovaných záznamů a rozsahu hodnot hashovací funkce (*faktor naplnění tabulky*). Předpokládáme, že klíče jsou generovány nezávisle na hashovací funkci (tu můžeme volit náhodně z větší množiny funkcí obáváme-li se úmyslné volby klíčů s větším množstvím kolizí).

Kolize je možno řešit mnoha způsoby ... od vytváření uspořádaných či neuspořádaných seznamů či použitím více či méně sofistikovaného algoritmu výpočtu náhradní adresy. Kvalita způsobu řešení kolizí se projeví pouze tehdy, nastávají-li kolize často.

Z obyčejné tabulky vzniká datová struktura ve chvíli, kdy začneme předcházet kolizím tím, že hlídáme faktor naplnění tabulky a při překročení stanovené meze tabulku zvětšíme (s tím také souvisí změna hashovací funkce) a položky „přehashujeme“. Impulsem k přehashování může být i velký počet smazaných záznamů v případě mazání „kaňkováním“ (nutnost při řešení konfliktů výpočtem náhradní adresy), či velký počet konfliktů oproti počtu očekávanému vzhledem k faktoru naplnění. Přehashování nemusíme dělat v rámci jediné operace, ale můžeme jej rozdělit na $\Theta(n)$ operací (kde n je velikost reprezentované množiny při startu přehashovávání). V takovém případě by měly být důvody pro přehashování definovány tak, aby nastávaly nejvýš jednou za $\Omega(n)$ operací tak, abychom měli vždy pouze dvě hashovací tabulky (zbytek původní a vytvářenou aktuální). Vyhledávání záznamů provádíme dotazem do obou tabulek.

Jaká je pravděpodobnost k násobné kolize při faktoru naplnění α (pro k klíčů vrací hashovací funkce stejnou adresu)? Jaká je průměrná doba (ne)úspěšného volání funkce **Read** při řešení konfliktů neuspořádaným seznamem? Jaký vliv mají konflikty vypočtených náhradních adres pokud jsou velké interference náhradních adres pro klíče „primárně“ hashované na různé adresy? Jaký vliv mají konflikty vypočtených náhradních adres pokud jsou malé interference náhradních adres pro klíče „primárně“ hashované na různé adresy? Jaký vliv mají konflikty vypočtených náhradních adres pokud jsou náhradní adresy hashovány z klíčů náhradními hashovacími funkcemi místo výpočtu na základě primární hashovací adresy? Odpovědi na tyto otázky naznačují, pro jaké faktory naplnění je očekávaná přístupová doba konstantní.

Podstatné je to, že cena paměti není tak velká, abychom si nemohli dovolit využít paměti konstanta krát větší než je nutno.

Mnohdy známe maximální počet záznamů, který bude v tabulce uložen. V takovém případě alokujeme tabulku na tomu odpovídající velikost (v mezích zvoleného faktoru naplnění). Uvědomme si, že inicializace prostoru n stojí čas $\Theta(n)$, takže by bylo chybou alokovat tabulky mnohem větší než jaká bude jejich maximální velikost (než je počet operací **Read** a **Write**, které budou se strukturou provedeny).

Hashování je tak jednoduchá datavá struktura, že mnohdy bývá zastíněna složitějšími datovými strukturami, které jí ale v reprezentaci obecné množiny nemohou konkurovat.

Hashování bývá použito jak pro výpočet v krátkodobé paměti, tak jako metoda ukládání dat v databázích. V druhém případě adresa určuje fyzický blok disku, typicky velikosti 2^k, kde $k > 10$. V takovém případě za jednotkovou operaci bereme načítání bloku disku. To že několik malých záznamů je hashováno do stejného bloku disku je vpořádku. Kolizi v takovém případě odpovídá to, že se záznamy hashované na adresu bloku do bloku nevejdou. Kolize mohou být řešeny například „souborem přetečení“. Důvodem změny velikosti adresového prostoru může být faktor naplnění definovaný jako celkový počet bytů uložených záznamů vůči celkovému počtu bytů adresovaného prostoru.

Při řešení kolizí „seznamem“ mimo hashovací tabulku je očekávaná délka seznamu pro náhodný klíč rovna faktoru naplnění α (součet délek přes všechny seznamy dělený počtem adresovatelných seznamů). Jsou-li klíče z lineárně uspořádaného univerza, může být užitečné udržovat seznamy uspořádané. Multiplikativní konstanta operace **Write** se tím příliš nezvětší, pro dlouhé konfliktní seznamy při porovnávání $\leq$ místo $=$ se tím urychlí neúspěšné vyhledávání. Při malém faktoru naplnění ale dlouhé konfliktní seznamy

nevznikají a vliv tohoto triku je minimální. Ze stejného důvodu se většinou nevyplatí použití vyhledávací struktury pro daný uspořádaný seznam (mnohem efektivnější je udržovat malé α a v případě neočekávaně velkého počtu kolizí změnit hashovací funkci).

Nejefektivnější způsob řešení kolizí (v rámci adresového prostoru hashovací tabulky) je následující: Máme systém hashovacích funkcí h_i^n určených dvěmi parametry - n, i (n udává velikost tabulky, i nemá další význam, funkce lišící se jen v i musejí být nezávislé). Při volbě velikosti n hashovací tabulky určíme náhodně i. Každý klíč hashujeme nejprve funkcí h_i^n. Nastane-li k-tá kolize, použijeme funkci h_{i+k}^n. V takovém případě nastává každá kolize s pravděpodobností α (faktor naplnění). Neúspěšné vyhledávání pracuje v očekávaném čase $1+(1-\alpha)\sum_{k=0}^{\infty} k\alpha^k = 1+\alpha/(1-\alpha)$ (postup zjednodušení: $S = \sum_{k=0}^{\infty} k\alpha^k = \sum_{k=1}^{\infty} \alpha^k + \sum_{k=0} k\alpha^{k+1} = \alpha/(1-\alpha)+\alpha S$, odtud $S = \alpha/(1-\alpha)^2$). Při faktoru naplnění $(k-1)/k$ dostáváme očekávaný čas k.

Vzhledem k tomu, že výpočet hashovací funkce většinou využívá dělení, je snaha výpočet adres zjednodušit. Někdy používáme 2 hashovací funkce. Po k-té kolizi použijeme adresu $h_1(x) + k^2 \cdot h_2(x)$ či hůře $h_1(x) + k \cdot h_2(x)$ (vždy modulo velikost tabulky). Další metoda používá $h(x) + kc_1 + k^2c_2$. Nejhorší výsledky dává $h(x) + kc$ kvůli interferenci konfliktů. Tyto interference analýzu očekávaného času těchto metod podstatně komplikují. Čím méně „trajektorií“ použijeme při řešení kolizí, tím více budou vznikat shluky kolizí. Pro poslední metodu vznikají významné interference při faktoru naplnění kolem 1/2. Pro ostatní metody při faktoru naplnění tabulky kolem 2/3 jsou interference malé a srovnatelné. Při větším povoleném faktoru naplnění si u těchto metod musíme dávat pozor, aby náhradní pozice procházely téměř celou tabulku. Čím méněkrát se opakují náhradní adresy před nalezením volné adresy, tím lépe.

Přirozený požadavek na operaci **List** je, aby její celkový čas byl úměrný velikosti reprezentované množiny. Pokud udržujeme faktor naplnění tabulky v rozsahu $\langle a_1, a_2\rangle$, kde $0 < a_1 < a_2 < 1$ jsou konstanty, pak přirozená implementace „hrubou silou“ má požadované vlastnosti. Alternativou je udržovat seznam prvků množiny. Je-li vyžadována efektivní implementace operace **List** a nemáme garantován dolní odhad faktoru naplněnní, je to dobrá volba. Závisí na aplikaci, zda očekáváme přístupy převážně pomocí operace **Read** či **List** i na tom, zda provádíme **List** dostatečně často, aby se vyplatilo zpomalit operaci **Write** a zvětšit datovou strukturu zavedením odkazů. Otázkou k zamyšlení je zabudování odkazů do hashovací tabulky a využití „nevyužitých“ odkazů pro strategii řešení kolizí (při použití obousměrného seznamu by implementace s oddělenými seznamy konfliktů nemusela činit potíže).

Pokud neočekáváme použití operace **Read**, ale veškeré přístupy k prvkům pouze pomocí operace **List** nebo přímé adresace, nepotřebujeme hashování a stačí nám seznam reprezentovaných prvků. Pokud nebudeme používat ani operaci **Delete**, stačí nám jednosměrný seznam. Pokud operaci **Delete** vyžadujeme (operace **Write**, **List** vracejí odkaz pro budoucí přímou adresaci), použijeme seznam obousměrný.

Jednosměrný seznam je triviální datová struktura, kde každý prvek má kromě reprezentované hodnoty navíc odkaz na jiný reprezentovaný prvek. Seznam je určen odkazem na první prvek. Odkazy jsou organizovány tak, abychom postupným používáním odkazů počínaje prvním prvkem navštívili všechny prvky množiny. Běžně je odkaz posledního prvku seznamu prázdný, cyklická varianta seznamu odkazuje z posledního prvku na prvek první.

Obousměrný seznam je obdobná datová struktura, kde každý prvek má navíc druhý odkaz na jiný reprezentovaný prvek. Druhý (neboli zpětný) odkaz prvku x míří na prvek, z něhož vede první (neboli dopředný) odkaz na prvek x. Výjimku tvoří první prvek seznamu, z něhož může vést zpětný odkaz na poslední prvek seznamu (cyklický zpětný seznam), nebo může být tento odkaz prázdný. Naprosto běžné je použití, kdy jeden ze seznamů je cyklický a druhý končí prázdným odkazem.

Implementace operací nad seznamy je triviální.

4. Reprezentace uspořádání

Reprezentaci uspořádání využíváme tam, kde je klíčové udržovat „uspořádaný spojový seznam“. Nejběžnější využití, je pokud očekáváme *Intervalové dotazy*, tedy, bude-li kladen požadavek vyjmenovat všechny prvky množiny s (sekundárním) klíčem v daném rozsahu hodnot (užitečné navíc je, že struktura bude vracet prvky v pořadí dle klíčů).

Jedinou věc, kterou zbývá dořešit, v případě *Intervalového dotazu*, je nalezení počátku a konce úseku seznamu, který máme vrátit.

Reprezentace uspořádání má velice různorodá využití a často při nich evidujeme dodatečné užitečné informace jako je například velikost reprezentované (pod)množiny. Operace běžně prováděné na reprezentaci uspořádání jsou **Insert**, **Delete**, **Join**, **Split**. (**Insert** vkládá nový prvek s daným klíčem v uspořádání do reprezentace, **Delete** odstraňuje daný prvek, který bývá určen buď jednoznačným klíčem, nebo, v případě potřeby opakujících se klíčů, odkazem, který v takovém případě navracel **Insert** v době vložení prvku. **Join** spojuje uspořádané množiny za předpokladu, že největší prvek jedné předchází nejmenší prvek druhé. Některé varianty operace **Join** zároveň vkládají na příslušné místo prvek s klíčem mezi maximem první a minimem druhé množiny. Operace **Split** rozdělí uspořádanou množinu v prvku, určeném obdobně jako při operaci **Delete**. Výsledkem bývá množina prvků předcházejících určený prvek, množina prvků následujících po určeném prvku a v závislosti na variantě struktury buď určený prvek, či prvky se stejnou hodnotou jako určený prvek, nebo jsou vráceny pouze dvě množiny a určený prvek je v jedné z nich zahrnut.)

Základní technikou umožňující nalezení nejbližšího místa k danému klíči v uspořádaném seznamu je vytvoření „vyhledávacího“ stromu nad daným uspořádaným (třeba jen implicitním) spojovým seznamem. Hlavní účel vyhledávacího stromu je nalezení jedné konkrétní z N hodnot s použitím $O(\log N)$ porovnání. Vlastní konstrukce vyhledávacích stromů bývá velmi různorodá. Příkladem jsou B-stromy s rozskoky dle B až $2B-1$ klíčů ve vnitřním vrcholu stromu (kořen jich může mít méně), kde B je optimalizováno pro blokové čtení disku (pro každý blok je možno vyhledávat půlením intervalu, ale běžně se vyhledává sekvenčně, vzhledem k zanedbatelnosti času vyhledání ve vnitřní paměti vůči času blokového čtení). Jiným příkladem jsou binární stromy, kde vyhledávací strom není nadstavba nad seznamem, ale vnitřní vrcholy vyhledávacího stromu tvoří spojový seznam reprezentovaného uspořádání.

Vyhledávání ve vyhledávacím stromu probíhá tak, že porovnáváme hodnotu vyhledávaného klíče s hodnot(ou/ami) v kořeni, a prohledávání pokračuje v jednoznačně určeném (kořeni) podstromu. Vyhledání končí v listu, s výjimkou varianty binárního podstromu, kde může končit předčasně nalezením klíče se stejnou hodnotou.

Dobře vyvážený vyhledávací strom garantuje, že porovnání bude $O(\log N)$. O vyváženost musíme, při práci s uspořádanou množinou, pečovat. U B-stromů je vyváženost garantována tím, že všechny větve stromu jsou udržovány stejně dlouhé. U binárních stromů tohoto stavu nemůžeme dosáhnout. Dříve byl používán invariant „AVL“ vyvažování, kdy se hloubky podstromů daného vrcholu mohou lišit nejvýš o 1. Rekurence $A_k = 1 + A_{k-1} + A_{k-2}$ snadno odhalí, že nejmenší takový strom hloubky n je velikosti řádově F_n (Fibonacciho čísla). Odkud hloubka stromu je $\log N / \log((1+\sqrt{5})/2)$. Nyní bývá častěji používán invariant „Červeno-černých stromů“, kde jsou vrcholy obarveny dvěmi barvami, červené vrcholy nemohou sousedit (pod sebou) a všechny cesty z kořene do nejbližšího „NIL“ obsahují stejný počet černých vrcholů (NIL zde označuje chybějícího následníka vrcholu v směru vyhledávání). Z invariantu snadno nahlédneme, že nejdelší větev stromu je nejvýš dvakrát delší než nejkratší větev, tedy je dlouhá nejvýš $2\log N$. Hloubka červeno-černých stromů je tedy větší než hloubka AVL stromů, výhodou červenočerných stromů je ale menší (konstantní) počet „rotací“ nutných k vyvážení stromu při operacích **Insert** a **Delete**. To, že je možno zvolený invariant udržovat je otázkou na cvičení resp. na nahlédnutí do implementačních detailů těchto struktur. Mám za to, že operace **Join** a **Split** jsou pro tyto metody vyvažování velice neefektivní, ale nerad bych se pletl. ?rozmyslet!?

Jinou metodou „vyvažování“ binárních stromů je technika samovyvažovacích se stromů (neboli splay stromů) [ST]. Základem techniky je operace **Splay**, která efektivním způsobem vyrotuje vybraný prvek do kořene binárního stromu. Operace **Insert**, **Delete**, **Split**, **Join** jsou pak snadno implementovány. U samovyvažujících se stromů se chvíli pozdržíme. Mnoho studentů si z „biologického popisu“ splay stromů odnese pouze to, že vybraný prvek je vyrotován do kořene. Způsob vyrotování je ale velice podstatný. Vyvažování stromu je zajištěno právě zvoleným způsobem rotování (provádíme „dvojrotace“ ... rotujeme otce vybraného vrcholu a pak vybraný vrchol, kdykoli rotace otce přiblíží vybraný vrchol ke kořeni. Jinak rotujeme dvakrát vybraný vrchol. Dvojrotace provádíme dokud není vybraný vrchol či jeho

aktuální otec kořenem stromu. V druhém případě poslední rotací vybraného vrcholu zakončíme operaci **Splay**). Proč je způsob rotování podstatný vyplyne až z rozboru časové složitosti operací **Splay**.

Pokud bychom postupně vkládali utříděnou posloupnost pomocí operací **Insert**, vznikl by nám strom hloubky N. Následný **Splay** minima by pak zákonitě trval $\Omega(N)$. Tento příklad ilustruje to, že **Splay** stromy bývají často nevyvážené a některé jednotlivé operace mohou trvat velice dlouho. Splay stromy jsou ale velice efektivní „amortizovaně“. Ono postupné vkládání utříděné posloupnosti operací **Insert** totiž netrvalo $\Theta(N \log N)$, jak by bylo u jiného způsobu vyvažování obvyklé, ale pouze $\Theta(N)$. Dá se ukázat (a vzápětí tak učiníme), že celkový čas $\Theta(N)$ operací **Splay**, začínající prázdným stromem netrvá nikdy více než $O(N \log N)$. Vzhledem k tomu, že operace **Insert**, **Delete**, **Split** i **Join** jsou implementovány pomocí nejvýše dvou operací **Splay** a v jinak konstantním čase, dostáváme tak jednoduchou a velice efektivní datovou strukturu pracující v malém amortizovaném čase.

Vlastní rozbor amortizované složitosti operace **Splay** začneme tím, že přiřadíme každému vrcholu v stromu nezápornou váhu w_v (pro základní rozbor nám stačí $w_\bullet = 1$, pro porovnávání s konkrétním statickým binárním stromem volíme váhu $w_x = 4^{d-d_x}$, kde d_x je hloubka vrcholu x v daném statickém stromu a d je hloubka statického stromu). Označme T_x podstrom s kořenem x a $\mu_x = \lfloor \log_2(\sum_{y \in T_x} w_y) \rfloor$. Definujeme potenciál $\Psi = \sum \mu_x$, kde sčítáme přes všechny vrcholy datové struktury (někdy říkáme, že každý vrchol x má na účtě μ_x jednotek, kterými může platit za čas některých dvourotací, kterých se účastní). Ukážeme, že počet dvourotací a $\Delta\Psi$ v průběhu operace **Splay** s vybraným vrcholem x dohromady nepřesáhne $3(\Delta\mu_x)$, kde Δ označuje rozdíl vzniklý odečtením původní hodnoty veličiny od nové hodnoty veličiny. Celkový čas operace **Splay** pak můžeme odhadnout pomocí $O(1 + \Delta\mu_x)$. Uvědomíme-li si, že v základním rozboru je $\mu_x \leq \log_2 N$, bude tím tvrzení předchozího odstavce dokázané.

Při volbě vah $w_x = 4^{d-d_x}$, tak jak bylo výše naznačeno, dostaneme pro kořen splay stromu $\mu_\rho < \log_2\bigl(4^d \sum_{i=0}^{\infty} 2^i/4^i\bigr) = \log_2(2 \cdot 4^d) = 2d + 1$, z celočíselnosti pak $\mu_\rho \leq 2d$, přitom $\mu_x \geq \lfloor \log_2 w_x \rfloor = 2(d - d_x)$. Proto $\Delta\mu_x \leq 2d_x$. Hodnota potenciálu Ψ se pohybuje v rozsahu $(0, 2dN)$. Cena každé operace je konstanta krát větší než čas $\Theta(1 + d_x)$ vyhledávání v daném statickém stromě. Celkový rozdíl časů od ceny je menší než $2Nd < 2N^2$, nezávisle na úvodním tvaru splay stromu. Proto pro libovolnou posloupnost dotazů délky $\Omega(N^2)$ dosahují **Splay** stromy celkový čas nejvýš konstanta krát horší než daný statický strom. Přitom například na konstantních posloupnostech dotazujících se na nejhlubší vrchol statického stromu v hloubce $h = \Omega(\log N)$ budou **Splay** stromy dosahovat konstantní časy, tedy časy $\Theta(h) = \Theta(\log N)$ krát lepší.

Odhad počtu dvourotací a $\Delta\Psi$ provedeme „teleskopicky“. Ukážeme, že pro každou dvourotaci zvlášť platí, že $1 + \Delta\Psi \leq 3\Delta\mu_x$. Potenciál Ψ ovlivňují pouze vrcholy, kterým se v průběhu dvourotace změnilo μ. Vrcholy jejichž podstromy se nezměnily mají μ nezměněné, vrcholy, jejichž podstrom byl dvourotací pouze přeorganizován, ale obsahuje tytéž vrcholy mají μ také nezměněné. Jediné vrcholy, které mohly změnit μ jsou vrchol x, jeho otec y a děd z. Označme μ' hodnoty po dvourotaci. Protože podstom z před dvourotací obsahuje stejné vrcholy jako podstrom x po dvourotaci, je $\mu'_x = \mu_z \geq \mu_y \geq \mu_x$. Navíc víme, že $\mu'_y \leq \mu'_x \geq \mu'_z$. Odtud dostáváme odhad $\Delta\Psi \leq 2\Delta\mu_x$. Je-li navíc $\Delta\mu_x > 0$, pak z celočíselnosti dostáváme $1 + \Delta\Psi \leq 3\Delta\mu_x$. Pokud je ale $\mu_x = \mu'_x$, ukážeme, že některá z nerovností $\mu'_y \leq \mu'_x \geq \mu'_z$ je ostrá. Opět vzhledem k celočíselnosti pak dostaneme $1 + \Delta\Psi \leq 0 = 3\Delta\mu_x$. Nyní konečně uvidíme, proč je důležitý zvolený systém rotací: Pokud by totiž nerovnosti nebyly ostré, dostali bychom samé rovnosti $(\mu =)\mu_x = \mu_y = \mu_z = \mu'_x = \mu'_y = \mu'_z$. Rotace byly ale zvoleny tak, že výsledný podstrom T'_z je disjunktní buď s původním podstromem T_x nebo s výsledným podstromem T'_y. Jsou-li váhy obou podstromů aspoň 2^μ, pak váha jejich sjednocení musí být aspoň $2 \cdot 2^\mu$, to je ale spor definice μ'_x s uvedenou rovností. Pokud bychom rotovali vždy vrchol x, takovéto disjunktní podstromy bychom nedostali. Pro pochopení rozdílu je užitečné porovnat správnou a nesprávnou volbu rotací na posloupnosti N insertů prvků dle velikosti následované postupným zavoláním **Splay** na prvky ve stejném pořadí.

Vraťme se nyní od Splay stromů zpět k obecným vyhledávacím stromům nad uspořádaným seznamem. Intervalové dotazy s dobře vyvažovanými vyhledávacímy stromy pak trvají $O(L + \log N)$, kde L je délka odpovědi. (Způsobů předávání výsledků je mnoho. Běžný je interface „FindFirst“, „FindNext“, předávající jeden ukazatal, kde stačí jedno vyhledání a následná kontrola, zda prvek stále ještě není mimo rozsah. Je-li porovnávání hodnot neefektivní, je možno použít interface „FindRange“, „FindNext“, kde „FindRange“ vrací dva ukazatele a „FindNext“ po dosažení druhého z nich již není volán.)

Ještě bych měl zmínit možnosti reprezentace uspořádané množiny, kde se klíče určující uspořádání mohou opakovat. Modifikace je jednoduchá. Stačí pod daným klíčem udržovat spojový seznam. Operace **Delete** pak nejprve promazává příslušný spojový seznam a teprve je-li prázdný, volá **Delete** „na hodnotu klíče“.

Dosud jsme se věnovali pouze jednorozměrným intervalovým dotazům. Vícerozměrné intervalové dotazy není snadné optimalizovat. Nejjednodušším řešením je provést intervalový dotaz v každém rozměru a mezi výsledky nejmenší odpovědi vybrat ty, které vyhovují ostatním kritériím. Pro tyto účely je užitečné, aby „FindRange“ vracel velikost odpovědi. Pokud vrcholy vyhledávacího stromu budou navíc obsahovat informaci o velikosti podstromu, pak zjištění výsledné velikosti zvládneme v čase úměrném času nalezení počátku a konce úseku spojového seznamu. Dá se ukázat, že udržování této informace zvyšuje časové odhady implementace operací **Insert**, **Delete**, **Join**, **Split** pouze konstanta krát ve všech zmíněných implementacích vyhledávacích stromů.

Konečně bych měl zmínit i reprezentaci implicitně uspořádané množiny (nemáme klíče, podle kterých je množina uspořádána, známe pouze pořadí prvků). V takovéto struktuře nedávají smysl intervalové dotazy, ale operace **Split** a **Join** jsou naprosto přirozené, ale i **Insert** a **Delete** by měly smysl, pokud by poloha byla určena ukazatelem. Mohlo by se zdát, že takováto struktura je k ničemu, ale hodí se například několika způsoby pro efektivní udržování koster grafů. (Jednoltivé stromy můžeme reprezentovat pomocí „Eulerovské procházky nad zdvojenými hranami stromu“. „Vyhledávací stromy“ (využíváme odkazy na otce) nad takovými Eulerovskými procházkami umožňují efektivně hledat jméno stromu. Stromová nadstavba umožňuje efektivní organizaci informací o nestromových hranách ... tato reprezentace není vhodná pro udržování informací o cestách v stromu. Jinou variantou jsou „stromy cest“, kde jednotlivé cesty jsou vnímány jako implicitně uspořádané množiny, což opět urychluje nalezení jména stromu, ale umožňuje zefektivnit evidenci informací pro mnohé další cestové dotazy.) Techniky vyvažování vyhladávacích stromů mohou být uplatněny i zde, ačkoli k vlastnímu vyhledávání nedochází. Zachování uspořádání je ale klíčové.

Odkazy:

[ST] Daniel Dominic Sleator & Robert Endre Tarjan, "Self adjusting Binary Search Trees", JACM Volume 32, No 3, July 1985, 652-686

5. Reprezentace množiny slov

Mnoho aplikací zpracovávajících textové soubory často řeší otázky, zda dané slovo textu patří do množiny slov se speciálním významem (kompilátory). Mnohdy je množina slov se speciálním významem „klíčová slova“ neměnná. Pro takovéto aplikace si „trie“ svojí efektivitou nijak nezadá s hashováním. Na rozdíl od hashování je ale trie možno zobecnit i k vyhledávání překlepů. Zpracování textových souborů se přiliš nevěnuji, proto tato kapitola bude prozatím velice krátká. Pouze poznamenám, že v implementaci TEXu je trie velice efektivně použito na pravidla dělení slov jazyků, která jsou předkompilována operací `\dump` při kompilaci formátu. Zájemce odkazuji na „TEX the program“.

Základní trie je strom s aritou odpovídající velikosti abecedy. Vyhledávání v trie postupuje tak, že vždy dle příslušného písmena vyhledávaného slova zvolíme pokračování v trie. Pokud v daném směru trie nepokračuje („NIL“), slovo v množině není. Na konci slova zkontrolujeme, zda aktuální vrchol trie má nastaven odkaz na reprezentanta množiny. Vyhledání slova v takovémto trie trvá $O(L)$, kde L je délka vyhledávaného slova. Uvědomme si, že hashovací funkce počítá adresu v čase $\Omega(L)$, porovnání dvou klíčů tvořených slovy délky L trvá až $\Theta(L)$. Neboli tento čas je v rozboru jiných datových struktur považován za jednotkový.

Nevýhodou základního trie je jeho veliká prostorová náročnost (až velikost abecedy krát počet písmen reprezentované množiny slov). Tuto nevýhodu je možno eliminovat tím, že kontrahujeme vrcholy trie mající jediného syna (odkaz na reprezentanta množiny je pro tento účel vnímán jako syn). V takovém případě bude vrchol trie uvidovat svoji „hloubku“ a při vyhledávání ve vrcholu hloubky h bude použito h-té písmeno slova k určení pokračování. Vzhledem k tomu, že některá písmena vyhledávaného slova vůbec nebyla použita, je potřeba v případě úspěšného nalezení reprezentanta množiny slov zkontrolovat, zda se jedná o totéž slovo. Časový odhad $\Theta(L)$ vyhledávání touto úpravou zůstal zachován, ale prostorová náročnost klesla na velikost abecedy krát počet reprezentovaných slov (plus počet písmen reprezentovaných slov).

Prostorovou náročnost je možno dále výrazně zredukovat komprimací „velice řídké matice“ odkazů evidovaných v jednotlivých vrcholech. Při komprimaci můžeme využít překrývání řádků tak, že hodnota NIL může být překryta hodnotou jiného řádku. Zajistíme aby vyhledávání, při kterém bychom pokračovali hodnotou nepatřící do původního řádku, bylo konečné a neúspěšné. Pokud by někdy klesla hloubka vrcholu trie, nutně jsme použili nesprávnou hodnotu a vyhledávání ukončíme neúspěchem. Pokud stále roste hloubka vrcholu trie, nejpozději po $\Theta(L)$ krocích dojdeme do reprezentanta množiny, nebo do hloubky aspoň L. Je-li hloubka L a vrchol odkazuje na reprezentanta množiny shodného s vyhledávaným slovem, nebo rovnou odkazujeme na reprezentanta shodného s vyhledávaným slovem, pak je vyhledávání úspěšné. Jinak slovo v množině není. Jak moc je možno matici zkomprimovat? Uvědomme si, že matice obsahuje tolik hodnot různých od NIL, kolik je reprezentovaných slov. Při komprimaci můžeme optimalizovat pořadí sloupců (přeuspořádat abecedu) a volit počátky jednotlivých řádků. ...

Strukturu trie není dobré přeceňovat. Multiplikativní konstanty spojené s nalezením dalšího vrcholu nejsou zenedbatelné ve srovnání s konstantami při lineárním porovnáním dvou řetězců.

Ke zpracování textů se hodí též tzv. „Suffix trees“. ...

6. H↔ladové algoritmy

Haldy jsou datové struktury využívané především hladovými algoritmy. Tyto algoritmy pracují s množinou očekávaných událostí, kde jednotlivé události jsou zpracovávány v pořadí dle své „priority“ (prvky s nižší hodnotou priority dříve). Základní operace charakterizující haldy jsou **FindMin**, **DeleteMin** a **Insert**. První nalezne, druhá odstraní událost s nejnižší hodnotou priority, třetí zařadí další událost s přidělenou prioritou. Některé algoritmy efektivně využijí operace **Decrement**, umožňující snížit prioritu určeného prvku, či operaci **Meld**, umožňující spojit dvě haldy dohromady.

Jsou-li priority malá (b bitová) celá čísla, může být tato vlastnost priorit zneužita k výpočtům přímé adresace. ?odkazy!? My se budeme věnovat haldám, založeným na porovnávání v obecném lineárním uspořádání. Vzhledem k tomu, že posloupnost N **Insert**ů, a N dvojic **FindMin**, **DeleteMin** dokáže setřídit libovolnou posloupnost N prvků pomocí porovnávání, posloupností je $N!$ a každé porovnání dává dva možné výsledky, musí být celkový průměrný počet porovnání aspoň $\log_2 N! \in \Theta(N \log N)$. Odtud obecná posloupnost N **Insert**ů, **FindMin**ů a **DeleteMin**ů vyžaduje $\Theta(N \log N)$ porovnání a tedy alespoň takovýto čas.

Přesto však implementace, která dosahuje časů $\Theta(\log N)$ pro libovolnou z těchto operací není optimální. Mnoho aplikací pracuje tak, že výpočet ukončí, s tím, že zbyde poměrně velká prioritní fronta, která již není potřeba. Odstraňovat nepotřebné prvky operací **DeleteMin** je neefektivní, stačí v takovém případě **Destroy**, který pouze v čase $O(N)$ uvolní použitou paměť. Pro takovéto aplikace je urychlení operace **Insert** podstatné. Ukážeme implementaci, kde amortizované časové odhady operací **Insert**, **FindMin**, **Decrement** a **Meld** jsou $O(1)$ a **DeleteMin** je $O(\log N)$.

Naše implementace se pokusí minimalizovat počet nutných porovnání, přičemž organizace bude trvat nejvýš konstantakrát déle, než kolik porovnání provedeme. Předem upozorňuji, že zde uvedená implementace se liší od implementace „Fibonacciho hald“, tak jak je publikovala dvojice Fredman, Tarjan [FT]. Zde uvedená implementace je v případě silně převažujících **Insert**ů nad **DeleteMin**y efektivnější. Navíc nemůže být horší! (Extrémním případem posloupnosti operací, kdy je zde uvedená implementace $\Omega(\log N)$ krát efektivnější je posloupnost dvojic insertů se stále menšími hodnotami priorit, prokládaná jednou dvojicí operací **FindMin**, **DeleteMin**.) Níže uvedený rozbor bude poměrně komplikovaný. Komplikace jsou způsobené tím, že v implementaci odkládáme nalezení minima až na volání funkce **FindMin**. Tím sice amortizovaně ušetříme, implementace se nezkomplikuje, ale rozbor je komplikovanější než u verzí, které minimum neustále přepočítávají. Uvádím zde detailní rozbor, protože o žádném popisu této verze Fibonacciho hald nevím.

Strukturu nejprve popíšeme matematicky, později naznačíme implementační detaily. Protože se pokoušíme minimalizovat počet porovnání, budeme si výsledek každého porovnání pamatovat a v matematickém modelu zachycovat orientovanou hranou od prvku s vyšší k prvku s nižší hodnotou (priority). Vzhledem k tomu, že vyhledáváme pouze minimum, je zcela zbytečné(přinejmenším předčasné) porovnávat prvek, vede-li z něj hrana. Ve chvíli, kdy dostaneme jediný orientovaný strom, je jeho kořen minimem. Nalezení minima po N **Insert**ech vyžaduje celkem $N - 1$ porovnání. Naší snahou ale bude zorganizovat tato porovnání tak, aby operace **DeleteMin** do budoucna vynutila co nejmenší počet dalších porovnání.

Operace **DeleteMin** vynutí tolik dalších porovnání, kolik hran vede do minimálního prvku (bez 1). Z tohoto důvodu bude naší snahou udržovat stromy porovnání co možná nejužší. Přesněji pro dané $q \in (1, 2\rangle$ budeme dodržovat invariant „úzkosti“, že, je-li k počet „řádných“ synů nějakého prvku, pak velikost podstromu tohoto prvku je alespoň q^k (syn je vnímán proti orientaci hran provnávání). Tento invariant nám zajistí, že odstraněním minima přibyde nejvýš $\log_q N$ řádných synů, mezi kterými bude nutno hledat minimum. (Čtenář určitě tuší, že budou existovat i „neřádní“ synové. Již brzy se k nim dostaneme. Užitečné je vědět, že není potřeba rozlišovat, kteří synové jsou řádní a kteří ne, důležité pouze je, kolik je těch řádných. Neřádní synové se od řádných liší tím, že mají předplaceno jedno zpracování při operaci **FindMin**.) Označme počet řádných synů prvku „řád“ prvku.

K zajištění invariantu úzkosti máme jediný prostředek . . . vhodně zorganizovat porovnávání „kořenů“ stromů porovnání. Pokud porovnáme kořeny řádu k, kořen výsledného stromu bude mít řád $k + 1$ a velikost podstromu bude aspoň $2q^k \geq q^{k+1}$, tedy invariant úzkosti bude zachován. Při porovnání prvků různých řádů může výsledek porovnání, který neovlivníme dopadnout tak, že kořenem se stane kořen stromu původně většího řádu a pokud bychom výsledek porovnání evidovali jako řádného syna, mohl by být invariant úzkosti porušen. Naší snahou proto bude, pokud možno, porovnávat kořeny stejných řádů. To můžeme zorganizovat následujícím způsobem: Berme kořeny jeden po druhém a zařazujme je „do množiny“ dle jejich řádu. Pokud v množině již prvek daného řádu existuje, pak jej odstraníme a dané

dva prvky porovnáme. Řád kořene se tak o jedna zvětší a pokus o zařazení do množiny můžeme zopakovat. Pokud „množinové“ operace zvládáme v konstantním čase, nezabere organizace porovnávání řádově více času než vlastní porovnávání. Skončíme s tím, že řády všech aktuálních kořenů jsou zařazeny do množiny, každý řád je tedy zastoupen nejvýš jednou. Řády jsou celočíselné, nezáporné, nejvýš rovny $\log_q N$, zbývá nám tedy nejvýš $O(\log N)$ stromů. Kořeny těchto stromů porovnáme v libovolném pořadí (nejlépe od nejmenšího po největší řád). Pokud by výsledek porovnání měl porušit invariant úzkosti, musíme syna evidovat jako neřádného. Neřádných synů tímto postupem tedy vznikne $O(\log N)$ (je zbytečné testovat, zda je invariant úzkosti zachován, nejjednodušší je všechny syny, připojené v této fázi, připojit jako neřádné). Čas nalezení nového minima po operaci **DeleteMin** je tedy $t_1 \in O(1)$ za každý strom struktury, toto hledání nového minima vytvoří $O(\log N)$ neřádných synů a vznikne jediný strom.

Definujeme-li potenciál Φ_0 rovný počtu stromů haldy (každý kořen má na účtě t_1 na zaplacení času pro své budoucí připojení), pak čas hledání minima vůči $\Phi_0 \cdot t_1$ bude konstantní, ale vznikne nám $O(\log N)$ neřádných synů. Vzhledem k tomu, že neřádní synové prvku, vynutí nárůst Φ_0 ve chvíli, kdy bude prvek odstraňován pomocí **DeleteMin**, vložíme pro účely amortizovaného rozboru na účet neřádného syna čas t_1 na budoucí porovnání ... definujeme potenciál Φ_1 jakožto součet počtu neřádných synů. Cenu pak evidujeme vůči $(\Phi_0 + \Phi_1) \cdot t_1$. Musíme ale něčím platit nárůst potenciálu Φ_1. My si i tento nárůst předplatíme. Zavedeme potenciál Φ_2, rovný počtu stromů nevzniklých operací **DeleteMin** (vzniklých operací **Insert** případně i **Decrement**), a potenciál Φ_3, který bude nulový, pokud po poslední operaci **DeleteMin** již následoval **FindMin** nebo roven $\log_q N_d$, kde N_d je velikost podstromů haldy vzniklých operací **DeleteMin**. Operace **DeleteMin** nejprve volá **FindMin**, čímž dojde k vynulování potenciálů Φ_2 i Φ_3. Potom již **DeleteMin** potenciál Φ_2 neovlivní a zvýšení potenciálu Φ_3 prodraží operaci **DeleteMin** o $O(\log N)$. Vynulováním potenciálů $\Phi_2 + \Phi_3$ při operaci **FindMin** zaplatíme nárůst potenciálu Φ_1. (Strom v množině řádů stromů bude zaplacen z potenciálu Φ_2, pokud nevznikl operací **DeleteMin**, případně vznikl při této operaci **FindMin** spojením s takovýmto stromem. Celková velikost ostatních stromů, je nejvýš N_d, proto jejich nejvyšší řád je nejvýš $\log_q N_d$ a na nárůst potenciálu Φ_1 způsobený těmito stromy potenciál Φ_3 stačí.)

Operaci **Meld** implementujeme jednoduše tak, že spojíme seznamy kořenů stromů hald. Rozbor složitosti je komplikován jen tím, že dosud oddělené rozbory pomocí potenciálů nemůžeme dále vnímat odděleně. U potenciálů Φ_0, Φ_1 a Φ_2 to nepřináší komplikace, a protože nerovnost $\log_q N_1 + \log_q N_2 = \log_q N_1 \cdot N_2 \geq \log_q(N_1 + N_2)$ platí pro $N_i \geq 2$, vzroste spojením hald potenciál Φ_3 nejvýš o konstantu.

Implementaci operace **Insert** můžeme vnímat jako **Meld** nově vytvořené jednoprvkové haldy s již existující haldou. Vytvořením jednoprvkové haldy vzrostou potenciály Φ_0 a Φ_2 o konstantu. Celková cena zůstává jednotková a rozbor ostatních operací zůstává zachován.

Zbývá implementovat operaci **Decrement**. Uvědomme si, že snížením hodnoty priority může být zneplatněna pouze hrana porovnání vedoucí z adresovaného prvku. Vzhledem k tomu, že chceme minimalizovat počet porovnávání, nebudeme zkoušet, zda ke zneplatnění skutečně došlo a hranu prohlásíme za neplatnou a odstraníme ji. Tím snížíme řád otce adresovaného prvku (pokud by z adresovaného prvku žádná hrana nevedla, nemusíme dělat nic). Vzniklý strom zařadíme pomocí operace **Meld** ke zbytku haldy. Pokud bychom tímto implementaci operace **Decrement** ukončili, porušili bychom tím invariant úzkosti (odebíráním vnuků vznikají široké stromy). Ukážeme že stačí jednoho z řádných synů prohlásit za neřádného (snížit řád prvku), kdykoli nějakému synovi byl podruhé snížen řád. Při prvním snížení řádu vrcholu v vrchol v začerníme, při druhém snížení řádu vrchol v odčerníme a jeho otci řád snížíme (je-li kladný). Při odstranění hrany vedoucí z vrcholu v vrchol také odčerníme. Každé snížení řádu bez odstranění syna zvyšuje potenciál Φ_1 o 1 a navíc nás stojí konstantní čas t_2. Odstranění syna zvyšuje potenciály Φ_0 a Φ_2 o 1.

Jeden **Decrement** může vyvolat sérii k snižování řádů, přičemž bude nejvýš jeden prvek očerněn a $k-1$ prvků odčerněno. Zavedeme-li potenciál Φ_4 rovný počtu očerněných vrcholů, nepokazíme tím rozbor ostatních operací, protože ty počet očerněných vrcholů nezvyšují. Naopak jsme tímto potenciálem schopni zaplatit nárůsty potenciálu Φ_1 i časy spojené se snížením řádu vrcholu i s přebarvením. Vůči potenciálu $(\Phi_0 + \Phi_1 + \Phi_2 + \Phi_3 + \Phi_4) \cdot t_1 + \Phi_4 \cdot t_2$ je cena operace **Decrement** konstantní.

Zbývá ověřit platnost invariantu úzkosti při zvolené strategii začerňování vrcholů. Minimální velikost stromu řádku k označme H_k. Očíslujme řádné syny vrcholu od 0 dle pořadí připojení. Pro H_k platí $H_k = H_0 + \sum_{i=1}^{k-1} H_{i-1}$, protože při připojení byl syn s číslem i řádu alespoň i, nyní je tedy řádu alespoň $i-1$. Dostáváme $H_{k+1} - H_k = (H_0 + \sum_{i=1}^{k} H_{i-1}) - (H_0 + \sum_{i=1}^{k-1} H_{i-1}) = H_{k-1}$ a známý charakteristický polynom $\lambda^2 = \lambda + 1$ Fibonacciho posloupnosti. Řešením je zlatý řez a lineární kombinace posloupností $((1 \pm \sqrt{5})/2)^k$. Stromy jsou tedy $(1 + \sqrt{5})/2$ úzké.

Nyní po ukončení matematického rozboru naznačíme implementační detaily. První drobnou komplikací by mohlo být to, že potřebujeme přistupovat k synům stromů porovnávání a arita těchto stromů není shora omezena. Přirozenou implementací je udržovat množinu synů vrcholu s tím, že otec pouze odkazuje na tuto množinu. Vzhledem k tomu, že při operaci **DeleteMin** z množiny synů vytváříme množinu stromů haldy, je užitečné použít tutéž reprezentaci. Množiny stromů spojujeme při operaci **Meld**, z množiny synů vrcholu odstraňujeme určený prvek při operaci **Decrement**. Při operaci **FindMin** potřebujeme postupně v libovolném pořadí přistoupit ke všechm prvkům množiny stromů. Přirozenou reprezentací splňující tyto požadavky je obousměrný spojový seznam (zpětný seznam cyklický). Kořeny stromů hald budou udržovány v takovémto seznamu, množina synů libovolného otce je též takovýto seznam, otec ukazuje na prvního ze svých synů. Je-li v množině stromů haldy jediný strom, je jeho kořen minimem.

Druhou drobnou komplikací je reprezentace množiny řádů v průběhu operace **FindMin** a s tím související reprezentace řádů prvků haldy. Jednou z možností je udržovat jeden globální obousměrný seznam možných řádů prvků a reprezentovat řád prvku haldy pomocí odkazu do tohoto seznamu (tento seznam se může automaticky prodlužovat). Při zvětšení řádu posuneme odkaz na následníka v seznamu a při snížení na předchůdce. V průběhu operace **FindMin** pak použijeme globální obousměrný seznam k reprezentaci příslušné množiny. První strom daného řádu nastaví zpětný ukazatel z globálního seznamu řádů. Druhý strom daného řádu tento odkaz použije a vynuluje .... Pokud můžeme používat nepřímou adresaci, je možno použít „globální pole“, ale automatické prodlužování stejně musíme implementovat, nemáme-li dobrý odhad logaritmu velikosti haldy. Další komplikaci přináší fakt, že potřebujeme efektivně vyjmenovat nepospojované stromy. To můžeme vyřešit tím, že strom z původního seznamu stromů odstraníme až tehdy, když jej připojujeme k jinému stromu v seznamu. Vzhledem k tomu, že obousměrné seznamy již při implementaci potřebujeme, přikláním se k reprezentaci s globálním seznamem. V takovém případě reprezentace vyžaduje pro obecný prvek haldy barvu, (odkaz na) hodnotu priority a reprezentovaný objekt, odkazy na otce a syna, odkazy na mladšího a staršího bratra a odkaz na řád.

Na závěr se zopakujeme implementaci jednotlivých operací, nyní již nekomplikovanou rozborem složitosti:

Operace **Insert** vytvoří záznam pro jeden prvek, prvek označí jako bílý (nebude černý), zaeviduje (odkaz na) hodnotu priority a reprezentovaný objekt, odkazy na otce, syna i staršího bratra nastaví na prázdné, odkaz na mladšího bratra ukáže na sebe (cyklický seznam) a odkaz na řád nastaví na začátek globálního seznamu. Pak zavolá **Meld** na původní haldu a tento prvek. Nakonec vrátí odkaz na vytvořený záznam pro případnou budoucí adresaci operací **Decrement**.

Operace **Meld** spojí obousměrné seznamy dvou hald (využijeme cykličnosti zpětného seznamu k nalezení posledního prvku), haldu reprezentuje odkaz na první prvek výsledného seznamu (musí fungovat i pro prázdný seznam).

Operace **DeleteMin** zavolá nejprve **FindMin**, haldu nyní reprezentuje odkaz na minimum, příslušný záznam odstraníme, současně ale zajistíme, aby haldu reprezentoval první syn odstraňovaného prvku.

Operace **FindMin** prochází seznam stromů, vždy si zapamatujeme, kde seznam pokračuje a s aktuálním kořenem provedeme následující akci: Pokud globální seznam dle řádu kořene někam odkazuje, pak odkazuje na kořen stejného řádu. Použijeme a odstraníme odkaz, porovnáme hodnoty priorit a kořen s větší hodnotou přesuneme z původního seznamu do seznamu synů druhého kořene, zároveň novému synovi zaevidujeme nový kořen jako otce. Novému kořeni zvětšíme řád (posun v seznamu řádů) a akci zopakujeme. Pokud globální seznam dle řádu kořene nikam neodkazuje, pak pouze zaevidujeme odkaz na tento kořen a akce končí. Po ukončení průchodu máme od každého řádu nejvýš jeden strom. Projdeme je ještě jednou, přičemž odstraníme zpětné odkazy z globálního seznamu, kořeny přitom po dvou porovnáváme a opět kořen s větší hodnotou přesouváme z původního seznamu do seznamu synů druhého kořene a nastavujeme otce novému synovi. Řády kořenů nyní neměníme. Na konci průchodu máme jednoprvkový seznam obsahující minimum. Vrátíme odkaz na jím reprezentovaný objekt.

Operace **Decrement** přesune adresovaný prvek ze seznamů bratrů do seznamu stromů haldy. Adresovaný prvek odčerní, zapamatuje si odkaz na otce a odkaz vyprázdní. Byl-li odkaz na otce prázdný, oparace končí. Jinak s otcem provádíme následující akci: Posuneme odkaz na řád dolů, nebyl-li první v globálním seznamu (předchůdce prvního prvku globálního seznamu odkazuje na sebe sama). Je-li otcův otec prázdný, operace končí. Zkontrolujeme černost otce. Není-li černý, očerníme jej a operace končí. Je-li otec černý, odčerníme jej a zopakujeme akci s jeho otcem.

A ještě poznámky:

Pokud bychom netrvali na optimalizaci počtu porovnání, mohli bychom provádět operaci **FindMin** po každé jiné operaci. Dostali bychom „Jednostromové Fibonacciho haldy“. V takovém případě může být tato operace optimalizována podle toho, po jaké operaci je provedena. (Po operacích **Insert**, **Meld** či **Decrement** spojujeme dva stromy, můžeme proto vynechat práci s globálním seznamem. Po operaci **DeleteMin** máme dost času na projití globálního seznamu až po nejvyšší řád v haldě. V obou případech nepotřebujeme udržovat původní seznam stromů.)

Pokud víme, že nebudeme potřebovat operaci **Decrement** nepotřebujeme barvu, odkaz na otce, stačí jednosměrný seznam bratrů, globální seznam by také mohl být jednosměrný. Kvůli operaci **Meld** potřebujeme odkaz na poslední prvek seznamu stromů nebo musí být seznam cyklický. Implementaci operace **FindMin** při použití jednosměrných seznamů také musíme pozměnit (při používání **FindMin** po každé operaci to nečiní problém).

Abuaiadh a Kingston se zabývali implementací s operací **Delete** místo operace **DeleteMin**, kdy provádíme častěji operaci **Delete** než operaci **FindMin**. Ukázali, že je-li operace **FindMin** použita T krát, **Decrement** D krát a ostatní operace dohromady $N > T$ krát, pak výsledný čas bude $O(N \log T + N + D)$. Navíc je v článku ukázáno, že pro libovolnou strategii porovnávání existuje posloupnost operací (a výsledků porovnání), která aspoň $\Omega(N \log T + N + D)$ porovnání vynucuje (přitom $T \to \infty$).

Pokud by aplikace operaci **Delete** vyžadovala, postupovali bychom jako při operaci **Decrement**, pouze místo zařazení adresovaného prvku do seznamu stromů bychom zařadili syny adresovaného prvku do seznamu stromů a adresovaný prvek bychom odstranili. Tím by potenciál Φ_0 vzrostl o řád adresovaného prvku a tolik, o kolik by poklesl potenciál Φ_1. Zároveň by po operaci musel být naplněn potenciál Φ_3 (podstatné při prvním **Delete**). Cena operace by tedy byla $O(\log N)$. Tak jak je v citovaném článku ukázáno, celkový součet řádů K vrcholů haldy je nejvýš $O(K(1 + \log(N/K)))$ a proto provedení K operací **Delete** nepřerušených operací **FindMin** stojí $O(K(1 + \log(N/K)))$. Po chvíli počítání z toho dostaneme, že N operací **Delete** proložených $T < N$ operacemi **FindMin** (a dalšími operacemi tak, že velikot haldy nepřesáhne N) stojí $O(N(1 + \log(T + 1)))$ (počítali jsme ceny pouze operací **Delete** a **FindMin**).

Odkazy:

[FT] Fredman M. L. & Robert Endre Tarjan "Fibonacci heaps and their uses in improved optimization algorithms", Journal of the ACM 34(3), 596-615

[AK] Diab Abuaiadh & Jeffrey H. Kingston "Are Fibonacci Heaps Optimal?" Technical Report Number 476, January 1994, The University of Sydney (http://www.cs.usyd.edu.au/research/tr/tr476.pdf)

7. Prohledávání

Prohledávání grafů či stromů je jednou z nečastějších aplikací nedatabázových algoritmů. Zdánlivě nejsnazším úkolem prohledávání grafů je nalezení nějakého vrcholu (či cesty do takového vrcholu) splňujícího předepsanou podmínku. Občas naším cílem není nalezení konkrétního vrcholu, ale pouze průchod všemi vrcholy grafu v algoritmem určeném pořadí. V takovém případě se při průchodu grafem snažíme spočítat něco, co se nám bude později hodit. Nejprve se budeme soustředit na první úlohu, později se budeme věnovat i možnostem dalších výpočtů.

Hlavním trikem při prohledávání grafů je zabránit tomu, abychom nějakou část grafu prohledávali opakovaně. Přitom nesmíme žádnou dosažitelnou část grafu vynechat. K tomu, abychom žádnou dosažitelnou část nevynechali stačí, když pro každý jednou dosažený vrchol prohledáme někdy všechny jeho sousedy. V principu jsou dvě možnosti jak to provést. Buď všechny sousedy ihned zařadíme mezi vrcholy, které bude potřeba v budoucnu prohledat nebo máme sousedy vrcholu nějak seřazeny a začneme prohledáváním prvního dosud neprohledaného souseda. Ve chvíli, kdy prohledání tohoto souseda skončí, začneme prohledávat souseda dalšího.

Při volbě první možnosti závisí na tom, jakým způsobem reprezentujeme množinu prvků k dalšímu zpracování. Výsledkem může být prohledávání do šířky, použijeme-li frontu (první zařazený bude první vyřazený), výsledkem může být prohledávání do hloubky, použijeme-li zásobník (poslední zařazený bude první vyřazený), výsledkem nebude ani prohledávání do šířky ani do hloubky, použjeme-li například haldy (máme-li určeny priority dalšího postupu).

Při volbě druhé možnosti dostaneme prohledávání do hloubky.

U malých grafů které mohou být vstupem algoritmu nezáleží příliš na tom, kterou z možností použijeme. U větších grafů, například na „implicitních“ grafech, kde je dán předpis, jak je z identifikátoru vrcholu možno generovat identifikátory jeho sousedů (příkladem je graf možných obarvení Rubikovy kostky, hrana odpovídá změně obarvení při otočení některé ze stěn), je rozdíl mezi první a druhou volbou velice výrazný. Při použití první volby algoritmu paměť počítače nebude stačit ani v případě, že hledaný vrchol je vzdálen od počátečního vrcholu jen „nepatrně“.

Při prohledávání malých grafů není těžké zabránit tomu, abychom nějakou část grafu prohledávali opakovaně — stačí abychom již navštívený vrchol označili. Pokud víme, že před spuštěním algoritmu není zádný vrchol označen, je situace mnohem jednodušší. Pokud o označení vrcholů předem nic nevíme, je nejjednodušší vrcholy číslovat a současně ukládat do pomocného pole zpětné odkazy. Zpětný odkaz nám umožňuje zkontrolovat, zda číslo u vrcholu je náhodné, či se jedná o číslo přidělené v průběhu číslování.

Při prohledávání implicitních grafů nemusíme mít dostatek paměti, kde bychom mohli vrcholy grafu označovat. V takovém případě použijeme co nejvíc dostupné paměti a místo označování do ní vrcholy hashujeme. Vzhledem k tomu, že nemáme prostor na řešení případných konfliktů, konflikty v tomto případě neřešíme a nově označený vrchol tak může nějaký jiný vrchol odznačit. Snažíme se použít hashovací funkci takovou, aby konflikty příliš neovlivňovaly chování algoritmu. Jsou-li sousedé konfliktních vrcholů hashováni odlišně, je odznačení konfliktného vrcholu nepodstatné, protože jeho již prohledaní sousedé zůstanou s velkou pravděpodobností označeni.

Jak tedy můžeme implementovat prohledávání implicitních grafů tak, abychom i s „malým“ množstvím paměti měli, v případě malé vzdálenosti od počátečního vrcholu, šanci na úspěch? Stanovíme si vzdálenost od počátečního vrcholu, do níž jsme ochotni graf prohledávat a prohledáme graf do hloubky. Na zásobník přitom ukládáme místo všech sousedů prohledávaného vrcholu pouze vrchol s pořadím dosud neprohledaného vrcholu. Evidujeme aktuální hloubku zásobníku (vrcholu) a pokud by překročila stanovenou mez, prohledávání vrcholu ukončíme. Navštívené vrcholy označujeme hashováním. V hashovací tabulce udržujeme dosaženou hloubku vrcholu. Pokud byl již vrchol označen v hloubce nepřesahující hloubku aktuální, prohledávání tohoto vrcholu ukončíme. U grafů, kde je velký stupeň vrcholů bývá čas prohledání grafu do hloubky h zanedbatelný vůči času prohledání grafu do hloubky $h + 1$. Proto není problém postupně prohledat hloubky 1, 2, ... dokud nenarazíme na hloubku, v níž se hledaný vrchol nachází. Ač takový algoritmus používá zanedbatelně paměti oproti prohledávání do šířky, je čas takového algoritmu s časem prohledávání do šířky srovnatelný.

Při prohledávání obrovských implicitních grafů by často prohledávání do šířky nebylo úspěšné ani z důvodu překročení vyhrazeného času. Bez dodatečných informací není taková úloha řešitelná. Mnohdy je ale možno nějakým způsobem pro vrcholy implicitního grafu (neboli „pozice“) zdola odhadnout vzdálenost k hledanému cíli. Pokud takový „heuristický“ odhad získáme mnohem rychleji než prohledáním příslušné části grafu a navíc je tento odhad dostatečně přesný, umožní nám to prohledávání do šířky

výrazně urychlit. Předpoklad, že odhad se pro dvě sousedící pozice bude lišit nejvýš o 1 není nepřirozený. V takovém případě můžeme použít modifikované prohledávání do šířky, kdy udržujeme současně tři fronty. Do první dáváme sousedy prohledávané pozice, pokud je pro ně heuristický odhad nižší. Do druhé fronty dáváme sousedy, pokud je pro ně heuristický odhad stejný a do třetí frony sousedy s horším heuristickým odhadem. Zpracováváme pozice první fronty, dokud se nestane, že je tato fronta prázdná. V takovém případě přečíslujeme fronty a založíme novou „třetí“ frontu. Pokud bychom neměli garantováno, že se heuristické odhady sousedících pozic liší nejvýš o 1, mohli bychom použít prioritní frontu. Logaritmický čas navíc bude nejspíš lepším pořadím prohledávaných pozic vykoupen. Efektivita takovéhoto algoritmu tuším pojmenovaného $A^\star$ závisí na kvalitě heuristického odhadu. Pokud by odhad byl naprosto přesný, byl by čas algoritmu roven počtu hran nejkratší cesty a hran z ní vybočujících krát čas na výpočet jednoho heuristického odhadu. Čím horší je heuristický odhad, tím více trpí algoritmus problémy prohledávání do šířky.

Metoda postupného prohlubování, která hlavní neduhy prohledávání do šířky řešila je použitelná i při prohledávání s heuristickým odhadem. V tomto případě je ale užitečné změnit pořadí sousedů při prohledávání a nejprve prohledávat vrcholy s menším heuristickým odhadem (liší-li se heuristické odhady nejvýš o 1, jedná se o konstantní zdržení, obecně je zdržení nejvýš logaritmické v maximálním stupni vrcholů). Stačí prohledávání do hloubky modifikovat tak, že ukončujeme prohledávání pozice, pokud její vzdálenost od počáteční pozice a heuristický odhad vzdálenosti do cílové pozice překročí stanovenou mez. Výhoda prohledávání do hloubky se může projevit při výpočtu heuristického odhadu. Při prohledávání do šířky můžeme vycházet pouze z aktuální pozice, při prohledávání do hloubky můžeme využít i mezivýsledky výpočtu heuristiky pro sousední vrchol. Při označování vrcholů je v tomto případě užitečné hashovat jak aktuálně dosaženou hloubku, tak heuristický odhad vzdálenosti do cíle. Mezivýsledky uložené v hashovací tabulce při prohledávání do hloubky h výrazně urychlují prohledávání do hlubky $h+1$. Popsaný algoritmus bývá označován $IDA^\star$.

Vraťme se k prohledávání malých grafů. U nich paměťové a časové nároky velikosti počtu hran a vrcholů nečiní problém a výběr způsobu prohledávání závisí na dalších požadavcích algoritmu. Pokud je naší snahou nalézt nejkratší (co do počtu hran) cestu vedoucí z počátečního do cílového vrcholu, bude prohledávání do šířky lepší. Pokud ale naší snahou je něco o grafu spočítat, je často zpracování v pořadí lokálně zachovávajícím sousednost užitečnější. V lokálních proměnných můžeme shromažďovat informace z procházených podstromů pod jednotlivými sousedy vrcholu a z těchto informací pro podstromy vytvořit novou informaci z „většího podstromu“. Při zpravocání počátečního vrcholu pak získáme informaci o celém prohledaném stromě(či grafu). Příkladem je určování komponent hranové či vrcholové dvousovislosti, testování rovinnosti grafu, určování komponent silné souvislosti(v orientovaném grafu).

Pro prohledávání do šířky znám jedinou netriviální aplikaci ... při pokusu aplikovat metodu rozděl a panuj na rovinné grafy se nám hodí rozdělit graf s n vrcholy na dvě části, kde větší z nich má nejvýš $2n/3$ vrcholů a jejichž hranice je tvořena nejvýš $4\sqrt{n}$ vrcholy. V první části rozdělovacího algoritmu je použito prohledávání do šířky, pro dokončení algoritmu je potřeba použít prohledávání do hloubky (rozdělovací algoritmus pracuje v čase $O(n)$).

Zde uvedené metody prohledávání fungují pro orientované grafy, tedy i pro neorientované grafy.

Následuje ukázka přibližného řešení rekurence ... rekurence ze čtvrtého příkladu úvodní kapitoly.

Výpočet rekurence $t(n) = \max_{k\le n/2}(t(k)+t(n-k)+\min\{n, ck\log n\}) = \max\{\max_{k\le n/(c\log n)} t(k)+t(n-k)+ck\log n, \max_{k=n/(c\log n)}^{n/2} t(k)+t(n-k)+n\}$.

První odhad ($k = n/2$): $t(n) \ge n\log n$. Odtud $t(n) = \max\{\max_{k\le n/(c\log n)} t(k)+t(n-k)+k\log n, t(n/(c\log n))+t(n-n/(c\log n))+n\}$.

Z prvního odhadu $t(n) \ge (n/c\log n)\cdot(\log n - \log(c\log n)) + (n(1-1/(c\log n)))\cdot(\log n + \log(1-1/(c\log n)))+n = n/c - n\log(c\log n)/(c\log n)+n\log n - n/c + n(1-1/(c\log n))\log(1-1/(c\log n))+n = n\log n - n\log(c\log n)/(c\log n) + n(1-1/(c\log n))\log(1-1/(c\log n))+n$. Výraz $(1-x)\log(1-x) = (1-x)\log(e)\ln(1-x)$ v okolí 0 můžeme odhadnout pomocí $-x(1-x)\log(e)$, tedy dostáváme $t(n) \ge n\log n - n\log(c\log n)/(c\log n) - n(1-1/(c\log n))\log e/(c\log n) + n$.

Zkusme hledat $t(n)$ ve tvaru $t(n) \ge K(n)\cdot n\log n$. Kde $K(n)$ bude velmi pomalu rostoucí funkce ($K(n/(c\log n)) \approx K(n(1-1/(c\log n)) \approx K(n)$).

Dostaneme rekurenci $t(n) \ge K(n)\big((n/(c\log n))\cdot(\log n - \log(c\log n)) + (n(1-1/(c\log n)))\cdot(\log n + \log(1-1/(c\log n)))\big)+n = K(n)\big(n/c - n\log(c\log n)/(c\log n)+n\log n - n/c + n(1-1/(c\log n))\log(1-1/(c\log n))\big)+n = K(n)\big(n\log n - n\log(c\log n)/(c\log n)+n(1-1/(c\log n))\log(1-1/(c\log n))\big)+n \approx K(n)\big(n\log n - n\log(c\log n)/(c\log n) - n(1-1/(c\log n))\log e/(c\log n)\big)+n$.

Pokud bude $K(n)n\log(c\log n)/(c\log n) \approx n$, budeme znát dostatečně přesný odhad $t(n)$. Volme $K(n) = K\cdot(c\log n)/\log(c\log n)$. Pak $K(n/(c\log n)) = K\cdot c(\log n - \log(c\log n))/\log(c\log n - c\log(c\log n))$ vyhovuje našim požadavkům.

Máme tedy $t(n) = Kcn\log^2 n/\log(c\log n)$.

Kontrolní dosazení: $t(n) = Kc(n/(c\log n))(\log n - \log(c\log n))^2/\log(c\log n - c\log(c\log n)) + Kcn(1-1/(c\log n))(\log n + \log(1-1/(c\log n)))^2/\log(c\log n + c\log(1-1/(c\log n)))+n \approx Kc(n/(c\log n))(\log^2 n - 2\log n\log(c\log n)+\log^2(c\log n))/\log(c\log n - c\log(c\log n)) + Kcn(1-1/(c\log n))(\log^2 n - 2\log e/c + \log^2 e/(c^2\log^2 n))/\log(c\log n - \log e/\log n)+n$. Další úpravou $t(n) \approx Kn\log n/(\log(c\log n)) - 2Kn + (Kcn\log^2 n - Kn\log n)/\log(c\log n)+n = Kcn\log^2 n/\log(c\log n)+n-2Kn$. Stačí $K \approx 1/2.12347$?

Pro $k = n/(c\log n)$ dostáváme vzhledem k tomu, že $t'(n) \approx Kc\log^2 n/\log(c\log n)$ a $t(n-k+1)+t(k-1)-t(n-k)-t(k) \approx t'(n-k)-t'(k) \approx Kc(\log n + \log(1-1/(c\log n)))^2/\log(c\log n + c\log(1-1/(c\log n))) - Kc(\log n - \log(c\log n))^2/\log(c\log n - c\log(c\log n))$. Další úpravou $t(n-k+1)+t(k-1)-t(n-k)-t(k) \approx Kc(\log n - \log e/(c\log n))^2/\log(c\log n - \log e/\log n) - Kc(\log^2 n - 2\log n\log(c\log n)+\log^2(c\log n))/\log(c\log n - c\log(c\log n))$. Po dalších odhadech $t(n-k+1)+t(k-1)-t(n-k)-t(k) \approx Kc\log^2 n/\log(c\log n) - 2K\log e/\log(c\log n) - Kc\log^2 n/\log(c\log n) + 2Kc\log n - Kc\log(c\log n) \approx 2Kc\log n < c\log n$. Proto je maximum v původní rekurenci nabýváno zhruba v bodě $n/(c\log n)$ a výsledek je zhruba $cn\log^2 n/2.12347\log(c\log n)$.

9. Spojování tříd ekvivalence

10. Reprezentace kostry