Datový typ prioritní fronta

Semestrální práce z předmětu 36PT

Martin Tůma
Cvičení 113, Út 18:00

22. května 2004

Specifikace problému

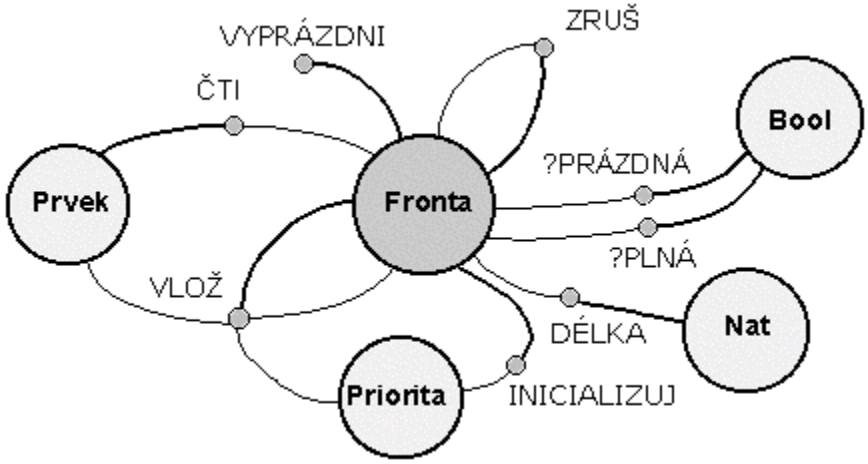
Často potřebujeme přístup k informacím, tak aby tyto byly seřazeny podle hodnot klíče, avšak ne najednou ke všem, ale z času na čas potřebujeme vybrat záznam s nejvyšší (nejnižší) hodnotou klíče. Navíc mezi jednotlivými výběry mohou být vloženy další záznamy.

Příkladem mohou výpočetní systémy, kde požadavky na přidělení procesoru jsou typicky obsluhovány podle svých priorit. Když se procesor stane volným, přidělí se úloze s nejvyšší prioritou. Během jejího výpočtu mohou přijít další úlohy a když její výpočet skončí vybere se opět úloha s aktuálně nejvyšší prioritou, atd.

Odpovídající abstraktní datový typ, tedy musí poskytovat operace vlož prvek a vyber největší prvek. Takovýto ADT je právě **prioritní fronta**.

Kromě použití v aplikacích, které mají uvedený charakter, se prioritní fronta používá také v některých složitějších algoritmech, např. grafových. Libovolnou prioritní frontu také můžeme použít na seřazení jejich prvků – Opakováním výběru největšího prvku, získáme jejich sestupné seřazení.

Signatura



INICIALIZUJ(i)	Vytvoření prioritní fronty s maximální prioritou *i*
VYPRÁZDNI	Vyprázdnění (zrušení) celé fronty
VLOŽ(p,i)	Vložení nového prvku *p* s prioritou *i* do fronty
ČTI(p)	Přečtení prvku *p* s maximální prioritou z fronty
ZRUŠ	Odebrání prvku s maximální prioritou z fronty
DÉLKA:n	Zjištění počtu prvků *n* ve frontě
?PLNÁ	Test na plnou frontu
?PRÁZDNÁ	Test na prázdnou frontu

Axiomy

?PRÁZDNÁ(INICIALIZUJ(i)) = true
?PRÁZDNÁ(VLOŽ(p,i)) = false
ČTI(p) = if not ?PRÁZDNÁ then $p=p_{max(i)}$ else chyba
ZRUŠ(VYPRÁZDNI) = chyba
VLOŽ(?PLNÁ) = chyba

Rešerše

Prioritní frontu lze implementovat v zásadě dvěma způsoby:

- Implementace pomocí lineární struktury (pole, spojový seznam)
- Implementace pomocí binárního stromu(haldy)

Implementace pomocí lineární struktury

Při použití pole uchováváme prvky prioritní fronty uspořádané, kdy operace výběru prvku přímo odebere největší prvek z konce fronty, ale vložení prvku musí zachovat uspořádaní. Operace odebrání maxima se vykoná v konstantním čase. Vkládaní, je založeno na myšlence vložení prvku na konec seřazené fronty a jeho postupnou výměnnou s prvky před ním, pokud prvek před ním je větší, až se nalezne pozice kam vkládaný prvek patří. Jde o stejný princip, který je používán v algoritmu řazení vkládáním (Insertsort) kde se postupně každý prvek vloží na správné místo do podposloupnosti uspořádaných prvků před ním. V nejhorším případě je nutno projít všechny prvky v prioritní frontě.

Alternativně lze prioritní fronta implementovat neuspořádaným polem, kdy se nevětší prvek hledá až při jeho vybíraní. V tomto případě má operace vložení prvku konstantní čas, kdežto operace výběru největšího prvku v nejhorším případě potřebuje projít všechny prvky pole.

Při implementaci pomocí spojového seznamu můžeme prvky opět udržovat seřazené nebo neuspořádané. Zatímco při implementaci pomocí pole v obou alternativách jsme vkládali a vybírali prvek z konce pole, při implementaci neuspořádaným obousměrným spojovým seznamem, prvek vkládáme na jeho začátek a největší prvek po jeho nalezení přímo odebereme. Implementace pomocí uspořádaného spojového seznamu pak odebírá prvek ze začátku seznamu a před jeho vložením se musí nalézt místo pro vložení.

Časová náročnost operací vložení a výběru největšího prvku v nejhorším případě v závislosti na uspořádanosti prvků ve spojovém seznamu je tedy stejná jako pro pole.

Uchovávaní prvků prioritní fronty jako neuspořádané posloupnosti je příkladem tzv. trpělivého (lazy) přístupu k řešení problému implementace její operací, kdy to co v rámci dané operace nemusíme udělat odložíme na později. Na druhé straně, uchovávaní prvků prioritní fronty jako uspořádané posloupnosti je příkladem tzv. netrpělivého (eager) přístupu k řešení problému, kdy v rámci operace vykonáme co nejvíce práce potřebné pro efektivní implementaci jiných operací.

Implementace pomocí binárního stromu(haldy)

Efektivnějším uložením prvků dynamické množiny zohledňující velikost klíčů je BVS, kde operace vložení a nalezení prvku se zadaným klíčem jsou O(h), kde h je výška stromu. V případě ADT prioritní fronta je operace výběru prvku specifikována jako výběr prvku s největším klíčem. Prvek v BVS s největším klíčem nalezneme tak, že budeme z kořene sledovat pořád ukazatele na pravý podstrom až k listu, kde ukazatel je nil.

Výška stromu a je v průměrném případě 1,39log2n, z čehož vyplívá O(1,39log2N), což je lepší než O(N/2) pro pole nebo seznam. V průměrném případě půjde při implementaci pomocí BVS o zlepšení těch operací, kterých časová náročnost je v nejhorším případě N při implementaci pomocí pole nebo seznamu.

Nejlepších výsledků pak dosáhneme při použití úplného binárního stromu s výškou O(log2 N). Navíc potřebujeme, aby uložení prvků ve stromě zohledňovalo velikosti jejich klíčů tak, aby bylo možno efektivně určit prvek s největším klíčem. Takovou strukturou je halda. Strom má vlastnost haldy, když klíč v každém vrcholu je větší nebo roven klíčům v jeho následnících, pokud je má. Ekvivalentně, klíč v každém vrcholu je menší nebo roven klíči v jeho předchůdci, pokud ho má. Z uvedené vlastnosti plyne, že žádný vrchol ve stromě s vlastností haldy nemá klíč větší než kořen.

Porovnání složitostí

	Vložení prvku	Výběr prvku
seřazený spojový seznam	O(N)	O(1)
neseřazený spojový seznam	O(1)	O(0)
Halda	$O(\log_2 N)$	$O(\log_2 N)$

Popis implementace

Fronta je implementováná nejlepší možnou metodou, tzn. pomocí BVS se strukturou haldy. V zásadě by šlo implementovat haldu ještě rychleji pomocí pole, kde by odpadlo složitější určování pozice posledního prvku, takováto implementace ale neumožňuje dynamickou změnu velikosti fronty.

Přečtení prvku s největší prioritou, tedy **funkce čti**, je zcela transparentní, jde pouze o přečtení hodnoty kořene. Jeho odstranění z fronty však vyžaduje náhradu prvku, který byl kořenem, jiným prvkem, což může vést k porušení vlastnosti hlady, pokud tento prvek má menší klíč. Na druhé straně, vložíme-li prvek, bude tento dalším listem a pokud má větší klíč než jeho předchůdce, opět došlo k porušení vlastnosti haldy. Uvedené situace můžeme zobecnit na dva případy, kdy se poruší vlastnost haldy a nutno ji obnovit, chceme-li využívat její vlastnost.

V prvním případě je vlastnost haldy porušena, protože klíč v některém vrcholu se stane menším než klíče v jednom nebo obou následnících. V tomto případě ho musíme vyměnit s větším z jeho následníku, což může vést k porušení vlastnosti o úroveň níže a postup výměny opakujeme. Tímto postupem prvek, který takto porušil vlastnost haldy putuje směrem dolů, k listům. Jeho postup skončí, když bude splněna vlastnost haldy anebo se stane listem. V implementaci tuto obnovu struktury provádí **procedura dolu**.

Celkový postup při odebrání prvku z fronty (**funkce zruš**) je následující:

- vyměníme kořen s posledním prvkem
- smažeme poslední prvek
- spustíme obnovu haldy od kořene (procedura dolu)

Ve druhém případě je vlastnost haldy porušena, protože klíč v některém vrcholu se stane větším než klíč v jeho předchůdci. Pro obnovení vlastnosti haldy ho musíme vyměnit s jeho předchůdcem, což v tomto případě může vést k porušení vlastnosti haldy o úroveň výše a postup opakujeme. Tímto způsobem prvek, který porušil vlastnost haldy putuje směrem nahoru, ke kořenu. Skončíme, když se obnoví vlastnost haldy anebo se nový prvek stane kořenem. Uvedený postup implementuje **procedura nahoru**.

Celkový postup při vkládání prvku (**funkce vlož**):

- nalezneme následující volnou pozici v haldě a umístíme na ní nový prvek
- spustíme obnovu haldy od nového listu (procedura nahoru)

Problém při implementaci haldy pomocí stromu je výše zmíněné nalezení následující volné pozice v haldě pro vložení prvku, respektivě nalezení předchozí poslední pozice při odebírání prvku z haldy. V případě, že je poslední prvek levý syn pro vkládání, respektivě pravý syn pro odebrání je situace jednoduchá - další/předchozí prvek je sourozenec aktuálního hledaného prvku. V opačném případě je pro nalezení správné pozice nutné poměrně složitě projít strom.

Při vkládání postupujeme od posledního prvku stromem nahoru, dokud je aktuální prvek pravým synem. Pokuď je levým synem nebo kořenem začneme postupovat od sourozence(u kořenu je sourozenec opět kořen) vlevo dolů, dokuď je prvek vnitřním prvkem, tzn dokuď má levého i pravého syna. Hledaná nová pozice je pak levý syn posledního prvku.

Při odebírání, postupujeme nahoru dokuď je aktuální prvek levým synem. Poté postupujeme vpravo dolů od sourozence. Hledaný prvek je posledním pravým synem.

Prioritní fronta je implementována jako objekt v samostatné knihovně **HTPRIFO**. Typ dat (**Prvek**) je definován **typem Tdata** a může být prakticky libovolný (pro testy použit integer). Priorita je, stejně jako délka fronty kterou vrací **funkce délka**, definována jako word. Maximální délka fronty a maximální priorita je tedy 65535.

Kromě výše popsaných a v signatuře uvedených operací je navíc ještě pro pohodlnější práci s frontou implementována **funkce vyber(p)**, která provede posloupnost funkcí čti a zruš.

Ošetření chybových stavů je řešeno tak, že každá z operací, která může vyvolat chybu(vlož, čti, zruš, vyber) je řešena jako funkce s návratovou hodnotou typu boolean podle úspěšnosti provedení akce. Chybové stavy jsou ošetřeny a funkce vrací false, při pokusu o nepovolenou operaci. V opačném případě vrací true.

Pojmenování operací v implementaci se díky omezením daným programovacím jazykem poněkud liší od pojmenování v signatuře, a to podle následujících pravidel:

- není použita diakritika
- znak „?“ je nahrazen slovem „je“ Příklad: ?PRÁZDNÁ -> jePrazdna

Více podrobností k použitým algoritmům lze nalézt v komentářích u vlastního zdrojového kódu unity.

Ověření algoritmu

Pro ověření funkce prioritní fronty byly provedeny testy ověřující axiomy pro prioritní frontu. Všechny testy proběhly úspěšně, lze se tedy domnívat, že implementace je korektní. Podrobný průběh testů by měl být zřejmý z výpisů testovacího programu a z komentářů v jeho zdrojovém kódu.

Překlad programu a test byl proveden jak v prostředí Windows s překladačem Borland turbo pascal 7.0, tak v prostředí Linuxu pomocí překladače Free Pascal 1.0.10. Při použití TP7.0 nastává problém s maximální velikostí volné paměti (640kB) pro aplikaci a nelze tudíž dosáhnou naplnění celé prioritní fronty. Implementované bezpečnostní mechanismy nicméně velikost volné paměti kontrolují a nedovolí zhroucení knihovny. Je vygenerována chyba, kterou může aplikace používající tuto knihovnu zachytit a podle potřeby se rozhodnout jak pokračovat.

Výpočet a ověření složitosti algoritmu

Maximální délka cesty při obnovování haldy je $\log_2 N$ (výška úplného stromu). Maximální délka cesty při hledání následující/předchozí poslední pozice je $2\log_2 N$ (Nastává při přechodu z levé části stromu do pravé, či naopak). Protože v každém uzlu provádíme konstantní počet porovnání a přiřazení *k*, je asymptotická operační složitost vkládání i vybírání z haldy:

$k \cdot \log_2 N + 2k \cdot \log_2 N = O(\log_2 N)$

Pro ověření složitosti algoritmu jsem uskutečnil test rychlosti seřazení posloupnosti náhodných čísel. Výsledky pro jednotlivé operace jsou uvedeny v tabulce. Pro porovnání jsou uvedeny i výsledky prioritní fronty implementované pomocí lineárního spojového seznamu. (knihovna LPRIFO). Maximální priorita činila při testu 10. Testovací sestava: Duron 900MHz, 256MB RAM, Linux 2.4.25, překladač fpc

	Spojový seznam		Halda	
Počet prvků	Vložení[ms]	Výběr[ms]	Vložení[ms]	Výběr[ms]
1000	1	<1	<1	<1
5000	10	<1	1	1
10000	82	1	2	2
20000	1060	1	4	4
30000	3226	1	6	7

Paměťová složitost reprezentace jednoho prvku fronty je $S_d + 3S_p$ kde S_d je paměť potřebná pro samotný datový typ dat Tdata a S_p je velikost ukazatele.

Závěr

Prioritní fronta implementovaná pomocí binárního stromu se strukturou haldy se ukázala jako velice efektivní. Při porovnání s nejjednodušší implementací pomocí spojového seznamu je s narůstajícím počtem prvků rozdíl efektivity algoritmů enormní. Rozdíl paměťové náročnosti je přitom nepatrný, pro implementaci pomocí dvoucestně zřetězeného seznamu je potřeba pouze o jeden ukazatel na prvek méně než při implementaci pomocí haldy.

Použít implementaci pomocí spojového seznamu je tedy výhodné akorát v případě, že nám při použití nezáleží na času seřazení, ale potřebujeme maximální rychlost odebírání z fronty, (či naopak při implementaci pomocí neseřazeného spojového seznamu), nebo je předpokládaný počet prvků velmi malý.

Přílohy:

HTPRIFO.PAS – zdrojový kód vlastní knihovny implementující prioritní frontu pomocí BVS se strukturou haldy
OVERENI.PAS – zdrojový kód programu pro ověření implementace prioritní fronty
TEST.PAS – zdrojový kód programu pro test časvé složitosti operací prioritní fronty
LPRIFO.PAS – zdrojový kód implementace prioritní fronty pomocí spojového seznamu použité pro srovnání.

Použitá literatura:

[1] Hudec Bohuslav, Programovací techniky, skripta FEL–CVUT, Vydavatelství CVUT 2001
[2] Heaps and Heapsort http://cis.stvincent.edu/swd/heaps/heaps.html
[3] Pascal a objekty http://www.sejda.net/programming/objectpascal/