UNICORN COLLEGE

ZÁKLADY NÁVRHU A OPTIMALIZACE ALGORITMŮ

Cvičení 3 - Vyhledávání

Autor: **Martin Budínský**

2019

1. Úvod

Tato práce si klade za cíl změřit rychlost tří operací - vložení (*insert*), vyhledání (*find*) a smazání (*delete*) - nad datovými strukturami pole (*Unsorted Array*), setřízené pole (*Sorted Array*) a binární vyhledávací strom (*Binary Search Tree*), naměřená data následně porovnat a diskutovat s očekávanou algoritmickou složitostí.

2. Popis podmínek testování

Testování popsaných operací bude probíhat následujícím způsobem:

Mějme pole dvojic N a M. V prvním kroku vytvoříme N polí o velikosti M, přičemž každé z polí bude obsahovat M náhodných hodnot z intervalu <0, M-1>. Tuto operaci provádíme třikrát pro vytvoření struktury dat *vkládaných*, *hledaných* a *mazaných*. V druhém kroku vytvoříme tři pole velikosti N prázdných instancí jednotlivých datových struktur (*UnsortedArray, SortedArray, BinarySearchTree*).

Následující krok je samotné měření. Iterujeme referenčními poli pro *insert* a *vkládáme* vždy do i-té instance třídy UnsortedArray, přičemž celou tuto operaci měříme. Tímto dostáváme dobu trvání N operací insert do M struktur; výsledek tedy dělíme číslem (N*M) pro získání doby trvání jedné operace.

Analogicky postupujeme pro měření operací *find* a *delete*, všechna měření opakujeme pro všechny dvojice M,N a datové struktury. Celý algoritmus zopakujeme desetkrát, čímž nakumulujeme 10 výsledků pro každou z dvojic M,N. Nad těmito výsledky počétáme aritmetický průměr a medián doby trvání jednotlivých operací.

3. Očekávané chování

Nejprve se podíváme na teoretickou složitost operací insert, find a delete pro každou z testovaných datových struktur:

Unsorted Array:

Operace Worst-case

Search O(n)

Insert O(n)

Delete O(n)

Vzhledem k tomu že ve strukturách nedovolujeme uchovávání duplicitních hodnot, *insert* samotný trvá *O(1)* – jednoduše vložíme prvek na konec pole – ale prvně musíme vyhledat, zda-li prvek již není v poli obsažen, taková operace nám v nejhorším případě bude trvat *O(n).*

Operace *find* lineárně prohledá pole – v nejhorším případě *O(n)* a operace *delete* funguje na úplně stejném principu – také v *O(n).*

Sorted Array:

Operace	Average	Worst-case
SearchB	O(log n)	O(log n)
SearchInt	O(n)	O(n)
Insert	O(n)	O(n)
Delete	O(n)	O(n)

Vyhledávání v setřízeném poli je oproti nesetřízenému poli rychlejší, protože využíváme principu binárního vyhledávání a tedy dosáhneme času *O(log n).*

Problémová je však operace *insert*, kde musíme nejprve v čase *O(log n)* vyhledat, zda-li prvek v poli existuje, poté ho vložit na jeho odpovídající místo a zbylé prvky posunout. To nám tedy zabere v nejhorším případě *O(n).*

Operace *delete* probíhá zcela analogicky, akorát prvky posuneme „doleva“ tak, abychom zaplnili prázdné místo.

Binary Search Tree:

Operace	Worst-case
Search	O(h)
Insert	O(h)
Delete	O(h)

h – výška stromu

Operace *insert* a *delete* v binárním vyhledávacím stromu obě závisí na operaci *search.* Vy- hledáváme-li v binárním stromu, v nejhorším případě můžeme jít až do listu, který je nejdále tak, jako je výška tohoto stromu.

Operaci *insert* provádíme vždy tak, že vytvoříme nový list, prvně tedy vyhledáme, zda-li prvek ve stromu je, v případě že není, pak ho na jeho korespondující pozici prostě vložíme, tedy tato operace trvá taky *O(h).*

Operace *delete* je zajímavá pouze v případě, že mažeme vnitřní vrchol, kdy musíme najít v podstromu levého potomka nejpravější prvek a ten vyměnit za mazaný vrchol. Toto může trvat opět *O(h).*

Nyní už tedy zbývá určit, jak je strom vysoký. Implementace binárního stromu, které využíváme v této práci nevyvažuje podstromy tak, aby skutečná výška stromu byla *O(log n)* jako v případě AVL stromů, avšak v momentě kdy do takovéto struktury budeme vkládat hodně náhodných dat ve velkém rozsahu, není příliš pravděpodobné, aby strom byl obzvláště nevyvážený a tedy můžeme očekávat asymptotickou složitost *O(log n)* u operací *insert* respektive *find* a *delete.*

Očekávání:

Nejrychlejší operaci *insert* na *velkých datech* by měl tedy mít binární vyhledávací strom z důvodu teoretické složitosti *O(log n).* Hned za ním by mělo být setřízené pole, protože princip vyhledání prvku funguje stejně jako v binárním vyhledávacím stromu, ale následné vložení musí posunout část pole. Nejpomalejší by mělo být nesetřízené pole, kde vyhledání prvku trvá nejdéle.

Operace *find* na velkých datech by měla být v principu stejně rychlá na binárním vyhledávacím stromu a setřízeném poli. Nesetřízenému poli by měla operace *find* trvat nejdéle.

Při operaci *delete* by měl být*,* obdobně jako u operace *insert,* na prvním místě binární vyhledávací strom, na druhém setřízené pole a na posledním místě nesetřízené pole. Operace *delete* má totiž ošekávanou složitost *O(log n)* v binárním vyhledávacím stromu, oproti tomu setřízené pole *O(n)* a stejně tak nesetřízené pole. V nesetřízeném poli nejen, že musíme vyhledat prvek v *O(n)*, ale následné smazání nám zabere dalších *O(n)* z důvodu posunu prvků.

Na malých datech si neodvažuji predikovat rychlost operací, zde se totiž asymptotická notace nedá brát vážně a tedy může klidně být nejrychlejší nesetřízené pole.

4. Výsledky měření

Jednotlivé výsledky jsou dostupné v souborech *times_ avg.txt* a *times_ med.txt.*
Výsledky operace *insert* pro 10 prvků z rozsahu 0-9 jsou v proměrů nepatrně rychlejší na nesetřízeném poli, než na poli setřízeném. Binární vyhledávací strom zůstává o celý desetinný řád pozadu. Toto se dá vysvětlit velkou objektovovou a tím pádem paměťovou složitostí. Binární vyhledávácí strom naopak, dle očekávání, exceluje v druhém extrému měření, vkládání 10 000 prvků.

Operace *find* je na malých datech průměrně nejrychlejší na střízeném poli. BVS opět zaostává ze stejných důvodů, jako u operace *insert.* Na velkých datech je průměrně nejrychlejší binární vyhledávání setřízeného pole. Nesetřízené pole zaostává díky asymptotické složitosti *O(n)* operace *find.*

Operace *delete* má na malých datech obdobný průměh s nejrychlejším nesetřízeným polem. Na datech velkých vítězí o řád binární vyhledávací strom díky očekávané asymptotické složitosti *O(log n)* operace *delete.*

5. Závěr

Z naměřených hodnot vyplývá, že každá struktura má rozdílné chování na základě velikosti a objemu dat, nad kterými pracuje, proto bychom měli vždy pečlivě uvážit, s jakými daty budeme pracovat a podle toho zvolit vhodnou strukturu pro jejich skladování. Např. pro malá data bude lepší využít struktury „primitivního“ pole (UnsortedArray), z důvodu velmi nízké paměťové náročnosti, případně objektové struktury, naopak pro větší objem dat bude vhodné sáhnout po „sofistikovanější“ struktuře, například v podobě SortedArray, či BinarySearchTree.