

Projet LSINF1121 - Algorithmique et structures de données

-

Rapport intermédiaire Mission 4

Groupe 26

Laurian DETIFFE
(6380-12-00)

Sundeeep DHILLON
(6401-11-00)

Alexis MACQ
(5910-12-00)

Xavier PÉRIGNON
(8025-11-00)

Thibaut PIQUARD
(4634-13-00)

Thomas WYCKMANS
(3601-12-00)



Année académique 2015-2016

Questions et réponses

1. Citez au moins quatre implémentations différentes d'un dictionnaire (table de symboles). Précisez, dans chaque cas, quelles sont les propriétés principales de ces implémentations. Dans quel(s) cas s'avèrent-elles intéressantes ? Quelles sont les complexités calculatoires de leurs principales méthodes ?

(a) **Recherche séquentielle (liste désordonnée) :**

Une option simple pour la structure de données d'une table de symboles est une liste chaînée de noeuds qui contiennent des clés et des valeurs. Le principe de l'algorithme est qu'il parcourt la liste en comparant la clé de recherche avec la clé de chaque noeud dans la liste. Si les clés concordent, il retourne la valeur associée. Sinon, il retourne *null*. La mise en œuvre d'une liste liée à la recherche séquentielle est trop lente pour qu'elle puisse être utilisée pour résoudre d'énormes problèmes. Cependant, elle est idéale pour les petits problèmes.

(b) **Binary tree search (BST) :**

Un arbre de recherche binaire est un arbre binaire où chaque noeud a une clé (et une valeur associée) et qui satisfait la restriction que la clé dans un noeud quelconque est plus grande que celle de tous les noeuds du sous-arbre gauche de ce noeud et plus petite que celle de tous les noeuds du sous-arbre droit de ce noeud. La mise en œuvre de cette structure est très facile à implémenter. Cependant, si l'arbre ressemble à une liste chaînée, cela risque d'augmenter fortement la complexité.

(c) **Separate chaining (tableau de listes) :**

Dans la méthode dite de chaînage séparé, chaque bucket est indépendant, et a une sorte de liste des entrées avec le même index. Le temps d'exécution d'une opération d'une table de hachage est le temps de trouver le bucket (qui est constant) plus le temps de l'opération concernant la liste. Dans une bonne table de hachage, chaque bucket a 0 ou 1 entrée, parfois 2 ou 3, mais rarement plus que cela. Par conséquent, les structures sont efficaces dans le temps et l'espace. Cependant, ce procédé hérite également des inconvénients des listes liées. En effet, lors du stockage de petites clés et de valeurs, la surcharge de l'espace du prochain pointeur dans chaque entrée peut être importante.

(d) **Linear probing (tableaux parallèles) :**

Linear probing est réalisée en utilisant deux valeurs : une étant comme une valeur de départ et une autre étant comme un intervalle entre les valeurs successives en arithmétique modulaire. La seconde valeur, qui est la même pour toutes les clés et connue sous le nom *stepsize*, est ajoutée à plusieurs reprises à la valeur de départ jusqu'à ce qu'un espace libre est trouvé, ou que toute la table soit parcourue.

$$newLocation = (startingValue + stepSize) \% arraySize$$

Cet algorithme offre une bonne mise en cache de la mémoire (si *stepsize* est égal à 1), grâce à la bonne localité de référence.

Résumé :

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		key interface	memory (bytes)
	search	insert	search hit	insert		
<i>sequential search (unordered list)</i>	N	N	$N/2$	N	<code>equals()</code>	$48 N$
<i>binary tree search (BST)</i>	N	N	$1.39 \lg N$	$1.39 \lg N$	<code>compareTo()</code>	$64 N$
<i>separate chaining (array of lists)</i>	$< \lg N$	$< \lg N$	$N/(2M)$	N/M	<code>equals()</code> <code>hashCode()</code>	$48 N + 64 M$
<i>linear probing (parallel arrays)</i>	$c \lg N$	$c \lg N$	< 1.50	< 2.50	<code>equals()</code> <code>hashCode()</code>	between $32 N$ and $128 N$

2. Souvenez vous de la question suivante proposée en bilan sur la mission sur les tris : Étant donné un ensemble S de taille n , et un nombre x . Décrivez un algorithme efficace utilisant une HashTable pour trouver s'il existe une paire (a, b) avec $a \in S$, $b \in S$ telle que $a + b = x$. Quelle est la complexité de votre algorithme ? Est-elle meilleure que votre solution qui utilisait un tri ?
3. Démontrez que $(a + b) \% M$ est équivalent à $((a \% M) + b) \% M$. En quoi cette propriété peut être utile pour construire une fonction de hachage sur les String. Expliquez comment Java calcule une fonction de hachage sur les String ? Quelle est la complexité pour calculer 1 fois et N fois le hashCode d'un String.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.
- 11.