Projet LSINF1121 - Algorithmique et structures de données
-
Rapport intermédiaire Mission 4

Groupe 26

Laurian DETIFFE (6380-12-00)
Sundeep 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 ? (Xavier)

   (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 nœuds du sous-arbre droit de ce noeud. La mise en oeuvre de cette structure est très facile à implémenter. Cependant, si l'abre 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 les 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) search	insert	average-case cost (after N random inserts) search hit	insert	key interface	memory (bytes)
sequential search (unordered list)	N	N	$N/2$	N	`equals()`	$48\,N$
binary tree search (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$	`compareTo()`	$64\,N$
separate chaining (array of lists)	$< \lg N$	$< \lg N$	$N/(2M)$	N/M	`equals()` `hashCode()`	$48\,N + 64\,M$
linear probing (parallel arrays)	$c \lg N$	$c \lg N$	< 1.50	< 2.50	`equals()` `hashCode()`	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 ? (Xavier)

   Concernant le bilan sur la mission sur les tris, la solution était de faire un tri efficace de l'ensemble S (par exemple Quicksort) et ensuite de tester l'addition du premier élément a avec le dernier élément b. Si le résultat de cette addition était inférieur à x, alors on testait l'addition de $(a + 1)$ avec b, et ainsi de suite. Sinon si le résultat de l'addition était supérieur à x, alors on testait l'addition de a avec $(b - 1)$, et ainsi de suite.

   Pour résoudre le même problème avec une HashTable, on pourrait penser à insérer tous les éléments de l'ensemble S dans une HashSet. Cette classe implémente l'interface Set, en utilisant une HashTable. HashSet est implémenté comme une HashMap, dont les clés sont les éléments du HashSet et les valeurs sont toutes une même valeur présente. Cette implémentation offre des performances constantes pour les opérations *add(T t)*, *remove(T t)*, *contains(T t)* et *size()*. Il suffira donc de vérifier si pour chaque élément a de l'ensemble S, il existe b dans la HashSet tel que $b = x - a$. La complexité de cet algorithme est donc de $O(1) + O(n)$ (parcours des éléments de S). Ce qui est plus rapide que l'algorithme de la mission sur les tris.

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. (Xavier)

Prenons G = $(a+b)\%M$ et D = $((a\%M)+b)\%M$.
On doit démontrer que G = D.
Nous pouvons écrire :

$$a = M * Q1 + R1 \text{ where } 0 \leq R1 < M \text{ où Q1 est un entier. a mod M} = R1$$
$$b = M * Q2 + R2 \text{ where } 0 \leq R2 < M \text{ où Q2 est un entier. b mod M} = R2$$
$$(a + b) = M * (Q1 + Q2) + R1+R2$$
$$G = (a + b) \bmod M$$
$$G = (M * (Q1 + Q2) + R1+ R2) \bmod M$$

On peut supprimer les multiples de M lorsqu'on prend mod M,

$$G = (R1 + R2) \bmod M$$
$$D = (a \bmod M + b) \bmod M$$
$$D = (R1 + R2) \bmod M$$

Et donc,

$$G = D = (R1 + R2) \bmod M$$

Cette propriété est utile pour construire une fonction de hashage sur les String car il n'est pas nécessaire de calculer le modulo de chaque valeur des caractères du String. En effet, il suffira de calculer la somme de la valeur des caractères et utiliser une seule fois le modulo à la fin.

La fonction de hashage pour un String est implémentée comme suit :

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$$

Dans cette fonction, le String s a n caractères. Elle calcule la somme des valeurs de chaque caractère, chacun multiplié par 31^r avec r étant l'emplacement inversé du caractère dans le String. La complexité du calcul de hashcode d'un String est de $O(n)$ puisqu'il faut parcourir tous les caractères du String. Mais une fois ce calcul fait, il n'est plus nécessaire de recalculer le hashcode de ce String.

4.
5.
6.
7.
8.
9.
10.
11.