LINGI1121 - Algorithmique et structures de données

Mission 1 : piles, files, listes chaînées

Groupe 9
Aghakhani Ghazaleh (1161-11-00)
Carlier Alexandre (5042-13-00)
Cleeremans Tanguy ()
Paris Antoine (3158-13-00)
Prieëls Antoine (3290-13-00)
Stévenart Meeus Florian (6273-13-00)

25 septembre 2015

1 Réponses aux questions

1. Un type abstrait est une spécification d'un ensemble de données et de l'ensemble des opérations qu'on peut lui appliquer (sans se préoccuper de l'implémentation de celles-ci). Il s'agit en quelque sorte d'un cahier des charges pour une structure de données. [4][2]

   Quant au choix d'utiliser une classe ou une interface en Java pour décrire un type abstrait de données, il semble tout indiqué. Une interface Java n'a pour but que de définir une implémentation ; elle est par définition abstraite et ne fixe aucun aspect de l'implémentation.[6] Cela colle parfaitement à la définition de type abstrait de données.

2. L'opération `push` pour une pile implémentée à l'aide d'une liste s'implémente en parcourant toute la liste jusqu'à arriver à la fin de la liste. À ce moment là, on modifie le pointeur vers l'élément suivant du dernier élement pour le faire pointer vers le nouvel élément.

   L'opération `pop` s'implément à peu près de la même façon. On parcourt la liste jusqu'à arriver à la fin. À ce moment là, on retourne le dernier élément de la liste et on modifie le pointeur de l'avant-dernier élément pour le faire pointer vers `null`.

   Cette implémentation n'est pas efficace parce qu'il faut parcourir toute la liste à chaque opération, ce qui peut vite devenir gênant pour des listes contenant des millions d'éléments...

3. L'implémentation d'une pile par la classe `java.util.Stack` fournit 5 fonctions :
   — `boolean empty()` : teste si la pile est vide ;
   — `E peek()` : retourne l'objet au sommet de la pile sans l'enlever ;
   — `E pop()` : retourne l'objet au sommet de la pile en l'enlevant ;
   — `E push(E item)` : ajoute un élément au sommet de la pile ;
   — `int search(Object o)` : retourne la position de l'objet dans la pile.

   En regardant le code source de cette classe, on constate que la plupart des fonctions sont héritées de la classe `Vector<E>`. En allant ensuite regarder le code source de cette classe, on se rend compte que celle-ci utilise un tableau pour stocker des éléments ainsi qu'un compteur d'éléments. Cela signifie donc qu'en Java, les éléments d'une liste chaînée sont

stockés dans un tableau. Cette solution est beaucoup plus efficace que celle proprosé à la question précédente puisqu'il est possible d'accèder au dernier élément de la liste sans la parcourir entièrement.

4. La méthode la plus efficace pour implémenter une pile avec deux files provient de [3]. Soient deux piles A et B. A contient les éléments au sommet de la pile tandis que B contient les éléments du bas de la pile. La taille de A doit toujours être inférieur à la racine carrée de la taille de B.

   `push` s'effectue simplement en effectuant `enqueue` du nouvel élément sur la pile A et ensuite en effectuant `dequeue` puis `enqueue` sur tous les autres élément de A. De cette manière le nouvel élément est bien le premier élément de A.

   Si le nombre d'élément contenu dans A devient plus grand que la racine carrée du nombre d'élément contenu dans B, on `enqueue` tous les éléments de B sur A un par un et on inverse A et B.

   Enfin, `pop` s'effectue en effectuant `dequeue` sur A et en retournant le résultat si A n'est pas vide et en effectuant `dequeue` sur B dans le cas contraire.

   En terme de complexité, `pop` s'effectue en $\mathcal{O}(1)$.

   Pour `push`, deux cas sont à analyser. Dans le premier cas, $|A| < \sqrt{|B|}$ et on a donc $\mathcal{O}(\sqrt{n})$. Dans le cas contraire, `push` s'effectue en $\mathcal{O}(n)$ mais après cela, A est vide et il faudra un temps $\mathcal{O}(\sqrt{n})$ avant que ce cas ne se reproduise, le coût amorti est donc en $\mathcal{O}(\sqrt{n})$.

5.

6.

7.

8. Un itérateur permet de parcourir une collection d'objets élément par élément. L'implémentation des itérateurs est très générique puisque ceux-ci peuvent être utilisés de la même manière sur n'importe quel type de structure de données.

   La modification des données par l'itérateur lui-même (en faisant appel à sa méthode `remove`) ne posera pas de souci. Cependant, la modification concurrente de la collection par un autre thread pourrait poser problème. En effet, un élément pourrait être renvoyé deux fois ou l'itérateur pourrait sauter un élément de la liste sans le renvoyer. C'est pourquoi la classe `Iterator` doir lancer une `ConcurrentModificationException` si n'importe quel élément de la collection a été modifié ou supprimé par une autre méthode que la sienne.[1]

   Il est possible d'empêcher la modification de la liste en ajoutant un booléen en tant que variable globale. Lors de l'initialisation de l'itérateur, ce booléen serait mis à `false` et chaque appel à `push` ou `pop` ferait passer cette variable à `true`. Ainsi, lors de la progression de l'itérateur une vérification de la variable permettra de déterminer si la pile a été modifiée. L'implémentation aura alors une forme semblable à la suivante [1].

```
1  public class Stack<Item> implements Iterable<Item> {
2
3      private boolean modified = true;
4      //Les autres variables restent inchangees
5
6      /**
7      * Add the item to the stack.
8      */
9      public void push(Item item) {
10         Node oldfirst = first;
```

1. Seules les lignes 3, 15, 28, 37 et 48 ont été modifiées. Le reste provient directement du code source.

```
11          first = new Node();
12          first.item = item;
13          first.next = oldfirst;
14          N++;
15          modified = true;
16      }
17
18      /**
19      * Delete and return the item most recently added to the
           stack.
20      * Throw an exception if no such item exists because the
           stack is empty.
21      */
22      public Item pop() {
23          if (isEmpty()) throw new RuntimeException("Stack␣
               underflow");
24          Item item = first.item;        // save item to return
25          first = first.next;            // delete first node
26          N--;
27          return item;                   // return the saved item
28          modified = true;
29      }
30
31      /**
32      * Return an iterator to the stack that iterates through the
33      * items in LIFO order.
34      */
35      public Iterator<Item> iterator()  {
36          return new ListIterator();
37          modified = false;
38      }
39
40      // an iterator, doesn't implement remove() since it's
           optional
41      private class ListIterator implements Iterator<Item> {
42          private Node current = first;
43          public boolean hasNext() { return current != null; }
44          public void remove() { throw new
               UnsupportedOperationException(); }
45
46          public Item next() {
47              if (!hasNext()) throw new NoSuchElementException();
48              if (modified) throw new
                   ConcurrentModificationException();
49              Item item = current.item;
50              current = current.next;
51              return item;
52          }
53      }
54
55      //Le reste du code n'est pas modifie
```

```
56
57              }
```

Laisser la méthode `remove()` vide ne poserait pas de problème lors de la compilation ou l'exécution du code. Cependant, il ne faut surtout pas oublier que cette méthode est vide car si le programme utilise cette méthode, aucun élément ne sera retiré de la liste. Après une rapide recherche du code source de la classe `Stack`, nous avons pu remarquer que pour remédier à cela, la méthode n'est pas vide mais une `UnsupportedOperationException` est lancée.

9. La notation tilde est définie comme étant[5]

$$g(N) \sim a * f(N) \quad avec \quad \lim_{N \longrightarrow +\infty} \frac{g(N)}{f(N)} = a$$

Comparons la notation $\sim$ aux autres notations que nous avons utilisé précédemment pour mesurer la complexité de fonctions : $\Omega(f(N))$, $\mathcal{O}(f(N))$ et $\Theta(f(N))$. Ces notations se différencient en deux grands types : $\mathcal{O}$ et Ω nous fournissent des bornes respectivement inférieure et supérieure tandis que $\sim$ et Θ nous donnent un ordre de grandeur quant à la complexité d'exécution d'un algorithme. Les deux notations les plus pratiques sont les notations $\mathcal{O}$ et $\sim$. Cependant, alors que $\mathcal{O}$ nous donne simplement une borne supérieure (ce qui est utile pour nous donner une idée générale de ce qui pourrait arriver dans le pire des cas), $\sim$ nous permet de définir aussi une borne inférieure. De plus, cette dernière conserve le coefficient du terme de plus haut degré et nous trouvons donc une précision plus grande pour les tableaux de très grande taille (cfr. Figure 1).

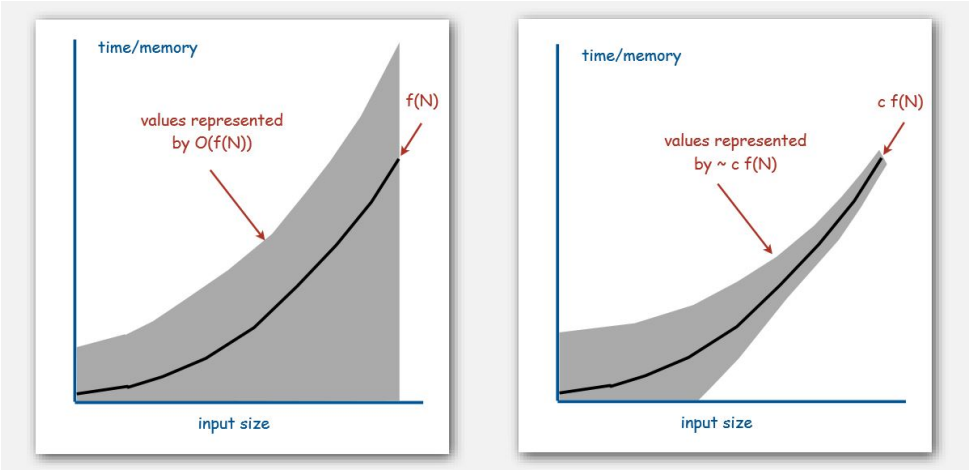

FIGURE 1 – Comparaison de la précision des notations $\mathcal{O}$ et $\sim$ [7].

10.

11.

Références

[1] Les collections d'objets. `https://openclassrooms.com/courses/apprenez-a-programmer-en-java/les-collections-d-objets`. Accessed : 22-09-2015.

[2] Modélisation objet. `http://www-perso.unilim.fr/tayed.ould-braham/C_java_pdf/TC_Modelisation.pdf`. Accessed : 22-09-2015.

[3] One stack, two queues. http://cstheory.stackexchange.com/questions/2562/one-stack-two-queues/2589#2589. Accessed : 22-09-2015.

[4] Type abstrait de données. https://fr.wikipedia.org/wiki/Type_abstrait. Accessed : 22-09-2015.

[5] François Aubry. Order of growth and theta notation. Accessed : 22-09-2015.

[6] Frederick A. Hosch Jaime Nino. *Programming and Object Oriented Design Using Java*. Wiley, 2008.

[7] Robert Sedgewick and Kevin Wayne. Analysis of algorithms. Accessed : 23-09-2015.