

TD n°9 bis

Complexité amortie et Union-Find

1 Analyse de complexité amortie

1.1 Évolution d'un compteur

On considère l'algorithme "école primaire" d'addition de nombres en base 10 avec retenue. L'opération élémentaire que l'on compte est l'addition entre deux chiffres. Soit a un entier. On considère l'algorithme suivant :

```
 $x = 0$   
pour  $i$  de 1 à  $10^a$  faire  
  |  $x = x + 1$ 
```

Exercice 1 Exprimez la complexité au pire et la complexité moyenne de l'algorithme. Exprimez un potentiel pour parvenir à ce résultat.

1.2 Allocation mémoire

On considère maintenant un objet ressemblant aux `Vector` Java, utilisé pour implémenter une file d'attente.

Au niveau interne, c'est un tableau de k éléments qui supporte les opérations

- Ajouter un élément en fin du tableau
- Retirer un élément en début du tableau

La structure maintient :

- Le tableau T lui-même
- k , sa taille
- d , l'indice de la première case valide du tableau
- f , l'indice de la dernière case valide du tableau

Ainsi ajouter un élément x se fait par `T[++d]=x` et l'extraire par `return T[f++]`.

Cependant si $d = k$ il faut ré-allouer. Quand on ré-alloue le tableau, on recopie toute la zone écrite, ce qui coûte un temps proportionnel à $f - d$.

Exercice 2 Supposons que l'on ne fasse que des insertions. L'heuristique est en général de doubler la taille du tableau ($k \rightarrow 2k$). Il faut donc tout recopier après allocation ! En supposant que la taille initiale est $k = 10$, donner le nombre d'opérations de copie à faire pour insérer n éléments et en déduire le temps amorti.

Exercice 3 Maintenant on fait des insertions et des suppressions. Si le tableau devient trop vide, on le recopie vers un tableau plus petit pour économiser la mémoire. Discuter des performances (temps amorti) des deux algorithmes suivants :

1. Si le tableau est à moitié vide, on supprime la moitié vide
2. Si le tableau est aux 3/4 vide, on le recopie vers un tableau à moitié vide (et deux fois plus petit, donc) les cases vides étant placées en queue

1.3 Arbres binaires de recherche

On considère un arbre binaire de recherche (ABR) T et un pointeur N sur un nœud de T . On a les quatre procédures :

- Min : N pointe maintenant sur le nœud de plus petite valeur de T
- Next : N pointe sur le successeur du nœud courant. Renvoie une erreur si impossible
- Prev : idem, prédécesseur

Exercice 4 Rappeler où se trouvent le minimum d'un ABR et le successeur d'un nœud. Puis donner le code de Min (attention, N n'est pas à la racine!) et de Next

- Exercice 5**
1. Donner la complexité amortie d'une séquence de un Min suivi de $n - 1$ Next.
 2. Après un Min initial, maintenant on fait deux Next puis un Prev en alternance (Next,Next,Prev,Next,Next,Prev,... en tout $2n - 4$ Next et $n - 2$ Prev)
 3. n Next et n Prev dans le désordre (on suppose qu'on ne fait pas d'erreur sur Min ou Max)

2 Union-Find

On rappelle l'algorithme de Kruskal calculant un arbre couvrant minimum :

```
Kruskal(G : graphe) début
|  $F \leftarrow$  ensemble vide
| Trier les arêtes de  $E$  par ordre croissant de poids pour  $i$  de 1 à  $m$  faire
|   | si ajouter l'arête  $e_i = (u, v)$  à  $F$  ne crée pas de cycle alors
|   |   |  $F \leftarrow F \cup e_i$ 
|   |
| fin
```

On suppose que l'on dispose d'une structure de données stockant toutes les composantes connexes de A à un instant donné. Plus précisément :

- Une fonction booléenne **Find**(u, v) dit si deux sommets sont dans la même composante
- Une procédure **Union**(u, v) modifie la structure de données en remplaçant la composante contenant u et la composante contenant v par une nouvelle composante qui les contient tous les deux.

Exercice 6 Réécrire l'algorithme en faisant appel à **Union** et à **Find**

Exercice 7 Combien d'appels à **Union** et à **Find** faut-il faire ?

Exercice 8 Proposez une implémentation de la structure de données des composantes

Exercice 9 Déduisez-en la complexité d'un appel à **Union**, d'un appel à **Find** et de l'algorithme en général