TD Algo2 – session 2 – Files de priorités et ensembles disjoints

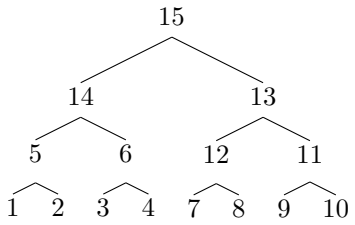
Objectifs d'apprentissage :
— étudier les structures de file de priorité et de forêt d'ensembles disjoints ;
— concevoir des algorithmes relatifs aux files de priorités et aux forêts d'ensembles disjoints ;
— mettre en pratique des techniques de preuve pour la validité ou la complexité d'algorithmes.

Les 6 premiers exercices sont essentiels. Plus d'exercices sur `https://algs4.cs.princeton.edu/24pq/index.php`.

Exercice 1 : Extraire-max-tas

Quel est le nombre minimum d'échanges lors d'un appel à l'opération `Extraire-Max-Tas` dans un tas de taille n sans clés dupliquées ? Fournir un tas de taille 15 pour lequel on obtient ce minimum. Même question si l'on appelle l'opération 2 fois, puis 3 fois de suite.

Le déplacement du dernier élément en première position n'est pas un échange. Il n'y a des échanges que lors de l'entassement. Exemple :



— 1 extraction : 1 échange.
— 2 extractions : $1 + 2$ échanges.
— 3 extractions : $1 + 2 + 2$ échanges.

Exercice 2 : Diminuer-clé-tas

Écrire le pseudo-code d'une procédure DIMINUER-CLÉ-TAS(A, i, k) dans un tas max. Quelle en est sa complexité en temps ?

Ça reprend le principe de l'entassement : c'est donc en $O(\log n)$.

DIMINUER-CLÉ-MAX(A, i, k)

si $k > A[i]$ **alors**
  erreur "nouvelle clé plus grande que clé actuelle"
$A[i] \leftarrow k$
ENTASSER-MAX(A, i)

Exercice 3 : Tri de listes triées

Donner un algorithme à temps $O(n \log k)$ qui fusionne k listes triées pour produire une liste triée unique, n étant ici le nombre total d'éléments toutes listes confondues. (Conseil : utiliser un tas min pour la fusion multiple.)

On note L la liste de listes triées, l la liste fusionnée et A le tas min utilisée.

```
TRI-LISTE-TRIÉES(L)
l ← {}
A ← file de priorités tas min
pour i = 1 jusqu'à k faire
  INSÉRER-TAS-MIN(A, (L[i][1], i, 1))
pour i = 1 jusqu'à n faire
  v, i_l, j ← EXTRAIRE-MIN-TAS(A)
  l[i] ← v
  si j < L[i_l].n alors
    INSÉRER-TAS-MIN(A, (L[i][j + 1], i_l, j + 1))
retourner l
```

Exercice 4 : Pile et file

Montrer comment implémenter une file FIFO (first-in first-out) avec une file de priorité. Montrer comment implémenter une pile avec une file de priorité.

Pour obtenir la même fonctionnalité qu'une file, on rajouterait les éléments avec une valeur croissante dans un tas min. Pareil dans un tas max pour une pile.

Exercice 5 : Pire cas

Donner une séquence de m opérations UNION et TROUVER-ENSEMBLE, contenant n opérations CRÉER-ENSEMBLE, dont le temps d'exécution est $O(m \log n)$ quand on se sert uniquement de l'union par rang.

Pour $n = 8$ et 7 UNION : $(0,1), (2,3), (4,5), (6,7), (0,2), (4,6), (0,4)$. Il suffit ensuite de faire m appels à TROUVER-ENSEMBLE pour 0 et 4.

Dans le cas général :

```
SEQUENCE-PIRE-CAS(n, m)
pour i = 0 jusqu'à n faire
  CRÉER-ENSEMBLE(i)
pour i = 1 jusqu'à ⌊log_2(n)⌋ faire
  pour j = 0 jusqu'à ⌊n/2^i − 1⌋ faire
    UNION(j2^i, j2^i + 2^(i−1))
pour i = 0 jusqu'à m faire
  TROUVER-ENSEMBLE(0, ⌊n/2⌋ − 1)
```

Exercice 6 : Compression de chemin

La compression de chemin modifie le parent dans une forêt d'ensembles disjoints pour que le parent pointer directement sur le représentant dès que c'est possible en $\Theta(1)$ opérations.

Il reste un cas où elle n'est pas mise en place dans l'algorithme. Compléter l'algorithme pour réaliser cette compression dans ce cas également.

```
UNION(x, y)
u ← TROUVER-ENSEMBLE(x)
v ← TROUVER-ENSEMBLE(y)
si u = v alors
  retourner
si u.rang > v.rang alors
  v.parent ← u
  y.parent ← u
sinon
  u.parent ← v
  x.parent ← v
  si u.rang = v.rang alors
    v.rang ← v.rang + 1
```

Exercice 7 : Échange

Chaque échange dans la boucle de AUGMENTER-CLÉ-TAS nécessite 3 affectations. Comment peut-on ajuster l'algorithme pour qu'il n'y ait plus qu'une seule affectation par itération de la boucle ?

```
AUGMENTER-CLÉ-TAS(A, i, k)
si k < A[i] alors
  erreur "nouvelle clé plus petite que clé actuelle"
A[i] ← k
tant que i > 1 et A[PARENT(i)] < k alors
  A[i] ← A[PARENT(i)]
  i ← PARENT(i)
A[i] ← k
```

Exercice 8 : Augmenter-clé-tas

Pour prouver la validité de AUGMENTER-CLÉ-TAS, on considère l'invariant de boucle suivant : Au début de chaque itération de la boucle, le tableau $A[1 : A.n]$ satisfait la propriété de tas max, à une infraction potentielle près : $A[i]$ risque d'être plus grand que $A[\text{PARENT}(i)]$. Cet invariant n'est pas suffisant pour prouver la validité de l'algorithme. Comment faudrait-il le préciser ?

Il faut préciser que $A[\text{PARENT}(i)].clé \geq A[\text{GAUCHE}(i)].clé$ et $A[\text{PARENT}(i)].clé \geq A[\text{DROITE}(i)].clé$. On a en effet besoin de cette propriété pour garantir dans l'étape de conservation que si on échange i et $\text{PARENT}(i)$, l'arbre enraciné en i respecte bien la propriété de tas max et donc l'invariant est respecté pour $\text{PARENT}(i)$.

Initialisation : le tableau respecte la propriété de tas max au début la procédure AUGMENTER-CLÉ-TAS et donc $A[\text{PARENT}(i)].clé \geq A[\text{GAUCHE}(i)].clé$ et $A[\text{PARENT}(i)].clé \geq A[\text{DROITE}(i)].clé$. Au début de la première itération, $A[i]$ vient de prendre une plus grande valeur et la seule violation ne peut être que $A[\text{PARENT}(i)] < A[i]$.

Conservation : en supposant que l'invariant est vrai pour i, on veut montrer qu'il sera aussi vrai pour la prochaine itération avec $\text{PARENT}(i)$, c'est-à-dire que la seule violation sera potentiellement que $A[\text{PARENT}(\text{PARENT}(i))] < A[\text{PARENT}(i)]$. D'autre part, il faut aussi montrer que $A[\text{PARENT}(\text{PARENT}(i))].clé \geq A[\text{GAUCHE}(\text{PARENT}(i))].clé$ et $A[\text{PARENT}(\text{PARENT}(i))].clé \geq A[\text{DROITE}(\text{PARENT}(i))].clé$. Pour le premier point, on sait que l'échange restaure la propriété de tas max en i car $A[\text{PARENT}(i)].clé \geq A[\text{GAUCHE}(i)].clé$ et $A[\text{PARENT}(i)].clé \geq A[\text{DROITE}(i)].clé$. Il ne peut donc plus y avoir de violation que en PARENT. Pour le second point, on suppose que i est le fils gauche de son parent (l'autre cas est symétrique) : $i = \text{GAUCHE}(\text{PARENT}(i))$. On a d'abord $A[\text{PARENT}(\text{PARENT}(i))].clé \geq A[\text{DROITE}(\text{PARENT}(i))].clé$ car ces éléments sont dans

une partie du tableau où la propriété de tas max est vérifié. On a aussi que, avant l'échange, $A[\textsc{Parent}(\textsc{Parent}(i))].clé \geq A[\textsc{Parent}(i)].clé$ et donc que $A[\textsc{Parent}(\textsc{Parent}(i))].clé \geq A[i].clé = A[\textsc{Gauche}(\textsc{Parent}(i))].clé$ après l'échange.
Terminaison : lorsque la boucle se termine, c'est que la propriété de tas max est vérifié en i ou que l'on a atteint la racine. Combiné à l'invariant, cela signifie que l'algorithme est valide.

Exercice 9 : Composantes connexes

Pendant l'exécution de `Composantes-Connexes` sur un graphe non orienté $G = (V, E)$ à k composantes connexes, combien de fois Union est-elle appelée ? Combien de fois exécute t-on la partie de Union qui modifie `parent` ? Les réponses seront exprimées en fonction de V, E et k.

— Union : E au total.
— Dernière partie d'Union : $V - k$.

Exercice 10 : Afficher ensemble

Considérons l'opération Afficher-Ensemble, qui reçoit un élément x et imprime tous les membres de l'ensemble de x, dans n'importe quel ordre. Montrer comment ajouter un seul attribut à chaque nœud dans une forêt d'ensembles disjoints de sorte que l'opération prenne un temps linéaire par rapport au nombre de membres de l'ensemble de x et que les temps d'exécution asymptotiques des autres opérations restent inchangés. On suppose que l'on peut imprimer chaque membre de l'ensemble en temps constant.

```
Créer-Ensemble(x)
x.parent ← x
x.voisin ← x
x.rang ← 0
```

```
Union(x, y)
u ← Trouver-Ensemble(x)
v ← Trouver-Ensemble(y)
u.voisin, v.voisin ← v.voisin, u.voisin
si u = v alors
  retourner
si u.rang > v.rang alors
  v.parent ← u
sinon
  u.parent ← v
  si u.rang = v.rang alors
    v.rang ← v.rang + 1
```

Tous les éléments d'un ensemble forment ainsi une liste chaînée et il suffit de les parcourir jusqu'à revenir au point de départ.