Structures de données (LU2IN006)

Cours 10 : Ensemble, partition, file de priorité

Nawal Benabbou

Licence Informatique - Sorbonne Université

2024-2025

Type de données abstrait : Ensemble

Type abstrait de données : ensemble

Ce type de données représente un ensemble fini d'éléments. Dans un ensemble, un élément ne peut apparaître qu'une seule fois. Pour détermner si un élément à ajouter serait un doublon, il faut une notion d'égalité entre les éléments (pas nécessairement une relation d'ordre).

Opérations classiques sur ce type de données :

- Vérifier(x, S) : vérifie la présence de l'élément x dans S.
- Insérer(x, S) : insère l'élément x dans l'ensemble S.
- Supprimer(x, S) : supprime l'élément x de l'ensemble S.
- Intersecter(S_1, S_2) : calcul l'intersection des ensembles S_1 et S_2.
- Unir(S_1, S_2) : calcul l'union des ensembles S_1 et S_2.

On notera n, n_1, et n_2 la cardinalité des ensembles S, S_1 et S_2 respectivement.

Une première implémentation possible : liste chaînée

Implémentation d'un ensemble avec une liste chaînée

On peut implémenter un ensemble à l'aide d'une liste chaînée, dont les éléments qui la compose correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S) est en

Une première implémentation possible : liste chaînée

Implémentation d'un ensemble avec une liste chaînée

On peut implémenter un ensemble à l'aide d'une liste chaînée, dont les éléments qui la compose correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S) est en $O(n)$ car il faut parcourir toute la liste.
- Insérer(x, S) est en

Une première implémentation possible : liste chaînée

Implémentation d'un ensemble avec une liste chaînée

On peut implémenter un ensemble à l'aide d'une liste chaînée, dont les éléments qui la compose correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S) est en $O(n)$ car il faut parcourir toute la liste.
- Insérer(x, S) est en $O(1)$ en réalisant une insertion en tête $O(1)$ (ou en $O(n)$ si la fonction vérifie son existence avant).
- Supprimer(x, S) est en

Une première implémentation possible : liste chaînée

Implémentation d'un ensemble avec une liste chaînée

On peut implémenter un ensemble à l'aide d'une liste chaînée, dont les éléments qui la compose correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S) est en $O(n)$ car il faut parcourir toute la liste.
- Insérer(x, S) est en $O(1)$ en réalisant une insertion en tête $O(1)$ (ou en $O(n)$ si la fonction vérifie son existence avant).
- Supprimer(x, S) est en $O(n)$ car il faut rechercher l'élément dans la liste pour le supprimer.
- Intersecter(S_1, S_2) et Unir(S_1, S_2) sont en

Une première implémentation possible : liste chaînée

Implémentation d'un ensemble avec une liste chaînée

On peut implémenter un ensemble à l'aide d'une liste chaînée, dont les éléments qui la compose correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S) est en $O(n)$ car il faut parcourir toute la liste.
- Insérer(x, S) est en $O(1)$ en réalisant une insertion en tête $O(1)$ (ou en $O(n)$ si la fonction vérifie son existence avant).
- Supprimer(x, S) est en $O(n)$ car il faut rechercher l'élément dans la liste pour le supprimer.
- Intersecter(S_1, S_2) et Unir(S_1, S_2) sont en $O(n_1 \times n_2)$ car il faut comparer deux à deux tous les éléments des listes. Si par contre on peut ordonner les éléments, ces opérations sont en $O(\max\{n_1 log(n_1); n_2 log(n_2)\})$ car on peut commencer par construire un tableau trié pour chaque liste (en $O(n_i log(n_i))$, puis parcourir en même temps les deux tableaux pour créer les éléments qui seront mis dans l'ensemble union ou intersection.

Une deuxième implémentation possible : table de hachage

Implémentation d'un ensemble avec une table de hachage

On utilise un tableau de taille fixée (disons m) et une fonction de hachage h qui associe à chaque élément possible e la position $h(e)$ dans le tableau.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S), Insérer(x, S) et Supprimer(x, S) sont en

Une deuxième implémentation possible : table de hachage

Implémentation d'un ensemble avec une table de hachage

On utilise un tableau de taille fixée (disons m) et une fonction de hachage h qui associe à chaque élément possible e la position $h(e)$ dans le tableau.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S), Insérer(x, S) et Supprimer(x, S) sont en $O(1)$ en pratique (si pas trop de collisions). En effet, il suffit de se rendre à la case $h(x)$, puis de réaliser la vérification, l'ajout ou la suppression.
- Intersecter(S_1, S_2) et Unir(S_1, S_2) sont en

Une deuxième implémentation possible : table de hachage

Implémentation d'un ensemble avec une table de hachage

On utilise un tableau de taille fixée (disons m) et une fonction de hachage h qui associe à chaque élément possible e la position $h(e)$ dans le tableau.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S), Insérer(x, S) et Supprimer(x, S) sont en $O(1)$ en pratique (si pas trop de collisions). En effet, il suffit de se rendre à la case $h(x)$, puis de réaliser la vérification, l'ajout ou la suppression.
- Intersecter(S_1, S_2) et Unir(S_1, S_2) sont en $O(m)$ car elles demandent de comparer les tables de hachage case par case (pour savoir quels sont les éléments présents dans les tables).

$\Rightarrow$ Cette implémentation est la plus efficace en pratique pour réaliser l'ensemble de ces opérations. Cependant, quand les éléments peuvent être ordonnés (entiers, mots, etc.), et que l'on doit souvent les parcourir dans l'ordre (numérique, alphabétique, etc), cette implémentation n'est pas très efficace. La suivante est plus adaptée.

Une troisième implémentation possible : AVL

Implémentation d'un ensemble avec un AVL

On peut implémenter un ensemble à l'aide d'un AVL (ou équivalent), dont les noeuds correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S), Insérer(x, S) et Supprimer(x, S) sont en

Une troisième implémentation possible : AVL

Implémentation d'un ensemble avec un AVL

On peut implémenter un ensemble à l'aide d'un AVL (ou équivalent), dont les noeuds correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S), Insérer(x, S) et Supprimer(x, S) sont en $O(log(n))$ car la hauteur d'un AVL est en $O(log(n))$.
- Unir(S_1, S_2) et Intersecter(S_1, S_2) sont en

Une troisième implémentation possible : AVL

Implémentation d'un ensemble avec un AVL

On peut implémenter un ensemble à l'aide d'un AVL (ou équivalent), dont les noeuds correspondent aux éléments de l'ensemble.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Vérifier(x, S), Insérer(x, S) et Supprimer(x, S) sont en $O(log(n))$ car la hauteur d'un AVL est en $O(log(n))$.
- Unir(S_1, S_2) et Intersecter(S_1, S_2) sont en $O(n_1+n_2)$: on crée un tableau trié pour chaque AVL (en $O(n_i)$), puis on les parcourt en même temps pour créer le tableau trié contenant les éléments à mettre dans l'AVL de l'union ou de l'intersection (en $O(n_1+n_2)$). Cet AVL peut ensuite être construit par une procédure dichotomique en $O(n_1+n_2)$: la racine de l'AVL est l'élément du milieu du tableau, puis récursivement, on utilise la première moitié du tableau pour le sous-arbre gauche et la deuxième moitié du tableau pour le sous-arbre droit.
- Parcourir(S) est en $O(n)$ car il suffit de faire un parcours infixe de l'arbre pour afficher les éléments dans l'ordre.

Type de données abstrait : Partition

Type abstrait de données : partition

Ce type de données représente une partition d'un ensemble fini d'éléments X. Une partition de X est un ensemble de sous-ensembles de X, deux à deux disjoints, et dont l'union est égale à l'ensemble X. Chaque sous-ensemble de la partition est communément appelée une *classe d'équivalence*, et on choisit arbitrairement un de ses éléments pour la représenter. Opérations classiques sur ce type de données :

- CreerClasse(x) : crée une classe dont le seul membre et représentant est x.
- Union(x,y) : fusionne les classes d'équivalence des éléments x et y.
- Find(x) : retourne le représentant de la classe de x.

On notera n la cardinalité de l'ensemble X.

Exemple : Composantes connexes

Dans un graphe non orienté, une composante connexe est un sous-graphe induit maximal connexe. Cela correspond à un ensemble de sommets tels que :

- tous les sommets de l'ensemble sont accessibles les uns des autres.
- aucun sommet en dehors de l'ensemble n'est accessible depuis les sommets de cet ensemble.

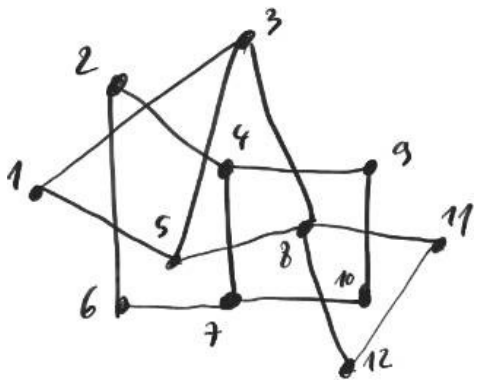
Les composantes connexes définissent une partition de l'ensemble des sommets.

Une première implémentation possible : table de hachage

Implémentation avec une table de hachage

On peut implémenter un partition à l'aide d'une table de hachage qui à tout élément associe un entier désignant le numéro de sa classe d'équivalence.

Exemple : Pour représenter les composantes connexes d'un graphe non orienté, on peut associer au sommet i la case i de la table, et elle contiendrait le numéro du sommet qui représente sa classe. Pour cet exemple, on pourrait donc utiliser la table suivante :



1	2	1	2	1	2	2	1	2	2	1	1

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

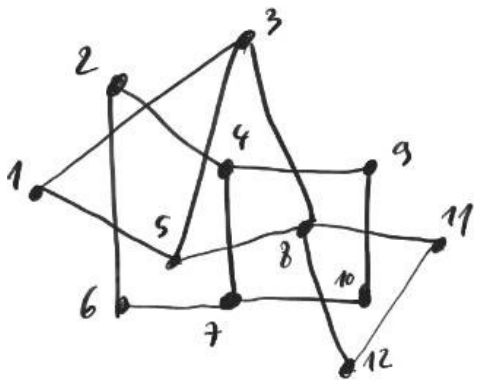
- Find(x) est en

Une première implémentation possible : table de hachage

Implémentation avec une table de hachage

On peut implémenter un partition à l'aide d'une table de hachage qui à tout élément associe un entier désignant le numéro de sa classe d'équivalence.

Exemple : Pour représenter les composantes connexes d'un graphe non orienté, on peut associer au sommet i la case i de la table, et elle contiendrait le numéro du sommet qui représente sa classe. Pour cet exemple, on pourrait donc utiliser la table suivante :



1	2	1	2	1	2	2	1	2	2	1	1

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

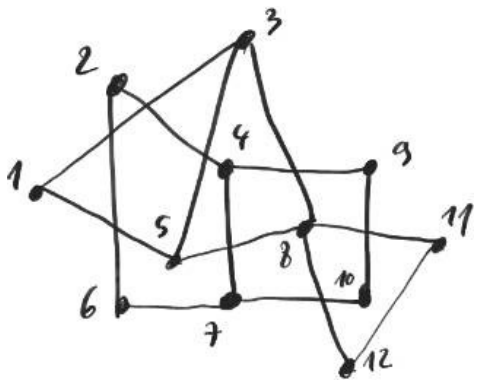
- Find(x) est en $O(1)$ car il suffit d'accéder à une case de la table.
- Union(x, y) est en

Une première implémentation possible : table de hachage

Implémentation avec une table de hachage

On peut implémenter un partition à l'aide d'une table de hachage qui à tout élément associe un entier désignant le numéro de sa classe d'équivalence.

Exemple : Pour représenter les composantes connexes d'un graphe non orienté, on peut associer au sommet i la case i de la table, et elle contiendrait le numéro du sommet qui représente sa classe. Pour cet exemple, on pourrait donc utiliser la table suivante :



1	2	1	2	1	2	2	1	2	2	1	1

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Find(x) est en $O(1)$ car il suffit d'accéder à une case de la table.
- Union(x,y) est en $O(n)$ car il faut parcourir toute la table pour trouver les éléments qui ont le même représentant que x, pour indiquer que leur représentant est maintenant celui de y.

Une deuxième implémentation : listes doublement chaînées

Implémentation avec une liste doublement chaînée par classe

On peut implémenter une partition à l'aide d'une liste doublement chaînée pour chaque classe, et on peut prendre la tête de la liste comme représentant. Dans ce cas, la complexité des opérations :

- Find(x) est en

Une deuxième implémentation : listes doublement chaînées

Implémentation avec une liste doublement chaînée par classe

On peut implémenter une partition à l'aide d'une liste doublement chaînée pour chaque classe, et on peut prendre la tête de la liste comme représentant. Dans ce cas, la complexité des opérations :

- Find(x) est en $O(n)$ car il faut remonter toute la liste pour connaître le représentant d'un élément.
- Union(x, y) est en

Une deuxième implémentation : listes doublement chaînées

Implémentation avec une liste doublement chaînée par classe

On peut implémenter une partition à l'aide d'une liste doublement chaînée pour chaque classe, et on peut prendre la tête de la liste comme représentant. Dans ce cas, la complexité des opérations :

- Find(x) est en $O(n)$ car il faut remonter toute la liste pour connaître le représentant d'un élément.
- Union(x,y) est en $O(1)$ car il suffit de concaténer les deux listes en mettant l'une à la fin de l'autre (mise à jour de deux pointeurs).

Remarques : On peut améliorer la complexité de Find en ajoutant à chaque élément un pointeur "représentant" qui pointe vers la tête de liste, ce qui conduit à une complexité en $O(1)$. Cependant, cela pénalise Union car, après avoir concaténer deux listes, il faut maintenant changer le pointeur représentant pour tous les éléments de la liste qui a été mise à la fin (complexité en $O(n)$). Pour améliorer cela, on pourrait en plus utiliser l'heuristique d'union pondérée, qui consiste à concaténer deux listes en mettant toujours la plus courte à la fin de la plus longue. Dans ce cas, on peut montrer que dans le pire des cas, le coût de k opérations successives (créer, union et/ou find) est en $O(k+nlogn)$.

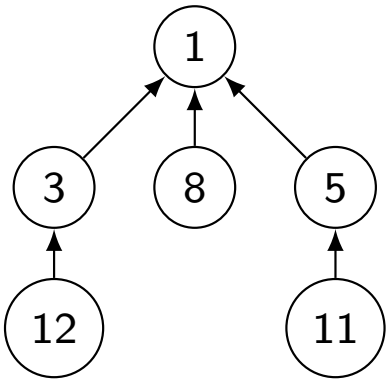
Une troisième implémentation : forêt

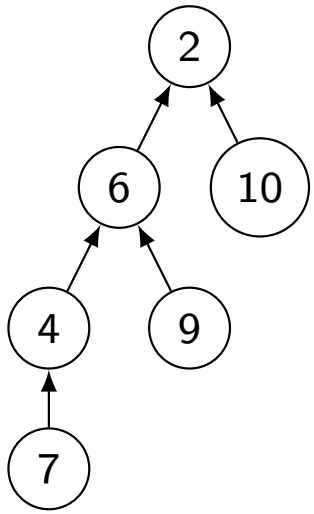
Pour représenter optimalement une partition, il faut utiliser une implémentation par forêt (c'est-à-dire un ensemble d'arbres).

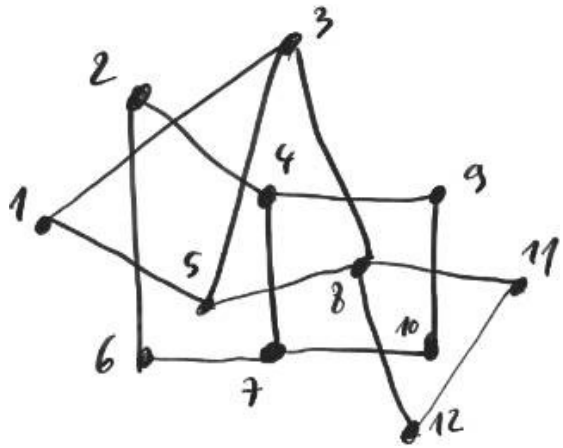
Implémentation avec une forêt

On peut représenter une partition avec une forêt, où chaque arbre correspond à une classe d'équivalence. Le représentant de chaque classe est la racine de l'arbre, et chaque noeud de l'arbre contient une référence vers son père.

Exemple : La forêt suivante permet de représenter les composantes connexes de notre graphe exemple.







Implémentation naïve des opérations avec une forêt

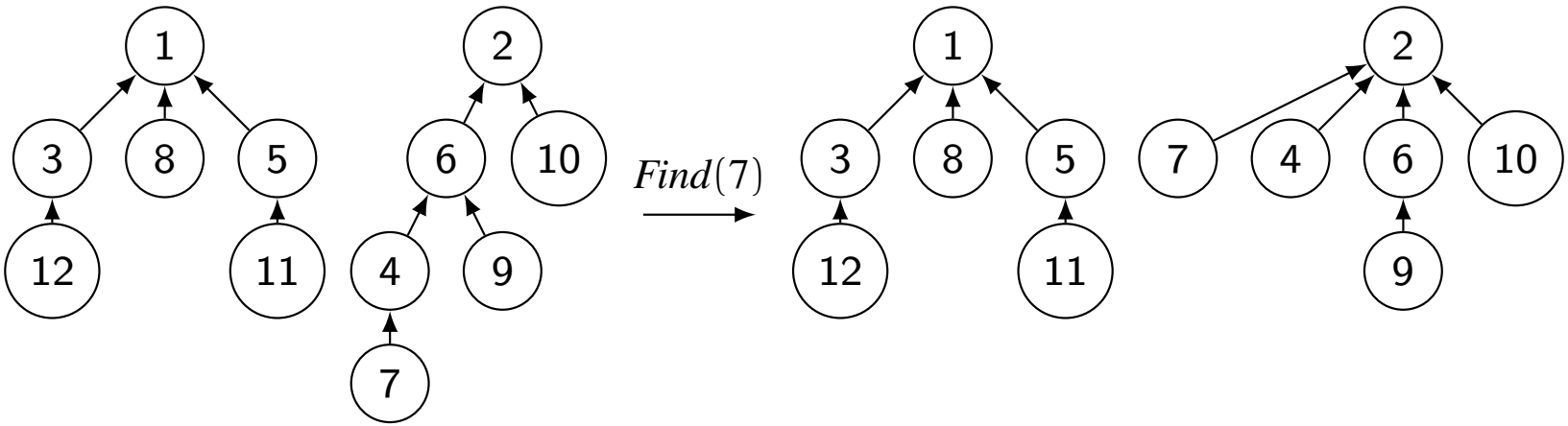
```
1  typedef struct noeud{
2      int elem;
3      struct noeud* pere;
4  } Noeud;
5
6  Noeud* creerClasse(int elem){
7      Noeud* n = (Noeud*)malloc(sizeof(Noeud));
8      n->elem = elem;
9      n->pere= n;
10     return n;
11 }
12 Noeud* find(Noeud* n){
13     if (n->pere != n)
14         return find(n->pere);
15     else
16         return n;
17 }
18 void union(Noeud* n1, Noeud* n2){
19     if (find(n1) != find(n2))
20         n1->pere=n2;
21 }
```

Remarque : Cette implémentation n'améliore pas la complexité des opérations par rapport à l'implémentation avec des listes doublement chaînées, car les arbres obtenus peuvent être très déséquilibrés.

Union-Find : une implémentation optimale avec une forêt

Pour améliorer ces opérations, il faut utiliser les deux heuristiques suivantes :

- *Find avec compression de chemin :* l'idée est de profiter de chaque appel à Find pour aplatir l'arbre. Plus précisément, tous les nœuds traversés quand on remonte jusqu'à la racine doivent être modifiés en chemin pour que leur père devienne la racine à la fin de l'appel.



- *Union par rang :* On stocke dans chaque nœud un majorant de la hauteur de son sous-arbre que l'on appelle rang (le rang d'un arbre contenant un seul noeud vaut 1). Lorsque l'on réalise une union, on décide que c'est la racine de plus grand rang qui devient le père de la racine de plus petit rang. Si les deux racines ont le même rang, on fait un choix arbitraire, et la racine qui devient père de l'autre doit augmenter son rang de 1.

Union-Find : une implémentation optimale avec une forêt

```
1  typedef struct noeud{
2      int elem;
3      int rang;
4      struct noeud* pere;
5  } Noeud;
6
7  Noeud* creerClasse(int elem){
8      Noeud* n = (Noeud*)malloc(sizeof(Noeud));
9      n->elem = elem;
10     n->pere= n;
11     n->rang= 1;
12     return n;
13 }
14
15 Noeud* find(Noeud* n){
16     if (n->pere != n)
17         n->pere = find(n->pere);
18     return n->pere;
19 }
```

Union-Find : une implémentation optimale avec une forêt

```
1  void union(Noeud* n1, Noeud* n2){
2      Noeud* racine1 = find(n1);
3      Noeud* racine2 = find(n2);
4      if (racine1 != racine2){
5          if (racine1->rang < racine2->rang){
6              racine1->pere = racine2;
7          }else{
8              racine2->pere = racine1;
9          }
10         if (racine1->rang == racine2->rang){
11             racine1->rang = racine1->rang+1;
12         }
13     }
14 }
```

Théorème

L'implémentation par forêt avec les deux heuristiques correspond à la structure de données appelée Union-Find. On peut montrer que dans le pire des cas, le coût de k opérations successives (créer, union et/ou find) est en $O(k\alpha(n))$ où $\alpha(n)$ est l'inverse de la fonction d'Ackermann (et $\alpha(n) \leq 5$ dans tous les cas pratiques). En conséquence, on peut considérer que le *coût amorti* par opération est une constante. Par ailleurs, il a été prouvé que ce n'est pas possible d'obtenir une meilleure complexité pour implémenter le type abstrait partition.

Type de données abstrait : file de priorité

Dans le deuxième cours, nous avons étudié les files basées sur le principe FIFO (*first in first out*). Dans une file de priorité, les premiers à être retirés de la file sont ceux les plus prioritaires.

Type abstrait de données : file de priorité

Ce type de données représente un ensemble de données homogènes, chacune associée à une valeur représentant son niveau de priorité. Une file de priorité permet l'extraction aisée de l'élément de priorité minimale (ou maximale), et l'ajout aisée de nouvelles données.
Opérations classiques sur ce type de données :

- Ajouter(x,p,F) : ajoute l'élément x de priorité p à la file de priorité F.
- ExtraireMin(F) : retire l'élément de priorité minimale.
- Fusionner(F_1, F_2) : fusionne les files de priorité F_1 et F_2.

On notera n, n_1, et n_2 le nombre d'éléments des files de priorité F, F_1 et F_2 respectivement.

Une première implémentation possible : tableau

Implémentation d'une file de priorité avec un tableau

On peut implémenter une liste de priorité par un tableau, où chaque case contient un élément de la file de priorité. Ce tableau peut éventuellement être trié par ordre de priorité.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Ajouter(x,p,F) est en

Une première implémentation possible : tableau

Implémentation d'une file de priorité avec un tableau

On peut implémenter une liste de priorité par un tableau, où chaque case contient un élément de la file de priorité. Ce tableau peut éventuellement être trié par ordre de priorité.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Ajouter(x,p,F) est en $O(1)$ si le tableau n'est pas trié. Si le tableau est trié, l'ajout est en $O(n)$ car il faut trouver la place du nouvel élément et décaler les autres éléments si besoin.
- ExtraireMin(F) est en

Une première implémentation possible : tableau

Implémentation d'une file de priorité avec un tableau

On peut implémenter une liste de priorité par un tableau, où chaque case contient un élément de la file de priorité. Ce tableau peut éventuellement être trié par ordre de priorité.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Ajouter(x,p,F) est en $O(1)$ si le tableau n'est pas trié. Si le tableau est trié, l'ajout est en $O(n)$ car il faut trouver la place du nouvel élément et décaler les autres éléments si besoin.
- ExtraireMin(F) est en $O(n)$ si le tableau n'est pas trié car il faut parcourir tout le tableau pour le trouver. L'opération est en $O(1)$ si le tableau est trié.
- Fusionner(F_1, F_2) est en

Une première implémentation possible : tableau

Implémentation d'une file de priorité avec un tableau

On peut implémenter une liste de priorité par un tableau, où chaque case contient un élément de la file de priorité. Ce tableau peut éventuellement être trié par ordre de priorité.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Ajouter(x,p,F) est en $O(1)$ si le tableau n'est pas trié. Si le tableau est trié, l'ajout est en $O(n)$ car il faut trouver la place du nouvel élément et décaler les autres éléments si besoin.
- ExtraireMin(F) est en $O(n)$ si le tableau n'est pas trié car il faut parcourir tout le tableau pour le trouver. L'opération est en $O(1)$ si le tableau est trié.
- Fusionner(F_1, F_2) est en $O(n_1 + n_2)$, car il faut parcourir les deux listes en même temps pour les intercaler efficacement.

Une deuxième implémentation possible : AVL

Implémentation d'une file de priorité avec un AVL

On peut implémenter une liste de priorité par un AVL où chaque noeud de l'arbre correspond à un élément de la file de priorité, et où les clés sont les niveaux de priorité des éléments.

Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Ajouter(x,p,F) est en $O(log(n))$ car la hauteur d'un AVL est $O(log(n))$.
- ExtraireMin(F) est en $O(log(n))$ car le min est la feuille la plus à gauche.
- Fusionner(F_1, F_2) est en $O(n_1 + n_2)$ (voir l'explication de l'opération Unir des ensembles).

Remarque : On pourrait utiliser un tas à la place d'un AVL mais on peut faire mieux pour l'opération de fusion avec des tas binomiaux.

Tas binomial

Un tas binomial est composé d'arbres binomiaux.

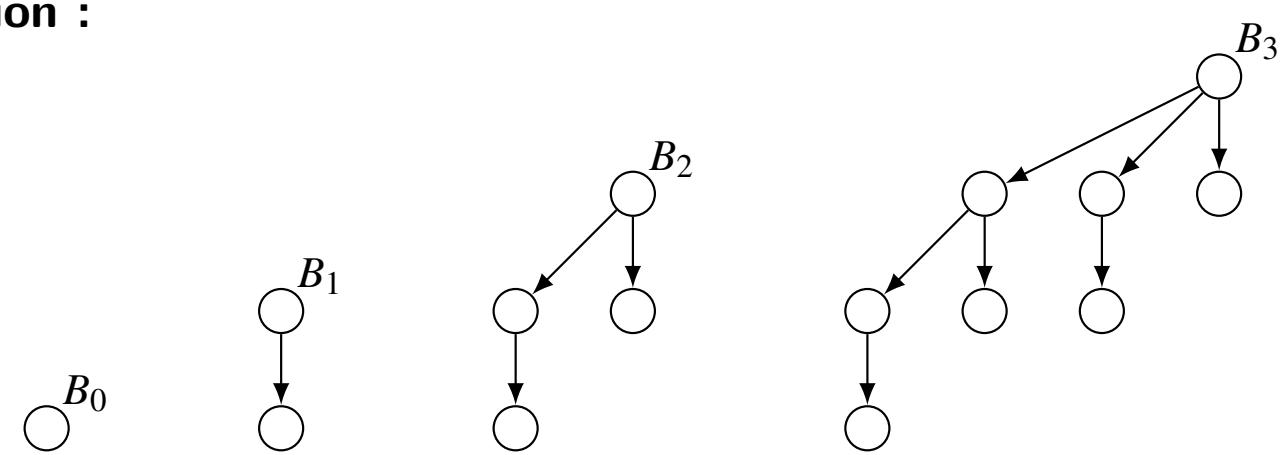
Définition : arbre binomial

Un arbre binomial d'ordre k, noté B_k, est défini récursivement de la manière suivante :

- B_0 est composé d'un seul noeud.
- B_k est un arbre dont la racine possède k fils, et ses fils sont les arbres binomiaux d'ordre $k-1, k-2, \ldots, 0$ dans cet ordre (de la gauche vers la droite).

Remarque : On peut aussi définir B_k à partir de deux arbres B_{k-1}, en plaçant l'un comme fils le plus à gauche de la racine de l'autre.

Illustration :



Tas binomial

Propriétés des arbres binomiaux

Un arbre binomial d'ordre k possède exactement 2^k noeuds, et sa hauteur est égale à k (la preuve par récurrence est triviale).

Tas binomial

Un tas binomial est une liste d'arbres binomiaux telle que :

- Dans chaque arbre binomial, la clé d'un noeud est toujours inférieure à celle de ses fils.
- Tous les arbres sont d'ordres différents.

Conséquences

- Le plus petit élément est une des racines des arbres binomiaux.
- Un tas binomial contenant n éléments possède au plus $\lfloor log(n)+1 \rfloor$ arbres binomiaux. Plus précisément, le nombre et les ordres de ses arbres est déterminé de manière unique par l'écriture binaire du nombre n. Par exemple, un tas binomial contenant $n = 11$ éléments sera composé d'un arbre d'ordre 3, 1 et 0 car 11 en binaire s'écrit 1011 $(11 = 2^3 + 2^1 + 2^0)$.

Fusion dans un tas binomial

Fusion

Pour fusionner deux tas binomiaux F_1 et F_2, on parcourt leur liste d'arbres binomiaux en même temps, en commençant par ceux de plus petits ordres. Si F_1 et F_2 possèdent un arbre d'ordre k, alors un arbre d'ordre $k+1$ est créé en fusionnant ces deux arbres de la manière suivante : la plus grande des deux racines devient le premier fils de la plus petite racine. Ce nouvel arbre fera partie du nouveau tas, sauf si F_1 et/ou F_2 possèdent un arbre d'ordre $k+1$, ce qui nécessiterait une autre fusion (car il doit y avoir au plus un seul arbre de chaque ordre). Ainsi, pour chaque ordre, on a au plus trois arbres binomiaux à traiter :

- S'il n'y en a qu'un, alors il est conservé pour faire partie du tas fusion.
- S'il y en a deux, ils sont fusionnés pour former un arbre d'ordre plus grand.
- S'il y en a trois, l'un d'eux est conservé pour le tas fusion, tandis que les deux autres sont fusionnés pour former un arbre d'ordre plus grand.

Complexité

L'opération de fusion est en $O(log(n))$ car l'ordre maximal d'un arbre binomial du tas fusion est $log(n)$.

Autres opérations dans un tas binomial

Insertion

Ajouter(x,p,F) est en

Autres opérations dans un tas binomial

Insertion

Ajouter(x,p,F) est en $O(log(n))$ car l'opération est effectuée en créant un tas contenant uniquement l'élément x de priorité/clé égale à p, puis en fusionnant ce tas avec F.

ExtraireMin

ExtraireMin(F) est en

Autres opérations dans un tas binomial

Insertion

Ajouter(x,p,F) est en $O(log(n))$ car l'opération est effectuée en créant un tas contenant uniquement l'élément x de priorité/clé égale à p, puis en fusionnant ce tas avec F.

ExtraireMin

ExtraireMin(F) est en $O(log(n))$ car il s'agit de trouver la plus petite racine parmi au plus $log(n)$ arbres binomiaux, la supprimer, créer un tas binomial avec la liste de ses fils, puis fusionner ce nouveau tas avec F.

Remarques :

- Les tas binomiaux permettent aussi de supprimer ou diminuer une clé quelconque du tas en $O(log(n))$.
- Il existe une variante des tas binomiaux (les tas de Fibonacci) qui permettent d'obtenir une meilleure complexité amortie.