Notes de cours d'algorithmique – L3
Tas et files de priorité

François Laroussinie
francois.laroussinie@univ-paris-diderot.fr
Page web du cours : https://www.irif.fr/~francoisl/l3algo.html

30 octobre 2020

Ici on définit des files de priorité où l'on extrait les éléments ayant la plus petite clé (en utilisant des ≪ tas min ≫). On peut de la même manière faire des structures où l'on extrait les éléments ayant une clé maximale (avec des ≪ tas max ≫).

1 Files de priorité vs tas.

Une file de priorité est un *type de données* pour stocker des éléments munis chacun d'une clé. On dispose des opérations classiques suivantes :

— Tester si la file est vide.

— Extraire de la file un élément ayant une clé de valeur minimale.

— Ajouter un élément (muni de sa clé) dans la file.

— Construire une file à partir d'un ensemble d'éléments et de leurs clés.

A ces opérations, nous allons ajouter une opération de mise à jour que l'on utilisera pour l'algorithme de Dijkstra et l'algorithme de Prim. Cette opération nous oblige d'étendre un peu la structure. Cette opération permet de diminuer la clé d'un élément présent dans la file. Nous allons donc présenter ce modèle en détail, mais nous commençons par présenter les principaux algorithmes sur des *tas* où ne sont stockées que les clés. En effet, pour réaliser (implémenter) une file de priorité, il est courant d'utiliser un tas, c'est-à-dire une structure de données arborescente qui permet d'avoir des opérations en complexité logarithmique pour l'insertion et l'extraction (et en temps constant pour le test du vide).
Dans un second temps (section 3), nous traiterons des files de priorité.

Définition 1 (Tas) *Un tas est un arbre binaire parfait où chaque noeud contient une clé inférieure à celles de ses sous-arbres.*

Un arbre binaire parfait est un arbre binaire où tous les niveaux sont remplis, sauf éventuellement le dernier où les feuilles sont alors regroupées à gauche. Voir la figure 1 pour deux exemples de tas contenant les clés $\{7, 5, 8, 8, 12, 12, 10, 15, 20, 2\}$. On voit donc que le même (multi)ensemble de clés peut avoir des tas différents (ils dépendent de l'ordre d'insertion des clés) mais la structure de l'arbre (le nombre de noeuds et leur positionnement) est fixée : elle dépend uniquement du nombre de valeurs à stocker. Notons aussi que la hauteur d'un arbre binaire parfait contenant n noeuds est $\lfloor \log_2(n) \rfloor$.

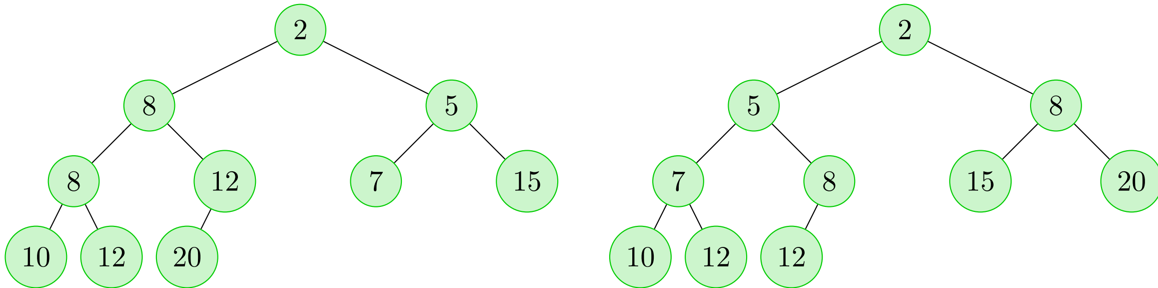

Figure 1: Exemples de tas

Implémentation d'un tas. On représente habituellement un tas par un tableau : la racine est stockée à la case d'indice 1, les noeuds du premier niveau sont dans les cases 2 et 3, ceux du second niveau sont en 4, 5, 6 et 7… Ce codage permet de « circuler » facilement dans l'arbre : étant donné un noeud i, on sait que son fils gauche est en $2 \cdot i$ et son fils droit en $2 \cdot i + 1$, et son père est en $\lfloor i/2 \rfloor$. Il permet aussi d'accéder facilement à la « prochaine feuille à ajouter » car elle sera à l'indice $n + 1$.
Le tas de gauche de la figure 1 peut donc être codé par le tableau suivant :

indices :	1	2	3	4	5	6	…							
clés :	2	8	5	8	12	7	15	10	12	20				

En plus, des valeurs (ici des clés) contenues dans T, il faut aussi disposer du nombre de clés stockées à un moment dans le tas, c'est-à-dire le nombre de cases utilisées (on le note T.Nb) et le nombre maximal de clés que peut contenir le tas (noté T.TailleMax).

2 Algorithmes sur les tas.

Le test du vide se fait directement en testant si T.Nb est égal à 0...

2.1 Insertion d'une clé.

Pour ajouter une clé c, on ajoute une feuille au tas (si le tableau n'est pas rempli !). Si c est suffisamment grande (c'-à-d. supérieure ou égale à la clé du père de la nouvelle feuille), c'est fini ! Sinon on place la clé du noeud père sur cette nouvelle feuille, et on remonte (en faisant descendre les clés au fur et à mesure) jusqu'à trouver un noeud dont le père a une clé inférieure (ou égale) à c : il suffit alors d'y placer la clé c.

```
Procédure Tas-Insertion(T, c)
begin
    si T.Nb == T.TailleMax alors
        return erreur
    T.Nb++
    i := T.Nb   //indice de la nouvelle feuille.
    tant que (i/2 ≥ 1) ∧ (T[i/2] > x) faire
        T[i] := T[i/2]
        i = i/2
    T[i] := c
end
```

Algorithme 1 : algorithme d'insertion

La complexité est donc en $O(\log_2(\mathsf{T.Nb}))$: dans le pire cas, on remontera jusqu'à la racine. La figure 2 montre l'insertion de la clé 6 pas à pas.

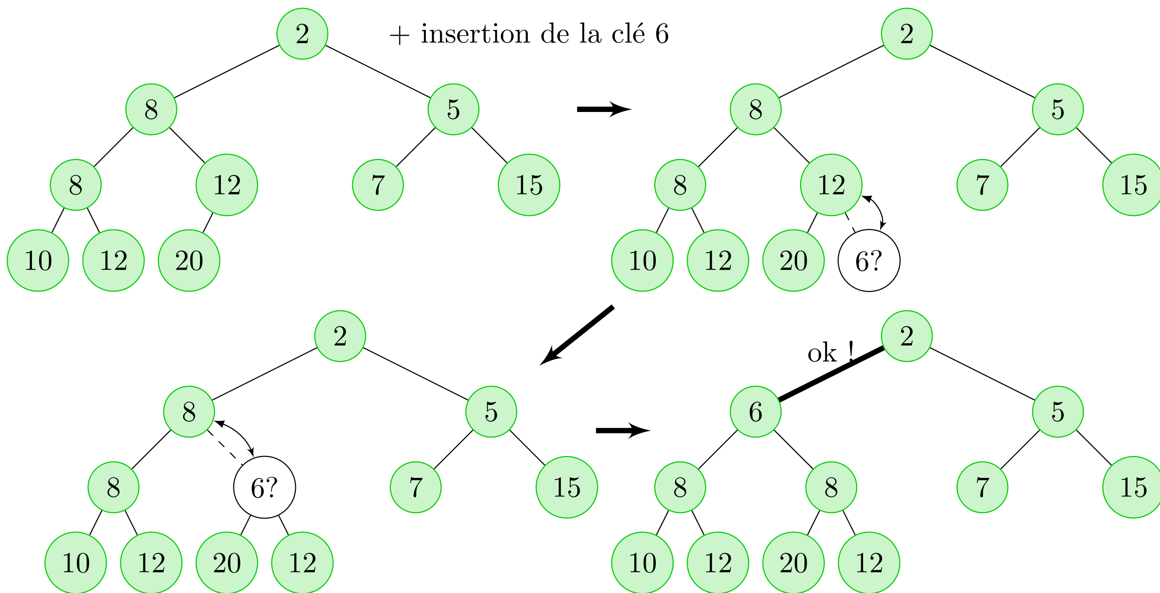

FIGURE 2: Insertion dans un tas

2.2 Extraction d'une clé.

Avec un tas on ne peut extraire que l'élément situé à la racine (c'est-à-dire une clé minimale). On stocke la clé de la racine dans une variable. On supprime une feuille et on place la clé qui s'y trouvait à la racine. Puis on fait un « tournoi » entre cette nouvelle racine et les deux clés des sous-arbres gauche et droit, le vainqueur sera placé à la racine, et si ce n'est pas la clé qui se trouvait à la racine qui a « gagné », celle-ci prendra la place du vainqueur et on continue ainsi dans le sous-arbre concerné… Cette opération va utiliser une procédure RemiseEnTas(T, i) qui organise donc un tournoi depuis le noeud i : si les sous-arbres de racine $2 \cdot i$ et $2 \cdot i + 1$ sont bien cohérents (pour les clés), la procédure RemiseEnTas(T, i) permet d'obtenir un tas correct. Cette fonction RemiseEnTas sera aussi utilisée pour la construction du tas.

```
Procédure Tas-ExtraireMin(T)
begin
    R := T[1]
    si T.Nb == 0 alors
        return erreur
    T[1] := T[T.Nb]
    T.Nb − −
    RemiseEnTas(T, 1)
    return R
end
```

Algorithme 2 : algorithme d'extraction

```
Procédure RemiseEnTas(T, i)
begin
    g := 2 · i
    d := 2 · i + 1
    win := i
    si g ≤ T.Nb ∧ T[g] < T[win] alors win := g
    si d ≤ T.Nb ∧ T[d] < T[win] alors win := d
    si win != i alors
        T[i] ↔ T[win]
        RemiseEnTas(T, win)
end
```

Algorithme 3 : algorithme de remise en tas depuis le noeud i

La complexité de RemiseEnTas est en $O(\log_2(\mathsf{T.Nb}))$. On a donc de même pour Tas-ExtraireMin. La figure 3 montre le processus d'extraction : la clé 12 est allée à la racine, puis elle a « glissé » à droite, puis a encore glissé à gauche (faisant remonté le 5 puis le 7).

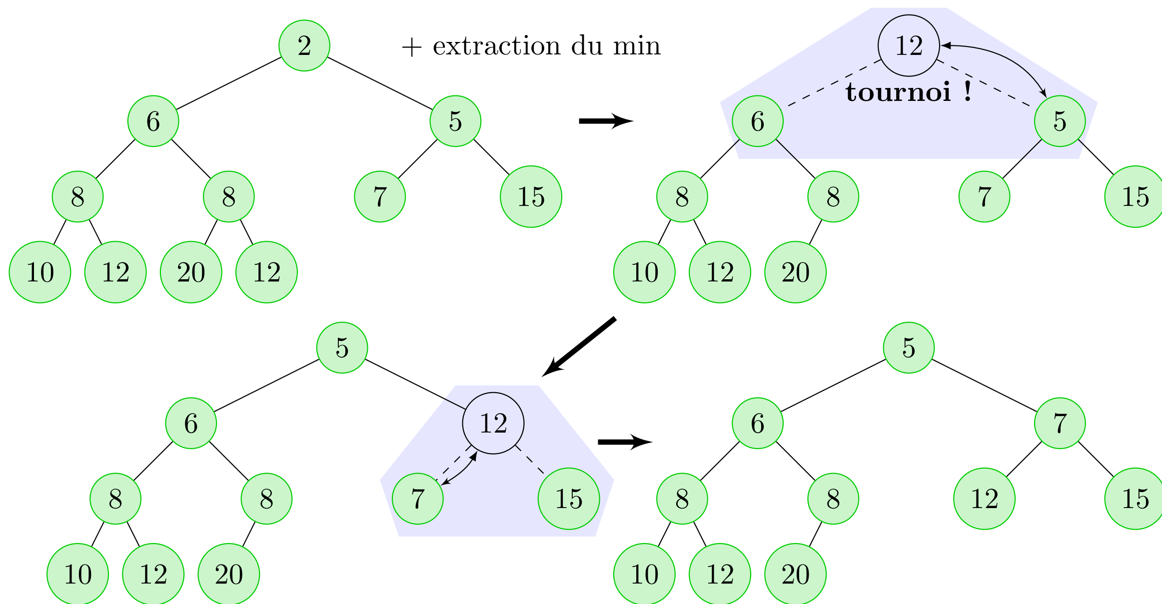

Figure 3: Extraction dans un tas

2.3 Construction efficace du tas.

Pour construire un tas à partir d'un ensemble de n clés, il est possible de procéder naivement en les insérant une à une à partir d'un tas vide. La complexité totale sera en $O(n \cdot \log_2 n)$. Une méthode plus efficace consiste à partir d'un tableau des clés (rangées de l'indice 1 à l'indice n dans un ordre quelconque) et de le transformer en un tas. Pour cela, on va utiliser la procédure RemiseEnTas. On part du constat que chacune des $\lceil \frac{n}{2} \rceil$ feuilles respecte la propriété des tas (que la clé d'un noeud soit inférieure ou égale aux clés contenues dans ses sous-arbres). Pour construire le tas, il suffit donc d'appeler RemiseEnTas sur les noeuds ayant deux (ou une) feuilles, donc sur l'indice $\lfloor \frac{\mathsf{T.Nb}}{2} \rfloor$, puis l'indice $\lfloor \frac{\mathsf{T.Nb}}{2} \rfloor - 1, \ldots$, puis d'appeler RemiseEnTas sur les noeuds supérieurs, *etc.* On en déduit donc l'algorithme 4 dont la complexité en $O(n)$, voir ci-dessous.

Procédure Tas-Créer(T)
begin
  pour $i = \lfloor \frac{\mathsf{T.Nb}}{2} \rfloor$ *à* 1 **faire**
    RemiseEnTas(T, i)
end

Algorithme 4 : algorithme de construction

La complexité de l'algorithme Tas-Créer s'évalue comme suit. On remarque d'abord que le coût de la procédure RemiseEnTas(T, i) est au pire de $2 \cdot h_i$ où h_i est la hauteur de l'arbre de racine i : on peut en effet faire des tournois (donc deux compariasons de clés) jusqu'en bas du sous-arbre de racine i. Dit autrement, ce coût est de $2(h - p_i)$ où h est la hauteur du tas global ($\log_2 n$) et p_i la profondeur du noeud i (c'est à dire $\lfloor \log_2 i \rfloor$). Soit $\mathcal{C}$ la complexité totale de la procédure Tas-Créer pour un tableau de taille n : $\mathcal{C}$ est donc la somme pour toutes les profondeurs $p = 0 \ldots h - 1$ (on ne s'intéresse pas aux feuilles), pour tous les noeuds de cette

profondeur (il y en a 2^p au maximum) de $2 \cdot (h-p)$. Et donc :

$$\mathcal{C} \leq \sum_{p=0}^{h-1} 2^p \cdot 2 \cdot (h-p) = 2^{h+1} \cdot \sum_{p=0}^{h-1} \frac{h-p}{2^{h-p}} = 2^{h+1} \cdot \sum_{i=1}^{h} \frac{i}{2^i} < 2^{h+1} \cdot \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}$$

D'où $\mathcal{C} \in O(n)$ car $2^h < n \leq 2^{h+1}$.

3 Files de priorité

3.1 Implémentation des files de priorités

Une file de priorité F pour stocker des éléments d'un ensemble X munis de clés dans $\mathbb{R}$ disposera d'un tableau F.T codant un tas, d'un champ F.Nb représentant le nombre de clés présentes dans la file et d'un champ F.TailleMax représentant la taille maximale du tableau [1]. On supposera que toute case du tableau stocke des paires $(x, c) \in X \times \mathbb{R}$. On utilisera aussi la notation F.T[i] $\leftrightarrow$ F.T[j] pour signifier l'échange des valeurs des deux cases (on permute alors les éléments et les clés). On utilise F.T[i].clé pour désigner la clé de l'élément situé au noeud i, et F.T[i].val pour l'élément.

Dans la suite, nous considérons des files de priorité <u>étendues</u> munies d'un tableau IndiceDansF qui associe à chaque élément de X l'indice de la case du tableau (le noeud) où il se situe (ou nil si il n'est pas présent dans F). On prendra comme indice de IndiceDansF les valeurs de X (on suppose qu'à tout moment, F ne contient qu'au plus une occurrence de x). Ce tableau IndiceDansF n'est utile que pour la fonction de mise à jour d'une clé dans la file utilisée pour les algorithmes de Dijkstra et Prim. Sa présence alourdit un peu le code des algorithmes précédents car il faut systématiquement mettre à jour le tableau IndiceDansF lorsqu'on modifie la place des éléments dans le tas (pour assurer l'invariant T[IndiceDansF[x]].val == x).

Notons qu'il existe d'autres structures de données (les « tas de Fibonacci ») qui sont plus efficaces (pour l'insertion et la mise à jour) mais elles sont bien plus élaborées, ici nous préférons donc utiliser des tas simples qui sont un bons compromis entre efficacité et simplicité de mise en oeuvre.

3.2 Algorithmes pour les files de priorité

Les algorithmes qui suivent reprennent exactement le même schéma que ceux des tas, mais en gérant les éléments à stocker en plus des clés et en mettant le tableau IndiceDansF à jour au fur et à mesure... On donne juste les algorithmes sans reexpliquer les idées sous-jacentes.

Pour la procédure de mise-à-jour de la clé d'un élément déjà stocké appelé ici FP-MaJ, on change légèrement la notation utilisée pour l'algorithme de Dijkstra ou Prim [2] mais l'essentiel est là. Attention ! Cette procédure suppose que la nouvelle clé (passée en argument) est inférieure à la clé actuelle.

1. on pourrait bien sûr utiliser des tableaux dynamiques et adapter le tableau aux besoins...
2. Le nom est changé et il n'y a pas ici de tableau d qui contient la valeur des clés dans les algorithmes de Dijkstra et Prim : on suppose comme dans le reste du document que les clés sont dans des champs clé des noeuds du tas.

```
Procédure FP-Insertion(F, x, c)
begin
    si F.Nb == F.TailleMax alors return erreur
    F.Nb++
    i := F.Nb
    tant que (i/2 ≥ 1) ∧ (F.T[i/2].clé > c) faire
        F.T[i] := F.T[i/2]
        IndiceDansF[F.T[i].val] := i
        i = i/2
    F.T[i].clé := c
    F.T[i].val := x
    IndiceDansF[x] := i
end
Procédure FP-ExtraireMin(F)
begin
    si F.Nb == 0 alors return erreur
    R := F.T[1].val
    F.T[1] := F.T[F.Nb]
    IndiceDansF[F.T[1].val] := 1
    F.Nb−−
    FP-RemiseEnTas(F, 1)
    return R
end
Procédure FP-RemiseEnTas(F, i)
begin
    g := 2 · i;  d := 2 · i + 1
    win := i
    si g ≤ F.Nb ∧ F.T[g].clé < F.T[win].clé alors win := g
    si d ≤ F.Nb ∧ F.T[d].clé < F.T[win].clé alors win := d
    si win! = i alors
        F.T[i] ↔ F.T[win]
        IndiceDansF[F.T[i].val] := i
        IndiceDansF[F.T[win].val] := win
        FP-RemiseEnTas(T, win)
end
Procédure FP-MaJ(F, x, c) //c est la nouvelle clé (inférieure) de x.
begin
    i := IndiceDansF[x]   //on récupère la position de x grâce au tableau.
    F.T[i].clé := c   //on met à jour la clé de x.
    tant que (i/2 ≥ 1) ∧ (F.T[i/2].clé > F.T[i].clé) faire
        F[i] ↔ F[i/2]
        IndiceDansF[F.T[i].val] := i
        IndiceDansF[F.T[i/2].val] := i/2
        i := i/2;
end
```

Algorithme 5 : algorithmes pour les files de priorité