Les arbres rouges et noirs et les tas

1. Introduction

Même si les arbres AVL et les B-arbres possèdent des propriétés intéressantes, ils n'en demeurent pas moins que des désavantages subsistent pour quelques applications. Par exemple, les arbres AVL peuvent nécessiter plusieurs rotations après une suppression; les B-arbres quant à eux peuvent nécessiter plusieurs opérations de regroupements ou d'éclatement après une opération insertion ou de suppression.

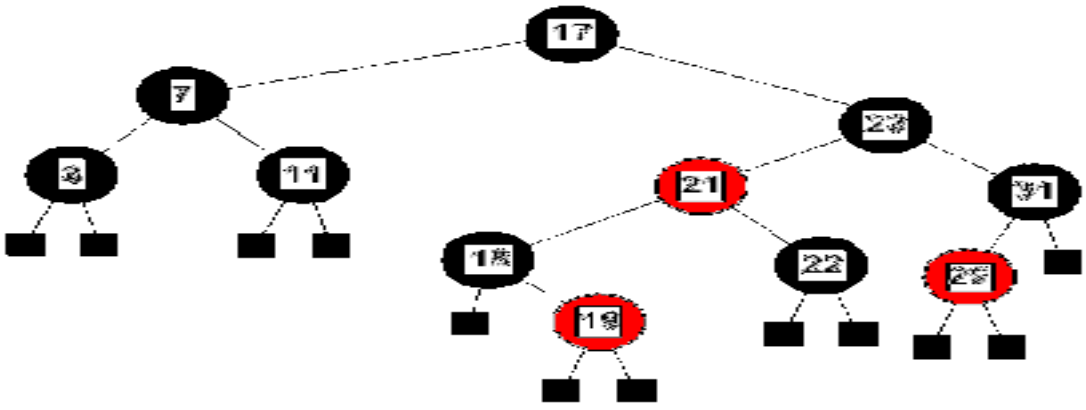
La structure de données d'arbre rouge et noir, discutée dans ce chapitre, ne possède pas ce désavantage car ne nécessitant qu'un temps de O(1) après une mise à jour pour conserver sa propriété d'arbre équilibré.

2. Définition

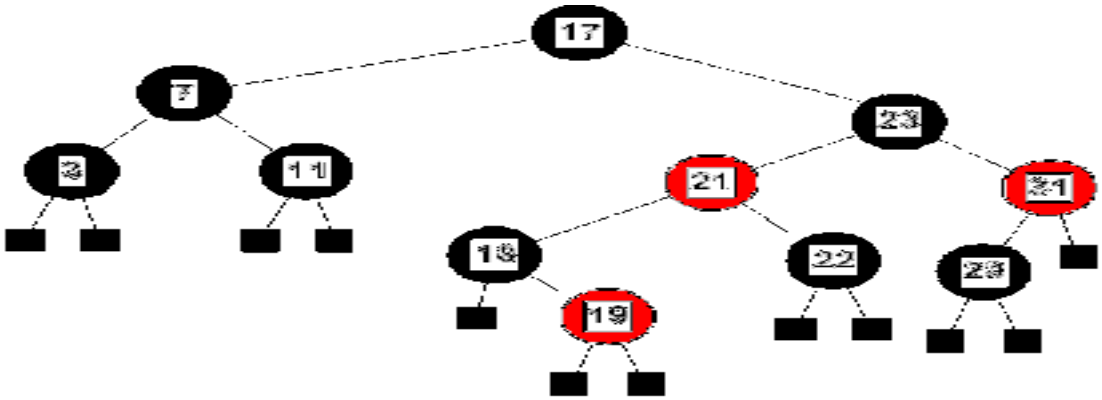
Un *arbre rouge et noir* est un arbre binaire de recherche où chaque nœud est de couleur rouge ou noire de telle sorte que

1. les feuilles (null) sont noires,
2. les enfants d'un nœud rouge sont noirs,
3. le nombre de nœuds noirs le long d'une branche de la racine à une feuille est indépendant de la branche. Autrement dit, pour tout noeud de l'arbre, les chemins de ce nœud vers les feuilles (nœud sentinel) qui en dépendent ont le même nombre de noeuds noirs.

Par commodité, on remplace les pointeurs null vers des sous-arbres vides par des feuilles sans clés. On les appelle nœuds externes. Les nœuds internes sont ceux qui ne sont pas externes.



Cet arbre est un arbre rouge et noir



Cet arbre n'est pas un arbre rouge et noir

Exercice : Montrer qu'un arbre ayant *n* nœuds internes possède n+1 noeuds externes.

La première condition est simplement technique et sans grande importance. La seconde condition stipule que les nœuds rouges ne sont pas trop nombreux. La dernière condition est une condition d'équilibre. Elle signifie que si on oublie les nœuds rouges d'un arbre on obtient un arbre binaire parfaitement équilibré.

Dans un arbre rouge et noir, on peut toujours supposer que la racine est noire. Si elle est rouge, on change sa couleur en noire et toutes les propriétés restent vérifiées.

Exercice: Montrer que l'on ne peut pas avoir deux nœuds rouges successifs le long d'un chemin d'un arbre rouge et noir (ARN).

En contrôlant cette information de couleur dans chaque nœud, on garantit qu'aucun chemin ne peut être deux fois plus long que n'importe quel autre chemin, de sorte que l'arbre reste équilibré. Même si l'équilibrage n'est pas parfait, on montre que toutes les opérations de recherche, d'insertion et de suppression sont en $O(\log n)$.

3. Hauteur d'un arbre rouge et noir

Les arbres rouges et noirs sont *relativement* bien équilibrés. La hauteur d'un arbre rouge et noir est de l'ordre de grandeur de log(n) où n est le nombre d'éléments dans l'arbre. En effet, la hauteur *h* d'un arbre rouge et noir ayant *n* éléments vérifie l'inégalité

$$h \leq 2\ln(n+1).$$

Pour un arbre rouge et noir, on appelle *hauteur noire* le nombre *hn(x)* de nœuds internes noirs le long d'une branche de la racine *x* à une feuille.

On montre par récurrence sur cette **hauteur noire**, qu'un arbre de hauteur noire égale à *hn(x)* possède au moins $2^{hn(x)}-1$ nœuds internes.

- Si la hauteur noire de *x* est 0, alors *x* doit être une feuille nulle. Le sous arbre enraciné en *x* contient $2^{hn(x)}-1=0$ nœud interne.

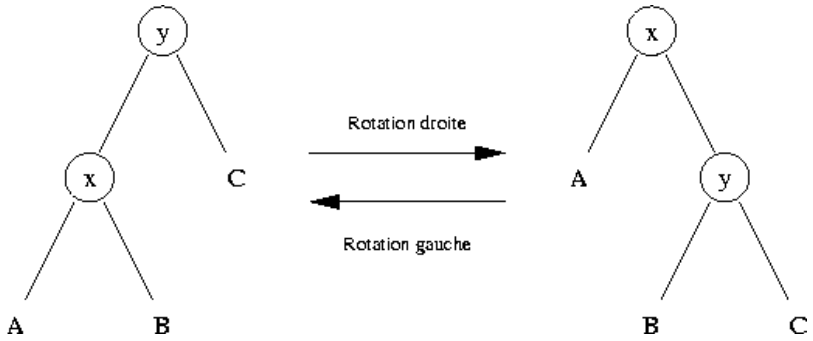
- Si la hauteur noire de *x* est >0, alors chacun de ses fils a une hauteur noire égale soit à *hn(x)* s'il est rouge, soit à *hn(x)-1* s'il est noir. Donc, en appliquant l'hypothèse d'induction aux deux sous arbres de *x*, le sous arbre enraciné en *x* contient au moins $2\times(2^{hn(x)-1}-1)+2=2^{hn(x)}$ nœuds internes.

Soit *h* la hauteur d'un arbre rouge-noir, la moitié au moins des nœuds vers une feuille doivent être noirs. Autrement dit, la hauteur noire d'un arbre rouge-noir est au moins *h/2*. Nous pouvons donc écrire :

$$n \geq 2^{hn(racine)}-1$$
$$\log_2(n+1) \geq hn(racine) \geq h/2$$
$$2\times\log_2(n+1) \geq h$$

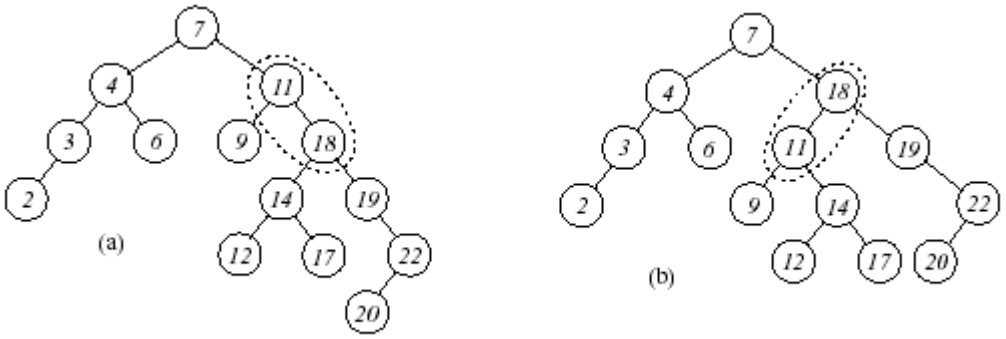
4. Rotations

Les rotations sont des modifications locales d'un arbre binaire. Elles consistent à échanger un nœud avec l'un de ses fils. Dans la rotation droite, un nœud devient le fils droit du nœud qui était son fils gauche. Dans la rotation gauche, un nœud devient le fils gauche du nœud qui était son fils droit. Les rotations gauche et droite sont inverses l'une de l'autre. Elle sont illustrées à la figure ci-dessous. Dans les figures, les lettres majuscules comme A, B et C désignent des sous-arbres.



L'opération de rotation est réalisable en temps $O(1)$.

La figure ci-dessous montre comment une rotation permet de rééquilibrer un arbre.



Rééquilibrage d'un arbre par une rotation. Une rotation gauche sur le nœud de clé 11 de l'arbre (a) conduit à un arbre (b) mieux équilibré: la hauteur de l'arbre est passée de 5 à 4.

5. Insertion d'une valeur

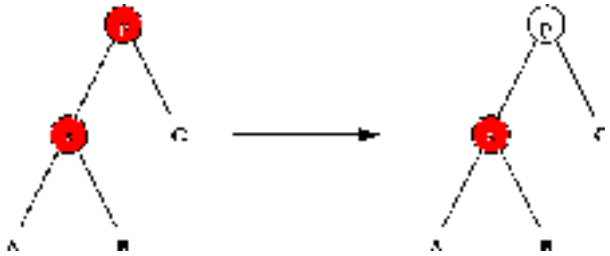
L'insertion d'une valeur dans un arbre rouge et noir commence par l'insertion usuelle d'une valeur dans un arbre binaire de recherche. Le **nouveau nœud** devient **rouge** de telle sorte que la propriété (3) reste vérifiée. Par contre, la propriété (2) n'est plus nécessairement vérifiée. Si le père du nouveau nœud est également rouge, la propriété (2) est violée.

Afin de rétablir la propriété (2), l'algorithme effectue des modifications dans l'arbre à l'aide de rotations. Ces modifications ont pour but de rééquilibrer l'arbre.

Soit x le nœud et p son père qui sont tous les deux rouges. L'algorithme distingue plusieurs cas.

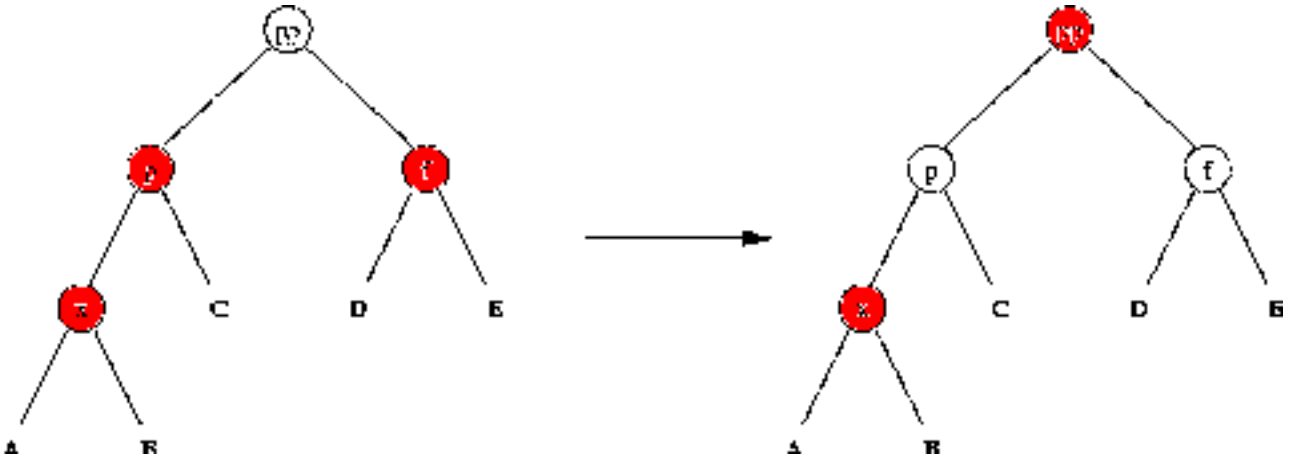
Cas 0 : le nœud père p est la racine de l'arbre

Le nœud père devient alors noir. La propriété (2) est maintenant vérifiée et la propriété (3) le reste. C'est le seul cas où la hauteur noire de l'arbre augmente.



Cas 1 : le frère f de p est rouge

Les nœuds p et f deviennent noirs et leur père pp devient rouge. La propriété (3) reste vérifiée mais la propriété ne l'est pas nécessairement. Si le père de pp est aussi rouge. Par contre, l'emplacement des deux nœuds rouges consécutifs s'est déplacé vers la racine.

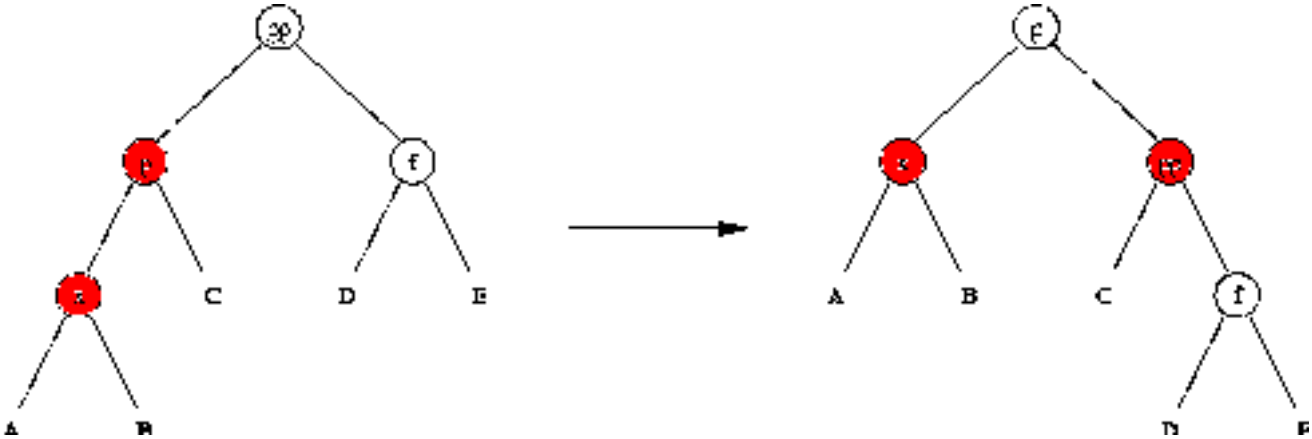


Cas 2 : le frère f de p est noir

Par symétrie on suppose que p est le fils gauche de son père. L'algorithme distingue à nouveau deux cas suivant que x est le fils gauche ou le fils droit de p.

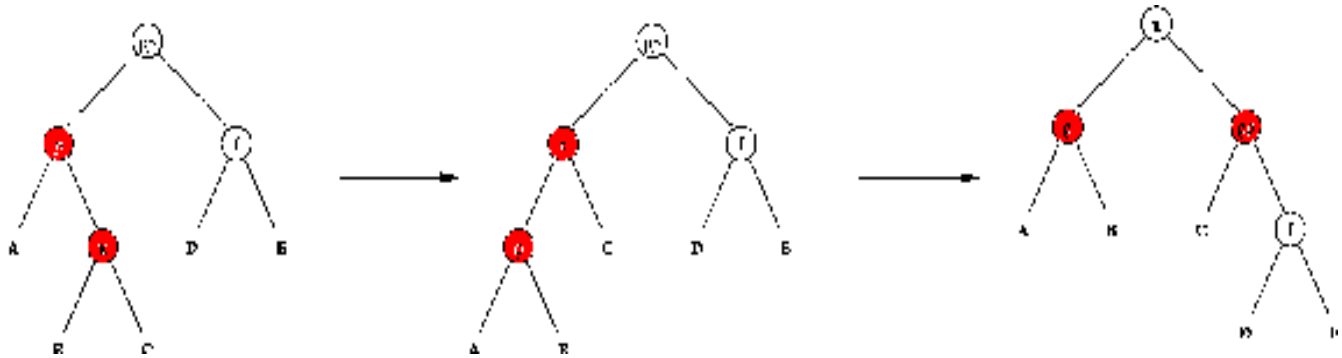
Cas 2a : x est le fils gauche de p.

L'algorithme effectue une rotation droite entre p et pp. Ensuite le nœud p devient noir et le nœud pp devient rouge. L'algorithme s'arrête alors puisque les propriétés (2) et (3) sont maintenant vérifiées.

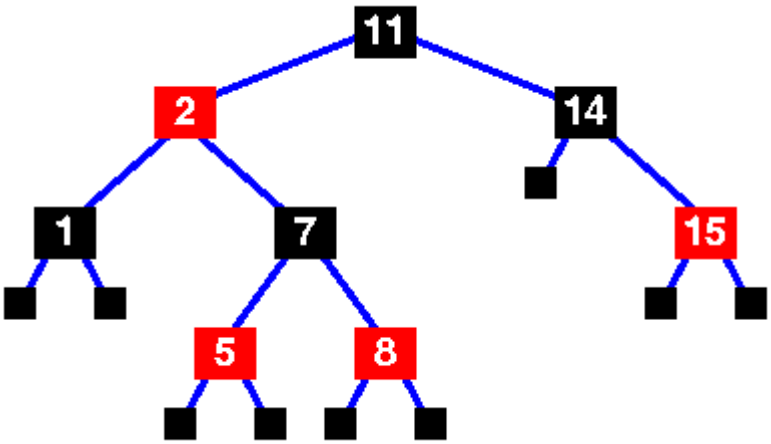


Cas 2b : **x est le fils droit de p.**

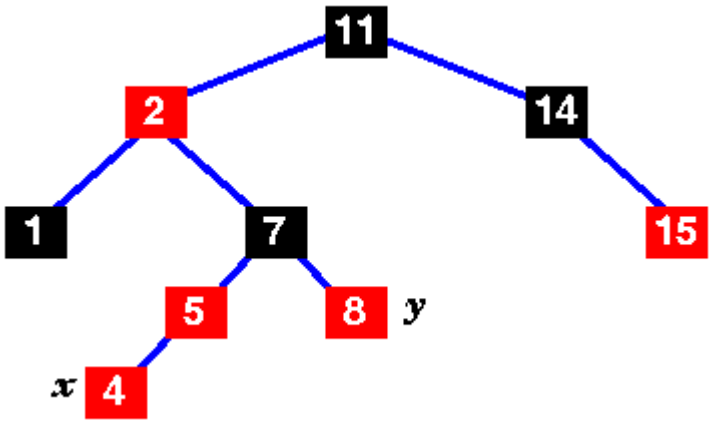
L'algorithme effectue une rotation gauche entre x et p de sorte que p devienne le fils gauche de x. On est ramené au cas précédent et l'algorithme effectue une rotation droite entre x et pp. Ensuite le nœud x devient noir et le nœud pp devient rouge. L'algorithme s'arrête alors puisque les propriétés (2) et (3) sont maintenant vérifiées.



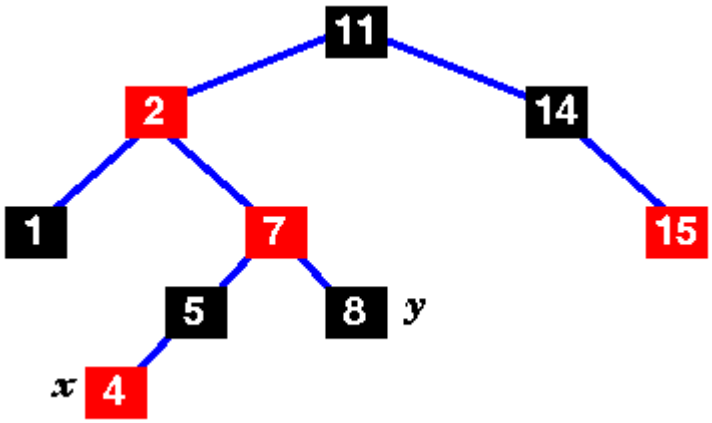
Exemple :



Arbre de départ

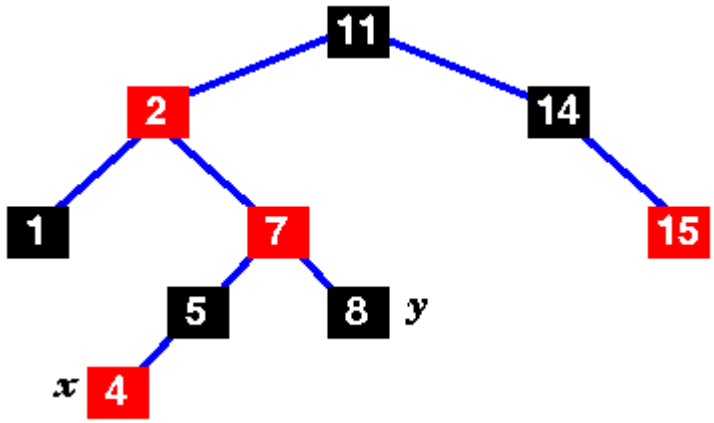


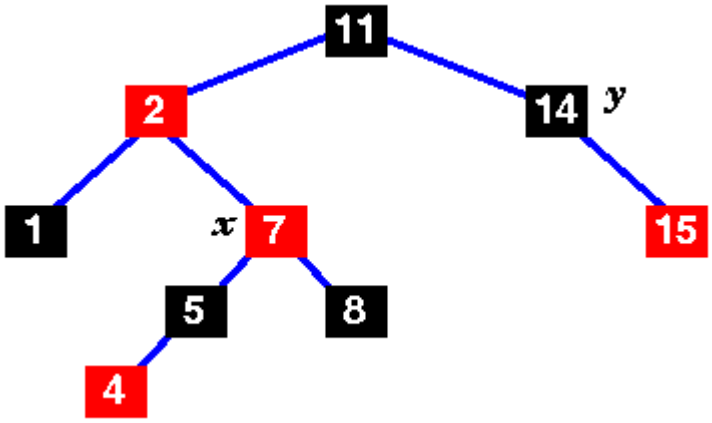
Ajout de 4



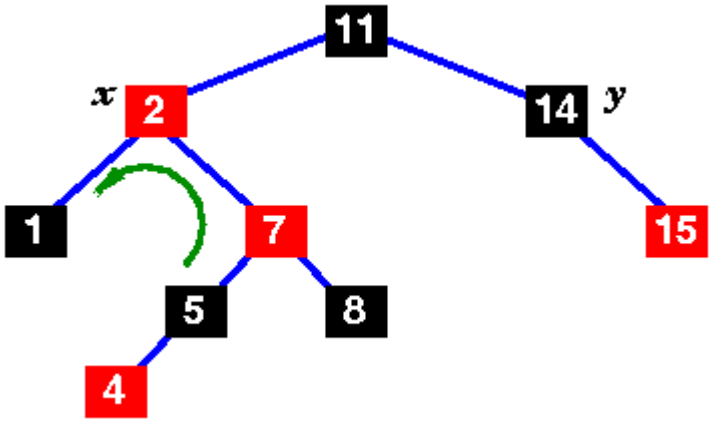
- Ce n’est plus un abre rouge et noir: Il y a deux noeuds rouges successifs sur le chemin : 11 - 2 - 7 - 5 - 4

- Changer de couleurs aux nœuds 5,7 et 8

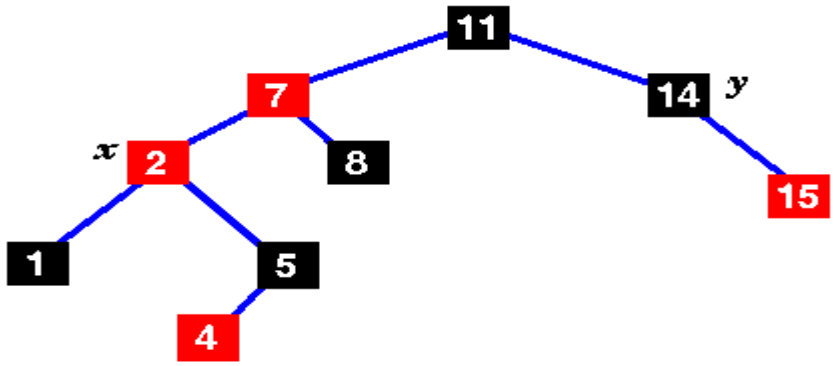




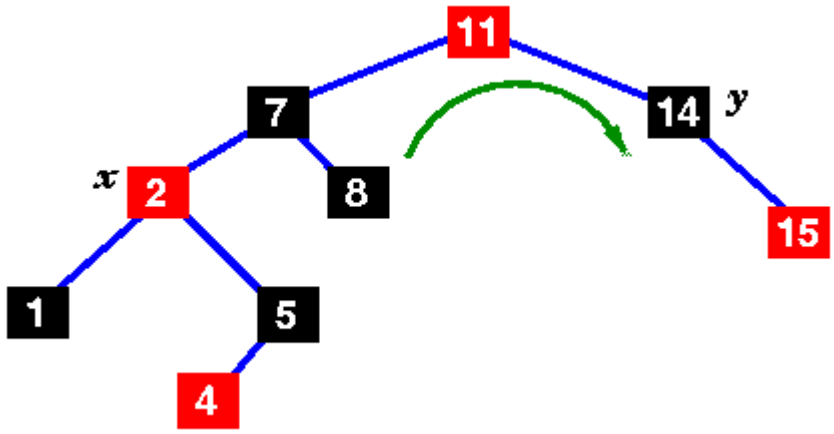
Prendre x jusqu'à son grand parent. Le parent de x est toujours rouges. Donc, l'arbre n'est pas rouget noir. Marquer l'oncle, y.



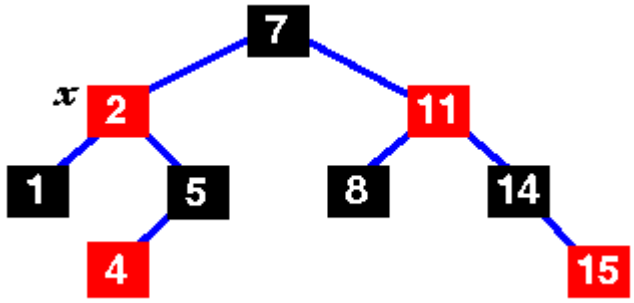
Prendre x en haut et faire une rotation gauche.



Pas encore rouge et noir



Changer de couleur à 7 et 11 et faire une rotation droite



Maintenant, l'arbre est rouge et noir

Exercice : Construire, dans cet ordre, un arbre rouge et noir, à partir des nœuds suivants A, L, G, O, R, I, T, H et M .

6. Suppression d'une valeur

Comme pour l'insertion d'une valeur, la suppression d'une valeur dans un arbre rouge et noir commence par supprimer un nœud comme dans un arbre binaire de recherche. Si le nœud qui porte la valeur à supprimer possède zéro ou un fils, c'est ce nœud qui est supprimé et son éventuel fils prend sa place. Si, au contraire, ce nœud possède deux fils, il n'est pas supprimé. La valeur qu'il porte est remplacée par la valeur suivante dans l'ordre et c'est le nœud qui portait cette valeur suivante qui est supprimé. Ce nœud supprimé est le nœud au bout de la branche gauche du sous-arbre droit du nœud qui portait la valeur à supprimer. Ce nœud n'a pas de fils gauche.

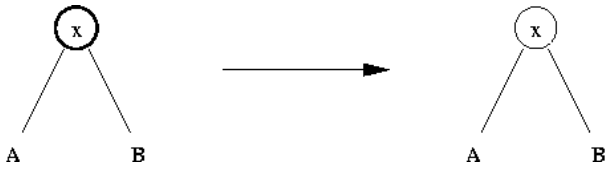
Si le nœud supprimé est rouge, la propriété (3) reste vérifiée. Si le nœud supprimé est noir, alors sa suppression va diminuer la hauteur noire de certains chemins dans l'arbre. La propriété est retrouvée en rectifiant les couleurs comme suit :

on considère que le nœud qui remplace le nœud supprimé porte une couleur noire en plus. Ceci signifie qu'il devient noir s'il est rouge et qu'il devient doublement noir s'il est déjà noir. La propriété (3) reste ainsi vérifiée mais il y a éventuellement un nœud qui est doublement noir.

Afin de supprimer ce nœud doublement noir, l'algorithme effectue des modifications dans l'arbre à l'aide de rotation. Soit x le nœud doublement noir.

Cas 0 : le nœud x est la racine de l'arbre

Le nœud x devient simplement noir. La propriété (2) est maintenant vérifiée et la propriété (3) le reste. C'est le seul cas où la hauteur noire de l'arbre diminue.

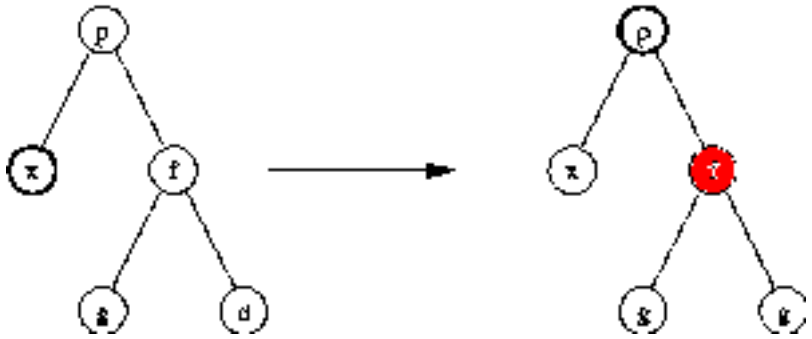


Cas 1 : le frère f de x est noir.

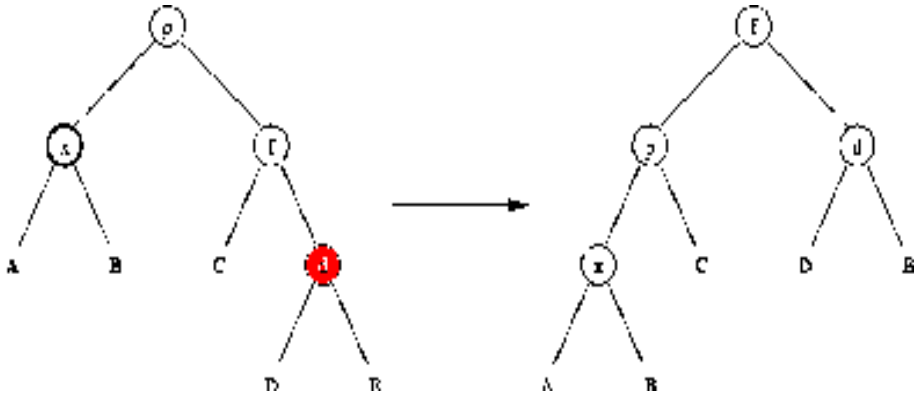
Par symétrie, on suppose que x est le fils gauche de son père p et que f est donc le fils droit de p. Soient g et d les fils gauche et droit de f. L'algorithme distingue à nouveau trois cas suivant les couleurs des nœuds g et d.

Cas 1a : les deux fils g et d de f sont noirs.

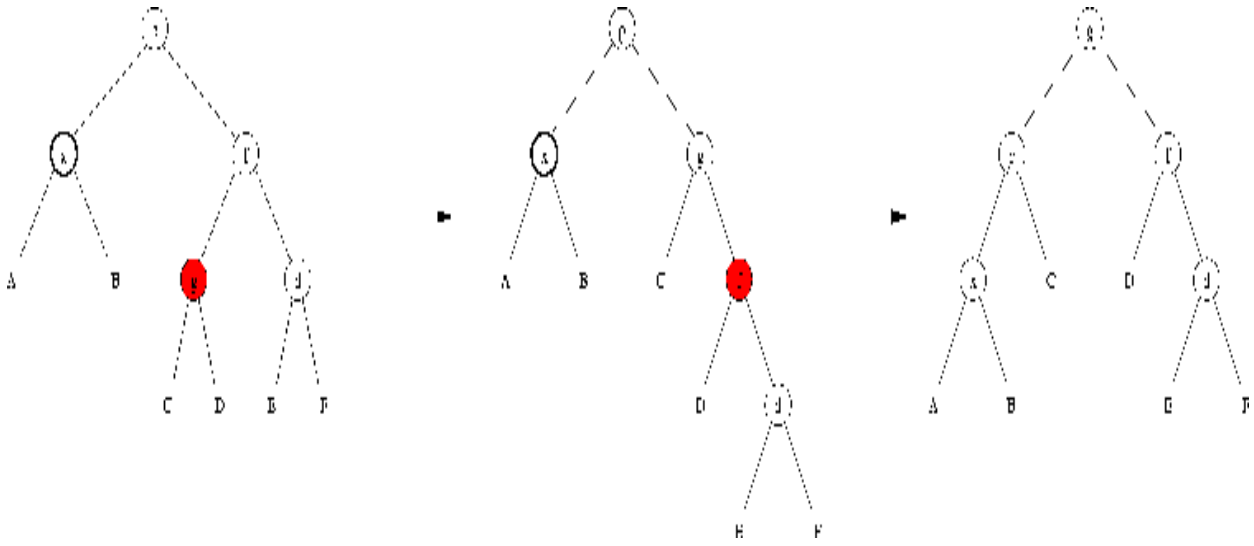
Le nœud x devient noir et le nœud f devient rouge. Le nœud p porte une couleur noire en plus. Il devient noir s'il est rouge et il devient doublement noir s'il est déjà noir. Dans ce dernier cas, il reste encore un nœud doublement noir mais il s'est déplacé vers la racine de l'arbre. C'est ce dernier cas qui représenté à la figure ci-dessous.



Cas 1b : le fils droit d de f est rouge.
L'algorithme effectue une rotation droite entre p et f. Le nœud f prend la couleur du nœud p. Les noeuds x, p et d deviennent noirs et l'algorithme se termine.

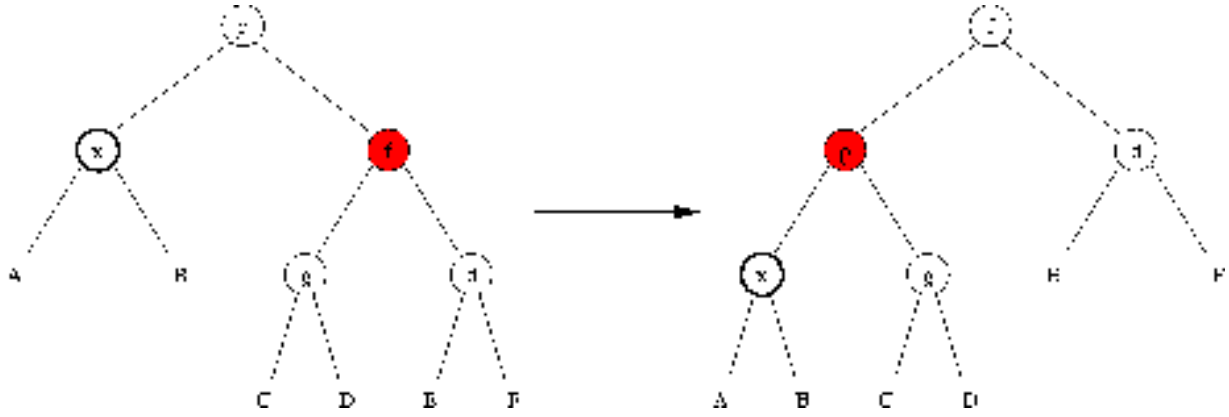


Cas 1c : le fils droit d est noir et le fils gauche g est rouge.
L'algorithme effectue une rotation gauche entre f et g. Le nœud g devient noir et le nœud f devient rouge. Il n'y a pas deux nœuds rouges consécutifs puisque la racine du sous-arbre D est noire. On est ramené au cas précédent puisque maintenant, le frère de x est g qui est noir et les deux fils de g sont noir et rouge. L'algorithme effectue alors une rotation entre p et g. Le nœud f redevient noir et l'algorithme se termine.

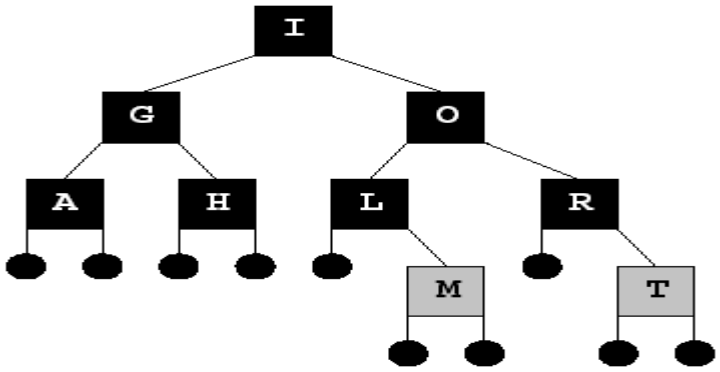


Cas 2 : le frère f de x est rouge.

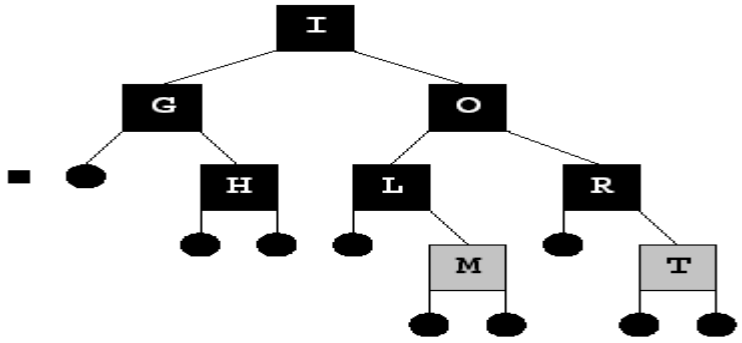
Par symétrie, on suppose que x est le fils gauche de son père p et que f est donc le fils droit de p. Puisque f est rouge, le père p de f ainsi que ses deux fils g et d sont noirs. L'algorithme effectue alors une rotation gauche entre p et f. Ensuite p devient rouge et f devient noir. Le nœud x reste doublement noir mais son frère est maintenant le nœud g qui est noir. On est donc ramené au cas 1.

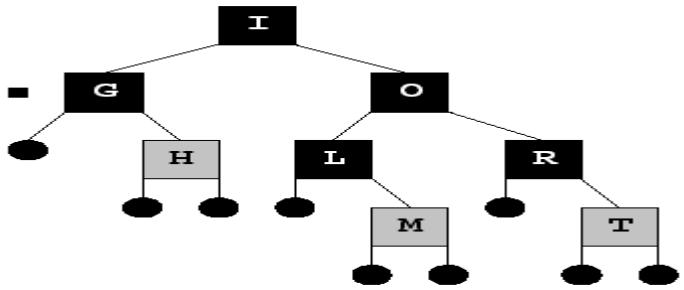


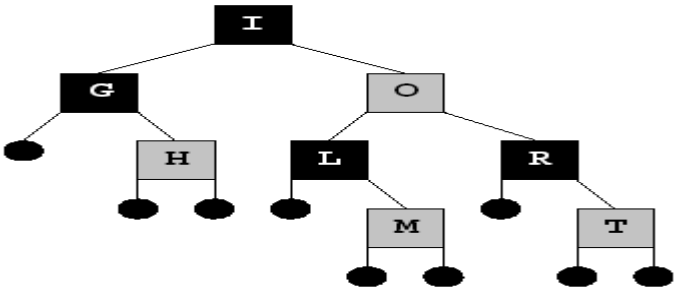
Exemple : supprimer les nœuds A, L, G, O, R, I, T H M de l'arbre rouge et noir suivant :



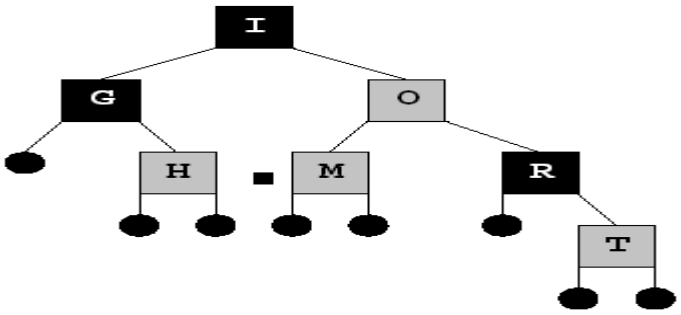
Supprimer A

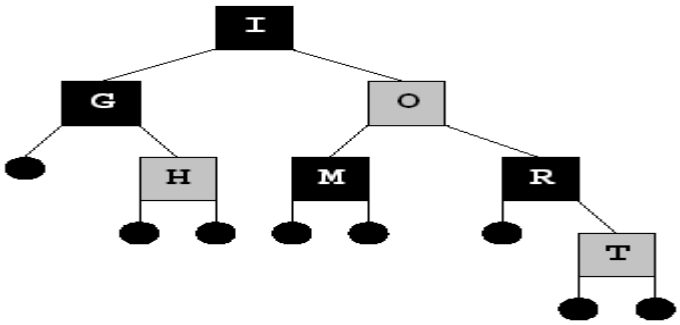




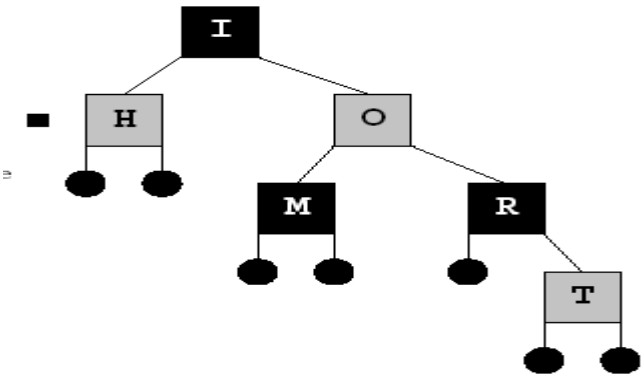


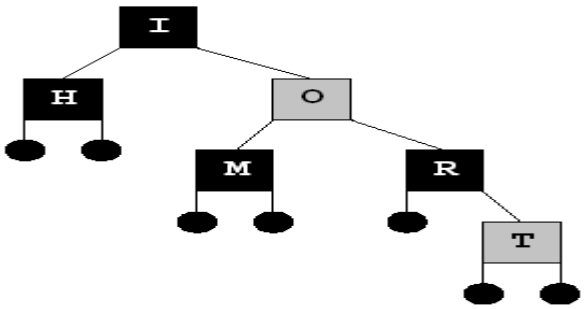
supprimer L



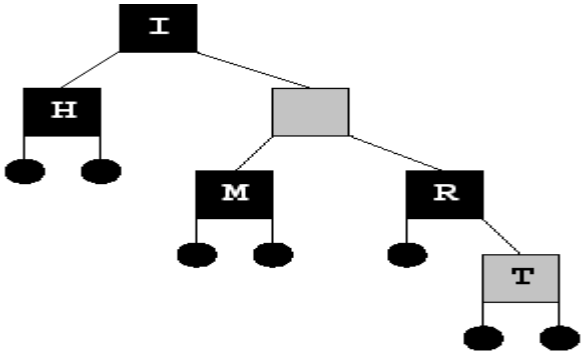


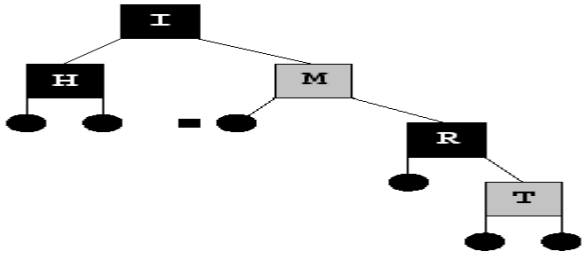
supprimer G

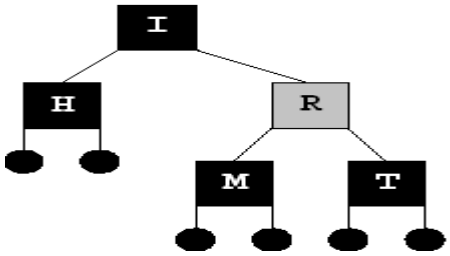




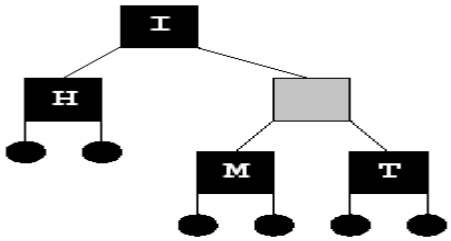
supprimer O

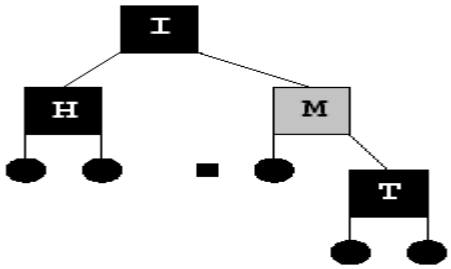


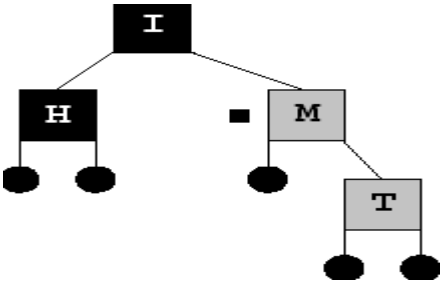




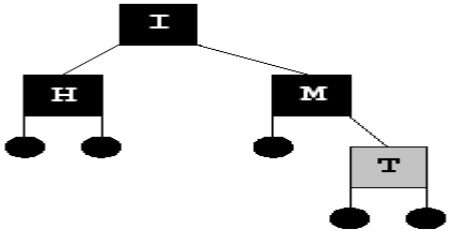
Supprimer R

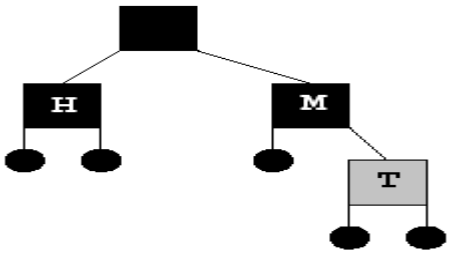


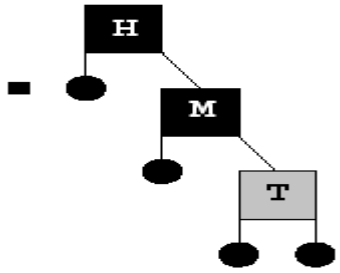


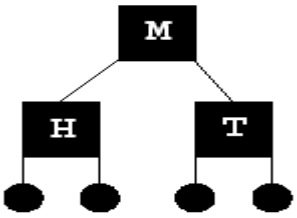


supprimer I

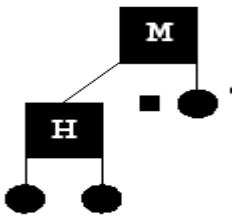


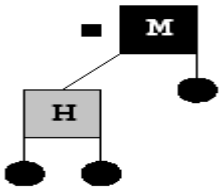




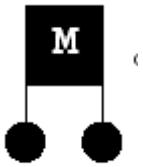


supprimer T





supprimer H



supprimer M

Exercice : À partir des explications présentées ci-dessus, écrire les fonctions d'insertion et de suppression dans un arbre rouge et noir.

Les files de priorité

Donnons d'abord une vision intuitive d'une file de priorité.

On suppose que des gens se présentent au guichet d'une banque avec un numéro écrit sur un bout de papier représentant leur degré de priorité. Plus ce nombre est élevé, plus ils sont importants et doivent passer rapidement. Bien sûr, il n'y a qu'un seul guichet ouvert, et l'employé(e) de la banque doit traiter rapidement tous ses clients pour que tout le monde garde le sourire. La file des personnes en attente s'appelle une *file de priorité*. L'employé de banque doit donc savoir faire rapidement les 3 opérations suivantes: trouver un maximum dans la file de priorité, retirer cet

élément de la file, savoir ajouter un nouvel élément à la file. Plusieurs solutions sont envisageables.

La première consiste à mettre la file dans un tableau et à trier la file de priorité dans l'ordre croissant des priorités. Trouver un maximum et le retirer de la file est alors simple: il suffit de prendre l'élément de droite, et de déplacer vers la gauche la borne droite de la file. Mais l'insertion consiste à faire une passe du tri par insertion pour mettre le nouvel élément à sa place, ce qui peut prendre un temps $O(n)$ où n est la longueur de la file.

Une autre méthode consiste à gérer la file comme une simple file et à rechercher le maximum à chaque fois. L'insertion est rapide, mais la recherche du maximum peut prendre un temps $O(n)$, de même que la suppression.

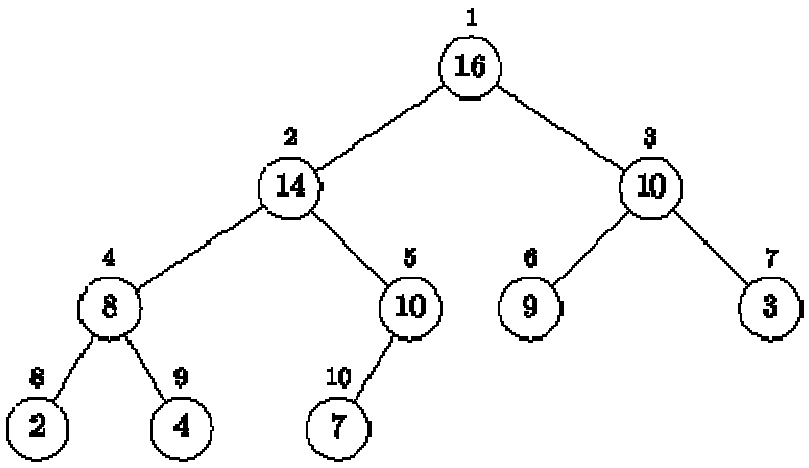
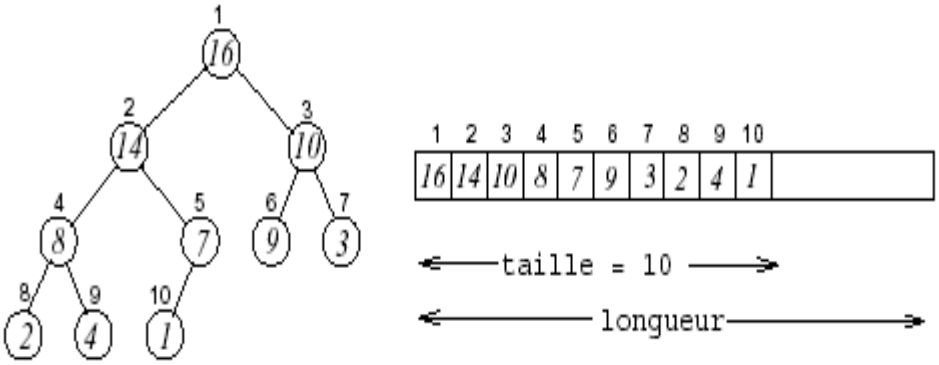


Figure 4.3 : Représentation en arbre d'un tas

Une méthode élégante consiste à gérer une structure d'ordre partiel grâce à un arbre. La file de n éléments est représentée par un arbre binaire contenant en chaque noeud un élément de la file (comme illustré dans la figure 4.3). L'arbre vérifie deux propriétés importantes: d'une part la valeur de chaque noeud est supérieure ou égale à celle de ses enfants, d'autre part l'arbre est quasi complet, propriété que nous allons décrire brièvement. Si l'on divise l'ensemble des noeuds en lignes suivant leur hauteur, on obtient en général dans un arbre binaire une ligne 0 composée simplement de la racine, puis une ligne 1 contenant au plus deux noeuds, et ainsi de suite (la ligne i contenant au plus 2^i noeuds). Dans un arbre quasi complet les lignes, exceptée peut être la dernière, contiennent toutes un nombre maximal de noeuds (soit 2^i). De plus les feuilles de la dernière ligne se trouvent toutes à gauche, ainsi tous les noeuds internes sont binaires, excepté le plus à droite de l'avant dernière ligne qui peut ne pas avoir de enfants droit. Les feuilles sont toutes sur la dernière et éventuellement l'avant dernière ligne.



Un tas. à gauche, la vue du tas sous forme d'arborescence, à droite le tableau qui le supporte.

On peut numéroter cet arbre en largeur d'abord, c'est à dire dans l'ordre donné par les petits numéros figurant au dessus de la figure 4.3. Dans cette numérotation on vérifie que tout noeud *i* a son père en position *i*/2, l'enfant gauche du noeud ***i*** est **2*i***, l'enfant droit **2*i* + 1**.

Ceci permet d'implémenter cet arbre dans un tableau a (voir figure 4.4) où le numéro de chaque noeud donne l'indice de l'élément du tableau contenant sa valeur.

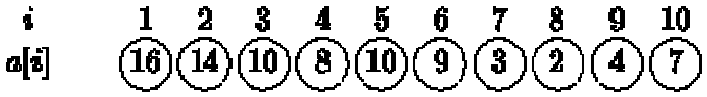


Figure 4.4 : Représentation en tableau d'un tas

L'ajout d'un nouvel élément *v* à la file consiste à incrémenter *n* puis à poser *a*[*n*] = *v*. Ceci peut ne plus représenter un tas car la relation $a[n/2] \geq v$ n'est pas nécessairement satisfaite. Pour obtenir un tas, il faut échanger la valeur contenue au noeud *n* et celle contenue par son père, remonter au père et réitérer jusqu'à ce que la condition des tas soit vérifiée. Ceci se programme par une simple itération (cf. la figure 4.5).

```
static void ajouter (int v)  {
  ++nTas;
  int i = nTas - 1;
  whi le (( i > 1) && ( a[pere (i)] < clé )) {
     a[i] = a[pere (i)];
   i = pere (i);
}
 a[i] = v;
```

ou plus précisément :

```
  while (i > 0 && a [i/2] <= v) {
     a[i] = a[i/2];
     i = i/2;
```

```
  }
  a[i] = v;
}
```

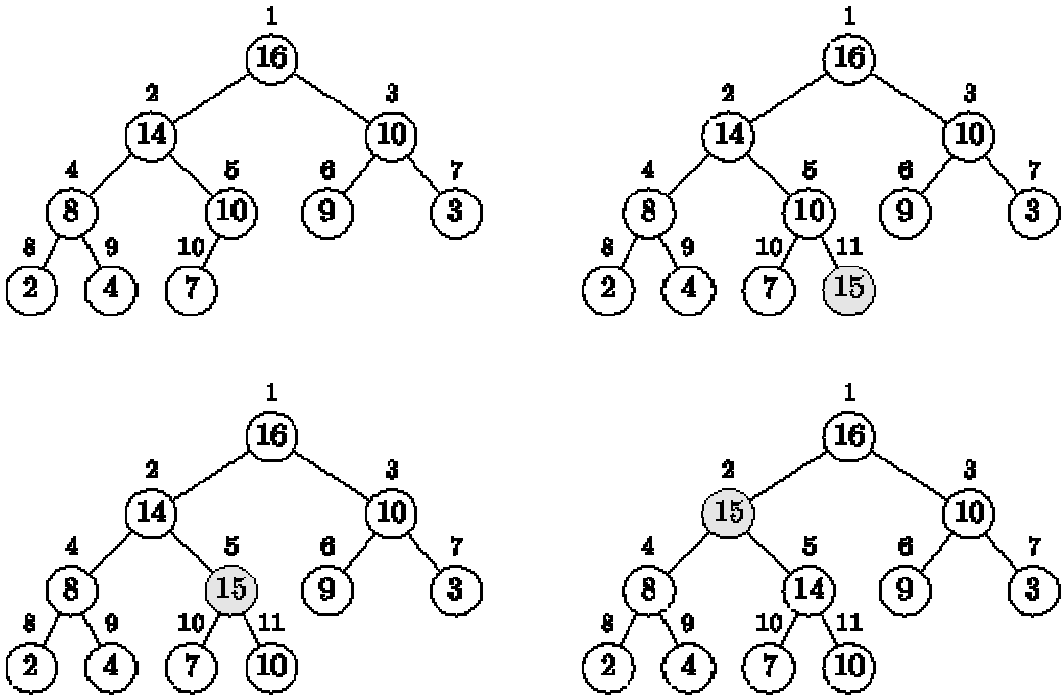


Figure 4.5 : Ajout dans un tas

On vérifie que, si le tas a *n* éléments, le nombre d'opérations n'excédera pas la hauteur de l'arbre correspondant. Or la hauteur d'un arbre binaire complet de *n* noeuds est log *n*. Donc Ajouter ne prend pas plus de $O(\log n)$ opérations.

On peut remarquer que l'opération de recherche du maximum est maintenant immédiate dans les tas. Elle prend un temps constant $O(1)$.

```
static int maximum ()  {
   return a[0];
}
```

Suppression d'un élément

Considérons l'opération de suppression du premier élément de la file. Il faut alors retirer la racine de l'arbre représentant la file, ce qui donne deux arbres! Le plus simple pour reformer un seul arbre est d'appliquer l'algorithme suivant:

On met l'élément le plus à droite de la dernière ligne à la place de la racine, on compare sa valeur avec celle de ses enfants, on échange cette valeur avec celle du vainqueur de ce tournoi, et on

réitère cette opération jusqu'à ce que la condition des tas soit vérifiée. Bien sûr, il faut faire attention, quand un noeud n'a qu'un enfants, et ne faire alors qu'un petit tournoi à deux. Le placement de la racine en bonne position est illustré dans la figure 4.6.

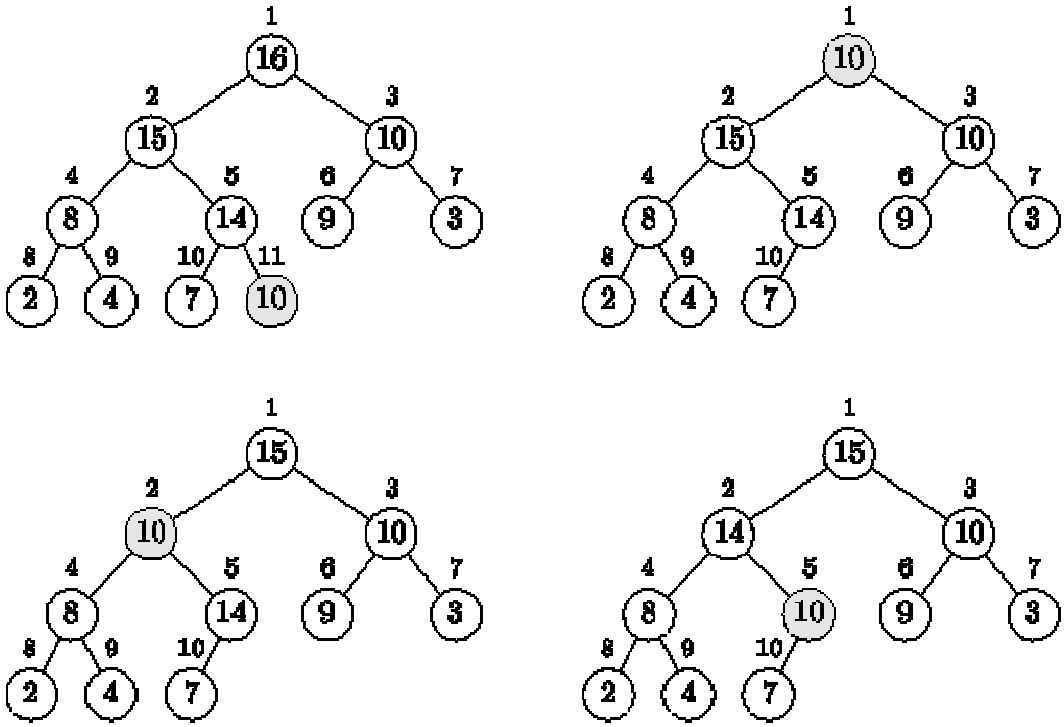


Figure 4.6 : Suppression dans un tas

```
static void supprimer () {

  int i, j;
  int v;
  a[0] = a[nTas - 1];
  --nTas;
  i = 0; v = a[0];
  while (2*i + 1 < nTas) {
     j = 2*i + 1;
     if (j + 1 < nTas)
        if (a[j + 1] > a[j])
           ++j;
     if (v >= a[j])
        break;
     a[i] = a[j]; i = j;
  }
  a[i] = v;
}
```

A nouveau, la suppression du premier élément de la file ne prend pas un temps supérieur à la hauteur de l'arbre représentant la file. Donc, pour une file de *n* éléments, la suppression prend

O(log *n*) opérations. La représentation des files de priorités par des tas permet donc de faire les trois opérations demandées: ajout, retrait, chercher le plus grand en log *n* opérations.

Ces opérations sur les tas permettent de faire le tri *HeapSort*. Ce tri possède la bonne propriété d'être toujours en temps *n* log *n*.

L'algorithme de HeapSort se divise en deux phases, la première consiste à construire un tas dans le tableau à trier, la seconde à répéter l'opération de prendre l'élément maximal, le retirer du tas en le mettant à droite du tableau. Il reste à comprendre comment on peut construire un tas à partir d'un tableau quelconque. On remarque d'abord que l'élément de gauche du tableau est à lui seul un tas. Puis on ajoute à ce tas, le deuxième élément avec la procédure Ajouter que nous venons de voir, puis le troisième, ... etc.

A la fin, on obtient bien un tas de *N* éléments dans le tableau à trier. Le programme est:

```
 static void heapSort ()  {

  int      i,v;
  nTas = 0;
  for (i = 0; i < N; ++i)
     ajouter (a[i]);
  for (i = N - 1; i >= 0; --i) {
     v = maximum();
     supprimer();
     a[i] = v;
   }
}
```

Sources et références

1. T.H. Cormen et al. (1990): Algorithms, MacGraw Hill.
2. John Morris (1998): Notes de cours de data structures and algorithms, University of western Autralia.
3. L.B. Holder (1999): Notes de cours d'algorithms and data structures, University of Texas at Arlington.
4. O. Carton (2003): Notes de cours d'algorithmique avancée en master de Bio-Info, Université Paris 7.