Table des matières

INTRODUCTION

I. DEFINITION ET COMPREHENSION DU THEME 4

1. Arbres binaires (Rappel).....4
2. Arbres binaires de recherché (Rappel).....4
   a. Recherche.....6
   b. Maximum et minimum 7
   c. Successeur et prédécesseur.....7
   d. Insertion 8
   e. Suppression 8

II. ETUDE DE CAS 10

A. ARBRES ROUGE-NOIR 10
   1. Présentation d'un arbre rouge noir.....10
   2. Manipulation et fonctionnement d'un arbre rouge-noir.....13
      a. Rotation.....13
      b. Insertion 10
      c. Suppression 18
B. ARBRES BINAIRES DE RECHERCHE OPTIMAUX 20
   1. Contexte et definition.....20
   2. Cout de recherche dans un Arbre binaire de recherche connaissant les fréquences de recherche de chaque nœud.....21
   3. Construction d'un arbre optimal.....23
      a. Sous structure optimale.....23
      b. Solution recursive 23
      c. Calcul du cout moyen de recherché dans l'arbre binaire optimal 25

CONCLUSION

Table des figures

Figure 1 4
Figure 2 5
Figure 3 8
Figure 4 11
Figure 5 11
Figure 6 12
Figure 7 12
Figure 8 14
Figure 10 21
Figure 11 22
Figure 12 26
Figure 13 26

INTRODUCTION

Un arbre binaire de recherche de hauteur *h* permet de mettre en œuvre les opérations fondamentales d'ensemble dynamique, telles RECHERCHER, PRÉDÉCESSEUR, SUCCESSEUR, MINIMUM, MAXIMUM, INSÉRER et SUPPRIMER, avec un temps $O(h)$. Les opérations sont donc d'autant plus rapides que la hauteur de l'arbre est petite ; mais si cette hauteur est grande, les performances risquent de ne pas être meilleures qu'avec une liste chaînée. Les arbres rouge-noir sont l'un des nombreux modèles d'arbres de recherche « équilibrés », qui permettent aux opérations d'ensemble dynamique de s'exécuter en temps $O(\lg n)$ dans le cas le plus défavorable.

I. DEFINITION ET COMPREHENSION DU THEME

1. Arbres binaires

Un **arbre binaire** est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé *racine*. Dans un arbre binaire, chaque élément possède au plus deux éléments appelés ‘‘fils’’ au niveau inférieur, habituellement appelés *gauche* et *droit*. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé *père*.

Au niveau le plus élevé il y a donc un nœud racine. Au niveau directement inférieur, il y a au plus deux nœuds fils. En continuant à descendre aux niveaux inférieurs, on peut en avoir quatre, puis huit, seize, etc. c'est-à-dire la suite des puissances de deux. Un nœud n'ayant aucun fils est appelé *feuille*.
La hauteur d’un arbre est le nombre de niveaux total, commençant par 0 ; autrement dit la distance entre la feuille la plus éloignée et la racine.
Le niveau où se trouve un nœud particulier est appelé *profondeur* de ce nœud.

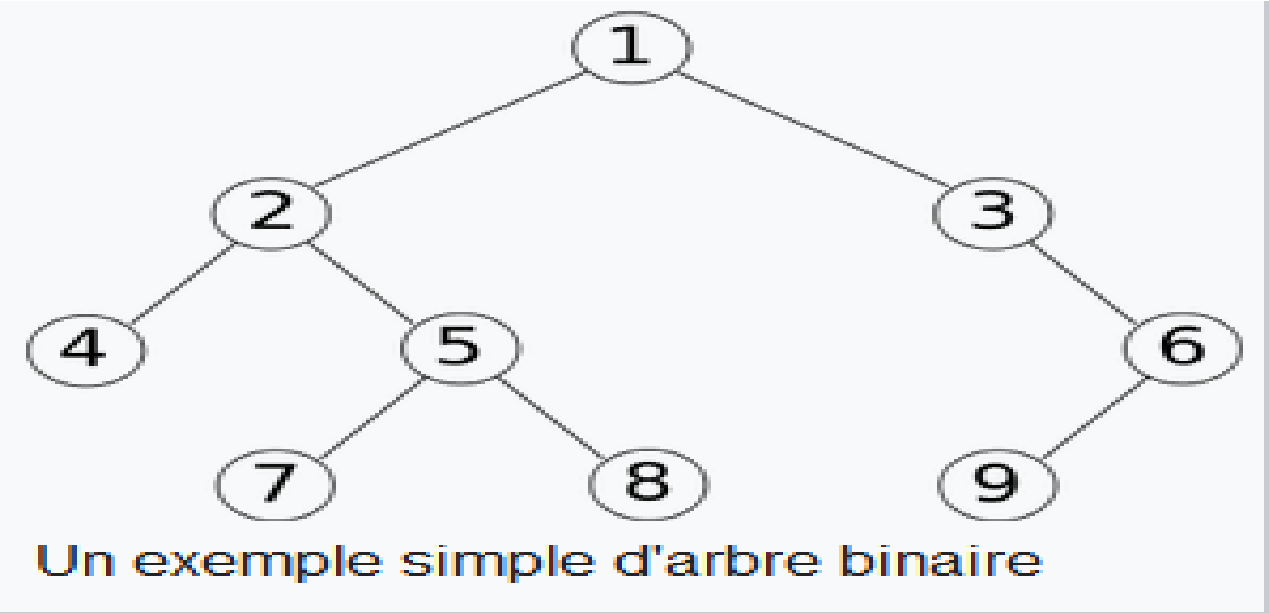


Figure 1 : Exemple simple d’arbre binaire

2. Arbre Binaire de Recherche (ABR)

Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud possède une clé, telle que chaque nœud du sous arbre *gauche* ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous arbre *droit* possède une clé supérieure ou égale à celle-ci ; Selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale. Les

nœuds que l'on ajoute deviennent des feuilles de l'arbre.
Un arbre binaire de recherche est dit équilibré si la différence de profondeur de deux feuilles quelconques est d'au plus 1.
Il existe de nombreux types d'arbres binaires de recherche, parmi lesquels les arbres rouge-noir et les arbres binaires de recherche optimaux (dont nous prendrons le soin de bien étudier).

Ci-dessous est un exemple d'arbre binaire de recherche équilibré.

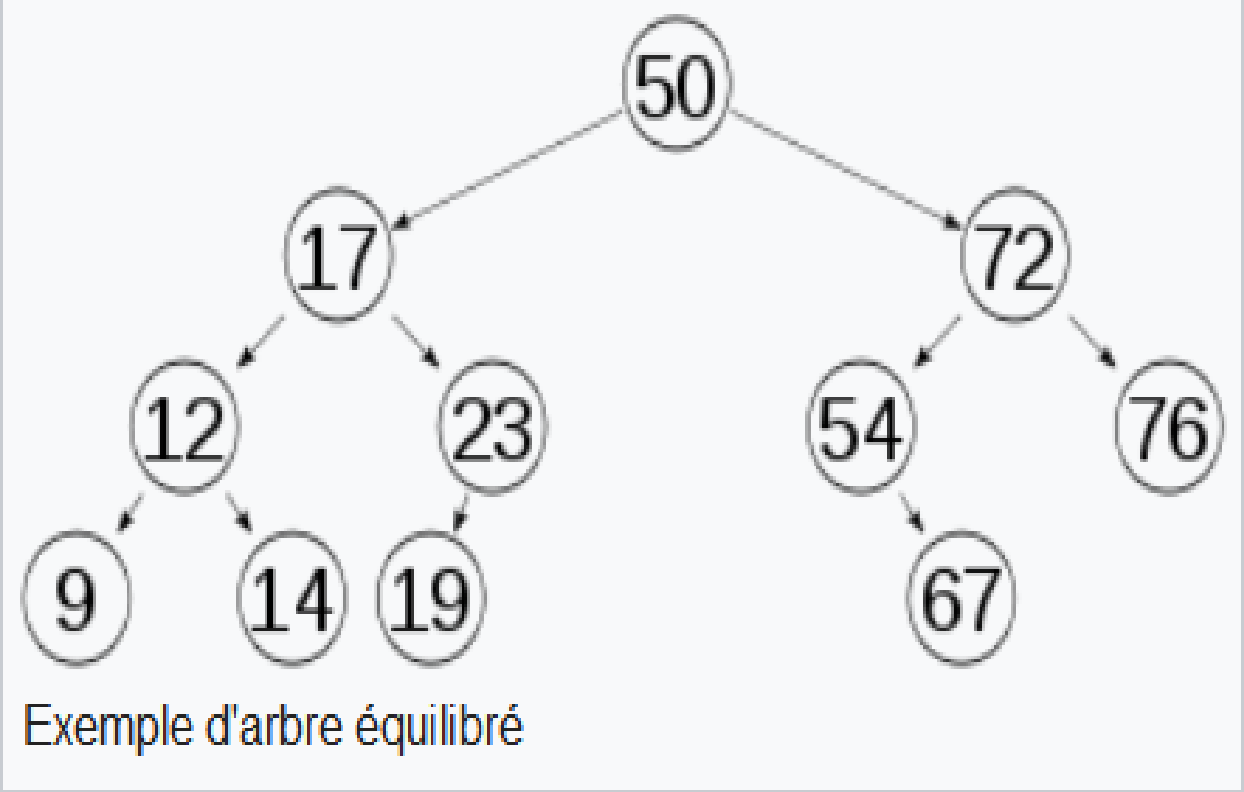


Figure 2 : Exemple d'arbre binaire équilibré

Un arbre binaire de recherche de hauteur *h* permet de mettre en œuvre les opérations fondamentales d'ensemble dynamique, telles RECHERCHER, PRÉDÉCESSEUR, SUCCESSEUR, MINIMUM, MAXIMUM, INSÉRER et SUPPRIMER, avec un temps $O(h)$.

a. Recherche

La recherche dans un arbre binaire d'un nœud ayant une clé particulière peut être considéré un procédé récursif ou itératif. On commence par examiner la racine. Si sa clé est la clé recherchée, l'algorithme se termine et renvoie la racine. Si elle est strictement inférieure, alors elle est dans le sous arbre gauche, sur lequel on effectue alors récursivement la recherche. De même si la clé recherchée est strictement supérieure à la clé de la racine, la recherche continue

dans le sous arbre droit. Si on atteint une feuille dont la clé n'est pas celle recherchée, on sait alors que la clé recherchée n'appartient à aucun nœud, elle ne figure donc pas dans l'arbre de recherche. L'algorithme itératif de l'opération rechercher dans l'arbre binaire de recherche peut se donner comme suit

```
ARBRE-RECHERCHER-ITÉRATIF(x, k)
1  tant que x ≠ NIL et k ≠ clé[x]
2       faire si k < clé[x]
3              alors x ← gauche[x]
4              sinon x ← droite[x]
5  retourner x
```

b. Minimum et maximum

On peut toujours trouver un élément d'un arbre binaire de recherche dont la clé est un minimum en suivant les pointeurs gauches à partir de la racine jusqu'à ce qu'on rencontre NIL. La procédure suivante retourne un pointeur sur l'élément minimal du sous arbre enracine au nœud x

```
ARBRE-MINIMUM(x)
1  tant que gauche[x] ≠ NIL
2       faire x ← gauche[x]
3  retourner x
```

Le pseudo code pour recherche le maximum dans un arbre binaire de recherche est symétrique

c. Successeur et Prédécesseur

Dans un arbre binaire de recherche, le successeur d'un nœud x est le nœud contentant la plus petite clé supérieure a x, et son prédécesseur est le nœud contenant la plus grande clé inferieure a x. La procédure suivante retourne le successeur d'un nœud x dans un arbre binaire de recherche, s'il existe, et NIL si x possède la plus grande clé de l'arbre

```
ARBRE-SUCCESSEUR(x)
1  si droite[x] ≠ NIL
2     alors retourner ARBRE-MINIMUM(droite[x])
3  y ← p[x]
4  tant que y ≠ NIL et x = droite[y]
5     faire x ← y
6           y ← p[y]
7  retourner y
```

d. Insertion

L'insertion d'un nœud commence par une recherche : on cherche la clé du nœud à insérer ; lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille : si elle est inférieure, le nouveau nœud sera à gauche ; sinon il sera à droite. La procédure suivante montre comment insérer un élément dans un arbre binaire de recherche.

```
ARBRE-INSÉRER(T, z)
 1  y ← NIL
 2  x ← racine[T]
 3  tant que x ≠ NIL
 4     faire y ← x
 5           si clé[z] < clé[x]
 6              alors x ← gauche[x]
 7              sinon x ← droite[x]
 8  p[z] ← y
 9  si y = NIL
10     alors racine[T] ← z                ▷ arbre T était vide
11     sinon si clé[z] < clé[y]
12              alors gauche[y] ← z
13              sinon droite[y] ← z
```

e. Suppression

Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé à partir de sa clé :

- **Suppression d'une feuille** : Il suffit de l'enlever de l'arbre vu qu'elle n'a pas de fils.
- **Suppression d'un nœud avec un enfant** : Il faut l'enlever de l'arbre en le remplaçant par son fils.

- **Suppression d'un nœud avec deux enfants** : Supposons que le nœud à supprimer soit appelé *N* (le nœud de valeur 7 dans le graphique ci-dessous). On échange le nœud N avec son successeur le plus proche (le nœud le plus à gauche du sous arbre droit - ci-dessous, le nœud de valeur 9) ou son plus proche prédécesseur (le nœud le plus à droite du sous arbre gauche - ci-dessous, le nœud de valeur 6). Cela permet de garder une structure d'arbre binaire de recherche. Puis on applique à nouveau la procédure de suppression à *N*, qui est maintenant une feuille ou un nœud avec un seul fils. Mais en général, il est conseillé de faire l'échange avec le successeur, et non avec le prédécesseur

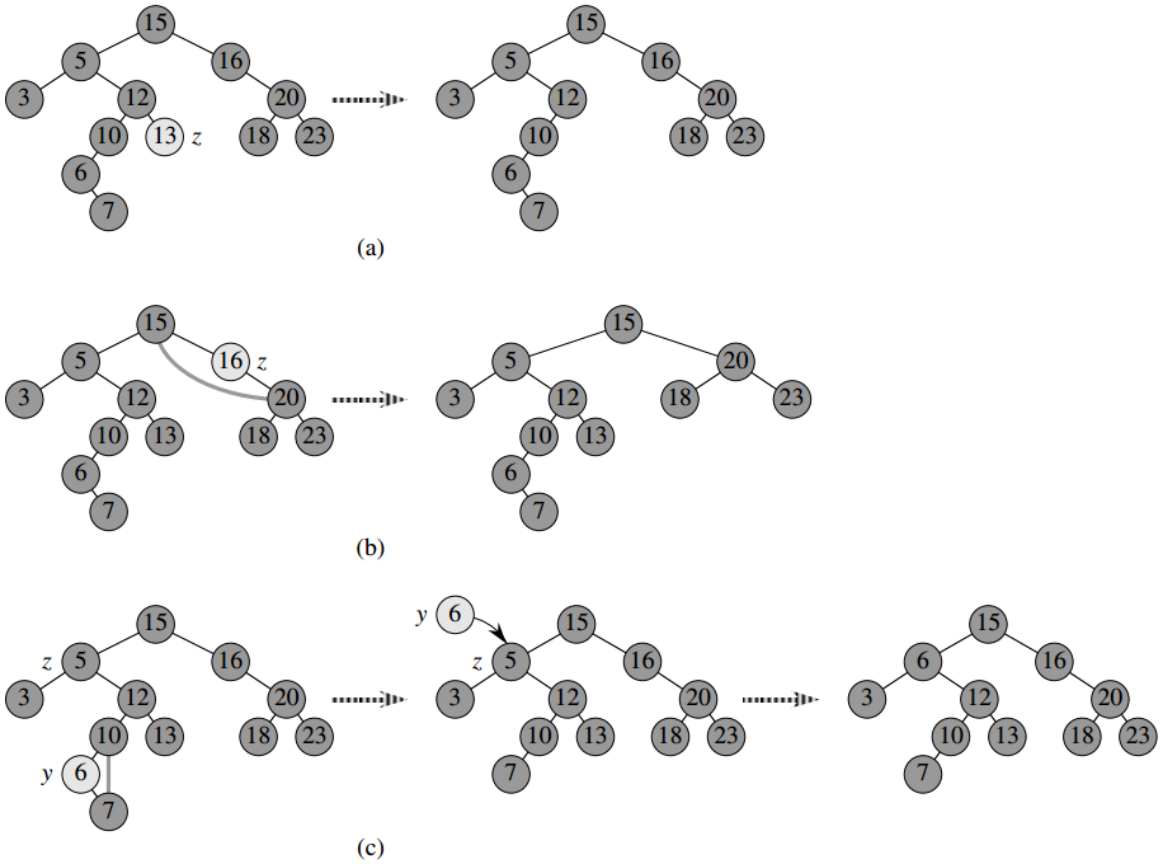


Figure 3 : Exemple de schéma de suppression

Le pseudo code suivant montre les différents cas de suppression dans un arbre binaire de recherche

```
ARBRE-SUPPRIMER(T, z)
 1  si gauche[z] = NIL ou droite[z] = NIL
 2     alors y ← z
 3     sinon y ← ARBRE-SUCCESSEUR(z)
 4  si gauche[y] ≠ NIL
 5     alors x ← gauche[y]
 6     sinon x ← droite[y]
 7  si x ≠ NIL
 8     alors p[x] ← p[y]
 9  si p[y] = NIL
10     alors racine[T] ← x
11     sinon si y = gauche[p[y]]
12              alors gauche[p[y]] ← x
13              sinon droite[p[y]] ← x
14  si y ≠ z
15     alors clé[z] ← clé[y]
16           copier données satellites de y dans z
17  retourner y
```

II. ETUDE DE CAS

A. ARBRES ROUGE-NOIR OU ARBRE BICOLORE

1. Présentation et définition d'un arbre rouge-noir

Un arbre rouge-noir est un arbre binaire de recherche satisfaisant les cinq propriétés suivantes :

1. *Chaque nœud est soit rouge, soit noir. **
2. *La racine est noire.*
3. *Chaque feuille (*NIL*) est noire.*
4. *Si un nœud est rouge, alors ses deux fils sont noirs.*
5. *Tous les chemins descendants reliant un nœud donné à une feuille (du sous arbres dont il est la racine) contiennent le même nombre de nœuds noirs.*

Dans un arbre binaire de recherche, la hauteur noire d'un nœud x est le nombre de nœuds noirs d'un chemin partant de ce nœud (le nœud en question n'est pas compte) vers une feuille, et on la note $bh(x)$. De toute évidence, la hauteur noire d'un arbre rouge-noir est donc la hauteur noire de sa racine. Un exemple d'arbre rouge-noir de hauteur noire 3 est représenté à la figure 4.
Comme nous l'avons dit plus haut, les opérations de recherche dans un arbre binaire de recherche sont de l'ordre de $O(h)$ avec h étant la hauteur de l'arbre. En considerant sa

hauteur, le lemme suivant nous montre pourquoi les arbres rouge-noir sont de bons arbres de recherche

Lemme : Un arbre rouge-noir ayant n nœuds internes à une hauteur au plus égale a $2\log_2(n+1)$

Preuve : Commençons par montrer que le sous arbre enraciné en un nœud *x* quelconque contient au moins $2^{bh(x)}-1$ nœuds internes. Cette affirmation peut se démontrer par récurrence sur la hauteur de *x*. Si la hauteur de *x* est 0, alors *x* est obligatoirement une feuille (*nil*[*T*]) et le sous arbre enraciné en *x* contient effectivement au moins $2^{bh(x)}-1=2^0-1=0$ nœuds internes. Pour l'étape inductive, soit un nœud interne *x* de hauteur positive ayant deux enfants. Chaque enfant a une hauteur noire $bh(x)$ ou $bh(x)-1$, selon que sa couleur est rouge ou noire. Comme la hauteur d'un enfant de *x* est inférieure à celle de *x* lui-même, on peut appliquer l'hypothèse de récurrence pour conclure que chaque enfant a au moins $2^{bh(x)-1}-1$ nœuds internes. Le sous arbre de racine *x* contient donc au moins $\left(2^{bh(x)-1}-1\right)+\left(2^{bh(x)-1}-1\right)+1=2^{bh(x)}-1$ nœuds internes, ce qui démontre l'assertion.
Pour compléter la preuve du lemme, appelons *h* la hauteur de l'arbre. D'après la propriété 4 sur les arbres rouges noirs, au moins la moitié des nœuds d'un chemin simple reliant la racine à une feuille, racine non comprise, doivent être noirs. En conséquence, la hauteur noire de la racine doit valoir au moins $\frac{h}{2}$; donc, $n \geq 2^{\frac{h}{2}}-1$.
En faisant passer le 1 dans le membre gauche et en prenant le logarithme des deux membres, on obtient.

$$\log(n+1) \geq \frac{h}{2}, soit\ h \leq 2\log(n+1)$$

Pour simplifier le traitement des conditions aux limites dans la programmation des arbres rouge-noir, on utilise une même sentinelle pour représenter NIL. Pour un arbre rouge-noir *T*, la sentinelle *nil*[*T*] est un objet ayant les mêmes champs qu'un nœud ordinaire. Son champ *couleur* vaut NOIR, et ses autres champs (*p*, *gauche*, *droite* et *clé*) peuvent prendre des valeurs quelconques. Comme le montre la **Erreur ! Source du renvoi introuvable.**(b), tous les pointeurs vers NIL sont remplacés par des pointeurs vers la sentinelle *nil*[*T*].

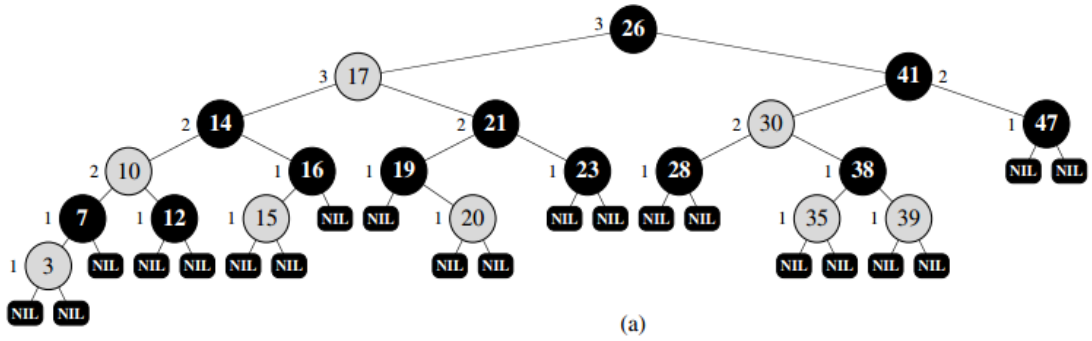


Figure 4 : Arbre rouge noir avec des nœuds nil

(a) Chaque feuille, représentée par (NIL), est noire. Chaque nœud non-NIL est étiqueté par sa hauteur noire ; les NIL ont une hauteur noire égale à 0.

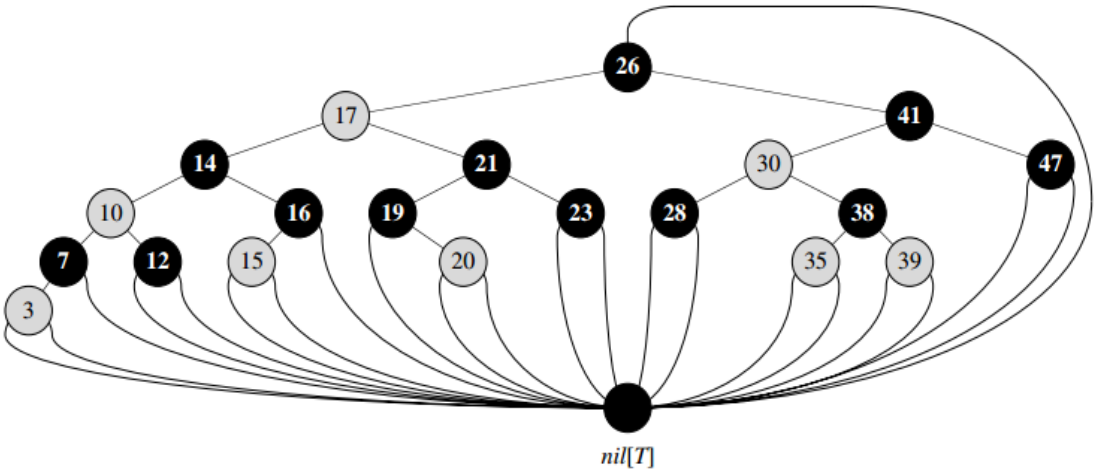


Figure 5 : Arbre rouge noir avec un seul nil

(b) Le même arbre rouge noir, mais où chaque NIL est remplacé par la sentinelle nil[T], qui est toujours noire, et où les Hauteurs noirs sont omises. Le parent de la racine est aussi la sentinelle.

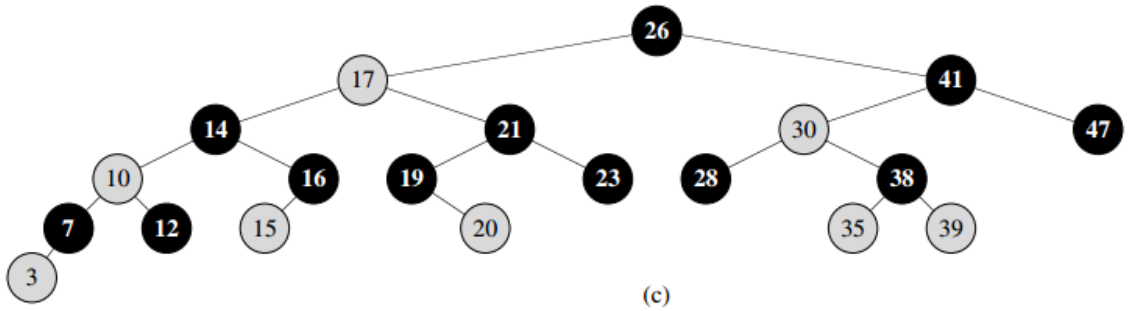


Figure 6 : Arbre rouge noir sans nœuds nil

(c) Le même arbre rouge-noir, mais où les feuilles et le parent de la racine ont été entièrement omis. Nous emploierons ce style de représentation dans le reste du chapitre

Le principal intérêt des arbres rouge-noir est qu'ils offrent la meilleure garantie sur le temps d'insertion, de suppression et de recherche dans les cas défavorables. Ceci leur permet non seulement d'être utilisable dans les applications en temps réel, mais aussi de servir comme fondement d'autres structures de données à temps d'exécution garanti dans les cas défavorables, par exemple en géométrie algorithmique (domaine de l'algorithmique qui traite des algorithmes manipulant des concepts géométriques). De plus, cette structure est économe en mémoire, puisqu'elle ne requiert qu'un bit supplémentaire d'informations par élément par rapport à un arbre binaire classique.

2. Manipulation et fonctionnement d'un arbre rouge-noir

2.1 Rotation

Lorsqu'on exécute les opérations ARBRE-INSÉRER et ARBRE-SUPPRIMER sur un arbre rouge-noir à n clés (Nous verrons ces procédures plus bas), prennent un temps O (lg n). Comme elles modifient l'arbre, le résultat pourrait violer les propriétés d'arbre rouge-noir énumérées plus haut, Pour restaurer ces propriétés, il faut changer les couleurs de certains nœuds de l'arbre et également modifier la chaîne des pointeurs. Pour retrouver les propriétés de l'arbre rouge-noir, nous utilisons un algorithme rotation.

Les rotations préservent la propriété des arbres de recherche mais pas la propriété des arbres rouge et noir.

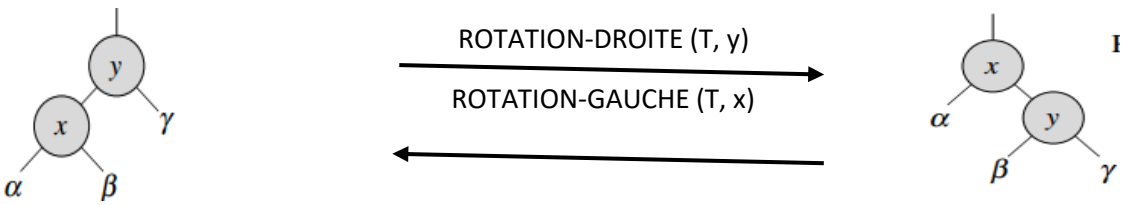


Figure 7 : Rotations sur un arbre binaire de recherche

Les opérations de rotation sur un arbre de recherche binaire. L'opération ROTATION-GAUCHE (T, x) transforme la configuration des deux nœuds de gauche pour aboutir à celle de droite, en modifiant un nombre constant de pointeurs. La configuration de droite peut être transformée en celle de gauche par l'opération inverse ROTATION-DROITE (T, y). Les deux nœuds peuvent se trouver n'importe où dans l'arbre binaire de recherche. Les lettres A, B et C représentent des sous arbres arbitraires. Une opération de rotation préserve la propriété d'arbre binaire de recherche : les clés de A précèdent clé[x], qui précède les clés de B, qui précède clé[y], B qui précède les clés de C.

Le pseudo code de ROTATION-GAUCHE suppose que droite[x]≠nil[T] et que le parent de la racine est nil[T].

```
ROTATION-GAUCHE(T, x)
  y ← droit(x)
  droit(x) ← gauche(y)
  si gauche(y) ≠ NIL alors père(gauche(y)) ← x
  père(y) ←père(x)
  si père(y) = NIL
     alors racine(T)← y
     sinon si x = gauche(père(x))
        alors gauche(père(x)) ← y
        sinon droit(père(x)) ← y
  gauche(y) ← x
  père(x) ← y
```

2.2 Insertion

L'insertion d'un nœud dans un arbre rouge noir se fait exactement comme dans un arbre binaire de recherche simple, a la seule exception que ce nœud prend la couleur rouge (si ce nœud prenait la couleur noire, on violerait la propriété 5 qui définit la hauteur noire dans un arbre rouge-noir.), et deux feuilles *nil[T]* lui sont attribuées, afin de ne pas violer la propriété 3 des arbres binaires de recherche. Ce qui se passe ensuite dépend de la couleur des nœud *''voisins''* a ce nœud nouvellement inséré. Remarquer qu'à la suite de cette insertion telle qu'expliqué plus haut, la propriété 2 ou la propriété 4 des arbres rouge-noir pourraient être violées. Plusieurs cas de figures sont envisageables tel qu'expliqués ici-bas.

Cas N° 1 : le nœud N est la racine de l'arbre.

Dans ce cas ce nœud est simple repeint en noir, et toutes les propriétés de l'arbre rouge noir sont vérifiées.

Cas N° 2 : le Père P de N est noir.

Dans ce cas, l'arbre résultant reste toujours rouge-noir, aucune des 5 propriétés de l'arbre rouge-noir n'étant pas violée.

Dans les cas 3-5, il est supposé et vérifiable que N a un grand parent G, car son parent étant rouge, ne peut pas être la racine qui est forcément noir. De plus, N a un oncle que l'on note U (même si ce nœud peut être une feuille)

Cas N° 3 : le père P et l'oncle U de N sont rouges.

Dans ce cas, P et U changent de couleur et deviennent noir, tandis que G prend la couleur rouge de noir qu'il était, afin de conserver la propriété 5 (Egalite de la hauteur noire) de l'ABR. G prenant la couleur rouge pourrait de nouveau violer la propriété 4 concernant les enfants d'un nœud rouge, ou la propriété 2 (la racine doit toujours être noire). Si une de ces propriétés est violée, alors on reprend le cycle en considérant le nœud problématique G, et on se repère dans l'un des 5 cas présenté pour la résolution du problème.

Cas N° 4 : le père P de N est rouge, tandis que l'oncle U de N est noir, et le nœud N est l'enfant droit de son père P.

Dans ce cas, le nœud problématique N devient son père P, et on effectue une rotation gauche de l'arbre autour de P, ensuite, le nouveau père de N prend la couleur N, le grand père G de N prend la couleur rouge, et une nouvelle rotation droite de l'arbre autour de G est effectuée. Si un problème se retrouve encore au niveau du nouveau nœud N, le cycle recommence, et on identifie dans quel cas on se trouve.

Cas N° 5 : le père P de N est rouge, tandis que l'oncle U de N est noir, et le nœud N est l'enfant gauche de son père P.

Dance ce cas, la couleur de son père P devient noire, celle de son grand père G devient rouge, et on effectue une rotation droite de l'arbre autour de du grand-père G de N. Si un problème se retrouve encore au niveau du nouveau nœud N, le cycle recommence.

Note : Deux cas de figures se présentent encore **(Cas N° 6 et Cas N° 7)**, qui sont les symétriques respectifs des **Cas N° 4 et Cas N° 5**, chaque gauche devenant droit, et chaque droit devenant gauche.

Aux exceptions du premier et du second cas (Cas élémentaires), tous les autres cas de figure expliqués plus haut se retrouvent dans l'algorithme de correction présenté ci bas

```
RN-INSÉRER(T, z)
1   y ← nil[T]
2   x ← racine[T]
3   tant que x ≠ nil[T]
4       faire y ← x
5            si clé[z] < clé[x]
6               alors  x ← gauche[x]
7               sinon x ← droite[x]


8   p[z] ← y
9   si y = nil[T]
10      alors  racine[T] ← z
11      sinon si clé[z] < clé[y]
12                alors  gauche[y] ← z
13                sinon droite[y] ← z
14  gauche[z] ← nil[T]
15  droite[z] ← nil[T]
16  couleur[z] ← ROUGE
17  RN-INSÉRER-CORRECTION(T, z)
```

```
RN-INSÉRER-CORRECTION(T, z)
 1  tant que couleur[p[z]] = ROUGE
 2      faire si p[z] = gauche[p[p[z]]]
 3             alors y ← droite[p[p[z]]]
 4                  si couleur[y] = ROUGE
 5                     alors couleur[p[z]] ← NOIR                    ▷ Cas 1
 6                           couleur[y] ← NOIR                       ▷ Cas 1
 7                           couleur[p[p[z]]] ← ROUGE                ▷ Cas 1
 8                           z ← p[p[z]]                             ▷ Cas 1
 9                     sinon si z = droite[p[z]]
10                              alors z ← p[z]                       ▷ Cas 2
11                                    ROTATION-GAUCHE(T, z)          ▷ Cas 2
12                           couleur[p[z]] ← NOIR                    ▷ Cas 3
13                           couleur[p[p[z]]] ← ROUGE                ▷ Cas 3
14                           ROTATION-DROITE(T, p[p[z]])             ▷ Cas 3
15             sinon (idem clause alors avec
                           permutation de « droite » et « gauche »)
16  couleur[racine[T]] ← NOIR
```

Un exemple d'insertion est présenté ici-bas : nous insérons un nœud avec une clé de 4 dans l'arbre proposé.

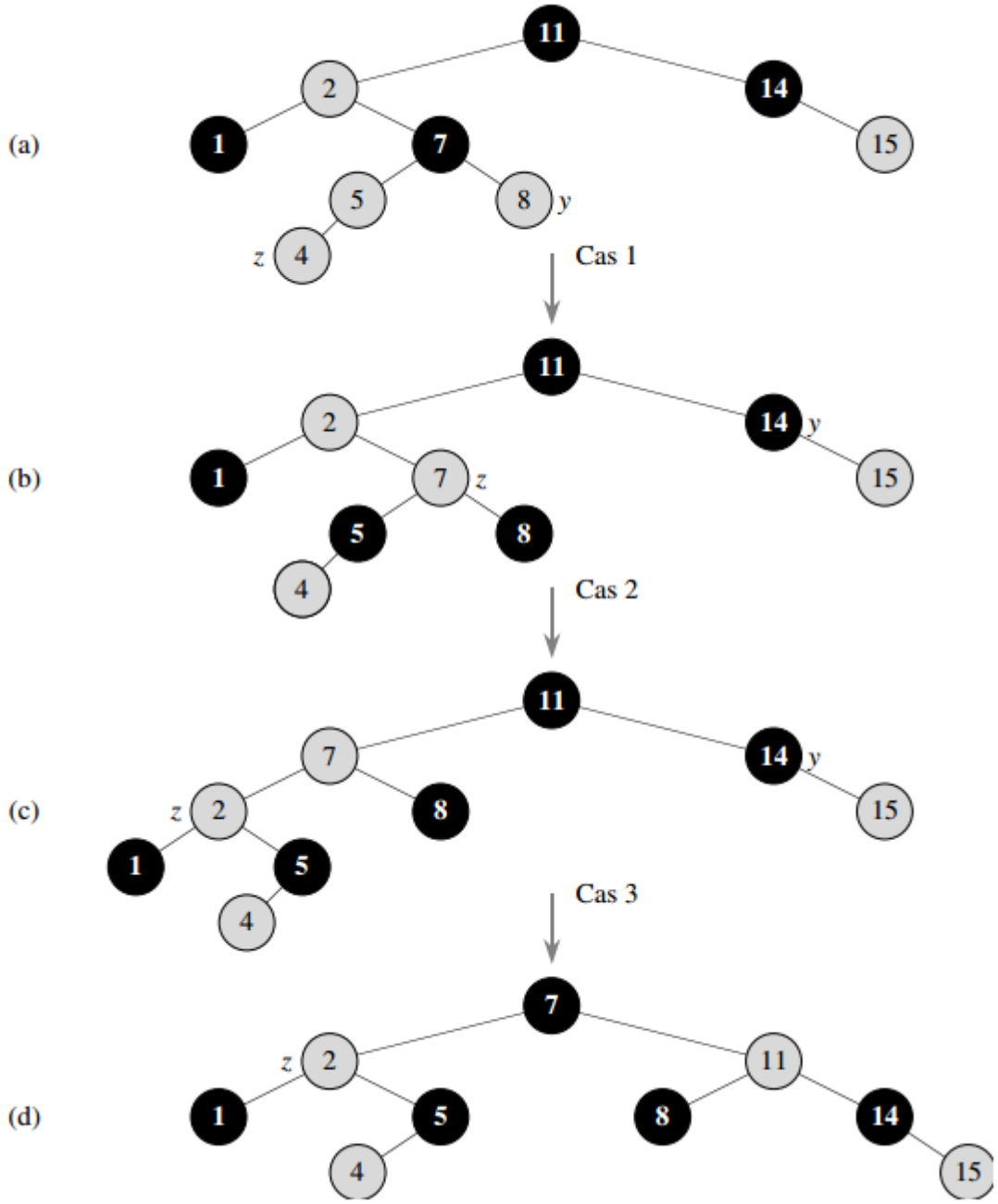


Figure 8 : Figure montrant 4 cas d'insertion dans un arbre rouge-noir

2.3. Suppression

Pour supprimer un élément dans un arbre rouge et noir, on commence par appliquer l'algorithme de suppression pour les arbres de recherche. Si l'élément supprimé était de couleur rouge, aucune des propriétés des arbres rouge et noir n'est violée. Par contre, si le

nœud supprimé était noir la propriété 4 (tous les chemins descendants d'un nœud à une feuille contiennent le même nombre de nœuds noirs) peut être violée. Le code ci-dessous présente l'algorithme de suppression avec utilisation de sentinelles et appel de l'algorithme de correction celui qui répartit les « noirs » surnuméraires.

```
ARBRE_RN_SUPPRESSION(T,x)
 1   si gauche(x)=NIL et droit(x) alors
 2      si père(x)=NIL alors
 3         racine(T)=NIL
 4      sinon
 5         si x=gauche(père(x)) alors
 6            gauche(père(x)) ←NIL
 7         sinon droit(père(x)) ←NIL
 8         si couleur(x)=noir alors
 9            RN_CORRECTION(T,x)
10   sinon si gauche(x)=NIL et droit(x)=NIL alors
11      si gauche(x)≠NIL alors
12         filsde_x=gauche(x)
13      sinon filsde_x=droit(x)
14      père(filsde_x) ←père(x)
15      si père(x) = NIL alors
16         racine(T )← filsde_x
17      sinon si gauche(père(x)) = x alors
18         gauche(père(x)) ← filsde_x
19         sinon droit(père(x)) ← filsde_x
20      si couleur(x) = NOIR alors
21         RN_CORRECTION(T , filsde_x)
22   sinon
23      min ← ARBRE_MINIMUM(droit(x))
24      clé(y) ←clé(min)
25      ARBRE-RN-SUPPRESSION(T,min)
26    renvoyer racine(T )
```

```
RN-CORRECTION(T , x)
1   tant que x 6= racine(T) et couleur(x) = NOIR faire
2      si x = gauche(père(x)) alors
3         w ← droit(père(x))
4         si w=NIL alors
5            x=père(x) ;
6            si x=gauche(père(x)) alors
7               w= gauche(père(x))
8            sinon w= droit(père(x))
9         sinon si couleur(w) = ROUGE alors
10           couleur(w) ← NOIR
11           couleur(père(w)) ← ROUGE
12            ROTATION-GAUCHE(T , père(x))
13            w ← droit(père(x))
14         sinon si couleur(gauche(w)) = NOIR et couleur(droit(w)) = NOIR alors
15            couleur(w) ← ROUGE
16            x ← père(x)
17         sinon
18               si couleur(droit(w)) = NOIR alors
19                  couleur(gauche(w)) ← NOIR
20                  couleur(w) ← ROUGE
21                   ROTATION-DROITE(T , w)
22                   w ← droit(père(x))
23               couleur(w) ← couleur(père(x))
24               couleur(père(x)) ← NOIR
25               couleur(droit(w))) ← NOIR
26               ROTATION-GAUCHE(T , père(x))
27               x ← racine(T)
28      sinon (même chose que précédemment en échangeant droit et gauche)
29  couleur(x) ← NOIR
```

B. ARBRES BINAIRES DE RECHERCHE OPTIMAUX

1. Contexte et définition

On se propose de construire un programme comme Google-translate, qui traduit un texte d'une langue en une autre (Français-Anglais par exemple). On se propose donc de construire un Arbre Binaire de Recherche a n mots comme étant les clés de cet arbre. On peut utiliser un arbre rouge-noir ou n'importe quel autre ABR équilibré, et s'assurer que le temps de recherche est de $O(\log n)$ dans le pire des cas. Mais en faisant ainsi, un mot fréquemment utilisé comme l'article « un » ou la préposition « de » peut se trouver loin de la racine, impliquant la consultation de plusieurs mots pour chaque fois qu'on le cherche (et donc temps de recherche élevé), et un mot

rarement utilise comme « anticonstitutionnellement » peut se trouver très proche de la racine de l'arbre. On veut donc construire un Arbre Binaire de Recherche tel que
les mots fréquemment utilisés soient proche de la racine, et les mots rarement utilise loin de celle-ci, réduisant ainsi le cout de recherche total. Comment peut-on organisé notre ABR, connaissant les fréquences d'apparition d'un mot, afin de réduire le nombre de nœuds visités lors d'une recherche ?
Ce qu'il nous faut est un **Arbre Binaire de Recherche Optimal.** Ainsi, un ABRO est un Arbre Binaire de Recherche dans lequel, connaissant la fréquence de recherche des clés, les nœuds sont arrangés de façon à ce que le cout de l'arbre soit minimum.

2. Cout de recherche dans un Arbre binaire de recherche connaissant les fréquences de recherche de chaque nœud

Considérons une séquence de n clés $K = < k_1, k_2, \dots, k_n >$ telle que $(k_1 < k_2 < \cdots < k_n)$. Nous voulons construire un ABR T à partir de ces clés, connaissant les probabilités p_i que lorsqu'une recherche sera effectuee dans l'arbre, elle concerne la cle k_i. Certaines recherches dans l'arbre pourront concernées des éléments qui ne sont pas dans K. Pour pouvoir s'occuper de ces cas, nous considérons également $n + 1$ clés factices $< d_0, d_1, d_2, \dots, d_n >$, représentant toutes les valeurs qui ne sont pas dans K, ayant des probabilites q_i d'être recherché. En particulier, d_0 représente toutes les valeurs inferieures a k_1, d_n représente toutes les valeurs supérieures a k_n, et pour $i = 1, 2, \dots, n-1$, d_i représente les valeurs comprises entre k_i et k_{i+1}

Nous supposons que le cout de recherche (que nous noterons E) est égal au nombre de nœud examiné avant de trouver le nœud cherche. Ainsi, pour une clé k_i, on a

$$E(k_i) = Prof(k_i) + 1 \; avec \; Prof(k_i) = Profondeur \; de \; la \; cle \; k_i \; dans \; l'abre \; T$$

Puisqu'une recherche dans un arbre est toujours soit successif ou pas, on a la somme de toutes les probabilités égale à un. viz

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

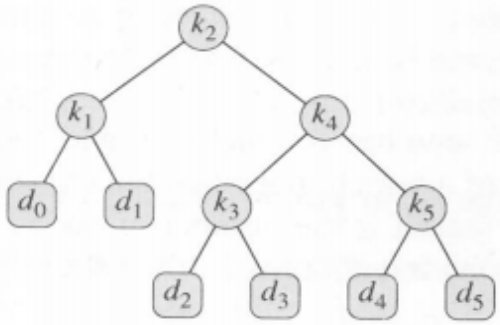
Le cout moyen de recherche dans l'arbre T est donne par

$$E[T] = \sum_{i=1}^{n} (Prof(k_i) + 1\,).p_i + \sum_{i=0}^{n} (Prof(d_i) + 1\,).q_i$$

$$= 1 + \sum_{i=1}^{n} Prof(k_i).p_i + \sum_{i=0}^{n} Prof(d_i).q_i$$

Un exemple de calcul du cout de recherche moyen est présenté dans la figure ci-dessous, ou ce cout vaut **2.8**

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



cost: 2.80

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Figure 10 : Scenario de construction d'un arbre optimal avec les données relatives

Mais, toujours pour ces mêmes clés et ces mêmes probabilités, on peut obtenir un arbre binaire de recherche avec un cout moyen de recherche moins élevé voir figure 11. D'ailleurs, c'est l'ABRO optimal pour les données considérées.

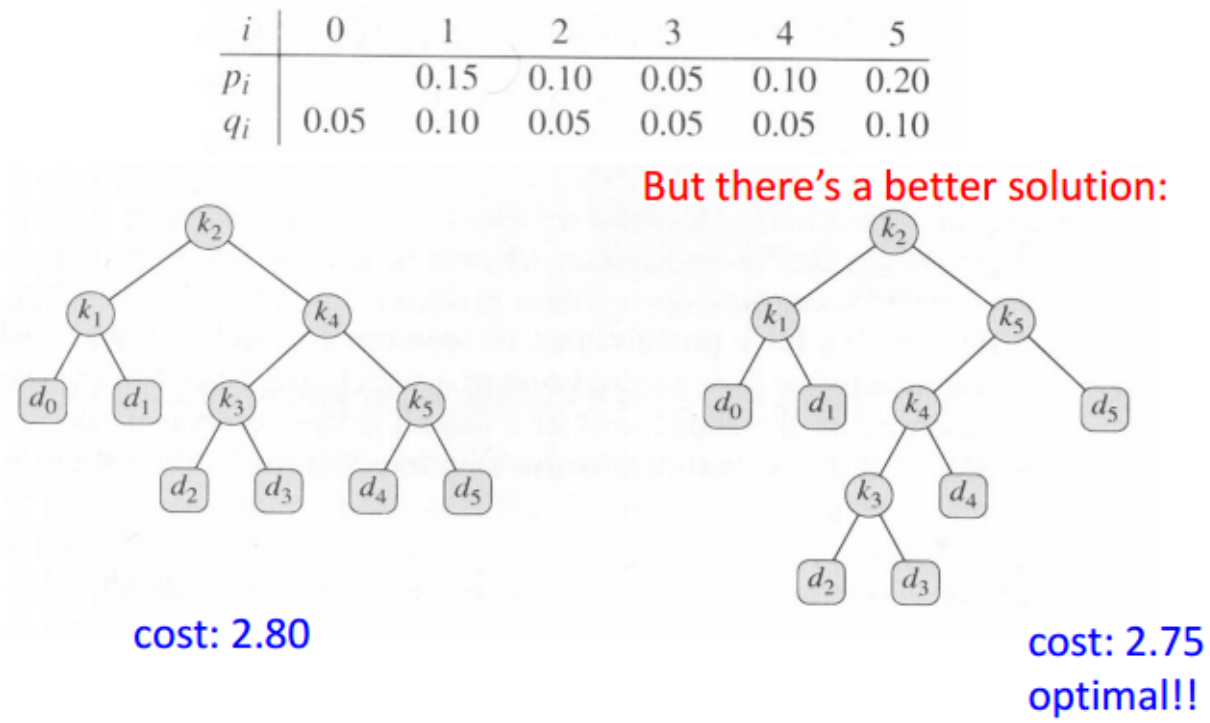


Figure 11 : Arbre binaire de recherche optimal résultant des données de la figure 10 plus-haut

De cet ABRO, on peut faire deux remarques capitales dans les ABRO :

- Un Arbre Binaire de Recherche Optimal n'a pas forcement la plus petite hauteur possible
- Un Arbre Binaire de Recherche Optimal n'a pas forcement comme racine la clé avec la plus grande probabilité d'être recherché.

La question qui naturellement se pose est de savoir comment à partir des clés et des probabilités de ces clés d'être recherché, on peut construire un ABRO
La première réponse serai de construire tous les ABR possible, et de calculer leurs couts moyens pour ensuite choisir le meilleur.
Mais cette option n'est du tout pas efficace, car lorsque le nombre de nœuds n d'un arbre devient grand, les différentes configurations d'ABR résultant grandissent exponentiellement. (Il est à remarquer que pour n nœud, on a environs $\frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}} = nombre\ de\ Catalan$ differentes configuratio d'ABR possible).

Une méthode assez efficace de retrouver un ABRO lorsque nous avons les données telles que présentées plus haut, est ***la programmation dynamique.*** Nous appliquerons les 4 étapes de la

programmation dynamique pour voir comment à partir des données nous pouvons construire un ABRO

3. Construction d'un Arbre Binaire de Recherche Optimal

a. *Sous structure optimale.*

Tout sous arbre de l'ABR contient les clés contiguë $k_i, k_{i+1}, \dots, k_j$. Si l'arbre T qui est un ABRO contient un sous arbre T' avec les clés $k_i, k_{i+1}, \dots, k_j$, alors, T' est forcement optimal. Car si T' n'est pas optimal, alors il suffit de changer T' par le sous arbre optimal correspondant et l'arbre résultant sera « plus optimal » que T, ce qui est une contradiction du fait que T est optimal.
Nous devons utiliser la sous-structure optimale pour montrer que l'on peut construire une solution optimale du problème à partir de solutions optimales de sous-problèmes. Soient les clés $k_i, k_{i+1}, \dots, k_j$; l'une de ces clés, par exemple k_r $(i \leq r \leq j)$, est la racine d'un sous arbre optimal contenant ces clés. Le sous arbre gauche de la racine k_r contiendra les clés $k_i, \dots, k_{r-1}$ (et les clés factices $d_{i-1}, \dots, d_{r-1}$) ; le sous arbre droite de la racine k_r contiendra les cles $k_{r+1}, \dots, k_j$ (et les clés factices $d_r, \dots, d_j$). Si l'on examine tous les candidats k_r pour la racine $(avec\ i \leq r \leq j)$ et si l'on détermine tous les arbres binaires de recherche optimaux contentant les clés $k_i, \dots, k_{r-1}$ et ceux contenant les clés $k_{r+1}, \dots, k_j$, alors on est certain de trouver un arbre binaire de recherche optimal
Remarquons une petite subtilité. Supposons que dans le sous arbre de clés $k_i, k_{i+1}, \dots, k_j$, nous choisissons la cle k_i comme etant la racine. Alors, le sous arbre de gauche du nœud k_i contiendra les clés $k_i, \dots, k_{i-1}$ qui sera interprété comme n'ayant pas de cle réel, mais la cle factice d_{i-1}. Utilisant un raisonnement symmetrique, si nous choisissons k_j comme etant la racine du sous arbre, alors son sous arbre de droite n'aura que la clé factice d_j

b. Solution récursive

Notre sous problème est de chercher l'ABRO ayant les clés $k_i, \dots, k_j$ avec $i \geq 1, j \leq n$ et $j \geq i - 1$. Pour ce faire, nous considérons $e[i, j]$ comme étant le cout moyen de recherche dans l'ABRO contenant les clés $k_i, \dots, k_j$. Ce que nous voulons a la fin c'est de trouver $e[1, n]$, pour le cas general a n nœud.
Le cas simple a lieu lorsque $j = i - 1$, et dans ce cas, nous avons juste la cle factice d_{i-1} le cout moyen minimal de recherche de l'ABR est $e[i, i-1] = q_{i-1}$
Lorsque $j > i - 1$, nous choisissons la racine k_r dans les clés $k_i, \dots, k_j$ et construisons l'ABRO contenant les clés $k_i, \dots, k_{r-1}$ comme sous arbre de gauche et les clés $k_{r+1}, \dots, k_j$ comme sous arbre de droite
Une subtilité a remarquer est que, lorsque l'arbre en question (contenant $k_i, \dots, k_j$ comme clés et k_r comme racine) est lui-même sous arbre d'un nœud, la profondeur de chaque nœud de $k_i\ a\ k_j$ augmente de 1, et le cout moyen de recherche augmente naturellement de la somme de toutes les probabilités des nœuds de l'arbre. Dans la suite, nous dénotons cette somme $w(i, j)$ avec

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

Ainsi, si k_r est la racine de l'ABRO contenant les clés $k_i, \dots, k_j$, alors

$$e[i,j] = p_r + \big(e[i,r-1] + w(i,r-1)\big) + \big(e[r+1,j] + w(r+1,j)\big)$$

$$= e[i,r-1] + e[r+1,j] + w(i,j)$$

$$car\ w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

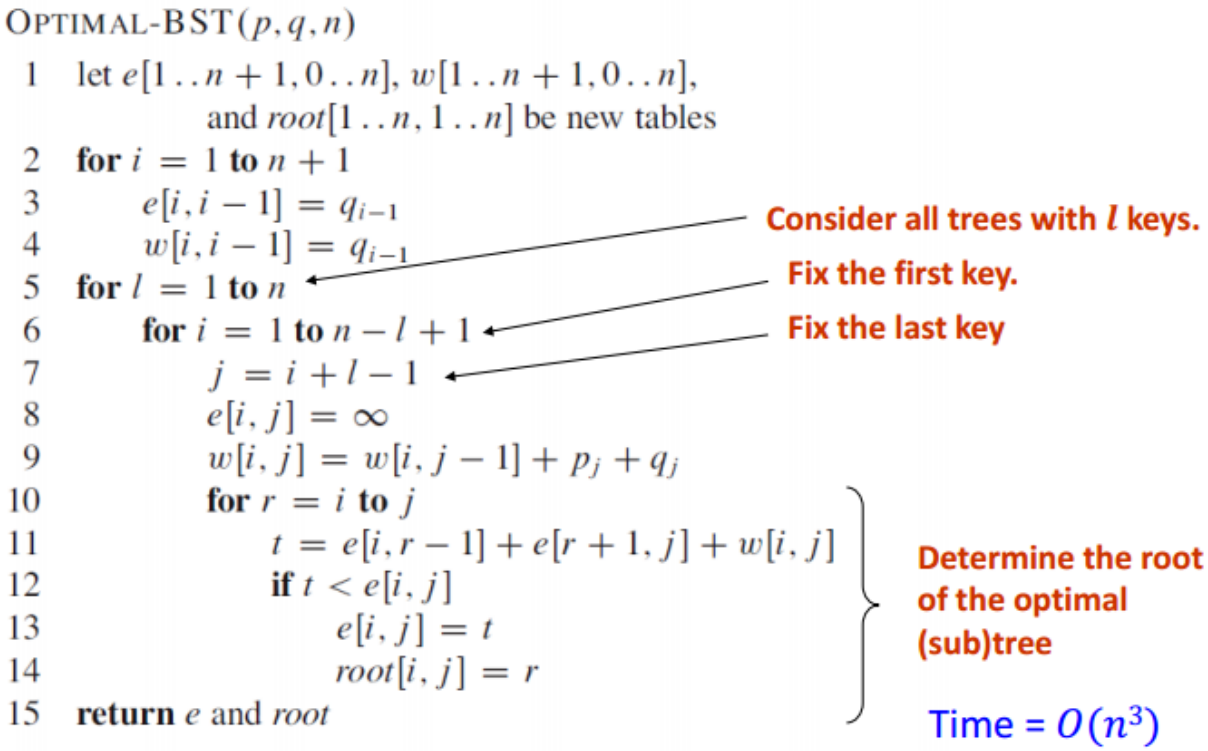
Puisque nous cherchons à construire un arbre optimal, nous choisissons la racine k_r pour laquelle $e[i,j]$ est minimal. Ainsi, nous avons

$$e[i,j] = \begin{cases} q_{i-1} \quad pour\ j = i-1 \\ \min\limits_{i \le r \le j} e[i,r-1] + e[r+1,j] + w(i,j)\ \ pour\ i \le j \end{cases}$$

c. Calcul du cout de recherche moyen dans un arbre binaire de recherche optimal

Les valeurs $e[i,j]$ donnent les coûts de recherche moyens dans des arbres binaires de recherche optimaux. Pour nous aider à gérer la structure des arbres binaires de recherche optimaux, définissons $racine[i,j]$, pour $1 \le i \le j \le n$, comme étant l'indice *r* pour lequel k_r est la racine d'un arbre binaire de recherche optimal contenant les clés $k_i, \dots, k_j$
Pour tout sous problème (i,j), on stocke les valeurs des couts moyen de recherche dans la table $e[1..n+1, 0..n]$ (On n'utilise que les entrées $e[i,j]$ de la table pour lesquelles $j \ge i-1$), on stocke aussi les racines des sous arbre a cles $k_i, \dots, k_j$ dans la table $racine[i,j]$ et enfin on utilise la table $w[1..n+1, 0 \dots n] = somme\ des\ probabilites$: $w[i,i-1] = q_{i-1}$ pour $1 \le i \le n+1$ et $w[i,j] = w[i,j-1] + p_j + q_j$ pour $1 \le i \le j \le n$. Remarquons que les valeurs de w sont stockées dans le tableau $w[1..n+1, 0 \dots n]$ juste à des fins d'efficacité ; Au lieu d'évaluer $w(i,j)$ de rien chaque fois que nous evaluons $e[i,j] = \min\limits_{i \le r \le j} e[i,r-1] + e[r+1,j] + w(i,j)\ \ pour\ i \le j$, on garde ces valeurs dans une table.

Nous avons donc le pseudo code qui prend en entrée les probabilités des clés réelles et des clés factices et le nombre de nœuds de l'arbre que nous souhaitons construire, et retourne les couts e et les racines de chaque sous arbre considéré



Ainsi, pour les probabilités et les clés de l'exemple précédent, le résultat de l'exécution du présent algorithme donne les tableaux (Que nous verrons comment remplir) de $e[i,j], r[i,j] et\ w[i,j]$ ci-dessous.
On a fait tourner les tableaux pour rendre horizontales les diagonales.

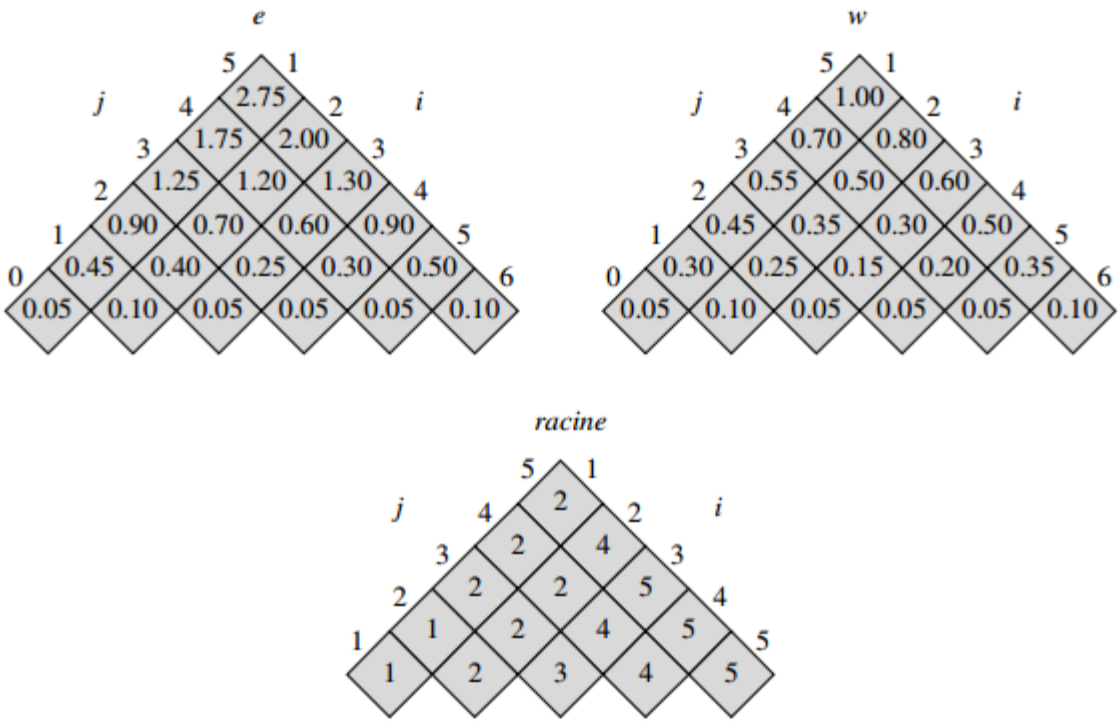


Figure 12 : Données assorties de l'exécution de l'algorithme OPTIMAL-BST avec les données de la figure 10

Et à partir de ces tableaux, nous avons l'Arbre Binaire de Recherche Optimal suivant :

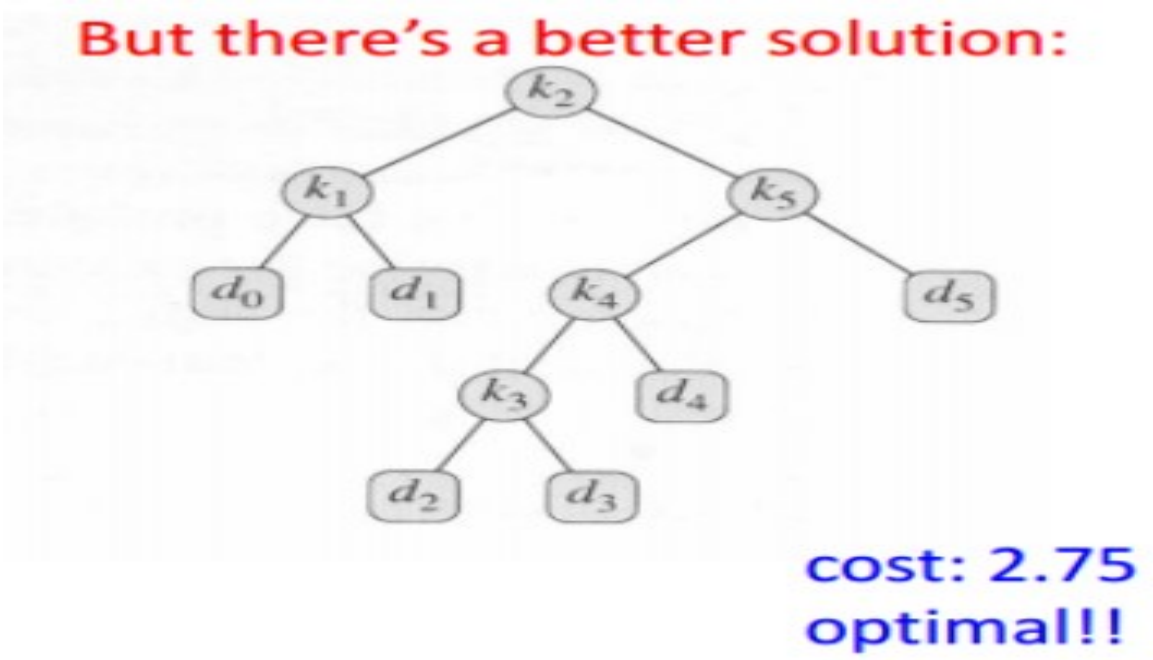


Figure 13 : Arbre binaire de recherche optimal construit à partir de la figure 12

A cause de l'imbrication a trois niveaux de la boucle for, la complexite de l'algorithme OBTIMAL-BST est de $O(n^3)$

CONCLUSION

Au terme de notre travail sur les arbres binaires optimaux et les arbres rouge-noir, ou il était question de montrer comment les arbres binaires de recherche sont un excellent outil ou structure de données en ce qui concerne les opérations élémentaires de recherche et de suppression, il n'en demeure pas moins que ces arbres présentent un léger inconvénient dans le nombre de fois qu'on doit accéder au disque lorsque les données à représenter par l'arbre sont volumineuses. Une généralisation des arbres binaires de recherche est donc introduite pour palier a ce problème : les B-arbres, que nous verrons avec le prochain groupe d'expose.

BIBLIOGRAPHIE

- **INTRODUCTION A L'ALGORITHMIQUE**, *Thomas Cormen* Professeur d'informatique au Darmouth College, *Charles Leiserson, Ronald Rivest et Clifford Stein* Professeurs d'informatique au MIT.
- ***ALGORITHMIQUE AVANCEE***, Frederic Vivien, IUP 2, 24 Avril 2002

WEBOGRAPHIE

- https://fr.wikipedia.org/wiki/Arbre_bicolore
- https:/www.cs.princeton.edu/~rs/talks/LLRB/RedBlack