Table des matières

LISTE DES TABLEAUX ET FIGURES

INTRODUCTION

De nos jours, on rencontre de nombreuses arborescences telles que l'arbre généalogique, table des matières d'un livre, les dossiers d'un ordinateur, l'organisation d'une entreprise, etc. De manière un peu plus formelle, les arbres binaires de recherche en abrégés (ABR), font partie de l'ensemble des structures de données telles que les enregistrements, les listes, les piles, les files, les tables, etc. Une structure de données étant elle-même une structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement. Elle est très utilisée en informatique, d'une part parce que les informations sont la plupart du temps structurées et hiérarchisées, et peuvent être représentées naturellement sous une forme arborescente, et d'autre part, parce que les structures de données arborescentes permettent de stocker des données volumineuses de façon à ce que leur accès soit simplifié. Pourquoi choisir les arbres comme structure de données pour la gestion de nos informations ? afin de mieux appréhender ce sujet, nous présenterons en primo les arbres binaires, en secundo les arbres rouge et noir et en tertio les arbres binaires de recherche optimaux.

I. Les 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 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 (niveau 0) il y a 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.

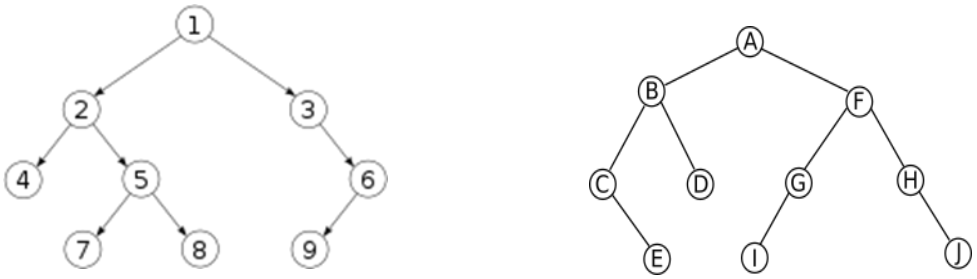


Figure 1: Exemple arbre

1. Types d'arbre binaire

- **Arbre binaire strict ou localement complet** est un arbre dont tous les nœuds possèdent zéro ou deux fils.
- **Arbre binaire dégénéré** est un arbre dans lequel tous les nœuds internes n'ont qu'un seul fils. Ce type d'arbre n'a qu'une unique feuille et peut être vu comme une liste chaînée.
- **Arbre binaire parfait** est un arbre binaire strict dans lequel toutes les feuilles (nœuds n'ayant aucun fils) sont à la même distance de la racine (c'est-à-dire à la même profondeur). Il s'agit d'un arbre dont tous les niveaux sont remplis : où tous les nœuds internes ont deux fils et où tous les nœuds externes ont la même hauteur.
- **Arbre binaire complet ou presque complet,** à ne pas confondre avec localement complet (ci-dessus), est un arbre dans lequel tous les niveaux sont remplis à l'exception éventuelle du dernier, dans lequel les feuilles sont alignées à gauche. On peut le voir comme un arbre parfait dont le dernier niveau aurait été privé de certaines de ses feuilles en partant de la plus à droite. Le caractère éventuel est important : un arbre parfait est nécessairement presque complet tandis qu'un arbre presque complet peut être parfait.

2. Vocabulaire lié aux arbres

- Degré d'un nœud : est le nombre de fils que possède ce nœud.
- Taille d'un arbre : est le nombre de nœud interne qui compose l'arbre. C'est à dire le nombre de nœuds totaux moins le nombre de feuille de l'arbre.

- Profondeur d'un nœud : la distance en termes de nœud par rapport à l'origine y compris le nœud.
- La hauteur de l'arbre : est la profondeur à laquelle il faut descendre dans l'arbre pour trouver le nœud le plus loin de la racine.

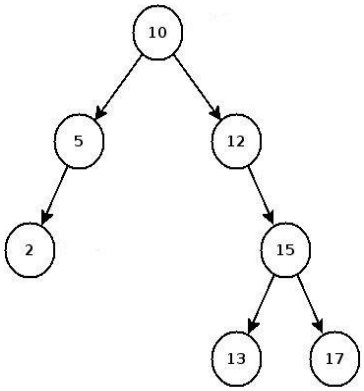


Figure 2: Arbre binaire de recherche

- ✓ {***2,5,10,12,13,15,17***} est l'ensemble de nœuds de l'arbre
- ✓ Le nœud ***10*** est appelé la racine de l'arbre ;
- ✓ Les nœuds ***2, 13*** et ***17*** sont appelés les feuilles de l'arbre.
- ✓ Le nœud contenant 5 a une profondeur égale à 2
- ✓ La hauteur d'un arbre est le plus long chemin qui mène de la racine à une feuille = 4.
- ✓ La taille de cet arbre est égale à 4.

3. Implémentation

Dans un arbre binaire un nœud est une structure qui contient au minimum trois champs, Un champ contenant l'élément du nœud, c'est l'information qui est importante. Cette information peut être un entier, une chaîne de caractère ou tout autre chose que l'on désire stocker. Les deux autres champs sont le fils gauche et le fils droit du nœud. Ces deux fils sont en fait des arbres, on les appelle généralement les sous arbres gauches et les sous arbres droit du nœud.

Tableau 1: Représentation d'un nœud.

Valeur	Ag	Ad

Donc d'après cette définition, un arbre ne pourra donc être qu'un pointeur sur un nœud. Voici donc une manière d'implémenter un arbre binaire

```
Type Arbre= pointeur( Noeud)
Noeud= structure
Valeur= TElement
Ag = Arbre ; {Fils gauche}
Ad = Arbre ; {Fils droit}
```

Pour pouvoir manipuler des arbres binaires, on doit donc disposer des fonctionnalités suivantes :

- Booléen : est_vide (AB a) : une fonction qui teste si un arbre est vide ou non.
- AB : fils_gauche (AB a) et AB : fils_droit (AB a), des fonctions qui retournent les fils gauche et droit d'un arbre non vide.
- CLE : val (AB a), une fonction qui retourne la valeur (clé) de la racine d'un arbre non vide.
- AB crée_arbre_vide (), une fonction qui crée un arbre vide.
- AB crée_arbre (CLE v, AB fg, fd), une fonction qui crée un arbre dont les fils gauche et droit soit fg et fd et dont la racine a pour valeur (clé) v.
- Change_val (CLE v, AB a), une fonction qui remplace la valeur de la racine de a par v, si a est non vide, et ne fait rien sinon.
- Change_fg (AB fg, AB a) et change_fd (AB fd, AB a), des fonctions qui remplacent le fils gauche de a (respectivement le fils droit de a) par fg (respectivement par fd), si a est non vide, et ne font rien sinon.

En C on utilise en généralement des pointeurs pour représenter les liens entre un nœud et ses fils dans un arbre binaire.

En Python, on peut représenter un arbre vide par une liste vide [] et un arbre non vide par une liste comprenant 3 éléments **[clé, fils gauche, fils droit].**

II. Les arbres binaires de recherche

Un arbre binaire de recherche (ABR) ou arbre binaire ordonné est un arbre binaire donc les éléments sont stockés à l'intérieur de chaque nœud (clé) de manière ordonnée, c'est à dire les clés de chaque nœud doivent être enregistrés selon un ordre croissant.

Donc lorsque le premier élément est inséré dans l'arbre il devient la racine. Ensuite, il suffit de mettre à gauche les éléments plus petits et à droite les éléments plus grands. C'est en effet cette particularité qui rend les ABR intéressants. Ce qui se traduit par : soit x un nœud dans un arbre binaire de recherche. Si y est un nœud dans le sous arbre gauche de x, alors clé(y) < clé(x). Si y est un nœud dans le sous arbre droit de x, alors clé(y)> clé(x).

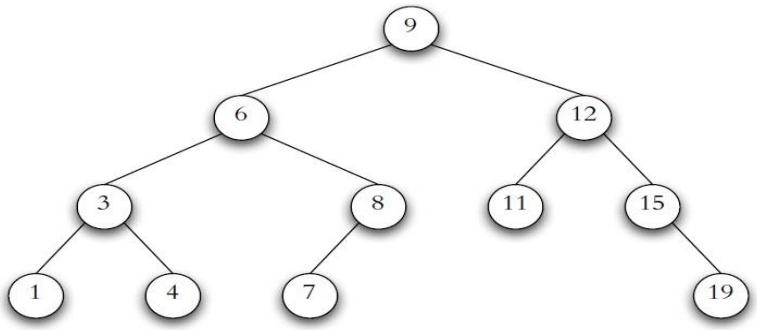


Figure 3: Arbre binaire de recherche

1. Différence entre un arbre binaire et un arbre binaire de recherche

Un arbre binaire: est une forme d'arbre spécialisée avec deux enfants (enfant gauche et enfant droit). C'est simplement une représentation des données dans la structure arborescente.

```
  1
 / \
2   3
```

Un arbre de recherche binaire (BST) ou "arbre binaire ordonné" est un type d'arbre binaire où les nœuds sont disposés dans l'ordre: pour chaque nœud, tous les éléments de son sous-arbre gauche sont inférieurs au nœud (<), et tous les éléments dans son sous-arbre droit sont supérieurs au nœud (>).

```
    5
   / \
  3   6
 / \   \
1   4   9
```

2. Parcours

Un parcours est un algorithme qui appelle une fonction, une méthode ou une procédure sur tous les nœuds d'un arbre.

L'affichage des éléments (clés) des nœuds d'un arbre binaire peut se faire dans un ordre préfixe, infixe ou Postfixe.

Soit un arbre A de type Arbre, de racine (a) et ses deux fils gauche(a) et droite(a). Nous pouvons écrire les fonctions suivantes (remarquez la position de la ligne Afficher (a)) :

a. Parcours Préfixe

Ici l'on affiche la racine, puis le sous arbre gauche, puis le sous arbre droit

```
AfficherPréfixe(Arbre a) :
    Afficher(a)
    Si nonVide(gauche(a))
          AfficherPréfixe(gauche(a))
    Si nonVide(droite(a))
          AfficherPréfixe(droite(a))
```

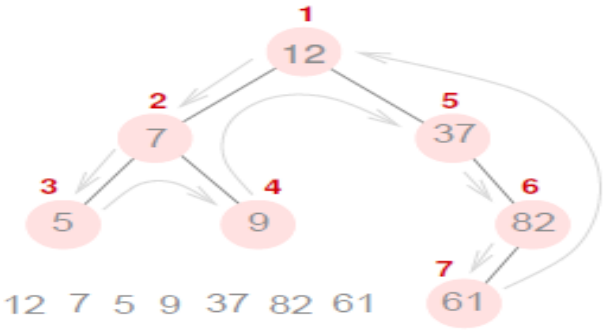


Figure 4: Parcours préfixe

b. Parcours infixe

Ici l'on affiche les nœuds dans l'ordre, puisqu'on affiche d'abord le sous arbre gauche dont les valeurs sont inferieures puis lui-même (la racine), et enfin le sous arbre droit qui contient les nœuds supérieurs.

```
AfficherInfixe(Arbre a) :
    Si nonVide(gauche(a))
       AfficherInfixe(gauche(a))
    Afficher (a)
    Si nonVide(droite(a))
       AfficherInfixe(droite(a))
```

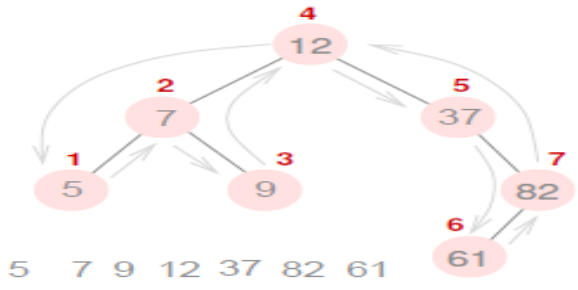


Figure 5:: Parcours infixe

c. Parcours postfixe ou suffixe

Ici on affiche le sous arbre gauche, puis le sous arbre droit, puis la racine.

```
AfficherPostfixe(Arbre a) :
    Si nonVide(gauche(a))
       AfficherPostfixe(gauche(a))
    Si nonVide(droite(a))
       AfficherPostfixe(droite(a))
    Afficher (a)
```

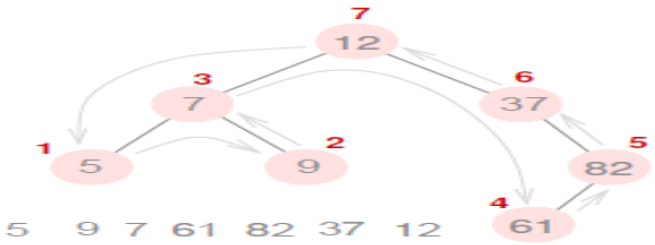


Figure 6: Parcours postfixe

d. Parcours en profondeur

Avec ce parcours, nous tentons toujours de visiter le nœud le plus éloigné de la racine que nous pouvons, à la condition qu'il soit le fils d'un nœud que nous avons déjà visité. À l'opposé des parcours en profondeur sur les graphes, il n'est pas nécessaire de connaître les nœuds déjà visités, car un arbre ne contient pas de cycles. Les parcours préfixe, infixe et postfixe sont des cas particuliers de ce type de parcours.

Pour effectuer ce parcours, il est nécessaire de conserver une liste des éléments en attente de traitement. Dans le cas du parcours en profondeur, il faut que cette liste ait une structure de pile (de type LIFO : dernier entré, premier sorti). Dans cette implémentation, on choisira également d'ajouter les fils d'un nœud de droite à gauche.

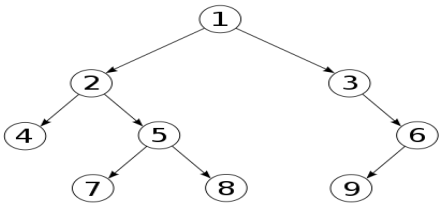
```
ParcoursProfondeur(Arbre a) {
   s = Pilevide
   ajouter(Racine(a), s)
   Tant que (s != Pilevide) {
       nœud = premier(s)
       retirer(s)
       Visiter(nœud) //On choisit de faire une opération
       Si (droite(nœud) != NIL) Alors
           ajouter(droite(nœud), s)
       Si (gauche(nœud) != NIL) Alors
           ajouter(gauche(nœud), s)
   }
}
```

e. Parcours en largeur

Contrairement au précédent, ce parcours essaie toujours de visiter le nœud le plus proche de la racine qui n'a pas déjà été visité. En suivant ce parcours, on va d'abord visiter la racine, puis les nœuds à la profondeur 1, puis 2, etc. D'où le nom parcours en largeur.

L'implémentation est quasiment identique à celle du parcours en profondeur à ce détail près qu'on doit cette fois utiliser une structure de file d'attente (de type FIFO : premier entré, premier sorti), ce qui induit que l'ordre de sortie n'est pas le même (i.e. on permute gauche et droite dans notre traitement).

```
ParcoursLargeur(Arbre a) {
   f = FileVide
   enfiler(Racine(a), f)
   Tant que (f != FileVide) {
       nœud = defiler(f)
       Visiter(nœud)              //On choisit de faire une opération
       Si (gauche(nœud) != NIL) Alors
           enfiler(gauche(nœud), f)
       Si (droite(nœud) != NIL) Alors
           enfiler(droite(nœud), f)
   }
}
```



Rendu du parcours en largeur : 1, 2, 3, 4, 5, 6, 7, 8, 9

Figure 7: Parcours en largeur

3. Opérations sur les arbres binaires de recherche

Les opérations caractéristiques sur les arbres binaires de recherche : sont l'insertion, la suppression, et la recherche d'une valeur. Ces opérations sont peu couteuses si l'arbre n'est pas trop grand en hauteur.

a. Etude de cas :

Soit E un ensemble muni d'une relation d'ordre, et soit A un arbre binaire portant des valeurs de E. L'arbre A est un arbre binaire de recherche si pour tout nœud p de A, la valeur de p est strictement plus grande que les valeurs se trouvant dans son sous arbre gauche et strictement plus petite que les valeurs se trouvant dans son sous arbre droit. Cette définition suppose donc qu'une valeur n'apparait au plus qu'une seule fois dans un arbre de recherche.

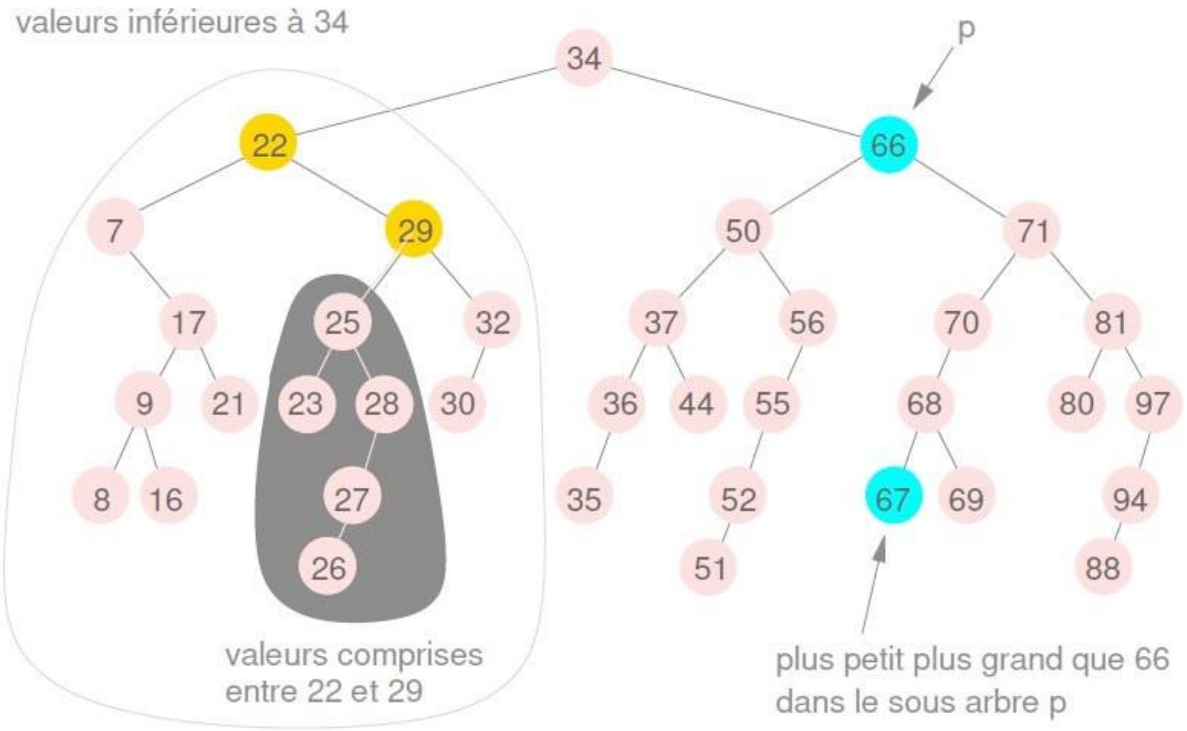


Figure 8: Etude de cas

Pour accéder à la clé la plus petite (respectivement la plus grande) dans un ABR il suffit de descendre sur le fils gauche (respectivement sur le fils droit) autant que possible. Le dernier nœud visité qui n'a pas de fils gauche (respectivement droit), porte la valeur la plus petite (respectivement la plus grande) de l'arbre. Les algorithmes de recherche du minimum, du maximum et du successeur sont les suivants :

Minimum

```
ARBRE-MINIMUM(x)
tant que gauche(x) != NIL
faire
        x = gauche(x) ;
renvoyer x ;
```

Maximum

```
ARBRE-MAXIMUM(x)
tant que droit(x) != NIL
faire
        x = droit(x) ;
renvoyer x ;
```

Successeur

```
ARBRE-SUCCESSEUR(x)
   si droit(x) != null alors
        renvoyer ARBRE-
MINIMUM(droit(x))
   y = père(x)
   tant que y != NIL et x = droit(y) faire
   x =  y
   y = père(x)
   renvoyer y
```

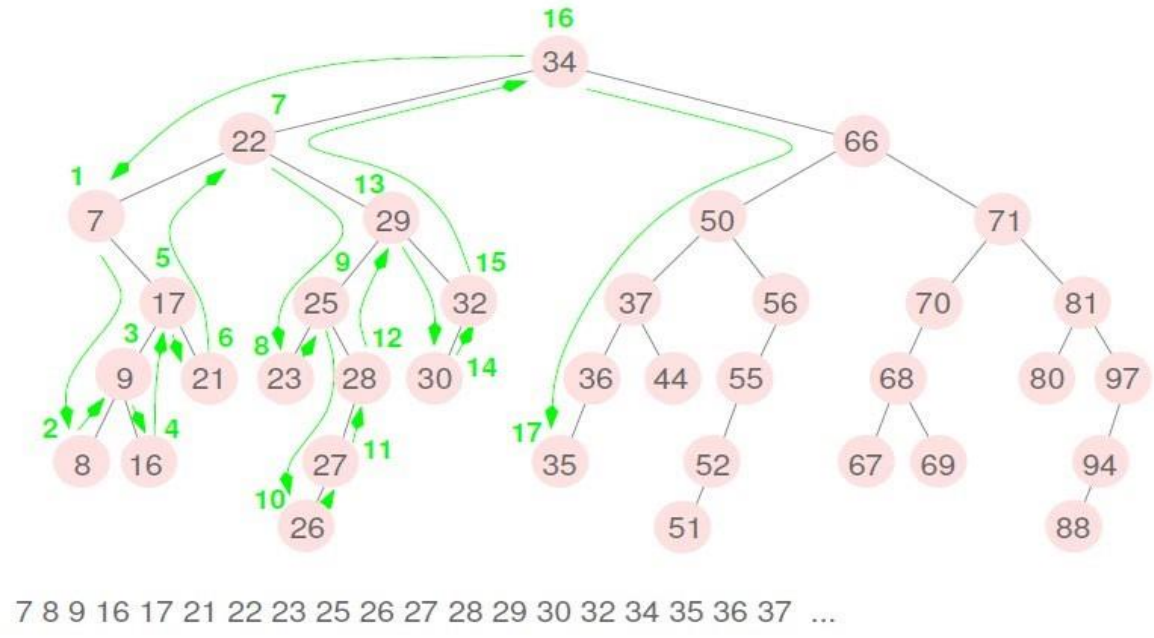


Figure 9: Etude de cas parcours infixe

b. Recherche d'une valeur

La recherche d'une valeur dans un ABR consiste à parcourir une branche en partant de la racine, en descendant chaque fois sur le fils gauche ou sur le fils droit, suivant que la clé portée par le nœud est plus grande ou plus petite que la valeur cherchée. La recherche s'arrête dès que la valeur est rencontrée ou que l'on a atteint l'extrémité d'une branche (le fils sur lequel il aurait fallu descendre n'existe pas).

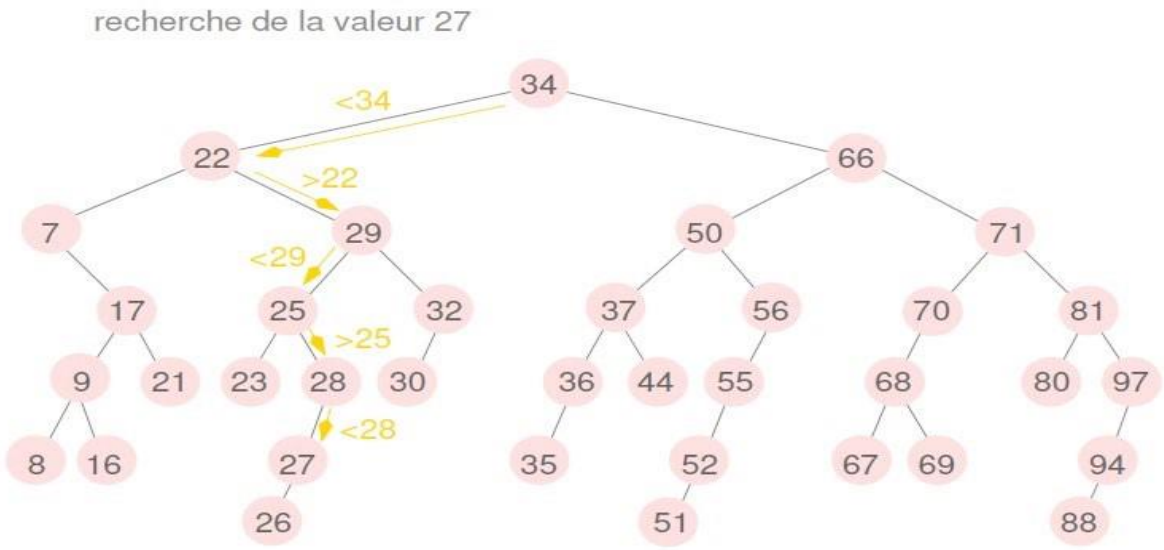


Figure 10: Recherche d'une valeur

Fonction recherche(a, v) : version itérative

entrée : a est un ABR, v est une clé.
sortie : **Vrai** si v figure dans a et **Faux** sinon.
début
  tant que *NON est_vide(a) ET* $v \neq val(a)$ **faire**
    si $v < val(a)$ **alors**
      a = fils_gauche(a)
    sinon
      a = fils_droit(a)
    finsi
  fintq
  si *est_vide(a)* **alors**
    retourner Faux
  sinon
    retourner Vrai
  finsi
fin

c. Insertion d'une nouvelle valeur

Le principe est le même que pour la recherche. Un nouveau nœud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arrêtée.

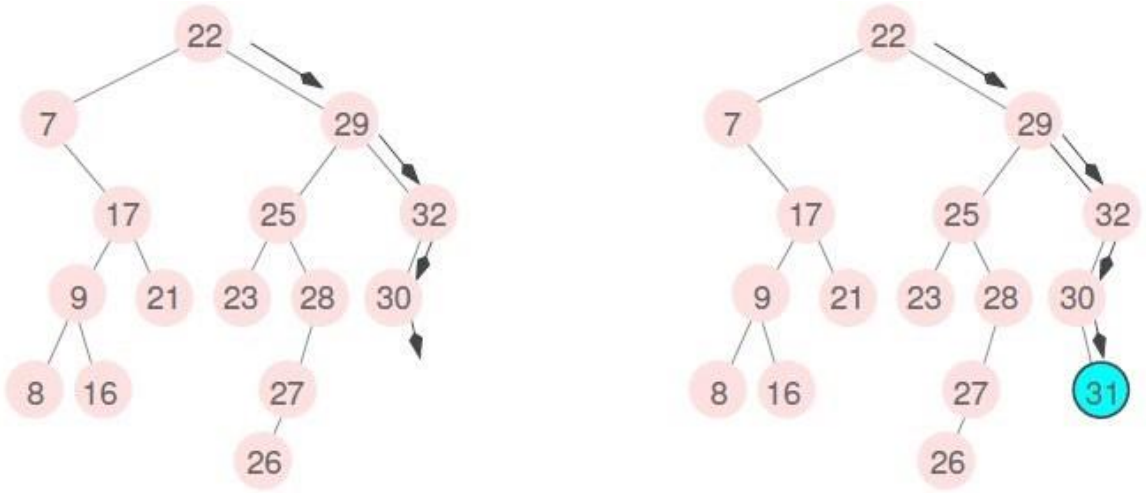


Figure 11: insérer une valeur

Algorithme Insertion: Insertion d'une nouvelle clé dans un ABR

```
entrée : a est un ABR, v est une clé.
résultat : v est insérée dans a
début
    si est_vide(a) alors
        a=cree_arbre(v,cree_arbre_vide(),cree_arbre_vide())
    sinon
        si v < val(a) alors
            Insertion(v,fils_gauche(a))
        sinon
            si v > val(a) alors
                Insertion(v,fils_droit(a))
            finsi
        finsi
    finsi
fin
```

d. Suppression d'un nœud

L'opération dépend du nombre de fils du nœud à supprimer

- **Cas 1 :** le nœud à supprimer n'a pas de fils, c'est une feuille. Il suffit de décrocher le nœud de l'arbre, c'est-à-dire de l'enlever en modifiant le lien du père, s'il existe, vers ce fils. Si le père n'existe pas l'arbre devient l'arbre vide.

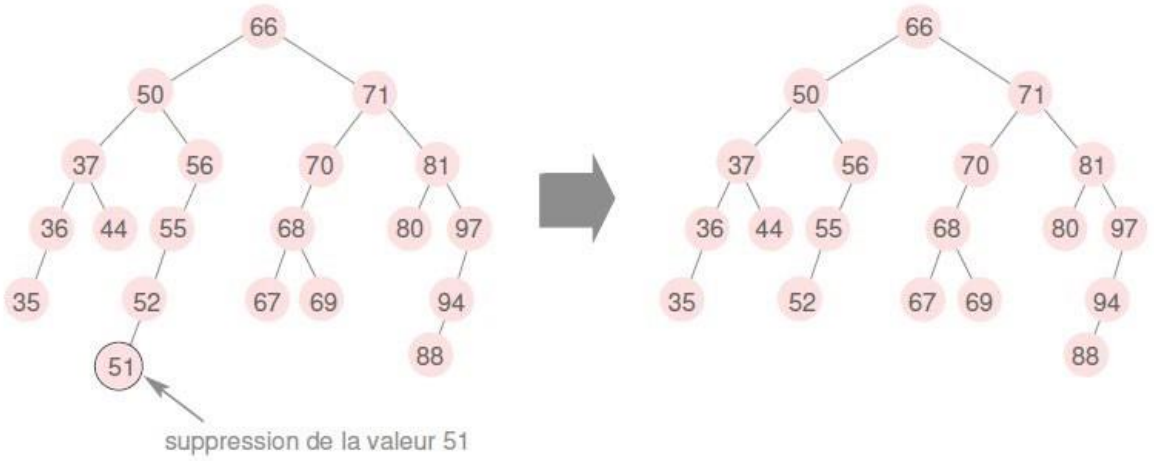


Figure 12: Supprimer la valeur 51

- **Cas 2 :** le nœud à supprimer à un fils et un seul. Le nœud est décroché de l'arbre comme dans le cas 1. Il est remplacé par son fils unique dans le nœud père, si ce père existe. Sinon l'arbre est réduit au fils unique du nœud supprimé.

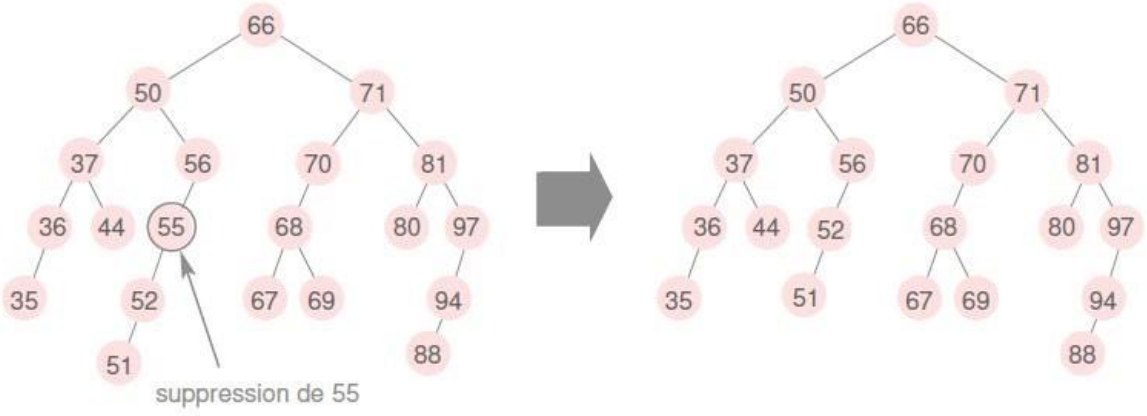


Figure 13: supprimer la valeur 55

- **Cas 3 :** le nœud à supprimer p a deux fils. Soit q le nœud de son sous arbre droit qui a la valeur la plus petite. Il suffit de recopier la valeur de q dans le nœud p et de décrocher le nœud q puisque le nœud q a la valeur la plus petite dans le fils droit.

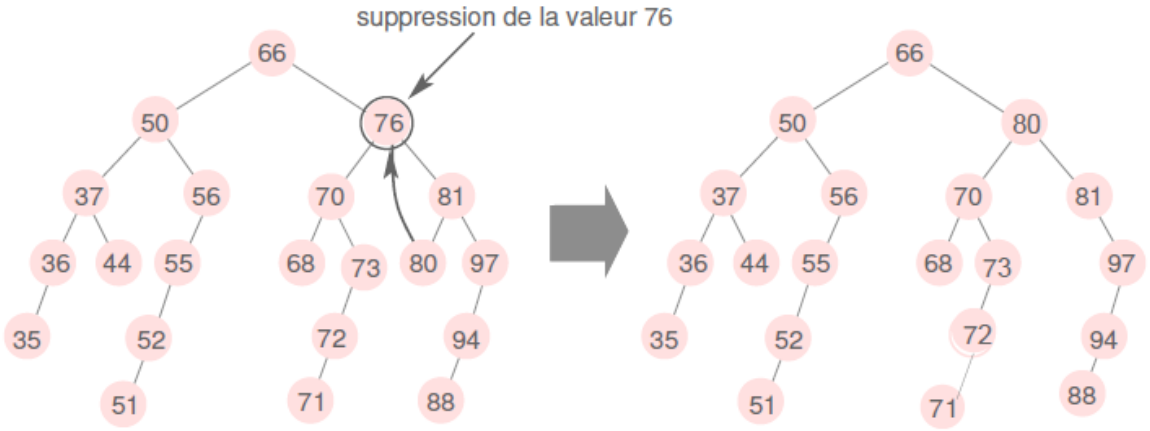


Figure 14: Supprimer la valeur 76

4. Notion d'équilibre d'un arbre binaire de recherche

Les arbres binaires de recherche présentent des avantages que nous avons vu plus haut, mais ils présentent aussi des inconvénients tels que le déséquilibre, etc. Prénons par exemple le cas d'un arbre dans lequel on insère successivement les valeurs 1, 2, 3, 4 et 5.

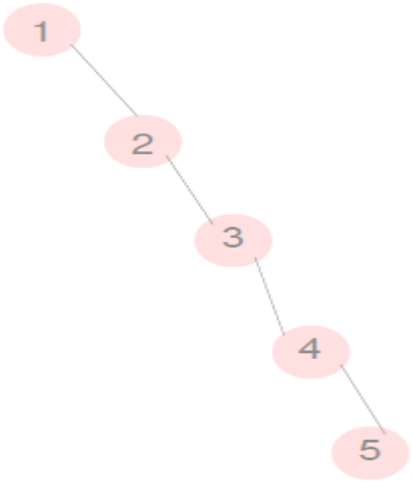


Figure 15: arbre non équilibré

Dans l'exemple ci-dessus l'on peut voir que l'on perd tout l'intérêt des arbres puisque l'on est revenu à une structure de données (une liste chainée) qui ne bénéficie pas des avantages des arbres. De manière plus générale on parle de dégénérescence en liste d'un arbre qui est un cas extrême de déséquilibre d'un arbre.

Un arbre est équilibré à partir du moment où la différence entre la hauteur de son sous arbre gauche et la hauteur de son sous arbre droit appelée balance est comprise entre -1 et 1.

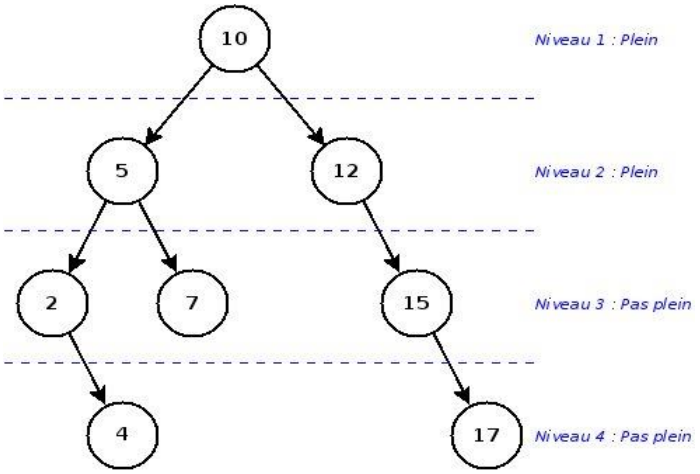


Figure 16:arbre binaire de recherche déséquilibré

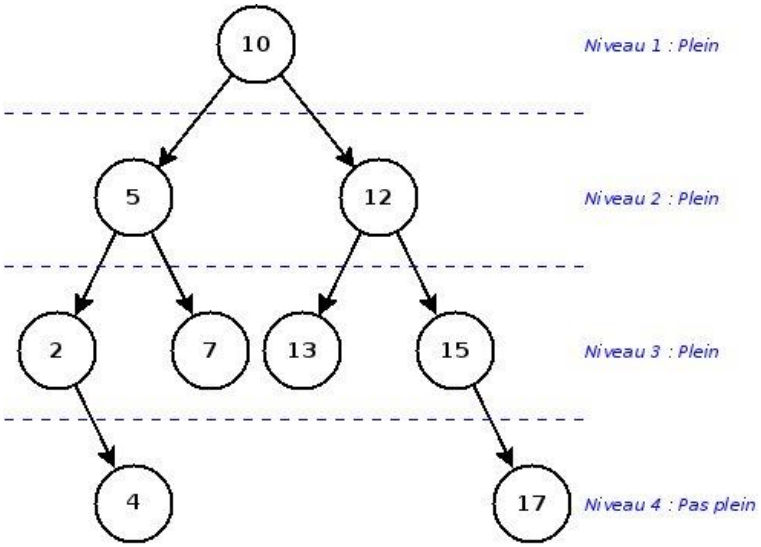


Figure 17: arbre binaire de recherche équilibré

Le déséquilibre d'un arbre provoque un rallongement du temps moyen des opérations, donc pour éviter cette situation on utilise des arbres binaires de recherche évolués tels que les arbres AVL, les arbres bicolores (Rouge et Noir), les arbres-tas qui se basent sur l'opération de rotation pour s'auto-équilibrer. Nous étudierons ici les arbres bicolores (Rouge et Noir).

III. Les Arbres rouges et noirs

1. Présentation des arbres rouges et noirs

a. Définition

Un Arbre rouge et noir ou Arbre rouge-noir ou Arbre bicolore est un type particulier d'arbre binaire de recherche équilibré, qui est une structure de données couramment utilisée en informatique pour organiser des données pouvant être comparées, par exemple des nombres ou des chaines de caractère. Les feuilles de l'arbre, c'est-à-dire les nœuds terminaux, ne contiennent aucune donnée et sont simplement représentées sans coût mémoire par les éléments **Nuls** (Pointeur nul en C, valeur NIL).

Comme tous les arbres binaires de recherche, les arbres bicolores peuvent être parcourus très efficacement en ordre infixe (ou ordre gauche - racine - droite), ce qui permet de lister les éléments dans l'ordre. La recherche d'un élément se fait en temps logarithmique $O(\log n)$, n étant le nombre d'éléments de l'arbre, y compris dans le pire des cas.

Les arbres rouges et noirs 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 alors utilisables dans des 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, en programmation fonctionnelle. Cette dernière est l'exemple le plus couramment utilisé de structure de données persistante qui peut être utilisée pour construire des tableaux associatifs capables de garder en mémoires les versions précédentes après un changement.

b. Propriétés

Un arbre rouge et noir est un arbre binaire de recherche dans lequel chaque nœud a un attribut supplémentaire : sa couleur, qui est soit rouge soit noire. Elle satisfait les cinq propriétés suivantes :

1. Propriété de couleur : chaque nœud est soit rouge, soit noir.
2. Propriété de racine : La racine est noire.
3. Propriété interne : si un nœud est rouge, alors ses deux fils sont noirs.
4. Propriété externe : chaque feuille (Nil) est noire.

5. Propriété de profondeur : pour tout nœud de l'arbre, les chemins de ce nœud vers les feuilles ont la même profondeur noire, qui est définie comme étant le nombre de nœud noir sur chaque branche.

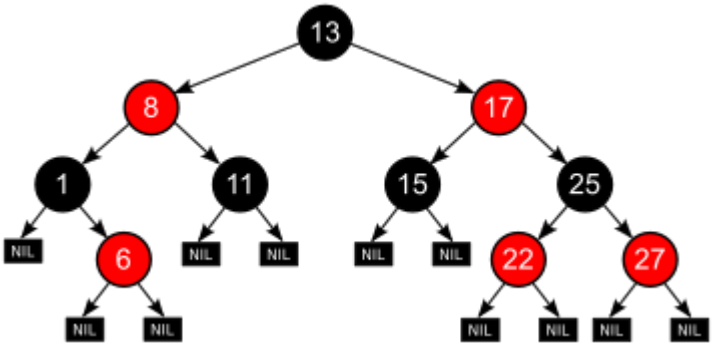


Figure 18: Exemple d'arbre rouge et noir de hauteur noire= 2

Pour comprendre comment ces contraintes garantissent la propriété ci-dessus, il suffit de s'apercevoir qu'aucun chemin ne peut avoir deux nœuds rouges consécutifs à cause de la propriété 3.

Remarque : La propriété 2 n'est pas nécessaire. Les seuls cas où la racine pourrait devenir rouge étant les deux cas où sa couleur n'a pas d'importance : soit la racine est le seul nœud, soit elle possède deux fils noirs.

1. Hauteur Noire

La hauteur noire d'un nœud x (noté hn(x)) dans un arbre rouge et noir est le nombre (unique) de nœud noir sur tout chemin du nœud à une feuille de l'arbre, le nœud x étant exclu.

Exemple : la hauteur noire des nœuds de notre figure précédente :

Tableau 2:Exemple de hauteur noire

Hn	x	1	8	11	13	17	14	25	27
Valeurs	2	0	1	0	1	1	0	0	0

Montrons que les arbres rouge et noir sont équilibrés

Propriété 1 : $hn(x) \geq \frac{h(x)-1}{2}$

Démonstration:

- Cas où nous avons le plus de nœud noir possible $hn(x) \leq h(x)$.
- Cas où nous avons le moins de nœud possible, nous avons une racine noire d'après la propriété 5 nous considérons un seul chemin et d'après la propriété 3 un nœud rouge a un nœud noir comme fils, puis le nœud noir a un fils rouge comme fils, ainsi de suite jusqu'à la fin où nous avons un nœud rouge. La hauteur **h(x)** est donc le nombre de nœud total utilisé et la hauteur noir **hn(x)** le nombre de nœud noir utilisé.

Quand nous avons plus que de nœuds rouges que de nœuds noirs, on a la relation dans la situation extrême : $2hn(x) + 1 \geq h(x)$ qui nous donne $\boldsymbol{hn(x) \geq \frac{h(x)-1}{2}}$

Propriété 2 : Dans un arbre rouge et noir à n nœuds et de racine r, $n \geq 2^{hn(r)}$

Démonstration : par récurrence, sur la hauteur de l'arbre rouge et noir.

- Cas de base : h=0 (arbre réduit à sa racine), n=1, hn(r) =0, d'où $1 \geq 2^0$
- Cas général : on suppose $n \geq 2^{hn(r)}$ vérifié pour tous les arbres rouges et noirs de hauteur $\leq h$ et on considère un arbre rouge et noir de hauteur h + 1.
  - Si $hn(r) = 0$ alors trivialement, $n \geq 2^0 = 1$ puisque l'arbre a au moins un nœud.
  - On suppose $hn(r) > 0$ cela signifie que la racine a deux enfants (sinon, il y aura violation de la propriété 5 des arbres rouges et noirs). Ces enfants sont racine d'arbre de hauteur noire $\geq hn(r) - 1$. $n \geq 2\ X\ 2^{hn(r)-1} + 1\ \geq 2^{hn(r)}$

Théorème : les arbres rouges et noirs sont équilibrés.

Démonstration : $n \geq 2^{hn(r)} \geq 2^{\frac{h-1}{2.}}$

$\Leftrightarrow \log n\ \geq \frac{h-1}{2} \Leftrightarrow h \leq 2(\log n) + 1$

Autrement dit $\boldsymbol{h = O(\log n)}$

2. Opérations sur les arbres rouges et noirs

La recherche sur un arbre bicolore s'effectue exactement comme dans les arbres binaires de recherche. Cependant, après une insertion ou une suppression, les propriétés de l'arbre rouge et noir peuvent être violées. La restauration de ces propriétés requiert un petit nombre de modification des couleurs (qui sont très rapide en pratique) et pas plus de trois rotation (deux pour l'insertion).

L'ensemble des opérations sur un arbre binaire de recherche rouge et noir a une complexité de O(h), h étant la hauteur de l'arbre. Et ceci n'est valable que si l'arbre est équilibré donc h = O(log n)

a. Recherche d'un élément

La recherche d'un élément se déroule de la même façon que pour un arbre binaire de recherche : en partant de la racine, on compare la valeur recherchée à celle du nœud courant de l'arbre. Si ces valeurs sont égales, la recherche est terminée et on renvoie le nœud courant. Sinon, on choisit de descendre vers le nœud enfant gauche ou droit selon que la valeur recherchée est inférieure ou supérieure. Si une feuille est atteinte, la valeur recherchée ne se trouve pas dans l'arbre. La couleur des nœuds de l'arbre n'intervient pas directement dans la recherche.

b. Rotation

Sur un arbre binaire de recherche, nous pouvons effectuer deux types d'opérations de rotation : la rotation à gauche et la rotation à droite.

La rotation à Gauche consiste : à transformer la configuration des deux nœuds de Gauche pour aboutir à celle de Droite, en modifiant un nombre constant de pointeur. La configuration de Droite peut être transformée par celle de Gauche par l'opération inverse. Soit le schéma suivant où l'opération Rotation-Droite (T, y) permet de faire la rotation à droite et Rotation-Gauche (T, x) la rotation à gauche.

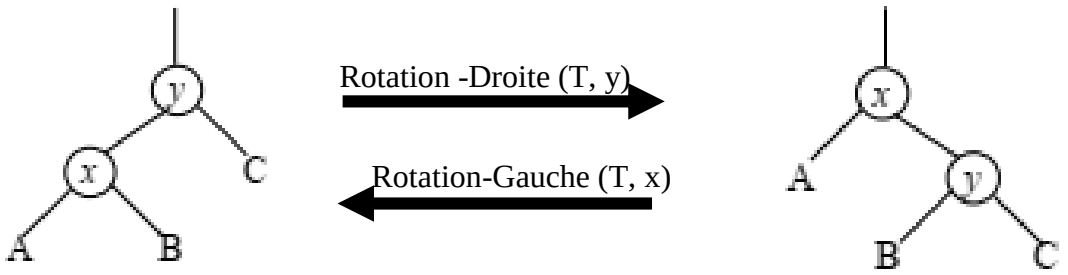


Figure 19: Rotation Gauche et Droite

Les rotations préservent la propriété des arbres binaires de recherche mais pas celle des arbres rouge et noir. Elles changent la structure de l'arbre mais en arrangeant les sous arbres. Son but est de diminuer la taille de l'arbre de plus, elles n'affectent pas l'ordre des éléments.

```
RotationGauche(T, x)
Début
   y = droit(x)
   droit(x) = gauche(y)
   Si gauche(y) != Nil Alors
         père(gauche(y)) = x
   FinSi
    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
       FinSi
    FinSi
   gauche(y) = x
   père(x) = y
Fin
```

c. Insertion

L'insertion commence de la même manière que sur un arbre binaire : en partant de la racine, on compare la valeur insérée à celle du nœud courant de l'arbre, et on choisit de descendre vers le nœud enfant gauche ou droit selon que la valeur insérée est inférieure ou supérieure. Le nouveau nœud est inséré lorsque l'on ne peut plus descendre, c'est-à-dire quand le nœud courant est une feuille de l'arbre. Cette feuille est remplacée par le nouveau nœud.

Une fois le nouveau nœud ajouté à l'arbre, il faut vérifier que les propriétés de l'arbre rouge et noir sont bien respectées et, dans le cas contraire, effectuer des opérations de changement de couleur et des rotations pour les rétablir. Le nœud inséré est initialement colorié en **rouge**. Il y a ensuite plusieurs cas possibles pour rétablir les propriétés de l'arbre, à partir du nœud inséré.

- **Cas 1 : Le nœud inséré n'a pas de parent.**

Il est en fait à la racine de l'arbre. La seule correction à apporter consiste à le colorier en **noir** pour respecter la propriété 2.

- **Cas 2 : Le parent du nœud inséré est noir.**

Alors l'arbre est valide : la propriété 3 est vérifiée, et la hauteur-noire de l'arbre est inchangée puisque le nouveau nœud est **rouge**. Il n'y a donc rien d'autre à faire.

- **Cas 3 : Le parent du nœud inséré est rouge.**

Alors la propriété 3 est invalide. L'action à effectuer dépend de la couleur de l'oncle du nœud inséré, c'est-à-dire le « frère » du parent du nœud inséré. En d'autres termes : en partant du nœud inséré (N), on considère son nœud parent (P), puis le nœud parent de P, ou grand-parent (G), et enfin l'oncle (U) qui est le fils de G qui n'est pas P. Si l'oncle est **rouge**, alors le parent et l'oncle sont coloriés en **noir**, et le grand-parent (qui était nécessairement noir) est colorié en **rouge**. Ce changement de couleur a pu toutefois créer une nouvelle violation des propriétés rouge et noir (propriété 3) plus haut dans l'arbre. Il faut maintenant recommencer la même analyse de cas mais cette fois en partant du nœud grand-parent ainsi colorié en rouge.

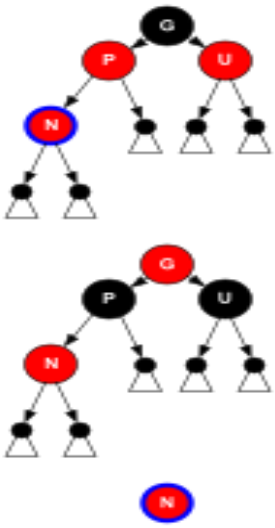


Figure 20: Insertion cas 3 le parent du nœud inséré est rouge

- **Cas 4 : Dans le cas où l'oncle est noir.**

Ici aussi la propriété 3 est violé, il faut effectuer des rotations qui dépendent de la configuration du nœud inséré autour de son parent et de son grand-parent, afin de ramener l'équilibre dans l'arbre.

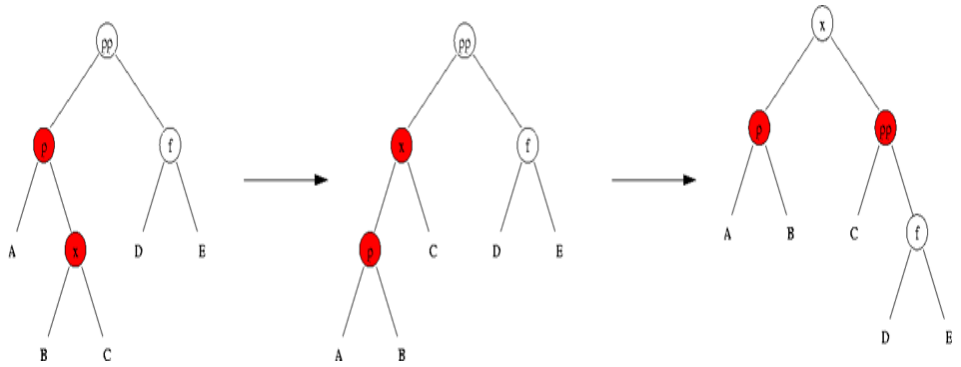


Figure 21: Insertion cas 4 l'oncle est noir

- **Cas 5 : Le parent vient prendre la place du grand-parent et le grand-parent celle de l'oncle.**

Le parent devient **noir** et le grand-parent **rouge** et l'arbre respecte alors les propriétés rouge et noir.

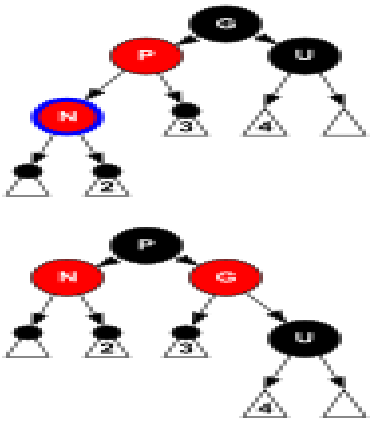


Figure 22: : Insertion cas 5 le parent vient prendre la place du grand-parent

Le seul cas où la correction ne se termine pas immédiatement est le cas 3, dans lequel on change le grand-parent de noir à rouge, ce qui oblige à effectuer une nouvelle vérification en partant du grand-parent. Cependant, il est aisé de vérifier que la fonction se termine toujours. Puisque le nœud à vérifier est toujours strictement plus haut que le précédent, on finira inévitablement par se retrouver dans l'un des cas non récursifs (dans le pire des cas, on remontera jusqu'à atteindre la racine de l'arbre, c'est-à-dire le cas 1). Il y aura donc aux plus deux rotations, et un nombre de changements de couleurs inférieur à la moitié de la hauteur de l'arbre. En pratique la probabilité de tomber plusieurs fois de suite sur le cas 3 est exponentiellement décroissante ; en moyenne le coût de la correction des propriétés est donc presque constant.

```
ROUGENOIR-INSERTION(T ,z)
        père(x) = NIL
        x = racine(T)
        Tant que x != NIL faire
                Père(x) = x
                Si clef(z)) < clef(x) alors
                        x = gauche(x)
                        x = droite(x)
        père(z) = père(x)
        Si père(x) == NIL alors
                racine(T) = z
        Sinon Si clef(z) < clef(x) alors
                Gauche(père(x) = z
        Sinon
                Droit(père(x) = z
```

```
ROUGENOIR-CORRECTION(T , x)
        ROUGENOIR-INSERTION(T , x)
        couleur(x) = Rouge
        Tant que x != racine(T) et couleur(père(x)) == Rouge faire
                Si père(x) == gauche(grand-père(x)) alors
                        y = droit(grand-père(x))
                        Si couleur(y) == Rouge alors
                                couleur(père(x)) = Noir
                                couleur(y) = Noir
                                couleur(grand-père(x)) = Rouge
                                x = grand-père(x)
                        Sinon Si x == droit(père(x)) alors
                                x = père(x)
                               RotationGauche(T, x)
                           couleur(père(x)) = Noir
                           couleur(grand-père(x)) = Rouge
                           RotationDroite (T , grand-père(x))
                Sinon (même chose que précédemment en échangeant droit et gauche)
        couleur(racine(T )) = Noir
```

La complexité de l'algorithme d'insertion se fait à **O(log n)** étant donné que c'est un arbre équilibre, démontré plus haut.

d. Suppression

Tout comme l'insertion d'une valeur, la suppression dans un arbre rouge et noir peut créer un viol de propriété dans un arbre rouge et noir et pour palier a ce viol nous pouvons procéder par une correction. Alors la suppression aimerait que nous respections certaines conditions de suppression et ces conditions sont les mêmes pour un arbre binaire de recherche

- Conditions de suppressions,

  Soit x le nœud qu'on désire supprimer

  - Si x est une feuille on la supprime
  - Si x est un nœud noir il faudra corriger les couleurs à l'ancien emplacement de x ce qui peut conduire à un déséquilibre de l'arbre.
  - Si x n'était pas une feuille mais avait un seul enfant(z), on remplace x par z.
    - ✓ Si x était un nœud (ce qui peut causer un problème de déséquilibre) alors il faudrait corriger les couleurs au niveau z
    - ✓ Si x a deux enfants, on recherche dans le sous arbre de x (sous arbre de l'enfant droit) l'élément minimal z.
    - ✓ Si z est de couleur noir, il faudra corriger les couleurs au niveau de l'emplacement actuelle de z. et on remplace l'enfant z par son enfant droit s'il en a et en fin on remplace x par z en concevant la couleur de z.

- Correction des couleurs

  S'il faut corriger une couleur au niveau d'un emplacement x cela veut dire qu'il manquait une couleur noire au niveau de x. soit x le nœud qu'on souhaite corriger :

  - Si x est rouge alors il devient noir
  - Si x c'est la racine on ne fait rien on a fini.
  - Sinon on suppose que x est un enfant gauche (pour le cas enfant droit est symétrique).
    - ✓ Si le frère(w) de x est rouge, on le rend noir puis on rend le parent(p) de w et x est rouge et on fait une rotation à gauche en p et on se retrouve dans le cas d'un frère noir qu'on va traiter dans les cas restants.

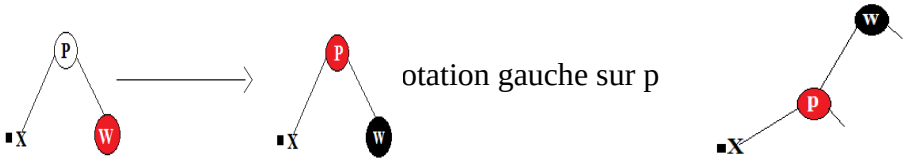


Figure 23:cas où le frère de W est rouge

    - ✓ Si w est noir et n'a que des enfants noirs on rend w rouge et on répète la procédure au niveau du parent p de w et x ou il manque une couleur noire.

Figure 24: cas où W est noir

✓ W est noir et son enfant gauche est rouge on rend w rouge et son enfant gauche noir puis on fait une rotation a droite en w et on passe au cas suivant

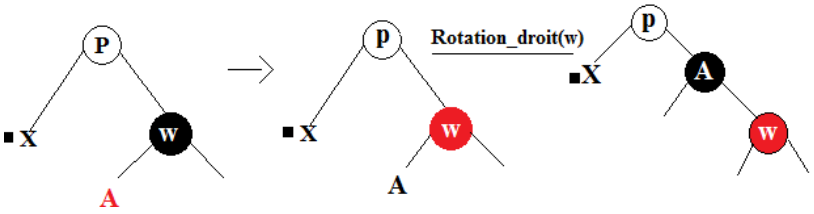


Figure 25:cas où W est noir et son enfant gauche est rouge

✓ Si w est noir et son enfant droit est rouge : on colorie w dans la couleur du parent p par la suite on colorie p en noire et l'enfant droit de w en noir et on fait une rotation gauche au niveau de p.

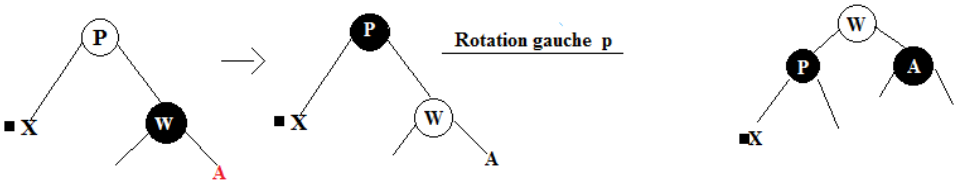


Figure 26:cas où W est noir et son enfant droit est rouge

```
ARBRE-RN-SUPPRESSION(T , x)
   Si gauche(x) == NIL(T) et droit(x) == NIL(T) alors
        Si père(x) == NIL(T ) alors
                racine(T ) NIL(T)
        Sinon
        Si x == gauche(père(x)) alors
                gauche(père(x)) = NIL(T)
        Sinon
                droit(père(x)) = NIL(T)
        Si couleur(x) == NOIR alors
                père(NIL(T )) = père(x)
                RN-CORRECTION(T , x)
   Sinon Si gauche(x) == NIL(T) ou droit(x) == NIL(T) alors
        Si gauche(x) != NIL(T) alors
                 filsde_x = gauche(x)
        Sinon
                filsde_x = droit(x)
        père(filsde_x) = père(x)
        Si père(x) == NIL(T ) alors
                racine(T ) = filsde_x
        Sinon Si gauche(père(x)) == x alors
                gauche(père(x)) = filsde_x
        Sinon
                 droit(père(x)) = filsde_x
        Si couleur(x) == NOIR alors
                RN-CORRECTION(T , filsde_x)
   Sinon
        min = ARBRE-MINIMUM(droit(x))
        clé(y) = clé(min)
        ARBRE-RN-SUPPRESSION(T, min)
renvoyer racine(T )
```

```
RN-CORRECTION(T , x)
        Tant que x != racine(T) et couleur(x) == NOIR faire
                Si x == gauche(père(x)) alors
                        w = droit(père(x))
                        Si couleur(w) == ROUGE alors
                                couleur(w) = NOIR
                                couleur(père(w)) = ROUGE
                                RotationGauche(T , père(x))
                                W = droit(père(x))
                        Si couleur(gauche(w)) == NOIR et couleur(droit(w)) == NOIR alors
                                couleur(w) = ROUGE
                                x = père(x)
                        Sinon Si couleur(droit(w)) == NOIR alors
                                couleur(gauche(w)) = NOIR
                                couleur(w) = ROUGE
                                RotationDroite(T , w)
                                w = droit(père(x))
                          couleur(w) = couleur(père(x))
                          couleur(père(x)) = NOIR
                          couleur(droit(w))) = NOIR
                          RotationGauche(T , père(x))
                          x = racine(T)
                        Sinon (même chose que précédemment en échangeant droit et gauche)
                couleur(x) = NOIR
```

IV. ARBRES BINAIRES DE RECHERCHE OPTIMAUX

1. Contexte et définition

Un Arbre Binaire de Recherche Optimal (ABRO) : est un Arbre Binaire de Recherche dans lequel, connaissant la probabilité de recherche des différentes clés de chaque nœud, les nœuds sont arrangés dans l'arbre de façon à ce que le cout de l'arbre soit minimum.

Supposons que l'on dispose d'un arbre de recherche dans lequel on fait de multiples accès (multiples recherches), comme par exemple un arbre binaire stockant les mots d'un dictionnaire en ligne ou des internautes iraient chercher des dentitions. Imaginons, maintenant que les recherches effectue ne sont pas équiprobables, mais que pour chaque clé k(i) (dans notre exemple, les mots), on a la probabilité p(i) qu'une recherche qui concerne k(i) (par exemple, on sait que les recherches faites sur le mot « Amour » sont plus fréquentes que celles sur le mot « Déception »). Comment doit être construit cet arbre de sorte que le cout moyen de recherche soit optimisé ? Pour répondre donc à cette question, il nous faut au préalable definir la notion de cout. Nous considérons ici que le cout d'une recherche dans un arbre binaire est le nombre de nœuds testes pour trouver la clé. On a donc que le cout de recherche d'une clé k(i) dans un arbre binaire A est hA(k(i)) + 1 (ou la hauteur h de k(i) dans A est la longueur de la chaıne qui va de k(i) a la racine).

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 effectuée dans l'arbre, elle concerne la clé 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 probabilités 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

$$\sum_{i=1}^{n} pi + \sum_{i=0}^{n} qi = 1$$

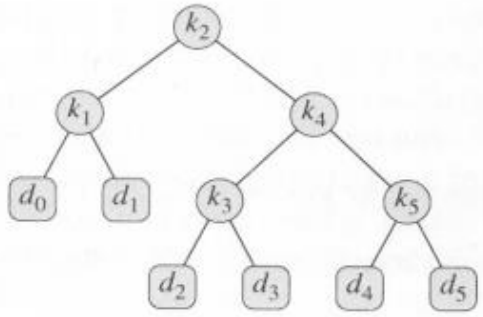
Le cout moyen de recherche dans l'arbre T est donne par

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

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

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 27 : 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 32. D'ailleurs, c'est l'ABRO optimal pour les données considérées.

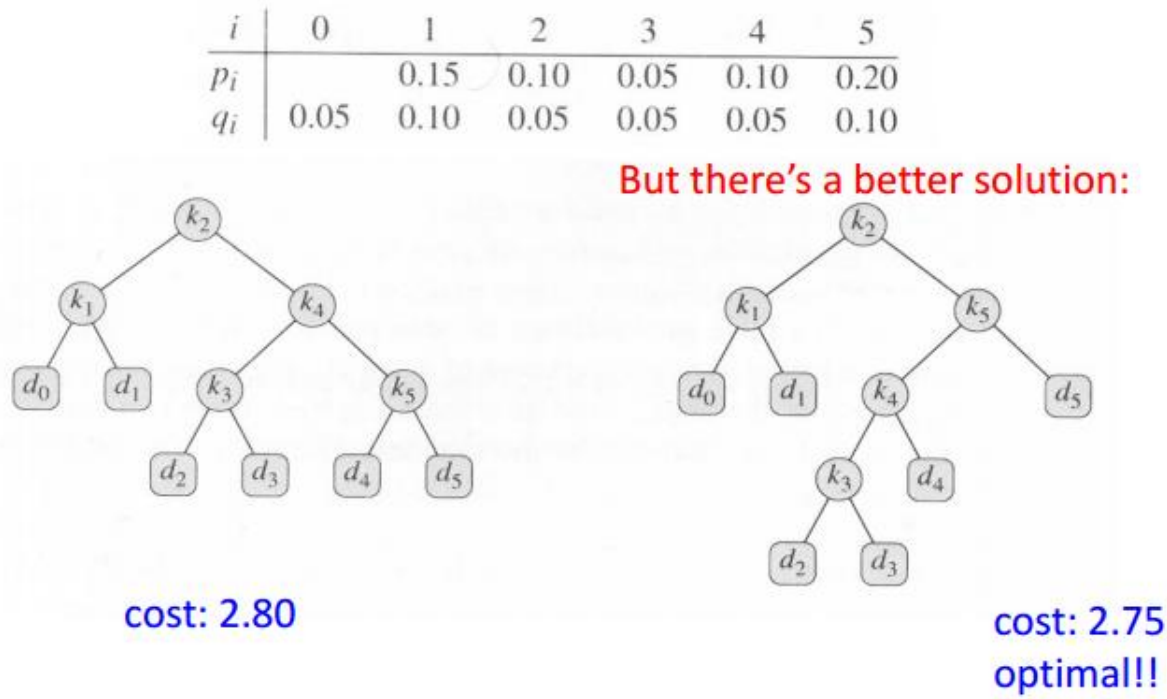


Figure 28 : Arbre binaire de recherche optimal résultant des données de la figure 31 plus-haut

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 $\frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}} = nombre\ de\ Catalan$ différentes configuration 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 forcément 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 $(i \le r \le j)$, 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 sou arbre droit de la racine k_r contiendra les clés $k_{r+1}, \dots, k_j$ (et les clés factices $d_r, \dots, d_j$). Si l'on examine tous les candidats k_r pour la racine ($avec$ $i \le r \le j$) 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 clé k_i comme étant 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 clé réel, mais la clé factice d_{i-1}. Utilisant un raisonnement symétrique, si nous choisissons k_j comme étant 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 $i \ge 1$, $j \le n$ et $j \ge i - 1$. 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 général a n nœud.

Le cas simple a lieu lorsque $j = i - 1$, et dans ce cas, nous avons juste la clé 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 remarqué 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 recherches optimales. 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 clés $k_i, \dots, k_j$ 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 évaluons $e[i, j] = \min\limits_{i \le r \le j} e[i, r - 1] + e[r + 1, j] + w(i, j)\ pour\ i \le j$, 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

Algorithme 1 : ABRO

Entrées : Les probabilités $p_1 \dots p_n$ et $q_0 \dots q_n$, et la taille n

Résultat : Les tableaux e et *racine*

Soient $e[1 \dots n+1, \dots n]$, $w[1 \dots n, 0 \dots n]$ et $racine[1 \dots n, 1 \dots n]$ des tableaux;

```
pour 1 ⩽ i ⩽ n + 1 faire
    e[i, i − 1] ← q_{i−1};
    w[i, i − 1] ← q_{i−1};
fin
pour 1 ⩽ l ⩽ n;                    /* l = le nombre de clefs dans le sous-arbre */
 faire
    pour 1 ⩽ i ⩽ n − l + 1 faire
        j = i + l − 1;
        e[i, j] ← ∞;
        w[i, j] ← w[i, j − 1] + p_j + q_j;
        pour i ⩽ r ⩽ j faire
            t = e[i, r − 1] + e[r + 1, j] + w[i, j];
            si t < e[i, j] alors
                e[i, j] ← t;
                racine[i, j] ← r
        fin
    fin
fin
retourner e et racine
```

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.

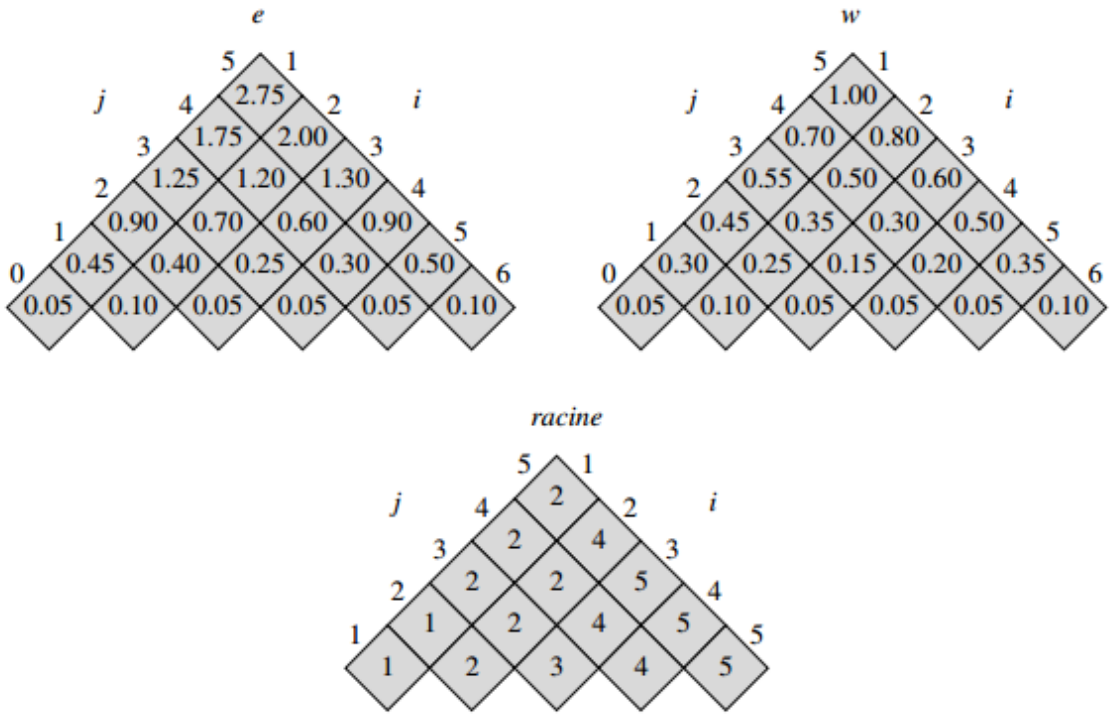


Figure 29: : Données assorties de l'exécution de l'algorithme OPTIMAL-BST avec les données de la figure 31

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

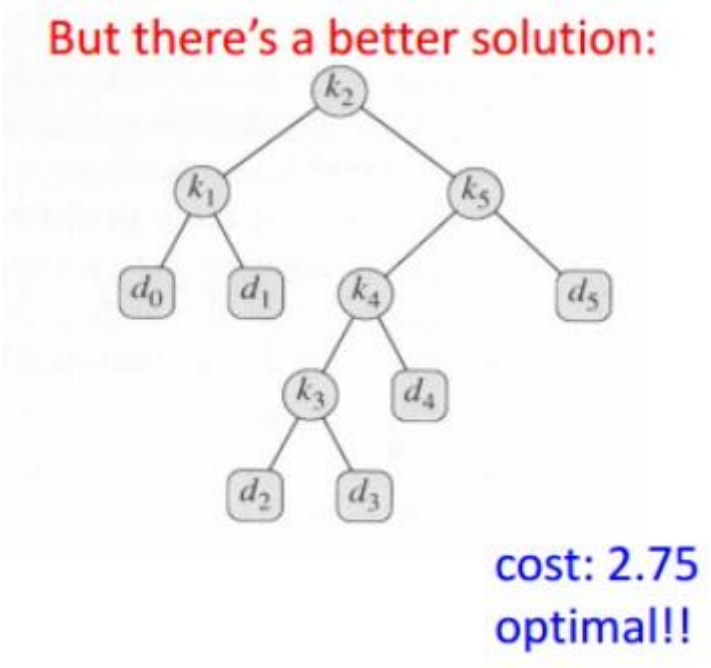


Figure 30: Arbre binaire de recherche optimal construit à partir de la figure 33

A cause de l'imbrication a trois niveaux de la boucle for, la complexité de l'algorithme OBTIMAL-BST est de $O(n3)$.

EXEMPLE PRATIQUE

On se propose de construire un arbre binaire de recherche optimal ayant les clés et les probabilité suivantes :

Numero cles	1	2	3	4
cles	10	20	30	40
probabiliter	4	2	6	3

On sait que :

$$e[i,j] = \begin{cases} q_{i-1} & \text{si } j = i-1 \\ \min\limits_{i \leqslant r \leqslant j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{si } i \leqslant j \end{cases}$$

Notre objectif est de trouver quel sera le numero de la cle racine afin de construire l'arbre optimale. Nous devons remplir le tableau suivant :

i\j	0	1	2	3	4
0					
1					
2					
3					
4					

Etape 1 : pour j - i = 0 on a :

0 – 0 = 0 (0 , 0)

1 – 1 = 0 (1 , 1)

2 – 2 = 0 (2 , 2)

3 – 3 = 0 (3 , 3)

4 – 4 = 0 (4 , 4)

E[i, j] = 0 pour i = j On remplit donc la première diagonale avec 0

Notre tableau devient :

i\j	0	1	2	3	4
0	0				
1		0			
2			0		
3				0	
4					0

Etape 2 : Pour j – i = 1

Lorsque j – 1 = 1, on remplit la diagonale avec la probabilité de chaque numéro de clé on a donc :

i\j	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Etape 3 : pour j – i = 2 on a

2 – 0 = 0 (0 , 2)

3 – 1 = 0 (1 , 3)

4 – 2 = 0 (2 , 4)

Nous avons 03 couple j – i = 2, nous allons donc calculer leurs couts

E[0 , 2] = min{ (e[0 , 0] + e[1 , 2]), (e[0 , 1] + e[2,2]) } + w(0 , 2)

= min{ (0 + 2), (4 + 0) } + (4 + 2)

= min {2 , 4} + 6

E[0 , 2] = 8 et ceci est obtenu bien evivemant lorsque le numero de cle est 1. On metra donc 8^1 dans le tableau.

E[1 , 3] = min{ (e[1 , 1] + e[2 , 3]) , (e[1 , 2] + e[3 , 3]) } + w(1 ,3)

= min{(0 + 6), (2 + 0)} + (2 + 6)

= min {6 , 2} + 8

E[1 , 3] = 10 et il est obtenu avec le numero de cle 3 on mettra donc 10^3 dans le tableau

E[2, 4] = min{ (e[2,2] + e[3,4]), (e[2,3] + e[4, 4]) } + w(2, 4)

= min{(0+3), (6+0)} + (6+3)

= min{3 ,6} + 9

E[2 , 4] = 12 obtenu avec la cle 3 donc 12^3 Notre tableau devient

i\j	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

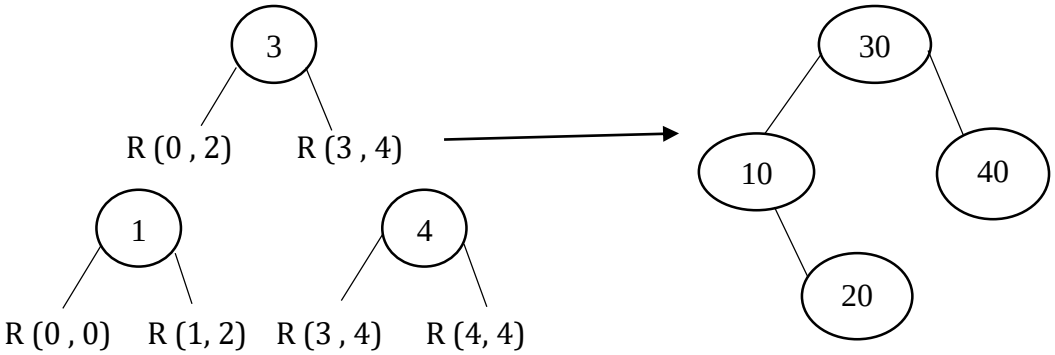
On applique la même méthode pour remplir les deux autre diagonales (c'est-a-dire pour j – i = 3 et pour j – i =

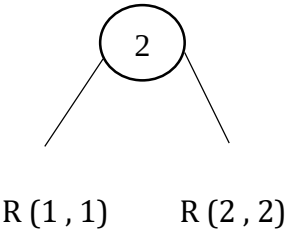
Notre tableau sera donc :

i\j	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^2
2			0	6	12^3
3				0	3
4					0

On constate que le numéro de clé dominant est 3 donc ça sera la racine de notre arbre optimal

R (0 , 4) = 3





CONCLUSION

Tout au long de notre travail qui portait sur les arbres binaires de recherche, les arbres rouges et noirs ainsi que les arbres binaires de recherche optimaux, il a été question pour nous de présenter les opérations de recherche, insertion et suppression pour ces arbres. Nous en sommes ressorti que utilité des arbres se trouve dans le fait qu'appliquer ces différentes opérations est plus trivial et plus rapide. Les arbres ont des inconvénients, lorsque les données à représenter sont volumineux en augmentant le nombre de fois qu'on accède au disque. Pour résoudre ce problème une généralisation des arbres binaires de recherche est donc introduite(les B-arbres).

BIBLIOGRAPHIE

- Algorithmique des structures arborescentes, *Marc Zeitoun L2 info et Math-info 2018-2019, 14 février 2019*
- Algorithmique Les arbres, *Florent Hivert, Mél :Florent.Hivert@lri.fr, 25 Mars 2018*
- Arbres rouge et noir [rn] Algorithmique, *Karine Zampieri, Stéphane Rivière, Version 21 mai 2018*

WEBOGRAPHIE

- https://fr.wikipedia.org/wiki/Arbre_bicolore
- https://www.irif.fr/~carton/Enseignement/Algorithmique/Programmation/RedBlackTree/
- https://pixees.fr/informatiquelycee/n_site/nsi_term_algo_arbre.html
- https://www.google.com/search?q=arbre+rouge+et+noir&oq=arbre+rouge+et+noir&aqs=chrome..69i57.13641j0j7&sourceid=chrome&ie=UTF-8

FICHE DE TRAVAUX DIRIGER

Exercice 1 : Arbres binaires de recherche

Considérer l'ensemble des clés 1,4,5,10,16,17, 21.

1) Dessiner des arbres binaires de recherche de cet ensemble de clés avec une hauteur de 3, puis 5, et ensuite 7.

2.) Voici une liste aléatoire de 15 éléments.

25	60	35	10	5	20	65	45	70	40	50	55	30	15

On s'intéresse aux arbres binaires de recherche.

a) Rappelez les propriétés des arbres binaires de recherche.

b) Construire l'arbre binaire de recherche avec racine 25.

c) Donner la liste infixée de l'arbre obtenu, en justifiant votre raisonnement. Quelle propriété a-t-elle ? Est-ce toujours le cas ?

d) Donner la liste préfixée de l'arbre obtenu.

e) Donner la liste postfixé de l'arbre obtenu.

f) Donner la liste du parcours en largeur de l'arbre obtenu.

g) Donner l'arbre obtenu par insertion de 14 dans l'arbre de la question H, en justifiant votre raisonnement.

h) Donner l'arbre obtenu par suppression de 30 dans l'arbre de la question H, en justifiant votre raisonnement.

Exercice 2 : Insertions dans les ABR

Soit la liste de clés L = (6, 11, 26, 28, 2, 3). Pour chacune des structures, en partant d'un arbre binaire vide, vous devez dessiner l'arbre après chacune des insertions des éléments de la liste L.

EXERCICE 3 :

Montrer les arbres rouge-noir qui résultent de l'insertion successive des clés 41, 38, 31, 12, 19, 8 dans un arbre rouge-noir initialement vide.

Rappel : lorsque nous avons adopté la représentions des arbres binaires qui place comme feuilles des nœuds qui ne contiennent pas d'éléments (les valeurs NULL sont les feuilles les autres nœuds, contiennent les éléments et ont toujours leurs deux fils), nous avons adapté la notion de hauteur en excluant ces nouvelles feuilles du décompte (la hauteur d'un arbre est la même qu'on représente ou non les feuilles NULL).

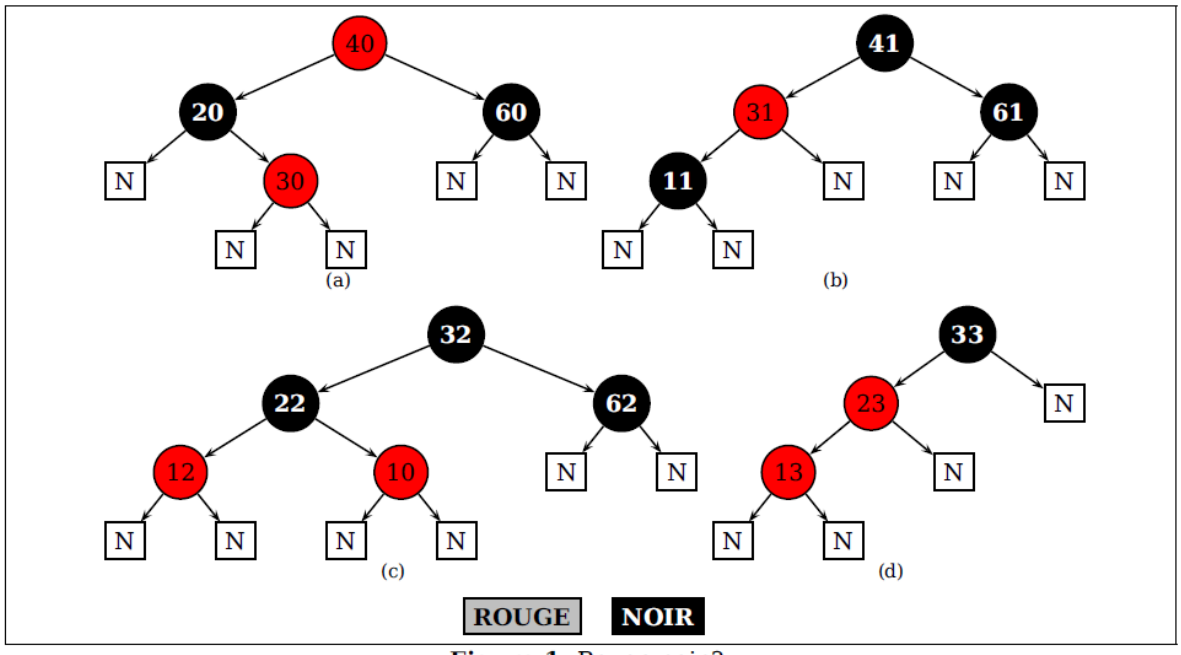


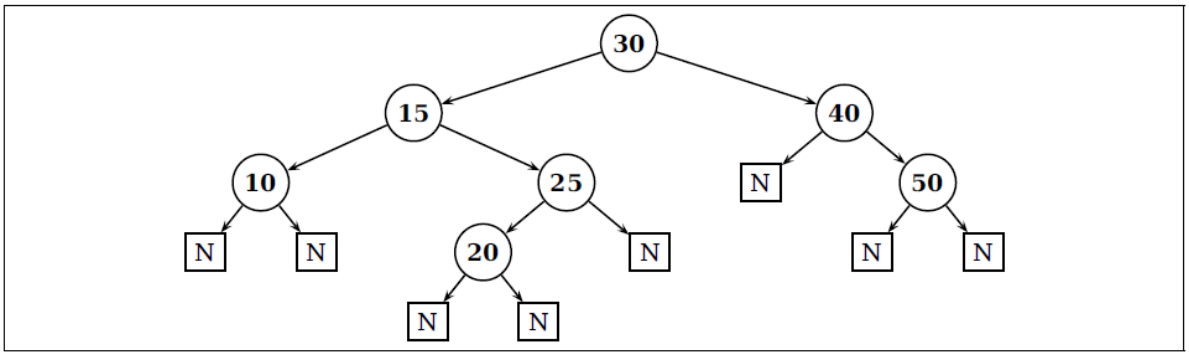
Figure 1: Rouge noir ?

Exercice 4 :

Pour chaque arbre de la figure 1, dire s'il s'agit d'un arbre rouge noir. Si non, pourquoi ?

Exercice 5 :

Est-il possible de colorier tous les nœuds de l'arbre binaire de recherche de la figure 2 pour en faire un arbre rouge noir ?



Exercice 6 :

Notre but est de montrer que la hauteur d'un arbre rouge noire est logarithmique en son nombre de nœuds.
Rappel. Soit x un nœud d'un arbre rouge noir. On appelle hauteur noire de x, notée Hn(x), le nombre de nœuds noirs présents dans un chemin descendant de x (sans l'inclure) vers une feuille de l'arbre.

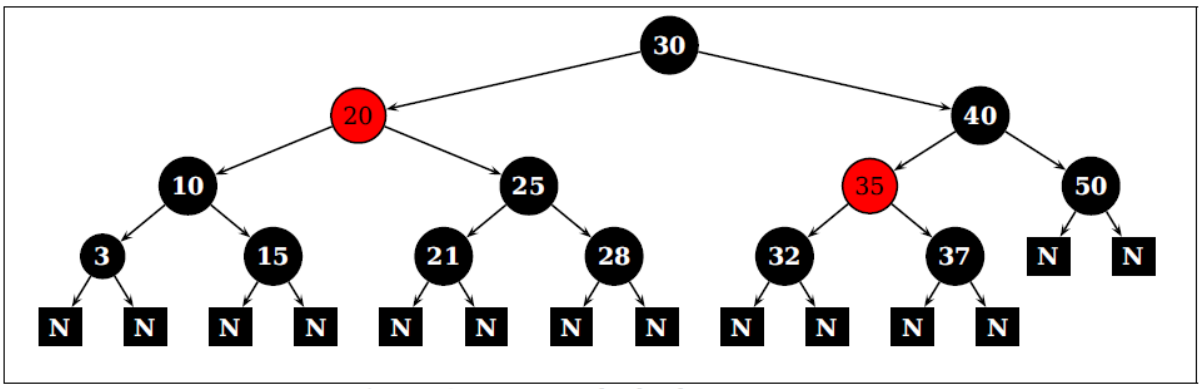


Figure 2: Un exemple d'arbre rouge noir

1. Dans l'arbre rouge noir donné en figure 2, que valent Hn(30), Hn(20), Hn(35), Hn(50) ?
Montrer que, pour un nœud x quelconque dans un arbre rouge noir, le sous-arbre enraciné à x contient au moins $2Hn(x) - 1$ nœuds internes.
En déduire qu'un arbre rouge noir comportant n nœuds internes a une hauteur au plus égale à $2 \log(n + 1)$.

EXERCICE 7 :

Déterminer le coût et la structure d'un arbre binaire de recherche optimal pour un ensemble de n = 7 clés ayant les probabilités suivantes :

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05