Introduction

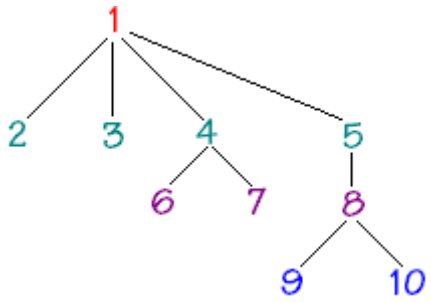
Le monde de l'informatique est en constante évolution et surtout à la recherche permanente de moyens pour optimiser la représentation des données en mémoire, la gestion de base de données et de système de fichiers. C'est pour cette raison que de nombreuses structures de données ont été créées et optimisées parmi lesquelles on retrouve les arbres binaires, les arbres binaires de recherche, les arbres rouges et noirs, les arbres B équilibrés. Dans le cadre de ce cours, nous nous intéresserons aux arbres B équilibrés qui sont mis en œuvre dans les mécanismes de gestion de bases de données et de systèmes de fichier, de stockage sur une unité de masse. Ainsi nous commencerons par définir un arbre B tout en donnant ses différentes propriétés. Ensuite, nous allons d'une part présenter les opérations élémentaires applicables dans un arbre B suivies des différents algorithmes et leurs complexités respectives. Enfin, nous donnerons les avantages et quelques applications des arbres B dans le domaine de l'informatique et de la technologie.

1. Formalisme

1.1. Généralité sur les arbres

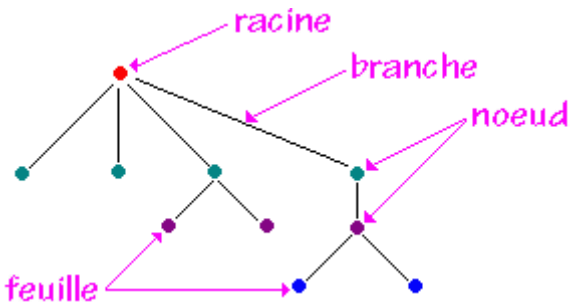
1.1.1. Etiquette, clé

Un arbre dont tous les nœuds sont nommés est dit étiqueté. L'étiquette (ou nom du sommet) représente la "Clé" du nœud ou bien l'information associée au nœud. Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :



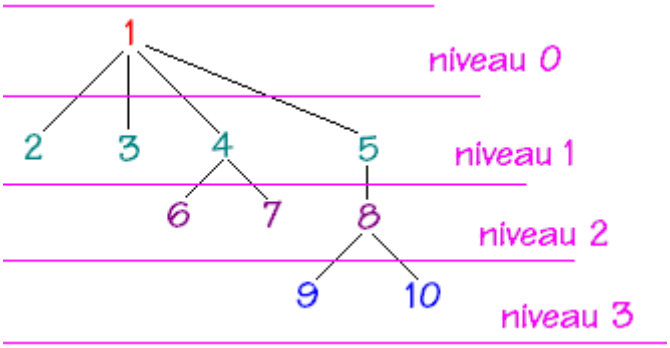
1.1.2. Racine, nœud, branche, feuille

Nous rappelons la terminologie de base sur les arbres sur le schéma ci-dessous :



1.1.3. Hauteur, profondeur ou niveau d'un nœud

Nous conviendrons de définir la hauteur(ou profondeur ou niveau) d'un nœud X comme égale au nombre de nœuds à partir de la racine pour aller jusqu'au nœud X. En reprenant l'arbre précédant et en notant **h** la fonction hauteur d'un nœud :

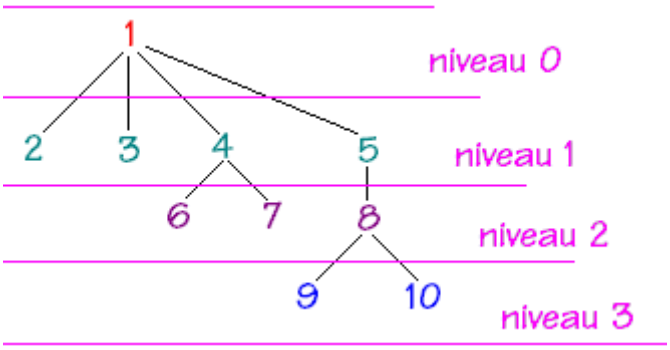


Pour atteindre le nœud étiqueté 9 , il faut parcourir le lien 1--5, puis 5--8, puis enfin 8--9 soient 4 nœuds donc 9 est de profondeur ou de hauteur égale à 4, soit h(9) = 4. Pour atteindre le nœud étiqueté 7, il faut parcourir le lien 1--4, et enfin 4--7, donc 7 est de profondeur ou de hauteur égale à 3, soit h(7)=3. Par définition la hauteur de la racine est égal à 1 mais Certains

auteurs adoptent une autre convention pour calculer la hauteur d'un nœud: la racine a pour hauteur 0 et donc n'est pas comptée dans le nombre de nœuds, ce qui donne une hauteur inférieure d'une unité à notre définition.

1.1.4. Chemin d'un nœud

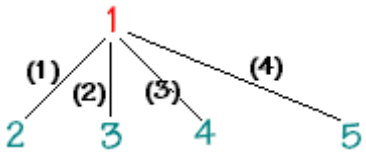
On appelle chemin du nœud X la suite des nœuds par lesquels il faut passer pour aller de la racine vers le nœud X :



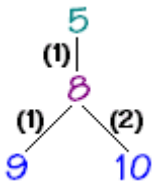
Chemin du nœud 10 = (1, 5, 8,10)
Chemin du nœud 9 = (1, 5, 8,9)
Chemin du nœud 7 = (1, 4,7)
Chemin du nœud 5 = (1,5)
Chemin du nœud 1 = (1)
Remarquons que la hauteur h d'un nœud X est égale au nombre de nœuds dans le chemin :
h(X) = NbrNoeud(Chemin(X)).

1.1.5. Degré d'un nœud

Par définition le degré d'un nœud est égal au nombre de ses descendants (enfants). Soient les deux exemples ci-dessous:



Le nœud 1 est de degré 4, car il a 4 enfants

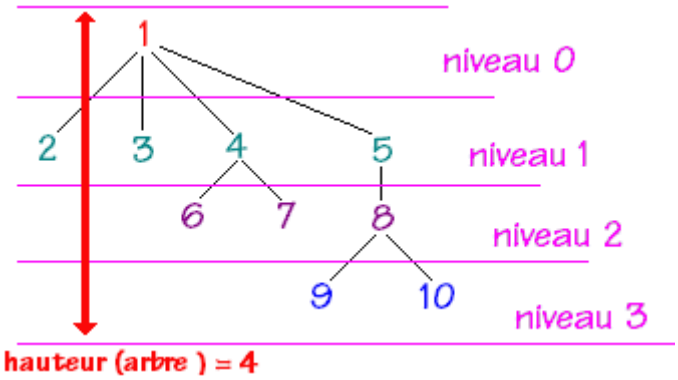


Le nœud 5 n'ayant qu'un enfant son degré est 1.Le nœud 8 est de degré 2 car il a 2 enfants.

1.1.6. Hauteur ou profondeur d'un arbre

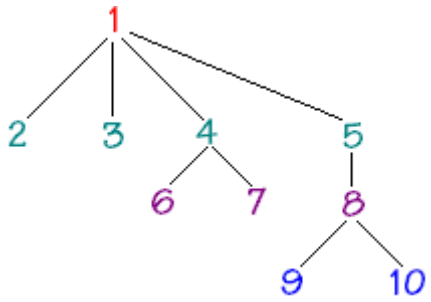
Par définition c'est le nombre de nœuds du chemin le plus long dans l'arbre. La hauteur h d'un arbre correspond donc au nombre de niveau maximum :

La hauteur de l'arbre ci-dessous :



1.1.7. Degré d'un arbre

Le degré d'un arbre est égal au plus grand des degrés de ses nœuds :
Soit à répertorier dans l'arbre ci-dessous le degré de chacun des nœuds :

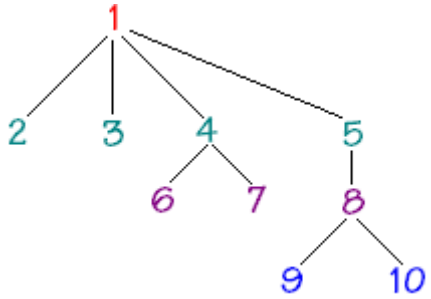


d°(1) = 4 d°(2) = 0
d°(3) = 0 d°(4) = 2
d°(5) = 1 d°(6) = 0
d°(7) = 0 d°(8) = 2
d°(9) = 0 d°(10) = 0

La valeur maximale est 4, donc cet arbre est de degré 4.

1.1.8. Taille d'un arbre

On appelle taille d'un arbre le nombre total de nœuds de cet arbre :



Cet arbre a pour taille 10 (car il a 10 nœuds)

1.2. Contexte

Les B-arbres (ou B-Tree) sont une structure de définition de données utilisée dans les domaines tels que:

- Systèmes de gestion de fichiers : ReiserFS (version modifiée des B-arbres) ou Btrfs (B-Tree file system) ;
- Bases de données : gestion des index

Les B-arbres reprennent le concept d'arbre binaire de recherche (ABR) équilibré mais en stockant dans un nœud **k** les valeurs nommées clés et en référençant k + 1 sous arbres, minimisant ainsi la taille de l'arbre et réduisant le nombre d'opération d'équilibrage ; Les B-arbre sont utilisés pour un stockage sur disque (Unité de masse).

1.3. Définition

Un B-arbre d'ordre m est un arbre tel que :

- Chaque nœud contient k clés triées avec : $\mathbf{m \leq k \leq 2m}$ pour un nœud non racine et $\mathbf{1 \leq k \leq 2m}$ pour un nœud racine.
- Chaque chemin de la racine à une feuille est de même longueur à 1 près
- Un nœud est :
  - ✓ Soit terminal : C'est une feuille
  - ✓ Soit possède (k + 1) fils tels que les clés du ième fils ont des valeurs comprises entre les valeurs du (i-1)ème et ième clés du père.

1.4. Structure d'un nœud :

Un nœud est composé de :

- k clés triées ;
- k+1 pointeur tels que :
  - ✓ Tous sont différents de NIL si le nœud n'est pas une feuille,
  - ✓ Tous sont égale à NIL si le nœud est une feuille.

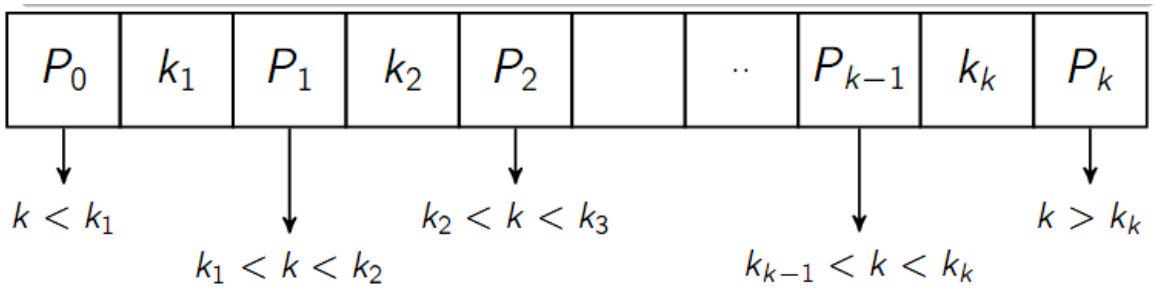


Figure 1: Représentation des clés et pointeur dans un nœud

1.5. Capacité d'un B-arbre

La capacité d'un B-arbre désigne le nombre total minimum et maximum des clés évalué en fonction de la hauteur et de l'ordre de l'arbre.
Soit un B-arbre d'ordre m et de hauteur h :

1. Nombre de clés minimal est : $\mathbf{2*(m+1)^{h}-1}$
2. Nombre de clés maximal est **:** $\mathbf{(2*m+1)^{h+1}-1}$

Remarque : Pour ce qui est du stockage sur disque (ou Unité de masse), un nœud du B-arbre désigne un bloc qui est un ensemble de secteur du disque.

EXEMPLE : B-arbre d'ordre 2

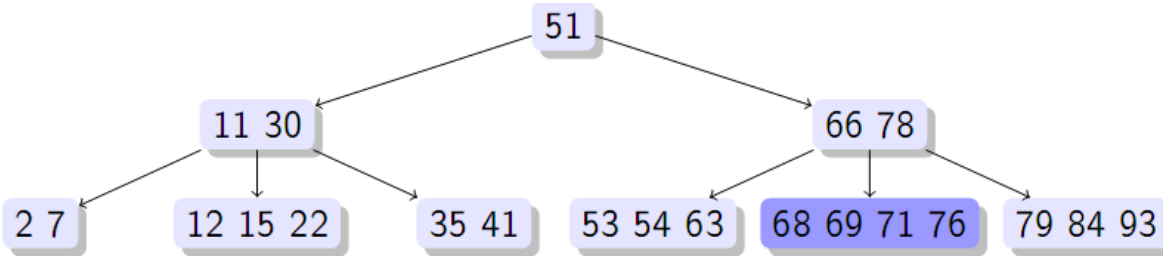


Figure 2:B-arbre d'ordre 2

1.6. Quelques variantes de B-arbre

- **B+ arbre:**

Dans un B+ arbres, les données sont enregistrées dans les feuilles. Les nœuds internes contiennent juste des clés qui sont utilisées pour diriger la recherche vers la feuille considérée.

- **B* arbres :**

Similaires aux B- arbre, mais occupés au 2/3

- **Compact B-arbre :**

Similaire aux B-arbres, seulement les nœuds sont occupés à au moins 90%.

2. Algorithme et Structure d'un B-arbre

2.1. Création d'un B-arbre

2.1.1. Algorithme

__ALGO__: B-arbre
Début
Type *Barbre = Nœud*
Type *Nœud =* ***Structure***
nbClés : ***Naturel_Non_Nul***
clés : ***Tableau*** *[1..MAX] de* ***Valeur***
sous_Arbres : Tableau [0..MAX] de Barbre
Finstructure
#Valeur = Possède des valeurs ordonnées
Fin

2.1.2. Complexité

Nombre d'opérations oe =0
Nombre d'affectations oa =1
Nombre de comparaisons oc =0
Nombre de boucles eb=0

O(1) : Complexité constante car ici il n'y a que identification des différentes variables constituant le B-arbre.

2.2. Recherche dans un B-arbre

2.2.1. Principe

Soit une clé **C** ;

A partir de la racine pour chaque nœud examiné :

- Si la clé **C** est présente alors retourner la clé C ;
- Si $C < K_1$ alors rechercher dans le sous-arbre le plus à gauche via le pointeur P_0
- Si $C > K_k$ alors rechercher dans le sous-arbre le plus à droite via le pointeur P_k
- Si $K_i < C < K_{i+1}$ alors rechercher la clé C dans le sous arbre via le pointeur P_i
- Si l'arbre est vide le pointeur vaut NIL et il y a échec.

Remarque : la recherche peut se faire à travers les processus de dichotomie.

2.2.2. Algorithmes

```
ALGO : RechercheCleBarbre
Fonction estPresent (a : B-Arbre, c : Valeur) : Booleen
Début
        Si a==NIL alors
                Retourner FAUX ;
        Sinon
          Si c<a.cles[1] alors
                  Retourner estPresent(a.sousArbres[0],c) ;
                Sinon
                        Si c>a.cles[a.nbCles] alors
                                Retourner estPresent(a.sousArbre[a.nbCles] , c) ;
                          Sinon
                                RechercherDansNoeud(a,c,res,ssArbre) ;
                                Si res alors
                                        Retourner VRAI ;
                                Sinon
                                        Retourner estPresent(ssArbre) ;
                          Finsi
                    Finsi
          Finsi
                Finsi
        Fin
ALGO : RechercheCleBarbre
Procedure rechercherDansNoeud(E n : Nœud, c : Valeur, S estPresent :Booleen,sousArbre :B-Arbre)
Declaration g, d, m : Naturel&NonNul
Début
        g ← 1;
```

```
d ← n.nbCles;
tant que g ≠ d faire
        m ← (g+d) div 2 ;
        si n.cles[m] ≥ c alors
                d ← m;
        sinon
                g ← m+1;
        finsi
fintanque
si n.cles[g]==c alors
        estPresent ← VRAI;
        sousArbre ← NIL;
sinon
        estPresent ← FAUX;
        sousArbre ← n.sousArbres[g-1];
finsi
```

Fin

2.2.3. Complexité

Calcul de la complexité de la fonction **rechercherDansNoeud**(E n : Nœud, c : Valeur, S estPresent : Booleen,sousArbre :B-Arbre)**:**

Soit **oe** le nombre d'opérations, **ae** le nombre d'affectations, **ce** le nombre de comparaisons.

Tn= ae +n/2(oe + ae +ce) +ce

Tn=O (n/2)

Complexité de la procédure de recherche =O ($\log_2(n)$);

Calcul de la complexité de la procédure **estPresent (a : B-Arbre, c : Valeur)** :

Complexité = O ($\log_2(n)$);

Complexité total ALGO : RechercheCleBarbre :

Complexité = O ($\log_2(n)$);

2.3. Insertion dans un B-arbre

2.3.1. Principe

- L'insertion se fait récursivement au niveau des feuilles
- Si un nœud a alors plus $2m + 1$ clés, il y a éclatement du nœud et remontée (grâce à la récursivité) de la clé médiane au niveau du père
- Il y a augmentation de la hauteur de l'arbre lorsque la racine se retrouve avec $2m + 1$ clés (l'augmentation de la hauteur de l'arbre se fait donc au niveau de la racine)

2.3.2. Algorithme

On suppose posséder les fonctions/procédures suivantes :

- **fonction** creerFeuille **(**c : **Tableau [**1..MAX**] de** Valeur, nb : **NaturelNonNul) :** ArbreB

- **fonction** estUneFeuille **(**a : ArbreB**) : Booleen**
- **fonction** eclatement **(**a : ArbreB, ordre : **NaturelNonNul) :** ArbreB (**précondition(s)** a.nbCles>2*ordre)
- **fonction** positionInsertion **(**a : ArbreB, c : Valeur**) : NaturelNonNul**
- **procédure** insererUneCleDansNoeud **(E/S** n : Noeud, **E** c : Valeur, pos :**NaturelNonNul)**
- **procédure** insererUnArbreDansNoeud **(E/S** n : Noeud, **E** a : ArbreB, pos : **NaturelNonNul)**
- **procédure** inserer **(E/S** a : ArbreB, **E** c : Valeur, ordre : **NaturelNonNul) debut** a←insertion(a,c,ordre)
- **fin**

```
Fonction insertion (a : ArbreB, c : Valeur, ordre : NaturelNonNul) : ArbreB
    Déclaration tab : Tableau[1..MAX] de Valeur
    debut
        si a = NIL alors
            tab[1] ←c
            retourner creerFeuille(tab,1)
            sinon
            pos← positionInsertion(a,c)
            si estUneFeuille(a) alors
            insererUneCleDansNoeud(a^,c,pos)
            si a^.nbCles≤2*ordre alors
            retourner a
            sinon
            retourner eclatement(a, ordre)
            finsi
            sinon
            ssArbre a^.sousArbres[pos]
            temp insertion(ssArbre,c,ordre)
            si ssArbre = temp alors
            retourner a
            sinon
            insererUnArbreDansNoeud(a^,temp,pos)
            si a^.nbCles≤2*ordre alors
            retourner a
            sinon
            retourner eclatement(a, ordre)
        finsi
            finsi
        finsi
        finsi
fin
```

2.3.3. Exemples

Exemple 1 : Insertion dans un nœud plein :

Principe : Eclatement du nœud en deux :

- Les (deux) plus petites clés restent dans le nœud

- Les (deux) plus grandes clés sont insérées dans un nouveau nœud
- Remontée de la clé médiane dans le nœud père

Insertion d'une clé dans un B-arbre d'ordre 2 (Insertion de 75)

On veut insérer la clé 75, on procède ainsi : On effectue tout d'abord la recherche et le repérage du nœud concerné, on vérifie à partir de l'ordre de l'arbre si le nombre de clé est atteint dans le nœud :

- Si le nœud n'est pas saturé on insère la clé 75 et on réordonne les valeurs des clés du nœud,
- Si le nœud est saturé, on effectue l'éclatement du nœud et la remonté de la clé médiane vers le nœud père.

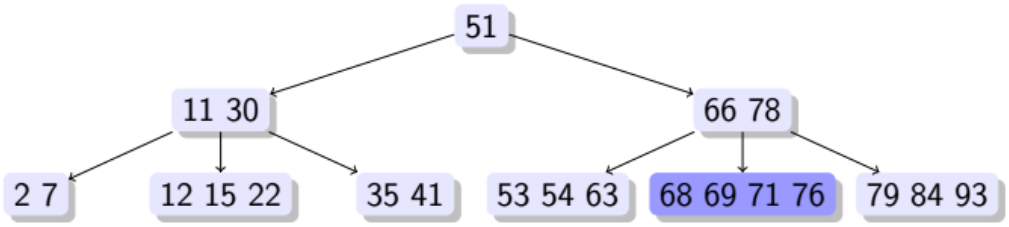


Figure 3: arbre avant insertion de clé

Rappel : ici nombre de clés par nœud ≤ 4

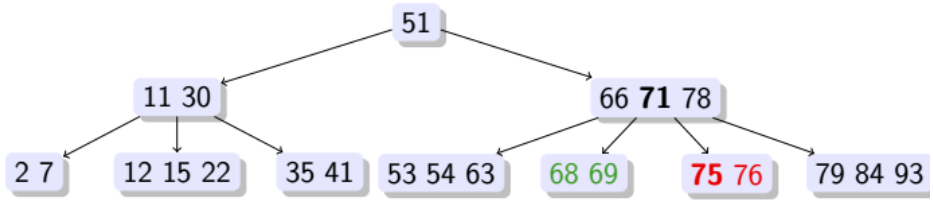


Figure 4: Arbre après insertion d'une clé

Exemple 2 : Insertion dans un nœud plein avec augmentation de la hauteur de l'arbre

Dans le cas de l'insertion dans un nœud plein qui produit une augmentation de la hauteur de l'arbre la méthode peut se décrire ainsi :

- Insertion dans un nœud plein telle que présentée précédemment (insertion de la nouvelle clé suivi de l'éclatement du nœud et de la remontée de la clé médiane au nœud père),
- Ensuite il y a éclatement du nœud père et création du nouveau nœud père ou d'une nouvelle **racine** (cas où le nœud père était la racine et l'ordre était atteint).
- Enfin il y a eu Augmentation d'une ou plusieurs unité(s) de la hauteur du B-arbre Arbre-B d'ordre 2

Exemple : Insertion de 9

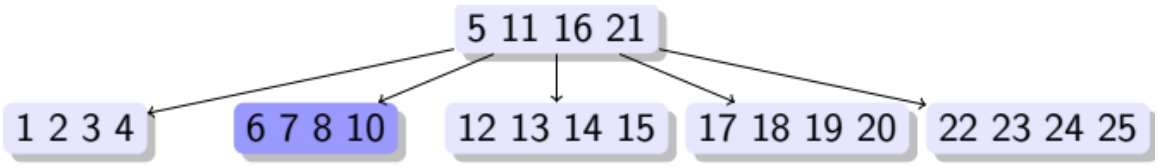


Figure 5: arbre initial avant éclatement

- Insertion clé **9** → Eclatement + remontée de la clé **8** au nœud père
- Remontée de la clé **8** au nœud père →Eclatement + création nouvelle racine

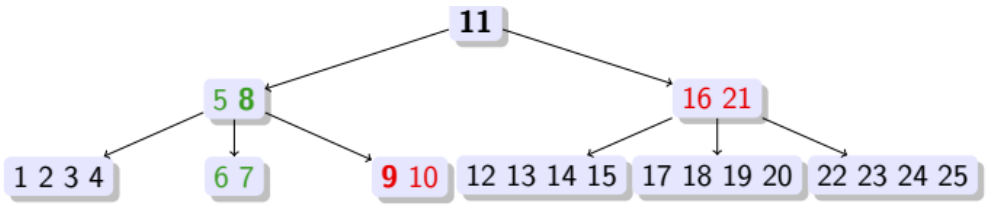


Figure 6: graphe après éclatement et insertion

2.3.4. Complexité

Complexité=O ($\log_2(n)$) ;

2.4. Suppression dans un B-arbre

2.4.1. Principe

Dans un B-arbre la suppression des clés se fait toujours au niveau des feuilles ; Si la clé à supprimer n'est pas dans une feuille, alors on la remplace par la plus grande valeur des plus petites clés (ou plus petite valeur des plus grandes clés) et on supprime cette dernière. Si la suppression de la clé d'une feuille (récursivement d'un nœud) amène à avoir moins de **m** clés dans un nœud : on fait une combinaison (fusion) de ce nœud avec un nœud voisin (avant ou après) et on descend la clé associant ces deux nœuds (avec éclatement du nœud si nécessaire), la récursivité de ce principe pouvant amener à diminuer la hauteur de l'arbre par le haut.

2.4.2. Exemple

Exemple 1 de suppression : combinaison avec un nœud voisin

Le B-arbre est d'ordre 2, le nombre de clés par nœud non racine > 1 ; Pour la suppression de la clé 25, on procède comme suit :

- Combinaison avec un nœud voisin ([12 14] et 20),
- Descente de la clé médiane (ici 15),
- Suppression du nœud.

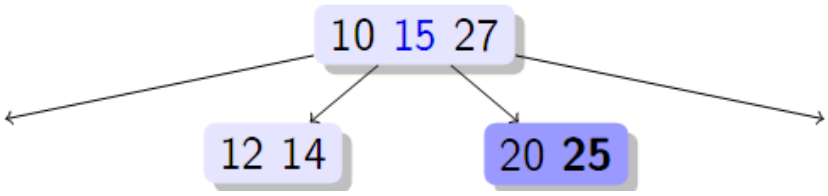


Figure 7: Exemple : B-arbre d'ordre 2- Suppression de la clé 25

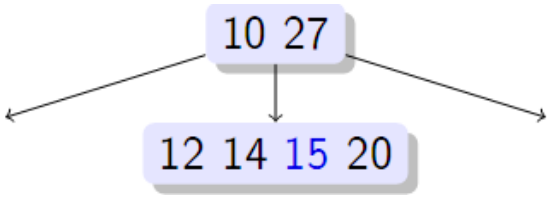


Figure 8: Exemple: Combinaison avec le Nœud voisin et descente de la clé médiane

Exemple 2 de suppression : Avec éclatement d'un nœud

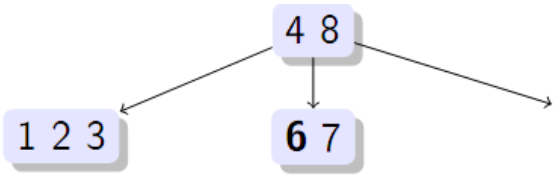


Figure 9: Suppression avec éclatement du nœud

Dans cet exemple, le nombre de clés par nœud non racine est < 5 et > 1 ; on veut supprimer la clé **6**, on procède comme suit:
Suppression clé **6**, Combinaison des clés **[1 2 3]** et **7** puis descente de la clé **4** au nœud fils ; ensuite redistribution des clés et remontée de clé médiane.

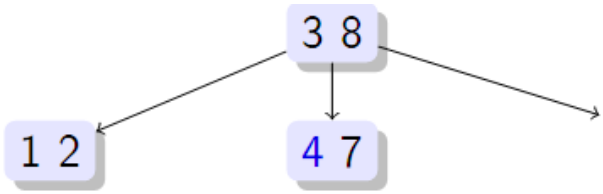


Figure 10: Suppression + éclatement du nœud : redistribution des clés et remontée clé médiane

Exemple 3 de suppression : Avec un nombre de clé inférieur à m

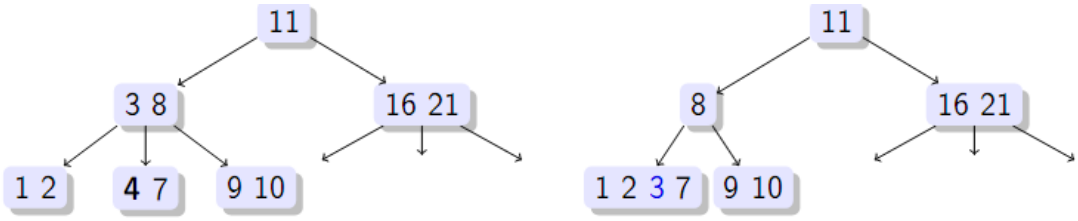


Figure 11: Suppression avec un nombre de clé inferieur à m

On veut supprimer la clé 4 :
On effectue tout d'abord une Combinaison des clés ([1 2] et 7) et une descente de la clé 3 ; Combinaison des clés (8 et [16 21]) et descente de la clé 11. Ceci entraine une diminution d'une unité de la hauteur de notre B-arbre.

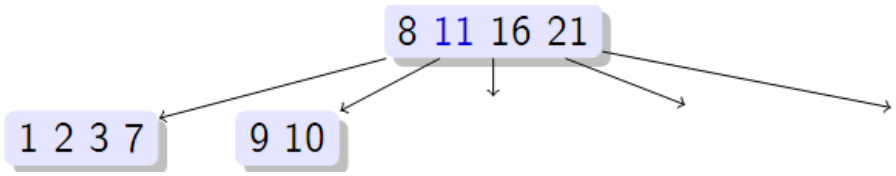


Figure 12: Suppression + nombre de clé inferieur à m : diminution hauteur B-arbre

Exemple 4 de suppression : dans un nœud non feuille

Principe :

Pour effectuer une telle suppression, on procède comme suit :

- Tout d'abord on effectue la recherche d'une clé adjacente **A** à la clé à supprimer puis on choisit la plus grande du sous arbre gauche ;
- On effectue le remplacement de la clé à supprimer par **A**
- On supprime la clé remplacé du sous arbre gauche.

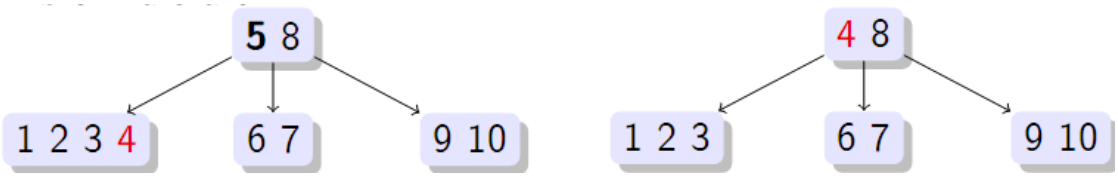


Figure 13: Suppression nœud non feuille

On veut supprimer la clé 5 du nœud racine de notre figure précédente, on procède ainsi :

Recherche de la plus grande clé adjacente à 5 dans le sous arbre gauche ici 4 ; on effectue ensuite une permutation entre la clé 5 de la racine et 4 du nœud feuille, enfin on supprime la clé 5.

2.4.3. Algorithmes

Fonction plusGrandeValeur (a : B-Arbre) : Valeur

précondition(s) a ≠ NIL ;

Fonction positionCleDansNoeudRacine (a : B-Arbre, c : Valeur) : Entier

précondition(s) a ≠ NIL

Fonction positionSsArbrePouvantContenirValeur (a : Arbre, c : Valeur) : Naturel

Procedure freres (E a : B-Arbre, posSSArbre : Naturel,S frereG, frereD : B-Arbre)

précondition(s) a ≠ NIL et a.sousArbres[posSsArbre] ≠ NIL ;

Procedure supprimerCleDansNoeudFeuille (E/S n : Nœud, E c : Valeur)

Procedure copierValeurs(S tDest ; Tableau[1..MAX] de Valeur, E tSource : Tableau[1..MAX] de Valeur, indiceDebutDest, indiceDebutSource, nb : NaturelNonNul)

précondition(s) indiceDebutSource+nb < MAX et indiceDebutDest+nb < MAX;

Procedure decalerVersGaucheClesEtSsArbres (E/S n : Noeud, E aPartirDe : NaturelNonNul, nbCran : NaturelNonNul)

ALGO :Suppression B-arbre

```
Procedure supprimer(E/S a : B-Arbre, E c : Valeur, ordre : NaturelNonNul)
Begin
        a ← suppression(a,c,ordre,NIL,NIL,uneCle) ;
 End
FIN_ALGO
```

Fonction *suppression(a : B-Arbre, c : Valeur, ordre : NaturelNonNul, frereG, frereD : B-Arbre, clePere : Valeur) : B-Arbre*

Déclaration . . .

Début

```
si a = NIL alors
        retourner a ;
sinon
        pos ← positionCleDansNoeudRacine(a,c) ;
        si estUneFeuille(a) alors
                si pos = -1 alors
                        retourner a ;
                sinon
                        si frereG = NIL et frereD = NIL et a.nbCles=1 alors
                                desallouer(a) ;
                        sinon
                                # supprimer c dans une feuille
                                supprimerCleDansNoeudFeuille(a,c) ;
                        finsi
                finsi
        sinon
                si pos = -1 alors
                        posSsArbre ← positionSsArbrePouvantContenirValeur(a,c) ;
                sinon
                        cleRemplacement ← plusGrandeValeur(a.sousArbres[pos-1]) ;
                        a.valeurs[pos] ← cleRemplacement ;
                         c ← cleRemplacement posSsArbre ← pos-1 ;
                finsi
                freres(a,posSsArbre,frereG,frereD)
        finsi
finsi
```

Fin

Fonction *supprimerCleDansNoeudFeuille(a,c)*

Début

```
si a.nbCles ≥ ordre ou (frereG = NIL et frereD = NIL) alors
        retourner a ;
sinon
        si frereG 6= NIL alors
                copierValeurs(tab,frereG.valeurs,1,1,frereG.nbCles) ;
```

```
                tab[frereG.nbCles+1] ← clePere ;
                copierValeurs(tab,a.valeurs,frereG.nbCles+2,1,a.nbCles) ;
                nb ← 1+a.nbCles+frereG.nbCles ;
                desallouer(frereG) ;
        sinon
                copierValeurs(tab,a^.valeurs,1,1,a.nbCles) ;
                tab[a.nbCles+1] ← clePere ;
                copierValeurs(tab,frereD.valeurs,a.nbCles+2,1,frereD.nbCles) ;
                nb ← 1+a.nbCles+frereD.nbCles ;
                desallouer(frereD) ;
        finsi
        res ← creerFeuille(tab,nb)
        desallouer(a) ;
        si nb>2*ordre alors
                retourner eclatement(res, ordre) ;
        finsi
        retourner res ;
    finsi
Fin

Fonction freres(a,posSsArbre,frereG,frereD)
Début
    res←suppression(a.sousArbres[posSsArbre],c,ordre,frereG,frereD,a.valeurs[
    posSsArbre])
    si res.nbCles=1 alors
        a.valeurs[posSsArbre] ← res.valeurs[1] ;
        a.sousArbres[posSsArbre-1] ← res.sousArbres[0] ;
        a.sousArbres[posSsArbre] ← res.sousArbres[1] ;
    sinon
        decalerVersGaucheClesEtSsArbres(a^,posSsArbre,1)
        a.sousArbres[posSsArbre] ← res ;
    finsi
    retourner a
Fin
```

2.5. Calcul de la complexité de l'algorithme de suppression :

Calcul de la complexité de la procédure **ALGO** : Suppression B-arbre

Nombre d'opérations eo =n/2

Nombre d'affectations ea = n/2

Nombre de comparaisons ec =n/2

Nombre de boucles eb=1

Complexité=O ($\log_2(n)$) ;

Avantages de B-arbre

Les b-arbres apportent de solides avantages en terme de rapidité et d'efficacité par rapport à d'autres mises en œuvre lorsque la plupart des nœuds sont dans le stockage secondaire, comme le disque dur. En maximisant le nombre de nœuds enfants pour chaque nœud, la hauteur de l'arbre est réduite, l'opération d'équilibrage est nécessaire moins souvent et donc l'augmentation de l'efficacité. En général, ce nombre est réglé de telle sorte que chaque nœud occupe tout un groupe de secteurs : ainsi étant donné que les opérations de bas niveau pour accéder au disque de cluster, il réduit au minimum le nombre d'accès à elle. Ils offrent d'excellences performances par rapport aux deux opérations de recherche et ceux de rénovation, car les deux peuvent être faites avec la complexité logarithmique et par utilisation des procédures très simples. Sur eux, il est également possible d'effectuer un traitement séquentiel d'archivage primaire sans qu'il soit nécessaire de le soumettre à une réorganisation.

Conclusion

En définitive, nous avons détaillé la structure des B-arbres et analysé ses performances à travers les algorithmes de recherche, d'insertion et de suppression. Ces algorithmes ont une complexité de O (log n) au pire des cas ce qui fait des B-arbre une structure de données efficace. Même si les B-arbres possèdent des propriétés intéressantes, ils n'en demeurent pas moins que des désavantages subsistent pour quelques applications. Par exemple les B-arbres peuvent nécessiter plusieurs opérations de regroupements ou d'éclatement après une opération insertion ou de suppression.

Bibliographie

- Introduction-à-l'algorithmique-Cours-et-exercices-corrigés Dunod, Paris, 2004, pour la présente édition ISBN 2 10 003922 9

Webographie

Les B-arbre

https://fr.wikipedia.org/wiki/Arbre_B

https://rmdiscala.developpez.com/cours/LesChapitres.html/Cours4/Chap4.7.htm

http://igm.univ-mlv.fr/~mac/ENS/DOC/barbr_17.pdf

https://www.labri.fr/perso/maylis/ASDF/supports/notes/FILM/B-arbreExemples.pdf