LES ARBRES N-AIRES

TABLE DES MATIERES

INTRODUCTION

DEFINITION

Les arbres binaires sont des structures de données. Ils sont définis ainsi :

- Le premier élément est désigné comme la “racine” de l’arbre.
- Découlent de cet élément appelé “père” n éléments dit “fils”. On utilise donc la terminologie de la généalogie pour décrire ces relations.
- Chaque élément de cet ensemble est appelé “nœud" et il y est associé une valeur ou information. Un nœud est également défini par ses relations, soit ses fils et son père.
- Les derniers éléments de l’arbre sont appelés “feuilles” de l’arbre.

On parle également de degré d’un nœud ; il s’agit du nombre de fils qu'il possède. On distingue en conséquence les nœuds de degré non nul, soit les nœuds non-terminaux, et les nœuds de degré nul qui sont les nœuds terminaux ou feuilles de l’arbre.

Comparatif des opérations binaires dans le pire des càs :

Implantation	Rechercher	Insérer	Supprimer
Tableau non ordonné	$O(n)$	$O(1)$	$O(n)$
Liste non ordonnée	$O(n)$	$O(1)$	$O(1)$
Tableau ordonné	$O(\log n)$	$O(n)$	$O(n)$
Liste ordonnée	$O(n)$	$O(n)$	$O(1)$
Pile	$O(n)$	$O(\log n)$	$O(n)$
Arbre de recherche*	$O(h)$	$O(h)$	$O(h)$

*vitesse dans le cas d'arbre de recherche simple, non équilibré.

LA THEORIE

LES DEUX TYPES D'ARBRES

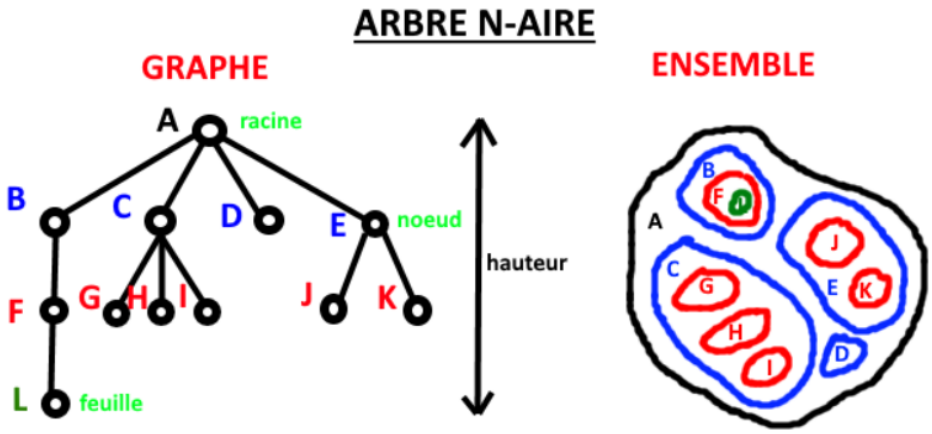
LES ARBRES BINAIRES

Définition : Un **arbre binaire** est une structure de données, hiérarchisée, constituée d’éléments nommés nœuds. Le nœud initial est appelé racine et chaque nœud n’ayant pas de fils est appelé feuille. Chaque nœud peut avoir 0 à 2 fils, le nombre de fils d’un père est appelé degré. La distance entre la racine et la plus éloignée des feuilles est appelée hauteur et le niveau d’un nœud par rapport à la racine s’appelle la profondeur.

Les arbres binaires peuvent notamment être utilisés en tant qu'arbre de recherche ou en tant que tas binaire.

LES ARBRES N-AIRES

Définition : Un **arbre n-aire** est un arbre binaire dont le degré n'est pas restreint à 2.



LES ARBRES PARTICULIERS

- Un **arbre binaire entier** est un arbre dont tous les nœuds possèdent zéro OU deux fils.
- Un **arbre binaire parfait** est un arbre binaire entier dans lequel toutes les feuilles sont à la même distance de la racine (c'est-à-dire à la même profondeur).

LES ARBRES BINAIRES DE RECHERCHE

Un arbre binaire de recherche (ou ABR) 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 la clé du nœud. Selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale.

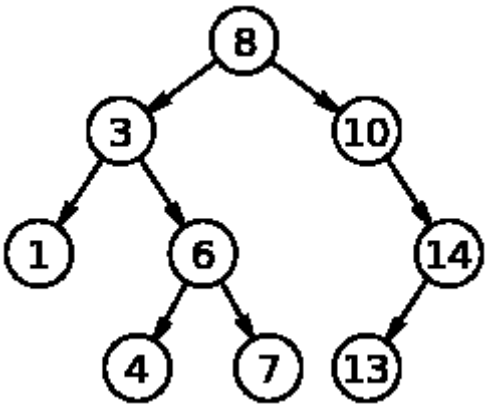


Figure 1 Exemple d'un ABR à 9 clés.

La recherche dans un arbre de recherche se fait par récursivité. On commence par la racine, on regarde si la clé correspond à au nœud en cours. Si le nœud correspond, on renvoie la valeur de ce nœud, si elle est supérieure, on prend le nœud du sous arbre de droite, si elle est inférieure, on passe au nœud du sous-arbre de gauche et ainsi de suite jusqu'à trouver la valeur. Si on atteint une feuille dont la clé n'est pas recherchée, on sait que la clé recherchée n'appartient à aucun nœud.

Cette opération requiert un temps en O(log(n)) dans le cas moyen, mais O(n) dans le cas critique où l'arbre est complètement déséquilibré et ressemble à une liste chaînée. Ce problème est écarté si l'arbre est équilibré par rotation au fur et à mesure des insertions pouvant créer des listes trop longues. La recherche requiert un temps en O(log(n)) dans le cas moyen, mais O(n) dans le cas critique

Les applications sont : le parcours ordonné, le tri, les files de priorité.

LES ARBRES AVL

Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un. La recherche, l'insertion et la suppression sont toutes en $O(\log_2(n))$ dans le pire des cas. L'insertion et la suppression nécessitent d'effectuer des rotations.

Les opérations de base d'un arbre AVL mettent en œuvre généralement les mêmes algorithmes que pour un arbre binaire de recherche, à ceci près qu'il faut ajouter des rotations de rééquilibrage nommées « rotations AVL ».

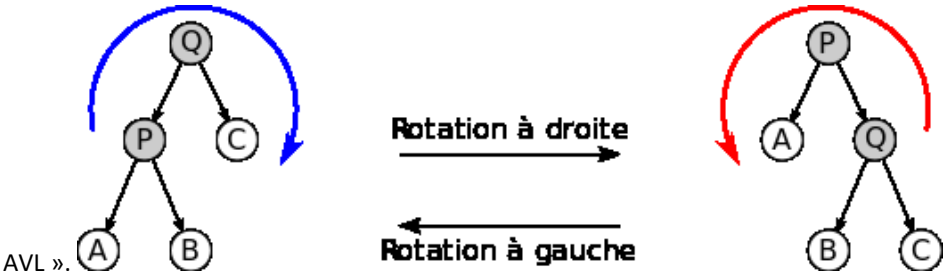


Figure 2 Schéma illustrant le principe de la rotation d'un arbre.

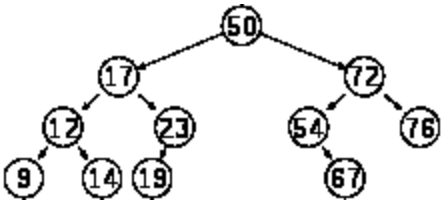


Figure 3 Exemple d'un arbre AVL équilibré (à la différence de l'arbre précédent).

Le facteur d'équilibrage d'un nœud est la différence entre la hauteur de son sous-arbre droit et celle de son sous-arbre gauche. Un nœud dont le facteur d'équilibrage est 1, 0, ou -1 est considéré comme équilibré.

LES ARBRES BICOLORES

Les arbres bicolore ou arbre rouge et noir sont un type particulier d'arbre de recherche équilibré. Un arbre bicolore est un arbre binaire de recherche dans lequel chaque nœud a un attribut supplémentaire : sa couleur,

qui est soit rouge soit noire. En plus des restrictions imposées aux arbres binaires de recherche, les règles suivantes sont utilisées :

- Un nœud est soit rouge soit noir ;
- La racine est noire ;
- Les enfants d'un nœud rouge sont noirs;
- Tous les nœuds ont 2 enfants. Ce sont d'autres nœuds ou des feuilles NIL, qui ne possèdent pas de valeur et qui sont les seuls nœuds sans enfants. Leur couleur est toujours noire et rentre donc en compte lors du calcul de la hauteur noire.
- Le chemin de la racine à n'importe quelle feuille (NIL) contient le même nombre de nœuds noirs. On peut appeler ce nombre de nœuds noirs la hauteur noire.

La recherche d'un élément se fait en temps logarithmique **O(log n)**, y compris dans le pire des cas.

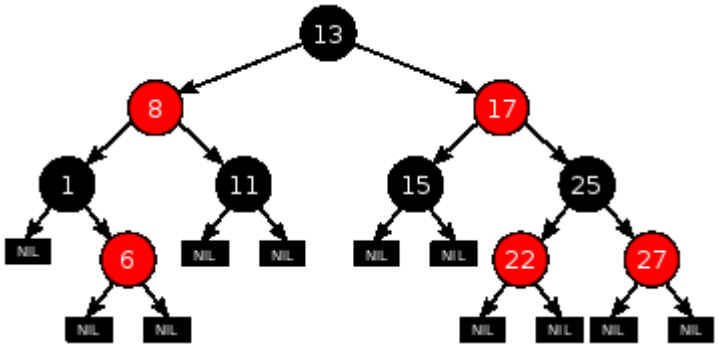


Figure 2 Exemple d'un arbre bicolore

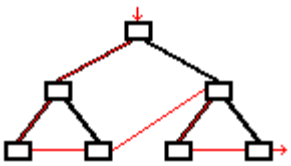
Ces contraintes impliquent une propriété importante des arbres bicolores : le chemin le plus long possible d'une racine à une feuille (sa hauteur) ne peut être que deux fois plus long que le plus petit possible : dans le cas le plus déséquilibré, le plus court des chemins ne comporte que des nœuds noirs, et le plus long alterne les nœuds rouges et noirs.

Applications : tableaux associatifs capables de garder en mémoires les versions précédentes après un changement. Les versions persistantes des arbres rouge-noir requièrent O(log n) en mémoire supplémentaire pour chaque insertion ou suppressions.

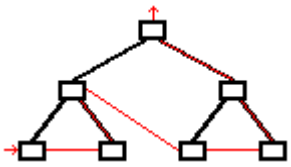
LES ARBRES COUSUS

Les arbres cousus (threaded tree) sont basées sur des arbres binaire mais les liaisons sont matérialisés différemment afin d'être optimisé pour chaque type de méthode de parcours.

Arbre cousu en pré ordre : le chaînage suit un parcours préfixe de l'arbre : nœuds parents en premier, nœuds enfants ensuite.



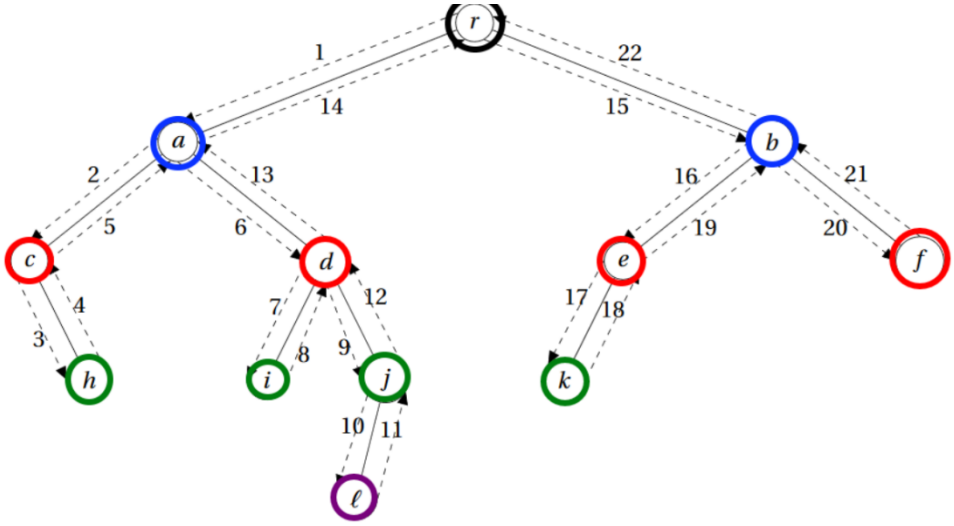
Arbre cousu en postordre: le chaînage suit un parcours postfixe de l'arbre : nœuds enfants en premier, nœuds parents en dernier.



Arbre cousu en inordre: Un arbre dont le chaînage suit un parcours infixe de l'arbre : nœud fils gauche, nœud parent, nœud fils droit. Dans le cas d'un arbre binaire de recherche, ce chaînage correspond à une liste chaînée triée.

COMMENT ON LES PARCOURT

On se balade autour d'un arbre en tournant autour des différents nœuds.

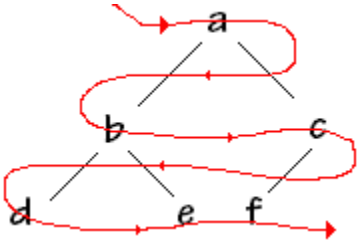


LES DIFFERENTS TYPES DE PARCOURS :

Le parcours en largeur consiste à parcourir l'arbre niveau par niveau. Les nœuds de niveau 0 sont sont d'abord parcourus puis les nœuds de niveau 1 et ainsi de suite. Dans chaque niveau, les nœuds sont parcourus de la gauche vers la droite.

Exemple 1 : **r, a, b, c, d, e, f, h, I, j, k, l.**

Exemple 2:



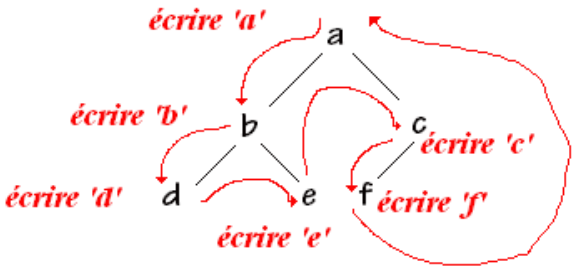
Le parcours en profondeurs qui se définissent de manière récursive.

Ce qui donne 3 types d'écriture possibles :

- L'écriture **préfixée** : la racine vers les feuilles.

  Exemple 1 : **(r (a (c (h)) (d (i) (j (l)))) (b (e (k)) (f)))**

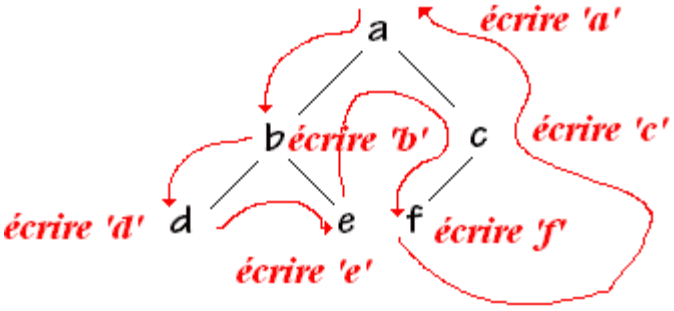
  Exemple 2:



- L'écriture **postfixée** : les feuilles vers la racine.

  Exemple 1: **((((l) j) (i) d) ((h) c) a) (((k) e) (f) b) r)**

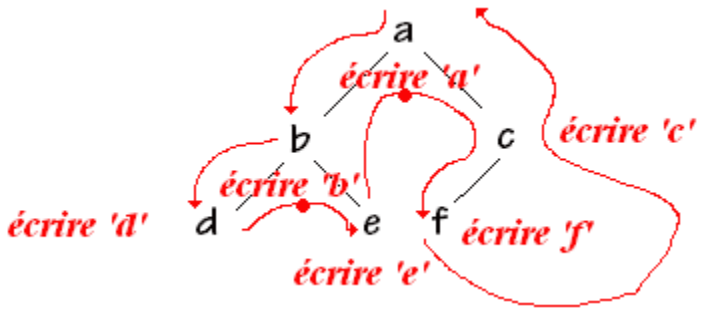
  Exemple 2:



- L'écriture **infixée** : de gauche à droite. *La complexité est dans le pire des càs de l'ordre O(n).*

  Exemple 1: **c, h, a, i, d, l, j, r, k, e, b, f.**

  Exemple 2 :

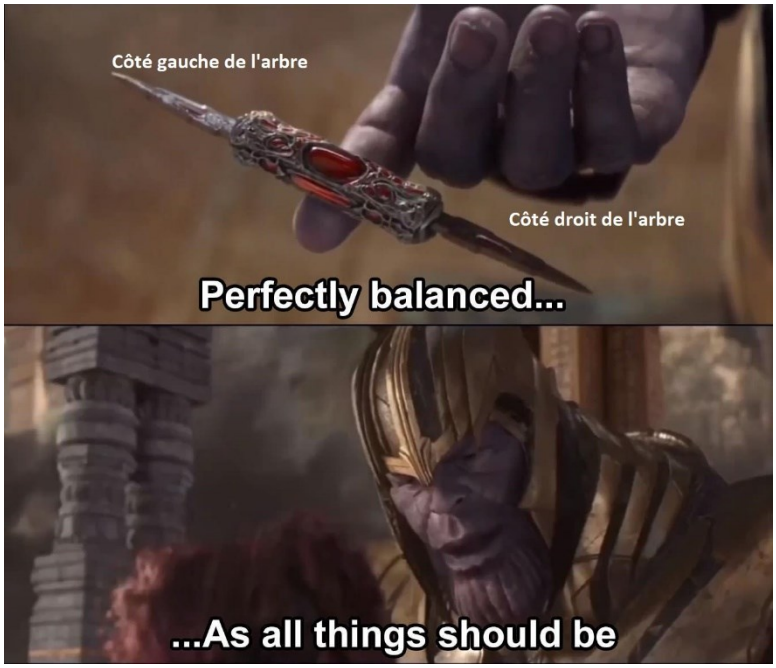


L'EQUILIBRAGE (POURQUOI EQUILIBRER ?)

Il est nécessaire d'équilibré un arbre si l'on souhaite optimiser la vitesse à laquelle on le parcourt et donc la vitesse à laquelle on obtient une donnée.

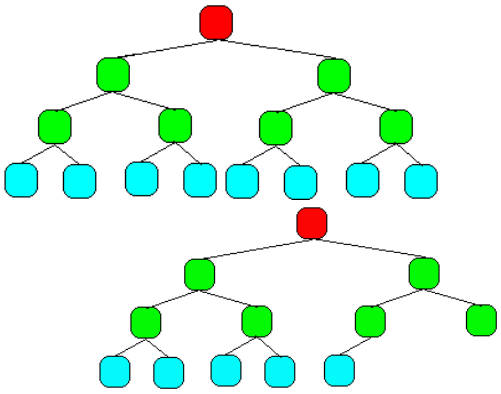
En effet les arbres équilibrés aussi appelé arbres parfaits maintiennent un nombre de niveaux équilibré voir égale pour chaque branche, cela permet en moyenne de réduire le nombre de pas nécessaire pour accéder à la donnée d'un nœud.

LES ARBRES PARFAITS



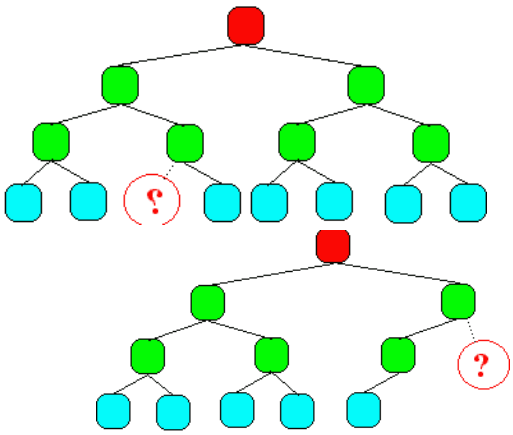
Par définition, un arbre parfait est un arbre dont tous les nœuds de tous niveau sont présents cependant, un arbre parfait peut tolérer une différence de 1 niveau entre ses branches, à condition que toute ces feuilles soient regroupées du côté gauche de l'arbrec'est-à-dire des nœuds terminaux, dans ce cas l'arbre parfait est un arbre binaire incomplet et **les feuilles du dernier niveau doivent être regroupées le plus à gauche de l'arbre**.

Exemple d’arbre parfait :



Arbre parfait complet
Arbre parfait incomplet

Exemple d’arbre non-parfait :

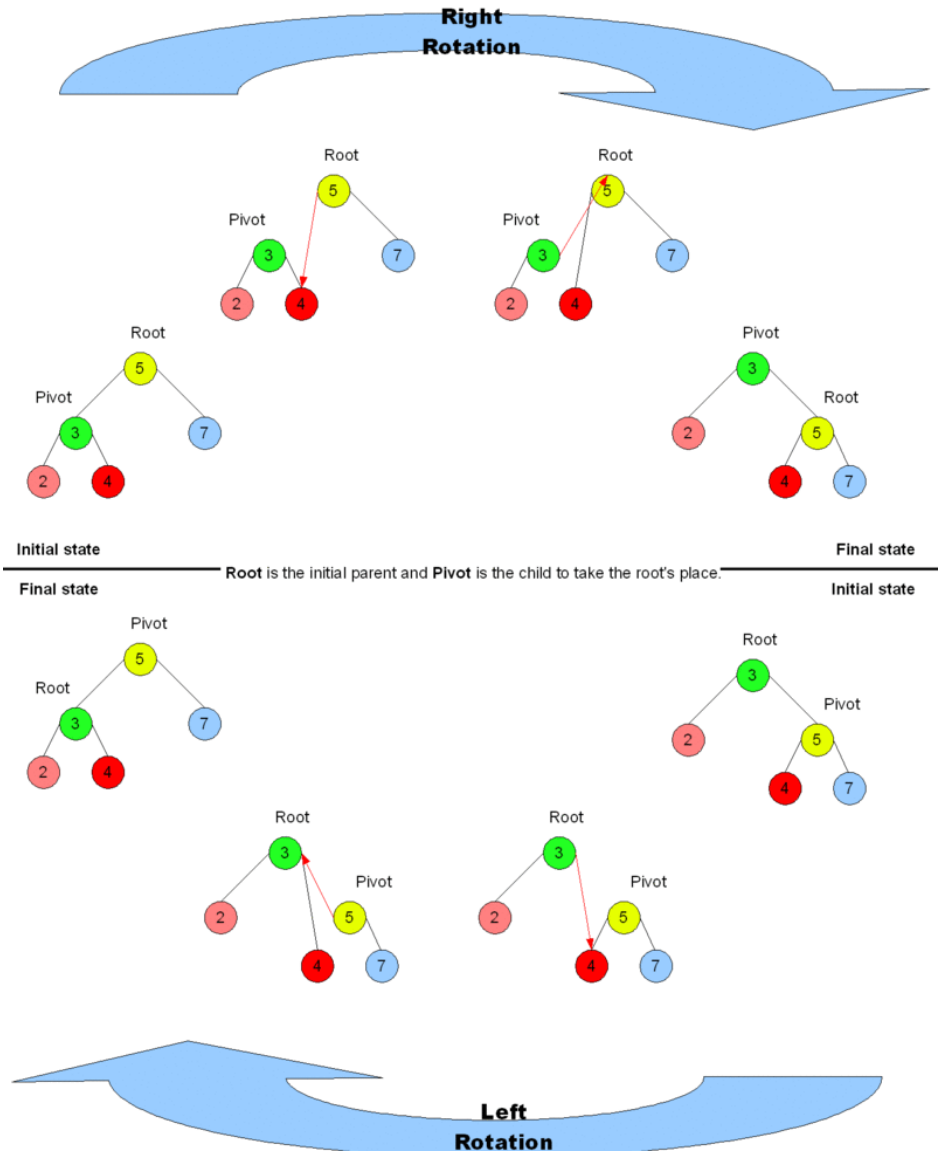


Les feuilles ne sont pas toutes à gauche

Les branches n’ont pas toutes le même niveau.

Equilibrage par rotation :

Un arbre peut être rééquilibré en utilisant des méthodes de rotations de ces branches. Après une rotation, un côté va augmenter son niveau de 1 tandis que le côté opposé va diminuer son niveau de manière équivalente. Ainsi on peut méthodiquement rééquilibrer par rotation les nœuds dont le niveau des enfants de gauches et de droites diffère de plus d'un.

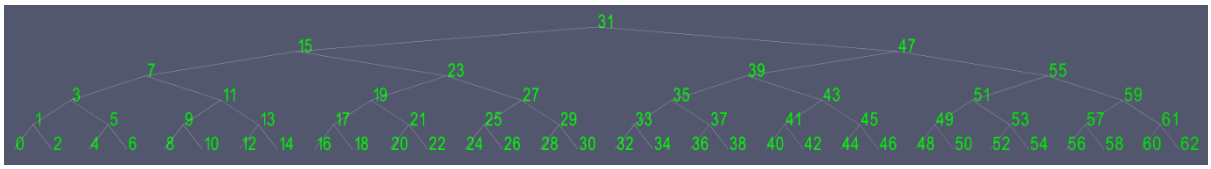


AUGMENTER ENCORE LA VITESSE DE RECHERCHE

Il existe plusieurs moyens augmenter la vitesse de recherche des arbres n-aire, nous avons déjà vu que l'équilibrage d'un arbre s'avère essentielle si l'on souhaite optimiser son arbre un minimum.

L'équilibrage s'effectue généralement une fois que l'arbre est fini, du moins une fois que de nombreux nœuds ont été placés. Cependant, dès la construction de l'arbre, les nœuds doivent être penser pour l'arbre puisse être traversé de la manière la plus optimale,

Exemple :



Ceci est la manière la plus optimale pour ranger 62 nombre dans un arbre.

Une autre méthode particulière pour augmenter la vitesse de recherche utilisé dans des structures similaires aux arbres mais dont les feuilles forment une liste chaînée : la skip list. Elle possède aussi un critère d'équilibre ; mais celui-ci est probabiliste, et non déterministe. C'est-à-dire que le plus souvent une donnée est recherché, le plus proche de la racine elle sera classée. On peut imaginer sa mise en pratique pour des auto-correcteurs ainsi que pour les poropositions des barres de recherches.

EN PRATIQUE

QUELQUES EXEMPLES D'UTILISATION DES ARBRES :

- **Arbre binaire de recherche** (Binary Search Tree) – Ils sont utilisés dans de nombreuses opérations de recherche, où la donnée entre et sort constamment, comme pour les map et les set d'objets dans la bibliothèque de nombreux langage.
- **Partition binaire de l'espace** (Binary Space Partition)- Ils sont utilises dans la la plupart des jeux-vidéo 3D pour déterminer quel objet doit être affiché au rendu.
- **Arbre Binaire** (Binary Tries) – Ils sont utilisés dans la plupart des routeurs à large bande passante, pour stocker les tables de routage.
- **Arbre de hachage** (Hash Trees) Ils sont utilisés dans les programmes qui utilise le peer to peer (p2p) et dans lequelle on doit verifier le hachage mais que l'intégralité du fichier n'est pas disponible.
- **Pile ou Tas** (Heaps) – Ils sont utilisés dans l'implémentation de file de priorité efficace, qui sont utilisés pour les processus dans de nombreux système d'exploitation, dans les algorithmes de path-finding, pour les jeux-vidéo, la robotique ou le tri.
- **Codage de Huffman** (Huffman Coding Tree)- Ils sont utilisés dans les algorithmes de compression comme le .jpeg ou le .mp3.
- **Générateur de pseudos pseudo-aléatoire** (GGM Trees) – Ils sont utilisés dans la cryptographie pour générer un arbre de nombres pseudo-aléatoire.
- **Arbre de Syntaxe** (Syntax Tree) – Ils sont utilisés dans les programmes de vérifications orthographique, syntaxique et de grammaire.

- **Arbre binaire de recherche randomisés** (Treap)- Ce sont des structures de données randomisés utilisées dans les connections sans-fil et les allocations mémoire.

- **T-Tree** (T-tree)- Presque toute les bases de données utilises des forms de B-Tree pour stocker les données sur les disquess dur, mais elles utilise le T-Tree quand elle utilise ces données dans leur mémoire de travail.

D'AUTRES EXEMPLES:

- Akinator et tous les jeux qui consiste à penser à un mot pour que la machine le devine en posant des questions au joueurs stock leurs données dans des arbre binaires dont le joueur peut directement renseigner des nœuds si le joueur apprend un nouveau mot au jeu. Il peut même utiliser des critères de probabilité pour suggérer des mots sans atteindre le niveau des feuilles.
- L'arborescence des fichiers est basée sur des arbres N-Aires.
- Toutes les intelligences artificielles sont composé d'arbre de décision.

LES DIFFERENTES METHODES

ALGORTIHME DES METHODES

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

PSEUDO-CODE D'UN PARCOURS PREFIXE :

```
visiterPréfixe(Arbre A) :
    visiter (A)
    Si nonVide (gauche(A))
          visiterPréfixe(gauche(A))
    Si nonVide (droite(A))
          visiterPréfixe(droite(A))

//Ceci affiche les valeurs de l'arbre en ordre préfixe. Dans cet ordre,
chaque nœud est visité ainsi que chacun de ses fils.
```

PSEUDO-CODE D'UN PARCOURS POSTFIXE OU SUFFIXE:

```
visiterPostfixe(Arbre A) :
    Si nonVide(gauche(A))
       visiterPostfixe(gauche(A))
    Si nonVide(droite(A))
       visiterPostfixe(droite(A))
    visiter(A)
// Dans un parcours postfixe ou suffixe, on affiche chaque nœud après
avoir affiché chacun de ses fils.
```

PSEUDO-CODE D'UN PARCOURS INFIXE

```
visiterInfixe(Arbre A) :
    Si nonVide(gauche(A))
       visiterInfixe(gauche(A))
    visiter(A)
    Si nonVide(droite(A))
       visiterInfixe(droite(A))

// Un parcours infixe visite chaque nœud entre les nœuds de son sous-
arbre de gauche et les nœuds de son sous-arbre de droite. C'est une
manière assez commune de parcourir un arbre binaire de recherche, car
il donne les valeurs dans l'ordre croissant.
```

PSEUDO CODE D'UN 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, Last In, First Out autrement dit : « 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) != null) Alors
```

```
            ajouter(droite(nœud), S)
        Si (gauche(nœud) != null) Alors
            ajouter(gauche(nœud), S)
    }
}
```

PSEUDO CODE D'UN 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, First in, first out autrement dit « 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) != null) Alors
            enfiler(gauche(nœud), f)
        Si (droite(nœud) != null) Alors
            enfiler(droite(nœud), f)
    }
}
```

EXEMPLE PRATIQUE D'UN ARBRE BINAIRE AVEC PARCOURS PREFIXE

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct noeud
{
    int nValeur;
    struct noeud *gauche;
    struct noeud *droite;
} noeud;

//Prototypes des fonctions
void ajouterNoeud(noeud **arbre, int nValeur);
int chercherNoeud(noeud *arbre, int nValeur);


//Programme principal (MAIN)
//BUT : Créer et gérer un arbre binaire
//ENTREE : L'environnement
//SORTIE : L'environnement
int main()
{
    noeud *Arbre = NULL;
    ajouterNoeud(&Arbre, 30);

    if(chercherNoeud(Arbre, 1))
    {
        printf("Il y a la valeur 1 dans l'arbre !");
    }
    else
    {
        printf("Il n'y a pas la valeur 1 dans l'arbre !");
    }
    return 0;
}
```

```c
//Implémentation de la méthode
//BUT : Ajouter un noeud au sein de l'arbre
//ENTREE : L'arbre et la valeur à insérer dans le noeud
//SORTIE : L'arbre modifié
//NOTE : Donc si l'élément n'est pas le premier, on boucle (boucle do while) afin d'avancer de nœud en nœud jusqu'à atteindre un emplacement
//libre (pointeur à NULL) et à chaque nœud on part à droite, si la clé est supérieure à celle du nœud courant, ou à gauche si elle est inférieure
//ou égale à celle du nœud courant.
void ajouterNoeud(noeud **arbre, int nValeur)
{
    noeud *tmpNoeud;
    noeud *tmpArbre = *arbre;

    noeud *elem = malloc(sizeof(noeud));
    elem->nValeur = nValeur;
    elem->gauche = NULL;
    elem->droite = NULL;

    if(tmpArbre)
    do
    {
        tmpNoeud = tmpArbre;
        if(nValeur > tmpArbre->nValeur )
        {
            tmpArbre = tmpArbre->droite;
            if(!tmpArbre) tmpNoeud->droite = elem;
        }
        else
        {
            tmpArbre = tmpArbre->gauche;
            if(!tmpArbre) tmpNoeud->gauche = elem;
        }
    } while(tmpArbre);
    else  *arbre = elem;
}
```

```
72    //Implémentation de la méthode
73    //BUT : Chercher une valeur dans l'arbre
74    //ENTREE : L'arbre et la valeur
75    //SORTIE : Un booléen
76    //NOTE : On suit le cheminement en partant à droite ou à gauche selon la valeur de la clé. À chaque nœud on vérifie si on est en présence
77    //de l'élément recherché, si oui on retourne la valeur 1. Quand on arrive au bout de la branche si on ne l'a pas trouvé on retourne 0.
78    int chercherNoeud(noeud *arbre, int nValeur)
79    {
80        while(arbre)
81        {
82            if(nValeur == arbre->nValeur)
83            {
84                return 1;
85            }
86
87            if(nValeur > arbre->nValeur)
88            {
89                arbre = arbre->droite;
90            }
91            else
92            {
93                arbre = arbre->gauche;
94            }
95        }
96        return 0;
```

EXEMPLE PRATIQUE D'UN ARBRE N-AIRE

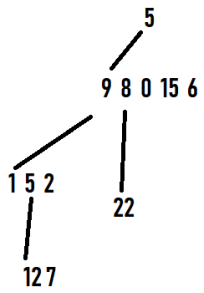


Figure 3 L'arbre que l'on cherche à obtenir.

ALGORITHME D'UN ARBRE N-AIRE

```
//definition du type structuré
TYPE noeud (
        INT valeur
        noeud enfant
        noeud frere
)

//initialise la racine
PROCEDURE initRacine (VAR noeud racine, INT valeurInit)
        racine.valeur -> valeurInit
        racine.frere -> NULL
        racine.enfant -> NULL
FINPROCEDURE

//ajoute un noeud
PROCEDURE addNoeud(VAR noeud pere, INT valeur)
```

```
        noeud newNoeud
        //allocation mémoire

        newNoeud.valeur -> valeur
        newNoeud.frere -> NULL
        newNoeud.enfant -> NULL

        SI pere.enfant = NULL ALORS
                pere.enfant = newNoeud
        SINON
                noeud grandFrere
                //allocation mémoire
                grandFrere -> pere.enfant
                TANT QUE (grandFrere.frere <> NULL) FAIRE //différent de
NULL
                        grandFrere -> grandFrere.frere
                FINTANTQUE
                grandFrere.frere -> newNoeud
        FINSI
FINPROCEDURE

//afficher l'arbre
PROCEDURE affichageArbre(VAR noeud *monNoeud)
        ECRIRE(monNoeud.valeur)
        SI (monNoeud.frere <> NULL) ALORS
                ECRIRE(" --")
                affichageArbre(monNoeud.frere)
        FINSI
        SI (monNoeud.enfant <> NULL) ALORS
                ECRIRE("Enfant de ", monNoeud.valeur)
                affichageArbre(monNoeud.enfant)
        FINSI
FINPROCEDURE

//programme principale
DEBUT
        noeud racine
        //allocation mémoire

        noeud pere
        //allocation mémoire

        initRacine(racine, 5)
```

```
        addNoeud(racine, 9)
        addNoeud(racine, 8)
        addNoeud(racine, 0)
        addNoeud(racine, 15)
        addNoeud(racine, 6)

        //pere = premier enfant de racine
    pere -> racine.enfant;
        addNoeud(pere, 1)
        addNoeud(pere, 5)
        addNoeud(pere, 2)

        //pere = deuxieme enfant de racine
    pere -> racine.enfant.frere;
        addNoeud(pere, 22)

        //pere -> deuxieme enfant du premier enfant de la racine
        pere -> racine.enfant.enfant.frere;
        addNoeud(pere, 12)
        addNoeud(pere, 7)

        affichageArbre(racine);

        //libération mémoire noeud racine
        //libération mémoire noeud pere
FIN
```

CODE PRATIQUE D'UN ARBRE N-AIRE

```
//noeud.h //prototype des méthode de gestion de l'arbre
#ifndef NOEUD_H_INCLUDED
#define NOEUD_H_INCLUDED

#include <stdio.h>
#include <stdlib.h>

typedef struct noeud{
    int valeur;
    struct noeud *enfant;
    struct noeud *frere;
}noeud;

extern void initRacine(noeud *noeudInit, int valeurInit);
```

```
extern void addNoeud(noeud *pere, int valeur);
extern void affichageArbre(noeud *monNoeud);

#endif // NOEUD_H_INCLUDED

//nœud.c //méthode de gestion des arbres
#include "noeud.h"

//initialise la racine
void initRacine(noeud *racine, int valeurInit){
    racine->valeur = valeurInit;
    racine->frere = NULL;
    racine->enfant = NULL;
}

//ajoute un noeud
void addNoeud(noeud *pere, int valeur){
    noeud *newNoeud;
    newNoeud = (noeud *) malloc (sizeof (noeud));

    newNoeud->valeur = valeur;
    newNoeud->frere = NULL;
    newNoeud->enfant = NULL;

    if(pere->enfant == NULL){
        pere->enfant = newNoeud;
    } else {
        noeud *grandFrere;
        grandFrere = (noeud *) malloc (sizeof (noeud));
        grandFrere = pere->enfant;
        while (grandFrere->frere != NULL){
            grandFrere = grandFrere->frere;
        }
        grandFrere->frere = newNoeud;
    }
}

//affiche l'arbre
void affichageArbre(noeud *monNoeud){
    printf("%d",monNoeud->valeur);
    if(monNoeud->frere!=NULL){
        printf(" -- ");
        affichageArbre(monNoeud->frere);
    }
```

```
    if(monNoeud->enfant!=NULL){
        printf("\nEnfant de %d :",monNoeud->valeur);
        affichageArbre(monNoeud->enfant);
    }
}
```

```
//main.c //test d'un nœud et affichage
#include "noeud.h"

int main()
{
    noeud *racine;//noeud à la base de tout
    racine = (noeud *) malloc (sizeof (noeud));

    noeud *pere;//noeud servie pour selectionner un pere qui n'est pas
la racine
    pere = (noeud *) malloc (sizeof (noeud));

    initRacine(racine, 5);

    addNoeud(racine, 9);
    addNoeud(racine, 8);
    addNoeud(racine, 0);
    addNoeud(racine, 15);
    addNoeud(racine, 6);

    //pere = racine->enfant = premier enfant de racine = 9
    pere = racine->enfant;
    addNoeud(pere, 1);
    addNoeud(pere, 5);
    addNoeud(pere, 2);

    //pere = racine->enfant->frere = deuxieme enfant de racine = 8
    pere = racine->enfant->frere;
    addNoeud(pere, 22);

    //pere = racine->enfant->enfant->frere = deuxieme enfant du premier
enfant de la racine = 5
    pere = racine->enfant->enfant->frere;
    addNoeud(pere, 12);
    addNoeud(pere, 7);
```

```
    affichageArbre(racine);

    free(racine);
    free(pere);
    return 0;
}
```

LEXIQUE

Racine = nœud à l'origine de l'arbre

Feuilles = nœuds terminaux situé à l'extrémité basse de l'arbre

BIBLIOGRAPHIE

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

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

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

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

https://fr.wikipedia.org/wiki/Rotation_d'un_arbre_binaire_de_recherche

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

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

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

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

https://rmdiscala.developpez.com/cours/LesChapitres.html/Cours4/TArbrechap4.6.htm

https://stackoverflow.com/questions/2130416/what-are-the-applications-of-binary-trees

https://www.gamedev.net/articles/programming/general-and-gameplay-programming/trees-part-2-binary-trees-r1433/