Arbres

Plan

1. Définitions
2. Arbres binaires
3. Arbres de recherche binaires
4. Implémentation en contigu
5. Quelques exemples d'application des arbres binaires
6. Arbres m-aire

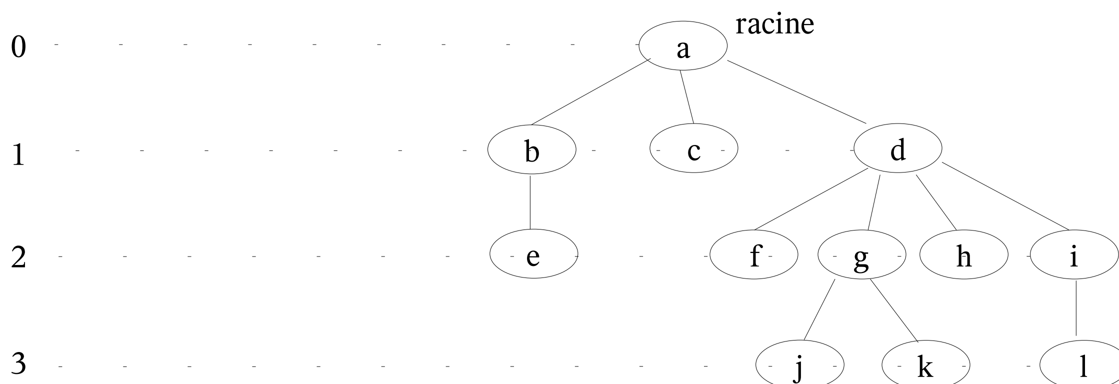
1) Définitions

Un arbre est une structure de données (souvent dynamique) représentant un ensemble de valeurs organisées hiérarchiquement. Chaque valeur est stockée dans un noeud. Les noeuds sont connectés entre eux par des relations parent/fils.

A part le noeud racine, tous les autres noeuds ont exactement un seul noeud parent et zéro ou plusieurs noeuds fils.

Le noeud racine n'a pas de parent et possède zéro ou plusieurs fils.

Les noeuds qui n'ont pas de fils sont appelés feuilles (ou noeuds externes), les autres (ceux qui ont au moins un fils) sont appelés noeuds internes.



Dans l'exemple de la figure ci-dessus, la racine de l'arbre est le noeud contenant l'information 'a'.
Les feuilles de cet arbres sont les noeuds contenant respectivement les informations suivantes :
'e', 'c', 'f', 'j', 'k', 'h' et 'l'
Les noeuds contenant les informations 'j' et 'k' sont les fils du noeud contenant 'g' qui est aussi le fils du noeud contenant 'd', lui même le fils de la racine.

Les noeuds issus d'un même parent direct sont des noeuds frères. Dans l'exemple précédent, les noeuds contenant 'b', 'c' et 'd' sont frères, de même que ceux contenant 'f', 'g', 'h' et 'i'. Les noeuds contenant 'j' et 'k' sont aussi des frères.

Une branche est une suite de noeuds connectés de père en fils (de la racine à une feuille). Par

exemple a-b-e est une branche de l'arbre précédent, de même que a-c, a-d-f, a-d-g-j, ...etc.

Le niveau d'un noeud est la distance qui le sépare de la racine. Par exemple, dans la figure précédente, le niveau du noeud contenant 'g' est égal à 2.

La profondeur d'un arbre (ou sa hauteur) est le plus grand niveau (c-a-d la distance entre la racine et la feuille la plus lointaine). Dans l'exemple précédent, la profondeur de l'arbre est égal à 3.

- Définition récursive d'un arbre

Cas particulier:

NIL est un arbre (l'arbre vide, contenant zéro noeud)

Cas général:

si n est un noeud et si T1, T2, ...Tm sont des arbres, alors on peut construire un nouvel arbre en connectant T1, T2, ...Tm comme des fils à n. chaque Ti est définit de la même manière (récursivement). T1, T2, ... Tm sont alors des sous-arbres de n

Les descendants d'un noeud n sont tous les noeuds appartenant aux sous-arbres de n. Dans l'exemple, les descendants de d sont : f,g,h,i,j,k et l.

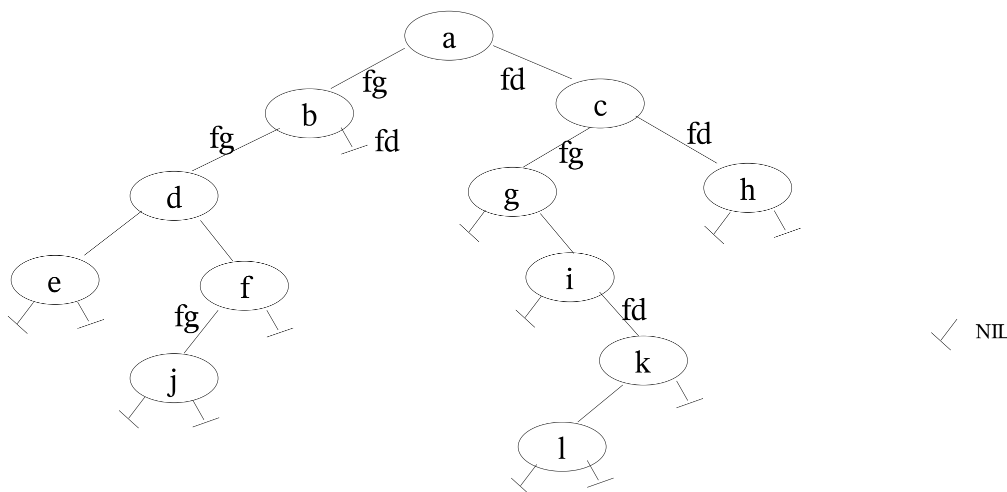
Les ascendants d'un noeud, sont tous les noeuds dont il est descendant. Dans l'exemple, les ascendants de k sont : g,d et a.

Le degré d'un noeud est le nombre de ses fils différents de NIL. Par exemple le degré de d est 4, celui de a est 3 alors que le degré de c est 0.

Une forêt est un ensemble d'arbres.

2) Arbres binaires

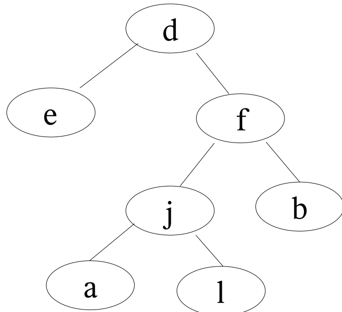
Un arbre binaire est un arbre où chaque noeud est connecté à deux sous-arbres (un sous-arbre gauche et un sous-arbre droit). Ainsi le premier fils d'un noeud n est appelé fils-gauche (fg) et le deuxième fils est appelé fils-droit (fd).



Dans cet exemple (la figure ci-dessus), le fils-droit du fils-gauche de la racine est un arbre vide (le

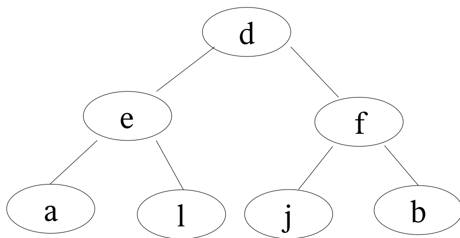
NIL à droite de b). De même le sous-arbre droit de la racine est un arbre binaire de racine un noeud contenant l'information 'c'... etc.

Un arbre binaire est dit 'strictement binaire' si chaque noeud interne a exactement 2 fils différents de NIL. (voir fig. ci-dessous)



Dans un arbre strictement binaire, le nombre de feuilles est toujours égal au nombre de noeuds internes + 1. Dans l'exemple précédent, il y a 4 feuilles (e, a, l et b) et 3 noeuds internes (d, f et j).

Un arbre binaire est dit 'complet' (ou complètement équilibré), s'il est strictement binaire et si toutes les feuilles se trouvent au même niveau :



Dans un arbre binaire complet de profondeur d :

le nombre total de noeuds = $2^0 + 2^1 + 2^2 + \dots 2^d = 2^{d+1}-1$

le nombre de noeuds internes = 2^d-1

le nombre de feuilles = 2^d

le nombre de noeuds dans le niveau i = 2^i

Ainsi on peut établir une équation entre la profondeur (d) d'un arbre binaire complet et le nombre total de noeuds (n) : $\mathbf{d = \log_2(n+1) - 1}$

- Modèle des arbres binaires

Dans les opérations formant le modèle des arbres binaire, les paramètres p et q sont des pointeurs vers des noeuds alors que v est un type quelconque (l'information ou la valeur du noeud).
Le noeud est alors formé d'au moins trois champs : l'information (typeqlq), le fg (ptr) et le fd (ptr).

CreerNoeud(v) : ptr // alloue un nouveau noeud contenant v comme information
// et NIL comme fils-gauche et fils-droit
// la fonction retourne l'adresse du nouveau noeud.
LibererNoeud(p) // libère (détruit) le noeud pointé par p

Info(p) : typeqlq // retourne l'information stockée dans le noeud pointé par p
fg(p) : ptr // retourne le contenu du champs 'fg' du noeud pointé par p
fd(p) : ptr // retourne le contenu du champs 'fd' du noeud pointé par p
Aff-info(p, v) // affecte v dans le champs 'info' du noeud pointé par p
Aff-fg(p, q) // affecte q dans le champs 'fg' du noeud pointé par p
Aff-fd(p, q) // affecte q dans le champs 'fd' du noeud pointé par p

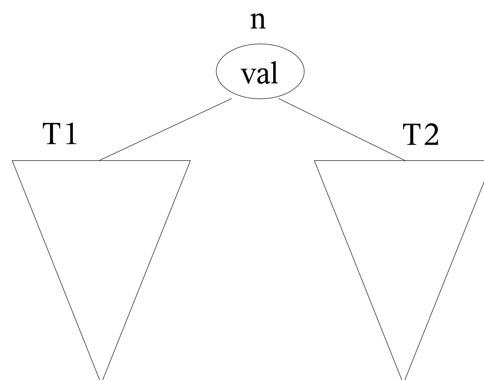
- Parcours d'arbres

Pour consulter ou faire un traitement particulier au niveau des noeuds d'un arbre binaire, il existe plusieurs ordres de parcours, parmi lesquels : le préordre, l'inordre et le postordre.

Ce sont des parcours simples à définir et simples à programmer (en récursif).

Soit n un arbre binaire (pouvant être vide : n=NIL).

S'il n'est pas vide (n pointe le noeud racine), alors il a la forme suivante (avec T1 et T2 des sous-arbres définis de la même manière que n) :



a) Le parcours préordre de n (s'il n'est pas vide) consiste à visiter le noeud racine (n) ensuite parcourir récursivement en préordre les sous-arbres T1 puis T2 --> [n , T1 , T2]

Voici une procédure (récursive) qui affiche les valeurs (informations) de tous les noeuds d'un arbre de racine R en le parcourant en préordre :

```
preordre( R:ptr )
debut
        SI R <> NIL
                ecrire( info(R) );
                preordre( fg(R) );
                preordre( fd(R) )
        FSI
fin
```

b) Le parcours inordre de n (s'il n'est pas vide) consiste d'abord à parcourir récursivement en inordre le sous-arbre gauche T1, puis visiter le noeud racine (n) ensuite parcourir récursivement en inordre le sous-arbre droit T2 --> [T1 , n , T2]

Voici une procédure (récursive) qui affiche les valeurs (informations) de tous les noeuds d'un arbre de racine R en le parcourant en inordre :

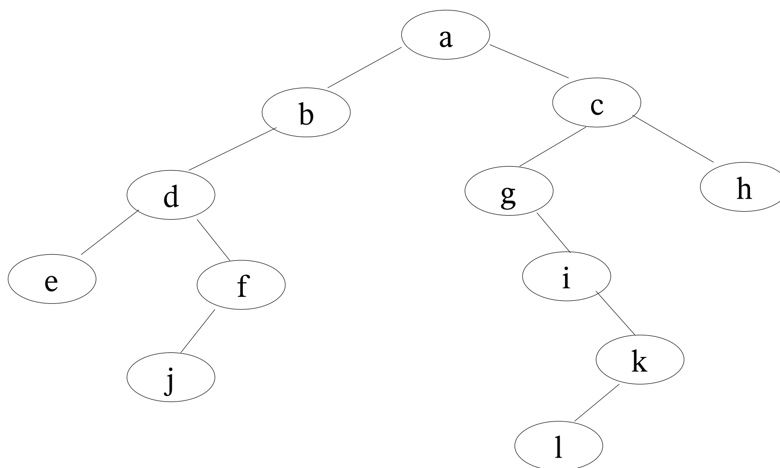
```
inordre( R:ptr )
debut
        SI R <> NIL
                inordre( fg(R) );
                ecrire( info(R) );
                inordre( fd(R) )
        FSI
fin
```

c) Le parcours postordre de n (s'il n'est pas vide) consiste d'abord à parcourir récursivement en postordre les sous-arbres T1 puis T2 ensuite visiter le noeud racine (n) --> [T1 , T2 , n]

Voici une procédure (récursive) qui affiche les valeurs (informations) de tous les noeuds d'un arbre de racine R en le parcourant en postordre :

```
postordre( R:ptr )
debut
        SI R <> NIL
                postordre( fg(R) );
                postordre( fd(R) );
                ecrire( info(R) )
        FSI
fin
```

Si on applique ces trois parcours sur l'arbre ci-dessous :



on aura pour le parcours préordre :
a b d e f j c g i k l h

pour le parcours inordre :
e d j f b a g i l k c h

et pour le parcours postordre :
e j f d b l k i g h c a

remarque : faites beaucoup d'exercices de ce type pour vous familiariser avec ces trois parcours.

On peut faire ces trois parcours sans utiliser la récursivité, il faudra alors un 'moyen' pour pouvoir remonter dans les branches de l'arbre. On pourra par exemple utiliser une structure de pile pour sauvegarder les adresses des noeuds par lesquels on est descendu et les dépiler quand on en aura besoin pour remonter.

On peut aussi enrichir la structure des noeuds en incluant un pointeur vers le noeud père, on pourra alors réaliser ces parcours sans utiliser une pile ou une autre structure de sauvegarde.

- Quelques astuces pour réaliser un parcours inordre itératif (non récursif)

On peut remarquer que le premier noeud visité en inordre dans un arbre de racine R, se trouve être le noeud le plus à gauche dans R (descendre de fg en fg jusqu'à trouver un noeud dont le fg est à NIL).

Le suivant inordre d'un noeud n (d'après la définition récursive) se trouve dans le sous-arbre droit de n (s'il existe). C'est donc le noeud le plus à gauche du sous-arbre droit de n.
Si le sous-arbre droit de n est vide (fd = NIL), alors le suivant de n se trouve plus haut que n (c'est un ascendant de n). Pour trouver de quel noeud il s'agit exactement, il faut encore voir la définition récursive où on remarque qu'un noeud n'est visité (ecrire(infor(R))) qu'après avoir terminer le parcours de son sous-arbre gauche (inordre(fg(R))). Donc le suivant inordre de n quand fd(n)=NIL, est le premier ascendant gauche de n (s'il existe). C'est à dire on remonte à partir de n, de père en père, jusqu'à trouver un noeud en remontant de son fils gauche.

Maintenant il est facile de réaliser le parcours inordre itératif (soit avec une pile soit avec l'opération pere(n)) :
On écrit une fonction 'Prem_Inordre(R) : ptr' qui recherche le noeud le plus à gauche de R, et une deuxième fonction 'Suiv_Inordre(n:ptr) :ptr' qui retourne le suivant inordre de n.
L'algorithme principal serait alors :

```
n := Prem_Inordre(R);
TQ n <> NIL
    ecrire( info(n) );
    n := Suiv_Inordre(n)
FTQ
```

Il existe d'autres types de parcours, comme par exemple le préordre inverse (n T2 T1), l'inordre inverse (T2 n T1) et le postordre inverse (T2 T1 n). Tous ces parcours et ceux qu'on déjà vu sont des parcours en profondeur, ils explorent l'arbre branche par branche. Il existe aussi des parcours dits 'en largeur' qui explorent l'arbre niveau par niveau.
Essayer de lister les valeurs d'un arbre niveau par niveau (en utilisant une file).

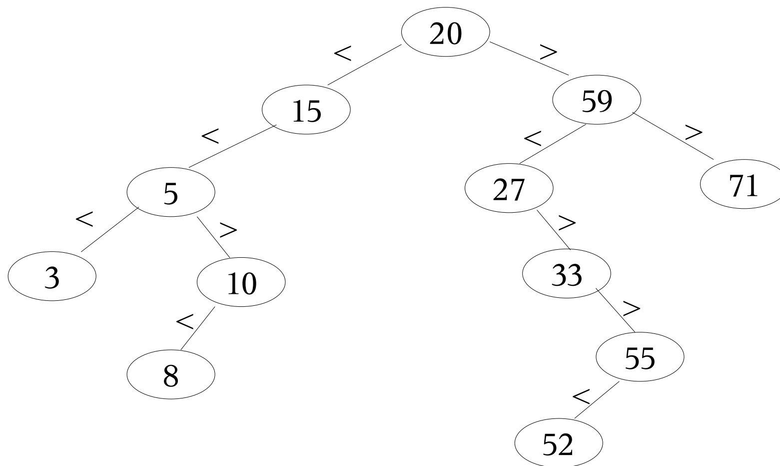
3) Arbres de recherche binaires

C'est des arbres binaires utilisés pour accélérer les opérations de recherche, insertion et suppression de valeurs.

Un arbre binaire de racine R est dit 'de recherche' ssi :
* toutes les valeurs du sous-arbre gauche de R sont inférieurs à info(R)
* toutes les valeurs du sous-arbre droit de R sont supérieurs à info(R)
* le sous-arbre gauche de R (l'arbre de racine fg(R)) doit être un arbre de recherche binaire
* le sous-arbre droit de R (l'arbre de racine fd(R)) doit être un arbre de recherche binaire

Le cas particulier de cette définition récursive est :
NIL (l'arbre vide) est un arbre de recherche binaire.

Toutes les valeurs dans un arbre de recherche sont distinctes (il n'y a pas de valeur en double).
Voici un exemple d'arbre de recherche binaire contenant des valeurs entières :



Pour rechercher une valeur v dans un arbre de recherche R, l'algorithme est simple et rapide :

Si l'arbre est vide (R=NIL), on s'arrête (v n'existe pas)
Sinon on compare v avec l'information de la racine (info(R)) :
    Si c'est égal, on s'arrête (v existe dans R)
    Sinon Si c'est inférieur, on continue récursivement dans le sous-arbre gauche de R
        Sinon c'est supérieur, on continue récursivement dans le sous-arbre droit de R

On remarque que le parcours inordre d'un arbre de recherche, permet de lister les valeurs de l'arbre en ordre croissant. Donc le premier noeud en inordre contient la plus petite valeur de l'arbre de recherche, et le suivant inordre de n contient le successeur en ordre croissant de la valeur info(n).

Pour l'arbre de la figure précédente, le parcours inordre donne la liste ordonnée suivante :
3, 5, 8, 10, 15, 20, 27, 33, 52, 55, 59, 71.

Les opérations d'insertion et de suppression dans un arbre de recherche doivent maintenir la propriété des arbres de recherche :

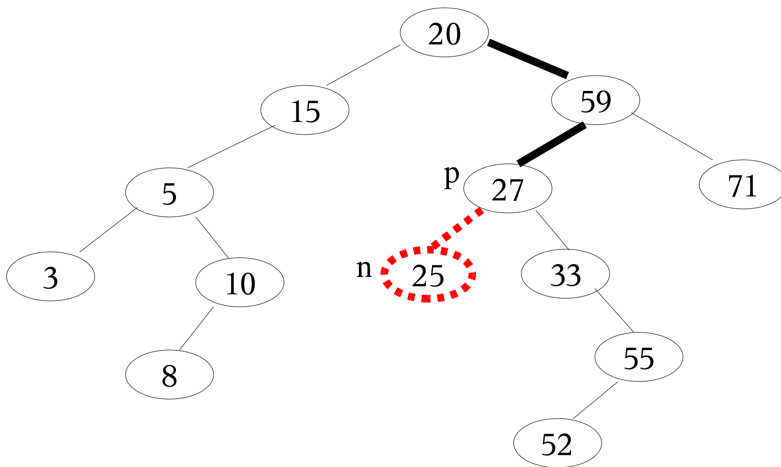
Pour insérer une valeur v dans un arbre de recherche R, on commence par rechercher v pour s'assurer qu'elle n'existe pas déjà dans R (pas de double) et pour localiser l'endroit où elle doit être insérée. Soit p le dernier noeud visité par la recherche, on crée alors un nouveau noeud n contenant v et on connecte n comme fils-gauche de p si v < info(p) ou comme fils droit de p si v > info(p).

Par exemple si on veut insérer 25 dans l'arbre de la figure précédente, on commence par rechercher 25, et voici le déroulement (textuel) de cette recherche :
on compare 25 par rapport à 20 (la racine), c'est supérieur, on descend donc à droite,
on compare 25 par rapport à 59, c'est inférieur, on descend à gauche,
on compare 25 par rapport à 27, c'est inférieur, on descend encore à gauche et on trouve NIL.
On s'arrête à ce niveau, 25 n'existe pas et le dernier noeud visité (p) est celui contenant 27.

Pour insérer 25, il suffit alors de créer un nouveau noeud (n) contenant 25 comme information, et le

connecter comme fils-gauche de p (car 25 est inférieure à 27).

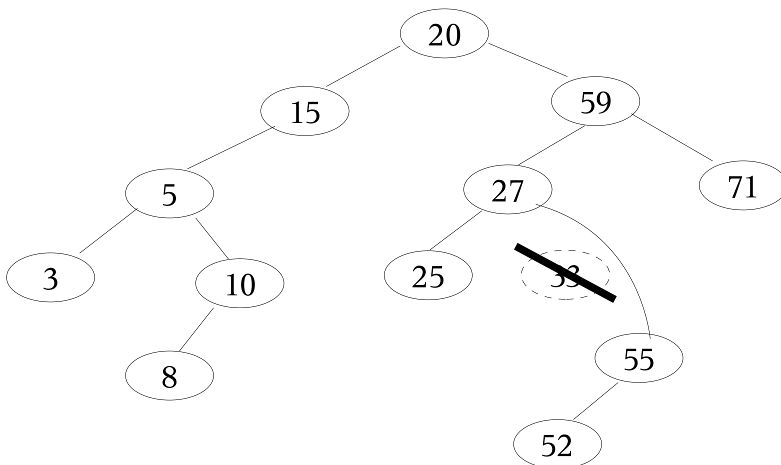
La figure ci-dessous schématise cette dernière phase :



L'insertion dans un arbre de recherche crée toujours une nouvelle feuille. L'arbre grandit vers le bas.

Pour supprimer une valeur v dans un arbre de recherche binaire R, on commence par la rechercher pour trouver le noeud (n) qui la contient, ensuite on procède à la suppression suivant l'un des deux cas suivants :
a) Si n est une feuille ou bien un noeud interne qui ne possède qu'un seul fils différent de NIL, on peut alors supprimer n en mettant à jour son père (le champs fg ou fd) pour remplacer n soit par NIL (si c'était une feuille), soit par l'adresse de l'unique fils de n (sinon).
b) Si n est un noeud avec deux fils différents de NIL, on remplace v à l'intérieur de n par son successeur en ordre croissant et on supprime le noeud qui contenait ce successeur. Comme le successeur de v se trouve dans le suivant inordre de n, il est facile de le localiser, il suffit de descendre une fois à droite de n, puis descendre le plus à gauche possible, jusquà trouver un noeud dont le fg est à NIL.

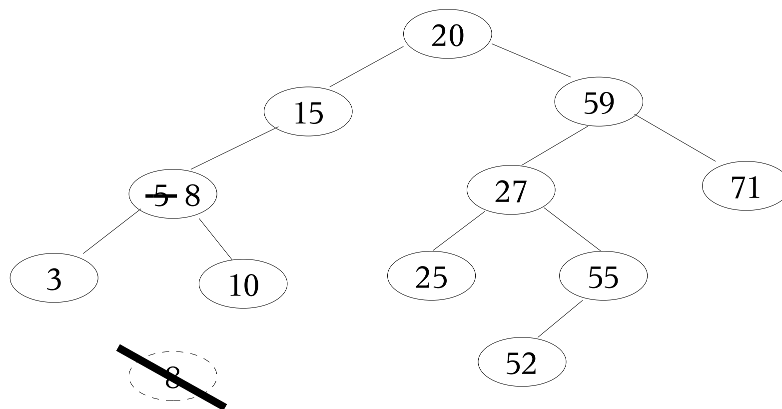
Par exemple, si on supprime 33 dans l'arbre précédent, on modifie le noeud contenant 27 pour pointé directement celui contenant 55 :



Si on supprime la valeur 5 dans l'arbre ci-dessus, on est alors dans le 2e cas (un noeud avec deux

fils différents de NIL), il faut donc remplacer la valeur 5 par son successeur (8), qui se trouve dans le noeud le plus à gauche du sous-arbre droit de 5. Le noeud contenant 8 sera bien sûr supprimé pour ne pas garder de double.

On obtient alors l'arbre suivant :



Toutes les opérations que l'on vient de voir (recherche, insertion et suppression) sont efficaces (rapides) si l'arbre binaire est équilibré (ou presque), car la profondeur serait alors logarithmique en fonction du nombre de noeuds (valeurs). Par exemple, dans un arbre de recherche binaire équilibré contenant 1000 valeurs, il faudrait au maximum 10 tests (ou itérations) pour une recherche, insertion ou suppression. De même qu'il en faudrait 20 tests si l'arbre renfermait 1 000 000 de valeurs, ...etc.

Le problème est que rien n'assure que l'arbre restera toujours équilibré (ou presque), si par exemple on insère, dans un arbre initialement vide, une suite ordonnée de 1000 valeurs, le résultat obtenu sera un arbre complètement 'dégénéré' qui ressemble en fait à une simple liste linéaire chaînée de 1000 maillons. Dans ce cas, le coût des opérations d'accès deviendra linéaire (proportionnel au nombre de valeur), et par exemple une recherche dans un tel arbre coutera au maximum 1000 tests, alors que dans un arbre dégénéré d'un million de valeurs, le coût sera d'un million de tests, ...etc.

Il existe d'autres algorithmes d'insertion et de suppression (un peu plus complexes) permettant de garder l'arbre équilibré dans tous les cas. Parmi les techniques les plus connues, il y a : AVL-trees, RedBlack-trees, B-trees, T-trees, ...etc. On verra dans un autre chapitre, la technique des B-trees qui généralise l'utilisation des arbres de recherche dans le stockage efficace de données volumineuses en mémoire secondaire.

4) Implémentation des arbres binaire en contigu

On présente ici, deux différentes manières de représenter des arbres binaires dans un espace de stockage statique (tableau).

- Représentation standard

Elle ressemble à celle utilisée pour implémenter des listes linéaires chaînées dans un tableau.

Les noeuds sont représentés par les cases d'un tableau ayant une structure d'au moins trois champs : (info : typeqlq, fg et fd : entier), on peut aussi rajouter un quatrième champs de type booléen pour indiquer si la case est libre ou occupée :

```
type
        Tcase = structure
                info : typeqlq;
                fg,fd : entier;
                vide : bool
        Fin

var
        T : tableau[1..N] de Tcase;
```

Avec cette représentation, on peut facilement implémenter le modèle standard des arbres binaire.

Si on veut en plus implémenter l'opération pere(n), on peut rajouter un autre champs de type entier à la structure Tcase.

- Représentation séquentielle

Dans cette reprsentation, on élimine les pointeurs entiers (fg,fd et éventuellement pere), en associant à chaque noeud de l'arbre une position fixe prédéfinie dans le tableau:
la case d'indice 1 sera toujours réservée au noeud racine de l'arbre,
la case d'indice 2 sera toujours réservée au fg de la racine,
la case d'indice 3 sera toujours réservée au fd de la racine,
....
en général, le fg de la case i se trouvera toujours à l'indice 2i et le fd de la case i se trouvera toujours à l'indice 2i+1, alors que le père de la case i il sera toujours positionné à la case i div 2.

On obtient ainsi une représentation compacte (qui consomme peu d'espace) mais pour laquelle, on ne peut pas implémenter le modèle standard des arbres binaire. C'est pour des utilisations un peu particulières (voir plus loin : exemples d'applications).

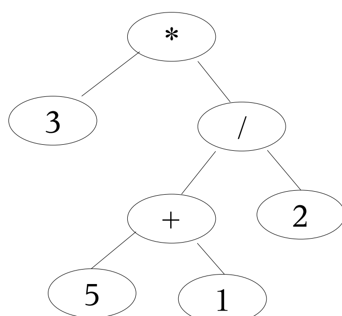
5) Quelques exemples d'applications des arbres binaires

Les arbres binaires sont très utilisés en programmation pour une multitude d'applications, on donne ici un aperçu sur quelques exemples d'utilisation.

- Représentation des expressions arithmétiques

Les expressions arithmétiques peuvent êtres représentées sous forme d'arbre binaire. Les noeuds internes contiennent des opérateurs, alors que les feuilles contiennent des valeurs (opérandes).

Par exemple, l'expression 3*((5+1)/2) sera représentée par l'arbre suivant :

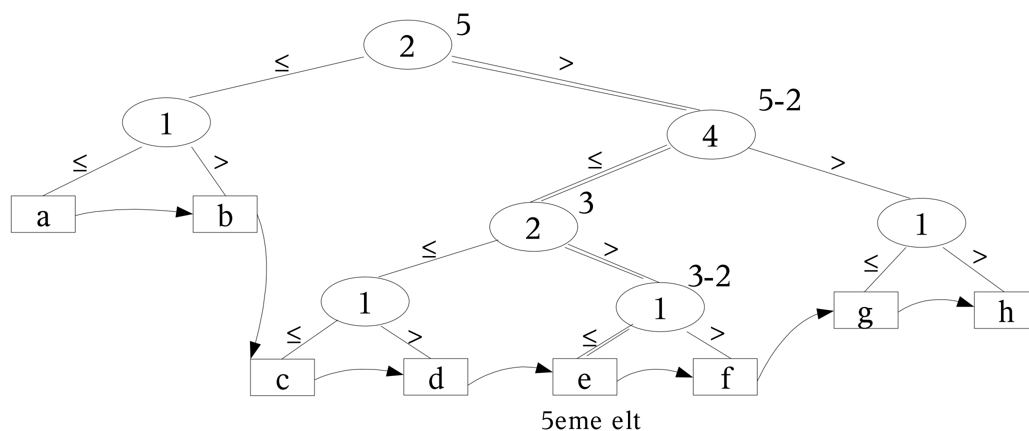


Donner un algorithme récursif pour évaluer une expression sous la forme d'un arbre binaire.

- Accélérer l'accès par position dans une liste

On peut utiliser un genre particulier d'arbre de recherche binaire pour rendre l'accès par position beaucoup plus rapide qu'un s'imple parcours séquentiel.

Dans ce type d'arbre, les feuilles contiennent les valeurs d'une liste donnée, alors que les noeuds internes forment un index de position : chaque noeud interne contient un entier désignant le nombre de feuilles de son sous-arbre gauche.
Cette information sera utilisée pour guider la recherche d'une position donnée :
si la position cherchée est inférieure ou égale à l'information du noeud interne on descend à gauche, sinon on descend à droite et en retranche à la position cherchée l'information du noeud.



Le schéma précédent, montre le déroulement de la recherche de la cinquième position :
On compare 5 par rapport à la racine (2), c'est supérieur, donc on descend à droite en retranchant 2 à la position cherchée, car en allant dans le sous arbre droit, on a écarté 2 feuilles (celles du sous arbre gauche).
Maintenant on recherche la position 3, alors que l'information du noeud courant indique qu'il y a 4 feuille à sa gauche, donc on descend à gauche.
On compare 3 avec l'information du noeud qui indique 2 feuilles à sa gauche, donc on descend à droite et on retranche 2.
Maintenant on recherche la position 1 et comme c'est inférieur ou égal à l'information du noeud courant (1) on descend à gauche.
Etant arrivé au niveau d'une feuille avec la position cherchée égale à 1, on est donc positionné sur le bon élément.

Trouver un moyen d'insérer et de supprimer par position sur telle structure.

Il existe plusieurs manières de construire une telle structure à partir d'un ensemble de valeurs (une liste ou un tableau par exemple).
Essayer de trouver un algorithme récursif qui permet de construire un arbre de ce type aussi equilibré que possible à partir d'un tableau de n éléments.

- Codage de Huffman

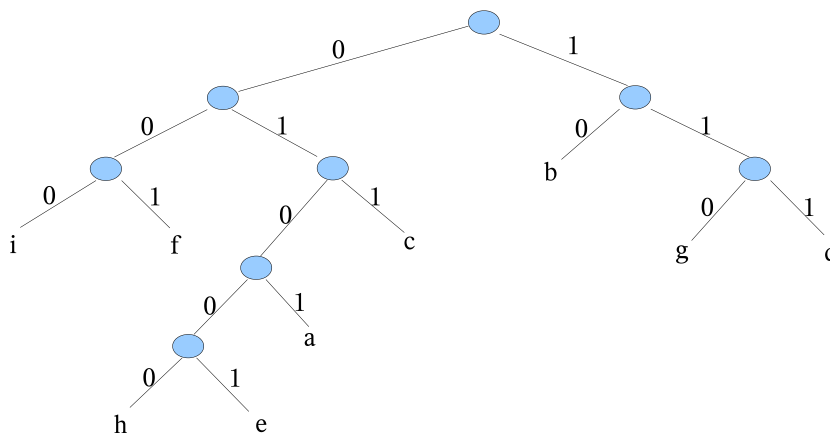
Une autre utilisation des arbres binaires est de représenter des codes de longueurs variables pour les

symboles utilisés dans un message.

L'algorithme de Huffman génère un code binaire de longueur variable pour chaque symbole d'un message de sorte que les symboles ayant une haute fréquence d'apparition (dans le message initial) se voient attribuer des codes de longueurs réduites, alors que les symboles avec une faible fréquence d'apparition sont codés avec un nombre de bits plus important.

L'objectif de cet algorithme est de compresser des fichiers (messages).

1. Au départ on calcule les probabilités d'apparition de chaque symbole (caractère). C'est le nombre d'occurrence du caractère dans le message divisé par la taille du message (en nombre de caractères).
2. A chaque caractère du message, on construit un noeud (initialement isolé) contenant ce caractère.
3. Choisir les 2 noeuds ayant la plus faible probabilité et non encore choisit
4. Créer un nouveau noeud (un caractère fictif) en connectant les 2 noeuds de l'étape précédente comme fils gauche et fils droit du nouveau noeud. Associé à ce nouveau noeud la somme des probabilités de ces 2 fils.
5. Répéter les deux dernières étapes jusqu'à ce qu'il n'y ait qu'un seul noeud non encore traité. C'est alors la racine de l'arbre de Huffman.



L'arbre ci-dessus est un exemple d'un codage à longueur variable pouvant être généré par l'algorithme de Huffman.
Dans cet exemple, le code associé au symbole 'a' est 0101, celui associé à 'h' est 01000 et celui associé à 'b' est 10.

Avec ce type de codage (représenté par un arbre), la propriété du préfixe est toujours vérifiée: « aucun code n'est le préfixe d'un autre » et c'est ce qui permet de décoder n'importe quelle chaîne de bits sans ambiguité.

Par exemple, la chaîne 1001110101011110 ne peut être décodée que d'une seule manière, en la parcourant de gauche à droite, on est guidé dans l'arbre pour atteindre les feuilles. A chaque fois qu'on est sur une feuille, on aura décodé le caractère correspondant, on remonte à la racine et continue le parcours de la chaîne de bit. On obtient alors le message décodé : bcbbbdb

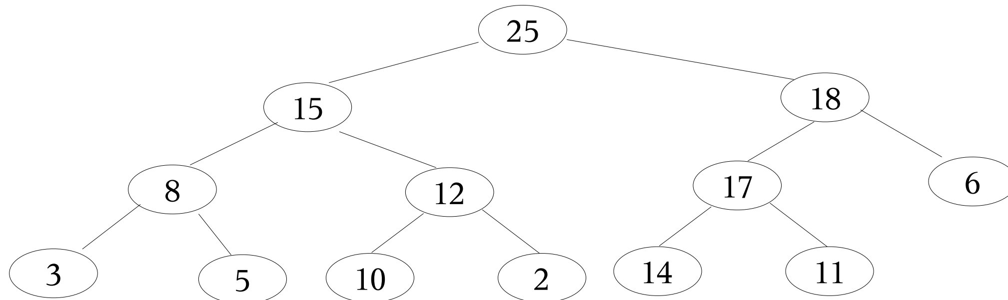
- File de priorité

Une file de priorité est un ensemble où on peut enfiler des valeurs associées à des priorités (des

poids) et quand on défile un élément, on doit retirer de l'ensemble la valeur ayant la plus forte priorité.

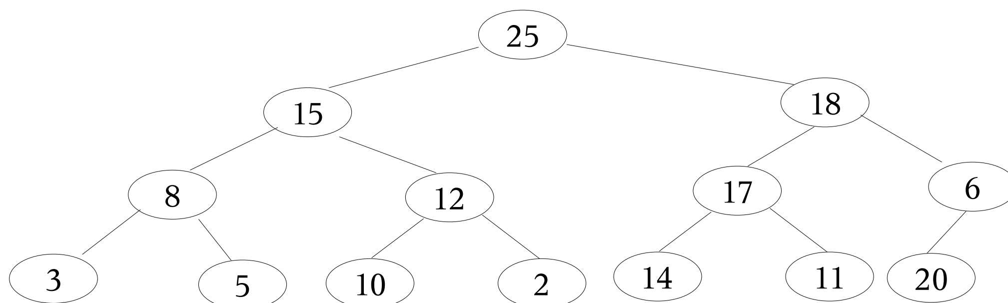
On peut implémenter de manière efficace ce type d'ensemble à l'aide d'un arbre binaire comme suit:

L'arbre est construit niveau par niveau et à chaque niveau est rempli de gauche à droite.
Les noeuds de l'arbre renferment les éléments de la file de priorité tout en respectant la propriété suivante : pour tout noeud interne p, priorité(p) >= priorité(de ses fils)
Voici un exemple :

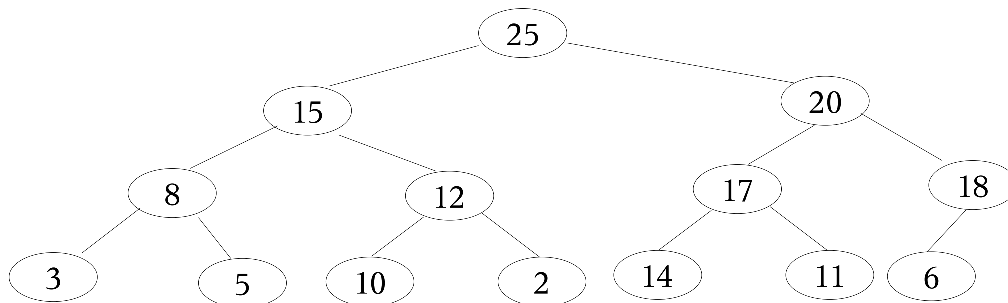


Quand on enfile un nouvel élément, on le rajoute temporairement à la première position libre (c-a-d ajouter une nouvelle feuille dans le dernier niveau et en partant de gauche à droite), ensuite on le fait remonter par une série de permutations avec le père, jusqu'à ce que sa propriété soit <= à celle de son père.

Par exemple, si on enfile un élément de priorité 20, on commence par l'insérer comme fils gauche de 6 (c'est la première position libre dans le dernier niveau)



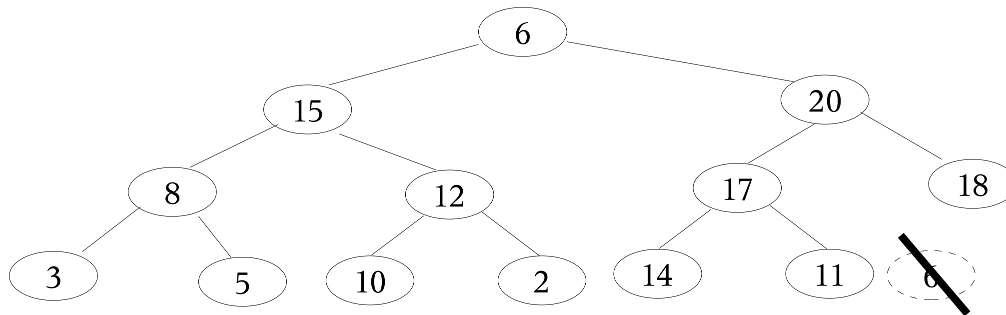
Puis en permute l'élément 20 avec son père (car 20 > 6), et on continue les permutations avec son nouveau père (18) et on s'arrête à ce niveau car 20 <= 25



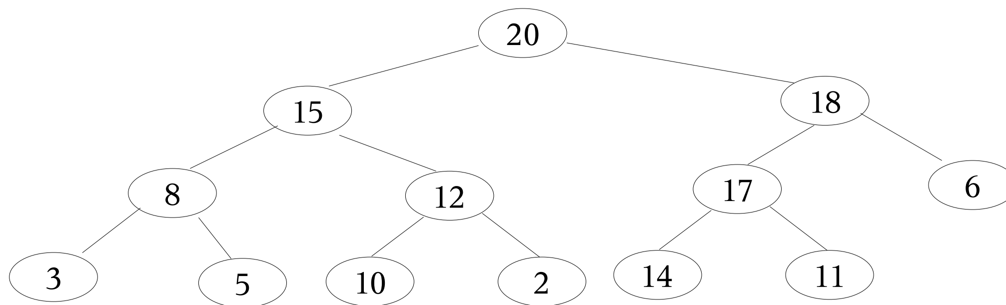
Pour défiler un élément (le plus prioritaire de l'ensemble), il suffit de retirer la racine de l'arbre.

Remplacer temporairement la racine par la feuille la plus à droite.
Puis faire descendre l'élément de la racine temporaire par une série de permutations avec le plus grand des fils, jusqu'à ce que sa priorité soit >= à celles de ses fils.

Dans l'exemple précédent, quand on défile l'élément 25, on le remplace temporairement par la feuille la plus à droite (6) que l'on supprime :



Puis on fait descendre l'élément 6, en le permutant avec 20, puis avec 18 :



Pour implémenter ces opérations de manière efficace, on utilise la représentation séquentielle des arbres binaire.

6) Arbres m-aire

Un arbre m-aire d'ordre d est un arbre où chaque noeud peut avoir un nombre de fils compris entre 0 et d. Si l'ordre n'est pas connu, on dit alors simplement un arbre m-aire. Dans ce cas le nombre de fils est théoriquement illimité.

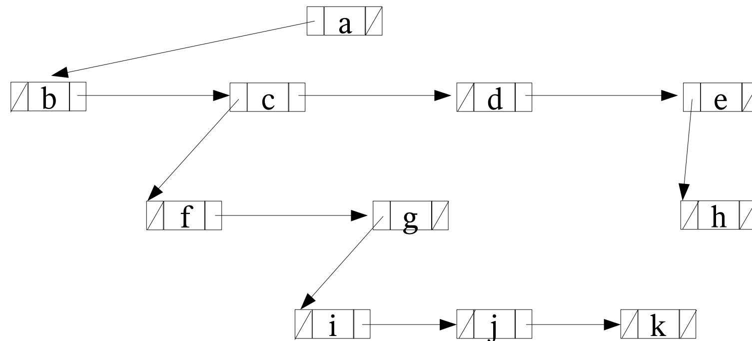
Pour écrire des algorithmes sur ce type d'arbres, on défini un modèle formé par les opérations suivantes :

```
CreerNoeud( v ) : ptr  // crée un noeud isolé contenant v comme information
LibérerNoeud( p )      // détruit le noeud pointé par p
info( p ) : typeqlq    // retourne l'information contenu dans le noeud pointé par p
Fils( i,p ) : ptr      // retourne le ième fils du noeud pointé par p
Aff-info( p,v )        // affecte v comme information du noeud pointé par p
Aff-Fils( i,p,q )      // connecte le noeud pointé par q, comme fils i du noeud pointé par p
degré( p ) : entier    // retourne le nombre de fils du noeud pointé par p
Aff-degré( p,x )       // affecte la valeur x, comme dégré du noeud pointé par p
```

Dans le cas d'un arbre m-aire, où l'ordre n'est pas connu, l'opération Add-degré(...) n'est pas implémentée. De plus, il n'existe pas de fils à NIL dans un tel arbre.

- Implémentation dynamique

Si l'ordre de l'arbre n'est pas connu, l'implémentation standard d'un arbre m-aire en dynamique consiste à allouer des noeuds ayant 3 champs : l'information, le premier fils et le frère droit.



Si l'ordre est connu, il plus simple de réserver un tableau de d pointeur pour représenter les fils.

- Implémentation statique

C'est le même principe qui est utilisé dans l'implémentation dynamique sauf que les noeuds sont localisés à l'intérieur d'un tableau.

Donc soit un tableau à 3 champs principaux (info, premier_fils, et frère_droit) dans le cas où l'ordre est inconnu, soit réservé un tableau de d entier à l'intérieur de chaque case du tableau principal si l'ordre d est connu.

- Arbres de recherche m-aire

C'est la généralisation des arbres de recherche binaires pour être utilisés en mémoire secondaire où les E/S sont « bufférisées ».

Chaque noeud renferme un tableau de d-1 valeurs (ordonnées) et un tableau de d fils, car dans un arbre de recherche m-aire d'ordre d, le nombre de fils d'un noeud (le degré) est toujours égal au nombre de valeur stockées + 1.

Ce type de structures de données étant le plus souvent utilisées comme une organisation de fichiers, on laissera leur présentation dans la partie structures de fichiers.