



Fin de chapitre I

Chapitre suivant : les listes



Chapitre 2: Les listes

- I. Les listes simplement chaînées
- II. Les listes doublement chaînées
- III. La programmation des listes simplement et doublement chaînées
 - Construction
 - Parcours
 - Insertion
 - Suppression
 - Fonctions de manipulation



Références bibliothèques

1. « Exercices et Problème d'algorithmes »
Auteurs: Bruno Baynat et collaborateurs
Code bibliothèque I-15/BAY
Edition Dunod
Année d'édition (2003)
2. « Programmer en Langage C cours et exercices »
Auteur: Claude Delannoy
Code bibliothèque I-22/DEL
Edition Eyrolles
5^{ème} Edition (2012)
3. « Types de données et Algorithmes »
Auteur: Christine Froidevaux et collaborateurs
Code bibliothèque I-1/FRO
Edition McGraw-Hill
Année d'édition (1990)

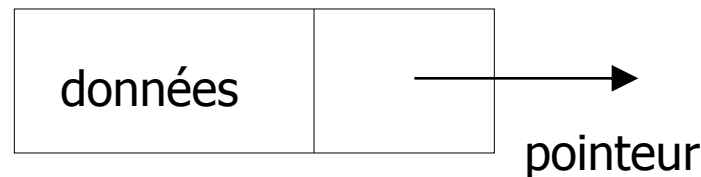
Les listes simplement chaînées

Définition

Une liste chaînée possède une **tête** (début), une **queue** (fin) et un ensemble d'éléments.

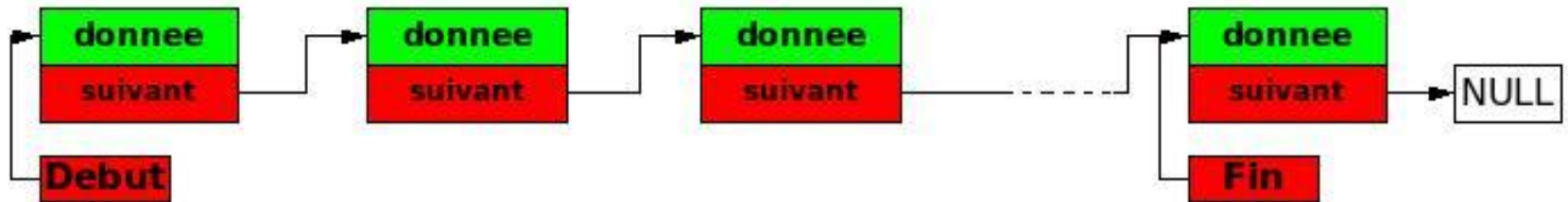
Chaque élément (**maillon**) contient l'information mémorisé et un pointeur sur l'élément suivant.

L'élément suivant de la queue de la liste n'existe pas, la valeur NULL en C.



Les listes simplement chaînées

Les listes sont un regroupement ordonné de données structurées de manière à ce que chaque composante sache où se trouve la suivante.



Une composante sera une structure contenant la valeur de l'information mémorisée, mais également un pointeur sur la composante suivante. Une liste est accessible par l'adresse de sa première composante.

On supposera dans la suite que les valeurs à mémoriser sont d'un type structures, composé de plusieurs champs, nommé information.

Les listes chaînées peuvent être utilisées quand plusieurs opérations d'insertion et/ou suppression d'éléments sont nécessaires.

Les listes simplement chaînées



La notion de liste chaînée n'est pas liée à une manière concrète de matérialiser la relation successeur.

Les éléments sont référencés par leurs adresses dans la mémoire.

Chaque élément est alloué dynamiquement, au moment où il commence à être utile.

Contrairement aux tableaux, il se peut que les maillons qui composent la liste ne soit pas placées dans l'ordre en mémoire et encore moins de façon contiguë.



Les listes simplement chaînées

Dans une liste chaînée :

- La taille est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet.
- Pour déclarer une liste chaînée il suffit de créer le pointeur qui va pointer sur le premier élément de votre liste chaînée, aucune taille n'est à spécifier donc.
- Il est possible d'ajouter, de supprimer, d'intervertir des éléments d'une liste chaînées en manipulant simplement leurs pointeurs.
- Il permet de manipuler des ensembles dynamiques.
- Les listes sont adaptées au parcours séquentiel d'un ensemble discret.
- Les listes sont à la base des Files et des Piles (chapitre 3).



Les listes simplement chaînées

Applications:

- La file d'attente de gestion et d'ordonnancement des processus et de spool d'impression dans les systèmes d'Exploitation (S5/Informatique),
- La gestion de la queue de message dans un environnement client-serveur,
- Etc....

Les listes simplement chaînées

Création et déclaration en C d'une liste :


Pour définir un élément de la liste, le type struct sera utilisé.

L'élément de la liste contiendra un champ donnee de type information qui dépend du problème posé et un pointeur suivant.

Le pointeur suivant doit être du même type que l'élément, sinon il ne pourra pas pointer vers l'élément. Le pointeur "suivant" permettra l'accès vers le prochain élément.


```
struct Info
{
    type1 champs1 ;
    type2 champs2 ;
    .... ;
    typen champsn ;
} ;
typedef struct Info Information;
```

Les listes simplement chaînées



```
struct maillon
{
    Information donnee;
    struct maillon *suivant;
};
typedef struct maillon Maillon;
struct liste
{
    Maillon *debut;
    Maillon *fin;
    int taille;
} ;
typedef struct liste Liste ;
Liste *L ; // déclaration d'un objet de type Liste *
L=CreerListe();
```

Les listes simplement chaînées



```
Liste *CreerListe(){
Liste *L=(Liste *)malloc(sizeof(Liste)) ;
if (L==NULL) {printf("Allocation non reussie\n");exit(-1);}
else {L->debut=NULL; L->fin=NULL; L->taille=0;}
return L;
}
Maillon *m;
m=CreerElement();
Maillon *CreerElement(){
Maillon *m;
m=(Maillon *)malloc(sizeof(Maillon)) ;
if (m==NULL) {printf("Allocation non reussie\n");exit(-1);}
else {m->donnee.champs1=0;...; m->donnee.champsn=0;
m->suivant=NULL;}
return m;
}
```

Les listes simplement chaînées



Caractéristiques

- Pour les tableaux les éléments sont contigus dans la mémoire.
- Pour les listes les éléments peuvent être éparpillés dans la mémoire. La liaison entre les éléments se fait grâce à un pointeur. En réalité, dans la mémoire la représentation est aléatoire en fonction de l'espace alloué.
- Le pointeur suivant du dernier élément doit pointer vers NULL (la fin de la liste).
- Pour accéder à un élément, la liste est parcourue en commençant avec la tête, le pointeur suivant permettant le déplacement vers le prochain élément. Le déplacement se fait dans une seule direction (séquentielle asymétrique), du premier vers le dernier élément. On va voir que pour le parcours inverse ou dans les deux directions (séquentielle symétrique) (en avant/en arrière), on utilise les listes doublement chaînées.

Les listes simplement chaînées



Accès aux champs d'une liste

Liste *L ;

L-> champs ; ou (*L).champs;

Exemple

L -> debut;

L -> fin;

L -> taille;

L -> debut -> donnee;

L -> debut -> suivant;

(L -> debut -> donnee).champs1;

(L->debut -> donnee).champs2;

...

etc

Les listes simplement chaînées

A retenir, les points clés des listes chaînées :

- Chaque élément de la liste est formé de $n+1$ champs :
 - n champs constituant l'information portée par le maillon, dépendante du problème particulier considéré,
 - un champ supplémentaire qui est la matérialisation de la relation successeur.
- Le pointeur debut contiendra l'adresse du premier élément de la liste ;
- Le pointeur fin contiendra l'adresse du dernier élément de la liste ;
- La variable entière taille contient le nombre d'éléments de la liste.

Quelque soit la position dans la liste, les pointeurs debut et fin pointent toujours respectivement vers le premier et le dernier élément. Le champ *taille* contiendra le nombre d'éléments de la liste quelque soit l'opération effectuée sur la liste.

Les listes simplement chaînées

Tableaux Vs listes chaînées

Cette question se pose au programmeur dans de nombreuses applications. Lorsque le choix est possible, on le fait en considérant les points suivants.

Avantages des tableaux par rapport aux listes :

- Accès direct. C'est la définition même des tableaux : l'accès au $i^{\text{ème}}$ élément se fait en un temps indépendant de i , alors que dans une liste chaînée ce temps est de la forme $k \times i$ (car il faut exécuter i fois l'opération $p = p \rightarrow \text{suivant}$).
- Pas de surencombrement : La relation successeur étant implicite (définie par la contiguïté des composantes), il n'y a pas besoin d'espace supplémentaire pour son codage. Alors que dans une liste chaînée l'encombrement de chaque maillon est augmenté de la taille du pointeur suivant.



Les listes simplement chaînées

Avantages des listes chaînées par rapport aux tableaux :

- Liberté dans la définition de la relation successeur. Cette relation étant explicite, on peut la modifier aisément (les maillons des listes chaînées peuvent être réarrangés sans avoir à déplacer les informations qu'ils portent).

Autre aspect de la même propriété, un même maillon peut faire partie de plusieurs listes.

- Encombrement total selon le besoin, si on utilise, comme dans les exemples précédents, l'allocation dynamique des maillons. Le nombre de maillons d'une liste correspond au nombre d'éléments effectivement présents, alors que l'encombrement d'un tableau est fixé d'avance et constant.

Les listes simplement chaînées



NOTE.

Une conséquence de l'encombrement constant d'un tableau est le risque de saturation : l'insertion d'un élément échoue parce que toutes les cases du tableau sont déjà occupées.

Bien sûr, ce risque existe aussi dans le cas des listes chaînées (il se manifeste par l'échec de la fonction malloc), mais il correspond à la saturation de la mémoire de l'ordinateur; c'est un événement grave, mais rare. Alors que le débordement des tableaux est beaucoup plus fréquent, car il ne traduit qu'une erreur d'appréciation de la part du programmeur.

Les listes simplement chaînées

Opérations sur les listes chaînées

Elles concernent, l'insertion, la suppression, la recherche, l'affichage d'une liste chaînée. Mais tout d'abord, il faut commencer par l'initialisation.

Initialisation

Prototype de la fonction :

```
void initialise_liste (Liste *L);
```

Cette opération doit être faite avant toute autre opération sur la liste. Elle initialise le pointeur debut et le pointeur fin avec la valeur NULL, et la taille avec la valeur zero.

Définition de la fonction

```
void initialise_liste (Liste *L)
```

```
{  
    L -> debut = NULL;  
    L -> fin = NULL;  
    L -> taille = 0;  
}
```

Les listes simplement chaînées

Remarque

La fonction Liste *CreerListe() crée, alloue et initialise les champs de la liste.

I- Insertion d'un élément dans la liste

Voici l'algorithme d'insertion et de sauvegarde des éléments :

- déclaration d'élément à insérer
- allocation de la mémoire pour le nouvel élément
- remplir le contenu du champ donnee
- mettre à jour les pointeurs vers le premier et le dernier élément si nécessaire.
- mettre à jour la taille de la liste

Pour ajouter un élément dans la liste il y a plusieurs situations :

1. Insertion au début de la liste
2. Insertion à la fin de la liste
3. Insertion ailleurs dans la liste

Les listes simplement chaînées



Les fonctions prototypes ainsi que les définitions des structures se placent dans le fichier *listes.h*. La définition des fonctions va être dans le fichier *listes.c* en incluant en plus des bibliothèques standards le fichier *listes.h* par `#include "listes.h"`.

1 Insertion au début de la liste

Prototype de la fonction :

```
void ins_debut_liste (Liste *L, Information val);
```

Les listes simplement chaînées

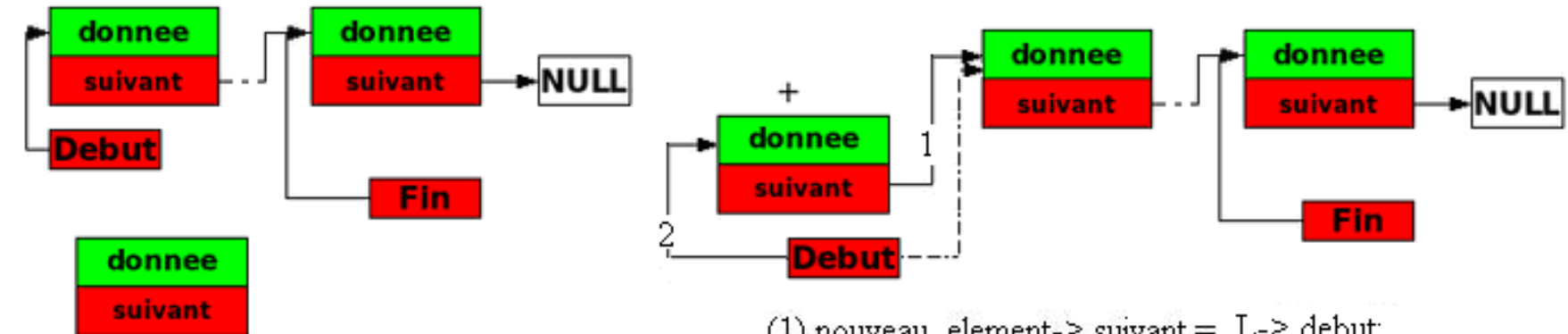


Définition de la fonction

```
void ins_debut_liste (Liste * L, Information val)
{
    Maillon *m;
    m=CreerElement();
    m-> donnee = val;
    m-> suivant = L -> debut;
    L -> debut = m;
    if (L -> taille==0) L -> fin= m;
    L -> taille++;
}
```

Les listes simplement chaînées

Insertion au début de la liste



alloc(nouveau_element)

- (1) nouveau_element->suivant = L->debut;
 - (2) L->debut = nouveau_element;
- L->taille ++;

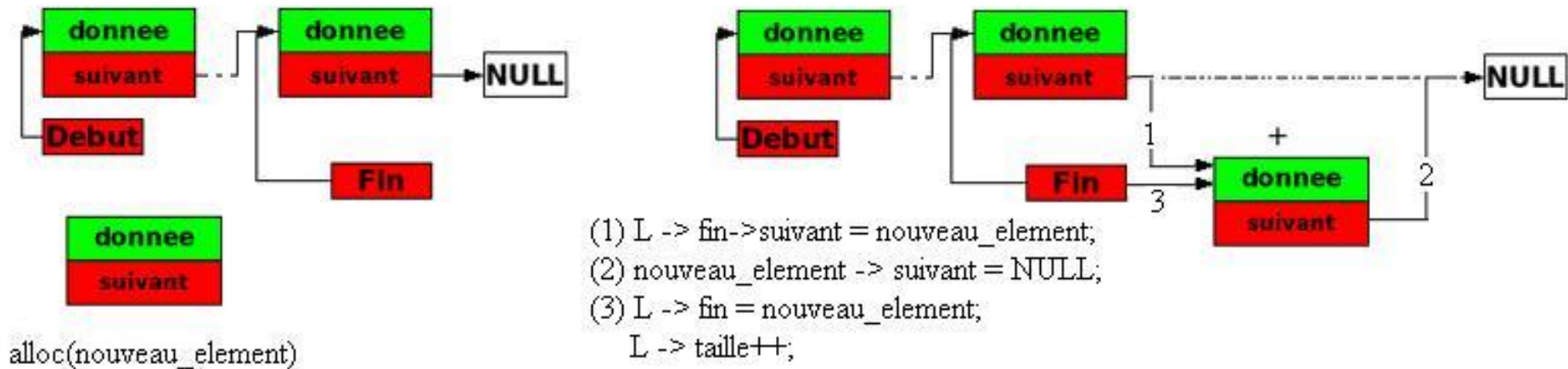
Les listes simplement chaînées

2 Insertion à la fin de la liste

Prototype de la fonction :

```
void ins_fin_liste (Liste *L, Information val);
```

Insertion à la fin de la liste



Les listes simplement chaînées

Définition de la fonction

```
void ins_fin_liste (Liste * L, Information val)
{
    Maillon *m;
    m=CreerElement();
    m-> donnee = val;
    if(L -> fin!=NULL) L->fin->suivant = m;
    L -> fin = m;
    if (L -> taille==0) L -> debut= m;
        L -> taille++;
}
```


Les listes simplement chaînées

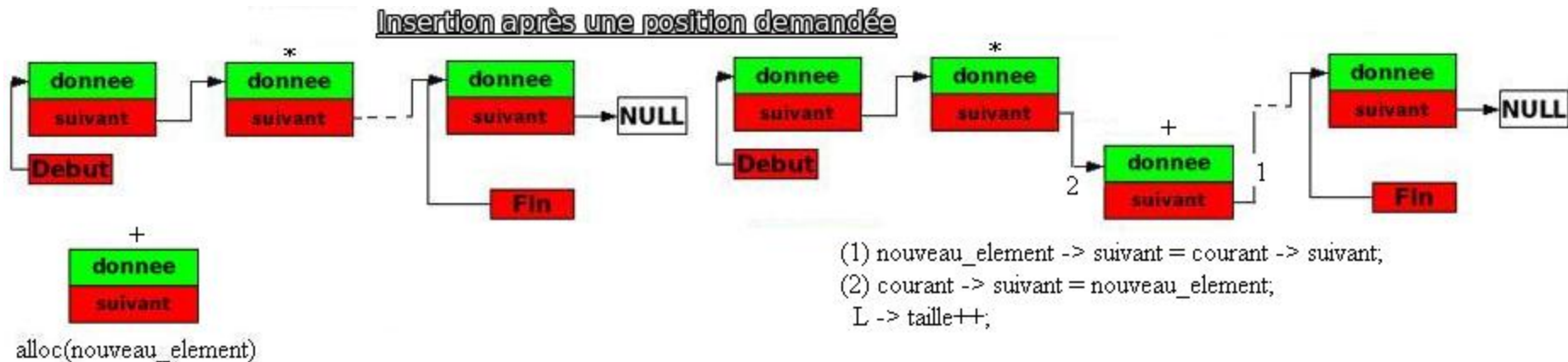
3 Insertion ailleurs dans la liste

Prototype de la fonction :

```
int ins_pos_liste (Liste *L, Information val, int pos);
```

La fonction renvoie -1 en cas d'échec sinon elle renvoie 0.

L'insertion s'effectuera après une certaine position passée en argument à la fonction. La position indiquée ne doit pas être le dernier élément. Dans ce cas il faut utiliser la fonction d'insertion à la fin de la liste.



Les listes simplement chaînées

Définition de la fonction

```
int ins_pos_liste (Liste * L, Information val, int pos)
{
    if (pos < 1 || pos >= L->taille)
        return -1;
    Maillon *courant, *m;
    int i;
    m=CreerElement(); m-> donnee = val;
    courant = L -> debut;
    for (i = 1; i < pos; i++) {courant = courant -> suivant;}
    m-> suivant = courant -> suivant;
    courant -> suivant = m;
    L -> taille++;
    return 0;
}
```

Les listes simplement chaînées

II- Retirer un élément dans la liste simplement chaînée

Voici l'algorithme pour retirer un élément de la liste :

- Utilisation d'un pointeur temporaire pour sauvegarder l'adresse d'éléments à retirer;
- L'élément à retirer se trouve après l'élément courant;
- Faire pointer le pointeur suivant de l'élément courant vers l'adresse du pointeur suivant de l'élément à retirer;
- Libérer la mémoire occupée par l'élément à retirer;
- Mettre à jour la taille de la liste.

Pour retirer un élément dans la liste il y a plusieurs situations :

1. Retirer au début de la liste
2. Retirer à la fin de la liste
3. Retirer ailleurs dans la liste

Les listes simplement chaînées

1 Retirer au début de la liste

Prototype de la fonction :

Information supp_debut_liste (Liste *L);

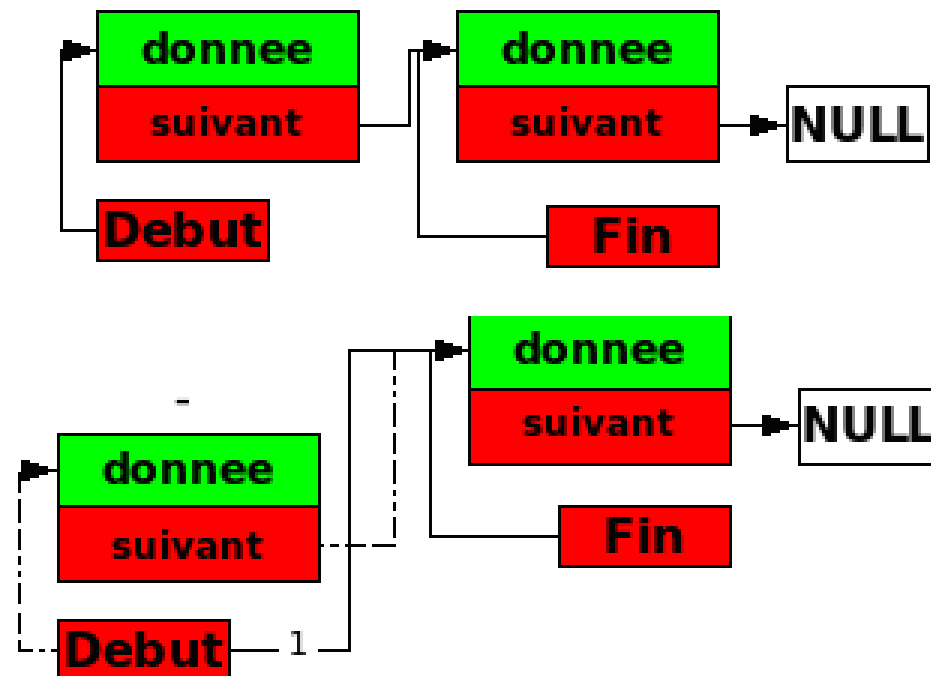
Définition de la fonction

Information supp_debut_liste (Liste * L)

```
{  
    Information res;  
    Maillon *supp_element;  
    if(L->taille==0) printf(" impossible de supprimer dans liste vide\n");  
    else{  
        supp_element = L-> debut ; L -> debut= L-> debut ->suivant;  
        L-> taille--;  
        if (L-> taille==0) L-> fin= NULL;  
        res=supp_element->donnee;  
        free(supp_element);}  
    return res;  
}
```

Les listes simplement chaînées

Suppression au début de la liste



```

supp_element = L -> debut;
(1) L -> debut = L -> debut -> suivant;
    L -> taille--;
  
```

Les listes simplement chaînées

2 Retirer à la fin de la liste

Prototype de la fonction :

Information supp_fin_liste (Liste *L);

Définition de la fonction

Information supp_fin_liste (Liste * L)

```
{  
    Information res ;  
    Maillon *pred, *courant;  
    if (L -> taille == 0)    printf(" impossible de supprimer dans liste vide\n");  
    else {  
        courant=L->debut;  
        pred=NULL;  
        while ( courant ->suivant != NULL)  
        {  
            pred=courant ;  
            courant=courant-> suivant;  
        }  
    }
```

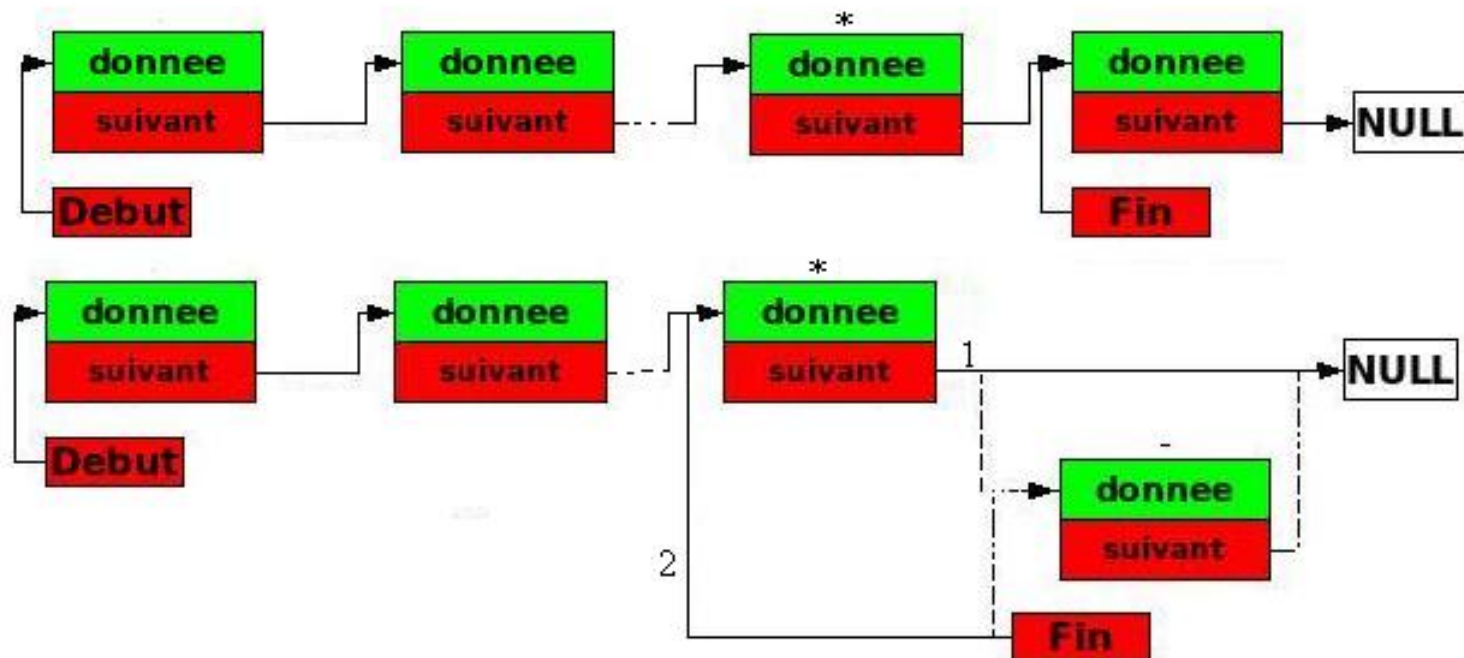


Les listes simplement chaînées

```
res= courant-> donnee;  
free(courant);  
L -> fin= pred;  
if (pred!=NULL)    pred->suivant=NULL;  
else  
    L -> debut=NULL;  
    L -> taille--;  
}  
return res;  
}
```

Les listes simplement chaînées

Suppression après l'avant dernier élément



```

supp_element = courant -> suivant;
(1) courant -> suivant = courant -> suivant -> suivant;
(2) L -> fin = courant;
    L -> taille--;
  
```


Les listes simplement chaînées

3 Retirer ailleurs dans la liste

Prototype de la fonction :

Information supp_dans_liste (Liste *L, int pos);


Définition de la fonction

/* supprimer un élément après la position demandée */

Information supp_dans_liste (Liste * L, int pos)

```
{  
    int i;  
    Information res;  
    Maillon *courant, *supp_element;  
    if (L -> taille <= 1 || pos < 1 || pos >= L -> taille)  
        printf("Impossible de supprimer dans liste à cette position \n");
```

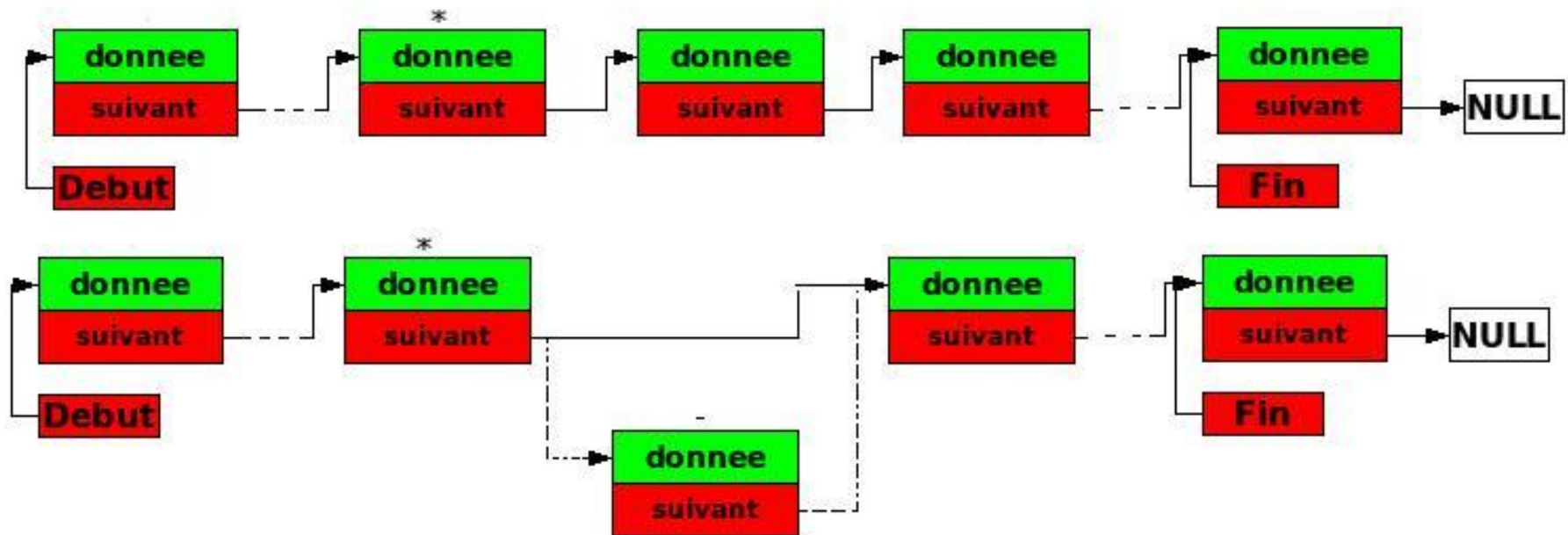
Les listes simplement chaînées



```
else
{
    courant = L -> debut;
    for (i = 1; i < pos; i++)
        { courant = courant -> suivant;}
    supp_element = courant -> suivant;
    courant -> suivant = courant -> suivant -> suivant;
    if(courant -> suivant == NULL) L -> fin = courant;
    res= supp_element -> donnee;
    free (supp_element);
    L -> taille--;
}
return res;
}
```

Les listes simplement chaînées

Suppression après une position demandée



```

supp_element = courant -> suivant;
(1) courant -> suivant = courant -> suivant -> suivant;
L -> taille--;
  
```

Les listes simplement chaînées



Affichage de la liste

Pour afficher la liste entière il faut se positionner au début de la liste. Ensuite en utilisant le pointeur suivant de chaque élément, la liste est parcourue du premier vers le dernier élément. La condition d'arrêt est donnée par le pointeur suivant du dernier élément qui vaut NULL ou la taille de la liste.

Prototype de la fonction

```
void affiche (Liste * L);
```

Les listes simplement chaînées

Définition de la fonction

```
/* affichage de la liste */  
void affiche (Liste * L)  
{  
    Maillon *courant;  
    courant = L -> debut;  
    while (courant != NULL)  
    {  
        printf (" format des champs \n", (courant -> donnee).champ1,...,  
                (courant -> donnee).champn);  
        courant = courant -> suivant;  
    }  
}
```

Les listes simplement chaînées

Destruction de la liste

Pour détruire la liste entière et libérer l'espace occupé par les maillons, on utilise par exemple la suppression au début de la liste tant que la taille est plus grande que zéro.

Prototype de la fonction

```
void detruire_liste (Liste * L);
```

Définition de la fonction

```
/* détruire la liste */
```

```
void detruire_liste (Liste * L)
{
    while (L -> taille > 0)
        supp_debut_liste (L);
}
```

À la sortie de la fonction on effectue un `free(L);`

Les listes simplement chaînées

Destruction de la liste

Une deuxième façon pour vider une liste entière et libérer l'espace occupé par les maillons , est d'utiliser la fonction suivante:

Prototype de la fonction

```
void vider_liste (Liste * L);
```

Définition de la fonction

```
void vider_liste (Liste * L)
{
    Maillon *curseur, *succ;
    curseur=L -> debut;
    while (curseur != NULL)
    { succ = curseur -> suivant; free(curseur ); curseur =succ;}
    L -> taille=0; L -> debut=NULL; L -> fin=NULL;
}
```

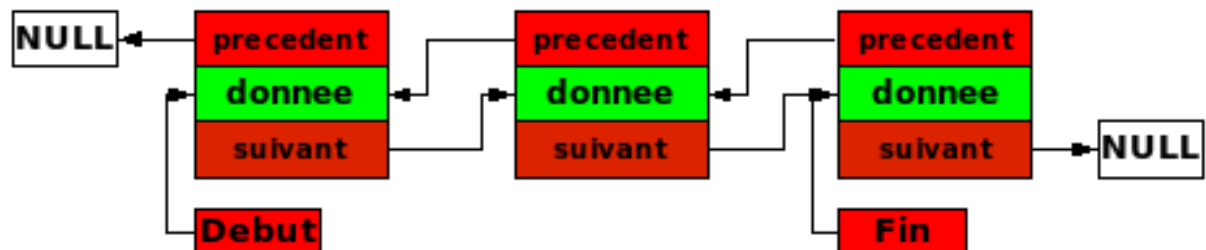
À la sortie de la fonction on effectue un free(L);

Les listes doublement chaînées

Principe

Une liste à double chaînage est une liste dont chaque élément pointe à la fois sur son successeur et son prédécesseur. De cette manière, la notion de tête n'a pas d'importance car la liste peut être repérée par l'intermédiaire de n'importe lequel de ses éléments en faisant une recherche avant ou une recherche arrière(symétrique).

Liste doublement chaînée



Les listes doublement chaînées




Structure

Pour définir un élément de la liste doublement chaînée, on procède comme pour une liste simplement chaînée. Le pointeur précédent du premier élément doit pointer vers NULL (le début de la liste). De même le pointeur suivant du dernier élément doit pointer vers NULL (la fin de la liste).

La construction du prototype d'un élément de la liste doublement chaînée

```
struct Info {  
    type1 champs1 ;  
    type2 champs2 ;  
    .... ;  
    typen champsn ;  
};  
typedef struct Info Information;
```

Les listes doublement chaînées



```
struct maillon_dc
{
    Information donnee;
    struct maillon_dc *suivant;
    struct maillon_dc *precedent;
};
typedef struct maillon_dc  Maillon_dc;
```

```
struct liste_dc
{
    Maillon_dc *debut;
    Maillon_dc *fin;
    int taille;
};
typedef struct liste_dc Liste_dc
```

Les listes doublement chaînées

Initialisation d'une liste doublement chaînée

Après la déclaration de la liste et l'allocation de la mémoire, on doit initialiser les pointeurs debut et fin avec le pointeur NULL et le champ taille par zéro.

```
Liste_dc *L_dc ; //déclaration d'un objet  
L_dc=CreerListe_dc();// prototype de la fonction
```

```
Liste_dc *CreerListe_dc()// définition de la fonction  
{  
    Liste_dc *L_dc=(Liste_dc *)malloc(sizeof(Liste_dc)) ;  
    if (L_dc==NULL) {printf("Allocation non reussie\n");exit(-1);}  
    else {L_dc->debut=NULL; L_dc->fin=NULL; L_dc->taille=0;}  
    return L_dc;  
}
```

Les listes doublement chaînées

Création d'un maillon

Maillon_dc *m; // déclaration un objet pointeur sur un maillon doublement chaîné

m=CreerElement_dc(); // prototype de la fonction

Maillon_dc *CreerElement_dc() // définition de la fonction

```
{
    Maillon_dc *m;
    m=(Maillon_dc *)malloc(sizeof(Maillon_dc)) ;
    if (m==NULL) {printf("Allocation non reussie\n");exit(-1);}
    else
    {m->donnee.champs1=0;...; m->donnee.champsn=0;
      m->suivant=NULL; m->precedent=NULL;}
    return m;
}
```

Les listes doublement chaînées



III- Insertion d'un élément dans la liste doublement chaînée

Pour ajouter un élément dans la liste il y a plusieurs situations :

1. Insertion au début de la liste ;
2. Insertion à la fin de la liste ;
3. Insertion avant un élément ;
4. Insertion après un élément.

Les listes doublement chaînées



1 Insertion au début de la liste doublement

Prototype de la fonction :

```
void insert_debut_liste_dc (Liste_dc *L_dc , Information val) ;
```

Les listes doublement chaînées



Définition La fonction :

```
void insert_debut_liste_dc (Liste_dc *L_dc , Information val)
{
    Maillon_dc *m;
    m=CreerElement_dc();
    m-> donnee = val;
    m-> suivant = L_dc -> debut ;
    if(L_dc -> debut !=NULL) L_dc -> debut -> precedent = m;
    L_dc -> debut = m;
    if (L_dc ->taille==0) L_dc -> fin= m;
    L_dc -> taille++;
}
```

Les listes doublement chaînées



2 Insertion a la fin de la liste doublement

Prototype de la fonction :

```
void insert_fin_liste_dc (Liste_dc *L_dc , Information val) ;
```




Les listes doublement chaînées

Définition La fonction :

```
void insert_fin_liste_dc (Liste_dc *L_dc , Information val)
{
    Maillon_dc *m;
    m=CreerElement_dc();
    m-> donnee = val;
    m-> precedent = L_dc -> fin;
    if(L_dc -> fin !=NULL) L_dc -> fin -> suivant = m;
    L_dc -> fin = m;
    if (L_dc -> taille==0) L_dc -> debut= m;
    L_dc -> taille++;
}
```

Les listes doublement chaînées

3 Insertion ailleurs dans la liste doublement

a) Insertion avant un élément de la liste

L'insertion s'effectuera avant une certaine position passée en argument à la fonction. On va parcourir la liste avec un pointeur courant jusqu'à atteindre la position. le pointeur suivant du nouvel élément pointe vers l'élément courant. le pointeur precedent du nouvel élément pointe vers l'adresse sur la quelle pointe le pointeur precedent d'élément courant.

Le pointeur suivant de l'élément qui précède l'élément courant pointera vers le nouveau élément.

Le pointeur precedent d'élément courant pointe vers le nouvel élément.

Le pointeurs fin ne change pas.

Le pointeur debut change si et seulement si la position choisie est la position 1

Prototype de la fonction :

```
void insert_avant_liste_dc (Liste_dc *L_dc , Information val, int pos) ;
```



Les listes doublement chaînées

Définition La fonction :

```
void insert_avant_liste_dc (Liste_dc *L_dc , Information val, int pos)
{
    Maillon_dc *m, *courant;
    m=CreerElement_dc();
    m-> donnee = val;
    courant = L_dc -> debut;
    for (int i=0; i < pos; i++)    courant = courant -> suivant;
    m-> precedent = courant -> precedent;  m-> suivant = courant ;
    if (courant->precedent == NULL) L_dc -> debut = m;
    else
        {courant -> precedent -> suivant = m;
         courant -> precedent = m;}
    L_dc -> taille++;
}
```



Les listes doublement chaînées

b) Insertion après un élément de la liste

L'insertion s'effectuera après une certaine position passée en argument à la fonction. La position indiquée ne doit pas être le dernier élément. Dans ce cas il faut utiliser la fonction d'insertion à la fin de la liste.

On va parcourir la liste avec un pointeur courant jusqu'à atteindre la position.

Le pointeur précédent du nouvel élément pointe vers l'élément courant.

Le pointeur suivant du nouvel élément pointe vers l'adresse sur la quelle pointe le pointeur suivant d'élément courant.

Le pointeur précédent de l'élément qui suit l'élément courant pointera vers le nouveau élément.

Le pointeur suivant d'élément courant pointe vers le nouvel élément.

Le pointeurs debut ne change pas;

Le pointeur fin change si et seulement si la position choisie est la position du dernier élément (taille de la liste doublement chaînée).



Les listes doublement chaînées

Prototype de la fonction :

```
void insert_apres_liste_dc (Liste_dc *L_dc , Information val, int pos) ;
```

Définition La fonction :

```
void insert_apres_liste_dc (Liste_dc *L_dc , Information val, int pos)
{
    Maillon_dc *m, *courant;
    m=CreerElement_dc();
    m-> donnee = val;
    courant = L_dc -> debut;
    for (int i=0; i < pos; i++)    courant = courant -> suivant;
    m-> precedent = courant; m-> suivant =  courant -> suivant;
    if (courant -> suivant == NULL)    L_dc -> fin = m;
    else { courant -> suivant -> precedent = m;
    courant -> suivant= m;}
    L_dc -> taille++;
}
```



Les listes doublement chaînées

IV- retirer un élément dans la liste doublement chaînée

Voici l'algorithme de suppression d'un élément de la liste doublement chaînée :

Utilisation d'un pointeur temporaire pour sauvegarder l'adresse de l'élément à supprimer. L'élément à supprimer peut se trouver dans n'importe quelle position dans la liste.

Retourner le champs donnée de l'élément à supprimer.

Libérer la mémoire occupée par l'élément supprimé;

Mettre à jour la taille de la liste.

Pour supprimer un élément dans la liste il y a plusieurs situations :

1. Retirer au début de la liste doublement chaînée;
2. Retirer à la fin de la liste doublement chaînée
3. Retirer à une position quelconque.



Les listes doublement chaînées

Nous allons créer une seule fonction.

Si nous voulons supprimer l'élément au début de la liste nous choisirons la position 1.

Si nous voulons supprimer l'élément à la fin de la liste nous choisirons la position `L_dc->taille`.

Si nous désirons supprimer un élément quelconque alors on choisit sa position dans la liste.

Prototype de la fonction:

Information `supp_pos_liste_dc (Liste_dc *L_dc, int pos);`

Les listes doublement chaînées

Définition de la fonction:

Information supp_pos_liste_dc (Liste_dc *L_dc, int pos)

```
{
    int i;
    Information res;
    Maillon_dc *supp_element,*courant;
    if (L_dc -> taille == 0)
        printf ("impossible de supprimer Liste vide! \n ");
    if( pos == 1)          /* suppression au début /
    {
        supp_element = L_dc -> debut;
        L_dc -> debut = L_dc ->debut -> suivant;
        if (L_dc ->debut == NULL)    L_dc -> fin = NULL;
        else
            L_dc -> debut -> precedent == NULL;
    }
}
```




Les listes doublement chaînées

```
elseif (pos == L_dc ->taille)    /* suppression à la fin */
{
    supp_element = L_dc -> fin;
    L_dc -> fin -> precedent -> suivant = NULL;
    L_dc ->fin = L_dc ->fin->precedent;
}
else                             /* suppression ailleurs */
{
    courant = L_dc ->debut;
    for(i=1 ;i < pos; i++)
        courant = courant -> suivant;
    supp_element = courant;
    courant -> precedent -> suivant = courant -> suivant;
    courant -> suivant -> precedent = courant -> precedent;
}
```

Les listes doublement chaînées



```
res= supp_element->donnee;  
free(supp_element);  
L_dc ->taille--;  
return res;  
}
```



Les listes doublement chaînées

Affichage de la liste doublement chaînée

Pour afficher la liste entière il existe deux possibilité, affichage direct ou affichage inverse.

En utilisant le pointeur suivant ou precedent de chaque élément, la liste est parcourue du premier vers le dernier élément (affichage direct) ou du dernier vers le premier élément (affichage inverse).



Les listes doublement chaînées

Prototype de la fonction

```
void affiche_direct_liste_dc(Liste_dc *L_dc);
```

Définition de la fonction

```
void affiche_direct_liste_dc(Liste_dc *L_dc)
{
    Maillon_dc *courant;
    courant = L_dc -> debut;
    while(courant != NULL)
    {
        printf ("format1\t...formatn\n\n", (courant -> donnee).champ1,...,
            (courant -> donnee).champn);
        courant = courant -> suivant;
    }
}
```



Les listes doublement chaînées

Prototype de la fonction

```
void affiche_inverse_liste_dc(Liste_dc *L_dc);
```

```
void affiche_inverse_liste_dc(Liste_dc *L_dc)
{
    Maillon_dc *courant;
    courant = L_dc -> fin;
    while(courant != NULL)
    {
        printf ("format1\t...formatn\n\n", (courant -> donnee).champ1,...,
        (courant -> donnee).champn);
        courant = courant->precedent;
    }
}
```



Les listes doublement chaînées

Destruction de la liste

Pour détruire la liste entière, on utilise par exemple la suppression à la position 1 de la liste tant que la taille est plus grande que zéro.

Prototpe de la fonction

```
void detruire_liste_dc (Liste_dc * L_dc);
```

Définition de la fonction

```
void detruire_liste_dc (Liste_dc * L_dc)
{
    while (L_dc -> taille > 0)
        supp_pos_liste_dc (L_dc, 1);
}
```

À la sortie de la fonction on doit récupérer la mémoire occupée par la liste elle-même.

```
free(L_dc);
```



Fin du chapitre 2

Chapitre suivant: les Piles et les Files



Chapitre 3: Les piles et les files

I. Les piles

II. Les Files

Les piles



I- le concept de pile: une pile c'est quoi?

Une pile est une structure de données dynamique caractérisée par un comportement particulier en ce qui concerne l'insertion et l'extraction des éléments (des éléments y sont introduits, puis extraits) ayant la propriété que, lors d'une extraction, l'élément extrait est celui qui a y été introduit le plus récemment. On dit « dernier entré, premier sorti », LIFO « Last In First Out »).

Ces caractéristiques sont :

- Insertion en tête : empiler => Push ;
- Suppression en tête : dépiler => Pop ;

On repère la pile par son pointeur de début ;

Définition

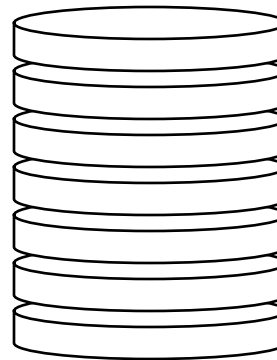
une pile est un ensemble d'éléments de même type, seul l'élément au sommet est visible.

Les piles

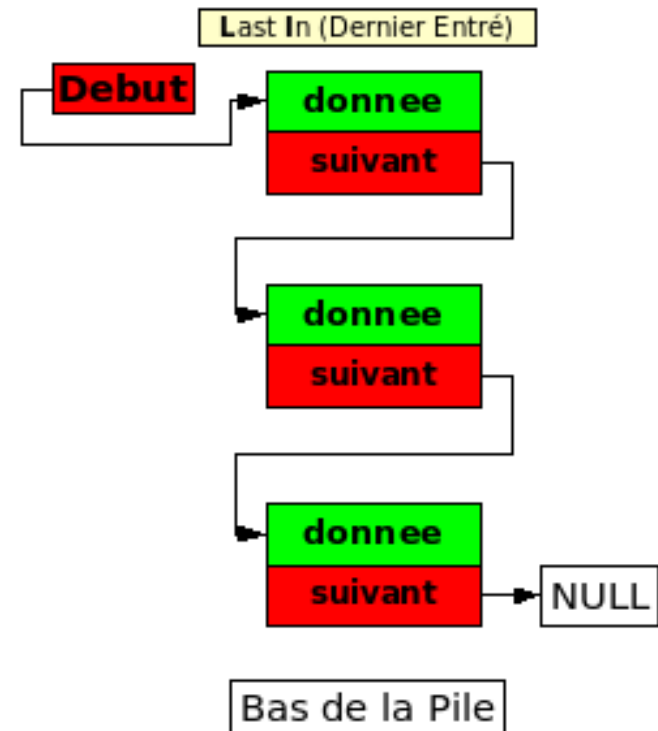
Les piles sont très utilisées en informatique

Notion intuitive : pile d'assiettes, pile de dossiers à traiter, ...

Une pile est une structure linéaire permettant de stocker et de restaurer des données selon un ordre LIFO (*Last In, First Out* ou « dernier entré, premier sorti »)



La pile





Les piles

Domaines d'application

Les problèmes qui suivent une stratégie **LIFO** : Last In First Out

La première information à être récupérée est celle qui a été mise en attente en dernier

→ Compilation

→ Évaluation d'expressions arithmétiques, logiques, ...

La construction du prototype d'un élément de la pile

Pour définir un élément de la pile, on procède comme pour une liste simplement chaînée

```
struct Info
```

```
{  
    type1 champs1 ;  
    type2 champs2 ;  
    .... ;  
    typen champsn ;  
};
```



Les piles

```
typedef struct Info Information;
struct maillon
{
    Information donnee;
    struct maillon *suivant;
};
typedef struct maillon Maillon;
struct pile
{
    Maillon *debut;
    int taille;
} ;
typedef struct pile Pile ;
Pile *P ;
P=CreerPile();
```



Les piles

Opérations sur les piles

a. Initialisation

Cette opération doit être faite avant toute autre opération sur la pile. Elle initialise le pointeur debut avec le pointeur NULL, et la taille avec la valeur zero.

Prototype de la fonction :

```
Pile * CreerPile();
```

Définition de la fonction

```
Pile * CreerPile()  
{Pile *P;  
  P=(Pile *)malloc(sizeof(Pile)) ;  
  if (P==NULL) {printf("Allocation non reussie\n");exit(-1);}  
  else {P>debut=NULL; P>taille=0;}  
  return P;  
}
```



Les piles

b. Insertion d'un élément dans la pile push

Voici l'algorithme d'insertion et de sauvegarde des éléments :

- déclaration de l'élément à insérer
- allocation de la mémoire pour le nouvel élément
- remplir le contenu du champ de donnée
- mettre à jour le pointeur debut vers le 1er élément (le sommet de la pile)
- mettre à jour la taille de la pile

Prototype de la fonction :

```
void empiler_pile (Pile *P, Information val);
```



Les piles

Définition de la fonction

```
void empiler_pile (Pile * P, Information val)
{
    Maillon *m;
    m= CreerElement();
    m->donnee = val;
    m->suivant = P -> debut;
    P ->debut = m;
    P ->taille++;
}
```

Les piles



c- Retirer un élément de la pile

Pour dépiler l'élément de la pile, il faut tout simplement retirer l'élément vers lequel pointe le pointeur debut (sommet de la pile).

Prototype de la fonction :

Information depiler_pile(Pile *P);

Les étapes de l'algorithme sont :

- le pointeur supp_elem contiendra l'adresse du 1er élément à supprimer;
- le pointeur debut pointera vers le 2ème élément (après la suppression du 1er élément, le 2ème sera en haut de la pile);
- la taille de la pile sera décrémentée d'un élément

Les piles

Définition de la fonction

Information depiler _pile(Pile * P)

```
{
    Maillon *supp_element; Information res;
    if (P -> taille == 0) printf("impossible de dépiler une pile vide");
    else {
        supp_element = P -> debut;
        P -> debut = P -> debut -> suivant;
        res=supp_element-> donnee;
        free (supp_element);
        P -> taille--;
    }
    return res;
}
```

Les piles



d. Affichage de la pile

Pour afficher la pile entière, il faut se positionner au début de la pile. Ensuite, en utilisant le pointeur suivant de chaque élément, la pile est parcourue du premier vers le dernier élément. La condition d'arrêt est donnée par la taille de la pile.

Prototype de la fonction :

```
void affiche_pile (Pile *P);
```

Les piles

Définition de la fonction

```
/* affichage de la pile */
void affiche_pile (Pile * P)
{
    Maillon *courant;
    int i;
    if(P->taille==0) printf(« Pile vide! \n« );
    else{
        courant = P -> debut;
        for(i=0; i < P -> taille; i++){
            printf("% format des champs \n", (courant -> donnee).champ1, ...,
                (courant -> donnee).champn);
            courant = courant -> suivant;}
    }
}
```



Les piles

e. Récupération de la donnée en haut de la pile

Pour récupérer la donnée au sommet la pile sans la supprimer, on peut utiliser une macro définie par comme suit:

```
#define pile_donnee(P) P -> debut -> donnee .
```

f. Destruction de la pile

Pour détruire la pile entière, on dépile l'élément au sommet tant que la taille est non nulle.

Prototpe de la fonction

```
void detruire_pile (Pile *P);
```

Définition de la fonction

```
void detruire_pile (Pile * P)
{
    while (P -> taille > 0) depiler_pile (P);
}
```

À la sortie de la fonction on doit récupérer la mémoire occupée par la pile elle-même par `free(P);`

Les files

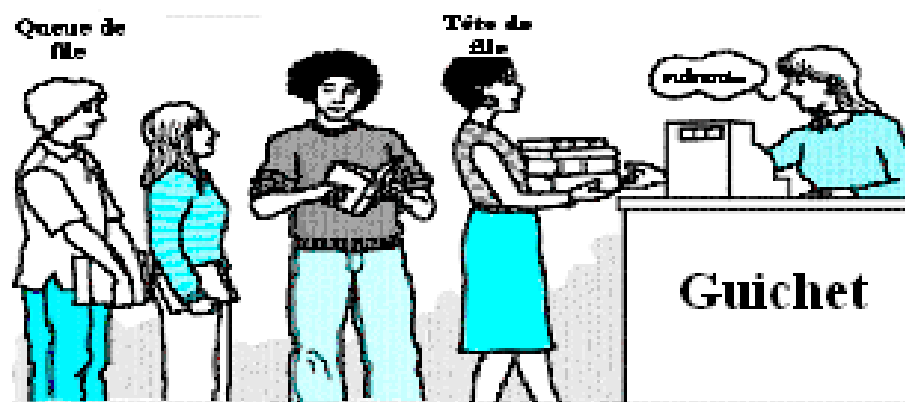
II- le concept de file:

C'est en générale des structures de listes appelées (FIFO) (First In First Out) «premier entré, premier sorti », (premier arrivé, premier servi). Ces caractéristiques sont :

- Insertion en queue : ajouter => enfiler ;
- Suppression en tête : défiler

Notion intuitive :

File d'attente à un guichet, file de documents à imprimer, ...



Les files



Une file est une structure linéaire permettant de stocker et de restaurer des données selon un ordre FIFO (*First In, First Out* ou « premier entré, premier sorti »)

Dans une file :

Les insertions (*enfilements*) se font à une extrémité appelée *queue* de la file et les suppressions (*défilements*) se font à l'autre extrémité appelée tête

Domaines d'applications

Les problèmes qui suivent une stratégie **FIFO** peuvent être résolus en utilisant une structure de données **File**

- Accès à un dispositif périphérique
- Gestions des fichiers à imprimer
- Gestion des processus/multitâche



Les files

La construction du prototype d'un élément de la file

Pour définir un élément de la file, on procède comme pour une liste simplement chaînée

```
struct Info
```

```
{  
    type1 champs1 ;  
    type2 champs2 ;  
    .... ;  
    typen champsn ;  
};
```

```
typedef struct Info Information;
```



Les files

```
struct maillon
{
    Information donnee;
    struct maillon *suivant;
};
typedef struct maillon Maillon;
struct file
{
    Maillon *debut;
    Maillon *fin;
    int taille;
} ;
typedef struct file File ;
File *F ;
F=CreerFile();
```


Les files

Opérations sur les files

a. Initialisation

Prototype de la fonction :

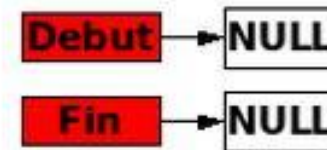
File *CreerFile();

Elle consiste à initialiser les pointeurs debut et fin avec le pointeur NULL, et la taille avec la valeur 0.

Définition de la fonction

File *CreerFile()

```
{File *F;
  F=(File *)malloc(sizeof(File)) ;
  if (F==NULL)
    {printf("Allocation non reussie\n");exit(-1);}
  else {F ->debut = NULL; F ->fin = NULL; F ->taille = 0;}
  return F;
}
```





Les files

b. Insertion d'un élément dans la file

Voici l'algorithme d'insertion et de sauvegarde des éléments

Prototype de la fonction :

```
void enfiler_file (File * F, Information val);
```

Définition de la fonction

```
void enfiler_file (File * F, Information val)
```

```
{  
    Maillon *m;  
    m= CreerElement(); m-> donnee = val;  
    if(F -> taille == 0) { F ->fin = m; F ->debut = m;}  
    else {F -> fin ->suivant = m; F ->fin = m;}  
    F -> taille++;  
}
```

Les files



c. supprimer un élément de la file

Pour retirer un élément de la file, il faut tout simplement supprimer l'élément vers lequel pointe le pointeur debut.

Prototype de la fonction :

Information defiler_file (File * F);

les étapes :

- Le pointeur supp_elem contiendra l'adresse du 1er élément;
- On retourne l'information au début
- Le pointeur debut pointera vers le 2ème élément (après la suppression du 1er élément, le 2ème sera au début de la file);
- La taille de la file sera décrémentée d'un élément.

Les files

Définition de la fonction

Information defiler_file (File * F)

```
{
    Maillon *supp_element; Information res;
    if (F -> taille == 0)
        printf("impossible de defiler une file vide");
    else
    {
        supp_element = F -> debut;
        F -> debut = F -> debut -> suivant;
        res= supp_element ->donnee;
        free (supp_element);  F -> taille--;
        if(F->taille==0) F->fin =NULL;
    }
    return res;
}
```



Les files

d. Affichage de la file

Pour afficher la file entière, il faut se positionner au début de la file (le pointeur debut le permettra). Ensuite en utilisant le pointeur suivant de chaque élément, la file est parcourue du premier vers le dernier élément. La condition d'arrêt est donnée par la taille de la file.

Prototype de la fonction :

```
void affiche_file(File *F);
```

Les files

Définition de la fonction

```
void affiche_file(File *F)
{
    Maillon *courant;
    int i;
    if (F->taille==0) printf(" File vide\n");
    else {   courant = F-> debut;
        for (i=0; i<F -> taille; i++)
        {
            printf(" % format du champs\n ", (courant ->donnee).champs1,....,
                (courant -> donnee).champsn));
            courant = courant -> suivant;
        }
    }
}
```



Les files

e. Destruction de la file

Pour détruire la file entière, on défile l'élément en tête tant que la taille est non nulle.

Prototpe de la fonction

```
void detruire_file (File *F);
```

Définition de la fonction

```
void detruire_file (File * F)
{
    while (F -> taille > 0) defiler_file (F);
}
```

À la sortie de la fonction on doit récupérer la mémoire occupée par la file elle-même par `free(F);`



Applications des piles

- I. Analyse syntaxique
- II. Calcul arithmétique
- III. Evaluation d'expression en Postfixée
- IV. Conversion Infixée à Postfixée

Applications des piles



Analyse syntaxique

L'analyse syntaxique est une phase de la compilation qui nécessite une pile. Par exemple, le problème de la reconnaissance des mots bien parenthésés. Nous devons écrire un programme qui :

- accepte les mots comme `()`, `()()` ou `((())())` ;
- rejette les mots comme `)`, `()()` ou `((())())`.

Nous stockons les parenthèses ouvrantes non encore refermées dans une pile de caractères,

Le programme lit caractère par caractère le mot entré :

- si c'est une parenthèse ouvrante, elle est empilée ;
- si c'est une parenthèse fermante, la parenthèse ouvrante correspondante est dépilée.

Le mot est accepté si

- la pile n'est jamais vide à la lecture d'une parenthèse fermante ;
- et si la pile est vide lorsque le mot a été lu.

Applications des piles



Calcul arithmétique

Le calcul arithmétique est une application courante des piles. L'ordre dans la pile permet d'éviter l'usage des parenthèses.

La notation infixée (parenthésée) consiste à entourer les opérateurs par leurs opérandes.

La notation préfixée (polonaise) consiste à placer les opérandes après l'opérateur.

La notation postfixée (polonaise inversée) consiste à placer les opérandes devant l'opérateur.

Les parenthèses sont nécessaires uniquement en notation infixée. Certaines règles permettent d'en réduire le nombre (priorité de la multiplication par rapport à l'addition, en cas d'opérations unaires représentées par un caractère spécial (-, !,...).

Les notations préfixée et postfixée sont d'un emploi plus facile puisqu'on sait immédiatement combien d'opérandes il faut rechercher.

Applications des piles

Exemple:

Infixée	PostFixée	Prefixée
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Applications des piles



Evaluation d'expression en Postfixée

Considérons l'évaluation de l'expression arithmétique en postfixée suivante:

6 5 2 3 + 8 * + 3 + *

Applications des piles



Algorithm

```
Initialiser la pile à vide;
while (ce n'est pas la fin de l'expression postfixée) {
    prendre l'item prochain de postfixée;
    if(item est une valeur)  empiler;
    else if(item operateur binaire ) {
        dépiler dans x;
        dépiler dans y;
        effectuer y operateur x;
        empiler le résultat obtenu; }
    else if (item opérateur unaire) {
        dépiler dans x;
        effectuer opérateur(x);
        empiler le résultat obtenu;
    }
}
```

Applications des piles

La seule valeur qui reste dans la pile est le résultat recherché.

Opérateur binaires: $+$, $-$, $*$, $/$, etc.,

Opérateur unaires: moins unaire, racine carrée, \sin , \cos , \exp , ... etc.

Pour l'expression $6\ 5\ 2\ 3\ +\ 8\ *\ +\ 3\ +\ *$

Le premier item est une valeur (6); elle est empilée.

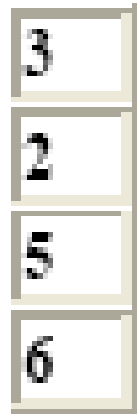
Le deuxième item est une valeur (5); elle est empilée.

Le prochain item est une valeur (2); elle est empilée.

Le prochain item est une valeur (3); elle est empilée.

La pile devient

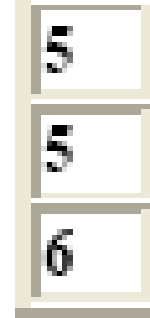
Sommet->



Applications des piles

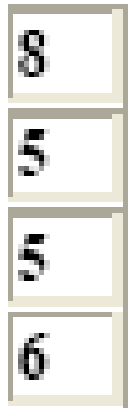
Les items restants à cette étape sont: $+ 8 * + 3 + *$

Le prochain item lu est '+' (opérateur binaire): 3 et 2 sont dépilés et leur somme '5' est ensuite empilée: Sommet->



Ensuite 8 est empilé
et le prochain opérateur *:

Sommet->

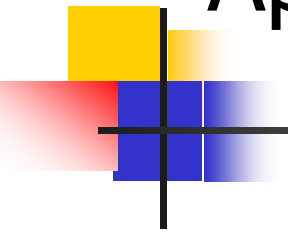


Sommet->



(8, 5 sont dépilés, 40 est empilé)

Applications des piles



Ensuite l'opérateur +
40, 5 sont dépilés
et 45 est empilé
Ensuite 3 est empilé

Sommet-> 45
6

Sommet-> 3
45
6

Ensuite l'opérateur +
3 et 45 sont dépilés
et $45+3=48$ est empilé

Sommet-> 48
6

Ensuite c'est l'opérateur *
48 et 6 sont dépilés
et $6*48=288$ est empilé

Sommet-> 288

Il n'y plus d'items à lire dans l'expression postfixée et aussi il n'y a qu'une seule valeur dans la pile représentant la valeur finale est: **288**.

Applications des piles

Autre Exemple : Evaluation de l'expression **623+-382/+*2^3+**

Symbol	opnd1	opnd2	valeur	Pile
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2

Applications des piles

(suite) Exemple : Evaluation de l'expression $623+-382/+*2^3+$

Symbol	opnd1	opnd2	valeur	Pile
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
^	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

La valeur finale est: **52.**

Applications des piles



Conversion Infixée en Postfixée

Bien entendu la notation postfixée ne sera pas d'une grande utilité s'il n'existait pas un algorithme simple pour convertir une expression infixée en une expression postfixée.

Encore une fois, cet algorithme utilise une pile.

Précédence des opérateurs (pour cet algorithme):

4 : '(' – dépilée seulement si une ')' est trouvée

3 : tous les opérateurs unaires

2 : / *

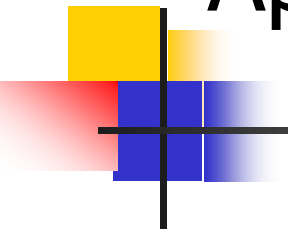
1 : + -

L'algorithme passe les opérands à la forme postfixée, mais sauvegarde les opérateurs dans la pile jusqu'à ce que tous les opérands soient tous traduits.

L'algorithme

initialise la pile et l'expression postfixée à vide;

Applications des piles



```
while(ce n'est pas la fin de l'expression infixée)
{
    prendre le prochain item infixé
    if(item est une valeur) concaténer item à postfixée
    else if(item == '(') empiler item
    else if(item == ')') {
        dépiler sur x
        while(x != '(') concaténer x à postfixe & dépiler sur x}
    else {
        while(precedence(sommet(pile)) >= precedence(item))
            dépiler sur x et concaténer x à postfixée;
        empiler item;}
}
while (pile non vide)
    dépiler sur x et concaténer x à postfixe;
```

Applications des piles

Exemple: considérons la forme infixe de l'expression $a+b*c+(d*e+f)*g$

Pile

Sommet =>



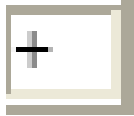
postfixée

ab

sommet=>



abc

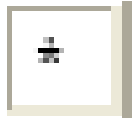


sommet=>



abc*+

sommet=>



abc*+de



Applications des piles

Suite conversion la forme infixe de l'expression $a+b*c+(d*e+f)*g$

Pile

Sommet =>

+

postfixée

$abc*+de*f$

(

+

sommet=>

+

$abc*+de*f+$

*

sommet=>

+

$abc*+de*f+g$

sommet=>

(pile vide)

$abc*+de*f+g*+$

Applications des piles

Un autre exemple : Essayons de convertir que $A + B * C$ qui va générer $ABC * +$, notre algorithme effectuera une succession d'empilement - dépilement suivants :

Item	Pile	Postfixe
-----	-----	-----
aucun	vide	aucun
A	vide	A
+	+	A
B	+	AB
À ce stade, l'algorithme détermine qui de * et + doit être placé à la fin de la pile. Comme l'opération * a une plus grande précedence, on empile *		
*	+ *	AB
C	+ *	ABC

Et on dépile

Comme les données en entrée sont terminées, on dépile tout le reste de la pile pour donner l'expression postfixée finale suivante: $ABC*+$

Applications des piles

Un autre exemple : Convertir $2*3/(2-1)+5*3$ à la forme Postfixée

Symbol	Pile	Postfixe
2	Vide	2
*	*	2
3	*	23
/	/	23*
(/(23*
2	/(23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/

Applications des piles

(suite) Convertir $2*3/(2-1)+5*3$ à la forme Postfixée

Symbol	Pile	Postfixe
5	+	23*21-/5
*	+*	23*21-/5
3	+*	23*21-/53
	Vide	23*21-/53*+

Enfin , l'expression en format Postfixe est $23*21-/53*+$

Fin du chapitre 3



Chapitre suivant: Les arbres binaires



Chapitre 4: les arbres binaires

- I. Définitions et représentation d'un arbre binaire
- II. Opérations sur les arbres binaires
- III. Parcours des arbres binaires (Parcours infixé, postfixé, préfixé)

Notion d'arbre



Notion d'arbre



I- Généralités

En informatique, un arbre est une structure de données récursive générale, représentant un arbre au sens mathématique. C'est un cas particulier de graphe qui n'a qu'une seule source et aucun cycle.

Dans un arbre, on distingue deux catégories d'éléments :

- les feuilles; éléments ne possédant pas de fils dans l'arbre ,
- les nœuds internes; éléments possédant des fils (sous-branches).

Les répertoires sous la plupart des systèmes d'exploitation actuels (Microsoft Windows, Unix dont Linux et Mac OS X ...) forment un arbre.

Les arbres binaires

Les arbres binaires de recherche

Les arbres n-aires



Notion d'arbre

Définition

Un arbre est une structure qui peut se définir de manière récursive : un arbre est un arbre qui possède des liens ou des pointeurs vers d'autres arbres.

Cette définition plutôt étrange au premier abord résume bien la démarche qui sera utilisée pour réaliser cette structure de données. Cette arborescence qualifie les arbres de structures dynamiques non linéaires.

Arité d'un arbre

On qualifie un arbre sur le nombre de fils qu'il possède. Ceci s'appelle l'arité de l'arbre. Un arbre dont les nœuds ne comporteront qu'au maximum n fils sera d'arité n . On parlera alors d'arbre n -aires.

Il existe un cas particulièrement utilisé : c'est l'arbre binaire. Dans un tel arbre, les nœuds ont au maximum 2 fils. On parlera alors de fils gauche et de fils droit pour les nœuds constituant ce type d'arbre. C'est ce type d'arbre que nous allons étudier dans ce chapitre.

Notion d'arbre



Atout des structures arbre

Accélère la recherche en diminuant le nombre de comparaisons.

Terminologie

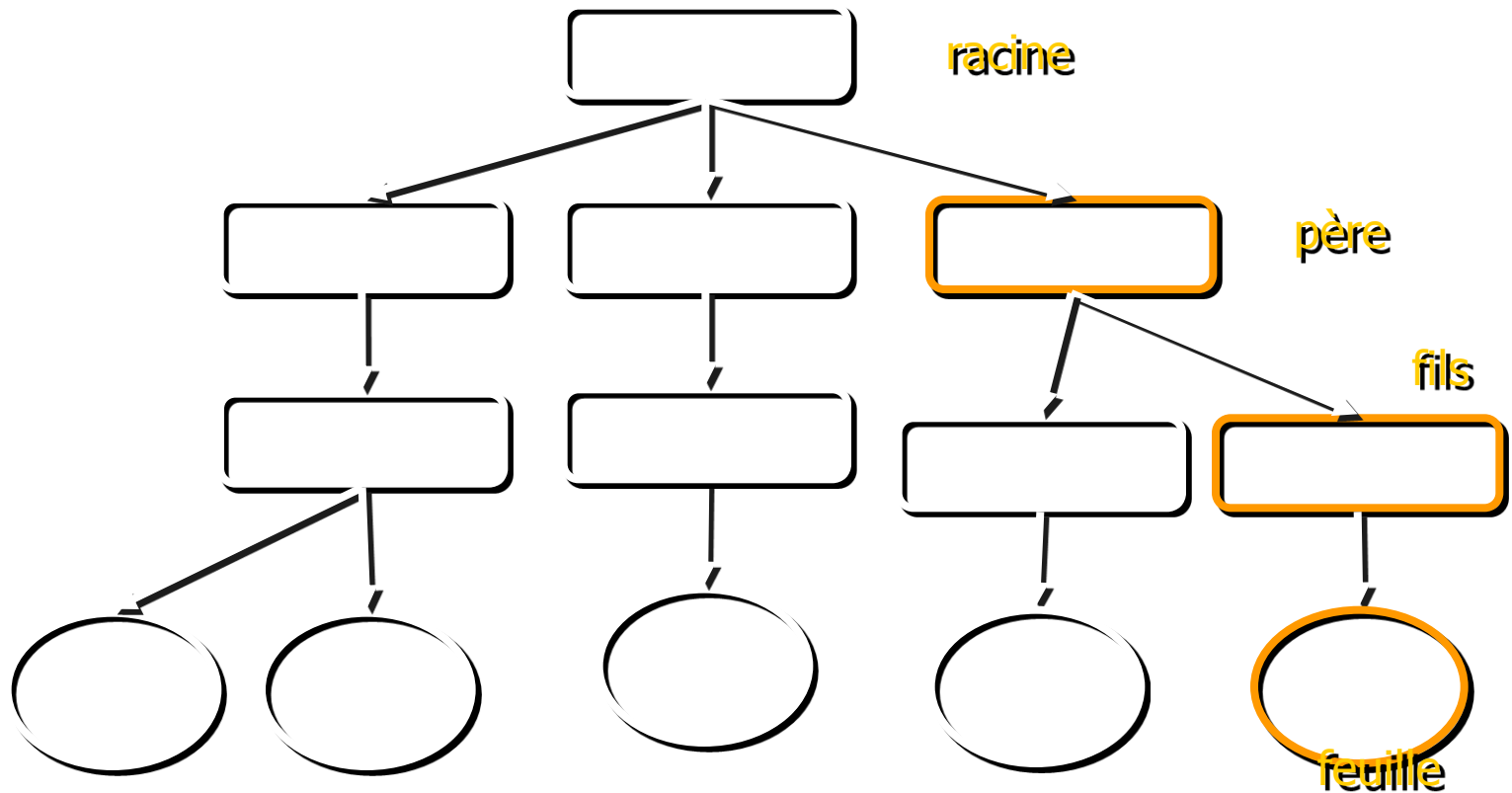
Précisons maintenant un peu plus les termes désignant les différentes composantes d'un arbre.

Tout d'abord, chaque élément d'un arbre se nomme **un nœud**. Les nœuds sont reliés les uns aux autres par des relations d'ordre ou de hiérarchie. Ainsi on dira qu'un nœud possède **un père**, c'est à dire un nœud qui lui est supérieur dans cette hiérarchie. Il possède éventuellement **un** ou **plusieurs fils**.

Il existe un nœud qui n'a pas de père, c'est donc **la racine** de l'arbre. Un nœud qui n'a pas de fils est appelé **une feuille**. Parfois on appelle **une feuille un nœud externe** tout autre nœud de l'arbre sera alors appelé **un nœud interne**.

Notion d'arbre

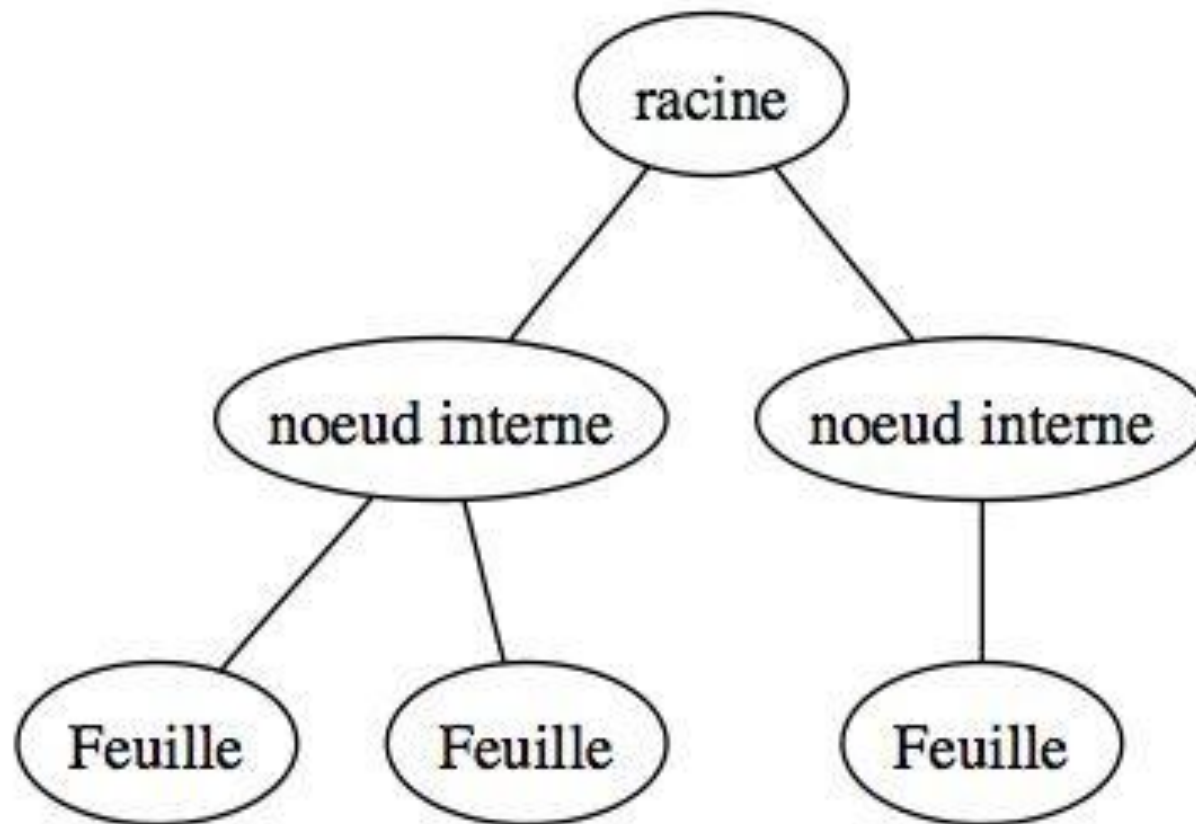
Terminologie



Arbre binaire

Terminologie

Description des différents composants d'un arbre binaire



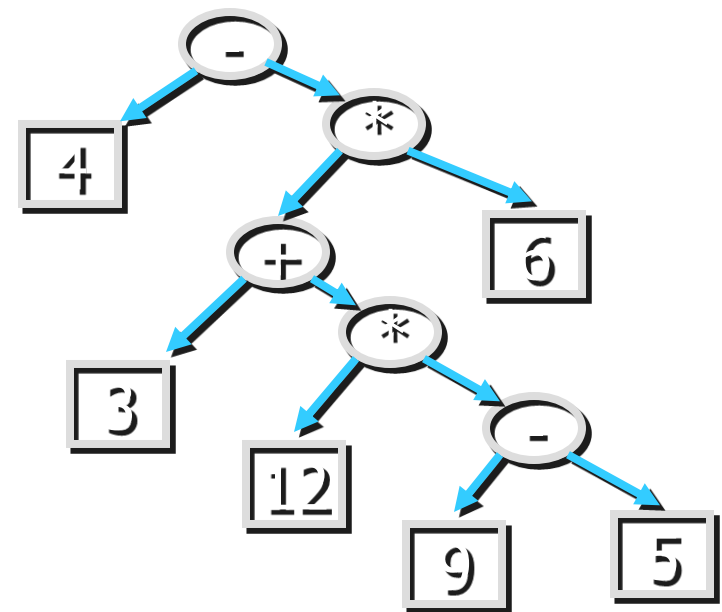
Arbre binaire

Domaines d'applications

Modèle pour les structures hiérarchisées

Expressions algébriques

$4-(3+12*(9-5))*6$



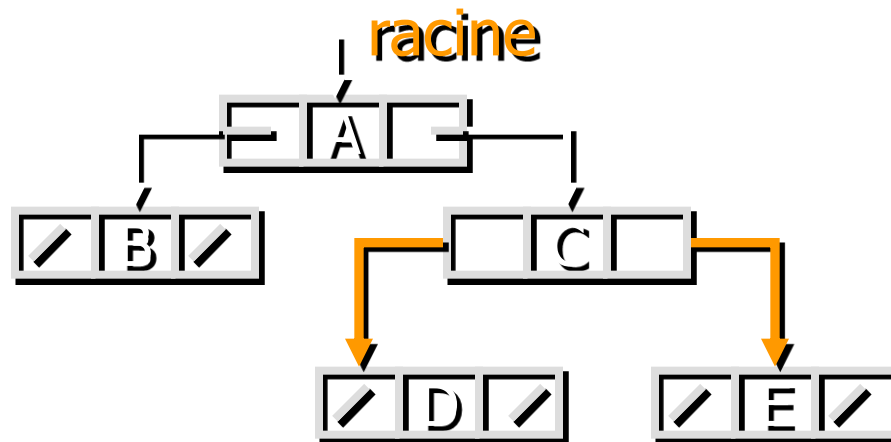
La priorité des opérations apparaît clairement

Arbre binaire

II-Mise en oeuvre

Représentation par **structure pointée**

Allocation dynamique des nœuds





Arbre binaire

Nous allons maintenant discuter de l'implémentation des arbres. Tout d'abord, définissons le nœud.

Un **nœud** est une **structure** (ou un enregistrement) 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.

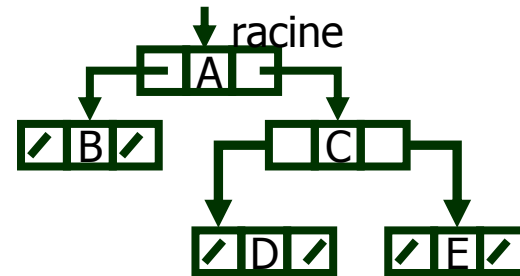
Les deux autres champs sont **le fils gauche et le fils droit du nœud**. Ces deux fils sont en fait des arbres, on les appelle généralement les sous arbres gauches et les sous arbres droit du nœud.

De part cette définition, un arbre ne pourra donc être qu'un pointeur sur un nœud.

Arbre binaire

Structure d'un noeud

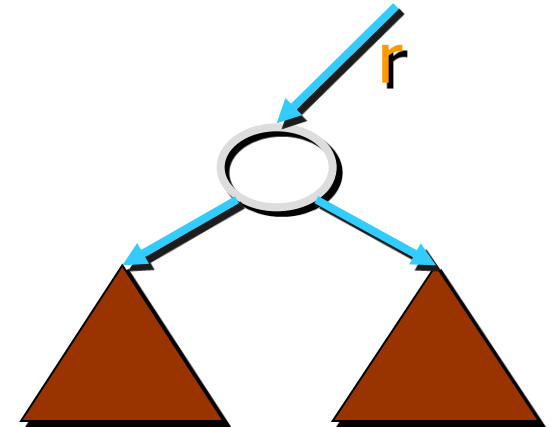
```
struct Noeud
{
    char val;
    struct Noued *gauche;
    struct Noued *droit;
};
typedef struct Noued Noued;
Nœud *racine;
```



Arbre binaire

Opération sur les arbres binaires

- Calcul de la **profondeur** maximale
- Calcul de la **taille** en nombre de nœuds
- **Exploration** de l'arbre



Arbre binaire



Taille et hauteur d'un arbre.

On appelle la **taille** d'un arbre, le nombre de nœud interne qui le compose. C'est à dire le nombre de nœud total moins le nombre de feuille de l'arbre.

On appelle également la **profondeur** d'un nœud la distance en terme de nœud par rapport à l'origine(racine).

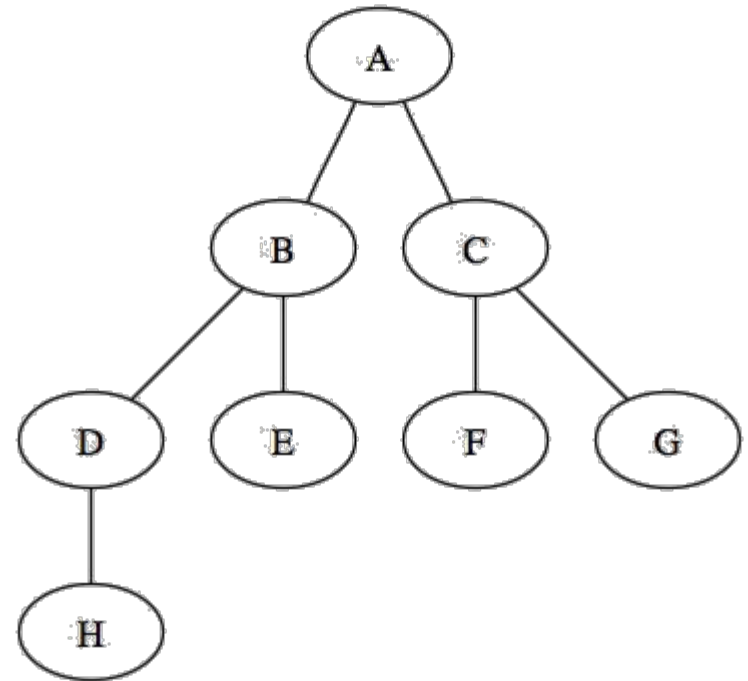
Par convention, la racine est de profondeur 0.

La **hauteur** de l'arbre est alors la profondeur maximale de ses nœuds. C'est à dire la profondeur à laquelle il faut descendre dans l'arbre pour trouver le nœud le plus loin de la racine.

Arbre binaire

Taille et hauteur d'un arbre.

La hauteur d'un arbre est très importante.
En effet, c'est un repère de performance.



Dans l'exemple le nœud F est de profondeur 2 et le nœud H est de profondeur 3.

La plupart des algorithmes que nous verrons dans la suite ont une complexité qui dépend de la hauteur de l'arbre.

Arbre binaire

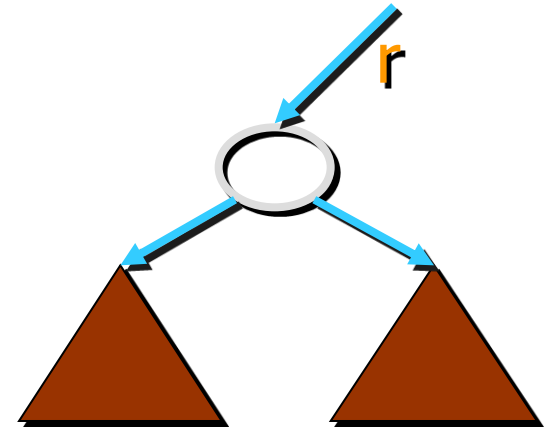
Calcul de la profondeur maximale

Prototype de la fonction

```
int profondeur_arbre(Nœud *ar);
```

Définition de la fonction

```
int profondeur_arbre(Nœud *ar)
{
    int p1,p2;
    if (ar==NULL) return 0;
    else
        p1= profondeur_arbre(ar->gauche); p2= profondeur_arbre(ar->droit);
    return 1+max(p1,p2);
}
```





Arbre binaire

III. Implémentation des arbres binaire en langage C :

Voici donc une manière d'implémenter un arbre binaire en langage C :

```
struct Noued
```

```
{
```

```
    char val;
```

```
    struct Noued *gauche;
```

```
    struct Noued *droit;
```

```
};
```

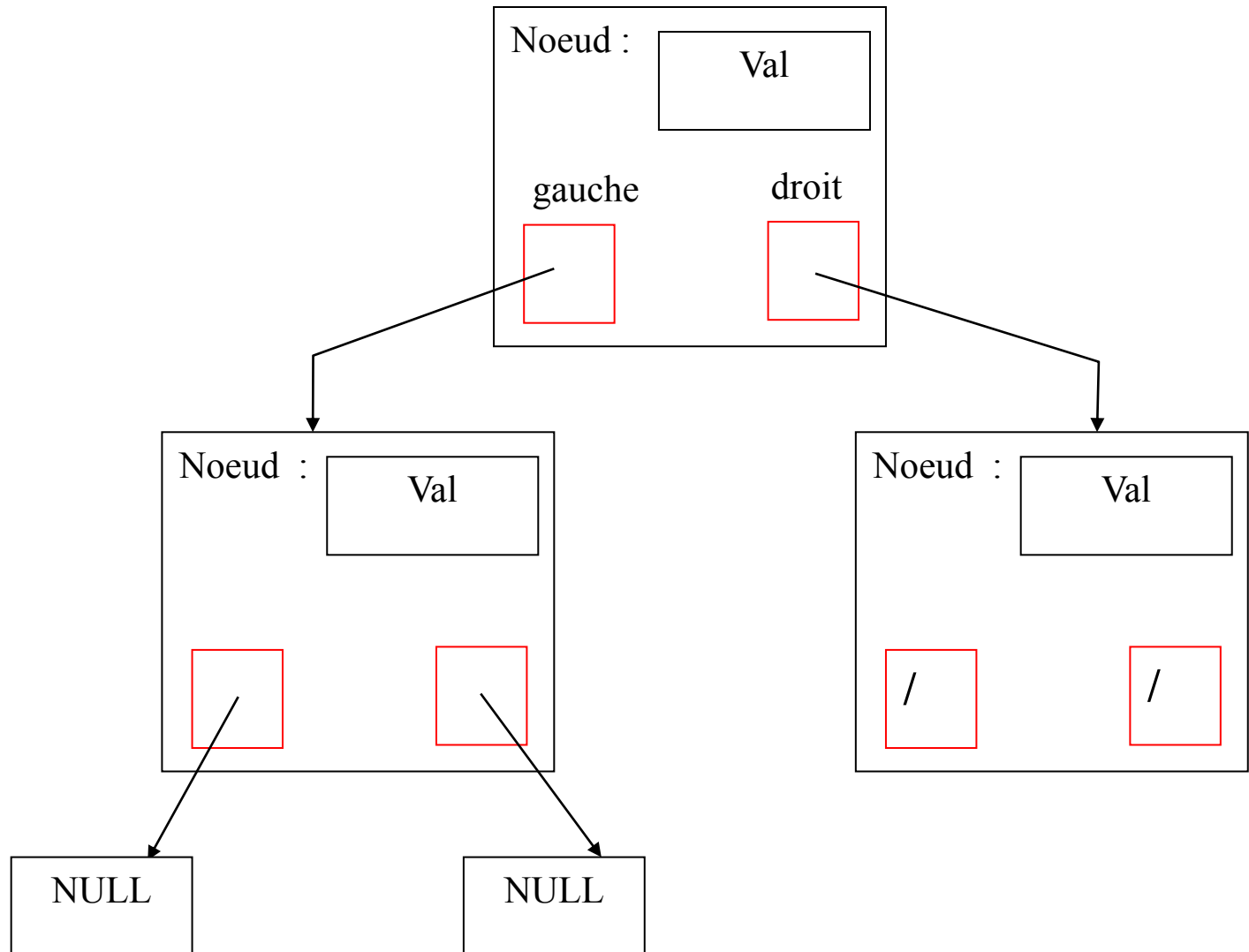
```
typedef struct Noued Noeud;
```

```
Noeud *Arbre
```

```
Arbre=(Noeud *)malloc(sizeof(Noeud));
```

De par cette définition, on peut donc aisément constater que l'arbre vide sera représenté par la constante NULL.

Arbre binaire





Arbre binaire

IV. Les fonctions de base sur la manipulation des arbres :

Voici ce que peut donner cette fonction en langage C :

Prototype de la fonction

```
int EstVide( Nœud *ar);
```

Définition de la fonction

```
int EstVide( Nœud *ar)
{
    return (ar == NULL);
}
```

On peut se demander tout simplement l'utilité d'une telle fonction. Tout simplement parce qu'il est plus simple de comprendre la signification de EstVide(ar) plutôt que ar==NULL.

Arbre binaire



Maintenant, prenons deux fonctions qui vont nous permettre de récupérer le fils gauche ainsi que le fils droit d'un arbre.

Il faut faire attention à un problème : le cas où l'arbre est vide.

En effet, dans ce cas, il n'existe pas de sous arbre gauche ni de sous arbre droit. Pour régler ce problème nous décidons arbitrairement de renvoyer l'arbre vide comme fils d'un arbre vide.

Prototype de la fonction

```
Nœud *fils_gauche( Nœud *ar);
```

Définition de la fonction

```
Nœud *fils_gauche( Nœud *ar)
{
    if ( EstVide(ar) ) return NULL;
    else return ar->gauche;
}
```



Arbre binaire

La fonction qui retourne le fils droit sera codé de la même manière mais tout simplement au lieu de renvoyer le fils gauche, nous renvoyons le fils droit.

Prototype de la fonction

```
Noued *fils_droit( Nœud *ar);
```

Définition de la fonction

```
Noued *fils_droit( Nœud *ar)
{
    if ( EstVide(ar) ) return NULL;
    else return ar->droit;
}
```



Arbre binaire

Passons à une autre fonction qui peut nous être utile : savoir si nous sommes sur une feuille.

Prototype de la fonction

```
int EstUneFeuille(Noeud *ar);
```

Définition de la fonction

```
int EstUneFeuille(Noeud *ar)
{
    if (EstVide(ar))
        return 0;
    else if EstVide(fils_gauche(ar)) && EstVide(fils_droit(ar))
        return 1;
    else
        return 0;
}
```



Arbre binaire

Enfin, nous pouvons créer une dernière fonction bien que très peu utile qui détermine si un nœud est un nœud interne.

Pour ce faire, deux méthodes: Soit on effectue le test classique en regardant si un des fils n'est pas vide, soit on utilise la fonction précédente.

Prototype de la fonction

```
int EstNœudInterne( Nœud *ar);
```

Définition de la fonction

```
int EstNœudInterne( Nœud *ar)
{
    return ! EstUneFeuille(ar) ;
}
```




Arbre binaire

Calcul de la hauteur d'un arbre

Pour calculer la hauteur d'un arbre, nous allons nous baser sur la définition récursive :

- un arbre vide est de hauteur 0.
- un arbre non vide a pour hauteur 1 + la hauteur maximale entre ses fils.

Prototype de la fonction

```
int hauteur_arbre (Nœud *ar);
```

Définition de la fonction

```
int hauteur_arbre (Nœud *ar)
{
    if ( EstVide(ar) )        return 0;
    else
        return
        1 + max( hauteur_arbre(fils_gauche(ar) ) , hauteur_arbre(fils_droit(ar)));
}
```



Arbre binaire

La fonction max n'est pas définie c'est ce que nous faisons maintenant :

```
int max(int a, int b)
{
    return (a>b)? a : b ;
}
```



Arbre binaire

Calcul du nombre de nœud

Le calcul du nombre de nœud est très simple. On définit le calcul en utilisant la définition récursive :

- Si l'arbre est vide : renvoyer 0
- Sinon renvoyer 1 plus la somme du nombre de nœuds des sous arbres.

Prototype de la fonction

```
int NombreNœud(Nœud *ar);
```

Définition de la fonction

```
int NombreNœud(Nœud *ar)
{
    if( EstVide(ar) ) return 0;
    else
        return
            1 + NombreNœud(fils_gauche(ar)) + NombreNœud(fils_droit(ar));
}
```



Arbre binaire

Calcul du nombre de feuilles

Le calcul du nombre de feuille repose sur la définition récursive :

- un arbre vide n'a pas de feuille.
- un arbre non vide a son nombre de feuille défini de la façon suivante :
 - si le nœud est une feuille alors on renvoie 1,
 - si c'est un nœud interne alors le nombre de feuille est la somme du nombre de feuille de chacun de ses fils.

Prototype de la fonction

```
int NombreFeuille( Nœud *ar);
```

Définition de la fonction

```
int NombreFeuille( Nœud *ar)
{
    if( EstVide(ar) )          return 0;
    else if ( EstUneFeuille(ar) ) return 1;
```

Arbre binaire



```
else
```

```
    return
```

```
        NombreFeuille(fils_gauche(ar)) + NombreFeuille(fils_droit(ar));
```

```
}
```

Nombre de nœud internes :

Nous allons calculer le nombre de nœud interne, Cela repose sur le même principe que le calcul du nombre de feuille.

La définition réursive est la suivante :

- un arbre vide n'a pas de nœud interne.
- si le nœud en cours n'a pas de fils alors renvoyer 0
- si le nœud a au moins un fils, renvoyer 1 plus la somme des nœuds interne des sous arbres.

Prototype de la fonction

```
int NombreNœudInterne( Nœud *ar);
```

Arbre binaire

Définition de la fonction

```
int NombreNœudInterne( Nœud *ar)
{ int p1,p2;
  if (EstVide(ar))          return 0;
  else if(EstUneFeuille(ar)) return 0;
else
  {
    p1= NombreNœudInterne(fils_gauche(ar));
    p2=NombreNœudInterne(fils_droit(ar))
    return 1+p1+p2;
  }
}
```



Arbre binaire

VI. Parcours d'un arbre :

Nous allons découvrir des algorithmes de parcours d'un arbre.

Cela permet de visiter tous les nœuds de l'arbre et éventuellement appliquer une fonction sur ces nœuds.

Nous distinguerons deux types de parcours : le parcours en profondeur et le parcours en largeur.

Le parcours en **profondeur** permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante.

Le parcours en **largeur** permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau un puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

Parcours en profondeur

Pour les trois fonctions de parcours, nous remarquerons que seul le placement de la fonction (ou procédure) `traiter_racine` diffère d'une fonction à l'autre.



Arbre binaire

Nous pouvons donc traiter ceci facilement, afin de ne pas à avoir à écrire trois fois de suite le même code.

Parcours en profondeur

Les types de parcours infixe, suffixe et postfixe sont les plus importants, en effet chacun à son application particulière. Nous verrons cela dans la suite.

Pour les trois fonctions de parcours, nous remarquerons que seul le placement de la fonction (ou procédure) `traiter_racine()` diffère d'une fonction à l'autre.

Nous pouvons donc traiter ceci facilement, afin de ne pas à avoir à écrire trois fois de suite le même code.

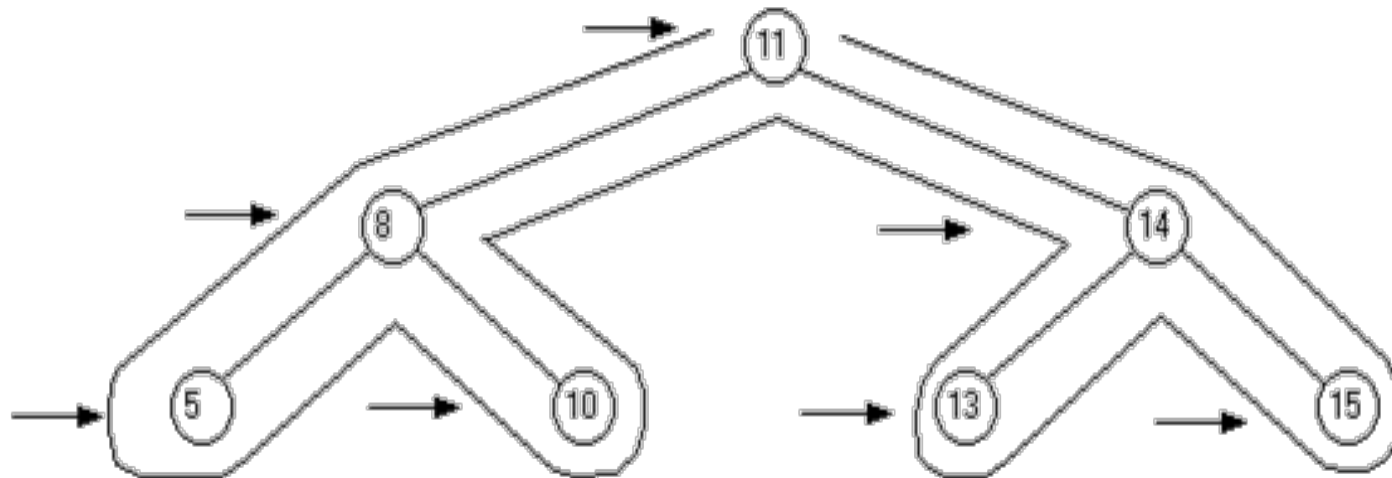
La fonction (ou procédure) `traiter_racine()`, est une fonction que vous définissez vous même, il s'agit par exemple d'une fonction d'affichage de l'élément , etc...

Arbre binaire

Parcours en profondeur préfixe(RGD)

On visite la racine, suivie de fils gauche et enfin le fils droit.

11,8,5,10,14,13,15

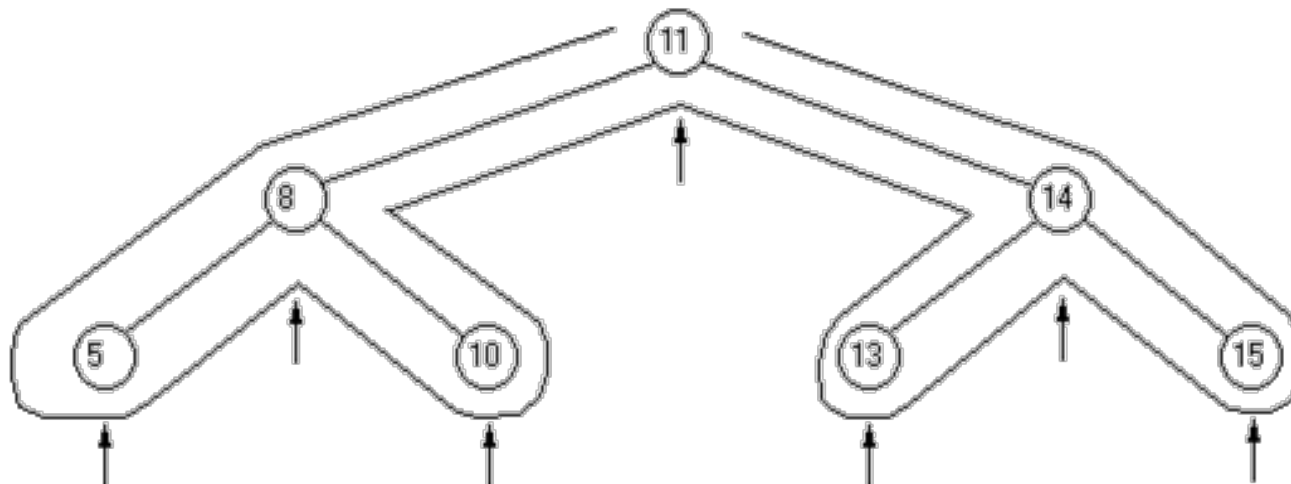


Arbre binaire

Parcours en profondeur infixe(GRD)

Un parcours infixe, comme ci-dessus, 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.

5,8,10,11,13,14,15

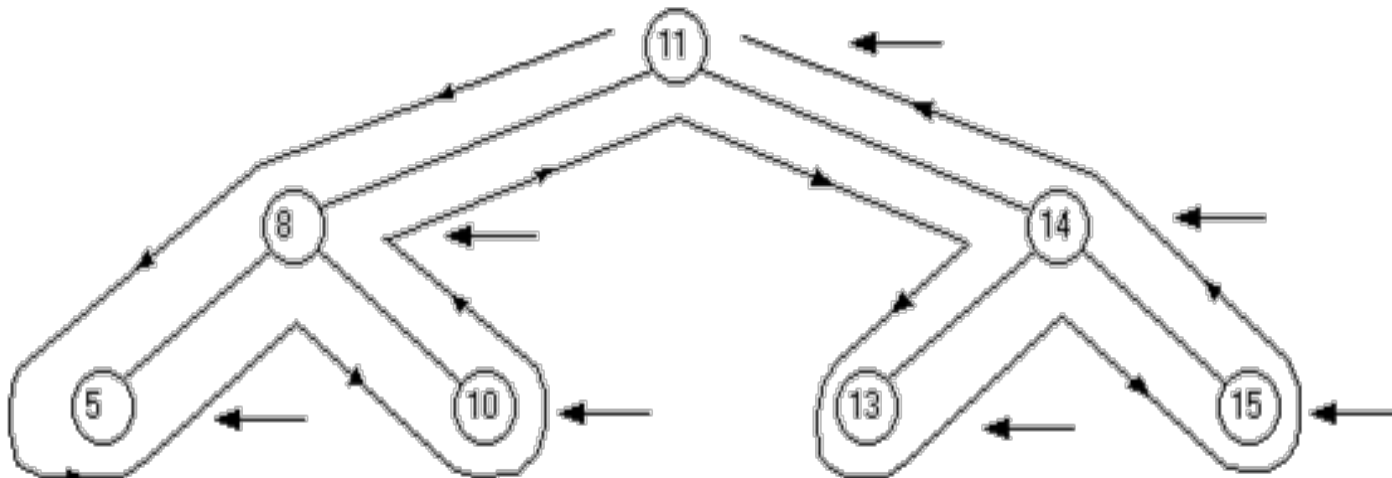


Arbre binaire

Parcours en profondeur postfixe(GDR)

Dans un parcours postfixe, on affiche chaque nœud après avoir affiché chacun de ses fils.

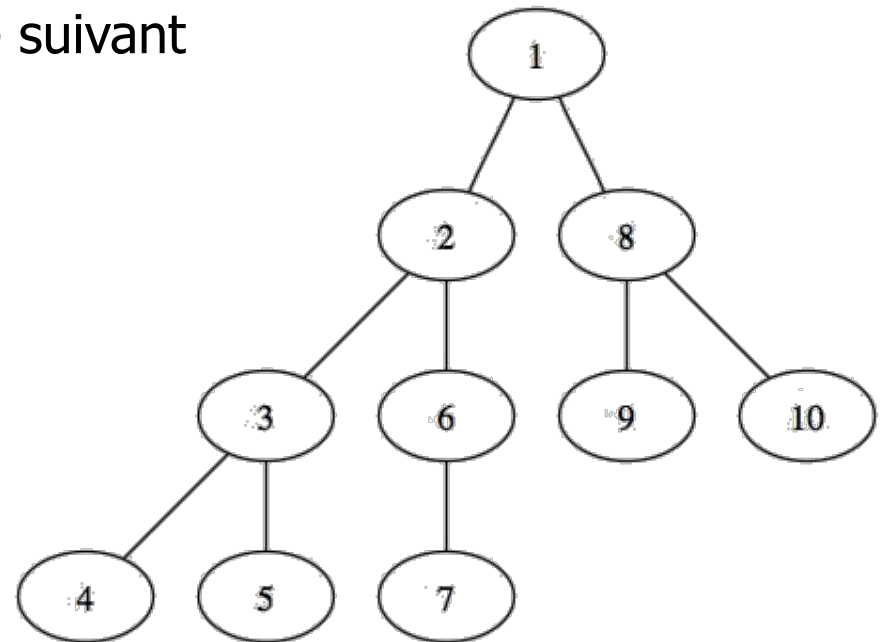
5,10,8,13,15,14,11



Arbre binaire

Parcours en profondeur

Le parcours en **profondeur** de l'arbre suivant donne : 1,2,3,4,5,6,7,8,9,10



Parcours **préfixe** de l'arbre : 1-2-3-4-5-6-7-8-9-10

Ceci n'est en fait qu'un seul parcours en profondeur de l'arbre.

Il s'agit d'un parcours d'arbre en profondeur à gauche d'abord et préfixe.



Arbre binaire

Prototype de la fonction

```
void DFS(Nœud *ar, int Type);
```

Définition de la fonction

```
void DFS(Nœud *ar, int Type)
{
    if( ! EstVide(ar) ){
        if( Type == 1 )
        { /* traiter racine */ }
        DFS(fils_gauche(ar), Type);
        if (Type == 2)
        { /* traiter racine */ }
        DFS(fils_droit(ar), Type);
        if( Type == 3)
        { /* traiter racine */ }
    }
}
```



Arbre binaire

à partir de ce code, on peut facilement créer trois fonctions qui seront respectivement le parcours préfixe, le parcours infixe et le parcours suffixe.

```
void DFS_prefix(Nœud *ar)
{
    DFS(ar,1);
}
void DFS_infix(Nœud *ar)
{
    DFS(ar,2);
}
void DFS_postfix(Nœud *ar)
{
    DFS(ar,3);
}
```

Arbre binaire



Les types de parcours infixe, suffixe et postfixe sont les plus importants, en effet chacun à son application particulière.

Pour les trois fonctions de parcours, nous remarquerons que seul le placement de la fonction (ou procédure) `traiter_racine` diffère d'une fonction à l'autre.

Nous pouvons donc traiter ceci facilement, afin de ne pas à avoir à écrire trois fois de suite le même code.

Arbre binaire



VII. Opérations élémentaires sur un arbre :

On définiras par la suite l'ensemble des opérations élémentaires sur un arbre:

- Création d'un arbre.
- Ajout d'un élément.
- Recherche dans un arbre.
- Suppression d'un arbre.



Arbre binaire

Création d'un arbre

On distingue deux types de création d'un arbre : création d'un arbre vide, et création d'un arbre à partir d'un élément et de deux sous arbres.

- **Création d'un arbre vide :**

Nous avons créé un arbre comme étant un pointeur, un arbre vide est donc un pointeur NULL.

La fonction de création d'un arbre vide est donc une fonction qui nous renvoie la constante NULL.

- **Création d'un arbre à partir d'un élément et de deux sous arbres :**

Il faut tout d'abord créer un nœud, ensuite, on place dans les fils gauches et droits les sous arbres que l'on a passés en paramètre ainsi que la valeur associée au nœud.

Enfin, il suffit de renvoyer ce nœud. En fait, ce n'est pas tout à fait exact, puisque ce n'est pas un nœud mais un pointeur sur un nœud qu'il faut renvoyer. Mais nous utilisons le terme nœud pour spécifier qu'il faut allouer un nœud en mémoire.

Arbre binaire

Créer une feuille

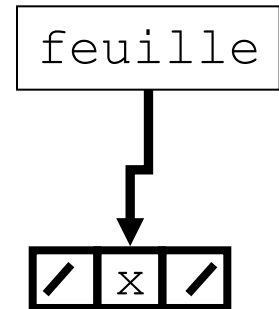
Prototype de la fonction

Nœud *creer_feuille(char x);

Définition de la fonction

Nœud *creer_feuille(char x)

```
{  
    Nœud *feuille;  
    feuille =(Nœud *)malloc(sizeof(Nœud));  
    if (feuille ==NULL) exit(-1);  
    else{  
        feuille ->val=x;  
        feuille ->gauche=NULL;  
        feuille ->droit=NULL;  
    }  
    return feuille;  
}
```



Arbre binaire

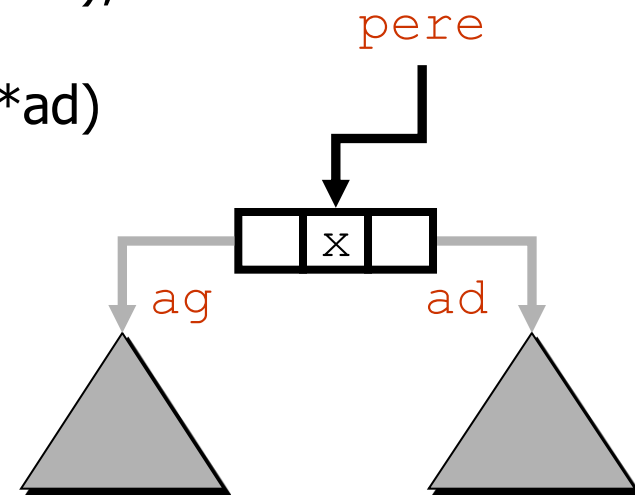
Création d'un arbre

Prototype de la fonction

```
Noued *creer_arbre(char x, Noued * ag, Noued *ad);
```

Définition de la fonction

```
Noued *creer_arbre(char x, Noued * ag, Noued *ad)
{
    Nœud *pere;
    pere =(Nœud *)malloc(sizeof(Nœud));
    if (pere ==NULL) exit(-1);
    else
    {
        pere ->val=x; pere ->gauche=ag;pere ->droit=ad;
    }
    return pere;
}
```



Arbre binaire



Ajout d'un élément

On distingue plusieurs cas :

Soit on insère dès que l'on peut. Soit on insère de façon à garder une certaine logique dans l'arbre.

Le premier type d'ajout est le plus simple, dès que l'on trouve un nœud qui a un fils vide, on y met le sous arbre que l'on veut insérer.

Cependant ce type de technique, si elle sert à construire un arbre depuis le début à un inconvénient : on va créer des arbres du type peigne. C'est à dire que l'on va insérer notre élément tel que l'arbre final ressemblera à ceci.

Arbre binaire



Ceci est très embêtant dans la mesure où cela va créer des arbres de très grande hauteur, et donc très peu performant.

Néanmoins, il peut arriver des cas où on a parfois besoin de ce type d'insertion. Dans ce cas, l'insertion se déroule en deux temps : on cherche d'abord un fils vide, puis on insère.

Ceci peut s'écrire récursivement :

- si l'arbre dans lequel on veut insérer notre élément est vide, alors il s'agit d'une création d'arbre.
- sinon, si le nœud en cours a un fils vide, alors on insère dans le fils vide.
- sinon, on insère dans le fils gauche.

Nous insérons du côté gauche, donc produire un peigne gauche, et si on insérerait du côté droit, nous aurions un peigne droit.

Arbre binaire



Prototype de la fonction

```
void AjouteNœud(Nœud *ar, Information val);
```

Définition de la fonction

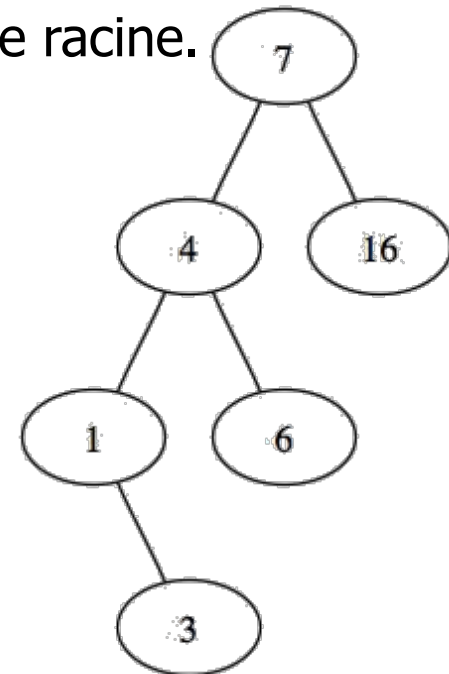
```
void AjouteNœud(Nœud *ar, Information val)
{
    if ( ar == NULL ) {ar = CreerArbre(val,NULL,NULL);}
    else if ( EstVide(fils_gauche(ar)) ) {
        ar->gauche = CreerArbre(val,NULL,NULL); }
    else if ( EstVide(fils_droit(ar)) ) {
        ar->droit = CreerArbre(val,NULL,NULL); }
    else
    {
        AjouteNœud(fils_gauche(ar),val);
    }
}
```

Arbre binaire

Ajout d'un élément dans un arbre binaire de recherche

Dans ce type d'arbre, il y a une cohérence entre les nœuds, c'est à dire que la hiérarchie des nœuds respecte une règle. Pour un arbre binaire de recherche contenant des entiers, nous considérerons que les valeurs des nœuds des sous arbres gauches sont inférieures à la racine de ce nœud et les valeurs des sous arbres droit sont supérieurs à cette racine.

Voici un exemple d'un tel arbre :





Arbre binaire

Le principe d'insertion est simple :

si on a un arbre vide, alors il s'agit de la création d'un arbre. Sinon, on compare la valeur de l'élément à insérer avec la valeur de la racine. Si l'élément est plus petit alors on insère à gauche. sinon, on insère dans le fils droit de la racine.

Prototype de la fonction

```
void InsereArbreRecherche(Nœud *ar, Information x);
```

Définition de la fonction

```
void InsereArbreRecherche(Nœud *ar, Information x)
{
    if( EstVide(ar) ){ar = CreerArbre(val,NULL,NULL);}
    else{
        if (x< ar->val) {InsereArbreRecherche(fils_gauche(ar), x);}
        else{ InsereArbreRecherche(fils_droit(ar), x);}
    }
}
```




Arbre binaire

Vous remarquerez que l'on insère les éléments qui sont égaux à la racine du côté droit de l'arbre.

Autre remarque, vous constaterez que nous effectuons des comparaisons sur les entités du type `Information`, ceci n'est valable que pour des valeurs numériques (entier, flottant et caractère). S'il s'agit d'autres types, vous devrez utiliser votre propre fonction de comparaison. (comme `strcmp` pour les chaînes de caractère).

Arbre binaire



Recherche dans un arbre

Il y a principalement deux méthodes de recherche. Elles sont directement liées au type de l'arbre : si l'arbre est quelconque et si l'arbre est un arbre de recherche.

Nos recherches se contenteront seulement de déterminer si la valeur existe dans l'arbre. Avec une petite adaptation, on peut récupérer l'arbre dont la racine contient est identique à l'élément cherché.



Arbre binaire

Recherche dans un arbre

Il y a principalement deux méthodes de recherche. Elles sont directement liées au type de l'arbre : si l'arbre est quelconque et si l'arbre est un arbre de recherche.

Nos recherches se contenteront seulement de déterminer si la valeur existe dans l'arbre. Avec une petite adaptation, on peut récupérer l'arbre dont la racine contient est identique à l'élément cherché.

Dans un arbre quelconque; on cherche dans tous les nœuds de l'arbre l'élément. Si celui ci est trouvé, on renvoie vrai (1), si ce n'est pas le cas, on renvoie faux (0).



Arbre binaire

Prototype de la fonction

```
int Existe(Noeud *ar, Information x);
```

Définition de la fonction

```
int Existe(Noeud *ar, Information x)
{
    if Estvide(ar) return 0;
    else
    {
        if (ar->val==x) return 1;
        else
            return (Existe(fils_gauche(ar, x)+Existe(fils_droit(ar, x)));
    }
}
```

Nous renvoyons un ou logique (+) entre le sous arbre gauche et le sous arbre droit, pour pouvoir renvoyer vrai (1) si l'élément existe dans l'un des sous arbres et faux sinon.



Arbre binaire

Ce genre de recherche est correct mais n'est pas très performante. En effet, il faut parcourir quasiment tous les nœuds de l'arbre pour déterminer si l'élément existe.

Recherche dans un arbre de recherche

C'est pour cela que sont apparus les arbres binaires de recherche. En effet, on les nomme ainsi parce qu'ils optimisent les recherches dans un arbre. Pour savoir si un élément existe, il faut parcourir seulement une branche de l'arbre. Ce qui fait que le temps de recherche est directement proportionnel à la hauteur de l'arbre.

L'algorithme se base directement sur les propriétés de l'arbre, si l'élément que l'on cherche est plus petit que la valeur du nœud alors on cherche dans le sous arbre de gauche, sinon, on cherche dans le sous arbre de droite.



Arbre binaire

Prototype de la fonction

```
int Existe(Noeud *ar, Information val);
```

Définition de la fonction

```
int Exist(Noeud *ar , Information val)
{
    if ( EstVide(ar) ) return 0;
    else if ( ar->value == val ) return 1;
    else if ( ar->donnee > val )
        return Exist(fils_gauche(ar), val);
    else
        return Exist(fils_droit(ar), val);
}
```



Arbre binaire

Suppression d'un arbre:

Par suppression de nœud, nous entendrons suppression d'une feuille.
En effet, un nœud qui possède des fils s'il est supprimé, entraîne une réorganisation de l'arbre.

Que faire alors des sous arbres du nœud que nous voulons supprimer ?

La réponse à cette question dépend énormément du type d'application de l'arbre. On peut les supprimer, on peut réorganiser l'arbre (si c'est un arbre de recherche) en y insérant les sous arbres qui sont devenus orphelins.

Bref, pour simplifier les choses, nous allons nous contenter de supprimer complètement l'arbre.

Arbre binaire



L'algorithme de suppression de l'arbre est simple :

On supprime les feuilles une par une. On répète l'opération autant de fois qu'il y a de feuilles. Cette opération est donc très dépendante du nombre de nœud.

En fait cet algorithme est un simple parcours d'arbre. En effet, lorsque nous devons traiter la racine, nous appellerons une fonction de suppression de la racine. Comme nous avons dit plutôt que nous ne supprimerons que des feuilles, avant de supprimer la racine, il faut supprimer les sous arbres gauche et droit.

On en déduit donc que l'algorithme de suppression est un parcours d'arbre postfixe.



Arbre binaire

Prototype de la fonction

```
void supprime(Nœud *ar);
```

Définition de la fonction

```
void supprime(Nœud *ar)
{
    Nœud *g = fils_gauche(ar);
    Nœud *d = fils_droit(ar);

    if( ! EstVide(ar ){
        supprime( g );
        supprime( d );

        free( ar );
        ar = NULL;
    }
}
```

Fin du chapitre 4

