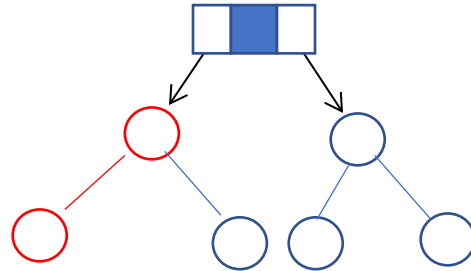
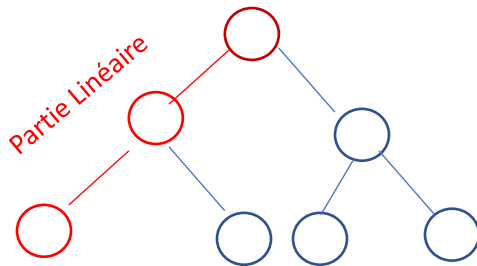


COURS M1 : ALGORITHME AVANCE

Introduction

Liste linéaire : vecteur, fichier, liste chaînée

Liste non-linéaire : table, arbre



Arbre \Leftrightarrow liste chaînée à double pointeur (2 pointeurs et 1 partie info)

Tableau \Leftrightarrow variable indicé

Déclaration variable \Leftrightarrow réservation d'une place mémoire pour que le var existe (occupation de la case mémoire)

Int tab [100] ;

- tab : adresse ou pointeur
- *tab : variable

LE TABLEAU

1. Algorithme de Parcours (à l'endroits)

Parcours Itératif	Parcours Récursif
<pre>void parcours (int t [], int n) { int i ; i = 0 ; while (i < n) { traiter (t [i]) ; i++ ; } }</pre>	<pre>void parcours (int t [], int i, int n) { if (i < n) { traiter (t[i]) ; parcours (t, i+1, n); } }</pre>

Appel récursif \Leftrightarrow appel d'une fonction à l'intérieur d'elle-même

\Leftrightarrow cas général + cas particulier (condition d'arrêt)

\Leftrightarrow si on ne connaît pas le condition d'arrêt, met dans le cas général

Exemple Somme des entiers (Sn)
$S = 1 + 2 + 3 + \dots + n$ $S(n) = n + S(n-1)$ <pre>int sommeEntier (int n) { if (n > 0) { return n + sommeEntier(n - 1) ; } }</pre>

NB : il y a aussi le parcourt à l'envers.

2. Algorithme d'accès à un élément dans un tableau

Parcours Itératif	Parcours Récursif
<pre>booléen acces (int t [], int n, int elm) { int i; booléen found; i = 0; found = false; while ((i < n) && (! found)) { if(t[i] == elm) { found = true; } else i++; } return found; }</pre> <p>booléen n'existe pas en langage C, vous pouvez utiliser int de valeur 0 ou 1</p>	<pre>booléen acces (int t [], int n, int elm) { if (n < 0) return false ; else if (t[n] == elm) return true; else return acces (t, n-1, elm); }</pre>

3. Recherche dichotomique

- Précondition \Leftrightarrow tous les éléments doivent être triés
- Recherche d'élément
- Milieu $m = (n + 1) / 2$
 - Si $t[m] == elm \Leftrightarrow$ arrêt
 - Si $elm < t[m] \Leftrightarrow$ recherche dans $t [inf... m-1]$
 - Si $elm > t[m] \Leftrightarrow$ recherche dans $t [m+1.... Sup]$

Parcours Itératif	Parcours Récursif
<pre> booléen acces (int t [], int n, int elm) { int inf, sup, m; booléen found; found = false; if ((elm >= t[1]) && (elm <= t[n])){ inf = 1; sup = n; while ((inf <= sup) && (! found)) { m = (inf + sup) / 2; if(t[m] == elm) found = true; else if(t[m] < elm) inf = m+1; else sup = m - 1; } return found; } } </pre> <p>booléen n'existe pas en langage C, vous pouvez utiliser int de valeur 0 ou 1</p>	<pre> booléen dichotomie (int t [], int inf, int sup, int elm) { if (inf > sup) return false; else { m = (inf + sup) / 2; if (elm < t[m]) return dichotomie (t, inf, m-1, elm); else if (elm > t[m]) return dichotomie (t, m+1, sup, elm); else return true; } } </pre>

4. Algorithme de Tri

Tri lent : tri par remplacement, par permutation, par insertion, et méthode de bulle

Tri rapide : tri par segmentation, par interclassement

- **Tri par insertion**

- Efficace pour faire le tri d'un petit nombre d'éléments
- On a besoin de fonctions : permettant d'effectuer une insertion + faire le tri


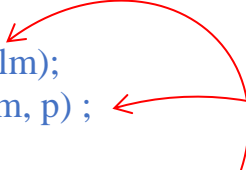
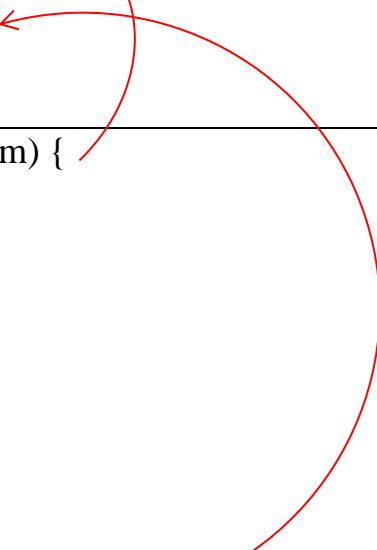
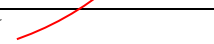
Donné-Résultat ⇔ passage par pointeur

Donné ⇔ passage par valeur

int *n ;

n = 6050
*n = 2

*n = 2 ;

Tri par Insertion	
<pre>void triInsertion (int t [], int n) { int i ; i = 0 ; while (i < n) { insertion (t, i, t[i+1]); i++ ; } }</pre>	
<pre>void insertion (int t [], int *n, int elm) { int p; if (*n < 0) { t [0] = elm; *n = 1; } else { p = position (t, n, elm); insertPlace (t, n, elm, p) ; } }</pre>	 
<pre>void position (int t [], int n, int elm) { int i, p ; if (t [0] > elm) p = 1 ; else { i = n - 1; while (t [i] > elm) i--; p = i + 1; } return p ; }</pre>	
<pre>int insertPlace (int t [], int n, int elm, int p) { for (i = n - 1; i >= p; i++) t [i + 1] = t [i]; n++; t [p] = elm; return n; }</pre>	

Partie déjà trié

4	10	12	40	21	15
---	----	----	----	----	----

t t [i+1]

- **Tri par segmentation (Quicksort)**

- Segmenter le tableau en 3 sous-tableaux
- t [place] est déjà, donc, seulement trié les 2 sous-tableaux

inf	place	sup
-----	-------	-----

t [inf...place - 1] <= t [place] < t [place+1...sup]

Tri par Segmentation
<pre> void triSegmentation (int t [], int inf, int sup) { int place ; if (inf < sup) { place = segmentation (t, inf, sup) ; triSegmentation (t, inf, place-1) ; triSegmentation (t, place+1, sup) ; } } </pre>
<pre> int segmentation (int t, int inf, int sup) { int i, j, pivot, place; pivot = t[inf]; i = inf + 1; j = sup; while (i <= j) { if (t[i] <= pivot) i++; else { permut(t[i], t[j]) ; j-- ; } } Permut(t[inf], t[j]) ; place = j ; return place ; } </pre>

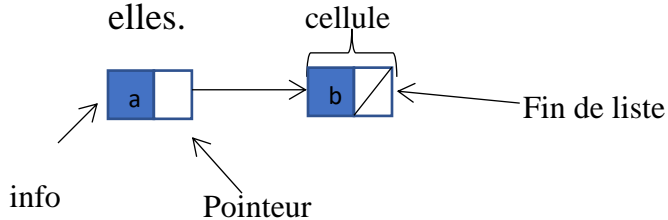
- **Tri par interclassement**

- Trier la 1ere moitié du tableau, puis la second
- Interclasser les 2 sous-tableaux trié

Tri par Interclassement
<pre>void triInterclassement (int t [], int inf, int sup) { int milieu, *nb3; if (inf < sup) { milieu = (inf + sup) / 2 ; triInterclassement (t1, inf, milieu) ; triInterclassement (t2, milieu+1, sup) ; interClasser(t1, milieu, t2, sup, t3, nb3) ; } }</pre>
<pre>void interClasser (int t1[], int n1, int t2[], int n2, int t3[], int *n3) { int i, j, k; i = k = 1; j = n1 + 1; while ((i <= n1) && (j <= n2)) { if (t1[i] <= t2[j]) { t3[k] = t1[i]; i = i + 1; } else { t3[k] = t2[j]; j = j + 1; } k = k + 1 ; } *n3 = k ; }</pre>

LISTE LINEAIRE CHAINEE

Une liste linéaire chaînée est constituée d'un ensemble de cellules chaînées entre elles.



1. Algorithme de création d'une liste chaînée

Définition d'une liste
<pre>struct cellule { [type] info ; struct cellule * suivant ; } struct cellule *liste ; typedef struct cellule * pointeur ; liste l ; l → info // obtenir la valeur de info l → suivant // pointer à la valeur suivant l → suivant → info l → suivant → suivant → info ...</pre>
Création d'une liste à partir de son dernière élément
<pre>struct cellule { type info ; struct cellule * suivant ; } typedef struct cellule * pointeur ; pointeur l, p ;</pre>


```

l = NULL ; // création d'une liste vide d'adresse l
p = (pointeur) malloc(sizeof(struct cellule)) ; // creation de cellule p contenant a
p→info = a;
p→suivant = l; // chaînage avec l
l = p ;

```

Exemple de liste chaîné : Passage d'un fichier à une liste

```

pointeur creerListe(FILE *f){
    pointeur l, p ;
    int val ;

    l = NULL ;
    f = fopen(« fich.txt », « r »);

    while(!feof){
        p = (pointeur) malloc(sizeof(struct cellule));
        fscanf(f, "%d", &val);
        p→info = val;
        p→suivant = l;
        l = p ;
    }
    return l;
}

```

2. Algorithme de Parcours d'une liste

- l = NULL (cas d'arrêt)
- l ≠ NULL (cas général)

Parcours Récursif	
Parcours à l'endroit	Parcours à l'envers
<pre> void parcours(pointeur l) { if(l != NULL) { traiter(l→info); parcours(l→suivant); } } </pre>	<pre> void parcours(pointeur l) { if(l != NULL) { parcours(l→suivant); traiter(l→info); } } </pre>

Parcours Itératif
<pre>void parcours(pointeur l) { while(l != NULL) { traiter(l→info); l = l→suivant ; } }</pre>

3. Algorithme d'accès dans une liste

Algorithme d'accès	
Schema itératif	Schema récursif
<pre>booléen acces(pointeur l, t val, pointeur *point){ booléen found = false; while((l != NULL) && (!found)) { if(l→info == val) found = true; else l = l→suivant; } *point = l; return found; }</pre>	<pre>pointeur acces(pointeur l, t val) { if(l == NULL) return NULL; else { if(l→info == val) return l; else return acces(l→suivant, val); } }</pre>

4. Accès associatifs dans une liste triée

Une liste triée :

- Une liste vide ou avec 1 élément est une liste triée
- Une liste avec plus de 2 éléments est triée si :

liste # NULL ; liste →suivant # NULL ;

liste →info <= liste →suivant →info ;

Schéma récursif
<pre> pointeur acces(pointeur l, t val) { if(l == NULL) return NULL; else { if(l->info < val) return acces(l->suivant, val); else { if(l->info > val) return NULL; else return l; } } } </pre>

5. Algorithme de Mise à Jour

Insertion	
Insertion en tête de liste	Insertion en fin de liste
<pre> void insertTete(pointeur *l, t elm) { pointeur p ; p = (pointeur) malloc(sizeof(struct cellule)); p->info = elm; p->suivant = *l; *l = p; } </pre>	<pre> void insertFin(pointeur l, t elm) { pointeur der, p ; if(l == NULL) insertTete(&l, eml); else { der = dernier(l); p = (pointeur) malloc(sizeof(...)); p->info = elm; p->suivant = NULL; der->suivant = p; } return l; } </pre>
Fonction dernier	
Forme itérative	Forme récursive
<pre> pointeur dernier(pointeur l) { pointeur precedent ; while (l != NULL) { precedent = l; l = l->suivant; } return precedent; } </pre>	<pre> pointeur dernier(pointeur l) { if(l->suivant == NULL) return l; else return dernier(l->suivant); } </pre>

Suppression	
Schéma itératif	Schéma récursif
<pre> booléen supprimek(pointeur *l, int k) { pointeur ptk, precedent ; booléen possible ; if((*l) != NULL && (k==1)) { possible = true; supTete(l); } else { possible = false; precedent = pointk(l, k-1); if(precedent != NULL) { ptk = precedent→suivant; if(ptk != NULL) { possible = true; precedent→suivant = ptk →suivant; free(ptk); } } } return possible; } </pre>	<pre> Void supprimek(pointeur *l, int k, booléen *possible) { if(*l == NULL) *possible = false; else { if(k==1){ *possible = true; supTete(l); } else { supprimek((*l)→suivant, k-1, *possible) ; } } } </pre>

LISTE LINEAIRE PARTICULIERE

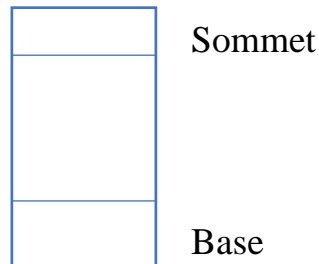
1) PILE

Pile → une liste linéaire

Mise à jour se font à partir de sommet

Jargon :

- Empiler(val)
- Depiler(val)
- Pile vide → $\text{sommet} = 0$, ou $\text{sommet} = -1$ en langage c
- Pile Plein → $\text{sommet} = \text{dimpile}$, ou $\text{sommet} = \text{dimpile} - 1$ en c
- SommetPile (), initPileVide ()



1.1. Vecteur Pile

Algorithme Empiler(val)

```
void empiler (t val) {  
    if(pilePlein()) {  
        printf("Pile Plein");  
        exit(1);  
    }  
  
    sommet += 1;  
    pile[sommet] = val;  
}
```

NB : *pile*, *dimpile* et *sommet* sont globales

Algorithme Depiler(val)
<pre> void dePiler (t *val) { if(<i>pileVide</i>()) { printf(“Pile Vide”); exit(1); } else { *val = pile[sommet] ; sommet -= 1 ; } } </pre> <p>NB : <i>pile, dimpile et sommet</i> sont globales</p>
Sommet Pile
<pre> void sommetPile () { return pile[sommet] ; } </pre> <p>NB : <i>pile, dimpile et sommet</i> sont globales</p>
Pile est vide
<pre> booléen pileVide() { return sommet == 0 ; } </pre>
Pile est Plein
<pre> booléen pilePlein() { return sommet == dimpile ; } </pre>
Initialiser une Pile
<pre> void initPileVide() { sommet = 0 ; } </pre>

1.2. Représentation Chaînée d'une Pile

Algorithme Empiler(val)
<pre>void empiler (t elm) { insertTete(&pile, elm) ; }</pre> <p>NB : <i>pile</i>, <i>pointeur</i> sont globales</p>
Depiler(val)
<pre>void depiler(t elm) { if(pileVide()) exit(1) ; else { *elm = pile→info ; suppTete(&pile) ; } }</pre>
Pile Vide
<pre>booléen pileVide() { return (pile == NULL); }</pre>
Initializer Pile Vide
<pre>void initPileVide() { pile = NULL ; }</pre>
Sommet Pile
<pre>t initPileVide() { return pile→info; }</pre>

2. TRAITEMENT DES EXPRESSIONS

2.1. Definition

- **Expression complètement parenthésée (excp)**
 - Une variable = une expression complètement parenthésée
 - Soit x et y sont excp, **opbin** un opérateur binaire $\Rightarrow (x \text{ opbin } y)$ est un excp
 - Soit x un excp, **opun** un opérateur unaire $\Rightarrow (\text{opun } x)$ est un excp

NB : Selon les règles de grammaire de BNF (Backus-Naur Form)

- **excp** : $\langle \text{variable} \rangle$; $(\langle \text{excp} \rangle \langle \text{opbin} \rangle \langle \text{excp} \rangle)$; $(\langle \text{opun} \rangle \langle \text{excp} \rangle)$
- **opbin** : $+$, $-$, $*$, $/$, $<$, $<=$, $=$, $>=$, $>$, $\#$, \wedge , \vee
- **opun** : Δ , ∇ , \neg ($+$ et $-$ unaires, $!$ pour les opérateurs logiques).

Ex :

- **excp** $\Rightarrow ((A+B) * C); (((A/B) = C) \wedge (E < F)); (\nabla A);$
- **Non excp** $\Rightarrow (A), ((A+B)), A \nabla B$

- **Expression sous forme préfixé (expref)**

Dans la grammaire de BNF, les règles sont :

- **expref**: $\langle \text{variable} \rangle$; $(\langle \text{opbin} \rangle \langle \text{expref} \rangle \langle \text{expref} \rangle)$; $\langle \text{opun} \rangle \langle \text{expref} \rangle$

ex: $+ - ABC \Rightarrow ((A-B) + C); \wedge \neg < A B C \Rightarrow ((\neg (A < B)) \wedge C)$

- **Expression sous forme postfixé (expost)**

Dans la grammaire de BNF, les règles sont :

- **expost** : $\langle \text{variable} \rangle$; $(\langle \text{expost} \rangle \langle \text{expost} \rangle \langle \text{opbin} \rangle)$; $\langle \text{expost} \rangle \langle \text{opun} \rangle$

ex : $((A-B) + C) \Rightarrow AB-C+$

- *Expression sous forme infixé (expinf)*

- C'est l'écriture habituel dans les langages de programmation
- Suppression des certaines parenthèses si l'ordre des priorités est admis

Ex : $((((A / B) * C) - (C / (D * E))) - (F - G)) \Rightarrow A/B * C - C/(D * E) - (F - G)$

Voir la grammaire de BNF dans le slide n°34

2.2. Evaluation d'une expression

Pour évaluer une expression, il faut utiliser **la pile**

- *Evaluation d'expression postfixé (expost)*

- **tind** \Rightarrow type indifférencié

faux \Rightarrow 0 ; **vrai** \Rightarrow autre que 0 et positif

B \Rightarrow booléen ; **N** \Rightarrow Numérique

D'où, faux \Rightarrow (B, 0) ; vrai \Rightarrow (B, 1) ; 1 \Rightarrow (N, 1)

Déclaration de tind en C
<pre>typedef struct tind{ char car ; int val ; }</pre>

Algorithme évaluation d'expression postfixé

Soit v , un vecteur de caractère

Soit $\#$, marque le signe de la fin de vecteur

- Si $v[i] == "\#"$:
 Arrêt de la pile
 Retourner la valeur au sommet de la pile

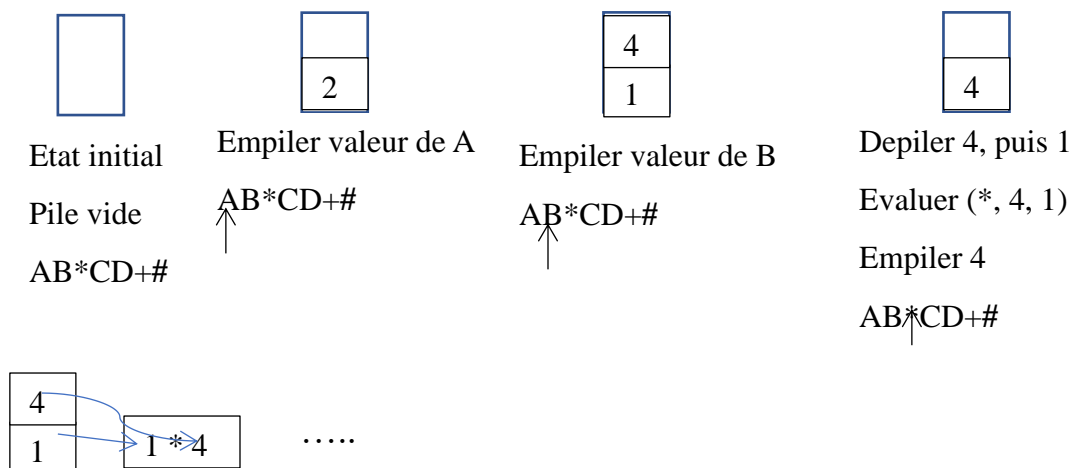
- Sinon :
 Si $\text{variable}(v[i])$:
 empiler (**valeur**($v[i]$))
 $i++$
 → H

- Sinon si $\text{opérateur}(v[i])$:
 depiler son/ses opérandes
 empiler résultat opération
 $i++$
 → H

Evaluation d'une expost à l'aide d'une pile

Soit une expression postfixé : $AB*CD+$

Avec l'évènement : $A = 2$; $B = 4$; $C = 9$; $D = 7$;



NB : initialisation Pile vide

- Vecteur => sommet = 1 / 0 en C
- Liste => sommet = NULL
- Résultat = valeur au sommet du pile

Evaluation Expression Postfixé



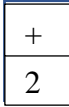
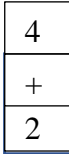


```
tind evalpost(char expost[]){
    tind vald, valg, evalp;
    int i;

    initPileVide();
    i = 1;

    while(expost[i] != '#') {
        if(variable(expost[i])) {
            empiler(valeur(expost[i]));
        }
        else if(unaire(expost[i]) {
            depiler(vald);
            empiler(operationUnaire(expost[i], vald);
        }
        else {
            depiler(vald);
            depiler(valg);
            empiler(operationBinaire(valg, expost[i], vald) ;
        }
        i++ ;
    }
    depiler(evalp) ;
    return evalp ;
}
```

- *Evaluation d'expression complètement parenthésée (excp)*

Algorithme évaluation d'expression complètement parenthésée
<p>Soit excp, symbole lu dans une expression complètement parenthésée</p> <ul style="list-style-type: none"> - Si excp est <i>une parenthèse gauche</i> : On ne fait rien - Sinon si c'est une <i>variable</i> : empiler la valeur de excp ; - Sinon si c'est <i>une parenthèse droite</i> : depiler les sous-expressions empiler le résultat de son évaluation

Evaluation d'une excp à l'aide d'une pile
<p>Soit une expression : (A+B) Avec l'évènement : A = 2 ; B = 4 ; C = 9 ;</p> <div style="display: flex; justify-content: space-around; align-items: flex-start;"> <div style="text-align: center;">  <p>Etat initial Pile vide (A+B)# ↑</p> </div> <div style="text-align: center;">  <p>Empiler valeur de A (A+B)# ↑</p> </div> <div style="text-align: center;">  <p>Empiler operateur * (A+B)# ↑</p> </div> <div style="text-align: center;">  <p>Empiler valeur de B (A+B)# ↑</p> </div> </div> <div style="display: flex; justify-content: space-around; align-items: flex-start; margin-top: 20px;"> <div style="text-align: center;">  <p>Dépiler et faire l'opération 2 + 4 = 6 Empiler le résultat 6 (A+B)# ↑</p> </div> <div style="text-align: center;">  <p>Fin de pile, depiler résultat (A+B)# ←</p> </div> </div>

Evaluation Expression Complètement Parenthésée

```
tind evalcp(t excp[]){
    tind vald, valg, evalc;
    char op;
    int i;

    initPileVide();
    i = 1;

    while(excp[i] != '#') {
        if(variable(excp[i])) {
            empiler(valeur(excp[i]));
        }
        else if(opérateur(excp[i]) {
            empiler(excp[i]);
        }
        else {
            if(excp[i] == '(') {
                depiler(vald);
                depiler(op);

                if(unaire(op)) {
                    empiler(operationUnaire(op, vald));
                }
                else {
                    depiler(valg);
                    empiler(operationBinaire(valg, op, vald));
                }
            }
        }
        i++;
    }
    depiler(evalc) ;
    return evalc ;
}
```

2.3. Passage d'une expression à une autre

- *Expression complètement parenthésée en Expression postfixé*

Algorithme de transformation
Soit excp , symbole lu dans une expression complètement parenthésée et expost , une variable
<ul style="list-style-type: none">- Si excp est <i>un opérateur</i> : empiler excp- Sinon si c'est une <i>variable</i> : ranger excp dans expost ;- Sinon si c'est <i>une parenthèse droite</i> : depiler un opérateur que l'on range dans expost

Excp en Expost
Soit une excp : (A+B)
<div>(A+B) # ↑ expost: A</div>
<div>(A+B) # ↑ empiler +</div>
<div>(A+B) # ↑ expost: AB</div>
<div>(A+B) # ↑ Depiler + expost: AB+</div>

- *Expression infixée en Expression postfixé*

Algorithme de transformation	
Soit expinf , symbole lu dans une expression infixé et expost , une variable	
<ul style="list-style-type: none"> - Les variables sont rangées directement dans expost - Les opérateurs sont empilés (tenir compte de la priorité) - Empiler les parenthèses gauches - Pour les parenthèses droites, depiler tous les operateurs jusqu'à la parenthèse gauche et les ranger dans expost 	
Ex :	
<ul style="list-style-type: none"> - Expinf: $A * B + C / (D * E)$ - Expost: $AB * CDE * / +$ 	

NB: Priorité des opérateurs

Priorité	Opérateurs
0	(
1	&&
2	== !=
3	< <= >= >
4	+ -
5	* / %
6	! ++ --

STRUCTURE DE DONNÉES DYNAMIQUE NON LINEAIRE

ARBRE

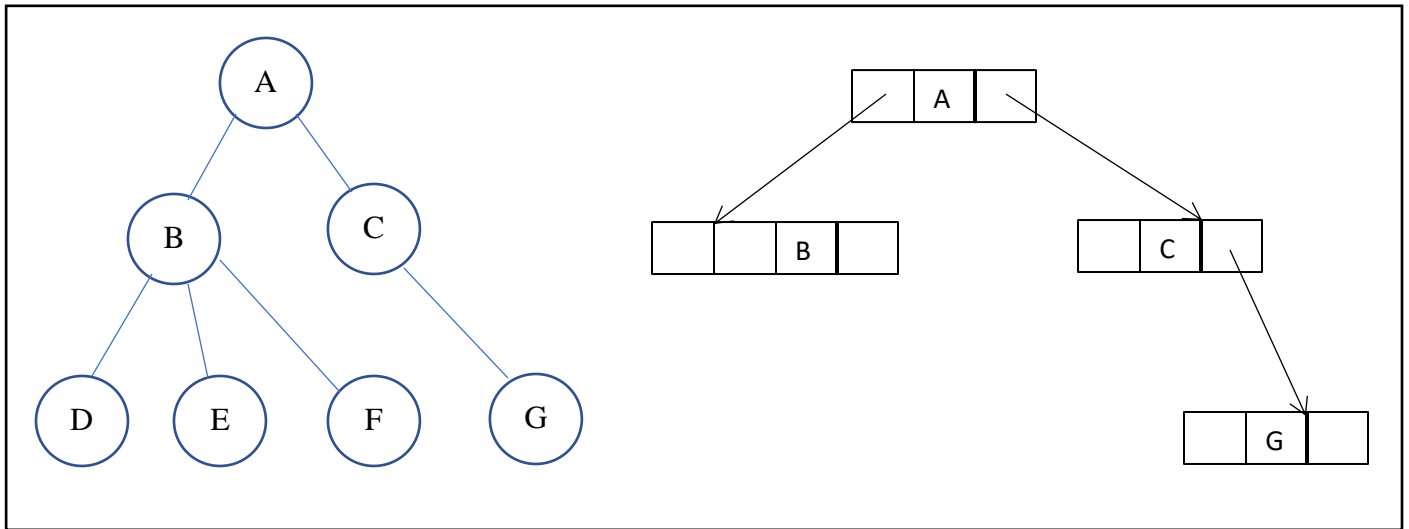
Arbre : une liste chaînée contenant au moins une partie *info* et *deux pointeurs*

Arborescence : arbre généalogique, dictionnaire

Feuille : nœud qui n'a pas de fils (nœud externe)

Racine : nœud interne (parent)

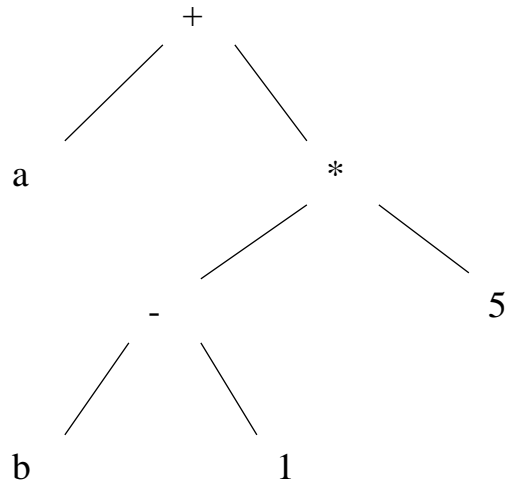
On peut représenter une relation d'inclusion entre plusieurs ensembles par une arborescence



NB :

- Tous les opérateurs se trouvent dans les nœuds des arbres, tandis que les opérands aux feuilles
- Même priorité => plus de priorité à gauche
- Le plus prioritaire à la feuille
- Une liste est un cas particulier d'une arborescence
- Une arborescence est un graphe particulier tel que :
 - Il existe un chemin unique d'un sommet à l'autre
 - Il n'y a pas de cycle

Arborescence représenté l'expression $a + (b-1) * 5$



Autre exemple : $a + b - 1 * 5$

Définition et terminologie

- **Graphe orienté** : ensemble fini des **nœuds** (sommets) associé à un ensemble fini des **arcs** joignant chacun un nœud à un autre
- **Graphe non orienté** : ensemble des nœuds et des arcs qui n'ont d'origine ni d'extrémité
- **Degré d'un nœud** = nombre de fils de ce nœud
- **Une feuille** = un nœud de degré 0
- **Profondeur** = niveau - 1 avec racine = niveau 1

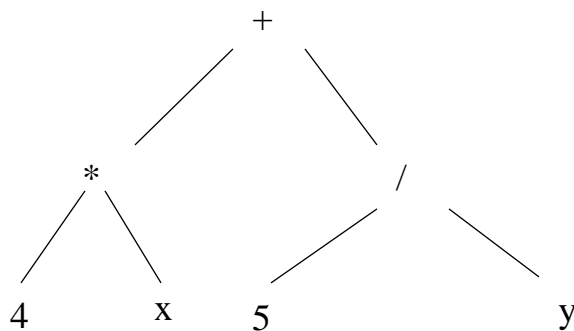
ARBRE BINAIRE

- **Arbre binaire** : un arbre où chaque nœud a **0, 1, ou 2 fils** (un fils gauche, un fils droit)
- **Arbre binaire complet** : chaque nœud a 0 ou 2 fils
- **Arbre binaire plein** : arbre binaire complet et toutes les feuilles sont des mêmes niveaux
- **Arbre n-aire** : arbre formé par n fils
- **Taille d'un arbre binaire** = nombre de nœuds
- **Hauteur d'un nœud** = maximum du niveau des feuilles
- **Hauteur d'un arbre** : C'est la hauteur de sa racine
- **Hauteur d'un arbre vide** = nulle
- **Hauteur d'un nœud** = max des hauteurs de sous-arbres gauche et du sous-arbre droit plus 1
- **Facteur d'équilibre de sous arbre** est associé à la racine
- **Facteur d'équilibre de nœud** = hauteur de sous-arbre gauche – hauteur de sous arbre droite

Représentation d'un arbre binaire sous-forme parenthésé

Racine (Sous-Arbre-Gauche, Sous-Arbre-Droite)

Exemple : $((4 * x) + (5 / y))$



$\Rightarrow +(* (4, x), / (5, y))$

Construction des arbres binaires

```
struct nœud {  
    struct nœud *gauche ;  
    type info ;  
    struct nœud *droite ;  
};  
typedef struct nœud *pointeur ;
```

Algorithme de parcours d'un arbre binaire

Il y a 4 algorithmes de parcours :

- **Préfixé** : RGD (Racine Gauche Droite)
- **Postfixé** : GDR (Gauche Droite Racine)
- **Infixé** : GRD (Gauche Racine Droite)
- **En largeur** : parcours par niveau, de gauche à droite

Exemple : soit une expression : $a + (b-1) * 5$

- Préfixé : $+ a * - b 1 5$
- Postfixé : $a b 1 - 5 * +$
- Infixé : $a + b - 1 * 5$
- En largeur : $+ a * - 5 b 1$

• *Algorithme de parcours récursif d'un arbre binaire*

Parcours préfixé	Parcours postfixé	Parcours infixé
<pre>void prefixe(pointeur raci){ if(raci != NULL) { traiter(raci→info); prefixe(raci→gauche); prefixe(raci→droite); } }</pre>	<pre>void postfixe(pointeur raci){ if(raci != NULL) { postfixe(raci→gauche); postfixe(raci→droite); traiter(raci→info); } }</pre>	<pre>void infixe(pointeur raci){ if(raci != NULL) { infixe (raci→gauche); traiter(raci→info); infixe (raci→droite); } }</pre>

- *Autre algorithme sur les arbres binaires*

Création d'un arbre binaire
<pre> pointeur creerArbre(pointeur raci) { pointeur p ; char filsg[20], filsd[20] ; printf(«Saisir fils gauche ») ; scanf(« %s », filsg) ; if(strcmp(filsg, « null ») != 0) { p = (pointeur) malloc(sizeof(struct noeud)); strcpy(p→info, filsg); raci→gauche = p; return creerArbre(raci→gauche); } else { raci→gauche = NULL. } printf(«Saisir fils droite ») ; scanf(« %s », filsd) ; if(strcmp(filsd, « null ») != 0) { p = (pointeur) malloc(sizeof(struct noeud)); strcpy(p→info, filsd); raci→droite = p; return creerArbre(raci→droite); } else { raci→droite = NULL. } } </pre>
Calcul taille d'un arbre
<pre> int taille(pointeur raci) { if(raci == NULL) return 0 ; else return (1 + taille(raci→gauche) + taille(raci→droite)) ; } </pre>

Calcul nombre de feuille

```
booléen feuille (pointeur nœud){  
    return ((nœud→gauche == NULL) && (noeud→droite == NULL)) ;  
}  
  
int nbFeuille(pointeur raci) {  
    if(raci == NULL) return 0;  
    else {  
        if(feuille(raci)) return 1;  
        else return (nbFeuille(raci→gauche) + nbFeuille(raci→droite)) ;  
    }  
}
```