

Liste

* Déclaration en C++

#include <stdlib.h>

```
typedef struct cellule  
{  
    type valeur p  
    cellule * suiv p  
}
```

};

// Le type liste est défini.

typedef cellule * llist

// Maintenant on peut déclarer une variable de type pointeur sur Notre liste (cellule)

↓ cellule

on a déclaré un pointeur llist qui pointe sur une cellule juste pour simplifier la déclaration.

* effectue la valeur à l'espace *p

p.valeur ← ... / p.suiv ← ...

* Déclaration en Algorithmique

type pointeur = liste = pointeur

liste = structure

valeur : type

suivant : pointeur

fin structure

var p : pointeur.

• Déclare varp "pointeur"

* Initialisation d'une *

Liste à NULL

pour initialiser une liste il faut d'abord que la liste soit vide. (NULL)

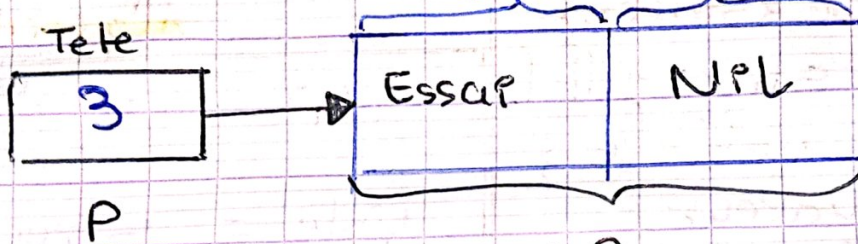
• Allouer(p) :

Allouer un espace dans la mémoire de size p et lui donne l'adresse de l'espace mémoire

*p

Si $p = Nil$ alors p ne pointe sur rien

pour définir ce exemple la:



En Algorithmique

Algorithme e Liste-essai.

Debut

Type pointeur = Liste & pointeur

Liste = structure

valeur: chaîne

suivant: pointeur

Fin structure

Var p = pointeur;

Alloquer(p);

$p.valeur \leftarrow \text{"Essai"};$

$p.suivant \leftarrow Nil;$

Fin

initialisation à 0

En C++ :

```
#include <iostream>
#include <stdlib.h>
using namespace std;
```

```
typedef struct element element;
struct element {
    char valeur;
    struct element *suiv;
};
```

```
typedef element * Liste;
```

```
int main (int argc, char **argv)
```

```
int main (int argc, char * argv)
```

```
Liste * maListe = NIL
```

```
/* tjr initialiser la Liste à NIL */
```

```
valeur = (element*) malloc (sizeof (element))
```

```
cin >> p -> valeur;
```

```
p -> valeur = "Essai";
```

```
p -> suiv = NIL;
```

```
systeme ("pause");
```

```
return 0;
```

```
}
```


* Création Une Liste en utilisant une procédure

En Algog

procédure Créer-Liste(tele:ptr)

var v : entier

□ Debut

Alloquer (Tele)

Lire (v)

val (Tele) \leftarrow v

suiv (Tele) \leftarrow Nil

□ Fin

En C++

~~void Créer-Liste(Liste* Tele)~~

~~int v~~

~~(Liste*) malloc(sizeof~~

~~Tele(Liste))~~

En C++

void Créer-Liste(Liste* tele)
{
int v;
Tele(Liste*) malloc(sizeof
(Liste));

cin >> v;

Tele->val = v;

Tele->suiv = Nil;

}

Une réalisation

liste . val == NIL

liste . suiv == NIL

Insertion dans une

Liste vide :

- Allocation de la mémoire pour le nouvel élément
- Remplir le champ de donnée du nouvel élément
- Le pointeur suivant
← NIL
- taille = taille + 1

Insertion au Début de la Liste

Affichage d'une liste en Algorithmique

procedure =

- définir un pointeur pour l'incrémenter.

- Tant que le pointeur $\neq \text{NULL}$
 $p = p \rightarrow \text{suivant}$

en C++:

```
void Affichage(  
  element * Tete) {  
  element * T;  
  T = Tete  
  (while T != NULL)  
  { cout << T->val.  
    endl;  
    T = T->suiv;  
  }  
}
```

procedure Affichage

(Tete: element)
var T: element

Tant que (T \neq NULL)
{ Ecrire (T.val)

T \leftarrow T.suiv
fin

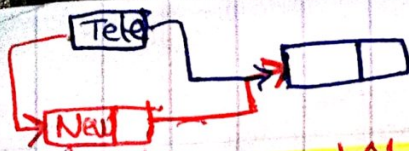
Nombre d'elements de la Liste

En C++

```
int Nombre_delem  
(element *T) {  
    element *T  
    int n;  
    while (T != NULL)  
    {  
        n = n + 1  
        T = T -> suiv;  
    }  
    return n;  
}
```

En Algorithmique

```
fonction Nomb_elem  
| T: element  
Debut  
var e: element  
n: entier  
TQ ( T ≠ Null )  
{  
    n ← n + 1  
    T ← T.suiv;  
    Nomb_elem ← n;  
Fin.
```

Insérer en tête

C++

```

element * Insert-tete
(element * Tele, string val)
{
    element * t;
    t = New(element);
    t->valeur = val;
    t->suiv = Tele;
    Tele = t;
    return t;
}

```

Algorithmes

Fonction Insert-tete
(Tele: element,
string val)

~~Debut~~
 Var t: element
~~Debut~~
 Affecter(t);
 t.valeur = val;
 t.suiv = Tele;
 Tele = t;
 Insert-tete ← t;
 Fin

Ajouter à une position
Donnée

C++

```

element * Ajout-pos
(element * Tele, string
valeur, int pos)
{

```

```

    element * p, * q;
    int i = 1;
    p = Tele;
    q = New(element);
    q->valeur = valeur;
q->suiv =

```

```

    if (pos == 1) {
        q->suiv = Tele;
        Tele = q;
    }

```

```

    else {
        while ((p->suiv != NULL) &&
        (p->i != pos - 1)) {
            p = p->suiv;
            i++;
        }
    }

```



```

q → suiv = p → suiv
p → suiv = q
return tele
}
}

```

Algorithme 8

Fonction insert_pos
 (tele : element, val :
 chaine, pos : entier)
 Var p, q : *element,
 i : entier

Debut :

```

i ← 1;
p ← tele;
q ← Allover(q);
q.valeur ← val;
Si (pos = 1) Alors
  q.suiv ← tele;
  tele = q;

```

Simon
 TQ ((p.suiv ≠ NULL)
 et (i ≠ pos - 1))
 faire :

```

p ← p.suiv;
i ← i + 1;

```

Fin Tant que.

```

q.suiv = p.suiv;
p.suiv = q;

```

Finsi

insert_pos ← tele

Fm

Insertion en queue

C++ =

element * insert-queue (element * tete, char v)

{ element * p, * q ;

p = New(element) ;

p->valeur = v ;

q = tete ;

while(q->suiv != NULL) {

q = q->suiv ; }

q->suiv = p ;

p->suiv = NULL ;

return tete ; }

Algorithmes

Fonction insert-que (tete : element, v : char)

Var : q, p : element ;

Debut :

Alouer (p)

p.valeur ← v ;

q ← tete ;

TQ (q.suiv ≠ NULL) Faire

q ← q.suiv ; Fin TQ

q.suiv ← p ;

p.suiv ← NULL ;

insert-que ← tete ;

Fin

Suppression A Une position donnée

En C++

```
element* supr - pos(element* tele, int pos)
{
    element * p, * q;
    int i;
    p = tele;
    if (pos == 1) {
        tele = tele -> suiv;
        delete (p);
    }
    else {
        q = tele;
        q = q -> suiv;
        i = 1;
        while (q -> suiv != Null) {
            if (i == pos - 1)
            {
                q = q -> suiv;
                t = t -> suiv;
                i++;
            }
            t -> suiv = q -> suiv;
            delete(q);
        }
        return tele;
    }
}
```


En Algortime

Fonction $\text{supr-pos}(\text{tele} = * \text{Element}; \text{pos} \in \text{Entier})$

Var $p, q = * \text{Element};$

$i = \text{entier};$

Debut $p \leftarrow \text{tele};$

si $(\text{pos} = 1)$ alors

$\text{tele} \leftarrow \text{tele} \cdot \text{suiv};$

des Allouer $(p);$

sinon

$q \leftarrow \text{tele};$

$q \leftarrow q \cdot \text{suiv};$

$i = 1;$

TQ $((q \cdot \text{suiv} \neq \text{Null}) \text{ et } (i \neq \text{pos} - 1))$ faire

$q \leftarrow q \cdot \text{suiv};$

$t \leftarrow t \cdot \text{suiv};$

$i = i + 1;$

Fin TQ;

$t \cdot \text{suiv} \leftarrow q \cdot \text{suiv};$

des Allouer (q)

$\text{supr-pos} \leftarrow \text{tele};$

Fin

Listes Doublement chaînées

Insertion en tête : C++ :

```
Element * insert_tete (element * Tete, char v)
{
    element * p;
    p = New(element);
    p -> val = v;
    p -> prec = Null;
    p -> suiv = Tete;
    Tete -> prec = p;
    Tete = p;
    return p;
}
```

Insertion en queue : C++ :

```
Element * inser_que (element * tete, char v)
{
    element * p, * q;
    q = tete;
    while (q -> suiv != Null) {
        q = q -> suiv;
    }
    p = New(element);
    p -> val = v;
    p -> prec = q;
    p -> suiv = Null;
    q -> suiv = p;
    return tete;
}
```


Insertion à une position donnée

```
Element * insert_pos (element * tete, char v, int pos) {  
    Element * p, * q, * store; int i;
```

```
    q = tete; i = 1;
```

```
    while ((q != NULL) && (i != pos - 1)) {
```

```
        q = q -> suiv;
```

```
        i++;
```

```
    p = New (element);
```

```
    p -> val = v;
```

```
    if (pos == 1) { p -> prec = NULL;
```

```
        p -> suiv = tete;
```

```
        tete -> prec = p;
```

```
        tete = p; }
```

```
    else { q -> suiv = p;
```

```
        p -> prec = q;
```

```
        p -> suiv = store;
```

```
        store -> prec = p; }
```

```
    return tete;
```

```
}
```


Suppression A une pos donner

Element *Supp_pos (Element *tele, int pos) {

Element *p, *q, *t;

int i = 1;

t = tele;

if (pos == 1) { tele = t->suiv;
delete(t); }

else { q = tele;

q = q->suiv;

while ((q != NULL) && (i != pos - 1)) {

q = q->suiv;

t = t->suiv;

i++;

t->suiv = q->suiv;

delete(q); }

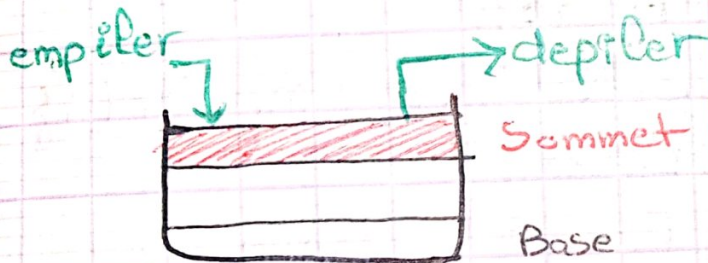
return tele;

}

Les piles & Les Files

1. Les piles

On peut ajouter ou supprimer juste de **Sommet**.



LIFO (Last In First Out)

1. declaration:

Algoz

```
pile = structure {
    int val;
    pile * suiv;
}
```

}

C++:

```
struct pile {
    int val;
    pile * suiv;
}
```

```
pile * sommet, * p;
char valeur;
```

2. Tester si vide

Algoz

```
procedure vide() {
```

```
Debut
```

```
Si (Sommet  $\neq$  Null) Alors
    Ecrire("N'est pas vide");
```

```
Sinon
```

```
    Ecrire("vide");
```

```
Finsi
```

```
Fin
```

C++ :

```
void vide() {
    if (Sommet  $\neq$  Null) {
        cout << "N'est pas vide" << endl;
    }
    else {
        cout << "Vide" << endl;
    }
}
```


3. Initialisation de la pile

C++

```
void initialiser() {  
    Sommet = New(pile);  
    Sommet = Null;  
    cout << "done !" << endl;  
}
```

Algo

```
procedure initi_p() "  
debut  
    Sommet ←  
    Allouer (Sommet);  
    Sommet ← Null;  
    Ecrire (" Done !");  
fin.
```

4. empiler

C++

```
void empiler() {  
    p = New(pile);  
    cout << "donne une valeur";  
    cin >> valeur;  
    p->valeur
```

C++

```
void empiler() {  
    cout << "donne une  
valeur";  
    cin >> valeur;  
    p->val = valeur;  
    p->suiv = Sommet;  
    Sommet = p;  
}
```

Algo

```
procedure empiler() "  
Debut  
    Ecrire ("donne une  
valeur");  
    Lire (valeur);  
    p.val ← valeur;  
    p.suiv ← Sommet;  
    Sommet ← p;  
Fin.
```


5. depiler

C++ :

```
void depiler() {  
    if (Sommet != NULL) {  
        p = Sommet;  
        Sommet = Sommet->suiv;  
        delete(p);  
    }  
}
```

Algo :

```
procedure depiler()  
debut  
si (Sommet ≠ NULL) Alors  
    p ← Sommet;  
    Sommet ← Sommet->suiv;  
    dealloquer(p);  
fin si  
fin
```

6. Affichage

C++ :

```
void Afficher() {  
    if (Sommet != NULL) {  
        p = Sommet;
```

```
while (p != NULL) {  
    cout << p->val;  
    p = p->suiv;  
}
```

Algo :

```
procedure Affichage()  
Debut  
si (Sommet = NULL) Alors  
    p ← Sommet;  
    Tant que (p ≠ NULL) faire  
        Ecrire(p->val);  
        p ← p->suiv;  
    Fin Tant que  
Fin si  
Fin
```

7. Hauteur-pile

C++ :

```
void Hauteur-pile() {  
    int h = 0;  
    p = Sommet;  
    while (p != NULL) {  
        h = h + 1;  
        p = p->suiv;  
    }  
    cout << "Hauteur est" << h;  
}
```


Algoes

```
procedure Hauteur-p() §  
var R : entier  
Debut  
R ← 0;  
p ← sommet;  
TQ (p ≠ Null) faire  
  R ← R + 1;  
  p ← p.suiv;  
FinTQ  
Ecrire("La Hauteur", R);  
Fin.
```

8. Remplacer_sommet =

C++

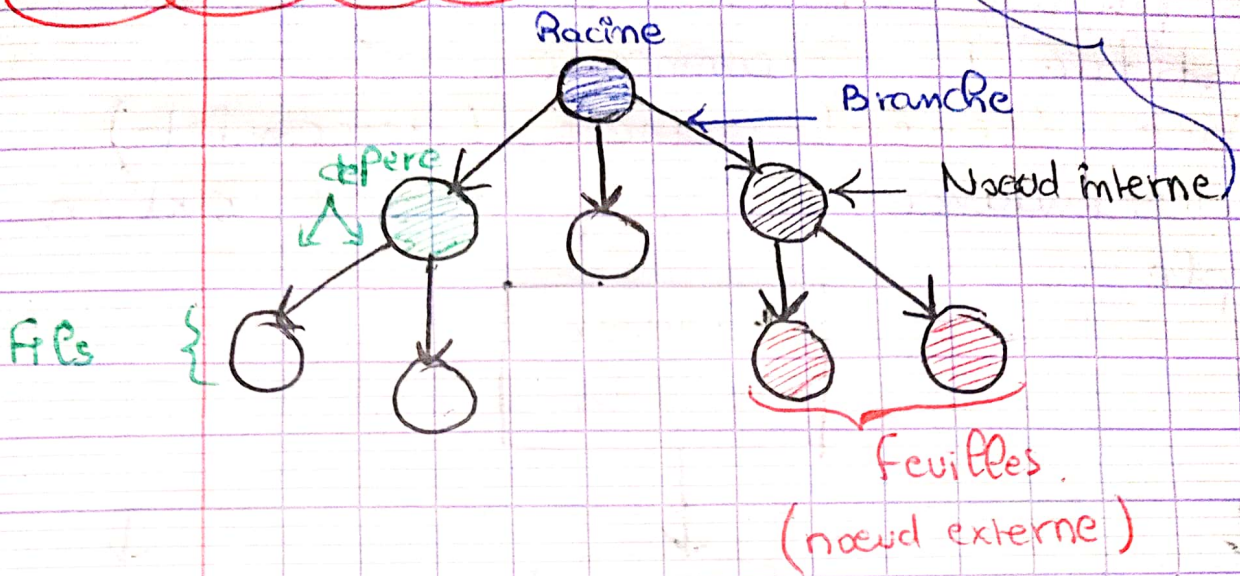
void


```
procedure Remplacer() {  
  int n;  
  if (Sommet != Null) {  
    cout << "donne la val de  
    Nouvelle_sommet";  
    cin >> n;  
    Sommet -> val = n; }  
  else {  
    cout << "La pile est vide";
```

Algoes

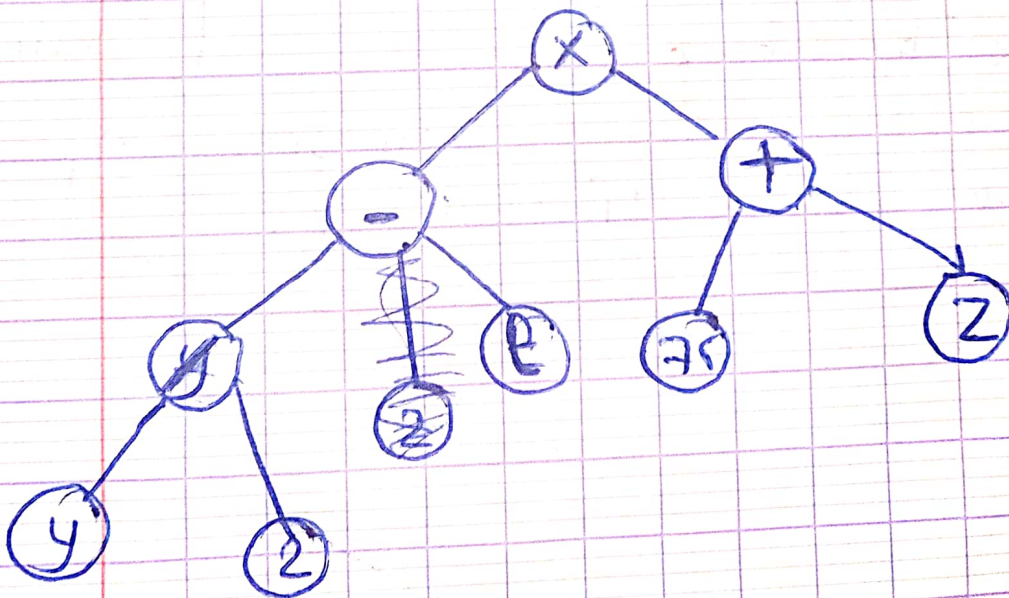
```
procedure Remplacer() §  
var n : entier  
Debut  
Si (Sommet ≠ Null) Alors  
  Ecrire("Donne la  
  Nouvelle_sommet");  
  Lire(n);  
  Sommet.val ← n;  
Sinon  
  Ecrire("La pile est  
  vide");  
Fin.
```


Les Arbres



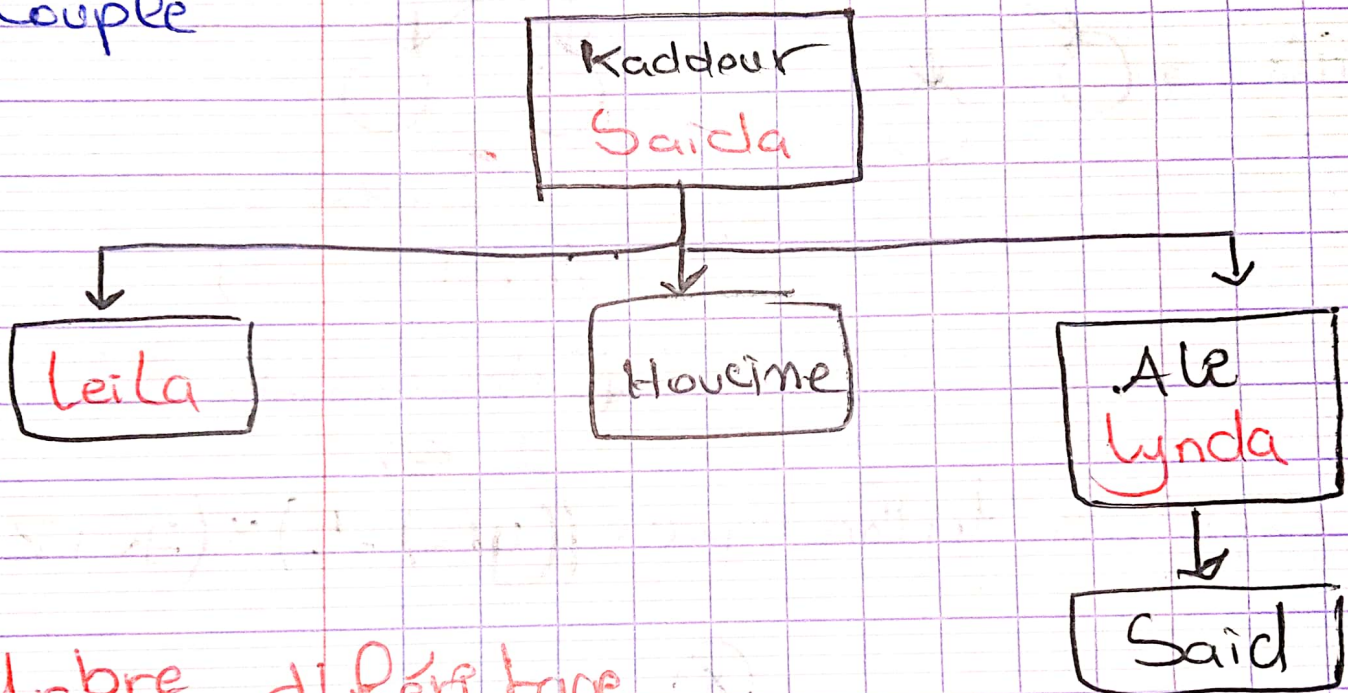
* descendants de 

Expression arithmétique $((y/2) - t) \times (x + z)$



Arbre Syntaxique Représente l'Analyse de phrase à partir de règles qui Constitue la Grammaire.

Arbre Généalogique (descendit)
la descendance d'une personne ou d'un couple



Arbre d'héritage

Arbre Lexicologique: Arbre en parh's
Commune, ou dictionnaire.

taille d'un arbre =
des Nœuds.

Le nombre totale

Arbre dégénéré



La Hauteur = Les Nb de Nœuds
de Haut en Bas. (par Niveau)



Chemin d'un Nœud La suite des
Nœuds Dans laquelle il faut passer
pour Aller d'une Racine à X,
 $Hauteur(x) = Nb\ Nœud\ (chemin(x))$

On a

Calcul Recursif

La Hauteur de

Noeud 9 :

$$R(9) = R(8) + 1$$

$$R(8) = R(5) + 1$$

$$R(5) = 1 + R(1)$$

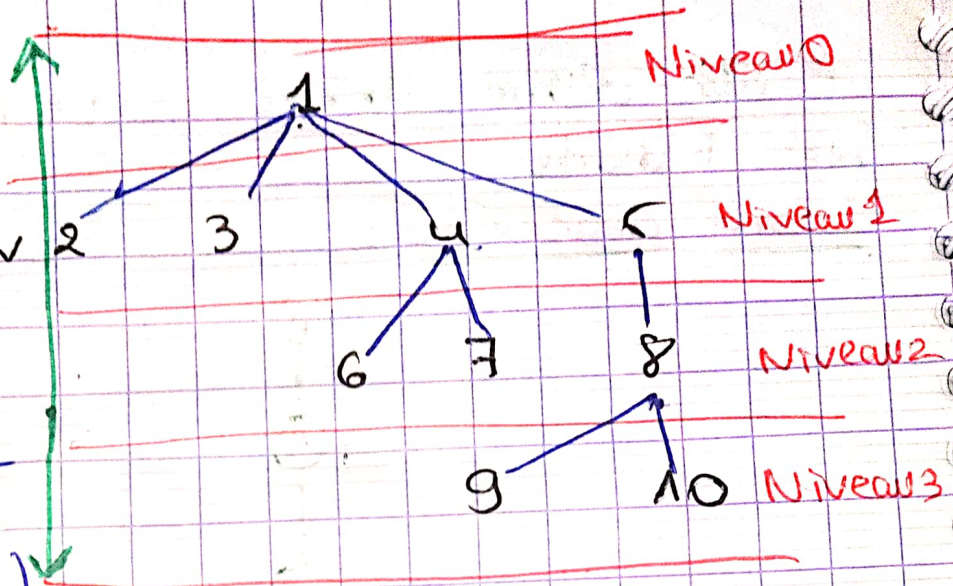
$$R(1) = 1$$

Hauteur d'Arbre = 4

$$\Rightarrow R(5) = 2$$

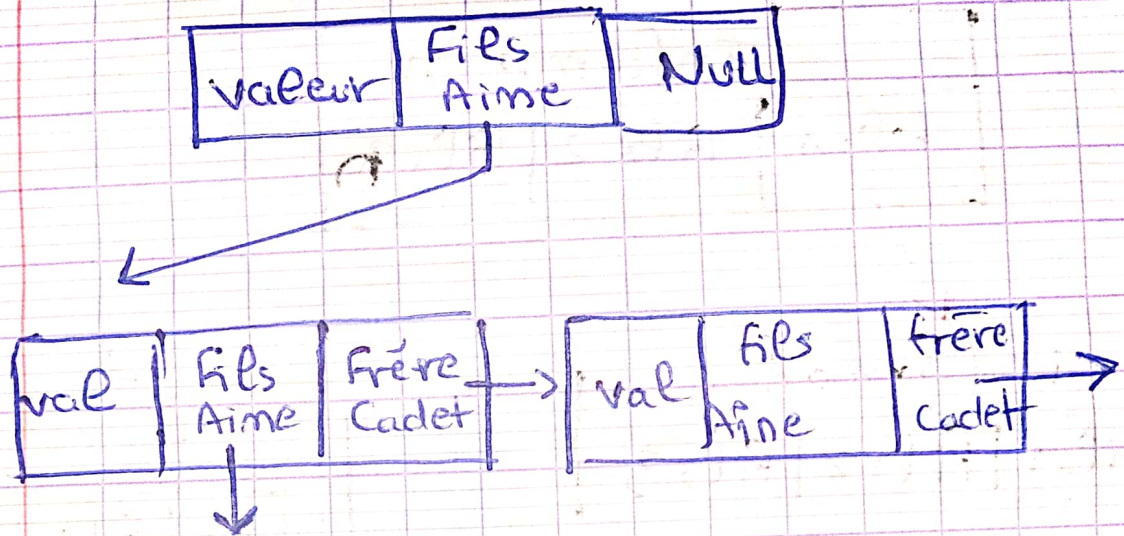
$$\Rightarrow R(8) = 3$$

$$\Rightarrow R(9) = 4$$



Implementation - Arbre

Une Liste:



Declaration d'Une Arbre

Algoe

Type Noeud = Structure

Debut

val : Element

Noeud Fils Aime : *Noeud

Frere Cadet : *Noeud

Fin

Arbre ~~Arbre~~ *Noeud;

C++

typedef struct Noeud {

Element val;

Noeud * Fils Aime;

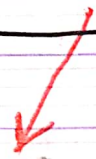
Noeud * Frere Cadet;

} * Arbre;

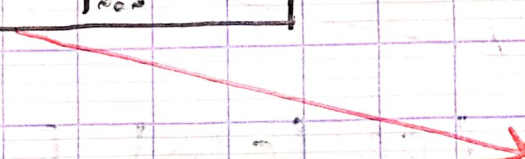
La Représentation :

Statique par un Tableau -
Dynamique Exemple (Liste)

Information		
Fils 1	Fils 2	Fils 3



information		
Fils 1	Fils 2	Fils 3



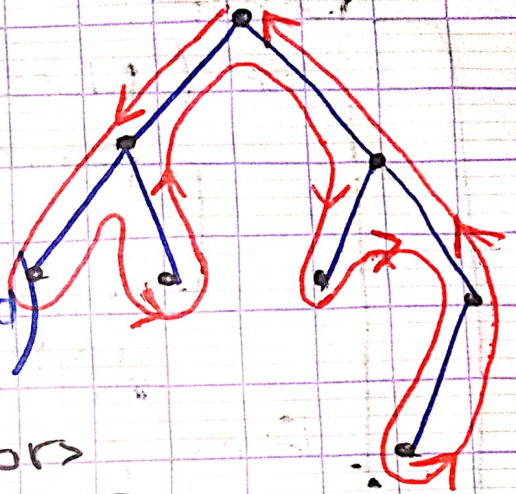
information		
Fils 1	Fils 2	Fils 3

Déclarations

```
Type Noeud = Structure {  
    info : Type;  
    fils : Tab [I] (NbdeFils) de pointeur  
            (TNoeud)  
};  
var Racine : pointeur (TNoeud);
```


parcours des Arbres

parcours en profondeur
Algorithme Recursive



procedure PP (TNoeud : *Noeud)

Debut

Si (Noeud \neq NULL) Alors

pour i de 1 à NbFils Faire

PP (Fils i (Noeud))

Fimpour

Finsi

Fim

profondeur prefixe (père avant fils)

procedure ppp (TNoeud : *Noeud)

Debut

Si (Noeud \neq NIL) faire

Apprche (valeur (Noeud))

pour i de 1 à NbFils faire

PPP (Fils i (Noeud))

Fimpour

Finsi

Fim

profondeur postfixe (Fils avant pere)

Procédure ppp2 (T Noeud, * Noeud)

Debut

Se (Noeud \neq Nil) faire
pour i de 1 à Nb fils faire
ppp2 (Fils i (Noeud))

Fin pour

Afficher (valeur (Noeud))

Fin se

Fin

La Taille de L'arbres

Fonction Taille (Noeud : pointeur (TNoeud))

enfer

Var : i, s : entier

Debut

Si (Noeud = Nil) Alors

Taille $\leftarrow 0$

Sinon

$s \leftarrow 1$

pour i de 1 à NbFils faire

$s \leftarrow s + \text{Taille}(\text{Fils}_i(\text{Noeud}))$;

Fin pour

Taille $\leftarrow s$;

Fin Si

Fin.

Arbre n-aire : d'ordre n , le degré Max d'un Noeud est n .

B-Arbres B-Arbre d'ordre n :

- Racine a au moins 2 fils.
- Chaque Noeud A entre $n/2$ et n fils.
- Tous les Noeuds feuilles sont au même Niveau.

Arbre Binaire Degré Max de Noeud 2.

Arbre Binaire de Recherche

$FG \leq R \leq FD$.

propriété d'Arbres

Racine, Taille (Nb de Noeud).

Hauteur (=) = 3, profondeur

Noeud & inférieur : لي حبيب لاخل

Feuille ($FG=0$ et $FD=0$)

parent(x) = Racine de \mathcal{N} .

freres(x) = Le Meme Niveau de \mathcal{N} .

Ancêtres(x) = $x \rightarrow \dots \rightarrow i$

descendants(x) = $x \rightarrow \dots \rightarrow i$

* La Racine d'Arbres

on ne trouve pas sont indice dans les
Fils Gauches ou droites.

Declaration de L'Arbre & En Algo

Type def ~~sub~~ Noeud = Structure

Debut

info : entier.

G, D : entier.

Fin.

T Noeud : Noeud ;

• Dans une Arbre binaire
parcours profondeur

procedure parcour pra (A: arbre, T: chaîne)

Debut

Si (A \neq NIL) Alors

Si (T = "préfixé")

Ecrire (A.info)

parcour pra (A.gauche);

Si (T = "infixé")

Ecrire (A.info)

parcour pra (A.droite);

Si (T = "postfixé")

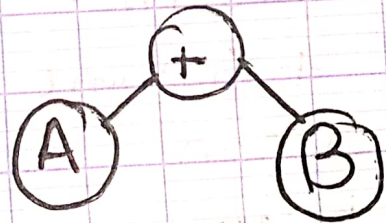
Ecrire (A.info).

Fin Si

inf = A + B

pre = + AB

post = AB +



Fonction Calcule Le Nombre de Noeud

Fonction Nb-Noeud (A : *Arbre) : entier

Debut

Si (A \neq NIL) Alors

Retourner $1 + \text{Nb-Noeud}(A.FG)$
 $+ \text{Nb-Noeud}(A.FD)$

Sinon

retourner 0

Fin Si

Fin

Fonction Calcule la Somme des Nbs

Fonction Somme (A : *Arbre) : entier

Debut

Si (A \neq NIL) Alors

Retourner $A.inf + \text{Somme}(A.FG)$
 $+ \text{Somme}(A.FD)$

Sinon

Retourner 0

Fin Si

Fin

Fonction Nbre de Feuilles

Fonction Nb_F (A : * Arbre) : entier

Debut

Si (A = Nil) Alors

retourner

Si ((A.FG = Nil) et (A.FD = Nil))
retourner 1;

Sinon

retourner Nb_F(A.FG) + Nb_F(A.FD)

finSi

Fonction Calcule La Hauteur

Fonction Haut(A : * Arbre) : entier

Debut

Si (A ≠ Nil)

Hd ← Haut(A.FD)

HG ← Haut(A.FG)

retourner 1 + max(Hd, HG);

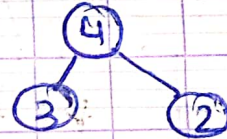
Sinon

retourner 0;

AB Complet

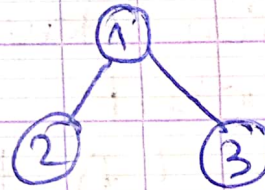
Tasse

tout Noeud a une valeur plus grande ou egale a celle de ces 2 fils.
plus grand element c'est la Racine



Tas Min

tout Noeud a une valeur plus petit. ou egale a celles de ces 2 fils.
plus petit element c'est la Racine.



Exemples de Cours

Chapitre N°1 & Les Listes chaînées

• Creation d'une Liste

procedure Creer (tete: ptr)

var v : entier

debut

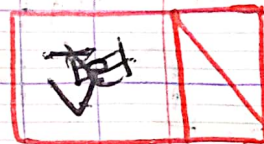
Allover (tete);

Lire (v)

val (tete) \leftarrow v

Suivant (tete) = Nil.

Fin



↑
tete.

• Creation d'une Liste a partir d'un vect

procedure Transférer V-L (elem & tete, V[n], n)

var i : entier

p : pointeur

Debut

p \leftarrow Nil

pour i = 1 à n faire

Allover (p)


```
val(p) ← v[i]
suivant(p) ← Tete
Tete ← p
Fimpour
Fin
```

Affichage d'une liste

```
procedure Affichage (Tete: *element)
var p: pointeur element
Debut
    p ← Tete;
    tant que (p ≠ NIL) faire
        Ecrire (val(p))
        p ← suivant(p)
    Fin TantQue
Fin
```


Ajouter en tête

procedure ~~Aj~~ Ajouter-T (Tete: ptr, v: entier)

var p: ptr

Debut

~~p ← Tete~~

Allocuer(p)

val(p) ← v

Suiv(p) ← Tete

Tete ← p

Fin

Ajouter en fin

procedure Aj-F (Tete: ptr, v: entier)

var p: ptr; q: ptr

Debut

p ← Tete

TQ (~~p~~ Suiv(p) ≠ NIL) faire

~~p~~ ← ~~p~~ Suiv(p)

Fin TQ

Lire (v)

Allocuer(q)

val(q) ← v

Suiv(q) ← NIL

Fin. $Suiv(p) \leftarrow q$

procedure Suppression en fin

procedure sup-f (Tete, ptr)

var p, p_1 : ptr.

Debut

$p \leftarrow Tete$

$p_1 \leftarrow Suiv(Tete)$

TQ ($p \neq Nil$) faire

$p_1 \leftarrow Suiv(p)$

$p \leftarrow Suiv(p)$

Fin TQ

$Suiv(p_1) \leftarrow Nil$

Libérer(p)

Fin

Appiche une Liste en Utilisant une Boucle

10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10

```
int Main() {  
    element * p = Null;  
    int i;  
    for (i = 1; i <= 10; i++) {  
        p = Ajout-T(p, i);  
        p = Ajout-f(p, i);  
    }  
    Affichage(p);  
    return 0;  
}
```


Supprimer un élément de valeur donnée d'une liste

Fonction Sup-Elém (élément à supprimer, valeur)

Var p, q : ptr

Debut

p ← Tete

q ← Tete

TQ (p ≠ Nil) Faire

~~if (val(p) = v) Faire~~

Si (val(Tete) = v) Alors

Tete ← Supr-T(Tete)

Fin Si sinon Si (val(p) = v) Alors

q ← Suiv(p)

libérer(p)

Si non

Ecrire ("La Valeur N'existe pas")

Fin Si

p ← Suiv(p)

retourner(Tete)

Fin

Chapitre N°2 Listes Doublement chaînées

Ajouter avant p

procedure Aj-Av-p(Tele:ph, v:ent)

Debut

$p_1 \leftarrow \text{Tele}$

TQ ($p_1 \neq \text{NIL}$) et ($p_1 \neq p$) faire

$p_1 \leftarrow \text{Suiv}(p_1)$

FinTQ

Si ($p_1 = p$) Alors

Allover (p_2)

$\text{val}(p_2) \leftarrow v$

$\text{Suiv}(p_2) \leftarrow p$

$\text{prec}(p_2) \leftarrow \text{prec}(p)$

$\text{Suiv}(\text{prec}(p)) \leftarrow p_2$

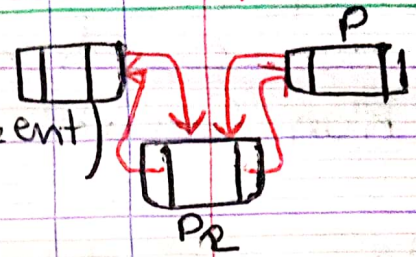
$\text{prec}(p) \leftarrow p_2$

Sinon

Ecrire ("p N'existe pas")

FinSi

Fin



Chapitre NEU

Les Arbres

Les Arbres Binaires

Fonction Creation d'un Nouveau Noeud

Fonction CreerNoeud (v : entier, fg : Arbre, fd : Arbre

{

Arbre p ;

$p = \text{New}(\text{Arbre})$

$p \rightarrow \text{val} = v$;

$p \rightarrow fg = fg$;

$p \rightarrow fd = fd$;

return p ; }

/ Dans le main */*

$a = \text{CreerNoeud}(n, p, \text{NULL})$

Hauteur d'Une Arbre de Recherche

Hauteur (Noeud: *A) : entier

Debut

Si (A \neq NULL) Alors
retourner 0

Sinon

Rd, Rg : entier

Rg = Hauteur (Fils Gauche(A));

Rd = Hauteur (Fils droite(A));

retourner 1 + Max(Rd, Rg);

Fin

Concatenation de deux Listes

Fonction Concat (L1, L2 : Liste)

Var p1, p2 : L1, L2;

Debut

Si (L1 \neq NIL) Alors

Si (L2 = NIL) Alors

L1 \leftarrow L2;

Sinon

p1 \leftarrow Tete(L1)

TQ p1 \neq NULL faire

p1 \leftarrow p1.suiv

FTQ

suiv(p1) = tete(L2)

FinSi

FinSi

Fin.

Ajouter un Nœud dans une ABRe

procedure AjouterABR (ABR \neq Arbre, ent v)

Debut

Si (ABR \neq Nil) faire

~~Ajouter~~ a = Arbre(v, Nil, Nil)

Sinon

Si $x \leq \text{valeur(ABR)}$

• Ajouter(v, FG(ABR))

Sinon :

• Ajouter(v, FD(ABR))

Finsi

Finsi

Fm

Rechercher Une clé // itérative //

bool Fonction: Rechercher (A = ptr (Arbre), int V)

Debut

TOP \neg est vide(A) et $V \neq \text{Val}(A)$ faire
 Si $V < \text{val}(a)$ Alors
 a = fils Gauche(A)
 Sinon
 a = fils droite(A)
 fin Si
fin TOP.

Si Est vide (A) Alors
 retourner Faux
Sinon
 retourner vrai.
fin Si

Fin

Rechercher Une cle dans ABQ / Recursive //

Fonction Rechercher (A: *Arbre, int v) : bool

Debut

Si EstVide(A) Alors

Retourner Faux.

Sinon si (v == val(A)) Alors

Retourner Vrai

Sinon si v < val(A) Alors

retourner recherche(FG(A), v)

Sinon

retourner recherche(FD(A), v)

Finsi

Finsi

Finsi

Fin

recursive \Rightarrow ékrakve.

X ~~éléme~~ procédure affichage préfixé (ArAr

pile p i

empiler(p, a)

TQ $p \neq \text{Nil}$ faire

$S \leftarrow \text{Sommet}(p)$

depiler(p)

Si $S \neq \text{Nil}$ Alors

Afficher(S)

empiler(p, fd(s))

empiler(p, dg(s))

fin Si

Fin TQ

Fin