Retour sur le passage de paramètres par variable : tableaux dimension 2

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

int *init1(int);
void init2(int **,int);
void rempli(int *,int);

int main(){
 int *ta,*tb,i;
 ta=init1(7);
 init2(&tb,7);
 rempli(tb,7);
 return EXIT_SUCCESS;}

int *init1(int n){
return (int *)malloc(n*sizeof(int));}

void rempli(int *tt,int n){
 int i;
 for(i=0;i<n;i++) tt[i]=2*(n-i);}

void init2(int **t,int n){
*t= (int *) malloc (n*sizeof(int));}
```


Retour sur le passage de paramètres par variable

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

int *init1(int);
void init2(int **,int);
void rempli(int *,int);

int main(){
 int *ta,*tb,i;
 ta=init1(7);
 init2(&tb,7);
 rempli(tb,7);
 return EXIT_SUCCESS;}

int *init1(int n){
return (int *)malloc(n*sizeof(int));}
```

...

Tableau simple
Dimension 2

```
int **init1(int);
void init2(int ***,int);
void rempli(int **,int);
void affiche(int **,int);

int main(){
 int **tb,nblg;
 printf("Triangle de Pascal : Nombre de
          lignes = ");
 scanf("%d",&nblg);
//  tb=init1(nblg);
 init2(&tb,nblg);
 rempli(tb,nblg);
 affiche(tb,nblg);
 return EXIT_SUCCESS;}

int **init1(int n){
 int **t,i;
 t= (int **)malloc(n*sizeof(int*));
 for(i=0;i<n;i++)
   t[i]=(int *)malloc((i+1)*sizeof(int));
 return t;}
```


Retour sur le passage de paramètres par variable

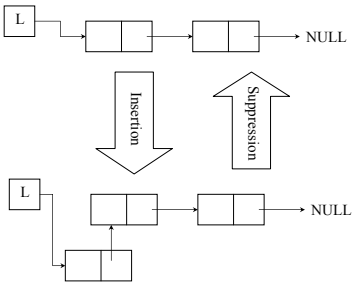
```
int *init1(int n){
return (int *)malloc(n*sizeof(int));}

void init2(int **t,int n){
*t= (int *) malloc (n*sizeof(int));}

void rempli(int *tt,int n){
 int i;
 for(i=0;i<n;i++) tt[i]=2*(n-i);}
```

Tableau simple
Dimension 2

```
int **init1(int n){
 int **t,i;
 t= (int **)malloc(n*sizeof(int*));
 for(i=0;i<n;i++)
   t[i]=(int *)malloc((i+1)*sizeof(int));
 return t;}

void init2(int ***t,int n){
 int i;
*t= (int **)malloc(n*sizeof(int*));
 for(i=0;i<n;i++)
   (*t)[i]=(int *)malloc((i+1)*sizeof(int));}

void rempli(int **tt,int n){
 int i,j;
 for(i=0;i<n;i++) {
  tt[i][0]=1;
  for(j=1;j<i;j++)
    tt[i][j]=tt[i-1][j]+tt[i-1][j-1];
  tt[i][i]=1;  } }

void affiche(int **tt,int n){
 int i,j;
 for(i=0;i<n;i++) {
  for(j=0;j<=i;j++)
    printf("%5d",tt[i][j]);
  printf("\n"); } }
```


Liste vide

Liste non vide

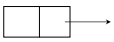


Insertion et suppression se font toujours en tête de liste

Algorithmes

- Insertion (en tête)
  - allouer une nouvelle cellule
  - Son suivant prend la valeur du pointeur de liste L
  - L pointe sur cette nouvelle cellule
- Suppression (en tête)
  - Mémoriser L (pointeur)
  - L pointe sur la valeur du pointeur de suivant du L
  - Détruire la cellule pointée par la mémorisation
- Parcours
  - copier L (pointeur) dans P (L fixe P change)
  - Boucler tant que P≠NULL
    - Afficher valeur pointée par P
    - P←suivant(P)

Structure de données



- Cellule
- Mauvaises idées

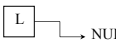
```
typedef struct {
  int valeur;
  cellule *suivant;
} cellule;
```

ou

```
typedef struct {
  int valeur;
  struct cellule *suivant;
} cellule;
```

- Bonne idée

```
struct cellule{
  int valeur;
  struct cellule *suivant;
};
```



- Éventuellement

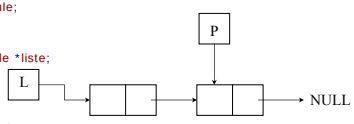
```
typedef cellule cellule;
```

- Liste

```
typedef struct cellule *liste;
```

- Ne pas oublier :

```
liste L=NULL, P;
```

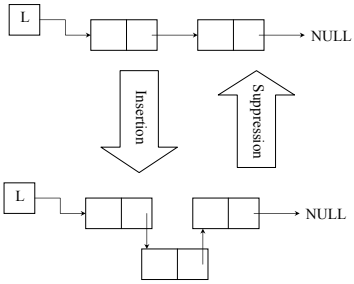


  - Initialiser la liste
  - Prévoir un pointeur pour se déplacer dans la liste

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

struct cellule{
  int valeur;
  struct cellule *suivant;
};

typedef struct cellule *liste;

liste insere(int element, liste Q){
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = Q;
  return(L);
}

void inserer(int element, liste *Q){
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = *Q;
  *Q=L;
  return;
}

typedef struct cellule cellule,*liste;

int main(){
  liste L=NULL, P;
  inserer(4,&L);
  inserer(7,&L);
  inserer(3,&L);
  inserer(8,&L);

/* L = insere(1,insere(2,insere(3,insere(4,NULL))));*/
printf("\n Affichage de la liste:\n");
 P = L;
 while (P != NULL) {
    printf("%d \t",P->valeur);
    P = P->suivant;
 }
 return 0;}
```

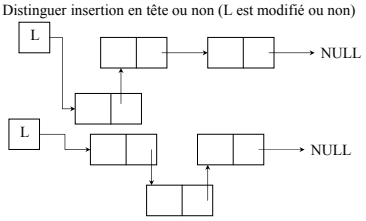

Liste vide

Liste non vide

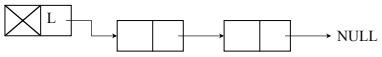


Insertion et suppression se font après recherche de la position de l'élément dans liste

Distinguer insertion en tête ou non (L est modifié ou non)

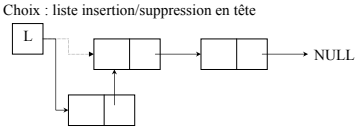


Solution : prendre pour L non pas un pointeur mais un élément dont la valeur est inutilisée

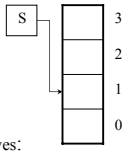


•Piles

Choix : liste insertion/suppression en tête



Autre choix : structure avec un tableau et un sommet (indice de tableau)



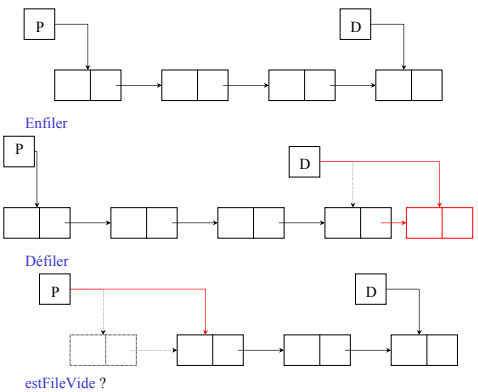
Ici - pile vide : S=-1
- pile pleine : S=3
- la pile comporte 2 éléments

•Primitives:

estPileVide , estPilePleine, empiler, dépiler (et new)

PAS de parcourir (comme pour les listes non triées)

•Files



estFileVide ?

•Structures ***linéaires***

- •Table
  - •accès indexé : **t[i]**
  - •traitement itératif : **FOR** , **WHILE** , **DO**
- •Liste chaînée
  - •un seul suivant, accès séquentiel : **l→suivant**
  - •traitement récursif : **IF (! L→estVide) ...**

•Structures ***arborescentes***

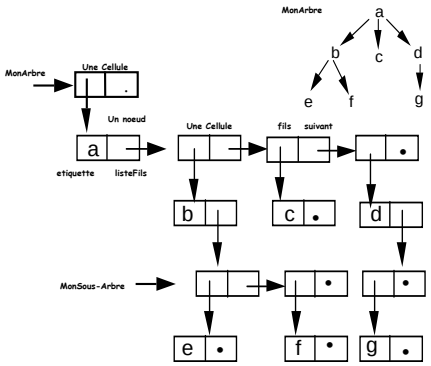
- •Arbre binaire : 2 suivants maximum

Arbre général : nombre quelconque de suivants

- •Traitement récursif : les suivants sont aussi des arbres
- •Un arbre n'a pas de cycle (moins général qu'un graphe)

•Arbre : structure de données
•Un ***arbre*** est une collection (finie) de ***noeuds*** de même type
•Un arbre est défini (récursivement) par :
  •un premier noeud (appelé ***racine*** de l'arbre)
  •0, 1, 2 ou plusieurs ***sous- arbres***
  •disjoints (appelés ***descendants*** de la racine)
Vocabulaire
• ***feuille*** : noeud sans descendant
• ***fils*** d'un noeud : racine d'un descendant de ce noeud
Sur l'exemple
Le noeud de valeur t est
un fils du noeud …
Les feuilles sont les noeuds …

Arbres n_aires

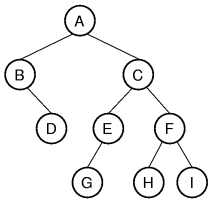



```
• typedef struct cell {
      struct noeud * fils ;
      struct cell * suivant ;
    } CELLULE, * ARBRE ;

• typedef struct noeud {
      ELEMENT etiquette ;
      ARBRE listeFils ;
    } NOEUD ;
```


Arbre binaire

Notation:

Noeud, enfant, arête, parent, ancêtre, descendant, chemin, profondeur, hauteur, niveau, feuille, noeud interne, sous-arbre.

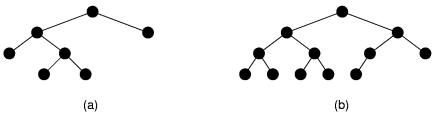


- l'ensemble ne contient aucun nœud, ou
- l'ensemble est constitué de trois sous-ensembles disjoints de nœuds:
  - un nœud unique appelé **racine**
  - un arbre binaire appelé **sous-arbre de gauche**
  - un arbre binaire appelé **sous-arbre de droite**.

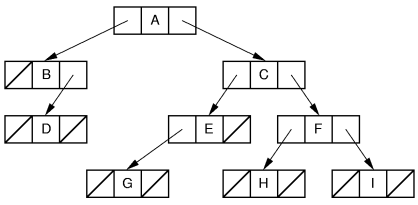
Arbre binaire plein et complet

Arbre binaire plein: Chaque noeud est soit une feuille soit possède exactement deux enfants

Arbre binaire complet: Si la hauteur de l'arbre est *d*, alors tous les niveau sont complets, sauf éventuellement le dernier. Les feuilles au dernier niveau sont complètement à gauche.

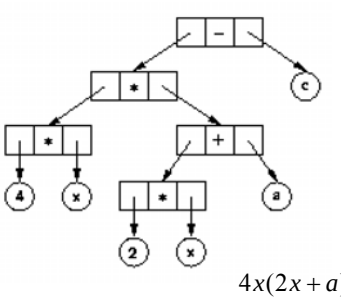


Implémentation




```
struct noeud{
  int valeur;
  struct noeud *fils_d,fils_g;
};
typedef noeud nœud,*arbre;
```

Exemple : expression algébrique



$$4x(2x+a)-c$$

Parcours

Une méthode permettant de visiter tous les nœuds d'un arbre dans un ordre quelconque est appelée un parcours (ou une fouille).

Un parcours où chaque nœud est visité exactement une fois est appelé une énumération.

- Parcours en préordre (préfixé): on visite un nœud avant de visiter ses enfants.
- Parcours en postordre (postfixé): on visite un nœud apès avoir visiter ses enfants.
- Parcours en ordre (infixé): on visite d'abord le sous-arbre de gauche, puis la racine et enfin le sous-arbre de droite.

Exercices:

1. Réaliser 2 bibliothèques pile_t et pile_af selon les 2 choix possibles (tableau ou structure auto-ref.) ainsi qu'un programme de test (empiler une suite d'entiers puis les afficher en ordre inverse en les dépilant) ainsi qu'un makefile adapté
2. Réaliser une calculatrice post fixée à l'aide de la pile
3. Lire une suite de mots les insérer dans une structure de liste triée puis les afficher dans l'ordre
4. Simuler le fonctionnement d'une imprimante : mettre des mots dans une file d'attente et les afficher dans l'ordre d'arrivée (file)
5. Lire une expression algébrique post-fixée et l'afficher en pré-fixé (arbre binaire)