Royaume du Maroc
Université Hassan II – Casablanca
Ecole Supérieure de Technologie
Casablanca

Polycopié du cours

structures de données en C.

M. OUZZIF

Chapitre 1 : Eléments de base

1. Introduction

Les structures que nous avons traitées jusqu'à présent sont des structures linéaires contiguës de type vecteurs ou fichiers séquentiels. Dans ce chapitre nous allons aborder une représentation dynamique de ces structures linéaires en utilisant la notion de pointeur permettant des chainages non contigus.

2. Notion de pointeur

Un pointeur est une variable dont le contenu est une adresse. Supposons que p est une variable de type pointeur qui contient l'adresse de l'information a. p sera représenté par la figure 1.

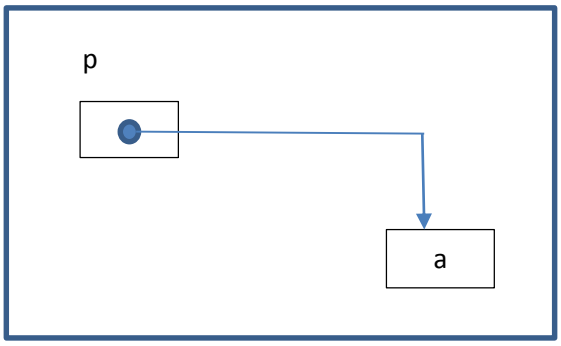


Figure 1. Représentation graphique d'un pointeur.

La déclaration utilisée pour p est la suivante :

Var p : pointeur ;

Pointeur est un type défini par :

Type pointeur : ^objet ;

Objet peut être de type simple ou structuré.

p désigne la variable qui contient l'adresse. Cette variable peut être utilisée en partie droite ou gauche d'une affectation contenant des adresses.

p^ : désigne l'objet dont l'adresse est rangée dans p. p^ peut apparaître en partie droite ou gauche d'une affectation concernant des objets de même type que celui pointé par p.

Exemple :

Soient deux variables p1 et p2 de types pointeur permettant respectivement l'accès à a et b (figure 2).

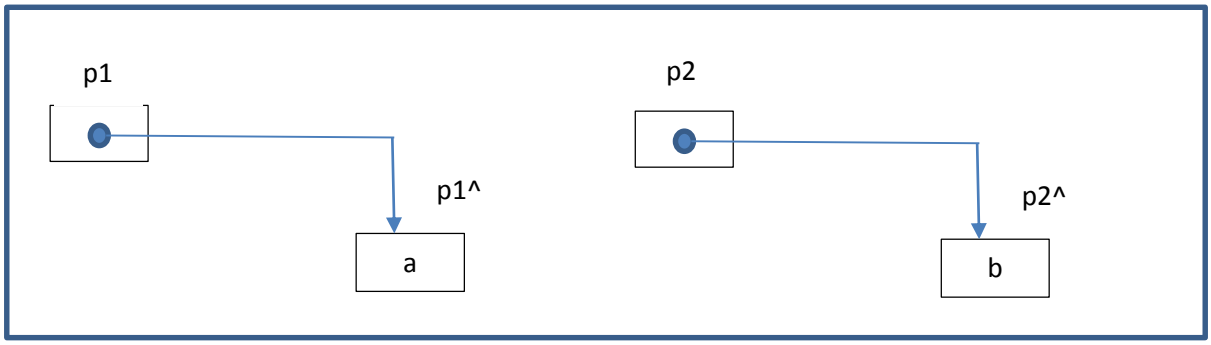


Figure 2. Pointeurs sur deux données différentes.

p1← p2 permet d'obtenir la situation représentée par la figure 3.

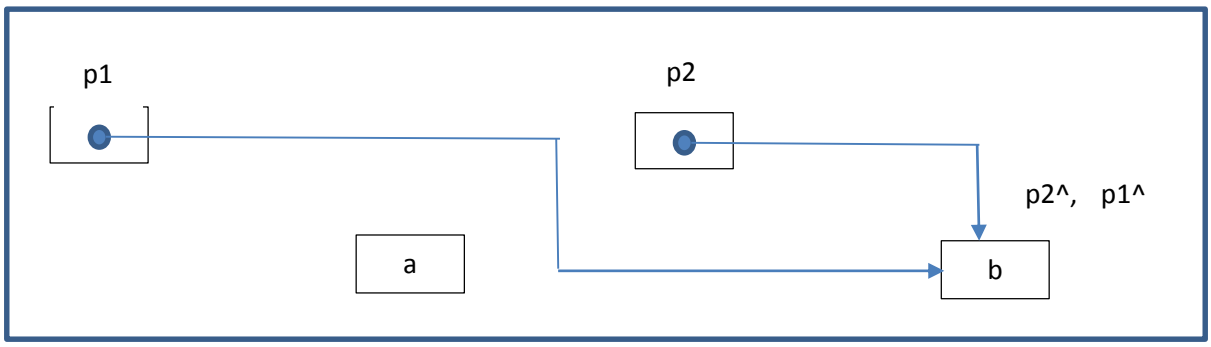


Figure 3. Perte d'accès à une donnée.

Remarque : a n'est plus accessible car son adresse qui se trouvait dans p1 a été perdu.

p1^ ← p2^ permet d'obtenir la situation représentée par la figure 4 dans laquelle a n'existe plus et la donnée b existe en deux exemplaires dans les zones mémoires adressées par les pointeurs p1 et p2.

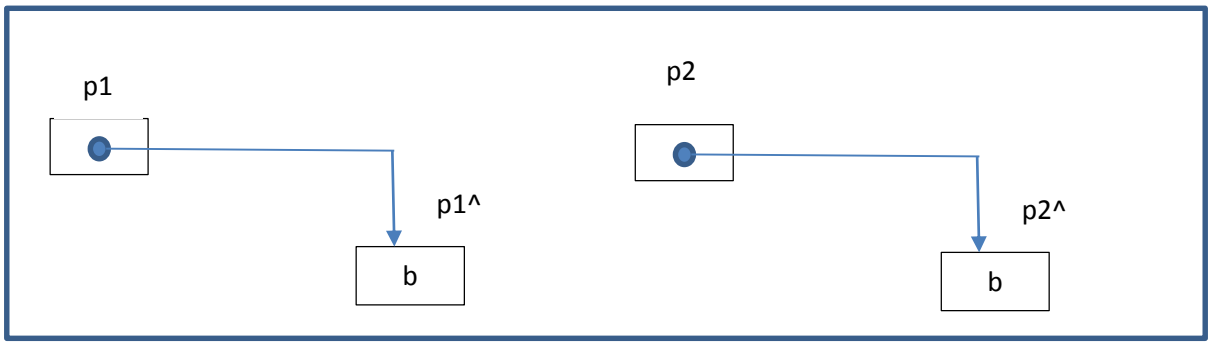


Figure 4. Perte de données par écrasement du contenu d'un pointeur.

- Valeur NIL

On dispose d'une valeur conventionnelle nommée NIL de type pointeur qui ne correspond à aucune adresse possible d'objet.

p=NIL est représentée graphiquement par la figure 5.

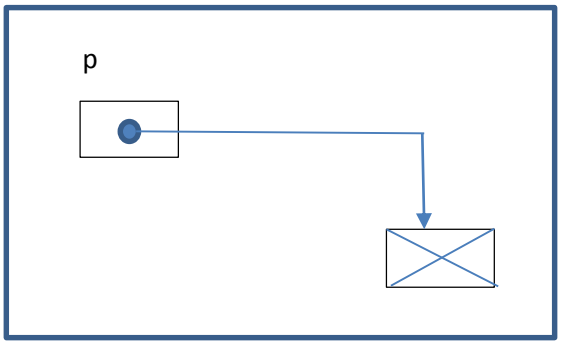


Figure 5. Représentation graphique du pointeur NIL.

3. Notion de cellule

En vue de construire dans la suite de ce chapitre des structures linéaires à composantes chaînées dynamiquement, nous introduisons la notion de cellule. Cette peut être représentée par un enregistrement comprenant une donnée « d » de type quelconque « t » et un pointeur « suivant » sur une structure de type cet enregistrement.

Type cellule = enregistrement
    d : t ;
    suivant : pointeur ;
Fin cellule ;

- t est un type simple, vecteur, enregistrement ou autre.

- Pointeur est le type défini par :
    - type pointeur : ^cellule ;

La figure 6 donne une représentation graphique d'une cellule.

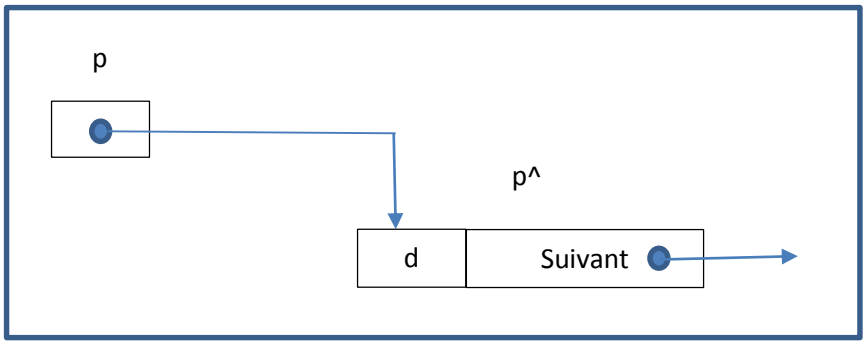


Figure 6. Représentation graphique d'un pointeur.

L'adresse de la cellule est rangée dans la variable p. On parle généralement de cellule pointée par p. La cellule contient deux champs. Un champ d de type t et un champ suivant contenant une adresse. Si p est différent de NIL, p^.d permet d'accéder à la donnée de la cellule p^.suivant permet d'accéder à la zonne pointée par p^.suivant qui, au cas où il est différent de NIL, représente la cellule suivante.

4. Gestion dynamique de la mémoire

Nous pouvons considérer la mémoire comme un réservoir de cellules. Nous pouvons effectuer deux opérations sur ce réservoir pour sa gestion dynamique :

- Créer une nouvelle cellule par la primitive « nouveau ».
- libérer une cellule par la primitive « libérer ».

Procédure nouveau (Var p : poiteur)
Spécification {reservoir non vide} ==> {p contient l'adresse d'une nouvelle cellule allouée}

Cette primitive permet à chaque appel d'obtenir une nouvelle cellule dont l'adresse est retournée dans la variable p.
Remarque : Var signifie que le paramètre est passé par variable (ou référence).

Procédure liberer (Val p : poiteur)

Spécification { } ==> {rend la cellule d'adresse p au resvoir}

Cette primitive permet à chaque appel de restituer la cellule pointée p.

Remarque : Val signifie que le paramètre est passé par valeur.

Ces deux primitives permettent d'obtenir ou de rendre des cellules à la mémoire au fur et à mesure des besoins de l'algorithme. ===> On parle de gestion dynamique de la mémoire contrairement à la gestion statiques des vecteurs (ou tableaux).

4. Pointeurs en C.

On appelle Lvalue (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une Lvalue est caractérisée par :

- son adresse, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- sa valeur, c'est-à-dire ce qui est stocké à cette adresse.

```
int i, j;
i = 3;
j = i;
```

Si le compilateur a placé la variable i à l'adresse 4831836000 en mémoire, et la variable j à l'adresse 4831836004, on a :

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

Deux variables différentes ont des adresses différentes. L'affectation i = j; n'opère que sur les valeurs des variables. Les variables i et j étant de type int, elles sont stockées sur 4 octets. Ainsi la valeur de i est stockée sur les octets d'adresse 4831836000 à 4831836003.

L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier quelque soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures. Sur un DEC alpha, par exemple, une adresse a toujours le format d'un entier long (64 bits).

L'opérateur & permet d'accéder à l'adresse d'une variable. Toutefois &i n'est pas une Lvalue mais une constante : on ne peut pas faire figurer &i à gauche d'un opérateur d'affectation. Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs. on définit un pointeur p qui pointe vers un entier i :

```
int i = 3;
int *p;
p = &i;
```

On se trouve dans la configuration :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

L'*opérateur unaire d'indirection* * permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si p est un pointeur vers un entier i, *p désigne la valeur de i. Par exemple, le programme :

```
main ()
{
  int i = 3;
  int *p;

  p = &i;
  printf("*p = %d \n",*p);
}
```

imprime *p = 3.

Dans ce programme, les objets i et *p sont identiques : ils ont mêmes adresse et valeur. Nous sommes dans la configuration :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Cela signifie en particulier que toute modification de *p modifie i. Ainsi, si l'on ajoute l'instruction *p = 0; à la fin du programme précédent, la valeur de i devient nulle.

On peut donc dans un programme manipuler à la fois les objets p et *p. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
}
```

et

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Après l'affectation *p1 = *p2; du premier programme, on a :

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Par contre, l'affectation p1 = p2 du second programme, conduit à la situation :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

5. Notion de récursivité.

La récursivité est un concept général qui peut être illustré dans (quasiment) tous les langages de programmation, et qui peut être utile dans de nombreuses situations. Elle est très utile dans les algorithmes de manipulation des structures de données.

La définition la plus simple d'une fonction récursive est la suivante : c'est une fonction qui s'appelle elle-même. Si dans le corps (le contenu) de la fonction, vous l'utilisez elle-même, alors elle est récursive.
L'exemple habituel est la fonction factorielle. Cette fonction, provenant des mathématiques (et très utilisée dans certains domaines mathématiques) prend en entrée un entier positif, et renvoie le produit des entiers inférieurs jusqu'à 1, lui compris :

fac(n) = n * (n-1) * ….. * 1
Par exemple la factorielle de 4 est 4 * 3 * 2 * 1 = 24.

Pour pouvoir trouver une solution récursive, il faut trouver la condition d'arrêt des appels appelée aussi condition de sortie.

Dans ce problème, la condition d'arrêt est n=0 ou n=1.

Le programme C permettant de calculer récursivement la factorielle est le suivant :

```
unsigned long factoriel (int n)
{
  if (n < 0) {  exit (EXIT_FAILURE);    }
        else if (n == 1 || n == 0)  return 1;
                  else   return n * factoriel (n - 1);
}
```

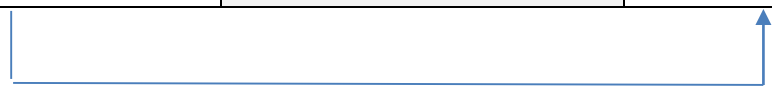
Nous pouvons observer ici que le dernier return est en fait l'appel récursif et nous soustrayons 1 à chaque appel jusqu'à ce que n == 1 qui est, comme décrit plus haut, notre condition de sortie.

Non, cela ne s'arrête pas là et c'est ici que nous allons voir le fonctionnement des fonctions récursives. Lorsque la fonction rencontre la condition de sortie, elle remonte dans tous les appels précédents pour calculer n avec la valeur précédemment trouvée !

Les appels des fonctions récursives sont en fait empilées (pile qui est une structure de donnée régie selon le mode LIFO: Last In First Out, Dernier Entré Premier Sorti). Chaque appel se trouve donc l'un à la suite de l'autre dans la pile du programme. Une fonction de ce type possède donc deux parcours: la phase de descente et la phase de remontée.

Voyons ceci grâce à un petit schéma:

Descente		Remontée
n==6		n*120=720
n==5		n*24=120
n==4		n*6=24
n==3		n*2=6
n==2		n*1=2
n==1		n==1



Condition de sortie : n==1

Les fonctions récursives étant un moyen assez puissant pour résoudre certains problèmes de façon élégante, elles n'en restent pas moins dangereuses et ce pour plusieurs raisons : le dépassement de capacité et le débordement de pile (Stack Overflow).

- **Dépassement de capacité**

Une des causes assez fréquente quand vous travaillez sur de très grands nombres, est le dépassement de capacité. C'est un phénomène qui se produit lorsque vous essayez de stocker un nombre plus grand que ce que peut contenir le type de votre variable.

Il est d'usage de choisir un type approprié, même si vous êtes certains que le type que vous avez choisi ne sera jamais dépassé, utilisez tant que possible une variable pouvant contenir de plus grandes données. Ceci s'applique à tous types de données.

Evitez le type int si vous travaillez avec une fonction récursive, comme les exemples précédents pour le calcul de factoriels. Ce type est très petit et dans une fonction récursive il peut très vite arriver de le dépasser et c'est donc le plantage assuré. Préférez-lui un type comme long pour assurer un minimum la viabilité de votre application.

- **Débordement de pile**

Ceci est sans doute une des causes les plus souvent rencontrés dans le plantage de programmes avec des fonctions récursives. Nous savons que les appels récursifs de fonctions sont placés dans la pile du programme, pile qui est d'une taille assez limité car elle est fixée une fois pour toutes lors de la compilation du programme.

Dans la pile sont non seulement stockés les valeurs des variables de retour mais aussi les adresses des fonctions entre autres choses, les données sont nombreuses et un débordement de la pile peut très vite arriver ce qui provoque sans conteste une sortie anormale du programme.

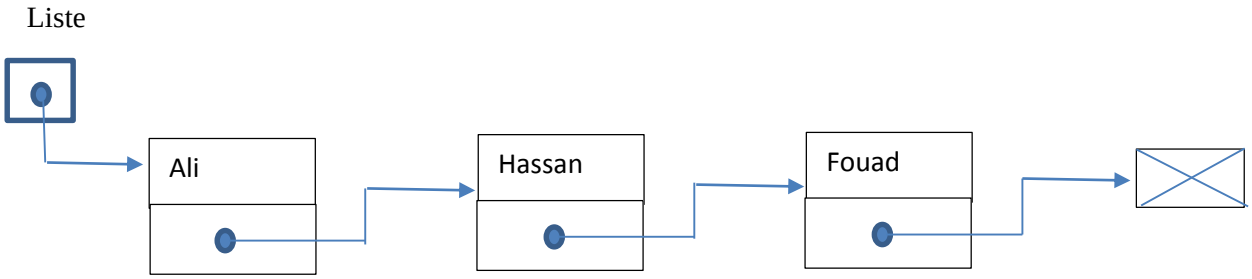
Une autre méthode existe cependant ! Si vous êtes presque sûr de dépasser ce genre de limites, préférez alors une approche itérative plutôt qu'une approche récursive du problème. Une approche récursive demande beaucoup de moyens en ressources car énormément de données doivent être stockées dans la pile d'exécution alors qu'en revanche, une approche itérative telle une boucle for est bien moins coûteuse en terme de ressources et est bien plus sûre, sauf dans le cas d'un dépassement de capacité bien sûr.

Chapitre 2 : Listes linéaires chaînées

1. Définition

Une liste linéaire chaînée est constitué d'une suite de cellules chaînées entre elles. L'adresse de la première cellule détermine toute la liste. Une liste est une structure souple : elle peut grandir ou se rétrécir à tout moment et de manière dynamique. Ses éléments peuvent être insérés ou supprimés à n'importe quel endroit. L'intérêt de l'utilisation de cette structure de données réside dans le fait qu'un nouvel élément n'occupe pas le mot mémoire suivant et par conséquent, sa création n'est pas liée à la contiguïté des zones mémoires (comme les tableaux).

Exemple d'une liste contenant des chaînes de caractères (prénoms):



2. Représentation

Nous représentons la liste linéaire chaînée comme étant un pointeur sur une cellule. On aura ainsi la description suivante :
Type cellule = enregistrement
d : t ;
suivant : pointeur ;

Fin cellule ;

Type pointeur = ^cellule ;
Type Tliste = ^cellule ;

Représentation en C :

typedef struct cel{
  int data;
  struct cel *next;
}cel;
typedef cel* cellule;
	Créer une cellule
	cellule createCellule(cellule next, int x){
	  cellule k = (cellule)malloc(sizeof(cellule));
	  k->data = x;
	  k->next = next;
	  return k;
	}

3. Création d'une liste

Supposons que nous voulons créer la liste linéaire chainée contenant les données a et b. Nous optons pour sa création à partir des derniers éléments pour des raisons de facilité.

Etape 1 : Création d'une liste vide d'adresse liste :

liste ← NIL

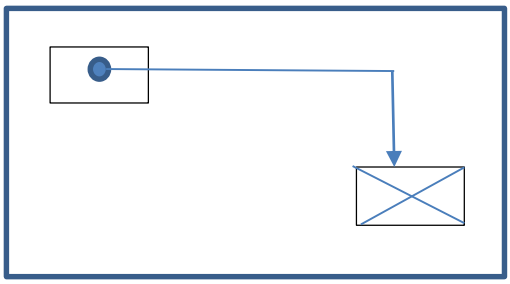


Figure 7. Création d'une liste NIL.

- Etape 2 : Création de la liste d'adresse liste et contenant l'information b :

Nous créons une cellule d'adresse p contenant l'information b

Nouveau(p) ;

p^.d ← b ;

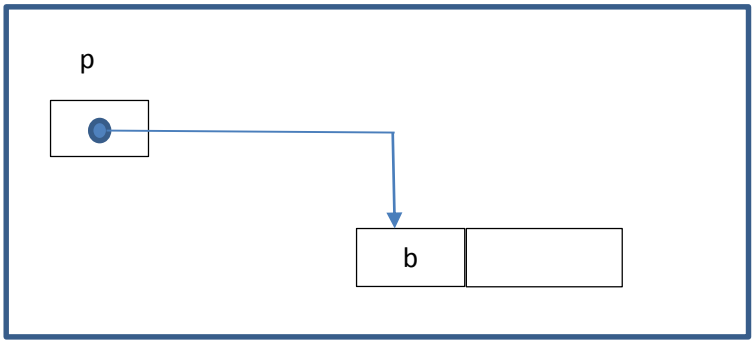


Figure 8. Insertion de l'élément b.

Puis nous chaînons avec la liste précédente :

p^.suivant←liste

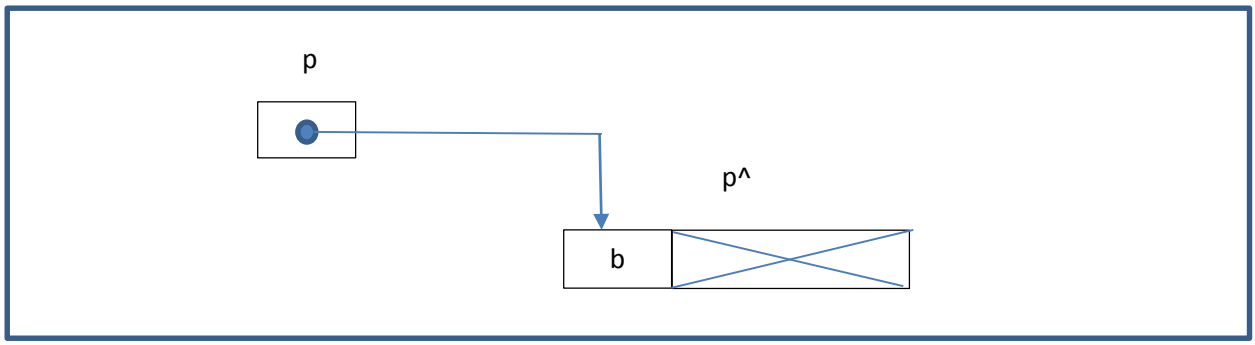


Figure 9. Chaînage avec la liste précédente.

Nous créons ensuite une liste d'adresse liste représentation de b :

Liste ← p ;

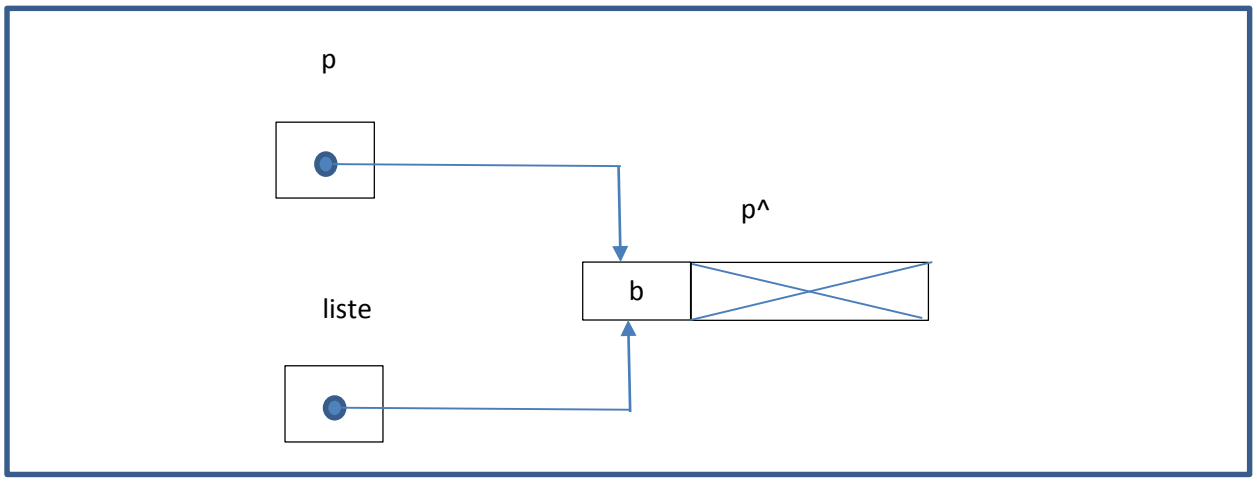


Figure 10. Création de la liste.

- Etape 3 : Création de la liste d'adresse liste représentant (a,b) :

Nous créons en premier lieu une nouvelle cellule d'adresse p contenant l'information a :

Nouveau (p) ;

P^.info ← a ;

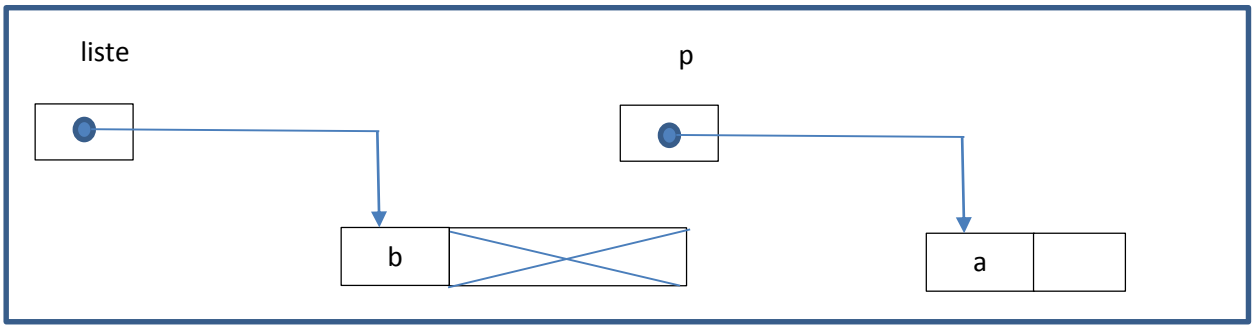


Figure 11. Création d'une nouvelle cellule contenant a.

Nous chainons ensuite avec la liste précédente :

p^.suivant ← liste ;

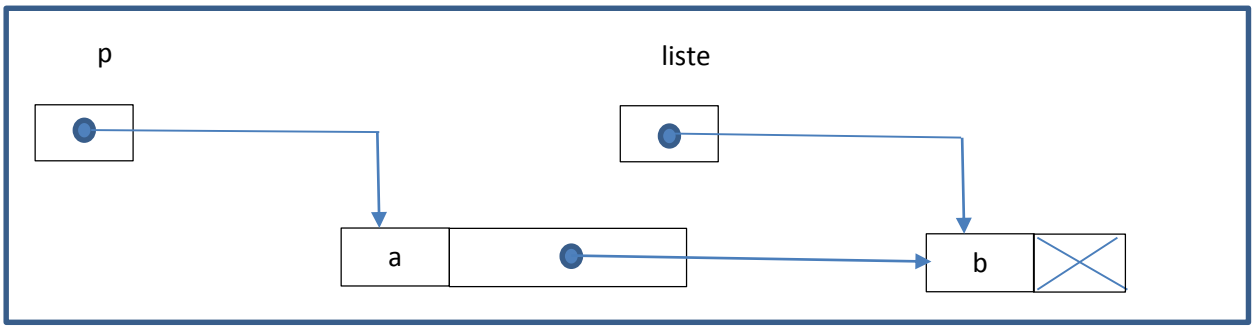


Figure 12. Chaînage avec la liste précédente.

Nous créons ensuite la liste d'adresse liste représentant (a,b) :

Liste ← p ;

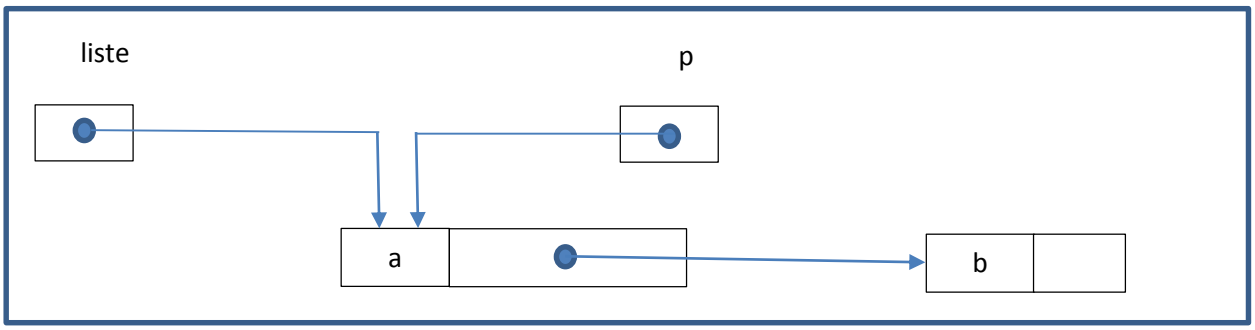


Figure 13. Création de la liste représentant (a,b) .

Nous conseillons les étudiants d'utiliser ce type de représentations graphiques en vue de matérialiser le raisonnement logique de l'algorithme à élaborer. Cet algorithme ne sera que la suite des instructions relative à chaque illustration graphique.

On peut généraliser cet algorithme en supposant que les éléments successifs se trouvent dans un fichier f. Dans ce cas, on crée la liste à partir de son dernier élément. On obtient ainsi une liste contenant les éléments rangés dans l'ordre inverse.

L'algorithme est le suivant :

```
Procédure créerliste (Val f :fichier de t ; Var liste : pointeur) ;
Spécification {} ====> {création de liste+ composée des éléments de f}
   p, liste1 : pointeur ;
   c : t ;
Debproc
liste1←NIL ; relire(f) ;
Tantque non(fdf(f)) faire
   Nouveau (p) ;
      lire(f,c) ;
      p^.d←c
      p^.suivant←liste1 ;
      liste1 ←p
Finfaire ;
liste ← liste1 ;
finproc
```

4. Passage d'une représentation vecteur à une représentation liste chaînée.

A partir des éléments V[i] d'un vecteur V, on crée la liste à partir de son dernier élément et du dernier élément du vecteur. On obtient ainsi les éléments dans le même ordre.

L'algorithme est alors le suivant:

```
Procédure vectliste (d V[1..n] : vecteur; r liste: pointeur) ;
   Spécification { } ====> {création de liste+ composée des éléments de V}
      i: entier;
      p, liste1: pointeur;
   debproc
      liste1←nil ;
      i← n;
```

```
Procédure vectliste (val V[1..n] : vecteur; Var liste: pointeur) ;
   Spécification { } ====> {création de liste+ composée des éléments de V}
      i: entier;
```

```
        p, liste1: pointeur;
    debproc
        liste1←nil ;
        i← n;
        tantque i >= 1 faire
        nouveau (p) ;
        pi .d ←V[i] ;
        pi.suivant := liste1 ;
        liste1←p;
        i ←i -1 ;
    finfaire;
    liste ←liste1 ;
finproc;
```

Pour obtenir les éléments du fichier dans le même ordre, on doit créer la liste à partir de son premier élément. Pour ce faire, on doit d'abord prévoir le cas du premier élément qui permet la création de la première cellule de la liste et dont l'adresse doit être préservée car elle permettra, comme nous l'avons vu, l'accès à toutes les cellules créées.

Ensuite, pour chaque nouvel élément de f, on doit créer une nouvelle cellule qu'il faudra insérer en fin de liste et non plus en tête. L'algorithme est alors le suivant:

```
Procédure créerliste (Val f : fichier de t; Var liste: pointeur) ;
Spécification { } ====>  {création de liste+, à l'endroit, composée des éléments du
fichier f}
    p, der: pointeur;
    c:
debproc
    relire (f) ;
    si fdf (f) alors
        liste ← NIL ;
        sinon
        nouveau (der) ; lire (f, c) ; der^.info ←c;
        liste := der; (adresse de la dernière cellule)
        tantque non fdf (f) faire
            lire (f, c) ;
            nouveau (p);
            (création d'une nouvelle cellule)
            p^.d ← c;
            der.suivant := p;
            (ajout en fin de liste)
```

```
                    der← p;
                    (adresse de la dernière cellule)
                finfaire ;
                deri.suivant ← nil;
                (fin de liste)
            finsi;
        finproc;
```

Comme on peut le constater, cet algorithme est moins simple que le précédent.

5. Parcours d'une liste chaînée

Schéma récursif

Pour l'écriture des algorithmes, sous forme récursive, on utilise la définition suivante d'une liste chaînée:

Une liste chaînée est:

- soit une liste vide (liste = NIL),
- soit composée d'une cellule (la première) chaînée à une sous-liste (obtenue après suppression de la première cellule).

• Exemple :

liste+ = (a, b, c, d, e) est composée d'une cellule contenant a et d'une sous-liste contenant (b, c, d, e). Cette sous-liste a pour adresse listei.suivant. On l'appelle sous-liste (ou liste) liste^.suivant+.

Premier parcours

Ce parcours traite les cellules de listé de gauche à droite. On appellera ce parcours parcoursgd.

Le schéma d'algorithme est alors le suivant:

```
procédure parcoursgd (d liste: pointeur) ;
    debproc
    si liste # NIL alors
        traiter (liste) ;
        parcoursgd (liste^.suivant) ;
    finsi;
finproc;
```

Remarque :
la variable liste est une donnée. Traiter (liste) ne doit donc pas modifier la valeur de liste

Deuxième parcours

Ce parcours traite les cellules de liste+ de droite à gauche. On appellera ce parcours parcoursdg.

Le schéma d'algorithme est alors le suivant:

```
procédure parcoursdg (d liste: pointeur) ;
        debproc
            si liste # NIL alors
                parcoursdg (liste^..suivant) ;
                traiter (liste) ;
        finsi;
finproc;
```

Exemple

Soit liste+ = (a, b, c, d). Pour les besoins de l'explication, nous avons appelé l'adresse de la

ième cellule listei.

Dans le cas de l'exécution de l'instruction parcoursgd, on obtient:

parcoursgd (listei) permet d'obtenir:

•• traiter (listei):

•• parcoursgd (listei+ 1);

parcoursgd (Iistel):

• • traiter (liste1);

•• parcoursgd (IisteZ):

••• traiter (liste2)

••• parcoursgd (liste3)

•••• traiter (liste3)

• • • • parcoursgd (liste4)

••••• traiter (liste4)

• • • • • parcoursgd (nil)

le parcours est terminé.

On constate que l'on traite les éléments dans l'ordre c'est-à-dire de gauche à droite. Dans le cas de l'exécution de l'instruction parcoursdg (listei), on obtient:

• parcoursdg(Iistei)

    • • parcoursdg (listei+1);

    • • traiter (Iistei):

Il faut traiter parcoursdg (Iistei+l) avant traiter (listei)

On obtient alors:

• parcoursdg (listel) ;

    • • parcoursdg (liste2) ;

        ••• parcoursdg (liste3) ;

            • • • • parcours dg (liste4) ;

            • • • • • parcoursdg (nil) ;

            ••••• traiter (liste4)

            •••• traiter (liste3)

        ••• traiter (liste2)

    • • traiter (liste 1) ;

Le parcours est terminé. On constate:

• que, pour chaque appel correspondant à une liste non vide, on doit exécuter les deux instructions prévues dans la condition,

• que la deuxième instruction n'est exécutée qu'après terminaison de l'exécution de la première,

• que les éléments sont alors traités dans l'ordre inverse, c'est-à-dire de droite à gauche.

Ces deux parcours sont très utilisés et sont à rapprocher de la création de la liste à l'endroit ou à l'envers que nous avons effectuée précédemment.

En C :

```
int voirKemeCelluleRecursive(cellule tete, int k){
   if(k == 1){
      return tete->data;
   }
   else
      voirKemeCelluleIterative(tete->next,k-1);
}
```

Schéma itératif

Le deuxième parcours n'est pas simple à obtenir sous forme itérative. Dans le cas général, il est nécessaire de disposer d'une pile, comme nous le verrons dans les chapitres suivants. Nous nous limiterons donc, pour le moment, au premier parcours qui correspond à ce qu'on appelle une récursivité terminale; c'est-à-dire qu'il n'y a pas d'instruction à exécuter après l'appel récursif. Le schéma itératif est alors le suivant:

```
procédure parcoursgd (Val liste: pointeur) ;
        listel: pointeur;
    debproc
        listel←liste;
        tant que listel # NIL faire
            traiter (Iiste l) ;
            listel←listel^.suivant ;
        finfaire;
    finproc;
```

Le paramètre liste étant un paramètre donnée (d), l'adresse de la première cellule est protégée par l'instruction : listel:= liste; Dans le cas paramètre serait une donnée-résultat (dr), il serait également nécessaire d'utiliser une variable auxiliaire de type pointeur pour effectuer le parcours afin de ne pas perdre l'accès à la liste. L'algorithme serait alors le même.

En C :

```
int voirKemeCelluleIterative(cellule tete,int k){
  while(k > 0){
        tete = tete->next;
        k--;
  }
    return tete->data;
}
```

6. Algorithmes sur les listes chaînées

On propose d'écrire des algorithmes de parcours d'une liste chaînée en commençant d'abord par l'écriture, sous forme récursive, des éléments d'une liste.

a- Ecriture des éléments d'une liste

L'écriture de gauche à droite d'une liste chaînée correspondra au premier parcours alors que l'écriture de droite à gauche correspondra au deuxième parcours.

Ecriture de gauche à droite

On utilise le premier parcours parcoursgd. On écrit donc les éléments de la liste à partir du premier élément.

L'algorithme est le suivant:

```
Procédure écritlistegd (d liste: pointeur) ;
    Spécification { } ====> {les éléments de liste+ ont été écrits de gauche à droite}
  debproc
    si liste # nil alors
      écritexte (Iiste^.d) ;
      écritlistegd (Iiste^.suivant) ;
    finsi;
  finproc;
```

b- Ecriture de droite à gauche

On utilise le deuxième parcours parcoursdg qui permet d'obtenir l'écriture des éléments à partir du dernier élément.

L'algorithme est le suivant:

```
Procédure écritlistedg (d liste: pointeur) ;
    Spécification { } ====> {les éléments de liste+ ont été écrits de droite à gauche}
debproc
    si liste # nil alors
        écritlistedg (liste^ .suivant) ;
        écritexte (liste^.info) ;
    finsi;
finproc;
```

Quiz :

Que font les deux procédures suivantes :

```
Procédure écritliste (d liste: pointeur) ;
debproc
    si liste ;f. nil alors
        écritexte (liste^.info) ;
        écritliste (liste^.suivant) ;
        écritexte (liste^ .info) ;
    finsi;
finproc;
```

```
Procédure écritliste (d liste: pointeur) ;
    debproc
        si liste ;f. nil alors
            écritliste (liste ^.suivant) ;
            écritexte (Iiste^.info) ;
            écritliste (liste^.suivant) ;
    finsi;
finproc;
```

c- Calcul de la longueur (nombre d'éléments) d'une liste

Schéma itératif

Hypothèse: supposons traités les n premiers éléments de la liste. On a la situation suivante:

n = long (liste1+) où long désigne le nombre d'éléments de la liste listel-.

Deux cas se présentent:

- liste1 + est vide : l'algorithme est terminé, la liste possède n éléments.
- liste1 + n'est pas vide

  La liste liste1+ a au moins un élément.
  L'exécution des instructions:
  n←n+ 1;
  listel:= liste1^.suivant ;
  permet de conserver l'assertion: n = long (listel-)

Initialisation : Il suffit que la variable liste1 contienne l'adresse de la première cellule de la liste et d'initialiser la variable n à zéro en exécutant les instructions:

```
liste1←liste;
n←0 ;
```

pour obtenir l'invariant: n = long (Iiste1-).

Ce raisonnement peut se résumer par: Hypothèse n = long (Iistel-)

Cet algorithme est une transposition de l'algorithme de calcul du nombre des éléments d'un fichier.

On peut écrire alors la fonction suivante:

```
Fonction long (Val liste : pointeur) :entier ;
Debfonc
n←0 ;
liste1← liste ;
tant que liste1#NIL faire
        n←n+1 ;
        liste1←liste1^.suivant ;
finfaire
retour (n) ;
finfonc
```

Schéma récursif

Le raisonnement est le suivant :

On a deux cas :

Si liste+ : sa longueur est alors égale zéro. Ceci permet de terminer l'algorithme.

Si liste+ n'est pas vide : elle est donc composée d'une cellule (le premier élément) chaînée à la sous liste liste^.suivant+. : la longueur est égale à un plus la longueur de la liste liste^.suivant.

```
Fonction long (Val liste : pointeur) :entier ;
Debfonc
Si liste =NIL alors retour 0 ;
        Sinon
                Retour 1+long (liste^.suivant)
finsi;
```

d- Insertion dans une liste

Insertion en tête de liste :

Pour insérer un élément en tête de liste, Il faut d'abord créer une cellule p d'adresse p par exécution de l'instruction nouveau (p). Ensuite, le champ information reçoit la valeur elem à insérer. Pour terminer par la réalisation de deux liaisons (a) et (b) dans cet ordre figure 14.

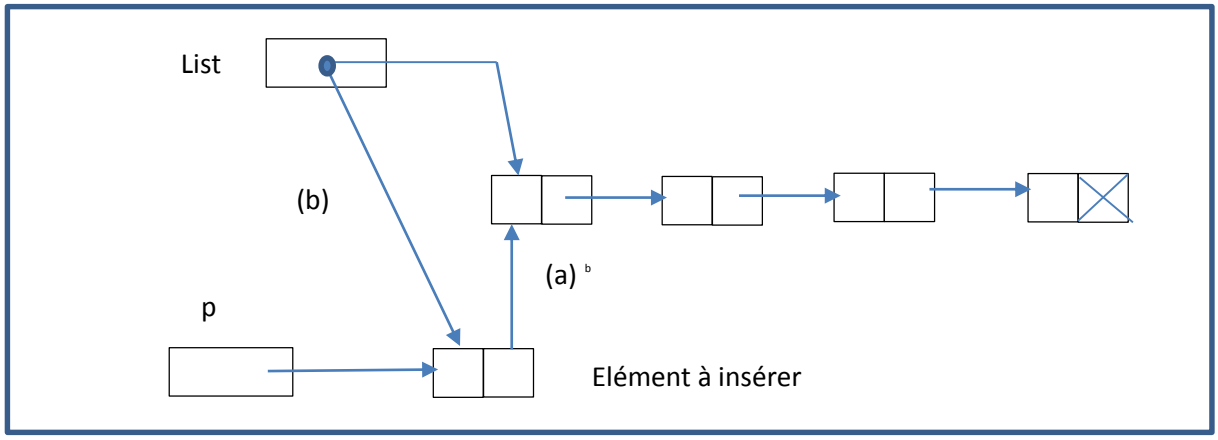


Figure 14. Insertion en tête de liste

```
Procédure inserertete (Var liste: pointeur, val elem ; t)) ;
p : pointeur
debproc
        nouveau(p) ;
        p^.info←elem ;
        p^.suivant←liste ;
        liste←p ;
finproc;
```

Insertion en tête de liste :

Si la liste est vide nous sommes ramenés à une insertion en tête. Dans le cas général, nous supposerons que la liste n'est pas vide. La figure 15 schématise cette insertion.

Après avoir créé une cellule d'adresse p contenant l'information à insérer elem, nous devons effectuer les liaisons a et b dans cet ordre. Pour pouvoir effectuer la liaison (b), il faut connître l'adresse der de la dernière cellule. Nous supposons que nous disposon d'une fonction dernier qui nous retourne ce pointeur.

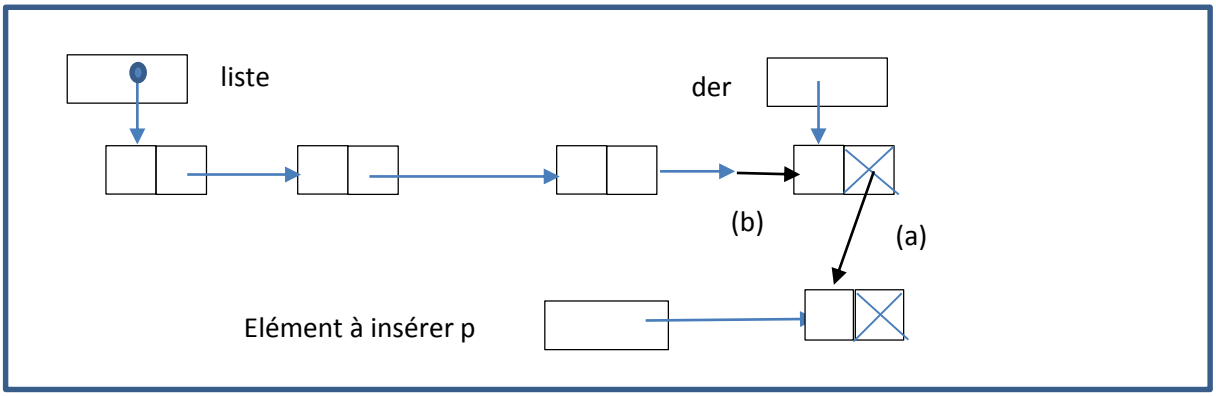


Figure 15. Insertion en fin de liste

```
Procédure insererfin (Var liste: pointeur, val elem ; t)) ;
p, der : pointeur ;

Debproc
        Si liste=NIL alors inserertete(liste,elem) ;
          Sinon
                  der←dernier(liste) ;
                  nouveau(p) ;
                  p^.info←elem ;
                  p^.suivant←NIL ;
                  der^.suivant=p ;
finproc;
```

Maintenant nous allons spécifier la fonction dernier :

```
Fonction dernier (Val liste: pointeur) : pointeur;
P, precedent : pointeur
debproc
        p←liste ;
        tant que p # NIL faire
          precedet←p ;
          p←p^.suivant ;
        finfaire ;
        retour precedent ;
finproc;
```

Exercice :
Ecrire la version récursive de la fonction dernier.

Insertion à la kième place :

L'insertion à la kième place consiste à créer les liaisons (a) et (b) dans cet ordre dans les deux figures : 16 et 17. Les élément suivants seront automatiquement décalés d'une position sans qu'on ait à les déplacer. Pour réaliser la liaison (b), il faut connaître l'adresse de la cellule précédente (la kième -1). L'insertion n'est possible que si k appartient à l'intervalle [1..n+1] où n est le nombre d'éléments de la liste. Il faudra prévoir l'insertion en tête (k=1) car elle modofoe l'adresse de la liste. Dans le cas général, on utilisera la fonction pointk pour déterminer l'adresse de la kième cellule.

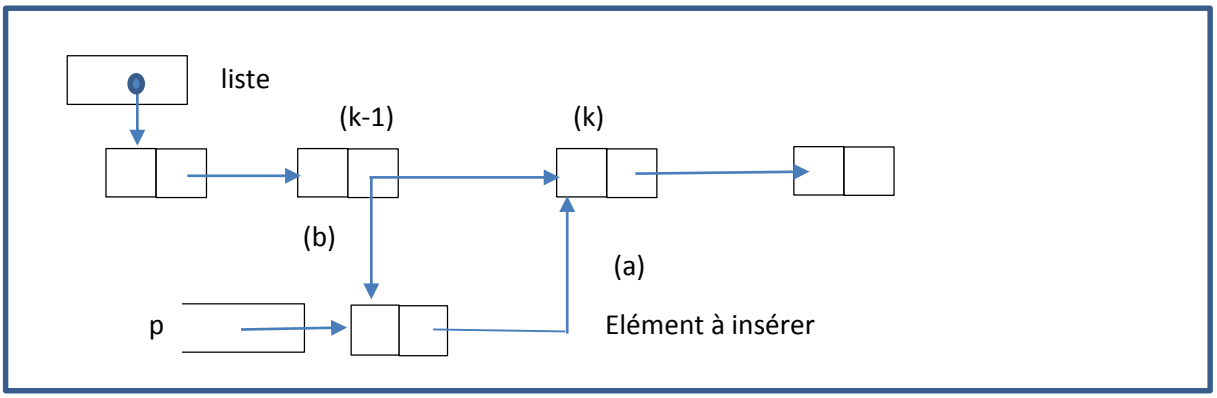


Figure 16. Insertion kième place (cas général)

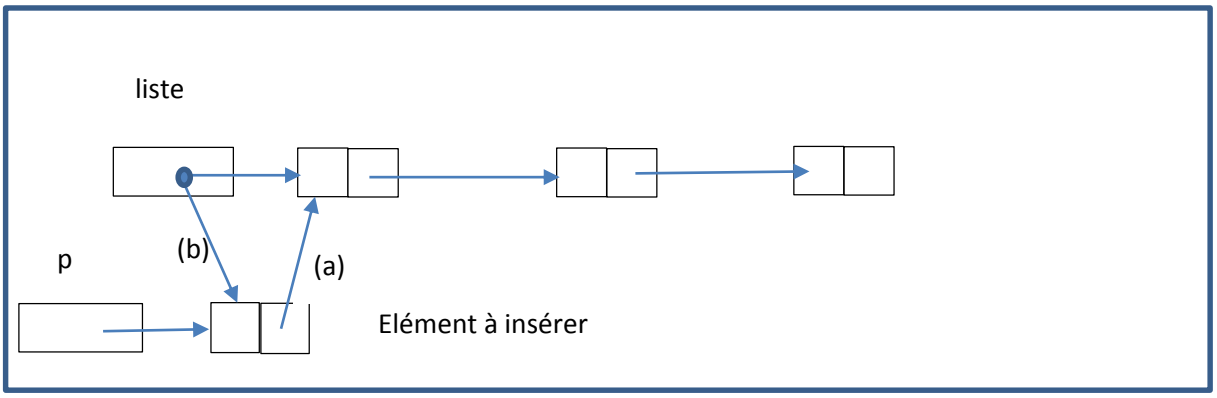


Figure 17. Insertion kième place (cas k=1)

Nous allons donner dans ce qui suit l'algorithme relatif à cette description.

```
Procédure insertk (Var liste: pointeur, Val k : entier ;val elem : t ; Var possible : booleen) ;
p, precedent : pointeur
debproc
        si k=1 alors inserertete(liste,elem) ;
                        possible ← vrai ;
          sinon
            precedent=pointk(liste, k-1) ;
            si precedent =NIL alors possible←faux ;
              sinon
                  nouveau(p) ;
                  p^.info←elem ;
                  p^.suivant← precedent^.suivant ;
                  precedent^.suivant ← p ;
                  possible ←vrai ;
          finsi
        finsi;
finproc
```

e- Suppression d'un élément d'une liste

Comme les insertions, on a le cas particulier de la suppression du premier élément qui a pour conséquence de modifier l'adresse de la liste. Dans le cas général, il suffit de modifier le contenu d'un pointeur comme le montre la figure 18.

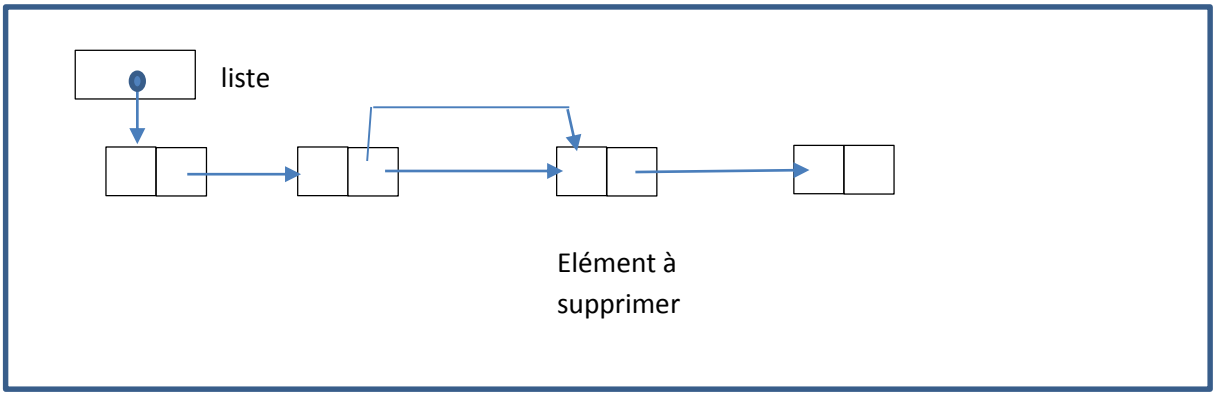


Figure 18. Suppression d'un élément de la liste.

Suppression du premier élément

On suppose que la liste n'est pas vide. On suppose également que l'on n'a plus besoin de l'élément et qu'on rend la cellule au réservoir de cellules. Il faut préserver par contre l'adresse de la tête de la liste avant d'effectuer la modification de cette adresse.

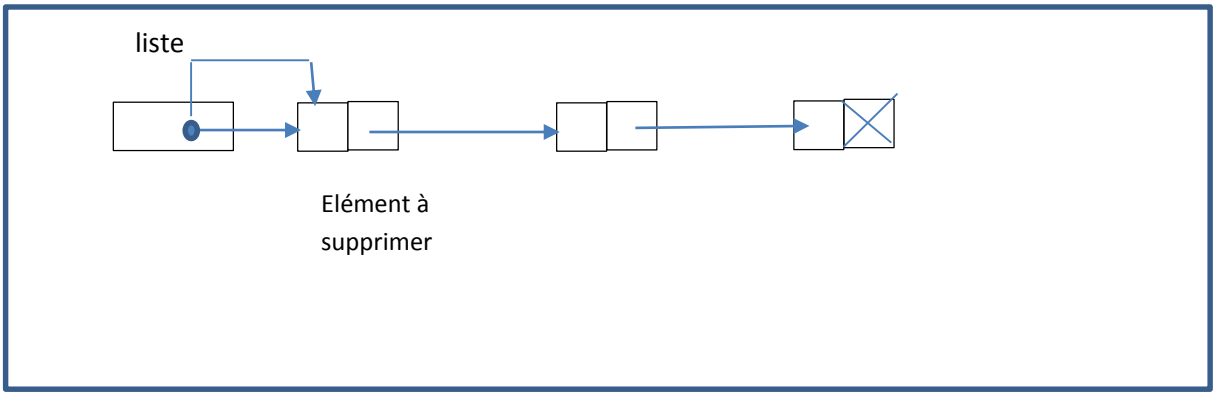


Figure 19. Suppression du premier élément

L'algorithme est le suivant :

```
Procédure supptete (Var liste: pointeur) ;
p, precedent : pointeur
debproc
        p ←liste ;
        liste ← liste^.suivant ;
        laisser(p) ;
finproc
```

Suppression par position

Nous voulons dans cette section supprimer le kième élément d'une liste. Il faut comme pour l'insertion, déterminer l'adresse « precedent » de la cellule qui précède celle que nous voulons supprimer : i.e. la kieme-1 cellule qui sera obtenue par un appel de la fonction pointk (à réaliser comme exercice). Ensuite, si la kième cellule existe, nous modifions la valeur de l'adresse contenue dans le champ suivant de la cellule d'adresse précédent comme indiqué dans la figure 20. Au préalable, nous aurions pris soin de traiter le cas particulier de la suppression du premier élément.

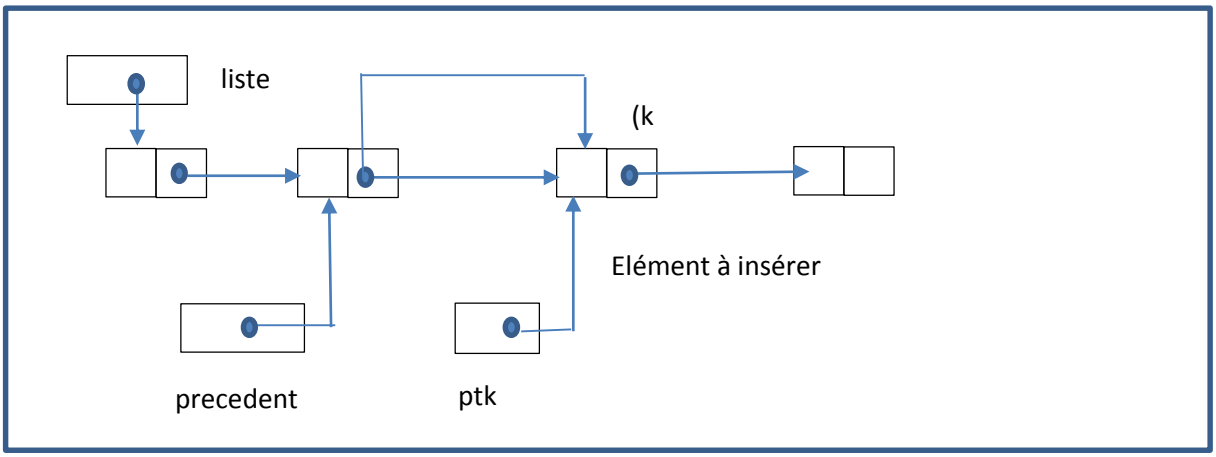


Figure 20. Suppression de la kième cellule

L'algorithme est le suivant :

```
Procédure supprimerk (Var liste: pointeur, Val k : entier ;val elem : t ; Var possible :
booleen) ;
p, der : pointeur ;

Debproc
        Si (liste#NIL) et (k=1) alors
                supprimertete(liste) ;
                possible ← vrai ;
          sinon
            possible ← faux ;
            precedent=pointk(liste, k-1) ;
            si precedent # NIL alors
                ptk ←precedent^.suivant;
                si ptk # NIL alors
                        possible=vrai ;
                        p^.suivant←ptk^.suivant ;
                        laisser (ptk) ;
                finsi
          finsi
finproc;
```

Exercice :

Ecrire la version récursive de la fonction supprimerk.

Plusieurs procédures et fonctions permettant de manipuler les listes sont fournies dans la partie programmes C des structures données fournie comme complément en fin de ce cours. Il est conseillé aux étudiants de les travailler avant de consulter la solution.

Chapitre 3 : Piles

1. Introduction

La structure de pile est une structure de liste particulière. Contrairement aux fichiers et vecteurs, elle ne sert généralement pas à garder de façon plus ou moins définitive des informations. On s'intéresse plutôt à la suite des états de la pile et on utilise le fait que le dernier élément ajouté se trouve au sommet de la pile, afin de pouvoir être atteint le premier.

On peut donner une image du fonctionnement de cette structure avec une pile d'assiettes : on peut ajouter et enlever des assiettes au sommet de la pile ; toute insertion ou retrait d'une assiette en milieu de pile La stratégie de gestion d'une pile est "dernier arrivé, premier servi". En anglais on dira Last ln First Out, plus connu sous le nom de LlFO. La structure de pile est utilisée pour sauvegarder temporairement des informations en respectant leur ordre d'entrée, et les réutiliser en ordre inverse.

2. Définitions et primitives d'accès

Une pile est une liste linéaire, telle que l'on passe d'un état de la pile au suivant par adjonction ou suppression d'un élément en tête. Une pile est en général schématisée ainsi, par la suite chronologique de ses états (figure 21).

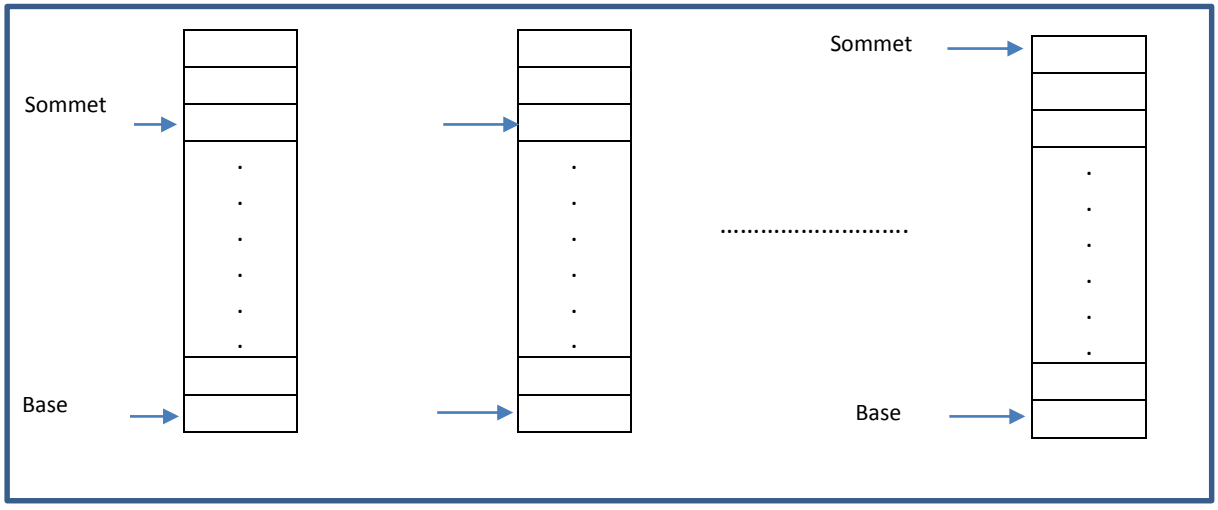


Figure 21. Successions des états d'une pile.

Le dernier élément de chaque liste est appelé base de la pile; le premier élément de chaque liste est appelé sommet de la pile. Les mises à jour de chaque liste, d'après la définition, se font à partir du sommet. Bien que théoriquement de taille infinie, une liste est toujours, dans la pratique, formée d'un nombre fini d'éléments qui correspond à la taille maximale de la pile.

Les adjonctions s'arrêtent lorsque cette taille maximale est atteinte ; on parle de débordement de la pile (ou sur-dépassement). Les suppressions s'arrêtent lorsque la pile devient vide (sous-dépassement). On définit habituellement les deux primitives suivantes:

- empiler (val) qui ajoute un nouvel élément val au sommet de la pile,
- dépiler(val) qui supprime l'élément sommet de pile et range sa valeur.

Remarques :

• En général, la variable qui permet l'accès au sommet de la pile, ainsi que la pile elle-même, sont des variables globales. On ne les passera donc pas en paramètres.

• Les valeurs des éléments d'une pile sont d'un type quelconque, que nous appellerons t.

• Exemples

Soient la pile schématisée en figure 22 (1) de et la variable val, qui contient z ; si l'on exécute l'instruction empiler(val), le nouvel état de la pile sera figure 22 (2) :

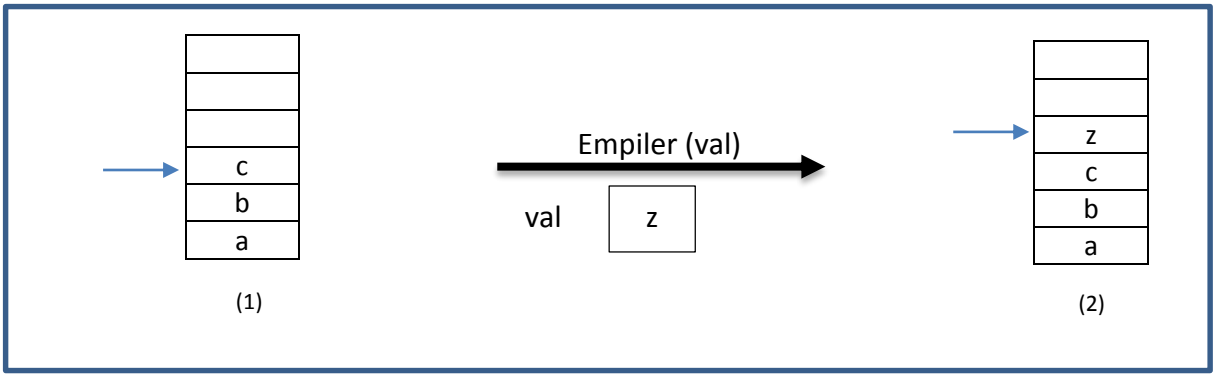


Figure 22. Opération Empiler.

si l'on exécute l'instruction dépiler(val), le nouvel état de la pile est (3), et val contient alors c :

On peut définir d'autres primitives qui facilitent l'utilisation d'une pile mais qui ne modifient pas la pile elle-même:

- une fonction sommetpile qui délivre la valeur de l'élément de sommet de la pile.

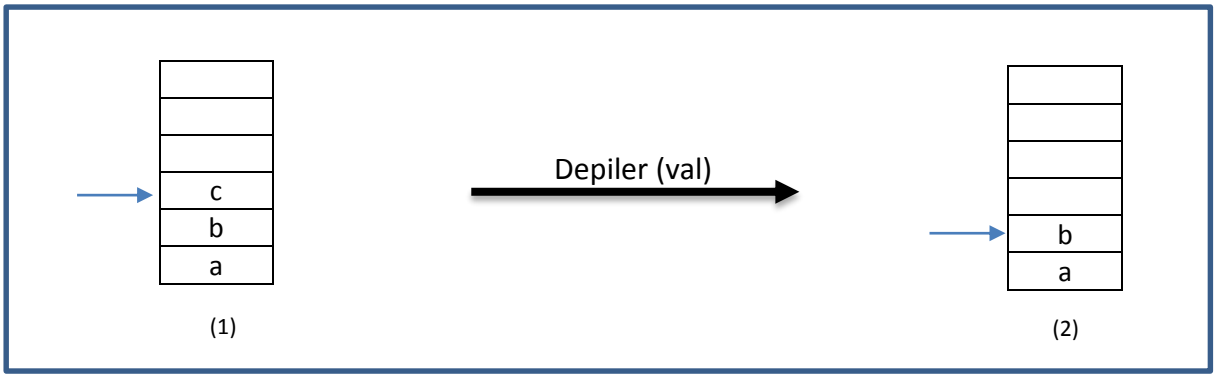


Figure 23. Opération Dépiler.

- deux prédicats qui permettent de tester éventuellement les sur- et sousdépassements de la pile: pilevide et pilepleine. Les débordements sont déterminés en comparant l'accès courant au premier élément de la pile aux deux valeurs extrêmes de cet accès qui déterminent entièrement la pile (base et valeur de l'accès pour la taille maximale).
- Enfin, une primitive initpilevide permet d'initialiser une pile.

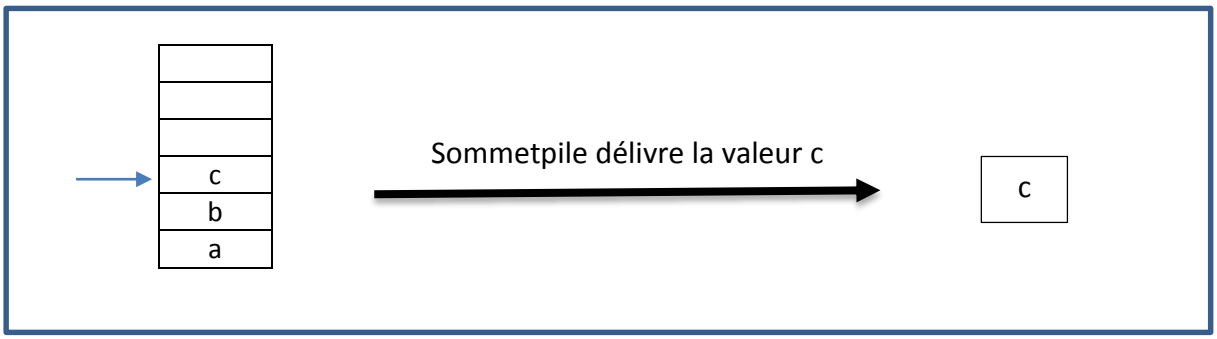


Figure 24. Opération de récupération du sommet de la pile.

Une pile est une liste linéaire, elle peut donc être représentée de manière contiguë ou chaînée. Les algorithmes proposés par la suite ne tiennent pas compte de la représentation. Notons bien que ces primitives doivent être utilisables quelle que soit la représentation interne choisie pour la pile. Seul le corps des primitives, inaccessible à l'utilisateur, dépend de cette représentation.

3. Représentation contiguë d'une pile

Soit une pile utilisant une représentation contiguë: un vecteur pile dont on donne a priori la taille maximale dimpile. Dans ce cas, l'accès au premier élément est déterminé par un indice sommet dont les valeurs extrêmes permettant de tester les dépassements sont 0 et dimpile. On

peut remarquer que la liste est représentée "à l'envers" dans le vecteur :la tête de liste correspond à l'indice le plus élevé, ou sommet, la base à l'indice 1.

Les dépassements sont testés dans les primitives dépiler et empiler (dans la pratique, ces deux tests ne sont pas toujours effectués; selon la nature du problème utilisant la pile, d'autres tests peuvent être envisagés).

Les variables pile, dimpile et sommet étant globales, les primitives sont les suivantes:

- **Primitive empiler**

```
procédure empiler (Val v: t) ;

spécification {} ===>  {v est empilé, sauf si la pile est pleine}

    debproc
      si pilepleine alors
            écritexte ('erreur: la pile est pleine') ;
            arrêt;
          sinon
            sommet ← sommet + 1;
            pile [sommet] ← v;
      finsi;
finproc;
```

- **Primitive dépiler**

```
procédure dépiler (Var v: t) ;
spécification { } ===> {v est dépilé, sauf si la pile est vide}
    debproc
        si pilevide alors
                écritexte ('erreur: la pile est vfde') ;
                arrêt ;
            sinon
                v ←pile [sommet] ;
                sommet :=sommet -1 ;
    finsi
finproc;
```

Remarques : On supposera que arrêt est une instruction qui permet d'arrêter le déroulement de l'algorithme qui a appelé ces primitives.

- **Primitive de récupération du sommet de la pile**

```
fonction sommetpile : t;
debfonc
    retour pile [sommet] ;
finfonc;
```

- **Primitive vérification si la pile est vide**

```
fonction pilevide : booléen;
debfonc
    retour sommet = 0 ;
finfonc;
```

- **Primitive vérification du débordement**

```
fonction pilepleine : booléen;
debfonc
    retour sommet = dimpile ;
finfonc;
```

- **Primitive d'initialisation de la pile**

```
procédure initpilevide ;
debproc
    sommet:=0;
finproc;
```

4. Représentation chaînée d'une pile

Une pile peut être représentée par une liste linéaire chaînée telle que les insertions ou les suppressions sont toujours effectuées en tête de liste.

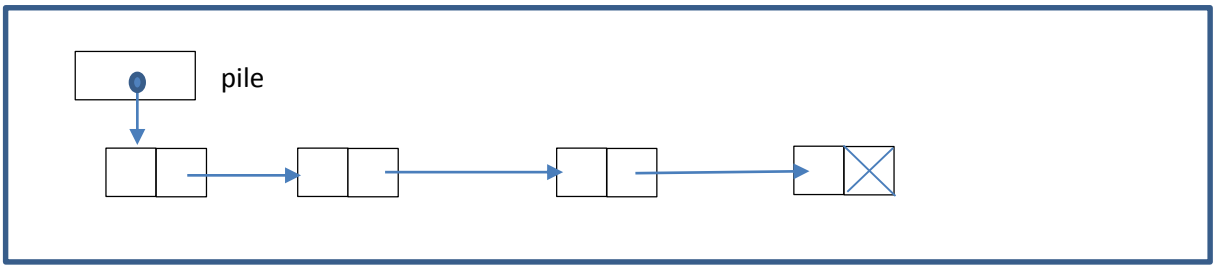


Figure 25. Représentation dynamique d'une pile.

La variable pile, de type pointeur, étant globale, les primitives deviennent:

- **Primitive empiler**

```
procédure empiler (d elem : t) ;
spécification {} ===> {insertion de elem en tête de pile}
debproc
      insertête (pile, elem) ;
finproc;
```

Notons qu'en raison de la gestion dynamique de la mémoire, la primitive pilepleine n'a plus de raison d'être, on supposera que l'instruction d'empilement est toujours possible.

- **Primitive dépiler**

```
procédure dépiler (Var elem : t) ;
spécification {} ===>  {elem = sommet de la pile, suppression en tête de l'élément, si possible}
debproc
      si pilevide alors
            écritexte ('erreur: la pile est vide') ;
            arrêt;
         sinon
            elem ←pile^.d ;
            supptête (pile) ;
      finsi;
finproc;
```

- **Primitive vérification si la pile est vide**

```
fonction pilevide : booléen;
debfonc
        retour pile = nil ;
finfonc;
```

- **Primitive d'initialisation de la pile**

```
procédure initpilevide ;
debproc
  pile ← NIL ;
finproc ;
```

Chapitre 4 : Files

1. Introduction

Les files correspondent au comportement d'une file d'attente devant un guichetoù la gestion est la suivante : premier arrivé, premier servi. En anglaison dira: First ln First Out ou plus simplement FIFO. Ces files d'attentesont d'un usage très répandu dans la programmation système, où ellesservent à gérer, par exemple, l'allocation de ressources.

Définition

Unefile d'attente est une liste linéaire telle que les :

- insertions sont toujours effectuées en fin de liste,
- lessuppressions sont toujours effectuées en tête de liste.

2. Représentation contiguë

Onpeut songer à représenter une file d'attente dans un vecteur simplement en ajoutant les nouveaux éléments à droite (queue de la file), et en déplaçant le pointeurde tête de la file vers la droite chaque fois que l'on supprime un élément.

• Exemple

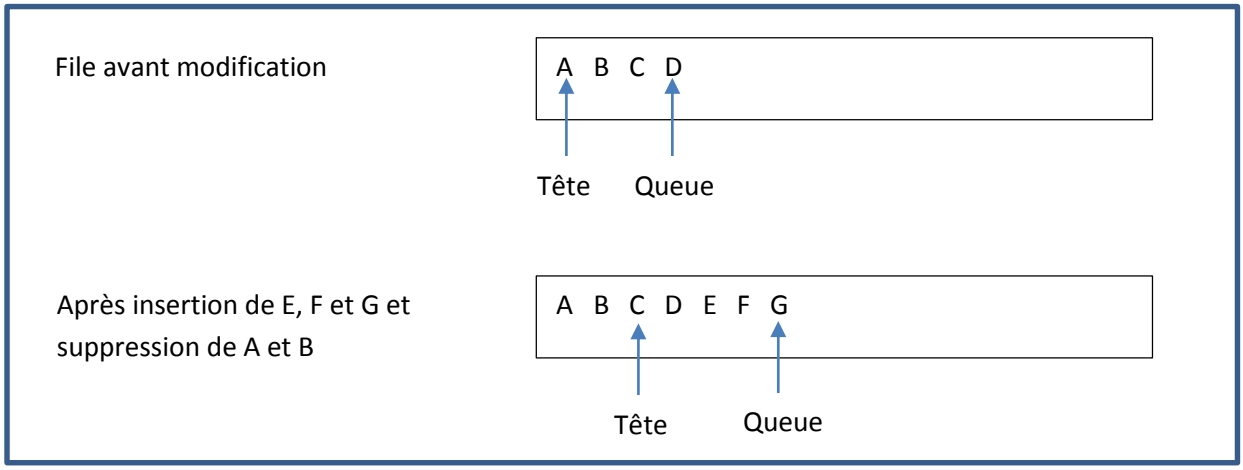


Figure 25. File d'attente.

Ensupposant que le vecteur FILE [1..n], ainsi que les entiers tête et queue, sont des variables globales, les primitives de gestion de la file seraient les suivantes:

- **Primitive d'initialisation**

```
procédure initfile ;
debproc
    tête←1;
    queue←0;
finproc;
```

- **Primitive de vérification si la file est vide**

```
fonction filevide : booléen;
    debfonc
    retour tête > queue;
finfonc;
```

- **Primitive de vérification si la file est pleine**

```
fonction filepleine : booléen;
    debfonc
    retour queue = n;
finfonc;
```

- **Primitive Enfiler (ajout en queue de la file)**

```
procédure enfiler (Val v: t) ;
debproc
si non filepleine alors
queue ←queue + 1;
FILE [queue] ←v;
finsi;
finproc;
```

- **Primitive Défiler**

```
procédure défiler (Var v: t) ;
debproc
    si non filevide alors
        v := FILE [tête] ;
        tête := tête + 1 ;
    fin si ;
finproc;
```

On voit que tête et queue augmentent toujours, et donc que le vecteur devra être très grand, même s'il ne contient que peu d'éléments à chaque instant. Pour éviter cet inconvénient, on peut envisager deux solutions.

Décaler les éléments de la file vers la gauche à chaque suppression

Tous les éléments avancent d'une place lorsque le premier est servi. Le pointeur de tête devient alors inutile, puisqu'il est toujours égal à 1.

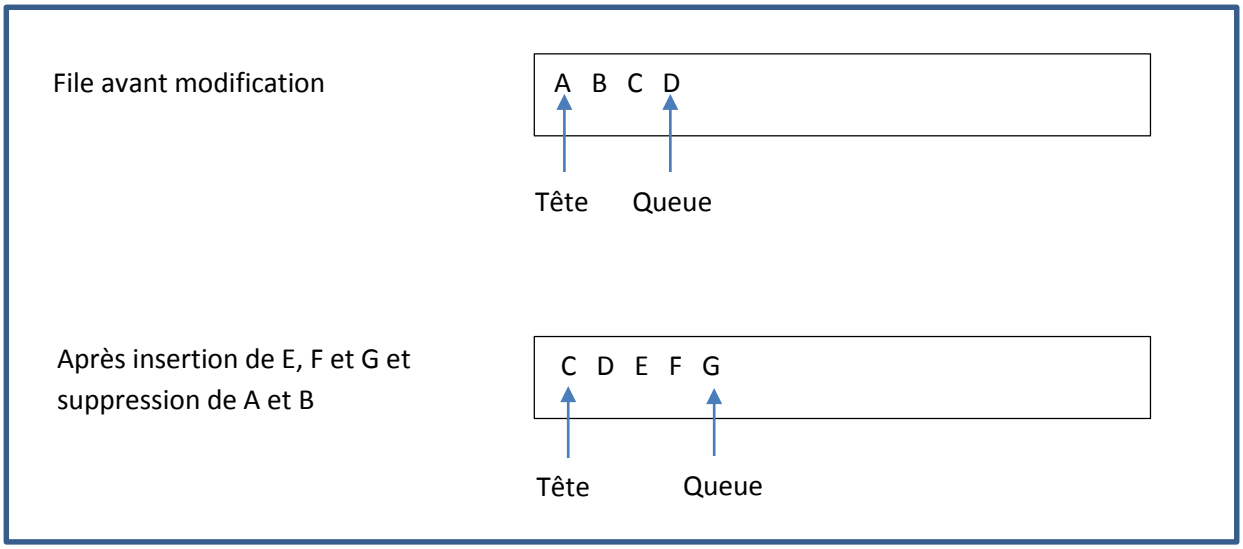


Figure 26. Décalage dans une file d'attente.

Les primitives modifiées sont les suivantes:

- **Primitive d'initialisation**

```
procédure initfile ;
debproc
      queue←0;
finproc;
```

- **Primitive de vérification si la file est vide**

```
fonction filevide : booléen;
debfonc
    retour queue = 0 ;
finfonc;
```

- **Primitive Défiler.**

```
procédure Défiler (Var v: t) ;
      i: entier;
      debproc
          si non filevide alors
                v ←FILE [1] ;
                i←1;
                tantque (i < queue) faire
                      FILE [i] := FILE[i+1] ;
                      i←i + 1;
                finfaire;
                queue ←queue -1 ;
          finsi;
finproc;
```

La taille du vecteur doit correspondre au nombre maximum d'éléments qui peuvent être simultanément présents dans la file d'attente.

- **Gestion circulaire de la file.**

C'est la meilleure solution, elle minimise la place nécessaire dans levecteur comme dans la solution précédente, tout en évitant les opérationsde décalage. Exemple : On suppose ici n=6 :

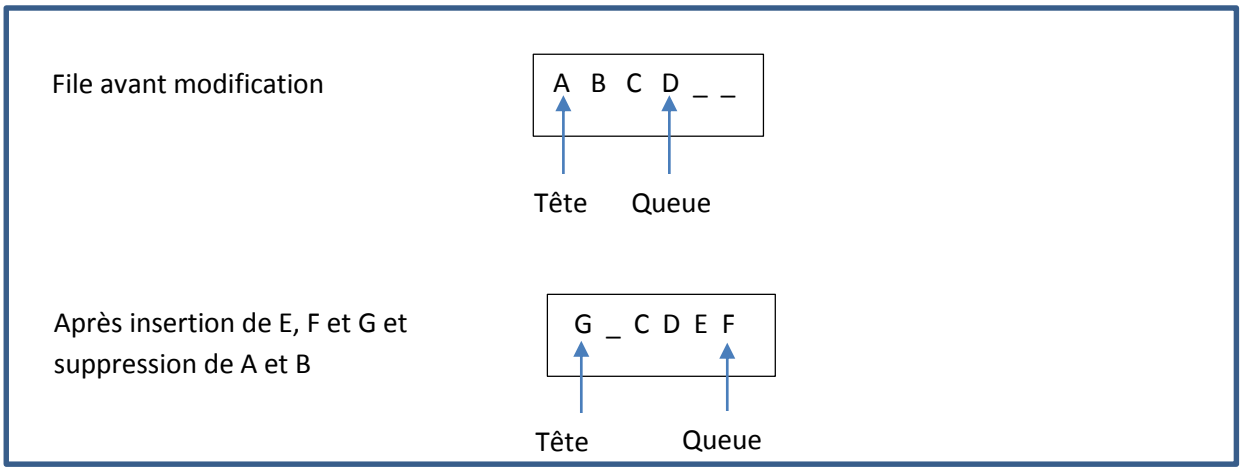


Figure 27. Gestion circulaire de la file.

Lorsque tête devient égal à (queue + 1) mod n, la file est soit vide, soit pleine. On est donc amené à distinguer les deux possibilités en utilisant deuxindicateurs booléens filevide et filepleine. Les primitives deviennent alors :

- **Primitive Enfiler (ajout en queue de la file)**

```
procédure enfiler (Val v: t) ;
debproc
si non filepleine alors
        si (queue < n) alors    queue ←queue + 1;
                sinon
                queue←1;  FILE [queue] := v;
                filevide ←faux;
                filepleine ← ((queue < n) et (tête = queue + 1))  ou ((queue = n) et (tête= 1))
        finsi ;
finsi ;
Finproc ;
```

- **Primitive d'initialisation de la file**

```
procédure initfile ;
    debproc
        tête := 1 ;
        queue:= 0;
        filevide := vrai;
        filepleine := faux;
finproc;
```

- **Primitive Défiler.**

```
procédure Défiler (Var v: t) ;
        debproc
            si non filevide alors
                v ←FILE [tête] ;
                si (tête < n )alors
                    tête ←tête + 1 ;
                    sinon
                    tête ←1;
                    filepleine ←faux;
                    filevide ← ((queue <n) et (tête = queue + 1))
                                                    ou ((queue = n) et (tête = 1))
                finsi;
            finsi;
finproc;
```

3. Représentation chaînée d'une file

C'est une représentation classique de liste chaînée, dans laquelle on utilise un pointeur supplémentaire: le pointeur sur la dernière cellule, ce qui permet d'accélérer l'algorithme d'insertion qui a toujours lieu en fin de liste.

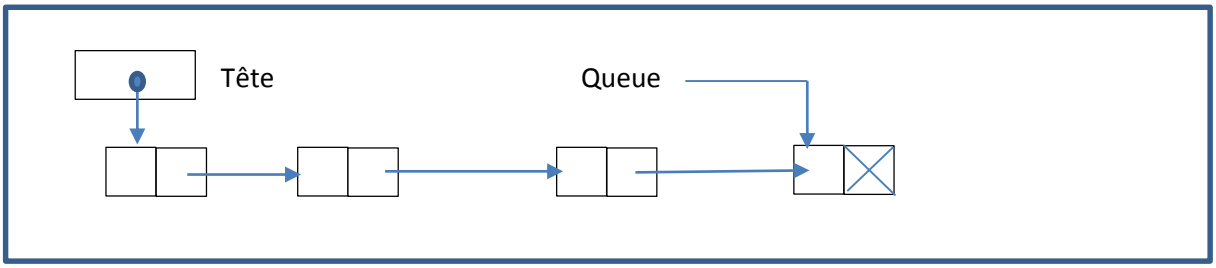


Figure 28. Représentation dynamique d'une file.

Les algorithmes d'insertion et de suppression sont les suivants (tête et queue sont des variables globales) :

- **Primitive Enfiler (ajout en queue de la file)**

```
procédure enfiler (Val v: t)
        debproc
        si (tête = nil) alors
                {tête = nil, queue = nil }
                insertête (tête, v) ;
                queue ←tête i ;
            sinon
                insertête (queue^..suivant, v) ;
                queue := queue^.suivant ;
        finsi ;
finproc ;
```

- **Primitive défiler (ajout en queue de la file)**

```
Procedure défiler (Var v : t)
        debproc
            si (tête #  nil) alors val ← tête i .info ;  supptête (tête) ;
                                        si (tête = nil) alors queue := NIL ;
                finsi ;
            finsi ;
finproc ;
```

Chapitre 5 : Arbre

1. Définition

Un arbre est une structure dynamique d'éléments organisés de façon hiérarchique (père, fils)et définie en fonction des nœuds qui la composent. Un nœud est l'élément élémentaire de l'arbre contenant des données et des pointeurs.

2. Exemple d'utilisation

Soit la figure suivante :

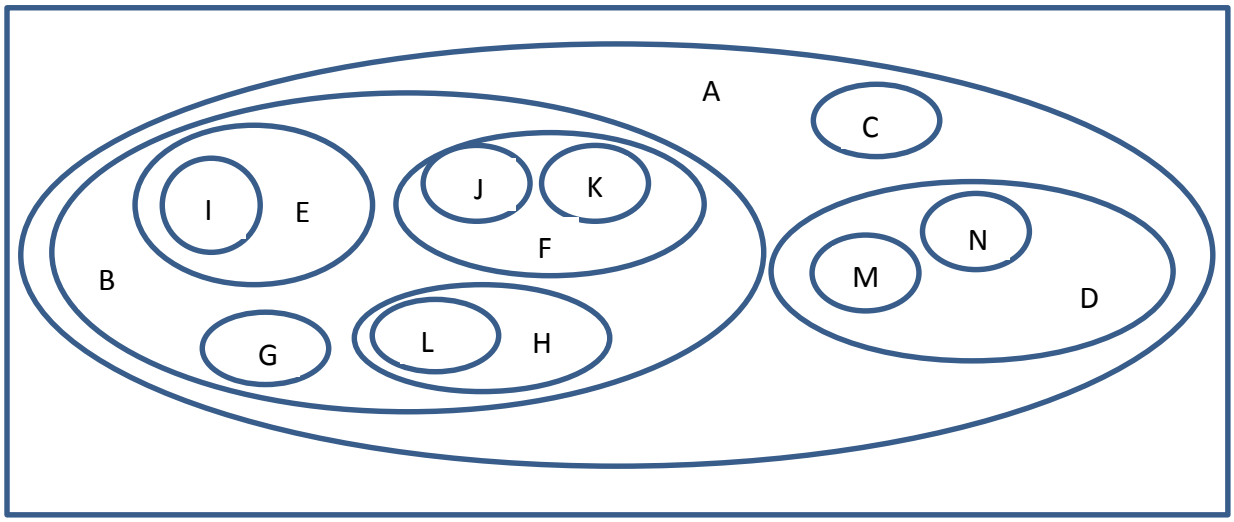


Figure 29. Relation d'inclusion d'ensembles.

La relation d'inclusion entre les ensembles de la figure 29 peut être représentée à l'aide de l'arbre de la figure 29.

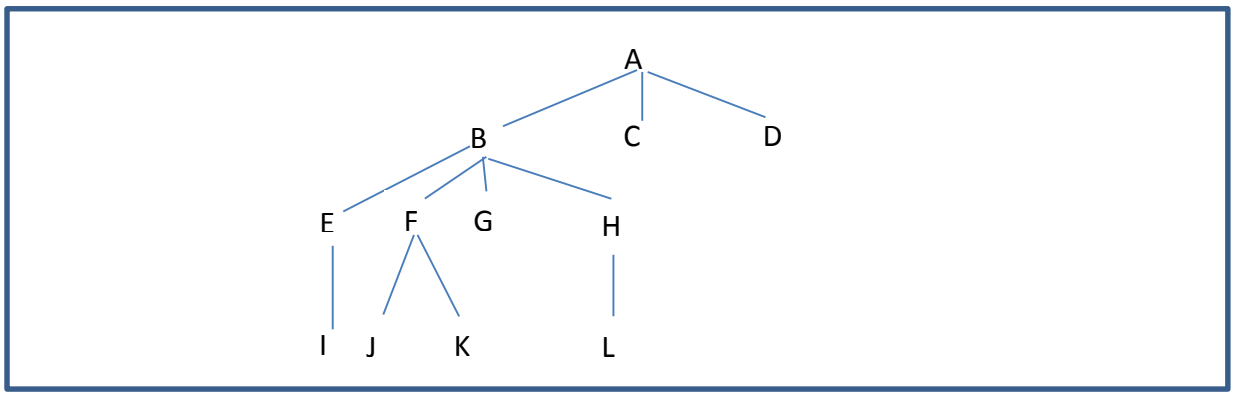


Figure 30. Arbre représentant la relation d'inclusion d'ensembles.

Un type structuré dans tout langage de programmation peut être réprésenté sous forme d'arbre.

```
Type tdate = structure
    Jour,mois, an : entier ;
Fin
Type tidentite = structure
    Nom, prenom : chaine 30 ;
        Date_naissance :tdate ;
        Lieu_naissance : tlieu ;
fin
Type tlieu = structure
        Ville  : chaine 30 ;
        Département : chaine 20 ;
Fin
Type  etablissement  = structure
        Universite :chaine 30 ;
        Lieu_etude :tlieu ;
fin
Type etudiant = structure
        Inscription : tetablissement ;
        identite : tidentite ;
fin
```

Le type étudiant peut être représenté à l'aide de l'arbre :

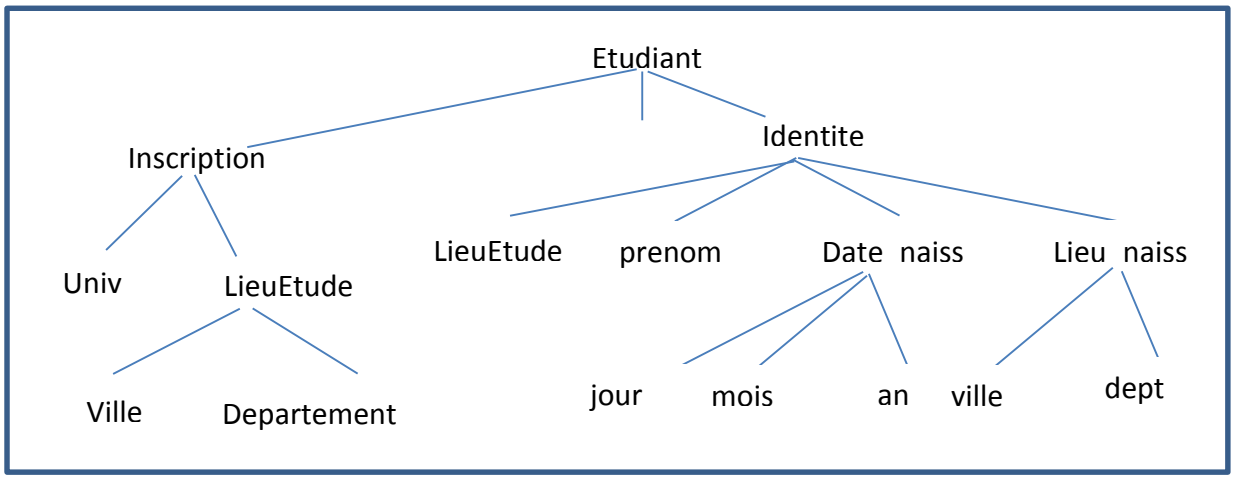


Figure 31. Arbre représentant le typage.

3. Notations et terminologie

On a remarqué que les schémas récursifs simplifiaient beaucoup l'écriture d'algorithmes sur les listes chaînées. On utilisait, pour ce faire, la définition récursive suivante d'une liste: une liste est

• soit vide,

• soit constituée d'un élément chaîné à une liste.

Il en est de même pour les arborescences, ou arbres, où on peut donner la définition suivante : un arbre est

• soit vide,

• soit constitué d'un élément auquel sont chaînés un ou plusieurs arbres.

- **Terminologie**

Dans un arbre nous employons le jargon suivant :

- Le premier nœud de l'arbre est la racine.
- Un fils directement en dessous d'un autre nœud est appelé fils ou descendant.
- Un nœud sans descendant est une feuille.
- Le degré d'un nœud est le nombre de sous-arbres de ce nœud (nombre de descendants).
- Le niveau de la racine est 1.
- Le niveau du descendant d'un nœud est le niveau de ce nœud +1.

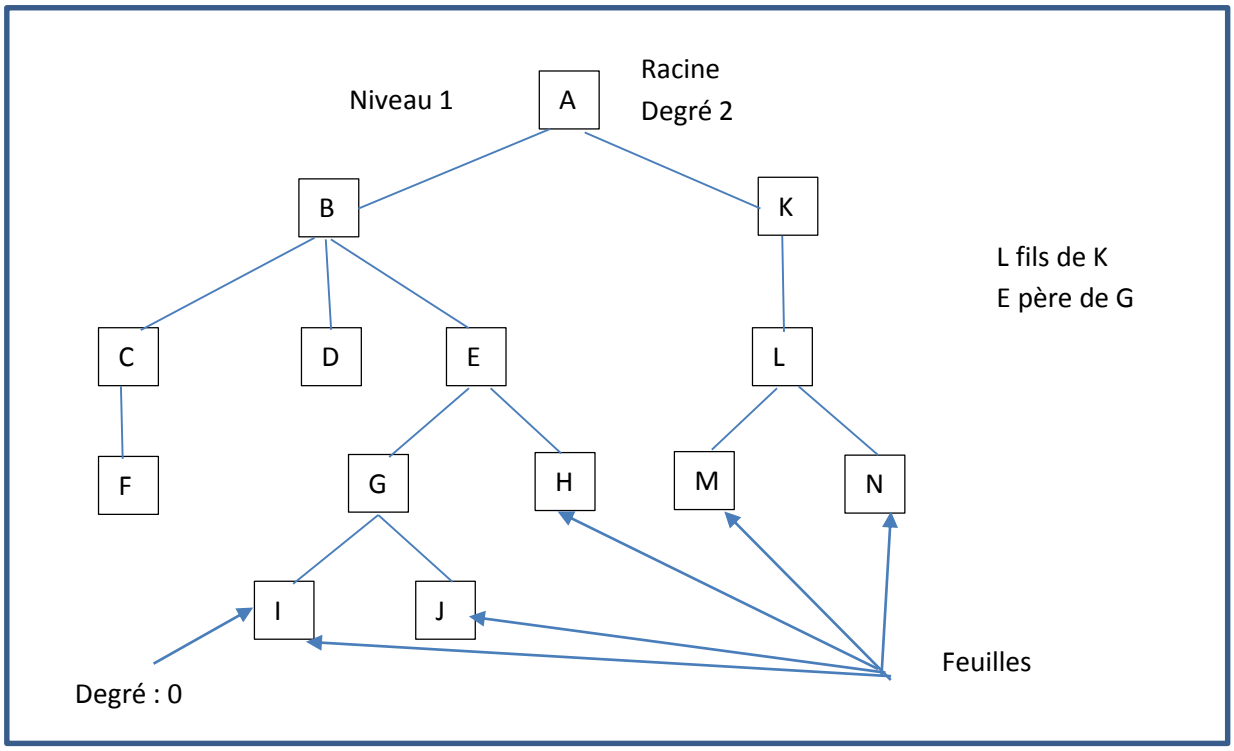


Figure 32. Terminologie.

4. Arbre n-aire et arbre binaire

Lorsqu'un arbre admet, pour chaque nœud, au plus n fils, l'arbre est dit n-aire. Si n est égale à 2, l'arbre est dit binaire. Nous nous interessons à ce deuxième type d'arbre. Dans cet arbre, nous disposons pour chaque nœud d'au plus deux descendants nommés fils gauche et fils droit (ainsi que de sous arbre gauche et sous arbre droit). Les algorithmes concernant les arbres binaires sont souvent écrits sous forme récursif.

- **Représentation d'un arbre binaire**

Nous définissons deux types :

```
Type    pointeur =^Nœud
        Nœud = enregistrement
                Gauche : pointeur ;
                Droit : pointeur ;
                Info : type_donnée ;
        Fin ;
```

Exemple :

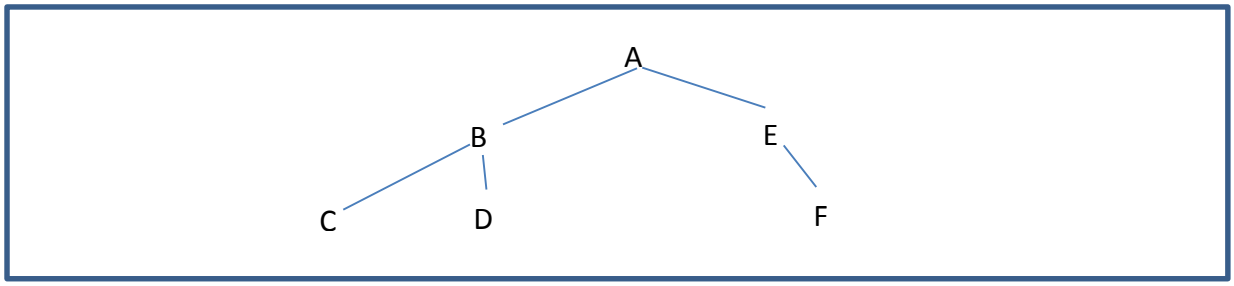


Figure 33. Exemple d'un arbre.

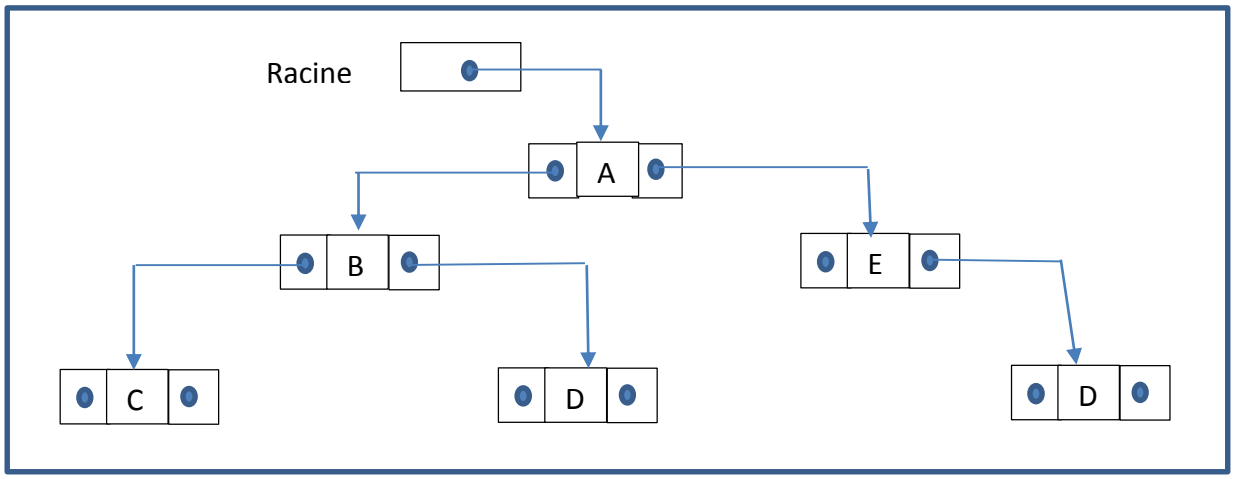


Figure 34. Représentation de l'arbre selon le typage adopté.

5. Arbre n-aire et arbre binaire

Le problème de parcours d'un arbre est analogue à celui du parcours d'une liste chaînée. Ces algorithmes sont d'une grande importance car ils servent de base à l'écriture de très nombreux algorithmes concernant les arbres. Il y a plusieurs manières pour parcourir un arbre binaire suivant l'ordre dans lequel on énumère un nœud et ses sous arbres gauche et droit. Nous nous interessons à trois types de parcours : préfixé, infixé et postfixé.

Exemple adopté :

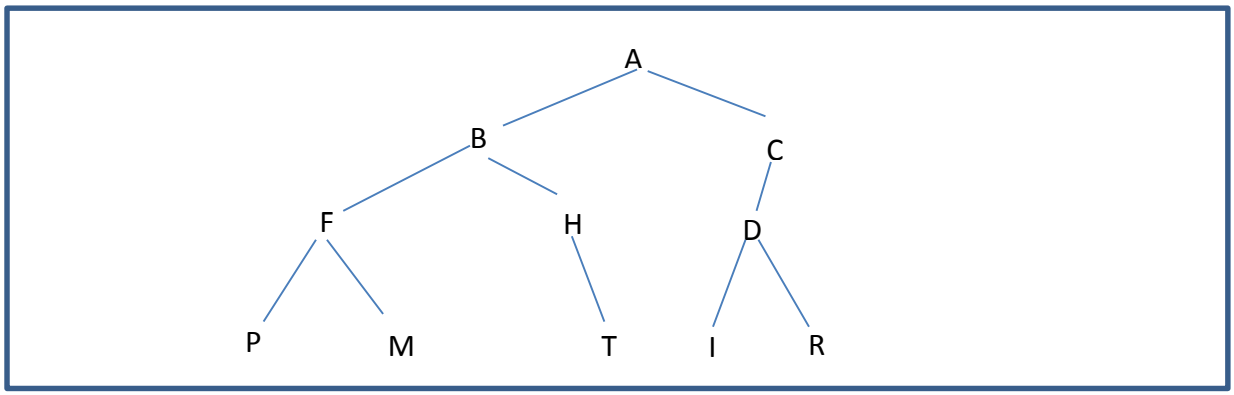


Figure 35. Arbre adopté pour le parcours.

a. Parcours préfixé

Ce parcours s'effectue dans l'ordre (ou en préordre) selon l'ordre suivant :

> Le traitement de la racine.
>
> Le parcours du sous arbre gauche.
>
> Le parcours du sous arbre droit

Le parcours préfixé de l'arbre de la figure 35 est le suivant :
A, B, F, P, M, H, T, C, D, I, R, E.

```
Procédure prefixe (Val racine : pointeur)
debproc
        si (racine # NIL) alors
                traiter (racine) ;
                prefixe (racine^.gauche) ;
                prefixe (racine^.droit) ;
finproc
```

b. Parcours infixe (projectif ou symétrique)

Ce parcours s'effectue dans l'ordre suivant :

> Le parcours du sous arbre gauche.
>
> Le traitement de la racine.
>
> Le parcours du sous arbre droit

Le parcours infixe de l'arbre de la figure 35 est le suivant :

P, F, M, B, H, T, A, I, D, R, C, E.

```
Procédure infixe (Val racine : pointeur)
debproc
        si (racine # NIL) alors
                infixe (racine^.gauche) ;
                traiter (racine) ;
                infixe (racine^.droit) ;
finproc
```

c. Parcours postfixé (ou ordre terminal)

Ce parcours s'effectue dans l'ordre suivant :

> Le parcours du sous arbre gauche.
>
> Le parcours du sous arbre droit
>
> Le traitement de la racine.

Le parcours postfixé de l'arbre de la figure 35 est le suivant :

P, F, M, B, H, T, A, I, D, R, C, E.

```
Procédure postefixe (Val racine : pointeur)
debproc
        si (racine # NIL) alors
                postefixe (racine^.gauche) ;
                postefixe (racine^.droit) ;
                traiter (racine) ;
finproc
```

7. Algorithme sur les arbres binaires

a. calcul de la taille d'un arbre

Nous souhaitons écrire une fonction permettant de prendre un arbre en entrée et de retourner le nombre de ses nœuds.

Fonction taille (val racine : pointeur) : entier

Specification {} ===> {résultat = nombre de nœuds de racine+}

Si racine+ est vide alors le nombre de nœuds est 0.

Si racine+ est non vide alors cet arbre est formé d'un élément, d'un sous arbre gauche et d'un sous arbre droit.

La taille de l'arbre est égale à 1 + la taille du sous arbre droit + la taille du sous arbre gauche.

```
Fonction taille  (Val racine : pointeur) : entier
debproc
        si (racine = NIL) alors retourner 0 ;
            sinon retourner (1+taille(racine^.droit) +taille(racine^.gauche))
        finsi
finproc
```

b. Nombre de feuilles d'un arbre

Nous réalisons la fonction qui détermine si un nœud est feuille en premier lieu.

Fonction taille (val racine : pointeur) : booleen

Specification {nœud #NIL} ===> {résultat : nœud est une feuille}

```
Fonction feuille  (Val noeud : pointeur) : booleen
debproc
        retourner ((noeud^.droit=NIL)  et  (noeud^.gauche=NIL))
        finsi
finproc
```

Pour la fonction nombre de feuille :

si racine+ est vide, le résultat est 0

si racine+ est non vide

si racine est feuille, le résultat est 1

sinon le résultat est le nombre de feuilles du sous arbre gauche + le nombre de feuilles du sous arbre droit.

```
Fonction nbfeuille  (Val racine : pointeur) : entier
debproc
        si (racine = NIL) alors retourner 0 ;
            sinon
                    si feuille(racine) alors
                            retourner (nbfeuille (racine^.droit) + nbfeuille
                           (racine^.gauche)) ;
        finsi
finproc
```

c. Recherche associative dans un arbre binaire

Nous voulons savoir si la valeur v est présente dans racine+

Fonction Recherche (val racine : pointeur, Val v :t) : booleen

Specification { } ===> {résultat : v appartient à racine +}

Si racine+ est vide, le résultat est faux.

Si racine+ n'est pas vide

Si racine^.info=v, le résultat est vrai

Si racine^.info #v

Effectuer la recherche dans le sous arbre droit.

Si le resultat est faux, Effectuer la recherche dans le sous arbre gauche.

```
Fonction Recherche  (Val racine : pointeur, Val v : t) : booleen
debproc
        si (racine = NIL) alors retourner faux ;
           sinon
                   si Recherche (racine^.gauche, v) alors retourner (vrai)
                           sinon retourner (Recherche (racine^.droit, v)
                   finsi
        finsi
finproc
```

D'autres procédures et fonctions qui seront discutés lors du cours pour illustrer leur raisonnement sont données dans la partie programmes C des structures données fournie comme complément en fin de ce cours.

Programmes C des algorithmes manipulant les structures de données.

1. Programmes sur les listes chaînées

Structure de la cellule

```
typedef struct cel{
   int data;
   struct cel *next;
}cel;
typedef cel* cellule;
```

Créer une cellule

```
cellule createCellule(cellule next, int x){
   cellule k = (cellule)malloc(sizeof(cellule));
   k->data = x;
   k->next = next;
   return k;
```

Calculer La taille d'une liste

```
int taille(cellule tete){
   if(tete == NULL)
      return 0;
   return ( 1 + taille(tete->next) );
}
```

Insérer au début d'une liste

```
void insertDebut(cellule *tete, int x){
   if(*tete == NULL)
      *tete = createCellule(NULL, x);
   else
      *tete = createCellule(*tete, x);
}
```

Inserer en fin de liste (itératif)

```
void insertFinIterative(cellule *tete, int x){
   if(*tete == NULL){
      insertDebut(tete, x);
      return;
   }
   cellule tmp = *tete;
   while(tmp->next != NULL)
      tmp = tmp->next;
   tmp->next = createCellule(NULL, x);
}
```

Inserer en fin de liste (itératif)

```
void insertFinIterative(cellule *tete, int x){
   if(*tete == NULL){
      insertDebut(tete, x);
      return;
   }
   cellule tmp = *tete;
   while(tmp->next != NULL)
      tmp = tmp->next;
   tmp->next = createCellule(NULL, x);
}
```

Insérer en fin de liste (récursif)

```
void insertFinRecursive(cellule *tete, int x){
   if(*tete == NULL || (*tete)->next == NULL){
      insertDebut(&(*tete)->next, x);
      return;
   }
   insertFinRecursive(&(*tete)->next, x);
}
```

Insérer à un rang de la liste (récursif)

```
void insertRangRecursive(cellule *tete, int x,
int rang){
   if(rang <=0 || rang > taille(*tete)+1 ){
      return;
   }
   if(rang == 1){
      insertDebut(tete ,x);
      return;
   }
   if(rang == 2){
      insertDebut(&(*tete)->next, x);
      return;
   }
   insertRangRecursive(&(*tete)->next, x,
rang-1);
}
```

Insérer à un rang de la liste (itératif)

```
void insertRangIterative(cellule *tete, int x, int rang){
   if(rang <= 0 && rang > taille(*tete)+1){
      printf("%d : Rang invalider.\n",rang);
      return;
   }
   if(rang == 1){
      insertDebut(tete, x);
      return;
   }
   cellule tmp = *tete;
   while(rang != 2){
      tmp = tmp->next;
      rang--;
   }
   tmp->next = createCellule(tmp->next, x);
}
```

Insérer en respectant l'ordre (itératif)

```
void insertOrdreIterative(cellule *tete, int x){
   if(*tete == NULL || x < (*tete)->data){
      insertDebut(tete ,x);
      return;
   }
   cellule tmp = *tete;
   cellule prev = *tete;
   while(tmp != NULL && x > tmp->data){
      prev = tmp;
      tmp = tmp->next;
   }
   prev->next = createCellule(prev->next, x);
}
```

Insérer en respectant l'ordre (itératif)

```
void insertOrdreRecursive(cellule *tete, int x){
   if(*tete == NULL || x < (*tete)->data){
      insertDebut(tete, x);
      return;
   }
   insertOrdreRecursive(&(*tete)->next, x);
}
```

Supprimer en fin de la liste (itératif)

```
void supprFinIterative(cellule *tete){
   cellule tmp = *tete;
   cellule prev = *tete;
   while(tmp->next != NULL){
      prev = tmp;
      tmp = tmp->next;
   }
   free(tmp);
   prev->next = NULL;
}
```

Supprimer le début de la liste

```
void supprDebut(cellule *tete){
   cellule tmp = *tete;
   *tete = tmp->next;
   free(tmp);
}
```

Supprimer en fin de la liste (récursif)

```
void supprFinRecursive(cellule *tete){
   if(*tete == NULL){
      return;
   }
   if((*tete)->next == NULL){
      cellule tmp = *tete;
      *tete = NULL;
      free(tmp);
      return;
   }
   supprFinRecursive(&(*tete)->next);
 }
```

Supprimer à un rang k de la liste (itératif)

```
void supprRangIterative(cellule *tete, int rang){
   if(rang <= 0 || rang > taille(*tete)){
      printf("%d : Invalide rang.\n",rang);
   }
   else if(rang == 1){
      supprDebut(tete);
      return;
   } ……..
```

Supprimer à un rang k de la liste (itératif)

```
…….
   else{
      cellule tmp = *tete;
      cellule tmp2 = NULL;
      while(tmp != NULL && rang != 2){    //parcourir jusqu'a rang-1 (la cellule precedente)
         tmp = tmp->next;
         rang--;
      }
      tmp2 = tmp->next;   //tmp2 contient la cellule qu'on veut supprimer
      tmp->next = tmp2->next; //  lier la cellule rang-1 avec rang+1
      printf("la cellule dont la valeur est %d est supprim%ce",(*tete)->data,130);
      free(tmp2);
   }
```

Supprimer à un rang k de la liste (Recursif)

```
void supprRangRecursive(cellule *tete, int rang){
   if(*tete == NULL || rang <= 0 || rang > taille(*tete)){
      return;
   }
   if(rang == 1){
      supprDebut(tete);
      return;
   }
   supprRangRecursive(&(*tete)->next, rang-1);
}
```

Supprimer la première occurrence d'une valeur dans la liste (itératif)

```
void supprPoccurenceIterative(cellule *tete ,int x){
   cellule prev = NULL;
   cellule tmp = *tete;
   while(tmp != NULL && x != tmp->data){
      prev = tmp;
      tmp = tmp->next;
   }
   if(tmp == NULL){    //s'elle ne trouve pas x
      printf("%d n'existe pas\n",x);
      getch();
      return;
   }
   supprDebut(&(prev)->next);
   printf("%d est supprim%ce\n",x,130);
   getch();
}
```

Supprimer la première occurrence d'une valeur dans la liste (récursif)

```
   getch();
} void supprPoccurenceRecursive(cellule *tete, int x){
   if(*tete == NULL){
      return;
   }
   if((*tete)->data == x){
      supprDebut(tete);
      return;
   }
   supprPoccurenceRecursive(&(*tete)->next, x);
}
```

Supprimer toutes les occurrences d'une valeur dans la liste (itératif)

```
void supprToutesOccurencesIterative(cellule *tete ,int x){
   int cpt = 0;
   cellule tmp = *tete;
   cellule prev = NULL;
   while(*tete != NULL && tmp->data == x){
      supprDebut(tete);
      cpt++;
   }
   tmp = (*tete)->next;
   prev = *tete;
   while(tmp != NULL){
      if(tmp->data == x){
         prev->next = tmp->next;
         free(tmp);
         tmp = prev->next;
      cpt++;
      }
      else{
         prev = tmp;
         tmp = tmp->next;
      }
   }
   printf("%d occurrences sont supprim%ces.\n",cpt,130);
   getch();
}
```

Supprimer toutes les occurrences d'une valeur dans la liste (récursif)

```
void supprToutesOccurencesRecursive(cellule *tete, int x){
   if(*tete == NULL){
      return;
   }
   if((*tete)->data == x){
      supprDebut(tete);
      supprToutesOccurencesRecursive(tete, x);
   }
   else
      supprToutesOccurencesRecursive(&(*tete)->next, x);
}
```

Supprimer les k occurrences d'une valeur dans la liste (itératif)

```
void supprKOccurencesIterative(cellule *tete, int x, int k){
   cellule tmp = *tete;
   cellule prev = NULL;
   while(*tete != NULL && tmp->data == x && k != 0){
      supprDebut(tete);   //l'incrémentation est fait dans la fonction supprDebut()
      k--;
   }
   if(*tete == NULL){      //si toutes les cellules(->data) sont egals a x
      printf("%d occurences sont supprim%ces.\n",k,130);
      getch();
      return;
   }
   tmp = (*tete)->next;
   prev = *tete;
 ……
```

```
    while(tmp != NULL && k != 0){
        if(tmp->data == x){
            prev->next = tmp->next;
            free(tmp);
            tmp = prev->next;
            k--;
        }
        else{
            prev = tmp;
            tmp = tmp->next;
        }
    }
    printf("%d occurences sont supprim%ces.\n",k,130);
    getch();
}
```

Supprimer les k occurrences d'une valeur dans la liste (récursif)

```
void supprKOccurencesRecursive(cellule *tete, int x, int k){
    if(*tete == NULL || k <= 0){
        return;
    }
   if((*tete)->data == x && k > 0){
        supprDebut(tete);
        supprKOccurencesRecursive(tete, x, k-1);
    }
    else
        supprKOccurencesRecursive(&(*tete)->next, x, k);
}
```

Supprimer la k ieme occurrence d'une valeur dans la liste (itératif)

```
void supprKemeOccurencesIterative(cellule *tete, int x, int keme){
   if(keme <= 0 || keme > taille(*tete)){ // max(k) = taille(Liste)
      printf("%d : invalide valeur\n",keme);
      getch();
      return;
   }
   if(keme == 1 && (*tete)->data == x){
      supprPoccurenceRecursive(tete, x);          //premiere occurence
      return;
   }
   cellule prev = *tete;
   cellule tmp = prev;
   while(tmp != NULL && keme > 0){
      if(tmp->data == x){
         keme--;
      }
      if(keme == 0)
         break;
      prev = tmp;
      tmp = tmp->next;
   }
   if(tmp == NULL){
      printf("L'occurence %d de %d n'existe pas.\n",keme,x);
      getch();
      return;
   }
```

Supprimer la k ieme occurrence d'une valeur dans la liste (récursif)

```
void supprKemeOccurencesRecursive(cellule *tete, int x, int keme){
   if(*tete == NULL || keme <= 0 || keme > taille(*tete)){
      return;
   }
   if((*tete)->data == x && keme == 1)
      supprDebut(tete);
   else if((*tete)->data == x)
      supprKemeOccurencesRecursive(&(*tete)->next, x ,keme-1);
   else
      supprKemeOccurencesRecursive(&(*tete)->next, x ,keme);
}
```

Supprimer la k ieme élément distinct d'une valeur dans la liste (itératif)

```
void supprKemeDistinctIterative(cellule* tete, int k){
   //  la premiere cellule
   if(nombreOccurence(*tete, (*tete)->data) == 1){
      k--;
      if(k == 0){
         supprDebut(tete);
         return;
      }
   }
   cellule prev = *tete;
   cellule tmp = prev->next;
   while(tmp != NULL){
      if(nombreOccurence(*tete,tmp->data) == 1)
         k--;
      if(k > 0){
         prev = tmp;
         tmp = tmp->next;
         }else
            break;
   } …….
```

```
    if(tmp == NULL){
        printf("%d nexiste pas\n");
        return;
    }
    supprDebut(&prev->next);
}
```

Supprimer la k ieme élément distinct d'une valeur dans la liste (récursif)

```
void supprKemeDistinctRecursive(cellule *tete, int k){
    if(*tete == NULL){
        return;
    }
    if((*tete)->data == val){
        supprK(&(*tete)->next, k , val);
    }
    else if((*tete)->next == NULL || (*tete)->data != (*tete)->next->data){
        //  S'elle est distincte
        k--;
        if(k==0)
            supprDebut(tete);
        else
            supprK(&(*tete)->next, k, (*tete)->data);
    }
    else
        supprK(&(*tete)->next, k, (*tete)->data);
}
```

2. Programmes sur les piles

- Représentation statique

Structure d'une pile statique

```
#define MAX 30
 struct{
        int sommet;
        int tab[max] ;
} pile;
```

Vérifier si la Pile est vide

```
int vide(pile p){
        if(p.sommet == -1)
                return 1;
        return 0;
}
```

Vérifier si la Pile est plein

```
int plein(pile p){
        if(p.sommet == MAX - 1)
                return 1;
        return 0;
}
```

Dépiler un élément(POP)

```
void depiler(pile *p){
        if(videS(*p)){
                printf("La pile est vide !");
        }
        else{
                int k = p->tab[p->sommet];
                p->sommet--;
                printf("\n%d est d%cpiler\n",k,130);
        }
```

Enpiler un élément dans la Pile(PUSH)

```
void enpiler(pile *p , int x){
        if(pleinS(*p)){
                printf("La pile est plein !");
                getch();
        }
                p->sommet++;
                p->tab[p->sommet] = x;
                printf("\n%d est empiler !",x);
        }
}
```

Vider la Pile est vide

```
void RAZ(pile *p){
        p->sommet = -1;
}
int sommet(pile *p){
   return p->tab[p->sommet];
}
```

- Représentation dynamique

Structure d'une pile dynamique

```
typedef struct cel{
   int data;
   struct cel *next;
}cellule;
typedef cellule* pile;
```

Vider la Pile est vide

```
int vide(pile p){
        if(p == NULL)
        return 1;
        return 0 ;
}
```

Enpiler un élément dans la Pile(PUSH)

```
void enpiler(pile *p, int x){
        pile nouveau =
(pile)malloc(sizeof(pile));
        nouveau->data = x;
        nouveau->next = *p;
        *p = nouveau;
        printf("L'op%cration est bien
effectu%ce",130,130);
}
```

Dépiler un élément (POP)

```
void depiler(pile *p){
        if(vide(*p)){
printf("La pile est vide !");
        return;
        }
        pile tmp = *p;
        *p = (*p)->next;
        free(tmp);
printf("L'op%cration est bien effectu%ce",130,130);
}
```

Vider la Pile

```
void RAZ(pile *p){
        //On peut faire tout simplement p == NULL
        pile tmp;
        while(*p != NULL){
                tmp = *p;
                *p= (*p)->next;
                free(tmp);
        }
}
```

3. Programmes sur les files

- Représentation statique

Structure d'une file statique

```
#define   MAX   30
typedef struct{
   int tete,queue;
   int tab[MAX];
}file;
```

Vérifier si la file est vide

```
int vide(file f){
        if(f.tete == -1 && f.queue == -1)
                return 1;
        return 0;
}
```

Vérifier si la file est pleine

```
int plein(file f){
        if((f.queue + 1)%MAX == f.tete)
                return 1;
        return 0;
}
```

Vider la file

```
void RAZ(file* f){
        f->tete = -1;
        f->queue = -1;
}
```

Afficher le contenu de la tête de la file

```
int tete(file f){
   return f.tab[f.tete];
}
```

Enfiler un élément à la file

```
void enfiler(file *f){
   int x1;
   printf("Donnez un nombre >> ");
   scanf("%d",&x1);
        if(vide(*f)){
                f->tete = 0;
                f->queue = 0;
        }
        else if(plein(*f)){
                printf("\nLa file est plein !\n");
                return;
        }
        else
                f->queue = (f->queue + 1)%MAX;
        f->tab[f->queue] = x1;
        printf("\n%d est enfil%c.\n",x1,130);
}
```

Défiler un élément de la file

```
void defiler(file *fS){
   int k;
        if(vide(*f)){
                printf("La file est vide !");
                return;
        }
        else if(f->tete == f->queue){
                k = f->tab[f->tete];
                f->tete = -1;
                f->queue = -1;
        }
        else{
                k = f->tab[f->tete];
                f->tete = (f->tete + 1)%MAX;
         }
      printf("\n%d est d%cfil%c\n",k,130,130);
}
```

- Représentation dynamique

Structure d'une file dynamique

```
typedef struct cel{
   int data;
   struct cel *next;
}cellule;
typedef cellule* filePtr;
typedef struct gr{
   filePtr tete;
   filePtr queue;
}file;}file;
```

Vérifier si la file est vide

Vérifier si la file est vide

```
int vide(file f){
        if(f.tete == NULL && f.queue == NULL)
                return 1;
        return 0;
}
```

Structure d'une file statique

```
#define   MAX   30
typedef struct{
   int tete,queue;
   int tab[MAX];
}file;
```

Vider la file

```
Vider la file
void RAZ(file *f){
        filePtr tmp;
        f->queue = NULL;
        while(f->tete != NULL){
                tmp = f->tete;
                f->tete = f->tete->next;
                free(tmp);
        }
printf("\nL'op%cration est bien
effectu%ce",130,130);
}
```

Afficher le contenu de la tête de la file

```
int tete(file f){
if (f.tete == NULL)
        return -1;     // -1 ∉ ensemble des
valeurs de la file
return f.tete->data;
}
```

Enfiler un élément à la file

```
void enfiler(file* f){
   int x;
   printf("Donnez un nobre >> ");
   scanf("%d",&x);
        filePtr Nouveau = (filePtr)malloc(sizeof(filePtr));
        Nouveau->data = x;
        Nouveau->next = NULL;
        if(vide(*f)){
                f->tete = Nouveau;
                f->queue = Nouveau;
        }
        else{
                f->queue->next = Nouveau;
                f->queue = Nouveau;
        }
        printf("\n%d est enfil%c\n",x,130);
}
```

Défiler un élément de la file

```
void defiler(file* f){
   int k;
        if(vide(*f)){
                printf("La file est vide !\n");
                return;
                } ……..
```

Défiler un élément de la file (suite)

```
……
        else if(f->queue == f->tete){
                k = f->tete->data;
                free(f->tete);    //Ou f->queue
                f->queue = NULL;
                f->tete = NULL;
        }
```

Défiler un élément de la file (suite)

```
…….
        else{
           k = f->tete->data;
                filePtr tmp = f->tete;
                f->tete = f->tete->next;
                free(tmp);
        }
        printf("\n%d est d%cfil%c.\n",k,130,130);
}
```

4. Les arbres

Structure d'un noeud

```
typedef strcut cel{
        int data ;
        strcut cel *d,*g ;
}cel ;
Typedef cel* arbre ;
```

Création d'un noeud

```
arbre create(int x){
   arbre k = (arbre)malloc(sizeof(arbre));
   k->data = x;
   k->g = NULL;
   k->d = NULL;
   return k;
}
```

Insertion d'un noeud dans un ABR

```
arbre insert(arbre a, int x){
   if(a == NULL){
      a = create(x);
   }else{
      if(x > a->data)
         a->d = insert(a->d, x);
      else
         a->g = insert(a->g, x);
   }
```

Procédures des parcours dans un arbre binaire

```
Void postfixe(arbre A){
        If(A != NULL){
                postfixe(A->g) ;
                postfixe(A->d) ;
                Printf("%d",A->data) ;
        }
}
Void Prefixe(arbre A){
        If(A != NULL){
                Printf("%d",A->data) ;
                Prefixe(A->g) ;
                Prefixe(A->d) ;
        }
}
Void infixe(arbre A){
        If(A != NULL){
                infixe(A->g) ;
                Printf("%d",A->data) ;
                infixe(A->d) ;
        }
}
```

Calcul de la hauteur d'un arbre

```
Int hauteur(arbre A){
        If(A==NULL)return 0 ;
        Else{
                Return(1+max(hauteur(A->g),hauteur(A-d) ) ) ;
        }
}
```

Calcul de la hauteur d'un arbre

- Nombre des feuilles

```
Int nbr_feuille(arbre A){
        If(A == NULL)return 0 ;
        If(A->d == NULL && A->g == NULL)return ;
        Return (nbr_feuille(A->g) + nbr_feuille(A->d)) ;
}
```

Calcul du nombre de feuilles d'un arbre

```
Int nbr_feuille(arbre A){
        If(A == NULL)return 0 ;
        If(A->d == NULL && A->g == NULL)return ;
        Return (nbr_feuille(A->g) + nbr_feuille(A->d)) ;
}
```

Recherche d'une valeur dans un arbre

```
Int recherche(arbre A, int x){
        If(A==NULL)return 0 ;
        If(A->data == x)return 1 ;
        Return(recherche(A->d) || recherche(A->g)) ;
}
```

Recherche d'une valeur dans un ABR

```
Int recherche_abr(arbre A, inr x){
        If(A == NULL)return 0;
        If(A->data == x)return 1 ;
        If(x > A->data)return recherche_abr(A->d) ;
        Else
                Recherche_abr(A->g) ;
}
```

Recherche de la plus grande valeur d'un arbre

```
Arbre grd(arbre A, int *max){
        If(A->d == NULL){
                *max = A->data ;
                Return A->g ;
        }
        A->d = grd(A->d, max) ;
        Return A ;
}
```

Suppression d'un noeud d'un ABR

```
Arbre suppr(arbre cellule, int x){
        If(A != NULL){
                if(x < A->data){
                        A->g = suppr(A->g, x) ;
                }
                else if(x > A->data){
                        A->d = suppr(A->d, x) ;
                }
                else{
                        if(A->g == NULL){
                                //il n y a pas de fils gauche
                                arbre tmp = A->d ;
                                free(A) ;
                                return tmp ;
                        }
                        else if(A->d == NULL){
                                //il n y a pas de fils droit
                                arbre tmp = A->g;
                                free(A);
                                return tmp;
                        } …….
```

```
…….
                        else{
                                int x;
                                //il y a les deux
                                A->g = gdr(A->g, &x);
                                A->data = x;
                                return A;
                        }
                }
                return A;
        }
}//fin
                return A;
        }
}//fin
```

Bibliographie

- Initiation à l'algorithmique et aux structures de données, volume1, Jacques Courtin et Irène Kowarski, Dunod, 1994.

- Algorithmes et structures de données, Niklaus Wirth, Eyrolles,1997.

- Algorithmes et structures de données : cours et exercices en langage C. Michel Divay, Dunod, 1999.