Table des matières

A. Listes chainées dynamiques

1. Qu'est ce qu'une liste chainée ?

a. Une chaine constituée de maillons

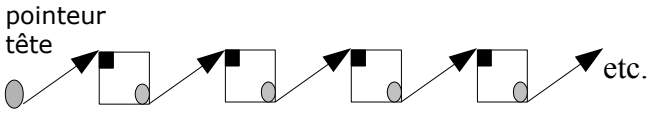
Une liste chainée est simplement une liste d'objet de même type dans laquelle chaque élément contient :

- Des informations relatives au fonctionnement de l'application.par exemple des noms et prénoms de personnes avec adresses et numéros de téléphone pour un carnet d'adresse.
- L'adresse de l'élément suivant ou une marque de fin s'il n'y a pas de suivant. C'est ce lien via l'adresse de l'élément suivant contenue dans l'élément précédent qui fait la "chaine" et permet de retrouver chaque élément de la liste.

L'adresse de l'objet suivant peut être :

- une adresse mémoire récupérée avec un pointeur (chainage dynamique).
- un indice de tableau récupéré avec un entier
- une position dans un fichier. C'est le numéro d'ordre de l'objet dans le fichier multipliée par la taille en octet du type de l'objet. Elle est récupérée avec un entier.

Que la liste chainée soit bâtie avec des pointeurs ou des entiers, c'est toujours le terme de pointeur qui est utilisé : chaque élément "pointe" sur l'élément suivant, c'est à dire possède le moyen d'y accéder et une liste chainée peut se représenter ainsi :

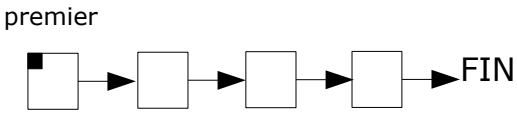


Chaque carré correspond à un maillon de la chaine. Le petit carré noir en haut à gauche correspond à l'adresse en mémoire de l'élément. Le petit rond en bas à droite correspond au pointeur qui "pointe" sur le suivant, c'est à dire qui contient l'adresse de l'élément suivant. Au départ il y a le pointeur de tête qui contient l'adresse du premier élément c'est à dire l'adresse de la chaine.

b. Trois types de listes chainées

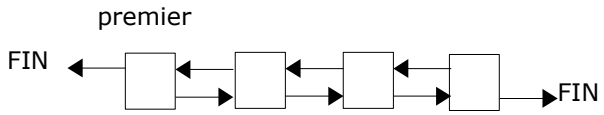
Liste simple

La liste chainée simple permet de circuler que dans un seul sens, c'est ce modèle :



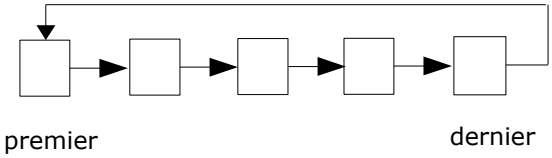
Liste symétrique ou doublement chainée

Avec le modèle double chaque élément possède l'adresse du suivant et du précédent ou des marques de fin s'il n'y en a pas. Il est alors possible de parcourir la chaine dans les deux sens :



Liste circulaire simple

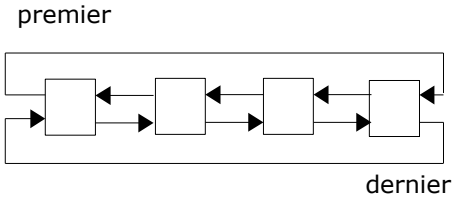
Dans une liste circulaire simple le dernier prend l'adresse du premier et la circulation est prévue dans un seul sens :



Dans ce modèle premier et dernier n'ont plus la même importance, le premier peut être n'importe quel maillon de la chaine et le dernier celui qui le précède.

Liste circulaire double

Même principe que précédemment mais avec une circulation possible dans les deux sens :



c. Les actions sur une liste chainée

Les actions sont en général toujours les mêmes et toutes ne sont pas toujours nécessaires, en gros il s'agit d'écrire des fonctions pour :

Liste vide :	savoir si une liste est vide ou pas.
Ajouter un maillon :	ajouter un maillon à la liste, soit au début, soit à la fin
Parcourir :	passer chaque élément en revue dans l'ordre du début vers la fin
Extraire :	récupérer les données d'un maillon et l'enlever de la chaine
Supprimer :	enlever un maillon de la liste sans récupérer ses données
Insérer :	ajouter un maillon quelque part dans la chaine
Détruire une liste :	désallouer tous les maillons de la liste

Copier une liste :	cloner la chaine
Sauvegarder une liste :	copier la liste sur fichier ou récupérer la liste dans le programme

d. Listes chainées contre tableaux

Voici un tableau qui récapitule les différences entre tableau et liste chainée :

TABLEAUX	LISTE CHAINEE
Taille en mémoire : Par nature, la tableau a une taille définie même dans le cas d'un tableau dynamique dont la taille peut être réévaluée périodiquement. La taille du tableau fait partie de la définition du tableau.	La liste chainée dynamique n'a pas pour sa définition de nombre d'éléments. Ils sont ajoutés ou soustraits à la demande, pendant le fonctionnement du programme.
Soustraire un élément : impossible dans un tableau d'enlever une case du tableau. Il est éventuellement possible de masquer un élément mais pas de retirer son emplacement mémoire.	Aucun problème pour soustraire un élément de la liste et libérer la mémoire correspondante.
Accès élément : Le tableau permet d'accéder à chaque élément directement. C'est très rapide.	Pour accéder à un élément il faut parcourir la liste jusqu'à lui, ça peut être long.
Tris : Les tris de tableau ne nécessitent pas de reconstruire les liens entre les emplacements mémoire du tableau. Il y a juste à manipuler les valeurs afin de les avoir dans l'ordre voulu	Il ne suffit pas de manipuler les valeurs il faut aussi reconstruire la chaine. Le mieux est de construire sa chaine en mettant les éléments dans l'ordre dés le départ plutôt que d'avoir à réorganiser l'ordre des éléments dans la chaine. Trier une chaine revient à construire une autre chaine en insérant dans l'ordre voulu chaque élément.

2. Implémenter une liste simple en dynamique

Nous allons détailler ici tous les aspects de l'implémentation d'une liste chainée. Des exemples sont notamment donnés pour l'insertion et la suppression d'éléments sur critère. Un exemple est donné également pour la sauvegarde et le chargement d'une liste.

a. Structure de donnée d'un maillon

Chaque élément de la liste est une structure qui contient des informations pour l'application (ici deux variables juste pour faire des tests) et un pointeur sur une structure de même type :

```
typedef struct elem{
```

```
   // 1 : les datas pour nos tests
   int val;
   char s[80];

   // 2 : le pointeur pour l'élément suivant
   struct elem* suiv;
}t_elem;
```

b. Début et fin de la liste

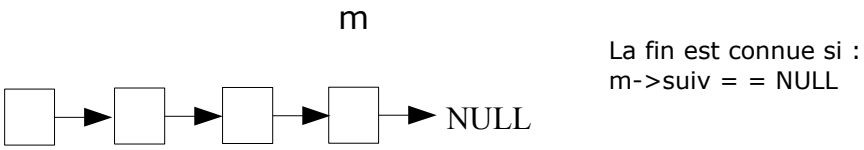
Deux positions sont très importantes dans une liste chainée : le début et la fin, souvent désignées par "premier et dernier" ou "tête et queue". Sans le premier impossible de savoir où commence la chaine et sans le dernier impossible de savoir où elle s'arrête.

Le début est donné par l'adresse du premier maillon. Une chaine prend ainsi le nom du premier maillon :

```
t_maillon*premier=NULL;
```

Lorsque la liste est vide le premier maillon prend TOUJOURS la valeur NULL.

La fin de la liste est indiquée par une sentinelle à savoir une valeur spécifique reconnaissable. Il y a plusieurs possibilités. Nous utiliserons ici la plus fréquente : la valeur NULL.



La fin est connue si :
m->suiv = = NULL

Attention

Avec les listes chainées il est recommandé de TOUJOURS METTRE A NULL UN POINTEUR NON ALLOUE. En effet un pointeur peut contenir une adresse mémoire qui n'est pas allouée, par exemple :

```
t_elem*prem;
```

prem contient ce qui traine en mémoire, une adresse quelconque non allouée. Une adresse mémoire non allouée est indétectable et écrire à une adresse non réservée fait planter le programme. La valeur NULL est détectable, elle signifie que le pointeur n'est pas alloué et le programme peut le détecter.

c. Initialiser un maillon

Il est indispensable d'allouer la mémoire pour chaque nouvel élément de la liste. Le mieux est de le faire avec l'initialisation des datas, dans une fonction d'initialisation.

Pour nos tests le champ val prend une valeur aléatoire entre 0 et 26 et le champ s prend une chaine de caractères composée d'une seule lettre de l'alphabet choisie au hasard. Le pointeur suiv est initialisé à NULL.

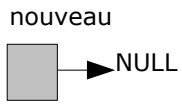
A la fin, la fonction d'initialisation retourne l'adresse du nouvel élément alloué et initialisé :

```
t_elem* init()
{
char* n[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","X","Y","Z"};

t_elem*e=malloc(sizeof(t_elem));
   e->val=rand()%26;
   strcpy(e->s,n[rand()%26]);
   e->suiv=NULL;
   return e;
}
```

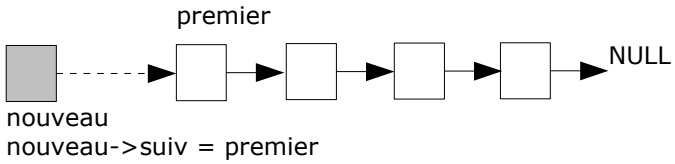
d. Ajouter au début

Pour construire une liste il faut simplement pouvoir ajouter des maillons. Le plus simple est d'ajouter en début de liste, avant la tête et de déplacer la tête ensuite.

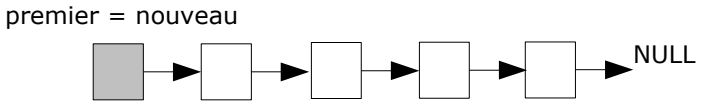
Soit un élément déjà initialisé avec la fonction init() :



Le nouvel élément est ajouté au début de la liste : le suivant c'est premier :



Le pointeur "premier" qui donne l'adresse de la liste, ne contient plus l'adresse du début qui est maintenant celle de nouveau, premier doit donc prendre cette adresse :



Pour la fonction il y a deux possibilités d'écriture. Soit prendre en paramètre l'adresse de début et utiliser le mécanisme de retour pour renvoyer la nouvelle adresse, soit prendre en paramètre l'adresse "perso" de la variable premier, à savoir passer le pointeur de début par référence.

Première version. La fonction ajout() prend en argument l'adresse du début de la liste, l'adresse du nouvel élément et retourne la nouvelle adresse du début :

```
t_elem* ajout_debut1(t_elem*prem,t_elem*e)
{
   e->suiv=prem;
   prem=e;
   return prem;
}
```

Exemple d'appel, dans le programme ci-dessous une chaine de 10 éléments est fabriquée :

```
int main()
{
t_elem* premier=NULL;
t_elem*nouveau;
int i;

   for (i=0; i<10; i++){
      nouveau=init();
      premier=ajout_debut1(premier,nouveau);
   }
   return 0;
}
```

Attention à bien initialiser premier sur NULL à la déclaration sinon la chaine n'aura pas de buttoir finale et il ne sera pas possible de repérer sa fin.

Deuxième version. Le mécanisme de retour n'est pas utilisé. Le premier paramètre est un pointeur de pointeur afin de pouvoir prendre comme valeur l'adresse de la variable pointeur qui contient l'adresse du premier élément : &premier

```
void ajout_debut2(t_elem**prem,t_elem*e)
{
   e->suiv=*prem;
   *prem=e;
}
```

Exemple d'appel, fabriquer dans le main une liste de 10 maillons :

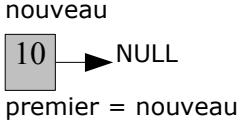
```
int main()
{
t_elem* premier=NULL;
t_elem*nouveau;
int i;

   for (i=0; i<10; i++){
      nouveau=init();
      ajout_debut2(&premier,nouveau);
   }
   return 0;
}
```

e. Insérer

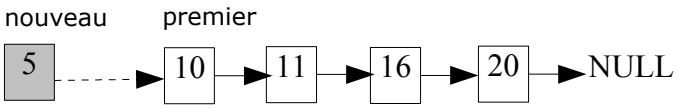
Le fait d'insérer plutôt qu'ajouter un élément dans la liste suppose un classement. Il faut un critère d'insertion, pourquoi ici et pas là ? Par exemple nous allons insérer nos éléments de façon à ce que les valeurs entières soient classées en ordre croissant.

Trois cas sont possibles : la liste est vide, insérer au début avant le premier, chercher la place précédente et insérer après.

1) Si la liste est vide, le nouveau à insérer devient le premier :



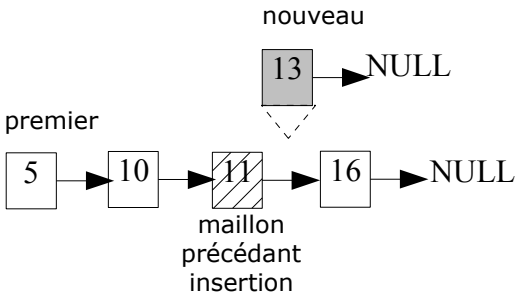
2) La liste n'est pas vide mais il faut insérer au début parce que la nouvelle valeur est inférieure à celle du premier maillon : si nouveau->val < premier->val



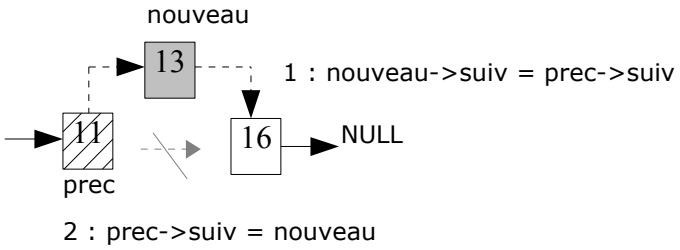
Soit les deux instructions :

```
nouveau->suiv = premier;
premier=nouveau;
```

3) Dernier cas, le maillon est à insérer quelque part dans la liste. Pour ce faire il faut chercher la bonne place et conserver l'adresse du maillon qui précède. Par exemple, insérer un nouveau maillon qui contient la valeur entière 13 dans la liste suivante :



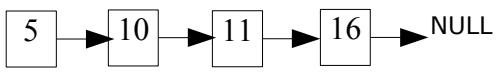
Lorsque le maillon précédent est trouvé, l'insertion est simple :



Pour trouver le maillon précédent la liste est parcourue tant que l'élément courant n'est pas NULL et que la valeur de l'élément à insérer est supérieure à la valeur de l'élément courant. Nous avons besoin de deux pointeurs : un pour l'élément courant et un pour l'élément précédent. A chaque tour, avant d'avancer l'élément courant de un maillon, l'adresse de l'élément précédent est sauvegardée :

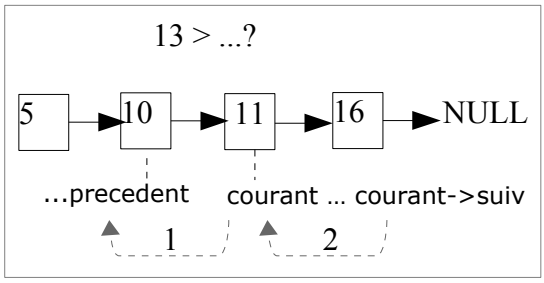
au départ courant et précédent prennent la valeur de premier :

précédent=
courant =
premier



Ensuite si courant n'est pas NULL, comparaison entre les valeurs entières de courant et nouveau. Si la valeur entière de nouveau est supérieure à la valeur entière de courant, précédent prend l'adresse de courant et courant prend l'adresse du suivant, ainsi de suite jusqu'à trouver l'emplacement, cet emplacement peut être à la fin de la liste, lorsque courant vaut NULL :

```
Si courant != NULL et nouveau->val > courant ->val alors :
    precedent=courant
    courant=courant->suiv

Par exemple insérer 13:
```



donne :

```
courant != NULL et 13 >5 alors
    precedent=courant
    courant=courant->suiv

courant != NULL et 13 >10 alors
    precedent=courant
    courant=courant->suiv

courant != NULL et 13 >11 alors
    precedent=courant
    courant=courant->suiv

courant != NULL et 13 >16   FAUX : fin de la boucle
    precedent contient l'adresse de l'élément 11
    la liaison peut être effectuée.
```

Si l'élément à insérer rencontre un élément qui a la même valeur que lui, par exemple pour insérer 16, on a alors :

```
courant != NULL et 16 >16   FAUX : fin de la boucle
precedent contient l'adresse de l'élément précédent
la liaison est effectuée en insérant le nouvel arrivant avant.
```

Mais il y a problème si l'élément à insérer doit être insérée en premier soit parce que la liste est vide, soit parce que sa valeur est inférieure à celle du premier. Dans ces deux cas il faut insérer au début de la liste et modifier la tête de la liste, le pointeur premier. De même si la valeur à insérer est égale à celle du premier. L'algorithme se divise donc en deux possibilités :

```
SI   premier==NULL OU element->valeur <= premier->valeur ALORS
    insérer l'élément au début
SINON
```

```
    courant = precedent = premier
    TANT QUE  premier != NULL ET element->valeur < courant->valeur
          precedent=courant
          courant=courant->suiv
    FIN TANT QUE
    element->suiv = courant
    precedent->suiv = element
FINSI
```

Première version sans passage par référence :

```
t_elem* inserer1(t_elem*prem,t_elem*e)
{
t_elem*n,*prec;
         // si liste vide ou ajouter en premier
   if(prem==NULL || e->val <= prem->val){  // attention <=
      e->suiv=prem;
      prem=e;
   }
   else{  // sinon chercher place précédente, insérer après
      n=prec=prem;
      while (n!=NULL && e->val > n->val){
        prec=n;
        n=n->suiv;
      }
      e->suiv=n;
      prec->suiv=e;
   }
   return prem;
}
```

Deuxième version avec passage par référence :

```
void inserer2(t_elem**prem,t_elem*e)
{
t_elem*n,*prec;
   if(*prem==NULL)                         // si liste vide
      *prem=e;
   else if ( e->val<(*prem)->val){  // si premier, ajouter debut
      e->suiv=*prem;
      *prem=e;
   }
   else{  // sinon chercher place
      n=prec=*prem;
      while (n!=NULL && e->val>n->val){
        prec=n;
        n=n->suiv;
      }
      e->suiv=n;
      prec->suiv=e;
   }
}
```

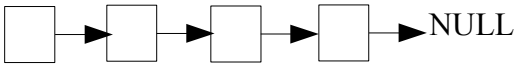
Les appels sont identiques aux autres fonctions d'ajout.

f. Parcourir la liste

Pour parcourir une liste chainée il suffit avec un pointeur de prendre successivement l'adresse de chaque maillon. Au départ le pointeur prend l'adresse

du premier élément qui est l'adresse de la liste, ensuite il prend l'adresse du suivant et ainsi de suite :

au départ :
p = premier



puis :
p = p->suiv
p = p->suiv
p = p->suiv
p = p->suiv
p == NULL

Pour chaque élément il est possible d'accéder aux datas de l'élément. Dans la fonction ci-dessous les valeurs sont simplement affichées :

```
void parcourir(t_elem*prem)
{
   if (prem==NULL)
      printf("liste vide");
   else
      while(prem!=NULL){
         printf("%d%s--",prem->val,prem->s);
         prem=prem->suiv;
      }
   putchar('\n');
}
```

Rappel :

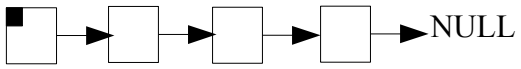
Attention au fait que le pointeur t_elem*prem déclaré en paramètre de fonction est une variable locale à la fonction. Elle ne dépend pas du contexte d'appel sauf pour la valeur qui lui est passée au moment de l'appel, exemple :

```
int main()
{
t_elem* premier=NULL;
t_elem*nouveau;
   int i;

   // fabriquer une liste de 10 éléments
   for (i=0; i<10; i++){
      nouveau=init();
      ajout_debut2(&premier,nouveau);
   }
   // afficher la liste :
   parcourir(premier);

   return 0;
}
```

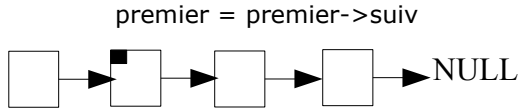
g. Supprimer au début

La suppression du premier élément suppose de bien actualiser la valeur du pointeur premier qui indique toujours le début de la liste, ce qui donne si la liste n'est pas vide :

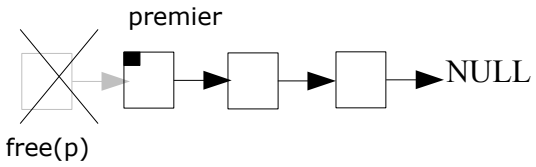
1)
p = premier



2)



3)



La modification du pointeur premier doit effectivement être répercutée sur le premier pointeur de la liste. De même que pour les fonctions d'ajout il faut retourner la nouvelle adresse de début ou utiliser un passage par référence afin de communiquer cette nouvelle adresse du début au contexte d'appel. La version présentée est avec un passage par référence du pointeur premier :

```
void supprimer_debut(t_elem**prem)
{
t_elem*n;
   if(*prem!=NULL){
      n=*prem;
      *prem=(*prem)->suiv;
      free(n);
   }
}
```

Appel :

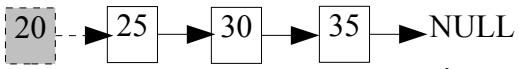
```
   supprimer_debut(&premier);
```

h. Supprimer un élément sur critère

L'objectif cette fois est de supprimer un maillon quelconque de la liste en fonction d'un critère donné. Le critère ici est que le champ val de l'élément soit égal à une valeur donnée. La recherche s'arrête si un élément répond au critère et nous supprimons uniquement le premier élément trouvé s'il y en a un. Comme pour les suppressions précédentes la liste ne doit pas être vide. Il faut prendre en compte le cas où l'élément à supprimer est le premier de la liste et pour tous les autre éléments il faut disposer de l'élément qui précéde celui à supprimer afin de pouvoir reconstruire la chaine.

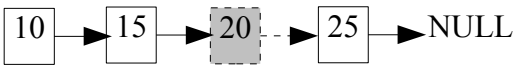
Prenons par exemple 20 comme valeur à rechercher. Si un maillon a 20 pour valeur entière il est supprimé et la recherche est terminée. Le principe est le suivant :

1) si c'est le premier élément :

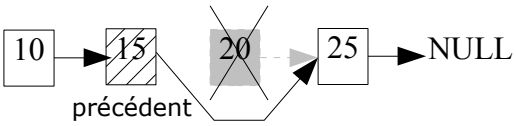


supprimer le premier comme indiqué plus haut

2) sinon trouver dans la liste un maillon où il y a 20 :



3) le supprimer, ce qui suppose d'avoir l'adresse du précédent afin de rétablir la chaine avec le suivant du maillon 20 :



La fonction reçoit en paramètre l'adresse du début de la liste et la valeur entière critère de sélection de l'élément à supprimer. Le premier élément trouvé qui a cette valeur dans son champ val est supprimé de la liste. Comme l'adresse de la liste est susceptible d'être modifiée, la fonction retourne cette adresse afin que cette modification puisse être récupérée dans le contexte d'appel. Nous proposons la fonction suivante :

```
t_elem* critere_supp_un1(t_elem*prem,int val)
{
t_elem*n,*prec;
   if(prem!=NULL){
      if(prem->val==val){// si premier
         n=prem;
         prem=prem->suiv;
         free(n);
      }
      else{             // sinon voir les autres
         prec=prem;
         n=prem->suiv;
         while(n != NULL ){
            if (n->val==val){
               prec->suiv=n->suiv;
               free(n);
               break;
            }
            prec=n;
            n=n->suiv;
         }
      }
   }
   return prem;
}
```

L'algorithme se résume ainsi :

```
Si la liste n'est pas vide
   S'il s'agit d'enlever le premier
```

```
        - l'adresse du premier est sauvée en n
        - le premier debvient le suivant du premier
        - l'adresse sauvée en n est libérée
     Si ce n'est pas le premier, voir parmi les autres, pour ce faire
        - le précedent est le premier
        - le courant n est le suivant de premier
        - Tant que n!= NULL
            - si val est trouvée
                - créer lien entre prec et suivant de n
                - libérer la mémoire del'élément n
                - provoquer la sortie de la boucle
            - sinon continuer la recherche
                - precedent = n et
                - n= suivant
           FinTant que
  à l'issue retourner prem qui a peut-être été modifié.
```

Voici une version avec passage par référence de la tête de liste, sans mécanisme de retour :

```
void critere_supp_un2(t_elem**prem,int val)
{
t_elem*n,*prec;
   if(*prem!=NULL){
      if((*prem)->val==val){// si premier
         n=*prem;
         *prem=(*prem)->suiv;
         free(n);
      }
      else{             // les autres
         prec=*prem;
         n=(*prem)->suiv;
         while(n != NULL ){
            if (n->val==val){
               prec->suiv=n->suiv;
               free(n);
               break;
            }
            prec=n;
            n=n->suiv;
         }
      }
   }
}
```

i. Détruire la liste

Pour détruire une liste il faut parcourir la liste, récupérer l'adresse du maillon courant, passer au suivant et désallouer l'adresse précédente récupérée. Au départ bien vérifier que la liste n'est pas vide. La tête de liste est modifiée, elle doit à l'issue passer à NULL. Il y a les deux possibilités habituelles : retourner la valeur NULL à la fin et la récupérer avec le pointeur de tête premier dans le contexte d'appel ou passer le pointeur premier par référence. Version avec passage par référence :

```
void detruire_liste(t_elem**prem)
{
t_elem*n;
   while(*prem!=NULL){
```

```
      n=*prem;
      *prem=(*prem)->suiv;
      free(n);
   }
   //ici *prem vaut NULL
}
```

Le contenu de la boucle while est identique à la fonction de suppression du début présentée plus haut, nous pouvons écrire, toujours avec passage par référence du pointeur premier :

```
void detruire_liste(t_elem**prem)
{
   while(*prem!=NULL)
      supprimer_debut(prem);
   //ici *prem vaut NULL
}
```

j. Sauvegarder la liste

Ecriture sur fichier

L'objectif est de sauver une liste chainée dynamique dans un fichier binaire, c'est à dire en langage C en utilisant la fonction standart fwrite(). Le principe est simple : si la liste n'est pas vide, ouvrir un fichier en écriture et copier dedans dans l'ordre où il se présente chaque maillon de la liste, ce qui donne :

```
// en paramètre passer l'adresse d'une liste au moment de l'appel
void sauver_liste(t_elem*prem)
{
FILE*f;
   // si liste non vide
   if(prem!=NULL){

      //ouvrir un fichier binaire en écriture : suffixe b
      if((f=fopen("save liste.bin","wb"))!=NULL){

          // parcourir la liste jusque fin
         while(prem!=NULL){

             // copier chaque maillon
            fwrite(prem,sizeof(t_elem),1,f);

            // passer au maillon suivant
           prem=prem->suiv;
         }
         fclose(f);    // fermer le fichier
      }
      else
         printf("erreur création fichier\n");
   }
   else
      printf("pas de sauvegarde pour une liste vide\n");
}
```

Pour sauver les données d'une liste chainée dynamique dans un fichier texte, il faut passer chaque maillon en revue et à chaque fois copier le contenu de chaque champ dans le fichier. Par exemple un maillon par ligne avec un séparateur qui

délimite chaque champ. Le séparateur peut être un caractère de ponctuation comme le point virgule. Les maillons arrivent dans l'ordre de la liste et les données sont recopiées au fur et à mesure.

Lecture à partir du fichier

Pour récupérer dans le programme une liste chainée dynamique préalablement sauvegardée dans un fichier binaire, il est nécessaire de reconstruire la liste au fur et à mesure. En effet les adresses contenues dans les pointeurs ne sont plus allouées et il faut réallouer toute la liste La fonction ci-dessous suppose qu'il y a au moins un maillon de sauvegardé dans le fichier et il ne doit pas y avoir de sauvegarde de liste vide (c'est à dire juste une création de fichier avec rien dedans). La fonction retourne l'adresse du début de la liste, c'est à dire l'adresse du premier élément, la tête de liste.

Tout d'abord, ouvrir le fichier dans lequel il y a la liste sauvegardée, ensuite :

- allouer la mémoire pour le premier maillon prem
- copier le premier maillon sauvé dans le fichier dans le premier maillon recréé prem avec la fonction fread()

Après cette première étape, tant qu'il y a des éléments à récupérer dans le fichier :

- récupérer dans une variable t_elem e les valeurs du maillon suivant :fread(&e, ...)
- allouer le suivant p->suiv du maillon courant p : p->suiv=malloc(...)
- le maillon suivant devient maillon courant : p=p->suiv
- copier les valeurs récupérée en e dans le maillon courant : *p=e;
- mettre systématiquement à NULL le maillon suivant : p->suiv=NULL, ceci afin d'avoir de façon simple NULL à la fin
- Fermer le fichier
- Retourner l'adresse de la tête de liste, adresse du premier élément.

Ce qui donne la fonction :

```
t_elem* load_liste()
{
FILE*f;
t_elem*prem=NULL,*p,e;
   if((f=fopen("save liste.bin","rb"))!=NULL){
      prem=malloc(sizeof(t_elem));
      fread(prem,sizeof(t_elem),1,f);
      p=prem;
      while(fread(&e,sizeof(t_elem),1,f)){
         p->suiv=malloc(sizeof(t_elem));
         p=p->suiv;
         *p=e;
         p->suiv=NULL;
      }
      fclose(f);
   }
   else
      printf("erreur ou fichier inexistant");
   return prem;
}
```

3. Implémenter une liste simple circulaire

a. Structure de données liste circulaire

La structure de données des maillons est la même que pour une liste non circulaire.

```
typedef struct elem{

  int val;             // 1 : les données
  char s[80];

  struct elem*suiv;    // 2 : pour construire la liste

}t_elem;
```

b. Liste vide

Comme pour une liste non circulaire, une liste vides est un pointeur à NULL :

```
t_elem*courant=NULL;
```

c. Début et fin de la liste

Une liste chainée circulaire est une liste dans laquelle la fin pointe sur le début.

Il y a deux différences avec des listes non circulaires :

- l'adresse de la chaine c'est l'adresse de n'importe quel maillon de la chaine. Au lieu d'avoir un pointeur premier nous aurons un pointeur courant qui contiendra l'adresse d'un maillon de la chaine.
- la fin de la chaine c'est aussi l'adresse du début contenue dans le pointeur courant et la valeur NULL n'est plus un indicateur de fin.

d. Initialiser un maillon

Par défaut chaque élément pointe sur lui-même, le pointeur suiv pointe sur l'élément lui-même, soit la fonction d'initialisation :

```
t_elem* init_elem(int val, char s[])
{
t_elem*e;
   e=(t_elem*)malloc(sizeof(t_elem));
   e->val=val;
   strcpy(e->s,s);
   e->suiv=e;     // pas de NULL, pointe sur lui-même
   return e;
}
```

e. Ajouter un maillon

Pour ajouter nous avons besoin du pointeur courant qui fait figure de premier et pour ne pas perdre la chaine il doit toujours posséder une adresse valide d'un maillon de la chaine. Si la liste est vide sa valeur va changer c'est pourquoi il est

passé par référence afin de ne pas perdre la modification. L'algorithme est le suivant :

- Si courant est NULL c'est que la liste est vide dans ce cas le nouvel élément donne son adresse à courant.
- Sinon, il y a au moins un élément dans la liste, peut-être plusieurs. Alors le nouvel élément est inséré derrière l'élément courant. Le nouvel élément prend pour suivant le suivant de l'élément courant et l'élément courant prend pour suivant le nouvel arrivant.

Ce qui donne :

```
void ajout_suiv(t_elem**cour,t_elem*e)
{
   if (*cour==NULL)    // cas avec modif de courant
      *cour=e;
   else{
      e->suiv=(*cour)->suiv; // pas de modif de courant
      (*cour)->suiv=e;
   }
}
```

f. Parcourir la liste

Si la liste n'est pas vide, pour parcourir la liste à partir du maillon courant, nous avons besoin de conserver l'adresse contenue dans ce maillon courant afin de pouvoir la reconnaître et savoir que le parcours est terminé. Nous n'utilisons pas le pointeur courant pour ce parcours mais un autre pointeur p qui prend la valeur de courant au départ. La valeur de courant ne bouge donc pas. Si la liste est vide nous affichons liste vide sinon :

- un pointeur prend l'adresse de courant
- la boucle est exécutée au moins une fois (il y a au moins un élément)
  - nous affichons les données,
  - nous passons au suivant

Tant que le pointeur ne retrouve pas l'adresse de courant.

Ce qui donne :

```
void affiche(t_elem*cour)
{
t_elem*p;
   if (cour==NULL)
      printf("liste vide\n");
   else{
      p=cour;
      do{
         printf("%d%s--",p->val,p->s);
         p=p->suiv;
      }while( p!=cour);
      putchar('\n');
   }
}
```

g. Supprimer un maillon

Nous ne pouvons pas supprimer facilement le maillon courant dans une liste simple parce qu'il faut disposer du maillon précédent pour rétablir la chaine. Alors nous allons supprimer le maillon qui suit le maillon courant. Algorithme :

- Tout d'abord, vérifier que la liste n'est pas vide
- Ensuite récupérer l'adresse du suivant à supprimer
- Modifier le suivant du courant qui devient le suivant de l'élément à supprimer
- Si l'élément à supprimer est le courant, c'est à dire s'il n'y a qu'un élément dans la liste, alors mettre courant à NULL pour indiquer liste vide.
- Dans tous les cas libérer la mémoire de l'élément supprimé.

Ce qui fait :

```
void supp_suiv(t_elem**cour)
{
t_elem*e;
   if (*cour!=NULL){
      e=(*cour)->suiv;
      (*cour)->suiv=e->suiv;
      if (*cour==e)
         *cour=NULL;
      free(e);
   }
}
```

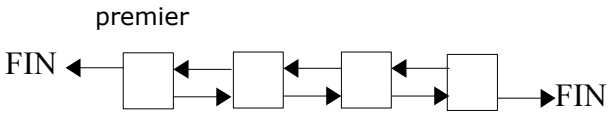
h. Détruire la liste

Pour détruire la liste nous allons parcourir autrement la liste. Si la liste n'est pas vide, nous partons du second élément jusqu'à retrouver le premier. L'adresse de chaque élément est supprimée via un pointeur intermédiaire sup. A l'issue de la boucle il ne reste plus que le pointeur courant et l'adresse qu'il contient est à son tour libérée :

```
void detruire_liste(t_elem**cour)
{
t_elem*p,*sup;
   if (*cour!=NULL){
      p=(*cour)->suiv;
      while (p!=*cour){
         sup=p;
         p=p->suiv;
         free(sup);
      }
      free(*cour);
      *cour=NULL;
   }
}
```

4. Implémenter une liste symétrique

une liste chainée est symétrique lorsqu'elle permet de circuler dans les deux sens de la façon suivante :



Nous allons implémenter une telle liste afin d'expérimenter les problèmes qui peuvent se poser.

a. Structure de donnée

L'élément est défini par le type t_elem. Pour les datas il a comme précédemment un champ pour une valeur entière et un champ pour une chaine de caractères. Pour la mécanique de la liste chainée il possède deux pointeurs struct elem* un vers l'élément précédent et un vers l'élément suivant :

```
  typedef struct elem{
   // les datas
   int val;
   char s[80];
   // mécanique de la chaine
   struct elem* suiv;
   struct elem*prec;
}t_elem;
```

b. Liste vide

Une liste vide est un pointeur premier à NULL

```
t_elem*prem=NULL ;
```

c. Début et fin de la liste

Le début c'est le pointeur premier qui contient l'adresse du premier élément. Éventuellement il peut être intéressant d'avoir un pointeur dernier pour le dernier élément de façon à pouvoir entrer dans la liste par un côté ou un autre. Dans les deux sens c'est toujours le principe de la sentinelle à NULL qui est utilisé.

d. Initialiser un élément

La fonction d'initialisation est identique à celle pour une liste chainée dynamique simple vue précédemment. Évidement pour une liste symétrique le deux pointeurs prec et suiv sont mis à NULL :

```
t_elem* init_elem()
{
char* n[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};

t_elem*e=malloc(sizeof(t_elem));
   e->val=rand()%26;
   strcpy(e->s,n[rand()%26]);
   e->suiv=NULL;
   e->prec=NULL;
   return e;
}
```

e. Ajouter un élément au début

L'élément e est au début de la liste :

- le premier courant devient son suivant : e->suiv=*prem
- e devient son précédent : (*prem)->prec=e;
- le pointeur premier devient e : *prem=e;

```
void ajouter_debut(t_elem**prem,t_elem*e)
{
   e->suiv=*prem;
   (*prem)->prec=e;    // lien vers précédent
   *prem=e;
}
```

f. Ajouter un élément à la fin

Si la liste est vide, l'élément e est le premier et c'est fini : *prem=e

sinon il faut trouver le dernier maillon p (à moins d'avoir un pointeur dernier) et ajouter le nouveau : le suivant de p c'est e et le précédent de e c'est p :

```
void ajouter_fin(t_elem**prem,t_elem*e)
{
t_elem*p;
   // si liste vide
   if (*prem==NULL)
      *prem=e;
   // sinon chercher dernier maillon
   else{
      p=*prem;
      while(p->suiv!=NULL)
         p=p->suiv;
      // et ajouter le nouveau
      p->suiv=e;
      e->prec=p;
   }
}
```

g. Parcourir, afficher la liste

Parcourir et afficher se fait de la même façon qu'avec une liste simple : partir du premier jusqu'à la fin avec un pointeur et afficher les valeurs de chaque élément. Pour parcourir la chaine dans l'autre sens il faut avoir un pointeur sur le dernier. C'est bien entendu possible de l'ajouter à la structure de données une liste c'est deux pointeurs un pour le premier et un pour le dernier. Il faut alors le mettre à jour en permanence dans toutes les fonctions d'ajout et de suppression. Voici le fonction d'affichage :

```
void afficher_chaine(t_elem*prem)
{
   if (prem==NULL)
      printf("liste vide\n");
   while(prem!=NULL){
      printf("%d%s--",prem->val,prem->s);
      prem=prem->suiv;
   }
   putchar('\n');
}
```

h. Supprimer un élément

Supprimer un élément est plus simple avec une liste symétrique qu'avec une liste simple. Il suffit de se positionner dessus et de rétablir les liens vers le précédent s'il y en a un et le suivant s'il y en à un :

- Si l'élément à supprimer a un suivant, le précédent du suivant devient le précédent de l'élément à supprimer.
- Si l'élément à supprimer a un précédent, le suivant du précédent devient le suivant de l'élément à supprimer.
- Si l'élément à supprimer est la tête de la liste, le premier de la liste devient le suivant

Dans la fonction ci-dessous nous ne libérons pas la mémoire de l'élément supprimé de la liste,son espace mémoire est toujours valide et accessible :

```
void supp_elem(t_elem**prem,t_elem*e)
{
   if (e->suiv!=NULL)
      e->suiv->prec=e->prec;
   if (e->prec!=NULL)
      e->prec->suiv=e->suiv;
   if (e==*prem)
      *prem=e->suiv;
}
```

i. Détruire la liste

Détruire une liste symétrique est identique à détruire une liste simple. Il s'agit de prendre une à une les adresses de chaque élément de la liste et de les libérer sans empêcher de pouvoir, pour chaque élément, passer à l'élément suivant :

```
void detruire_liste(t_elem**prem)
{
t_elem*sup;
   while (*prem!=NULL){
      sup=*prem;
      *prem=(*prem)->suiv;
      free(sup);
   }
}
```

j. Copier une liste

Copier une liste dynamique, simple ou symétrique, signifie reconstruire une liste en parallèle de la liste copiée. Il ne faut pas utiliser les même adresses mémoire sinon ce serait en mémoire la même liste manipulée par des pointeurs différents. Tout d'abord il faut créer le premier élément. C'est son adresse qui indique le début de la liste qui est retournée à l'issue de la copie, c'est la copie. Pour chaque maillon de la chaine à copier créer un nouveau maillon lui affecter les mêmes valeurs et le relier au précédent de la chaine copie, passer au maillon suivant de la chaine copie , c'est à dire celui qui vient d'être créé et passer au suivant de la chaine à copier jusqu'à arriver à la fin de la chaine à copier. Alors retourner la tête de la chaine copiée. Ça donne :

```
t_elem* copie_liste(t_elem*prem)
```

```
{
t_elem*prem2=NULL,*e,*p;

   if (prem!=NULL){
      // le premier pour la nouvelle liste
      prem2=init_elem(prem->val,prem->s);
      p=prem2;
      prem=prem->suiv;

      // les suivants
      while(prem!=NULL){
         e=init_elem(prem->val,prem->s);
         p->suiv=e;
         e->prec=p;
         p=e;
         prem=prem->suiv;
      }
   }
   return prem2;
}
```

5. Mise en pratique listes chainées

Exercice 1
Dans un programme mettre en place une liste chainée simple à partir de la structure de données du cours, tester les fonctions d'initialisation, d'ajout de suppression, de destruction de la liste et de sauvegarde de la liste.
Ajouter ensuite :

- une fonction copie de liste
- une fonction d'insertion en ordre croissant
- une fonction de suppression selon un critère donnée (par exemple tous les éléments dont le champ val est égal à une valeur ou inférieur à une valeur)

Exercice 2
Dans un programme mettre en place une liste chainée circulaire simple à partir de la structure de données du cours, tester les fonctions d'initialisation, d'ajout de suppression, de destruction de la liste.
Ajouter ensuite :

- une fonction copie de liste
- une fonction d'insertion en ordre croissant
- une fonction de suppression selon un critère donnée (par exemple tous les éléments dont le champ val est égal à une valeur ou inférieur à une valeur)

Exercice 3
Dans un programme mettre en place une liste symétrique, utiliser les fonctions du cours. Ajouter la sauvegarde et le chargement d'une liste, la suppression d'éléments sur critère.

Exercice 4
L'objectif est de faire une suite de nombre entiers sous la forme d'une liste chainée.
Le programme à faire :

- initialise chaque entier avec une valeur aléatoire comprise entre 0 et 1000
- affiche une liste de nb entiers, nb entré par l'utilisateur

- peut détruire la liste afin d'en faire une nouvelle.
- calcule la somme des entiers de la liste
- met à -1 le maillon nb/2 de la liste
- passe en négatif tous les maillons inférieurs à un seuil déterminé par l'utilisateur. Afficher le résultat.
- Efface tous les maillons dont la valeur est comprise entre un seuil haut et un seuil bas entrés par l'utilisateur. Afficher le résultat.
- Duplique les maillons qui ont la même valeur qu'une valeur entrée par l'utilisateur. Afficher le résultat.
- Sauve la liste sur fichier (suppose chapitre fichiers)
- Charge une liste sauvegardée

Exercice 5
L'objectif est d'écrire les deux fonctions suivantes : la première permet de transformer en liste chainée un tableau dynamique de nb éléments. La seconde transforme à l'inverse une liste chainée de nb éléments en un tableau dynamique. Faire un programme de test.

Exercice 6
Écrire une fonction qui prend en paramètre une liste chainée et renvoie une autre liste ayant les mêmes éléments mais dans l'ordre inverse. Tester dans un programme.

Exercice 7
Écrire une fonction de concaténation de deux listes chainées. Il y a deux versions de la fonction : une destructrice des deux listes données au départ et l'autre qui préserve ces deux listes. Tester dans un programme.

Exercice 8
Écrire une fonction qui détermine si une liste chainée d'entiers est triée ou non en ordre croissant.
Écrire une fonction qui insère un élément à sa place dans une liste chainée triée.
Tester dans un programme.

Exercice 9
Ecrire une fonction qui permet de fusionner deux listes chainées. La fusion se fait en alternant un élément d'une liste avec un de l'autre liste. Tous les éléments des deux listes doivent trouver leur place dans la liste résultante même en cas de différence de taille. Faire deux versions, une qui conserve les deux listes de départ et une autre qui les détruit. Tester dans un programme.

Exercice 10
Écrire une fonction qui prend en entrée une liste chainée d'entiers et qui ressort deux listes chainées, une pour les nombres pairs et une autre pour les nombres impairs. La liste initiale est détruite. Tester dans un programme.

Exercice 11
Écrire les instructions qui saisissent puis affichent une liste de chevaux de course. Chaque cheval est entré séparément par l'utilisateur. Un cheval est défini par un nom, un dossard, un temps réalisé dans la course, un classement et la liste est une liste chainée. Le programme permet d'ordonner la liste selon le classement des chevaux et de supprimer des chevaux de la liste.

Exercice 12 (suppose fichier)
Ecrire un programme permettant de lire un texte à partir d'un fichier. Chaque mot du texte est récupéré dans une liste chainée qui regroupe tous les mots du texte. Les mots sont dans l'ordre du texte sur le fichier et il n'y a pas de répétition de mot.

Exercice 13
Écrire un programme qui permet de distribuer les cartes d'un jeu de 52 cartes entre nb joueurs entré par l'utilisateur. Le jeu de chaque joueur est constitué par une liste chainée. Tester dans un programme.

Exercice 14
Le problème de Joséphus Flavius : Dans un bureau de recrutement, n personnes numérotées de 1 à n sont disposées en cercle suivant l'ordre de leur numéro. Le chef de bureau est au centre, puis se dirige vers la personne n°1. Sa stratégie est d'éliminer chaque deuxième personne qu'il rencontre en tournant sur le cercle. la dernière personne restante est embauchée. Par exemple s'il y a 10 personnes, n=10, les personne 2, 4, 6, 8, 10, 3, 7, 1, 9 sont éliminées et la personne restante est le n°5. Faire un programme de simulation :

1) Pour n entré par l'utilisateur donner le numéro de la personne restante.

2) au lieu de prendre chaque deuxième personne généraliser en prenant la k-éme personne, k entré par l'utilisateur.
Il s'agit de faire une liste circulaire. Chaque élément est une personne (nom, numéro)

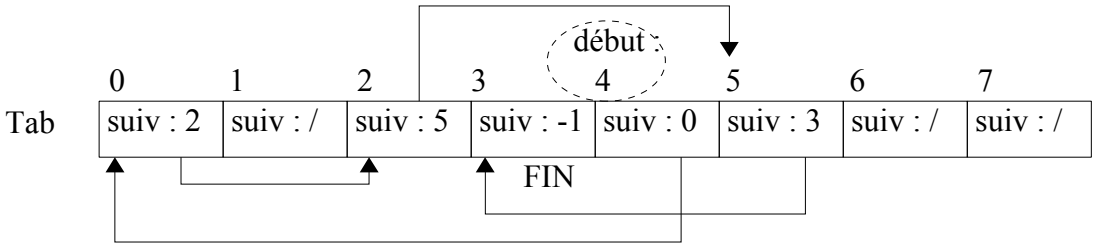
Exercice 15
Division cellulaire de l'algue Anabaena Catenula. Au départ nous avons une cellule de taille 4 symbolisée par un sens, par exemple une flèche ← . Elle grandit de 1 à chaque étape de son évolution. A l'étape 5 elle a la taille 9 qui est le maximum qu'une cellule peut atteindre. Alors elle se divise en deux cellules, une de 4, sens ←, et une de 5 symbolisée par le sens → et l'évolution reprend. Dés qu'une cellule arrive à la taille 9 elle se divise en deux cellules de 4 et 5. Programmer une simulation. Là encore il s'agit de faire une liste circulaire. Définir pour commencer une structure cellule avec une représentation du sens. Trouver un mode d'affichage et pouvoir afficher une étape quelconque de la progression.

B. Listes chainées statiques

1. Principe

a. Liste dans un tableau ou sur fichier

Une liste chainée peut également s'implémenter dans un tableau. Dans ce cas le pointeur sur l'élément suivant est un entier qui prend comme valeur l'indice du tableau ou se trouve l'élément suivant de la liste. Le tableau en tant que tel n'est jamais parcouru comme un tableau normal dans un sens ou dans l'autre. C'est toujours la liste qui est lue et utilisée dans l'ordre ou sont rangés les éléments qui la composent. Par exemple :



La liste ci-dessus commence à 4, l'élément suivant est à 0, le suivant à 2, le suivant à 5, le suivant à 3 et le suivant à l'indice -1 qui est hors tableau ce qui signifie la fin de la liste avec l'élément à 3. Dans cette liste il reste trois positions libres, une à l'indice 1, une à l'indice 6 et une à l'indice 7. Le nombre total d'éléments dans la liste ne peut pas dépasser la taille du tableau, ici 8. Mais si le tableau est dynamique, il est possible de le réallouer soit avec une taille supérieure, soit avec une taille inférieure si dans un contexte donné la gestion de la mémoire totale disponible est nécessaire. Par ailleurs il convient de remarquer qu'il est possible d'avoir plusieurs listes différentes qui se partagent un même tableau.

Notons également qu'une liste peut être implémentée dans un fichier à accès direct, c'est à dire un fichier dont la taille est fixée au départ et ouvert en écriture-lecture. Dans ce cas l'indice du tableau est remplacé par la position de l'élément dans le fichier et les algorithmes sont les mêmes.

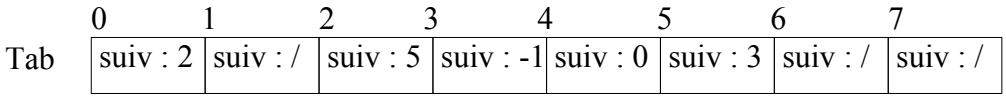
b. Attribution mémoire : table, ramasse-miettes, liste libre

La différence avec une liste chainée dynamique est que l'allocation de mémoire a lieu au départ et ne varie pas à chaque nouvelle entrée ou chaque sortie d'un élément de la liste. Il n'y a pas une allocation de chaque élément entrant dans la liste ni de libération de la mémoire à chaque élément sortant de la liste. Éventuellement le bloc initial peut être réalloué une fois de temps en temps. Par exemple s'il n'y a pas assez de place il peut être augmenté de nb éléments. Mais ça ne change rien à la gestion des éléments du bloc.

Les entrées et sorties successives des éléments dans la ou les listes font que les éléments des listes occupent le tableau de façon non systématique et les places libres sont disséminées de façon anarchique. Il est nécessaire de pouvoir les retrouver afin de les attribuer aux nouveaux éléments entrants dans une liste. Pour ce faire trois méthodes sont possibles.

Table d'allocation

Une première solution consiste à entretenir une table d'allocation c'est à dire un tableau de booléens qui indique pour chaque position du tableau si elle est libre 1 ou pas 0. La table d'allocation du bloc Tab serait :



	0	1	2	3	4	5	6	7
Table allocation	0	1	0	0	0	0	1	1

Il apparaît que les position libres sont 1, 6, 7. A chaque nouvelle entrée dans la liste une position libre de la table passe à 0, non libre. A chaque élément supprimé de la liste sa position dans la table passe à 1 libre.

Ramasse-miette

Le ramasse-miette est un algorithme qui permet de trouver une place libre dans le tas en s'appuyant sur les données sans avoir à entretenir une table d'allocation en parallèle. C'est la méthode que nous utilisons dans l'exemple ci-dessous

Liste des positions libres

Une autre solution consiste à entretenir une liste des places libres. Pour entrer un élément on prend une position dans la liste des positions disponibles, cette position est alors supprimée de la liste des places libres. A l'inverse à chaque fois qu'un élément est supprimé de la liste sa position est rajoutée à la liste des positions libres.

2. Liste simple dans un tableau avec ramasse-miettes

a. Structure de données

L'élément

Nous reprenons le type d'élément utilisé pour tester les listes chainées dynamiques. L'élément à deux champs de données : un champ val qui contient une valeur entière et un champ s pour stocker une chaine de caractères. Le champ suiv est un pointeur d'indice c'est à dire une variable entière destinée à contenir la valeur de l'indice du tableau qui fait de l'élément suivant dans la liste, ce qui donne :

```
typedef struct elem{
   int val;
   char s[80];
   int suiv;
}t_elem;
```

Le bloc (tas)

Le bloc est le tableau global des éléments potentiels. Il a ici une taille maximum définie par la macro :

```
#define BMAX    16
```

Bien entendu il peut faire l'objet d'une allocation dynamique voir de réallocation pendant le fonctionnement du programme mais dans le test ci-dessous nous nous en sommes tenus à un tableau statique de BMAX éléments et nous avons défini un type BLOC :

```
typedef t_elem BLOC[BMAX];
```

Élément libre

Les positions libres sont indiquées par la valeur -1 définie par une macro constante :

```
#define LIBRE          -1
```

Les listes

Une liste est caractérisée par l'indice de départ et nous définissons une liste uniquement par un entier. En fait le bloc peut être partagé très facilement entre plusieurs listes et nous pourrions même avoir un tableau de liste ! A savoir un tableau d'entiers ou chaque entier correspond à un départ de liste. Dans le test ci-dessous nous avons uniquement deux listes déclarées dans le main() :

```
int l1, l2;
```

Fin de liste, liste vide

La fin de la liste est indiquée par la taille du bloc BMAX afin de différencier la fin de la liste d'une position libre. Si une liste est vide sa tête est à BMAX. Au départ les listes sont vides :

```
int l1=BMAX, l2=BMAX;
```

b. Initialisation

Initialiser le bloc

Au départ le bloc doit être initialisé. Pour toutes les positions le champ suiv de chaque élément est mis à LIBRE, pour ce faire la fonction standard memset() peut être utilisée :

```
void init_bloc(BLOC b)
{
   memset(b,LIBRE,sizeof(t_elem)*BMAX);
}
```

La fonction memset met à LIBRE tous les octets couverts par le bloc b.

Initialiser un élément

Les valeurs que l'on souhaite donner à l'élément sont données en paramètre, un entier et une chaine de caractère et la fonction retourne un élément initialisé avec ces valeurs :

```
t_elem init_elem(int val, char s[])
{
t_elem e;
   e.val=val;
   strcpy(e.s,s);
   e.suiv=LIBRE;  // par défaut une position libre
   return e;
}
```

c. Fonction d'attribution de position

Pour obtenir une position libre dans le tableau un moyen simple est de parcourir le tableau depuis le début et de renvoyer l'indice de la première position libre trouvée. L'inconvénient est que ce parcours peut s'avérer long si le tableau est très grand. Nous allons opter pour une attribution avec un facteur hasard. Au lieu de toujours

commencer à parcourir le tableau du début, nous partirons d'une position sélectionnée au hasard et à partir de cette position nous cherchons la première position libre. La fonction ci-dessous retourne la position libre trouvée dans le tableau d'éléments donné en paramètre. A défaut, lorsque le tableau et plein, c'est la valeur NBFIN qui est retournée. Le tableau BLOC est passé en paramètre :

```
int attrib_pos(BLOC b)
{
int pos,i;
   i=rand()%BMAX;
   for (pos=(i+1)%BMAX; pos!=i; pos=(pos+1)%BMAX)
      if (b[pos].suiv==LIBRE)
         break;
   if (pos==i)
      pos=BMAX;
   return pos;
}
```

d. Ajouter un élément dans une liste

Pour ajouter un élément au début de la liste, le premier point est d'obtenir une position libre dans le tas avec la fonction attrib_pos(). Si la position retournée est valable, c'est à dire différente de NBFIN, alors :

- recopier à cette position les données voulues transmises par le paramètre t_elem e
- cette position prend la tête de la liste : b[pos].suiv = *prem; La tête actuelle de la liste est transmise par référence afin de pouvoir enregistrer le changement de tête :
- la première position de la liste devient la position pos : prem = pos

Ce qui donne la fonction suivante :

```
void ajout_debut(BLOC b,int*prem,t_elem e)
{
int pos;
   if ((pos=attrib_pos(b))!=BMAX){
      b[pos]=e;
      b[pos].suiv=*prem;
      *prem=pos;
   }
}
```

e. Insérer après

Il s'agit ici de pouvoir insérer un élément après un autre qui est dans la liste et dont on a l'indice dans le tableau. Un éventuel choix sur critère doit être fait ailleurs, dans une ou plusieurs autres fonctions à part.

La fonction reçoit en paramètre le bloc, la liste concernée, l'indice de l'élément derrière lequel insérer et les valeurs à insérer, ce qui donne :

```
void inserer_apres(BLOC b, int prem,int id_elem,t_elem e)
{
int pos;

    if ( prem==BMAX)
```

```
        printf("ajout impossible liste vide\n");
    else if(b[id_elem].suiv == LIBRE)
        printf("ajout impossible apres element libre\n");
    else if ( (pos=attrib_pos(b)) == BMAX)
        printf("ajout impossible liste pleine\n");
    else{
         b[pos]=e;
         b[pos].suiv=b[id_elem].suiv;
         b[id_elem].suiv=pos;
    }
}
```

La fonction ne fait rien si la liste est vide, si l'indice de l'élément passé correspond à une position libre ou si la liste est pleine. Sinon

- une position libre pos est attribuée
- le nouvel élément est copié à cette position
- l'élément suivant du nouveau est le suivant de celui après lequel il faut insérer
- le suivant de celui après lequel insérer est le nouveau.

f. Supprimer début

Supprimer le début d'une liste suppose toujours la gestion du premier élément de la liste à savoir la tête de liste indispensable pour savoir où commence la liste. Comme toujours il faut commencer par vérifier que la liste n'est pas vide. Ensuite :

- récupérer l'indice de la position à libérer, c'est à dire l'indice du premier élément.
- Ensuite affecter au premier élément la valeur d'indice de l'élément suivant.
- Pour finir l'ancienne première position à l'indice qui a été récupéré au début est mise à LIBRE

ce qui donne :

```
void supprimer_debut(BLOC b, int *prem)
{
int supp;
    if(*prem!=BMAX){  // si la liste n'est pas vide
        supp=*prem;
        *prem=b[*prem].suiv;
        b[supp].suiv=LIBRE;
    }
}
```

g. Supprimer après

Il s'agit là de pouvoir supprimer un élément après un autre dont l'indice est donné en paramètre à la fonction. En premier l'indice de l'élément à effacer est récupéré dans pos et plusieurs vérifications sont faites :

si la liste n'est pas vide et si cette position à supprimer n'est pas libre et si cette positon à supprimer n'indique pas la fin de liste (impossible de supprimer après le dernier élément puisqu'il n'y a plus d'élément) alors :

- le suivant de la position après laquelle supprimer est le suivant de la position à supprimer et la position supprimée est mise à LIBRE.

Ce qui donne la fonction :

```
void supprimer_apres(BLOC b, int prem, int id_elem)
{
int suivant;
    if (prem!=BMAX){ // si liste non vide
       suivant=b[id_elem].suiv;
       if( suivant!=LIBRE && suivant!=BMAX){
           b[id_elem].suiv=b[suivant].suiv;
           b[suivant].suiv=LIBRE;
       }
    }
}
```

h. Afficher une liste

Même principe que dans le test précédent, le bloc et l'indice tête de liste de la liste à afficher sont donnés en paramètre :

```
void affiche(BLOC b,int l)
{
   if (l==BMAX)
      printf("liste vide");
   while (l!=BMAX){
      printf("%d%s--",b[l].val,b[l].s);
      l=b[l].suiv;
   }
   putchar('\n');
}
```

i. Supprimer une liste

Il n'est pas possible de réinitialiser le bloc parce qu'il contient plusieurs listes simultanément. Il est nécessaire d'effacer chaque liste élément par élément sans toucher aux autres listes.

Le bloc et la tête de liste sont donnés en paramètre. La tête de liste est passée par référence. Elle va en effet être modifiée, elle passe à BMAX à l'issue de la destruction de la liste. Chaque position supprimée est mise à LIBRE, ce qui donne :

```
void supprime_liste(BLOC b, int *l)
{
int sup;
   while (*l!=BMAX){
      sup=*l;
      *l=b[*l].suiv;
      b[sup].suiv=LIBRE;
   }
}
```

j. Test dans le main()

Toutes les fonctions ci-dessus sont testées dans le main(). Voici le menu :

```
int menu()
{
int res=-1;
   printf(  "1 : ajout liste\n"
```

```
            "2 : inserer apres\n"
            "3 : supprimer debut\n"
            "4 : supprimer apres\n"
            "5 : supprimer liste\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Le montage suivant permet d'accomplir les actions du menu et de mettre en œuvre les fonctions que nous avons présentées :

```
int main()
{
BLOC b;
int L=BMAX;
t_elem e;
int id_elem;
int fin=0;

   init_bloc(b);
   while (!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,"L");
            ajout_debut (b,&L,e);
            affiche(b,L);
            break;
         case 2 :
            id_elem=L;
            e=init_elem(rand()%26,"L");
            inserer_apres(b, L,id_elem,e);
            affiche(b,L);
            break;

         case 3 :
            supprimer_debut(b,&L);
            affiche(b,L);
            break;

         case 4 :
            supprimer_apres(b, L, L);
            affiche(b,L);
            break;

         case 5 :
            supprime_liste(b, &L);
            affiche(b,L);
            break;

         default : fin=1;

      }
   }
   return 0;
}
```

k. Plusieurs listes dans un même bloc

Iles facile d'ajouter dans Le programme ci-dessus une seconde liste pour le même tas. Juste pour tester le menu permet de créer et supprimer deux listes

```
int menu()
{
int res=-1;
   printf(  "1 : ajout liste 1\n"
            "2 : supprime liste 1\n"
            "3 : ajout liste 2\n"
            "4 : supprime liste 2\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

et le main ensuite correspond aux actions :

```
int main()
{
BLOC b;
int L=BMAX;
int L2=BMAX;
t_elem e;
int id_elem;
int fin=0;

   init_bloc(b);
   while (!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,"L");
            ajout_debut (b,&L,e);
            affiche(b,L);
            break;
         case 2 :
            id_elem=L;
            e=init_elem(rand()%26,"L");
            inserer_apres(b, L,id_elem,e);
            affiche(b,L);
            break;

         //LISTE 2
         case 3 :
            e=init_elem(rand()%26,"L2");
            ajout_debut (b,&L2,e);
            affiche(b,L2);
            break;
         case 4 :
            supprime_liste(b, &L2);
            affiche(b,L2);
            break;

         default : fin=1;

      }
   }
   return 0;
}
```

3. Gestion d'une liste libre

Au début de ce chapitre nous avons mentionné une alternative au ramasse-miette pour l'allocation de positions libres dans le bloc, c'est la gestion d'une liste de positions libres. En effet toutes les positions libres peuvent être reliées en une liste. Lorsque le programme a besoin d'attribuer une position libre pour un élément entrant cette position est prise dans la liste des positions libres. Si le programme supprime un élément il libère une position qui est alors ajoutée à la liste des positions libres.

Pour tester cette organisation nous reprenons le programme précédent, avec juste les modifications suivantes :

- initialisation de la liste libre
- fonction d'attribution de position libre qui supprime une position de la liste libre
- la suppression d'un élément d'une liste du tas qui ajoute une position à la liste libre

a. Structure de données

La seule modification par rapport au programme précédent est une liste supplémentaire nommée libre :

```
int libre;
```

Cette liste regroupe en permanence les postions libres et il n'y a plus dans le programme à tester dans le bloc si une position est libre ou pas, ainsi la macro LIBRE est inutiles et disparaît.

b. Initialisation liste libre

Au départ toutes les listes sont vides sauf la liste des éléments libres qui est pleine. Le premier élément libre est l'élément 0 et tous les éléments sont reliés entre eux dans l'ordre du début à la fin du bloc. Le dernier pointe sur BMAX qui indique la fin de liste, ce qui donne la fonction d'initialisation suivante :

```
void init_liste_libre(BLOC b, int*libre)
{
int i;
   *libre=0;                  // premier élément à 0
   for (i=0; i<BMAX; i++)
      b[i].suiv=i+1;          // le dernier est à BMAX
}
```

c. Attribution de position libre

Si la liste des positions libres n'est pas vide, l'indice de la première est retournée pour attribution d'un nouvel élément. La tête de liste libre prend alors la valeur de la position libre suivante. Si la liste libre est vide la valeur de retour est BMAX :

```
int attrib_pos(BLOC b, int*libre)
{
int pos;
   pos=*libre;
```

```
   if (*libre!=BMAX)
      *libre=b[*libre].suiv;
   return pos;
}
```

d. Ajout d'un élément au début d'une des listes

Pas de modification pour l'ajout qui s'opère sur la liste courante après attribution d'une position libre. Mais la fonction prend en plus la liste libre en paramètre avec un passage par référence parce que la modification de sa valeur doit être conservée.

```
void ajout_debut(BLOC b, int*l, int*libre, t_elem e)
{
int pos;
   if( (pos=attrib_pos(b,libre))!=BMAX){
      b[pos]=e;
      b[pos].suiv=*l;
      *l=pos;
   }
   else
      printf("liste pleine\n");
}
```

e. suppression d'un élément au début d'une des listes

Supprimer un élément d'une liste c'est l'enlever de la liste concerné et ajouter sa position dans la liste des positions libres. Là encore la fonction reçoit la tête de la liste vide en paramètre, avec passage par référence :

```
void supp_debut(BLOC b, int*l, int*libre)
{
int i;
   if (*l!=BMAX){
      i=*l;
      *l=b[*l].suiv;    // suppression de la liste courante
      b[i].suiv=*libre; // ajout dans liste libre
      *libre=i;
   }
}
```

f. Test dans le main()

Le menu :

```
int menu()
{
int res=-1;
   printf(  "1 : ajout liste 1\n"
            "2 : supprime debut liste 1\n"
            "3 : ajout liste 2\n"
            "4 : supprime debut liste 2\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
```

```
}
```

Le test suivant permet de mettre en oeuvre toutes les fonctions précédentes :

```
int main()
{
BLOC b;
int l1=BMAX,l2=BMAX,libre; // les trois listes
t_elem e;
int fin=0;

   init_liste_libre(b, &libre);
   while (!fin){
      switch(menu()){
         // liste 1
         case 1 :
            e=init_elem(rand()%26,"L1");
            ajout_debut (b,&l1,&libre,e);
            affiche(b,l1);
            break;
         case 2 :
            supp_debut(b, &l1,&libre);
            affiche(b,l1);
            break;
         // liste 2
         case 3 :
            e=init_elem(rand()%26,"L2");
            ajout_debut (b,&l2,&libre,e);
            affiche(b,l2);
            break;
         case 4 :
            supp_debut(b, &l2,&libre);
            affiche(b,l2);
            break;
         default : fin=1;
      }
   }
   return 0;
}
```

C. Tableaux d'indices

1. Principe

A partir d'un tas qui est un tableau il y a la possibilité de faire plusieurs listes différentes simultanément sur les mêmes données, par exemple pour proposer plusieurs ordres possibles dans l'organisation des données sans les bouger à chaque utilisation. Pour ce faire nous pouvons sortir le champ du suivant des structures qui correspondent aux éléments dans le tas et avoir plusieurs tableaux d'indices à l'extérieur qui ordonnent ces éléments. Soit par exemple le tas ci-dessous qui est un tableau d'entiers :

	0	1	2	3	4	5	6	7
Tas	45	3	32	145	59	7	19	23

voici un tableau d'indices qui permet de stocker un parcours en ordre croissant :

tête = 1

	0	1	2	3	4	5	6	7
suiv	4	5	0	1 ou 8	3	6	7	2

L'indice ou se trouve la plus petite valeur donne la tête de la liste, le premier de la liste. C'est l'indice 1 avec la valeur trois dans le tas, ensuite viennent les indices des éléments du tas n°5, 6,7, 2, 0, 4, 3. A l'indice 3 du tableau suiv, fin de la liste nous avons deux possibilités. Soit comme nous l'avons fait jusque maintenant avoir la valeur de fin de liste, à savoir la taille du tableau 8, soit retrouver la valeur de l'indice de tête 1 pour avoir une liste circulaire. Dans le premier cas le parcours peut s'effectuer avec une boucle for de la façon suivante :

```
for (i=tete; suiv[i]!=8; i=suiv[i])
    printf("%d", tas[i]);
```

Dans le second nous avons :

```
for (i=tete; suiv[i]!=tete; i=suiv[i])
    printf("%d", tas[i]);
```

Lorsque la fin de la liste correspond au début la liste est dite circulaire. Une liste circulaire peut boucler sur elle-même peut être utilisée en prenant n'importe quelle position comme tête et point de départ du parcours. L'ordre est alors modifié et ne respecte plus complètement l'ordre croissant. Par exemple si nous prenons comme tête la position 7 nous obtenons la suite des indices 7, 2, 0, 4, 3, 1, 5, 6 pour le tas soit la suite des nombres : 23, 32, 45, 59, 145, 3, 7, 19. Nous pouvons tester cette propriété de la façon suivante :

```
tete=rand()%TASMAX;
for (i=tete; suiv[i]!=tete; i=suiv[i])
    printf("%d",tas[i]);
```

2. Test 1 : parcourir un tableau avec ses propres valeurs

Il s'agit ici tester un tableau d'indices sur lui-même afin de pouvoir le parcourir avec ses propres valeurs de façon circulaire. Le tableau a une taille BMAX et il est initialisé avec les valeurs d'indice de 0 à BMAX-1. Il n'y a aucun doublon, l'entrée est indifférente et à la fin nous devons toujours retrouver le début après avoir passé par toutes les cases du tableau.

a. Suite naïve incomplète

Pour atteindre cet objectif la première idée que nous proposons est d'initialiser le tableau avec des valeurs toutes différentes comprises entre 0 et BMAX-1. Pour ce faire, l'idée est de doubler le tableau par un tableau de booléens de même taille. Chaque position servira à indiquer si la position à déjà été utilisée 1 ou pas encore 0. Au départ toutes les postions du tableau de contrôle sont initialisées à 0. Ensuite BMAX valeurs aléatoires modulo BMAX qui restent dans la fourchette 0-BMAX-1 sont tirées. Pour chacune nous regardons si elle est déjà utilisée. Si oui plutôt que de refaire un tirage aléatoire jusqu'à en trouver une bonne, nous partons de cette position pour trouver la première disponible soit en avançant avec un pas de 1 soit

en reculant avec un pas de -1. Le pas est lui-même à chaque fois initialisé avec de façon aléatoire. Dés qu'une position libre est trouvée, elle est affectée à la position courante du tableau et sa position dans le tableau de contrôle passe à 1. La fonction reçoit le tableau à initialiser en paramètre, ce qui donne :

```
void init_liste_naive(int t[BMAX])
{
int cntl[BMAX]={0};
int i,suiv, pas;

   // pour chaque position du tableau
   for (i=0; i<BMAX; i++){

      // un pas pour chercher au besoin une valeur d'indice
      // aléatoire correcte
      pas=rand()%2*2-1;

      // indice aléatoire
      suiv=rand()%BMAX;

      // si il est utilisé en chercher un autre
      while (cntl[suiv]==1)
         // formule pour rester entre 0 et BMAX-1
         suiv=(suiv+pas+BMAX)%BMAX;

      // une fois trouvée l'affecter dans le tableau
      t[i]=suiv;

      // et le marquer comme utilisée
      cntl[suiv]=1;

   }
}
```

Qu'est ce que ça donne ? le schéma suivant en fait une simulation :

indices :	0	1	2	3	4	5	6
valeurs :	1	2	4	5	0	6	3

```
Parcours avec tête à 2 :      2, 4, 0, 1, 2
manque passages en 3, 5, 6

Parcours avec tête à 5 :      5, 6, 3, 5
manque passages en 0, 1, 2, 4

etc.
```

Cette idée ne permet pas de faire une liste qui passe par toutes les cases avec une entrée aléatoire. La taille de la liste dépend du moment où la tête de la liste est retrouvée. Comment résoudre cette question ?

b. Suite complète, circulaire

Pour éviter ce problème de boucle trop courte, nous sommes obligés pour construire la liste de préciser une tête au départ, c'est à dire l'indice de départ et de rendre cette valeur inutilisable pour la suite. La liste construite de la sorte assure que quelque soit la tête choisie il faudra passer par toutes les cases du tableau

avant d'arriver à la fin de la liste. Reprenons notre simulation avec par exemple au départ la tête à 2. Cette valeur est interdite pour les choix avenir et elle sera donnée comme valeur finale. Nous pouvons avoir par exemple :

départ :
tête = 2

indices :	0	1	2	3	4	5	6
valeurs :	1	5	4	2	0	6	3

```
Parcours avec tête à 2 :      2, 4, 0, 1, 5, 6, 3, 2
Fin boucle avec retour début, complet

Parcours avec tête à 5 :      5, 6, 3, 5, 2, 4, 0, 5
Fin boucle avec retour début, complet

etc.
```

Les modifications à apporter à la fonction précédente sont les suivantes :

- au départ décider d'une tête et mettre cette position à occupée dans le tableau de contrôle. Cette position est conservée dans deux variables comme tête et position courante.
- Ensuite l'algorithme est le même que précédemment avec un détail important supplémentaire : la position courante n'est plus donnée par le compteur de la boucle for mais par la variable courante qui prend à chaque tour la valeur du nouvel indice de suivant trouvé.
- A la sortie de la boucle for, la dernière position de la suite prend la valeur de la tête donnée initialement pour avoir une liste circulaire. Pour avoir une liste non circulaire il suffit de donner la valeur de fin BMAX.
- Pour finir la valeur de la tête est retournée au contexte d'appel

Ces modifications donnent la fonction suivante :

```
int init_liste_complete(int t[BMAX])
{
int tete,suiv,cmpt,courant,pas;
int cntl[BMAX]={0};
   // décider indice de tête, qui donne aussi première positon
   // courante
   courant=tete=rand()%BMAX;
   // marquer cet indice comme pris
   cntl[tete]=1;
   for (cmpt=1; cmpt<BMAX; cmpt++){
      // obtenir chaque suivant parmi les possibles
      suiv=rand()%BMAX;
      pas=rand()%2*2-1;
      while(cntl[suiv]==1)
         suiv=(suiv+pas+BMAX)%BMAX;
      // une fois qu'un suivant correcte est obtenu
      // le marquer comme déjà utilisé
      cntl[suiv]=1;
      // l'affecter à la position courante
      t[courant]=suiv;
      // prendre la position suivante
      courant=suiv;
   }
```

```
   // à la fin boucler le dernier sur le premier pour avoir
   // une liste circulaire (ou mettre à BMAX)
   t[courant]=tete;
   return tete;
}
```

c. Affichage tableau

Pour afficher le tableau, le parcourir et afficher les valeurs qu'il contient de l'indice 0 à l'indice BMAX-1. La fonction n'a besoin que du tableau en paramètre et bien sur de connaître la taille du tableau :

```
void affiche_tab(int t[BMAX])
{
int i;
   for (i=0; i<BMAX; i++)
      printf("t[%2d]=%2d\n",i,t[i]);
   putchar('\n');
}
```

d. Affichage liste

Pour afficher la liste la fonction a besoin de connaître quelle est la tête à partir d'où commence la liste pour pouvoir l'afficher. La tête est donc donnée en paramètre. Ensuite le plus simple est d'afficher tout de suite la tête puis les éléments suivants de la liste avec une boucle for de la façon suivante :

```
void affiche_liste(int suiv[BMAX], int tete)
{
int i;
   printf("suiv[%2d]=%2d\n", tete, suiv[tete]);
   for (i=suiv[tete];i!=tete; i=suiv[i])
      printf("suiv[%2d]=%2d\n", i, suiv[i]);
}
```

e. Test in main()

Le menu propose de tester les deux fonctions d'initialisation de suite, celle qui ne fonctionne pas complètement et l'autre. Pour chacune il y a la possibilité d'afficher le tableau indépendamment de la liste et la liste constituée :

```
int menu()
{
int res=-1;
   printf(  "1 : init liste naive\n"
            "2 : affiche tableau\n"
            "3 : affiche liste\n"
            "4 : init liste complete\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Voici dans le main() les déclaration des variables têtes et tableau et les appels des fonctions qui correspondent aux actions du menu : :

```
int main()
{
int tete,fin=0;
int suiv[BMAX];

   while (!fin){
      switch(menu()){
         case 1 :
            init_liste_naive(suiv);
            break;
         case 2 :
            affiche_tab(suiv);
            break;
         case 3 :
            // affichage en liste avec tête aléatoire
            affiche_liste(suiv,rand()%BMAX);
            break;
         case 4 :
            tete=init_liste_complete(suiv);
            break;
         default : fin=1;
      }
   }
    return 0;
}
```

3. 4.3 Test 2 : produire des parcours aléatoires pour un tas

Nous allons appliquer l'algorithme précédent de génération de suites pour produire des parcours aléatoires d'un tas. L'intérêt va être d'avoir plusieurs ordres possibles pour le parcours d'un bloc d'informations. Dans une application ce peut être plusieurs présentations et tris des données accessibles simultanément et tenus en réserve dans l'application.

a. Structure de données

Nous reprenons l'élément de base utilisé pour les études précédentes. Un champ pour une valeur entière, un champ pour une chaine de caractères. La différence c'est que les listes sont maintenant extériorisées et le champ suiv est devenu inutile au niveau des datas, ce qui donne la structure suivante :

```
typedef struct elem{
   int val;
   char s[80];
}t_elem;
```

La taille du tas est donnée par une macro et un type TAS est créé :

```
#define BMAX   16
typedef t_elem TAS[BMAX];
```

Le tas lui-même sera déclaré dans le main(à) sous la forme :

```
TAS t;
```

Un tableau de chaines de caractères est utilisé en global juste à titre illustratif afin d'avoir des valeurs possibles pour les champ s des éléments du tas :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

Notre objectif est de fournir des parcours aléatoires sans les stocker, pour ce faire il suffit d'une tête de liste et d'un ensemble de suivants de la taille du tas :

```
int tete,suiv[BMAX];
```

b. Initialisation du tas

Au départ chaque élément du tas est initialisé avec son numéro d'ordre dans le tas et une lettre choisie au hasard dans le tableau global S.

```
void init_tas(TAS t)
{
int i;
   for (i=0; i<BMAX; i++){
    t[i].val=i;
    strcpy(t[i].s,S[rand()%26]);
   }
}
```

c. Création d'un parcours

La création d'un parcours reprend exactement la fonction vue précédemment dans le test 1, init_liste complete() sauf qu'elle n'est pas circulaire. A la fin la dernière position de la liste prend la valeur BMAX :

```
int creer_parcours(int p[BMAX])
{
int tete,suiv,cmpt,courant,pas;
int cntl[BMAX]={0};

   courant=tete=rand()%BMAX;
   cntl[tete]=1;
   for (cmpt=1; cmpt<BMAX; cmpt++){
      suiv=rand()%BMAX;
      pas=rand()%2*2-1;
      while(cntl[suiv]==1)
         suiv=(suiv+pas+BMAX)%BMAX;
      cntl[suiv]=1;
      p[courant]=suiv;
      courant=suiv;
   }
   p[courant]=BMAX;    // ce n'est pas une liste circulaire
   return tete;
}
```

d. Affichage du tas

Il s'agit d'afficher chaque élément du tableau du tas. Seul le paramètre TAS est nécessaire à la fonction d'affichage de tas :

```
void affiche_tas(TAS t)
{
```

```
int i;
   for  (i=0; i<BMAX; i++)
      printf("%2d%s--",t[i].val,t[i].s);
   putchar('\n');
}
```

e. Affichage du parcours

Afficher le parcours nécessite d'avoir la tête et la suite c'est à dire le tableau d'entiers suiv. Ensuite c'est un parcours de liste ordinaire, à partir de la tête prendre chaque position suivante arriver à la fin avec la valeur BMAX. Pour chaque position i il faut afficher l'élément correspondant du tas, ce qui donne :

```
void affiche_parcours(TAS t, int suiv[BMAX], int tete)
{
int i;

   for (i=tete; i!=BMAX; i=suiv[i])
      printf("%2d%s--",t[i].val,t[i].s);
   putchar('\n');
}
```

f. Test in main()

Il y a uniquement deux actions proposées : afficher le tas et afficher un parcours :

```
int menu()
{
int res=-1;
   printf(  "1 : affiche tas original\n"
            "2 : affiche un parcours\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

L'initialisation du tas a lieu juste avant la boucle d'événement. Pour chaque affichage de parcours un nouveau parcours est créé :

```
int main()
{
int tete,fin=0;
int suiv[BMAX];
TAS t;

   init_tas(t);
   while (!fin){
      switch(menu()){
         case 1 :
            affiche_tas(t);
            break;
         case 2 :
            tete=creer_parcours(suiv);
            affiche_parcours  (t, suiv, tete);
            break;
         default : fin=1;
      }
```

```
   }
    return 0;
}
```

4. 4.4 Test 3 : faire un tableau d'indices ordonnés

L'objectif maintenant est de constituer une suite d'indices rangés normalement dans un tableau (de 0 à N) qui permette d'avoir un parcours ordonné du tas. Par exemple pour avoir une suite d'éléments classés en en ordre croissant nous aurons à l'indice 0 l'indice de l'élément du tas ayant la plus petite valeur entière, à l'indice 1 l'indice de l'élément du tas ayant la plus petite valeur entière suivante etc jusqu'à ce soient classés tous les éléments du tas en fonction chacun de la valeur entière qu'il a.

	0	1	2	3	4	5	6	7
Tas	45	345	32	145	59	9	19	23

Tableau des indices rangés pour obtenir une suite croissante :

	0	1	2	3	4	5	6	7
suite	5	6	7	2	0	4	3	1

L'ordre de la suite correspond à l'ordre du tableau. Pour parcourir le tas selon cette suite ça donne :

```
   for (i=0; i<8; i++)
   printf("%d ",Tas[ suite[i] ]);
```

La difficulté vient uniquement du tri, comment trier un tableau sans rien bouger ? C'est ce que nous allons faire dans l'étude ci-dessous.

a. Structure de données

La structure de donnée est la même que dans l'étude précédente :

```
#define BMAX         16             // nombre total d'éléments du
tas

typedef struct elem{                // type d'un élément
   int val;
   char s[80];
}t_elem;
typedef t_elem TAS[BMAX];           // type tas, tableau de BMAX
éléments

char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

Notre application permet de trier par val en ordre croissant ou par ordre alphabétique ou encore les deux simultanément (plus petit OU plus prés de A passe avant)

Les trois types de tri sont identifiés par les macros :

```
#define TRINOM       0
```

```
#define TRIVAL       1
#define TRIVALNOM    2
```

b. Initialisation et affichage du tas

L'initiation affecte une valeur aléatoire entière comprise entre 0 et 26 pour le champ val de chaque élément et une lettre sous forme de chaine de caractères choisie aléatoirement dans l'alphabet fourni par le tableau global S pour le champ s :

```
void init_tas(TAS t)
{
int i;
   for (i=0; i<BMAX; i++){
    t[i].val=rand()%26;
    strcpy(t[i].s,S[i%26]);
   }
}
```

Les valeurs de chaque élément sont simplement affichées avec la fonction affiche_tas() qui reçoit le tas en paramètre :

```
void affiche_tas(TAS t)
{
int i;
   for  (i=0; i<BMAX; i++)
      printf("%2d%s--",t[i].val,t[i].s);
   putchar('\n');
}
```

c. Parcours ordonnés par tris

fonction de tri dérivée du tri par insertion. Comme il ne faut pas toucher aux données du tas, les données prises sont identifiées dans un tableau de booléen 1 prise, 0 disponible. La suite ordonnée des indices des éléments du tas est stockée dans un tableau d'entiers dynamique retourné au contexte d'appel. L'algorithme est le suivant :

- chercher la plus petite valeur dans les éléments disponibles du tas,
- stocker l'indice correspondant à la position courante de la suite ordonnée,
- cocher la position de l'élément dans le tableau de contrôle booléen
- avancer la positon courante de la suite de une position
- boucler sur le nombre total des éléments du tas

La fonction reçoit en paramètre le tas et permet plusieurs tris. Le type de tri souhaité doit être précisé au paramètre type au moment de l'appel par une des trois macros TRINOM, TRIVAL ou TRIVALNOM

```
int* parcours_tri(TAS t, int type)
{
int*liste;
int cntl[BMAX]={0};
int k,i,suiv;
   liste=(int*)malloc(sizeof(int)*BMAX);
   // correspond aux éléments de la liste
```

```
   for (k=0;k<BMAX;k++){
      // trouver élément suivant disponible du tas
      for (suiv=0; suiv<BMAX; suiv++)
         if (!cntl[suiv])
            break;
      // comparer avec les autres libres et sélectionner celui
      // souhaité selon critère du tri
      for (i=0; i<BMAX; i++){
         if (i==suiv) // si lui-même passer
            continue;
         if (!cntl[i])// si libre comparer
            switch(type){
               case TRINOM :
                  if (strcmp(t[i].s,t[suiv].s)<0)
                     suiv=i;
                  break;
               case TRIVAL :
                  if (t[i].val<t[suiv].val)
                     suiv=i;
                  break;
               case TRIVALNOM :
                  if (strcmp(t[i].s,t[suiv].s)<0 ||
t[i].val<t[suiv].val)
                     suiv=i;
                  break;
            }
      }
      // nouvel élément trouvé :
      cntl[suiv]=1;  // n'est plus disponible pour la suite
      liste[k]=suiv; // ajouter à la liste
   }
   return liste;
}
```

d. Affichage parcours ordonné

Une fois la liste ordonnée des indices du tas obtenue, elle est rangée en ordre dans le tableau liste et il est simple de l'afficher. La fonction reçoit en paramètre, le tas et la liste ordonnée :

```
void affiche_parcours_tri (TAS t, int p[])
{
int i,k;
   for (i=0; i<BMAX; i++){ // Pour chaque position de la liste
      k=p[i];  // récupérer de la position dans le tas
      printf("%2d%s--",t[k].val,t[k].s);
   // ou directement : printf("%2d%s--",t[p[i]].val,t[p[i]].s);
   }
}
```

e. Test dans le main()

Dans l'application test le menu propose d'initialiser et d'afficher le tas et les trois possibilités de tri, par valeur, alphabétique ou les deux à la fois

```
int menu()
{
int res=-1;
```

```
   printf(  "1 : initialiser et afficher tas\n"
            "2 : tri par valeur, affichage\n"
            "3 : tri alphabetique, affichage\n"
            "4 : tri val ou nom, affichage\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Chaque appel de la fonction tri récupère le tableau des indices ordonnés du tas dans le int*p. Ensuite le parcours trié est affiché avec la fonction affiche_parcours() et la mémoire allouée au tableau d'indices est libérée.

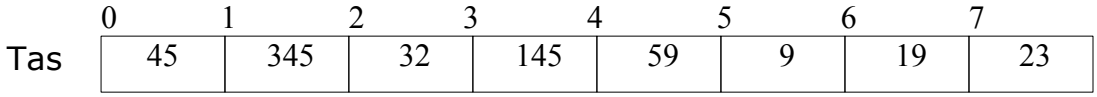
```
int main()
{
int fin=0;
int*p;
TAS t;
   srand(time(NULL));
   while (!fin){
      switch(menu()){
         case 1 :
            init_tas(t);
            affiche_tas(t);
            break;
         case 2 :
            p=parcours_tri(t,TRIVAL);
            affiche_parcours_tri (t, p);
            free(p);
            break;
         case 3 :
            p=parcours_tri(t,TRINOM);
            affiche_parcours_tri (t, p);
            free(p);
            break;
         case 4 :
            p=parcours_tri(t,TRIVALNOM);
            affiche_parcours_tri (t, p);
            free(p);
            break;
         default : fin=1;
      }
   }
    return 0;
}
```

5. Test 4 : faire une liste ordonnée

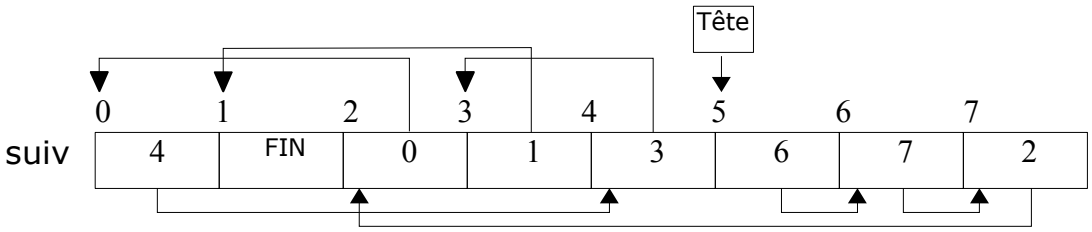
Dans l'étude précédente nous avons envisagé de créer une suite d'indices qui ordonne le tas en ordre croissant. Cette suite a été rangée dans un tableau normal de l'indice 0 à l'indice final du tableau de même taille que le tas. L'idée maintenant est de faire la même chose mais sur le modèle d'une liste chainée dans un tableau. Si le résultat est le même : une suite d'indices des éléments du tas qui correspond à un classement du tas en ordre croissant, le principe algorithmique est différent. Nous allons prendre l'indice de chaque élément du tas et en fonction de la valeur de l'élément correspondant l'insérer à chaque fois à la bonne place dans une liste

chainée. Au départ la liste chainée est vide et elle va se construire au fur et à mesure des insertions.

Soit le tas :



Voici la liste chainée d'indices pour obtenir une suite croissante :



Le parcours de la liste ne respecte pas l'ordre normal du tableau, il s'effectue de la façon suivante :

```
for (i= tete; i!=FIN; i=suiv[i])
   printf("%d ", tas[i]);
```

Pour réaliser cet algorithme nous aurons besoin essentiellement de quatre fonctions : une fonction d'insertion au début, une fonction d'insertion après un élément donné, une fonction de tri dans laquelle les deux précédentes sont utilisées et une fonction de comparaison qui intervient également dans le processus du tri.

a. Structure de données

Comme dans les études précédentes la macro BMAX donne le nombre d'éléments du tas. Chaque élément du tas est défini par une structure t_elem. Nous avons défini également pour le tas lui-même le type TAS qui correspond à un tableau de BMAX t_elem :

```
#define BMAX         16

typedef struct elem{
   int val;
   char s[80];
}t_elem;
typedef t_elem TAS[BMAX];
```

Pour cette étude nous avons défini également un type t_liste qui permet de regrouper toutes les variables nécessaires à une liste afin de faciliter l'utilisation de plusieurs listes en simultané dans le programme. La liste des pointeurs suiv a la même taille BMAX que le tas. Une liste complète nécessite une tête pour la liste des indices d'éléments et une tête pour la gestion des positions libres avec une liste libre.

```
typedef struct liste{
   int tete;          // tëte liste des indices d'éléments du tas
```

```
   int libre;          // tête liste des positions libres
   int suiv[BMAX];     // tableau des pointeurs d'indices ( des
entiers)
}t_liste;
```

Les listes sont toujours manipulées via des pointeurs t_liste* de sorte que nous aurons le programme les variables suivantes déclarées au sommet dans le main() :

```
TAS t;
t_liste*l=NULL;
```

Comme dans le programme précédent il y a trois sortes de tri possibles identifié chacun par une macro :

```
#define TRINOM       0
#define TRIVAL       1
#define TRIVALNOM    2
```

Et nous utiliserons également un tableau global de chaines de caractères qui donne les lettres de l'alphabet en majuscule :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

b. Initialisation et affichage du tas

Initialisation du tas et affichage sont identiques au programme précédent :

```
void init_tas(TAS t)
{
int i;
   for (i=0; i<BMAX; i++){
    t[i].val=rand()%26;
    strcpy(t[i].s,S[rand()%26]);
   }
}

void affiche_tas(TAS t)
{
int i;
   for  (i=0; i<BMAX; i++)
      printf("%2d%s--",t[i].val,t[i].s);
   putchar('\n');
}
```

c. Initialisation d'une liste

L'initialisation d'une liste suppose d'allouer de la mémoire pour un liste. Ensuite la liste d'élément est vide et sa tête prend la valeur de l'indicateur de fin BMAX. La première position libre est tout simplement la première position du tableau. En fait la gestion de la liste libre sera plus simple que celle que nous avons vue en 3.4 parce qu'il n'y aura pas de suppression dans la liste des éléments mais uniquement des ajouts. Les positions libres sont dans l'ordre des indices du tableau d'indices suiv, de sorte que pour passer à la position libre suivante il suffira de faire libre++.

A l'issue la fonction retourne une liste proprement initialisée :

```
t_liste* init_liste()
```

```
{
t_liste*l;
   l=(t_liste*)malloc(sizeof(t_liste));
   l->tete=BMAX; // indicateur de fin, liste vide
   l->libre=0;
   return l;
}
```

d. Insérer au début

Pour insérer au début le processus algorithmique est le suivant :

```
le suivant de la première position libre est la tête :
suiv[libre]=tete;

la tête prend la valeur de cette position libre :
tete=libre

et pour finir passer à la position libre suivante :
libre++
```

La fonction qui en découle prend l'adresse mémoire d'une liste en paramètre afin que les modifications faites à cette adresse soient retournées au contexte d'appel. Ensuite il y a juste à ajouter le contrôle par pointeur avec l'opérateur flèche -> devant les variables de l'algorithme, ce qui donne :

```
void inserer_debut(t_liste*l)
{
   l->suiv[l->libre]=l->tete;
   l->tete=l->libre;
   l->libre++;
}
```

e. Insérer après une position donnée

Pour insérer après une position id donnée le processus algorithmique est le suivant :

```
le suivant de la première position libre est le suivant de la
position id :
suiv[libre]=suiv[id];

ensuite le suivant de la position id devient cette position libre :
suiv[id]=libre;

pour finir passer à la position libre suivante :
libre++;
```

Comme pour insérer au début la fonction prend en paramètre l'adresse d'une liste. Elle prend en plus la position id après laquelle insérer, ce qui donne :

```
void inserer_apres(t_liste*l,int id)
{
   l->suiv[l->libre]=l->suiv[id];
   l->suiv[id]=l->libre;
   l->libre++;
}
```

f. Création listes ordonnées par tris

Comme nous l'avons mentionné plus haut, pour créer une liste ordonnée il suffit de prendre un par un les éléments du tas et de les insérer à chaque fois à la bonne place dans la liste en fonction du type de tri réalisé, par valeur, par odre alphabétique ou les deux simultanément. Pour ce faire la fonction liste_tri () a besoin en paramètre de recevoir le tas et que soit précisé le type de tri souhaité. Elle retourne à l'issue du traitement l'adresse d'une liste ordonnée allouée dans la fonction. L'algorithme est le suivant :

- La première chose est d'allouer une t_liste*l en utilisant la fonction init_liste() présentée plus haut :
l=init_liste();

- Il faut également insérer le premier élément du tas au début de la liste avec la fonction inserer_debut(). Nous aurions pu d'ailleurs avoir dans la fonction d'initialisation de liste le réglage : tête = 0, suiv[0]=BMAX et libre =1 puisqu'à chaque nouvelle liste ce sera la même chose au départ.

- Ensuite tant qu'il reste de la place dans la liste libre, c'est à dire tant qu'il reste des éléments dans le tas à classer, pour chaque élément il y a deux cas possibles : soit insérer en tête de liste avec la fonction insérer_début() soit trouver la position où insérer l'élément et l'insérer dans la liste après la position précédente avec la fonction insérer_après(). A chaque fois l'élément du tas à insérer est déterminé par la position équivalente à la première position libre de la liste libre. En effet la liste libre progresse régulièrement de un en un de 0 à BMAX-1 (à chaque insertion les fonctions inserer_debut() et inserer_apres() incrémente de 1 l'indice de la première position libre)

    -> L'élément est inséré au début si le résultat de la comparaison avec le premier élément de la liste en fonction du type de tri souhaité vaut 1. La comparaison est effectuée avec une fonction à part compare(). Cette fonction reçoit en paramètre les deux éléments à comparer ainsi que le type de tri qui détermine comment les comparer. Elle retourne le résultat, 0 ou 1 selon les cas.

    -> Sinon, il faut trouver la position en conservant toujours la position précédente qui va permettre l'insertion. A l'issue l'indice de l'élément est inséré à sa position dans la liste ordonnée.

- Pour finir retourner la nouvelle liste créée.

Ce qui donne la fonction :

```
t_liste* liste_tri(TAS t,int type)
{
t_liste*l;
int i,prec;
   // initialisation liste
   l=init_liste();
   // insérer premier (tete en 0, suiv en BMAX, libre en 1)
   inserer_debut(l);
   while(l->libre!=BMAX){
      // insérer en premier si le suivant t[libre] est <= t[tete]
      if (compare(t[l->libre],t[l->tete],type))
         inserer_debut(l);
      else{
         // sinon trouver position
```

```
         prec = i = l->tete;
        // avancer si le suivant t[libre] est >= à pos courante
        // t[i]
         while (i!=BMAX && compare(t[i],t[l->libre],type)){
            prec=i;
            i=l->suiv[i];
         }
         inserer_apres(l,prec);
      }
   }
   return l;
}
```

La comparaison se fait en fonction du type de tri passé en paramètre, c'est gérer par un switch qui renvoie à chaque type de tri. Le résultat des tests de comparaison est stocké dans un entier et renvoyé à la fin. Au moment de l'appel il faut bien faire attention à l'ordre des éléments qui sont testés.

```
int compare (t_elem e1, t_elem e2, int type)
{
int res=0;
   switch(type){
      case TRIVAL :
         res= (e1.val < e2.val);
         break;
      case TRINOM :
         res=(strcmp(e1.s,e2.s)<0);
         break;
      case TRIVALNOM :
         res= ( (e1.val <e2.val) || (strcmp(e1.s,e2.s)<0) );
         break;
   }
   return res;
}
```

g. Affichage liste ordonnée

L'affichage est très simple, juste un boucle for avec i qui commence avec la tête et fini à BMAX en prenant successivement toutes les valeurs de positions de la liste :

```
void affiche_liste(TAS t, t_liste*l)
{
int i;
   for (i=l->tete; i!=BMAX; i=l->suiv[i])
      printf("%2d%s--",t[i].val,t[i].s);
   putchar('\n');
}
```

h. Test dans le main()

Dans le main() le menu propose d'initialiser un tas et de créer des listes selon chaque possibilité de tri par valeur TRIVAL, alphabétique TRINOM, les deux associés TRIVALNOM :

```
int menu()
{
int res=-1;
   printf(  "1 : initialiser et afficher tas\n"
```

```
            "2 : liste par valeur, affichage\n"
            "3 : liste alphabetique, affichage\n"
            "4 : liste val ou nom, affichage\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

La liste de chaque tri est récupérée dans le contexte d'appel ensuite elle est affichée et détruite, ce qui donne :

```
int main()
{
int fin=0;
TAS t;
t_liste*l=NULL;

   srand(time(NULL));
   while (!fin){
      switch(menu()){
         case 1 :
            init_tas(t);
            affiche_tas(t);
            break;
         case 2 :
            l=liste_tri(t,TRIVAL);
            affiche_liste(t,l);
            free(l);
            break;
         case 3 :
            l=liste_tri(t,TRINOM);
            affiche_liste(t,l);
            free(l);
            break;
         case 4 :
            l=liste_tri(t,TRIVALNOM);
            affiche_liste(t,l);
            free(l);
            break;
         default : fin=1;
      }
   }
    return 0;
}
```

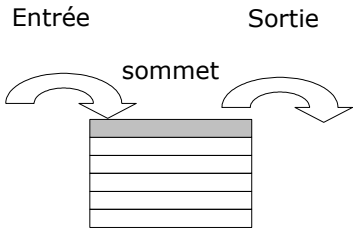
D. Piles

1. Principes de la pile

a. Modèle de donnée pile

La pile est une liste qui stocke de manière ordonnée les éléments. Ça marche comme une pile d'assiettes, les assiettes à laver sont empilées les unes par dessus les autres au fur et à mesure de leur arrivée et la personne qui lave les prends en commençant par le haut, ce qui fait que la dernière arrivée est la première lavée. De

même dans une pile informatique les éléments sont ajoutés un par un selon leur ordre d'arrivée et ils sont retirés un par un en partant du dernier arrivé appelé le sommet de la pile :



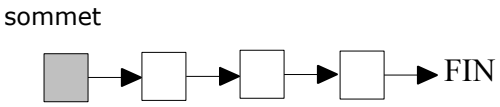
C'est la règle du dernier entré premier sorti : LIFO (last in first out).

b. Implémentations statique ou dynamique

Du point de vue de l'implémentation une pile est une liste simplifiée qui n'accepte que des entrées-sorties en tête de liste. De même que pour les listes nous pouvons avoir une pile en dynamique réalisée à la demande à l'aide de pointeurs ou une pile en mémoire contiguë réalisée selon une taille définie au départ avec un tableau.

Allocation dynamique de mémoire

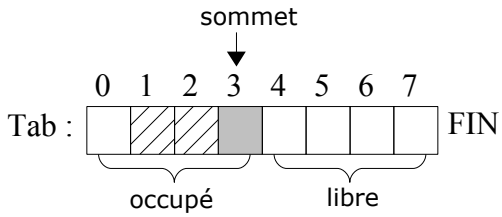
Représentation d'une pile en dynamique :



Une pile en dynamique n'a pas de taille limite, hormis celle de la mémoire centrale (RAM) disponible.

Allocation contiguë de mémoire

Représentation d'une pile en contiguë :



La taille d'une pile en contiguë nécessite d'être spécifiée à un moment donné et gérée s'il s'agit d'un tableau dynamique.

Éventuellement il est envisageable d'avoir une pile sur fichier. Dans ce cas la pile en contiguë n'a plus de limite en taille. Mais l'écriture est plus lourde et les accès disque sont plus lents.

c. Primitives de gestion des piles

Les opérations de base effectuées sur les piles sont les suivantes :

- Initialisation d'une pile : créer une pile vide
- Connaître si la pile est vide : renvoyer 1 si vrai 0 si non
- Connaître si la pile est pleine : renvoyer 1 si vrai 0 si non
- Lire sommet : récupérer les données du sommet de la pile sans retirer de la pile
- Empiler (push) : ajouter un nouvel élément au sommet de la pile
- Dépiler (pop): retirer l'élément sommet de la pile
- Vider la pile : retirer tous les éléments de la pile
- Détruire la pile : désallouer tout ce qui concerne la pile

d. Applications importantes des piles

En système les piles sont utilisées pour pour implémenter les appels de fonctions : ce sont les piles d'appels. En particulier lors d'appels récursifs se constitue une pile de récursion (ce point est abordé dans le chapitre sur la récursivité avec le principe : récursivité = itération + pile). Elles sont utilisées également dans l'évaluation d'expression arithmétiques. Dans toute situation modélisée ou l'ordre d'entrée-sortie des données correspond à celui d'une pile...

2. Implémentation d'une pile en dynamique

L'implémentation d'une pile ne pose aucun problème particulier, c'est un cas simple de liste chainée avec un ensemble réduit de fonctions associées à sa gestion.

a. Structure de données

Nous reprenons notre élément de base utilisé pour les études de listes chainées à savoir deux champs de données, un entier val et une chaine de caractères s, additionné d'un pointeur sur un structure de même type pour l'implémentation de la liste.

```
typedef struct elem{
   int val;
   char s[80];

   struct elem*suiv;
}t_elem;
```

Tableau en global de chaine s de caractères pour l'initialisation aléatoire du champ s :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

b. Pile vide, pile pleine

Si une pile dynamique est vide ça signifie que le pointeur de tête est mis à NULL. Pour le savoir il suffit de faire un test sur la valeur du pointeur de tête. La fonction pile_vide retourne le résultat du test sur la valeur du pointeur de tete, premier de la liste, 1 si oui, 0 si non :

```
int pile_vide(t_elem*tete)
{
   return (tete==NULL);
}
```

La question de la pile pleine ne se pose pas avec une liste dynamique. Il n'y a en théorie pas de limite à la liste constituée. En pratique toutefois il pourrait arriver de dépasser la capacité mémoire de la machine. Mais c'est un cas limite qui ne concerne pas la pile. C'est plutôt au moment de l'initialisation d'un nouvel élément qu'il faudrait vérifier s'il reste assez de mémoire pour pouvoir le faire.

c. Initialisation

Comme pour une liste ordinaire le pointeur premier doit être où à NULL si liste vide où avec une adresse valide pour une liste non vide. Ainsi au départ une pile vide est mise à NULL.

Ensuite, au fur et à mesure des ajouts dans la pile chaque élément nouveau est initialisé comme dans le chapitre précédent pour une liste ordinaire :

```
t_elem*init_elem(int val, char s[])
{
t_elem*e;
   e=(t_elem*)malloc(sizeof(t_elem));
   e->val=val;
   strcpy(e->s,s);
   e->suiv=NULL;
   return e;
}
```

Par défaut le suivant est mis à NULL.

Remarque

Le retour du malloc() peut être testé pour savoir si une erreur c'est produite au moment de l'allocation mémoire. Le retour vaut NULL en cas d'erreur, par exemple s'il n'y a pas assez de mémoire disponible.

d. Empiler

Empiler c'est ajouter au début de la liste :

```
void empiler(t_elem**p,t_elem*e)
{
   e->suiv=*p;
   *p=e;
}
```

e. Lire le sommet

Le sommet de la pile est donné en permanence par le pointeur tête adresse de la liste. Pour lire le sommet il suffit de considérer les valeurs du premier élément à cette adresse.

f. Dépiler

Dépiler c'est retirer le premier élément de la liste et le supprimer de la liste. Le premier de la liste passe alors au suivant et l'élément retiré est retourné au contexte d'appel :

```
t_elem* depiler(t_elem**p)
{
t_elem*e=NULL;

   if(!pile_vide(*p)){
      e=*p;
      *p=(*p)->suiv;
   }
   return e;
}
```

g. Vider, détruire

Vider et détruire sont identiques dans une pile dynamique Il s'agit de dépiler un à un tous les éléments de la pile et de libérer la mémoire qu'ils occupent. A la fin il est important de bien initialiser le pointeur de tête à NULL :

```
void detruire_pile(t_elem**p)
{
t_elem*e;
   while (!pile_vide(*p)){
     e=depiler(p);
     free(e);
   }
   *p=NULL;
}
```

h. Affichage

Les deux fonctions d'affichage ci-dessous n'ont rien à voir directement avec les piles, mais elles permettent de visualiser les actions effectuées. La première permet d'afficher un élément seul. Elle sert pour voir un élément dépilé. Pour cette fonction attention de bien vérifier que l'élément à afficher n'est pas NULL. Ce peut être le cas lorsque la pile vide est néanmoins dépilée :

```
void affiche_elem(t_elem*e)
{
   if (e!=NULL)
      printf("%d%s",e->val,e->s);
   else
      printf("pas d'element a aficher");
   putchar('\n');
}
```

La seconde affiche toute la pile comme n'importe quelle liste :

```
void affiche_pile(t_elem*p)
{
   if(p==NULL)
      printf("pile vide");
   while(p!=NULL){
      printf("%d%s",p->val,p->s);
      p=p->suiv;
   }
```

```
   putchar('\n');
}
```

i. Test dans le main()

Les fonctions précédentes sont testées selon le menu suivant :

```
int menu()
{
int res=-1;
   printf(  "1  : empiler et affiche pile\n"
            "2  : depiler, affiche element\n"
            "3  : detruire pile, affiche pile vide\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Les choix de l'utilisateur sont effectués de la façon suivante :

```
int main()
{
int fin=0,id=0;
t_elem*e,*p=NULL;
   while(!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,S[(id++)%26]);
            empiler(&p,e);
            affiche_pile(p);
            break;
         case 2 :
            e=depiler(&p);
            affiche_elem(e);
            break;
         case 3 :
            detruire_pile(&p);
            affiche_pile(p);
            break;
         default : fin=1;
      }

   }
   return 0;
}
```

3. Implémentation d'une pile en statique (tableau)

a. Structure de données

Pour cette étude nous optons pour une pile générique, susceptible de fonctionner avec n'importe quel type de donnée. La taille du bloc est fixée avec une macro constante. Le bloc est un tableau statique de pointeurs génériques void*. La pile fonctionne avec une variable sommet n associée au tableau. Le type t_pile est défini de la façon suivante :

```
#define NBMAX  16    // la taille du bloc
typedef struct pile{
   int n;            // le sommet
   void* t[NBMAX];  // le bloc pour la pile
}t_pile;
```

L'élément est toujours le même sans le pointeur suiv inutile dans ce cas de figure :

```
typedef struct elem{
   int val;
   char s[80];
}t_elem;
```

Tableau en global pour l'initialisation du champ s des éléments :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

b. Initialisation

Initialiser la pile c'est allouer de la mémoire pour une structure t_pile et mettre à 0 le sommet ce qui signifie que la pile est vide, ensuite retourner l'adresse de la pile :

```
t_pile* init_pile()
{
t_pile*p;
   p=(t_pile*)malloc(sizeof(t_pile));
   p->n=0;
   return p;
}
```

Les éléments sont initialisés à part avec la fonction suivante mais nous pourrions avoir n'importe quel autre type d'élément pour cette pile générique :

```
t_elem*init_elem(int val, char s[])
{
t_elem*e;
   e=(t_elem*)malloc(sizeof(t_elem));
   e->val=val;
   strcpy(e->s,s);
   return e;
}
```

c. Pile vide, pile pleine

Avec une pile implémentée en tableau nous avons besoin de savoir si la pile est pleine pour tout ajout et si elle est vide pour pour toute suppression. Le sommet de la pile en est l'indicateur. Si le sommet est à 0 la pile est vide si le sommet est à NBMAX la pile est pleine. Nous pouvons nous contenter de faire ce test à chaque fois que nécessaire mais les deux noms des fonctions suivantes présentent l'avantage de formuler plus clairement la question dans un test. Chacune des fonctions retourne le résultat 1 ou 0 selon que le test est vrai ou faux :

```
int pile_vide(t_pile*p)
{
   return (p->n==0);
}
```

```
int pile_pleine(t_pile*p)
{
   return (p->n==NBMAX);
}
```

d. Empiler

Empiler c'est ajouter au début. Si la pile n'est pas pleine l'adresse e de l'élément à ajouter est affectée à la position sommet n de la pile et le sommet est avancé de une position :

```
void empiler(t_pile*p,void*e)
{
   if (!pile_pleine(p)){
      p->t[p->n]=e;
      p->n++;
   }
   else
      printf("pile pleine\n");
}
```

e. Lire le sommet

Lire le sommet consiste à retourner l'adresse de l'élément au sommet de la pile sans dépiler, c'est à dire sans retirer cet élément de la pile. Ce n'est possible que si la pile n'est pas vide :

```
void* sommet(t_pile*p)
{
void*e=NULL;
   if (!pile_vide(p))
      e = p->t[p->n-1];
   return e;
}
```

f. Dépiler

Dépiler consiste à retirer l'élément au sommet de la pile. Si la pile n'est pas vide récupérer l'adresse de l'élément au sommet et décrémenter le sommet de une position. A l'issue retourner l'adresse récupérée ou NULL si la pile est vide ;

```
void* depiler(t_pile*p)
{
void*e=NULL;
   if (!pile_vide(p)){
      p->n--;
      e=p->t[p->n];
   }
   return e;
}
```

g. Vider, détruire

Vider la pile c'est retirer tous ses éléments et libérer la mémoire allouée pour chacun. Il suffit là d'une boucle for pour libérer la mémoire de chaque élément et de mettre ensuite le sommet n de la pile à 0 :

```
void vider_pile(t_pile*p)
{
int i;
   for (i=0; i<p->n; i++)
      free(p->t[i]);
   p->n=0;

}
```

Détruire la pile c'est vider la pile et en plus libérer la mémoire allouée pour la pile elle-même. Le pointeur t_pile* doit être mis à NULL à l'issue :

```
void detruire_pile(t_pile**p)
{
   vider_pile(*p);
   free(*p);
   *p=NULL;
}
```

h. Affichage

Les fonctions d'affichage sont là uniquement pour tester le fonctionnement de la pile et visualiser les éléments mais elles ne font pas partie du fonctionnement d'une pile. La fonction affiche_elem() permet d'afficher un élément de type t_elem via une adresse mémoire t_elem*

```
void affiche_elem(t_elem*e) // permet de caster le void* en
t_elem*
{
   if (e!=NULL)
      printf("%d%s--",e->val,e->s);
   else
      printf("pas d'element");
}
```

L'affichage de la pile consiste à afficher tous les éléments de la pile en commençant par le sommet. Les cas de pile non initialisée ou vide sont pris en compte. Les adresses void* sont passées au paramètre t_elem* de la fonction affiche_elem(). Elles sont de ce fait castés en t_elem*, condition obligatoire pour avoir accès aux champs de ces structures.

```
void affiche_pile(t_pile*p)
{
int i;
   if(p==NULL)
      printf("pas de pile");
   else if (p->n<=0)
      printf("pile vide");
   else{
      for (i=p->n-1; i>=0; i--)
         affiche_elem(p->t[i]); // void* casté en t_elem*
   }
   putchar('\n');
}
```

i. Test dans le main()

Le menu propose les actions suivantes :

```
int menu()
{
int res=-1;
   printf(  "1  : empiler et affiche pile\n"
            "2  : depiler, affiche element\n"
            "3  : vider pile, affiche pile vide\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Elle sont mises en oeuvre dans le main() de la façon suivante :

```
int main()
{
int fin=0,id=0;
t_elem*e;
t_pile*p=NULL;

   p=init_pile();
   while(!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,S[(id++)%26]);
            empiler(p,e);
            affiche_pile(p);
            break;
         case 2 :
            e=depiler(p);
            printf("element retire :\n");
            affiche_elem(e);
            putchar('\n');
            break;
         case 3 :
            vider_pile(p);
            affiche_pile(p);
            break;
         default : fin=1;
      }
   }
   detruire_pile(&p);
   return 0;
}
```

4. Mise en pratique de piles

Exercice 1
Faire un programme qui affiche une liste chainée simple à l'envers en utilisant une pile.

Exercice 2
Donner le contenu de la pile pour chaque opération de la suite :
Q*UES***TI*ON*FAC***IL***E**.
Chaque lettre provoque un empilement et chaque astérisque un dépilement.

Faire de même avec la suite : EAS*Y*QUE***ST***IO*N***.

Exercice 3
Un fichier texte peut contenir des parenthèses( ), des crochets [ ] et des accolades { }. Ces éléments peuvent être imbriqués les uns dans les autres, par exemple : { a(bc[d])[{ef}(g)]}. Ecrire une fonction qui parcourt le fichier texte et détermine si le fichier est correctement parenthésé. A savoir parenthèses, crochets et accolades doivent être correctement refermés et imbriqués. Par exemple ({]}) et ({)} ne sont pas correctes.

Exercice 4
Faire un programme qui, à l'aide d'une pile, évalue l'expression arithmétique postfixée :
5 11 9 + 2 12 * - 8 / *.
Puis à l'aide d'une autre pile, transformez l'expression postfixée en son expression infixée (opérateur entre 2 opérandes le tout entre parenthèses).

Exercice 5
Les internautes utilisent un logiciel pour naviguer sur Internet. Parmi ces fonctions il y a celle qui permet de revenir sur une page déjà accédée via son adresse URL (http://www...). Cette fonction est activée à l'aide de l'icône représentant la flèche. Pour réaliser cette fonctionnalité, le navigateur conserve un historique des adresses URL de chaque page accédée de sorte à pouvoir y accéder de nouveau au besoin. Les adresses des pages sont mémorisées selon l'ordre d'accès du plus récent au moins récent. Après un certain temps, cet historique peut contenir la même page plus d'une fois. On souhaite pouvoir éviter les doublons : ne garder en mémoire qu'une seule version URL de chaque page multiple, ceci tout en préservant l'ordre d'accès du plus récent au moins récent.
Programmer une simulation.

1) Quelle structure de données envisagez-vous ?

2) Tester avec un programme qui initialise un historique d'adresses URL contenant des doublons.

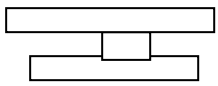
3) Comment évitez-vous les doublons dans l'historique ? Quelle solution pour référencer toutes les pages, y comprises multiples, sans garder leur adresse URL en doublon ?

Exercice 6
Nous disposons de trois piles P1, P2 et P3 pouvant contenir un nombre illimité d'objets.

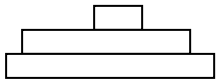
Au début, des objets de tailles différentes peuvent être empilés de manière désordonnée dans la première pile P1. Cette pile n'est soumise à aucune contrainte : un objet plus grand peut donc être empilé sur un plus petit. Nous supposerons donc que l'appel à la procédure EmpilerEnVrac(P1) empile de manière désordonnée plusieurs objets dans la pile P1.

On souhaite trier ces objets par taille dans la seconde pile P2, soumise quant à elle à la contrainte suivante : un objet plus petit ne peut être empilé que sur un objet plus grand. Pour cela, on dépile les objets de la première pile P1 un par un, et on les empile dans la seconde pile P2 en respectant cette contrainte d'ordre d'empilement.

Si la taille de l'objet en sommet de P1 est plus grande que celle en sommet de P2, la troisième pile P3 intermédiaire s'avère indispensable. Tout comme P1, cette troisième pile P3 n'est soumise à aucune contrainte d'ordre d'empilement.



première pile P1 désordonnée



seconde pile P2 triée

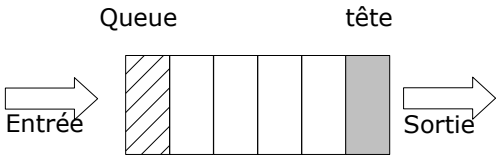
Faire un programme de simulation pour n objets entrés par l'utilisateur.

E. Files

1. Principes de la file

a. Modèle de donnée file

La file est une liste qui stocke les éléments avec un ordre spécifique. C'est comme une file d'attente à la caisse d'un magasin. Les personnes arrivent une à une et constituent la file et elles partent une à une dans l'ordre d'arrivée après avoir payé. Il y a la personne en tête de file, la première et la personne en queue de file, la dernière. C'est l'ordre du premier arrivé premier sorti, "first in first out" dit FIFO en informatique :

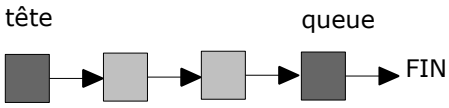


b. Implémentations statique ou dynamique

Une file est une liste particulière qui utilise les deux côtés début et fin à la différence d'une pile qui n'en utilise qu'un. L'implémentation peut se faire en dynamique avec des pointeurs ou en contiguë avec un tableau.

Allocation dynamique de mémoire

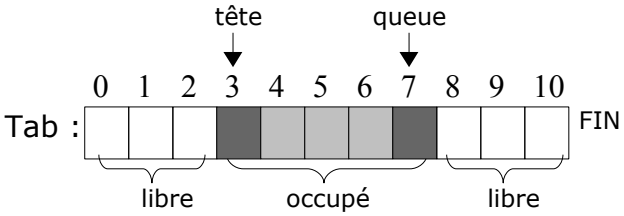
Représentation d'une file en dynamique :



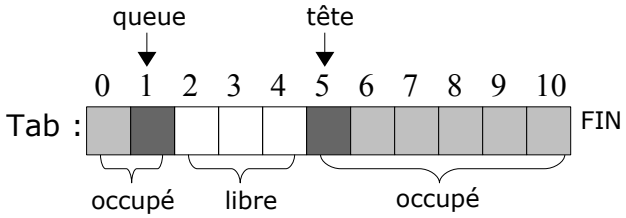
Il est nécessaire d'avoir deux pointeurs pour une file. Un pour gérer les entrées en queue et un pour gérer les sorties en tête. En dynamique une file n'a pas de taille limite, hormis celle de la mémoire centrale (RAM) disponible.

Allocation contiguë de mémoire

Représentation d'une file en contiguë :



La gestion d'une file en contiguë est moins aisée qu'elle n'en à l'air au premier abord. Il y a deux pointeurs d'indices, un pour la tête un pour la queue mais il faut utiliser le tableau de façon circulaire. Les nouveaux entrants peuvent être placés au début ou à la fin selon la place restante dans le tableau. Si nous ajoutons 5 entrées et en supprimons 2 dans le schéma ci dessus nous obtenons :



La taille d'une file en contiguë nécessite d'être spécifiée à un moment donné et gérée s'il s'agit d'un tableau dynamique. Éventuellement il est envisageable d'avoir une file sur fichier. Dans ce cas la file en contiguë n'a plus de limite en taille. Mais l'écriture est plus lourde et les accès disque sont plus lents.

c. Primitives de gestion des files

Les opérations de base effectuées sur les files sont les suivantes :

- Initialisation d'une file : créer une file vide
- Connaître si la file est vide : renvoyer 1 si vrai 0 si non
- Connaître si la file est pleine : renvoyer 1 si vrai 0 si non
- Lire tête : récupérer les données de la tête de la file sans retirer de la file
- Enfiler : ajouter un nouvel élément en queue de file
- Défiler : retirer l'élément en tête de file
- Vider la file : retirer tous les éléments de la file
- Détruire la file : désallouer tout ce qui concerne la file

d. Applications importantes des files

En système les files sont utilisées pour gérer tous les processus en attente de ressource système ou ans la lecture de certains fichiers fichiers multimédia (son en particulier). D'une façon générale elles interviennent dans un modèle dés qu'il est question d'une file d'attente : systèmes de réservation, gestion de pistes d'aéroport etc.

2. Implémentation d'une file en dynamique

Une file est une liste dont nous conservons en permanence le premier élément qui représente la tête de la file et le dernier élément qui représente la queue de la file. Une file est ainsi définie par deux pointeurs, celui de tête pour les sorties et celui de queue pour les entrées. A part cette spécificité c'est c'est un cas normal de liste chainée.

a. Structure de données

L'élément témoin est toujours le même :

```
typedef struct elem{
   int val;
   char s[80];
   struct elem*suiv;
}t_elem;
```

Initialisé de façon aléatoire avec le tableau global S de chaines de caractères pour le champ s :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

Le type file est défini par une structure qui comprend deux pointeurs, celui prem pour la tête qui gére les sorties d'éléments et celui dern pour la queue qui gère les arrivées d'éléments :

```
typedef struct file{
   t_elem*prem;  // sortie
   t_elem*dern;  // entrée
}t_file;
```

b. File vide, file pleine

Pour savoir si une file est vide il suffit de tester le dernier ou le premier, si NULL alors la file est vide. Notons qu'il n'est pas possible sauf erreur de programmation que l'un des deux soit NULL et l'autre pas :

```
int file_vide(t_file*f)
{
   return f->dern==NULL;
}
```

Il n'y a pas théoriquement (à part la mémoire sur la machine) de limite à la taille d'une file dynamique et donc pas de test sur file pleine.

c. Initialisation

L'initialisation d'un élément se fait comme dans les études précédentes, le champ suiv par défaut à NULL, la fonction reçoit les valeurs à affecter à l'élément en paramètre et retourne l'adresse valide d'un élément initialisé :

```
t_elem*init_elem(int val, char s[])
{
t_elem*e;
   e=(t_elem*)malloc(sizeof(t_elem));
   e->val=val;
   strcpy(e->s,s);
   e->suiv=NULL;
   return e;
}
```

Initialiser une file c'est allouer la mémoire nécessaire pour les deux pointeurs prem et dern d'une structure t_file et leur affecter la valeur NULL. La fonction retourne l'adresse valide d'une t_file

```
t_file*init_file()
{
t_file*f;
   f=(t_file*)malloc(sizeof(t_file));
   f->prem=f->dern=NULL;
   return f;
}
```

d. Enfiler

Enfiler un élément, c'est à dire ajouter un nouvel arrivant à la file c'est ajouter un élément en queue, à la fin au niveau du pointeur dern. La fonction reçoit en paramètre l'adresse de la file et l'adresse supposée valide de l'élément à ajouter. La connexion est très simple :

- le suivant du dernier devient le nouveau,
- le dernier devient le nouveau.

Si la file est vide le premier et le dernier prennent la même valeur, celle du nouvel élément, ce qui donne :

```
void enfiler(t_file*f,t_elem*e)
{
   // ajouter en queue (en dern)
   if (file_vide(f))// si file vide
      f->dern=f->prem=e;
   else{ // sinon ajouter à la fin
      f->dern->suiv=e;
      f->dern=e;
   }
}
```

e. Lire tête, lire queue

Récupérer les données du premier ou du dernier élément c'est récupérer les adresses des pointeurs prem pour la tête et dern pour la queue, sous réserve que la

file ne soit pas vide. Les fonctions retournent l'adresse valide de l'élément ou NULL en cas de file vide :

```
t_elem* lire_tete(t_file*f)
{
t_elem*e=NULL;
   if (!file_vide(f))
      e=f->prem;
   return e;
}

t_elem* lire_queue(t_file*f)
{
t_elem*e=NULL;
   if (!file_vide(f))
      e=f->dern;
   return e;
}
```

f. Défiler

Il s'agit de retourner l'adresse de l'élément sortant en tête et de passer la tête au suivant. Si la file est vide le retour est NULL. Deux cas sont à envisager :

- Si un seul élément dans la liste retourner son adresse et mettre les deux pointeurs prem et dern à NULL.
- Si plusieurs, retrouner l'adresse du premier et passer le premier prend l'adresse du suivant

Ce qui donne :

```
t_elem*defiler(t_file*f)
{
t_elem*e=NULL;
   if(!file_vide(f)){       // si non vide retirer en tête (en
prem)
      if (f->prem==f->dern){// si un seul élément
         e=f->prem;
         f->prem=f->dern=NULL;
      }
      else{                 // si plusieurs
        e=f->prem;
        f->prem=f->prem->suiv;
      }
   }
   return e;
}
```

g. Vider, détruire

Vider la file consiste à défiler tous les éléments jusqu'à ce que la file soit vide et à chaque fois libérer la mémoire allouée pour l'élément, ça donne :

```
void vider_file(t_file*f)
{
t_elem*sup;
   while(!file_vide(f)){
      sup=defiler(f);
```

```
      free(sup);
   }
}
```

Détruire la file c'est vider la file et libérer la mémoire allouée pour la file :

```
void detruire_file(t_file**f)
{
   vider_file(*f);
   free(*f);
   *f=NULL;
}
```

h. Affichage

Nous avons deux fonctions d'affichage, un pour afficher un seul élément et l'autre pour afficher la file. La fonction pour afficher un élément est la suivante, elle prend en compte le cas où l'élément est à NULL :

```
void affiche_elem(t_elem*e)
{
   if (e!=NULL)
      printf("%d%s",e->val,e->s);
   else
      printf("pas d'element");
   putchar('\n');
}
```

La fonction pour afficher la file prend en compte le ca où la file est vide et sinon elle affiche les valeurs de cahque élément en commençant par celui de tête :

```
void affiche_file(t_file*f)
{
t_elem*e;
    e=f->prem;
    if(file_vide(f))
      printf("file vide");
    while(e!=NULL){
      printf("%d%s--",e->val,e->s);
      e=e->suiv;
    }
    putchar('\n');
}
```

i. Test dans le main()

Dans le main() les opérations de test sont les suivantes :

```
int menu()
{
int res=-1;
   printf(  "1  : enfiler et affiche file\n"
            "2  : defiler, affiche element, file\n"
            "3  : vider file, affiche file vide\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Et voici la mise en application dans le main() :

```
int main()
{
int fin=0,id=0;
t_elem*e;
t_file*f;

   f=init_file();
   while(!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,S[(id++)%26]);
            enfiler(f,e);
            affiche_file(f);
            break;
         case 2 :
            e=defiler(f);
            affiche_elem(e);
            affiche_file(f);
            break;
         case 3 :
            vider_file(f);
            affiche_file(f);
            break;
         default : fin=1;
      }
   }
   detruire_file(&f);
   return 0;
}
```

Remarque

Dans tout le fonctionnement du programme la file est sensée être initialisée. C'est pourquoi elle est initialisée au départ juste avant la boucle d'événements et il n'est pas possible ensuite de la détruire pendant le fonctionnement. Elle est détruite en sortie lorsque le programme termine son exécution. Si on décide de pouvoir détruire la file pendant le fonctionnement du programme il faut alors ajouter dans toutes les fonctions qui utilise un t_file*f le test suivant pour éviter un accès à une adresse mémoire non valide :

```
if (f!=NULL){

          ...

}
```

3. Implémentation d'une file en statique (tableau)

a. Structure de données

Toujours l'élément elem pour les tests avec le champ suiv qui a disparu :

```
typedef struct elem{
   int val;
   char s[80];
}t_elem;
```

avec le tableau de chaines de caractères S en globale :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

Ce qui concerne la file est réuni dans une structure : un pointeur d'indice pour la tete de la file (prem), un pointeur d'indice pour l'indice de queue (dern) et un tableau pour les éléments de taille NBMAX défini par une macro constante. Nous avons opté pour un tableau de void*, générique de sorte que la file peut servir pour n'importe quel type d'objet :

```
#define NBMAX  8
typedef struct file{
   int prem,dern;
   void*tab[NBMAX];
}t_file;
```

b. File vide, File pleine

Il y a besoin de savoir si la file est vide pour défiler ou lire tête et queue. Nous savons que la file est vide si le premier à la même position que le dernier dans le tableau, c'est à dire si prem==dern :

```
int file_vide(t_file*f)
{
   return f->dern==f->prem;
}
```

Nous avons besoin de savoir si la file est pleine lorsque des éléments sont enfilés parce que la taille est fixe, limitée à NBMAX éléments. Le principe est que la file est pleine si le prochain retrouve le premier dans le tableau circulaire, après un tour complet du tableau en partant de n'importe quelle position. c'est à dire si : (dern+1) modulo NBMAX égal prem :

```
int file_pleine(t_file*f)
{
   return ((f->dern+1)%NBMAX == f->prem) ;
}
```

c. Initialisation

L'initialisation de la file consiste à allouer de la mémoire pour une structure t_file et retourner l'adresse du bloc. Les indicateurs de position prem et dern sont au départ tous les deux à 0

```
t_file* init_file()
{
t_file*f;
   f=(t_file*)malloc(sizeof(t_file));
   f->prem=f->dern=0;
   return f;
}
```

L'initialisation d'un élément ne change pas :

```
t_elem*init_elem(int val, char s[])
{
```

```
t_elem*e;
   e=(t_elem*)malloc(sizeof(t_elem));
   e->val=val;
   strcpy(e->s,s);
   return e;
}
```

d. Enfiler

Pour enfiler, si la file n'est pas pleine, il s'agit d'ajouter un élément en queue de file et d'avancer le pointeur d'indice dernier de un en contrôlant bien la circularité du tableau (arrivée à la fin égal repartir au début)

```
void enfiler(t_file*f,void*e)
{
   // ajouter en queue (en dern)
   if (!file_pleine(f)){
      f->tab[f->dern]=e;
      f->dern=(f->dern+1)%NBMAX;
   }
}
```

Remarque :

dern est toujours en avance de un par rapport au dernier élément effectif contenu dans le tableau. Il correspond à la position libre suivante pour le prochain arrivant. Le test de file pleine rajoute encore un pour savoir si la position encore d'après est différente de la tête. De sorte que la dernière case du tableau reste toujours libre et sert de sentinelle de fin. La solution qui consiste à dire la file est pleine si dern == prem pose tout un tas de petits problèmes qui alourdissent le code sans grand intérêt.

e. Lire tête, lire queue

Lire la tête c'est récupérer l'adresse du premier élément de la file sans changer la file :

```
void* lire_tete(t_file*f)
{
void*e=NULL;
   if (!file_vide(f))
      e=f->tab[f->prem];
   return e;
}
```

Lire la queue c'est récupérer l'adresse du dernier élément de la file sans rien changer à la file :

```
void* lire_queue(t_file*f)
{
void*e=NULL;
int q;
   if (!file_vide(f)){
      q=(f->dern-1+NBMAX)%NBMAX; //cntl si dern à 0
      e=f->tab[q];
   }
   return e;
```

```
}
```

f. Défiler

Défiler consiste, si la file n'est pas vide, à récupérer l'adresse de la tête et passer la tête à l'élément suivant :

```
void*defiler(t_file*f)
{
void*e=NULL;
   if(!file_vide(f)){ // si non vide retirer en tête (en prem)
      e=f->tab[f->prem];
      f->prem=(f->prem+1)%NBMAX;
   }
   return e;
}
```

g. Vider, détruire

Vider la file consiste à libérer la mémoire pour chaque élément et réinitialiser tête et queue sur la même position, 0 pour simplifier.

```
void vider_file(t_file*f)
{
t_elem*sup;
   while(!file_vide(f)){
      sup=defiler(f);
      free(sup);
   }
}
```

Détruire la file c'est vider la file et libérer la mémoire allouée pour la file elle-même sans oublier de mettre le pointeur de file à NULL :

```
void detruire_file(t_file**f)
{
   vider_file(*f);
   free(*f);
   *f=NULL;
}
```

h. Affichage

Pour afficher nous avons deux fonctions, une pour afficher la file et une pour afficher un élément. La fonction pour afficher la file affiche un par un tous les éléments de la file avec la fonction d'affichage d'un élément. Le cast nécessaire au pointeur générique void* pour l'accès aux données est ainsi réalisé au passage de paramètre t_elem* :

```
void affiche_file(t_file*f)
{
int e;
   if(file_vide(f))
      printf("file vide");
   for (e=f->prem;e!=f->dern;e=(e+1)%NBMAX)
      affiche_elem(f->tab[e]);
   putchar('\n');
```

```
}

void affiche_elem(t_elem*e)
{
   if (e!=NULL)
      printf("%d%s--",e->val,e->s);
   else
      printf("pas d'element");
}
```

i. Test dans le main()

Les actions proposées sont les mêmes que dans l'étude précédente :

```
int menu()
{
int res=-1;
   printf(  "1  : enfiler et affiche file\n"
            "2  : defiler, affiche element, file\n"
            "3  : vider file, affiche file vide\n"
          );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

De même pour les appels des fonctions dans la boucle d'événements du main() :

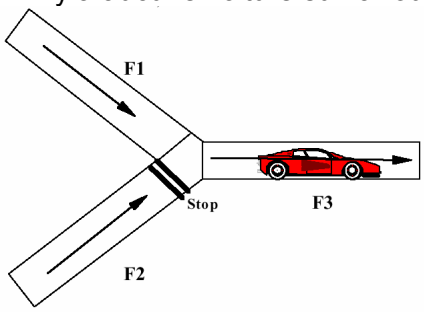
```
int main()
{
int fin=0,id=0;
t_elem*e;
t_file*f;

   f=init_file();
   while(!fin){
      switch(menu()){
         case 1 :
            e=init_elem(rand()%26,S[(id++)%26]);
            enfiler(f,e);
            affiche_file(f);
            break;
         case 2 :
            e=defiler(f);
            affiche_elem(e);
            putchar('\n');
            affiche_file(f);
            break;
         case 3 :
            vider_file(f);
            affiche_file(f);
            break;
         default : fin=1;
      }
   }
   detruire_file(&f);
   return 0;
}
```

4. Mise en pratique de files

Exercice 1
Donner le contenu de la file pour chaque opération de la suite : Q*UES***TU*ON*FAC***IL***E**.
Chaque lettre provoque un enfilement et chaque astérisque un défilement.

Exercices 2
Dans une gare un guichet est ouvert. Les clients arrivent à des dates aléatoires et font la queue. L'interval entre l'arrivée de deux clients successifs est un nombre aléatoire entre 0 et INTERVAL_MAX en secondes ou en minutes. Lorsque le guichetier a fini avec un client il appelle le client suivant et le traitement a une durée entre 0 et DUREE_TRAITEMENT_MAX.
Faire un programme de simulation.

Exercice 3
Pour simuler un croisement routier, à sens unique, on utilise 3 files f1, f2 et f3 représentant respectivement les voitures arrivant sur les routes R1 et R2, et les voitures partant sur la route R3. La route R2 a un STOP, les voitures de la file f2 ne peuvent avancer que s'il n'y a aucune voiture sur la route R1, donc dans la file f1.



06RI01

L'algorithme de simulation utilisera une boucle sans fin.
A chaque itération, il sera fait un appel à la procédure arrivée(f1, f2) qui simule l'arrivée d'une ou plusieurs voitures des files f1 et f2, modifiant ainsi leur état en mémoire.

- □ Si l'on considère que les files sont infinies quelle structure de données choisir ?
- □ Admettons que les files ne sont pas infinies. La taille de nos files est limitée à une variable MAX saisie par l'utilisateur et symbolisant le maximum de voitures que peut accueillir une route et la procédure arrivée(f1, f2) prend en compte cette nouvelle hypothèse.

Programmer une simulation.

Nous ajoutons maintenant une nouvelle hypothèse à notre problème : le STOP est respecté mais la voiture de la route R2 peut être prioritaire par rapport à la route R1. C'est à dire que si la distance entre la première voiture de la route R1 et le croisement est jugé suffisante par votre simulateur, on préférera défiler f2 plutôt que f1. La vitesse des voitures est jugée constante.

Modifier le programme précédent en ajoutant cette précision.

Exercice 4
Le but de cet exercice est d'écrire un programme qui simule le déroulement d'une partie du jeu de la bataille.

Rappel des règles :

- On dispose d'un jeu de 32 cartes (4 couleurs et 8 puissances de carte).
- Chaque carte possède une couleur (COEUR, CARREAU, TREFLE ou PIQUE) et une puissance (SEPT, HUIT, NEUF, DIX, VALET, DAME, ROI ou AS). On utilisera les codes ASCII 3, 4, 5 et 6 pour représenter les symboles , ♣, ♦, ♥, ♠. Par exemple, la dame de coeur sera affichée D♥.

Le jeu est d'abord mélangé aléatoirement (avis aux arnaques !) pour ensuite être coupé en deux tas de 16 cartes. On en donne un à chaque joueur. La partie peut enfin commencer. Chaque joueur montre la carte au sommet de son tas. Le joueur qui a la carte de plus forte valeur ramasse sa carte et celle de son adversaire et les met sous son tas. En cas d'égalité, les deux cartes sont placées sur un tas d'attente avec, pour chaque joueur, une autre carte prise au sommet de son tas. Ensuite, la partie reprend. Le joueur qui remportera la manche suivante remportera non seulement les deux cartes en jeu mais également toutes celles qui se trouvent dans le tas d'attente.

Le perdant sera le joueur qui n'a plus de carte dans son tas et le vainqueur celui qui réalise le meilleur score à la fin de la partie.

Définissez des structures de données adaptées pour une carte, un tas de cartes et un jeu de cartes

Définir les joueurs. Commencer avec deux joueurs.

Quelles sont les actions à effectuer et dans quel ordre ?

Soigner l'affichage du déroulement de la partie.

Ensuite généraliser pour unre partie de bataille entre nb joueurs avec un jeu de carte imaginaire de nb carte avec nb couleurs.

Exercice 5

Ecrire un programme qui transforme une expression infixée (avec parenthèses) en notation postfixée (polonaise inversée). L'expression suivante :

3*(((12-3)/3)-1)

devra être traduite en :

3 12 3 – 3 / 1 - *

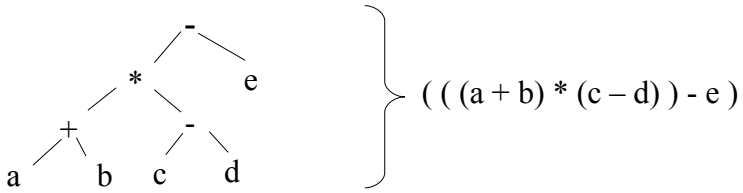
Les opérateurs valides sont : +, -, *, /. L'algorithme lit une suite de caractères et range le résultat de la conversion dans une file qui est affichée à la fin.

F. Introduction des arbres

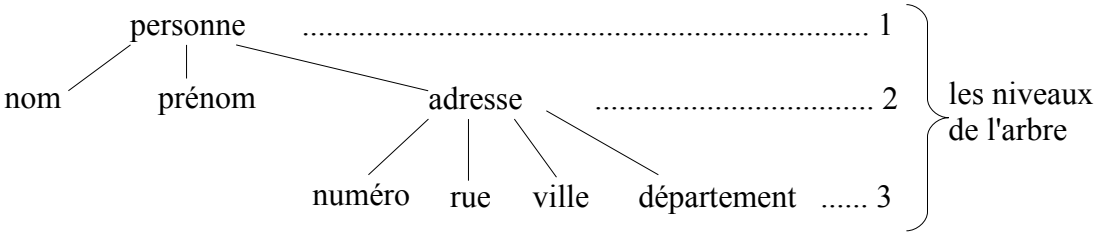
1. Généralités sur les arbres

a. Principe

Un arbre est une structure de données qui permet de stocker des informations et aussi une organisation des informations :



L'arbre est constitué de noeuds reliés entre eux de façon hiérarchique par des arrêtes. Le premier noeud est appelé racine les derniers, ceux après lesquels il n'y a plus de noeud sont les feuilles. Un parcours de la racine vers une feuille est une branche. Chaque noeud peut être considéré comme la racine d'un sous-arbre constitué de sa descendance et de lui-même. L'arbre est de ce fait une structure récursive. Dans l'exemple ci-dessus chaque sous-arbre correspond à un parenthésage. Une information parenthésée peut traduire un arbre. L'expression : (personne(nom, prénom, adresse(numéro, rue, ville, département))) est celle de l'arbre :



Chaque étage de l'arbre est appelé niveau et les niveaux sont numérotés de 1 la racine à n la feuille la plus basse. La hauteur d'un arbre c'est le niveau maximum atteint par une feuille. La hauteur de l'arbre ci-dessus est 3.

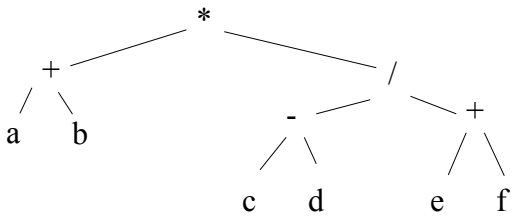
Souvent on parle de noeuds parents et de noeud fils. Les noeuds fils sont ceux qui descendent d'un noeud parent et seule la racine n'a pas de parent. Dans l'exemple ci-dessus, personne est la racine, il est parent de nom, prénom et adresse qui sont ses fils. Adresse est parent de numéro, rue, ville,département et numéro, rue, ville,département sont fils de adresse. Les fils d'un même parent sont frères

b. Exemples d'utilisations des arbres

Quelques uns des exemples suivants sont extraits du livre de Michel Divay *Algorithmes et structures de données* (1999).

Expression arithmétique

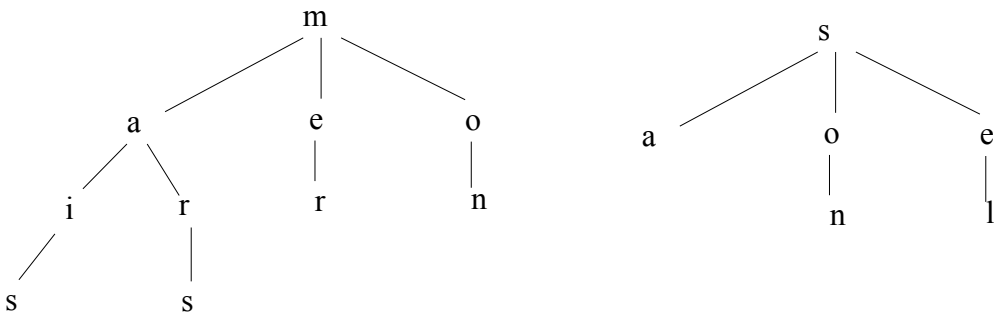
Une expression arithmétique, comme nous l'avons déjà mentionné peut être représentée par un arbre. L'expression : (a + b) * ((c - d) / (e + f)) donne l'arbre suivant :



Représentation de mots

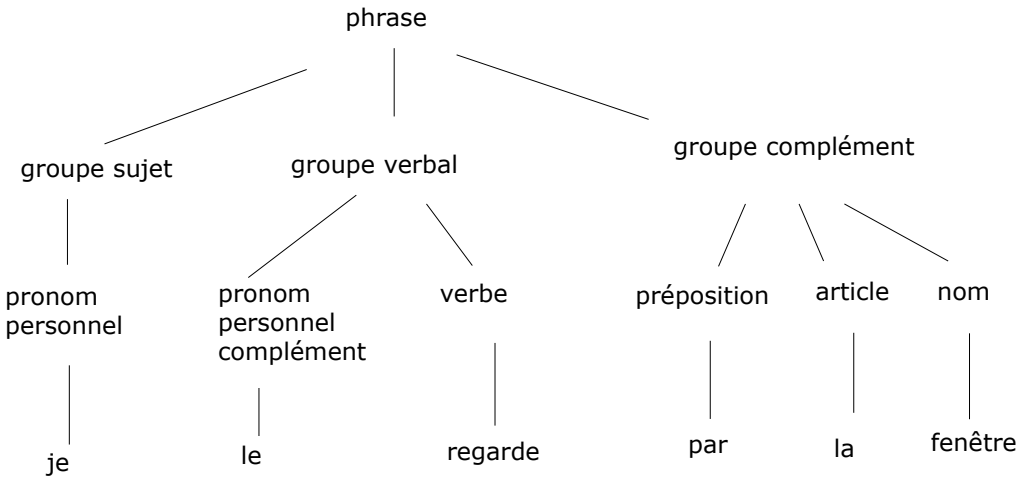
Voici deux arbres l'un pour représenter des mots qui commencent par m : mais, mars, mer, mon et l'autre des mots qui commencent par s : sa, son, sel. Chaque branche de l'arbre constitue un mot. Ces arbres peuvent également se noter avec des parenthèses :

(m(a(is, rs),er, on))) et (s (a, on, el))



Structure de phrase

Michel Divay donne également l'exemple suivant :



Et il en précise l'intérêt : la structure de la phrase ainsi modélisée en arbre peut être connue dans un programme informatique ceci afin d'atteindre des objectifs comme la traduction dans une autre langue, la prononciation via un synthétiseur vocal ou des aspects sémantiques dans la perspective par exemple de donner des ordres à un robot "poser la sphère rouge sur le cube bleu".

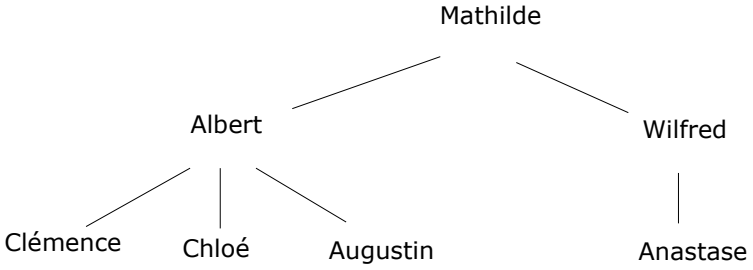
Arbre généalogique

Matilde a deux enfants : Albert et wilfred.

Albert a trois enfants : Clémence, Chloé, Augustin.

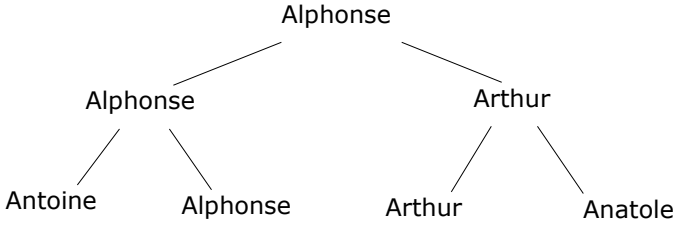
Wilfred a un enfant : Anastase.

L'arbre correspondant est le suivant :



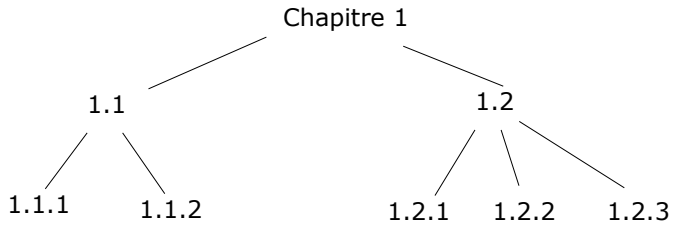
Résultat d'un tournois de tennis

Les matchs se déroulent des feuilles vers la racine qui désigne le vainqueur du tournois. La structure va par paire, chaque paire désigne une rencontre et le noeud parent le vainqueur du match.c'est un arbre binaire. Au départ du tournoi il y a quatre joueurs, il y a trois matchs :



Un chapitre de cours

Un sommaire de livre scientifique, un chapitre de cours sont construits selon une arborescence, par exemple :



Répertoire des fichiers d'un système d'exploitation

Dans l'arborescence du système de fichier les dossiers sont des noeuds et les fichiers des feuilles.

Nomenclature d'un objet

Par exemple l'arbre des composants d'une voiture (moteur, carrosserie, sièges etc.), la structure de la Terre (continents, pays, etc.) du corps humain (tête, tronc, membres etc.), d'un animal …

Du point de vue de la programmation objet, une classe et ses classes dérivées constituent également un arbre (voir chapitre suivant, découvrir C++)

Une classification

Par exemple la classification des espèces animales en vertébrés (poissons, batraciens, reptiles, oiseaux, mammifères) et invertébrés (arthropodes (crustacés, myriapodes, arachnides, insectes), vers ou mollusques) forme un arbre. etc.

c. Nomenclature des arbres

Racine
C'est le noeud "sommet" pointé par aucun autre

Niveau :
Le niveau de la racine est 1, les autres noeuds ont un niveau augmenté de 1 par rapport au niveau du noeud dont ils dépendent.

Une branche :
Une branche est constituée par le chemin à partir d'un noeud jusqu'à une feuille

Hauteur / profondeur d'un arbre :
Pour un arbre donné c'est le niveau atteint par la branche la plus longue.

Degré d'un noeud
C'est le nombre de fils d'un noeud.

Degré d'un arbre
C'est le degré maximum atteint par les noeuds de l'arbre.

Noeuds externes ou feuilles
Ce sont les derniers noeuds suivis par aucun autre.

Noeuds internes :
Ensemble des noeuds qui sont suivis par au moins un noeud.

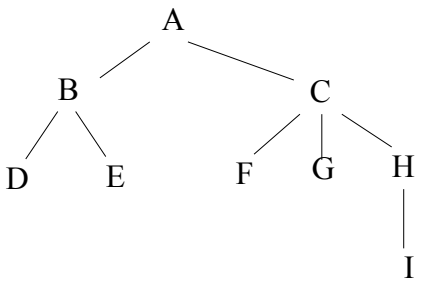
Taille de l'arbre :
Nombre total des noeuds de l'arbre.

Arbre ordonné :
L'ordre des sous-arbres est significatifs.

Arbre binaire :
Arbre ordonné de degré 2

Arbre binaire complet :
Arbre binaire de taille $2^h - 1$ ou h représente la hauteur de l'arbre

Exemple :



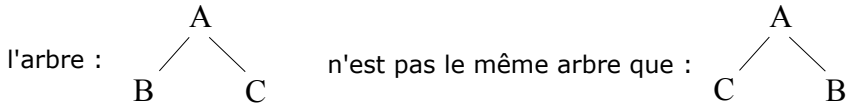
A est la racine.
B est parent de D et E
D et E sont enfants de B
D et E sont frères (ou siblings)
D, E, F, G, I sont les noeuds externes ou feuilles
A, B, C, H sont les noeuds internes
La hauteur (profondeur) de l'arbre est 4
E est à la profondeur 3
Le degré du noeud C est 3, du noeud B est 2

2. Deux types d'arbre

Les arbres sont divisés en deux catégories, les arbres binaires et les arbres n-aires.

a. Arbre binaire

Ce sont des arbres au plus de degré 2, c'est à dire dans lesquels chaque noeud a au plus deux fils. Les fils sont en général nommés fils gauche et fils droit. C'est un arbre ordonné, c'est à dire que les fils gauche et droite ne sont pas interchangeables :



Les arbres binaires sont majoritairement utilisés dans les algorithmes.

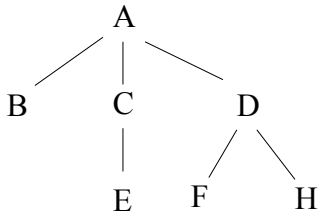
b. Arbre n-aire

C'est un arbre dans lequel au moins un noeud possède plus de deux fils, c'est à dire un arbre de degré supérieur à 2. Ces arbres sont en général peu utilisés dans les algorithmes parce qu'ils peuvent toujours être convertis en arbres binaires qui sont plus simples à manipuler.

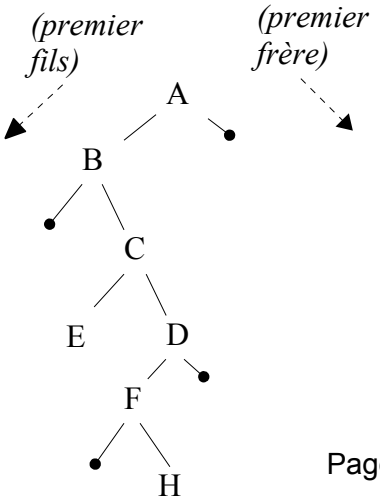
c. Transformer un arbre n-aire en arbre binaire

Un arbre binaire peut toujours être formé à partir d'un arbre n-aire en utilisant les liens premiers fils, frère immédiat. Pour chaque noeud de l'arbre n-aire, le fils gauche de l'arbre binaire est donné par le premier fils dans l'arbre n-aire et le fils droit par le premier frère de l'arbre n-aire, exemple :

L'arbre n-aire :



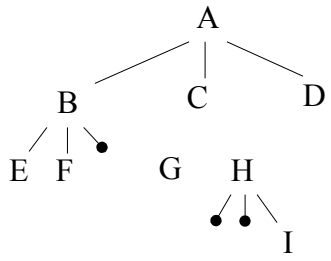
devient l'arbre binaire :



3. Représentations en mémoire

a. Arbre N-aire

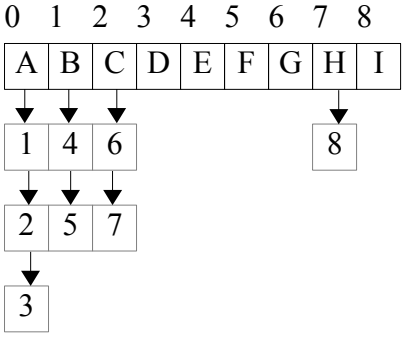
Soit l'arbre suivant :



Il peut être représenté en mémoire sous trois formes :

- tableau de listes chainées
- tableau de structures à deux dimensions
- dynamique à partir de pointeurs

Tableau de listes chainées :



Avec une structure de données de la forme :

```
#define NBMAX   9
typedef struct noeud{
          char data[16];
          struct noeud *suiv;
}t_noeud;
```

```
t_noeud*tab[NBMAX];
```

Le champ data représente plus largement le champ associé aux données rangées dans l'arbre, ici juste une petite chaine de caractères pour stocker des chaines du type "A", "B" etc. afin de réaliser des tests.

Tableau de structures ou à deux dimensions

	data	P1	P2	P3
0	A	1	2	3
1	B	4	5	-
2	C	6	-	7
3	D	-	-	-
4	E	-	-	-
5	F	-	-	-
6	G	-	-	-
7	H	-	-	8
8	I	-	-	-

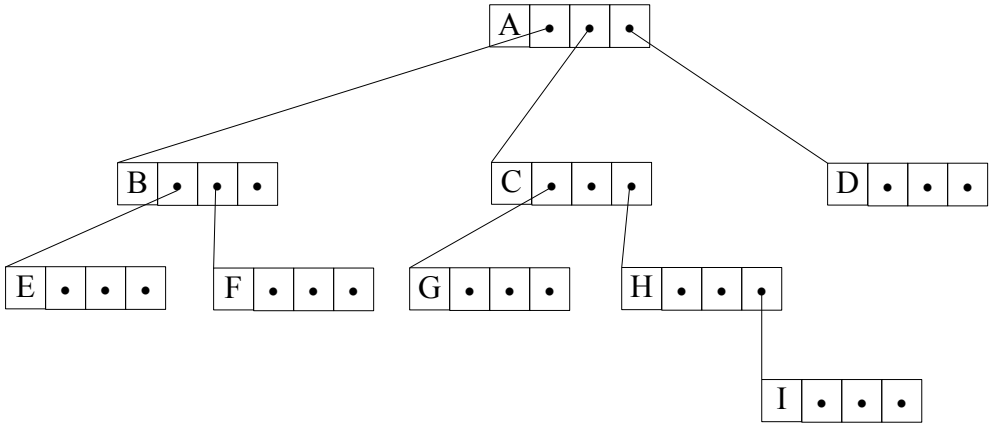
Avec une structure de données de la forme :

```
#define NBMAX   9
typedef struct noeud{
          char data[16];
          int P1,P2,P3;
}t_noeud;

t_noeud tab[NBMAX];
```

ou encore en séparant data des pointeurs d'indice :

```
char* data[NBMAX];
int mat[NBMAX][NP];    // avec NP pour le nombre de pointeurs,
                       // degré de l'arbre
```

Dynamique à partir de pointeurs

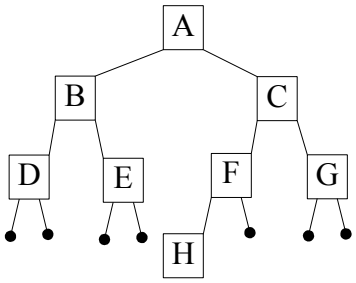


Avec une structure de données de la forme :

```
typedef struct noeud{
   char data[16];
   struct noeud*P1,*P2,*P3;
}t_noeud;
```

b. Arbre binaire

Soit l'arbre binaire suivant :



C'est un arbre à quatre niveau. Le nombre maximum possible de noeuds avec quatre niveaux est 2^4-1 noeuds, c'est à dire 16-1, 15 noeuds. Il faudrait pour ce faire que tous les noeuds du niveau trois possèdent deux fils, un à gauche, un à droite. Un tel arbre binaire est dit un arbre binaire complet.

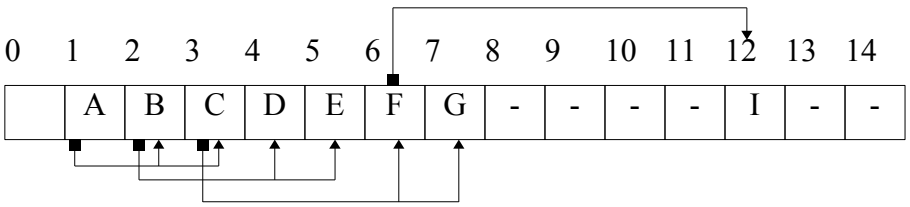
Tableau à une dimension

Un arbre binaire peut se ranger dans un tableau à une dimension avec la règle suivante :

Soit la racine à l'indice 1, quelque soit le noeud à l'indice i du tableau :

- le fils gauche de i est à i*2 (si dans le tableau)
- le fils droit de i est à i*2+1 (si dans le tableau

Pour notre arbre ci-dessus nous avons le tableau suivant :



Ce tableau sera probablement dynamique dans un programme, alloué en fonction de la taille maximum de l'arbre. La structure de données peut être :

```
typedef struct noeud{
   char data[16];            // les datas du noeud
   int g,d;                  // fils gauche et droite
}t_noeud;

t_noeud*tab;                 // tableau de noeud à allouer
dynamiquement
```

Construit à partir de pointeurs

L'arbre binaire peut bien sur être construit dynamiquement à partir de pointeurs. La structure de données est alors :

```
typedef struct noeud{
          char data[16];         //  les datas du noeud
          struct noeud*g,*d;     // fils gauche et droit
}t_noeud;

t_noeud*racine;           // adresse de l'arbre donnée par le
noeud racine
```

c. Structures de données statiques ou dynamiques

Comme pour les listes chainées il y a deux modes possibles pour l'écriture des arbres.

- En mémoire contiguë, c'est à dire en tableau ou sur fichier avec pour spécialité le stockage et la transmission de l'arbre ou des données de l'arbre. Les pointeurs sont alors des entiers, pointeurs d'indice ou indicateurs de position d'un noeud dans un fichier. La structure de données des arbres statique utilise des pointeurs d'indice :

```
typedef struct noeud{
   (..)                              // les datas du noeud

   int g,d;                          // fils gauche et droit
}t_noeud;
```

- Soit le mode dynamique avec pour spécialité un arbre destiné à évoluer, supportant des retraits et des ajouts fréquents. Les pointeurs sont alors des pointeurs ordinaires qui contiennent l'adresse mémoire d'un noeud.

```
typedef struct noeud{
   (..)                              // les datas du noeud

   struct noeud *g,*d; // fils gauche et droit
}t_noeud;
```

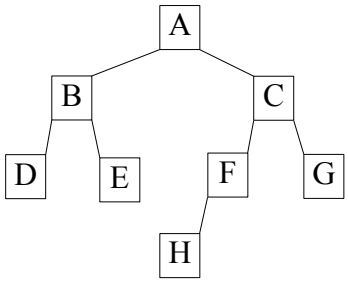
G. Contrôler un arbre binaire

1. Faire un arbre binaire

Les fonctions proposées ici permettent de créer des arbres soit à partir d'une description, soit totalement aléatoire. Notre objectif est de les utiliser ensuite pour tester les fonctions de gestion des arbres.

a. Créer un arbre à partir d'un schéma descriptif

Voici notre arbre :



Créer l'arbre "à la main",avec une fonction de création de noeud

Nous allons créer un arbre dynamique. Nous avons besoin d'un tableau de caractères pour stocker une chaine (ou éventuellement juste un caractère puisqu'il n'y a que des caractères à stocker) et de deux pointeurs, un pour le fils gauche et un pour le fils droit. La structure de données est la suivante :

```
typedef struct noeud{
   char dat[80];
   struct noeud*g, *d;
}t_noeud;
```

Voici une fonction pour initialiser un noeud de l'arbre :

```
t_noeud* CN(char s[],t_noeud*g, t_noeud*d)
{
t_noeud*n;
   n=(t_noeud*)malloc(sizeof(t_noeud));
   strcpy(n->s, s);
   n->g=g;
   n->d=d;
   return n;
}
```

Et la fonction pour créer l'arbre :

```
t_noeud* creerArbre1()
{
   return CN("A",
                  CN("B",
                        CN("D",NULL,NULL),
                        CN("E",NULL,NULL)),
                  CN("C",
                        CN("F",
                               CN("H",NULL,NULL),
                               NULL),
                        CN("G",NULL,NULL))
              );
}
```

Cette méthode n'est pas pratique et des erreurs avec les parenthèses sont difficiles à éviter.

Créer l'arbre en utilisant sa description et un tableau

Autre possibilité, toutes les adresses des noeuds de l'arbre sont allouées et stockées dans un tableau et ensuite l'arbre est construit :

```
t_noeud* creerArbre2()
{
const int nbNoeud=9;
t_noeud**tab;
t_noeud*r;
int i;
   // création du tableau de pointeurs
   tab=(t_noeud**)malloc(sizeof(t_noeud*)*nbNoeud);

   // allocation des adresses des noeuds
   for (i=0; i<nbNoeud; i++)
      tab[i]=(t_noeud*)malloc(sizeof(t_noeud));

   // initialisation de chaque noeud en fonction du schéma de
   // l'arbre
   strcpy(tab[0]->dat,"A");
   tab[0]->g=tab[1];
   tab[0]->d=tab[2];

   strcpy(tab[1]->dat,"B");
   tab[1]->g=tab[3];
   tab[1]->d=tab[4];

   strcpy(tab[2]->dat,"C");
   tab[2]->g=tab[5];
   tab[2]->d=tab[6];

   strcpy(tab[3]->dat,"D");
   tab[3]->g=NULL;
   tab[3]->d=NULL;

   strcpy(tab[4]->dat,"E");
   tab[4]->g=NULL;
   tab[4]->d=NULL;

   strcpy(tab[5]->dat,"F");
   tab[5]->g=tab[7];
```

```
   tab[5]->d=NULL;

   strcpy(tab[6]->dat,"G");
   tab[6]->g=NULL;
   tab[6]->d=NULL;

   strcpy(tab[7]->dat,"H");
   tab[7]->g=NULL;
   tab[7]->d=NULL;

   // la racine r de l'arbre est le premier élément
   r=tab[0];
   free(tab); // libération de l'adresse du tableau
   return r;  // retour de l'adresse de l'arbre
}
```

b. Créer un arbre à partir des données aléatoires d'un tableau

Pour faire des tests il peut être intéressant d'avoir facilement des arbres avec des valeurs et des tailles différentes. A cet effet nous allons faire une fonction de génération d'arbres binaires.

Le principe est simple, avoir un tableau, l'initialiser avec des valeurs aléatoires et construire un arbre binaire en utilisant la règle mentionnée en 1.5 au chapitre 1 :

Soit la racine à l'indice 1, quelque soit le noeud à l'indice i du tableau :

- le fils gauche de i est à i*2 (si dans le tableau)
- le fils droit de i est à i*2+1 (si dans le tableau

Nous pourrions bien sur prendre des noeuds au hasard dans le tableau et construire l'arbre de la sorte mais l'intérêt d'utiliser ce principe est d'éviter d'avoir à contrôler quels sont les noeuds restants disponibles dans le tableau pour construire l'arbre. En effet cela revient à prendre les noeuds dans l'ordre où il se présente dans le tableau pour construire l'arbre. Le tableau donne l'équivalent d'un parcours en largeur de l'arbre et la racine est toujours à l'indice 0.

Le champ dat de chaque noeud est initialisé avec une chaine de caractères choisie aléatoirement parmi les suivantes :

```
char* S[]={"A","B","C","D","E","F","G","H","I","J","K","L","M",
           "N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
```

Arbre dynamique

L'arbre dynamique se traduit par l'adresse du premier noeud qui est la racine de l'arbre. Tous les autres sont déductibles via le réseau de pointeurs fils gauche, fils droit.

```
t_noeud* creerArbre3()
{
t_noeud*r;
t_noeud**tab;
int i,k,taille;

   // taille aléatoire
   taille=1+rand()%100; // min 1 max 100 noeuds
```

```
   // création tableau de pointeurs de noeuds
   tab=(t_noeud**)malloc(sizeof(t_noeud*)*taille);

   // initialisation aléatoire des datas des noeuds
   for(i=0;i<taille;i++){
      tab[i]=(t_noeud*)malloc(sizeof(t_noeud));
      strcpy(tab[i]->dat,S[rand()%26]);
   }

   // construction de l'arbre, racine en tab[0]
   for (i=0,k=0; i<taille; i++){
      tab[i]->g=(++k<taille)?tab[k]: NULL;
      tab[i]->d=(++k<taille)?tab[k]: NULL;
   }
   r=tab[0];
   free(tab);
   return r;
}
```

Arbre statique

L'arbre statique est en fait un tableau de noeuds et l'arbre est défini dedans via un réseau d'indice fils gauche, fils droit. Dans la structure de données les pointeurs fils gauche et droit deviennent des pointeurs d'indice, c'est à dire des entiers qui correspondent à des indices du tableau :

```
typedef struct noeuds{
   char dat[80];       // les datas
   int g, d;           // pointeurs d'indices
}t_noeuds;
```

Ensuite l'algorithme de création de l'arbre statique est sensiblement le même que celui d'un arbre dynamique. La fonction retourne le tableau des noeuds et implicitement la racine est à l'indice 0 :

```
t_noeuds* creerArbre_stat()
{
t_noeuds*t;
int nb,i,k;
   nb=1+rand()%50;
   t=(t_noeuds*)malloc(sizeof(t_noeuds)*nb);
   for(i=0,k=0; i<nb; i++){
      strcpy(t[i].dat,S[rand()%26]);
      t[i].g=(++k<nb)?k:-1;
      t[i].d=(++k<nb)?k:-1;
   }
   return t;
}
```

Si nous souhaitons avoir une racine avec une position aléatoire il faut alors penser à la retourner via une variable passée par référence.

c. Créer un arbre en insérant des éléments ordonnés

L'objectif est ici de créer un arbre en ajoutant au fur et à mesure des nouveau noeuds dans l'arbre. Les nouveaux noeuds sont toujours ajoutés au niveau des feuilles (les noeuds suivis par aucun noeud), pas entre les noeuds internes. Ils sont

répartis selon les valeurs alphabétiques. L'algorithme est le suivant : si la valeur lexicographique de la lettre du nouveau noeud est avant celle du noeud courant partir à gauche sinon partir à droite et en partant de la racine jusqu'à arriver à une feuille. Pour pouvoir accrocher le nouveau noeud il est nécessaire de conserver pendant la descente l'adresse du noeud précédent afin d'avoir l'adresse de la feuille finale à laquelle accrocher le nouveau noeud qui devient une nouvelle feuille :

```
void inserer(t_noeud**racine,t_noeud*n)
{
t_noeud*x,*prec;
   // si pas de racine il devient racine
   if(*racine==NULL)
      *racine=n;
   // sinon descendre jusqu'à une feuille (fils g et d à NULL)
   else{
      x=*racine;// pour descendre
      prec=NULL;// pour conserver position précédente
      while(x!=NULL){
         prec=x;  // save position précédente
         x=(strcmp(n->dat,x->dat)<0)?x->g:x->d;// descente selon n
      }
      // à l'issue accrocher
      if (strcmp(n->dat,prec->dat)<0)
         prec->g=n;
      else
         prec->d=n;
   }
}
```

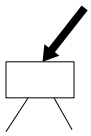
2. Parcourir l'arbre

Le parcours d'un arbre est important pour pouvoir consulter ses données et il y a deux modes de parcours des arbres, le parcours en profondeur et le parcours en largeur.

a. Parcours en profondeur

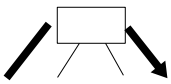
Pour le parcours en profondeur chaque noeud est visité trois fois. Il y a de ce fait trois possibilités pour consulter les données des noeuds : à l'aller, au premier retour ou au second retour ce qui donne en allant de la gauche vers la droite les trois parcours suivants :

préfixé :



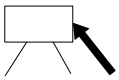
Les données sont visitées lors de la première rencontre du noeud, avant le parcours du sous-arbre gauche.

infixé :



Les données sont visitées après parcours de l'arbre gauche, avant parcours du sous arbre droit.

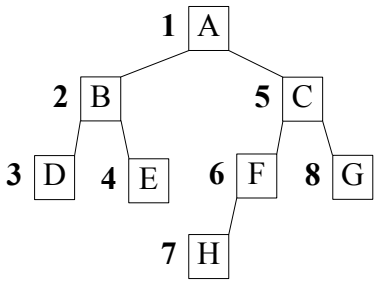
postfixé :



Les données sont visitées après les parcours du sous arbre gauche et du sous arbre droit.

C'est le même principe dans l'autre sens en commençant par la droite. Le parcours sera alors dans le sens inverse. Dans les exemples qui suivent nous présentons toujours des parcours gauche-droite.

Parcours en profondeur préfixé



Du point de vue algorithmique il y a deux possibilités : un algorithme récursif et une version itérative de cet algorithme. La version itérative utilise une pile. Ces algorithmes sont les mêmes pour un arbre statique ou un arbre dynamique mais l'écriture est légèrement différente Voici les deux algorithmes :

Algorithme récursif

Version dynamique

```
void prefixe(t_noeud*r)
{
   if(r!=NULL){
      printf("%s",r->dat);
      prefixe(r->g);
      prefixe(r->d);
   }
}
```

Version statique

```
void prefixe_s
       (t_noeuds t[],int r)
{
   if(r!=-1){
     printf("%s",t[r].dat);
     prefixe(t,t[r].g);
     prefixe(t,t[r].d);
   }
}
```

Algorithme itératif, gestion pile

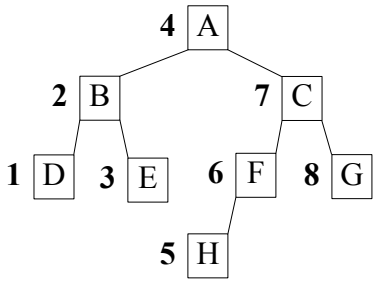
```
prefixe_iter(t_noeud*r)
{
   empiler(r);
   while(!pile_vide()){
      r=depiler();
      visiter(r);
   // attention inversion
   // droite d'abord
   // puis gauche
   if(r->d) empiler(r->d);
   if(r->g) empiler(r->g);
   }
}
```

La version itérative peut être mise en place sans développer à part un module de gestion de pile. Une pile c'est juste un tableau et un sommet et nous pouvons obtenir la taille de l'arbre (la fonction qui donne la taille de l'arbre, c'est à dire le nombre total de noeuds qui le compose, est détaillée au chapitre 2.4). Il reste à

allouer un tableau dynamique de noeuds et gérer l'indice de sommet qui avance de un si un élément est ajouté et recule de un si un élément est soustrait.

```
void prefixe_iter2(t_noeud*r)
{
t_noeud**pile;
int n=0;
   if(r==NULL)
      printf("arbre inexistant");
   else{
      pile=(t_noeud**)malloc(sizeof(t_noeud*)*taille_dyn(r));
      pile[n++]=r;
      while(n>0){
         r=pile[--n];
         printf("%d ",r->val);
         if (r->d)
            pile[n++]=r->d;
         if (r->g)
            pile[n++]=r->g;
      }
      free(pile);
   }
   putchar('\n');
}
```

Parcours en profondeur infixé

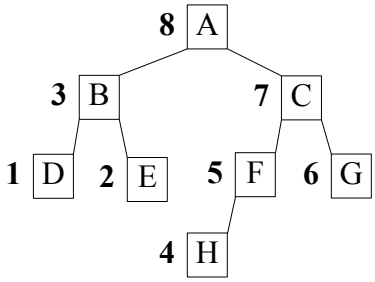


Voici l'algorithme du parcours infixé, les modifications à faire pour un arbre statique sont les mêmes que précédemment pour un parcours préfixé.

Algorithme récursif	Algorithme itératif

Version dynamique
`void infixe(t_noeud*r)`
`{`
`  if(r!=NULL){`
`  infixe(r->g);`
`  printf("%s",r->dat);`
`  infixe(r->d);`
`  }`
`}`
	?
	En théorie il est toujours possible de supprimer la récursion pour établir une version itérative. Mais l'algorithme itératif du parcours infixé est plus difficile à mettre en place et il est rarement utilisé.

Parcours en profondeur postfixé



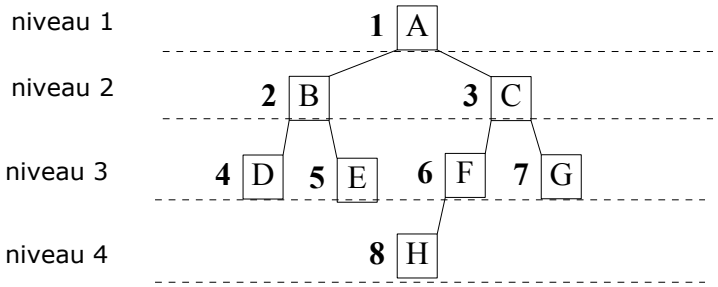
Voici l'algorithme du parcours postfixé, les modifications à faire pour un arbre statique sont toujours les mêmes :

Algorithme récursif	Algorithme itératif

Version dynamique
`void postfixe(t_noeud*r)`
`{`
`  if(r!=NULL){`
`  postfixe(r->g);`
`  postfixe(r->d);`
`  printf("%s",r->dat);`
`  }`
`}`
	?
	Là également une version itérative est envisageable, mais elle n'est quasiment pas utilisée et pas immédiate à mettre en place

b. Parcours en largeur, par niveau

Le parcours en largeur consiste à prendre les noeuds par niveau de la racine aux feuilles et chaque noeud est visité une seule fois :



Algorithme itératif uniquement

L'algorithme est identique à la version itérative du parcours préfixé mais avec l'utilisation d'une file. La file d'attente contient au départ la racine. L'élément en tête de file est extrait et ses fils sont ajoutés à la file jusqu'à ce que la file soit vide :

Version dynamique :

```
void parcoursLargeur(t_noeud*r)
{
   enfiler(r);
   while(!file_vide(r)){
      r=defiler();
      visiter(r);
      if (r->g) enfiler(r->g);
      if (r->d) enfiler(r->d);
   }
}
```

Il n'est pas nécessaire de développer tout un module de gestion de file. Nous connaissons la taille maximum de la file, c'est la taille de l'arbre, il suffit d'allouer un tableau de pointeurs de noeud et de gérer l'indice de tête pour les sorties et l'indice de queue pour les entrées :

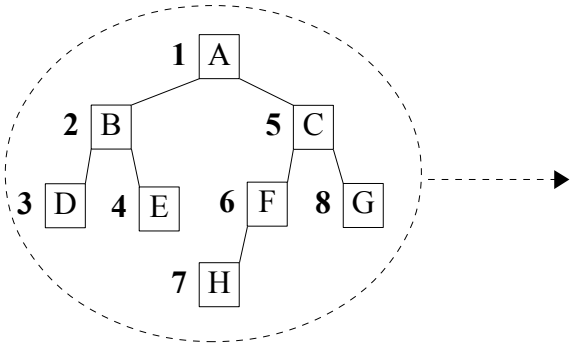
```
void largeur_iter(t_noeud*r)
{
t_noeud**file;
int q=0,t=0;
   if(r==NULL)
      printf("arbre inexistant");
   else{
      file=(t_noeud**)malloc(sizeof(t_noeud*)*taille_dyn(r));
      file[t++]=r;
      while(q!=t){
         r=file[q++];
         printf("%d ",r->val);
         if (r->g)
            file[t++]=r->g;
         if (r->d)
            file[t++]=r->d;
      }
   }
   putchar('\n');
}
```

3. 3. Afficher l'arbre

Il s'agit d'afficher l'arbre très simplement afin de pouvoir contrôler son contenu.

a. Afficher un arbre avec indentation

L'affichage avec une indentation correcte pour chaque différent niveaux est fait ici selon un parcours préfixé de l'arbre. L'arbre test donne l'affichage console suivant :




```
A
    B
        D
        E
C
        F
            H
        G
```

Voici l'algorithme avec les versions pour arbre dynamique et statique :

Arbre dynamique

```
void aff_indent(t_noeud*r,int niveau)
{
int i;
   if (r!=NULL){
      for(i=0; i<niveau; i++) // 5 espaces par niveau
         printf("%5s"," ");
      printf("%s\n",r->dat);
      aff_indent(r->g,niveau+1);
      aff_indent(r->d,niveau+1);
   }
}
```

Arbre statique

```
void aff_indent_stat(t_noeuds*t[],int r,int niveau)
{
   if (r!=-1){
      for(i=0; i<niveau; i++) // 5 espaces par niveau
         printf("%5s"," ");
      printf("%s",t[r].dat);
      aff_indent_stat(t,t[r].g,niveau+1);
      aff_indent_stat(t,t[r].d,niveau+1);
   }
}
```

b. Dessin de l'arbre sans les liens

Dessiner l'arbre en forme d'arbre comme dans l'arbre test le rend plus naturel et lisible mais pour ce faire il est nécessaire de pouvoir positionner les noeuds dans l'espace de la fenêtre console. Il faut donc disposer des coordonnées horizontales et verticales adéquates pour chaque noeud et d'une fonction qui déplace le curseur en écriture à la bonne position sur la page. Nous utilisons la fonction gotoxy()

```
// déplacer le curseur en écriture dans une fenètre console :
void gotoxy(int x, int y)
{
HANDLE h=GetStdHandle(STD_OUTPUT_HANDLE);
COORD i;
    i.X = x;
    i.Y = y;
    SetConsoleCursorPosition (h, c);
}
```

Les fonctions GetStdHandle() et SetConsoleCursorPosition () nécessite l'include de :

```
#include <windows.h>
```

Arbre dynamique

Voici l'algorithme pour le dessin d'un arbre dynamique :

```
void dessin_arbre(t_noeud*r, int*x, int y)
{
   if (r!=NULL){
      dessin_arbre(r->g, x,y+2); // vertical
      *x+=5;   // horizontal
      gotoxy(*x,y);
      printf("%s",r->dat);
      dessin_arbre(r->d, x,y+2);
   }
}
```

Au départ x et y sont initialisés pour le bord gauche de l'arbre en x et la position verticale de départ en y.

4. Obtenir des propriétés de l'arbre binaire

Voici un lot de fonctions qui peuvent être utiles dans différentes circonstances. Elles sont toutes récursives. Nous avons utilisé un entier pour les datas symbolique des nœuds et nous proposons à chaque fois une version pour un arbre dynamique et une version pour un arbre statique :

```
// arbre dynamique
typedef struct noeud{
   int val;
   struct noeud*g, *d;
}t_noeud;
```

```
// arbre statique :
typedef struct noeuds{
   int val;
   int g, d;
}t_noeuds;
```

a. Avoir la taille

Une fonction qui donne le nombre total des noeuds :

Arbre dynamique

```
int taille_dyn(t_noeud*r)
{
int res=0;
   if (r!=NULL)
      res= 1 + taille_dyn(r->g) + taille_dyn(r->d);
   return res;
}
```

Arbre statique

```
int taille_stat(t_noeuds t[],int r)
{
int res=0;
   if (r!=-1)
      res= 1 + taille_stat(t,t[r].g) + taille_stat(t,t[r].d);
   return res;
}
```

b. Donner la hauteur

```
int max(int v1, int v2)
{
   return (v1>v2)?v1:v2;
}
```

Arbre dynamique

```
int hauteur_dyn(t_noeud*r)
{
int h=0;
   if(r!=NULL)
      h= 1 +  max(hauteur_dyn(r->g),hauteur_dyn(r->d));
   return h;
}
```

Arbre statique

```
int hauteur_stat(t_noeuds t[],int r)
{
int h=0;
   if(r!=-1)
      h= 1 +  max(hauteur_stat(t,t[r].g),hauteur_stat(t,t[r].d));
   return h;
}
```

c. Savoir si un noeud est une feuille

Arbre dynamique

```
int feuille_dyn(t_noeud*r)
{
   return (r->g==NULL && r->d==NULL);
}
```

Arbre statique

```
int feuille_stat(t_noeuds t[],int r)
{
   return (t[r].g==-1 && t[r].d==-1);
}
```

d. Compter le nombre des feuilles de l'arbre

Arbre dynamique

```
int compteFeuille_dyn(t_noeud*r)
{
int nb=0;
   if (r!=NULL){
      if (feuille_dyn(r))
         nb=1;
      else
         nb=compteFeuille_dyn(r->g)+compteFeuille_dyn(r->d);
   }
   return nb;
}
```

Arbre statique

```
int compteFeuille_stat(t_noeuds t[],int r)
{
int nb=0;
   if (r!=-1){
      if (feuille_stat(t,r))
         nb=1;
```

```
      else
         nb = compteFeuille_stat(t,t[r].g)
                  + compteFeuille_stat(t,t[r].d);
   }
   return nb;
}
```

e. Lister toutes les feuilles

Arbre dynamique

```
void listerFeuille_dyn(t_noeud*r)
{
   if (r!=NULL){
      if (feuille_dyn(r))
         printf("%d\n",r->val);
      listerFeuille_dyn(r->g);
      listerFeuille_dyn(r->d);
   }
}
```

Arbre statique

```
void listerFeuille_stat(t_noeuds t[],int r)
{
   if (r!=-1){
      if (feuille_stat(t,r))
         printf("%d\n",t[r].val);
      listerFeuille_stat(t,t[r].g);
      listerFeuille_stat(t,t[r].d);
   }
}
```

f. Faire la somme des valeurs des noeuds

Arbre dynamique

```
int sommeVal_dyn(t_noeud*r)
{
int res=0;
   if (r!=NULL)
      res =   sommeVal_dyn(r->g)
            + sommeVal_dyn(r->d)
            + r->val;
   return res;
}
```

Arbre statique

```
int sommeVal_stat(t_noeuds t[],int r)
{
int res=0;
   if (r!=-1)
      res =  sommeVal_stat(t,t[r].g)
           + sommeVal_stat(t,t[r].d)
           + t[r].val;
   return res;
}
```

g. Comparer des valeurs des noeuds de l'arbre

Arbre dynamique

```
int maxVal_dyn(t_noeud*r)
{
int res=0;
   if (r!=NULL){
      res = max(maxVal_dyn(r->g),maxVal_dyn(r->d));
      res = max(res,r->val);
   }
   return res;
}
```

Arbre statique

```
int maxVal_stat(t_noeuds t[],int r)
{
int res=0;
   if (r!=-1){
      res = max(maxVal_stat(t,t[r].g),maxVal_stat(t,t[r].d));
      res = max(res,t[r].val);
   }
   return res;
}
```

h. Ramener un noeud de l'arbre à partir d'une valeur

Arbre dynamique

```
t_noeud* chercheVal_dyn(t_noeud*r, int val)
{
t_noeud*res=NULL;
   if (r!=NULL){
      if (r->val==val)
         res=r;
      else{
         res=chercheVal_dyn(r->g,val);
         if (res==NULL)
            res=chercheVal_dyn(r->d,val);
      }
   }
   return res;
}
```

Arbre statique

```
int chercheVal_stat(t_noeuds t[],int r, int val)
{
int res=-1;
   if (r!=-1){
      if (t[r].val==val)
         res=r;
      else{
         res=chercheVal_stat(t,t[r].g,val);
         if (res==-1)
            res=chercheVal_stat(t,t[r].d,val);
      }
   }
   return res;
```

```
}
```

5. Dupliquer l'arbre

Arbre dynamique

```
t_noeud* copieArbre_dyn(t_noeud*r)
{
t_noeud*res=NULL;
   if (r!=NULL){
      res=(t_noeud*)malloc(sizeof(t_noeud));
      res->val=r->val;
      res->g=copieArbre_dyn(r->g);
      res->d=copieArbre_dyn(r->d);
   }
   return res;
}
```

Arbre statique

```
t_noeuds* copieArbre_stat(t_noeuds t[],int r)
{
t_noeuds* res;
int taille;
   taille=taille_stat(t,r);
   res=(t_noeuds*)malloc(sizeof(t_noeuds)*taille);
   memcpy(res,t,sizeof(t_noeuds)*taille);
   return res;
}
```

6. Détruire l'arbre

Arbre dynamique

```
void detruire_arbre_dyn(t_noeud**r)
{
   if(*r!=NULL){
      detruire_arbre_dyn(&(*r)->g);
      detruire_arbre_dyn(&(*r)->g);
      free(*r);
      *r=NULL;
   }
}
```

Arbre statique

```
void detruire_arbre_stat(t_noeuds**r)
{
   if(*r!=NULL){
      free(*r);
      *r=NULL;
   }
}
```

7. Conversion statique-dynamique d'un arbre binaire

Il s'agit là de pouvoir passer d'un arbre dynamique à un arbre statique et inversement d'un arbre statique à un arbre dynamique. Ces opérations de conversion sont importantes si le programme utilise un arbre dynamique et doit pouvoir le sauvegarder. En effet il n'est pas possible de sauvegarder un arbre dynamique. Il doit nécessairement être converti en statique pour la sauvegarde et en dynamique au moment du chargement.

a. Conversion arbre statique en dynamique

```
t_noeud* statToDyn(t_noeuds*rs,int r)
{
t_noeud*rd=NULL;
   if (r!=-1){
      rd=(t_noeud*)malloc(sizeof(t_noeud));
      rd->val=rs[r].val;
      rd->g=statToDyn(rs,rs[r].g);
      rd->d=statToDyn(rs,rs[r].d);
   }
   return rd;
}
```

b. Conversion arbre dynamique en statique

conversion arbre dynamique en arbre statique :

- avoir la taille de l'arbre dynamique
- créer un tableau dynamique de taille noeuds statiques
- recopier les données de chaque noeuds avec la structure de l'arbre
- regrouper 1,2,3 dans une seule fonction

Ce qui donne :

```
int taille_dyn(t_noeud*rd)
{
int res=0;
   if (rd!=NULL)
      res= 1 + taille_dyn(rd->g) + taille_dyn(rd->d);
   return res;
}

t_noeuds*allouer_tab(t_noeud*rd)
{
t_noeuds*rs=NULL;
   if (rd)
      rs=(t_noeuds*)malloc(sizeof(t_noeuds)*taille_dyn(rd));
   return rs;
}
int copie_dyn_stat(t_noeud*rd,t_noeuds*rs,int *pos)
{
int res=-1;
   if(rd!=NULL){
      res=*pos;
      (*pos)++;
      rs[res].val=rd->val;
      rs[res].g=copie_dyn_stat(rd->g,rs,pos);
```

```
      rs[res].d=copie_dyn_stat(rd->d,rs,pos);
   }
   return res;
}

t_noeuds* dynToStat(t_noeud*rd)
{
t_noeuds*rs=NULL;
int pos=0;
   rs=allouer_tab(rd);
   copie_dyn_stat(rd,rs,&pos);
   return rs;
}
```

8. Sauvegarde et chargement d'un arbre binaire

a. Sauver un arbre dynamique

Si l'arbre est dynamique il est nécessaire de le convertir en statique pour pouvoir le sauvegarder à cause de la perte des adresses mémoires. La fonction de sauvegarde fait appel à la fonction de conversion dynToStat().

```
void save_arbre_dyn(t_noeud*rd)
{
FILE*f;
int nb;
t_noeuds*rs;
   f=fopen("save arbre.bin","wb");
   if (f==NULL||rd==NULL)
      printf("probleme fichier ou arbre inexistant\n");
   else{
      rs=dynToStat(rd);
      nb=taille_dyn(rd);
      fwrite(rs,sizeof(t_noeuds),nb,f);
      fclose(f);
   }
}
```

b. Charger (loaod) un arbre dynamique

Lire un noeud sur le fichier en fonction de sa position:

```
void lireNoeud(FILE*f, int nieme, t_noeuds*enr)
{
   fseek(f,sizeof(t_noeuds)*nieme,SEEK_SET);
   fread(enr,sizeof(t_noeuds),1,f);
}
```

Convertir chaque noeud récupérés sur le fichier en dynamique :

```
t_noeud* fileToDyn(FILE*f, int r)
{
t_noeud*n=NULL;
t_noeuds enr;
   if(r!=-1){
      lireNoeud(f,r,&enr);
```

```
      n=(t_noeud*)malloc(sizeof(t_noeud));
      n->val=enr.val;
      n->g=fileToDyn(f, enr.g);
      n->d=fileToDyn(f, enr.d);
   }
   return n;
}
```

Gestion de l'ouverture du fichier en lecture, conversion de l'arbre en dynamique dans le programme :

```
t_noeud* loadArbre_dyn()
{
FILE*f;
t_noeud*rd=NULL;
   if ((f=fopen("save arbre.bin","rb"))==NULL)
      printf("probleme ouverture du fichier\n");
   else{
      rd=fileToDyn(f, 0);
      fclose(f);
   }
   return rd;
}
```

c. Sauver un arbre statique

```
void save_arbre_stat(t_noeuds*rs,int r)
{
FILE*f;
int nb;
   f=fopen("save arbre.bin","wb");
   if (f==NULL||rs==NULL)
      printf("probleme fichier ou arbre inexistant\n");
   else{
      nb=taille_stat(rs,r);
      fwrite(rs,sizeof(t_noeuds),nb,f);
      fclose(f);
   }
}

int taille_stat(t_noeuds*rs,int r)
{
int res=0;
   if (r!=-1)
      res= 1 + taille_stat(rs,rs[r].g) + taille_stat(rs,rs[r].d);
   return res;
}
```

Eventuellement il peut être intéressant de sauver la taille de l'arbre en début de fichier avec une instruction du type :

```
fwrite (&nb,sizeof(int),1,f);
```

d. Charger (load) un arbre statique

```
t_noeuds* loadArbre_stat()
{
```

```
FILE*f;
t_noeuds*rs=NULL;
int nb;

   if ((f=fopen("save arbre.bin","rb"))==NULL)
      printf("probleme fichier\n");
   else{
      nb=taille_file(f,0);
      rewind(f);
      rs=(t_noeuds*)malloc(sizeof(t_noeuds)*nb);
      fread(rs,sizeof(t_noeuds),nb,f);
      fclose(f);
   }
   return rs;
}
```

Permet d'obtenir la taille de l'arbre sur le fichier :

```
int taille_file(FILE*f,int r)
{
t_noeuds enr;
int res=0;
   if (r!=-1){
      lireNoeud(f,r,&enr);
      res= 1 + taille_file(f,enr.g) + taille_file(f,enr.d);
   }
   return res;
}
```

9. Arbres binaires sur fichiers

Le principe ici est d'avoir un arbre statique mais sur fichier et non en RAM. Les noeuds sont enregistrés comme dans un tableau, les uns à la suite des autres et désignés par leur numéro d'ordre dans le fichier.

a. Structure de données

```
// noeud arbre statique
typedef struct noeuds{
   int val;
   int g, d;   // position en noeud dans le fichier des fils
}t_noeuds;
```

b. Lecture d'un noeud à partir de son numéro d'enregistrement

```
void lireNoeud(FILE*f, int nieme, t_noeuds*enr)
{
   fseek(f,sizeof(t_noeuds)*nieme,SEEK_SET);
   fread(enr,sizeof(t_noeuds),1,f);
}
```

c. Adaptation des fonctions pour les arbres binaires dynamiques ou statiques

Toutes les fonctions définies sur les arbres binaires peuvent être adaptées pour l'utilisation d'un arbre sur fichier. Voici par exemple la fonction d'affichage indenté d'un arbre sur fichier à partir d'un parcours préfixé :

```
void aff_indent_file(FILE*f, int r,int niveau)
{
t_noeuds enr;
int i;
   if (r!=-1){
      lireNoeud(f,r,&enr);
      for (i=0; i<niveau; i++)
         printf("%5s"," ");
      printf("%d",enr.val);
      aff_indent_file(f,enr.g,niveau+1);
      aff_indent_file(f,enr.d,niveau+1);
   }
}
```

A chaque opération sur un noeud il faut d'abord récupérer le noeud dans un noeud tempon en RAM qui permet de lire ses informations. C'est le rôle de la variable t_noeuds enr. C'est ce qu'il faut ajouter à toutes les fonctions vues précédemment pour les adapter à la gestion d'un arbre sur fichier.

10. Mise en pratique : arbre binaire

Exercice 1
Faire générateur d'arbres binaires DYNAMIQUES contenant des données aléatoires. Générer un arbre. Parcourir l'arbre en profondeur (trois parcours à tester). Parcourir l'arbre en largeur. A chaque fois afficher l'arbre. Obtenir toutes les propriétés de l'arbre (taille, hauteur, nombre de feuilles, lister les feuilles, somme des nœuds de l'arbre). Ramener tel ou tel nœud selon une valeur donnée. Sauver l'arbre. Détruire l'arbre. Charger l'arbre.

Exercice 2
Faire générateur d'arbres binaires STATIQUES contenant des données aléatoires. Générer un arbre. Parcourir l'arbre en profondeur (trois parcours à tester). Parcourir l'arbre en largeur. A chaque fois afficher l'arbre. Obtenir toutes les propriétés de l'arbre (taille, hauteur, nombre de feuilles, lister les feuilles, somme des nœuds de l'arbre). Ramener tel ou tel nœud selon une valeur donnée. Sauver l'arbre. Détruire l'arbre. Charger l'arbre.

Exercice 3
Faire générateur d'arbres binaires SUR FICHIER contenant des données aléatoires. Générer un arbre. Parcourir l'arbre en profondeur (trois parcours à tester). Parcourir l'arbre en largeur. A chaque fois afficher l'arbre. Obtenir toutes les propriétés de l'arbre (taille, hauteur, nombre de feuilles, lister les feuilles, somme des nœuds de l'arbre). Ramener tel ou tel nœud selon une valeur donnée. Sauver l'arbre. Détruire l'arbre. Charger l'arbre.

Exercice 4
Représenter l'arbre généalogique d'une famille de son choix ou de son invention. Les informations peuvent être fournies par un fichier texte du style :
Sonia : Paul, Catherine, Benoit;
Paul : Isabelle, Armand ;

Catherine : Anatole, Eloïse, Brigitte ;
etc.

Exercice 5
Une expression arithmétique comme (14 * 5) – (7 / 10) peut prendre la forme d'un arbre. Dans un programme :

- □ Construire un arbre à partir d'une expression arithmétique.
- □ Écrire une fonction d'évaluation de l'expression arithmétique à partir d'un arbre et donner son résultat.
- □ Ecrire une fonction qui affiche la traduction en expression postfixée.

```
Une expression postfixée (notation polonaise inversée) est du type :
5  17  6 – 4 * 2 + *
les parenthèses implicites sont :
( 5 ( ( ( 17  6  – )  4 * )  2 + ) * )
ce qui équivaut, en notation classique infixée, à :
( ( ( 17 – 6 ) * 4 ) + 2 ) * 5
```

- □ Ecrire une fonction pour construire un arbre à partir d'une expression postfixée donnée
- □ Ecrire une fonction qui construit un arbre à partir d'une expression arithmétique infixée classique (avec parenthèses).

Exercice 6
Soit la description suivante :

Homme : tete, cou, tronc, bras, jambe ;

tete : crane yeux, oreille, cheveux, bouche ;

tronc : adbomen, thorax ;

thorax : cœur, foie, poumon ;

jambe : cuisse, mollet, pied ;

pied : cou de pied, orteil, talon

bras : epaule, avant bras, main

main : doigt

Dans un programme faire l'arbre n-aire, puis l'arbre binaire correspondant.
Sur le modèle de cette description, faire l'arbre des composants d'une voiture. Toujours sur ce modèle analysez un sujet ou un objet de votre choix et représentez l'arbre correspondant dans un programme.

Exercice 7
Simuler la gestion d'un tournoi de tennis sous la forme d'un arbre binaire. Les informations de chaque match sont conservées. A l'issue du tournois le match de final se trouve à la racine de l'arbre.

Exercice 8

Soit la description suivante :

Terre : Europe, Asie, Afrique, Amérique, Océanie ;

Europe : France, Espagne, Italie, Allemagne, Belgique ;

Asie : Chine, Inde, Japon ;

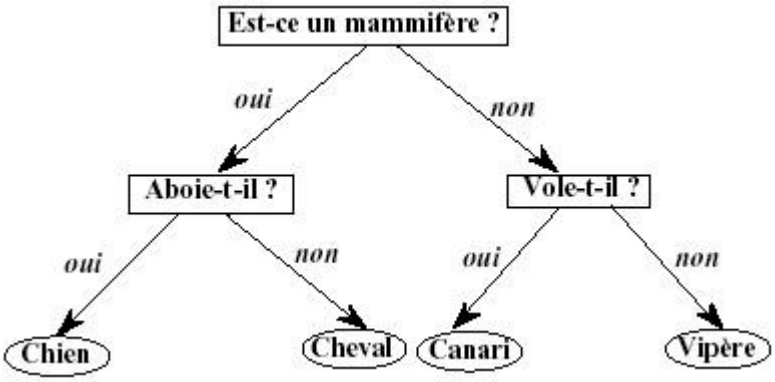
Afrique : Burkina, Côte ivoire ;

Dans un programme faire l'arbre n-aire, puis l'arbre binaire correspondant.
Sur ce modèle faire, par exemple la carte de vos lieux favoris.

Exercice 9

Objectifs : parcourir et ajouter dans un arbre binaire.

La représentation d'un domaine dont les concepts sont hiérarchisés sous forme d'arbre binaire peut s'appliquer aussi bien à des tâches de classification / identification (d'animaux, de végétaux, de minéraux...) qu'à des tâches de diagnostic (diagnostic médical, détection de panne...). Par exemple :



06RI02

Dans cet exemple l'ordinateur pose des questions pour essayer de découvrir un animal auquel vous pensez. Vous ne pouvez répondre que par OUI ou NON. S'il échoue, il vous demande de lui fournir une question qui caractérise l'animal qu'il n'a pas trouvé, ce qui permet de réaliser une certaine forme d'apprentissage. Trouver un animal consiste à parcourir un arbre binaire dont les nœuds internes sont des questions et les feuilles des animaux. La phase d'apprentissage consiste, si l'utilisateur le souhaite, à ajouter l'animal non trouvé dans l'arbre. (voir trace d'exécution en 2ème page)

Afin de garder une trace de cet arbre binaire, celui-ci est sauvegardé dans un fichier.

Deux approches sont possibles : sur fichier en accès direct ou en dynamique en mémoire centrale. Pour commencer faites plutôt un arbre dynamique (mais vous pouvez opter pour un arbre sur fichier si vous préférez). La sauvegarde sur fichier d'un arbre dynamique pourra être réalisée grâce à un parcours en largeur de l'arbre.

A faire :

Définir en C la structure de données pour un nœud de l'arbre.

- A quelle(s) condition(s) sait-on différencier un nœud interne d'une feuille ?

- Quel est l'ordre de parcours (infixé, préfixé ou postfixé) de l'arbre affiché dans la trace d'exécution de la page suivante ? Justifiez. Donnez les 2 autres ordres de parcours.
- Ecrire, commenter et tester les fonctions suivantes en respectant la trace d'exécution de la page suivante :
  - Affichage d'un arbre de jeu.
  - Parcours d'un arbre de jeu en posant les questions à l'utilisateur
  - Apprentissage d'un nouvel animal par ajout d'une question et d'une réponse
  - Toute autre fonction jugée nécessaire
- Envisager la sauvegarde et le chargement de l'arbre dans le programme.

Pour la trace d'exécution, on a utilisé les caractères normaux pour ce qui est affiché à l'écran par le programme, les commentaires sont en *italiques* et les réponses tapées par l'utilisateur en caractères **gras souligné**.

```
Ouverture du fichier animaux.bin          ouverture du fichier contenant un arbre de jeu

            Vipere                        affichage de l'arbre de jeu
      Vole t-il ?
            Canari
Est-ce un mammifere ?
            Cheval
      Aboie t-il ?
            Chien

Est-ce un mammifere ? O                   parcours de l'arbre
Aboie t-il ? N                            en posant les questions à l'utilisateur
Cheval
Etes vous d'accord ? N                    apprentissage
A quel animal pensiez-vous ? Chat
Entrez une question permettant de caracteriser cet animal
et pour laquelle la reponse est OUI : Miaule t-il ?
Maintenant je connais cet animal

Encore ? O                                boucle du programme principal

            Vipere                        affichage de l'arbre de jeu
      Vole t-il ?
            Canari
Est-ce un mammifere ?
                  Cheval
            Miaule t-il ?
                  Chat
      Aboie t-il ?
            Chien

Est-ce un mammifere ? O                   parcours de l'arbre
Aboie t-il ? N                            en posant les questions à l'utilisateur
Miaule t-il ? O
Chat
Etes vous d'accord ? O

Encore ? N                                sortie de la boucle du programme principal

Fermeture du fichier animaux.bin          fermeture du fichier à accès direct
```

06RI03

H. Arbres binaires de recherche

1. Définition

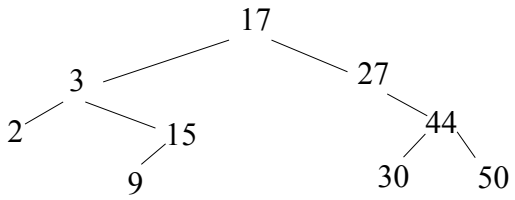
Dans un arbre binaire de recherche tous les éléments sont dotés d'une clé et ils sont positionnés dans l'arbre en fonction de cette clé. La clé rend lisible et utilisable l'ordre de l'arbre. La clé est en général un nombre qui est ajouté à l'élément. C'est grâce à ce nombre que l'élément peut être rangé et identifié à une place précise dans l'arbre. Tous les éléments peuvent ensuite être retrouvés rapidement grâce à leur clé, sans être obligé de parcourir tout l'arbre systématiquement. Les éléments rangés dans l'arbre sont en quelque sorte triés en fonction de leur clé. Il est facile d'ajouter des nouveaux éléments ou de supprimer des éléments sans modifier l'ordre de l'arbre.

Les clés obéissent aux règles suivantes :

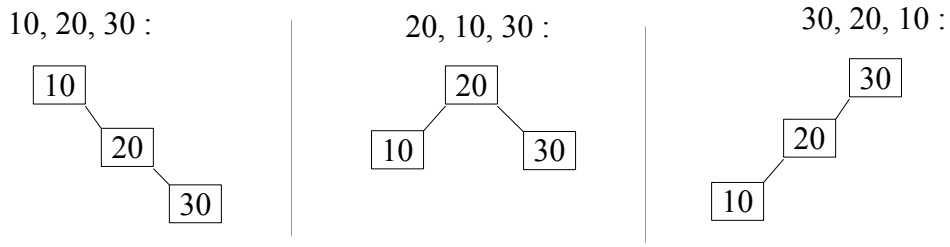
Pour chaque noeud de l'arbre :

- □ les clés du sous arbre gauche (SAG) sont inférieures à celle de la racine
- □ les clés du sous arbre (SAD) droit sont supérieures à celle de la racine
- □ toutes les clés sont différentes, il n'y a pas de clé identique

Par exemple la suite de clés 17, 3, 15, 9, 27, 44, 2, 50, 30 donne l'arbre suivant :



L'arbre ci-dessus suppose que les éléments sont entrés dans l'ordre de la suite, mais la forme de l'arbre dépend de l'ordre d'insertion :



Remarque

Quelque soit l'ordre d'entrée dans l'arbre des éléments avec leur clé, à l'issue, un parcours infixé de l'arbre donnera toujours une suite croissante des clés.

Ci-dessus ce sont les suites:

```
2, 3, 9, 15, 17, 27, 30, 44, 50
et
10, 20, 30
```

2. Structure de données

Notre objectif est de mettre en place un arbre binaire de recherche avec ses fonctions usuelles de traitement. Il s'agit d'un arbre dynamique. La clé peut être un entier mais souvent elle est intégrée au début d'une chaine de caractères et nous allons prendre cette option pour écrire les fonctions autour de l'arbre. Nous utilisons la structure suivante :

```
typedef struct noeud{
   char*dat;
   struct noeud*g,*d;
}t_noeud;
```

Pour afficher des données de façon un peu plus graphique dans l'espace de la fenêtre console nous utilisons des fonctions fournies en annexe.

Tous les arbres sont générés automatiquement avec des données aléatoires dans cet étude et nous avons besoin de limiter la taille en noeud des arbres, c'est l'objet de la macro :

```
#define MAXNOEUD  15
```

Pour l'affichage de l'arbre en forme d'arbre (mais sans les liens) la distance verticale des noeuds est fixée avec la macro :

```
#define PASY      2
```

3. 3. Insérer un élément dans l'arbre selon sa clé

L'insertion d'un élément se fait toujours en feuille de l'arbre, jamais au niveau des noeuds internes. Pour connaître l'endroit adéquat où insérer il faut descendre dans l'arbre en s'orientant à chaque noeud en fonction de la clé. Si la clé du nouveau noeud est plus petite que celle du noeud courant de l'arbre partir à gauche sinon partir à droite jusqu'à arriver au niveau des feuilles, à la bonne place. La première fonction à écrire est cette fonction qui permet de comparer deux clés. Elle servira à chaque fois qu'il faut rechercher un élément ou une position dans l'arbre selon une clé.

a. Comparer deux clés

La fonction reçoit en argument les deux chaines qui contiennent une clé et ce sont uniquement les clés qui sont comparées. Elle retourne 0 si les clés sont égales, -1 si la première est inférieure à la seconde et 1 si la première est supérieure à la seconde. Pour séparer la clé du reste de la chaine et la convertir en entier il y a plusieurs possibilités. Ce sont les fonctions strtol(), atol() et atoi(). Nous avons choisi atoi() qui retourne le nombre entier en ascii du début de chaine sous forme d'entier numérique int. Les deux nombres sont ensuite comparés à l'aide de l'opérateur conditionnel :

```
int compareCle(char*dat1, char*dat2)// d1 comparé à d2
{
int d1,d2;
   d1=atoi(dat1);
   d2=atoi(dat2);
```

```
   return(d1==d2)?0:((d1<d2)?-1:1);
}
```

b. Insérer à la bonne place

Pour insérer dans l'arbre un élément selon sa clé il suffit de partir de la racine et de s'orienter à chaque noeud en fonction de la valeur de sa clé comparée à celle du nouveau noeud à insérer. Si la nouvelle clé est inférieure partir à gauche sinon partir à droite et il ne peut pas y avoir de clé identique par définition dans l'arbre de recherche. La descente continue jusqu'à arriver à une feuille, tout en bas de l'arbre. Une fois arrivé à la bonne place sur un pointeur à NULL de la feuille, créer un noeud, lui affecter les datas et affecter l'adresse de ce nouveau noeud au pointeur de la feuille.

Au départ la fonction reçoit en paramètre la racine de l'arbre. Comme elle est susceptible d'être modifiée il s'agit d'un passage par référence. C'est l'adresse du pointeur lui-même qui est passé en paramètre à savoir un pointeur de pointeur. La fonction est récursive et ce paramètre prend successivement les adresses des pointeurs fils gauche ou fils droit en fonction de la descente. Lorsque le pointeur dont la fonction a l'adresse contient NULL c'est l'arrivée et il est possible d'affecter une adresse valide à ce pointeur. Par ailleurs les datas sont transmises sous la forme d'une chaine de caractères char*dat.

```
void inserer(t_noeud**r,char*dat)
{
t_noeud*n;
int cmp;
   if(*r==NULL){
      n=(t_noeud*)malloc(sizeof(t_noeud));
      n->dat=(char*)malloc(strlen(dat)+1); // +1 pour \0
      strcpy(n->dat,dat);
      n->g=n->d=NULL;
      *r=n;
   }
   // attention, ne pas avoir 2 clés identiques
   else if( (cmp=compareCle(dat,(*r)->dat)) ==0)
      printf("erreur : la cle %s existe deja\n",dat);
   else if (cmp<0)
      // passage adresse du pointeur fils gauche
      inserer(&(*r)->g,dat);
   else
      // passage adresse du pointeur fils droit
      inserer(&(*r)->d,dat);
}
```

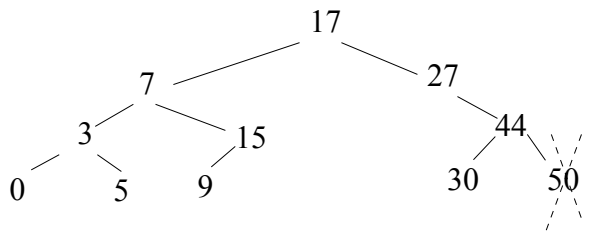
4. Rechercher dans l'arbre un élément selon sa clé

Pour rechercher un élément il suffit de partir de la racine avec la clé de l'élément à rechercher. La fonction prend en paramètre la racine et les datas qui contiennent la clé. La racine ne sera pas modifiée, c'est juste un parcours. Il n'y a donc pas lieu de la passer par référence. La fonction retourne l'adresse du noeud qui correspond à la clé et NULL si cette clé ne correspond à aucun élément de l'arbre. La fonction est récursive. Elle s'appelle elle-même vers la gauche ou vers la droite tant que le noeud n'est pas trouvé :

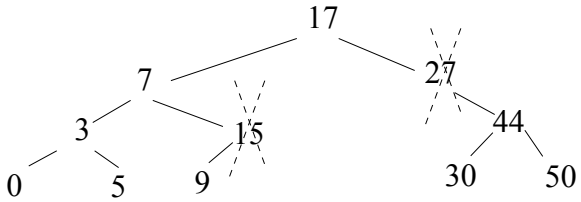
```
t_noeud* rechercher(t_noeud*r, char*dat)
{
int cmp;
t_noeud*res=NULL;
   if(r!=NULL){
      if ((cmp=compareCle(dat,r->dat))==0)
         res=r;
      else if (cmp<0)
         res=rechercher(r->g, dat);
      else
         res=rechercher(r->d,dat);
   }
   return res;
}
```

5. Supprimer un élément dans l'arbre de recherche

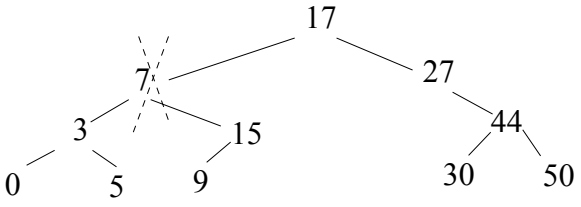
Supprimer une feuille d'un arbre est assez simple :



Pour supprimer le noeud 50 il suffit de mettre le fils droit du noeud 44 à NULL. C'est simple également pour chaque noeud qui n'a qu'un seul fils. Par exemple :



Pour supprimer le noeud 15 il suffit de faire pointer le fils droit du noeud 7 sur le noeud 9. Pour supprimer le noeud 27 il faut faire pointer les fils droit du noeud 17 sur le noeud 44. Nous sommes là dans la configuration d'une liste chainée. Mais ça se complique si le noeud à supprimer a deux fils. Par exemple, comment supprimer le noeud 7 ?



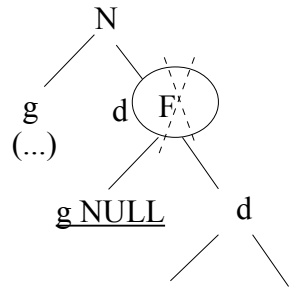
Que faire des noeud 3 et 15 ? Comment les accrocher ? La solution consiste à :

- chercher le plus grand dans le sous arbre de gauche ou le plus petit dans le sous arbre de droite (5 ou 9 pour le noeud 7). Ils seront nécessairement ou une feuille ou un noeud n'ayant qu'un seul fils. Ensuite :
- le supprimer en conservant sa valeur et
- copier cette valeur dans le noeud à supprimer

a. Trois cas

Dans cette situation nous avons trois cas. Nous voulons supprimer le noeud F à droite de N sur le schéma :

1) Le nœud F n'a pas de fils gauche :

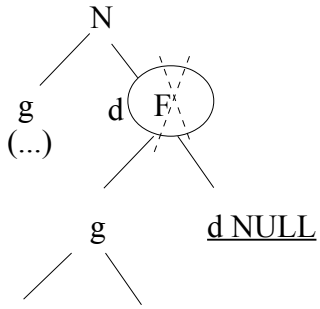


le noeud F n'a pas de fils gauche :

Si (F->g == NULL) Alors
N->d = F->d

(ou N->g = F->d si nous sommes à gauche de N)

2) Le nœud F n'a pas de fils droit

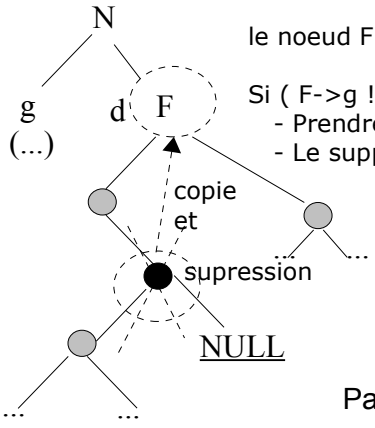


le noeud F n'a pas de fils droit :

Si (F->d == NULL) Alors
N->d = F->g

(ou N->g = F->g si nous sommes à gauche de N)

3) Le nœud F a fils gauche et fils droit



le noeud F a fils gauche et fils droit :

Si (F->g != NULL && F->d != NULL) Alors
- Prendre plus grand à gauche (ou plus petit à droite)
- Le supprimer de l'arbre et copier sa valeur en F

b. Fonction de recherche du noeud max

Pour le cas n°3 nous avons besoin d'une fonction qui ramène le noeud le plus grand à gauche ou le plus petit à droite. Nous avons opté pour le plus grand à gauche. La fonction ramène ce noeud et le supprime de l'arbre. Pour l'obtenir, nous partons au départ à gauche et il faut ensuite chercher toujours sur la droite jusqu'à tomber sur NULL. Lorsque la fonction a trouvé ce noeud elle le retourne via le mécanisme de retour. Elle retourne NULL en cas d'échec.

Comme la racine est susceptible d'être modifiée elle est passée par référence, c'est donc un pointeur de pointeur t_noeud**. La fonction est récursive. A chaque sous appel de la fonction c'est l'adresse du pointeur fil droit du noeud courant qui est passée. La fin des appels a lieu lorsque le pointeur courant dont la fonction a l'adresse pointe sur NULL soit :

```
si ( (*r)->d==NULL ) alors
   *r donne l'adresse du noeud cherché.
```

Cette adresse est retournée et le pointeur *r prend la valeur du fils gauche de l'arbre, c'est à dire que ce noeud fils droit est supprimé de l'arbre. Cette adresse toujours valide remonte toute la pile des appels et est restituée au contexte du premier appel de la fonction. Voilà la fonction :

```
t_noeud* rameneNoeudMax(t_noeud**r)
{
t_noeud*pg=NULL;
   if(*r!=NULL){
      if((*r)->d==NULL){ // plus grand trouvé
         pg=*r;
         *r=(*r)->g;
      }
      else
         pg=rameneNoeudMax(&(*r)->d);
   }
   return pg;
}
```

c. Fonction de suppression

La fonction de suppression est récursive. Elle prend la racine en paramètre et les datas avec la clé de l'élément à supprimer. Comme la racine est susceptible d'être modifiée elle est passée par référence. C'est un pointeur de pointeur t_noeud**.

Le premier point est de trouver l'élément à supprimer dans l'arbre et pour ce faire de comparer la clé de l'élément à supprimer avec celle du noeud courant. Si le résultat est inférieur à 0 la fonction de suppression s'appelle elle-même en partant à gauche. Si au contraire le résultat est supérieur à 0 la fonction de suppression s'appelle elle-même en partant à droite. Si le résultat est égal à 0 le noeud à supprimer est trouvé en *r et nous devons traiter les trois cas mentionnés pour la suppression d'un noeud :

- Si pas de fils droit, prendre le fils gauche comme suite de l'arbre.
- Si pas de fils gauche prendre le fils droit comme suite de l'arbre.
- Si un fils droit et un fils gauche alors supprimer le plus grand à gauche et ramener sa valeur pour la copier sa valeur en (*r)->dat, c'est à dire dans l'élément courant à supprimer. Si la chaine de caractères dat est dynamique attention de bien veiller à ce qu'elle soit suffisamment grande.

```
t_noeud* suppNoeud(t_noeud**r,char*dat)
{
int cmp;
t_noeud*res=NULL;

   if(*r!=NULL){
      if ( (cmp=compareCle(dat,(*r)->dat))<0)
         res=suppNoeud(&(*r)->g,dat);
      else if (cmp>0)
         res=suppNoeud(&(*r)->d,dat);
      else{     // noeud trouvé, suppression
         res=*r;
         if (res->d==NULL)
            *r=res->g;
         else if (res->g==NULL)
            *r=res->d;
         else{  // si un fils gauche et un fils droit alors
                // suppression plus grand à gauche
            res=rameneNoeudMax(&(*r)->g);
           (*r)->dat=(char*)realloc((*r)->dat,strlen(res->dat)+1);
            strcpy((*r)->dat,res->dat);
         }
      }
   }
   return res;
}
```

6. Lister tous les éléments de l'arbre (parcours en largeur)

Le principe ici est de remonter un tableau qui contient tous les noeuds de l'arbre rangés par niveau dans le sens d'un parcours en largeur. Le premier point est d'avoir la taille de l'arbre. Ensuite nous avons besoin d'allouer un tableau t_noeud ou de pointeurs t_noeud*, nous avons opté pour un tableau de pointeurs. Le premier noeud à l'indice 0 est le noeud racine de l'arbre. Ensuite sont rentrés s'ils existent les fils gauche et droite de chaque noeud qui entrent dans le tableau. A l'issue l'adresse du tableau dynamique est retournée, ce qui donne :

```
t_noeud** listerArbre(t_noeud*r, int*nbNoeud)
{
int t,q;
t_noeud**tab=NULL;
   if (*nbNoeud>0){
      *nbNoeud=taille_dyn(r);
      tab=(t_noeud**)malloc(sizeof(t_noeud*)*(*nbNoeud));
      t=0;
      q=1;
      tab[t]=r;                  // racine en tête
      while (q<*nbNoeud){
         if (tab[t]->g !=NULL )  // si fils gauche ajouter
```

```
            tab[q++]=tab[t]->g;
         if (tab[t]->d != NULL)  // si fils droit ajouter
            tab[q++]=tab[t]->d;
         t++;                    // passer au suivant
      }
   }
   return tab;
}
```

La fonction pour avoir la taille, déjà mentionnée est la suivante :

```
int taille_dyn(t_noeud*r)
{
int res=0;
   if (r!=NULL)
      res=1+taille_dyn(r->g)+taille_dyn(r->d);
   return res;
}
```

Une fois que la liste est constituée, il est simple de l'afficher :

```
void affiche_liste(t_noeud*t[],int nb)
{
int i;
   for (i=0; i<nb; i++){
      printf("%s--",t[i]->dat);
   }
   putchar('\n');
}
```

Ou de la détruire.

```
void detruireListe(t_noeud***l)
{
   if(*l){
      free(*l);
      *l=NULL;
   }
}
```

Remarque

La suppression de la liste ne supprime pas l'arbre. Les adresses mémoires de tous les éléments de l'arbre ne sont pas touchées.

7. Afficher l'arbre

L'affichage de l'arbre avec la bonne hiérarchie se fait à partir d'un parcours infixé à l'aide de deux variables, une pour la position horizontale, l'autre pour la position verticale. La position horizontale dépend de la taille de la chaine de caractère de chaque noeud à afficher. Aussi la progression horizontale n'est elle pas régulière. De plus elle est constante, il n'y a pas de noeud à la même position horizontale. C'est pourquoi un passage par référence est nécessaire qui permet de stocker au fur et à mesure l'ancienne position. La progression verticale est relative à chaque contexte d'appel de la fonction selon l'empilement des appels récursifs. Elle est définie par la macro PASY.

```
void graphInfixe(t_noeud*r, int*x, int y)
```

```
{
   if (r!=NULL){
      afficheInfixe(r->g,x,y+PASY);
      gotoxy(*x,y);
      printf("%s",r->dat);
      *x+=strlen(r->dat);//PASX;
      afficheInfixe(r->d,x,y+PASY);
   }
}
```

L'affichage complet de l'arbre suppose de savoir où est le curseur en écriture au départ pour la position verticale. La position horizontale est donnée, contre bord en 0 à gauche. L'arbre est affiché. Le curseur est déplacé en dessous de l'arbre contre le bord gauche et la hauteur de l'arbre est affichée.

```
void afficheArbre(t_noeud*r)
{
int y=wherey()+1;
int x=0;
   graphInfixe(r,&x,y);
   gotoxy(0,y+(hauteur(r)*PASY));
   printf("hauteur de l'arbre : %d\n",hauteur(r));
}
```

Rappel de la fonction qui permet d'obtenir la hauteur de l'arbre

```
int hauteur(t_noeud*r)
{
int res=0;
   if (r!=NULL)
      res=1+max(hauteur(r->g),hauteur(r->d));
   return res;
}
```

8. Test dans le main

Le test propose les actions suivantes :

```
int menu()
{
int res=-1;
   printf(  "1  : creer arbre de recherche, afficher \n"
            "2  : inserer afficher\n"
            "3  : rechercher\n"
            "4  : supprimer\n"
            "5  : lister\n"
         );
   scanf("%d",&res);
   rewind(stdin);
   return res;
}
```

Voici le main complet avec les appels des fonctions pour les actions présentées :

```
int main()
{
int fin=0,nb;
t_noeud*r=NULL;
t_noeud*n=NULL;
t_noeud**t=NULL;
```

```
char cle[16];

   srand(time(NULL));
   while(!fin){
      switch(menu()){
         case 1 :
            detruireArbre(&r);
            r=creerArbre_Alea(1+rand()%MAXNOEUD);
            afficheArbre(r);
            break;

         case 2 :
            printf("entrer la cle du noeud a trouver :\n");
            fgets(cle,16,stdin);
            n=rechercher(r, cle);
            if (n!=NULL)
               printf("trouve : %s\n",n->dat);
            else
               printf("non trouve dans l'arbre\n");
            break;

         case 3 :
            printf("entrer la cle du noeud a supprimer :\n");
            fgets(cle,16,stdin);
            n=suppNoeud(&r,cle);
            if (n!=NULL){
               free(n->dat);
               free(n);
            }
            afficheArbre(r);
            break;

         case 4 :
            t=listerArbre(r, &nb);
            affiche_liste(t,nb);
            detruireListe(&t);
            break;

         default : fin=1;
      }
   }
   detruireArbre(&r);
   return 0;
}
```

9. Mise en pratique : arbres

Exercice 1
Générer un arbre binaire de recherche. Rechercher dedans un élément selon sa clé. Supprimer un élément. Afficher l'arbre. Faire la somme des éléments de l'arbre. Lister tous les éléments de l'arbre etc.

Exercice 2
Faire un compteur d'occurrence de mots à partir d'un texte fourni par exemple sur fichier texte.