ALGO

1) Les piles

```
int empiler (struct T **Adresse_Pile,  struct T *t)
{
    new = allouer de la place pour nouvel élément ;
    SI place disponible
          ALORS copier t dans nouvel espace ;
                new->suivant ⇦ *Adresse_Pile ;
                *Adresse_Pile ⇦ new ;
                empiler ⇦ 1 ;
          SINON empiler ⇦ 0 ;
    FIN SI
}

OU

struct T * empiler (struct T *Pile,  struct T *t)
{
    new = allouer de la place pour nouvel élément ;
    SI place disponible
          ALORS copier t dans nouvel espace ;
                new->suivant ⇦ Pile ;
                empiler ⇦ new ;
          SINON empiler ⇦ NULL ;
    FIN SI
}
```

```
struct T *depiler (struct T **Adresse_Pile)
{
    SI *Adresse_Pile = NUL
          ALORS   depiler ⇦ NUL
          SINON   depiler ⇦ *Adresse_Pile;
                  *Adresse_Pile ⇦
                                (*Adresse_Pile)->suivant;
    FIN SI
}
```

```
int pile_vide (struct T *Pile)
{
    pile_vide ⇦ (Pile == NUL)
}
```

```
struct T *dernier (struct T *Pile)
{
    dernier ⇦ Pile
}
```

2) Les files

```
int enfiler (struct T **Adresse_File,  struct T *t)
{
    struct T *tmp;

    new = allouer de la place pour nouvel élément ;
    SI place disponible
          ALORS copier t dans new ;
                new->Suivant ⇦NUL
                SI *Adresse_File = NUL
                      ALORS *Adresse_File ⇦New ;
                      SINON
                            Tmp ⇦*Adresse_File
                            TANT QUE Tmp->Suivant <> NUL FAIRE
                                  Tmp ⇦Tmp->Suivant ;
                            FIN TANT QUE
                            Tmp->Suivant ⇦New ;
                FIN SI
                enfiler ⇦1 ;
          SINON enfiler ⇦0 ;
    FIN SI
}

OU

struct T * enfiler (struct T *File,  struct T *t)
{
    struct T* tmp;

    new = allouer de la place pour nouvel élément ;
    SI place disponible
          ALORS copier t dans new ;
                SI File = NUL
                      ALORS enfiler ⇦New ;
                      SINON
                            Tmp ⇦File
                            TANT QUE Tmp->Suivant <> NUL FAIRE
                                  Tmp ⇦Tmp->Suivant ;
                            FIN TANT QUE
                            Tmp->Suivant ⇦New ;
                            enfiler ⇦File;
                FIN SI
          SINON enfiler ⇦NULL ;
    FIN SI
}
```

```
struct T *defiler (struct T **Adresse_File)
{
    SI *Adresse_File est NUL
          ALORS   defiler ⇦NUL
          SINON   defiler ⇦*Adresse_File;
                  *Adresse_File ⇦(*Adresse_File)->Suivant;
    FIN SI
}
```

3) La récursivité

Une procédure est dite **récursive** quand, de manière directe ou indirecte, elle s'appelle elle-même.

La récursivité est **directe** si l'appel de la procédure est défini de manière explicite dans le corps de la procédure.

La récursivité est dite **implicite** ou **indirecte** si l'appel de la fonction P appartient au corps d'une fonction appelée par la fonction P. Dans ce cas, on dit encore que les procédures sont mutuellement récursives.

Un problème se prête particulièrement bien à l'analyse récursive lorsqu'il peut être décomposé en plusieurs sous-problèmes de même type, mais de taille plus petite ! Dans un tel cas, la méthode générale comportera trois étapes :

- **Paramétrage du problème**, faisant apparaître les différents éléments dont dépend la solution, et en particulier la taille du problème à résoudre, qui (dans les cas favorables) devra décroître à chaque appel récursif ;
- **Recherche d'un cas trivial** et de sa solution. C'est souvent l'étape clé de l'algorithme. Un "cas trivial" est un cas qui peut être réglé directement, sans appel récursif. Ce sera souvent le cas où la *taille* du problème est nulle ou égale à 1.
- **Décomposition du cas général** visant à le ramener à un ou plusieurs problèmes, en principe plus simples (de taille plus petite).

5.4.2 ARBIN_VIDE

ARBIN_VIDE est une fonction qui, étant donné un arbre binaire existant, c'est-à-dire vide ou ayant une racine répond à la question, est-il vide ? Elle prend donc en argument **parbin** et renvoie une valeur booléenne.

```
int ARBIN_VIDE (ELEARBIN *parbin)
{
    if (parbin == 0)
          return 1;
    else
          return 0;
}
```

5.4.3 RACINE

Cette fonction **RACINE** est une fonction qui affiche le contenu de la racine d'un arbre, elle prend donc en argument uniquement **parbin**, elle n'a aucun donnée à renvoyer.

```
void RACINE (ELEARBIN *parbin)
{
    if (! ARBIN_VIDE (parbin))
    {
          printf ("....", parbin->...);
          printf ("....", parbin->...);
          ....
    }
    else
    printf ("l'arbre est vide");
}
```

5.4.4 CONSARBIN

CONSARBIN transforme un arbre vide en le remplaçant par un élément de type ELEARBIN. Par cette opération la variable de type pointeur détenant l'adresse de l'arbre va être modifiée, il est donc intéressant que cette fonction renvoie l'adresse de la racine celle-ci étant l'adresse fournie par la nouvelle allocation de mémoire créée en C par la fonction **malloc**.

Une autre solution aurait été de donner à la fonction l'adresse du pointeur de la racine de l'arbre (double indirection), cela permettant en plus de retourner une valeur disant si la fonction s'est exécutée normalement ou non.

```
ELEARBIN * CONSARBIN_1 (ELEARBIN *parbin)
{
    parbin = (ELEARBIN *) malloc(sizeof(ELEARBIN));
    parbin->... = ... ;
    parbin->... = ... ;
    ...
    parbin->pdroite = NULL ;
    parbin->pgauche = NULL ;

    return parbin;
}
```

Ou alors on peut avoir une solution plus complète :

- On va écrire une fonction destinée à introduire des informations dans une structure de type ELEARBIN que l'on appellera CRÉER_ENR
- Et une fonction CONSARBIN destinée à construire un arbre dont les données de la racine sont dans une structure passée en argument à la fonction et pouvant prendre en argument les adresses des 2 sous-arbres. Ce qui pourra être utile si les deux sous-arbres existent déjà.

```
ELEARBIN * CREER_ENR (ELEARBIN *pel)
{
     pel = (ELEARBIN *) malloc(sizeof(ELEARBIN));
     pel->... = ... ;
     pel->... = ... ;
     ...
     pel->pdroite = NULL ;
     pel->pgauche = NULL ;

     return pel;
}


ELEARBIN * CONSARBIN (ELEARBIN *pel, ELEARBIN *parbgauche,
                      ELEARBIN *parbdroite)
{
     ELEARBIN *parbin;

     parbin = pel;
     parbin->pgauche = parbgauche ;
     parbin->pdroite = parbdroite ;

     return parbin;
}
```

5.4.5 INSERER

Cette fonction va prendre en argument un pointeur sur une structure de type ELEARBIN contenant les nouvelles données à introduire dans l'arbre et un pointeur sur la racine de l'arbre dans lequel on veut l'introduire.

Cette fonction peut prendre 2 formes suivant qu'on utilise ou non la récursivité. Notons que ces fonctions créent l'arbre de telle manière que tout élément plus petit que la racine se trouve dans son sous-arbre de gauche, tandis que tout élément plus grand que la racine se trouve dans le sous-arbre de droite, cette règle étant appliquée récursivement à tous les sous-arbres.

```
ELEARBIN * INSERER (ELEARBIN *pel, ELEARBIN *parbin)
{
     if (ARBIN_VIDE (parbin))
          parbin = CONSARBIN(pel, NULL, NULL) ;
     else
          if (pel->car > parbin->car)
               parbin->pdroite = INSERER (pel, parbin->pdroite) ;
          else
               parbin->pgauche = INSERER (pel, parbin->pgauche) ;

     return parbin ;
}
```

Version non récursive

```
ELEARBIN * INSERER (ELEARBIN *pel, ELEARBIN *parbin)
{
    ELEARBIN *pcher, *pprec;

    if (ARBIN_VIDE(parbin))
          parbin = CONSARBIN (pel, NULL, NULL);
    else
    {
          pcher = parbin;
          while (pcher != 0)
          {
                pprec = pcher;
                if (pel->car > pcher->car)
                      pcher = pcher->droite;
                else
                      pcher = pcher->gauche ;
          }
          if (pel->car > pprec->car)
                pprec->pdroite = pel;
          else
                pprec->pgauche = pel;
    }

    return parbin;
}
```

5.4.6 PARCOURIR

Pour les arbres binaires, les algorithmes de *parcours* les plus simples consistent à effectuer les trois actions suivantes dans un certain ordre :

- Traiter racine ;
- *Parcourir* le sous-arbre gauche ;
- *Parcourir* le sous-arbre droit.

Ces modes de parcours sont définis de façon récursives (comme le suggèrent les italiques). Il existe 6 modes de parcours possibles, que nous noterons GRD (Gauche, Racine, Droite), RGD, GDR, DRG, RDG, DGR.

RGD est aussi appelée **préfixe** ou **pré ordre** (racine d'abord, enfants ensuite) ; GDR **post fixe** ou **post ordre** (enfants d'abord, racine ensuite) ; GRD **infixe**.

Dans l'algorithme ci-dessous, la fonction TRAITER peut être n'importe quelle opération que l'on désire appliquer à chaque élément, par exemple une impression si l'on veut seulement avoir la liste des éléments dans l'ordre

```
PARCOURIR (ELEARBIN *parbre)
{
    if ( !ARBIN_VIDE (parbre))
    {
          PARCOURIR (parbre->pgauche) ;
          TRAITER (parbre->car) ;
          PARCOURIR (parbre->pdroite) ;
    }
}
```

5.4.7 RECHERCHER

Une application fort utile des arbres binaires, qui nous fournira un exemple de GRD, est la notion d'**arbre binaire de recherche**. Supposons **T** muni d'une relation d'ordre ; on peut alors convenir d'insérer systématiquement dans le sous-arbre de gauche d'un sous-arbre quelconque les éléments plus petits que la racine, et dans le sous-arbre de droite, les éléments plus grands. On en revient donc à l'arbre donné en exemple dans le paragraphe de la fonction INSERER. Cette organisation de l'arbre permet de faciliter la recherche d'un élément, intuitivement, on voit assez vite que s'il y a **n** éléments dans l'arbre, le nombre de comparaisons nécessaires pour accéder à l'un d'eux est au maximum la profondeur de celui-ci, de l'ordre de $\log_2 n$ si tout se passe bien, au lieu de **n** si les éléments étaient rangés en liste linéaire.

On obtient alors l'algorithme de recherche suivant :

```
ELEARBIN * RECHERCHER (ELEARBIN *parbre, int val)
{
     if (ARBIN_VIDE (parbre))
           return NULL;
     else
           if (val == parbre->val)
                 return parbre;
           else
                 if (val > parbre->val)
                       return RECHERCHER (parbre->pdroite,val);
                 else
                       return RECHERCHER (parbre->pgauche,val);
}
```

5.4.8 Hauteur

```
int HAUTEUR (ELEARBIN *parbin)
{
       int ht ;

       if(ARBIN_VIDE(parbin))
      {
           ht = 0 ;
      }
      else
     {
          ht = 1 + MAX(HAUTEUR(parbin→pgauche),HAUTEUR(parbin→pdroite)) ;
     }
     return ht ;
}
```

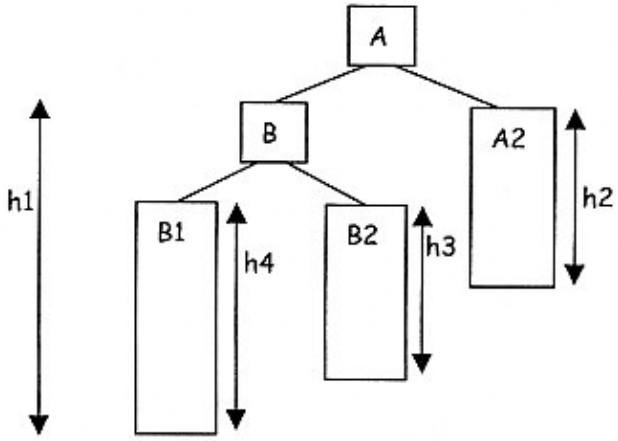
5.4.9 NBELEM

```
Int nbelem (ELEARBIN *parbin)
{
     Int nb ;

     If (ARBIN_VIDE(parbin))
     {
          nb = 0 ;
     }
     Else
     {
          nb = 1 + NBELEM(parbin->pgauche) + NBELEM('parbin->pdroite);
     }
     Return nb;
}
```

5.4.10.2.1 L'arbre penche à gauche

L'arbre penche à gauche à cause du sous arbre de gauche de son sous arbre de gauche.



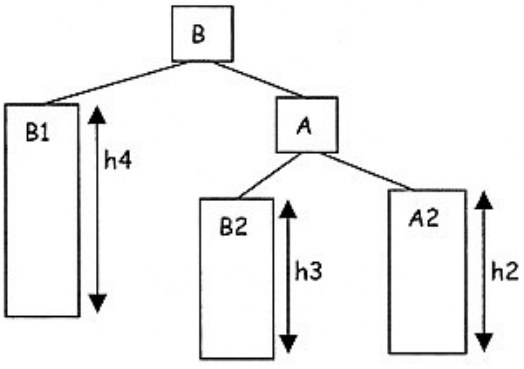
Tenant compte des hypothèses, à savoir que l'arbre est déséquilibré et que ce déséquilibre est provoqué par le sous arbre de gauche et en particulier par le sous arbre de gauche de ce sous arbre de gauche, ainsi que du schéma ci-dessus, on peut faire les constatations suivantes :

h1 = h2 + 2 (sinon, l'arbre ne serait pas déséquilibré ou le serait trop !)
h1 = h4 + 1 (car h1 tient compte de l'élément B)
h3 < h4 (par hypothèse)

Cela nous permet de tirer les relations suivantes :

h4 = h2 + 1
h3 = h2

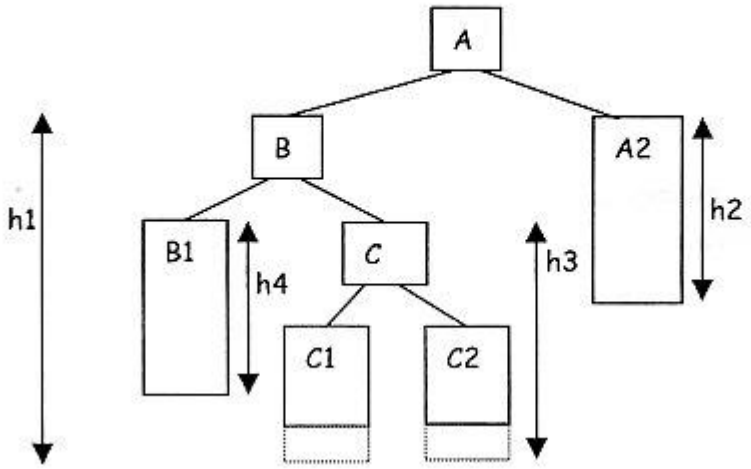
Nous allons réorganiser l'arbre de manière à ce qu'il soit à nouveau rééquilibré. Pour cela, nous allons considérer l'élément B comme la nouvelle racine de l'arbre et nous allons répartir les autres parties de l'arbre comme dans la figure suivante, afin d'obtenir à nouveau un arbre équilibré et qui respecte toujours la relation d'ordre : tous les éléments plus petits ou égaux à la racine sont dans le sous arbre de gauche, tandis que tous les éléments plus grands sont dans les sous arbre de droite.



Puisque h2+1 = h4 et h2 = h3, cet arbre est à nouveau équilibré !

L'ordre établi dans l'arbre a également été conservé : tout élément qui se trouvait à gauche d'un élément quelconque est resté à gauche de cet élément tandis que tout élément qui se trouvait à droite d'un élément quelconque est resté à droite de cet élément.

L'arbre penche à gauche à cause du sous arbre de droite de son sous arbre de gauche.



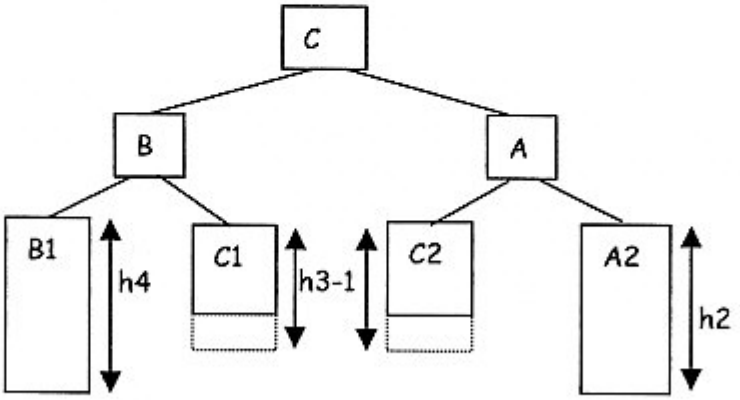
Nous pouvons à nouveau faire quelques constatations :

$h1 = h2 + 2$
$h1 = h3 + 1$
$h4 < h3$

Ce qui implique que :

$h3 = h2 + 1$
$h4 = h2$

Cette fois-ci, c'est l'élément C qui va jouer le rôle de racine dans le nouvel arbre :

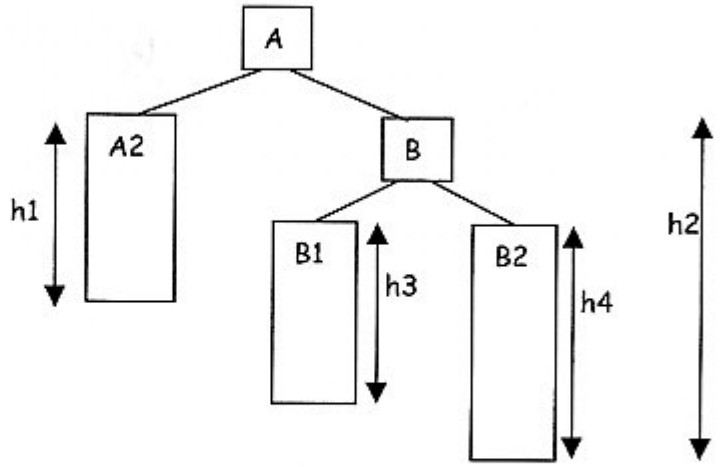


Puisque h3-1 = h2 et h4 = h2, cet arbre est à nouveau équilibré.

Et en suivant le même raisonnement que dans le cas précédent, l'arbre est resté organisé suivant la relation d'ordre définie au départ.

5.4.10.2.2 L'arbre penche à droite

L'arbre penche à droite à cause du sous arbre de droite de son sous arbre de droite.



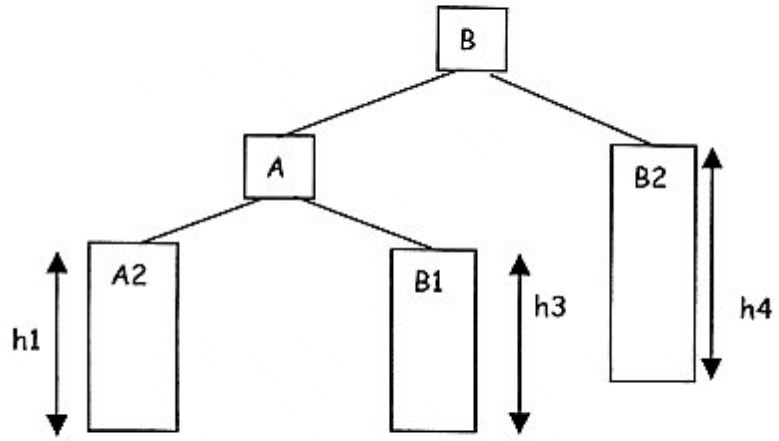
Les hypothèse et le schéma nous permettent de dire :

$h2 = h1 + 2$
$h2 = h4 + 1$
$h3 < h4$

et donc

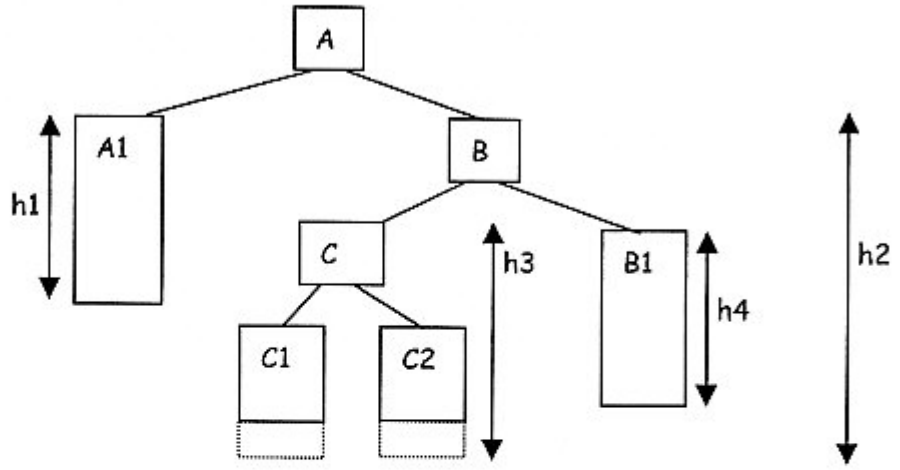
$h4 = h1 + 1$
$h3 = h1$

Cette fois-ci, B devient la racine du nouvel arbre équilibré :



Et puisque h4 = h1+1 et h4 = h3+1, on peut à nouveau conclure que l'arbre a été rééquilibré. L'ordre est toujours conservé, suivant le même principe que chaque élément a gardé la même position relative par rapport à tous les autres au niveau gauche/droite.

L'arbre penche à droite à cause du sous arbre de gauche de son sous arbre de droite.



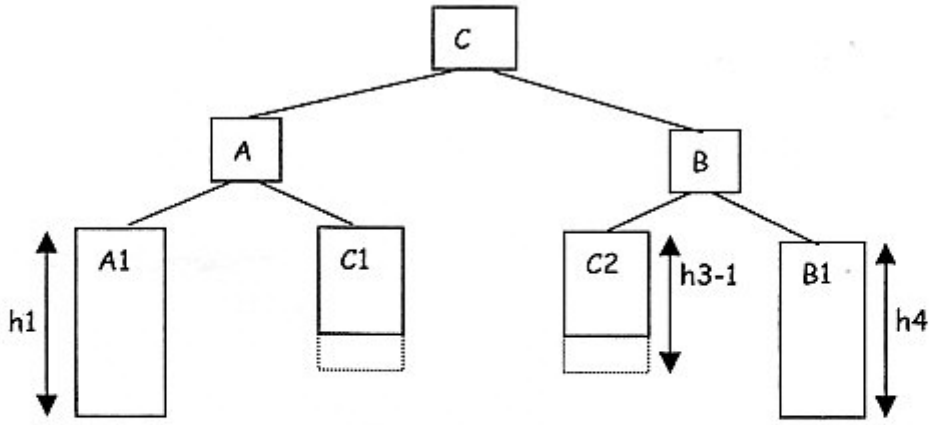
Dans ce dernier cas,

$h2 = h1 + 2$
$h2 = h3 + 1$
$h4 < h3$

et donc

$h3 = h1 + 1$
$h4 = h1$

C devient la nouvelle racine de l'arbre équilibré, ce qui donne :



Et puisque h4 = h1et h1 = h3-1 ou h1-1 = h3-1, on peut à nouveau conclure que l'arbre a été rééquilibré. L'ordre est toujours conservé, suivant le même principe que chaque élément a gardé la même position relative par rapport à tous les autres au niveau gauche/droite.

5.4.10.3 Algorithme

Nous allons maintenant traduire les transformations faites au paragraphe précédent en fonctions C. Pour cela, nous aurons besoin pour la plus grande clarté de l'algorithme des 2 fonctions suivantes :

SOUSARBG(parbin) et **SOUSARBD(parbin)** qui vont prendre en argument l'adresse de l'arbre et renvoyer respectivement le sous arbre de gauche et le sous arbre de droite. leur écriture ne pose aucune difficulté et elles vont rendre de précieux services pour la rédaction et la lecture de notre fonction.

La fonction **RACINE**, quant à elle, retourne la valeur de la racine de l'arbre qu'elle reçoit en paramètre.

```
ELEARBIN *REEQUILIBRER (ELEARBIN *parbin)
{
  if (HAUTEUR(SOUSARBG (parbin))>(HAUTEUR(SOUSARBD(parbin))+1))
  {
    if (HAUTEUR (SOUSARBG (SOUSARBG(parbin)))>
                                    HAUTEUR(SOUSARBD( SOUSARBG(parbin))))
    {
      parbin =CONSARBIN(RACINE (SOUSARBG (parbin)),
                        SOUSARBG(SOUSARBG(parbin)),
                        CONSARBIN(RACINE(parbin),
                                  SOUSARBD(SOUSARBG (parbin)),
                                  SOUSARBD(parbin) )  );

    }
    else
    {
      parbin= CONSARBIN(RACINE (SOUSARBD (SOUSARBG(parbin))),
                        CONSARBIN(RACINE(SOUSARBG(parbin)),
                                  SOUSARBG(SOUSARBG (parbin)),
                                  SOUSARBG(SOUSARBD(SOUSARBG (parbin)))),
                        CONSARBIN(RACINE(parbin),
                                  SOUSARBD(SOUSARBD(SOUSARBG(parbin))),
                                  SOUSARBD(parbin) ) );
    }
  else
  {
    if (HAUTEUR (SOUSARBD(SOUSARBD(parbin))) >
                                  HAUTEUR(SOUSARBG( SOUSARBD(parbin))))
    {
      parbin =CONSARBIN(RACINE (SOUSARBD(parbin)),
                        CONSARBIN(RACINE(parbin),
                                  SOUSARBG(parbin),
                                  SOUSARBG(SOUSARBD (parbin)) ),
                        SOUSARBD(SOUSARBD(parbin))  );

    }
    else
    {
      parbin= CONSARBIN(RACINE (SOUSARBG (SOUSARBD(parbin))),
                        CONSARBIN(RACINE(parbin),
                                  SOUSARBG(parbin),
                                  SOUSARBG(SOUSARBG(SOUSARBD(parbin)))),
                        CONSARBIN(RACINE(SOUSARBD(parbin)),
                                  SOUSARBD(SOUSARBG(SOUSARBD (parbin))),
                                  SOUSARBD(SOUSARBD (parbin))  )  );

    }
  }

retun(parbin) ;
}
```

6.1.1 Technique classique

```
i ⇦ 1
TANT QUE  i < n+1  ETC  t[i] != cible  FAIRE
      i ⇦ i+1
FIN TANT QUE
SI
      ALORS position ⇦ i
      SINON position ⇦ -1
FIN SI
```

|| sortie => i = n+1 ou (i ≤ n et t[i] = cible) ||

Le nombre d'itérations de cet algorithme se situe entre n/2 (cas où la présence est presque certaine) et n (cas où la présence a peu de chance d'arriver). On a donc O.C.T = O(n)

Quant au nombre d'opérations par itération, il est de 3.

6.1.2 Technique du booléen

Dans cette technique, on utilise une variable booléenne **trouve** qui passera de 0 à 1 si la cible est présente et a été trouvée dans le tableau.

```
i ⇦ 1
trouve ⇦ 0
position ⇦ -1
TANT QUE  i < n+1  ETC  trouve = 0  FAIRE
      SI  t[i] = cible
            ALORS  trouve ⇦ 1
                   position ⇦ i
            SINON  i ⇦ i+1
      FIN SI
FIN TANT QUE
```

|| sortie => i = n+1 ou trouve = 1 ||

Cet algorithme demande le même nombre d'itérations que le précédent, par contre, on peut compter que le nombre d'opérations par itération est ici de 4.

6.1.3 Technique du forçage de boucle

Dans cette technique, on suppose que la recherche va se faire dans tout le vecteur (i < n+1), la boucle s'arrêtera lorsqu'on aura trouvé la cible en donnant à i une valeur plus grande que n.

```
i ⇦1
position ⇦-1
TANT QUE  i < n+1  FAIRE
      SI  t[i] = cible
            ALORS  position ⇦i
                   i ⇦n+1
            SINON  i ⇦i+1
      FIN SI
FIN TANT QUE
```

|| sortie => i = n+1 ou t[position] = cible ||

Le nombre d'itérations est à nouveau le même. Par contre, le nombre d'opérations par itération est de 3. La variable position n'étant affectée que une ou deux fois, son introduction n'entraîne pas d'augmentation du nombre total d'opérations.

6.1.4 Technique de la sentinelle

Cette technique consiste à placer la valeur de la cible après la dernière donnée de l'ensemble dans lequel s'effectue la recherche. Dans un tableau contenant n informations, on place la valeur cible dans la case n+1. Il est évident que dans ce cas, toute recherche va aboutir. On pourra conclure sur la présence de la cible dans le vecteur si on a trouvé la cible avant la case n+1.

Ce qui donne l'algorithme suivant :

```
i ⇦1
t[n+1] ⇦cible
TANT QUE  t[i] != cible  FAIRE
      i ⇦i+1
FIN TANT QUE
SI i < n+1
      ALORS position ⇦i
      SINON position ⇦-1
FIN SI
```

6.3.3 Ordre de complexité approximatif de cette méthode (O.C.T.)

Le tableau initial contient **n** objets ou clés.

Après la première comparaison,	il reste	n/2 clés	soit $n/2$
Après la deuxième	il reste	n/4 clés	soit $n/2^2$
Après la troisième	il reste	n/8 clés	soit $n/2^3$
Après la quatrième	il reste	n/16 clés	soit $n/2^4$
Après la $X^{ième}$	il reste		$n/2^X$

Pour que la recherche soit terminée, il faut que le vecteur soit réduit à une seule clé. Le nombre d'itérations sera donc égal à un nombre X tel que

$$n/2^X \leq 1$$

Autrement dit, **X** est le nombre de fois que l'on peut diviser **n** par 2 pour trouver un nombre inférieur ou égal à 1.

Ou bien, **x** est le nombre de fois que l'on peut multiplier 1 par 2 pour trouver un nombre supérieur à **n**.

Soit $2^x = n \rightarrow x = \log_2 n$

Remarque :

Si n = 100 $2^6 = 64$

$2^7 = 128$

On va pouvoir arrêter la recherche après 6 ou 7 itérations !

Si n = 1000 $2^8 = 256$

$2^9 = 512$

$2^{10} = 1024$

La recherche ne demandera que 10 itérations soit seulement 3 de plus !

Soit :

```
inf ⇦ 1
sup ⇦ n
TANT QUE  inf < sup  FAIRE
      m ⇦ (inf + sup)/2
      SI cible ≤ t[m]
            ALORS sup ⇦ m
            SINON inf ⇦ m + 1
      FIN SI
FIN TANT QUE
SI t[inf] = cible
      ALORS position ⇦ inf
      SINON position ⇦ -1
FIN SI
```

7.1.1 Le tri bulle

7.1.1.1 Méthode

L'idée est de comparer 2 éléments successifs du vecteur, si ceux-ci ne sont pas dans l'ordre souhaité (croissant par exemple), on les permute.

Si on fait cette opération depuis les 2 premiers jusqu'aux 2 derniers, on aura « poussé » le plus grand en dernière position.

Si le vecteur contenait **n** éléments au départ, il suffira de recommencer l'opération pour le vecteur des n-1 premiers éléments pour obtenir de nouveau le plus grand élément en dernière position. Pas à pas, les derniers éléments seront triés en ordre croissant.

On recommencera l'opération tant qu'il restera encore 2 éléments.

```
j ⇦n
bool ⇦0
TANT QUE  bool = 0
     i ⇦1
     bool ⇦1
     TANT QUE  i < j  FAIRE
          SI t[i] > t[i+1]
               ALORS Echanger (t[i] et t[i+1])
                     bool ⇦0
          FIN SI
          i ⇦i+1
     FIN TANT QUE
     j ⇦j - 1
FIN TANT QUE
```

7.1.2 Le tri par extraction

7.1.2.1 Méthode

L'idée est d'**extraire** des éléments du vecteur, soit le plus grand, soit le plus petit, et de l'installer en bonne position, c'est-à-dire à la fin du vecteur pour le plus grand.

Lorsqu'on a fait l'opération pour le vecteur entier de **n** éléments, on recommence pour le vecteur des **n-1** premiers et ainsi de suite.

7.1.2.2 Algorithme

```
j ⇦n
TANT QUE  j > 1 FAIRE
     max ⇦t[1]
     imax ⇦1
     i ⇦2
     TANT QUE  i <= j  FAIRE
          SI t[i] > max
               ALORS     max ⇦t[i]
                         imax ⇦i
          FIN SI
          i ⇦i+1
     FIN TANT QUE
     t[imax] ⇦t[j]
     t[j] ⇦max
     j ⇦j - 1
FIN TANT QUE
```

7.1.3 Le tri par insertion

7.1.3.1 Méthode

On partage le vecteur en 2 parties, la partie triée et la partie non triée. On prend le premier élément de la partie non triée que l'on met dans une variable réserve. On déplace d'une case vers la droite tous les éléments plus grands que lui dans la partie triée ; puis on l'installe en bonne place.

Exemple :

2	**4**	**7**	**10**	3	8	15	20
res ⇦3							
2	4	7	10	10	8	15	20
2	4	7	7	10	8	15	20
2	4	4	7	10	8	15	20
2	**3**	**4**	**7**	**10**	8	15	20

Au premier tour, le vecteur trié est réduit à un seul élément, le premier élément à **insérer** est donc celui de la case 2

7.1.3.2 Algorithme

```
POUR  j  ALLANT DE  2  A  n FAIRE
    i ⇦j - 1
    res ⇦t[j]
    TANT QUE  i ≥ 1  ET  t[i] > res  FAIRE
          t[i+1] ⇦t[i]
          i ⇦i-1
    FIN TANT QUE
    t[i+1] ⇦res
FIN POUR
```

La condition i ≥ 1 sert à vérifier si on ne se trouve pas en dehors du vecteur, ce qui pourrait arriver si on ne trouvait pas de valeur inférieure à **res** dans celui-ci.

Cette condition ajoute bien sûr une opération par itération.

On peut améliorer l'algorithme en plaçant dans la case précédant le début du vecteur (case 0 par exemple), une valeur plus petite que n'importe quelle valeur qui pourrait intervenir dans celui-ci et que l'on appelle sentinelle. Il faut cependant être certain de pouvoir disposer de cet emplacement.

L'algorithme devient :

```
t[0] ⇦plus_petit (on place une valeur plus petite que toutes celles
                                  pouvant intervenir dans le vecteur)
POUR  j  ALLANT DE  2  A  n FAIRE
    i ⇦j - 1
    res ⇦t[j]
    TANT QUE  t[i] > res  FAIRE    (la condition i ≥ 1 est
                                  supprimée)
          t[i+1] ⇦t[i]
          i ⇦i-1
    FIN TANT QUE
    t[i+1] ⇦res
FIN POUR
```

7.2.1 Description de la méthode du heap-sort

La méthode de ce tri consiste à avancer en première position (de manière très originale et particulière) le plus grand élément d'un tableau de n éléments, à le mettre à sa place en dernière position dans le tableau en l'échangeant avec l'élément situé à cet endroit puis à refaire l'opération pour le tableau de n-1 éléments

Il suffit d'ignorer la dernière case et de reproduire ensuite le même procédé.

Pour expliquer la méthode qui va servir à extraire et positionner le plus grand élément en tête du tableau il faut comprendre l'introduction suivante.

```
TRI_EPI (t, 1, n)
{
     POUR i allant de n/2 à 1 par pas de -1 FAIRE
          PATERNER (t, i, n)
     FIN POUR
     POUR i allant de n à 2 par pas de -1 FAIRE
          Echanger (t[1], t[i])
          PATERNER (t, 1, i-1)
     FIN POUR
}
```

```
PATERNER( t , inf , sup )
{
     ipere←inf              {initialisation de la case à examiner}
     ifils←ipere*2          {indice du premier fils}
     tmp← t[ipère]

     TANT QUE ifils<= sup FAIRE

          SI ifils<sup ET t[ifils+1]>t[ifils] ALORS ifils←fils+1
                         {choix du plus grand des 2fils
                         lorsqu'il y en a 2}

          SI t[ifils]> tmp
               ALORS
                    t[ipere] ←t[ifils]
                    ipere←fils          {on doit recommencer pour
                                         vérifier le niveau en dessous}
                    ifils←ipere*2


               SINON
                    Sortir de la boucle
                         {en effet puisqu'on a fait aucun échange
                         rien ne risque d'être perturbé dans les
                         niveaux inférieurs.}
          FIN SI
     FIN TANT QUE
     t[ipere]←tmp
}
```