RECURSIVITE
ARBRES BINAIRES

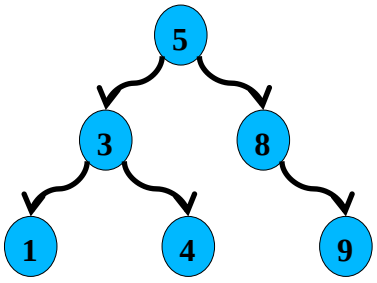
Insertion,
Parcours pré, post et in ordre,
Recherche,
Suppression.

ARBRE BINAIRE DEFINITION RECURSIVE

Les arbres binaires sont des arbres dont les nœuds n'acceptent que deux fils au maximum.

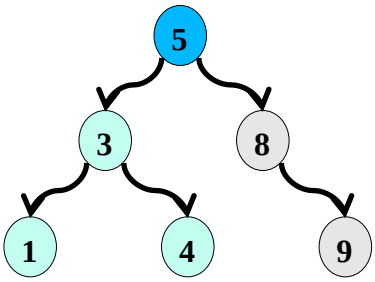
Un arbre binaire est :
- soit un arbre vide (NULL en C)
- soit un triplet (I,G,D) constitué d'une racine I et de deux arbres binaires :
  - G (fils gauche)
  - D (fils droit).

Les arbres binaires sont des structures utilisées pour le classement et l'accès rapide aux données



ARBRE BINAIRE ORDONNE

On ne s'intéressera qu'aux arbres binaires ordonnés :



Tout élément à gauche de la racine est inférieur à la valeur de la racine,

Tout élément à droite de la racine est supérieur à la valeur de la racine.

Cette propriété doit être vérifiée récursivement à tous les niveaux pour que l'arbre binaire soit dit ordonné.

DONNEES MANIPULEES

```
typedef struct noeud
{
  char info[32] ;
  struct noeud* gauche;
  struct noeud * droit;
} NOEUD ;
```

La création d'un nouveau nœud est une fonction indépendante :

NOEUD* CreerNoeud(void)

Elle est appelée avant l'appel de la fonction d'insertion.

INSERTION A LA BONNE PLACE PRINCIPE RECURSIF

L'insertion d'une nouvelle information donne lieu à la création d'un nœud.

La fonction d'insertion réclame donc 2 paramètres :

NOEUD* AjNoeud(NOEUD *R, NOEUD *N)

- R est le nœud courant (la racine au début)
- N est le nœud crée précédemment,
- Elle retourne la nouvelle adresse du nœud courant (qui n'a changé qu'en cas d'insertion).

CREATION D'UN NOEUD

```
NOEUD* CreerNoeud(void)
{ NOEUD* N=(NOEUD *)malloc(sizeof(NOEUD));
  if (N == NULL)
  { printf("\nErreur allocation memoire sortie");
    return NULL;
  }
  printf("\nDonner un mot");
  scanf("%s", N->info);
  N->gauche = NULL;
  N->droit = NULL;
  return N;
} /* fin CreerNoeud */
```

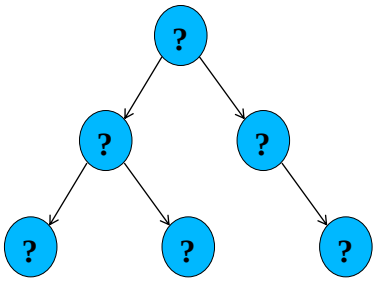
Attention : la création d'un noeud doit se faire en dehors de l'insertion récursive

INSERTION A LA BONNE PLACE

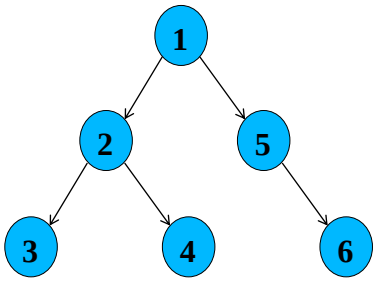
```
NOEUD* AjNoeud(NOEUD *R, NOEUD *N)
{ if (R == NULL) return N;
  if (strcmp(N->info,R->info)<0)
     R->gauche = AjNoeud(R->gauche,N);
  else
     if (strcmp(N->info,R->info)>0)
        R->droit = AjNoeud(R->droit, N);
     else
     { printf("\nLe mot existe, pas d'ajout");
       free(N);
     }
  return R;
} /* fin AjNoeud */
```


PARCOURS PRE-ORDRE

```
void ParcoursPre(NOEUD *R)
{
  if (R != NULL)
  {
    printf("\n%s", R->info);
    ParcoursPre(R->gauche);
    ParcoursPre(R->droit);
  }
} /* fin ParcoursPre */
```



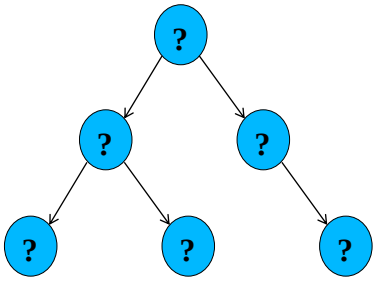
PARCOURS PRE-ORDRE : AFFICHAGE



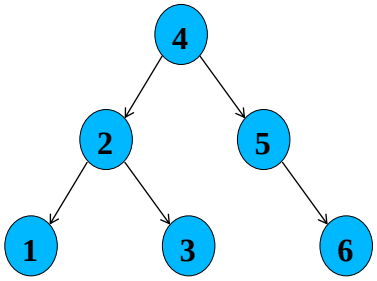
Parcours en pré-ordre :
le traitement (printf) se fait avant les appels récursifs

PARCOURS IN-ORDRE

```
void ParcoursIn(NOEUD *R)
{
  if (R != NULL)
  {
    ParcoursIn(R->gauche);
    printf("\n%s", R->info);
    ParcoursIn(R->droit);
  }
} /* fin ParcoursIn */
```



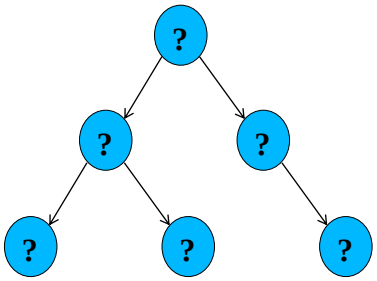
PARCOURS IN-ORDRE : AFFICHAGE



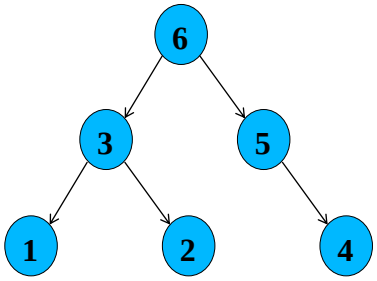
Parcours en in-ordre :
le traitement (printf) se fait entre l'appel récursif à la branche gauche et l'appel récursif à la branche droite
l'affichage se fait dans le bon ordre

PARCOURS POST-ORDRE

```
void ParcoursPost(NOEUD *R)
{
  if (R != NULL)
  {
    ParcoursPost(R->gauche);
    ParcoursPost(R->droit);
    printf("\n%s", R->info);
  }
} /* fin ParcoursPost */
```



PARCOURS POST-ORDRE : AFFICHAGE



Parcours en post-ordre :
le traitement (printf) se fait après les traitements de la branche gauche et de la branche droite

RECHERCHE – PRINCIPE

La recherche d'une information se fait en log2(n) opérations (au lieu de n opérations au pire).
Ainsi pour 1024 valeurs la recherche pour se faire en 10 opérations.
Attention cela suppose que l'arbre binaire soit équilibré.

Méthode de parcours utilisée :

- A chaque nœud : comparer la valeur recherchée avec la valeur du nœud.
  - Si c'est la même arrêter,
  - Si elle est plus petite, relancer récursivement la recherche sur la branche gauche,
  - Si elle est plus grande, relancer récursivement la recherche sur la branche droite,

DESALOCATION DES NOEUDS D'UN ARBRE

```
void DesalouerArbre(NOEUD *R)
{
if (R != NULL)
  {
    DesalouerArbre(R->gauche);
    DesalouerArbre(R->droit);
    free(R);
  }
}
```

Remarque : la désalocation est un traitement de type post-ordre

RECHERCHER UNE INFORMATION

```
int fRechercherMot(NOEUD *A, char motRech[])
{int icmp=0;
 if (A==NULL)
   { printf("\nMot non trouvé"); return -1; }
 icmp= strcmp(motRech,A->info);
   printf("\nvaleur courante : %s ",A->info);
 if (icmp==0)
   { printf("\nMot trouvé"); return 0; }
. . .
```

RECHERCHER UNE INFORMATION

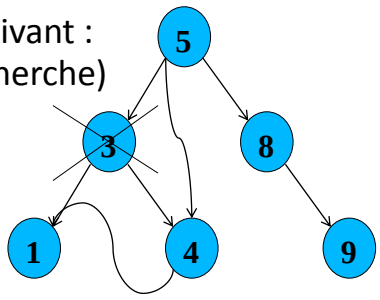
```
int fRechercherMot(NOEUD *A, char motRech[])
{int icmp=0;
 if (A==NULL)
   { printf("\nMot non trouvé"); return -1; }
 icmp= strcmp(motRech,A->info);
   printf("\nvaleur courante : %s ",A->info);
 if (icmp==0)
   { printf("\nMot trouvé"); return 0; }
 if (icmp<0)
   return fRechercherMot(A->gauche,motRech);
 if (icmp>0)
   return fRechercherMot(A->droit,motRech);
} /* fin fRechercherMot */
```


SUPPRESSION D'UN NOEUD : PRINCIPE

La suppression d'une information est en général suivie d'un ré-équilibrage de l'arbre. Le cas traité ici ne se préocupe pas de rééquilibrage.

Le principe de la suppression est le suivant :

- Si le nœud est trouvé (voir la recherche)
- Le remplacer par son fils droit,
- Insérer complètement à gauche du fils droit l'ancien fils gauche



SUPPRESSION D'UN NOEUD : INSERTION A L'EXTREME GAUCHE

```
NOEUD *fInsererGauche(NOEUD *A, NOEUD *Ins)
{
  if(A== NULL)
    return Ins;

  A->gauche=fInsererGauche(A->gauche,Ins);
  return A;
} /* fin fInsererGauche */
```


SUPPRESSION D'UN NOEUD : FONCTION

```
NOEUD *fSuppMot(NOEUD *A, char motS[])
{ int icmp=0; NOEUD *S=NULL;
  if(A==NULL)
   {  printf("\nMot non trouvé!"); return NULL;}
  icmp= strcmp(motS,A->info);
  if (icmp==0)
   {  printf("\nMot trouvé!"); S=A;
      A=fInsererGauche(A->droit,A->gauche);
      free(S); return A;  }
. . .
```

SUPPRESSION D'UN NOEUD : FONCTION

```
NOEUD *fSuppMot(NOEUD *A, char motS[])
{ int icmp=0; NOEUD *S=NULL;
  if(A==NULL)
   {  printf("\nMot non trouvé!"); return NULL;}
  icmp= strcmp(motS,A->info);
  if (icmp==0)
   {  printf("\nMot trouvé!"); S=A;
      A=fInsererGauche(A->droit,A->gauche);
      free(S); return A;  }
  if (icmp<0)
    { A->gauche=fSuppMot(A->gauche,motS);
      return A; }
  if (icmp>0)
    { A->droit=fSuppMot(A->droit,motS);
      return A; }
}  /* fin fSupprimerMot */
```