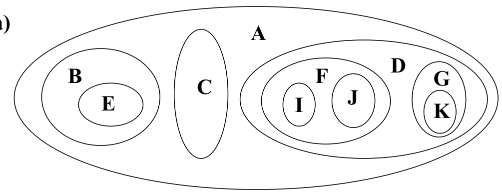
Arbres

On appelle **arbre** de type T une structure formée d'une donnée de type T, appelée **racine** de l'arbre, et d'un ensemble fini de taille variable, éventuellement vide, d'arbres de type T, appelés sous- arbres de l'arbre.

a)

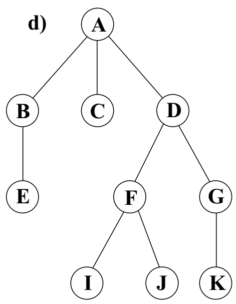


b) (A (B (E)) (C) (D (F (I) (J)) (G (K))

c)

1 A
2 B
3 E
2 C
2 D
3 F
4 I
4 J
3 G
4 K

d)



Une **feuille** est la racine d'un sous- arbre n'ayant pas lui- même de sous- arbre.

Un **noeud (sommet)** est
- une racine d'un arbre ou
- une racine d'un sous- arbre ou
- une feuille.

Un noeud se trouvant directement sous un autre noeud s'appelle son **enfant,** son **fils** ou son **descendant direct**; celui- ci est inversement appelé le **parent** ou **l'ascendant** de celui- là.

Deux enfants du même parent sont **frères**.

Le **degré** d'un noeud est égal au nombre de ses enfants.

Un **arbre binaire** est un arbre ordonné de degré 2: un ensemble fini de noeuds qui peut être vide ou comporter une racine avec deux sous- arbres binaires disjoints.

Type abstrait Arbre.

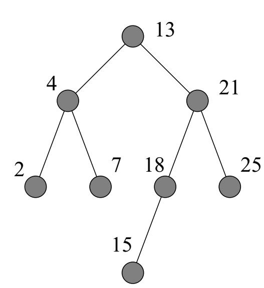
```
typedef ... Info;
class arbre {
public:
      bool Vide ();
      arbre (Info V);
                  // crée un nouvel arbre binaire contenant un
                  // noeud avec le champ d'information égal à V
      Info Valeur ();
                  // accès au champ d'information de la racine
                  // de l'arbre
      void CreerGauche (Info V);
                  // crée le fils gauche avec le champ
                  // d'information égal à V
      void CreerDroit (Info V);
                  // crée le fils droit avec le champ
                  // d'information égal à V
      arbre Gauche ();
                  // accès au sous- arbre gauche
      arbre Droit ();
                  // accès au sous- arbre droit
      arbre Pere ();
                  // accès au père
private:
  ...
};
```


Exemple: Application des arbres binaires

Compter le nombre de répétitions dans une liste de nombres:
5 25 10 5 35 25 0 5 35

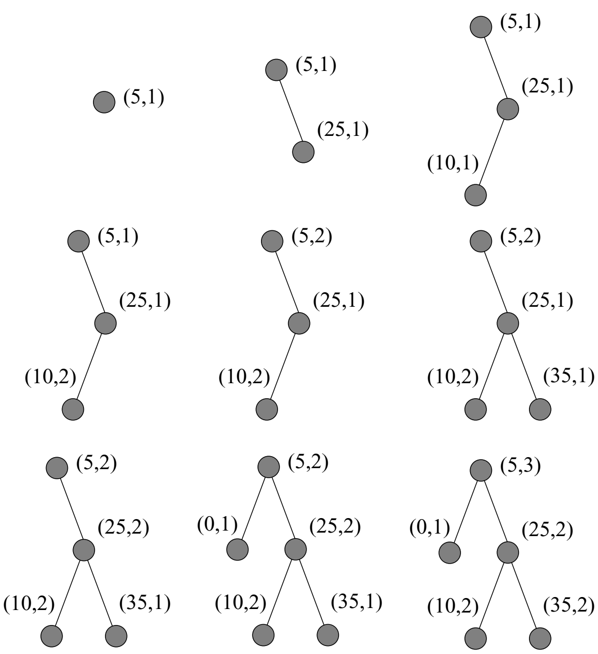
5 - 3 fois
25 - 2 fois
10 - 1 fois
35 - 2 fois
0 - 1 fois

Idée: représenter les valeurs dans un arbre binaire de sorte que: pour tout noeud N toutes les clés du sous- arbre gauche sont inférieures à la clé du noeud et toutes les clés du sous- arbre droit sont supérieures à la clé du noeud .



Les données: 5 25 10 5 35 25 0 5 35

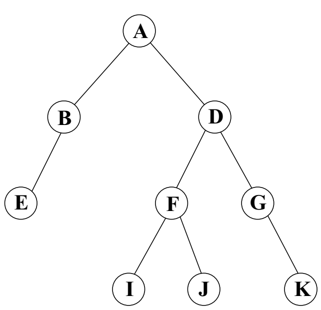
La création de l'arbre:




```
#include "arbre. h"
typedef struct {
        int V, Compte;
    } rec;
typedef rec* Info;

main() {
    arbre A, Racine, Pere;
    rec R; bool Trouve; int Nombre;
    cin >> Nombre; R.V = Nombre; R.Compte = 1;
    Racine = arbre(& R);
    cin >> Nombre;
    while (Nombre > -1) { // il reste des nombres
        Trouve = false;
        A = Racine;
        while (! Trouve && !A.Vide()) {
            Pere = A;
            if (A.Valeur()-> V == Nombre) {
                A.Valeur()->Compte++;
                Trouve = true; }
            else if (A.Valeur()->V < Nombre) A = A.Droit();
            else A = A.Gauche();
        }
        if (! Trouve) {
            R.V = Nombre; R.Compte = 1;
            if (Pere.Valeur()->V < Nombre) Pere.CreerDroit(& R);
                else Pere. CreerGauche(& R);
        } cin >> Nombre;
    }
    Impression( Racine);
}
```


Parcours de l'arbre binaire.



Parcours de l'arbre en ordre

a) préfixé A B E D F I J G K
- visiter la racine
- visiter le SAG
- visiter le SAD

b) infixé E B A I F J D G K
- visiter le SAG
- visiter la racine
- visiter le SAD

c) postfixé E B I J F K G D A
- visiter le SAG
- visiter le SAD
- visiter la racine

PARCOURS-INFIXE (x)
si x ≠ NIL
  alors PARCOURS-INFIXE (*gauche[x]*)
    afficher *clé[x]*
    PARCOURS-INFIXE (*droite[x]*)

PARCOURS-PREFIXE (x)
si x ≠ NIL
  alors afficher *clé[x]*
    PARCOURS-PREFIXE (*gauche[x]*)
    PARCOURS-PREFIXE (*droite[x]*)

PARCOURS-POSTFIXE (x)
si x ≠ NIL
  alors PARCOURS-POSTFIXE (*gauche[x]*)
    PARCOURS-POSTFIXE (*droite[x]*)
    afficher *clé[x]*

Si x est la racine d'un sous-arbre à n noeuds, alors chacun des appels PARCOURS-INFIXE (x), PARCOURS-PREFIXE (x) et PARCOURS-POSTFIXE (x) prend un temps θ(n).

Arbres binaires de recherche.

Soit un ensemble de clés K et une relation d'ordre "<" définie sur K. Un arbre binaire de recherche pour (K, <) possède des noeuds contenant les valeurs de K, de sorte que: pour tout noeud N toutes les clés du sous- arbre gauche sont inférieures à la clé du noeud et toutes les clés du sous- arbre droit sont supérieures à la clé du noeud.

Nous seront intéressés d'effectuer les opérations suivantes définies pour un arbre binaire de recherche enraciné dans x et une clé k∈ K:
- RECHERCHER (x,k)
- MINIMUM (x)
- MAXIMUM (x)
- SUCCESSEUR (x,k)
- PRÉDÉCESSEUR (x,k)

Recherche

```
ARBRE-RECHERCHER (x,k)
si x = NIL ou k = clé[x]
    alors  retourner x
si  k < clé[x]
    alors  retourner ARBRE-RECHERCHER (gauche[x],k)
    alors  retourner ARBRE-RECHERCHER (droite[x],k)
```

```
ARBRE-RECHERCHER-ITÉRATIF (x,k)
tant que x ≠ NIL ou k ≠ clé[x]
    faire si k < clé[x]
            alors x ← gauche[x]
            sinon x ← droite[x]
retourner x
```

Minimum et maximum

```
ARBRE-MINIMUM (x)
tant que gauche[x] ≠ NIL
    faire   x ← gauche[x]
retourner x
```

```
ARBRE-MINIMUM (x)
tant que droite[x] ≠ NIL
    faire   x ← droite[x]
retourner x
```


Successeur et prédécesseur

```
ARBRE-SUCCESSEUR (x)
si droite[x] ≠ NIL
    alors  retourner ARBRE-MINIMUM (droite[x])
y ← p[x]
tant que y ≠ NIL et x = droite[y]
    faire   x ← y
            y ← p[y]
retourner y
```

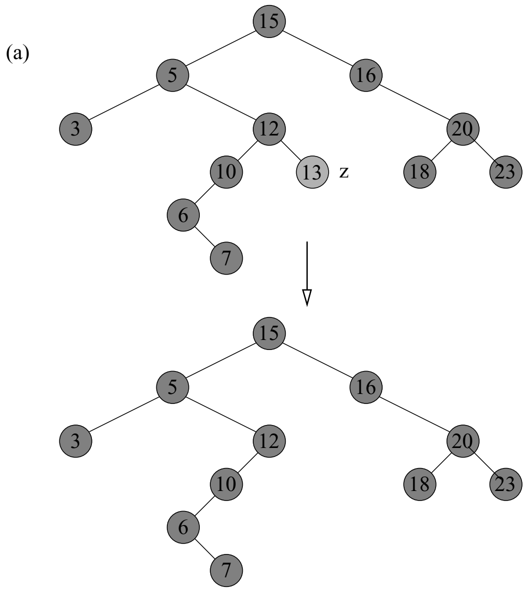
Les opérations d'ensemble dynamique RECHERCHER, MINIMUM, MAXIMUM, SUCCESSEUR et PRÉDÉCESSEUR peuvent se faire en temps *O(h)* sur un arbre binaire de recherche de hauteur *h*.

Insertion et suppression

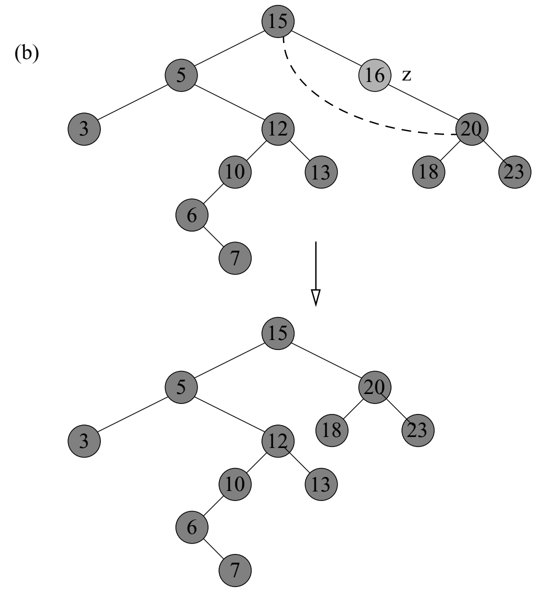
Insertion

```
ARBRE-INSÉRER (T,z)
    // clé[z] = v,  gauche [z] = NIL et droite[z] = NIL
y ← NIL
x ← racine[T]
tant que x ≠ NIL
    faire   y ← x
            si clé[z] < clé[x]
                alors x ← gauche[x]
                sinon x ← droite[x]
p[z] ← y
si y = NIL
    alors racine[T] ← z              // arbre T était vide
    sinon  si clé[z] < clé[y]
            alors gauche[y] ← z
            sinon droite[y] ← z
```

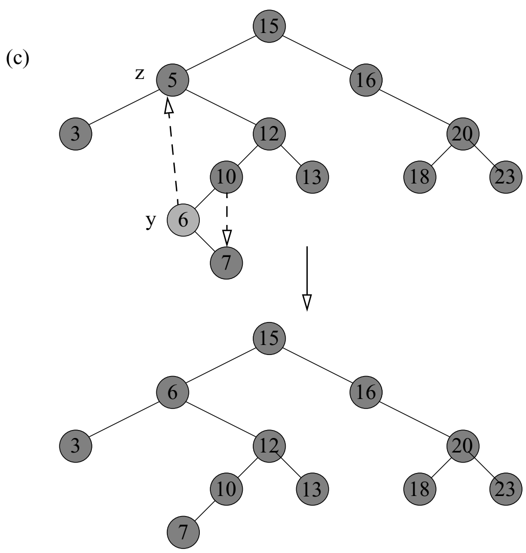
Suppression



Suppression d'un noeud z d'un arbre binaire de recherche. Pour savoir quel est le noeud à supprimer effectivement, on regarde le nombre d'enfants de z (ce noeud est colorié en gris clair). (a) Si z n'a pas d'enfant, on se contente de le supprimer.



(b) Si z n'a qu'un seul enfant, on détache z.



(c) Si z a deux enfants, on détache son successeur y qui possède au plus un enfant et on remplace alors la clé et les données satellites de z par celles de y.

ARBRE-SUPPRIMER (T,z)

```
si gauche[z] = NIL ou droite[z] = NIL
   alors y ← z
   sinon y ← ARBRE-SUCCESSEUR (z)
si gauche[y] ≠ NIL
   alors x ← gauche[y]
   sinon x ← droite[y]
si x ≠ NIL
   alors p[x] ← p[y]
si p[y] = NIL
   alors racine[T] ← x
   sinon  si y = gauche[p[y]]
             alors gauche[p[y]] ← x
             sinon droite[p[y]] ← x
si y ≠ z
   alors  clé[z] ← clé[y]
          copier données satellites de y dans z
retourner y
```

On peut exécuter les opérations INSÉRER et SUPPRIMER d'ensemble dynamique en temps *O(h)* sur un arbre binaire de recherche de hauteur *h*.

On peut prouver, que la hauteur moyenne d'un arbre de recherche construit aléatoirement à partir de *n* clés est $O(\lg n)$.

Représentation des arbres binaires. Représentation tableau.

```
typedef int arbre;          // 0..1000
struct noeud {
      info val;
      arbre g, d;
};
noeud N[1000];
       ...
// création de la liste libre
for (int I= 0; I< 999; I++)
      N[I].g = I + 1;
N[999].g = -1;
int Libre = 0;

arbre CreerArbre( info X) {
      // joue le rôle du constructeur
      if (Libre == -1)
            Erreur (" arbre: debordement");
      else {
            arbre A = Libre;
            Libre = N[Libre].g;
            N[A].val = X;
            N[A].g = -1;
            N[A].d = -1;
            return A;
      }
}
```


Représentation pointeur

```
class arbre {
      ...
private:
      struct noeud {
            info val;
            noeud* g;
            noeud* d;
      } *a;
};
arbre:: arbre (info v) {
      a = new noeud;
      a-> g = NULL; a-> d = NULL;
      a-> val = ...;
}
bool arbre:: Vide () {
      return (a== NULL);
}
info arbre:: Valeur () {
      return (a-> val);
}
void arbre:: CreerGauche (info v) {
      noeud* n = new noeud;
      n-> g = NULL;
      n-> d = NULL;
      n-> val = v;
      a-> g = n;
}
```