STRUCTURE DE DONNEES COMPLEXE

CHAPITRE 1 : LES ARBRES

Les arbres sont comme les listes - une structure de données permettant de représenter des collections d'objets. Tout comme les listes, les arbres sont définis de manière récursive, ce qui distingue les listes et les arbres c'est la façon d'organiser ces éléments :

-dans une liste ils sont présents séquentiellement (d'abord la tête, puis le reste de la liste - reste qui est également une liste)

-Dans un arbre, on trouve un élément appelé la racine puis le reste des éléments est divise dans deux (ou plusieurs) branches (qui sont-elles-mêmes des arbres).

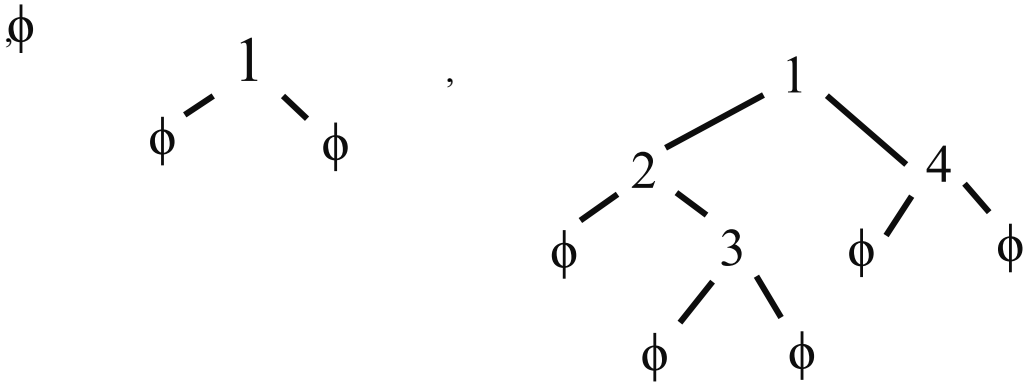
Ce type de structure arborescente a de très nombreuses applications dont deux que nous verrons : le tri (fTD) et la planification (f projet)

1-<u>Type abstrait</u>

Définition : soit e un ensemble (non vide) d'éléments. Un arbre binaire sur E est soit l'arbre vide, note **ϕ,** soit un triplet constitué d'un élément de E et de deux arbres, respectivement appelés fils (ou sous-arbre) gauche et droit.

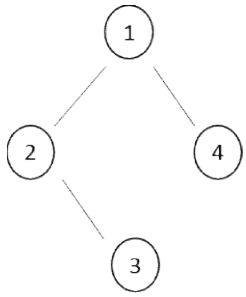
Ensembles d'arbres sur les entiers :

ϕ, (1, ϕ, ϕ), (1, (2, ϕ, (3, ϕ, ϕ)), (4, ϕ, ϕ))
Qu'on présentera sous la forme



Ou plus simplement

ϕ, 1,



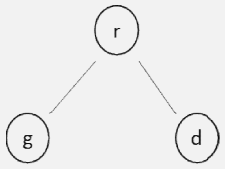
On notera l'ensemble des arbres T. définissons les opérations du type abstrait.

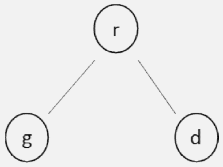
Opération : **is_empty**

Type : T→{T, F}

Description : indique si l'arbre donné en argument est vide

Definition : is_empty (ϕ) = T

is_empty ( **) = F**

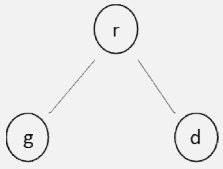
La notation () désigne un arbre non vide, de racine r, de fils gauche g et de fils droit d.

Opération : **cardinal**

Type : T →N

Description : indique le nombre d'éléments dans un arbre

Définition : cardinal(ϕ) = 0

Cardinal () = 1 + cardinal(g) + cardinal(d)

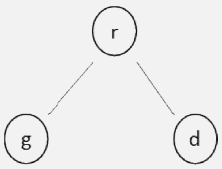
On appelle profondeur d'un élément de l'arbre la distance entre cet élément et la racine. La hauteur de l'arbre est égale à la plus grande profondeur trouvée dans cet arbre.

Opération : **hauteur**

Type : T →N

Description : retourne la hauteur d'un arbre

Définition : hauteur(ϕ) = 0

hauteur (

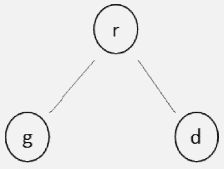
) = 1 + max(hauteur(g), hauteur(d))

Opération : **member**

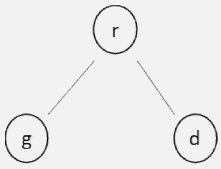
Type : T →E →{T, F}

Description : indique si un élément est présent dans l'arbre

Définition : member (ϕ, e) = F ∨ e appartenant à E

member(

, e) = T si e = r sinon

member(, 

e) = member (g, e) ∪ member(d, e)

(rappel :∪ désigne le « ou » logique)

Operations : equals

Type : T→T →{T, F}

Description : teste si les deux arbres sont égaux

Définition : **equals**(ϕ, ϕ) = T,

equals(

) = (r1 = r2) ∩**equals**(f1,f2) ∩**equals**(g1, g2)

equals (x, y) = F dans tous les autres cas.

De même que pour les listes on peut définir l'égalité (dite structurelle) entre les arbres.

Nous verrons les opérations d'insertion et suppression dans le cadre des arbres de recherche.

2- **implémentation**

De la même manière que pour les listes, implémenter un type abstrait « arbre » consiste d'abord à établir une correspondance entre un objet mathématique et une représentation en mémoire. Celle-ci est déterminée par un type C et un constructeur. Ensuite il s'agit de traduire les opérations de type abstrait en respectant scrupuleusement cette correspondance.

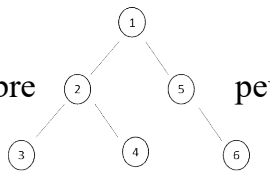
2.1 **Représentation mémoire**

Comme pour les listes, on représente les arbres par les pointeurs, en prenant pour convention que le pointeur NULL représente l'arbre vide. Pour les arbres non vides, le pointeur désignera une structure (qui, encore une fois, est le moyen en C de représenter un produit cartésien) contenant la racine et les deux arbres fils. Cela donne:

```
Struct tree_t {
        Int root;
        Struct tree *left;
        Struct tree *right;
} ;
typedef struct tree_t *tree ;
```

Pour représenter des arbres sur les entiers, le constructeur s'écrit alors

```
        tree cons (int root, tree left, tree right) {
                tree t = (tree) malloc (signeof (struct tree_t));
                t→ root = root;
                t→left = left;
                t→right = right;
                return t;
        }
```

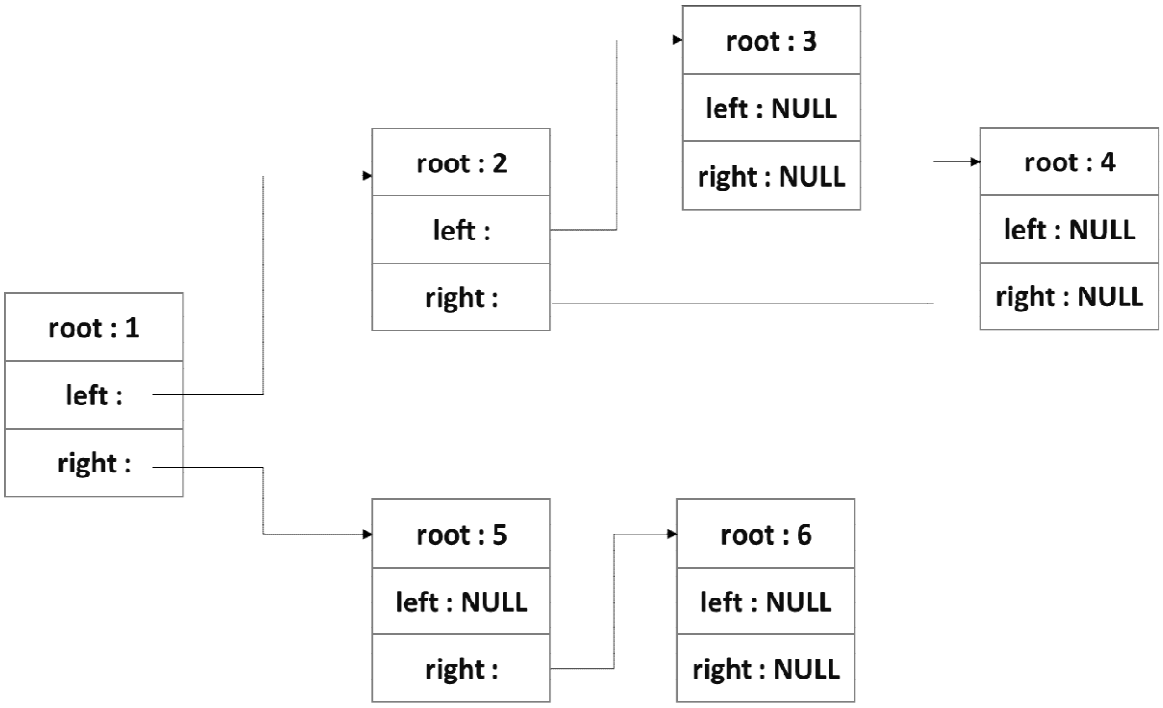
Exemple : l'arbre 

peut être construit avec l'instruction :

Tree_t = cons(1, cons(2,

```
                        cons(3, NULL, NULL),
                        cons(4, NULL, NULL) ,
cons(5,
                        NULL,
                        cons(6, NULL, NULL))) ;
```

L'état de la mémoire après cet appel est :



2.2 **Observateurs**

On applique directement les récurrences vues dans le type abstrait :

```
Int cardinal(tree t){
        if (t==NULL) return 0;
        else return 1+ cardinal(t→left) + cardinal(t→right)
}
```

```
int profondeur(tree t){
        if (t==NULL) return 0 ;
        else return 1+ max(profondeur (t→left), profondeur(t→right)) ;
        }
```

```
Int equals(tree t1, tree t2){
        If (t1 == NULL) return (t2 == NULL);
        Else    if (t2 == NULL) return 0;
        else return (t1 → root == t2→root)
                        && (equals (t1→left, t2→left))
                        && (equals (t1→right, t2→right));
}
```

3. **parcours d'arbres**

Parcourir une collection (liste, arbre, ensemble, …) signifie visiter successivement et dans un certain arbre les éléments qu'il contient pour produire un résultat. Le traitement appliqué à chaque nœud dépend bien sûr du résultat que l'on cherche à obtenir. On prendra pour exemple ici l'affichage du nœud à l'écran.

Dans le cas des listes, il existe pour un traitement donné 2 parcours possibles « naturels » : de gauche à droite ou de droite à gauche. Concrètement, on facilite ces parcours à l'aide d'appels récursifs :

```
void left_traverse (list l) {
        if (! is_empty(l)) {
                printf(l→head);
                left_traverse(l-tail) ;
        }
}
```

```
Void right_traverse (list l) {
        If(!is_empty(l)) {
                right_traverse(l-tail);
                printf(l→head) ;
        }
}
```

La seule différence entre ces deux fonctions porte sur l'ordre d'affichage (le traitement) et l'appel récursif. Voyons leur effet sur un exemple avec le code suivant :

```
list l = cons(1, cons(2, cons(3, NULL))) ;
left_traverse(l);
printf (''\n'');
```

Right_traverse(l) ;

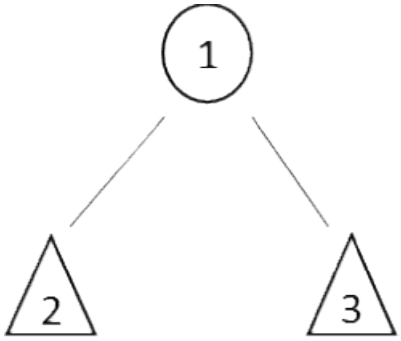
Qui produit à l'écran :

```
123
321
```

On le voit sur cet exemple, left_traverse visite les éléments de la liste de gauche à droite quand right_traverse les visites de droite à gauche.

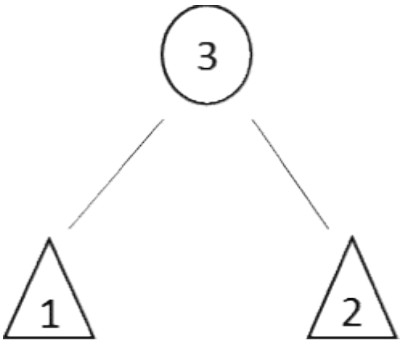
Sur les arbres, on peut définir une variété de parcours, parmi lesquels les parcours dit « en profondeur » :

- Le parcours préfixe



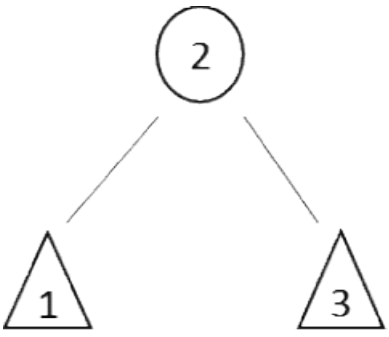
D'abord la racine,
Puis le fils gauche
Et enfin le fils droit

- Le parcours suffixe



Fils gauche, fils droit
Et racine

- Le parcours infixe



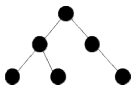
Fils gauche, racine, fils droit

Comme pour le parcours des listes, ces trois variantes correspondent à des ordres différents dans les appels successifs aux traitements et aux appels récursifs :

```
void prefixe (tree t) {
        if (t! =NULL) {
                printf(t→root);
                prefixe(t→left);
                prefixe(t→right) ;
        }
```

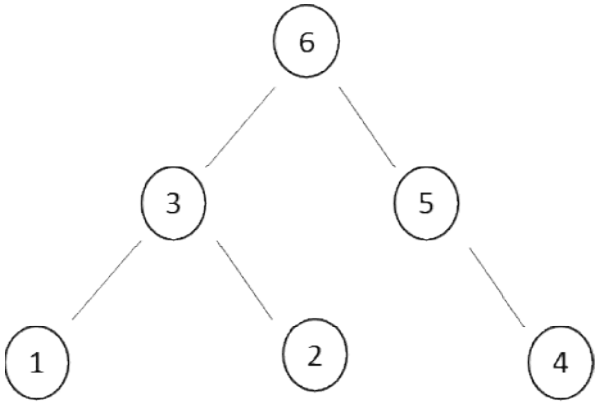
```
void suffixe (tree t) {
        if (t! =NULL) {
                suffixe(t→left) ;
                suffixe(t→right);
                printf(t→root) ;
        }
```

```
void infixe(tree t){
        if( t != NULL){
                infixe(t→left) ;
                printf(t→root) ;
                infixe(t→right) ;
        }
```

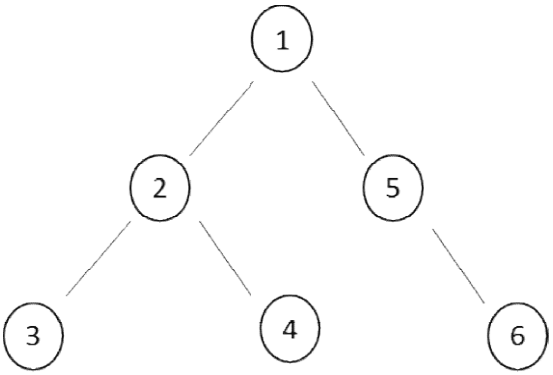
Illustration : soit un arbre dont la structure est : 

L'arbre de parcours de ses éléments est :

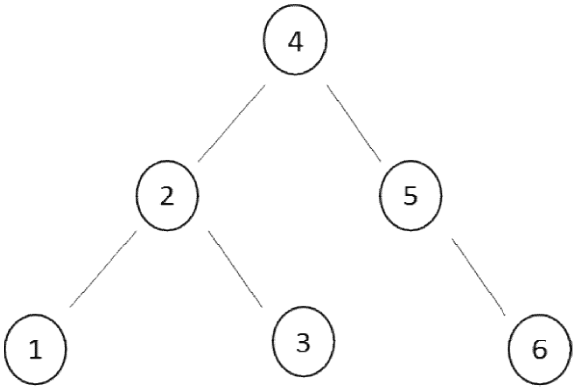
Parcours suffixe :



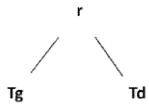
Parcours préfixe :



Parcours infixe :



4- **arbres de recherche**

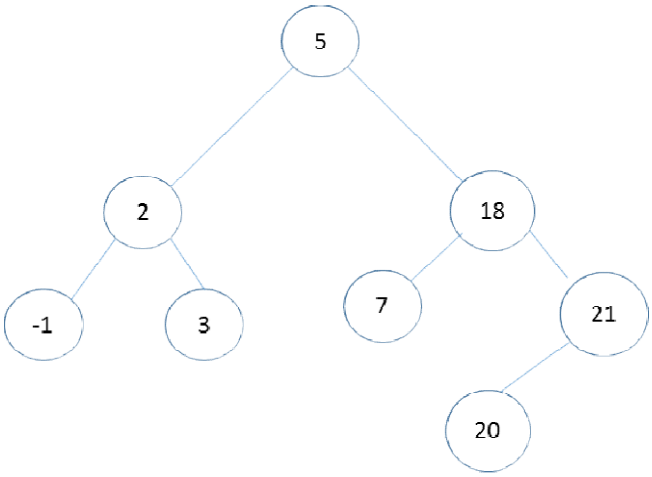


Définition : un arbre binaire de recherche est un arbre binaire dont tout sous-arbre non vide vérifie :

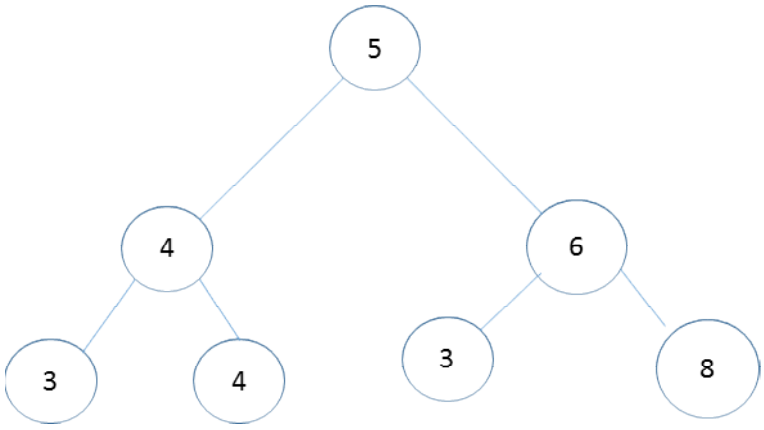
-tout élément de Tg est inférieur à r

-tout élément de Td est supérieur à r.

Exemple :



contre-exemple :



Propriete : un arbre est un arbre de recherche si et seulement si un parcours infixe visite ses éléments dans l'ordre croissant

Demonstration : exercice (par recurence sur la profondeur de l'arbre)

Moralité : construire un arbre de recherche est un moyen de trier une collection d'objets

Pour clore ce chapitre, nous passons en détail sur l'insertion et la suppression dans un arbre de recherche. Dans les deux cas l'arbre doit etre modifie en maintenant la propriéte des arbres de recherche.

Operation : **insertion**

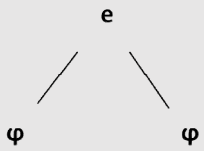
Type : T x E → T

Precondition : insertion (T, e) si T est un arbre de recherche

Postcondition : insertion(T, e) est un arbr de recherche contenant e et non les elements de T.

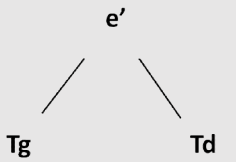
Definition :

Insertion(**ϕ, e) =**

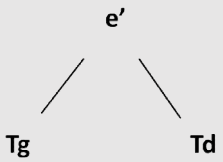


si e < r

Insertion(

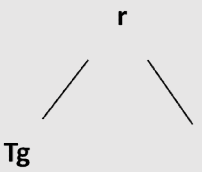


, e) =



si e = e'

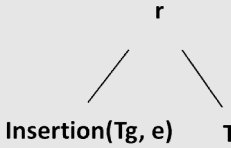
Insertion(



, e) = si

e > r

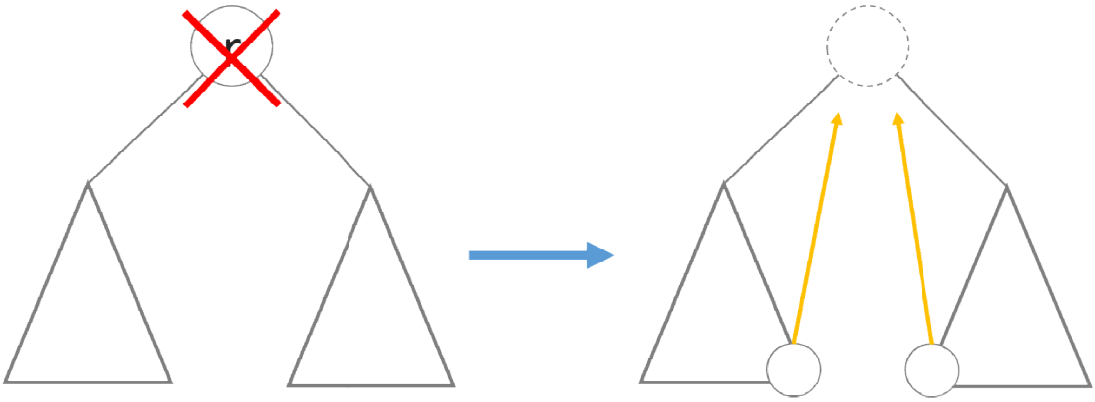
=



sinon

La suppression est un problème un peu plus compliqué : si on supprime une racine il faut (dans la plupart des cas) en trouver une autre pour raccrocher les 2 branches.

Or on ne peut pas choisir n'importe quel élément si on veut conserver un arbre de recherche. Il y a en fait aux plus deux candidats : le maximum de l'arbre gauche ou le minimum de l'arbre droit :



Par conséquent, une fois arrivé au nœud (élément) de l'arbre à supprimer, on le remplacera par le maximum (respectivement le minimum) du fils gauche (respectivement droit) qu'on aura préalablement supprimé dudit fils gauche (respectivement droit)

Décomposons le problème, avec pour commencer la suppression du maximum d'un arbre :

Opération : **suppressionmax**

Type : T→ TxE

Précondition : suppression(max(t)) si t est un arbre de recherche non vide

Définition :

Suppresionmax(r / φ φ) = (ϕ, r)

Suppressionmax(r / g φ) = (g, r)

Suppressionmax(r / g d) = r / g Suppressionmax(d)

```
tree insertion (tree t, int e) {
if(t==null) return cons (e, null, null);
if(t→ root > e) t→ left = insertion(t→ left, e);
else if (t→ root < e) t→ right = insertion(t→ right, e);
return t ;
}
```

Pour suppressionmax, il faut renvoyer deux résultats, ce qui n'est pas directement possible en C.il faut en renvoyer un normalement et l'autre par l'intermédiaire d'un pointeur.

```
tree suppressionmax (tree t, int* max) {
assert (t! =null);
if (t→ right == null) {
*max = t→ root;
tree r = t→ left;
free(t)
return r;
}
else {
t→ right = suppressionmax(t→ right, max)
return t;
}
}
```

On voit que la fonction suppressionmax renvoie deux résultats : l'arbre où l'on a supprimé le maximum, et ledit maximum. Cela n'est pas très commode en C mais on règlera ce problème plus tard. Nous passons maintenant à la suppression proprement dite.

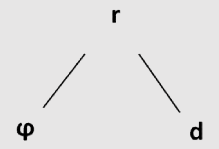
Opération : **suppression**

Type : T x T → E

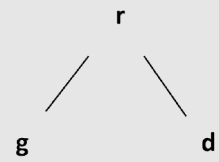
Précondition : la suppression (t, e) si T est un arbre de recherche

Définition :

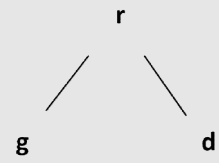
Suppression (ϕ, e) = ϕ

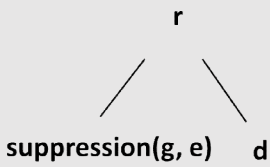
Suppression(

, e) = d si e = r

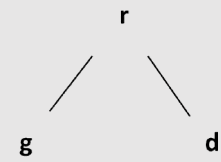
Suppression(

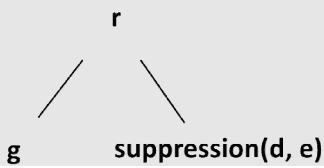
, e) = si e = r où (g', M) = suppressionmax(g)

Suppression (

, e) = 

si e < r

Suppression (

, e) = 

si e > r

Pour finir passons à l'implémentation en langage C la principale difficulté pour adapter les définitions de types abstraits concerne la gestion mémoire : attention aux fuites !

```
tree suppression (tree t, int e) {
        if(t==null) return t;
        else if (t→root ==e) {
if (t→left == null)
tree r = t→right;
                        free(t);
                        return r;
else {
              int m;
              t→left = suppresionmax (t→left, 4m);
              t→root = m;
              return t;
}
       }
if (t→root >e) t→left = suppression (t→left, e);
  else if (t→root <e) t→right = suppression (t→right, e);
Return t ;
```