Chapitre

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 présentés 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 racine puis le reste des éléments est divisé dans deux (ou plusieurs) branches (qui sont elles-même des arbres). → répartis

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

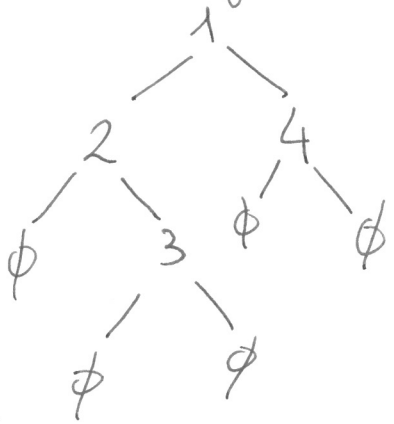
1. Type abstrait

Def : soit E un ensemble (non vide) d'éléments. Un arbre binaire sur E est soit l'arbre vide, noté $\emptyset$, soit un triplet constitué d'un élément de E et de deux arbres, respectivement appelés fils (ou sous-arbre) gauche et droit.

Exemples d'arbres sur les entiers :

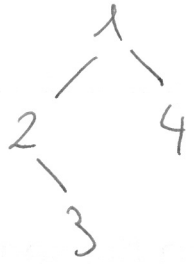
ϕ, $(1, \phi, \phi)$, $(1, (2, \phi, (3, \phi, \phi)), (4, \phi, \phi))$

qu'on représentera sous la forme



ou plus simplement

ϕ, 1,



On notera l'ensemble des arbres Π. Définissons les opérations du type abstrait

opération : is_empty
type : $\Pi \longrightarrow \{T, F\}$
description : indique si l'arbre donné en argument est vide
définition : is_empty $(\phi) = T$

is_empty $\left(\begin{smallmatrix} & r & \\ g & & d \end{smallmatrix} \right) = F$

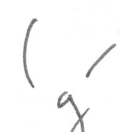
La notation $\begin{smallmatrix} & r & \\ g & & d \end{smallmatrix}$ désigne un arbre non vide, de racine r, de fils gauche g et fils droit d.

opération : cardinal
type : $\mathbb{T} \to \mathbb{N}$
description : indique le nombre d'éléments dans un arbre
définition : cardinal(ϕ) = 0

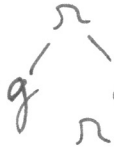
cardinal() = 1 + cardinal(g) + cardinal(d)

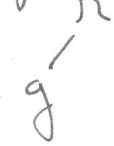
On appelle profondeur d'un élément de l'arbre la distance entre cet élément et la racine. La hauteur de l'arbre est égale à la plus grande profondeur trouvée dans cet arbre.

opération : hauteur
type : $\mathbb{T} \to \mathbb{N}$
description : retourne la hauteur d'un arbre
définition : hauteur(ϕ) = 0

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

opération : member
type : $\mathbb{T} \to \underline{\mathbb{E}} \to \{T, F\}$
description : indique si un élément est présent dans l'arbre.
définition : member(ϕ, e) = F $\quad \forall e \in \underline{\mathbb{E}}$

member( , e) = T si e = r

member( , e) = member(g, e) $\vee$ member(d, e) sinon

(rappel $\vee$ désigne le "ou" logique).

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

opération : equals

type : $\Pi \to \Pi \to \{T, F\}$

description : teste si 2 arbres sont structurellement égaux

définition :

$equals(\phi, \phi) = T$

$equals\left(\begin{array}{ccccc} & r_1 & & r_2 & \\ f_1 & & g_1 & & f_2 & & g_2 \end{array}\right) = (r_1 = r_2) \wedge equals(f_1, f_2) \wedge equals(g_1, g_2)$

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

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

2. Implémentation

De la même manière que pour les listes, implémenter le 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 du type abstrait, en respectant scrupuleusement cette correspondance.

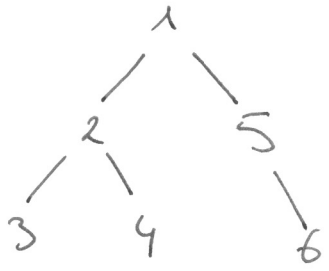
2.1 Représentation mémoire

Comme pour les listes, on représentera les arbres par des pointeurs, en prenant pour convention que le pointeur NULL représente l'arbre vide. Pour les arbres non vide, le pointeur désignera une structure (qui, encore une fois, est le moyen en C de représenter un produit cartésien) contenant la racine et les 2 arbres fils. Cela donne :

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

pour représenter des arbres sur des entiers. Le constructeur s'écrit alors

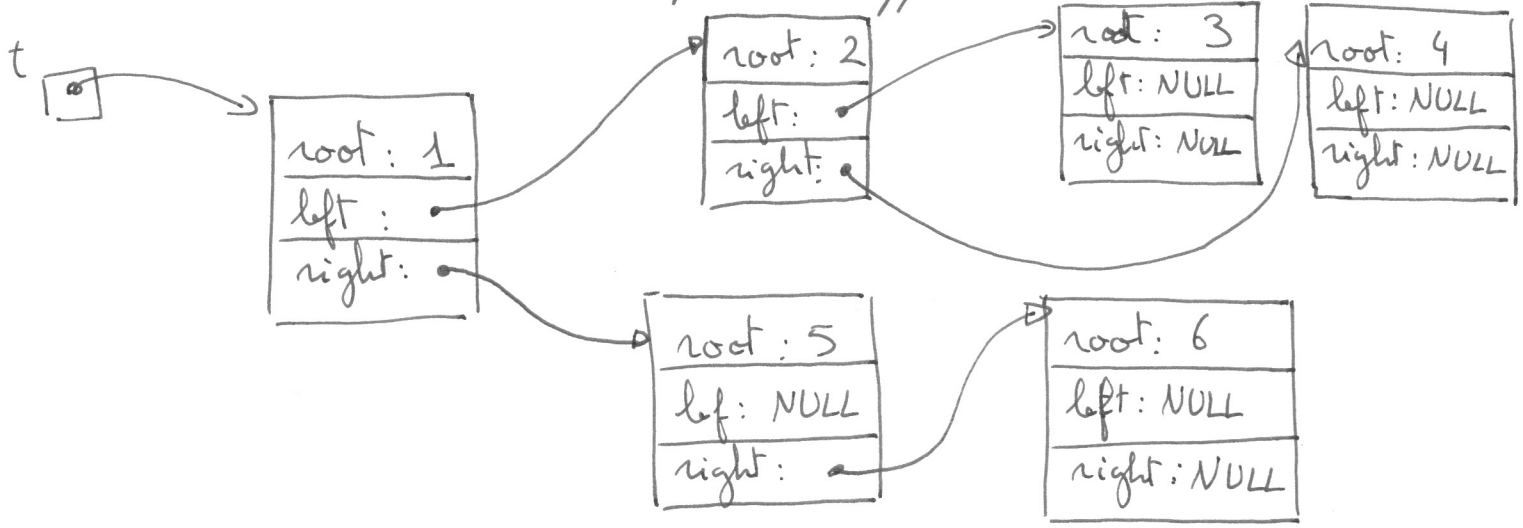
```
tree cons(int root, tree left, tree right) {
    tree t = (tree) malloc(sizeof(struct tree_t));
    t->root = root;
    t->left = left;
    t->right = right;
    return t;
}
```

Exemple : l'arbre 

peut être construit par 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 ordre les éléments qu'elle contient pour produire un résultat. Le traitement appliqué à chaque noeud dépend bien sûr du résultat que l'on cherche à obtenir. On prendra pour exemple ici l'affichage du noeud 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 implémente facilement ces parcours à l'aide d'appels récursifs :

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

```
void right_traverse(list l) {
    if(!is_empty(l)) {
        right_traverse(l->tail);
        print(l->head);
    }
}
```

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

```
list l = cons(1, cons(2, cons(3, NULL))); // représente 1::2::3::∅
left_traverse(l); printf("\n");
right_traverse(l);
```

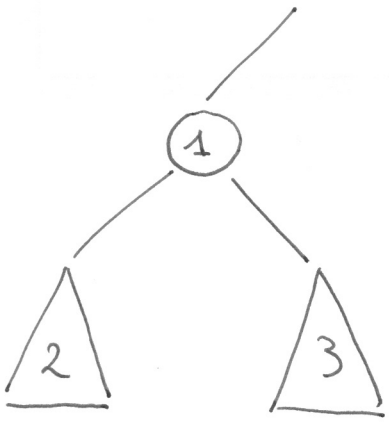
qui produit à l'écran :

```
1 2 3
3 2 1
```

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

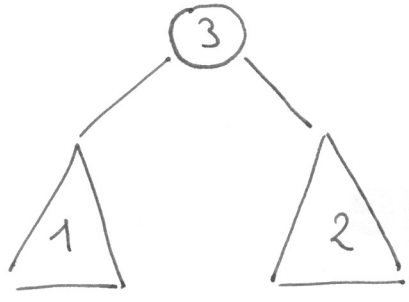
Sur les arbres, on peut définir une variété de parcours, parmi lesquels les parcours dits "en profondeur" :

- le parcours préfixe



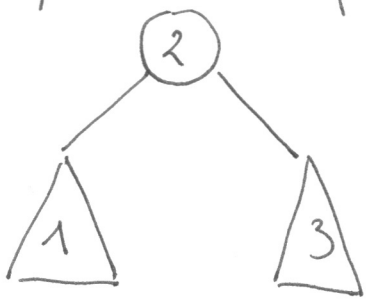
d'abord la racine,
puis le fils gauche
et enfin le fils droit

- le parcours suffixe



fils gauche, fils droit, et racine

- le parcours infixe



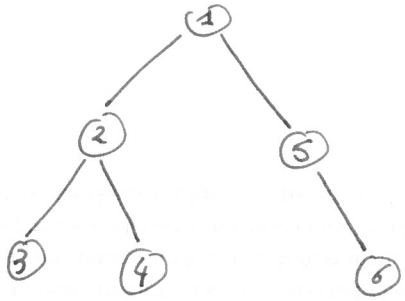
fils gauche, racine, fils droit

Comme pour le parcours de liste, ces 3 variantes correspondant à des ordres différents dans les appels successifs aux traitements et aux appels récursifs:

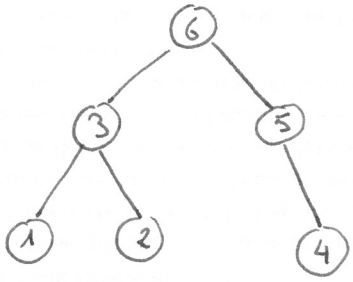
```
void prefixe (tree t) {
    if (t != NULL) {
        print (t->root);
        prefixe (t->left);
        prefixe (t->right);
    }
}

void suffixe (tree t) {
    if (t != NULL) {
        suffixe (t->left);
        suffixe (t->right);
        print (t->root);
    }
}

void infixe (tree t) {
    if (t != NULL) {
        infixe (t->left);
        print (t->root);
        infixe (t->right);
    }
}
```

Illustration: soit un arbre dont la structure est  l'ordre de parcours de ses éléments est:

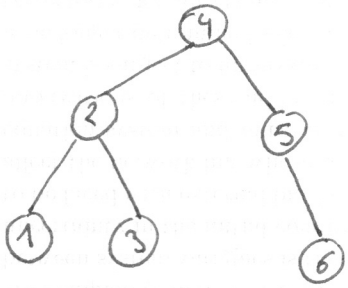
parcours préfixe:



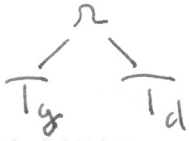
parcours suffixe:



parcours infixe:

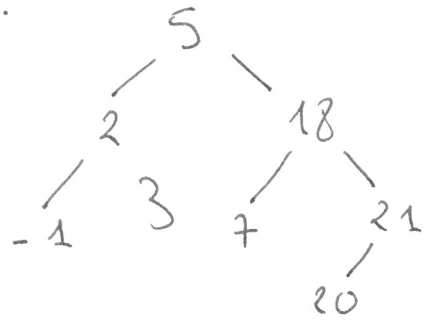


4. Arbres de recherche

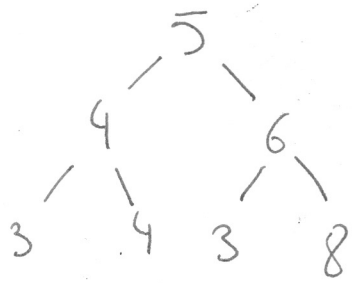
Def: un arbre binaire de recherche est un arbre binaire dont tout sous-arbre non vide  vérifie:

- tout élément de T_g est inférieur à r
- tout élément de T_d est supérieur ou égal à r.

ex:



contre ex:



Prop: un arbre est un arbre de recherche ssi un parcours infixe visite ses éléments dans l'ordre croissant.

démonstration: exercice (par récurrence 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/suppression dans un arbre de recherche. Dans les 2 cas, l'arbre ~~modifié~~ doit être modifié en maintenant la propriété des arbres de recherche!

opération : insertion

type : $\mathbb{T} \times \mathbb{E} \longrightarrow \mathbb{T}$

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

post condition : insertion (T, e) est un arbre de recherche contenant e et tous les éléments de T

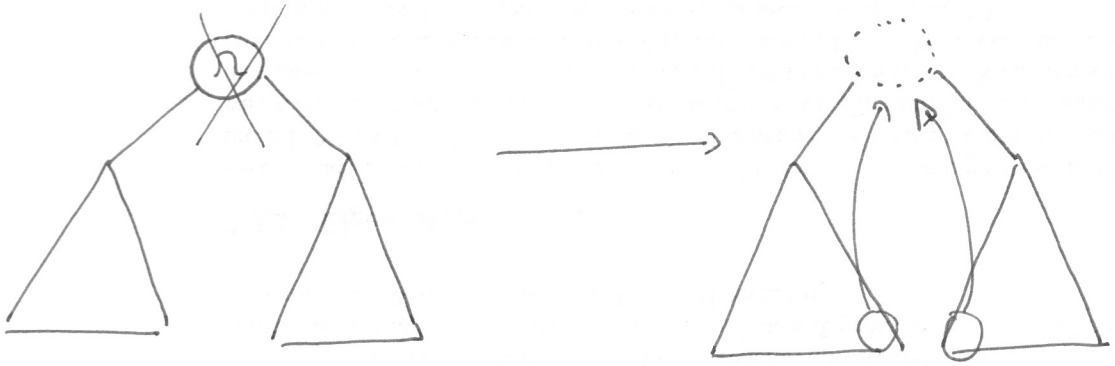
définition :

$$\text{insertion}(\phi, e) = \begin{array}{c} e \\ \phi \quad \phi \end{array}$$

$$\text{insertion}\left(\begin{array}{c} r \\ T_g \quad T_d \end{array}, e\right) = \begin{cases} \begin{array}{c} r \\ \text{insertion}(T_g, e) \quad T_d \end{array} & \text{si } e < r \\ \begin{array}{c} r \\ T_g \quad \text{insertion}(T_d, e) \end{array} & \text{si } e > r \end{cases}$$

$$\text{insertion}\left(\begin{array}{c} e' \\ T_g \quad T_d \end{array}, e\right) = \begin{array}{c} e' \\ T_g \quad T_d \end{array} \text{ si } e = e'$$

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 ~~pa~~ branches. Or on ne peut pas choisir n'importe quel élément, si l'on veut conserver un arbre de recherche. Il y a en fait au plus 2 candidats : le maximum de l'arbre gauche ou le minimum de l'arbre droit :



Par conséquent, une fois arrivé au noeud (élément) de l'arbre à supprimer, on le remplacera par le max (resp. le min) du fils gauche (resp. droit) qu'on aura préalablement supprimé dudit fils gauche (resp. droit).

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

opération : suppression_max

type : ~~$\mathbb{T} \to \mathbb{T} \times \mathbb{E}$~~ $\mathbb{T} \to \mathbb{T} \times \mathbb{E}$

précondition : suppressionmax(t) si t est un arbre de recherche non vide

définition :

$$\begin{cases} \text{suppressionmax}\left(\phi \overset{r}{\frown} \phi\right) = (\phi, r) \\ \text{suppressionmax}\left(g \overset{r}{\frown} \phi\right) = (g, r) \\ \text{suppressionmax}\left(g \overset{r}{\frown} d\right) = g \overset{r}{\frown} \text{suppressionmax}(d) \end{cases}$$

```
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 2 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;
    }
}
```

Nous arrivons à la fonction de suppression.

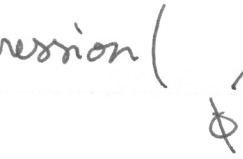
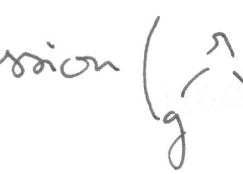
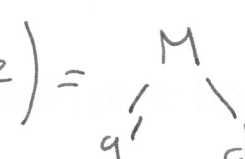
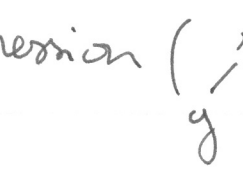
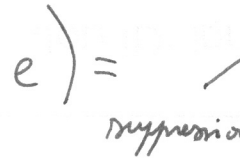
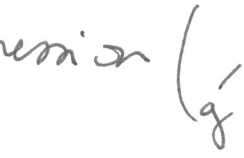
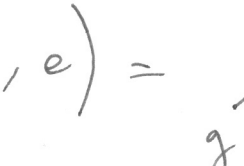
On voit que la fonction suppressionmax renvoie 2 résultats : l'arbre où l'on a supprimé le maximum, et ledit maximum. Cela n'est pas très commode en C, mais on réglera ce problème plus tard. Nous passons maintenant à la suppression proprement dite.

opération : suppression

type : $\Pi \times \mathbb{E} \to \Pi$

précondition : 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 du type abstrait 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 = suppressionmax (t -> left, &M);
            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;
}
```

adresse de M