I Rappels sur les arbres binaires de recherche

On utilisera les types suivants d'arbres binaires enracinés :

```
type 'a arb = V | N of 'a arb * 'a * 'a arb
```

```
typedef struct arb {
    struct arb* g;
    int r;
    struct arb* d;
} arb;
```

En C, un arbre vide est représenté par un pointeur `NULL`.

Théorème : Induction sur les arbres binaires

Soit $P(a)$ une propriété sur un arbre binaire a. Supposons :
- $P(\texttt{V})$ vraie.
- Pour tout arbre binaire g, d et tout élément r, si $P(g)$ et $P(d)$ sont vraies, alors $P(\texttt{N}(g,r,d))$ est vraie.

Alors $P(a)$ est vraie pour tout arbre binaire a.

Remarque : On peut aussi raisonner par récurrence sur la hauteur de l'arbre ou le nombre de sommets.

Définition : Hauteur

La hauteur $h(a)$ d'un arbre binaire a est la longueur maximum d'un chemin (en nombre d'arêtes) de la racine à une feuille.

```
let rec h a = match a with
    | V -> 0
    | N(g, _, d) -> 1 + max (h g) (h d)
```

```
int h(arb* a) {
    if(!a) return 0;
    return 1 + max(h(a->g), h(a->d));
}
```

Si a est un arbre binaire, on note $f(a)$ son nombre de feuilles, $n(a)$ son nombre de nœuds, $n_i(a)$ son nombre de nœuds internes (non feuilles) et $h(a)$ sa hauteur.

Exercice 1.
Montrer que si a est un arbre binaire strict (chaque nœud a 0 ou 2 fils) alors $f(a) = n_i(a) + 1$.

Théorème : Formules sur les arbres binaires

Si a est un arbre binaire alors :

$$f(a) \leq 2^{h(a)}$$
$$h(a) + 1 \leq n(a) \leq 2^{h(a)+1} - 1$$
$$\log_2(n(a)+1) - 1 \leq h(a) \leq n(a) - 1$$

On dit que a est équilibré si $h(a) = \mathrm{O}(\log(n(a)))$.

Exercice 2.
Si $G = (S, A)$ est un graphe, on note $d(G) = \max_{u,v \in S} d(u,v)$ la distance maximale entre deux sommets de G, où $d(u,v)$ est la longueur du plus court chemin entre u et v.

1. Justifier que pour un arbre a, $d(a)$ est la longueur maximum d'un chemin élémentaire entre deux sommets de a.
2. Écrire une fonction `d : 'a arb -> int` qui calcule le diamètre d'un arbre binaire en temps linéaire.
3. Généraliser avec un arbre enraciné quelconque défini par `type 'a arb = N of 'a arb list`.

4. Avec quelle complexité peut-on trouver le diamètre d'un graphe quelconque ?

Si a est un arbre binaire, on note $r(a), g(a), d(a)$ respectivement la racine, le sous-arbre gauche et le sous-arbre droit de a.

Définition : Arbre binaire de recherche

Un arbre binaire de recherche (ABR) est un arbre binaire a tel que pour chaque nœud b de a, toutes les étiquettes de $g(b)$ sont inférieures ou égales à $r(b)$ et toutes les étiquettes de $d(b)$ sont supérieures ou égales à $r(b)$.

Remarque : La propriété d'ABR est héréditaire : un sous-arbre d'un ABR est un ABR. C'est utile pour écrire une fonction récursive et raisonner par récurrence sur les ABR.

Exercice 3.
Prouver ou réfuter l'affirmation suivante : un arbre binaire a est un ABR si et seulement si $f(a)$, $g(a)$ sont des ABR et $r(g(a)) \leq r(a) \leq r(d(a))$.

Théorème

Soit a un arbre binaire. Alors a est un ABR si et seulement si son parcours infixe est croissant.

Remarque : Cela donne un algorithme de tri et permet de tester si un arbre binaire est un ABR.
Preuve :

```
let rec add e a = match a with
    | V -> N(V, e, V)
    | N(g, r, d) ->
        if e < r then N(add e g, r, d)
        else N(g, r, add e d)

let rec has e a = match a with
    | V -> false
    | N(g, r, d) ->
        if e = r then true
        else if e < r then has e g
        else has e d

let rec del_min a = match a with
    | V -> failwith "Empty"
    | N(V, r, d) -> r, d
    | N(g, r, d) ->
        let m, g' = del_min g in
        m, N(g', r, d)

let rec del e a = match a with
    | V -> V
    | N(g, r, d) ->
        if e = r then
            match g, d with
            | V, _ -> d
            | _, V -> g
            | _, _ ->
                let m, d' = del_min d in
                N(g, m, d')
        else if e < r then N(del e g, r, d)
        else N(g, r, del e d)
```

```
arb* add(int e, arb* a) {
    if(!a) return N(NULL, e, NULL);
    if(e < a->r)
        a->g = add(e, a->g);
    else
        a->d = add(e, a->d);
    return a;
}

bool has(int e, arb* a) {
    if(!a) return false;
    if(e == a->r) return true;
    if(e < a->r) return has(e, a->g);
    return has(e, a->d);
}

int del_min(arb** a) {
    if(!(*a)->g) {
        int r = (*a)->r;
        *a = (*a)->d;
        return r;
    }
    return del_min(&(*a)->g);
}

arb* del(int e, arb* a) {
    if(!a) return NULL;
    if(e == a->r) {
        if(!a->d) return a->g;
        if(!a->g) return a->d;
        int m = del_min(&a->d);
        a->r = m;
        return a;
    }
    if(e < a->r)
        a->g = del(e, a->g);
    else
        a->d = del(e, a->d);
    return a;
}
```

I.1 Arbre rouge-noir

Définition : Arbre rouge-noir

Un arbre rouge-noir (ARN) a est un ABR dont les noeuds sont coloriés en rouge ou noir et vérifiant :

- La racine est noire (non obligatoire, mais rend le code plus simple).
- Si un sommet est rouge, ses éventuels fils sont noirs.
- Le nombre de noeuds noirs le long de n'importe quel chemin de la racine à un emplacement vide (`V`) est le même, que l'on appelle hauteur noire et note $h_b(a)$.

Théorème

Soit a un ARN avec $n(a)$ noeuds et de hauteur $h(a)$. Alors :

- $h(a) \leq 2h_b(a)$.
- $n(a) \geq 2^{h_b(a)} - 1$.
- $h \leq 2\log_2(n+1)$ (un arbre rouge-noir est donc équilibré).

Preuve :

II *Treap*

On peut montrer qu'un arbre binaire de recherche (ABR) construit en ajoutant un à un n entiers choisis « uniformément au hasard » a une hauteur moyenne $O(\log(n))$.
Si les éléments à rajouter ne sont pas générés aléatoirement, mais sont tous connus à l'avance, on peut commencer par les mélanger aléatoirement puis les rajouter dans cet ordre aléatoire pour obtenir à nouveau une hauteur moyenne $O(\log(n))$. Pour cela, on considère l'algorithme suivant (mélange de Knuth), où `Random.int (i+1)` renvoie un entier uniformément au hasard entre `0` et `i` :

```
let shuffle a =
    for i = 1 to Array.length a - 1 do
        swap a i (Random.int (i + 1))
    done
```

1. Écrire la fonction `swap : 'a array -> int -> int` utilisée par `shuffle`, telle que `swap t i j` échange `t.(i)` et `t.(j)`.

Solution :

```
let swap t i j =
    let tmp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- tmp
```

2. Montrer que `shuffle t` applique une permutation choisie uniformément au hasard sur le tableau `t`.

Solution : `shuffle t` applique un certain nombre de transpositions sur `t`, donc il effectue bien une permutation sur `t`.
Soit σ une permutation. On sait que σ est produit de transpositions donc on peut l'écrire $\sigma = (a_1 b_1)(a_2 b_2)...(a_p b_p)$ avec $a_k < b_k$ pour tout k et $b_1 < b_2 < ... < b_p$. Alors il existe exactement une possibilité pour que `shuffle t` applique σ sur `t` : que `Random.int (i+1)` renvoie a_1 pour `i` $= b_1$ (probabilité $\frac{1}{b_1}$), a_2 pour `i` $= b_2$, ... a_p si `i` $= b_p$ (probabilité

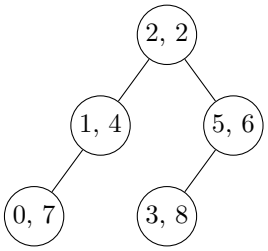
$\frac{1}{b_p}$) et que `Random.int (i+1)` renvoie `i` (probabilité $\frac{1}{i}$) dans tous les autres cas.
La probabilité d'obtenir la permutation σ sur `t` est donc le produit de ces probabilités (ce sont des évènements indépendants), c'est à dire $\frac{1}{n!}$.

Lorsque la totalité des éléments à rajouter n'est pas connue à l'avance, on peut utiliser une structure appelée **arbretas** qui est un arbre binaire (défini par `type 'a arb = V | N of 'a * 'a arb * 'a arb`) dont les noeuds sont étiquetés par des couples (élément, priorité), où la priorité est un nombre entier choisi uniformément au hasard au moment de l'ajout de l'élément. De plus :

- les éléments doivent vérifier la propriété d'ABR.
- la priorité d'un sommet doit être inférieure à la priorité de ses éventuels fils (propriété de tas min sur les priorités).

3. Dessiner un arbretas dont les couples (élément, priorité) sont : (1, 4), (5, 6), (3, 8), (2, 2), (0, 7).

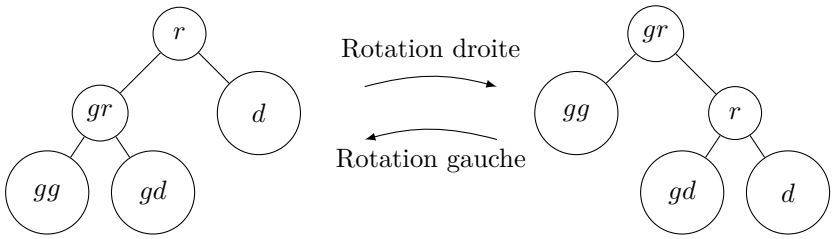
Solution :



4. Étant donnés des éléments et priorités tous distincts, montrer qu'il existe un unique arbretas les contenants.

Solution : Montrons P_n : « il existe un unique arbretas contenants n couples (élément, priorité) tous distincts » par récurrence sur n.
P_0 est vraie : l'arbre vide convient et c'est le seul.
Soit $n \in \mathbb{N}^*$ et supposons P_k vraie, $\forall k \leq n$. Considérons un ensemble C de $n+1$ couples (élément, priorité) tous distincts. Soit (e, p) le couple (élément, priorité) ayant la plus petite priorité, G les couples dont les éléments sont inférieurs à e et D les couples dont les éléments sont supérieurs à e. Les arbretas contenant les couples de C sont ceux s'écrivant $\mathrm{N}((e, p), g, d)$ avec g et d arbretas contenant G et D. D'après l'hypothèse de récurrence, il existe un unique tel g et un unique tel d. Donc il existe aussi un unique arbretas contenant les couples de C.

Nous allons utiliser des opérations de rotation sur un arbretas `N(r, N(gr, gg, gd), d)` :



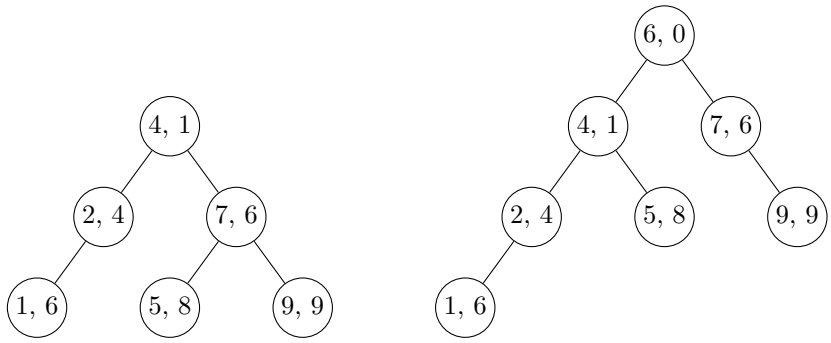
5. Écrire une fonction `rotd` effectuant une rotation droite sur un arbre `N(r, N(gr, gg, gd), d)`.

Solution : `let rotd = function N(r, N(gr, gg, gd), d) -> N(gr, gg, N(r, gd, d));;`

On supposera définie dans la suite une fonction `rotg` pour effectuer « l'inverse » d'une rotation droite. On remarquera que, si `a` est un ABR, `rotg a` et `rotd a` sont aussi des ABR.
Pour ajouter un sommet s dans un arbretas (en conservant la structure d'arbretas), on l'ajoute comme dans un ABR classique (en ignorant les priorités) puis, si sa priorité est inférieure à celle de son père, on applique une rotation sur son père pour faire remonter s et on continue jusqu'à rétablir la structure d'arbretas.

6. Dessiner l'arbretas obtenu en rajoutant (6, 0) à l'arbretas suivant :

Solution : On obtient l'arbretas de droite.



7. Écrire une fonction utilitaire `prio` renvoyant la priorité de la racine d'un arbre (on renverra `max_int`, c'est à dire le plus grand entier représentable en base 2 sur le processeur, si cet arbre est vide).

Solution :

```
let prio = function
    | V -> max_int
    | N(r, g, d) -> snd r
```

8. Écrire une fonction `add a e` ajoutant `e` (qui est un couple (élément, priorité)) à un arbretas `a`, en conservant la structure d'arbretas.

Solution : `add` ajoute l'élément puis applique éventuellement une rotation sur le nouvel arbre.

```
let rec add a e = match a with
    | V -> N(e, V, V)
    | N(r, g, d) when e < r -> let g' = add g e in
                               if snd r < prio g' then N(r, g', d)
                               else rotd (N(r, g', d))
    | N(r, g, d) -> let d' = add d e in
                    if snd r < priod d' then N(r, g, d')
                    else rotg (N(r, g, d'))
```

Pour supprimer un élément d'un arbretas, on commence par le chercher comme dans un ABR classique (en ignorant les priorités) puis on le fait descendre avec des rotations jusqu'à ce qu'il devienne une feuille qu'on peut alors supprimer librement.

9. Écrire une fonction `del` supprimant un élément dans un arbretas, en conservant la structure d'arbretas.

Solution : Il faut effectuer la rotation sur le fils de priorité minimum pour conserver la structure. On utilise le fait que `prio V` renvoie `max_int` pour s'assurer qu'une rotation est appliquée sur un arbre convenable (par ex. `g` doit être non vide pour appliquer `rotd N(r, g, d)`).

```
let rec del a e = match a with
    | N(r, g, d) when e < fst r -> N(r, del g e, d)
    | N(r, g, d) when fst r < e -> N(r, g, del d e)
    | N(r, V, V) -> V (* dans les deux cas suivants, e = r *)
    | N(r, g, d) -> del ((if prio g < prio d then rotd else rotg) a) e
```