Informatique *Cours*
Structures de données hiérarchiques Ipesup

Objectifs

À l'issue de cette leçon, l'étudiant doit être capable de :
— q

Structures de données hiérarchiques

Comme on a vu précédemment, la table de hachage permet d'implémenter efficacement des tableaux associatifs, mais elle présente certaines limites. Elle n'est notamment pas adaptée aux structures immuables, ni aux situations où les données sont totalement ordonnées et où l'on souhaite accéder au n-ième élément ou à un intervalle de valeurs.

Nous introduisons donc dans cette section des structures arborescentes, conçues pour répondre à ces besoins.

1 Arbres binaires

Parmi les structures arborescentes, les arbres binaires occupent une place centrale, ce qui motive leur étude en premier.

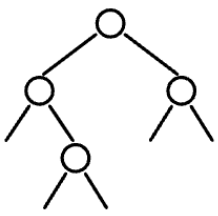
Definition - Arbre binaire

Un *arbre binaire* est un ensemble de nœuds organisés hiérarchiquement, défini inductivement comme suit : un arbre binaire est
— soit l'arbre vide, noté E, sans nœud ;
— soit un nœud, appelé *racine*, relié à deux arbres binaires ℓ et r, appelés respectivement sous-arbre gauche et sous-arbre droit. Un tel arbre, dont la racine porte l'étiquette x, se note $\mathsf{N}(\ell, x, r)$.

Le nombre de nœuds d'un arbre binaire t, noté $n(t)$, est défini récursivement par

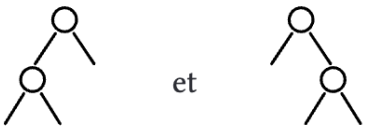
$$n(\mathsf{E}) = 0, \qquad n(\mathsf{N}(\ell, x, r)) = 1 + n(\ell) + n(r).$$

Voici un exemple d'arbre binaire à quatre nœuds.



La racine est en haut et chaque nœud est relié à ses deux sous-arbres. Ici, le sous-arbre gauche contient deux nœuds, le sous-arbre droit un seul, et l'on compte cinq sous-arbres vides. Si un nœud a possède un sous-arbre non vide de racine b, alors a est le *père* de b et b le *fils* de a. Un nœud dont les deux sous-arbres sont vides est une *feuille* ; dans cet exemple, il y en a deux. Tout nœud non feuille est un *nœud interne*.

Il existe deux arbres binaires distincts à deux nœuds : ce sont des arbres *positionnels*.



Hauteur

La hauteur d'un arbre binaire t, notée $h(t)$, est définie récursivement par

$$h(\mathsf{E}) = -1, \qquad h(\mathsf{N}(\ell, x, r)) = 1 + \max\Big(h(\ell), h(r)\Big).$$

Certains auteurs posent la hauteur de l'arbre vide à 0. Les deux conventions sont possibles et influencent peu les résultats, car les hauteurs sont surtout comparées entre elles ou étudiées asymptotiquement.

La *profondeur* d'un nœud est sa distance à la racine. La hauteur d'un arbre est donc la profondeur maximale de ses nœuds, c'est-à-dire la plus grande distance entre la racine et un nœud.

Propriété

Soit t un arbre binaire, n son nombre de nœuds et h sa hauteur. On a les propriétés suivantes :

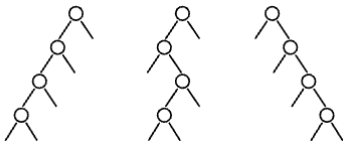
- $h + 1 \leq n \leq 2^{h+1} - 1$;
- le nombre de sous-arbres vides de t est $n + 1$.

Démonstration
L'inégalité $h + 1 \leq n$ est immédiate par définition de la hauteur.

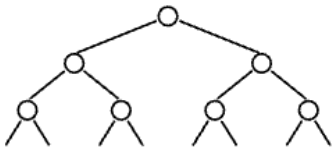
- Prouvons $n \leq 2^{h+1} - 1$ par induction structurelle (ou récurrence forte sur n).
  - Si $n = 0$, l'arbre est vide et $h = -1$.
  - Si $n > 0$, l'arbre a une racine et deux sous-arbres ℓ et r, avec $n = 1 + n(\ell) + n(r)$. L'un des deux sous-arbres a hauteur $h - 1$; supposons $h(\ell) = h - 1$ et $h(r) \leq h - 1$. Par hypothèse de récurrence, $n(\ell) \leq 2^h - 1$ et $n(r) \leq 2^h - 1$, donc
$$n \leq 1 + (2^h - 1) + (2^h - 1) = 2^{h+1} - 1.$$
- Le nombre de sous-arbres vides se montre aussi par induction structurelle.
  - Si $n = 0$, il y en a un : l'arbre lui-même.
  - Si $n > 0$, ℓ contient $n(\ell) + 1$ sous-arbres vides et r en contient $n(r) + 1$, d'où au total
$$n(\ell) + 1 + n(r) + 1 = n + 1.$$

On remarque que ces propriétés valent aussi pour l'arbre vide, avec $n = 0$ et $h = -1$, et que les deux bornes sur n sont atteignables.

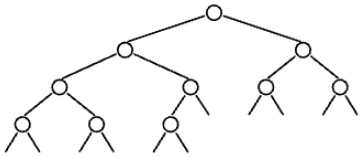
L'égalité $h+1=n$ correspond à des arbres totalement linéaires, avec un seul nœud à chaque profondeur ; lorsque le sous-arbre non vide est toujours du même côté, on parle de *peigne*, structurellement équivalent à une liste chaînée.



À l'inverse, l'égalité $n = 2^{h+1} - 1$ est réalisée par un arbre binaire *parfait*, dont tous les niveaux sont remplis : l'arbre ici est parfait de hauteur 2 et a taille $2^3 - 1 = 7$.



Enfin, si tous les niveaux sont remplis sauf le dernier, rempli de gauche à droite, on obtient un arbre binaire *complet* ; un arbre complet à 10 nœuds a la forme suivante :



1.1 Représentation en machine

On décrit ici la représentation en machine des arbres binaires, d'abord en OCaml puis en C.

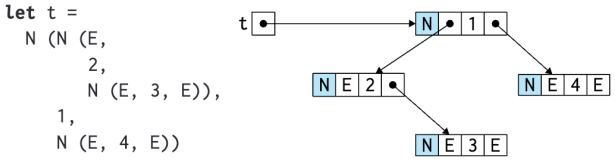
OCaml. Le programme suivante contient la définition d'un type `'a bintree` pour des arbres binaires polymorphes. Le constructeur `E` représente l'arbre vide et le constructeur `N` représente un nœud, contenant une étiquette de type `'a` et deux sous-arbres.

Le programme contient également deux exemples de fonctions sur ce type : une fonction `size` qui calcule le nombre de nœuds et une fonction `perfect` qui construit un arbre binaire parfait où chaque nœud est étiqueté par la hauteur du sous-arbre dont il est la racine.

Arbres binaires en OCaml

```
type 'a bintree =
  | E
  | N of 'a bintree * 'a * 'a bintree

let rec size (t : 'a bintree) : int =
  match t with
  | E -> 0
  | N (l, _, r) -> 1 + size l + size r

```

```ocaml
let rec perfect (h : int) : int bintree =
  if h = -1 then
    E
  else
    N (perfect (h-1), h, perfect (h-1))
```



La figure montre la construction d'un arbre binaire à quatre nœuds, étiquetés 1, 2, 3 et 4, stocké dans une variable `t`. Une valeur de type `bintree` de la forme `N(...)` correspond à un pointeur vers un bloc mémoire identifié par le constructeur `N` et contenant ses trois champs. À l'inverse, une valeur `E` est stockée directement dans le champ, en pratique sous la forme d'un entier.

Il faut souligner que les arbres binaires en OCaml sont *immuables*, propriété essentielle pour leur utilisation comme structures de données.

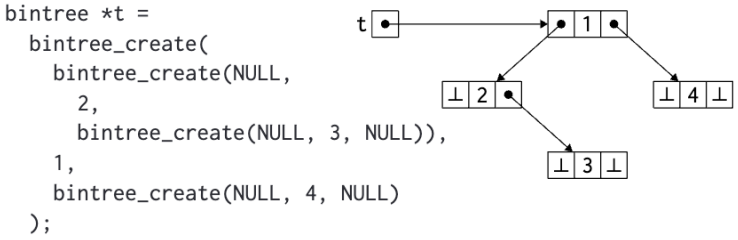
C. Le programme suivante contient la définition d'un type `bintree` pour les arbres binaires. Un arbre binaire est un pointeur vers une structure `Node` contenant trois champs, `left`, `value` et `right`. L'arbre vide est représenté par le pointeur `NULL`.

Ici, les nœuds sont étiquetés par des valeurs entières mais il serait tout aussi simple de les étiqueter avec une valeur d'un autre type. La fonction `bintree_create` construit un nouveau nœud, dont les trois champs sont initialisés avec trois valeurs passées en arguments.

Arbres binaires en C

```c
typedef struct Node {
  struct Node *left;
  int value;
  struct Node *right;
} bintree;

bintree *bintree_create(bintree *l, int v, bintree *r) {
  bintree *t = malloc(sizeof(struct Node));
  t->left = l;
  t->value = v;
  t->right = r;
  return t;
}

int bintree_size(bintree *t) {
  if (t == NULL) return 0;
  return 1 + bintree_size(t->left) + bintree_size(t->right);
}

```

```c
bintree *bintree_perfect(int h) {
  if (h == -1) return NULL;
  return bintree_create(bintree_perfect(h-1),
                        h,
                        bintree_perfect(h-1));
}
```



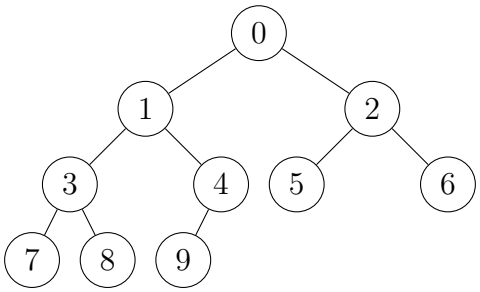
La figure illustre la construction d'un arbre binaire contenant quatre nœuds (étiquetés avec les entiers 1, 2, 3, 4) et stocké dans une variable `t`. Une valeur du type `bintree*` est un pointeur, soit `NULL` (noté $\perp$), soit vers un bloc mémoire contenant les valeurs des trois champs de la structure `Node`.

Le programme contient également une fonction `bintree_size` qui calcule le nombre de nœuds d'un arbre et une fonction `bintree_perfect` qui construit un arbre binaire parfait.

À la différence des arbres binaires en OCaml, les arbres binaires en C sont *mutables*.

1.2 Représentation d'un arbre complet dans un tableau

Dans le cas particulier d'un arbre binaire *complet*, une représentation par tableau est possible et efficace. Il suffit de numéroter les nœuds de haut en bas et de gauche à droite, à partir de zéro, puis de placer l'étiquette de chaque nœud dans la case correspondante du tableau. Voici un exemple avec $n = 10$ nœuds :



Cette représentation est très compacte, puisqu'elle ne stocke que les étiquettes, tout en permettant de naviguer dans l'arbre. Si un nœud est numéroté i, son fils gauche (resp. droit) est $2i + 1$ (resp. $2i + 2$), à condition que $2i + 1 < n$ (resp. $2i + 2 < n$).
Réciproquement, le père de i est $\lfloor (i - 1)/2 \rfloor$ pour $i > 0$.

1.3 Parcours d'un arbre binaire

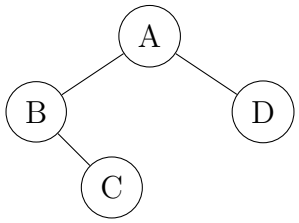
Un *parcours* d'un arbre binaire est un algorithme qui visite chaque nœud exactement une fois et y applique un traitement. Il peut servir à synthétiser une information à partir des étiquettes ou de la structure, à rechercher un nœud, ou à modifier l'arbre.

Une approche naturelle consiste à utiliser une fonction récursive prenant l'arbre en argument. Le cas de base est l'arbre vide E. Pour un nœud $\mathsf{N}(\ell, x, r)$, on parcourt récursivement les sous-arbres ℓ et r. Trois parcours canoniques apparaissent :

— *préfixe* : traitement avant les sous-arbres ;
— *infixe* : traitement entre les deux sous-arbres ;
— *postfixe* : traitement après les sous-arbres.

Exemple

Considérons l'exemple d'un arbre dont les étiquettes sont des chaînes et où le parcours imprime chaque étiquette. Sur l'arbre binaire



En supposant ici que le parcours du sous-arbre gauche est toujours effectué avant celui du sous-arbre droit
— le parcours préfixe va afficher `ABCD`,
— le parcours infixe `BCAD`
— le parcours postfixe `CBDA`,

En OCaml, on écrit un parcours préfixe comme ceci :

```
let rec preorder t = match t with
| E -> ()
| N (l, x, r) -> print_string x; preorder l; preorder r
```

Supposons que l'on souhaite non plus afficher les étiquettes, mais les renvoyer sous forme de liste, par exemple en ordre préfixe. On peut alors procéder ainsi :

```
let rec preorder t = match t with
| E -> []
| N (l, x, r) -> x :: preorder l @ preorder r
```

C'est correct, mais potentiellement inefficace : le temps peut devenir quadratique en la taille de l'arbre. Soit C_n la complexité de `preorder` sur un arbre de taille n. On a

$$C_0 = 1, \qquad C_n = 1 + C_\ell + \ell + C_{n-1-\ell} \quad \text{avec } 0 \leq \ell \leq n-1,$$

où ℓ est la taille du sous-arbre gauche. Les termes C_ℓ et $C_{n-1-\ell}$ proviennent des deux appels récursifs, le 1 correspond à `::`, et ℓ au coût de `@`, proportionnel à la longueur de la première liste (ici ℓ).

Dans un peigne à gauche (donc $\ell = n-1$), on obtient

$$C_n = 1 + C_{n-1} + (n-1) + C_0 = n + 1 + C_{n-1},$$

d'où

$$C_n = \frac{(n+1)(n+2)}{2},$$

soit un coût quadratique, qui constitue le pire cas. On montre en effet par récurrence que $C_n \leq (n+1)(n+2)/2$ pour tout arbre. La complexité n'est toutefois pas toujours quadratique : pour un peigne à droite (donc $\ell = 0$), on trouve $C_n = 2n + 1$, donc un coût linéaire. La complexité de `preorder` dépend donc de la forme de l'arbre.

On peut néanmoins obtenir une implémentation toujours linéaire en ajoutant un second argument `acc`, appelé *accumulateur* :

```
let rec preorder t acc = match t with
| E -> acc
| N (l, x, r) -> x :: preorder l (preorder r acc)
```

Au lieu de renvoyer la liste des éléments de l'arbre dans l'ordre préfixe, la fonction `preorder` renvoie maintenant cette liste concaténée à l'accumulateur `acc`. En particulier, on en déduit la liste des éléments d'un arbre `t` avec un appel initial à `preorder t []`.

$$C_0 = 1, \qquad C_n = 1 + C_\ell + C_{n-1-\ell},$$

d'où il est facile de déduire par récurrence que $C_n = 2n+1$. Cette nouvelle fonction `preorder` a donc bien une complexité linéaire.

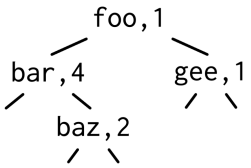
2 Arbres binaires de recherche

Si les valeurs sont comparables, on peut les stocker dans un arbre binaire en plaçant les plus petites dans le sous-arbre gauche et les plus grandes dans le sous-arbre droit : on obtient un *arbre binaire de recherche*. Cette organisation permet des opérations efficaces d'insertion et de recherche, comme avec une table de hachage, mais aussi des opérations fondées sur l'ordre (minimum, n-ième élément, etc.), impossibles avec une table de hachage.

Definition - Arbre binaire de recherche

Un *arbre binaire de recherche* est un arbre binaire dont les éléments sont totalement ordonnés et tel que, pour chaque sous-arbre $\mathsf{N}(\ell, x, r)$, l'élément x est supérieur à tous les éléments de ℓ et inférieur à tous les éléments de r.

Voici un exemple d'arbre binaire de recherche où chaque nœud contient une clé (chaîne) et une valeur (entier), comparées selon l'ordre alphabétique :



Cet exemple montre comment implémenter un tableau associatif avec un arbre binaire de recherche. Nous le faisons maintenant : d'abord en OCaml avec des arbres immuables, puis en C avec des arbres mutables.

En Ocaml.

Arbres binaires de recherche en OCaml.

```
type ('k, 'v) bst =
  | E
  | N of ('k, 'v) bst * 'k * 'v * ('k, 'v) bst

let empty : ('k, 'v) bst =
  E

let rec find (k : 'k) (t : ('k, 'v) bst) : 'v =
  match t with
  | E ->
      raise Not_found
  | N (l, k', v', r) ->
      if k < k' then find k l
      else if k > k' then find k r
      else v'

let rec add (k : 'k) (v : 'v) (t : ('k, 'v) bst) : ('k, 'v) bst =
  match t with
  | E ->
      N (E, k, v, E)
  | N (l, k', v', r) ->
      if k < k' then N (add k v l, k', v', r)
      else if k > k' then N (l, k', v', add k v r)
      else N (l, k, v, r)
```

Le programme implémente en OCaml un tableau associatif par arbres binaires de recherche. Le type polymorphe `bst` utilise `'k` pour les clés et `'v` pour les valeurs ; chaque nœud (`N`) contient une clé et une valeur. Le type est immuable et les clés sont comparées par la comparaison structurelle polymorphe (`<`, `>`).

La fonction `find` renvoie la valeur associée à une clé `k` dans un arbre `t`. Elle compare `k` à la clé racine `k'` : recherche dans le sous-arbre gauche si $k < k'$, droit si $k > k'$, et renvoie `v'` en cas d'égalité. Si un sous-arbre vide `E` est atteint, la clé est absente et l'exception `Not_found` est levée.

La fonction `add` ajoute une entrée : `add k v t` renvoie, par immutabilité, un nouvel arbre contenant les entrées de `t` et $k \mapsto v$. La comparaison avec la clé racine `k'` détermine gauche, droite, ou mise à jour si la clé existe ; si un sous-arbre vide est atteint, une feuille est créée. La fonction `add` préserve la propriété d'arbre binaire de recherche : si `t` l'est, `add k v t` l'est aussi. Ainsi, tout arbre construit via `empty` et `add` est un arbre binaire de recherche, garantissant le bon comportement de `find`. Une interface rendant `bst` abstrait assure que toute valeur de ce type est bien un arbre binaire de recherche.

En C. Implémentons maintenant des arbres binaires de recherche en C avec une *structure mutable* : l'ajout d'une entrée modifie l'arbre en place. On définit d'abord un type de nœud, analogue à celui en OCaml, mais avec deux champs `key` et `value` :

```
typedef struct Node {
  struct Node *left;
  char *key;
  int value;
  struct Node *right;
} node;
```

On rappelle que l'arbre vide est `NULL`. La recherche d'une clé suit le schéma OCaml : descente à gauche ou à droite jusqu'à la clé ; on peut l'écrire récursivement ou avec une boucle `while`. Le programme suivante donne la version itérative `bst_getn` ; si l'on atteint `NULL`, on signale l'échec en renvoyant -1.

L'insertion est plus délicate. On voudrait

`void bst_put(node *t, char *k, int v)`

pour ajouter $k \mapsto v$ dans t, mais si $t =$ `NULL` (première insertion), aucun emplacement n'existe pour accrocher le premier nœud. Tester avant l'appel déplacerait le problème côté client, ce qui est inacceptable. Plus généralement, on ne veut pas révéler au client que le vide est `NULL`. On encapsule donc l'arbre dans une structure `Bst` :

```
typedef struct Bst {
  node *root;
} bst;
```

On revient alors à l'implémentation de `bst_put`. Tester systématiquement si `root` vaut `NULL`, puis répéter ce test lors des insertions récursives, alourdirait le code. On adopte donc une autre approche : une fonction auxiliaire sur le type `node`, qui renvoie la racine après insertion :

`node *bst_putn(node *t, char *k, int v).`

Le retour de valeur résout le cas de l'arbre vide ; sinon, on renvoie simplement `t`. Le programme suivante donne le code de `bst_putn`. On peut vérifier qu'avec n appels récursifs, il y a $n+1$ affectations, mais une seule modifie réellement l'arbre ; les autres réaffectent `t->left` ou `t->right` à leur valeur courante.

La fonction `bst_put` s'en déduit alors trivialement en appelant `bst_putn` sur le champ `root` :

```
void bst_put(bst *t, char *k, int v) {
  t->root = bst_putn(t->root, k, v);
}
```

Le programme suivante contient le code qui encapsule les arbres dans la structure `Bst` et qui ne présente que le type `bst`, abstrait, dans l'interface. En particulier, on ne sait même plus qu'il s'agit d'arbres binaires de recherche.

Arbres binaires de recherche en C

```
typedef struct Node {
    struct Node *left;
    char *key;
    int value;
    struct Node *right;
} node;

node *bst_createn(char *k, int v) {
    node *t = malloc(sizeof(struct Node));
    t->key = k;
    t->value = v;
    t->left = t->right = NULL;
    return t;
}

int bst_getn(node *t, char *k) {
    while (t != NULL) {
        int c = strcmp(k, t->key);
        if (c == 0) return t->value;
        t = c < 0 ? t->left : t->right;
    }
    return -1;
}

node *bst_putn(node *t, char *k, int v) {
    if (t == NULL) return bst_createn(k, v);
    int c = strcmp(k, t->key);
    if (c < 0) {
        t->left = bst_putn(t->left, k, v);
    }
    else if (c > 0) {
        t->right = bst_putn(t->right, k, v);
    }
    else t->value = v;
    return t;
}


struct Bst {
    node *root;
};

bst *bst_create(void) {
    bst *t = malloc(sizeof(struct Bst));
    t->root = NULL;
    return t;
}
```

```
void bst_put(bst *t, char *k, int v) {
    t->root = bst_putn(t->root, k, v);
}

int bst_get(bst *t, char *k) {
    return bst_getn(t->root, k);
}
```

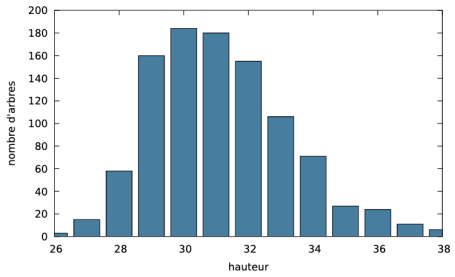
2.1 Complexité.

La complexité des opérations sur un arbre binaire de recherche (recherche, insertion) est bornée par sa hauteur. Dans le cas idéal d'un arbre parfait, $h = \log n$ et ces opérations sont en $O(\log n)$, complexité très satisfaisante mais peu probable après insertions successives ; en pratique, h est généralement plus grand que $\log n$.

Lorsque les insertions sont aléatoires, la hauteur moyenne est $2\ln(n)$, donc proche du minimum.

Exemple

Pour $10,000$ éléments, aucune hauteur supérieure ou égale à 40 n'apparaît.



Distribution de la hauteur d'un arbre binaire contenant 10 000 éléments insérés aléatoirement, pour 1000 tirages

En revanche, des insertions triées produisent un peigne avec $h(t) = n(t) - 1$, et plus généralement certains arbres ont une hauteur linéaire, rendant les opérations comparables à celles d'une liste chaînée car presque tous les nœuds doivent être examinés.

Pour éviter ces situations pathologiques, on cherche à *équilibrer* les arbres binaires de recherche afin de garantir une hauteur logarithmique, indépendamment des opérations effectuées.

Definition- Arbres équilibrés

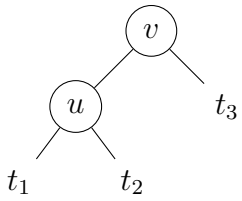
Un ensemble S d'arbres binaires est dit *équilibré* s'il existe une constante C telle que, pour tout arbre non vide $t \in S$,

$$h(t) \leq C \times \log(n(t)).$$

Cette inégalité est toujours vraie pour un arbre à un seul nœud ($h = 0$, $n = 1$).

2.2 Rotations.

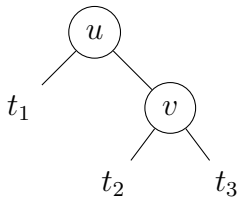
Considérons un arbre binaire de recherche composé de deux nœuds portant u et v, et de trois sous-arbres t_1, t_2, t_3.



Par la propriété BST, on a : tous les éléments de t_1 sont $< u$, ceux de t_2 sont entre u et v, et ceux de t_3 sont $> v$, ce que l'on résume par

$$t_1 < u < t_2 < v < t_3. \tag{1}$$

On peut alors réorganiser localement ces deux nœuds tout en conservant 1 :



on obtient une *rotation droite*, ainsi nommée car u et v se déplacent visuellement vers la droite. Le nouvel arbre est encore un arbre binaire de recherche grâce à 1.

Réciproquement, on peut transformer le second arbre en le premier par une *rotation gauche*. Le programme suivante présente ces deux rotations et leur code.

Rotations dans les arbres binaires de recherche

- En OCaml, avec des arbres immuables :

```
let rotate_right = function
  | N (N (t1, ku, vu, t2), kv, vv, t3) ->
      N (t1, ku, vu, N (t2, kv, vv, t3))
  | t -> t

let rotate_left = function
  | N (t1, ku, vu, N (t2, kv, vv, t3)) ->
      N (N (t1, ku, vu, t2), kv, vv, t3)
  | t -> t
```

- En C, avec des arbres mutables :

```
node *bst_rotate_right(node *t) {
    if (t == NULL || t->left == NULL) return t;
    node *l = t->left;
    t->left = l->right;
    l->right = t;
```

```
    return l;
}

node *bst_rotate_left(node *t) {
    if (t == NULL || t->right == NULL) return t;
    node *r = t->right;
    t->right = r->left;
    r->left = t;
    return r;
```

En OCaml comme en C, on implémente une fonction renvoyant la racine après rotation si elle est possible, sinon l'arbre inchangé.

Une rotation s'exécute en temps constant, indépendamment de la taille des trois sous-arbres : en OCaml, deux nœuds sont alloués ; en C, deux pointeurs sont échangés. Le code est symétrique entre rotation droite et gauche.

Ces rotations sont l'outil central de l'équilibrage des arbres binaires de recherche. Lorsqu'un arbre devient déséquilibré, on applique une ou plusieurs rotations pour rétablir un meilleur équilibre. De nombreux algorithmes existent ; nous étudions ici les *arbres rouge-noir*.

3 Arbres rouge-noir

Les arbres rouge-noir, encore appelés arbres bicolores, constituent une famille d'arbres binaires de recherche équilibrés.

Définition — Arbre rouge-noir

Un *arbre rouge-noir* est un arbre binaire dont chaque nœud est coloré en rouge ou en noir, vérifiant :

1. un nœud rouge n'a jamais de père rouge ;
2. tout chemin de la racine à un sous-arbre vide contient le même nombre de nœuds noirs.

Dans la suite, on note $b(t)$ le nombre de nœuds noirs le long de tout chemin de la racine à un sous-arbre vide d'un arbre rouge-noir t. On note que, pour un arbre rouge-noir non vide, ses deux sous-arbres sont également des arbres rouge-noir.

Lemma

Pour tout arbre rouge-noir t, on a les deux inégalités

$$h(t) \leq 2b(t), \qquad 2^{b(t)} \leq n(t) + 1.$$

Demonstration On montre ces inégalités par induction structurelle sur t.

— Le cas de base d'un arbre vide est trivial, car $h(t) = -1$ et $n(t) = b(t) = 0$.

— Soit t un arbre rouge-noir non vide, et ℓ et r ses deux sous-arbres.

  — Si la racine de t est noire, alors $b(\ell) = b(r) = b(t) - 1$. Dès lors,

$$h(t) = 1 + \max(h(\ell), h(r)) \leq 1 + 2(b(t) - 1) < 2b(t).$$

De plus,

$$n(t) + 1 = n(\ell) + n(r) + 1 + 1 \geq 2^{b(t)-1} - 1 + 2^{b(t)-1} - 1 + 1 + 1 = 2^{b(t)}.$$

— Si la racine de t est rouge, alors $b(\ell) = b(r) = b(t)$. Si t ne contient qu'un seul nœud, alors $h(t) = b(t) = 0$ et $n(t) = 1$, d'où les deux inégalités. Sinon, t contient deux sous-arbres non vides et donc les racines sont noires par la propriété d'arbre rouge-noir. On a donc quatre sous-arbres $\ell\ell, \ell r, r\ell, rr$ sous ces deux nœuds noirs et

$$h(t) = 2 + \max(h(\ell\ell), h(\ell r), h(r\ell), h(rr)) \leq 2 + 2(b(t) - 1) = 2b(t).$$

De plus,

$$n(t) + 1 = n(\ell) + n(r) + 1 + 1 \geq 2^{b(t)} - 1 + 2^{b(t)} - 1 + 1 + 1 > 2^{b(t)}.$$

Le corollaire s'en déduit aisément. En effet,

$$h(t) \leq 2b(t) \leq 2\log(n(t) + 1) \leq 4\log(n(t)) \quad \text{pour } n(t) \geq 2,$$

et par ailleurs l'inégalité $h(t) \leq 4\log(n(t))$ tient trivialement pour $n(t) = 1$.

3.1 Réalisation en OCaml.

On montre comment construire des arbres rouge-noir. On le fait avec des arbres immuables, en OCaml, mais une implémentation mutable ou en C serait également possible.

Un arbre rouge-noir est un arbre binaire de recherche auquel on ajoute, dans chaque nœud, une information de couleur.

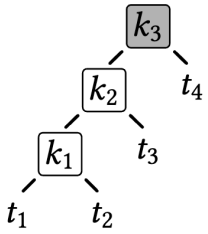
```
type color = R | B

type ('k, 'v) rbt =
  | E
  | N of color * ('k, 'v) rbt * 'k * 'v * ('k, 'v) rbt
```

Les opérations de simple consultation de l'arbre binaire de recherche restent inchangées : la fonction `find` est identique à celle du programme 7.24, la couleur des nœuds étant ignorée.

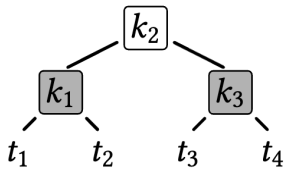
En revanche, les opérations de construction doivent préserver à la fois la propriété d'arbre binaire de recherche et celle d'arbre rouge-noir. Pour l'opération `add`, il faut choisir la couleur du nouveau nœud : le noir ne doit pas violer la propriété 2, le rouge la propriété 1. On choisit néanmoins d'insérer les nouveaux nœuds en rouge, et le code de `add` débute ainsi :

```
let rec add k v t = match t with
| E -> N (R, E, k, v, E)
```

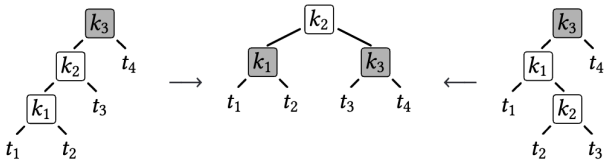
Dans un arbre non vide, l'insertion se fait récursivement à gauche si k est plus petite que la clé racine, et à droite si elle est plus grande. Si le nouveau nœud rouge est ajouté sous un nœud rouge, deux nœuds rouges consécutifs apparaissent, par exemple après une insertion à gauche du sous-arbre gauche.



On peut alors utiliser une rotation pour restaurer la propriété rouge-noir. L'arbre



contient les mêmes éléments, reste un arbre binaire de recherche, et rétablit la propriété rouge-noir. Si l'insertion problématique se fait à droite du sous-arbre gauche, deux rotations suffisent.



Ce rééquilibrage peut placer un nœud rouge à la racine, créant éventuellement deux rouges consécutifs plus haut. La fonction d'insertion traite ce cas de la même manière, avec rééquilibrage si nécessaire, qu'il s'agisse d'un rouge nouvellement inséré ou issu d'une rotation.

Supposons une fonction `lbalance` réalisant ces deux rééquilibrages si besoin, de type

`color * ('k,'v) rbt * 'k * 'v * ('k,'v) rbt -> ('k,'v) rbt}.`

Elle agit comme le constructeur `N` si aucun rééquilibrage n'est requis et renvoie toujours la racine obtenue. L'insertion dans le sous-arbre gauche s'écrit alors en utilisant `lbalance` à la place de `N`.

```
| N (c, l, k', v', r) ->
    if k < k' then lbalance (c, add k v l, k', v', r)
```

On termine la fonction `add` avec un cas similaire pour l'insertion à droite, en supposant avoir écrit également une fonction `rbalance` symétrique de la fonction `lbalance`, et enfin le cas où l'insertion se fait à la racine :

```
else if k > k' then rbalance (c, l, k', v', add k v r)
else N (c, l, k, v, r) (* on ecrase la valeur *)
```

Dans ce dernier cas, on ne change pas la couleur du nœud, car la structure de l'arbre ne change pas.

Il reste encore un problème. Nos deux fonctions de rééquilibrage éliminent la présence de nœuds rouges consécutifs situés en profondeur. Mais elles ne vont pas éliminer en revanche deux nœuds rouges consécutifs à la racine de l'arbre. Pour y remédier, il suffit de toujours colorier en noir la racine de l'arbre après l'insertion.

```
let add k v t = match add k v t with
| E -> assert false
| N (_, l, k', v', r) -> N (B, l, k', v', r)
```

En particulier, c'est cette toute dernière opération qui va faire apparaître des nœuds noirs dans nos arbres rouge-noir ! Le programme suivante contient la totalité du code de les fonctions.

Arbres rouge-noir en OCaml

```
type color = R | B

type ('k, 'v) rbt =
  | E
  | N of color * ('k, 'v) rbt * 'k * 'v * ('k, 'v) rbt

let empty : ('k, 'v) rbt =
  E

let lbalance = function
  | (B, N (R, N (R, t1, k1, v1, t2), k2, v2, t3), k3, v3, t4)
  | (B, N (R, t1, k1, v1, N (R, t2, k2, v2, t3)), k3, v3, t4) ->
      N (R, N (B, t1, k1, v1, t2), k2, v2, N (B, t3, k3, v3, t4))
  | (c, l, k, v, r) -> N (c, l, k, v, r)

let rbalance = function
  | (B, t1, k1, v1, N (R, N (R, t2, k2, v2, t3), k3, v3, t4))
  | (B, t1, k1, v1, N (R, t2, k2, v2, N (R, t3, k3, v3, t4))) ->
      N (R, N (B, t1, k1, v1, t2), k2, v2, N (B, t3, k3, v3, t4))
  | (c, l, k, v, r) -> N (c, l, k, v, r)

let rec add (k: 'k) (v: 'v) (t: ('k, 'v) rbt) : ('k, 'v) rbt =
  match t with
  | E ->
      N (R, E, k, v, E)
  | N (c, l, k', v', r) ->
      if k < k' then lbalance (c, add k v l, k', v', r)
      else if k > k' then rbalance (c, l, k', v', add k v r)
      else N (c, l, k, v, r) (* on ecrase la valeur *)

let add (k: 'k) (v: 'v) (t: ('k, 'v) rbt) : ('k, 'v) rbt =
  match add k v t with
  | E -> assert false
  | N (_, l, k', v', r) -> N (B, l, k', v', r)
```

Complexité. Les fonctions `lbalance` et `rbalance` sont en temps constant. La fonction `add` effectue une seule descente récursive (gauche ou droite) suivie d'une opération constante, donc sa complexité est majorée par la hauteur de l'arbre. Or les arbres rouge-noir sont équilibrés, donc de hauteur logarithmique ; l'insertion est ainsi en temps logarithmique.

Théorème d'insertion dans un arbre rouge-noir

L'insertion d'un élément dans un arbre rouge-noir t a une complexité $O(\log(n(t)))$.

Par le même raisonnement, toute opération avec une unique descente (par exemple trouver le minimum ou supprimer une entrée) est aussi logarithmique.

4 Tas et files de priorité

Les arbres binaires ont de nombreuses applications, notamment la structure de tas, utilisée pour implémenter des files de priorité.

Une file de priorité stocke un multiensemble d'éléments totalement ordonnés et fournit deux opérations : ajouter un élément et retirer le plus petit. Un même élément peut apparaître plusieurs fois, d'où la notion de multiensemble.

Les files de priorité interviennent dans de nombreux algorithmes, par exemple Dijkstra et Kruskal, ainsi que dans le *tri par tas*, qui consiste à insérer tous les éléments puis à les extraire dans l'ordre croissant. Dans cette section, on définit d'abord la structure de tas, puis on en déduit une file de priorité immuable en OCaml et mutable en C.

Définition – structure de tas

Un arbre binaire d'éléments totalement ordonnés possède la *structure de tas* ssi :
- il est `E`, ou
- il est `N`(ℓ, x, r), où ℓ et r sont des tas et x est inférieur ou égal à tous les éléments de ℓ et r.

Ainsi, dans un tas non vide, la racine contient l'élément minimal, propriété clé pour implémenter efficacement une file de priorité.

4.1 Files de priorité immuablesz en OCaml

En OCaml, la structure de tas est celle d'un arbre binaire, avec le type polymorphe

```
'a heap = E | N of 'a heap * 'a * 'a heap}
```

Les éléments sont comparés par comparaison structurelle. Obtenir le minimum est immédiat : il se trouve à la racine.

La suppression du minimum et l'insertion sont plus délicates : il faut combiner les sous-arbres gauche et droit, ou faire descendre la plus grande valeur, sans savoir a priori de quel côté poursuivre. On résout ces deux problèmes en définissant une fonction de fusion `merge`, qui construit récursivement un tas contenant tous les éléments de deux tas. Si l'un est vide, on renvoie l'autre ; sinon, on compare les racines et la plus petite devient la nouvelle racine.

Dans le cas $x_1 \leq x_2$, on construit le résultat à partir de l_1, r_1 et t_2 en plaçant r_1 à gauche et l_1 à droite afin de favoriser l'équilibre ; le cas $x_2 < x_1$ est symétrique.

Une fois `merge` définie, l'insertion consiste à fusionner le tas avec un singleton contenant l'élément, et la suppression du minimum à fusionner les sous-arbres gauche et droit.

Le code complet est donné dans le programme suivante.

Files de priorité immuables en OCaml

```
type 'a heap = E | N of 'a heap * 'a * 'a heap

let empty : 'a heap =
  E

let is_empty (t: 'a heap) : bool =
  t = E

let rec merge (t1: 'a heap) (t2: 'a heap) : 'a heap =
  match t1, t2 with
  | E, t | t, E ->
      t
  | N (l1, x1, r1), N (l2, x2, r2) ->
      if x1 <= x2 then
        N (merge r1 t2, x1, l1)
      else
        N (merge r2 t1, x2, l2)

let insert (x: 'a) (t: 'a heap) : 'a heap =
  merge (N (E, x, E)) t

let get_min (t: 'a heap) : 'a =
  match t with
  | E -> invalid_arg "get_min"
  | N (_, x, _) -> x

let extract_min (t: 'a heap) : 'a * 'a heap =
  match t with
  | E -> invalid_arg "extract_min"
  | N (l, x, r) -> x, merge l r
```

Les tas que nous venons de présenter s'appellent en anglais des *skew heaps*, ce que l'on pourrait traduire par *tas obliques*, même s'il n'y a pas vraiment de traduction officielle.

Ils sont *auto-équilibrés* dans le sens où une séquence de n insertions successives dans un tas initialement vide a tout de même un coût total $O(n \log n)$. Tout se passe donc comme si chacune des insertions avait un coût $O(\log n)$. En réalité, certaines insertions coûtent plus cher et d'autres moins cher, seul le coût moyen étant logarithmique. On a donc une *complexité amortie* $O(\log n)$ pour l'insertion (et la suppression) dans un tas oblique.

4.2 Complexité.

L'insertion successive de n éléments dans un tas initialement vide coûte au total $O(n \log n)$.

4.3 Files de priorité mutables, en C

Comme un arbre binaire complet peut être représenté dans un tablea, on exploite cette idée pour implémenter une file de priorité en C.
On suppose que la file contient des entiers et a une capacité maximale `capacity`.
On utilise un tableau `data` de taille `capacity`, dont les `size` premières cases représentent un arbre binaire complet satisfaisant la structure de tas.

```
typedef struct Pqueue {
  int capacity, size; // 0 <= size <= capacity
  int *data; // tableau de taille capacity
} pqueue;

pqueue *pqueue_create(int capacity) {
  pqueue *q = malloc(sizeof(struct Pqueue));
  q->capacity = capacity;
  q->data = calloc(capacity, sizeof(int));
  q->size = 0;
  return q;
}
```

Insertion

Insérer un élément dans la file de priorité ne peut se faire que si la file n'est pas déjà pleine, ce que l'on vérifie avec une assertion.

```
void pqueue_add(pqueue *q, int x) {
  assert(!pqueue_is_full(q));
```

Le principe d'insertion est le suivant : on place le nouvel élément dans la première case libre (en bas à droite du tas), puis on le fait remonter jusqu'à rétablir la propriété de tas. On utilise une variable i pour sa position courante et on poursuit la remontée tant que l'on n'a pas atteint la racine

```
int i = q->size;
while (i > 0) {

  int fi = (i - 1) / 2; (*L element au-dessus du noeud i se trouve a la
                          position (i-1)/2*).
  int y = q->data[fi];
```

On le compare alors au nouvel élément x.
Si la propriété de tas est respectée, c'est-à-dire si $y \leq x$, on sort de la boucle ; la remontée est terminée.Sinon, on fait descendre l'élément y à la place i, puis i prend la valeur du nœud qui contenait y et la boucle reprend.

```
  if (y <= x) break;
  q->data[i] = y;
  i = fi;
}
```

Une fois sorti de la boucle, soit parce qu'on est arrivé à la racine, soit parce que la propriété de tas était rétablie, on affecte l'élément x au nœud i et on n'oublie pas d'incrémenter le nombre d'éléments.

```
q->data[i] = x;
q->size++;
}
```

Suppression du plus petit élément.

Le minimum se trouve à la racine, donc à l'indice 0 du tableau. Pour le supprimer, on le remplace par l'élément situé en bas à droite du tas, puis on fait descendre ce dernier jusqu'à rétablir la propriété de tas. On commence par vérifier que la file n'est pas vide et par stocker le minimum dans une variable r.

```
int pqueue_remove_min(pqueue *q) {
  assert(q->size > 0);
  int r = q->data[0];
```

On diminue ensuite le nombre d'éléments de la file de priorité.
S'il tombe à zéro, c'est que la file ne contenait qu'un seul élément et il suffit alors de renvoyer r.
Sinon, on récupère l'élément x situé tout en bas à droite du tas, et on se donne une variable i pour la position candidate à recevoir cet élément. Initialement, i vaut 0, c'est-à-dire qu'on envisage de placer x à la racine du tas, à la place de r.

```
  int n = --q->size;
  if (n == 0) return r;
  int x = q->data[n];
  int i = 0;
```

Commence alors la descente de x dans l'arbre jusqu'au rétablissement de la propriété de tas. On procède dans une boucle infinie, quittée par `break`. On calcule d'abord l'indice j correspondant à la racine du sous-arbre gauche.

```
  while (true) {
    int j = 2 * i + 1;
```

Si cet indice est en dehors de l'arbre, alors on sort de la boucle.

```
    if (j >= n) break;
```

Sinon, il faut considérer le sous-arbre droit afin de déterminer quel sous-arbre contient la plus petite valeur à sa racine. Le cas échéant, on remplace j par $j + 1$.

```
    if (j + 1 < n && q->data[j] > q->data[j + 1]) j++;
```

On a vérifié que $j + 1 < n$, car j peut être le dernier nœud du tas. Une fois j fixé, on compare l'élément du nœud j avec x.

Si la propriété de tas est satisfaite, on quitte la boucle.
Sinon, on fait remonter l'élément du nœud j dans le nœud i puis i prend la valeur de j et la boucle reprend.

```
    if (x <= q->data[j]) break;
    q->data[i] = q->data[j];
    i = j;
  }
```

Une fois sorti de la boucle, on place la valeur x dans le nœud i et on termine en renvoyant r.

```
  q->data[i] = x;
  return r;
}
```

Le code complet est donné dans le programme suivante.

Files de priorité mutables en C

```
typedef struct Pqueue {
  int capacity, size;   // 0 <= size <= capacity
  int *data;            // tableau de taille capacity
} pqueue;

void move_up(int a[], int i, int x) {
  while (i > 0) {
    int fi = (i - 1) / 2;
    int y = a[fi];
    if (y <= x) break;
    a[i] = y;
    i = fi;
  }
  a[i] = x;
}

void pqueue_add(pqueue *q, int x) {
  assert(!pqueue_is_full(q));
  move_up(q->data, q->size, x);
  q->size++;
}

void move_down(int a[], int i, int x, int n) {
  while (true) {
    int j = 2 * i + 1;
    if (j >= n) break;
    if (j + 1 < n && a[j] > a[j + 1]) { j++; }
    if (a[j] >= x) break;
    a[i] = a[j];
    i = j;
```

```
  }
  a[i] = x;
}

int pqueue_remove_min(pqueue *q) {
  assert(!pqueue_is_empty(q));
  int r = q->data[0];
  int n = --q->size;
  if (n > 0) move_down(q->data, 0, q->data[n], n);
  return r;
}
```

Théorème d' insertion et suppression dans un tas

L'insertion et la suppression d'un élément dans un tas contenant n éléments a une complexité $O(\log n)$.

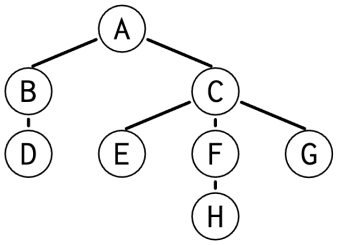
5 Arbres

Les arbres binaires ont une structure rigide (deux sous-arbres par nœud). Pour plus de flexibilité, on utilise des *arbres* où chaque nœud peut avoir un nombre variable de sous-arbres.

Définition d'arbre

Un *arbre* est un ensemble de $n \geq 1$ nœuds tel que : un nœud r est la racine ; les $n-1$ autres sont partitionnés en $k \geq 0$ sous-arbres disjoints ; la racine r est reliée à la racine de chacun de ces sous-arbres.
Un arbre réduit à un seul nœud est une *feuille*.

Exemple



La racine A possède deux sous-arbres B, D et C, E, F, G, H, de racine B puis C. Les sous-arbres d'un nœud forment une séquence ordonnée appelée *forêt*, éventuellement vide ; un arbre peut donc être vu comme une racine et une forêt.

Définition - hauteur

La *hauteur* est la plus grande distance entre la racine et un nœud ; un arbre à un seul nœud a hauteur 0. Équivalemment, la profondeur de la racine est 0, celle des racines de ses sous-arbres est 1, etc., et la hauteur est la profondeur maximale.

Attention

Contrairement à l'intuition, un arbre binaire n'est pas un arbre. D'abord, un arbre binaire peut être vide, alors qu'un arbre contient toujours au moins un nœud. Ensuite, l'arbre binaire distingue sous-arbre gauche et droit : deux arbres binaires symétriques sont donc distincts, d'où la notion d'arbres *positionnels*. À l'inverse, il n'existe qu'un seul arbre à deux nœuds.
Enfin, les dessins d'arbres binaires montrent des « petites pattes » pour les sous-arbres vides, ce qui n'est pas le cas pour les arbres.

5.1 Représentation en machine

On décrit maintenant la représentation en machine des arbres, à la fois en OCaml et en C.

OCaml.

Le programme suivant définit le type polymorphe `'a tree`, où la forêt des sous-arbres est représentée par des listes OCaml. Il inclut aussi une fonction `size` calculant le nombre de nœuds, définie récursivement avec `size_forest`, qui calcule la taille d'une forêt.

Arbres en OCaml

```
type 'a tree =
  | N of 'a * 'a tree list

let rec size (N (_, tl): 'a tree) : int =
  1 + size_forest tl
and size_forest (tl: 'a tree list) : int =
  match tl with
  | []      -> 0
  | t :: tl -> size t + size_forest tl
```

C.

Le programme suivante définit un type `tree` (étiquettes entières) comme pointeur vers une structure `Tree` contenant : `value`, un pointeur `children` vers le premier sous-arbre, et un pointeur `next` vers le nœud suivant de la forêt (`NULL` s'il n'existe pas). L'arbre est ainsi représenté par des structures liées via `children` (sous-arbres) et `next` (frères).

Le programme fournit `tree_create` (création d'un nœud, pointeurs à `NULL`) et `tree_add_first_child` (ajout en tête des sous-arbres). L'exemple se construit en ajoutant d'abord `C` comme second sous-arbre de `A`, puis `B` comme premier.

Arbres en C

```
typedef struct Tree {
  int value;
  struct Tree *children;   // premier sous-arbre
  struct Tree *next;       // suivant dans la forêt
```

```
} tree;

tree *tree_create(int v) {
  tree *t = malloc(sizeof(struct Tree));
  t->value = v;
  t->children = NULL;
  t->next = NULL;
  return t;
}

void tree_add_first_child(tree *t, tree *c) {
  assert(c->next == NULL);
  c->next = t->children;
  t->children = c;
}

int tree_size(tree *t) {
  int s = 1;
  for (tree *c = t->children; c != NULL; c = c->next) {
    s += tree_size(c);
  }
  return s;
}
```

5.1.1 Comparaison

Bien que différentes en apparence, les représentations C et OCaml sont proches. Les champs `next` de `Tree` forment une liste simplement chaînée pour la forêt, analogue au type `list` d'OCaml. La seule différence est mémoire : C utilise un seul bloc (`Tree`), OCaml en utilise deux (`N` et `::`).

5.2 Conversion de et vers les arbres binaires

Il existe un isomorphisme naturel entre arbres binaires et arbres. Les types C correspondants sont structurellement identiques :

```
struct Node { int value; struct Node *left, *right; }
struct Tree { int value; struct Tree *children, *next; }
```

Ils ne diffèrent que par les noms. Plus précisément, il y a isomorphisme entre :

- un arbre binaire et une forêt ;
- un arbre binaire non vide dont le sous-arbre droit est vide et un arbre.

Le programme suivante donne quatre fonctions OCaml réalisant ces isomorphismes. La fonction `bintre_of_tree` réussit toujours, tandis que `tree_of_bintree` échoue si l'argument n'est pas un arbre binaire non vide au sous-arbre droit vide : sinon on obtient une forêt vide (arbre vide) ou une forêt avec au moins deux arbres (sous-arbre droit non vide).

Conversions entre arbres et arbres binaires

Conversion d'un arbre en un arbre binaire :

```
let rec bintree_of_forest (tl: 'a tree list) : 'a bintree =
  match tl with
  | [] ->
      Bintree.E
  | (Tree.N (x, ch)) :: tl ->
      Bintree.N (bintree_of_forest ch, x, bintree_of_forest tl)

let bintree_of_tree (t: 'a tree) : 'a bintree =
  bintree_of_forest [t]
```

Et inversement :

```
let rec forest_of_bintree (t: 'a bintree) : 'a tree list =
  match t with
  | Bintree.E ->
      []
  | Bintree.N (l, x, r) ->
      Tree.N (x, forest_of_bintree l) :: forest_of_bintree r

let tree_of_bintree (t: 'a bintree) : 'a tree =
  match forest_of_bintree t with
  | [t] -> t
  | _ -> invalid_arg "tree_of_bintree"
```

5.3 Parcours préfixe et postfixe

Comme pour les arbres binaires, on peut définir des parcours sur les arbres. La différence est que le nombre de sous-arbres est arbitraire, donc le parcours infixe n'a plus de sens ; seuls les parcours préfixe et postfixe restent pertinents, selon que le nœud est visité avant ou après ses sous-arbres. On les exprime de façon générique par une fonction de type

$$('a \to unit) \to' atree \to unit$$

qui applique une fonction à chaque nœud. Le programme suivante fournit les fonctions OCaml `preorder` et `postorder`, utilisant `List.iter` pour parcourir la forêt, bien qu'une fonction récursive dédiée soit aussi possible.

Parcours d'arbre

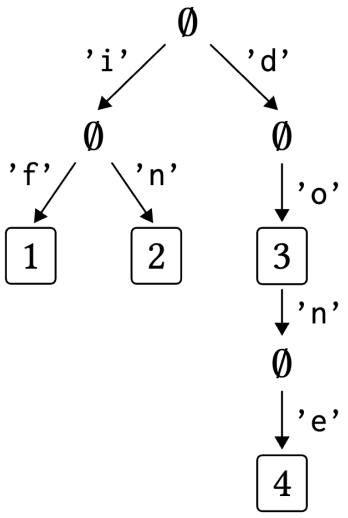
```
let rec preorder f (N (x, tl)) =
  f x;
  List.iter (preorder f) tl

let rec postorder f (N (x, tl)) =
  List.iter (postorder f) tl;
  f x
```

5.4 Arbres préfixes

On présente une structure d'arbre pour représenter des tableaux associatifs à clés chaînes. Chaque branche est étiquetée par une lettre et chaque nœud contient une valeur si le mot formé par le chemin depuis la racine appartient au tableau. Par exemple, l'arbre code

$$\text{"if"} \mapsto 1,\ \text{"in"} \mapsto 2,\ \text{"do"} \mapsto 3,\ \text{"done"} \mapsto 4.$$

Chaque nœud correspond à un mot w (la racine au mot vide) et contient soit $\varnothing$, soit la valeur associée à w. Cette structure s'appelle un *arbre préfixe* ou *trie* (de « retrieval »).



Réalisation en OCaml.

On implémente en OCaml un arbre préfixe mutable (tableau associatif) à clés chaînes et valeurs polymorphes, avec la même interface qu'une table de hachage.

Un nœud est défini par :

```
type 'a trie = {
  mutable value : 'a option;
  branches : (char, 'a trie) Hashtbl.t;
}
```

Le champ mutable `value` contient la valeur éventuelle, et `branches` (table `Hashtbl`) associe chaque caractère au sous-arbre correspondant ; une feuille a une table vide.

Un trie vide est un unique nœud avec `value = None` et `branches` vide :

```
let create () =
  { value = None; branches = Hashtbl.create 8 }
```

La valeur 8 est choisie ici arbitrairement. De toutes façons, les tables de hachage d'OCaml s'adaptent dynamiquement au nombre d'entrées.

5.4.1 Recherche d'une clé.

La fonction `get` renvoie la valeur associée à une clé `s` ou lève `Not_found`. La recherche descend récursivement en suivant les caractères de `s`, via une fonction `find` et un indice `i`.

```
let get t s =
  let rec find t i =
    if i = String.length s then
      (match t.value with None -> raise Not_found | Some v -> v)
    else
      find (Hashtbl.find t.branches s.[i]) (i + 1)
  in
  find t 0
```

5.4.2 Ajout d'une nouvelle entrée.

Insertion d'une clé. L'insertion d'une entrée pour la clé `s` consiste à descendre le long des branches étiquetées par les caractères de `s`, en créant si nécessaire de nouvelles branches. On utilise une fonction récursive locale `add`.

```
    let put t s v =
  let rec add t i =
    if i = String.length s then
      t.value <- Some v
    else
      let b =
        try
          Hashtbl.find t.branches s.[i]
        with Not_found ->
          let b = create () in
          Hashtbl.add t.branches s.[i] b;
          b
      in
      add b (i + 1)
  in
  add t 0
```

Si la fin du mot est atteinte, on écrit `v` (en remplaçant éventuellement l'ancienne valeur). Sinon, on récupère ou crée le sous-arbre correspondant au caractère courant, puis on poursuit la descente. L'insertion fonctionne aussi pour la chaîne vide.

Complexité. Les fonctions `get` et `put` parcourent la clé caractère par caractère. À chaque étape, elles effectuent des opérations en temps constant et des appels à `Hashtbl.find` ou `Hashtbl.add`, considérés en temps constant amorti. Ansi, recherche et insertion s'exécutent en temps proportionnel à la longueur de la clé, indépendamment du nombre d'entrées.

Si la longueur des clés est bornée, ces opérations sont donc en temps constant amorti. On peut alors comparer les deux approches : en pratique, la table de hachage est plus efficace en temps et en espace. Par exemple, stocker tous les mots du dictionnaire français prend environ 3 fois moins de temps et 7 fois moins de mémoire avec une table de hachage qu'avec un trie.