Chapitre 3

Révisions : tableaux et listes

Ce chapitre présente quelques rappels sur les structures linéaires : tableaux et listes. On compare notamment l'intérêt algorithmique de chacune de ses structures.

1 Rappels sur les tableaux

Un tableau est une suite d'éléments consécutifs en mémoire, tous de même type. Un tableau est déterminé par :

- sa taille,
- le type des éléments qu'il peut contenir,
- l'adresse de sa première case.

La figure 3.1 représente un tableau contenant les éléments du multi-ensemble $S = \{\!\{15\,,17\,,42\,,42\,,45\,,84\,,99\}\!\}$ (dans l'ordre où ils sont écrits).

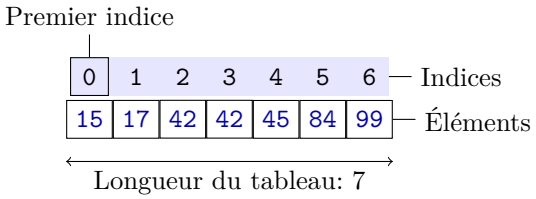


FIG. 3.1 : Un tableau contenant 7 entiers

Remarque. 3.1 Même si on veut seulement représenter un multi-ensemble, un tableau contient en fait plus d'information, car ses éléments sont naturellement ordonnés par leur indice dans le tableau. Un tableau représente donc en fait, mathématiquement, un *vecteur*.

Cela n'empêche pas d'utiliser cette structure pour représenter un multi-ensemble, en ignorant simplement l'ordre des éléments. Autrement dit, si on représente un multi-ensemble par un tableau, permuter les éléments de ce tableau ne change pas le multi-ensemble représenté. Il peut alors être avantageux de maintenir une représentation particulière de ce multi-ensemble (par exemple triée), pour pouvoir effectuer des tâches plus efficacement (par exemple, en utilisant la dichotomie).

Un tableau est implanté en rangeant ses éléments de façon *consécutive en mémoire*, à partir de l'adresse du premier élément du tableau. Ce choix a un avantage et deux inconvénients :

1. **Accès direct**. L'avantage des tableaux est qu'ils permettent l'accès à un élément dont l'indice est connu en temps $O(1)$ (cette propriété est appelée *accès direct*). En effet, pour accéder à l'élément

se trouvant en $i^{\text{ème}}$ case d'un tableau, il suffit d'ajouter à l'adresse de son premier élément i fois la taille de chaque élément. Le calcul de l'adresse de l'élément `p[i]` nécessite donc seulement une addition et une multiplication. L'accès à la valeur `p[i]`, ou `*(p+i)`, nécessite en plus de dé-référencer l'adresse. On a donc un nombre constant d'opérations « élémentaires » à effectuer (une addition, une multiplication et un dé-référencement), ce qui nécessite un temps $O(1)$, c'est-à-dire constant, ***indépendant de l'indice i et de la taille du tableau.***

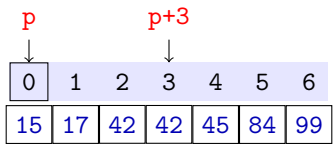


FIG. 3.2 : Accès direct

Remarque. 3.2 En langage C, l'arithmétique des pointeurs effectue la multiplication par la taille des éléments du tableau sans que l'on ait à l'écrire : si on écrit

```
int p[10];
float q[20];
```

alors `p + 5` est l'adresse de la $6^{\text{ème}}$ case de `p` et `q + 15` est l'adresse de la $16^{\text{ème}}$ case de `q`. On n'a pas besoin de (et il ne faut pas, en fait) multiplier par `sizeof(int)` dans le cas de `p`, ou par `sizeof(float)` dans le cas de `q`.

2. **Redimensionnement coûteux.** Dans certains langages, le nombre d'éléments d'un tableau est fixe et décidé au moment de l'allocation mémoire. Dans ce cas, il n'est pas possible d'étendre un tableau dont toutes les cases sont occupées pour y ranger un élément supplémentaire, sans faire soi-même une recopie à la main.

   Il existe cependant des langages permettant d'étendre un tableau. En C, vous avez vu la fonction de la bibliothèque standard de prototype `void * realloc(void *ptr, size_t size);`. Cette fonction permet de redimensionner un tableau déjà alloué par une des fonctions `malloc` ou `calloc`. Par exemple, l'instruction C :

   ```
   realloc(p, 10*sizeof(int));
   ```

   fait passer de la situation de la Figure 3.3 (a) à celle de la Figure 3.3 (b) (si elle réussit, et en supposant que `p` est un tableau contenant des éléments de type `int`).

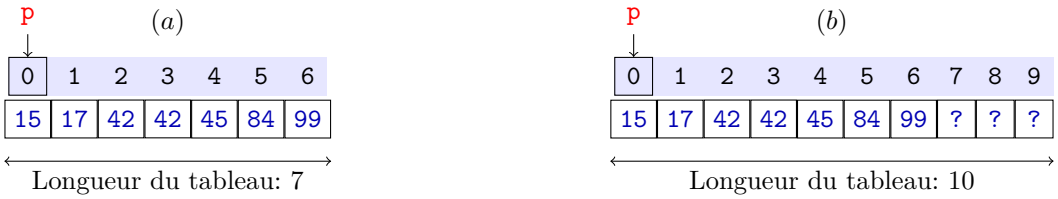


FIG. 3.3 : Redimensionnement d'un tableau

Cependant, même s'il y a assez de mémoire disponible, redimensionner un tableau peut conduire à *recopier* tout le tableau. C'est le cas s'il n'y a pas assez d'espace contigu en mémoire, par exemple si d'autres données ont déjà été allouées « de chaque côté du tableau », comme dans la Figure 3.4.

Remarque. 3.3 Dans l'exemple de la figure 3.4, redimensionner `p2` serait coûteux même s'il y avait de la place disponible « à sa gauche », puisque l'extension doit se faire sur la droite.

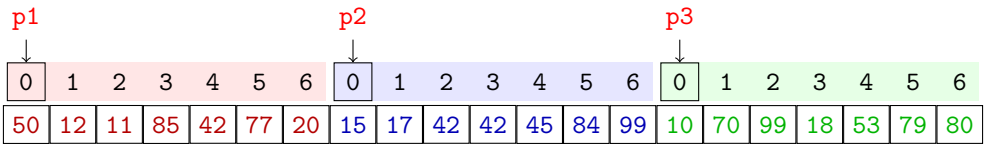


FIG. 3.4 : Une situation où redimensionner `p2` est coûteux

Remarque. 3.4 Au niveau de la programmation, lorsqu'un gestionnaire de mémoire comme celui de la bibliothèque standard C a besoin d'étendre un tableau de taille n, dire que le coût de l'extension est $O(n)$ dans le cas le pire n'est pas tout à fait exact. En effet, pour allouer le nombre voulu de cases, l'algorithme de gestion mémoire doit trouver l'espace ***consécutivement*** en mémoire (car dans un tableau, on range les éléments de façon consécutive). Il est possible qu'il y ait suffisamment d'espace en mémoire, mais pas consécutivement. Si l'algorithme décale certaines zones occupées mémoire pour créer un espace contigu assez grand, le coût du redimensionnement peut être très important (et dépend de la taille de la mémoire disponible, et non plus de celle du tableau).

3. **Insertion et suppression coûteuses**. Enfin, si on veut insérer un élément dans un tableau à un indice i (en supprimant son dernier élément), il faut décaler tous les éléments à un indice $j > i$, ce qui nécessite à nouveau, dans le cas le pire, $O(n)$ affectations. De même, pour supprimer un élément d'un tableau, il faut décaler d'une case vers la gauche tous les éléments d'indice supérieur à l'indice de l'élément à supprimer, ce qui nécessite $O(n)$ affectations dans le cas le pire.

Tableaux : résumé

- L'avantage des tableaux est l'accès ***en temps constant*** à n'importe quel élément dont on connaît la position : le nombre d'opérations pour accéder à un élément ne dépend pas de la taille du tableau.
- Un inconvénient des tableaux est qu'on ne peut pas toujours les redimensionner, et même si on le peut, un redimensionnement peut induire une ***recopie complète*** de tout le tableau. Le nombre d'affectations nécessaires est, dans ce cas, proportionnel au nombre d'éléments du tableau.
- Un second inconvénient est que l'insertion dans un tableau nécessite de décaler tous les éléments à la droite de celui qu'on veut insérer, ce qui demande à nouveau une ***recopie***. Dans le cas le pire, si on insère un élément en première place, ce décalage demande un nombre d'affectations proportionnel au nombre d'éléments du tableau.

2 La structure de donnée de liste

Une autre structure de données dite « linéaire » qui permet de représenter en mémoire des multi-ensembles est la *liste*. Au lieu de représenter un multi-ensemble sous la forme d'un tableau, c'est-à-dire par une suite de cases *consécutives* en mémoire, nous utilisons l'idée suivante : chaque élément est placé dans une structure de deux champs, le premier contenant l'élément à mémoriser, et le second contenant l'adresse de la structure suivante. Le dernier élément de la liste sera associé à une valeur spéciale, notée traditionnellement `NULL`. La liste elle-même peut être représentée par un pointeur sur le premier élément de la liste, appelé tête de la liste, ou `head` (si elle est vide, la liste sera représentée par la valeur `NULL`). Du point de vue de la programmation, en langage C par exemple, la valeur `NULL` est une adresse invalide, ce qui permet de la différencier des pointeurs de la liste, et ainsi de détecter la fin de la liste. La Figure 3.5 représente le même multi-ensemble que précédemment, $S = \{\!\{15, 17, 42, 42, 45, 84, 99\}\!\}$, les éléments apparaissant dans cet ordre.

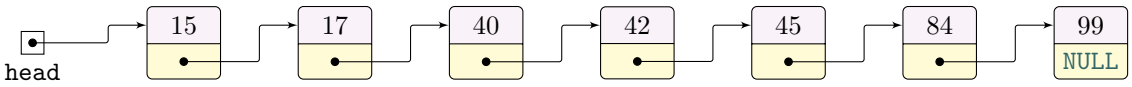


FIG. 3.5 : Une liste simplement chaînée d'entiers

Remarque. 3.5 Comme dans le cas des tableaux, une liste fournit implicitement un ordre sur les éléments (voir Remarque 1).

À nouveau, ce choix d'implantation a des avantages et des inconvénients.

1. Un inconvénient est que contrairement au cas des tableaux, l'accès à un élément n'est pas direct : accéder à un élément nécessite de « suivre » la chaîne de pointeurs, ce qui, à partir du début de la liste, a un coût $O(n)$ dans le cas le pire (où n est la taille de la liste).

2. Au contraire des tableaux, les opérations d'insertion en tête et de suppression peuvent être implantées en $O(1)$. En particulier, le coût de l'ajout de k éléments en tête d'une liste de taille n a un coût $O(k)$, qui ne dépend pas de n.

La Figure 3.6 montre les $O(1)$ étapes de suppression d'un élément de la liste, en supposant qu'il a déjà été recherché (rappel : la recherche elle-même a une complexité $O(n)$, où n est la taille de la liste). On suppose donc l'élément à supprimer pointé par `x`, et on appelle `next` le champ de la structure pointant sur la cellule suivante de la liste.

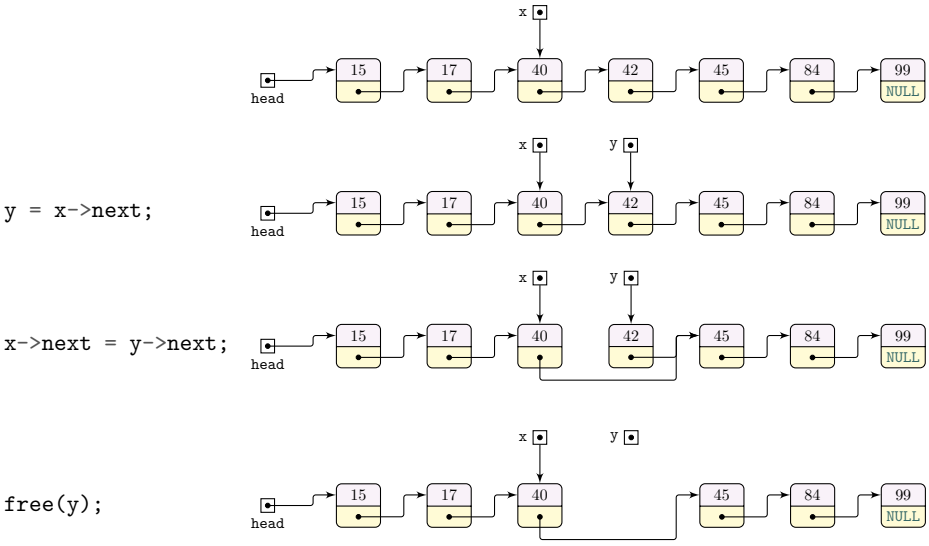


FIG. 3.6 : Suppression dans une liste en temps constant

Listes : résumé

- L'inconvénient principal des listes est l'accès d'un élément en temps $O(n)$ dans le cas le pire, si la liste est de taille n, même si on sait que l'élément est le $i^{\text{ème}}$ dans la liste.
- Un avantage est qu'on peut ***ajouter/supprimer*** un élément en tête, en temps $O(1)$.
- De façon plus générale, si on dispose d'un pointeur sur un élément de la liste, on peut ***insérer*** ou ***supprimer*** un élément en temps $O(1)$, sans ***sans recopie*** des éléments présents.

2.1 Les listes en OCaml

Tout comme dans la plupart des langages, on pourrait définir un type liste en langage OCaml. La définition peut se faire de la façon suivante, si on veut définir les listes d'entiers.

```
type int_list =
  | Empty
  | Cell of int * int_list
```

C'est une définition de type récursive. Elle exprime qu'une liste est :

- soit la liste vide, qu'on choisit de représenter par la valeur `Empty`.
- soit composée d'un couple (entier, liste).

Par exemple, la liste de la figure 3.5 s'écrirait

```
Cell (15,
      Cell (17,
            Cell (40,
                  Cell (42,
                        Cell(45,
                             (Cell(84,
                                   Cell(99, Empty)))))))
```

Remarque. 3.6 Les identificateurs permettant de construire les types, ici `Empty` et `Cell`, doivent commencer par une majuscule (au contraire du nom du type, `int_list`, qui doit commencer par une minuscule).

Une remarque importante est que définir une liste de flottants, ou une liste de chaînes de caractères, se ferait de façon analogue. Comme de nombreux algorithmes sur les listes ne dépendent pas du type des éléments de la liste, il serait intéressant d'avoir une notion de liste dont le type des éléments est un paramètre. Il est possible d'écrire une telle définition en OCaml. Dans ce cas, il faut utiliser un paramètre de type, commençant par une apostrophe, par exemple `'a`, ou `'foo`. Par exemple :

```
type 'foo list =
  | Empty
  | Cell of 'foo * 'foo list
```

OCaml déterminera le type d'une liste en fonction de celui de ces éléments. Par exemple :

```
# Empty;;
- : 'a list = Empty
# Cell(1, Empty);;
- : int list = Cell (1, Empty)
# Cell(1.5, Empty);;
- : float list = Cell (1.5, Empty)
```

Enfin, il est en fait inutile d'écrire notre propre type, car Ocaml fournit déjà un type `'a list`, ainsi que des fonctions et opérateurs. De ce fait,

- Plutôt que la notation lourde ci-dessus pour écrire la liste de la Figure 3.5, on écrit simplement `[15;17;40;42;45;84;99]`. Attention, les éléments sont séparés par des « ; » et non des « , ».
- La liste vide se note donc `[]`.
- On peut ajouter un élément en tête de liste avec l'opérateur `::`. Par exemple, `1::[2;3]` est la liste `[1;2;3]`, qu'on peut aussi obtenir par `1::2::3::[]`.

- On peut concaténer deux listes grâce à l'opérateur `@`. Par exemple, `[1;2;3] @ [4;5]` est la liste `[1;2;3;4;5]`.
- Plusieurs fonctions utiles sont disponibles dans la bibliothèque : voir https ://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html. Il est possible de visualiser leur code à l'aide de l'utilitaire `ocamlbrowser`.

Chapitre 4

Arbres et arbres de recherche

Les tableaux et les listes permettent de représenter des multi-ensembles de deux manières différentes, et chacune, du fait de son organisation en mémoire, possède ses propres propriétés algorithmiques. La table suivante suppose que les éléments des multi-ensembles peuvent être ordonnés. Les éléments d'un multi-ensemble peuvent donc être triés par ordre croissant (d'où les colonnes *Tableau trié* et *Liste triée*). Dans cette table, n désigne le nombre d'éléments du tableau ou de la liste représentant le multi-ensemble. La table récapitule les propriétés de ces deux structures vues dans le cours d'algorithmique du 1$^{\text{er}}$ semestre de L2 Informatique et dans le chapitre précédent.

	Tableau	**Tableau trié**	**Liste**	**Liste triée**
Recherche d'un élément	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$
Recherche du minimum	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Insertion d'un élément	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Suppression d'un élément	$O(n)$	$O(n)$	$O(n)$	$O(n)$

TAB. 4.1 : Résumé des complexités de certaines tâches sur tableaux et listes

Un avantage du tableau trié est qu'il permet la recherche d'un élément en temps $O(\log n)$, en utilisant un algorithme dichotomique, alors que celui des listes est l'insertion ou la suppression d'un élément en temps constant si on dispose d'un pointeur sur l'élément avant l'emplacement où on veut insérer (voir le Chapitre 3). L'insertion d'un élément dans un tableau non nécessairement trié peut se faire en $O(1)$, mais cela nécessite de mémoriser une information supplémentaire. On peut par exemple mémoriser l'indice du premier emplacement libre, en maintenant les emplacements libres uniquement en fin de tableau. En contrepartie, la suppression d'un élément nécessite un décalage, pour maintenir les emplacements libres en fin de tableau. Enfin, la complexité $O(n)$ de la suppression d'un élément dans une liste provient de la recherche de cet élément. Une fois cette recherche effectuée, l'insertion et la suppression ne nécessitent plus que $O(1)$ opérations élémentaires, comme on l'a vu au Chapitre 3. De même, l'insertion dans une liste dans laquelle on ne veut pas de répétitions d'éléments se ferait en $O(n)$ et non $O(1)$. En effet, avant d'insérer, il faudrait parcourir la liste pour rechercher si l'élément est ou non présent.

Remarque. 4.1 En pratique, la différence entre des complexités en $O(n)$ et en $O(\log n)$ est significative. Par exemple, si $n = 10^{15}$ (un *peta-octet*, c'est-à-dire un million de *giga-octets*), alors $\log_2(n)$ vaut environ seulement 35. Cela signifie qu'un algorithme de complexité $O(\log n)$ sera utilisable même pour de très grandes valeurs de n (en supposant raisonnable la constante multiplicative cachée dans la notation $O(\cdot)$). Pour de telles grandeurs, la difficulté réside en fait dans l'accès efficace aux données, qui peuvent être réparties sur plusieurs supports.

1 De la liste à l'arbre

On aimerait combiner l'efficacité de la recherche dans les tableaux triés avec la facilité d'insertion/suppression d'un élément déjà trouvé fournie par les listes. La Remarque 1 montre qu'à défaut de parvenir à des complexités en $O(1)$, obtenir des complexités en $O(\log n)$ est très intéressant.

Par ailleurs, garder les éléments à stocker contigus en mémoire pose des problèmes lorsqu'on veut insérer ou supprimer un élément, car cette structure de données force à faire des décalages. On part donc de l'idée suivante : la complexité $O(\log n)$ de recherche d'un élément dans un tableau trié provient de l'algorithme de dichotomie. Le problème dans une liste est que l'on n'a pas accès à l'élément central (supposons pour simplifier que cet élément existe, c'est-à-dire qu'on travaille sur une liste de taille impaire). Pour résoudre ce problème d'accès à l'élément central, on pourrait représenter une liste triée par :

- sa partie gauche, constituée des éléments à gauche de l'élément central,
- son élément central,
- sa partie droite, constituée des éléments à droite de l'élément central.

Par exemple, on pourrait représenter la liste de la Figure 3.5 de la façon suivante :

FIG. 4.1 : Découpage d'une liste triée en partie gauche, élément central, partie droite

Par ailleurs, pour implanter ce triplet, on peut utiliser la représentation suivante (notez que la nouvelle structure nécessaire, celle qui est pointée par `x`, utilise deux pointeurs) :

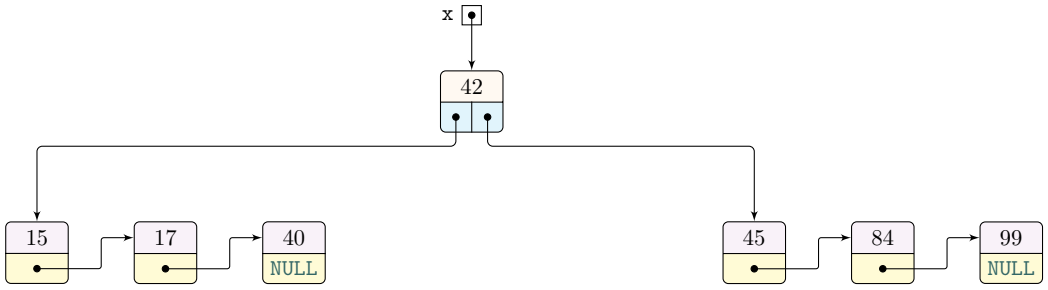


FIG. 4.2 : Structure de données pour découper une liste triée élément central, parties gauche et droite

Cette idée permet bien d'accélérer la recherche dans une liste triée. En effet, on a maintenant accès à l'élément central, et on peut donc commencer à appliquer la méthode dichotomique : si l'élément cherché est l'élément central, il est trouvé. Sinon, s'il est inférieur à l'élément central, on le cherche dans la partie gauche, et sinon, dans la partie droite. On a donc divisé l'espace de recherche par 2.

Il est naturel de continuer cette idée et de subdiviser à nouveau les sous-listes gauche et droite. Sur le même exemple, on obtiendrait :

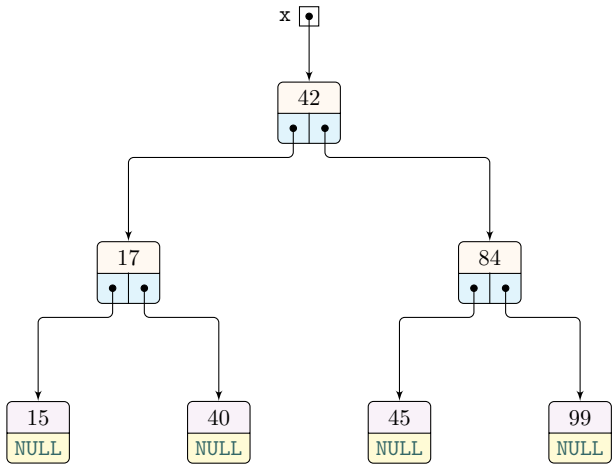


FIG. 4.3 : Arbre découpant récursivement une liste

On obtient un *arbre binaire* : chaque nœud de l'arbre a au maximum 2 fils. De plus, comme on est parti d'une liste triée, on peut vérifier que l'arbre a la propriété suivante : pour tout nœud z de l'arbre,

- les étiquettes des nœuds présentes dans le sous-arbre gauche de z sont inférieures à l'étiquette de z.
- les étiquettes des nœuds présentes dans le sous-arbre droit de z sont supérieures ou égales à l'étiquette de z.

Ces arbres binaires, dont les nœuds sont étiquetés par des valeurs ayant cette propriété, s'appellent ***arbres binaires de recherche***. Cette propriété est importante, car elle guide la navigation vers un élément que l'on recherche, de la même façon que l'on guide la navigation dans l'algorithme de recherche dichotomique. Pour rechercher un élément, la longueur de la navigation dans le cas le pire est la ***hauteur de l'arbre*** (c'est-à-dire, pour rappel, la ***longueur d'une des branches les plus longues de l'arbre***). Par convention, la hauteur de l'arbre vide est -1, et la longueur d'une branche est son nombre d'***arêtes*** reliant les nœuds de la branche. Par exemple, la hauteur de l'arbre de la Figure 4.3 est 2.

Si on part d'une liste triée de longueur $n = 2^k - 1$ et que l'on construit l'arbre comme dans l'exemple ci-dessus, on obtiendrait un arbre dont toutes les branches auraient la même hauteur, $k - 1$ (l'exemple est le cas où $k = 3$, $n = 7$). Autrement dit, le nombre de choix à effectuer serait $k - 1 = \log_2(n + 1) - 1$.

2 Arbres binaires de recherche

Rappelons que nos objectifs sont les suivants. On veut :

- une structure de donnée permettant de représenter des multi-ensembles, et
- qui permet d'effectuer les opérations suivantes de façon efficace :
  - rechercher un élément dans le multi-ensemble,
  - ajouter un élément,
  - supprimer un élément.

On va organiser les éléments d'un multi-ensemble à représenter selon l'idée expliquée en Section 1. Sur l'exemple présenté dans cette section, nous avons ignoré deux détails :

1. Une valeur peut être présente en plusieurs exemplaires dans le multi-ensemble à représenter. Cela pose un problème potentiel, puisqu'on voudrait qu'en tout nœud de l'arbre, les valeurs présentes dans le sous-arbre gauche soient inférieures à la valeur du nœud, et celles présentes dans le sous-arbre droit soient supérieures. Pour gérer le cas de valeurs en *plusieurs exemplaires* et diminuer le nombre de nœuds des arbres construits, on procède ainsi : au lieu de stocker individuellement chaque valeur,

on stocke des *couples* (x, n) où x est une valeur présente dans le multi-ensemble, et $n \geqslant 1$ désigne le nombre de fois où x apparaît dans l'ensemble. Par exemple, on représentera le multi-ensemble $\{\!\{15, 17, 42, 42, 45, 84, 99\}\!\}$ par l'ensemble $\{(15, 1), (17, 1), (42, 2), (45, 1), (84, 1), (99, 1)\}$, parce que 42 apparaît 2 fois. Ainsi, il n'y aura plus de duplication de valeurs. Cet ensemble pourrait être représenté par l'arbre de la Figure 4.4.

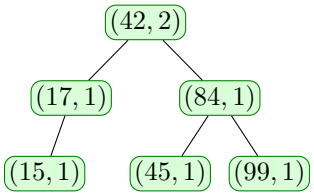


FIG. 4.4 : Un arbre représentant un multi-ensemble

On peut noter sur cet arbre que les 1$^{\text{ères}}$ coordonnées suivent la règle des arbres binaires de recherche : la 1$^{\text{ère}}$ coordonnée d'un nœud est (strictement) supérieure aux 1$^{\text{ère}}$ coordonnées des nœuds de son sous-arbre gauche et (strictement) inférieure aux 1$^{\text{ère}}$ coordonnées des nœuds de son sous-arbre droit.

Pour alléger les notations, on indiquera les secondes coordonnées, c'est-à-dire le nombre d'occurrences de l'élément, au dessus du nœud plutôt que dans un couple. L'arbre de la Figure 4.4 peut donc se dessiner comme en Figure 4.5.

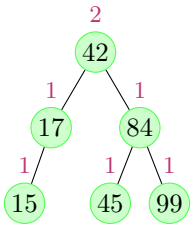


FIG. 4.5 : Une autre représentation de l'arbre de la Figure 4.4

2. Le second détail ignoré en exemple de la Section 1 est que, dans cet exemple, tout nœud avait 0 ou 2 fils. Autrement dit, l'arbre de l'exemple de la Section 1 est *complet* (il est même parfait). En général, on ne peut pas assurer cette propriété : par exemple le nœud 17 de la Figure 4.5 a un seul fils. Il faut donc retenir que certains nœuds des arbres binaires de recherche auront comme arité 1.

En résumé, l'arbre de la Figure 4.5 représente le multi-ensemble $\{\!\{15, 17, 42, 42, 45, 84, 99\}\!\}$. Les feuilles, représentées par un carré, peuvent aussi être vues comme l'arbre vide. Chaque nœud (nœud interne, ou feuille) contient une clé et un entier (noté au dessus) qui représente le nombre de fois que la clé apparaît dans le multi-ensemble.

Remarque. 4.2 Le point 1 montre en fait que l'on peut sans plus de travail représenter des *dictionnaires* en utilisant les arbres binaires de recherche. En effet, on s'apprête à mémoriser dans les nœuds des couples (x, m) où x n'apparaît pas plus d'une fois et m est un entier. De façon plus générale, on peut mémoriser des couples (k, v), celui où k est une clé et v la valeur associée. La différence principale est que v n'est pas nécessairement un entier (ça pourrait être une chaîne de caractères, ou une information structurée). La représentation d'un multi-ensemble est un cas particulier de dictionnaire : les clés sont les éléments du multi-ensemble et la valeur associée à une clé est le nombre de fois où l'élément apparaît.

On va maintenant définir formellement le type *arbre binaire de recherche*. Ce type permet de représenter des ensembles, des multi-ensembles, et même des dictionnaires (comme expliqué en Remarque 2). La Définition 1 utilise donc deux ensembles : le premier, K, correspond aux clés, et le second, V, aux valeurs associées aux clés. Suivant la Remarque 2, la représentation des multi-ensembles est un simple cas particulier : pour représenter un multi-ensemble d'éléments de E, on prendra $K = E$ (par exemple les entiers dans l'exemple de la Section 1), et $V = \mathbb{N}$ représentant le nombre d'occurrences de chaque valeur.

Definition 4.1 — Arbre binaire de recherche. Soit K un ensemble muni d'une relation d'ordre total $\leqslant$, et V un ensemble. On écrit $k_1 < k_2$ si $k_1 \leqslant k_2$ et $k_1 \neq k_2$. Un ***arbre binaire de recherche*** sur K, V est un arbre

- dont chaque nœud est étiqueté par un couple (k, v) avec $k \in K$ et $v \in V$,
- dont ***tout*** nœud interne vérifie la propriété suivante : si le nœud est étiqueté par (k, v), alors,
  - pour tout nœud de son sous-arbre gauche étiqueté par (k_ℓ, v_ℓ), on a $k_\ell < k$.
  - pour tout nœud de son sous-arbre droit étiqueté par (k_r, v_r), on a $k < k_r$.

Remarque. 4.3 La Définition 1 peut conduire au type OCaml suivant :

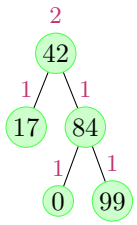
```
type ('k,'v) tree = Empty | Bin of ('k * 'v) * ('k,'v) tree * ('k,'v) tree
```

Cependant, on utilisera, sous Moodle, le type plus général suivant :

```
type 'a tree = Empty | Bin of 'a * 'a tree * 'a tree
```

Ce type est plus simple, et effectivement plus général, puisque le type `'a` représente n'importe quel type (même un type de la forme `('k * 'v)`).

Remarque. 4.4 — Important. On pourrait être tenté de simplifier la définition, en disant seulement qu'en chaque nœud interne, si on appelle x, x_ℓ, et x_r les clés de ce nœud, de son fils gauche éventuel et de son fils droit éventuel, on a $x_\ell < x < x_r$. Cette condition est nécessaire mais elle n'est pas suffisante. Par exemple, elle est vraie pour l'arbre suivant, mais ce n'est pas un arbre binaire de recherche.



Le problème est le nœud qui porte la valeur 0. En effet, comme il se trouve dans le sous-arbre droit de la racine qui porte 42, la valeur qu'il porte devrait être au moins 43, donc pas 0.

Dans le cas de la représentation d'un multi-ensemble, rappelons à nouveau que $V = \mathbb{N}$ permet de compter le nombre d'apparitions d'un élément dans le multi-ensemble : pour chaque nœud interne étiqueté par (x, m), l'entier m représente le nombre de fois que x apparaît dans le multi-ensemble représenté par l'arbre binaire. Il est appelé ***multiplicité*** de x. Par exemple, dans l'arbre de la Figure 4.5, la multiplicité de 42 est 2 et la multiplicité des autres nœuds est 1.

2.1 Recherche dans un ABR

Si un arbre est un ABR, il est simple de rechercher une clé x (pour retourner, par exemple, sa multiplicité). L'algorithme itératif est le suivant :

- Si l'arbre est vide, on retourne 0 (la clé est absente, c'est-à-dire « présente en 0 exemplaire »).
- Sinon, si la clé de la racine est x, on retourne la multiplicité de la racine.
- Sinon, si x est inférieur à la clé de la racine, on cherche récursivement dans le sous-arbre gauche.
- Sinon, x est supérieur à la clé de la racine, et on cherche récursivement dans le sous-arbre droit.

Lors d'un appel récursif, on passe de l'arbre de départ à l'un de ses sous-arbres : la hauteur de l'arbre dans lequel on recherche diminue donc d'au moins 1. Cet argument montre que la recherche a une complexité $O(h)$ dans le cas le pire, où h est la hauteur de l'arbre dans lequel on recherche la clé.

2.2 Insertion et suppression dans un ABR

L'insertion dans un arbre binaire de recherche est simple, parce qu'elle intervient au niveau des feuilles. On peut obtenir un algorithme en $O(h)$ où h est la hauteur de l'arbre dans lequel on insère. L'algorithme est similaire à celui d'insertion. Il faut faire attention, pour construire l'arbre résultat, de conserver la structure (à part la feuille qu'on ajoute). Par exemple, si on a détecter qu'on doit insérer dans le sous-arbre gauche, il ne faut pas se contenter de relancer récursivement l'insertion sur le sous-arbre gauche, en oubliant la racine et le sous-arbre droit (qui doivent faire partie du résultat).

La suppression est plus difficile. Pour supprimer une occurrence d'un élément x de multiplicité n dans un arbre binaire de recherche qui représente un multi-ensemble, on peut procéder ainsi. L'algorithme est décrit de façon itérative, mais il faudra le programmer de façon récursive.

- On recherche le nœud de clé x (qui doit exister si on veut supprimer une occurrence de x). Puis,
  - Si la multiplicité de x est strictement supérieure à 1, il suffit de la décrémenter.
  - Sinon, la multiplicité de x est 1. C'est plus difficile dans ce cas, où on doit produire un arbre ayant **un nœud de moins** que l'arbre d'origine. La difficulté est aussi que l'on veut garder la propriété que l'arbre obtenu reste un arbre binaire de recherche. Comme la suppression va affecter uniquement le sous-arbre enraciné au nœud à supprimer, on peut décrire l'algorithme en supposant que ce nœud, étiqueté $(x\,,1)$, se trouve à la racine. On distingue plusieurs cas :
    - Si les deux fils du nœud racine $(x\,,1)$ sont vides, le résultat est simplement l'arbre vide.
    - Si le nœud fils droit du nœud racine $(x\,,1)$ est vide, le résultat est le sous-arbre gauche.
    - Si le nœud fils gauche du nœud racine $(x\,,1)$ est vide, le résultat est le sous-arbre droit.
    - Sinon, le nœud racine $(x\,,1)$ a deux fils non vides. On calcule l'élément dont la clé est juste avant celle de x. C'est la clé maximale dans le sous-arbre gauche (elle se trouve en suivant les fils droits à partir de la racine du sous-arbre gauche). Appelons y cette clé et m sa multiplicité. Le résultat est l'arbre obtenu de l'arbre original en supprimant la clé $(y\,,m)$ et en étiquetant la racine par $(y\,,m)$ au lieu de $(x\,,1)$.

On peut à nouveau écrire cet algorithme pour que sa complexité soit $O(h)$ où h est la hauteur de l'arbre dans lequel on veut supprimer une clé.

Arbres binaires de recherche (ABR) : résumé

Qu'avons nous gagné par rapport aux listes et aux tableaux pour représenter un multi-ensemble ? On a en fait introduit deux points allant dans des directions orthogonales.

- D'une part, si un élément est présent en plusieurs exemplaires dans le multi-ensemble, on ne le représente qu'une fois, associé à sa ***multiplicité***. Cela aurait très bien pu être fait de la même façon au niveau des listes ou des tableaux, en mémorisant des couples $(x\,,m)$ où x est un élément et $m \geqslant 1$ est sa multiplicité.

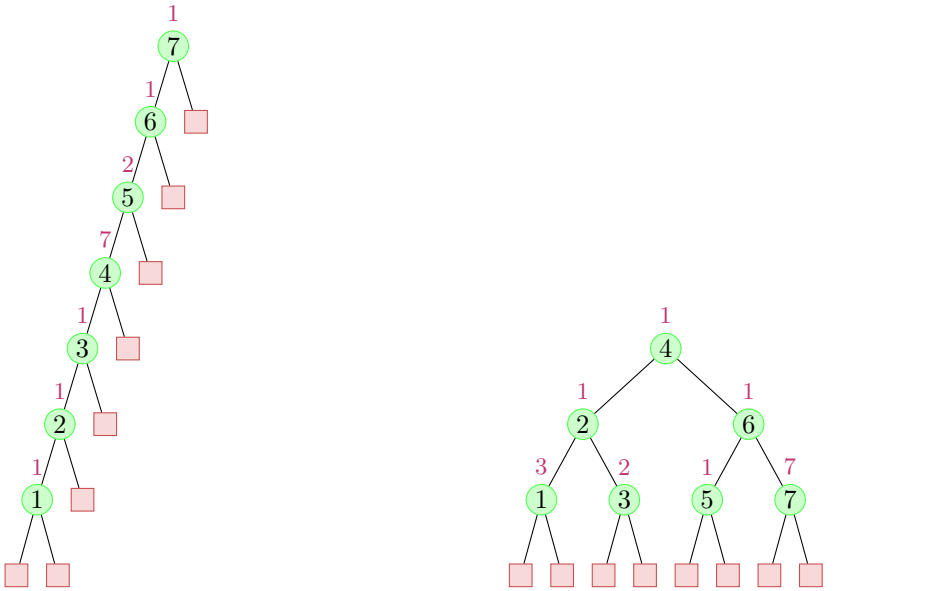
- L'autre point important est une modification plus profonde de la ***structure de donnée*** : arbres binaires de recherche au lieu de listes ou tableaux. Les opérations importantes
  - de recherche d'un élément,
  - d'insertion d'un élément,
  - de suppression d'un élément,
  - de recherche du minimum ou du maximum,

  se font dans un temps ***proportionnel à la hauteur de l'arbre*** qui représente le multi-ensemble, dans le cas le pire. Rappelons que la ***hauteur*** est la longueur de la plus longue branche. On compte cette longueur en nombre de d'***arêtes*** entre les nœuds de la branche, avec comme convention que la hauteur de l'arbre vide est -1. Voir les 2 exemples de la Figure 4.6.

Plus précisément, appelons n le nombre de nœuds internes d'un arbre binaire de recherche permettant de représenter un multi-ensemble (c'est-à-dire que n est le nombre d'éléments *différents* du multi-ensemble à représenter).

Appelons $h(t)$ la hauteur minimale d'un tel arbre de recherche, et n le nombre de clés différentes dans l'arbre, c'est-à-dire le nombre de nœuds de t. On a

- $h(t) \leqslant n - 1$: cette hauteur maximale est atteinte dans le cas d'un arbre « filiforme »,
- $h(t) \geqslant \log_2(n+1) - 1$: cette hauteur minimale est atteinte dans le cas d'un arbre parfait (c'est-à-dire, dont toutes les feuilles sont à la même profondeur, et tous les nœuds internes ont exactement 2 fils).



(a) Arbre filaire, $n = 7$, $h = 6 = n - 1$ (b) Arbre parfait, $n = 7$, $h = 2 = \log_2(n+1) - 1$

FIG. 4.6 : Deux arbres à 7 clés

Nous obtenons alors le tableau de complexité suivant.

	Tableau	**Tableau trié**	**Liste**	**Liste triée**	**ABR**
Recherche d'un élément	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(h(t))$
Recherche du minimum	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(h(t))$
Insertion d'un élément	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(h(t))$
Suppression d'un élément	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(h(t))$

Avec la remarque précédente reliant $h(t)$ à n, la complexité de tous les algorithmes (recherche, insertion, suppression) se situe donc dans le cas le pire, à une constante multiplicative près, entre $\log(n)$ et n.

Dans le chapitre suivant, nous verrons comment maintenir une hauteur $O(\log n)$, qui garantira cette complexité pour toutes les opérations dans le cas le pire.