Cours : Arbre binaire de recherche

Table des matières

1 Arbre binaire de recherche **1**
- 1.1 Définition . . . 1
- 1.2 Algorithmes élémentaires . . . 3
- 1.3 Insertion, suppression . . . 4
- 1.4 Réalisation d'un dictionnaire par un Abr . . . 8

2 Arbre auto-équilibré **9**

3 File de priorité **10**
- 3.1 Structure abstraite de file de priorité . . . 10
- 3.2 Tas binaire . . . 11
- 3.3 Tas impératif à l'aide d'un arbre implicite . . . 12
- 3.4 Opérations sur un tas binaire . . . 13
- 3.5 Tri par tas . . . 15

1 Arbre binaire de recherche

1.1 Définition

Les *arbres binaires de recherche* fournissent une réalisation naturelle et efficace des types abstraits Set et Dict. Nous nous concentrerons d'abord sur le type Set, et nous verrons ultérieurement que l'on peut facilement adapter au type Dict.

Dans toute cette section, $(A, \preceq)$ désigne un ensemble totalement ordonné.

Définition 1.1: Ensemble des étiquettes

On rappelle que l'ensemble $\mathcal{T}(A)$ des arbres binaires homogènes est défini par induction structurelle :
- $\perp \in \mathcal{T}(A)$.
- Quel que soit $g \in \mathcal{T}(A)$, $d \in \mathcal{T}(A)$ et $x \in A$, on a $(g, x, d) \in \mathcal{T}(A)$.

Si $t \in \mathcal{T}(A)$ est un arbre, on définit l'ensemble $\varphi(t)$ des étiquettes de t par :
- $\varphi(\perp) := \varnothing$
- $\forall g, d \in \mathcal{T}(A), \quad \forall x \in A, \quad \varphi(g, x, d) := \{x\} \cup \varphi(g) \cup \varphi(d)$

Définition 1.2: Arbre binaire de recherche

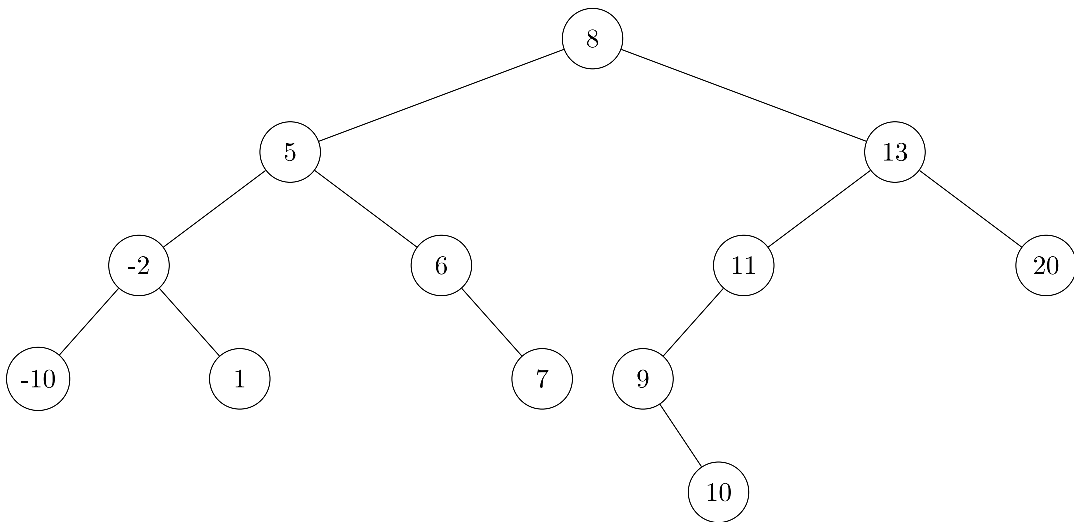
L'ensemble $\text{Abr}(A)$ des *arbres binaires de recherche* sur A est défini par induction structurelle
- $\perp \in \text{Abr}(A)$.
- Quel que soit $g \in \mathcal{T}(A)$, $d \in \mathcal{T}(A)$ et $x \in A$, alors $(g, x, d) \in \text{Abr}(A)$ lorsque
  - g et d sont dans $\text{Abr}(A)$.
  - $\max(\varphi(g)) < x < \min(\varphi(d))$.

Remarques

- ⇨ Dans un tel arbre, une feuille est un nœud de la forme $(\perp, x, \perp)$. Quand on représente graphiquement l'arbre, on omet systématiquement les fils vides.
- ⇨ On prend la convention $\max \varnothing < x < \min \varnothing$ pour tout $x \in A$. Si A est une partie de $\mathbb{R}$, on prend par exemple $\min \varnothing = +\infty$ et $\max \varnothing = -\infty$.
- ⇨ Le fait qu'on ait choisi des inégalités strictes dans la définition d'un Abr garantit que les étiquettes sont deux à deux distinctes. On manipule donc des ensembles et non des multi-ensembles.

Exemple

⇨ *Arbre binaire de recherche*

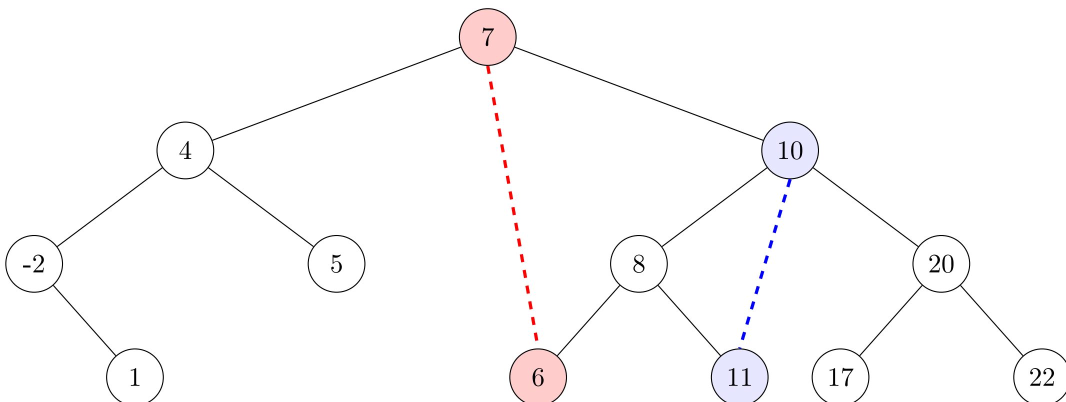


Un arbre binaire de recherche pour l'ensemble $\{-10, -2, 1, 5, 6, 7, 8, 9, 10, 11, 13, 20\}$

Attention, la propriété d'Abr n'est pas « locale » : il ne suffit pas de vérifier que chaque nœud a une étiquette plus grande que celle de son fils gauche et plus petite que celle de son fils droit.

Exemple

⇨ *Violation de la propriété d'Abr* : L'arbre ci-dessous *n'est pas* un Abr. Il y a deux violations de la propriété qui ont été mises en évidence. Si l'on se limitait à comparer les étiquettes des pères avec celles de leurs fils, on ne détecterait pas de problème.



Propriété d'Abr non vérifiée

Pour un ensemble donné d'étiquettes, de nombreux Abr sont possibles et leur forme peut varier énormément.

Exercice 1

⇨ Donner deux Abr de hauteur maximale et deux de hauteur minimale pour l'ensemble $\{1, 2, 3, 4, 5, 6\}$.

Proposition 1.3

Soit t un arbre binaire homogène. Alors t est un Abr si et seulement si infixe(t) est triée par ordre strictement croissant, où infixe(t) désigne la liste de étiquettes de t dans l'ordre infixe.

Remarques

⇨ À chaque niveau de l'arbre, les étiquettes lues de gauche à droite forment une suite croissante. En fait, si l'on représente « proprement » l'arbre, alors la lecture des étiquettes de gauche à droite, indépendamment du niveau, correspond à un parcours infixe et donne donc une suite croissante.

⇨ Le minimum est obtenu en descendant à gauche depuis la racine jusqu'à arriver à la fin de la branche. De même pour le maximum en descendant à droite.

1.2 Algorithmes élémentaires

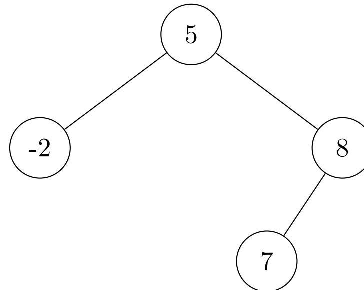
1.2.1 Représentation en OCaml

Pour représenter en OCaml un arbre binaire dans lequel un nœud peut n'avoir qu'un seul fils, on utilise le type suivant :

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

Exercices 2

⇨ 1. Comment représente-t-on une feuille avec ce type ?
2. Définir en OCaml l'arbre suivant :



3. Dessiner l'arbre défini par :

```
N (N (V, 0, V),
   1,
   N (N (V, 2, N (V, 4, V)),
      5,
      N (V, 6, V)))
```

⇨ On souhaite écrire une fonction déterminant si une arbre t est un arbre binaire de recherche. Écrire une fonction `isAbr : 'a abr -> bool * 'a option * 'a option` prenant en entrée un arbre binaire de recherche et renvoyant `(b, min, max)` où

- `b` vaut `true` si t est un ABR et `false` sinon.
- `min` est la valeur du plus petit élément de t si t est un ABR, et vaut `None` lorsque t est vide. Si t n'est pas un ABR, cette valeur n'est pas spécifiée.
- `max` est la valeur du plus grand élément t si t est un ABR, et vaut `None` lorsque t est vide. Si t n'est pas un ABR, cette valeur n'est pas spécifiée.

1.2.2 Représentation en C

En C, on représentera un nœud d'un ABR par une `struct`.

```
struct bst {
    int key;
    struct bst *left;
    struct bst *right;
};

typedef struct bst bst;
```

L'arbre vide est représenté par un pointeur nul, et la fonction suivante permet donc de créer une nouvelle feuille :

```
bst *bst_newNode(int x) {
    bst *node = malloc(sizeof(bst));
    node->key = x;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

Dans la suite, on supposera toujours que les nœuds d'un Abr ont été créés par des appels à cette fonction `bst_newNode`.

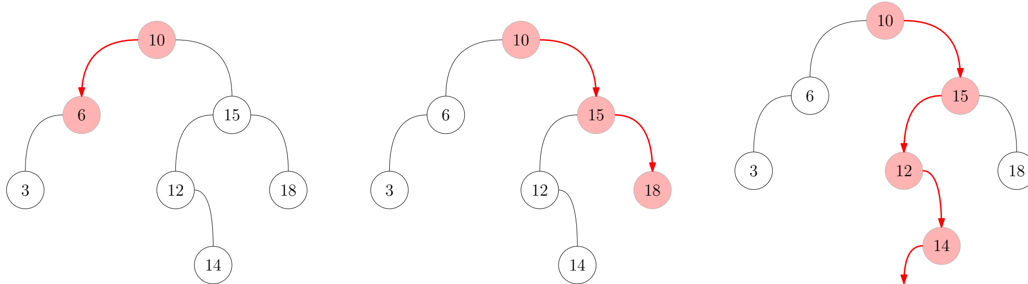
Exercice 3

⇨ Écrire une fonction `bst_delete` qui libère la mémoire occupée par un Abr.

```
void bst_delete(bst *t);
```

1.2.3 Recherche

Pour rechercher un élément dans un Abr, on descend le long d'une branche depuis la racine jusqu'à trouver l'élément ou arriver à un sous-arbre vide. Tout l'intérêt de la propriété d'Abr est de permettre de n'explorer *qu'une seule branche.*



Recherche de 6, de 18 et de 13 dans un Abr

On notera la grande similarité entre la recherche dans un Abr et la *recherche dichotomique dans un tableau trié.*

Exercices 4

⇨ 1. Écrire une fonction OCaml `appartient : 'a abr -> 'a -> bool`.
2. Montrer la correction de cette fonction.

⇨ Écrire une fonction `minimum : 'a abr -> 'a` renvoyant la plus petite étiquette d'un Abr. On lèvera une exception si l'arbre est vide.

⇨ Écrire des version C de ces deux fonctions. Pour chaque fonction, on écrira une version récursive, puis une version itérative. Quels sont les avanatages et les inconvénients des deux versions ?

```
bool bst_isMember(bst *t, int x);
int bst_minimum(bst *t);
```

1.3 Insertion, suppression

1.3.1 Insertion

Les insertions se font toujours en bas de l'arbre : l'élément ajouté sera une feuille. On descend donc le long d'une branche, comme lors d'une recherche, de manière à insérer l'élément au bon endroit. Si l'élément est déjà présent, il sera nécessairement rencontré lors du parcours : dans ce cas, on renvoie l'arbre inchangé. Tout l'intérêt d'un Abr par rapport à un tableau trié vient de la possibilité de réaliser efficacement des insertions et des suppressions.

Exercice 5

⇨ 1. Écrire une fonction `insere : 'a abr -> 'a -> 'a abr` tel que `insere a x` renvoie un Abr contenant les mêmes éléments que `a`, ainsi que l'élément `x`. Si `x` était déjà présent dans `a`, on renverra l'arbre original.
2. Quelle modification faut-il apporter au code précédent si l'on souhaite qu'un élément puisse avoir plusieurs occurrences comme dans le cas d'une structure MultiSet ?

En C, même si le principe reste le même, il va falloir procéder un peu différemment : la structure d'arbre que nous utilisons est mutable. La fonction `bst_insert` peut être programmée ainsi :

```c
bst *bst_insert(bst *t, int x) {
    if (t == NULL) {
        return bst_newNode(x);
    }
    if (t->key < x) {
        t->right = bst_insert(t->right, x);
    } else if (t->key > x) {
        t->left = bst_insert(t->left, x);
    }
    return t;
}
```

Exercices 6

⇨ 1. Ré-écrire la fonction `bst_insert` de manière itérative.
  2. Est-il encore nécessaire que cette fonction renvoie un `bst*`, ou peut-elle renvoyer `void` ? Si oui, pourquoi ?

⇨ 1. Écrire une fonction `construit : 'a list -> 'a abr` renvoyant un Abr dont les étiquettes sont les éléments de la liste passée en argument.
  2. Cette fonction est-elle récursive terminale ? Peut-on facilement en écrire une version terminale ?
     Cette fonction appelle `insere` *qui n'est pas récursive terminale, mais ce n'est pas la question : on s'intéresse juste à la fonction* `construit` *elle-même.*
  3. Préciser l'ordre dans lequel les insertions sont faites, dans les deux versions.
  4. Prévoir, puis vérifier, l'arbre obtenu à partir des listes suivantes, en insérant les éléments dans leur ordre d'apparition dans la liste :

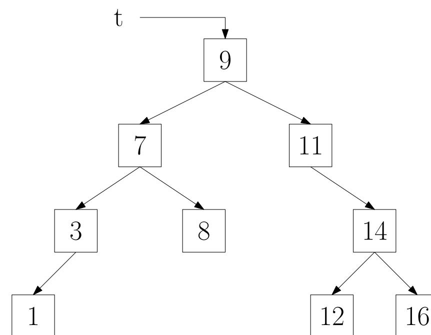
     (a) `[1; 4; 2; 3; 0; 7; 5]` (b) `[0; 1; 2; 3; 4; 5]` (c) `[5; 4; 3; 2; 1; 0]`

⇨ Écrire une fonction `bst_build`, prenant en entrée un tableau d'objets de type `T` et sa taille, et renvoyant l'Abr obtenu en insérant les éléments du tableau, dans l'ordre, dans un arbre initialement vide.

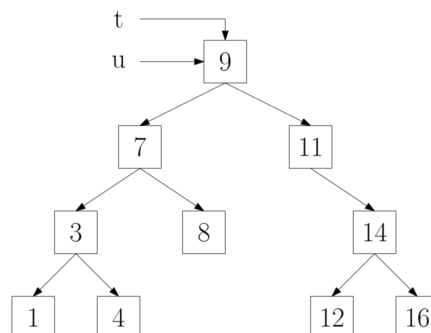
```c
bst *bst_build(int a[], int n);
```

1.3.2 Partage

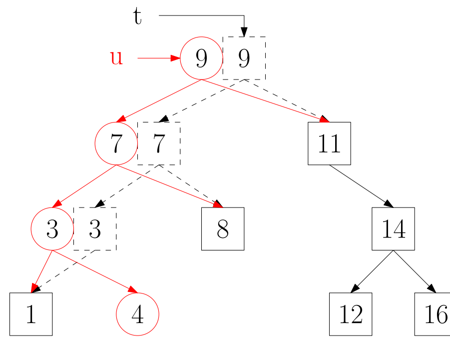
Il faut bien comprendre que les structures que nous avons définies en OCaml et en C, bien qu'elles aient essentiellement la même représentation en mémoire, sont complètement différentes. En effet, on a une structure immuable en OCaml et mutable en C : dans un cas, l'insertion d'un élément dans un arbre crée un nouvel arbre, sans modifier l'arbre initial, alors que dans l'autre cas on a une mutation.



Un Abr t (en C ou en OCaml).



Insertion en C : résultat de l'opération `bst *u = bst_insert(t, 4)`. Les pointeurs `u` et `t` sont égaux.



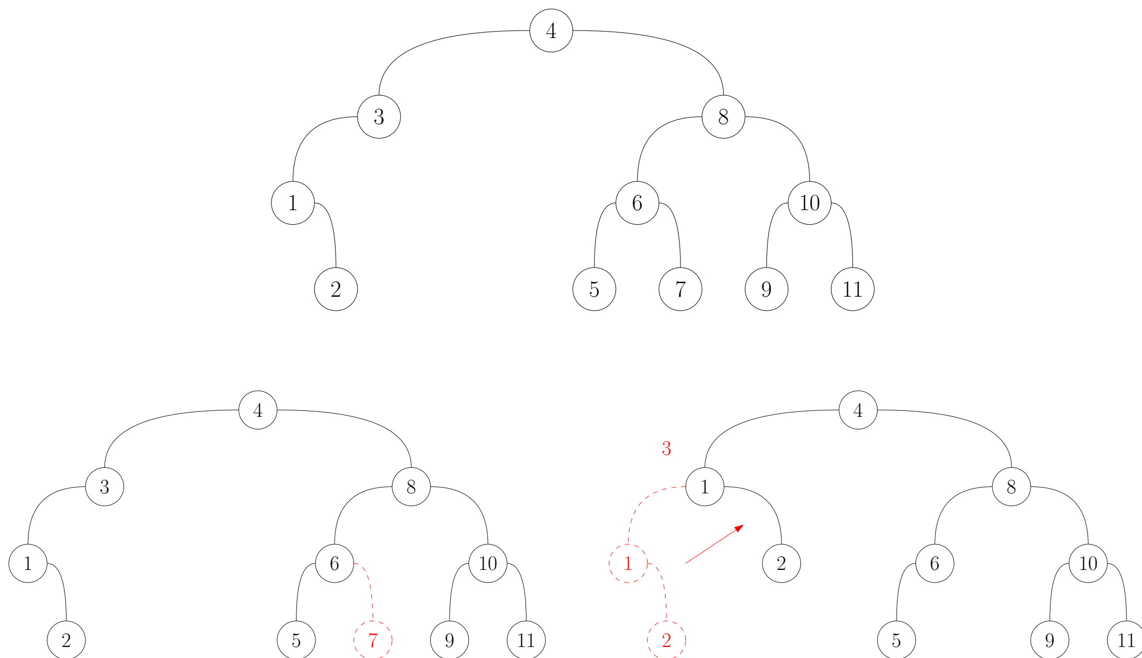
Insertion en OCaml : résultat de l'opération `let u = insere t 4`. La partie pointillée appartient à `t` mais pas à `u`, la partie rouge à `u` mais pas à `t`, le reste est partagé.

Il est tout-à-fait possible de définir une structure d'Abr mutable en OCaml, mais l'intérêt est assez limité. Le code est nettement moins élégant, en particulier. En revanche, définir et utiliser de manière sûre une structure immuable d'arbre en C est réellement problématique : on en vient rapidement à écrire une version minimale d'un *garbage collector*.

1.3.3 Suppression

Étant donnés un Abr t représentant un ensemble $\varphi(t)$ et un élément x, on souhaite renvoyer un Abr t' représentant $\varphi(t) \setminus \{x\}$. Il y a trois cas simples :

- Si l'élément à supprimer n'apparaît pas dans l'arbre, il n'y a rien à faire.
- Si l'élément est une feuille, il suffit de la supprimer.
- Si l'élément n'a qu'un seul fils non vide, il suffit de supprimer l'élément et de remonter son fils.



Suppression de 7 (feuille) Suppression de 3 (un seul fils)

Cas simples pour la suppression

On considère le code OCaml suivant, que l'on va compléter :

```
let rec supprime (t : 'a abr) (elt : 'a) : 'a abr =
  match arbre with
  | V ->
  | N (V, x, d) when x = elt ->
  | N (g, x, V) when x = elt ->
  | N (g, x, d) when elt < x ->
  | N (g, x, d) when elt > x ->
  | N (g, x, d) ->
```

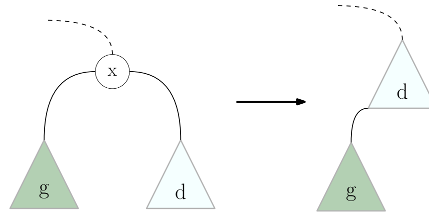
Exercice 7

⇨ *Suppression d'un élément, cas simples* : Compléter les lignes 3 à 7 de la fonction `supprime`.

Dans le cas où le nœud à supprimer possède deux fils, c'est un peu plus compliqué. Une possibilité assez simple, mais *non satisfaisante* est la suivante :

— On remonte le sous-arbre droit d de l'élément x à supprimer.

— Le sous-arbre gauche g de x devient le fils gauche du minimum de d.



Suppression d'un élément, méthode naïve.

Exercice 8

⇨ Justifier la correction de la méthode ci-dessus, et expliquer pourquoi elle n'est pas satisfaisante.

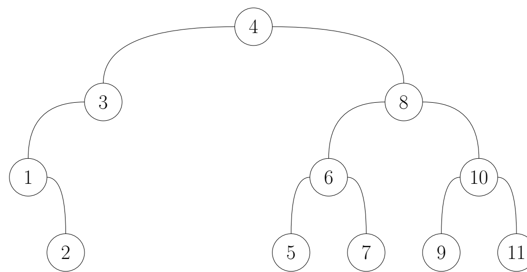
Pour faire mieux, il faut d'abord remarquer que le minimum d'un ABR ne peut avoir de fils gauche. Par conséquent, la suppression du minimum rentre toujours dans l'un des cas simples traités plus haut. On peut donc procéder ainsi :

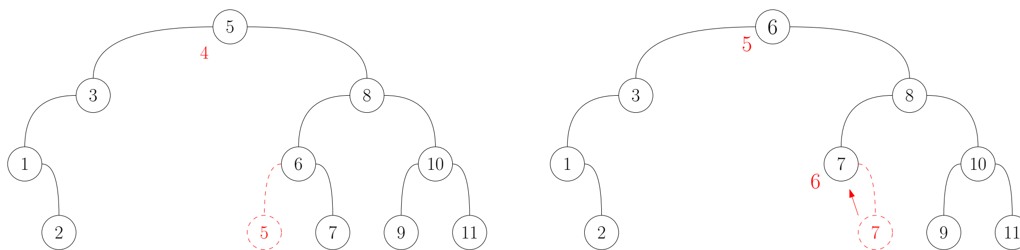
— On considère le sous-arbre droit d de l'élément x à supprimer ; d est non vide, sinon on serait dans l'un des cas simples.

— On récupère le minimum m de d.

— On calcule d', résultat de la suppression de m dans d.

— On renvoie l'arbre (g, m, d') ; autrement dit, on remplace x par le minimum de d, et l'on supprime ce minimum dans d.





Suppression d'un élément ayant deux fils

Exercices 9

⇨ *Suppression d'un élément, cas général*

1. Écrire une fonction `supprime_min : 'a abr -> 'a abr` qui prend un ABR t supposé non vide et renvoie l'ABR t' obtenu en supprimant le minimum de t.
2. Compléter la ligne 8 de la fonction `supprime`.
3. Supposons que l'on dispose d'un ABR `a` ainsi que d'un élément `x` de `a`, et que l'on définisse :

```
let b = insere (supprime a x) x
```

A-t-on, en général, `a = b` ?

⇨ *Suppression d'un élément, langage C*

1. Écrire une fonction `bst_deleteMin` supprimant le minimum d'un ABR supposé non vide.

```
bst *bst_deleteMin(bst *t);
```

2. Écrire une fonction `bst_deleteElement` supprimant un élément d'un ABR. Si l'élément n'apparaît pas dans l'ABR, ce dernier sera renvoyé inchangé.

```
bst *bst_deleteElement(bst *t, int x);
```

1.3.4 Complexité des opérations

Définition 1.4: Hauteur d'un ABR

La *hauteur* $h(t)$ d'un ABR t est définie par induction structurelle
- $h(\perp) = -1$;
- $h(g, x, d) = 1 + \max(h(g), h(d))$

Remarques

⇨ Fixer la hauteur de l'arbre vide à -1 plutôt qu'à 0 est une convention. Elle n'est pas universelle, mais c'est celle du programme.

⇨ Autrement dit, la hauteur d'un ABR est le nombre maximal d'arêtes traversées le long d'un chemin reliant la racine à une feuille de l'arbre ; où une feuille est un nœud de la forme `N (V, x, V)`.

Proposition 1.5

Soit t un ABR de hauteur h. Une opération de recherche, d'insertion ou de suppression dans t effectue au plus $h + 1$ comparaisons entre clés.

Proposition 1.6

Soit t un ABR de hauteur h. En supposant que la comparaison de deux clés peut se faire en temps constant, les opérations de recherche, d'insertion et de suppression peuvent s'effectuer en temps $\mathrm{O}(h)$. Si le nombre d'étiquettes de t est $n \geqslant 1$, on a

$$\lfloor \log_2 n \rfloor \leqslant h \leqslant n - 1$$

Remarques

⇨ Si les clés sont des entiers ou des flottants, les comparaisons sont bien en temps constant. Si on a par exemple des chaînes de caractères, c'est plus compliqué puisque la comparaison prend un temps proportionnel à la longueur du plus grand préfixe commun : en pratique, on utilise des variantes spécialisées dans ce cas. On peut bien sûr toujours affirmer qu'on fera au plus h comparaisons.

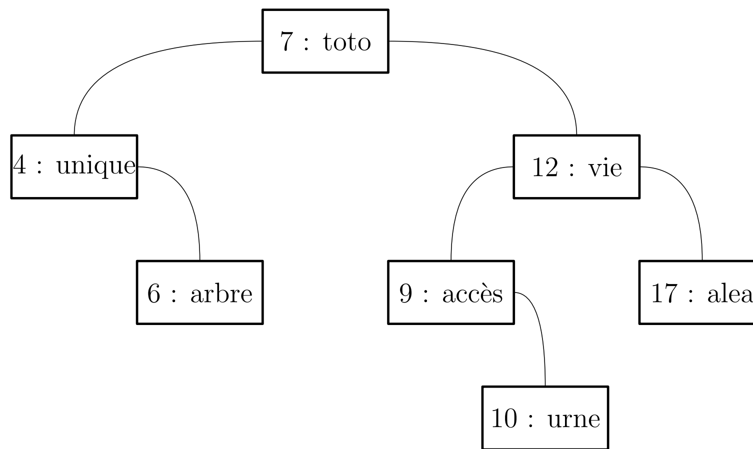
⇨ Le pire cas est celui d'un *peigne* pour lequel les performances sont catastrophiques : complexité temporelle en $\Theta(n)$.

⇨ En insérant les étiquettes par ordre croissant, on obtient justement un peigne.

⇨ Si l'arbre est le résultat d'une suite aléatoire d'insertions et de suppressions, on aura en moyenne du $\Theta(\log n)$.

1.4 Réalisation d'un dictionnaire par un ABR

Les ABR permettent de réaliser de manière fonctionnelle, c'est-à-dire persistante la structure de dictionnaire, à condition que les clés soient choisies dans un ensemble totalement ordonné. Pour ce faire, on stocke dans les nœuds les couples (clé, valeur) et toutes les comparaisons se font uniquement sur les clés.



Exemple de `(int, string) dict`.

On remarquera que dans l'exemple ci-dessus, la propriété d'ordre des ABR s'applique uniquement aux clés. Les valeurs n'ont d'ailleurs pas de raison d'être choisies dans un type ordonné.

Exercice 10

⇨ *Opérations élémentaires sur les dictionnaires fonctionnels* : On considère le type suivant :

```
type ('k, 'v) dict =
  | Vide
  | Noeud of 'k * 'v * ('k, 'v) dict * ('k, 'v) dict
```

Écrire les fonctions suivantes :

```
get : 'k -> ('k, 'v) dict -> 'v option
set : 'k -> 'v -> ('k, 'v) dict -> ('k, 'v) dict
```

On obtient ainsi des complexités en $O(h)$ pour `get`, `set` et `remove`, ce qui est satisfaisant si l'arbre est raisonnablement équilibré car on a alors $h = O(\log n)$.

Remarque

⇨ Pour un `(string, int) dict`, un dictionnaire dont les *clés* sont des chaînes de caractères, il est possible d'utiliser un ABR (l'ensemble des chaînes de caractères est totalement ordonné par l'ordre lexicographique), mais c'est rarement la bonne solution. On préférera utiliser une variante de *trie* (*cf* en travaux pratiques) ou une table de hachage. En effet, la comparaison de deux chaînes de caractères est coûteuse : elle prend un temps proportionnel à leur plus grand préfixe commun.

2 Arbre auto-équilibré

3 File de priorité

3.1 Structure abstraite de file de priorité

Une file de priorité est une structure de données contenant des couples (valeur, priorité) ; les valeurs peuvent être d'un type quelconque, mais les priorités doivent être choisies dans un type totalement ordonné (le plus souvent, ce sont des entiers ou des flottants).

Comme pour une pile ou une file, les deux opérations fondamentales sont l'ajout d'un élément – c'est-à-dire ici d'un couple (valeur, priorité) – et l'extraction du « prochain » élément. Dans une pile, ce « prochain » élément est celui qui a été inséré le plus récemment ; dans une file, c'est au contraire le premier à avoir été inséré ; dans une *file de priorité*, c'est celui ayant *la priorité la plus faible* , ou la plus grande priorité, si l'on choisit l'autre convention.

Un exemple d'application très naturelle des files de priorité est celui du *scheduler* d'un système d'exploitation. Sur un ordinateur actuel, de nombreux *processus* s'exécutent « en parallèle » ; certains de ces processus correspondent à des applications lancées par l'utilisateur, d'autres à des services tournant en arrière-plan. Chaque processus peut posséder plusieurs *fils d'exécution* (ou *threads*), mais cela ne change rien au problème. En réalité, un processeur (ou un cœur) ne peut exécuter qu'un seul processus à la fois, et le nombre de processus à exécuter sera systématiquement supérieur au nombre de processeurs : il faut donc gérer la pénurie de processeurs. Pour cela, on peut maintenir une file de priorité contenant tous les processus à exécuter. Ensuite, à chaque fois qu'un processeur se libère :

— On choisit la tâche la plus prioritaire parmi celles à exécuter, c'est-à-dire celle ayant la priorité p la plus basse, si l'on prend la convention usuelle.
— On l'exécute pendant un temps fixé, de l'ordre de la milli-seconde, sur le processeur libre.
— Si cette exécution a donné lieu à la création de nouveaux processus, ils sont ajoutés à la file de priorité.
— Dans tous les cas, sauf si le processus s'est terminé pendant la fenêtre d'exécution, il est remis dans la file de priorité. Cependant, on l'insère avec une priorité $p' > p$, c'est-à-dire qu'il est à présent moins prioritaire ; la valeur de $p' - p$ dépendra du temps d'exécution dont il vient de bénéficier et du type de processus. Par exemple, on souhaite qu'une application interactive dispose de plus de temps processeur qu'un processus de sauvegarde tournant en arrière-plan.

Opération	Type	Commentaire
Opérations caractéristiques		
`insert`	`'p -> 'v -> ('p, 'v) pq -> unit`	Ajout d'un élément (avec sa priorité)
`extract_min`	`('p, 'v) pq -> 'p * 'v`	Extraction de l'élément de priorité minimum (la file est modifiée)
Opérations complémentaires		
`get_min`	`('p, 'v) pq -> 'p * 'v`	Lecture de l'élément de priorité minimum (la file reste inchangée)
`create`	`unit -> ('p, 'v) pq`	Création d'une file vide
`card`	`('p, 'v) pq -> int`	Nombre d'éléments dans la file

Signature possible pour un type PRIORITYQUEUE impératif.

Remarques

⇨ Les opérations données correspondent à une *file min* : pour une *file max*, on aura `get_max` et `extract_max`.

⇨ Les valeurs seront souvent (mais pas toujours) uniques. En revanche, il sera presque toujours possible d'avoir plusieurs éléments de même priorité.

⇨ `get_min` peut bien sûr être implémenté à l'aide de `extract_min` et `insert`, mais rarement de manière efficace.

⇨ Dans certaines applications, il est souhaitable de disposer d'autres opérations :
— *fusionner* deux files.
— *modifier la priorité* d'un élément déjà présent dans la file.
Certaines réalisations de la structure permettent de réaliser ces opérations de manière efficace, d'autres non.

⇨ La signature donnée est impérative, mais il est tout à fait possible de définir des files de priorité fonctionnelles.

Exercice 11

⇨ *Réalisation par un* Abr

1. Expliquer comment on pourrait réaliser une file de priorité à l'aide d'un arbre binaire de recherche.
2. Si l'on utilise un arbre rouge-noir, quelle complexité obtient-on pour les opérations `get_min`, `extract_min` et `insert` ?

3.2 Tas binaire

Définition 3.1: Arbre binaire complet

Un *arbre binaire complet gauche* de hauteur h est un arbre binaire qui vérifie les propriétés suivantes :

— Tous les niveaux sauf éventuellement le dernier sont complets. Autrement dit, il y a exactement 2^i nœuds à profondeur i, pour $0 \leqslant i < h$.

— Le dernier niveau est rempli de gauche à droite.

Remarques

⇨ Dans la suite, on écrira parfois simplement « arbre binaire complet » pour « arbre binaire complet *gauche* ».

⇨ La forme d'un arbre binaire complet est entièrement déterminée par son nombre de nœuds : plus précisément, pour tout $n \in \mathbb{N}$, il y a exactement un arbre binaire complet ayant n nœuds.

⇨ Tous les nœuds internes d'un arbre binaire complet ont un fils gauche. Tous, sauf éventuellement un, ont un fils droit.

⇨ Les feuilles d'un arbre binaire complet de hauteur h sont toutes à profondeur h ou $h - 1$.

Proposition 3.2

Soit un arbre binaire complet de hauteur h et de taille (nombre de nœuds) $n \geqslant 1$. Alors

$$h = \lfloor \log_2 n \rfloor \qquad \text{et} \qquad 2^h \leqslant n < 2^{h+1}.$$

Remarque

⇨ Un arbre binaire complet est donc « parfaitement équilibré » : les opérations nécessitant de parcourir un chemin de la racine à une feuille y sont rapides.

Définition 3.3: Propriété d'ordre des tas

Soit T un arbre dont les nœuds (internes ou pas) sont étiquetés par des éléments d'un ensemble totalement ordonné $(E, \leqslant)$.

— On dit que T a *la propriété d'ordre des tas-min* si l'étiquette d'un nœud est toujours inférieure ou égale à celles de ses (éventuels) enfants.

— On dit que T est un *tas(-min) binaire* si, de plus, c'est un arbre binaire complet gauche.

Remarques

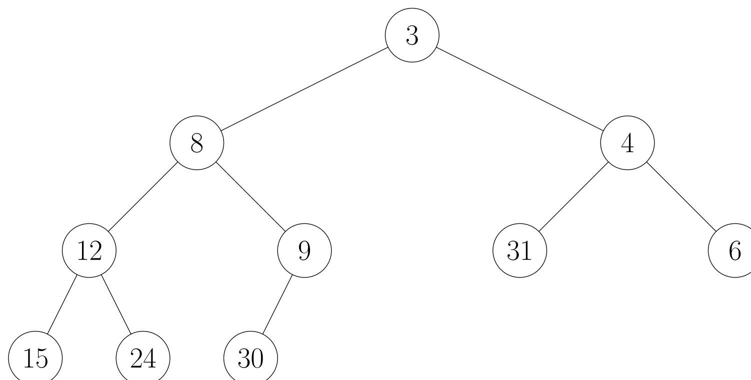
⇨ On ne sait rien sur l'étiquette du fils gauche par rapport à celle du fils droit (à part qu'elles sont toutes deux plus grandes que celle du père).

⇨ Contrairement aux arbres binaires de recherche, la propriété d'ordre des tas est *purement locale*.

⇨ Si l'étiquette d'un nœud est toujours *supérieure ou égale* à celle de ses enfants, on parlera de *tas-max*.

⇨ Le terme anglais est *(binary) heap* (et donc *min-heap*, *max-heap*).

⇨ Autant la pile d'appels a bien une structure de pile, autant le tas sur lequel on fait des `malloc` n'a absolument rien à voir avec la structure de tas que l'on vient de décrire.



Proposition 3.4

Quelques propriétés immédiates des tas-min :
- Si a est un ancêtre de b, alors l'étiquette de a est inférieure ou égale à celle de b.
- La racine contient un élément de clé minimale.

Remarque

⇨ Pour un ensemble donné d'étiquettes, la forme d'un tas binaire est entièrement déterminée. Cette forme ne dépend que du nombre d'étiquettes puisqu'il s'agit d'un arbre complet gauche. En revanche, il y a de nombreuses manières de placer les étiquettes en respectant la propriété d'ordre des tas : les dénombrer est un bon exercice, même s'il ne semble pas y avoir de formule close simple pour le cas général.

3.3 Tas impératif à l'aide d'un arbre implicite

Le principe général pour réaliser une file de priorité à l'aide d'un tas est de stocker les couples *(priorité, valeur)* dans un arbre en imposant la propriété d'ordre de tas *sur les priorités*. Les valeurs jouent donc un rôle uniquement passif : pour simplifier, on les ignore complètement dans ce qui suit. Bien entendu, quand nous réaliserons cette structure en C ou en OCaml, nous en tiendrons compte.

Théorème 3.5

On peut représenter un arbre binaire *complet gauche* ayant n nœuds dans un tableau de longueur n en énumérant ses étiquettes dans l'ordre du parcours en largeur. Cette représentation est injective : à un tableau donné correspond un seul arbre.

On a les propriétés suivantes :
- La racine est à l'indice 0.
- Les nœuds de profondeur i sont numérotés consécutivement à partir de l'indice $2^i - 1$.
- Les fils gauche et droit du nœud d'indice k sont respectivement les nœuds d'indice $2k + 1$ et $2k + 2$.
- Le père d'un nœud d'indice $k > 0$ a pour indice $\left\lfloor \frac{k-1}{2} \right\rfloor$.

Remarques

⇨ On peut bien sûr stocker le parcours en largeur de n'importe quel arbre binaire dans un tableau, mais si on ne se limite pas aux arbres complets gauches, un même tableau correspondra à plusieurs arbres.

⇨ Un tas binaire étant par définition un arbre binaire complet gauche, il peut être représenté de cette manière.

⇨ Une telle représentation a deux avantages par rapport à la représentation classique d'un arbre :
- Elle est beaucoup plus compacte, puisqu'on ne stocke aucun pointeur.
- Elle permet facilement de passer d'un nœud à son père. Cela nécessiterait dans la représentation usuelle de rajouter un pointeur parent à la structure, et augmenterait donc encore la consommation mémoire.

Exercice 12

⇨ 1. Écrire le tableau obtenu à partir de l'exemple de tas binaire donné plus haut :
   (a) En faisant un parcours en largeur.
   (b) En utilisant les propriétés données dans le théorème.
   Vérifier que l'on obtient bien la même chose !
2. Dessiner l'arbre correspondant au tableau suivant : `[20, 13, 14, 7, 8, 1]`.

En pratique, les opérations intéressantes sur les tas (insertion d'un élément, extraction du minimum) font varier le nombre d'éléments. La longueur du tableau ne sera donc pas en général égale au nombre d'éléments présents dans le tas, qu'on aura besoin de stocker. En C, on pourrait par exemple utiliser la structure suivante :

```c
struct heap {
  int size;
  int capacity;
  T *arr;
};

typedef struct heap heap;
```

On remarque que cette structure est exactement celle que nous avons définie pour les tableaux dynamiques. Dans la plupart des applications, il ne sera en fait jamais nécessaire de redimensionner le tableau (parce que l'on disposera

d'une borne sur le nombre d'éléments présents, et que l'on pourra directement créer un tableau de taille adéquate) ; le reste du temps, on utilisera la stratégie habituelle pour le redimensionnement :

— Si l'on doit rajouter un élément et que le tableau est « plein » (`size == capacity`), on réalloue un tableau deux fois plus grand ;
— Si l'on enlève un élément et qu'on obtient `size < capacity / 4`, on réalloue un tableau deux fois plus petit.

En OCaml, une version non redimensionnable utilisera le type suivant :

```
type 'a heap = {mutable size : int; data : 'a array}
```

Pour une version redimensionnable, il suffit de rendre le champ `data` mutable :

```
type 'a heap = {mutable size : int; mutable data : 'a array}
```

Pour simplifier (très légèrement), on utilisera dans le cours une version non redimensionnable, et l'on réservera la version redimensionnable aux TP. On définit quelques fonctions utilitaires :

```
heap *heap_new(int capacity) {
    heap *h = malloc(sizeof(heap));
    h->arr = malloc(capacity * sizeof(T));
    h->size = 0;
    h->capacity = capacity;
    return h;
}
```

```
void heap_delete(heap *h) {
    free(h->arr);
    free(h);
}

void swap(T *arr, int i, int j) {
    T tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

```
int up(int i) {
    return (i - 1) / 2;
}

int left(int i) {
    return 2 * i + 1;
}

int right(int i) {
    return 2 * i + 2;
}
```

3.4 Opérations sur un tas binaire

Le but est ici de réaliser de manière efficace la structure abstraite de file de priorité. *On suppose ici que l'on travaille avec des tas-min.*

3.4.1 Lecture du minimum

C'est immédiat : le minimum est à la racine.

Proposition 3.6

On peut accéder au minimum d'un tas de taille n en temps $O(1)$.

3.4.2 Insertion

Pour insérer une nouvelle clé dans un tas, on écrit cette clé dans la première case libre du tableau, on incrémente le cardinal, puis l'on fait une *percolation vers le haut* (ou *sift-up* en anglais) pour rétablir la propriété de tas. Pour faire percoler un élément vers le haut :
- si la clé de l'élément est supérieure ou égale à celle de son père (ou si on est à la racine), on s'arrête ;
- sinon, on échange l'élément avec son père et l'on recommence.

Remarque

⇨ Essentiellement, on insère le nouvel élément à la bonne place dans la liste triée correspondant au chemin qui le relie à la racine.

Proposition 3.7

Si $\mathcal{T}$ est un tas binaire, alors le résultat $\mathcal{T}'$ de l'insertion d'un élément x dans $\mathcal{T}$ par le processus décrit ci-dessus est encore un tas binaire (qui contient les éléments de $\mathcal{T}$ plus une occurrence supplémentaire de x).

Exercices 13

⇨ Simuler les insertions successives de 2, 5 et 1 dans le tas donné en exemple plus haut.

⇨ 1. Programmer en C la fonction `sift_up` qui effectue une percolation vers le haut du nœud dont on fournit l'indice.
2. Écrire la fonction `heap_insert` qui permet d'insérer un élément dans le tas.

```c
void sift_up(heap *h, int i);
void heap_insert(heap *h, T x);
```

Proposition 3.8: Complexité de l'insertion

Insérer un élément dans un tas de taille n se fait en temps $\mathrm{O}(\log n)$.

3.4.3 Extraction du minimum

Pour extraire le minimum d'un tas binaire, on commence par le lire (évidemment !), puis l'on recopie le dernier élément du tableau (la feuille tout en bas et tout à droite de l'arbre) sur la racine. On supprime cette feuille et l'on fait une *percolation vers le bas* (*sift-down*) de la nouvelle racine. Pour faire percoler un élément vers le bas :
- on le compare à ses deux fils ;
- s'il est inférieur ou égal à ses deux fils, on s'arrête ;
- sinon, on l'échange avec le plus petit de ses deux fils et l'on recommence.

Remarque

⇨ Évidemment, on s'arrête aussi si l'élément n'a pas de fils et l'on s'adapte s'il n'en a qu'un seul.

Exercice 14

⇨ Effectuer trois opérations successives d'extraction du minimum sur le tas donné en exemple.

Proposition 3.9

Si $\mathcal{T}$ est un tas binaire non vide, alors le résultat $\mathcal{T}'$ de l'extraction du minimum de $\mathcal{T}$ est encore un tas binaire.

Exercice 15

⇨ 1. Écrire la fonction `sift_down` qui prend en entrée un pointeur vers un tas et l'indice d'un nœud et effectue une percolation vers le bas de ce nœud.
2. Écrire la fonction `heap_extract_min` qui renvoie le minimum d'un tas et le supprime.

```c
void sift_down(heap *h, int i);
T heap_extract_min(heap *h);
```

Proposition 3.10: Complexité de l'extraction du minimum

On peut extraire le minimum d'un tas contenant n éléments en temps $\mathrm{O}(\log n)$.

3.4.4 Heapify

Une autre opération possible, très utile en particulier pour le tri par tas, est communément appelée *heapify* (*entassement*, si l'on veut). Elle consiste à prendre un tableau quelconque que l'on interprète comme représentant un arbre binaire complet gauche et à le modifier pour qu'il respecte la propriété de tas.

Exercice 16

⇨ Comment peut-on réaliser cette opération à partir de celles dont nous disposons ? Quelle complexité obtient-on, en fonction de la taille n du tableau ?

On peut faire plus efficace en utilisant l'algorithme suivant, qui prend en entrée un tableau de taille n (éléments numérotés de 0 à $n-1$) et le modifie en place pour en faire un tas binaire :

Algorithme 1 Algorithme efficace pour *heapify*.

fonction HEAPIFY(t)
  pour $i = \lfloor \frac{n-2}{2} \rfloor$ à 0 **faire**
    SIFTDOWN(t, i)
  fin pour
fin fonction

Exercices 17

⇨ Écrire une fonction `heapify` ayant la spécification suivante :

Entrées : un entier `size` et un tableau `arr` de `size` élements (de type `T`) ;

Sortie : un pointeur `h` vers un tas, ayant `arr` comme tableau de données. Le tableau `arr` aura été modifié de manière à ce que `h` soit effectivement un tas binaire.

On supposera que le tableau `arr` passé en argument est le résultat d'un `malloc`, et que la responsabilité de sa libération est transférée à `h`. Pourquoi est-ce important ?

```
heap *heapify(T arr[], int size);
```

⇨ Analyse de HEAPIFY

1. En remplaçant le $i = \lfloor \frac{n-2}{2} \rfloor$ par $i = n-1$, justifier la correction de l'algorithme.
2. Expliquer pourquoi il est correct de commencer à $i = \lfloor \frac{n-2}{2} \rfloor$.
3. Justifier que la complexité de HEAPIFY est en $\Theta(n)$.

3.5 Tri par tas

À partir d'une réalisation de la structure de file de priorité, il est toujours possible d'obtenir un algorithme de tri :

— on commence par *insérer* les n éléments à trier dans une file initialement vide ;
— on fait ensuite n *extractions du minimum*.

Remarque

⇨ Nous verrons plus tard que dans le cadre assez général des *tris par comparaison*, il est impossible d'obtenir un algorithme de tri en temps $o(n \log n)$. On peut donc en déduire qu'aucune réalisation de la structure de file de priorité ne peut proposer *simultanément* des opérations d'insertion et d'extraction du minimum en temps $o(\log n)$. Certaines variantes plus sophistiquées que les tas binaires (tas binomiaux, tas de Fibonacci, ...) peuvent en revanche fournir l'une de ces opérations (l'insertion, typiquement) en temps $O(1)$ et l'autre en $O(\log n)$.

Exercices 18

⇨ *Tri par tas* Écrire une fonction `heapsort` prenant en entrée un tableau `arr` et effectuant un tri en place de ce tableau par ordre décroissant.

```
void heapsort(T arr[], int len);
```