# Chapitre 13

# Structures de données (S2)

| Notions | Commentaires |
|---|---|
| Structure de tableau associatif implémenté par une table de hachage. | La construction d'une fonction de hachage et les méthodes de gestion des collisions éventuelles ne sont pas des exigibles du programme. |

*Extrait de la section 3.2 du programme officiel de MP2I : « Structures de données séquentielles ».*

| Notions | Commentaires |
|---|---|
| Implémentation d'un tableau associatif par un [...] arbre bicolore. | On note l'importance de munir l'ensemble des clés d'un ordre total. |
| Propriété de tas. Structure de file de priorité implémentée par un arbre binaire ayant la propriété de tas. | Tri par tas. |

*Extrait de la section 3.3 du programme officiel de MP2I : « Structures de données hiérarchiques ».*

## Sommaire

# 0 Tableaux associatifs

## 0.0 Type abstrait

### 0.0.0 Motivation

Les tableaux que l'on a manipulé jusqu'à présent stockent une valeur par indice : autrement dit, à chaque indice ils *associent* une valeur.

*Exemple.* Dans les graphes, on a beaucoup manipulé des tableaux qui à un noeud associe son parent dans l'arbre de parcours, ou le fait que le noeud soit marqué ou non.

Cependant, quand il s'agit de stocker des associations, les tableaux ont deux limitations :

- Les indices d'un tableau sont obligatoirement des entiers.
- Les indices sont obligatoirement un segment $[\![0; n[\![$ avec $n$ connu à l'avance.

On voudrait pouvoir échapper à ces deux limitations.

**Définition 1 (Tableau associatif).**
Un **tableau associatif**, aussi appelé **dictionnaire**, est une structure de données abstraite qui permet de stocker des associations de la forme

$$clef \mapsto valeur$$

*Exemple.*

FIGURE 13.1 – Un tableau associatif

*Exemple.*

- Un... dictionnaire (par exemple du français) est un tableau associatif qui à un mot (clef) associe sa définition (valeur).
- Un annuaire à un nom (clef) associe un numéro de téléphone (valeur).
- Si on se place dans un graphe (où les sommets ne sont pas forcément des entiers), la représentation par listes d'adjacences à un sommet (clef) associe la liste de ses voisins (valeur).

### 0.0.1 Type abstrait

Les opérations que l'on veut pouvoir faire sur un dictionnaire sont :

- Obtenir un dictionnaire vide.
- Tester si une clef est associée à quelque chose.
- Accéder à la valeur associée à une clef.
- Ajouter une association clef $\mapsto$ valeur

- Enlève l'association d'une clef.
- Tester si un dictionnaire est vide.
- Obtenir une association d'un dictionnaire (n'importe laquelle).

*Remarque.* Il y a une subtilité : que se passe-t-il quand on ajoute une nouvelle association pour une clef déjà associée ? Différents fonctionnements sont possibles :

- On peut lever une erreur. C'est peu pratique, réécrire dans une « case » d'un dictionnaire est courant pour la même raison que réécrire dans une case d'un tableau est courant.
- La nouvelle association peut écraser la précédente (comme dans un tableau).
- La nouvelle association se rajoute « par dessus la précédente » : la nouvelle est l'association « active » (donc quand on accède on trouve la nouvelle valeur), mais si on supprime à l'avenir l'association de la clef est enlevée seule la nouvelle association sera enlevée.

FIGURE 13.2 – Plusieurs d'associations pour une même clef

Voici le type abstrait du type dictionnaire :

| | |
|---|---|
| Type : | **Dict** |
| Utilise : | Clef, Valeur, Bool |
| Opérations : | |
| | `dict_vide` : Rien → Dict |
| | `est_asso` : Dict × Clef → Bool |
| | `acces` : Dict × Clef → Valeur |
| | `ajoute` : Dict × Clef × Valeur → Dict |
| | `enlève` : Dict × Clef → Dict |
| | `est_vide` : Dict -> Bool |
| | `choisir` : Dict -> Clef × Valeur |

TABLE 13.1 – Signature fonctionnelle du type abstrait dictionnaire

## 0.1 Implémentation avec des listes d'associations

**Définition 2 (Liste d'associations).**
Une liste d'associations est une liste linéaire de paires (*clef*, *valeur*).
C'est une façon naïve d'implémenter des tableaux associatifs.

*Exemple.* Ci-dessus une liste d'associations `string` ↦ `int`, où la liste linéaire est implémentée par une `list` :

```ocaml
let dict = [
  ("dragon", 6);
  ("forêt", 5);
  ("chateau", 7);
  ("cave", 4);
  ("abrac", 11);
  ("réponse", 42);
  ("souterrain", 10)
]
```

En OCaml, plusieurs fonctions du module `List` sont pré-codées pour manipuler des listes d'associations : `https://ocaml.org/manual/4.14/api/List.html#1_Associationlists`

L'avantage principal de cette façon de faire est sa simplicité, l'inconvénient est sa complexité temporelle [1]...

## 0.2 Implémentation avec des ABR équilibrés

Pour cette implémentation, il *faut* que les clefs soient munies d'un ordre total.

**Définition 3 (Arbres équilibrés pour tableaux associatifs).**
Étant donnés un ensemble $\mathscr{K}$ totalement ordonné (ensemble de clefs), et $\mathscr{V}$ (ensemble de valeurs) de valeur, on peut représenter $A \subseteq \mathscr{K} \times \mathscr{V}$ dans un arbre binaire de recherche comme suit :

- Un noeud stocke une paire (une association) $(k, v) \in A$
- Pour comparer deux noeuds, on compare uniquement les clefs.

*Exemple.* L'arbre ci-dessous stocke les mêmes associations que la liste d'associations précédente :

FIGURE 13.3 – Implémentation d'un tableau associatif par un arbre binaire de recherche

**Proposition 4.**
Pour que cette implémentation soit efficace, il faut que l'arbre soit équilibré. On peut par exemple utiliser des arbres Rouge-Noir.

*Remarque.*

- On peut aussi utiliser des AVL pour équilibrer les ABR (hors-programme, mais vous avez un lien vers une très bonne explication dans le cours sur les arbres).
- Si les clefs sont des strings et ont beaucoup de préfixes en commun, utilisert un *trie* au lieu d'un ABR peut-être pertinent.

1. Donc pour des petits dictionnaires, pas de problèmes !

## 0.3 Implémentation avec des tables de hachage

### 0.3.0 Fonction de hachage

Le cas facile des tableaux associatifs est celui où l'ensemble des clefs est de la forme $[\![0;n[\![$, puisque l'on peut alors simplement utiliser des tableaux. L'objectif des tables de hachage est de se ramener à ce cas facile.

**Définition 5 (Fonction de hachage).**
Soit $M \in \mathbb{N}^*$ (plutôt petit) et $\mathscr{K}$ un (grand) ensemble de clefs.
Une **fonction de hachage** pour $\mathscr{K}$ est une fonction $h : \mathscr{K} \to [\![0;M[\![$.
Pour $k \in \mathscr{K}$, on dit que $h(k)$ est le **hash** (ou **empreinte**) de $k$.
Si $h(x) = h(y)$, on dit que $x$ et $y$ sont en **collision**.

*Remarque.* Une fonction de hachage **doit** être déterministe : on veut que hacher deux fois le même objet donne deux fois la même empreinte !

*Exemple.*

- Si $\mathscr{K} =$ `string`, un hachage possible est de prendre la somme des lettres (`a` = 1, `b` = 26, ...) modulo $M$. En pratique, cette façon de faire ne répartit pas assez uniformément (car certaines lettres sont bien plus probables que d'autres).
- Si $\mathscr{K} =$ `double`, on peut prendre le code binaire de chaque flottant, l'interpréter comme un entier et le prendre modulo $M$.
- Plus généralement, une fonction de hachage hache souvent la clef en un entier, puis se ramène $[\![O;m[\![$ par un modulo.

FIGURE 13.4 – Fonctionnement de beaucoup de fonctions de hachage

**Proposition 6 (Qualités d'une fonction de hachage).**
Une « bonne » fonction de hachage est une fonction qui a peu de collision. Autrement dit, elle répartit « à peu près uniformément » les éléments de $\mathscr{K}$ :

$$\mathbb{P}\big(h(x) = h(y)\big) \simeq \frac{1}{M}$$

*Remarque.*

- En pratique, les clefs ne sont pas toujours équiprobables, on ne prend donc pas toujours la loi uniforme (sur l'ensemble des $(x,y)$) pour $\mathbb{P}$.
- Une autre propriété désirable est la **pseudo-injectivité** : que deux éléments proches aient des empreintes éloignées. En effet, en pratique, quand on hache des éléments on hache souvent beaucoup d'éléments « les uns à la suite des autres ». La pseudo-injectivité permet alors d'à peu près garantir que tous ces éléments auront une empreinte différente.

  En fait, la pseudo-injectivité peut même servir à remplacer le hachage uniforme : en pratique, c'est plus simple à obtenir et fonctionne au moins aussi bien.

### 0.3.1 Table de hachage

**Définition 7 (Table de hachage).**
Soit $\mathscr{K}$ ensemble de clefs et $\mathscr{V}$ de valeurs. On fixe $M \in \mathbb{N}^*$ et on se dote d'une fonction de hachage associée.
Une **table de hachage** est une façon d'implémenter un tableau associatif qui fonctionne ainsi :

- La table est un tableau à $M$ cases, appelées **alvéoles**.
- Dans l'**alvéole** d'indice $e$ se trouve la liste des paires (clef, valeur) dont l'empreinte de la clef est $e$.

FIGURE 13.5 – Schéma d'une table de hachage

*Exemple.* Pour les mêmes associations que précédemment, avec $M = 5$ et hachage la somme des lettres modulo $M$ :

FIGURE 13.6 – Implémentation d'un tableau associatif par une table de hachage

Les tables de hachage sont fournies en OCaml par le module `Hashtbl` : `https://ocaml.org/manual/4.14/api/Hashtbl.html`

*Remarque.* Dans la définition donnée ici, les collisions sont gérées en stockant la liste des valeurs en collision pour chaque empreinte. Il existe d'autres façons de faire, par exemple en faisant un deuxième niveau de hachage pour encore diminuer le nombre de collisions :

Figure 13.7 – Table à double hachage

### 0.3.2 Attention aux objets mutables !

Les objets dont le contenu est mutable (comme les tableaux) sont assez embêtants quand on les hash. En effet, est-ce que l'on veut hacher :

- Leur contenu : dans ce cas, hacher deux fois le même objet mutable (par exemple le même tableau) pourra tout à fait ne pas donner deux fois la même empreinte ! C'est embêtant, ça veut dire qu'en modifiant l'objet mutable on « oublie » où se trouve toutes ses associations dans le dictionnaire...
- Leur adresse : dans ce cas, deux variables différentes mais ayant exactement le même contenu (par exemple deux tableaux distincts dont le contenu des cases est égal, ou pire encore deux pointeurs qui pointent vers la même mémoire) ne seront pas identifiés comme identiques...

**Proposition 8.**
En OCaml, le hachage d'un objet mutable hache son *contenu*. Ainsi, hacher deux fois le même tableau peut tout à fait ne pas donner la même empreinte.
(HP) En Python, hacher un objet mutable est interdit. On va alors généralement calculer la valeur du contenu et la hacher elle.

## 0.4 Complexités

En notant $A$ l'ensembles des associations à stcoker, et $M$ la taille de la table de hachage (et on supposant que les compraisons/hachage de clefs se font temps constant) :

| Opération \ Implémentation | Liste d'associations | ARN | Table de hachage |
|---|---|---|---|
| `dict_vide` | 1 | 1 | 1 |
| `est_asso` | $\|A\|$ | $\log_2 \|A\|$ | 1 |
| `acces` | $\|A\|$ | $\log_2 \|A\|$ | dépend des collisions (gros 1 en pratique) |
| `ajoute` *(avec doublons de clefs)* | 1 | $\log_2 \|A\|$ | dépend des collisions (gros 1 en pratique) |
| `ajoute` *(sans doublons de clefs)* | $\|A\|$ | $\log_2 \|A\|$ | dépend des collisions (gros 1 en pratique) |
| `enleve` | $\|A\|$ | $\log_2 \|A\|$ | dépend des collisions (gros 1 en pratique) |
| `est_vide` | 1 | 1 | $M$ (ou moins selon l'implem) |
| `choisir` | 1 | 1 | $M$ |

Table 13.2 – Complexités des opérations usuelles sur les dictionnaires

*Remarque.* La complexité des tables de hachage est en pratique un (un peu gros) $O(1)$.

## 0.5 Tri par dénombrement

Les dictionnaires permettent d'écrire l'algorithme de tri ci-dessous, où je note $d[x]$ la valeur associée à $x$ dans le dictionnaire $d$ :

**Algorithme 21 :** Tri par dénombrement

**Entrées :** *arr* un tableau à trier
**Sorties :** Rien, mais modifie *arr* pour le trier

```
1  d ← dictionnaire vide
2  pour chaque x ∈ arr faire
3  |  si x est une clef de d alors  d[x] ← d[x] + 1
4  |  sinon  d[x] ← 1
5  |
6  i ← 0
7  pour chaque clef x de d dans l'ordre croissant faire
8  |  pour k de 0 à d[x] exclu faire
9  |  |  arr[i] ← x
10 |  |  i ← i + 1
```

Remarquez que pour conclure, cet algorithme a besoin de parcourir les clefs dans l'ordre croissant... Autrement dit, il a besoin de trier les clefs. Sans hypothèse supplémentaire sur les données, il est donc en $\Omega(n \log_2 n)$.

**Cependant**, dans le cas où les données sont des entiers on peut procéder ainsi :

- Commencer par trouver le min $m$ et le max $M$ du tableau.
- Utiliser la fonction de hachage suivante : $x \mapsto x - m$. Notez que cette fonction est croissante !
- Créer une table à $M - m + 1$ cases, et appliquer l'algorithme précédent avec.
- Pour obtenir les clefs dans l'ordre, il suffit alors de parcourir la table dans l'ordre (le hachage est croissant) !

*Exemple.*

Figure 13.8 – Tri par dénombrement

**Proposition 9 (Tri par dénombrement).**
Le tri par dénombrement d'un tableau de *len* entiers dont le min est $m$ et le max $M$ s'effectue en $\Theta(len + m + M)$.

*Démonstration.* Analyse du pseudo-code précédemment expliqué. □

*Remarque.* On avait pourtant prouvé qu'un tri se fait en $\Omega(n \log_2 n)$, non ? Non ! On avait prouvé qu'un tri *par comparaisons*[2] est en $\Omega(n \log_2 n)$ comparaisons.

Or, ce tri-ci ne fonctionne pas par comparaisons : on utilise le fait que les entrées sont des entiers pour fonctionner autrement.

## 0.6 Un type d'ensemble

Les tableaux associatifs permettent de facilement représenter des ensembles : les éléments qui sont dans l'ensemble sont associés à quelque chose, et ceux qui n'y sont pas ne sont pas associés.

*Remarque.*

- Cela permet notamment de faire des « ensemble de sommets marqués » dans un parcours ! Attention toutefois : lorsque l'on peut utiliser un tableau de booléens, cela sera (presque) toujours plus efficace.
- Dans le cas d'une implémentation par arbre... autant utiliser directement un arbre équilibré qui stocke les éléments de l'ensemble, et oublier les « quelque chose » qui jouent le rôle de valeur. Le module `Set` fournit de tels ensembles en OCaml, mais il est très hors-programme[3].

2. Un tri où la seule façon de distinguer deux données est de les comparer
3. Car il est défini à l'aide d'un foncteur, c'est à dire d'une « fonction » qui prend en argument un module et renvoie un module.

# 1 Tas

Dans toute cette section, on se donne $(\mathscr{E}, \leq)$ un ensemble totalement ordonné d'étiquettes.

## 1.0 Définition

### 1.0.0 Définition

**Définition 10 (Tas min).**
Un tas binaire min (anglais : *binaray min heap*) (resp. tas binaire max / *binary max heap*), abrégé **tas min** (resp. **tas max.**), est un arbre binaire complet étiquetté par $\mathscr{E}$ dont l'étiquette de chaque noeud est inférieure (resp. supérieure) à celle des racines de ses enfants.

*Exemple.*

FIGURE 13.9 – Un tas min contenant $[\![0; 9[\![$

*Remarque.*

- On peut aussi définir des tas min (resp. max) ternaires, ou k-aire en général. Dans ce cours (et dans le programme) on se limite aux tas binaires.
- **Tas et ABR ne sont pas la même notion !** Quelques différences :
  - La propriété d'ABR n'est *pas* locale (on ne peut pas juste la vérifier entre chaque noeud et les racines de ses enfants) ; celle de tas ci (elle est définie comme étant locale).
  - Dans un ABR, le minimum est obtenu en allant toujours à gauche, dans un tas min il est à la racine.
  - Rechercher un élément dans un ABR est rapide, pas dans un tas (doit-on aller à gauche ou à droite ?).
  - Un ABR est trié par parcours infixe, un tas pas du tout (cf exemple ci-dessus).

**Proposition 11 (Définition non-locale d'un tas).**
Un arbre binaire $A$ est un tas min si et seulement si pour tout noeud, l'étiquette du noeud est inférieure à l'étiquette de tous les noeuds de ses sous-arbres.

*Démonstration.* La réciproque est immédiate. L'implication s'obtient par transitivité de $\leq$ le long de tout chemin descendant. □

*Remarque.* La définition non-locale ne sert pas en pratique, puisque la version locale ets plus simple à tester. Toutefois, c'est bon de l'avoir en tête pour prouver certains propriétés théoriques.

**Définition 12 (Rappel : arbre binaire complet dans un tableau).**
On peut représenter un arbre binaire complet dans un tableau en stockant ses éléments dans l'ordre d'un parcours en largeur (de gauche à droite). Cela fait que :

- La racine est à l'indice 0.
- L'étage $p > 0$ correspond aux noeuds d'indice $[\![2^i - 1; 2^{i+1} - 1[\![$
- Si $i$ est l'indice d'un noeud, son noeud parent est à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$.
- Si $i$ est l'indice d'un noeud, ses (évenntuels) enfants gauche et droit sont respectivement à l'indice $2 * i + 1$ et $2 * i + 2$.

*Démonstration.* On peut prouver ces propriétés :

- Immédiat pour la racine.
- Soit $i$ un étage. Comme l'arbre est complet, tous les étages d'avant sont entièrement remplis : d'après le cours sur les arbres, ils contiennent donc au total :

$$\sum_{p=0}^{i-1} 2^p = 2^i - 1$$

  Il y a donc au total exactement $2^i - 1$ noeuds dans les étages $0, ..., i-1$, qui sont ceux rangés avec les noeuds de l'étage $i$ dans le tableau (BFS). Comme on indice à partir de 0, on obtient bien que le premier élément de l'étage $i$ est d'indice $2^i - 1$. Comme de plus l'étage $i$ contient au plus $2^i$ noeuds, ils vont au plus jusqu'à l'indice $2^{i+1} - 2$ inclus.
- Corollaire du point suivant.
- Écrivons $i$ sous la forme $2^p - 1 + k$ (avec $p$ la profondeur du noeud $i$), et supposons que le noeud correspondant a des enfants. Ce noeud a donc $k$ adelphes avant lui sur son étage. Par BFS, les enfants de ces adelphes sont exactement les noeuds avant les enfants de $i$ sur la rangée $p + 1$.

FIGURE 13.10 – Notations de la preuve

  Ainsi, l'enfant gauche du noeud d'indice $i$ a pour indice $2^{p+1} - 1 + 2k$. Or :

$$2 * (2^p - 1 + k) + 1 = 2^{p+1} - 1 + 2k$$

  De même pour le noeud droit.

□

**Proposition 13 (Tas dans un tableau).**
Pour implémenter un tas, on peut stocker ses valeurs dans un tableau (comme ci-dessus). Il est utile d'anticiper en faisant un tableau trop grand, aussi on stocke à part la taille (le nombre de noeuds) du tas.

### 1.0.1 Implémentation en OCaml

Comme en C, on peut faire des « `struct` » en OCaml. On appelle cela des **enregistrements**. Par défaut les champs d'un enregistrement sont non-mutables ; le mot-clef `mutable` permet de les rendre mutables [4] [5].

Pour les tas, on peut donc procéder ainsi :

minHeap.ml
```
type 'a minHeap = {
  mutable len : int;      (* Taille du tas *)
  mutable tab : 'a array (* Tableau qui contient les éléments *)
}
```

*Remarque.*

- Le champ `len` est mutable car on va ajouter et enlever des éléments, donc la taille du tas va évoluer.
- Le champ `tab` est mutable car on va parfois remplacer le tableau par un autre ! On va en effet utiliser le même redimensionnement que les tableaux dynamiques pour « modifier » efficacement la longueur du tableau [6].

*Remarque.* Une variante fréquente de la représentation des tas est de stocker la racine dans la case d'indice 1, et d'utiliser la case d'indice 0 pour stocker la taille (le nombre de noeuds) du tas.

*Exercice.* Quels sont les liens entre indices des enfants et des parents dans cette variante ?

## 1.1 Opérations

*Remarque.* Dans toute la suite de ce cours, on considère uniquement des tas min. Les tas max se construisent de manière symétrique.

On se donne les fonctions suivantes :

minHeap.ml
```
(* Renvoie l'indice du parent du noeud d'indice i *)
let parent i = (i-1) / 2

(* Renvoie l'indice de l'enfant gauche du noeud d'indice i *)
let gauche i = 2*i+1

(* Renvoie l'indice de l'enfant droite du noeud d'indice i *)
let droite i = 2*i+2
```

### 1.1.0 Percolation vers le haut

**Définition 14 (Pseudo-tas).**
~~On appelle **pseudo-tas** (min) un arbre binaire complet dont la propriété de tas est rompue sur au plus une arête.~~ À reprendre.

*Exemple.*

4. Il ne marche qu'en définissant un type enregistrement, pas sur tout et n'importe quoi...
5. Fun fact : une référence est en fait un enregistrement constituée d'un seul champ mutable !
6. Pour rappel : quand le tableau est plein, le remplacer par un deux fois plus long, quand il est à 75% vide le remplacer par un deux fois plus court.

FIGURE 13.11 – Un pseudo-tas min avec en exergue l'arête qui rompt la propriété de tas

**Définition 15 (Percolation vers le haut).**
La **percolation vers le haut** est une opération qui permet de « réparer » un pseudo-tas (min) en tas (min). Elle procède comme suit : tant qu'il existe un noeud plus petit que son parent, l'échanger avec son parent.

*Remarque.* La percolation vers le haut consiste à « pousser » un élément vers le haut, « à travers » les autres : d'où le terme de *percolation*.

On se donne `swap t i j` une fonction qui échange les cases `i` et `j` d'un tableau. Avec elle, on écrit la percolation :

minHeap.ml

```
27 (* Percolation vers le haut dans le pseudo-tas
28   [t] depuis l'indice [i] *)
29 let rec percoleHaut t i =
30   if i > 0 && t.tab.(parent i) > t.tab.(i) then begin
31     swap t.tab i (parent i);
32     percoleHaut t (parent i)
33   end
```

*Remarque.* On aurait aussi pu implémenter avec une boucle `while`.

**Proposition 16 (Correction de `percoleHaut`).**
~~Si~~ `t` ~~correspond à un pseudo-tas, et que~~ `i` ~~est l'indice de l'enfant éventuellement plus petit que son parent, alors~~ `percoleHaut t i` ~~est un tas.~~ Faux. À reprendre.

*Démonstration.* La correction totale de `swap` est admise[7]. Pour la corrcetion totale de `percoleHaut` :

- La fonction termine car `i` est un variant. En effet il entier, minoré strictement par 0 (ligne 29) et décroit strictement en cas d'appel récursif (ligne 31 ; comme `i > 0` on a bien `parent i < i`).
- On montre la correction partielle par récurrence forte sur $i \in \mathbb{N}$ :
  - Initialisation : Si $i = 0$, alors `i` n'a pas de parent. Il ne peut donc pas être plus petit que son parent, et le pseudo-tas est en fait un tas. De même si `t.tab.(parent i) <= t.tab.(i)`.
  - Hérédité : Soit $i > 0$ tel que la correction soit vraie jusqu'au rang $i$ exclu, montrons-la vraie au rang $i$. Par hypothèse, la seule rupture possible de la propriété de tas est entre $i$ et $parent(i)$.
    On suppose que `t.tab.(parent i) > tab.(i)` (l'autre cas a été traité en initialisation). On va alors faire un échange. Les arêtes potentiellement impactées par l'échange sont toutes celles dont une des extremités est d'indice $i$ ou $parent(i)$ :

7. Et triviale.

(a) Avant swap (b) Après swap

FIGURE 13.12 – Arêtes impactées par l'échange entre $i$ et $parent(i)$

On utilise les conventions de nommage des noeuds et des arêtes de la figure ci-dessus. Avant l'échange, comme par hypothèse seule $C$ enfreint la propriété de tas, on a :

| Arête | Extrémités | Analyse |
|---|---|---|
| A | $pp \to p$ | Valide par hypothèse, donc $pp \leq p$ |
| B | $p \to a$ | Idem, donc $p \leq a$ |
| C | $p \to x$ | Invalide : $p > x$ par hypothèse |
| D | $x \to y$ | Idem que $A$, donc $x \leq y$ |
| E | $x \to z$ | Idem, donc $x \leq z$ |

Après l'échange :

| Arête | Extrémités | Analyse |
|---|---|---|
| A | $pp \to x$ | Peut-être invalide |
| B | $x \to a$ | Valide car $x < p \leq a$ |
| C | $x \to p$ | Valide car $x < p$ |
| D | $p \to y$ | Prblm !! |
| E | $p \to z$ | Prblm !! |

Ainsi, après l'échange, la seule possible violation de la propriété de tas est sur l'arête A (entre l'indice $parent(i)$ et son parent). En appliquant l'hypothèse de récurrence à l'appel récursif on obtient qu'après l'appel récursif $t$ vérifie bien la propriété de tas.

□

### 1.1.1 Insertion

Grâce à la percolation, on obtient un algorithme très simple pour insérer :

**Définition 17 (Insertion dans un tas min).**
Pour insérer dans un tas min, on ajoute le nouvel élément au premier emplacement disponible, puis on le fait percoler vers le haut pour le mettre au bon endroit.

En plus de l'implémenter, on va redimensionner le tableau du tas lorsque nécessaire. À cette fin, on se donne `Array.redim tab new_length` qui crée une copie de `tab` de longueur `new_length` (quitte à enlever la fin de `tab` ou à remplir la fin avec des valeurs en plus).

Voici donc l'insertion :

minHeap.ml

```
42 (** Insertion dans un tas binaire *)
43 let add t x =
44   (* Si besoin, doubler le tableau *)
45   if t.len = Array.length t.tab then
46     t.tab <- redim t.tab (2 * Array.length t.tab);
47
48   (* Puis insérer en percolant vers le haut *)
49   t.tab.(t.len) <- x;
50   t.len <- t.len + 1;
51   percoleHaut t (t.len-1)
```

*Exemple.*

(a) Début de l'insertion (b) Percolation vers le haut (c) Fin de l'insertion

FIGURE 13.13 – Insertion d'un noeud dans un tas

### 1.1.2 Percolation vers le bas

**Définition 18 (Percolation vers le bas).**
La **percolation vers le bas** est une opération qui permet de « réparer » un pseudo-tas (min) en tas (min). Elle procède comme suit : tant qu'il existe un noeud plus grand que son enfant, l'échanger avec le plus petit de ses enfants.

*Remarque.*

- La percolation vers le bas consiste à « pousser » un élément vers le bas.
- Il est important de bien échanger avec le plus *petit* des enfants :

FIGURE 13.14 – La percolation vers le bas ne répare pas si on échange avec le plus grand enfant

Voici une implémentation. C'est un peu plus long que la percolation vers le haut car il faut trouver quel est l'enfant minimal, tout en gérant les cas où il y a 0 ou 1 enfant :

minHeap.ml

```
54 (* Percolation vers le bas dans le tableau
55   [tab] depuis l'indice [i] *)
56 let rec percoleBas t i =
57   if gauche i < t.len then
58
59     (* Calculer l'indice du plus petit enfant*)
60     let i_enf_min =
61       if droite i < t.len && t.tab.(gauche i) > t.tab.(droite i)
62       then droite i
63       else gauche i
64     in
65
66     (* Si besoin, échanger et continuer de percoler*)
67     if t.tab.(i) > t.tab.(i_enf_min) then begin
68       swap t.tab i (i_enf_min);
69       percoleBas t i_enf_min
70     end
```

**Proposition 19 (Correction de `percoleBas`).**
~~Si `t` correspond est un « pseudo-pseudo-tas », c'est à dire que c'est un tas partout sauf éventuellement entre un noeud $i$ et ses enfants, et que `i` est l'indice de ce noeud, alors~~ `percoleBas t i` ~~est un tas.~~ Faux. À reprendre.

*Remarque.* Ici la notion de pseudo-tas n'est pas la bonne : il peut y avoir 2 erreurs puisqu'un noeud est mal placé par rapport à ses enfants, qui peuvent être deux. D'où « pseudo-pseudo-tas ».

*Démonstration.* Pour la correction totale de `percoleBas` :

- La fonction termine car `i` est un variant. En effet il entier, majoré strictement par `t.len` (ligne 57) et croit strictement en cas d'appel récursif (ligne 69 ; puisque `gauche i` et `droite i` sont strictements supérieurs à `i`).
- On montre la correction partielle par récurrence forte décroissante sur $i \in [\![0; \texttt{t.len}[\![$ :
  - Initialisation : Si $i =$ `t.len`, alors `i` n'a pas d'enfants. Il ne peut donc pas être plus grand que ses enfants, et le pseudo-pseudo-tas est en fait un tas.
    Remaruqons également que par définition `i_enf_min` est l' indice du plus petit des enfants (lorsqu'il y en a). Donc si on ne rentre pas dans le `if` ligne 67, le noeud $i$ est plus petit que son plus petit enfant, donc est bien placé et le pseudo-pseudo-tas est en fait un tas.
  - Hérédité : Soit $0 \leq i <$ `t.len` tel que la correction soit vraie de `t.len` jusqu'au rang $i$ exclu, montrons-la vraie au rang $i$. Par hypothèse, la seule rupture possible de la propriété de tas est entre $i$ et ses enfants.
    On procède ensuite comme dans la preuve précédente : on liste les arêtes avant et après l'échange, on conclut que les problèmes éventuels sont descendus sur les arêtes D et E, et on conclut par hypothèse de récurrence.

(a) Avant swap (b) Après swap

FIGURE 13.15 – Arêtes impactées par l'échange entre $i$ et $i\_en\_f\_min$

□

### 1.1.3 Extraction du minimum

Les tas min sont optimisés pour lire et extraire le minimum. Notons immédiatement que le minimum est à l'indice 0 (d'après la verison non-locale de la définition des tas), et que donc le trouver est immédiat.

Cependant, l'enlever n'est *a priori* pas si trivial. On va utiliser la même astuce que dans les ABR : échanger l'élément à supprimer avec un élément facile à supprimer.

**Définition 20 (Extraction du minimum d'un tas).**
Pour extraire le minimum d'un tas, on l'échange avec le dernier élément du tas et on le supprime. Puis, on fait percoler vers le bas cet ancien dernier élément du tas qui vient d'être placé à la racine.

Cela donne le code suivant (avec redimensionnement dynamique) :

minHeap.ml

```
(** Extracition dans un tas binaire *)
let take_min t =
  (* Si besoin, contarcter le tableau *)
  if t.len < 3 * Array.length t.tab / 4 then
    t.tab <- redim t.tab (Array.length t.tab /2);

  let mini = t.tab.(0) in
  swap t.tab 0 (t.len-1);
  t.len <- t.len -1;
  percoleBas t 0;
  mini
```

*Exemple.*

(a) Échange entre la racine et le dernier

(b) Percolation vers le bas

(c) Fin de l'extraction

FIGURE 13.16 – Extraction du minimum dans un tas

## 1.2 Complexités

Analysons pour l'instant la complexité sans prendre en compte le redimensionnement dynamique.

**Proposition 21.**
L'insertion et l'extraction dans un tas binaire à $n$ éléments se fait en temps $\Theta(\log_2 n)$.

*Démonstration.* Dans l'insertion et l'extraction, il y a des opérations en temps constant (dont l'éventuel `swap`), et un appel à `percoleHaut` ou `percoleBas`. Analysons la première, la seconde se fera de même.

`percoleHaut` fait à chaque appel $O(1)$ opération plus un appel sur `parent i`. Ainsi, `i` est divisé par 2 à chaque fois. Or il vaut initialement au plus `t.len-1` (puisque c'est un indice valide), d'où une complexité en $O(\log_2 \texttt{t.len})$.

Pour prouver que cette borne est atteinte, on peut considérer l'insertion d'un noeud étiquetté par 0 dans le tas où les noeuds sont étiquettés par leur profondeur : il faut percoler jusqu'en haut, donc faire un appel par étage donc $\Omega(\log_2 \texttt{t.len})$ appels. □

Le redimensionnement dynamique a déjà été analysé au premier semestre : il coute $O(1)$ *en amorti*. On obtient donc $\Theta(\log_2 n)$ en amorti pour insérer et extraire.

## 1.3 Tri par tas

Les tas donnent un algorithme de tri, qui « correspond » au tri par sélection : créer le tas contenant tous les éléments, puis extraire les minimums successifs et les placer dans l'ordre :

minHeap.ml

```
86 (** Trie en place [arr] *)
87 let heapSort arr =
88   let t = {len = 0; tab = Array.make 1 tab.(0)} in
89   for i = 0 to Array.length arr -1 do
90     add t arr.(i)
91   done;
92   for i = 0 to Array.length arr -1 do
93     arr.(i) <- take_min t
94   done
```

*Remarque.*

- C'est le seul tri explicitement au programme, vous devez donc le maitriser sur le bout des doigts !
- Cela étant dit, les autres tris que l'on a vus sont des immenses classiques qui seront probablement considérés comme acquis par les concours.

**Proposition 22 (Complexité du tri par tas).**
Le tri par tas trie un tableau à $n$ éléments en temps $\Theta(n \log_2 n)$

*Démonstration.* Il s'agit d'additionner les $\log_2()$ successifs : la phase d'insertion (ligne 89-90) insère à chaque itération un noeud dans un tas à $i$ élements. Or :

$$\sum_{i=2}^{n-1} \log_2 i = \log_2 \left( \prod_{i=2}^{n-1} i \right) = \log_2((n-1)!) = \Theta(n \log_2 n)$$

D'où le fait que la phase d'insertion soit en temps $\Theta(n \log_2 n)$. De même pour les extractions successives. □

*Remarque.*

- Il existe un algorithme pour construire un tas en temps *linéaire* ! Mais l'extraction, elle, se fait toujours en temps quasi-linéaire.
- Il existe des variantes des tas qui cherchent à proposer de nouivelles opérations, implémentées (très) efficacement. On peut par exemple citer les tas binomiaux [8] ou les tas de Fibonacci [9]

# 2 Files de priorité

*Remarque.* On fait ici des files de priorité *min* ; les *max* se définissent de manière symétrique.

## 2.0 Type abstrait

**Définition 23.**
Soit $(\mathscr{K}, \leq)$ un ensemble totalement ordonné d'éléments appelés *priorités* et $\mathscr{V}$ un ensemble de *valeurs*.
Une **file de priorité min** sur $\mathscr{K} \times V$ est une structure de données de paires (priorité, valeur) où l'on peut insérer une nouvelle paire, trouver la paire de priorité minimale et extraire la paire de priorité minimale.

*Exemple.*

8. Je vous renvoie à Wikipédia.
9. Idem.

Figure 13.17 – Insertions et extractions dans une file de priorité

Voici le type abstrait du type file de priorité :

| | |
|---|---|
| Type : | **FilePrio** |
| Utilise : | Prio, Valeur, Bool, Entier |
| Opérations : | |
| | `file_vide` : Rien → FilePrio |
| | `ajoute` : File_Prio × Prio × Valeur → File_Prio |
| | `trouve_min` : File_Prio × Prio × Valeur |
| | `enlève_min` : File_Prio → Prio × Valeur |
| | `est_vide` : File_Prio → Bool |
| | `taille` : Fille_Prio → Entier |

Table 13.3 – Signature fonctionnelle du type abstrait file de priorité

## 2.1 Implémentation avec des ABR équilibrés

**Proposition 24.**
Un arbre binaire équilibré contenant des paires (priorité, valeur) ordonnées par priorité permet d'implémenter une file de priorité efficacement.

*Démonstration.* Si besoin, rappelons que l'on a vu comment extraire le minimum d'un ABR en temps linéaire en la hauteur : c'est plus simple que d'extraire un élément quelconque ! □

*Exemple.*

Figure 13.18 – Un arbre rouge-noir implémentant une file de priorité

## 2.2 Implémentation avec des tas

**Proposition 25.**
Un tas contenant des paires (priorité, valeur) ordonnées par priorité permet d'implémenter une file de priorité efficacement.

*Exemple.*

Figure 13.19 – Un tas implémentant la même file de priorité

## 2.3 Complexités

Pour $F$ une file de priorité, en notant $|F|$ sa taille :

| Opération \ Implémentation | ARN | Tas binaire |
|---|---|---|
| `file_vide` | 1 | 1 |
| `ajoute` | $\log_2 \lvert F \rvert$ | $\log_2 \lvert F \rvert$ |
| `trouve_min` | $\log_2 \lvert F \rvert$ | 1 |
| `enleve_min` | $\log_2 \lvert F \rvert$ | $\log_2 \lvert F \rvert$ |
| `est_vide` | 1 | 1 |
| `taille` $\lvert F \rvert$ ou 1 (selon l'implem) | 1 | |

Table 13.4 – Complexités des opérations usuelles sur les files de priorité

## 2.4 Une opération supplémentaire

On veut parfois pouvoir *modifier* la priorité d'un élément (notamment dans l'algorithme de Dijkstra). Une solution simple pour cela est de mémoriser un dictionnaire qui à une valeur associe une information qui permet de retrouver la « position » la structure :

- Dans un ABR on associe la priorité, qui permet de faire des recherches dans l'ABR et donc de trouver l'élément.
- Dans un tas on associe l'indice dans le tableau.

Pour modifier la priorité, on peut alors :

- Dans un ABR, supprimer le noeud et l'insérer avec la nouvelle priorité.
- Dans un tas, modifier la priorité puis percoler vers le haut ou le bas le noeud modifié.

Dans les deux cas c'est en temps logarithmique... Mais très couteux en mémoire puisqu'il faut maintenir un dictionnaire !