TAS ET FILE DE PRIORITÉ

Idée

La file de priorité est une structure de donnée abstraite permettant de gérer un (multi). ensemble dynamique d'éléments muni d'une valeur appelée clé ou priorité, à valeur dans un ensemble totalem^t ordonné.

Elle doit permettre notamment d'ajouter des élem^t (d'où le côté dynamique de l'ensemble) et de récupérer et supprimer 1 élément de plus petite (ou plus grande) priorité.

Type abstrait

FILE_DE_PRIORITÉ_MIN

CRÉER_FdP_MIN () : file de priorité min

AJOUTER (F: fdP_min, x: élem^t, p: priorité) : void

DIMINUER_PRIO (F: fdp min, x: élem^t, p: priorité) : void

DONNER_MIN (F: fdp min) : élem^t

SUPPRIMER_MIN (F: fdp min) : void

Implémenta° concrète

- On peut implémenter cette structure de données abstraite grâce au tas qui est une structure de données concrète basée sur un arbre binaire parfait et tournoi représenté par un tableau.

Def Un arbre binaire est dit parfait si tous ses niveaux sont pleins sauf éventuellem^t le dernier, où les nœuds sont à gauche

Autrement dit, si la plus grande profondeur d'un nd d'un AB est K, alors $\forall k \in [0..K]$, A compte 2^k nd de prof. k, et les nœuds de prof. K sont le + à gauche possible.

Def Un arbre binaire dont les nœuds sont étiquetés par une clé à valeur dans un ens. totalem^t ordonné est dit arbre tournoi ssi chaque nd distinct de la racine a une clé plus grande que celle de son père ds l'arbre.

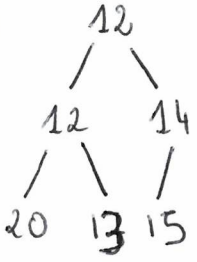
ex

est un A.B. parfait

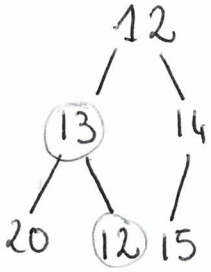
les nds du dernier niveau ne sont pas collés à gauche
↳ cet AB n'est pas parfait

l'avant dernier niveau n'est pas plein
↳ cet AB n'est pas parfait

ex

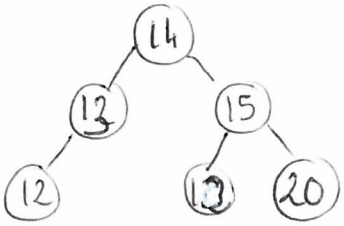


est un arbre tournoi



n'est pas tournoi car "12" est + specht que son père

⚠ AB tournoi ≠ ABR de recherche.



est un ABR mais pas tournoi

Numérota° d'un AB parfait

Convention ici
[profondeur de la racine = 0
hauteur d'un arbre = prof. max]

On numérote (à partir de 1) les nœuds d'un AB de haut en bas (=par prof. croissante) et de gauche à droite.

Ainsi si A est un AB parfait de hauteur H à N nœuds on a:

- $\forall R \in [0..H[$, les nd. de A de prof R sont numérotés de 2^R à $2^{R+1}-1$
- les nd de A de prof H sont numérotés de 2^H à N.

et si x est un nœud numéroté i, on a:

- x est racine $\Leftrightarrow i = 1$
- la profondeur de x est $\lfloor \log_2(i) \rfloor$, en particulier $H = \lfloor \log_2(N) \rfloor$
- si x n'est pas racine, le numéro du père de x est $\lceil \frac{i}{2} \rceil$
- si x a un fils gauche, son numéro est $2i$
- _____ droit, _____ $2i+1$.

En effet, en notant $R = \lfloor \log_2(i) \rfloor$ la hauteur de x, il existe $k \in [0..2^R[$ tq $i = 2^R + k$.
Ainsi au niveau R, x est précédé à gauche par k nœuds, et si x a fils gauche y, celui-ci est au niveau R+1, précédé à gauche par les $2 \times k$ fils des nd à gauche de x.
Ainsi y est numéroté $2^{R+1} + 2k = 2(2^R + k) = 2i$.

Grâce à cette numérotation, on représente facilement un AB parfait par un tableau à une dimension, indexé à partir de 1.

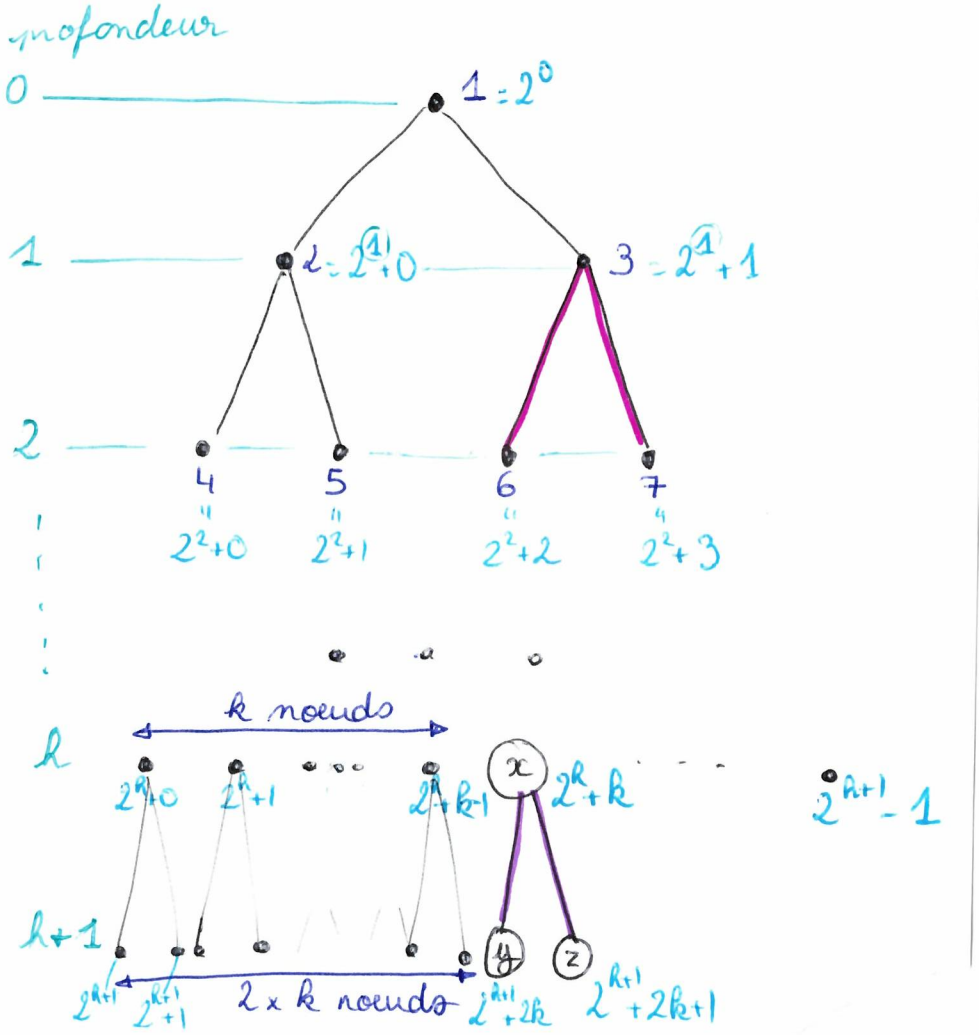
Rmq En fait il n'existe qu'un seul AB parfait ayant N noeuds, donc donner le nombre de nd suffit à décrire la forme d'un AB parfait.

On utilise un tableau pour stocker l'étiquette de chacun des noeuds puisqu'on a affaire à des AB parfait étiquetés.

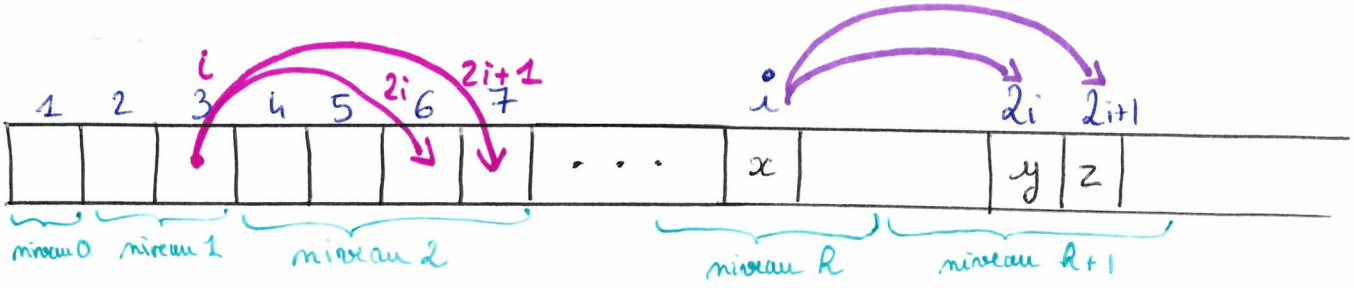
[En pratique, le numéro d'un noeud donne l'indice de la case du tableau où l'on a stocké son étiquette.

schémas

- de la numérota° d'un AB parfait :



- du tableau correspondant :



Il arrive parfois que l'on identifie les éléments et leurs priorités, par exemple lorsqu'on gère un ensemble de valeur, entières ou réelles, dont on veut facilement avoir le minimum (resp. le max.).

Mais les files de priorité sont aussi utilisées dans des contextes où cette identification est impossible

- parce que les élém^t ne peuvent être ordonnés
- parce qu'assimiler les élém^t à leur priorité ferait disparaître des élém^t qui ont la m^e priorité que d'autres

ex. de tels contextes

- algorithmes sur les graphes, où les élém^t sont des sommets

  Dijkstra : la priorité est la distance à la racine si l'on ne considère que les chemins dont les sommets intermédiaires sont fermés (à une étape donnée de l'algo),

  Prim : la priorité est le coût de raccordem^t du sommet à l'arbre en construc° (à une étape de l'algo)

  NB: dans les 2 cas, la priorité d'un sommet ne fait que décroître au cours de l'algorithme.

- algorithmes d'ordonnancem^t, où les élém^t sont des tâches

  "EDD" : la priorité d'une tâche est sa due-date (date d'échéance)

  NB: ici la priorité ne change pas au cours du temps, mais les tâches ne sont pas toutes initialem^t dans la file : elles arrivent au fur et à mesure qu'on avance dans le tps et qu'elles deviennent dispo.

Cadre On se place dans le cas où l'on ne peut pas identifier éléments et priorités. On doit alors s'assurer de pouvoir

- à partir d'un indice i qui représente un nd de l'arbre accéder à l'élément correspondant
- à partir d'un élém^t (dont on veut mettre à jour la priorité par ex) pouvoir accéder à la case correspondante, donc avoir l'indice.

HYP: On suppose que le type des élém^ts contient un champ "indice" capable de stocker un entier, initialem^t vide

HYP: De plus on suppose ici que les élém^t qui peuvent arriver dans la file font partie d'un ensemble fini (ex: les sommets, les tâches) de taille connue

TAS_MIN

- **attributs**
  - `n` entier
  - `capacité` entier
  - `elem` tableau d'éléments (pointeurs) indexé par [1..capacité]
  - `prio` tableau de priorités (type ordonné) indexé par [1..capacité]
- **méthodes**
  - CREER_TAS_VIDE (c entier) :
  - EST_VIDE () : bool
  - DONNER_MIN() : élément (pointeur)
  - SUPPRIMER_MIN() : void
  - DESCENDRE_RACINE() : void
  - AJOUTER (x : elem., p : prio) : void
  - DIMINUER_PRIO (x : elem p : prio) : void

CREER_TAS_MIN_VIDE (c : entier)

```
T ← nouvel objet de type TAS_MIN
T.taille ← 0
T.capacité ← c
T.elem ← tableau de pointeurs initialement vide
         indexé par [1..c]
T.prio ← tableau de "priorité" init. vide
         indexé par [1..c]
```

EST_VIDE()

retourne taille == 0

DONNER_MIN() Hypothèse : EST_VIDE() = faux

retourne elem[1]

SUPPRIMER_MIN() hyp : EST_VIDE() = faux

```
elem[1] ← elem[m]
(elem[m] → indice) ← 1
prio[1] ← prio[m]
m ← m - 1
DESCENDRE_RACINE()
```

DESCENDRE_RACINE()

```
i ← entier initialisé à 1
x_ok ← booléen initialisé à faux
tant que (non x_ok)
    Si 2i+1 ≤ m
        Si prio[2i] ≤ prio[2i+1]
            prio_min ← prio[2i]
            fils_min ← 2i
        Sinon
            prio_min ← prio[2i+1]
            fils_min ← 2i+1
        Si prio[i] > prio_min
            échanger prio[i] et prio[fils_min]
                     ——— elem ——— elem ———
            (elem[i] → indice) ← i
            i ← fils_min
        Sinon
            (elem[i] → indice) ← i
            x_ok ← vrai
    Sinon
        Si 2i ≤ m et prio[i] > prio[2i]
            échanger prio[i] et prio[2i]
                     ——— elem ——— elem ———
            (elem[i] → indice) ← i
            i ← 2i
        sinon
            (elem[i] → indice) ← i
            x_ok ← vrai
```

AJOUTER (x : elem, p prio) hyp : x→indice = ∅ (donc x ∉ T normalement et T.capacité > T.n)

```
n ← n + 1
prio[n] ← +∞
elem[n] ←
(x → indice) ← n
DIMINUER_PRIO (x, p)
```

DIMINUER_PRIO (x : elem, p : prio) hyp : x→indice ≠ ∅ donc x ∈ T.

```
i ← entier initialisé à x→indice.
prio[i] ← p
tant que i > 1 et prio[i] < prio[⌊i/2⌋]
    échanger prio[i] et prio[⌊i/2⌋]
             ——— elem[i] - elem[⌊i/2⌋]
    (elem[i] → indice) ← i
    i ← ⌊i/2⌋
x→indice (i)
```