Arbres binaires de recherche, tas

Lycée Thiers

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

Crédits

- ≪ Option informatique MPSI, MP/MP* ≫, Roger Mansuy, paru chez Vuibert.

Crédits

- ≪ Option informatique MPSI, MP/MP* ≫, Roger Mansuy, paru chez Vuibert.
- Wikipédia :

Crédits

- « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
- Wikipédia :
  - Tas

Crédits

- « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
- Wikipédia :
  - Tas
  - Tas binaires

Crédits

- « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
- Wikipédia :
  - Tas
  - Tas binaires
  - Tri par tas

Crédits

- « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
- Wikipédia :
  - Tas
  - Tas binaires
  - Tri par tas
- OpenClassRoom Arbres binaires de recherche

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

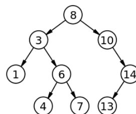
Binary search trees (BST)

Définition

Un *arbre binaire de recherche* (ABR) sur un type totalement ordonné est un arbre binaire tel que pour tout nœud interne, les étiquettes apparaissant dans le sous-arbre gauche (resp.droit) sont strictement[a] inférieures (resp. supérieures) à celle la racine.

a. Selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale.

FIGURE – Un ABR. Les étiquettes de gauche ont des valeurs plus petites que celle de la racine, celle de droite sont plus grandes

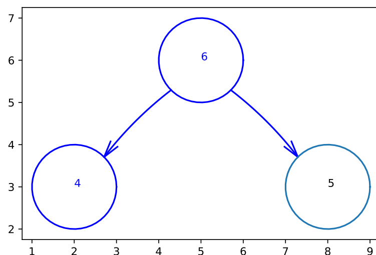


DSF

On peut facilement récupérer les clés d'un arbre binaire de recherche dans l'ordre croissant en réalisant un parcours en profondeur infixe.

Contre exemple

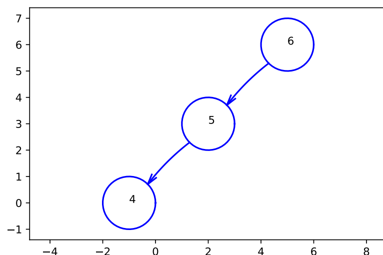
Figure – Un arbre binaire qui n'est pas un ABR



Passage liste ordonnée/ABR

A une liste ordonnée correspondent plusieurs ABR.

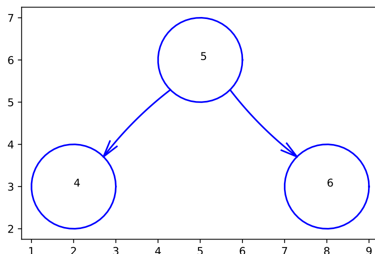
FIGURE – Un ABR pour représenter **[4,5,6]**



Passage liste ordonnée/ABR

A une liste ordonnée correspondent plusieurs ABR.

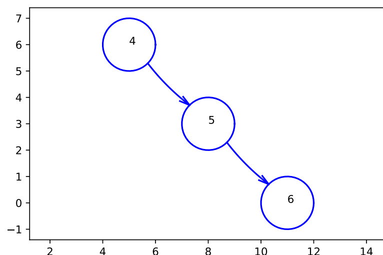
FIGURE – Un ABR équilibré pour représenter **[4,5,6]**



Passage liste ordonnée/ABR

A une liste ordonnée correspondent plusieurs ABR.

FIGURE – Un ABR pour représenter **[4,5,6]**



1 Arbre binaire de recherche
- ABR
- Implémentation
- Rotations

2 Dictionnaires

3 Tas
- Généralités
- Représentation par tableaux
- Opérations

4 Files de priorités (une application des tas)

Type de données

Nous utilisons le type suivant :

```
type 'a tree =
    | Nil
    | N of 'a * ( 'a tree ) * ('a tree);;
```

- Une feuille est implémentée par **N(x, Nil, Nil)**,

Type de données

Nous utilisons le type suivant :

```
type 'a tree =
    | Nil
    | N of 'a * ( 'a tree ) * ('a tree);;
```

- Une feuille est implémentée par **N(x, Nil, Nil)**,
- une nœud d'arité 1 par **N(x, t, Nil)** ou **N(x, Nil, t)** avec **x** et **t** de type convenable.

Type de données

Nous utilisons le type suivant :

```
type 'a tree =
    | Nil
    | N of 'a * ( 'a tree ) * ('a tree);;
```

- Une feuille est implémentée par **N(x, Nil, Nil)**,
- une nœud d'arité 1 par **N(x, t, Nil)** ou **N(x, Nil, t)** avec **x** et **t** de type convenable.
- Une telle structure modélise les arbres binaires, pas seulement les ABR. C'est lors de la *création* d'un arbre que nous ferons attention à ce qu'il respecte la contrainte d'ordre.

Primitives

- Une fonction de création d'un ABR à partir d'une liste.

Primitives

- Une fonction de création d'un ABR à partir d'une liste.
- une fonction d'insertion d'une valeur dans un ABR.

Primitives

- Une fonction de création d'un ABR à partir d'une liste.
- une fonction d'insertion d'une valeur dans un ABR.
- une fonction de recherche d'une valeur dans un arbre.

Primitives

- Une fonction de création d'un ABR à partir d'une liste.
- une fonction d'insertion d'une valeur dans un ABR.
- une fonction de recherche d'une valeur dans un arbre.
- une fonction de suppression d'une valeur dans un arbre.

Insérer sous une feuille

```
1 let rec insert x t = match t with
2   | Nil -> N(x,Nil, Nil)
3   | N(y,g,d) when x<y -> N(y, insert x g, d)
4   | N(y, g, d) when x >y -> N(y,g, insert x d)
5   | _ -> t (*pas de doublon*);;(*le laisser sinon
      Warning 8.*)
```

- Le choix qui est fait ici est celui d'un ABR sans étiquettes de mêmes valeurs (pas de doublon).

Insérer sous une feuille

```
1  let rec insert x t = match t with
2     | Nil -> N(x,Nil, Nil)
3     | N(y,g,d) when x<y -> N(y, insert x g, d)
4     | N(y, g, d) when x >y -> N(y,g, insert x d)
5     | _ -> t (*pas de doublon*);;(*le laisser sinon
        Warning 8.*)
```

- Le choix qui est fait ici est celui d'un ABR sans étiquettes de mêmes valeurs (pas de doublon).
- On insére la nouvelle valeur sous une feuille.

Insérer sous une feuille

```
1  let rec insert x t = match t with
2     | Nil -> N(x,Nil, Nil)
3     | N(y,g,d) when x<y -> N(y, insert x g, d)
4     | N(y, g, d) when x >y -> N(y,g, insert x d)
5     | _ -> t (*pas de doublon*);;(*le laisser sinon
          Warning 8.*)
```

- Le choix qui est fait ici est celui d'un ABR sans étiquettes de mêmes valeurs (pas de doublon).
- On insére la nouvelle valeur sous une feuille.
- On pourrait aussi insérer x à la racine :

Insérer sous une feuille

```
1 let rec insert x t = match t with
2   | Nil -> N(x,Nil, Nil)
3   | N(y,g,d) when x<y -> N(y, insert x g, d)
4   | N(y, g, d) when x >y -> N(y,g, insert x d)
5   | _ -> t (*pas de doublon*);;(*le laisser sinon
      Warning 8.*)
```

- Le choix qui est fait ici est celui d'un ABR sans étiquettes de mêmes valeurs (pas de doublon).
- On insére la nouvelle valeur sous une feuille.
- On pourrait aussi insérer x à la racine :
  - ≪ Couper ≫ l'arbre en deux sous-ABR g, d contenant respectivement les éléments plus petits et plus grands que x.

Insérer sous une feuille

```
1  let rec insert x t = match t with
2    | Nil -> N(x,Nil, Nil)
3    | N(y,g,d) when x<y -> N(y, insert x g, d)
4    | N(y, g, d) when x >y -> N(y,g, insert x d)
5    | _ -> t (*pas de doublon*);;(*le laisser sinon
        Warning 8.*)
```

- Le choix qui est fait ici est celui d'un ABR sans étiquettes de mêmes valeurs (pas de doublon).
- On insére la nouvelle valeur sous une feuille.
- On pourrait aussi insérer x à la racine :
  - « Couper » l'arbre en deux sous-ABR g, d contenant respectivement les éléments plus petits et plus grands que x.
  - construire l'arbre $N(x, g, d)$

Complexité de l'insertion sous une feuille

Description informelle pour un arbre de heuteur h à n nœuds.

- On descend le long d'une branche jusqu'à la feuille.

Complexité de l'insertion sous une feuille

Description informelle pour un arbre de heuteur h à n nœuds.

- On descend le long d'une branche jusqu'à la feuille.
- Il y a $O(h)$ pour cette descente. Pour chaque nœud interne, les opérations hors appel récursif sont à coût constant.

Complexité de l'insertion sous une feuille

Description informelle pour un arbre de heuteur h à n nœuds.

- On descend le long d'une branche jusqu'à la feuille.
- Il y a $O(h)$ pour cette descente. Pour chaque nœud interne, les opérations hors appel récursif sont à coût constant.
- Dans le cas d'arrêt, le coût est également constant.

Complexité de l'insertion sous une feuille

Description informelle pour un arbre de heuteur h à n nœuds.

- On descend le long d'une branche jusqu'à la feuille.
- Il y a $O(h)$ pour cette descente. Pour chaque nœud interne, les opérations hors appel récursif sont à coût constant.
- Dans le cas d'arrêt, le coût est également constant.
- Donc complexité en $O(h)$.

Complexité de l'insertion sous une feuille

Description informelle pour un arbre de heuteur h à n nœuds.

- On descend le long d'une branche jusqu'à la feuille.
- Il y a $O(h)$ pour cette descente. Pour chaque nœud interne, les opérations hors appel récursif sont à coût constant.
- Dans le cas d'arrêt, le coût est également constant.
- Donc complexité en $O(h)$.
- On comprend l'intérêt de « contrôler » h. En pratique, on essaye de conserver $h \leq C \log n$ pour une certaine constante C. Si on arrive à maintenir cette contrainte au fil des insertions, on aobtient un *arbre équilibré*).

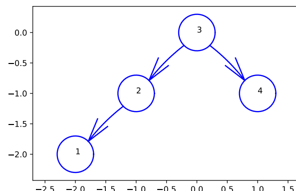
Création

Créer un ABR à partir d'une liste :

```
let rec create l = match l with
  | [] -> Nil
  | e::q -> insert e (create q);;

create([1;4;2;3]);;
```

FIGURE – ABR obtenu par **create([1 ;4 ;2 ;3]) ; ;**

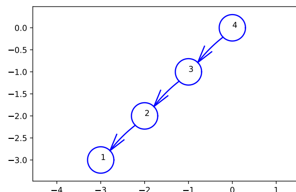


Création

Liste triée

Si la liste est déjà triée, on obtient une liste chaînée.

FIGURE – ABR obtenu par **create([1 ;2 ;3 ;4]) ; ;**



Complexité de la création

- Si la liste est déjà triée, l'ABR n'a qu'une branche.

Complexité de la création

- Si la liste est déjà triée, l'ABR n'a qu'une branche.
- Pour une liste triée de longueur n, la complexité vérifie une relation de la forme

$$
\begin{aligned}
c_n &= c_{n-1} + \underbrace{n}_{\text{cpx insertion}} + \underbrace{c}_{\text{autres opérations}} \\
&= c_{n-2} + 2c + n + \underbrace{n-1}_{\text{cpx insertion}} = \dots \\
&= c_0 + nc + \sum_{k=1}^{n} k = \Theta(n^2)
\end{aligned}
$$

Complexité de la création

- Si la liste est déjà triée, l'ABR n'a qu'une branche.
- Pour une liste triée de longueur n, la complexité vérifie une relation de la forme

$$\begin{aligned} c_n &= c_{n-1} + \underbrace{n}_{\text{cpx insertion}} + \underbrace{c}_{\text{autres opérations}} \\ &= c_{n-2} + 2c + n + \underbrace{n-1}_{\text{cpx insertion}} = \dots \\ &= c_0 + nc + \sum_{k=1}^{n} k = \Theta(n^2) \end{aligned}$$

- On voit l'intérêt de maintenir un certain équilibre dans l'arbre au moment des insertions.

Utilité de l'équilibrage des arbres

- A priori, cela ne semble pas si grave d'avoir une liste chaînée et non un bel arbre binaire « équilibré ».

Utilité de l'équilibrage des arbres

- A priori, cela ne semble pas si grave d'avoir une liste chaînée et non un bel arbre binaire « équilibré ».
- Mais les opérations sur les ABR (insertion, suppression, recherche) ont une complexité au pire qui dépend de la hauteur...

Utilité de l'équilibrage des arbres

- A priori, cela ne semble pas si grave d'avoir une liste chaînée et non un bel arbre binaire ≪ équilibré ≫.
- Mais les opérations sur les ABR (insertion, suppression, recherche) ont une complexité au pire qui dépend de la hauteur...
- ... d'où l'idée qu'il faut limiter la hauteur des arbres lors de l'insertion.

Utilité de l'équilibrage des arbres

- A priori, cela ne semble pas si grave d'avoir une liste chaînée et non un bel arbre binaire « équilibré ».
- Mais les opérations sur les ABR (insertion, suppression, recherche) ont une complexité au pire qui dépend de la hauteur...
- ... d'où l'idée qu'il faut limiter la hauteur des arbres lors de l'insertion.
- C'est le principe du *rééquilibrage* des ABR.

Création d'un arbre équilibré à partir d'une liste

- Un arbre A est dit *équilibré* lorsque $h(A) = O(\log(|A|))$.

Création d'un arbre équilibré à partir d'une liste

- Un arbre A est dit *équilibré* lorsque $h(A) = O(\log(|A|))$.
- Exemple arbres AVL : pour chaque nœud, la différence entre les hauteurs de ses fils (l'un éventuellement vide) est $0, 1$ ou -1.
  On peut établir que, dans un AVL de taille $|A| = n$, on a $\frac{3}{2}\log_2(n+1) \geq h(A)$ (voir exercice en TD).

Création d'un arbre équilibré à partir d'une liste

- Un arbre A est dit *équilibré* lorsque $h(A) = O(\log(|A|))$.
- Exemple arbres AVL : pour chaque nœud, la différence entre les hauteurs de ses fils (l'un éventuellement vide) est $0, 1$ ou -1.
  On peut établir que, dans un AVL de taille $|A| = n$, on a $\frac{3}{2}\log_2(n+1) \geq h(A)$ (voir exercice en TD).
- Autre exemple : arbres Rouge-Noir (voir (TD)).

Création d'un arbre équilibré à partir d'une liste

- Dans un arbre équilibré, insérer x se fait en $O(\log_2 n)$ appels internes au pire plus un nombre borné d'autres opérations en $\Theta(1)$.

Création d'un arbre équilibré à partir d'une liste

- Dans un arbre équilibré, insérer x se fait en $O(\log_2 n)$ appels internes au pire plus un nombre borné d'autres opérations en $\Theta(1)$.
- Si on maintient le caractère équilibré (le rééquilibrage a un coût logarithmique -admis-), le coût de la création à partir d'une liste est donc de l'ordre de

$$\Theta\left(\sum_{k=1}^{n} \log_2(k)\right) = \Theta\left(\log_2(n!)\right) = \Theta(n \log_2 n)$$

Rechercher

```
1 let rec search x t = match t with
2     (*cherche x dans t*)
3   | Nil -> false
4   | N(y,_,_) when y = x-> true
5   | N(y,g,_) when x < y -> search x g
6   | N(y,_,d) when x > y -> search x d
7   | _ -> false;;(*le laisser sinon 'this pattern-
      matching is not exhaustive.'*)
```

- Si x est égal à la racine de t, c'est bon. Sinon on cherche récursivement dans le sous arbre gauche lorsque x est plus petit que la racine, et à droite sinon.

Rechercher

```
1  let rec search x t = match t with
2      (*cherche x dans t*)
3    | Nil -> false
4    | N(y,_,_) when y = x-> true
5    | N(y,g,_) when x < y -> search x g
6    | N(y,_,d) when x > y -> search x d
7    | _ -> false;;(*le laisser sinon 'this pattern-
       matching is not exhaustive.'*)
```

- Si x est égal à la racine de t, c'est bon. Sinon on cherche récursivement dans le sous arbre gauche lorsque x est plus petit que la racine, et à droite sinon.
- Si x est à la profondeur k, il y a k appels internes pour le trouver.

Rechercher

```
1  let rec search x t = match t with
2      (*cherche x dans t*)
3    | Nil -> false
4    | N(y,_,_) when y = x-> true
5    | N(y,g,_) when x < y -> search x g
6    | N(y,_,d) when x > y -> search x d
7    | _ -> false;;(*le laisser sinon 'this pattern-
       matching is not exhaustive.'*)
```

- Si x est égal à la racine de t, c'est bon. Sinon on cherche récursivement dans le sous arbre gauche lorsque x est plus petit que la racine, et à droite sinon.
- Si x est à la profondeur k, il y a k appels internes pour le trouver.
- Si x n'est pas dans l'arbre, il y a au pire $h(t)$ appels internes.

Suppression

Opération de fusion

On veut « fusionner » deux ABR **G** et **D** tels que les étiquettes de **G** sont toutes plus petites que celles de **D**. Ceci afin d'obtenir un ABR unique construit à partir des nœuds des deux arbres.

```
let rec merge a b = match a,b with
  (*fusion qui privilégie l'arbre gauche*)
  | Nil,t |  t, Nil -> t
  | N(x,ga,da), N(y,gb,db)-> (*on a max a <= min b*)
             N(x, ga, N(y, merge da gb, db));;
```

Dans cette fusion, la racine de l'arbre gauche devient systématiquement la racine de l'arbre retourné. On aurait pu privilégier l'arbre droit.

Suppression

Opération de fusion

```
let a1 = N(3,N(2,Nil, Nil),N(4, Nil, Nil)) in let
a2 = N(30,N(20,Nil, Nil),N(40, Nil, Nil)) in
merge a1 a2;;
```

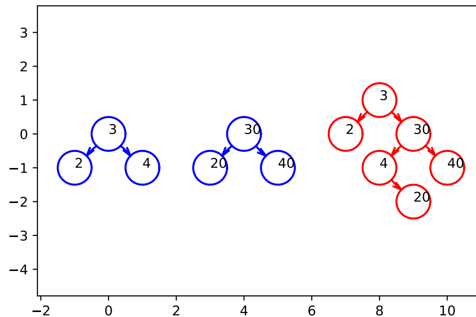
Et on obtient :

```
N (3, N (2, Nil, Nil),
      N (30, N (4 , Nil, N (20, Nil, Nil)),
             N (40, Nil, Nil)))
```

Suppression

Opération de fusion

FIGURE – En rouge, la fusion des deux arbres bleus



Correction de la fusion

Preuve par induction. On montre que si on fusionne deux ABR a, b tels que $\max a \leq \min b$, alors le nouvel arbre formé est un ABR contenant toutes les étiquettes de a, b.

- Cas de base : si un des deux arbres est vide, on renvoie l'autre. C'est un ABR par hyp. et il contient bien toutes les étiquettes des deux arbres.

Correction de la fusion

Preuve par induction. On montre que si on fusionne deux ABR a, b tels que $\max a \leq \min b$, alors le nouvel arbre formé est un ABR contenant toutes les étiquettes de a, b.

- Cas de base : si un des deux arbres est vide, on renvoie l'autre. C'est un ABR par hyp. et il contient bien toutes les étiquettes des deux arbres.
- Hérédité. Soient $a = N(x, g_a, d_a)$ et $b = N(y, g_b, d_b)$ non vides avec $\max a \leq \min b$.

Correction de la fusion

Preuve par induction. On montre que si on fusionne deux ABR a, b tels que $\max a \leq \min b$, alors le nouvel arbre formé est un ABR contenant toutes les étiquettes de a, b.

- Cas de base : si un des deux arbres est vide, on renvoie l'autre. C'est un ABR par hyp. et il contient bien toutes les étiquettes des deux arbres.
- Hérédité. Soient $a = N(x, g_a, d_a)$ et $b = N(y, g_b, d_b)$ non vides avec $\max a \leq \min b$.
- Notre hypothèse d'induction (HI) est que la fusion d'un sous-terme immédiats de a avec un sous-terme immédiat de b est un ABR contenant toutes les étiquettes de ses deux sous-termes.

Correction de la fusion

Preuve par induction. On montre que si on fusionne deux ABR a, b tels que $\max a \leq \min b$, alors le nouvel arbre formé est un ABR contenant toutes les étiquettes de a, b.

- Cas de base : si un des deux arbres est vide, on renvoie l'autre. C'est un ABR par hyp. et il contient bien toutes les étiquettes des deux arbres.
- Hérédité. Soient $a = N(x, g_a, d_a)$ et $b = N(y, g_b, d_b)$ non vides avec $\max a \leq \min b$.
- Notre hypothèse d'induction (HI) est que la fusion d'un sous-terme immédiats de a avec un sous-terme immédiat de b est un ABR contenant toutes les étiquettes de ses deux sous-termes.
- La racine du nouvel arbre est x et elle est plus grande que toutes les étiquettes de son fils gauche g_a (puisque a est un ABR)

Correction de la fusion

Hérédité (suite)

- Le fils droit est $d = N(y, \text{merge}(d_a, g_b), d_b)$. Par (HI) la fusion $\text{merge}(d_a, g_b)$ est un ABR avec toutes les étiquettes de d_a, g_b.

Correction de la fusion

Hérédité (suite)

- Le fils droit est $d = N(y, \text{merge}(d_a, g_b), d_b)$. Par (HI) la fusion $\text{merge}(d_a, g_b)$ est un ABR avec toutes les étiquettes de d_a, g_b.
- Comme y est supérieur aux étiquettes de g_b, elles-mêmes plus grandes que celles de d_a, il vient que y est supérieur aux étiquettes de l'ABR résultant de la fusion.

Correction de la fusion

Hérédité (suite)

- Le fils droit est $d = N(y, \mathrm{merge}(d_a, g_b), d_b)$. Par (HI) la fusion $\mathrm{merge}(d_a, g_b)$ est un ABR avec toutes les étiquettes de d_a, g_b.
- Comme y est supérieur aux étiquettes de g_b, elles-mêmes plus grandes que celles de d_a, il vient que y est supérieur aux étiquettes de l'ABR résultant de la fusion.
- Or y est inférieur aux étiquettes de d_b puisque b est un ABR. Donc $d = N(y, \mathrm{merge}(d_a, g_b), d_b)$ est un ABR dont toutes les étiquettes sont au moins plus grandes que celles de la plus petite de d_a. De plus il contient y et toutes les étiquettes de d_a, g_b, d_b.

Correction de la fusion

Hérédité (suite)

- Le fils droit est $d = N(y, \text{merge}(d_a, g_b), d_b)$. Par (HI) la fusion $\text{merge}(d_a, g_b)$ est un ABR avec toutes les étiquettes de d_a, g_b.
- Comme y est supérieur aux étiquettes de g_b, elles-mêmes plus grandes que celles de d_a, il vient que y est supérieur aux étiquettes de l'ABR résultant de la fusion.
- Or y est inférieur aux étiquettes de d_b puisque b est un ABR. Donc $d = N(y, \text{merge}(d_a, g_b), d_b)$ est un ABR dont toutes les étiquettes sont au moins plus grandes que celles de la plus petite de d_a. De plus il contient y et toutes les étiquettes de d_a, g_b, d_b.
- Donc x est plus petit que les étiquettes de d (qui est un ABR) et plus grand que celles de g_a (qui est un ABR) donc $N(x, g_a, d)$ est un ABR avec toutes les étiquettes de a, b.

Complexité de la fusion

Soient deux ABR g, d :

- Il y a au plus autant d'appels internes que le minimum de hauteur des sous arbres.

Complexité de la fusion

Soient deux ABR g, d :

- Il y a au plus autant d'appels internes que le minimum de hauteur des sous arbres.
- La complexité est en $\Theta(\min(h(a), h(b))$ où $h(g)$ et $h(d)$ sont les hauteurs respectives de a, b.

Complexité de la fusion

Soient deux ABR g, d :

- Il y a au plus autant d'appels internes que le minimum de hauteur des sous arbres.
- La complexité est en $\Theta(\min(h(a), h(b))$ où $h(g)$ et $h(d)$ sont les hauteurs respectives de a, b.
- La fusion est une fonction auiliaire utilisée dans le cadre de la suppression d'un nœud d'étiquette donnée.
  Dans ce cas, les deux arbres à fusionner sont des sous-arbres de l'ABR initial. En notant h la hauteur de l'ABR initial, on a $h(g) = O(h)$ et $h(d) = O(h)$. Il vient donc que la fusion a une complexité temporelle en $\Theta(h)$.

Suppression

Code

```
1 let rec remove x t =match t with
2     (*on commence par chercher x dans t*)
3   | Nil -> failwith "x not found" (*on n'a pas trouvé
      x*)
4   | N(y,g,d) when x < y-> N(y, remove x g, d)
5   | N(y,g,d) when x > y-> N(y, g, remove x d)
6     (*A partir d'ici, on a trouvé x*)
7   | N(y,g,d) when y=x -> merge g d (*fusion des deux
      sous-arbres*)
8   | _-> failwith "ne devrait pas arriver";;
```

Remarques :

- Lorsque le nœud **A** d'étiquette **x** n'a qu'un fils, celui-ci prend la place de son père (on le « remonte »).

Suppression

Code

```
1  let rec remove x t =match t with
2       (*on commence par chercher x dans t*)
3     | Nil -> failwith "x not found" (*on n'a pas trouvé
        x*)
4     | N(y,g,d) when x < y-> N(y, remove x g, d)
5     | N(y,g,d) when x > y-> N(y, g, remove x d)
6       (*A partir d'ici, on a trouvé x*)
7     | N(y,g,d) when y=x -> merge g d (*fusion des deux
        sous-arbres*)
8     | _-> failwith "ne devrait pas arriver";;
```

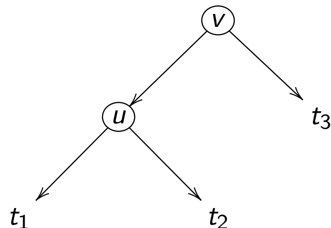
Remarques :

- Lorsque le nœud **A** d'étiquette **x** n'a qu'un fils, celui-ci prend la place de son père (on le « remonte »).
- Si **A** est une feuille, on se contente de la supprimer (la fusion met **Empty** à la place de **A**).

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

Rotation droite

Soit un ABR avec 2 nœuds contenant les valeurs u, v et 3 sous-arbres t_1, t_2, t_3

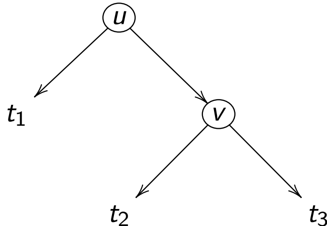


On a (en notant de façon abusive)

$$t_1 < u < t_2 < v < t_3$$

Rotation droite

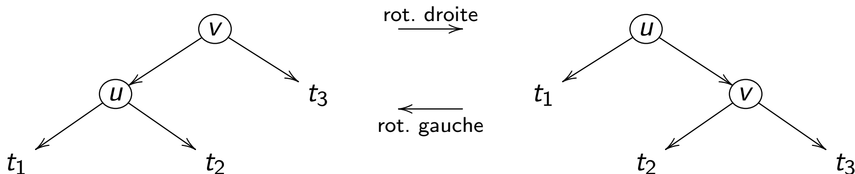
On peut réorganiser l'arbre en conservant la propriété d'ABR :



L'inégalité $t_1 < u < t_2 < v < t_3$ assure qu'il s'agit bien d'un ABR.
Cette transformation s'appelle *rotation droite* (car les 2 nœuds u, v ont été déplacés vers la droite).

Rotations droite et gauche

L'opération inverse est également possible



Rééquilibrage

```
let rotate_right = function
    | N(N(t1,u,t2),v,t3) -> N(t1,u,N(t2,v,t3))
    | t -> t (*si opération impossible, renvoyer t*)


let rotate_left t = match t with
    | N(t1,u,N(t2,v,t3)) -> N(N(t1,u,t2),v,t3)
    | _ -> t (*si opération impossible, renvoyer t*)
```

- Ces deux rotations constituent l'outil principal de rééquilibage des ABR.

Rééquilibrage

```
1 let rotate_right = function
2      | N(N(t1,u,t2),v,t3) -> N(t1,u,N(t2,v,t3))
3      | t -> t (*si opération impossible, renvoyer t*)
4
5
6 let rotate_left t = match t with
7      | N(t1,u,N(t2,v,t3)) -> N(N(t1,u,t2),v,t3)
8      | _ -> t (*si opération impossible, renvoyer t*)
```

- Ces deux rotations constituent l'outil principal de rééquilibage des ABR.
- Lorsqu'un arbre se retouve déséquilibré parce qu'il commece à « trop pencher d'un côté », on utilise une ou plusieurs rotations pour le ramener à un arbre plus équilibré.

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.
- Chaque clef est associée à une valeur unique : un dictionnaire correspond donc à une application en mathématiques.

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.
- Chaque clef est associée à une valeur unique : un dictionnaire correspond donc à une application en mathématiques.
- Peut être vu comme une généralisation du tableau dont les indices ne serait pas nécessairement des entiers.

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.
- Chaque clef est associée à une valeur unique : un dictionnaire correspond donc à une application en mathématiques.
- Peut être vu comme une généralisation du tableau dont les indices ne serait pas nécessairement des entiers.
- Opérations usuellement fournies :

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.
- Chaque clef est associée à une valeur unique : un dictionnaire correspond donc à une application en mathématiques.
- Peut être vu comme une généralisation du tableau dont les indices ne serait pas nécessairement des entiers.
- Opérations usuellement fournies :
    - ajout : association d'une nouvelle valeur à une nouvelle clef ;

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.
- Chaque clef est associée à une valeur unique : un dictionnaire correspond donc à une application en mathématiques.
- Peut être vu comme une généralisation du tableau dont les indices ne serait pas nécessairement des entiers.
- Opérations usuellement fournies :
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;
  - modification : association d'une nouvelle valeur à une ancienne clef ;

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.
- Chaque clef est associée à une valeur unique : un dictionnaire correspond donc à une application en mathématiques.
- Peut être vu comme une généralisation du tableau dont les indices ne serait pas nécessairement des entiers.
- Opérations usuellement fournies :
    - ajout : association d'une nouvelle valeur à une nouvelle clef ;
    - modification : association d'une nouvelle valeur à une ancienne clef ;
    - suppression : suppression d'une clef ;

Dictionnaire

- *Tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) : type de données associant à un ensemble de *clefs* un ensemble correspondant de *valeurs*.
- Chaque clef est associée à une valeur unique : un dictionnaire correspond donc à une application en mathématiques.
- Peut être vu comme une généralisation du tableau dont les indices ne serait pas nécessairement des entiers.
- Opérations usuellement fournies :
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;
  - modification : association d'une nouvelle valeur à une ancienne clef ;
  - suppression : suppression d'une clef ;
  - recherche : détermination de la valeur associée à une clef, si elle existe.

Dictionnaire

- Les dictionnaires peuvent être implémentés concrètement par des ABR. Ce sont alors des données persistantes.

Dictionnaire

- Les dictionnaires peuvent être implémentés concrètement par des ABR. Ce sont alors des données persistantes.
- L'ensemble des clés doit être totalement ordonné.

Dictionnaire

- Les dictionnaires peuvent être implémentés concrètement par des ABR. Ce sont alors des données persistantes.
- L'ensemble des clés doit être totalement ordonné.
- Les étiquettes des nœuds de l'ABR sont des couples (clés, valeurs) et le placement d'un nœud dans l'arbre est fait selon sa clé et non sur sa valeur.

Dictionnaire

- Les dictionnaires peuvent être implémentés concrètement par des ABR. Ce sont alors des données persistantes.
- L'ensemble des clés doit être totalement ordonné.
- Les étiquettes des nœuds de l'ABR sont des couples (clés, valeurs) et le placement d'un nœud dans l'arbre est fait selon sa clé et non sur sa valeur.
- En OCAML, explorons 3 façons de définir les dictionnaires.

Dictionnaires

Dictionnaires par liste de paires

Méthode la plus simple. Persistante.

```
(*dictionnaires par liste de paires*)
let m = ["Sally Smart", "555-9999";
         "John Doe", "555-1212";
         "J. Random Hacker", "553-1337"];;

List.assoc "John Doe" m;;
(*#   - : string = "555-1212"*)
```

Dictionnaires

Dictionnaires par AVL

- Les arbres AVL ont été historiquement les premiers arbres binaires de recherche automatiquement équilibrés.

Dictionnaires

Dictionnaires par AVL

- Les arbres AVL ont été historiquement les premiers arbres binaires de recherche automatiquement équilibrés.
- Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un.

Dictionnaires

Dictionnaires par AVL

- Les arbres AVL ont été historiquement les premiers arbres binaires de recherche automatiquement équilibrés.
- Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un.
- La recherche, l'insertion et la suppression sont toutes en $O(\log_2(n))$ dans le pire des cas.

Dictionnaires

Dictionnaires par AVL

- Les arbres AVL ont été historiquement les premiers arbres binaires de recherche automatiquement équilibrés.
- Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un.
- La recherche, l'insertion et la suppression sont toutes en $O(\log_2(n))$ dans le pire des cas.
- L'insertion et la suppression nécessitent d'effectuer des *rotations* (opérations de rééquilibrage).

Dictionnaires

Dictionnaires par AVL

- Les arbres AVL ont été historiquement les premiers arbres binaires de recherche automatiquement équilibrés.
- Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un.
- La recherche, l'insertion et la suppression sont toutes en $O(\log_2(n))$ dans le pire des cas.
- L'insertion et la suppression nécessitent d'effectuer des *rotations* (opérations de rééquilibrage).
- Le nom AVL vient des deux inventeurs Georgii Adelson-Velsky et Evguenii Landis (1962).

Dictionnaires

Dictionnaires par AVL

```
(*dictionnaire applicatif réalisé par arbres équilibrés*)
 include (Map.Make(String));;

 let m = empty
           |> add "Sally Smart" "555-9999"
           |> add "John Doe" "555-1212"
           |> add "J. Random Hacker" "553-1337";;

find "John Doe" m;;
(*#   - : string = "555-1212"*)
```

Structure persistante basée sur les arbres équilibrés.
Ajout/Suppression/Recherche en temps logarithmique.

Dictionnaires

Dictionnaires par table de hachage

```
1
2  (*dictionnaires par table de hachage polymorphe*)
3  let m = Hashtbl.create 3;;(*taille attendue 3, ça peut
       changer*)
4  Hashtbl.add m "Sally Smart" "555-9999";
5  Hashtbl.add m "John Doe" "555-1212";
6  Hashtbl.add m "J. Random Hacker" "553-1337";;
7
8  Hashtbl.find m "John Doe";;
9  (*#   - : string = "555-1212"*)
```

Structure impérative. Modifications en place.
Ajout/Suppression/Recherche en temps constant (en moyenne pour ajout, et pour la recherche, ça dépend en fait de la fonction de hash).

1 Arbre binaire de recherche
- ABR
- Implémentation
- Rotations

2 Dictionnaires

3 Tas
- Généralités
- Représentation par tableaux
- Opérations

4 Files de priorités (une application des tas)

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

Définition

tas-max

On se place dans un type totalement ordonné.

Définition

Un *tas-max* de hauteur h est un arbre binaire vérifiant :

- **Condition d'ordre** : les fils d'un nœud ont une étiquette inférieure ou égale à celle du père.

Définition

tas-max

On se place dans un type totalement ordonné.

Définition

Un *tas-max* de hauteur h est un arbre binaire vérifiant :

- **Condition d'ordre** : les fils d'un nœud ont une étiquette inférieure ou égale à celle du père.
- **Condition de structure** : Un tas est *complet gauche* :

Définition

tas-max

On se place dans un type totalement ordonné.

Définition

Un *tas-max* de hauteur *h* est un arbre binaire vérifiant :

- **Condition d'ordre** : les fils d'un nœud ont une étiquette inférieure ou égale à celle du père.
- **Condition de structure** : Un tas est *complet gauche* :
  - Tous les niveaux sont remplis sauf éventuellement le dernier.

Définition

tas-max

On se place dans un type totalement ordonné.

Définition

Un *tas-max* de hauteur h est un arbre binaire vérifiant :

- **Condition d'ordre** : les fils d'un nœud ont une étiquette inférieure ou égale à celle du père.
- **Condition de structure** : Un tas est *complet gauche* :
  - Tous les niveaux sont remplis sauf éventuellement le dernier.
  - Le dernier niveau (éventuellement incomplet) est rempli sans trou en partant de la gauche.

Précisions et conséquences

- Un arbre du type précédent est dit de type *tas-max* (racine=max).

Précisions et conséquences

- Un arbre du type précédent est dit de type *tas-max* (racine=max).
- *tas-min* : la condition d'ordre deveint « l'étiquette du père est plus petite que celle des fils. »

Précisions et conséquences

- Un arbre du type précédent est dit de type *tas-max* (racine=max).
- *tas-min* : la condition d'ordre deveint « l'étiquette du père est plus petite que celle des fils. »
- les branches sont toutes de longueur h ou $h - 1$,

Précisions et conséquences

- Un arbre du type précédent est dit de type *tas-max* (racine=max).
- *tas-min* : la condition d'ordre deveint « l'étiquette du père est plus petite que celle des fils. »
- les branches sont toutes de longueur h ou $h-1$,
- enlever les feuilles de profondeur h donne un arbre parfait,

Précisions et conséquences

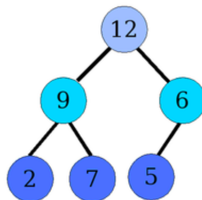
- Un arbre du type précédent est dit de type *tas-max* (racine=max).
- *tas-min* : la condition d'ordre deveint « l'étiquette du père est plus petite que celle des fils. »
- les branches sont toutes de longueur h ou $h-1$,
- enlever les feuilles de profondeur h donne un arbre parfait,
- les nœuds internes de profondeur $h-1$ d'arité ≥ 1 sont à gauche des feuilles de profondeur $h-1$,

Précisions et conséquences

- Un arbre du type précédent est dit de type *tas-max* (racine=max).
- *tas-min* : la condition d'ordre deveint « l'étiquette du père est plus petite que celle des fils. »
- les branches sont toutes de longueur h ou $h-1$,
- enlever les feuilles de profondeur h donne un arbre parfait,
- les nœuds internes de profondeur $h-1$ d'arité ≥ 1 sont à gauche des feuilles de profondeur $h-1$,
- si il y a un nœud interne de profondeur $h-1$ avec un seul fils, son fils est une feuille et c'est le dernier nœud dans le parcours en largeur.

Exemple

FIGURE – Un tas. Si on enlève les feuilles de profondeur 2, l'arbre est parfait (cf. fig. [13] pour la représentation en array)



Tous les niveaux sont remplies, sauf le dernier, lequel est partiellement rempli en commençant par la gauche.

Contre-exemples

Les arbres suivants ne sont pas des tas :

FIGURE – Un nœud de hauteur $h-1$ et d'arité 1 est à gauche d'un nœud de hauteur $h-1$ d'arité 2

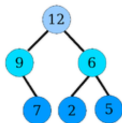
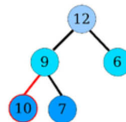


FIGURE – Un père a un fils d'étiquette plus grande que la sienne



Hauteur d'un arbre complet gauche

Soit A un arbre complet gauche à n nœuds et de hauteur p

- L'avant dernier niveau (qui correspond à un arbre parfait) est rempli. Et le dernier niveau contient au moins une feuille.

$$2^p - 1 < n \leq \underbrace{2^{p+1} - 1}_{\text{taille min. d'un arbre parfait plus gros que } A}$$

Hauteur d'un arbre complet gauche

Soit A un arbre complet gauche à n nœuds et de hauteur p

- L'avant dernier niveau (qui correspond à un arbre parfait) est rempli. Et le dernier niveau contient au moins une feuille.

$$2^p - 1 < n \leq \underbrace{2^{p+1} - 1}_{\text{taille min. d'un arbre parfait plus gros que } A}$$

- Alors

$$2^p \leq n < 2^{p+1}$$

Hauteur d'un arbre complet gauche

Soit A un arbre complet gauche à n nœuds et de hauteur p

- L'avant dernier niveau (qui correspond à un arbre parfait) est rempli. Et le dernier niveau contient au moins une feuille.

$$2^p - 1 < n \leq \underbrace{2^{p+1} - 1}_{\text{taille min. d'un arbre parfait plus gros que } A}$$

- Alors

$$2^p \leq n < 2^{p+1}$$

- Donc

$$p \leq \log_2 n < p + 1$$

Hauteur d'un arbre complet gauche

Soit A un arbre complet gauche à n nœuds et de hauteur p

- L'avant dernier niveau (qui correspond à un arbre parfait) est rempli. Et le dernier niveau contient au moins une feuille.

$$2^p - 1 < n \leq \underbrace{2^{p+1} - 1}_{\text{taille min. d'un arbre parfait plus gros que } A}$$

- Alors

$$2^p \leq n < 2^{p+1}$$

- Donc

$$p \leq \log_2 n < p + 1$$

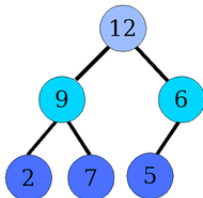
- Il vient

$$\lfloor \log_2 n \rfloor = p.$$

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

Exemple

FIGURE – Un tableau représentant un tas



BFS

Observer le lien entre le parcours en largeur et la représentation en tableau.

Contrexemples

Les arbres suivants ne sont pas des tas :

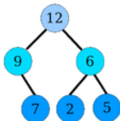


FIGURE – Un nœud de hauteur $h-1$ et d'arité 1 est à gauche d'un nœud de hauteur $h-1$ d'arité 2

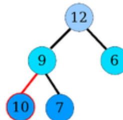


FIGURE – Un père a un fils d'étiquette plus grande que la sienne

Tas et tableaux

On peut stocker un tas dans un tableau dont on n'utilise pas (pour le moment) le premier élément.

- La racine occupe l'élément d'indice 1,

Tas et tableaux

On peut stocker un tas dans un tableau dont on n'utilise pas (pour le moment) le premier élément.

- La racine occupe l'élément d'indice 1,
- Les fils du nœud d'indice k (avec $k > 0$) sont aux indices $2k$ et $2k + 1$ (si ceux-ci ne dépassent pas la longueur du tableau).

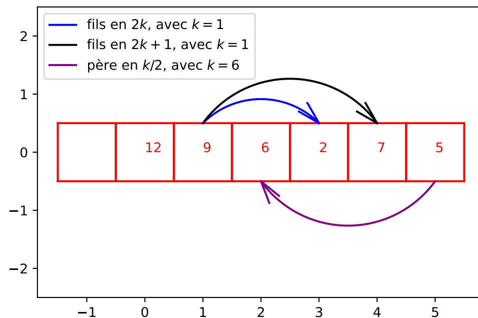
Tas et tableaux

On peut stocker un tas dans un tableau dont on n'utilise pas (pour le moment) le premier élément.

- La racine occupe l'élément d'indice 1,
- Les fils du nœud d'indice k (avec $k > 0$) sont aux indices $2k$ et $2k + 1$ (si ceux-ci ne dépassent pas la longueur du tableau).
- le père du nœud d'indice k est à l'indice $\lfloor \frac{k}{2} \rfloor$.

Tas et tableaux

FIGURE – Relations père/fils dans un tableau représentant un tas (cf. tas de la fig. [7])



Tas et tableaux

En **CAML**

```
1  (*correspond au schéma précédent en OCAML*)
2  let t = [|6; 12; 9; 6; 2; 7; 5; 0; 67; 33|];;
```

Lorsqu'on crée un tableau représentant un tas :

- Il faut prévoir la taille du tableau à l'instant initial (6 ici) et les éventuels ajouts à effectuer (en clair prévoir plus de place que la simple taille du tableau à l'instant 0).

Tas et tableaux

En **CAML**

```
1 (*correspond au schéma précédent en OCAML*)
2 let t = [|6; 12; 9; 6; 2; 7; 5; 0; 67; 33|];;
```

Lorsqu'on crée un tableau représentant un tas :

- Il faut prévoir la taille du tableau à l'instant initial (6 ici) et les éventuels ajouts à effectuer (en clair prévoir plus de place que la simple taille du tableau à l'instant 0).
- Le premier élément du tableau désigne la taille du tas (qui est différente de celle du tableau).

Tas et tableaux

En **CAML**

```
(*correspond au schéma précédent en OCAML*)
let t = [|6; 12; 9; 6; 2; 7; 5; 0; 67; 33|];;
```

Lorsqu'on crée un tableau représentant un tas :

- Il faut prévoir la taille du tableau à l'instant initial (6 ici) et les éventuels ajouts à effectuer (en clair prévoir plus de place que la simple taille du tableau à l'instant 0).
- Le premier élément du tableau désigne la taille du tas (qui est différente de celle du tableau).
- Les éléments $0, 67, 33$ en fin de tableau ne sont pas considérés comme appartenant au tas. Ils seront remplacés par les valeurs éventuellement ajoutées au tas.

Tas et tableaux

OCAML

Un problème : le type de la taille du tas (**int**) fige le type du tableau avec l'implémentation précédente. Toutes les valeurs du tas doivent être des entiers...

```
1 (*pour dissocier le type de la taille du tas
2 de celui de ses éléments,  on peut utiliser
3 un type enregistrement*)
4 type 'a myHeap =
5 {mutable length : int; heap : 'a array};;
```

1. Arbre binaire de recherche
   - ABR
   - Implémentation
   - Rotations
2. Dictionnaires
3. Tas
   - Généralités
   - Représentation par tableaux
   - Opérations
4. Files de priorités (une application des tas)

Opérations sur les tas

- Ajouter : ajout d'un élément dans le tas binaire en préservant la structure de tas.

Opérations sur les tas

- Ajouter : ajout d'un élément dans le tas binaire en préservant la structure de tas.
- Retirer : retirer un élément d'indice donné et rectifier le tableau pour qu'il corresponde de nouveau à un tas.

Opérations sur les tas

- Ajouter : ajout d'un élément dans le tas binaire en préservant la structure de tas.
- Retirer : retirer un élément d'indice donné et rectifier le tableau pour qu'il corresponde de nouveau à un tas.
- Construire : construction du tas binaire à partir d'un ensemble d'éléments.

Ajouter un élément

Principe

Considérons que l'on veuille ajouter le nœud **x** à notre tas binaire :

- On insère **x** à la prochaine position libre (la position libre la plus à gauche possible sur le dernier niveau),

Ajouter un élément

Principe

Considérons que l'on veuille ajouter le nœud **x** à notre tas binaire :

- On insère **x** à la prochaine position libre (la position libre la plus à gauche possible sur le dernier niveau),
- puis on effectue l'opération suivante (que l'on appelle *percolation* vers le haut ou *percolate-up)* pour rétablir si nécessaire la propriété d'ordre du tas binaire :

Ajouter un élément

Principe

Considérons que l'on veuille ajouter le nœud **x** à notre tas binaire :

- On insère **x** à la prochaine position libre (la position libre la plus à gauche possible sur le dernier niveau),
- puis on effectue l'opération suivante (que l'on appelle *percolation* vers le haut ou *percolate-up)* pour rétablir si nécessaire la propriété d'ordre du tas binaire :
- Tant que **x** n'est pas la racine de l'arbre et que l'étiquette de **x** est strictement supérieure à celle du père, échanger les positions entre **x** et son père.

Ajouter un élément

Shéma

On veut ajouter 50 dans un tas-max :

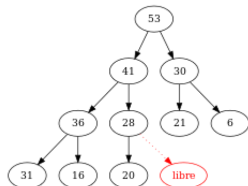
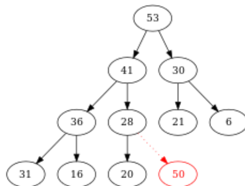


FIGURE – On cherche le seul emplacement possible pour préserver la structure d'arbre complet gauche

Ajouter un élément

Shéma

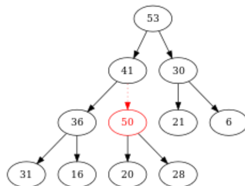
FIGURE – le nœud d'étiquette 50 est placé provisoirement. On le compare à son père (28)



Ajouter un élément

Shéma

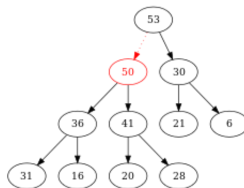
FIGURE – Comme $50 > 28$, on échange les positions de 50 et 28. Et on compare 50 avec son nouveau père (41)...



Ajouter un élément

Shéma

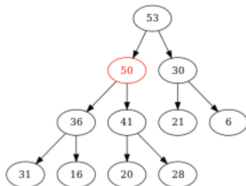
FIGURE – Comme $50 > 41$, on échange les positions de 50 et 41. Et on compare 50 avec son nouveau père (53)...



Ajouter un élément

Shéma

FIGURE – Comme $50 \leq 53$, il n'y a rien à faire : 50 a trouvé sa bonne place



Ajouter un élément

La fonction auxiliaire d'échange

```
let swap i j t=
  (*échange deux éléments d'un tableau*)
  let  c = t.(i) in t.(i)<-t.(j); t.(j)<-c;;
```

Complexité en $\Theta(1)$.

Ajouter un élément

La fonction auxiliaire de percolation

```
1  let rec percolate_up n t=
2    (*n est l'indice de l'élément à percoler*)
3    if n > 1 then
4      begin (*t.(n) n'est pas la racine*)
5       let m = n/2 (*indice du père*)
6       in if t.(m) < t.(n) then
7        begin
8          swap m n t; (*échange père/fils*)
9          percolate_up m t; (*percolate_up avec m*)
10       end;
11     end;;
```

Ajouter un élément

Complexité de la percolation

On applique cette fonction à un tas d'entiers de n nœuds (représenté par un tableau) :

- Dans le pire des cas, l'élément remonte la branche la plus longue du tas : $\log_2(n)$ étapes puisque le tas est un arbre binaire presque parfait.

Ajouter un élément

Complexité de la percolation

On applique cette fonction à un tas d'entiers de n nœuds (représenté par un tableau) :

- Dans le pire des cas, l'élément remonte la branche la plus longue du tas : $\log_2(n)$ étapes puisque le tas est un arbre binaire presque parfait.
- A chaque étape, il y a un nombre d'opérations élémentaires borné par (disons) c.

Ajouter un élément

Complexité de la percolation

On applique cette fonction à un tas d'entiers de n nœuds (représenté par un tableau) :

- Dans le pire des cas, l'élément remonte la branche la plus longue du tas : $\log_2(n)$ étapes puisque le tas est un arbre binaire presque parfait.
- A chaque étape, il y a un nombre d'opérations élémentaires borné par (disons) c.
- Au total, entre $\log_2(n)$ opérations et $c \log_2(n)$. Complexité en $\Theta(\log_2(n))$.

Ajouter un élément

La fonction d'insertion

Principe : on insère l'élément après le dernier élément du tableau ($\Theta(1)$) et on percole (au pire $\Theta(\log_2(n))$ si n est le nombre de nœud). Donc complexité au pire $\Theta(\log_2(n))$:

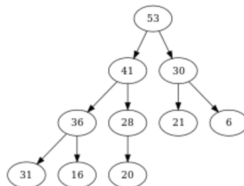
```
1  let insert v t =
2    t.(0) <- t.(0)+1;(*maj lg du tas*)
3    t.(t.(0)) <- v;(*placer v à la dernière position*)
4    percolate_up (t.(0)) t;;
```

Supprimer un élément

Shéma

On souhaite supprimer la racine du tas-max suivant :

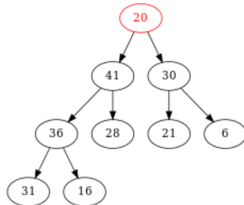
FIGURE – Le tas-max de référence



Supprimer un élément

Shéma

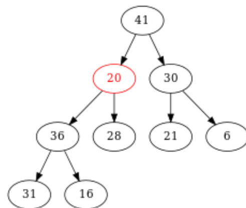
FIGURE – On remplace la racine par le dernier nœud



Supprimer un élément

Shéma

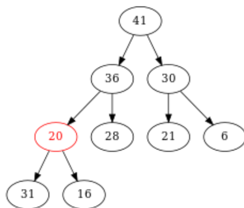
FIGURE – On compare 20 et son fils max (41). Comme $41 > 20$, on échange 41 et 20



Supprimer un élément

Shéma

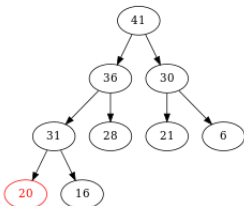
FIGURE – On compare 20 et son fils max (36). Comme $36 > 20$, on échange 36 et 20



Supprimer un élément

Shéma

FIGURE – On compare 20 et son fils max (31). Comme $31 > 20$, on échange 31 et 20. On est alors dans un des deux cas d'arrêt : plus de fils ou pas de fils plus grand. Ici, 20 n'a plus de fils. On a fini.



Supprimer un élément

Principe

On veut supprimer la racine

- Lorsqu'on supprime le dernier nœud d'un tas, celui-ci reste un tas.

Supprimer un élément

Principe

On veut supprimer la racine

- Lorsqu'on supprime le dernier nœud d'un tas, celui-ci reste un tas.
- On supprime le dernier nœud et on le met à la place du nœud racine (la propriété d'ordre est perdue).

Supprimer un élément

Principe

On veut supprimer la racine

- Lorsqu'on supprime le dernier nœud d'un tas, celui-ci reste un tas.
- On supprime le dernier nœud et on le met à la place du nœud racine (la propriété d'ordre est perdue).
- On percole vers le bas (*percolate-down*) pour retrouver la propriété d'ordre.

Supprimer un élément

Recherche du plus grand fils

```
let aux_max n t =
  (*cette fonction retourne l'indice
  du fils le plus grand de t.(n);
  -1 si pas de fils*)
  if 2 * n < t.(0) then
    begin
    (*t.(n) a deux fils*)
      if t.(2*n) < t.(2*n+1) then 2*n+1 else 2*n
    end
    else
    (* moins de deux fils *)
    begin
      if 2 * n = t.(0) then 2 * n (*un seul fils*)
      else  -1 (*pas de fils*)
    end;;
```

Complexité en $\Theta(1)$.

Supprimer un élément

Percolate-down

```
1  let rec percolate_down k t =
2    let m = aux_max k t (* m vaut -1,2k ou 2k+1*)
3    in
4    if m > -1 && t.(m) > t.(k) then
5      (* rqe : si m=-1 ou t.(m) <= t.(k)
6         on ne fait rien*)
7      begin
8           swap k m t;
9           percolate_down m t;
10     end;;
```

Supprimer

Complexité de la percolation

- A chaque appel interne on descend d'un étage dans l'arbre.

Supprimer

Complexité de la percolation

- A chaque appel interne on descend d'un étage dans l'arbre.
- Le nombre d'appel est majoré par la hauteur ($\log_2(n)$ pour n nœuds dans cet arbre complet gauche).

Supprimer

Complexité de la percolation

- A chaque appel interne on descend d'un étage dans l'arbre.
- Le nombre d'appel est majoré par la hauteur ($\log_2(n)$ pour n nœuds dans cet arbre complet gauche).
- En dehors de l'appel interne, il y a à chaque étape moins de (disons) c opérations élémentaires.

Supprimer

Complexité de la percolation

- A chaque appel interne on descend d'un étage dans l'arbre.
- Le nombre d'appel est majoré par la hauteur ($\log_2(n)$ pour n nœuds dans cet arbre complet gauche).
- En dehors de l'appel interne, il y a à chaque étape moins de (disons) c opérations élémentaires.
- Coût total entre $\log_2(n)$ et $c \log_2(n)$. Complexité en $\ominus(\log_2(n))$

Supprimer la racine

Remove

```
1  let remove t =
2        (*mettre dernier elt dans t.(1) :*)
3        t.(1) <- t.(t.(0));
4        (*chger taille du tas : *)
5        t.(0) <- t.(0) - 1;
6        percolate_down 1 t;;
```

Complexité : la même que la percolation.

Création

Par remontée : percolation haute du nœud courant

- Pour **t**, tableau de taille n, on fait une copie de taille assez grande, disons **n+1** :

6	12	8	7	15

$\mapsto$

5	6	12	8	7	15

Et on maintient « **t[1 :k+1]** *est un tas* » (notation slicing `Python`).

Création

Par remontée : percolation haute du nœud courant

- Pour **t**, tableau de taille n, on fait une copie Et on maintient « **t[1 :k+1]** *est un tas* » (notation slicing `Python`).
- $k = 1$:

5	6	12	8	7	15

t[1 :2] est un tas

Création

Par remontée : percolation haute du nœud courant

- Pour **t**, tableau de taille n, on fait une copie Et on maintient « **t[1 :k+1]** *est un tas* » (notation slicing `Python`).
- $k = 1$: | 5 | 6 | 12 | 8 | 7 | 15 | **t[1 :2]** est un tas
- $k = 2$: | 5 | 6 | 12 | 8 | 7 | 15 | On percole up 12.

  | 5 | 12 | 6 | 8 | 7 | 15 | **t[1 :3]** est un tas.

  $k = 3$: 8 est à sa place. **t[1 :4]** est un tas.

Création

Par remontée : percolation haute du nœud courant

- Pour **t**, tableau de taille n, on fait une copie Et on maintient « **t[1 :k+1]** *est un tas* » (notation slicing `Python`).
- $k = 1$:

5	6	12	8	7	15

  t[1 :2] est un tas
- $k = 2$:

5	6	12	8	7	15

  On percole up 12.

5	12	6	8	7	15

  t[1 :3] est un tas.

  $k = 3$: 8 est à sa place. **t[1 :4]** est un tas.
- $k = 4$:

5	12	6	8	7	15

  On percole up 7.

5	12	7	8	6	15

  t[1 :5] est un tas.

Création

Par remontée : percolation haute du nœud courant

- Pour **t**, tableau de taille n, on fait une copie Et on maintient « **t[1 :k+1]** *est un tas* » (notation slicing `Python`).
- $k = 1$:

5	6	12	8	7	15

t[1 :2] est un tas

- $k = 2$:

5	6	12	8	7	15

On percole up 12.

5	12	6	8	7	15

t[1 :3] est un tas.

$k = 3$: 8 est à sa place. **t[1 :4]** est un tas.

- $k = 4$:

5	12	6	8	7	15

On percole up 7.

5	12	7	8	6	15

t[1 :5] est un tas.

- $k = 5$:

5	12	7	8	6	15

On percole up 15 deux fois

Création

Par remontée : percolation haute du nœud courant

- Pour **t**, tableau de taille n, on fait une copie Et on maintient « **t[1 :k+1]** *est un tas* » (notation slicing `Python`).
- $k = 1$:

5	6	12	8	7	15

t[1 :2] est un tas

- $k = 2$:

5	6	12	8	7	15

On percole up 12.

5	12	6	8	7	15

t[1 :3] est un tas.

$k = 3$: 8 est à sa place. **t[1 :4]** est un tas.

- $k = 4$:

5	12	6	8	7	15

On percole up 7.

5	12	7	8	6	15

t[1 :5] est un tas.

- $k = 5$:

5	12	7	8	6	15

On percole up 15 deux fois

5	12	15	8	6	7

5	12	15	8	6	7

5	15	12	8	6	7

t[1 :6] est un tas.

Création

Par remontée (Complexité)

- Il y a n nœud, donc une hauteur de $p = \lfloor \log_2 n \rfloor$

Création

Par remontée (Complexité)

- Il y a n nœud, donc une hauteur de $p = \lfloor \log_2 n \rfloor$
- Pire des cas : chaque remontée aboutit à la racine.

Création

Par remontée (Complexité)

- Il y a n nœud, donc une hauteur de $p = \lfloor \log_2 n \rfloor$
- Pire des cas : chaque remontée aboutit à la racine.
- Niveau k : au plus 2^knœuds ; lesquels remontent à la racine en k étapes.

Création

Par remontée (Complexité)

- Il y a n nœud, donc une hauteur de $p = \lfloor \log_2 n \rfloor$
- Pire des cas : chaque remontée aboutit à la racine.
- Niveau k : au plus 2^knœuds ; lesquels remontent à la racine en k étapes.
-

$$\begin{aligned} C(n) &\leq \sum_{k=1}^{p} k2^k = 2\sum_{k=1}^{p} k2^{k-1} \\ &= 2\frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{k=0}^{p} x^k\right)[2] \\ &= 2\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{x^{p+1}-1}{x-1}\right)[2] = 2\left(x \mapsto \frac{-px^p + px^{p+1} - x^p + 1}{x^2 - 2x + 1}\right)[2] \end{aligned}$$

Création

Par remontée (Complexité)

- Il y a n nœud, donc une hauteur de $p = \lfloor \log_2 n \rfloor$
- Pire des cas : chaque remontée aboutit à la racine.
- Niveau k : au plus 2^knœuds ; lesquels remontent à la racine en k étapes.
- $$C(n) \leq 2(x \mapsto \frac{-px^p + px^{p+1} - x^p + 1}{x^2 - 2x + 1})[2]$$
- $$C(n) \leq 2(p2^p - 2^p + 1) \leq 2 \times p2^p = 2 \times \underbrace{2^{\lfloor \log_2 n \rfloor} \lfloor \log_2 n \rfloor}_{O(n \log_2 n)}$$

Création

Par descente : percolation basse du nœud courant

A partir d'un tableau d'entiers :

- On le considère comme un arbre complet gauche en décalant ses éléments d'un cran à droite et en insérant sa longueur.

Création

Par descente : percolation basse du nœud courant

A partir d'un tableau d'entiers :

- On le considère comme un arbre complet gauche en décalant ses éléments d'un cran à droite et en insérant sa longueur.
- Parcours du tableau : on maintient l'invariant *Tous les sous-arbres dont la racine est à droite du sommet courant sont des tas binaires*.

Création

Par descente : percolation basse du nœud courant

A partir d'un tableau d'entiers :

- On le considère comme un arbre complet gauche en décalant ses éléments d'un cran à droite et en insérant sa longueur.
- Parcours du tableau : on maintient l'invariant *Tous les sous-arbres dont la racine est à droite du sommet courant sont des tas binaires*.
- On parcourt les sommets par indices décroissants à partir du premier nœud interne (position $\lfloor n/2 \rfloor$).
  Lors de ce parcours on effectue un percolate-down à partir du nœud courant pour maintenir le status de tas.

Création

Par descente : percolation basse du nœud courant

A partir d'un tableau d'entiers :

- On le considère comme un arbre complet gauche en décalant ses éléments d'un cran à droite et en insérant sa longueur.
- Parcours du tableau : on maintient l'invariant *Tous les sous-arbres dont la racine est à droite du sommet courant sont des tas binaires*.
- On parcourt les sommets par indices décroissants à partir du premier nœud interne (position $\lfloor n/2 \rfloor$).
  Lors de ce parcours on effectue un percolate-down à partir du nœud courant pour maintenir le status de tas.
- Donc après avoir traité la racine, comme elle vérifie l'invariant, notre arbre est un tas binaire.

Création

Par descente : percolation basse du nœud courant

- Pour **t**, tableau de taille n, on fait une copie de taille disons **n+1**

6	12	8	7	15	9

$\rightarrow$

6	6	12	8	7	15	9

7,15, 9 aux indices $> \lfloor n/2 \rfloor$ sont des feuilles donc des tas.

Création

Par descente : percolation basse du nœud courant

- Pour **t**, tableau de taille n, on fait une copie de taille disons **n+1**

6	12	8	7	15	9

$\rightarrow$

6	6	12	8	7	15	9

  7,15, 9 aux indices $> \lfloor n/2 \rfloor$ sont des feuilles donc des tas.
- $k = \lfloor n/2 \rfloor = 3$ (Profondeur 1). 8 est le père de 9.

6	6	12	8	7	15	9

$\rightarrow$

6	6	12	9	7	15	8

Création

Par descente : percolation basse du nœud courant

- Pour **t**, tableau de taille n, on fait une copie de taille disons **n+1**

6	12	8	7	15	9

→

6	6	12	8	7	15	9

7,15, 9 aux indices $> \lfloor n/2 \rfloor$ sont des feuilles donc des tas.

- $k = \lfloor n/2 \rfloor = 3$ (Profondeur 1). 8 est le père de 9.

6	6	12	8	7	15	9

→

6	6	12	9	7	15	8

- $k = 2$ (Profondeur 1). 12 est père de 7 et 15

6	6	12	9	7	15	8

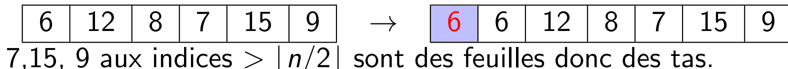
→

6	6	15	9	7	12	8

Création

Par descente : percolation basse du nœud courant

- Pour **t**, tableau de taille n, on fait une copie de taille disons **n+1**



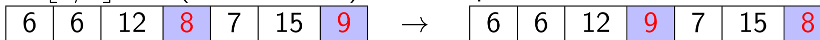
6	12	8	7	15	9

$\rightarrow$

6	6	12	8	7	15	9

7,15, 9 aux indices $> \lfloor n/2 \rfloor$ sont des feuilles donc des tas.

- $k = \lfloor n/2 \rfloor = 3$ (Profondeur 1). 8 est le père de 9.

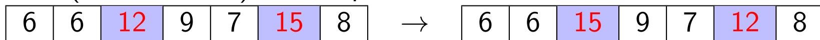


6	6	12	8	7	15	9

$\rightarrow$

6	6	12	9	7	15	8

- $k = 2$ (Profondeur 1). 12 est père de 7 et 15

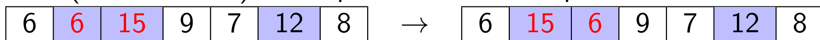


6	6	12	9	7	15	8

$\rightarrow$

6	6	15	9	7	12	8

- $k = 1$ (Profondeur 0). 6 est père de 15 et 9. 2 percos.



6	6	15	9	7	12	8

$\rightarrow$

6	15	6	9	7	12	8

$\rightarrow$

6	15	12	9	7	6	8

Création

Par descente (Complexité)

Hauteur $p = \lfloor \log_2 n \rfloor$ pour n nœuds.

- Dans le pire des cas, chaque descente d'un nœud situé au niveau k aboutit aux feuilles : $p - k$ échanges AU PLUS.

Création

Par descente (Complexité)

Hauteur $p = \lfloor \log_2 n \rfloor$ pour n nœuds.

- Dans le pire des cas, chaque descente d'un nœud situé au niveau k aboutit aux feuilles : $p - k$ échanges AU PLUS.
- Il y a au plus $\lfloor \frac{n}{2^{p-k}} \rfloor$ nœuds de profondeur k (cf. TD).

Création

Par descente (Complexité)

Hauteur $p = \lfloor \log_2 n \rfloor$ pour n nœuds.

- Dans le pire des cas, chaque descente d'un nœud situé au niveau k aboutit aux feuilles : $p - k$ échanges AU PLUS.
- Il y a au plus $\lfloor \frac{n}{2^{p-k}} \rfloor$ nœuds de profondeur k (cf. TD).
- Complexité $C(n)$ au pire. Début des descentes : niveau $p - 1$

Création

Par descente (Complexité)

Hauteur $p = \lfloor \log_2 n \rfloor$ pour n nœuds.

- Dans le pire des cas, chaque descente d'un nœud situé au niveau k aboutit aux feuilles : $p - k$ échanges AU PLUS.
- Il y a au plus $\lfloor \frac{n}{2^{p-k}} \rfloor$ nœuds de profondeur k (cf. TD).
- Complexité $C(n)$ au pire. Début des descentes : niveau $p - 1$

$$C(n) \leq \sum_{k=0}^{p-1} \underbrace{\lfloor \frac{n}{2^{p-k}} \rfloor}_{\text{nb node de hauteur k}} \overbrace{\underbrace{(p-k)}_{\text{...pour chaque node}}}^{\text{nb échanges au pire...}} \leq \sum_{k=0}^{p} \frac{n}{2^{p-k}}(p-k)$$

Création

Par descente (Complexité)

Hauteur $p = \lfloor \log_2 n \rfloor$ pour n nœuds.

- Dans le pire des cas, chaque descente d'un nœud situé au niveau k aboutit aux feuilles : $p - k$ échanges AU PLUS.
- Il y a au plus $\lfloor \frac{n}{2^{p-k}} \rfloor$ nœuds de profondeur k (cf. TD).
- Complexité $C(n)$ au pire. Début des descentes : niveau $p - 1$

$$C(n) \leq \sum_{k=0}^{p-1} \underbrace{\lfloor \frac{n}{2^{p-k}} \rfloor}_{\text{nb node de hauteur k}} \overbrace{\underbrace{(p-k)}_{\text{...pour chaque node}}}^{\text{nb échanges au pire...}} \leq \sum_{k=0}^{p} \frac{n}{2^{p-k}} (p-k)$$

$$C(n) \overbrace{\leq}^{\substack{\text{cgt. var.} \\ k=p-k}} \sum_{k=0}^{p} \frac{n}{2^k} k \leq n \overbrace{\sum_{k=0}^{p} \frac{k}{2^k}}^{\text{série CV donc majorée}} = O(n)$$

Création

Par descente (Complexité)

Hauteur $p = \lfloor \log_2 n \rfloor$ pour n nœuds.

- Dans le pire des cas, chaque descente d'un nœud situé au niveau k aboutit aux feuilles : $p - k$ échanges AU PLUS.
- Il y a au plus $\lfloor \frac{n}{2^{p-k}} \rfloor$ nœuds de profondeur k (cf. TD).
- Complexité $C(n)$ au pire. Début des descentes : niveau $p - 1$

$$C(n) \overbrace{\leq}^{\substack{\text{cgt. var.} \\ k=p-k}} \sum_{k=0}^{p} \frac{n}{2^k} k \leq n \overbrace{\sum_{k=0}^{p} \frac{k}{2^k}}^{\text{série CV donc majorée}} = O(n)$$

Donc création par descente ($O(n)$) moins coûteuse que par remontée ($O(n \log_2 n)$).

Tri (croissant) par tas

- A partir d'un tableau de n nombres on crée un tas-max **t** ;

Tri (croissant) par tas

- A partir d'un tableau de n nombres on crée un tas-max **t** ;
- Étape 1 : la racine **t[1]** est le maximum du tas, on l'échange avec **t[n]** (notation slicing `Python`).

Tri (croissant) par tas

- A partir d'un tableau de n nombres on crée un tas-max **t** ;
- Étape 1 : la racine **t[1]** est le maximum du tas, on l'échange avec **t[n]** (notation slicing `Python`).
- Le max du tableau se retrouve à la fin du tas en position **t[n]**. **t[n :]** est trié et contient le max.

Tri (croissant) par tas

- A partir d'un tableau de n nombres on crée un tas-max **t** ;
- Étape 1 : la racine **t[1]** est le maximum du tas, on l'échange avec **t[n]** (notation slicing `Python`).
- Le max du tableau se retrouve à la fin du tas en position **t[n]**. **t[n :]** est trié et contient le max.
- On met à jour la longueur du tas (qui représente le nombre d'éléments qu'il reste à trier) en la décrémentant (puisque l'ancienne racine a trouvé sa place).

Tri (croissant) par tas

- A partir d'un tableau de n nombres on crée un tas-max **t** ;
- Étape 1 : la racine **t[1]** est le maximum du tas, on l'échange avec **t[n]** (notation slicing `Python`).
- Le max du tableau se retrouve à la fin du tas en position **t[n]**. **t[n :]** est trié et contient le max.
- On met à jour la longueur du tas (qui représente le nombre d'éléments qu'il reste à trier) en la décrémentant (puisque l'ancienne racine a trouvé sa place).
- On percole bas de la nouvelle racine. Alors **t[0 :n]** est de nouveau un tas. Et on itère (swap puis percolation)...

Tri (croissant) par tas

- A partir d'un tableau de n nombres on crée un tas-max **t** ;
- Étape 1 : la racine **t[1]** est le maximum du tas, on l'échange avec **t[n]** (notation slicing `Python`).
- Le max du tableau se retrouve à la fin du tas en position **t[n]**. **t[n :]** est trié et contient le max.
- On met à jour la longueur du tas (qui représente le nombre d'éléments qu'il reste à trier) en la décrémentant (puisque l'ancienne racine a trouvé sa place).
- On percole bas de la nouvelle racine. Alors **t[0 :n]** est de nouveau un tas. Et on itère (swap puis percolation)...
- L'invariant qui assure la correction est ≪ *À la fin de l'étape* **k**, **t[0 :n-k+1]** *est un tas et* **t[n-k+1 :]** *est un tableau trié par ordre croissant dont les éléments sont plus grands que ceux de* **t[1 :n-k+1]** ≫. Il y a n étapes.

Tri par tas

```
1 let tri_tas l =
2   let t = create_down l  in (*O(n)*)
3   while t.(0)>1 do
4     (*mettre le max à la fin du tab :*)
5     swap 1 t.(0) t;
6     t.(0)<-(t.(0)-1);(*le tas à étudier a un elt de
        moins*)
7     percolate_down 1 t (*O(log(n))*)
8   done; Array.sub t 1 (Array.length l)  ;;
```

Tri par tas

Complexité

- $O(n)$ pour la création d'un tas (rappel : hauteur $O(\log_2(n))$)

Tri par tas

Complexité

- $O(n)$ pour la création d'un tas (rappel : hauteur $O(\log_2(n))$
- Chaque échange d'éléments et chaque décrémentation de **t.(0)** en $O(1)$

Tri par tas

Complexité

- $O(n)$ pour la création d'un tas (rappel : hauteur $O(\log_2(n))$)
- Chaque échange d'éléments et chaque décrémentation de **t.(0)** en $O(1)$
- Chaque appel à `percolate_down` en $O(\log_2 n)$ (majoration grossière car la longueur du tas décroît).

Tri par tas

Complexité

- $O(n)$ pour la création d'un tas (rappel : hauteur $O(\log_2(n))$
- Chaque échange d'éléments et chaque décrémentation de **t.(0)** en $O(1)$
- Chaque appel à `percolate_down` en $O(\log_2 n)$ (majoration grossière car la longueur du tas décroît).
- n passages dans la boucle.

Tri par tas

Complexité

- $O(n)$ pour la création d'un tas (rappel : hauteur $O(\log_2(n))$
- Chaque échange d'éléments et chaque décrémentation de **t.(0)** en $O(1)$
- Chaque appel à `percolate_down` en $O(\log_2 n)$ (majoration grossière car la longueur du tas décroît).
- n passages dans la boucle.
- Complexité au pire en $O(n \log_2(n))$ à la louche.

Tri par tas

Complexité

- $O(n)$ pour la création d'un tas (rappel : hauteur $O(\log_2(n))$)
- Chaque échange d'éléments et chaque décrémentation de **t.(0)** en $O(1)$
- Chaque appel à `percolate_down` en $O(\log_2 n)$ (majoration grossière car la longueur du tas décroît).
- n passages dans la boucle.
- Complexité au pire en $O(n \log_2(n))$ à la louche.
- On ne peut pas faire mieux pour un tri par comparaison.

1 Arbre binaire de recherche
- ABR
- Implémentation
- Rotations

2 Dictionnaires

3 Tas
- Généralités
- Représentation par tableaux
- Opérations

4 Files de priorités (une application des tas)

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :
  - insérer un élément avec sa priorité ;

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :
  - insérer un élément avec sa priorité ;
  - extraire l'élément ayant la plus grande clé ;

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :
  - insérer un élément avec sa priorité ;
  - extraire l'élément ayant la plus grande clé ;
  - tester si la file de priorité est vide ou pas.

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :
  - insérer un élément avec sa priorité ;
  - extraire l'élément ayant la plus grande clé ;
  - tester si la file de priorité est vide ou pas.
  - On ajoute parfois à cette liste l'opération « augmenter/diminuer » la clé d'un élément », utilisée par exemple dans l'algorithme de Dijkstra.

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :
  - insérer un élément avec sa priorité ;
  - extraire l'élément ayant la plus grande clé ;
  - tester si la file de priorité est vide ou pas.
  - On ajoute parfois à cette liste l'opération « augmenter/diminuer » la clé d'un élément », utilisée par exemple dans l'algorithme de Dijkstra.
- Les *priorités* sont d'un type totalement ordonné.

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :
  - insérer un élément avec sa priorité ;
  - extraire l'élément ayant la plus grande clé ;
  - tester si la file de priorité est vide ou pas.
  - On ajoute parfois à cette liste l'opération « augmenter/diminuer » la clé d'un élément », utilisée par exemple dans l'algorithme de Dijkstra.
- Les *priorités* sont d'un type totalement ordonné.
- Une file de priorité permet d'implémenter efficacement des planificateurs de tâches, où un accès rapide aux tâches d'importance maximale est souhaité.

Définition

- Une *file de priorité* est une structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.
- Il y a trois primitives :
  - insérer un élément avec sa priorité ;
  - extraire l'élément ayant la plus grande clé ;
  - tester si la file de priorité est vide ou pas.
  - On ajoute parfois à cette liste l'opération « augmenter/diminuer » la clé d'un élément », utilisée par exemple dans l'algorithme de Dijkstra.
- Les *priorités* sont d'un type totalement ordonné.
- On trouve une file de priorité, par exemple, dans les ordonnanceurs des systèmes d'exploitation, notamment le noyau LINUX.

Exemple

Les urgences d'un hôpital :

- chaque nouveau patient est ajouté à la file,

Exemple

Les urgences d'un hôpital :

- chaque nouveau patient est ajouté à la file,
- chaque fois qu'un médecin est libre, il s'occupe du patient avec l'état le plus critique.

Exemple

Les urgences d'un hôpital :

- chaque nouveau patient est ajouté à la file,
- chaque fois qu'un médecin est libre, il s'occupe du patient avec l'état le plus critique.
- Le tri des patients se fait sur des critères quantifiables et ordonnés comme

Exemple

Les urgences d'un hôpital :

- chaque nouveau patient est ajouté à la file,
- chaque fois qu'un médecin est libre, il s'occupe du patient avec l'état le plus critique.
- Le tri des patients se fait sur des critères quantifiables et ordonnés comme
  - l'état de conscience : Échelle de Glasgow $[\![1,4]\!] \times [\![1,5]\!] \times [\![1,6]\!]$ (priorité aux scores bas, ce qui entraîne d'utiliser un tas-min)

FIGURE – Score de Glasgow : somme des trois critères

Ouverture des yeux	Réponse verbale	Réponse motrice
1 - nulle	1 - nulle	1 - nulle
2 - à la douleur	2 - incompréhensible	2 - extension stéréotypée (rigidité décérébrée)
3 - à la demande	3 - inappropriée	3 - flexion stéréotypée (rigidité de décortication)
4 - spontanée[2]	4 - confuse	4 - évitement (retrait)
	5 - normale	5 - orientée (localisatrice)
		6 - aux ordres[3]

Exemple

Les urgences d'un hôpital :

- chaque nouveau patient est ajouté à la file,
- chaque fois qu'un médecin est libre, il s'occupe du patient avec l'état le plus critique.
- Le tri des patients se fait sur des critères quantifiables et ordonnés comme
  - l'état de conscience : Échelle de Glasgow $[\![1,4]\!] \times [\![1,5]\!] \times [\![1,6]\!]$ (priorité aux scores bas, ce qui entraîne d'utiliser un tas-min)
  - s'ils respirent (1 ou 0), ou s'ils saignent (volume de la perte de sang)...

Type de données

- Comme le tas est un tableau de données (type **data**), on ne peut plus réserver le premier élément à l'indication de sa longueur(type **int**).

```
1 type ('a, 'b) data = {priority: 'a; value: 'b} ;;
2 type ('a, 'b) priority_file = {mutable n: int; tbl: ('
      a, 'b) data array} ;;
```

Type de données

- Comme le tas est un tableau de données (type **data**), on ne peut plus réserver le premier élément à l'indication de sa longueur(type **int**).
- Toutefois, on a pris l'habitude de stocker les données à partir de l'élément 1 et non 0. Donc nos tableaux auront une première case qui ne servira à rien.

```
1 type ('a, 'b) data = {priority: 'a; value: 'b} ;;
2 type ('a, 'b) priority_file = {mutable n: int; tbl: ('
      a, 'b) data array} ;;
```

Type de données

- Comme le tas est un tableau de données (type **data**), on ne peut plus réserver le premier élément à l'indication de sa longueur(type **int**).
- Toutefois, on a pris l'habitude de stocker les données à partir de l'élément 1 et non 0. Donc nos tableaux auront une première case qui ne servira à rien.
- La longueur du tas est toujours susceptible d'évoluer. Il faut la définir comme *mutable*.

```
1 type ('a, 'b) data = {priority: 'a; value: 'b} ;;
2 type ('a, 'b) priority_file = {mutable n: int; tbl: ('
      a, 'b) data array} ;;
```

Type de données

- Comme le tas est un tableau de données (type **data**), on ne peut plus réserver le premier élément à l'indication de sa longueur(type **int**).
- Toutefois, on a pris l'habitude de stocker les données à partir de l'élément 1 et non 0. Donc nos tableaux auront une première case qui ne servira à rien.
- La longueur du tas est toujours susceptible d'évoluer. Il faut la définir comme *mutable*.
- **f.n+1** désigne la première case libre du tas **f.tbl**.

```
1 type ('a, 'b) data = {priority: 'a; value: 'b} ;;
2 type ('a, 'b) priority_file = {mutable n: int; tbl: ('
      a, 'b) data array} ;;
```

Création

```
let creer_file n (p,v) = {n = 0; tbl = Array.make (n
    +1) {priority = p; value = v}};;
let empty_queue = creer_file 5 (1, 2);;
```

Exceptions

- On peut ajouter des éléments dans la file tant qu'elle n'est pas pleine. Si elle est pleine, on soulève une exception **Full**.

Exceptions

- On peut ajouter des éléments dans la file tant qu'elle n'est pas pleine. Si elle est pleine, on soulève une exception **Full**.
- On peut retirer des éléments de la file tant qu'elle n'est pas vide. Si elle est vide, on soulève une exception **Empty**.

Exceptions

- On peut ajouter des éléments dans la file tant qu'elle n'est pas pleine. Si elle est pleine, on soulève une exception **Full**.
- On peut retirer des éléments de la file tant qu'elle n'est pas vide. Si elle est vide, on soulève une exception **Empty**.
- Création des exceptions :

```
exception Empty ;;
exception Full ;;
```

Ajouter

Il faut adapter la fonction de percolation haute à nos priorités. En exo.

```
let ajouter d f =
  if f.n+1 = Array.length f.tbl then raise Full;
  f.n<-f.n+1;
  f.tbl.(f.n)<-d;
  percolate_up (f.n) f;;
```

Retirer

Il faut adapter la fonction de percolation basse à nos priorités. En exo.

```
let retirer f =
  if f.n = 0 then raise Empty;
  f.tbl.(1)<-f.tbl.(f.n);
  f.n<-(f.n)-1;
  percolate_down 1 f;;
```

Augmenter la priorité

Quand on augmente l'urgence, donc qu'on diminue la valeur de priorité d'un élément, on le fait remonter par percolation haute.

```
let plus_prioritaire k p f=
  (*augmente la priorité de l'élément k*)
  if f.tbl.(k).priority <= p
  then failwith "la nouvelle priorité doit être plus
     petite";
  f.tbl.(k)<-{priority=p;value=f.tbl.(k).value};
  percolate_up k f;;
```

Augmenter la priorité

Quand on augmente l'urgence, donc qu'on diminue la valeur de priorité d'un élément, on le fait remonter par percolation haute.

```
1 let plus_prioritaire k p f=
2   (*augmente la priorité de l'élément k*)
3   if f.tbl.(k).priority <= p
4   then failwith "la nouvelle priorité doit être plus
       petite";
5   f.tbl.(k)<-{priority=p;value=f.tbl.(k).value};
6   percolate_up k f;;
```

Et quand on diminue l'urgence, on fait descendre l'élément par percolation basse.

Augmenter la priorité

Quand on augmente l'urgence, donc qu'on diminue la valeur de priorité d'un élément, on le fait remonter par percolation haute.

```
1  let plus_prioritaire k p f=
2    (*augmente la priorité de l'élément k*)
3    if f.tbl.(k).priority <= p
4    then failwith "la nouvelle priorité doit être plus
         petite";
5    f.tbl.(k)<-{priority=p;value=f.tbl.(k).value};
6    percolate_up k f;;
```

Et quand on diminue l'urgence, on fait descendre l'élément par percolation basse.
Il peut être utile de maintenir dans la structure un dictionnaire qui indique, pour chaque valeur, à quelle position on peut la trouver.