Chapitre 1 : Les arbres (saison 3)

Option Informatique – MP

Lycée Chateaubriand

Table des matières

Plan

Implémentation

ABR

Tas et liste de priorité

Liste de priorité

Liste de priorité

Nous voulons étudier la structure de liste de priorité. Le principe est le suivant, on imagine des personnes qui arrivent à un guichet mais, contrairement à la file classique, les personnes ont des importances différentes (on peut par exemple penser à des processus qui arrivent à un processeur). L'opérateur du guichet doit donc pouvoir traiter la personne qui a la plus grande importance. De ce fait, on veut faire trois opérations :

- Trouver le maximum
- Insérer un élément
- Retirer le maximum.

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum :

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum : linéaire
  - Insérer un élément :

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum : linéaire
  - Insérer un élément : constant
  - Suppression du maximum :

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum : linéaire
  - Insérer un élément : constant
  - Suppression du maximum : linéaire il faut trouver le nouveau maximum.

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum : linéaire
  - Insérer un élément : constant
  - Suppression du maximum : linéaire il faut trouver le nouveau maximum.
- Un tableau ordonné. Là encore, on peut regarder les complexité :
  - Trouver le maximum :

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum : linéaire
  - Insérer un élément : constant
  - Suppression du maximum : linéaire il faut trouver le nouveau maximum.
- Un tableau ordonné. Là encore, on peut regarder les complexité :
  - Trouver le maximum : constant
  - Insérer un élément :

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum : linéaire
  - Insérer un élément : constant
  - Suppression du maximum : linéaire il faut trouver le nouveau maximum.
- Un tableau ordonné. Là encore, on peut regarder les complexité :
  - Trouver le maximum : constant
  - Insérer un élément : linéaire
  - Suppression du maximum :

Implémentations naïves

Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum : linéaire
  - Insérer un élément : constant
  - Suppression du maximum : linéaire il faut trouver le nouveau maximum.
- Un tableau ordonné. Là encore, on peut regarder les complexité :
  - Trouver le maximum : constant
  - Insérer un élément : linéaire
  - Suppression du maximum : linéaire car les tableaux ne sont pas mutables

Nous allons voir une implémentation qui permet d'insérer et de retirer le maximum en $O(\log n)$ et de trouver le maximum en temps constant où n désigne la taille de la file.

Tas

Tas

Définition 1

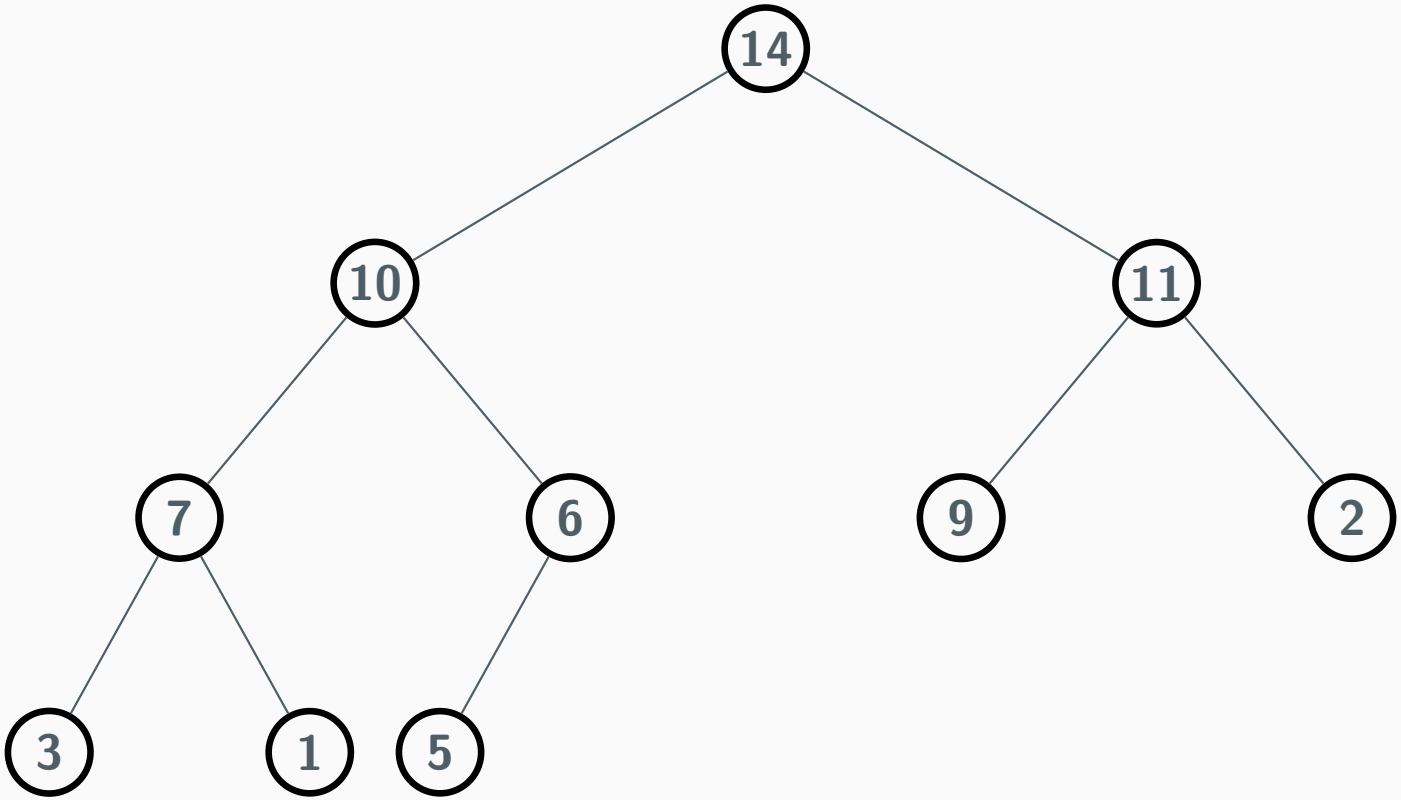
Un tas (maximal) de hauteur h est un arbre binaire tel que :

- *Toutes les feuilles sont de profondeur h ou $h-1$.*
- *Pour tout $p < h$, il y a exactement 2^p nœuds de profondeur p.*
- *Tous les nœuds de profondeur h sont « à gauche ».*
- *Chaque nœud a une étiquette supérieure ou égale à celle de ses fils. En particulier, la racine est le maximum de toutes les étiquettes.*

Remarques

- Les trois premières conditions concernent le squelette, la dernière concerne les étiquettes.
- On peut aussi considérer un tas minimal où, au contraire, la racine est le plus petit élément et ainsi de suite.

Exemple



Remarques

- On peut encadrer la taille n de l'arbre en fonction de sa hauteur h (et réciproquement) en décomposant les cas selon le nombre de nœuds du dernier niveau. On a :

$$2^h = 1 + 2 + 2^2 + \cdots + 2^{h-1} + 1 \leq n \leq 1 + 2 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$$

  Et donc

$$\log_2(n+1) - 1 \leq h \leq \log_2(n).$$

- L'intérêt est que l'on accède en temps constant au plus grand élément du tas (c'est la racine). Cela permet donc d'utiliser cela pour implémenter une file de priorité.

Insertion

On veut pouvoir insérer un élément dans un tas (de manière à garder un tas) de plus, cette méthode doit être en temps logarithmique par rapport à la taille du tas (ce qui revient à linéaire par rapport à la hauteur[1]).

[1] on rappelle que, par définition, un tas est presque complet.

Insertion

On veut pouvoir insérer un élément dans un tas (de manière à garder un tas) de plus, cette méthode doit être en temps logarithmique par rapport à la taille du tas (ce qui revient à linéaire par rapport à la hauteur[1]). La méthode est la suivante :

- On insére l'élément « en bas » du tas, c'est-à-dire que si le tas est un arbre binaire complet on augmente la profondeur de 1 et, dans le cas contraire, on rajoute l'élément « dans la dernière couche », juste à droite du dernier élément. De fait, c'est la seule place disponible au niveau du squelette du tas.

[1] on rappelle que, par définition, un tas est presque complet.

Insertion

On veut pouvoir insérer un élément dans un tas (de manière à garder un tas) de plus, cette méthode doit être en temps logarithmique par rapport à la taille du tas (ce qui revient à linéaire par rapport à la hauteur[1]). La méthode est la suivante :

- On insére l'élément « en bas » du tas, c'est-à-dire que si le tas est un arbre binaire complet on augmente la profondeur de 1 et, dans le cas contraire, on rajoute l'élément « dans la dernière couche », juste à droite du dernier élément. De fait, c'est la seule place disponible au niveau du squelette du tas.
- On remonte l'élément ajouté en l'échangeant avec son père tant que ce dernier est inférieur à l'élément qui vient d'être ajouté.

[1] on rappelle que, par définition, un tas est presque complet.

Complexité

Théorème 1 (Complexité de l'insertion d'un élément dans un tas)

L'algorithme d'insertion ci-dessus a une complexité linéaire en la profondeur du tas c'est-à-dire une complexité logarithmique en la taille du tas.

Suppression

On veut aussi pouvoir supprimer la racine. La méthode est la suivante

- On échange la racine avec « le dernier élément du tas »,
- On supprime le dernier élément du tas l'élément. C'est la seule possibilité au niveau du squelette du tas.
- On « descend » l'élément qui a été échangé :

Suppression

On veut aussi pouvoir supprimer la racine. La méthode est la suivante

- On échange la racine avec « le dernier élément du tas »,
- On supprime le dernier élément du tas l'élément. C'est la seule possibilité au niveau du squelette du tas.
- On « descend » l'élément qui a été échangé :
  tant que cet élément est inférieur au maximum de ses deux fils, on l'échange avec ce maximum

Complexité

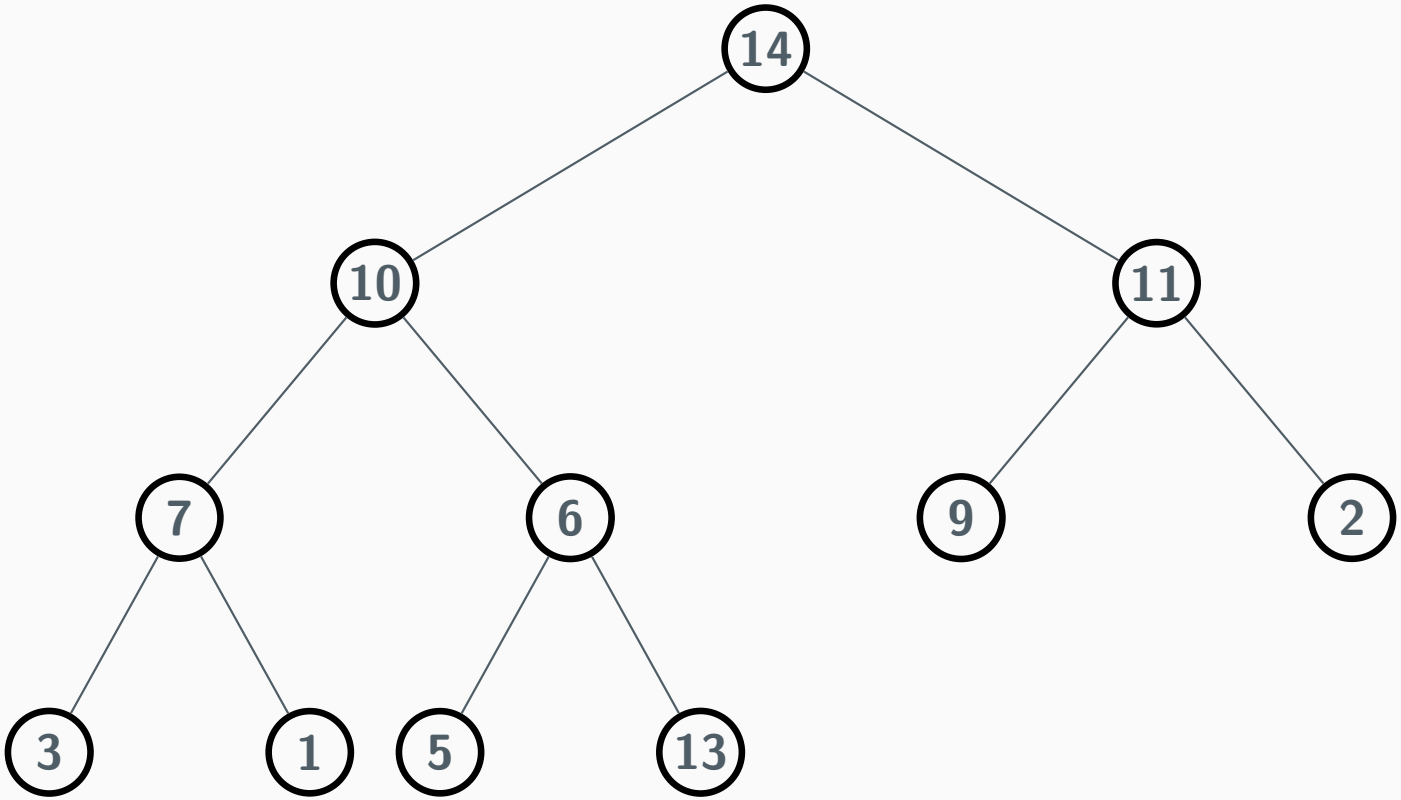
Théorème 2 (Complexité de la suppression de la racine dans un tas)

L'algorithme de suppression ci-dessus a une complexité linéaire en la profondeur du tas c'est-à-dire une complexité logarithmique en la taille du tas.

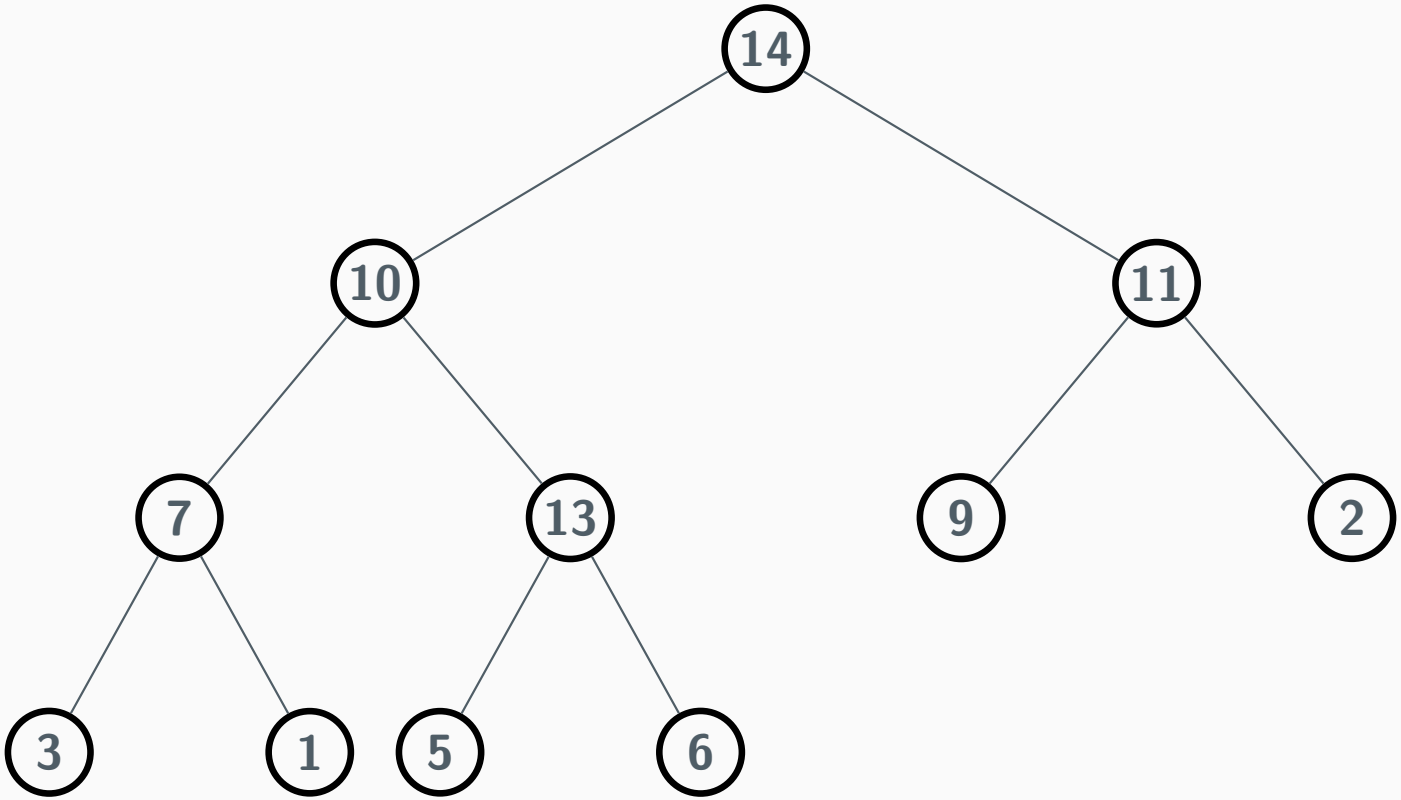
Exemple

Si on reprend le tas ci dessus et que l'on insère 13, on obtient successivement

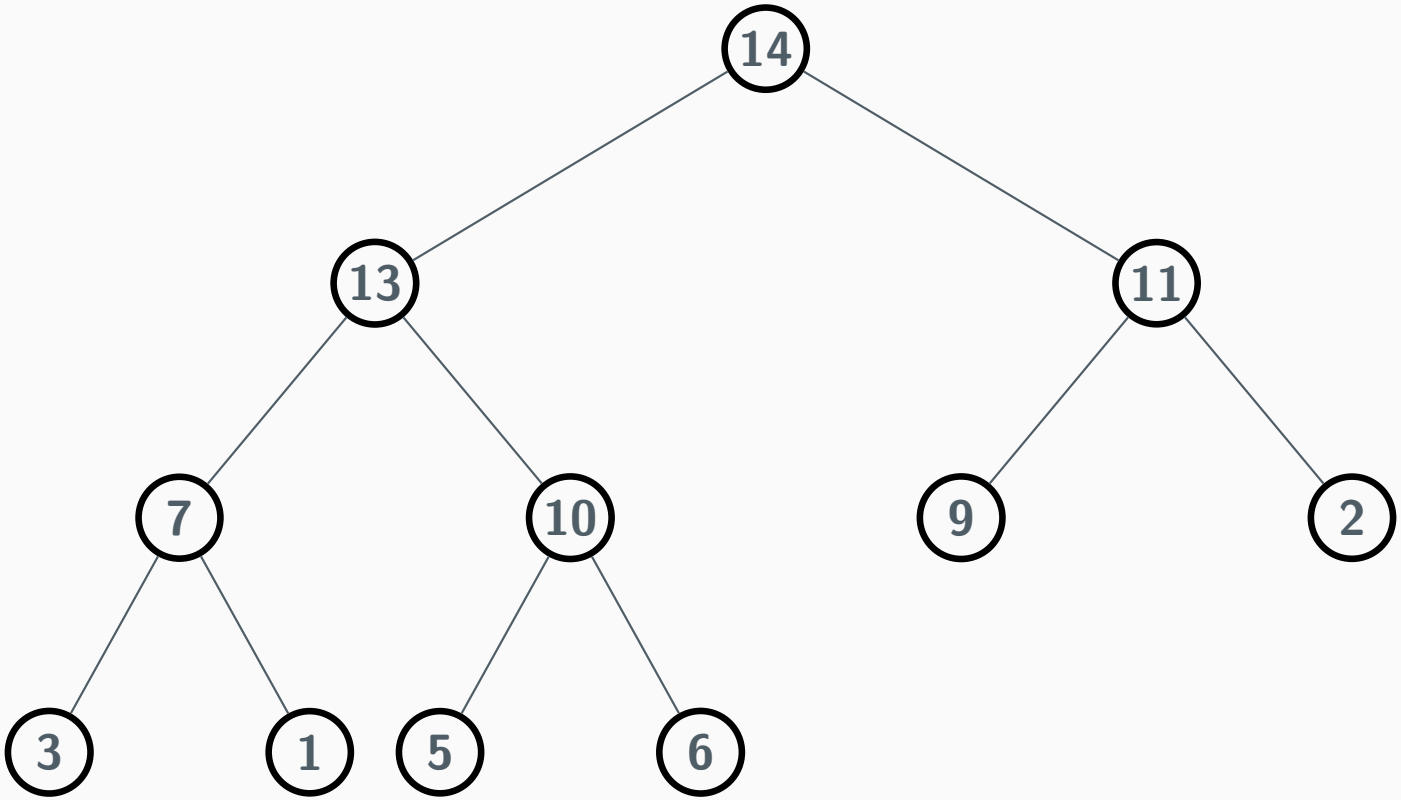
Exemple



Exemple



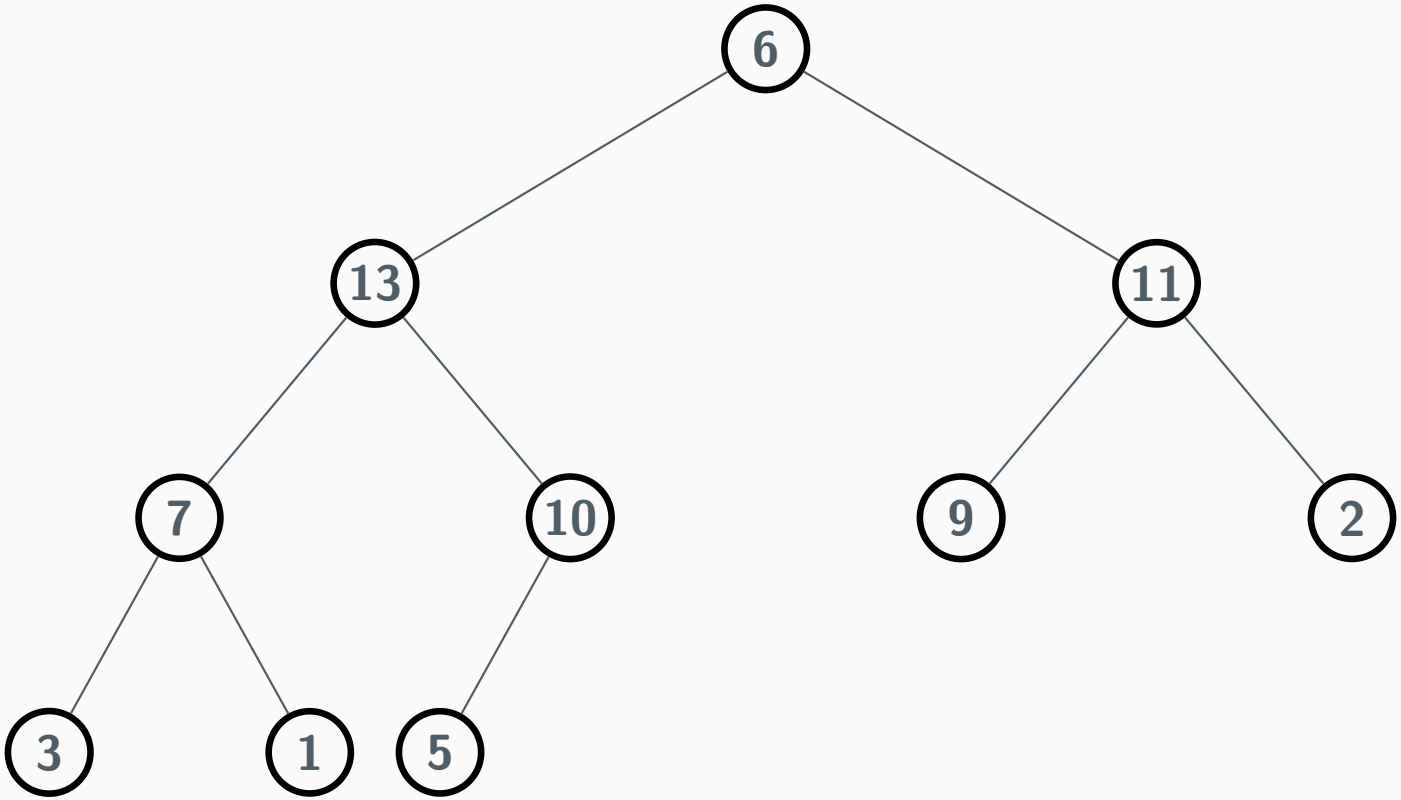
Exemple



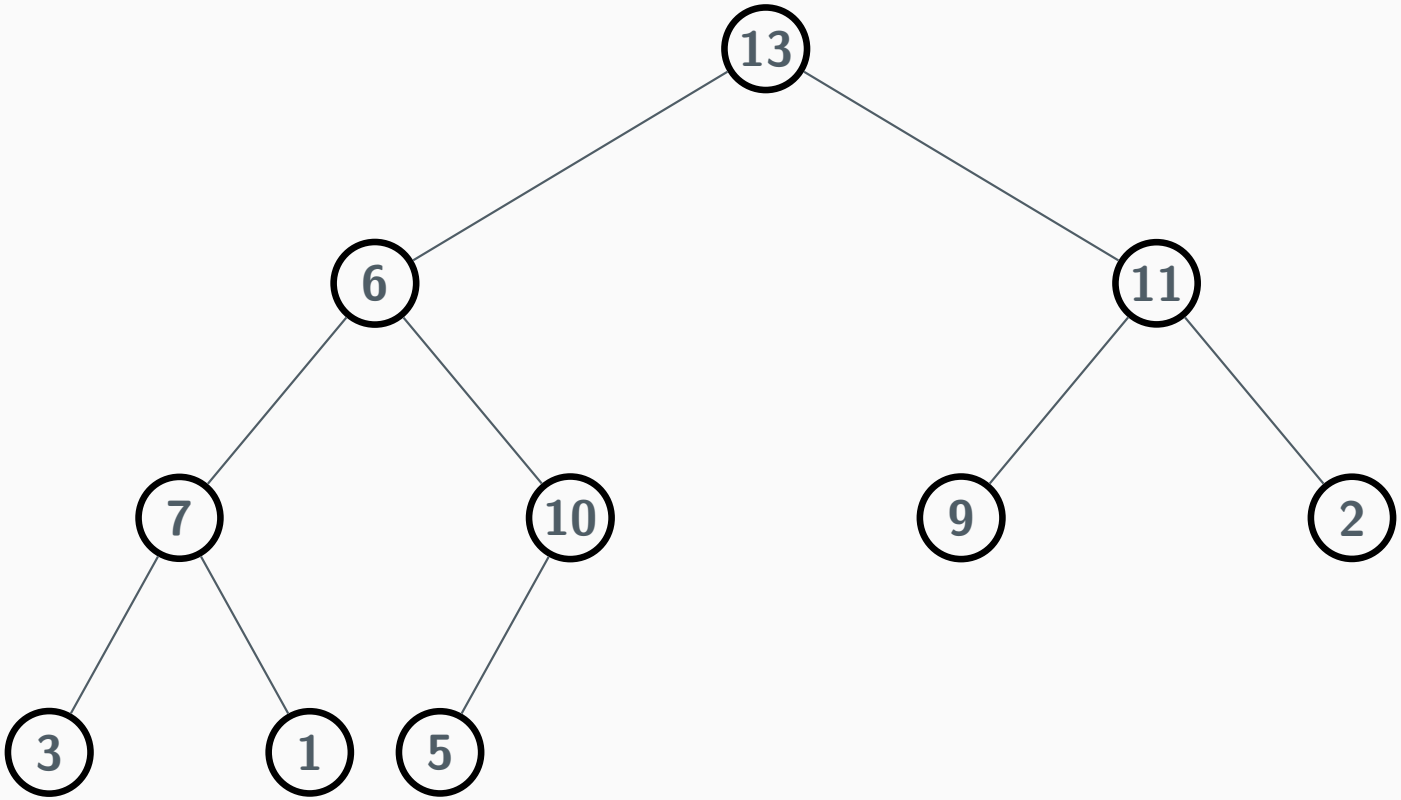
Exemple

Maintenant si on veut supprimer la racine, on obtient

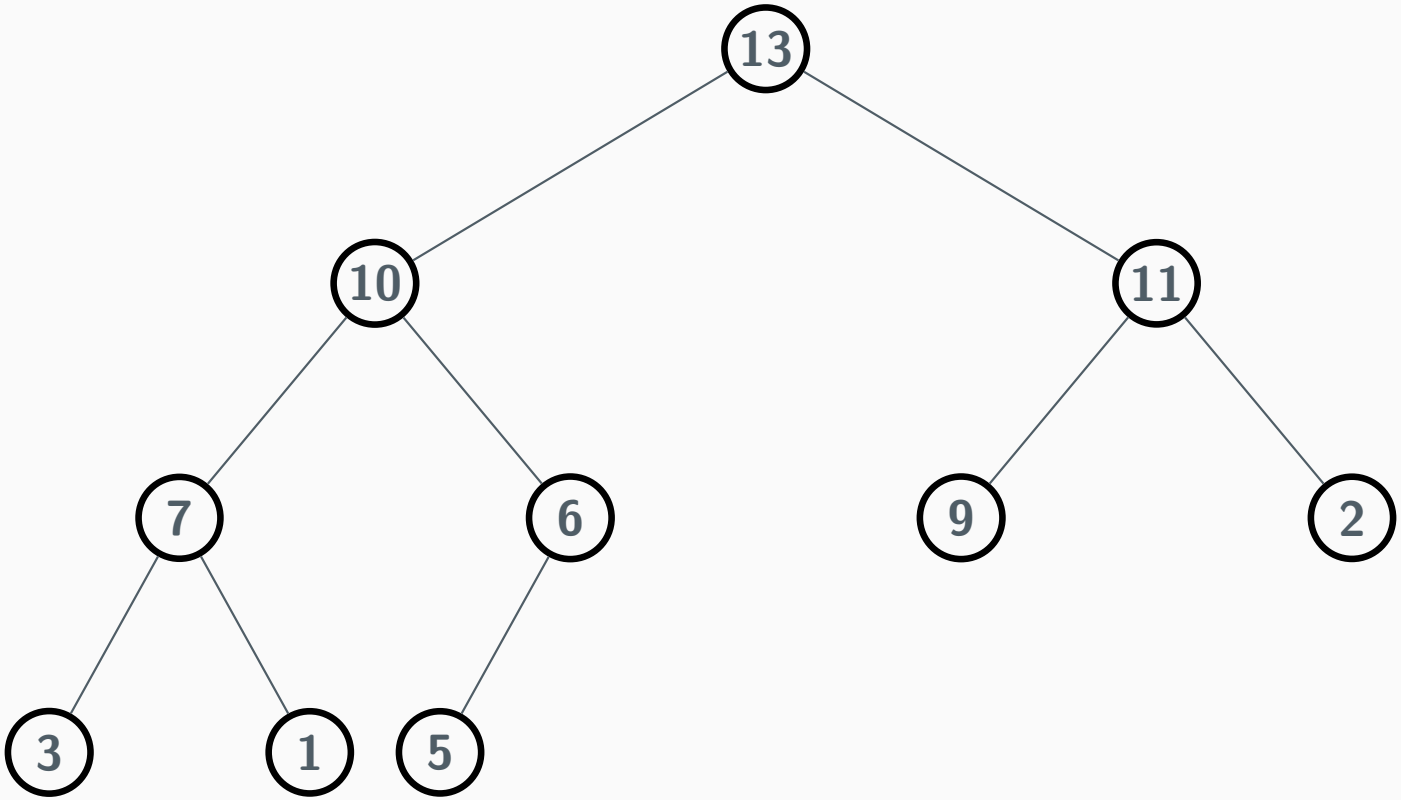
Exemple



Exemple



Exemple



Implémentation à l'aide d'un tableau

On peut implémenter une file de priorité à l'aide d'un tableau.

On se donne un tableau `t` de type `int array` de longueur N (supposée suffisamment grande). La case `t.(0)` sert à repérer la taille du tas. En pratique on ne considère dans le tas que les éléments `t.(i)` pour i compris en 1 et `t.(0)`. Ensuite on remplit le tableau de la manière suivante :

- La racine du tas est `t.(1)`.
- Ses deux fils sont `t.(2)` et `t.(3)`.
- De manière plus générale, les éventuels fils de `t.(k)` sont `t.(2k)` et `t.(2k+1)`. De ce fait, le père de `t.(k)` est `t.(k/2)`[2].

[2] ici k/2 désigne la partie entière de $\frac{k}{2}$

Remarque

Si on veut utiliser une file de priorité dont les éléments ne sont pas des entiers, on peut utiliser un type enregistrement :

```
type filePriorite = {mutable taille : int ; donnee : 'a array}
```

On peut alors garder les mêmes conventions que ci-dessus (on suppose alors que `t.(0)`ne sert pas) ou commencer à `t.(0)`et il y a lieu de décaler les formules ci-dessus.

Exercice

Si on stocke les éléments à partir de `t.(0)` quel sont les indices des fils éventuels de `t.(k)` ?

Exercice

Si on stocke les éléments à partir de `t.(0)` quel sont les indices des fils éventuels de `t.(k)` ?

Les fils de `t.(k)` ont pour indices $2k+1$ et $2k+2$.

Exercice

Si on stocke les éléments à partir de `t.(0)` quel sont les indices des fils éventuels de `t.(k)` ?

Les fils de `t.(k)` ont pour indices $2k+1$ et $2k+2$.

Quel est l'indice du père de `t.(k)` ?

Exercice

Si on stocke les éléments à partir de `t.(0)` quel sont les indices des fils éventuels de `t.(k)` ?

Les fils de `t.(k)` ont pour indices $2k+1$ et $2k+2$.

Quel est l'indice du père de `t.(k)` ?

Le père de `t.(k)` a pour indice $\lfloor \frac{k-1}{2} \rfloor$

Fonctions

On peut alors écrire nos fonctions

- Création d'un tas vide

```
let creTasVide taille_max = Array.make taille_max 0;;
```

- Recherche du maximum :

```
let maxiFile t = if t.(0) = 0 then failwith "le tas est vide"
    else t.(1);;
```

Remarque

On peut bien évidemment tester si la file est vide `t.(0) = 0`et éventuellement lever une exception.

Fonction d'insertion

On commence par écrire une fonction `echange : 'a array -> int -> int -> unit` telle que, par effet de bords, `t i j` échange les éléments d'indice `i` et `j` dans `t`.

Fonction d'insertion

On commence par écrire une fonction `echange : 'a array -> int -> int -> unit` telle que, par effet de bords, `t i j` échange les éléments d'indice `i` et `j` dans `t`.

```
let echange t i j =
  let temp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- temp;;
```

Fonction d'insertion

On peut alors écrire la fonction `insert : 'a array -> 'a -> unit` telle que `insert t x` insert l'élément `x` dans le tas donné par le tableau `t`.

Fonction d'insertion

On peut alors écrire la fonction `insert : 'a array -> 'a -> unit` telle que `insert t x` insert l'élément `x` dans le tas donné par le tableau `t`.

```
let insert t x =
  t.(0) <- t.(0) +1;
  t.(t.(0)) <- x;
  let k = ref t.(0) in
  let p = ref !k/2 in
  while !k <> 1 && t.(!k) > t.(!p) do
     echange t !k !p;
     k := !p;
     p := !k /2
     done;;
```

Fonction d'insertion

On peut aussi écrire une fonction qui utilise une fonction auxiliaire récursive

Fonction d'insertion

On peut aussi écrire une fonction qui utilise une fonction auxiliaire récursive

```
let insert2 x t =
  t.(0) <- t.(0) +1;
  t.(t.(0)) <- x;
  let rec remonte i =
    let p = i/2 in
    if i <> 1 && t.(i) > t.(p) then
    begin
     echange t i p;
     remonte p
    end
  in remonte t.(0);;
```

Remarque

La fonction procède par effets de bords (on a une implémentation impérative de la structure de file de priorité).

Fonction de suppression

On veut écrire une fonction

`supprime : 'a array -> unit`

telle que `supprime t` supprime la racine d'une file de priorité donnée par le tableau `t`.

On commence par une fonction récursive

`descente : 'a array -> int -> unit`

telle que `descente t k` va « descendre » l'élément d'indice `k` afin d'obtenir un tas.

Fonction de suppression

```
let rec descente t k =
   if (2*k <= t.(0)) &&  t.(k) < t.(2*k)) ||   (2*k+1 <= t.(0)) &&
         t.(k) < t.(2*k+1)) then
     if (2*k+1 <= t.(0)) && t.(2*k+1) > t.(2*k) then
    (echange t k 2*k+1;
    descente t (2*k+1))
      else
      (echange t k (2*k);
      descente t (2*k));;
```

Fonction de suppression

On peut alors écrire la fonction `supprime`

Fonction de suppression

On peut alors écrire la fonction `supprime`

```
let supprime t =
  echange t t.(0) 1;
  t.(0) <- t.(0)-1;
  descente t 1;;
```

Tri par tas

Tri pas tas

On peut utiliser la structure de tas pour construire un tri. En effet en partant d'une liste on peut insérer successivement ses éléments dans un tas. Par la suite, en retirant tous les éléments les uns après les autres du tas (en les prenant au sommet) on récupère une liste triée.

Les ajouts et suppression dans un tas étant de complexité logarithmique (par rapport à la taille du tas) on en déduit un algorithme de tri quasi-linéaire. Notons que cela peut se faire en place.

Exemple

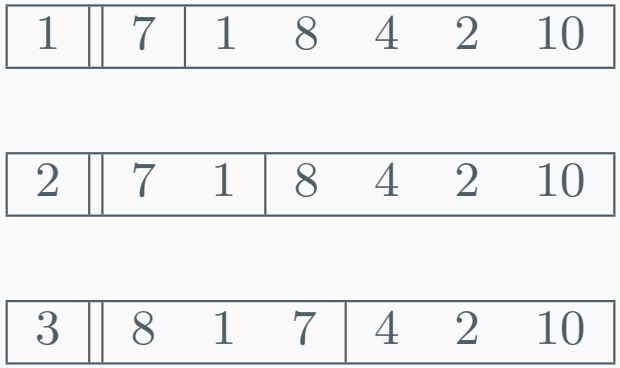
Trions le tableau $\boxed{7 \quad 1 \quad 8 \quad 4 \quad 2 \quad 10}$.

On considère que le début du tableau est un tas. Il a donc un élément qui est l'élément 7.

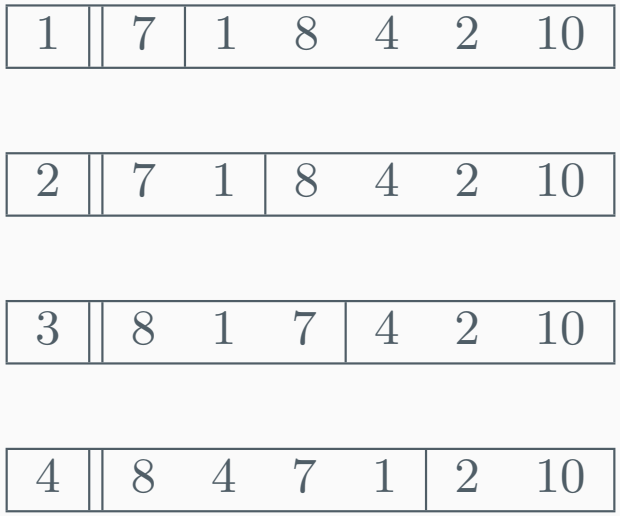
Exemple

Exemple

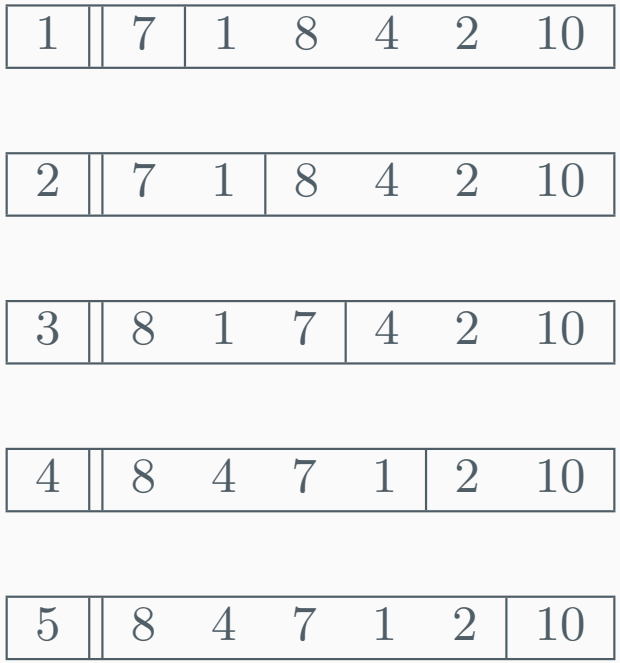
Exemple



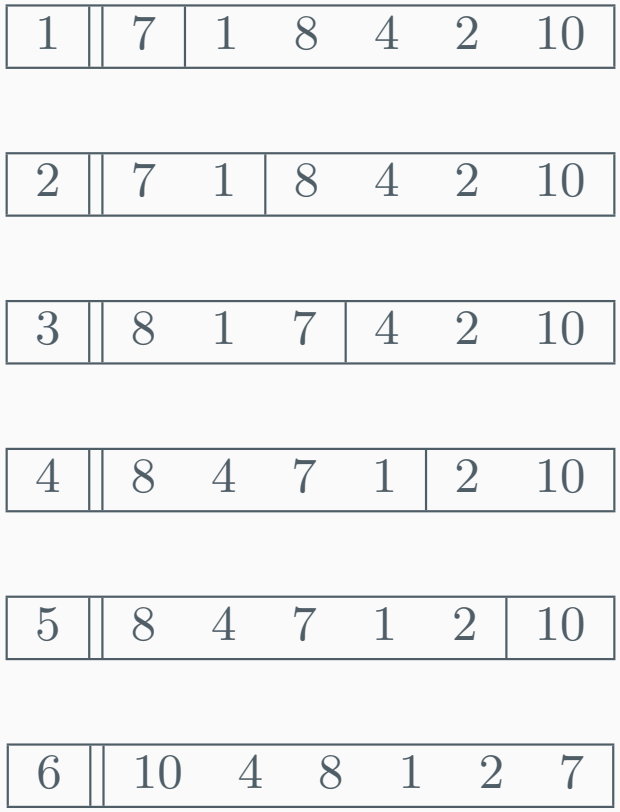
Exemple



Exemple



Exemple



Exemple

Il reste à supprimer les éléments du tas

6	10	4	8	1	2	7

Exemple

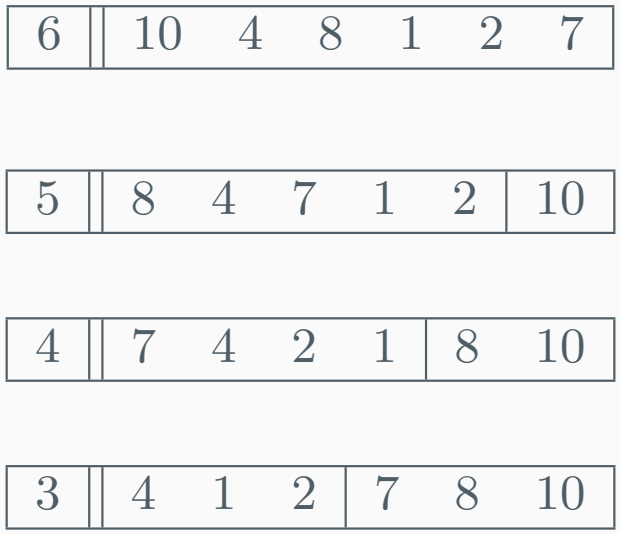
Il reste à supprimer les éléments du tas

Exemple

Il reste à supprimer les éléments du tas

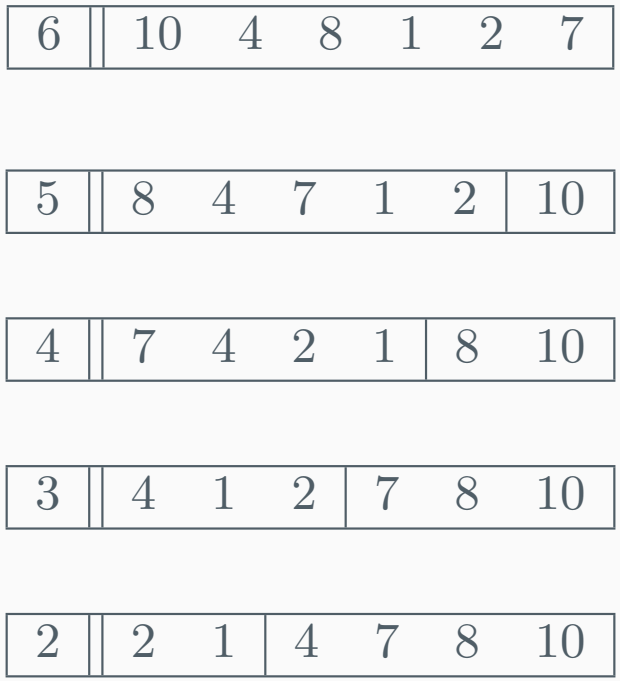
Exemple

Il reste à supprimer les éléments du tas



Exemple

Il reste à supprimer les éléments du tas



Exemple

Il reste à supprimer les éléments du tas