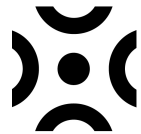
Algorithmique des arbres

Files de priorités : implémentation par un tas binaire

L2 Mathématique et Informatique



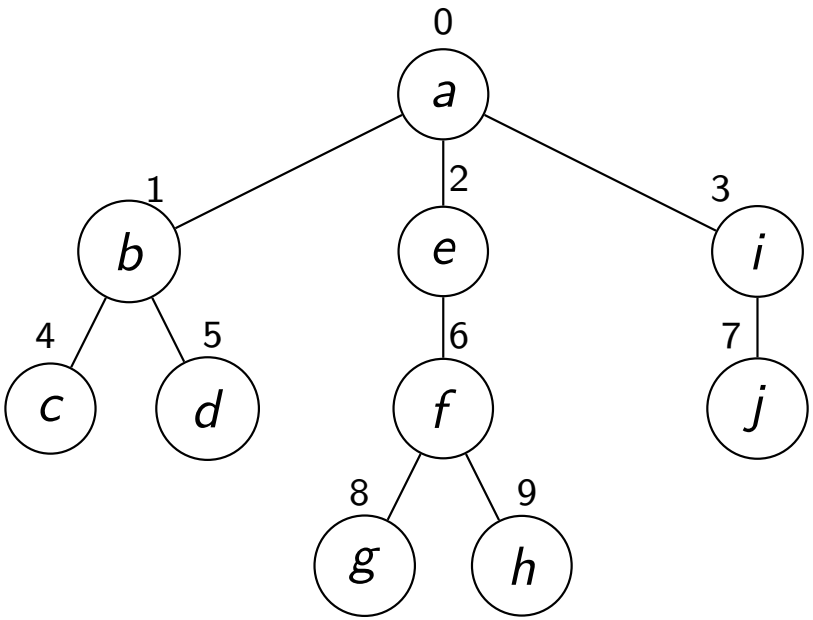
Outline

Rappel sur la définition d'arbres

Définition : **Arbres** *(Version itérative)*

Un *arbre* est défini par :

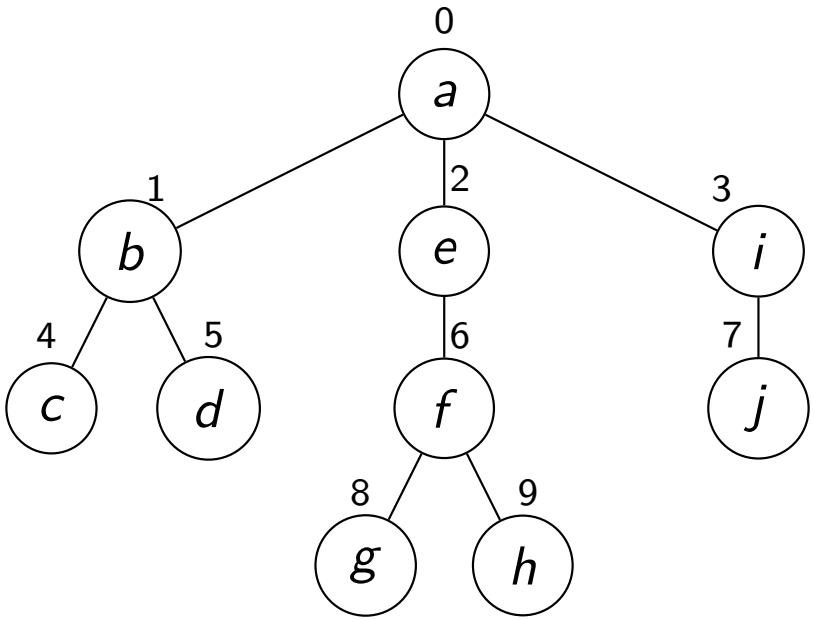
- un ensemble N de *nœuds* ;
- un nœud particulier r : la *racine* ;
- P une relation binaire "est parent de".



aPb aPe aPi
bPc bPd
ePf
iPj
fPg fPh

Implantation par table des pères

Nécessite un indiçage des nœuds de 0 à $nbnoeud - 1$. On utilise un tableau dans lequel on trouve à l'indice i le numéro du père de i



Avantages + Inconvénients :

$+++$ Économique en mémoire ;

$+++$ Accès facile des feuilles vers la racine ;

$---$ Accès difficile de la racine vers les feuilles.

Table des pères :	-1(ou 0)	0	0	0	1	1	2	3	6	6
Valeurs :	a	b	e	i	c	d	f	j	g	h

Outline

Type abstrait de données `File de priorité` 1 / 2

Définition : *(Version 1)*

Soit C un ensemble quelconque et E un ensemble ordonné.
Une *file de priorité* est un ensemble de couples $(\texttt{cle},\texttt{priorite}) \in C \times E$, sur lequel on souhaite pouvoir réaliser les opérations suivantes :

- $\texttt{maximum} : \texttt{File} \longrightarrow C \times E$;
- $\texttt{inserer} : \texttt{File} \times C \times E \longrightarrow \texttt{File}$;
- $\texttt{supprimer} : \texttt{File} \longrightarrow \texttt{File}$.

Remarque : On peut inverser l'ordre pour renvoyer les éléments avec la priorité minimale.

Exemple : gestion des urgences dans un hôpital

- $\begin{cases} \text{Perte de conscience, ne peut plus communiquer} \rightsquigarrow 100 \\ \text{Polytraumatisé (= patient tout cassé de partout)} \rightsquigarrow 100 \\ \text{Arrêt cardiaque} \rightsquigarrow 100 \end{cases}$
- Bras cassé $\rightsquigarrow 20$
- Coupure légère $\rightsquigarrow 10$
- Confusion entre Urgences et un médecin généraliste $\rightsquigarrow 0$

Type abstrait de données `File de priorité` 2 / 2

En général, les éléments que l'on insère dans une file de priorités ont une clé et une priorité.

D'un point de vue algorithmique la clé de l'élément n'a aucune importance : seule la priorité compte.

$\Longrightarrow$ On assimilera donc les éléments à leur priorité.

Définition : *(Version 2)*

Soit E un ensemble ordonné.
Une *file de priorité* est un ensemble d'éléments de E sur lequel on souhaite pouvoir réaliser les opérations suivantes :

- `maximum` : `File` $\longrightarrow E$;
- `inserer` : `File` $\times E \longrightarrow$ `File` ;
- `supprimer` : `File` $\longrightarrow$ `File`.

Différence entre piles, files et files de priorité

Les files de priorité sont donc très proches des piles et des files :

- Toutes les trois implémentent une opération d'insertion et de suppression sur un ensemble de valeurs.

Différence fondamentale :

⇝ Les piles et les files ressortent leurs éléments dans un ordre dépendant de l'ordre d'insertion.

⇝ Les files de priorité ressortent leurs éléments dans un ordre qui dépend de leur priorité.

Applications des files de priorités

- Planificateurs de tâches où un accès rapide aux tâches d'importance maximale est souhaité.
  - ⇝ ordonnanceurs des systèmes d'exploitation, notamment le noyau Linux ;
  - ⇝ contrôle aérien.

- Recherche du plus court chemin dans un graphe valué par l'algorithme de Dijkstra (Voir cours d'algo de L3).

- Recherche d'un arbre couvrant de poids minimal par l'algorithme de Prim (Voir cours d'algo de L3).

1ière implémentation : par un tableau non ordonnée 1/2

```
typedef struct {
    Elt * tab;  // Tableau d'Elt non triés de taille `nb_elts`
    int nb_elts;
    int taille_max;
} FilePriorite;

int inserer(FilePriorite * f, Elt elt){
    if (f->nb_elts == f->taille_max)
        return 0;
    f->tab[f->nb_elts] = elt;
    (f->nb_elts)++;
    return 1;
}
```

1ière implémentation : par un tableau non ordonnée 2/2

```
typedef struct {
    Elt * tab;  // Tableau d'Elt non triés de taille `nb_elts`
    int nb_elts;
    int taille_max;
} FilePriorite;

int supprimer(FilePriorite * f, Elt * res){
    int pos_max, pos;
    if (f->nb_elts == 0)
        return 0;
    pos_max = position_max(f->tab, f->nb_elts);
    *res = f->tab[pos_max];
    for(pos = pos_max; pos < f->nb_elts - 1; pos++)
        f->tab[pos] = f->tab[pos + 1];
    (f->nb_elts)--;
    return 1;
}
```

2nd implémentation : par un tableau ordonnée 1/2

```
typedef struct {
    Elt * tab;  // Tableau d'Elt non triés de taille `nb_elts`
    int nb_elts;
    int taille_max;
} FilePriorite;

int inserer(FilePriorite * f, Elt elt){
    int pos_ins = 0, pos;
    if (f->nb_elts == f->taille_max)
        return 0;
    while (pos_ins < f->nb_elts && f->tab[pos_ins] < elt){
        pos_ins++;
    }
    for (pos = f->nb_elts; pos > pos_ins; pos--)
        f->tab[pos] = f->tab[pos - 1];
    f->tab[pos_ins] = elt;
    (f->nb_elts)++;
    return 1;
}
```

2nd implémentation : par un tableau ordonnée 2/2

```
typedef struct {
    Elt * tab;  // Tableau d'Elt non triés de taille `nb_elts`
    int nb_elts;
    int taille_max;
} FilePriorite;

int supprimer(FilePriorite * f, Elt * res){
    if (f->nb_elts == 0)
        return 0;
    *res = f->tab[f->nb_elts - 1];
    (f->nb_elts)--;
    return 1;
}
```

Compléxité pour différentes implémentations

Implémentation	`inserer`	`supprimer`
Par un tableau non ordonné Par une liste chainée	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Par un tableau ordonné Par une liste chainée triée	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Compléxité pour différentes implémentations

Implémentation	`inserer`	`supprimer`
Par un tableau non ordonné Par une liste chainée	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Par un tableau ordonné Par une liste chainée triée	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Par un tas max	$\mathcal{O}\big(\log_2(n)\big)$	$\mathcal{O}\big(\log_2(n)\big)$

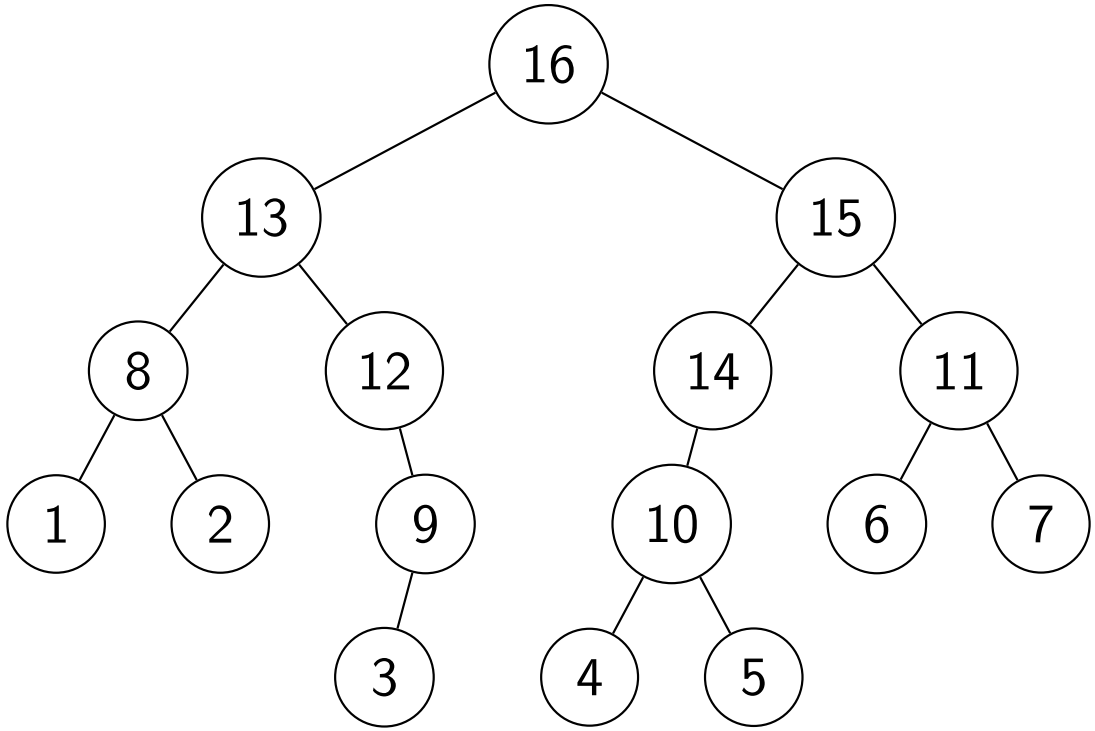
Outline

Outline

Arbre tournoi

Définition

Soit E un ensemble ordonné.
Un arbre *tournoi* sur E est un arbre étiqueté par des éléments de E dans lequel l'étiquette de chaque nœud est supérieure ou égale à celles de ses enfants (s'ils existent).



Un arbre tournoi sur les entiers positifs

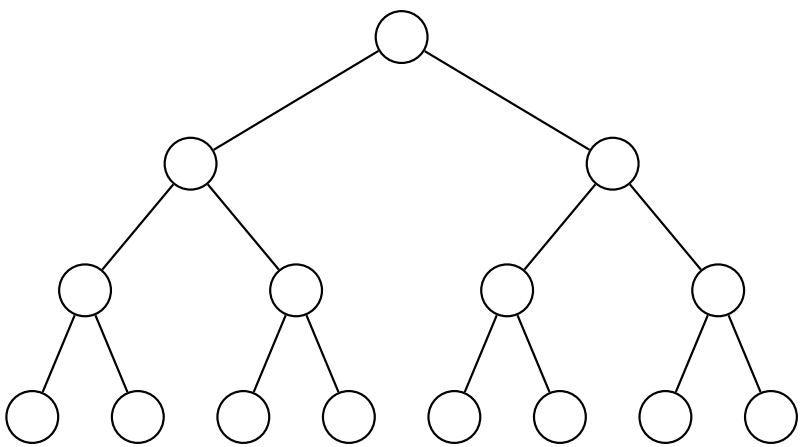
Arbre binaire presque complet

Définition

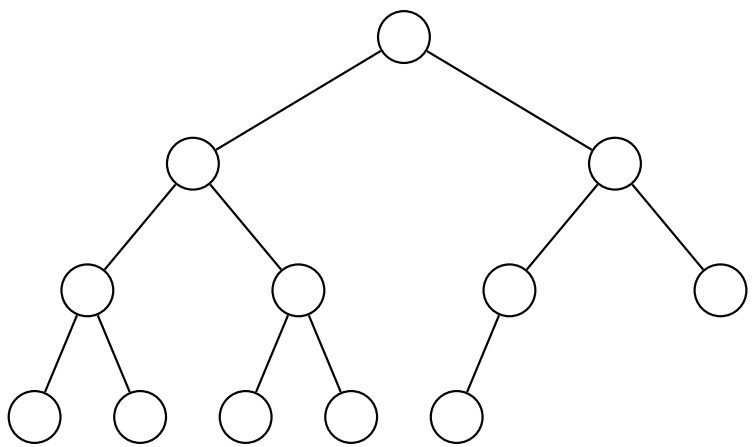
Un *arbre binaire complet* est un arbre strictement binaire dans lequel toutes les feuilles sont à même hauteur : tous les niveaux sont pleins.

Définition

Un *arbre binaire presque complet* est un arbre strictement binaire dans lequel tous les niveaux sont pleins, sauf peut être le dernier dans lequel les feuilles sont le plus à gauche possible.



Un arbre binaire complet



Un arbre binaire presque complet

Arbre binaire presque complet

Définition

Un *arbre binaire complet* est un arbre strictement binaire dans lequel toutes les feuilles sont à même hauteur : tous les niveaux sont pleins.

Définition

Un *arbre binaire presque complet* est un arbre strictement binaire dans lequel tous les niveaux sont pleins, sauf peut être le dernier dans lequel les feuilles sont le plus à gauche possible.

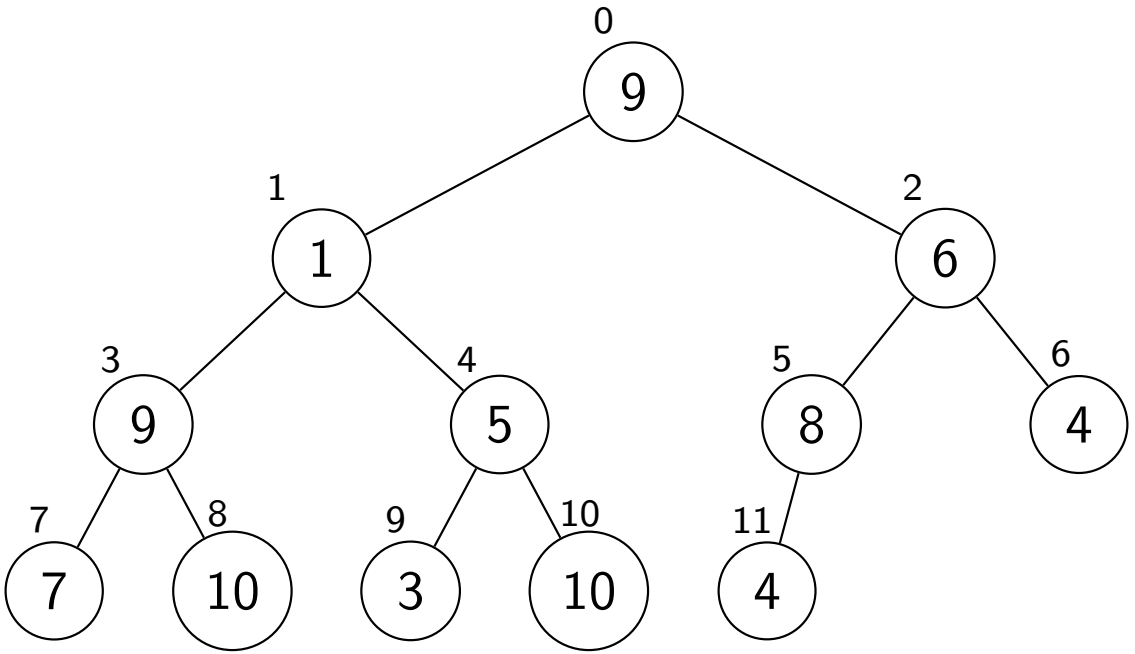
Propriété :

La hauteur d'un arbre binaire presque complet à n nœuds est $\lfloor \log_2(n) \rfloor$.

Rappel : Un arbre complet de hauteur h a $2^{h+1} - 1$ nœuds.

La relation Parent dans un arbre complet ou presque complet

Une numérotation par niveaux d'un arbre presque parfait permet un calcul direct de la relation Parent.



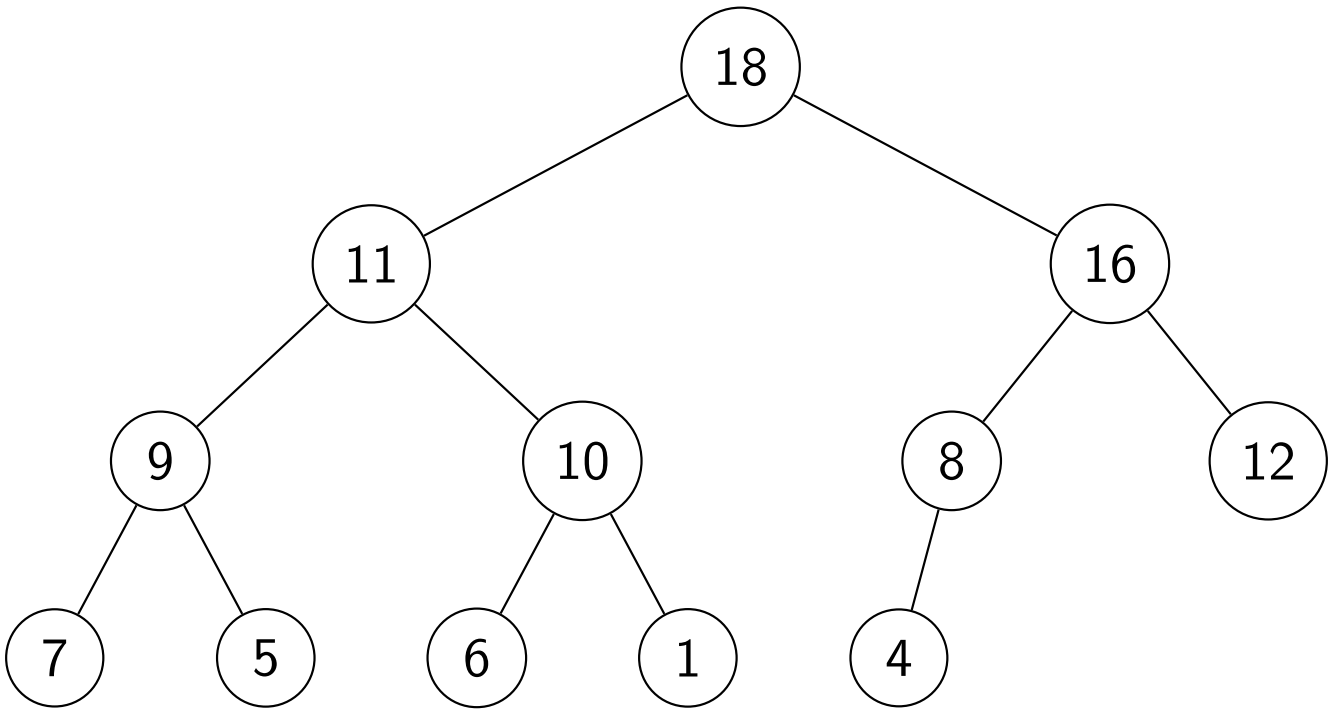
Propriété :

- Le nœud parent du nœud numéro i, $i > 0$, est $\frac{i-1}{2}$.
- Le fils gauche, s'il existe, du nœud de numéro i est $2i + 1$.
- Le fils droit, s'il existe, du nœud de numéro i est $2i + 2$.

Notion de *Tas binaire*

Définition

Soit E un ensemble ordonné.
Un *tas binaire* sur E est un arbre binaire tournoi sur E qui est aussi presque complet.



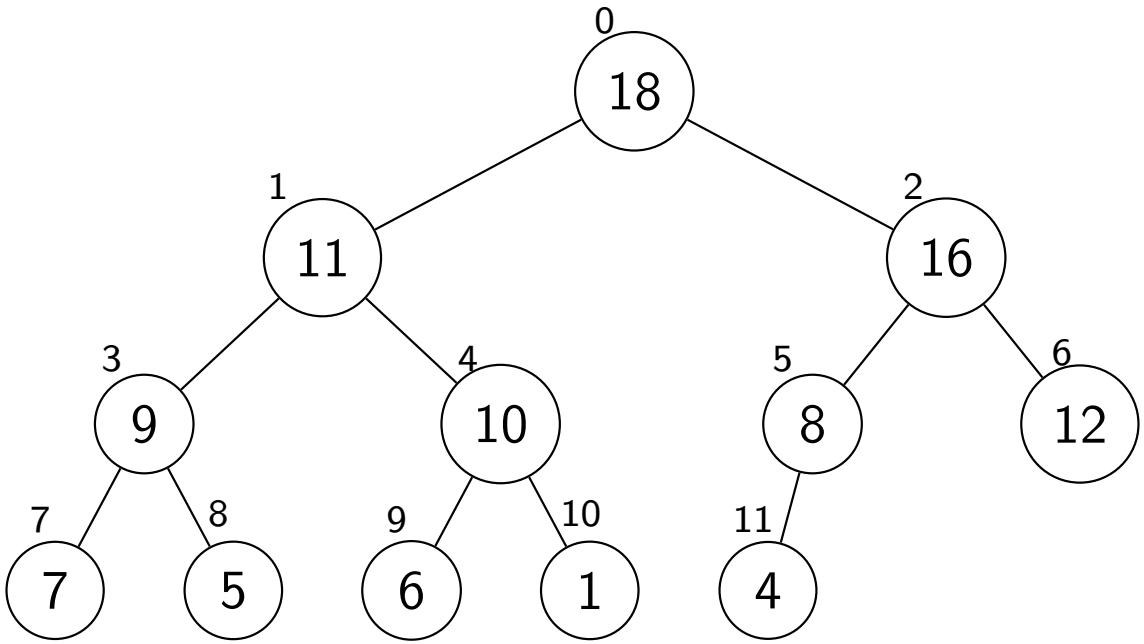
Un tas sur les entiers positifs

Outline

Implantation d'un tas par la table des pères 1 / 2

Rappel :

Une numérotation par niveaux d'un arbre presque parfait permet un calcul direct de la relation Parent.



En mémorisant les nœuds dans l'ordre d'un parcours en largeur, il n'est pas nécessaire de représenter les liens.

18	11	16	9	10	8	12	7	5	6	1	4	...

Implantation d'un tas par la table des pères 2 / 2

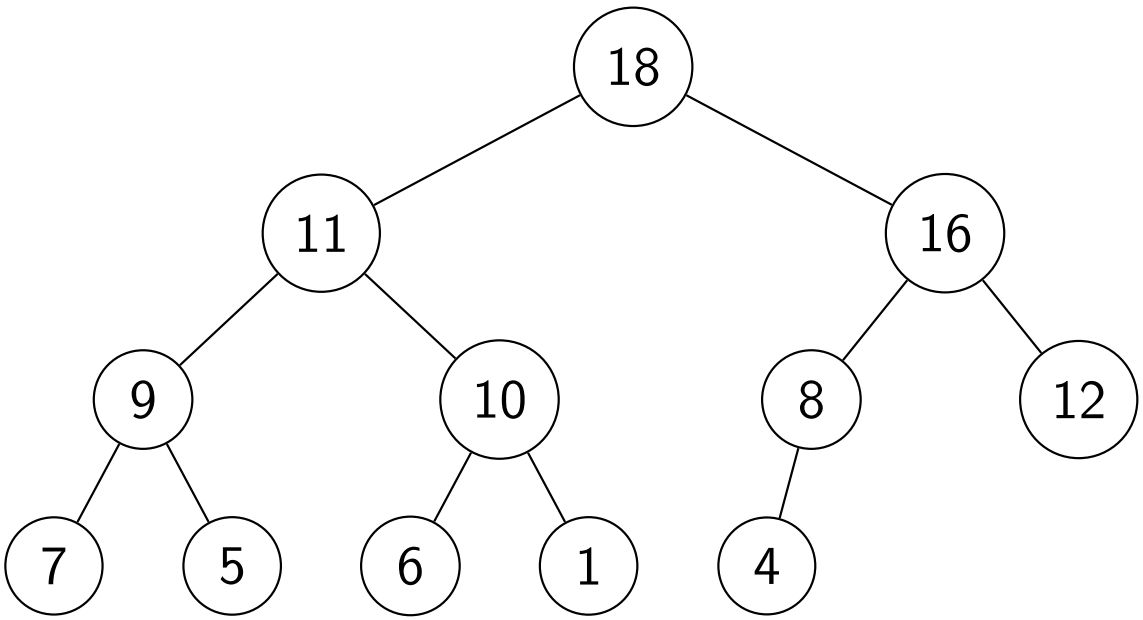
```
typedef struct {
    int *valeur;  /* zone de taille int */
    int dernier;
    int taille;
} Tas;
```

Outline

Ajout d'un élément dans un tas

On préserve la structure d'arbre parfait en ajoutant une feuille le plus à gauche possible du dernier niveau.

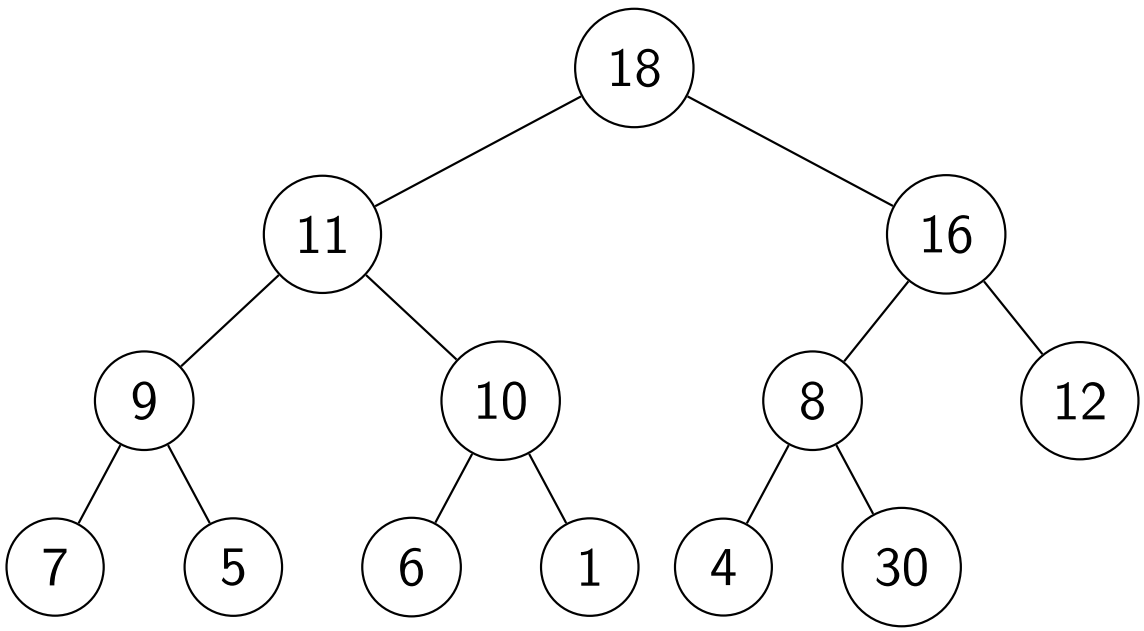
On remonte ensuite l'étiquette par échange avec celle du nœud parent pour retrouver une structure de tournoi.



Ajout d'un élément dans un tas

On préserve la structure d'arbre parfait en ajoutant une feuille le plus à gauche possible du dernier niveau.

On remonte ensuite l'étiquette par échange avec celle du nœud parent pour retrouver une structure de tournoi.

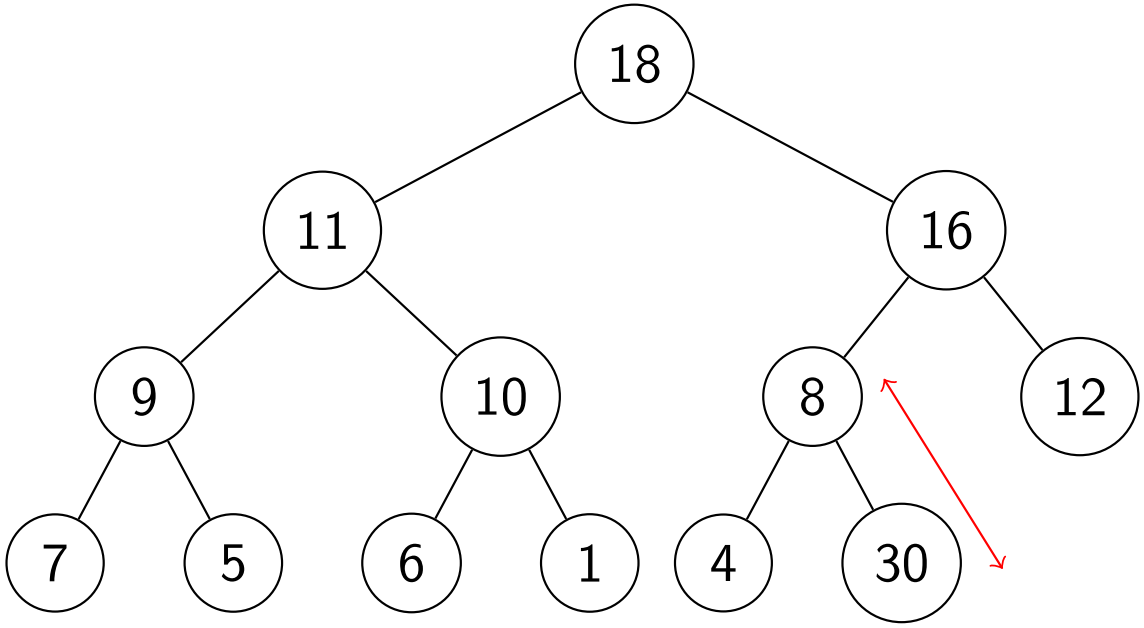


Ajout de 30 **au dernier niveau à droite**

Ajout d'un élément dans un tas

On préserve la structure d'arbre parfait en ajoutant une feuille le plus à gauche possible du dernier niveau.

On remonte ensuite l'étiquette par échange avec celle du nœud parent pour retrouver une structure de tournoi.

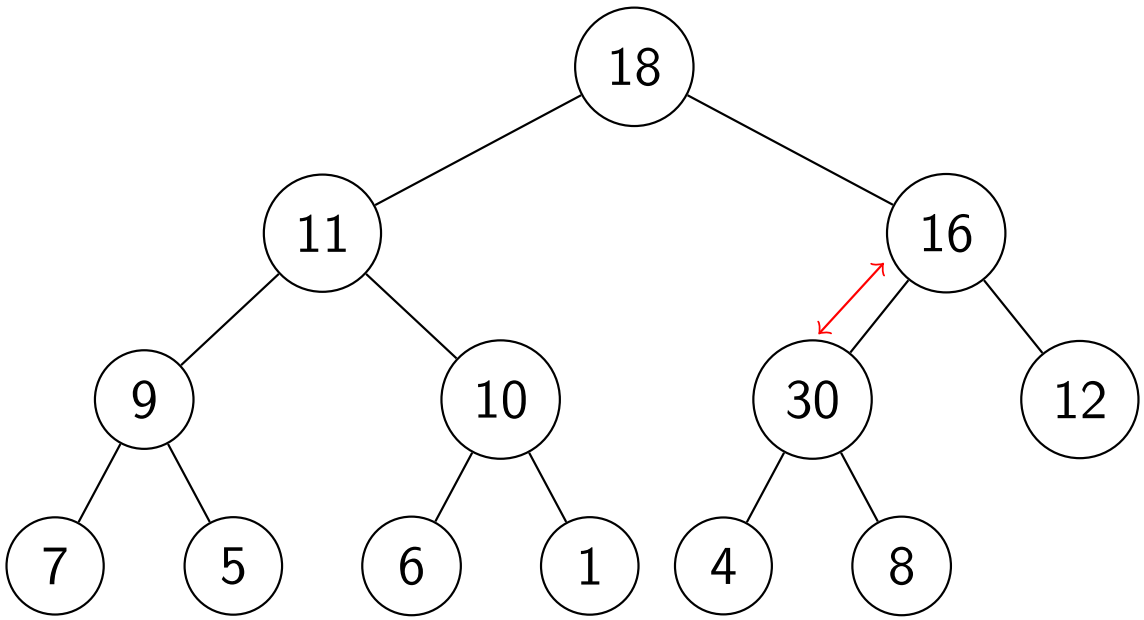


Remonté de 30 le long de sa branche pour retrouver une structure de tas binaire

Ajout d'un élément dans un tas

On préserve la structure d'arbre parfait en ajoutant une feuille le plus à gauche possible du dernier niveau.

On remonte ensuite l'étiquette par échange avec celle du nœud parent pour retrouver une structure de tournoi.

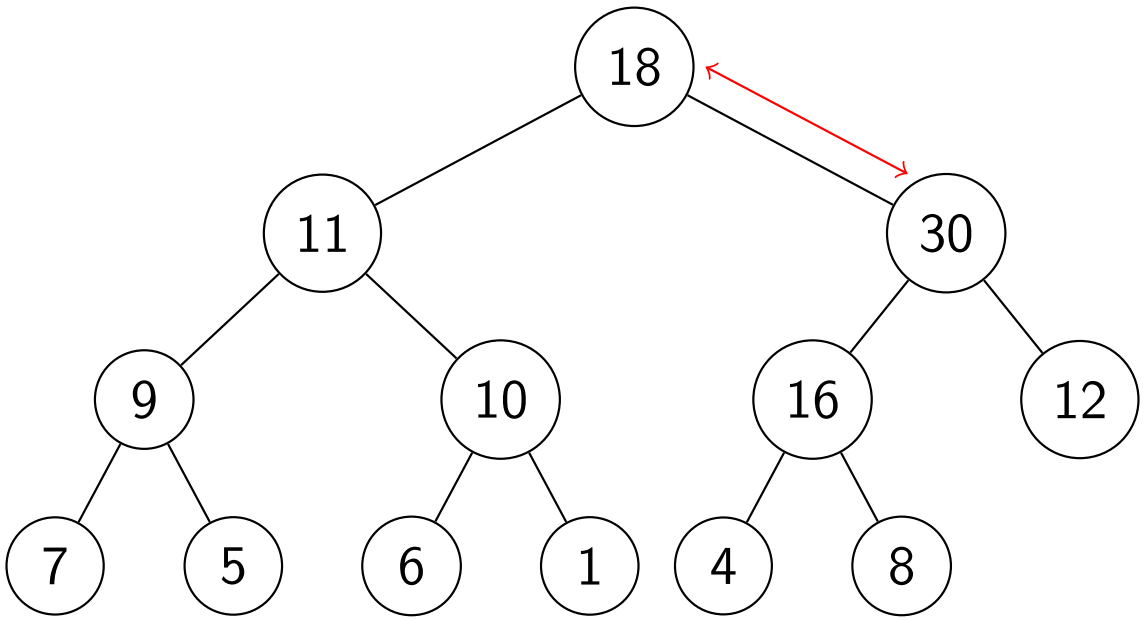


Remonté de 30 le long de sa branche pour retrouver une structure de tas binaire

Ajout d'un élément dans un tas

On préserve la structure d'arbre parfait en ajoutant une feuille le plus à gauche possible du dernier niveau.

On remonte ensuite l'étiquette par échange avec celle du nœud parent pour retrouver une structure de tournoi.

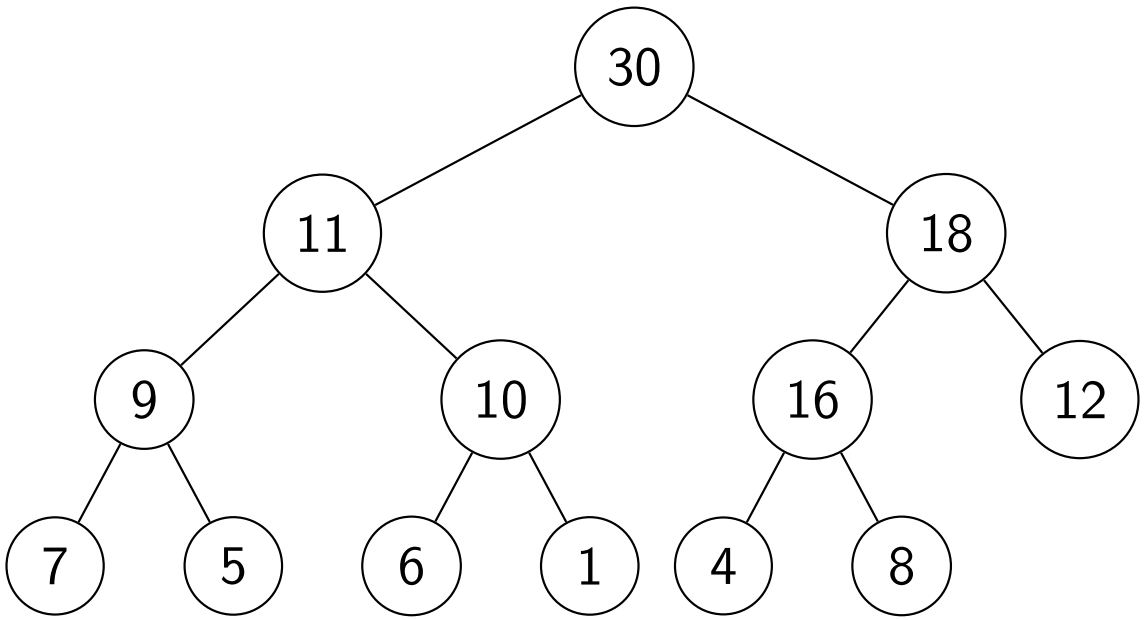


Remonté de 30 le long de sa branche pour retrouver une structure de tas binaire

Ajout d'un élément dans un tas

On préserve la structure d'arbre parfait en ajoutant une feuille le plus à gauche possible du dernier niveau.

On remonte ensuite l'étiquette par échange avec celle du nœud parent pour retrouver une structure de tournoi.



Remonté de 30 le long de sa branche pour retrouver une structure de tas binaire

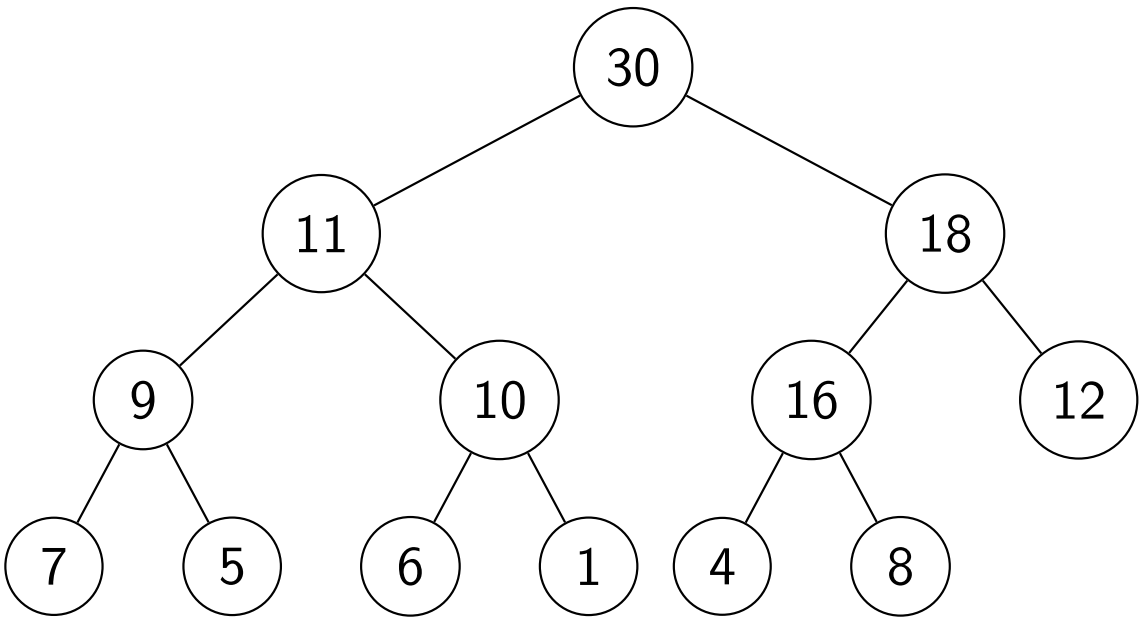
Implémentation de l'ajout dans un tas

```
int Ajoute(Tas *T,int x){
    int enfant,parent;
    if (T->dernier == taille - 1)
        return 0;
    T->dernier++;
    T->valeur[T->dernier] = x;
    enfant = T->dernier; parent = (enfant - 1) / 2;
    while ((enfant > 0) &&
           (T->valeur[parent] > T->valeur[enfant])){
        Echange(&(T->valeur[enfant]), &(T->valeur[parent]));
        enfant = parent; parent = (enfant - 1) / 2;
    }
    return 1;
}
```

Suppression du maximum d'un tas

Le maximum est à la racine. On supprime la feuille la plus à droite du dernier niveau et on remplace la racine par son étiquette.

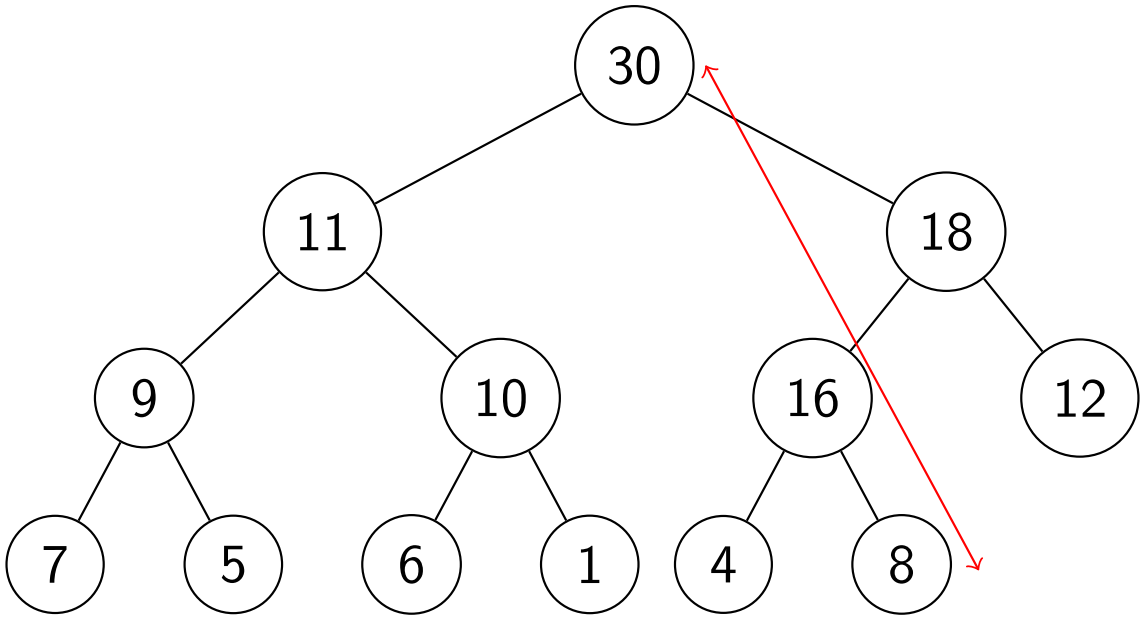
On descend la nouvelle étiquette de la racine par échange avec la plus grande de ses enfants pour retrouver une structure d'arbre tournoi.



Suppression du maximum d'un tas

Le maximum est à la racine. On supprime la feuille la plus à droite du dernier niveau et on remplace la racine par son étiquette.

On descend la nouvelle étiquette de la racine par échange avec la plus grande de ses enfants pour retrouver une structure d'arbre tournoi.

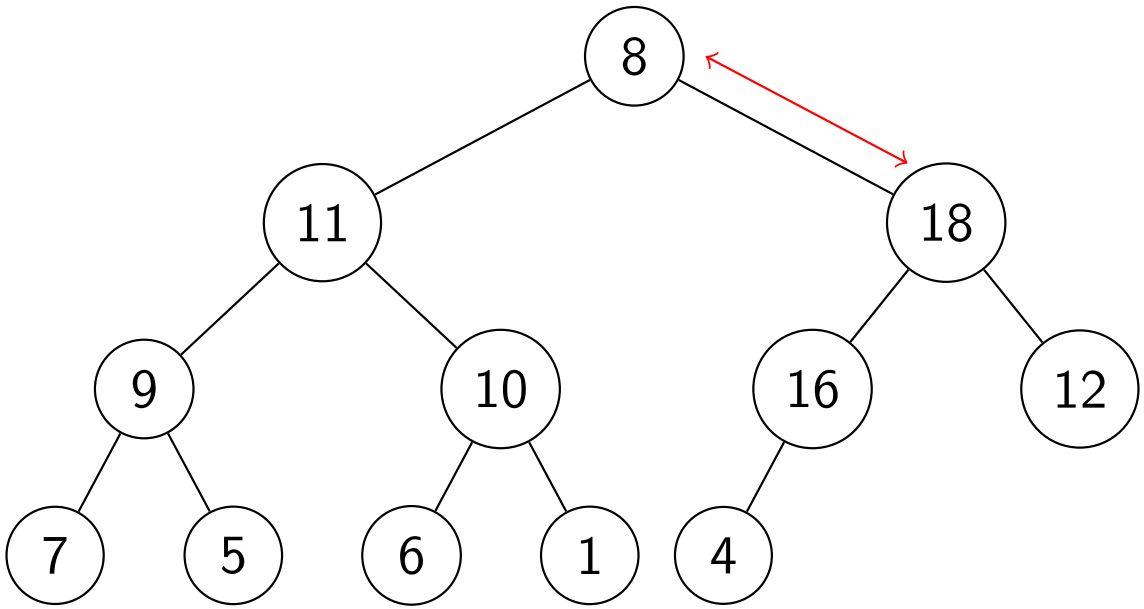


Suppression du maximum et échange avec le dernier élément

Suppression du maximum d'un tas

Le maximum est à la racine. On supprime la feuille la plus à droite du dernier niveau et on remplace la racine par son étiquette.

On descend la nouvelle étiquette de la racine par échange avec la plus grande de ses enfants pour retrouver une structure d'arbre tournoi.

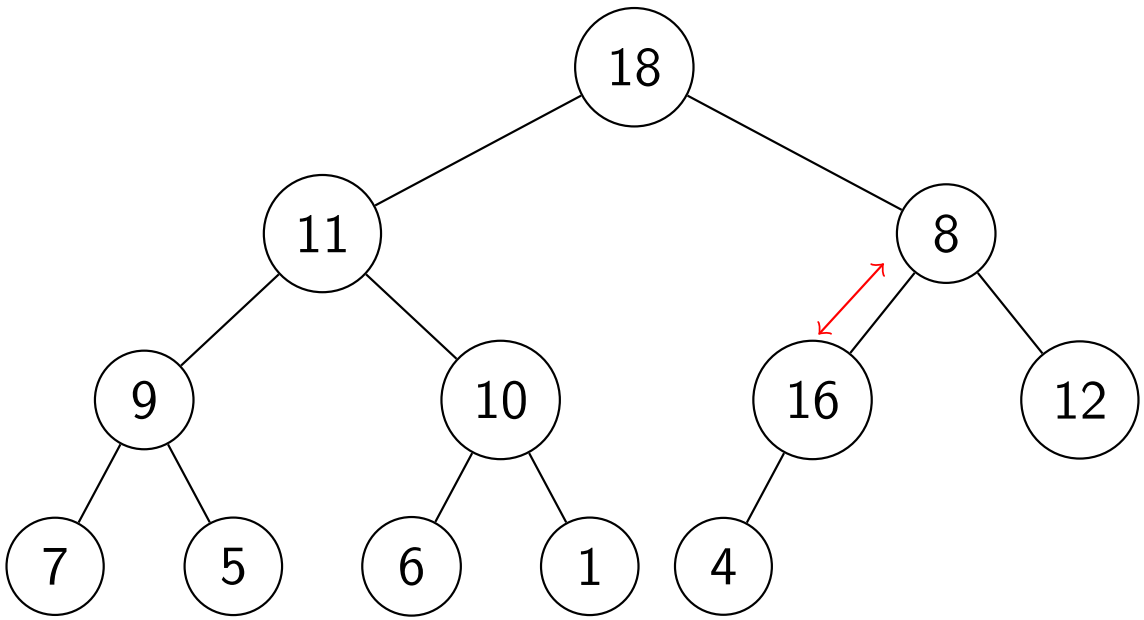


Descente de la racine le long de sa branche pour retrouver une structure d'arbre tournoi

Suppression du maximum d'un tas

Le maximum est à la racine. On supprime la feuille la plus à droite du dernier niveau et on remplace la racine par son étiquette.

On descend la nouvelle étiquette de la racine par échange avec la plus grande de ses enfants pour retrouver une structure d'arbre tournoi.

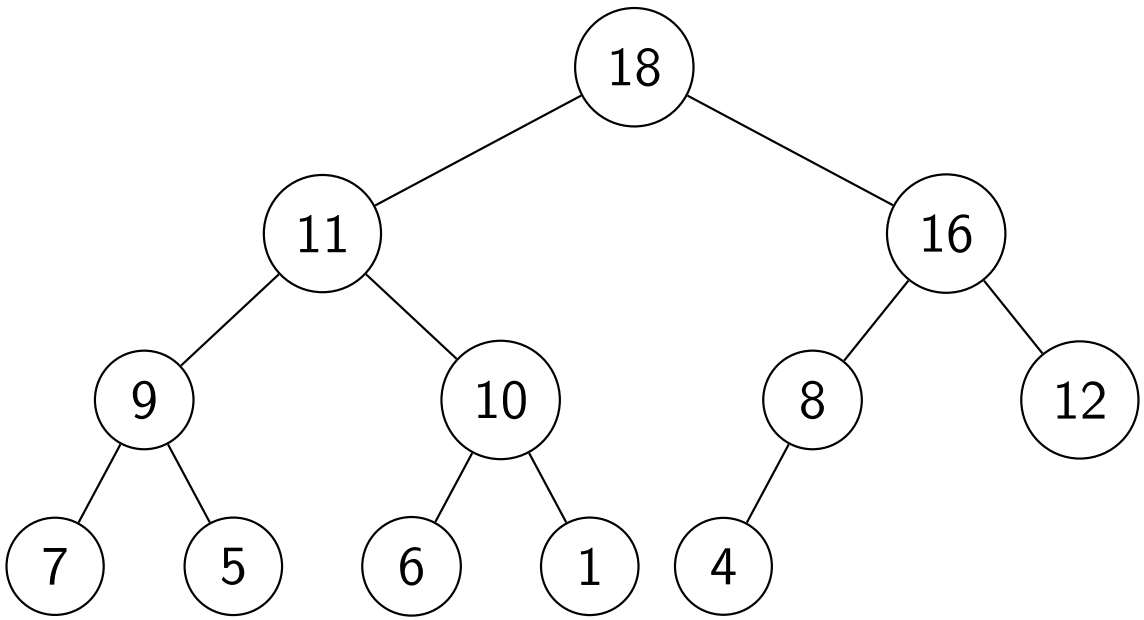


Descente de la racine le long de sa branche pour retrouver une structure d'arbre tournoi

Suppression du maximum d'un tas

Le maximum est à la racine. On supprime la feuille la plus à droite du dernier niveau et on remplace la racine par son étiquette.

On descend la nouvelle étiquette de la racine par échange avec la plus grande de ses enfants pour retrouver une structure d'arbre tournoi.



Implémentation de la suppression dans un tas

Laissé en exercice !

Compléxité de l'ajout et de la suppression

Le nombre d'échanges pour l'ajout ou la suppression du minimum est borné par la hauteur du tas.

La hauteur d'un arbre binaire presque complet à n nœuds est $\lfloor \log_2(n) \rfloor$.

L'ajout et la suppression dans un tas se font donc en $\mathcal{O}\big(\log_2(n)\big)$.

Outline

Implémentation des file de priorités

```
typedef struct {
    int *valeur;  /* zone de taille int */
    int dernier;
    int taille;
} Tas, FilePriorite;
```

Avec les opérations :

- `Elt maximum(FilePriorite f)` en $\mathcal{O}(1)$.
- `int inserer(FilePriorite * f, Elt x)` en $\mathcal{O}\big(\log_2(n)\big)$.
- `int supprimer(FilePriorite * f, Elt * res)` en $\mathcal{O}\big(\log_2(n)\big)$.

Tris par tas

Le nombre d'échanges pour l'ajout ou la suppression du minimum est borné par la hauteur du tas. On a donc naturelement un algorithme de tri en $\mathcal{O}(n \log(n))$:

- Ajouter un à un les n éléments à trier dans un tas ;
- Supprimer n fois le maximum.

On obtient une suite triée par ordre croissant. Oui, j'ai bien dit par ordre croissant !

Pour un tableau, ce tri peut s'effectuer sur place :

- construction du tas :

  le début du tableau contient le tas déjà construit, auquel on ajoute les éléments successivement.

- destruction du tas :

  on supprime les maximums successifs en les échangeant avec la dernière feuille du tas.

Exemple de tri par tas 1 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

9	1	7	16	4	8	6	5

Exemple de tri par tas 1 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

9	1	7	16	4	8	6	5

$\longleftrightarrow$
tas

Exemple de tri par tas 1 / 2

Tableau initial à trier :

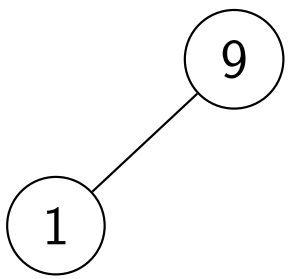
9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

9	1	7	16	4	8	6	5

$\longleftrightarrow$
tas



Exemple de tri par tas 1 / 2

Tableau initial à trier :

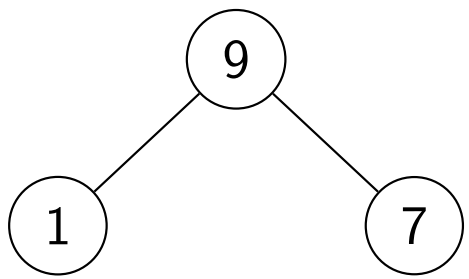
9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

9	1	7	16	4	8	6	5

$\longleftrightarrow$
tas



Exemple de tri par tas 1 / 2

Tableau initial à trier :

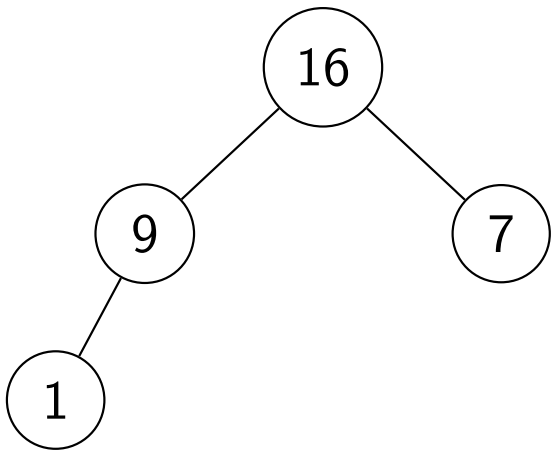
9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

9	1	7	16	4	8	6	5

tas



Après échanges, le tableau devient :

16	9	7	1	4	8	6	5

Exemple de tri par tas 1 / 2

Tableau initial à trier :

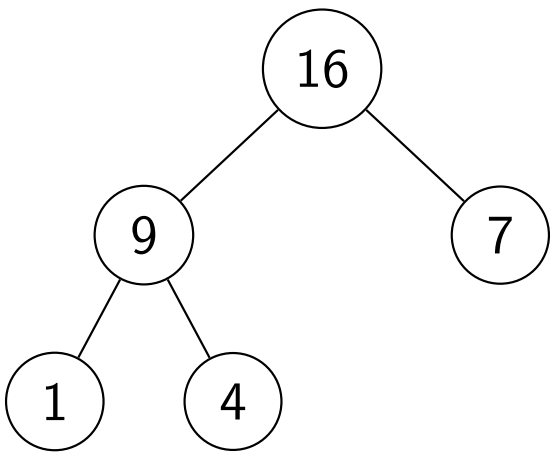
9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

16	9	7	1	4	8	6	5

tas



Exemple de tri par tas 1 / 2

Tableau initial à trier :

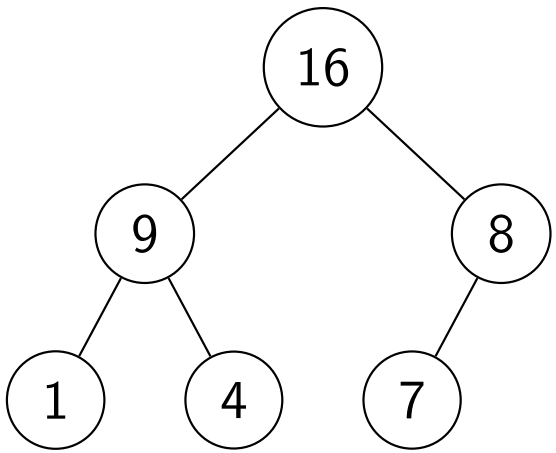
9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

16	9	7	1	4	8	6	5

tas



Après échanges, le tableau devient :

16	9	8	1	4	7	6	5

Exemple de tri par tas 1 / 2

Tableau initial à trier :

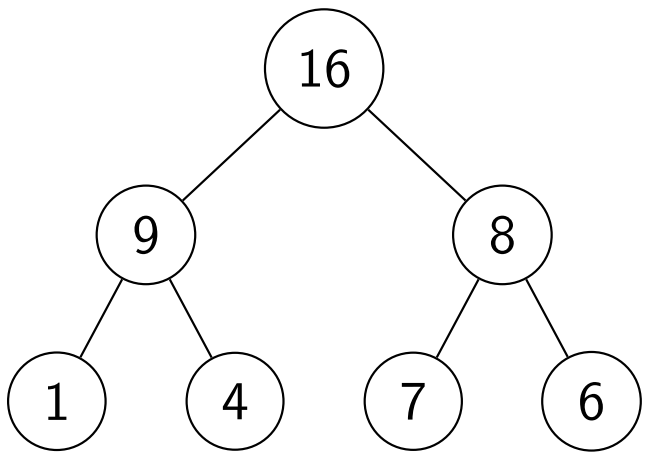
9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

16	9	8	1	4	7	6	5

tas



Exemple de tri par tas 1 / 2

Tableau initial à trier :

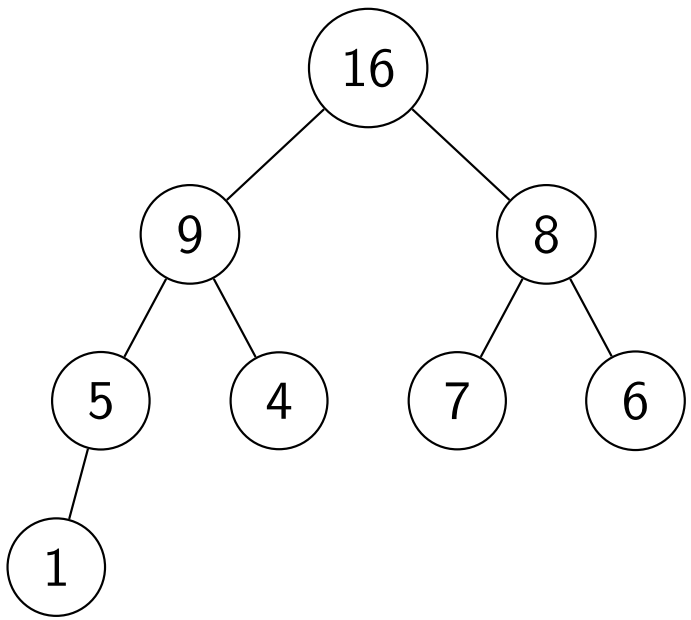
9	1	7	16	4	8	6	5

Construction du tas :

Tableau à trier :

16	9	8	1	4	7	6	5

tas



Après échanges, le tableau devient :

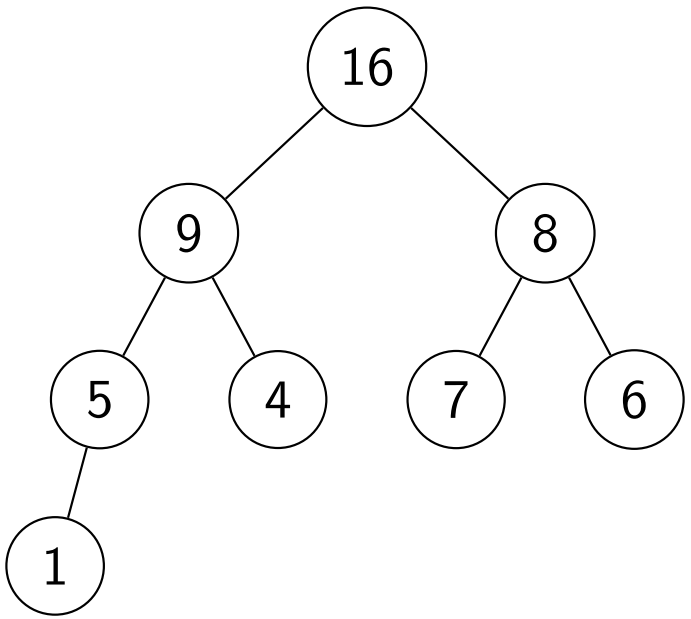
16	9	8	5	4	7	6	1

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas :

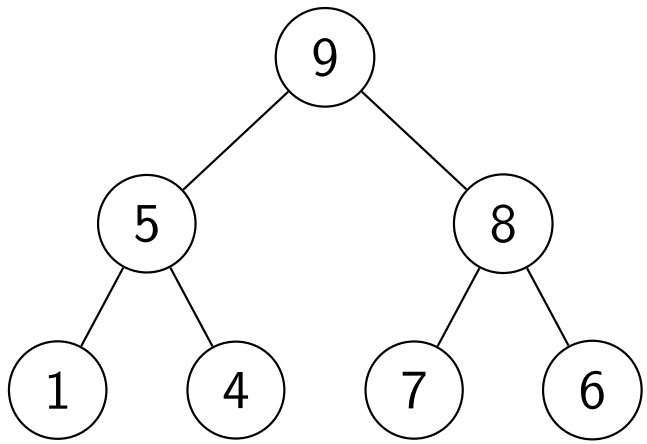


Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 16



Après échanges, le tableau devient :

9	5	8	1	4	7	6	16

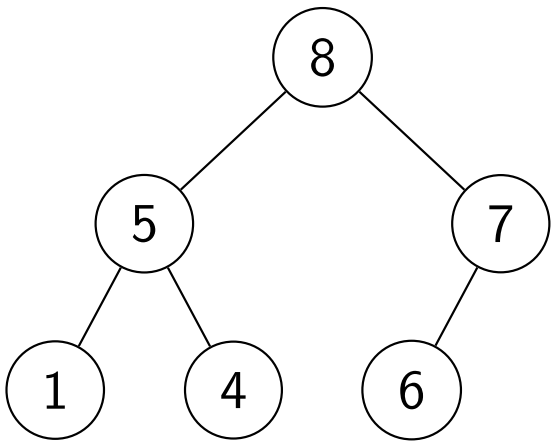
$\longleftrightarrow$ tas

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 9



Après échanges, le tableau devient :

8	5	7	1	4	6	9	16

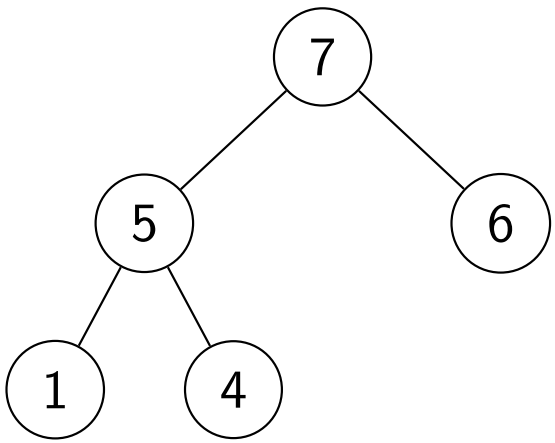
$\longleftrightarrow$
tas

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 8



Après échanges, le tableau devient :

7	5	6	1	4	8	9	16

$\longleftrightarrow$

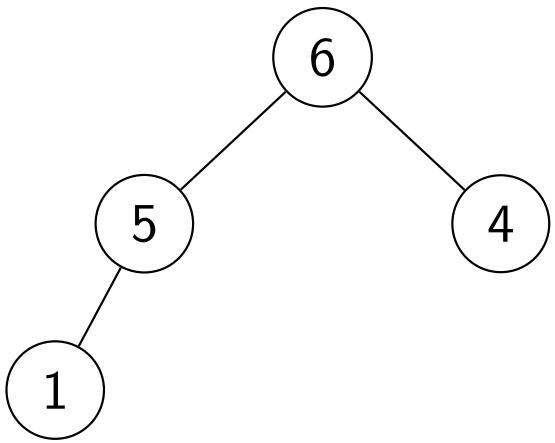
tas

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 7



Après échanges, le tableau devient :

6	5	4	1	7	8	9	16

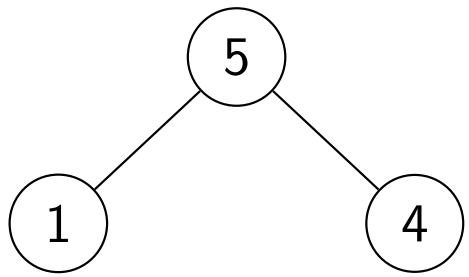
$\longleftrightarrow$
tas

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 6



Après échanges, le tableau devient :

5	1	4	6	7	8	9	16

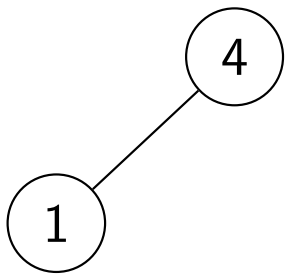
$\longleftrightarrow$
tas

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 5



Après échanges, le tableau devient :

4	1	5	6	7	8	9	16

$\longleftrightarrow$
tas

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 4

Après échanges, le tableau devient :

1	4	5	6	7	8	9	16

$\longleftrightarrow$
tas

Exemple de tri par tas 2 / 2

Tableau initial à trier :

9	1	7	16	4	8	6	5

Destruction du tas : suppression du max actuel $\rightsquigarrow$ 1

Après échanges, le tableau devient :

1	4	5	6	7	8	9	16

Il est désormais trié !