INF421-a
Bases de la programmation et de l'algorithmique

(Bloc 6/ 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

29 septembre 2006

Aujourd'hui

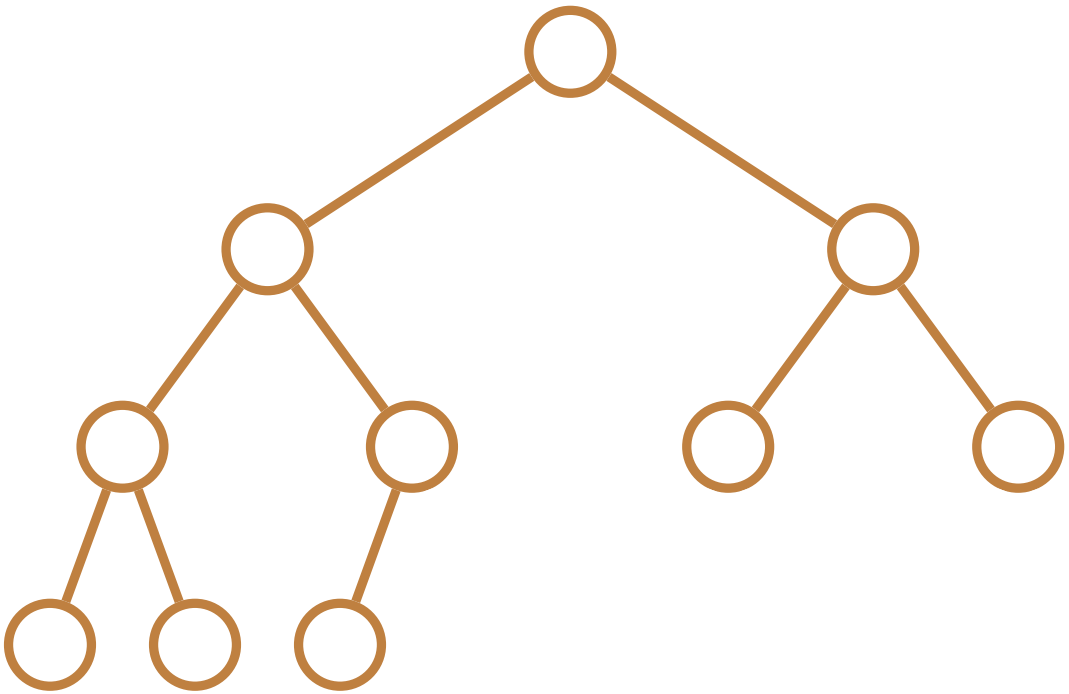
Tas

Un tas en Java

Arbres binaires de recherche

Définition des tas

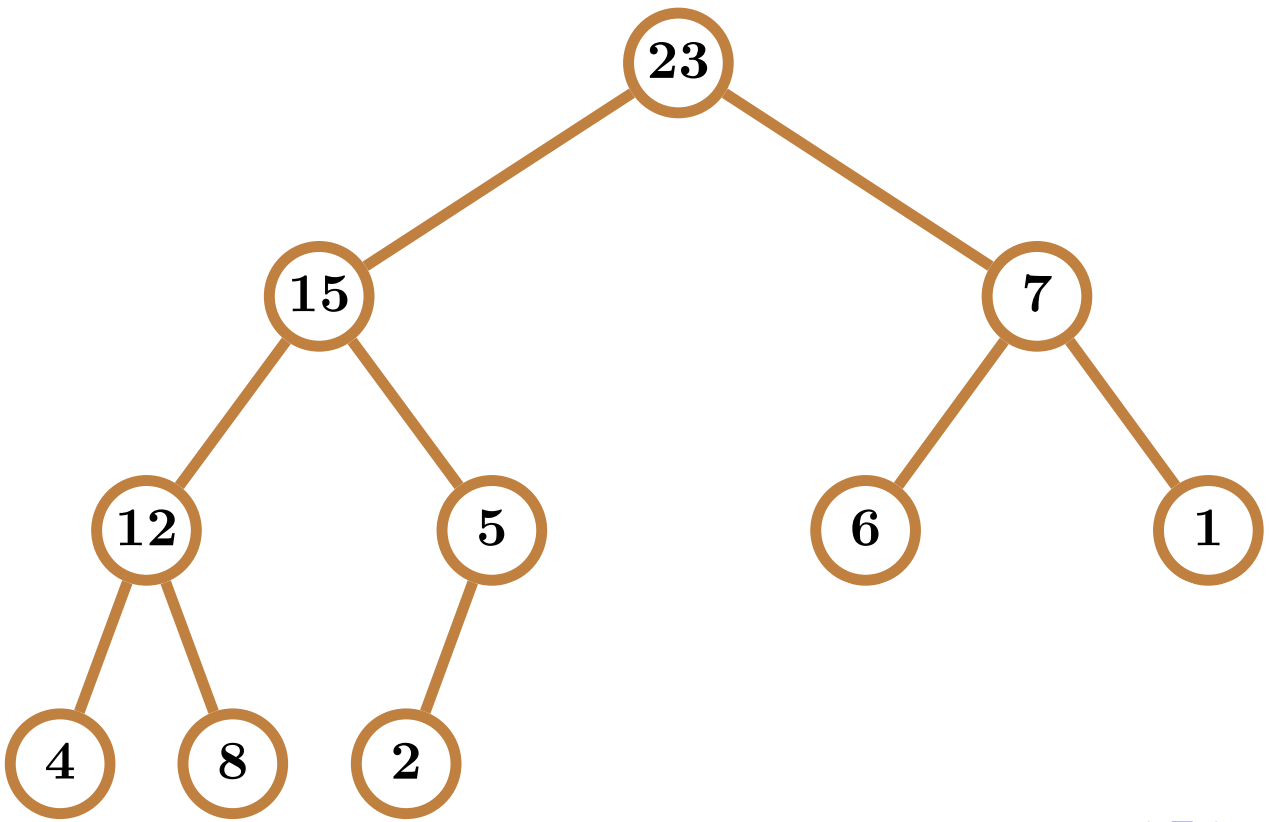
Un arbre binaire est tassé quand tous ses niveaux sont entièrement remplis à l'exception peut-être du dernier niveau, et ce dernier niveau est rempli à gauche.



La hauteur d'un arbre tassé à n nœuds est $\lfloor \log_2 n \rfloor$.

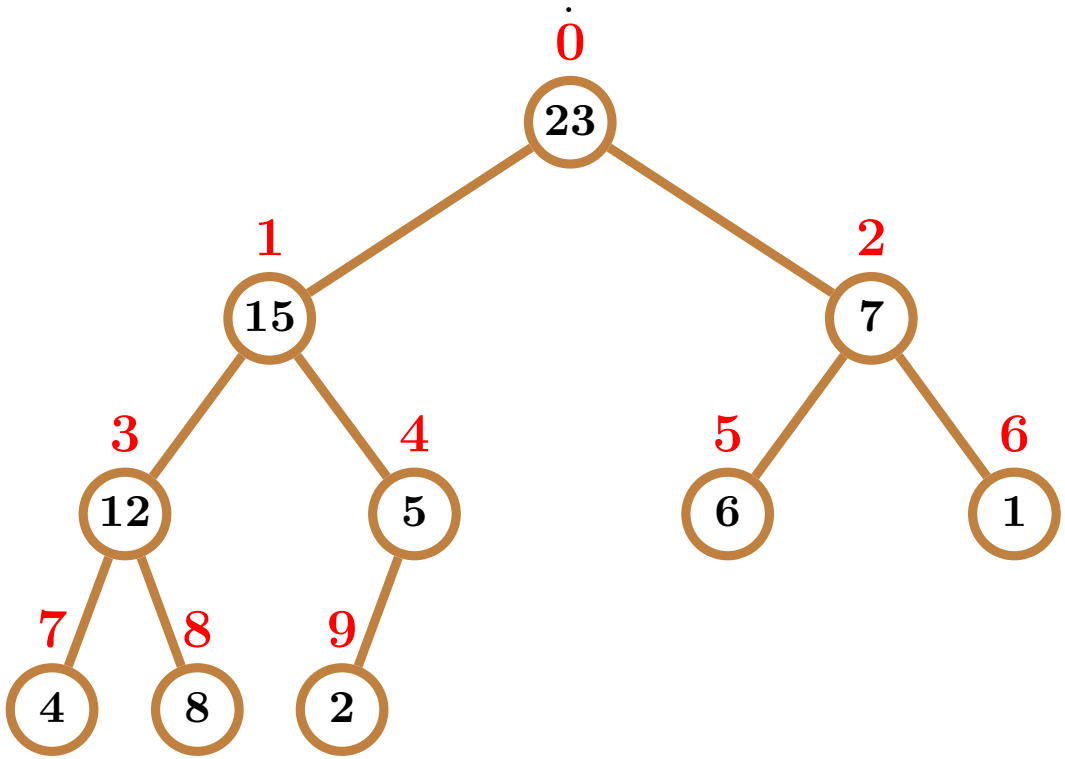
Définition des tas d'entiers

Un tas est un arbre binaire tassé dans lequel le contenu de chaque noeud est supérieur ou égal à celui de ses fils



Définition des tas d'entiers

- Les nœuds d'un arbre tassé sont numérotés en largeur, de gauche à droite
- Ces numéros sont des indices dans un tableau
- L'élément d'indice i = le contenu du nœud de numéro i



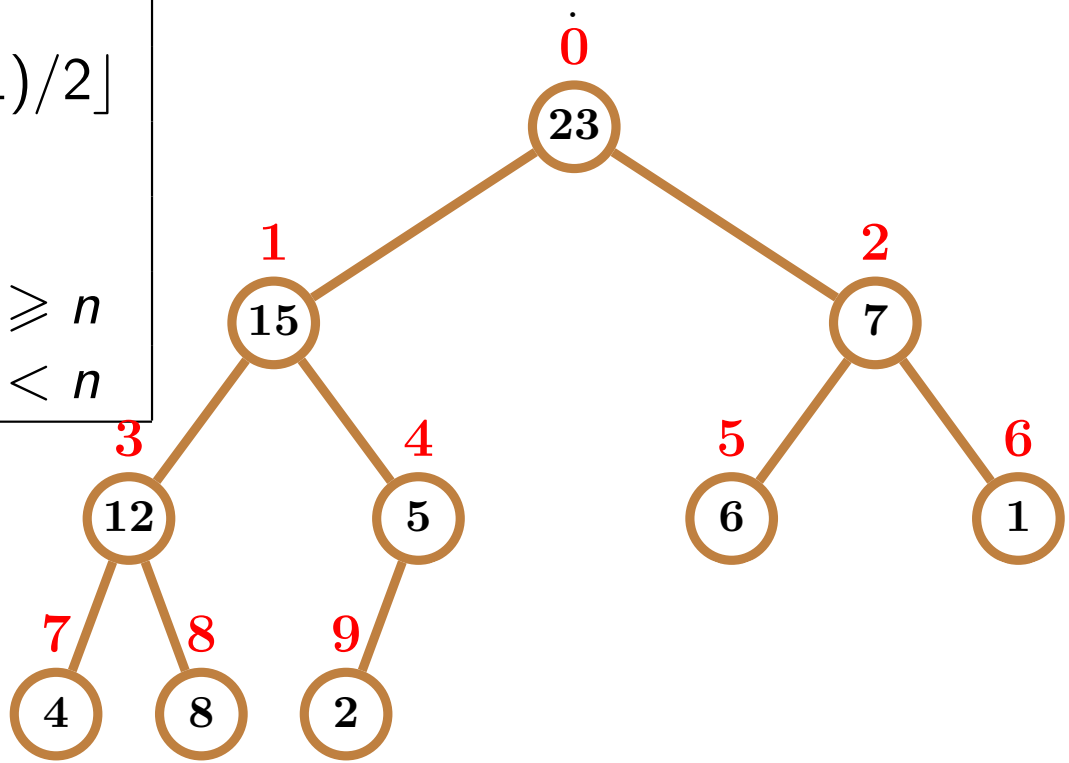
i	0	1	2	3	4	5	6	7	8	9
a_i	23	15	7	12	5	6	1	4	8	2

L'hérédité des tas d'entiers

racine	:	0
père de i	:	$\lfloor (i-1)/2 \rfloor$
fils gauche de i	:	$2i+1$
fils droit de i	:	$2i+2$
i est une feuille	:	$2i+1 \geqslant n$
i a un fils droit	:	$2i+2 < n$

.

.



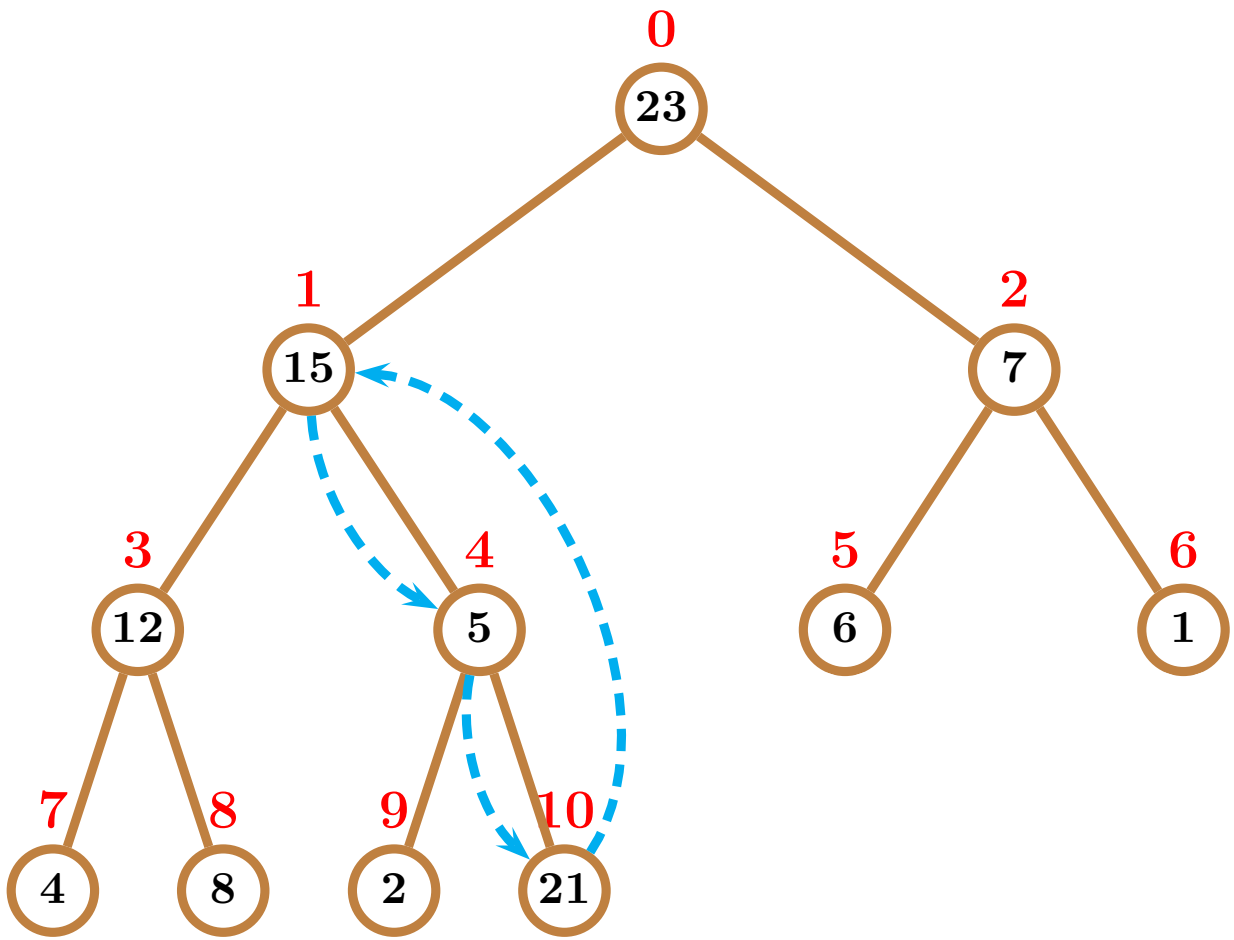
L'hérédité des tas d'entiers

- Entre le niveau 1 et h, nous avons $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ éléments
- Les indices des éléments de niveau h sont compris entre $2^{h-1} - 1$ et $2^h - 2$
- Soit j une élément de niveau h, *i.e.*, $j = 2^{h-1} - 2 + \lambda$ avec $\lambda \in \{1, ..., 2^h - 2 - 2^{h-1} + 1\}$
- Son fils droit : $2^h - 2 + 2 * \lambda = 2^h - 2 + 2j - 2^h + 4 = 2j + 2$
- Son fils gauche : $2j + 2 - 1 = 2j + 1$

Insérer dans un tas

Insérer v

- L'élément est ajouté comme contenu d'un nouveau nœud à la fin du dernier niveau de l'arbre
- Tant que le contenu du père est plus petit que v, le contenu du père est descendu vers le fils.
- Remplacer par v le dernier contenu abaissé

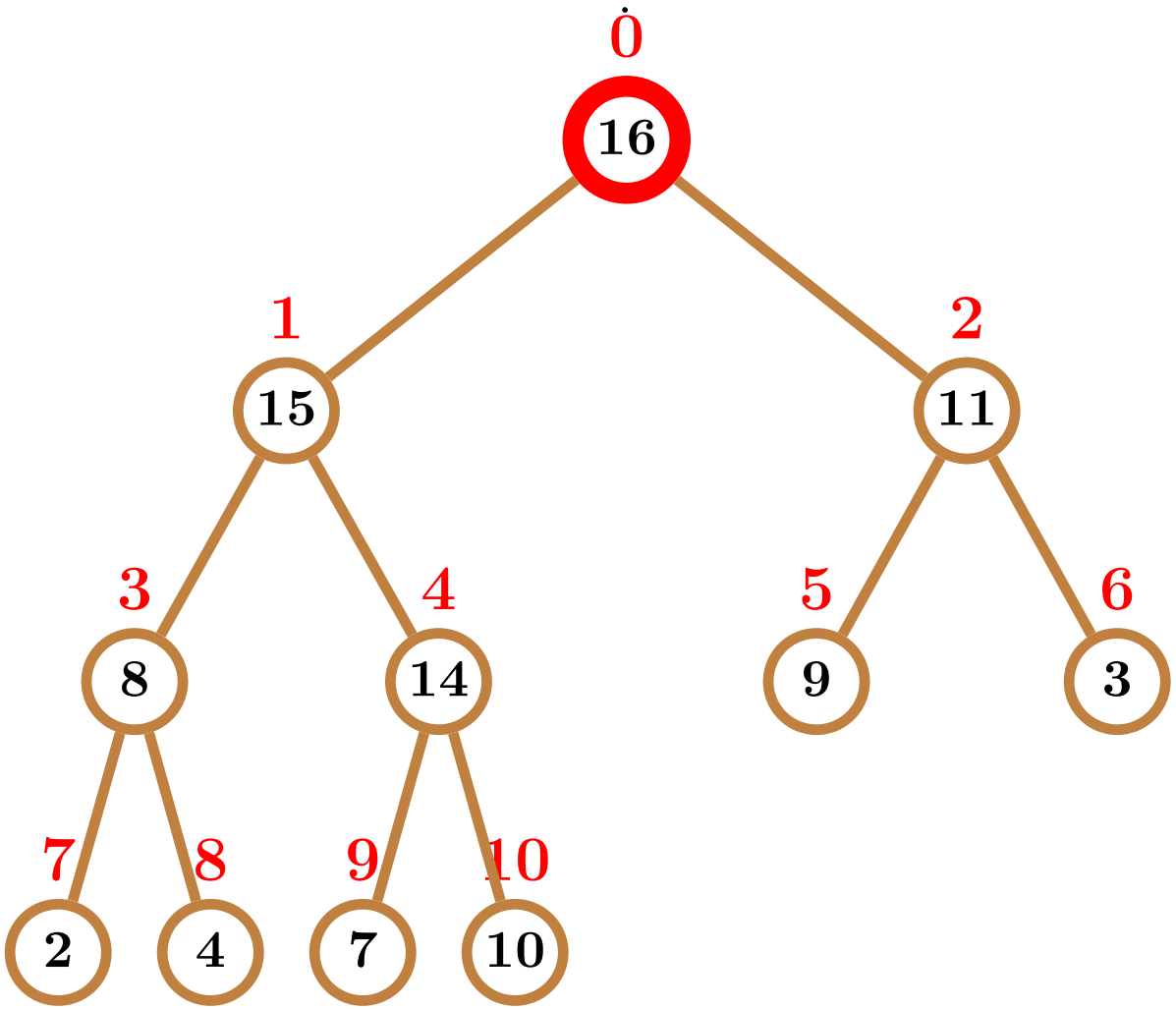


Supprimer dans un tas (la racine)

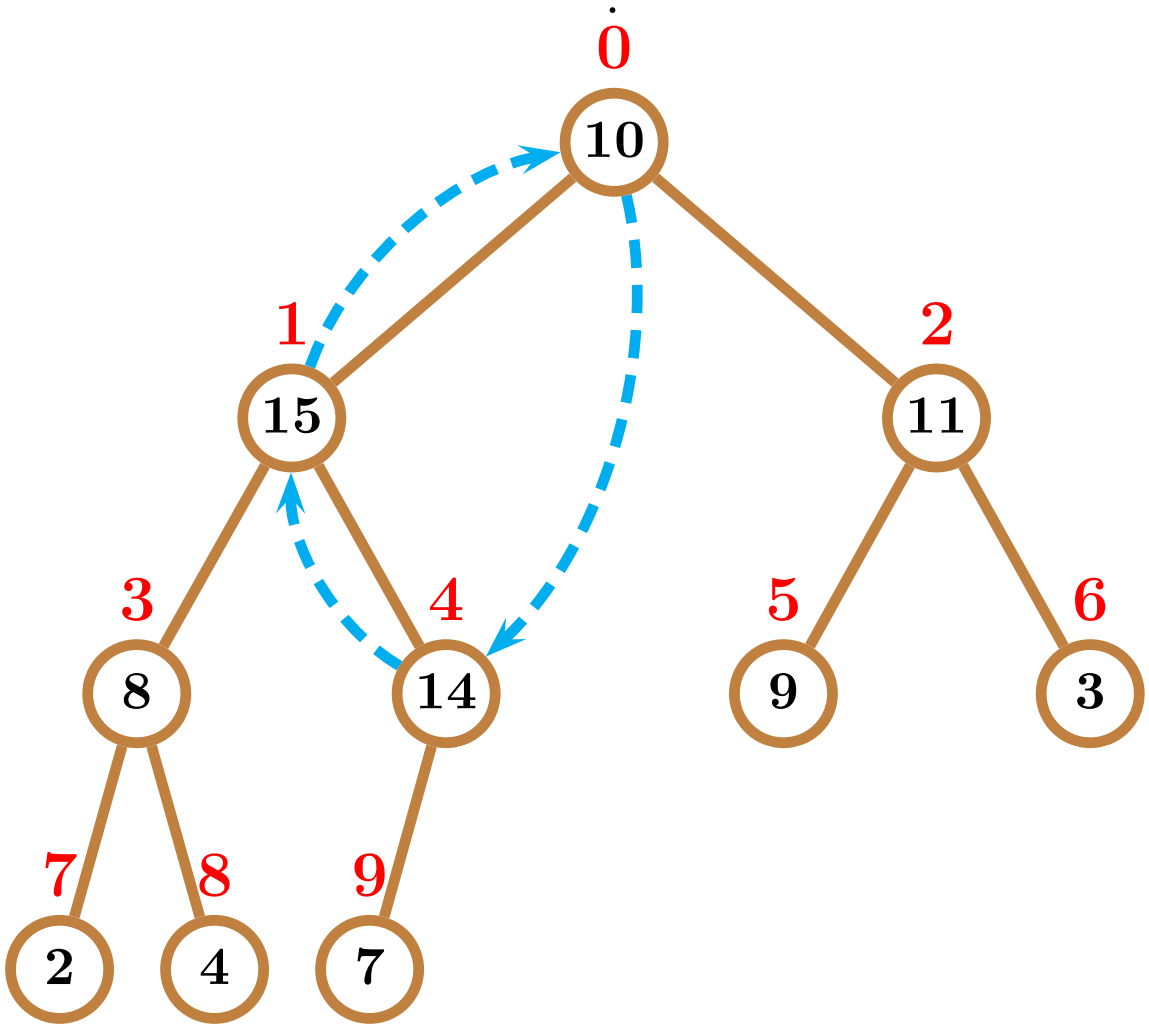
On enlève la racine du tas

- Retasser : le contenu du nœud le plus à droite du dernier niveau est transféré vers la racine
- Comparer : la racine est comparée au + grand des fils
  - Si fils $\geq$ père remonter et remplacer le contenu du père
  - itérer

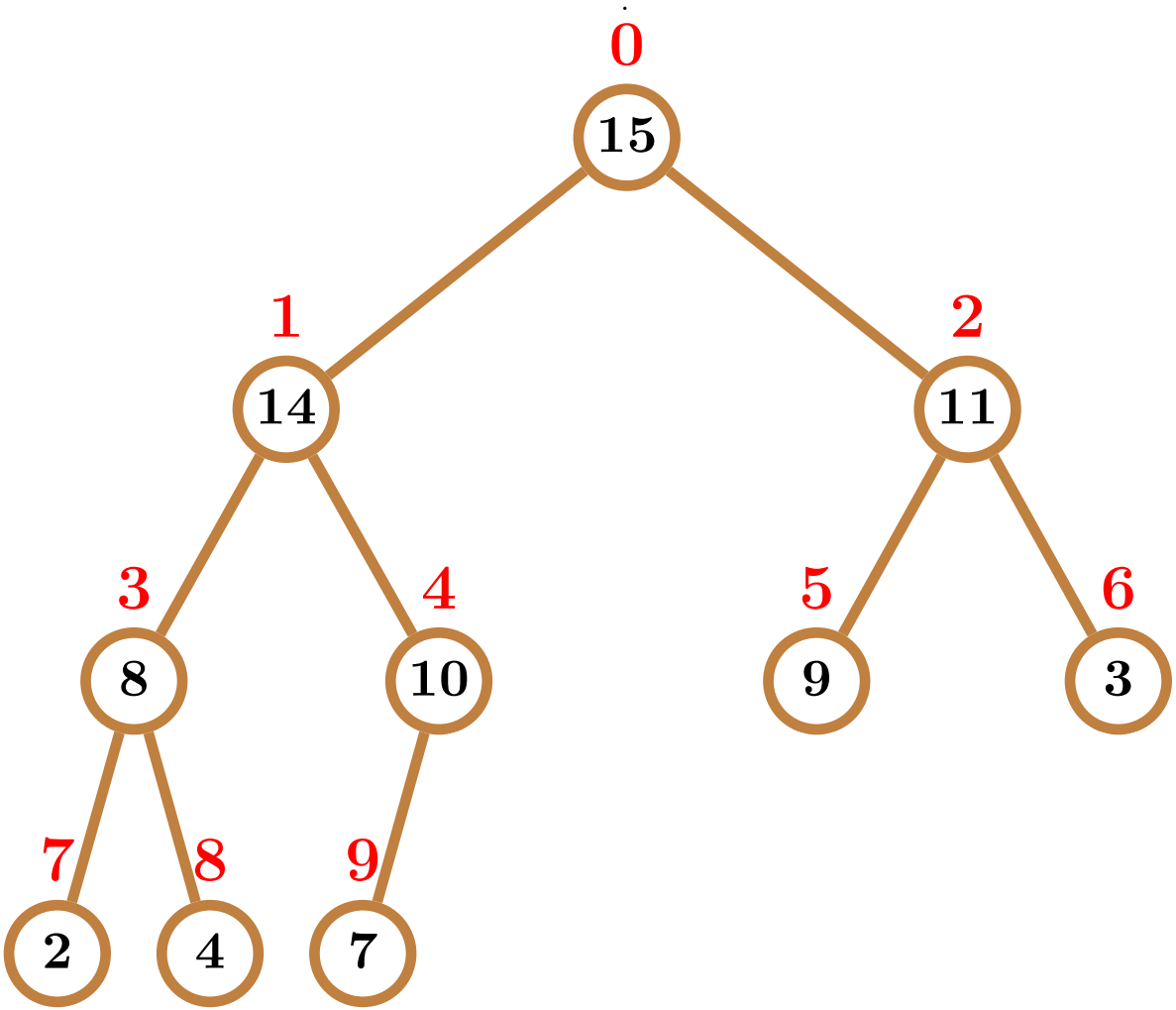
Supprimer dans un tas (la racine)



Supprimer dans un tas (la racine)



Supprimer dans un tas (la racine)



Insertion / suppression : complexité

La complexité des opérations de suppression et d'insertion est majorée par la hauteur de l'arbre

La hauteur d'un arbre tassé à n nœuds est $\lfloor \log_2 n \rfloor$

Insertion et suppression en $O(\log n)$

Aujourd'hui

Tas

Un tas en Java

Arbres binaires de recherche

Un tas en Java

- Un tableau pour représenter l'arbre. Rappel :
  - Les nœuds d'un arbre tassé sont numérotés en largeur, de gauche à droite
  - Ces numéros sont des indices dans un tableau a
  - L'élément d'indice i = le nœud de numéro i
- Trois méthodes `maximum()`, `inserer()`, `supprimer()`

Un tas en Java

```
class Tas {
    int[] a;
    int nTas = 0;
    Tas(int n) {
        nTas = 0;
        a = new int[n];
    }
    int maximum() {
        return a[0];
    }
```

```
int pere(int i) {
  return (i - 1) / 2
}
int gauche(int i) {
  return 2 * i + 1;
}
int droite(int i) {
  return 2 * i + 2;
}
boolean estUneFeuille(int i) {
  return (2 * i + 1 ≥ nTas);
}
boolean aUnFilsDroit(int i) {
  return (2 * i + 2 <  nTas);
}
```

Un tas en Java

```
void ajouter(int v) {
    int i = nTas;
    ++nTas;
    while (i > 0 && a[pere(i)] ≤ v) {
        a[i] = a[pere(i)];
        i = pere(i);
    }
    a[i] = v;
}
```

Un tas en Java

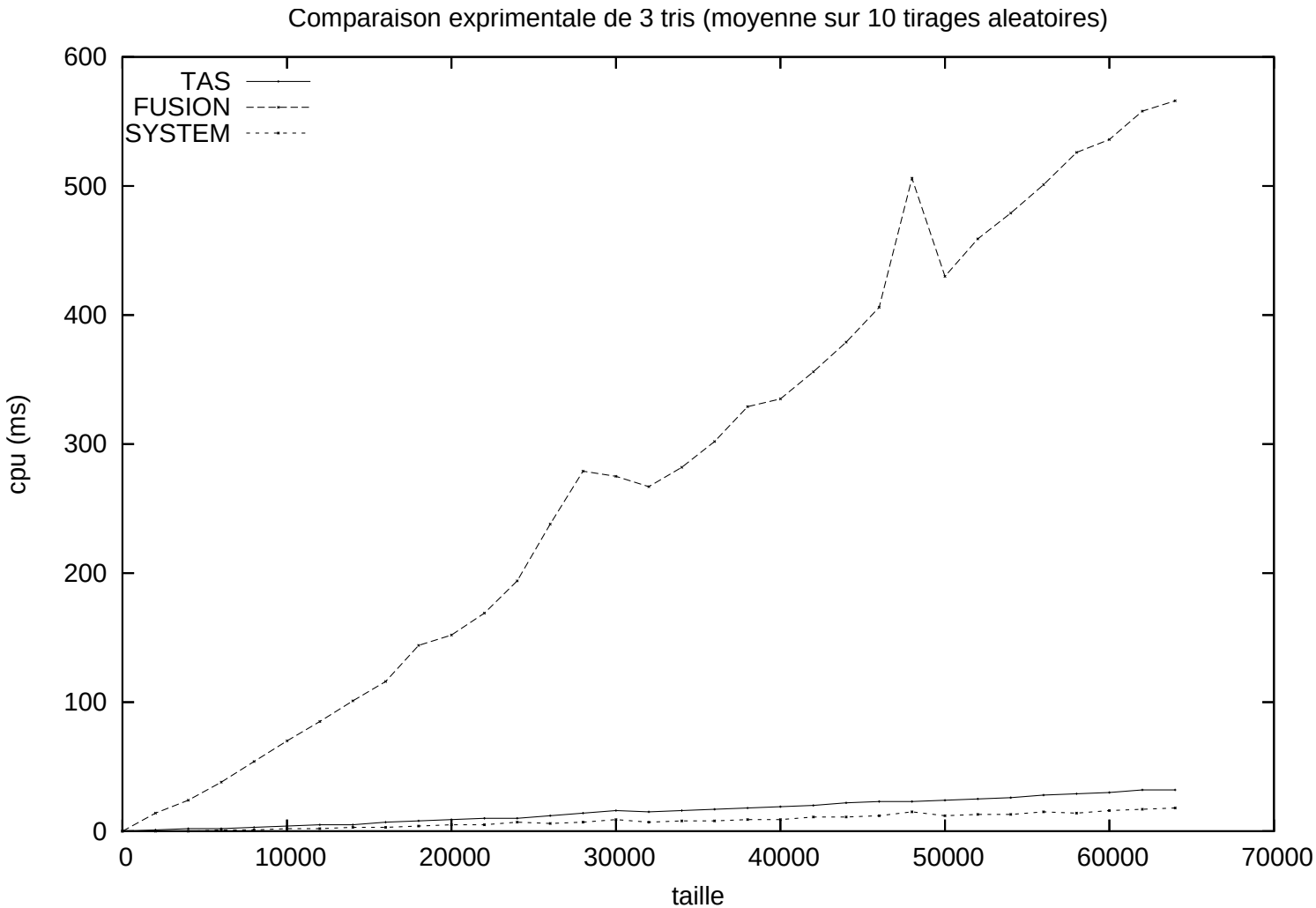
```
void supprimer() {
    nTas = nTas - 1;
    a[0] = a[nTas];
    int v = a[0];
    int i = 0;
    while (!estUneFeuille(i)) {
        int j = gauche(i);
        if (aUnFilsDroit(i) && a[droite(i)] > a[gauche(i)])
            j = droite(i);
        if (v ≥ a[j])
            break;
        a[i] = a[j];
        i = j;
    }
    a[i] = v;
}
```

Trier avec des tas

- n éléments à trier
- Les mettre dans le tas $n \times O(\log n)$
- Et les retirer ! $n \times O(\log n)$

```
static void triParTas(int[] a) {
    int n = a.length;
    Tas t = new Tas(n);
    for (int i = 0; i < n; i++)
        t.ajouter(a[i]);
    for (int i = n - 1; i ≥ 0; --i) {
        int v = t.maximum();
        t.supprimer();
        a[i] = v;
    }
    return;
}
```

Un tas en Java



Résumé sur les Tas

Structure de données utile pour

- trier des données
- gérer un ensemble de données en ne faisant
  - qu'ajouter des données
  - que retirer de l'ensemble des données la plus grande

Toutes les opérations en $O(\log n)$

Attention : Un tas n'est pas une "bonne" structure de donnée pour rechercher un élément quelconque

Aujourd'hui

Tas

Un tas en Java

Arbres binaires de recherche

Les arbres binaires de recherche

But du jeu : "Gérer" (insertion, suppression, recherche, *etc.*) les éléments d'un ensemble totalement ordonné

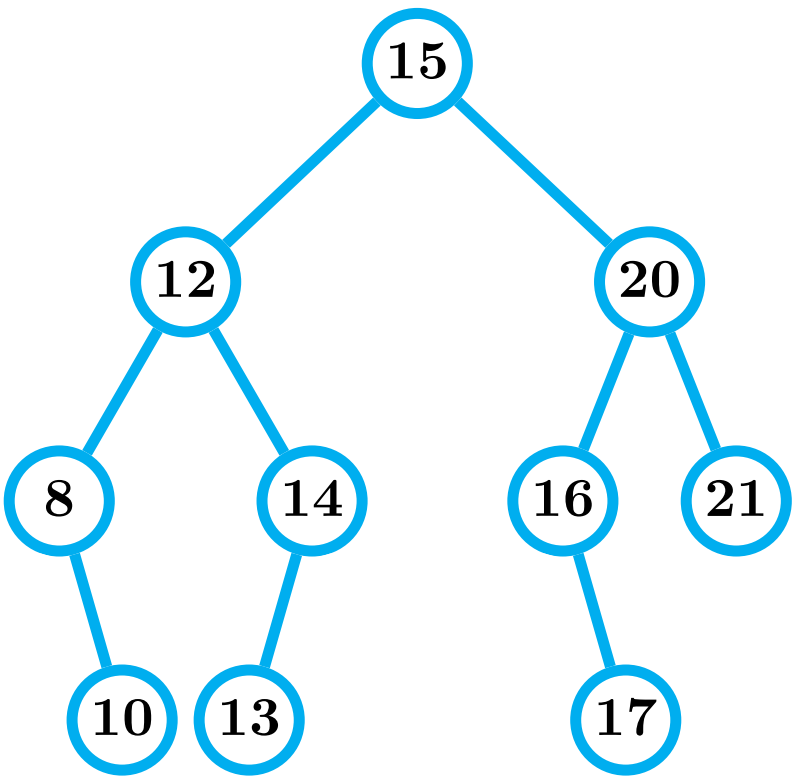
Un arbre binaire de recherche est un arbre binaire dans lequel pour tout nœud s,

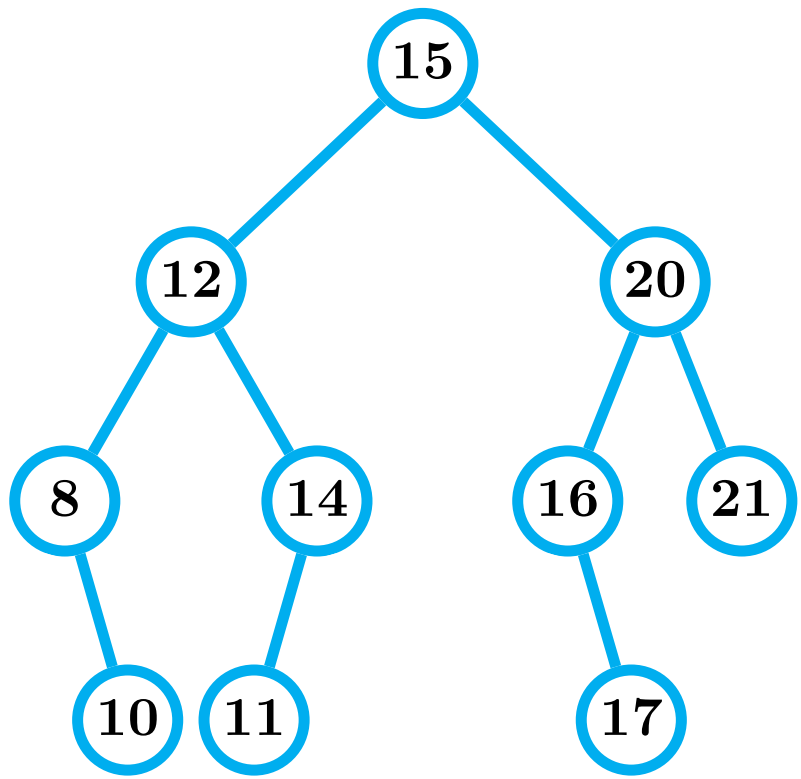
- tous les nœuds du sous-arbre gauche de s sont strictement inférieurs au contenu de s
- tous les nœuds du sous-arbre droit de s sont strictement supérieurs au contenu de s

1. Pas de doublet dans un arbre binaire de recherche
2. Rappel : Un tas est un arbre binaire tassé dans lequel le contenu de chaque noeud est $\geq$ à celui de ses fils
   - Dans un cas on regarde les fils (tas)
   - Dans l'autre (abn), tous les éléments des sous-arbres !

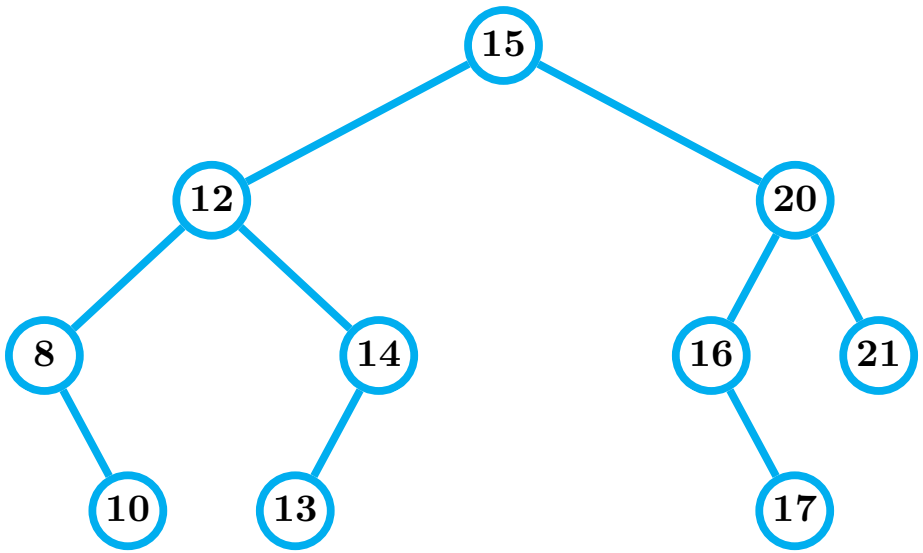
Des exemples d'arbres binaires de recherche

De ces deux arbres, lequel est un arbre binaire de recherche ?





Quelques propriétés des arbres binaires de recherche



Rappel : Parcours “infixe” : sous-arbre gauche puis sommet courant puis sous-arbre droit

Soit donc ici :

Dans un arbre binaire de recherche, le parcours infixe fournit les contenus des nœuds en ordre croissant

Un arbre binaire de recherche en Java

Un arbre binaire de recherche est un arbre binaire (rappel)

```
class Arbre {
    int val;
    Arbre gauche, droite;
    public String toString() {
        String s1 = "Vide";
        String s2 = "Vide";
        if (gauche ≠ null) s1 = gauche.toString();
        if (droite ≠ null) s2 = droite.toString();
        return "(" + s1 + ", " + val + ", " + s2 + ")";
    }
    int hauteur() {
        return 1 + Math.max((gauche == null) ? 0 : gauche.hauteur(),
                            (droite == null) ? 0 : droite.hauteur());
    }
}
```

Chercher un élément dans un arbre binaire de recherche

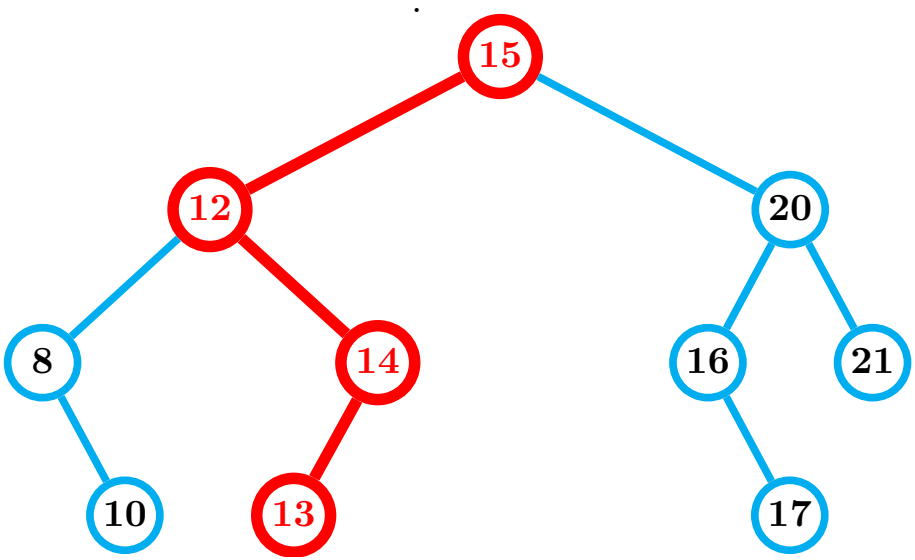
x est-il dans l'arbre t ?

- Si $t =$ `null` $\longrightarrow$ NON
- Si $x = t.val \longrightarrow$ OUI
- Si $x < t.val \longrightarrow$ Se reposer la question pour x et `t.gauche`
- Si $x > t.val \longrightarrow$ Se reposer la question pour x et `t.droite`

C'est une simple transposition de la recherche dichotomique

Complexité : $O(h)$ où h est la hauteur de l'arbre

Chercher 13

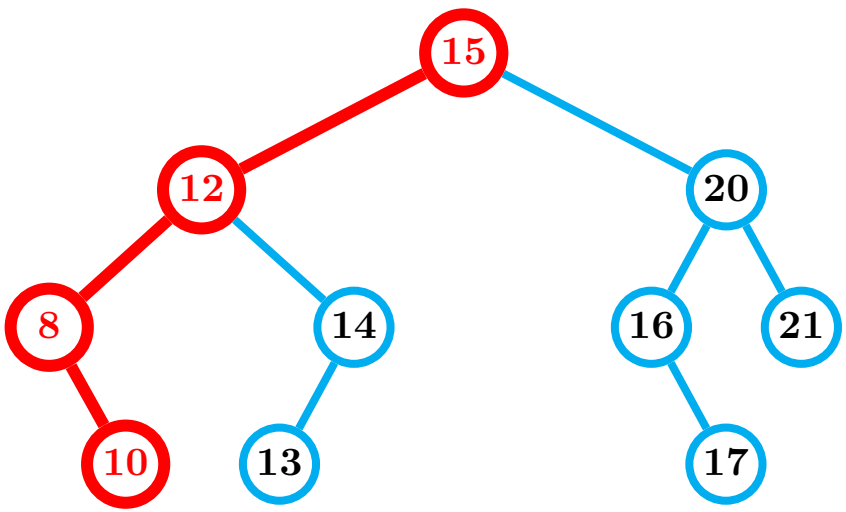


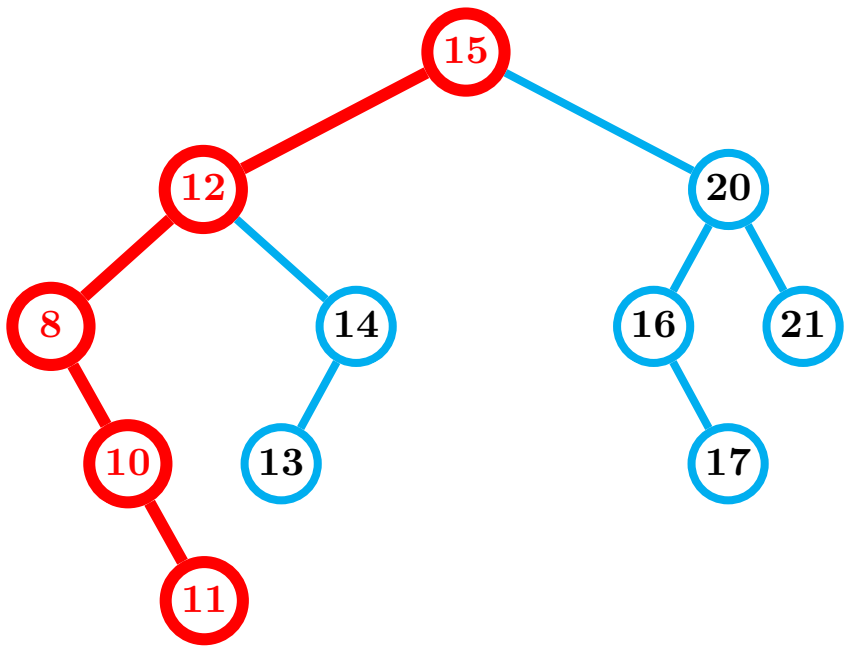
La meilleure façon de chercher ?

```
static Arbre
chercher(int x, Arbre a) {
  if (a == null || x == a.val)
    return a;
  if (x < a.val)
    return chercher(x, a.gauche);
  return chercher(x, a.droite);
}
Arbre
chercher(int x) {
  if (x == val) return this;
  if (x < val && gauche ≠ null)
    return gauche.chercher(x);
  if (x > val && droite ≠ null)
    return droite.chercher(x);
  return null;
}
```

```
static Arbre
chercherI(int x, Arbre a) {
  while(a ≠ null &&
        x ≠ a.val)
    if (x < a.val)
      a = a.gauche;
    else
      a = a.droite;
  return a;
}
```

Insérer 11





Quel est le coût d'une insertion ?

Insérer dans un arbre binaire de recherche

Quelle différence entre ces deux variantes ?

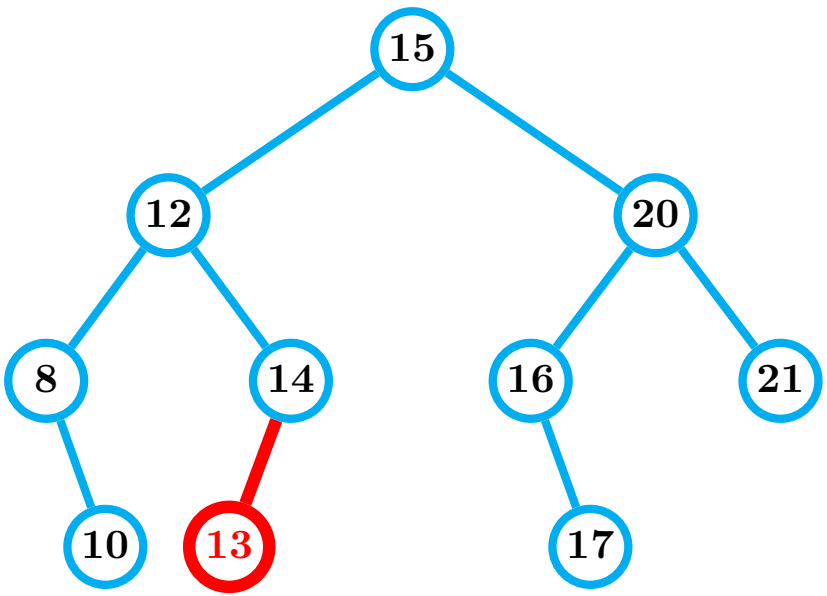
```
static Arbre inserer1(int x, Arbre a) {
  if (a == null) return new Arbre(null, x, null);
  if (x < a.val) a.gauche = inserer1(x, a.gauche);
  else if (x > a.val) a.droite = inserer1(x, a.droite);
  return a;
}
static Arbre inserer2(int x, Arbre a) {
  if (a == null) return new Arbre(null, x, null);
  if (x < a.val) {
    Arbre b = inserer2(x, a.gauche);
    return new Arbre(b, a.val, a.droite); }
  else if (x > a.val) {
    Arbre b = inserer2(x, a.droite);
    return new Arbre(a.gauche, a.val, b); }
  return a;
}
```

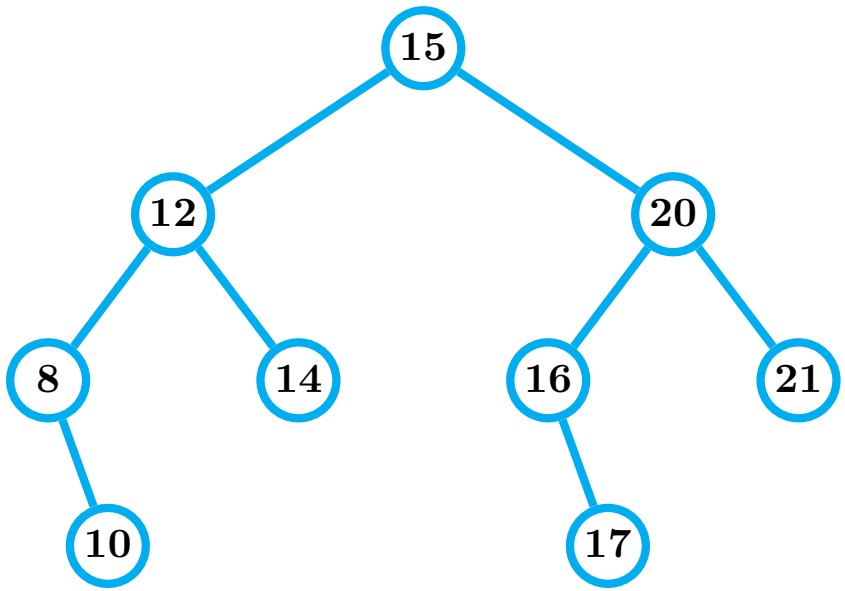
Supprimer un élément d'un arbre binaire de recherche

Nous devons distinguer 3 cas en fonction de l'arité du sommet s à enlever :

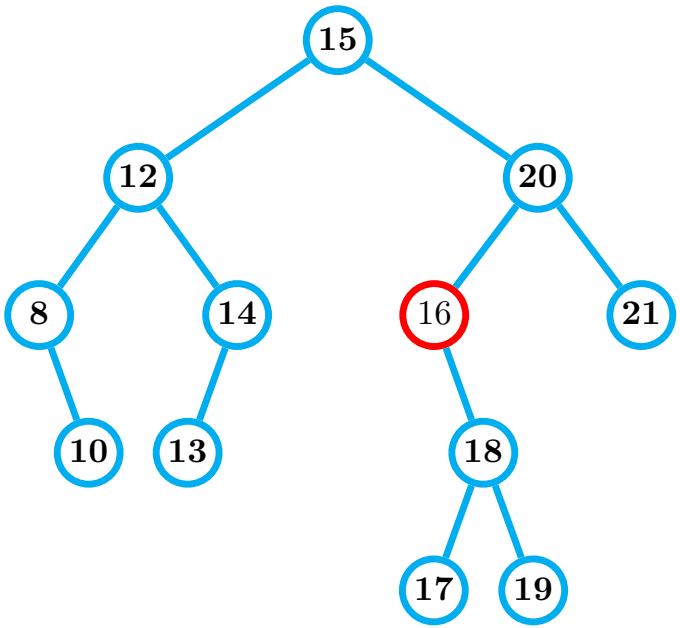
- Cas trivial : si le nœud s est une feuille, l'enlever
- Plus ennuyeux : si le nœud s a un fils unique, l'éliminer et remonter le fils
- Très ennuyeux : si le nœud s a deux fils

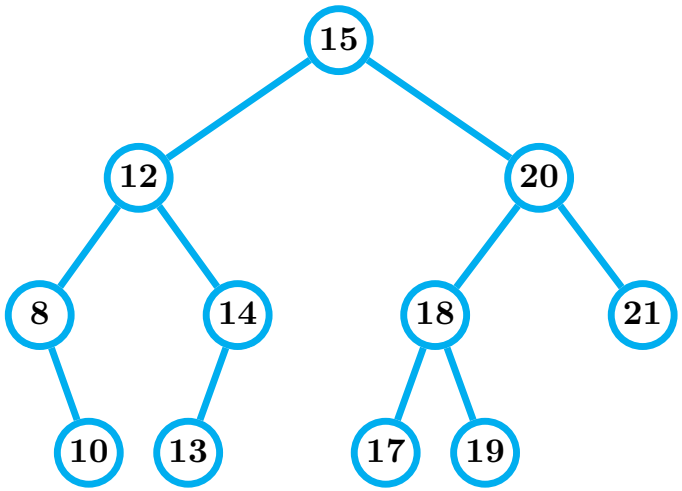
Supprimer une feuille d'un arbre binaire de recherche





Supprimer un nœud d'arité 1





Règle : le nœud s à supprimer n'a qu'un fils qui devient le fils du père de s

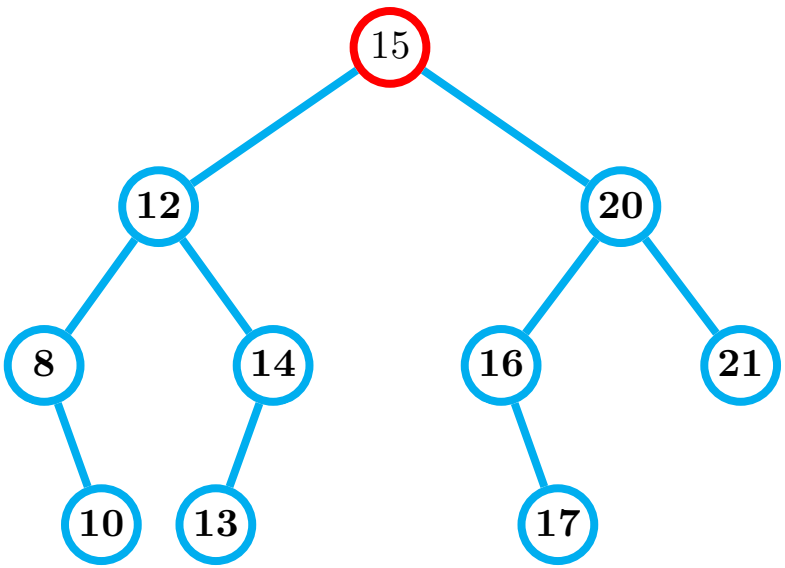
Supprimer un nœud s d'arité 2

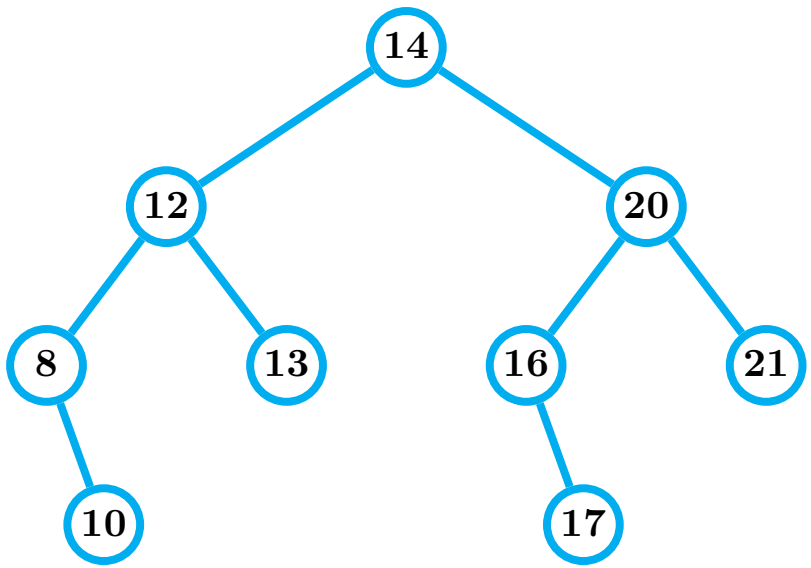
- Le nœud s lui-même n'est pas supprimé
- On échange le contenu de s avec g l'élément maximal du sous-arbre gauche
- Et on supprime g

Remarquons

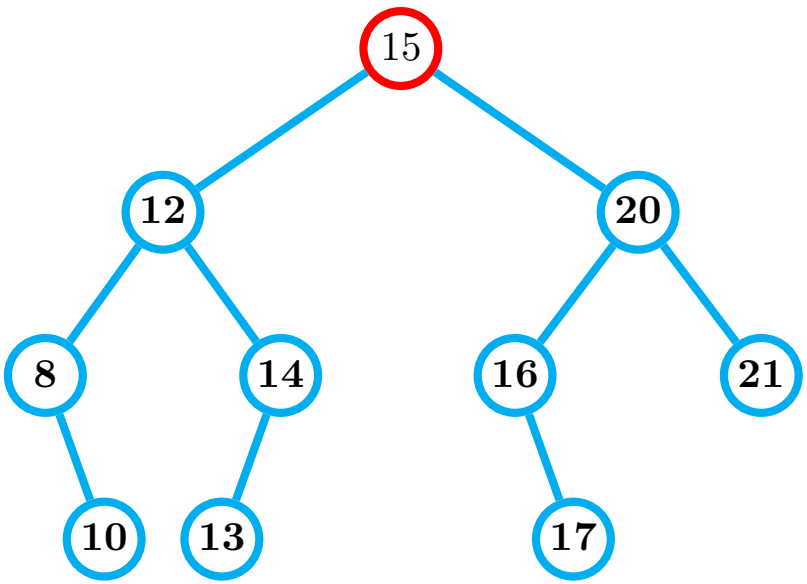
- g n'a pas de fils droit sinon il serait maximal et donc on retombe sur les cas connus
- Pour trouver g il suffit d'aller à droite tant qu'on peut !

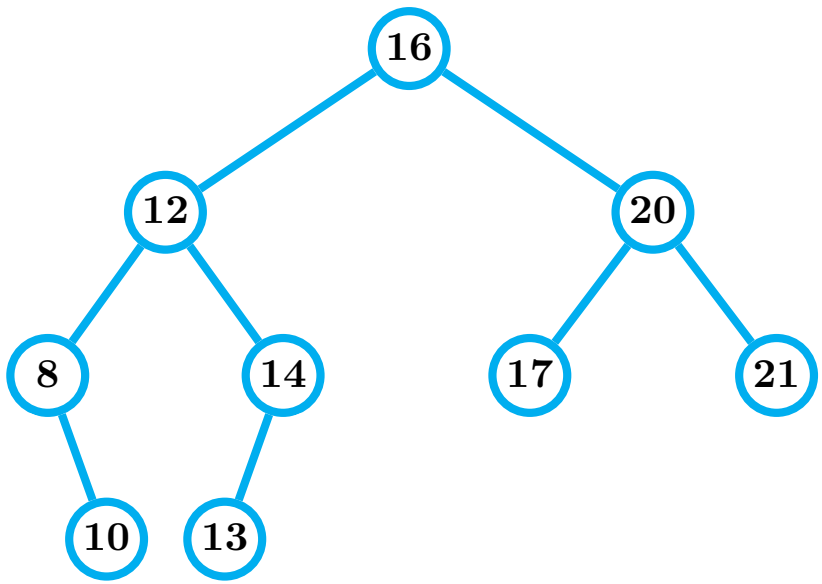
Supprimer un nœud s d'arité 2





Supprimer un nœud s d'arité 2 (variante)





Supprimer un nœud s d'arité 2

```
static Arbre rechercherEtSupprimer(int x, Arbre a) {
    if (a == null) return null;
    if (x == a.val) return supprimerRacine(a);
    if (x < a.val) a.gauche = rechercherEtSupprimer(x, a.gauche);
    else  a.droite = rechercherEtSupprimer(x, a.droite);
    return a;}
static Arbre supprimerRacine(Arbre a) {
    if (a.gauche == null) return a.droite;
    if (a.droite == null) return a.gauche;
    Arbre f = dernierDescendant(a.gauche);
    a.val = f.val;
    a.gauche = rechercherEtSupprimer(f.val, a.gauche);
    return a;}
static Arbre dernierDescendant(Arbre a) {
    if (a.droite == null) return a;
    return dernierDescendant(a.droite);}
```

Complexité (pire cas)

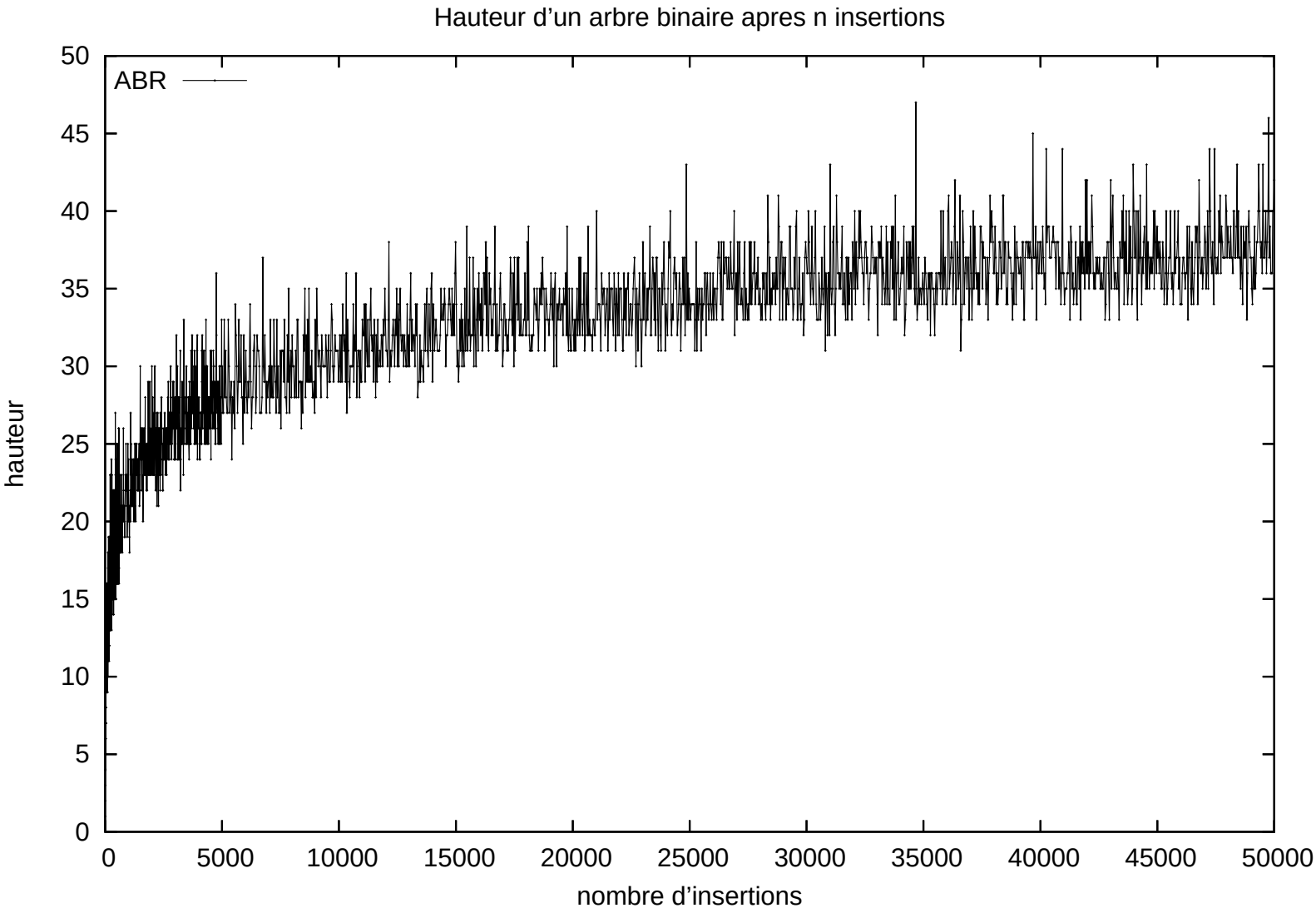
Soit n le nombre de nœuds et h la hauteur de l'arbre

- Pour l'insertion ?
- Pour la suppression ?

Comment évaluer h ?

- Quelle est la hauteur d'un arbre binaire tassé ?
- Quelle est la hauteur d'un arbre binaire réduit à une chaîne ?

Quelques expériences ⟶ pas si catastrophiques



Hauteur moyenne d'un arbre binaire de recherche

- Insertion de n valeurs distinctes (clefs) dans un arbre binaire de recherche initialement vide
- Hypothèse : les $n!$ permutations sont équiprobables
- Question : Quelle est la hauteur moyenne de l'arbre ?

Pourquoi se poser une telle question ? Pour faire une analyse en moyenne du comportement des arbres binaires de recherche

Les opérations élémentaires sur un arbre binaire de recherche coûtent $O(\log n)$ en moyenne

Attention : C'est en moyenne, on est jamais à l'abri d'un mauvais comportement

Hauteur moyenne d'un arbre binaire de recherche

Quelques rappels simples

- Insérons un élément x dans un arbre binaire de recherche vide
- Insérons les $n-1$ autres éléments
  - Ceux $> x$ vont à droite
  - Ceux $< x$ vont à gauche
- La hauteur de l'arbre $= 1 +$ la plus grande des deux hauteurs des deux sous-arbres

Hauteur moyenne d'un arbre binaire de recherche

- Soit H_n la hauteur d'un arbre de n clefs
- Soit $Y_n = 2^{H_n}$

La probabilité que le premier élément inséré soit le $i$ème est $\frac{1}{n}$ et donc, pour $n \geq 2$,

$$\mathbb{E}[Y_n] = \frac{1}{n} \sum_{i=1}^{n} 2 \times \mathbb{E}[\max(Y_{i-1}, Y_{n-i})]$$

Or, $\mathbb{E}[\max(Y_{i-1}, Y_{n-i})] \leq \mathbb{E}[Y_{i-1}] + \mathbb{E}[Y_{n-i}]$ et donc

$$\mathbb{E}[Y_n] = \frac{4}{n} \sum_{i=1}^{n-1} \mathbb{E}[Y_i]$$

Hauteur moyenne d'un arbre binaire de recherche

Montrons par induction que $\mathbb{E}[Y_n] \leq \frac{1}{4} C_{n+3}^3$

Rappel : $\mathbb{E}[Y_n] = \frac{4}{n} \sum_{i=1}^{n-1} \mathbb{E}[Y_i]$

- Trivial pour $n = 1$
- Or $\sum_{i=0}^{n-1} C_{i+3}^3 = C_{n+3}^4$ et donc

$$\begin{aligned} \mathbb{E}[Y_n] &\leq \frac{4}{n} \sum_{i=1}^{n-1} \frac{1}{4} C_{i+3}^3 \leq \frac{1}{n} \sum_{i=0}^{n-1} C_{i+3}^3 \\ &\leq \frac{1}{n} C_{n+3}^4 \leq \frac{1}{4} C_{n+3}^3 \end{aligned}$$

Hauteur moyenne d'un arbre binaire de recherche

Lemme de Jensen : Soit f convexe et A une variable aléatoire alors $\mathbb{E}[f(A)] \geq f(\mathbb{E}[A])$.

Dans notre cas, $\mathbb{E}[Y_n] = \mathbb{E}[2^{H_n}] \geq 2^{\mathbb{E}[H_n]}$ et donc

$$2^{\mathbb{E}[H_n]} \leq \frac{1}{4} C_{n+3}^3$$

Ce qui conduit directement à $\mathbb{E}[H_n] = O(\log n)$