3.24pt

Structures de Données

CM4 – Les arbres

Olivier FESTOR

Telecom Nancy

26 Avril 2022

Plan

3.24pt

1. Présentation des arbres
   - Vocabulaire
   - Les arbres binaires
   - Les ABR
2. Spécification algébrique des arbres binaires
   - Opérations
   - Axiomes
3. Spécification algébrique des ABR
   - Opérations
   - Axiomes
4. Limite des ABR
   - Illustration
   - Solution

3.24pt

5. Les arbres AVL
   - Présentation
   - Les poids
   - Insertion
   - Mise en pratique
6. Les tas
   - Présentation
   - Ajout
   - Suppression
   - Heapsort
7. Implantations Java
   - Arbre binaire complet
   - Arbre binaire quelconque
8. Complexités

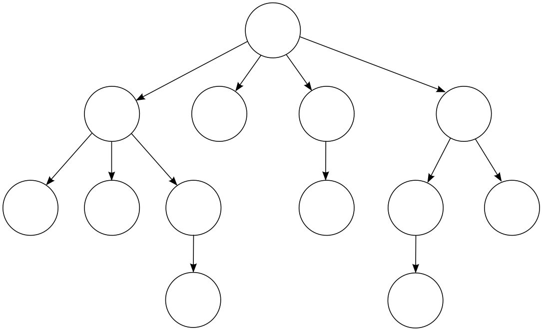
Plan

Contributeurs

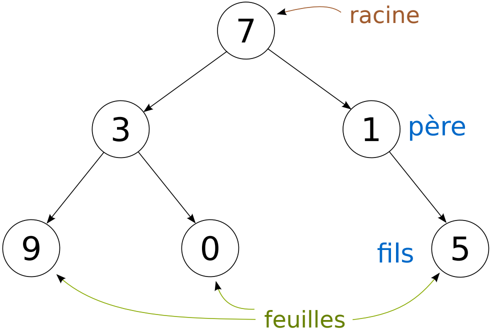
Depuis la création du cours

- Martin Quinson (créateur)
- Adrien Kranenbuhl
- Sébastien da Silva
- Gérald Oster
- Olivier Festor

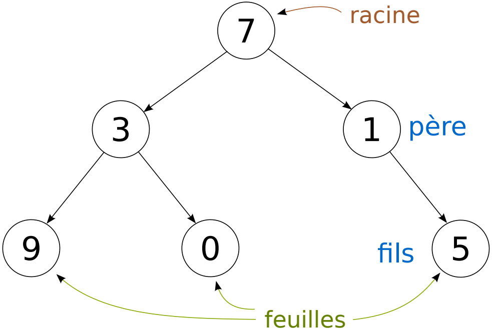
Arbres quelconques



Arbres quelconques

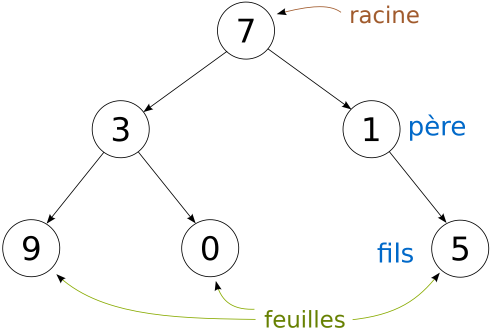


Arbres quelconques



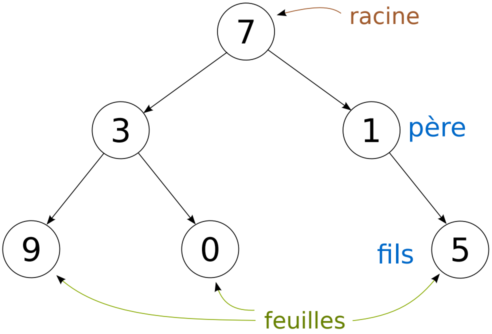
- Un arbre est constitué de **nœuds** et de **branches**

Arbres quelconques



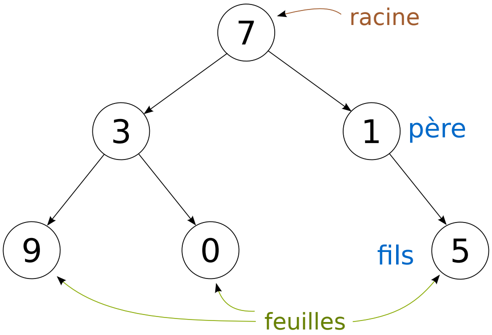
- Un arbre est constitué de **nœuds** et de **branches**
- Le nœud 7 est la **racine** de l'arbre.

Arbres quelconques



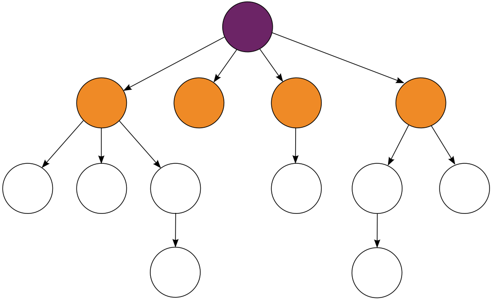
- Un arbre est constitué de **nœuds** et de **branches**
- Le nœud 7 est la **racine** de l'arbre.
- Le nœud 1 est le **père** du nœud **5** qui est son **fils**.

Arbres quelconques



- Un arbre est constitué de **nœuds** et de **branches**
- Le nœud 7 est la **racine** de l'arbre.
- Le nœud 1 est le **père** du nœud **5** qui est son **fils**.
- Les nœuds $9, 0$ et 5 sont les **feuilles** de l'arbre.

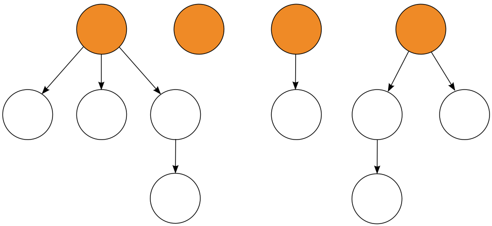
Récursivité



1 arbre : 1 racine avec 4 fils

Les arbres sont une structure **intrinsèquement récursive**.
Les **fils** d'un nœud constituent un ensemble de **sous-arbres** disjoints.

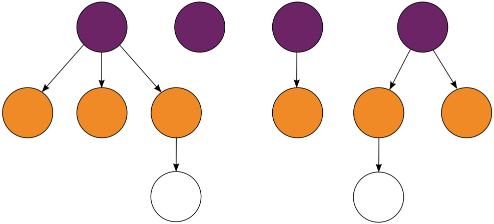
Récursivité



4 fils ⟶ 4 sous-arbres

Les arbres sont une structure **intrinsèquement récursive**.
Les **fils** d'un nœud constituent un ensemble de **sous-arbres** disjoints.

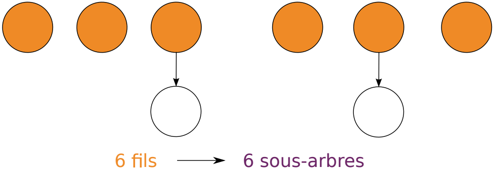
Récursivité



4 arbres : 4 racines avec 6 fils

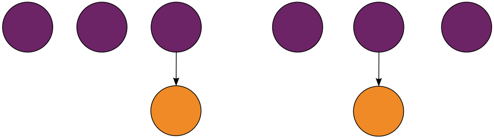
Les arbres sont une structure **intrinsèquement récursive**.
Les **fils** d'un nœud constituent un ensemble de **sous-arbres** disjoints.

Récursivité



Les arbres sont une structure **intrinsèquement récursive**.
Les **fils** d'un nœud constituent un ensemble de **sous-arbres** disjoints.

Récursivité



6 arbres : 6 racines avec 2 fils

Les arbres sont une structure **intrinsèquement récursive**.
Les **fils** d'un nœud constituent un ensemble de **sous-arbres** disjoints.

Récursivité

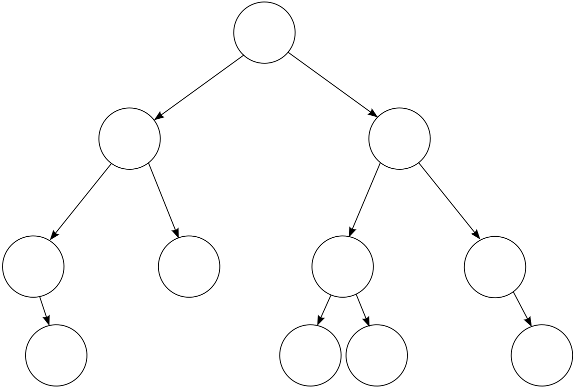
Les arbres sont une structure **intrinsèquement récursive**.
Les **fils** d'un nœud constituent un ensemble de **sous-arbres** disjoints.

Récursivité

2 arbres : 2 racines avec 0 fils

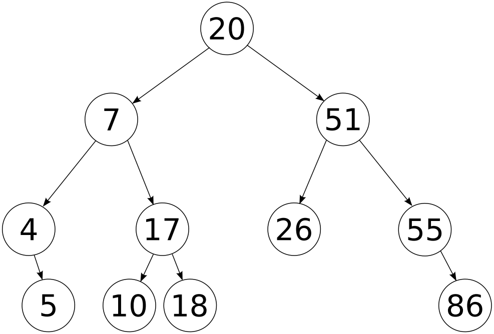
Les arbres sont une structure **intrinsèquement récursive**.
Les **fils** d'un nœud constituent un ensemble de **sous-arbres** disjoints.

Les arbres binaires



Tous les nœuds d'un arbre binaire ont **au plus deux fils** :
un fils **gauche** et un fils **droit**

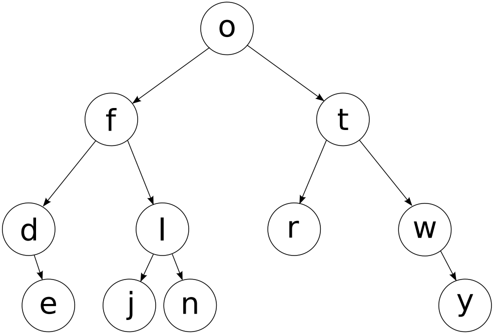
Les arbres binaires de recherche



Deux contraintes vérifiées en **chaque nœud** :

- Toutes les valeurs des nœuds du **sous-arbre gauche** sont **inférieures** à la valeur du **père**.
- Toutes les valeurs des nœuds du **sous-arbre droit** sont **supérieures** à la valeur du **père**.

Les arbres binaires de recherche



Deux contraintes vérifiées en **chaque nœud** :

- Toutes les valeurs des nœuds du **sous-arbre gauche** sont **inférieures** à la valeur du **père**.
- Toutes les valeurs des nœuds du **sous-arbre droit** sont **supérieures** à la valeur du **père**.

Plan

Type BinaryTree<T>

$$
\begin{array}{rcll}
\textit{empty} & : & & \rightarrow \textit{BinaryTree}{<}T{>} \\
\textit{makeRoot} & : & BT{<}T{>} \times T \times BT{<}T{>} & \rightarrow \textit{BinaryTree}{<}T{>} \\
\textit{isEmpty} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{Boolean} \\
\textit{hasLeft} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{Boolean} \\
\textit{hasRight} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{Boolean} \\
\textit{has} & : & \textit{BinaryTree}{<}T{>} \times T & \rightarrow \textit{Boolean} \\
\textit{value} & : & \textit{BinaryTree}{<}T{>} & \rightarrow T \\
\textit{left} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{BinaryTree}{<}T{>} \\
\textit{right} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{BinaryTree}{<}T{>} \\
\textit{height} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{Integer} \\
\textit{count} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{Integer} \\
\textit{prefix} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{MyList}{<}T{>} \\
\textit{infix} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{MyList}{<}T{>} \\
\textit{postfix} & : & \textit{BinaryTree}{<}T{>} & \rightarrow \textit{MyList}{<}T{>}
\end{array}
$$

Type BinaryTree<T>

$$
\begin{aligned}
empty &: && \to BinaryTree{<}T{>}\\
makeRoot &: BT{<}T{>} \times T \times BT{<}T{>} && \to BinaryTree{<}T{>}\\
isEmpty &: BinaryTree{<}T{>} && \to Boolean\\
hasLeft &: BinaryTree{<}T{>} && \to Boolean\\
hasRight &: BinaryTree{<}T{>} && \to Boolean\\
has &: BinaryTree{<}T{>} \times T && \to Boolean\\
value &: BinaryTree{<}T{>} && \to T\\
left &: BinaryTree{<}T{>} && \to BinaryTree{<}T{>}\\
right &: BinaryTree{<}T{>} && \to BinaryTree{<}T{>}\\
height &: BinaryTree{<}T{>} && \to Integer\\
count &: BinaryTree{<}T{>} && \to Integer\\
prefix &: BinaryTree{<}T{>} && \to MyList{<}T{>}\\
infix &: BinaryTree{<}T{>} && \to MyList{<}T{>}\\
postfix &: BinaryTree{<}T{>} && \to MyList{<}T{>}
\end{aligned}
$$

Type BinaryTree<T>

$$
\begin{array}{rll}
empty: & & \rightarrow BinaryTree{<}T{>} \\
makeRoot: & BT{<}T{>} \times T \times BT{<}T{>} & \rightarrow BinaryTree{<}T{>} \\
isEmpty: & BinaryTree{<}T{>} & \rightarrow Boolean \\
hasLeft: & BinaryTree{<}T{>} & \rightarrow Boolean \\
hasRight: & BinaryTree{<}T{>} & \rightarrow Boolean \\
has: & BinaryTree{<}T{>} \times T & \rightarrow Boolean \\
value: & BinaryTree{<}T{>} & \rightarrow T \\
left: & BinaryTree{<}T{>} & \rightarrow BinaryTree{<}T{>} \\
right: & BinaryTree{<}T{>} & \rightarrow BinaryTree{<}T{>} \\
height: & BinaryTree{<}T{>} & \rightarrow Integer \\
count: & BinaryTree{<}T{>} & \rightarrow Integer \\
prefix: & BinaryTree{<}T{>} & \rightarrow MyList{<}T{>} \\
infix: & BinaryTree{<}T{>} & \rightarrow MyList{<}T{>} \\
postfix: & BinaryTree{<}T{>} & \rightarrow MyList{<}T{>}
\end{array}
$$

Les préconditions

Les préconditions

$value(b)$ défini ssi

Les préconditions

$$value(b) \quad \text{défini ssi} \quad non\ isEmpty(b)$$

Axiomes avec *isEmpty*

Axiomes avec *isEmpty*

$$isEmpty(empty()) \quad =$$

Axiomes avec *isEmpty*

$$isEmpty(empty()) = vrai$$

Axiomes avec *isEmpty*

$$\begin{aligned} isEmpty(empty()) &= vrai \\ isEmpty(makeRoot(l, v, r)) &= \end{aligned}$$

Axiomes avec *isEmpty*

$$\begin{array}{rcl} isEmpty(empty()) & = & vrai \\ isEmpty(makeRoot(l, v, r)) & = & faux \end{array}$$

Axiomes avec *hasLeft* et *hasRight*

Axiomes avec *hasLeft* et *hasRight*

$$hasLeft(empty()) \quad =$$

Axiomes avec *hasLeft* et *hasRight*

$$hasLeft(empty()) = faux$$

Axiomes avec *hasLeft* et *hasRight*

$$
\begin{aligned}
hasLeft(empty()) &= faux \\
hasRight(empty()) &=
\end{aligned}
$$

Axiomes avec *hasLeft* et *hasRight*

$$\begin{aligned} hasLeft(empty()) &= faux \\ hasRight(empty()) &= faux \end{aligned}$$

Axiomes avec *hasLeft* et *hasRight*

$$
\begin{aligned}
hasLeft(empty()) &= faux \\
hasRight(empty()) &= faux \\
hasLeft(makeRoot(l, v, r)) &=
\end{aligned}
$$

Axiomes avec *hasLeft* et *hasRight*

$$
\begin{aligned}
hasLeft(empty()) &= faux \\
hasRight(empty()) &= faux \\
hasLeft(makeRoot(l, v, r)) &= non\ isEmpty(l)
\end{aligned}
$$

Axiomes avec *hasLeft* et *hasRight*

$$
\begin{array}{rcl}
hasLeft(empty()) & = & faux \\
hasRight(empty()) & = & faux \\
hasLeft(makeRoot(l, v, r)) & = & non\ isEmpty(l) \\
hasRight(makeRoot(l, v, r)) & = &
\end{array}
$$

Axiomes avec *hasLeft* et *hasRight*

$$\begin{aligned}
hasLeft(empty()) &= faux\\
hasRight(empty()) &= faux\\
hasLeft(makeRoot(l, v, r)) &= non\ isEmpty(l)\\
hasRight(makeRoot(l, v, r)) &= non\ isEmpty(r)
\end{aligned}$$

Axiomes avec *has*

Axiomes avec *has*

$$has(empty(), v) \quad =$$

Axiomes avec *has*

$$has(empty(), v) \quad = \quad faux$$

Axiomes avec *has*

$$
\begin{aligned}
has(empty(), v) &= faux \\
has(makeRoot(l, v_1, r), v_2) &=
\end{aligned}
$$

Axiomes avec *has*

$$
\begin{array}{rcl}
has(empty(), v) & = & faux \\
has(makeRoot(l, v_1, r), v_2) & = & \begin{cases} vrai & \text{si } v_1 = v_2 \\ has(l, v_2) || has(r, v_2) & \text{sinon} \end{cases}
\end{array}
$$

Axiomes avec *value*

Axiomes avec *value*

$$value(empty()) \quad =$$

Axiomes avec *value*

$$value(empty()) \quad = \quad \text{Violation de précondition !}$$

Axiomes avec *value*

$$
\begin{aligned}
value(empty()) &= \text{Violation de précondition !} \\
value(makeRoot(l, v, r)) &=
\end{aligned}
$$

Axiomes avec *value*

$$
\begin{aligned}
value(empty()) &= \text{Violation de précondition !}\\
value(makeRoot(l, v, r)) &= v
\end{aligned}
$$

Axiomes avec *height*

Axiomes avec *height*

$$height(empty()) \quad =$$

Axiomes avec *height*

$$height(empty()) \quad = \quad 0$$

Axiomes avec *height*

$$
\begin{aligned}
height(empty()) &= 0 \\
height(makeRoot(l, v, r)) &=
\end{aligned}
$$

Axiomes avec *height*

$$
\begin{array}{rcl}
height(empty()) & = & 0 \\
height(makeRoot(l, v, r)) & = & 1 + max(height(l), height(r))
\end{array}
$$

Axiomes avec *count*

Axiomes avec *count*

$$count(empty()) \quad =$$

Axiomes avec *count*

$$count(empty()) \quad = \quad 0$$

Axiomes avec *count*

$$
\begin{aligned}
count(empty()) &= 0 \\
count(makeRoot(l, v, r)) &=
\end{aligned}
$$

Axiomes avec *count*

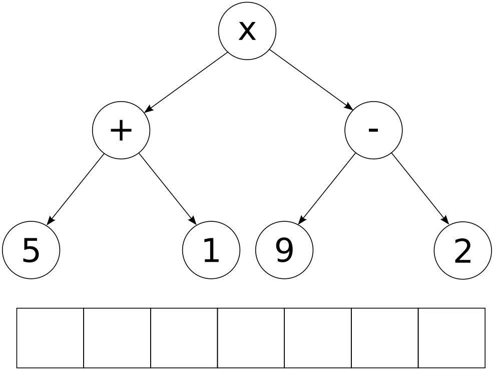
$$
\begin{aligned}
count(empty()) &= 0 \\
count(makeRoot(l, v, r)) &= 1 + count(l) + count(r)
\end{aligned}
$$

Axiomes avec *prefix*

$$prefix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$

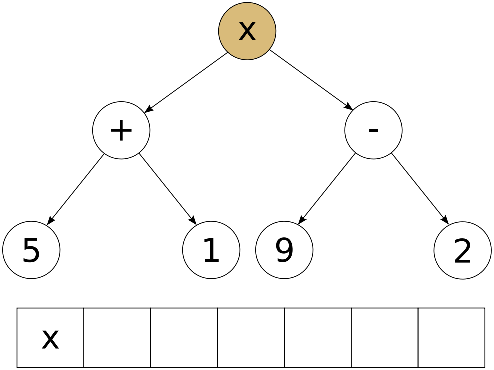
Axiomes avec *prefix*

$$prefix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



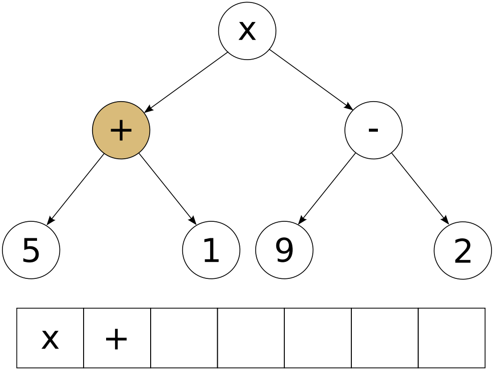
Axiomes avec *prefix*

$$prefix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



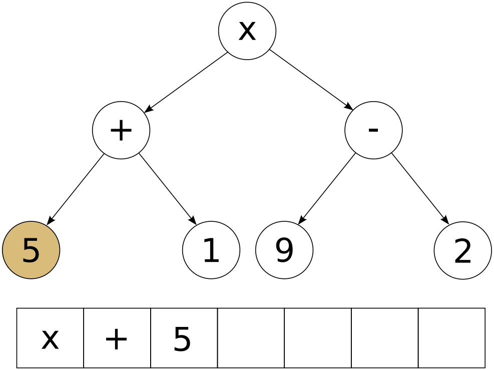
Axiomes avec *prefix*

$prefix : BinaryTree<T> \rightarrow MyList<T>$



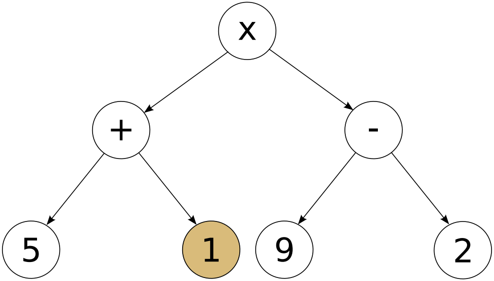
Axiomes avec *prefix*

$$prefix : BinaryTree<T> \to MyList<T>$$



Axiomes avec *prefix*

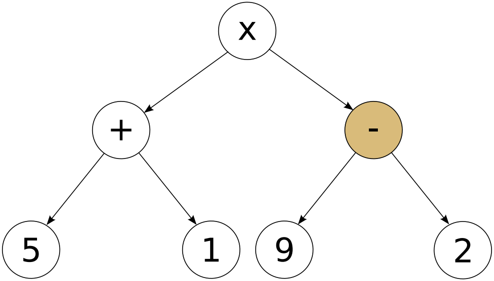
$$prefix : BinaryTree\textless T\textgreater \rightarrow MyList\textless T\textgreater$$



x	+	5	1			

Axiomes avec *prefix*

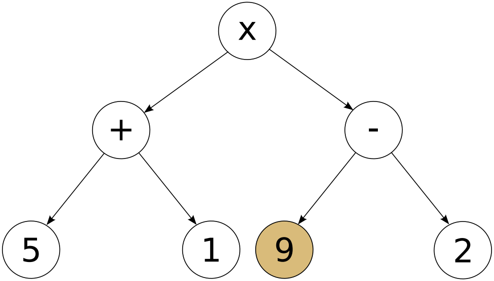
$prefix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



x	+	5	1	-		

Axiomes avec *prefix*

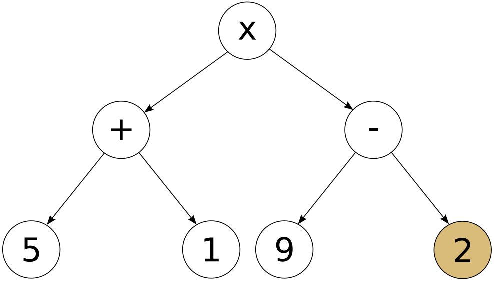
$$prefix : BinaryTree<T> \rightarrow MyList<T>$$



x	+	5	1	-	9	

Axiomes avec *prefix*

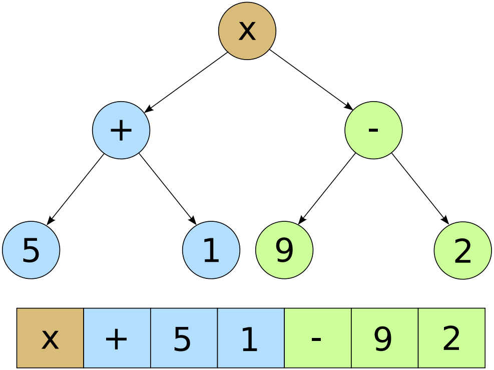
$$prefix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



x	+	5	1	-	9	2

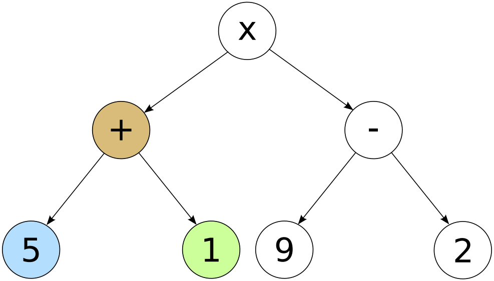
Axiomes avec *prefix*

$$prefix : BinaryTree\textless T\textgreater \rightarrow MyList\textless T\textgreater$$



Axiomes avec *prefix*

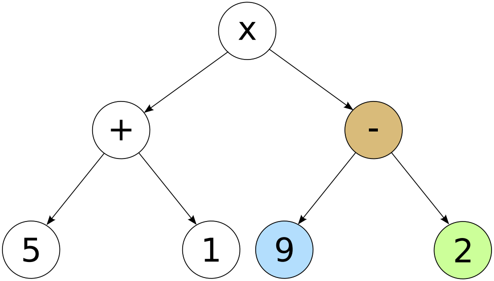
$$prefix : BinaryTree<T> \rightarrow MyList<T>$$



x	+	5	1	-	9	2

Axiomes avec *prefix*

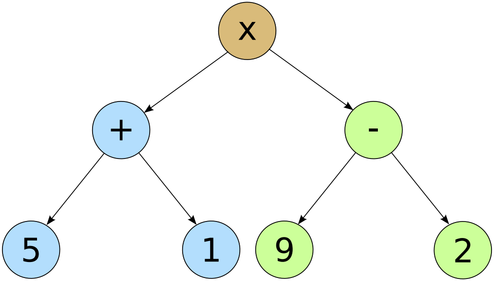
$$prefix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



x	+	5	1	-	9	2

Axiomes avec *prefix*

$$prefix : BinaryTree<T> \rightarrow MyList<T>$$

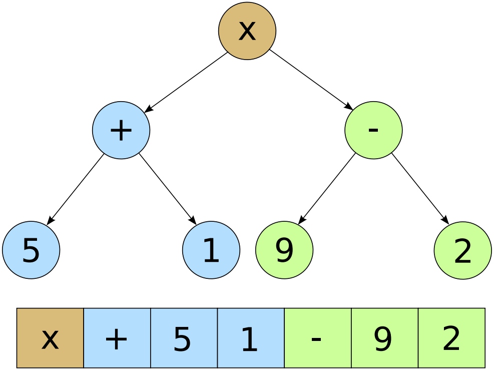


x	+	5	1	-	9	2

$$prefix(empty()) \quad =$$

Axiomes avec *prefix*

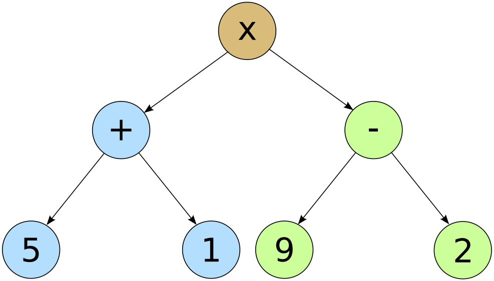
$$prefix : BinaryTree<T> \rightarrow MyList<T>$$



$$prefix(empty()) = empty()$$

Axiomes avec *prefix*

$$prefix : BinaryTree{<}T{>} \to MyList{<}T{>}$$

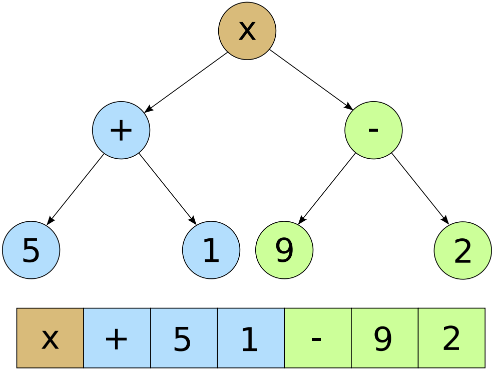


x	+	5	1	-	9	2

$$\begin{aligned} prefix(empty()) &= empty() \\ prefix(makeRoot(l, v, r)) &= \end{aligned}$$

Axiomes avec *prefix*

$$prefix : BinaryTree\langle T \rangle \rightarrow MyList\langle T \rangle$$



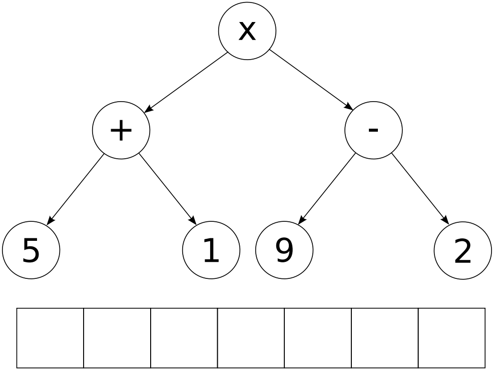
$$\begin{array}{rcl} prefix(empty()) & = & empty() \\ prefix(makeRoot(l, v, r)) & = & addFirst(add(prefix(l), prefix(r)), v) \end{array}$$

Axiomes avec *infix*

$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$

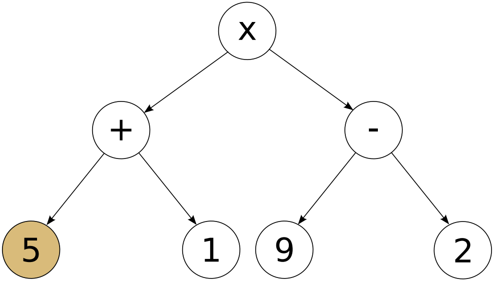
Axiomes avec *infix*

$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



Axiomes avec *infix*

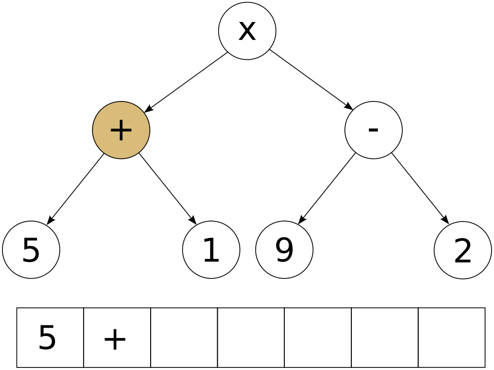
$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



5						

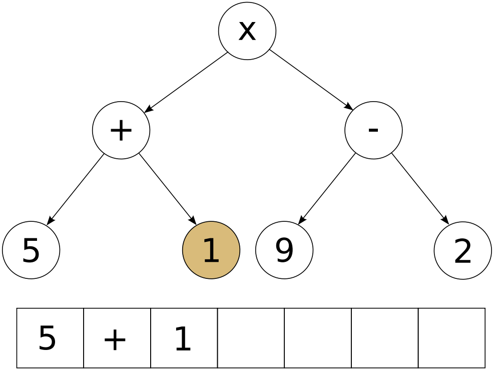
Axiomes avec *infix*

$$\mathit{infix} : \mathit{BinaryTree}{<}T{>} \rightarrow \mathit{MyList}{<}T{>}$$



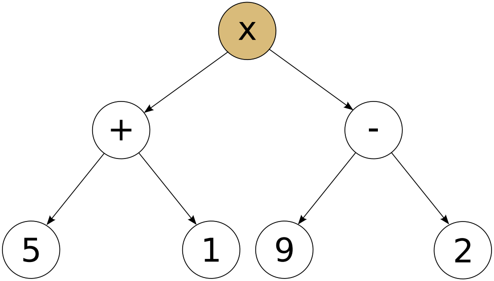
Axiomes avec *infix*

$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



Axiomes avec *infix*

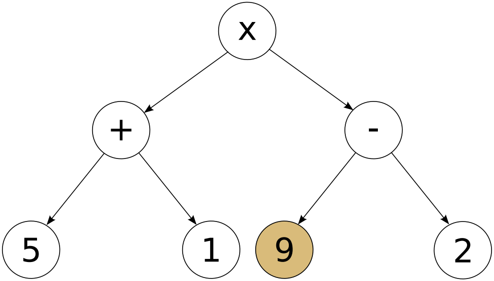
$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



5	+	1	x			

Axiomes avec *infix*

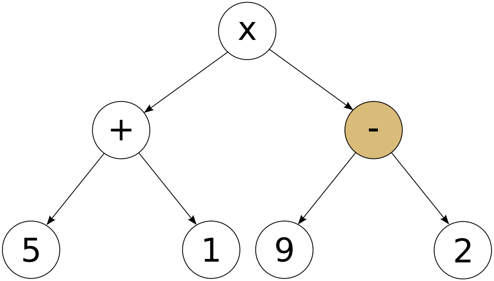
$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



5	+	1	x	9		

Axiomes avec *infix*

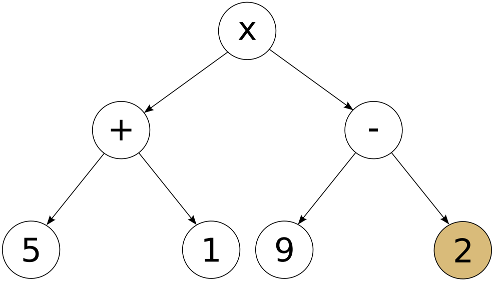
$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



5	+	1	x	9	-	

Axiomes avec *infix*

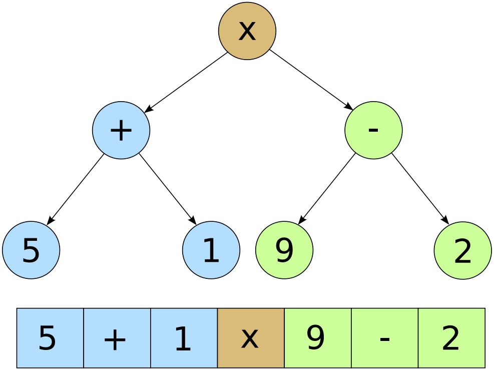
$$\textit{infix} : \textit{BinaryTree}{<}T{>} \rightarrow \textit{MyList}{<}T{>}$$



5	+	1	x	9	-	2

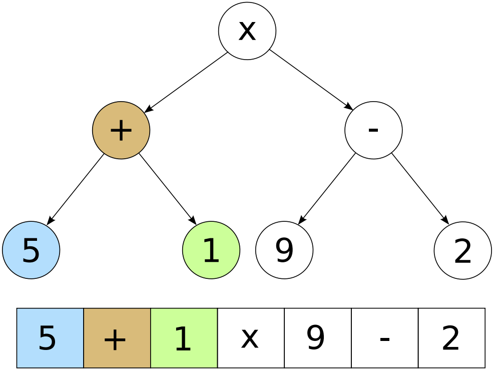
Axiomes avec *infix*

$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



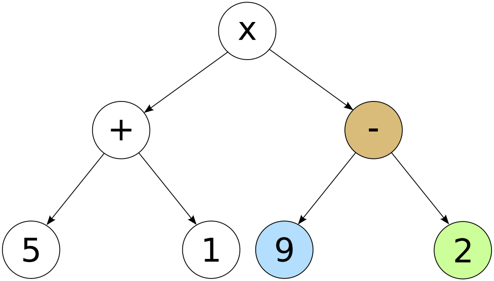
Axiomes avec *infix*

$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



Axiomes avec *infix*

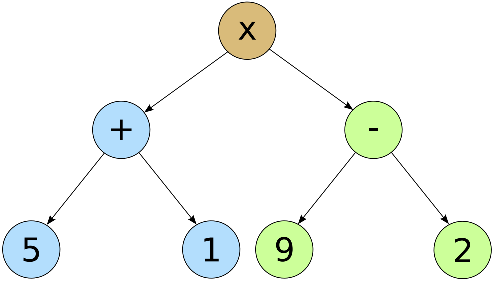
$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



5	+	1	x	9	-	2

Axiomes avec *infix*

$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$

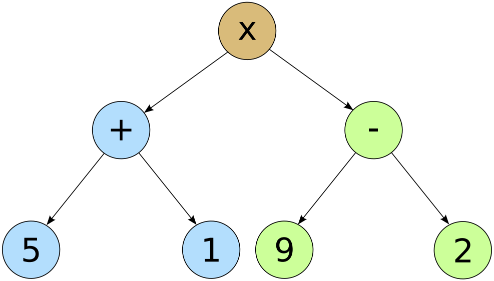


5	+	1	x	9	-	2

$$infix(empty()) \quad =$$

Axiomes avec *infix*

$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$

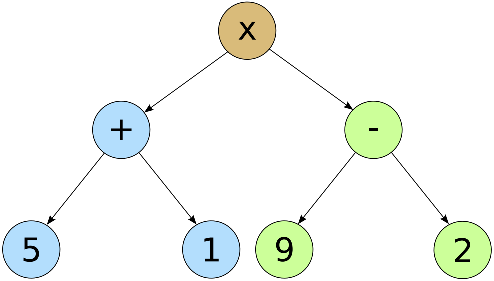


5	+	1	x	9	-	2

$$infix(empty()) \quad = \quad empty()$$

Axiomes avec *infix*

$$infix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$

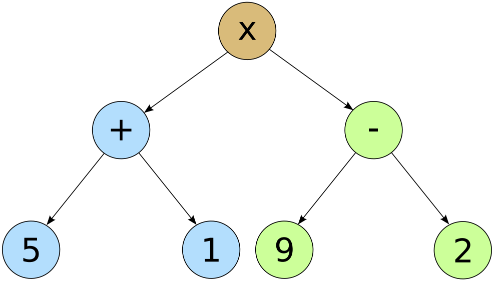


5	+	1	x	9	-	2

$$\begin{aligned} infix(empty()) &= empty() \\ infix(makeRoot(l, v, r)) &= \end{aligned}$$

Axiomes avec *infix*

$$infix : BinaryTree\!<\!T\!> \rightarrow MyList\!<\!T\!>$$



5	+	1	x	9	-	2

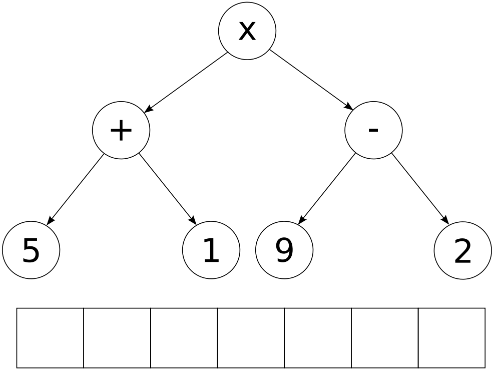
$$\begin{aligned} infix(empty()) &= empty() \\ infix(makeRoot(l, v, r)) &= add(addLast(infix(l), v), infix(r)) \end{aligned}$$

Axiomes avec *postfix*

$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$

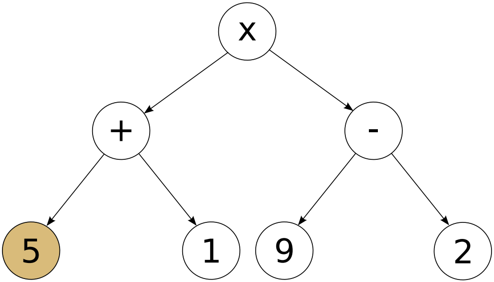
Axiomes avec *postfix*

$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



Axiomes avec *postfix*

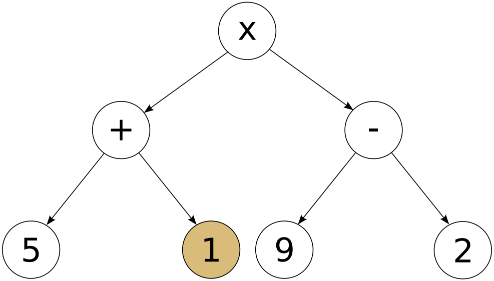
$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



5						

Axiomes avec *postfix*

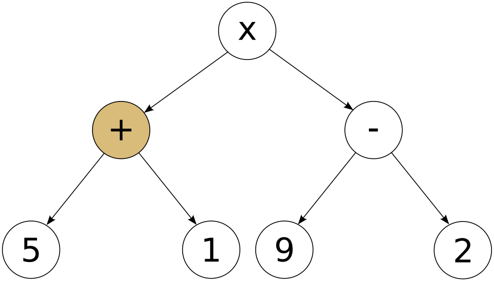
$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



5	1					

Axiomes avec *postfix*

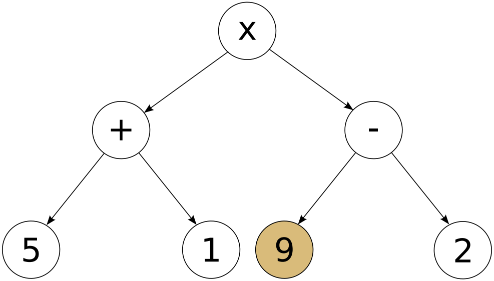
$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



5	1	+				

Axiomes avec *postfix*

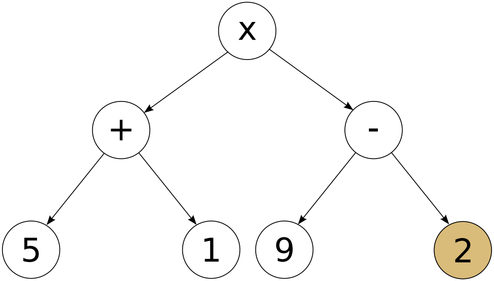
$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



5	1	+	9			

Axiomes avec *postfix*

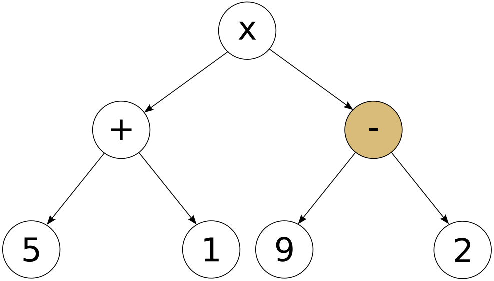
$postfix : BinaryTree\langle T\rangle \rightarrow MyList\langle T\rangle$



5	1	+	9	2		

Axiomes avec *postfix*

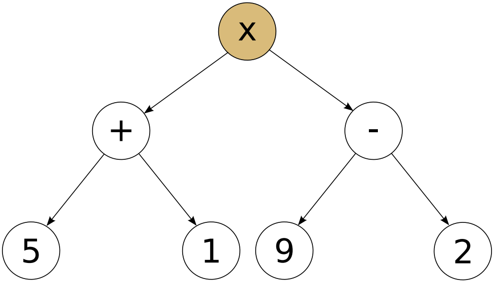
$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



5	1	+	9	2	-	

Axiomes avec *postfix*

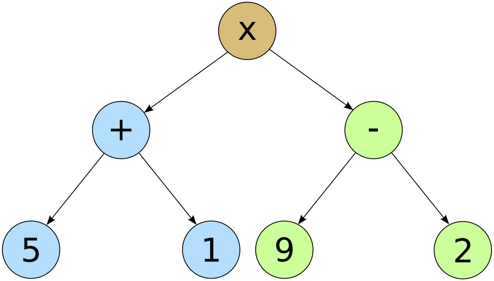
$$postfix : BinaryTree<T> \to MyList<T>$$



5	1	+	9	2	-	x

Axiomes avec *postfix*

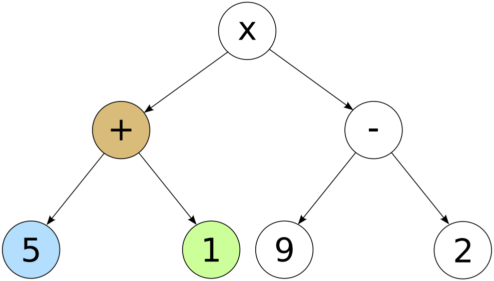
$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$



5	1	+	9	2	-	x

Axiomes avec *postfix*

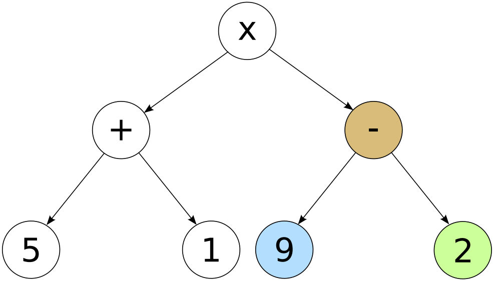
$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



5	1	+	9	2	-	x

Axiomes avec *postfix*

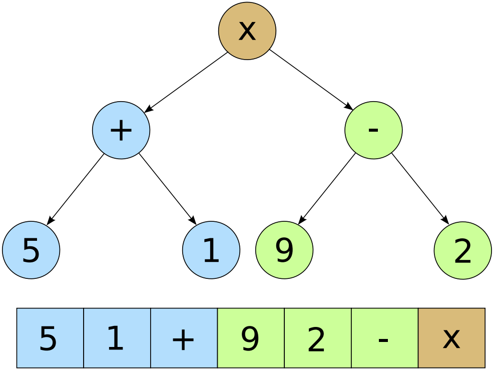
$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



5	1	+	9	2	-	x

Axiomes avec *postfix*

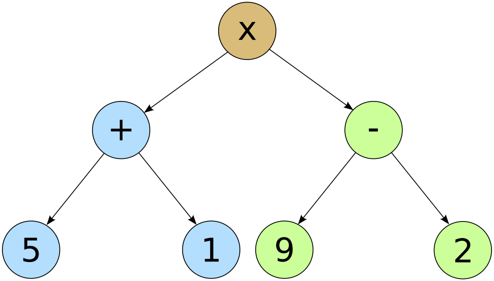
$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



$$postfix(empty()) \quad =$$

Axiomes avec *postfix*

$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$

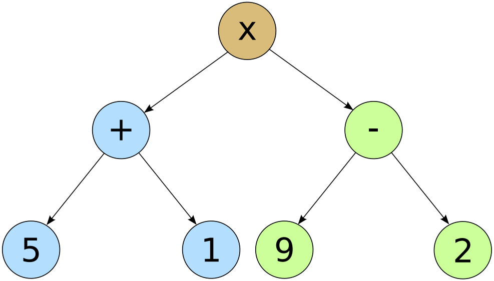


5	1	+	9	2	-	x

$$postfix(empty()) \quad = \quad empty()$$

Axiomes avec *postfix*

$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$

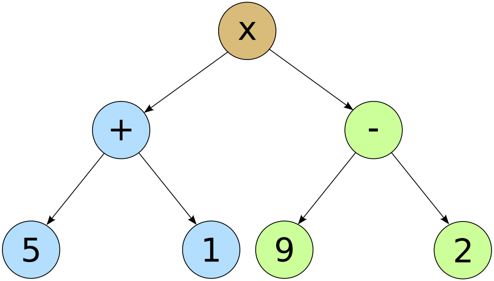


5	1	+	9	2	-	x

$$\begin{aligned} postfix(empty()) &= empty() \\ postfix(makeRoot(l, v, r)) &= \end{aligned}$$

Axiomes avec *postfix*

$$postfix : BinaryTree{<}T{>} \rightarrow MyList{<}T{>}$$



5	1	+	9	2	-	x

$$\begin{aligned} postfix(empty()) &= empty() \\ postfix(makeRoot(l, v, r)) &= addLast(add(postfix(l), postfix(r)), v) \end{aligned}$$

Plan

3.24pt

3.24pt

TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

Type BinarySearchTree<T> sous-type de BT<T>

$$
\begin{array}{rll}
least: & BST{<}T{>} & \rightarrow T \\
greatest: & BST{<}T{>} & \rightarrow T \\
predecessor: & BST{<}T{>} & \rightarrow T \\
successor: & BST{<}T{>} & \rightarrow T \\
add: & BST{<}T{>} \times T & \rightarrow BST{<}T{>} \\
deleteGreatest: & BST{<}T{>} & \rightarrow BST{<}T{>} \\
deleteLeast: & BST{<}T{>} & \rightarrow BST{<}T{>} \\
delete: & BST{<}T{>} \times T & \rightarrow BST{<}T{>}
\end{array}
$$

Type BinarySearchTree<T> sous-type de BT<T>

$$
\begin{aligned}
\mathit{least} :&\quad BST{<}T{>} &&\rightarrow T\\
\mathit{greatest} :&\quad BST{<}T{>} &&\rightarrow T\\
\mathit{predecessor} :&\quad BST{<}T{>} &&\rightarrow T\\
\mathit{successor} :&\quad BST{<}T{>} &&\rightarrow T\\
\mathit{add} :&\quad BST{<}T{>} \times T &&\rightarrow BST{<}T{>}\\
\mathit{deleteGreatest} :&\quad BST{<}T{>} &&\rightarrow BST{<}T{>}\\
\mathit{deleteLeast} :&\quad BST{<}T{>} &&\rightarrow BST{<}T{>}\\
\mathit{delete} :&\quad BST{<}T{>} \times T &&\rightarrow BST{<}T{>}
\end{aligned}
$$

Type BinarySearchTree<T> sous-type de BT<T>

$$
\begin{aligned}
least : &\quad BST{<}T{>} & \rightarrow T \\
greatest : &\quad BST{<}T{>} & \rightarrow T \\
predecessor : &\quad BST{<}T{>} & \rightarrow T \\
successor : &\quad BST{<}T{>} & \rightarrow T \\
add : &\quad BST{<}T{>} \times T & \rightarrow BST{<}T{>} \\
deleteGreatest : &\quad BST{<}T{>} & \rightarrow BST{<}T{>} \\
deleteLeast : &\quad BST{<}T{>} & \rightarrow BST{<}T{>} \\
delete : &\quad BST{<}T{>} \times T & \rightarrow BST{<}T{>}
\end{aligned}
$$

Les préconditions

Les préconditions

$makeRoot(l, v, r)$ defini ssi

Les préconditions

$$makeRoot(l, v, r) \quad \text{defini ssi} \quad \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases}$$

Les préconditions

$$
\begin{array}{rl}
makeRoot(l, v, r) & \text{defini ssi} \quad \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\
least(b) & \text{defini ssi}
\end{array}
$$

Les préconditions

$$\begin{array}{rll} makeRoot(l, v, r) & \text{defini ssi} & \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\ least(b) & \text{defini ssi} & non\ isEmpty(b) \end{array}$$

Les préconditions

$$
\begin{array}{rll}
makeRoot(l, v, r) & \text{defini ssi} & \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\
least(b) & \text{defini ssi} & non\ isEmpty(b) \\
greatest(b) & \text{defini ssi} &
\end{array}
$$

Les préconditions

$$
\begin{array}{rll}
makeRoot(l, v, r) & \text{defini ssi} & \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\
least(b) & \text{defini ssi} & non\ isEmpty(b) \\
greatest(b) & \text{defini ssi} & non\ isEmpty(b)
\end{array}
$$

Les préconditions

$$
\begin{array}{rll}
makeRoot(l,v,r) & \text{defini ssi} & \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\
least(b) & \text{defini ssi} & non\ isEmpty(b) \\
greatest(b) & \text{defini ssi} & non\ isEmpty(b) \\
predecessor(b) & \text{defini ssi} &
\end{array}
$$

Les préconditions

$$\begin{array}{rll}
makeRoot(l, v, r) & \text{defini ssi} & \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\
least(b) & \text{defini ssi} & non\ isEmpty(b) \\
greatest(b) & \text{defini ssi} & non\ isEmpty(b) \\
predecessor(b) & \text{defini ssi} & hasLeft(b)
\end{array}$$

Les préconditions

$$
\begin{array}{rll}
makeRoot(l, v, r) & \text{defini ssi} & \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\
least(b) & \text{defini ssi} & non\ isEmpty(b) \\
greatest(b) & \text{defini ssi} & non\ isEmpty(b) \\
predecessor(b) & \text{defini ssi} & hasLeft(b) \\
successor(b) & \text{defini ssi} &
\end{array}
$$

Les préconditions

$$
\begin{array}{rll}
makeRoot(l, v, r) & \text{defini ssi} & \begin{cases} & (isEmpty(r) || v < least(r)) \\ \&\& & (isEmpty(l) || v > greatest(l)) \end{cases} \\
least(b) & \text{defini ssi} & non\ isEmpty(b) \\
greatest(b) & \text{defini ssi} & non\ isEmpty(b) \\
predecessor(b) & \text{defini ssi} & hasLeft(b) \\
successor(b) & \text{defini ssi} & hasRight(b)
\end{array}
$$

Axiomes avec *has*

Axiomes avec *has*

$$has(empty(), v) \quad =$$

Axiomes avec *has*

$$has(empty(), v) \quad = \quad faux$$

Axiomes avec *has*

$$
\begin{aligned}
has(empty(), v) &= faux \\
has(makeRoot(l, v_1, r), v_2) &=
\end{aligned}
$$

Axiomes avec *has*

$$
\begin{aligned}
has(empty(), v) &= faux \\
has(makeRoot(l, v_1, r), v_2) &= \begin{cases} vrai & \text{si } v_1 == v_2 \\ has(l, v_2) & \text{si } v_2 < v_1 \\ has(r, v_2) & \text{si } v_2 > v_1 \end{cases}
\end{aligned}
$$

Axiomes avec *least* et *greatest*

least : plus petite valeur de l'arbre.
greatest : plus grande valeur de l'arbre.

Axiomes avec *least* et *greatest*

least : plus petite valeur de l'arbre.
greatest : plus grande valeur de l'arbre.

$$least(makeRoot(l, v, r)) \quad =$$

Axiomes avec *least* et *greatest*

least : plus petite valeur de l'arbre.
greatest : plus grande valeur de l'arbre.

$$least(makeRoot(l, v, r)) = \begin{cases} v & \text{si } isEmpty(l) \\ least(l) & \text{sinon} \end{cases}$$

Axiomes avec *least* et *greatest*

least : plus petite valeur de l'arbre.
greatest : plus grande valeur de l'arbre.

$$least(makeRoot(l, v, r)) = \begin{cases} v & \text{si } isEmpty(l) \\ least(l) & \text{sinon} \end{cases}$$

$$greatest(makeRoot(l, v, r)) =$$

Axiomes avec *least* et *greatest*

least : plus petite valeur de l'arbre.
greatest : plus grande valeur de l'arbre.

$$\begin{aligned} least(makeRoot(l, v, r)) &= \begin{cases} v & \text{si } isEmpty(l) \\ least(l) & \text{sinon} \end{cases} \\ greatest(makeRoot(l, v, r)) &= \begin{cases} v & \text{si } isEmpty(r) \\ greatest(r) & \text{sinon} \end{cases} \end{aligned}$$

Axiomes avec *predecessor* et *successor*

predecessor : valeur qui précède directement celle de la racine.
successor : valeur qui succède directement à celle de la racine.

Axiomes avec *predecessor* et *successor*

predecessor : valeur qui précède directement celle de la racine.
successor : valeur qui succède directement à celle de la racine.

$$predecessor(makeRoot(l, v, r)) \quad =$$

Axiomes avec *predecessor* et *successor*

predecessor : valeur qui précède directement celle de la racine.
successor : valeur qui succède directement à celle de la racine.

$$predecessor(makeRoot(l, v, r)) \quad = \quad greatest(l)$$

Axiomes avec *predecessor* et *successor*

predecessor : valeur qui précède directement celle de la racine.
successor : valeur qui succède directement à celle de la racine.

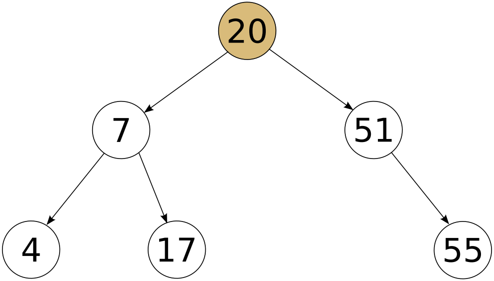
$$\begin{aligned} predecessor(makeRoot(l, v, r)) &= greatest(l) \\ successor(makeRoot(l, v, r)) &= \end{aligned}$$

Axiomes avec *predecessor* et *successor*

predecessor : valeur qui précède directement celle de la racine.
successor : valeur qui succède directement à celle de la racine.

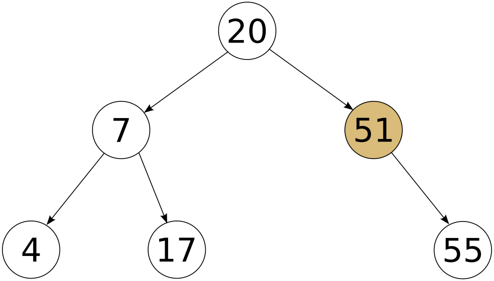
$$
\begin{aligned}
predecessor(makeRoot(l, v, r)) &= greatest(l) \\
successor(makeRoot(l, v, r)) &= least(r)
\end{aligned}
$$

Axiomes avec *add*



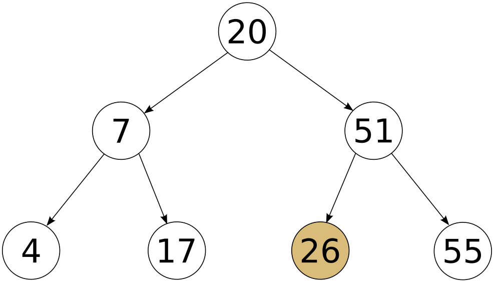
Exemple : ajout de la valeur 26.

Axiomes avec *add*



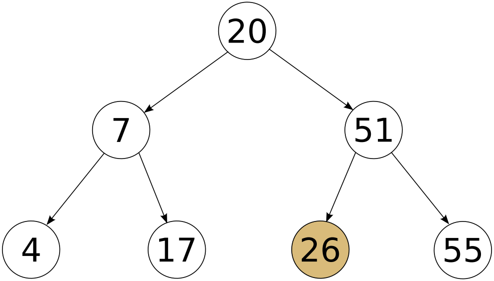
Exemple : ajout de la valeur 26.

Axiomes avec *add*



Exemple : ajout de la valeur 26.

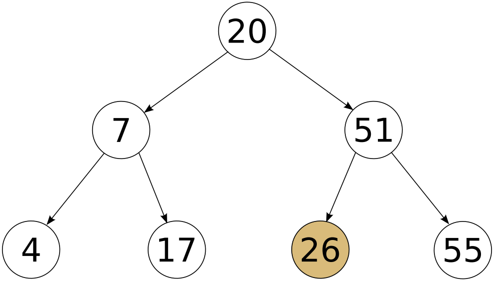
Axiomes avec *add*



Exemple : ajout de la valeur 26.

$$add(empty(), v) =$$

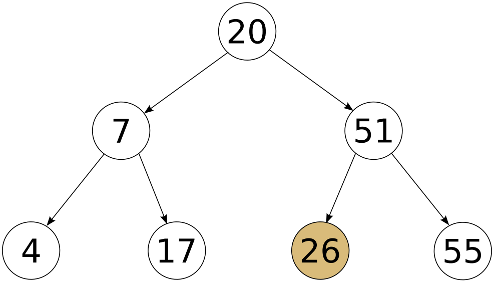
Axiomes avec *add*



Exemple : ajout de la valeur 26.

$$add(empty(), v) = makeRoot(empty(), v, empty())$$

Axiomes avec *add*

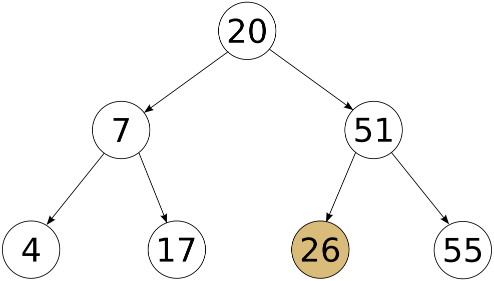


Exemple : ajout de la valeur 26.

$$add(empty(), v) = makeRoot(empty(), v, empty())$$

$$add(makeRoot(l, v_1, r), v_2) =$$

Axiomes avec *add*

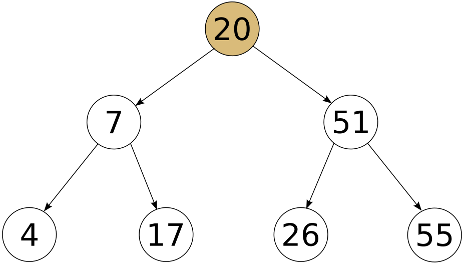


Exemple : ajout de la valeur 26.

$$add(empty(), v) = makeRoot(empty(), v, empty())$$

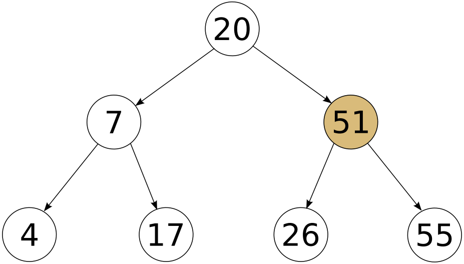
$$add(makeRoot(l, v_1, r), v_2) = \begin{cases} makeRoot(l, v_1, r) & \text{si } v_2 = v_1 \\ makeRoot(l, v_1, add(r, v_2)) & \text{si } v_2 > v_1 \\ makeRoot(add(l, v_2), v_1, r) & \text{si } v_2 < v_1 \end{cases}$$

Axiomes avec *deleteGreatest* et *deleteLeast*



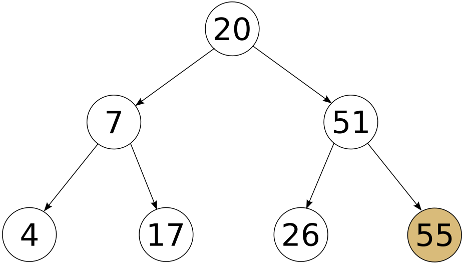
Exemple : *deleteGreatest* appliqué 2 fois.

Axiomes avec *deleteGreatest* et *deleteLeast*



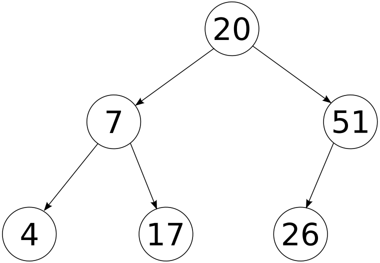
Exemple : *deleteGreatest* appliqué 2 fois.

Axiomes avec *deleteGreatest* et *deleteLeast*



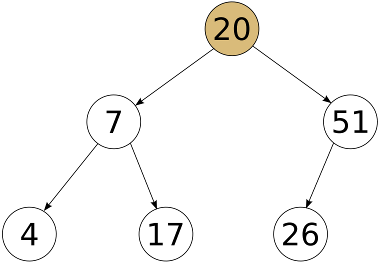
Exemple : *deleteGreatest* appliqué 2 fois.

Axiomes avec *deleteGreatest* et *deleteLeast*



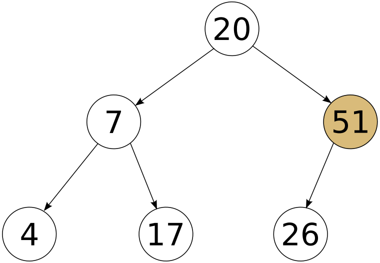
Exemple : *deleteGreatest* appliqué 2 fois.

Axiomes avec *deleteGreatest* et *deleteLeast*



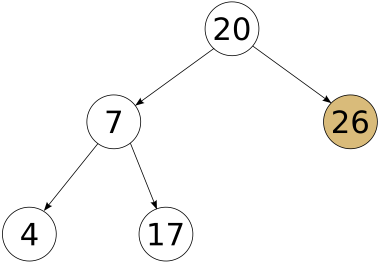
Exemple : *deleteGreatest* appliqué 2 fois.

Axiomes avec *deleteGreatest* et *deleteLeast*



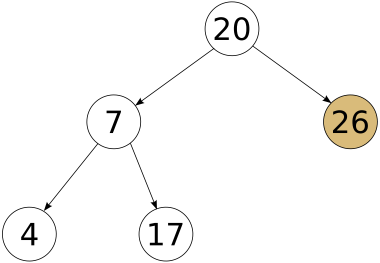
Exemple : *deleteGreatest* appliqué 2 fois.

Axiomes avec *deleteGreatest* et *deleteLeast*



Exemple : *deleteGreatest* appliqué 2 fois.

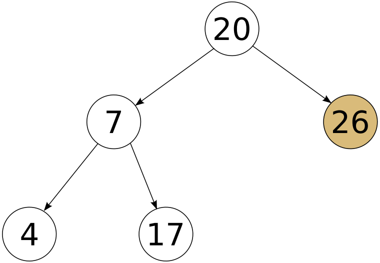
Axiomes avec *deleteGreatest* et *deleteLeast*



Exemple : *deleteGreatest* appliqué 2 fois.

$$delG(empty()) \quad =$$

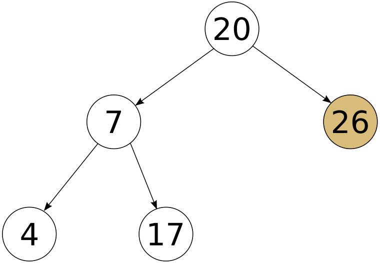
Axiomes avec *deleteGreatest* et *deleteLeast*



Exemple : *deleteGreatest* appliqué 2 fois.

$$delG(empty()) \quad = \quad empty()$$

Axiomes avec *deleteGreatest* et *deleteLeast*

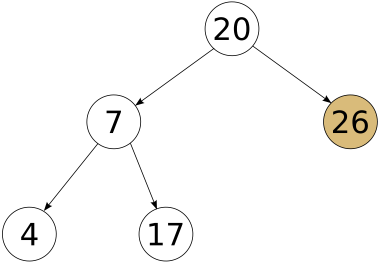


Exemple : *deleteGreatest* appliqué 2 fois.

$$delG(empty()) = empty()$$

$$delG(makeR(l, v, r)) =$$

Axiomes avec *deleteGreatest* et *deleteLeast*

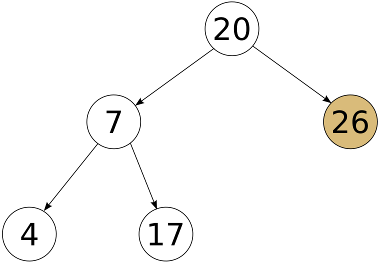


Exemple : *deleteGreatest* appliqué 2 fois.

$$delG(empty()) = empty()$$

$$delG(makeR(l, v, r)) = \begin{cases} l & \text{si } isEmpty(r) \\ makeR(l, v, delG(r)) & \text{sinon} \end{cases}$$

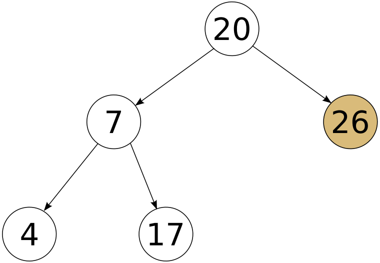
Axiomes avec *deleteGreatest* et *deleteLeast*



Exemple : *deleteGreatest* appliqué 2 fois.

$$
\begin{aligned}
delG(empty()) &= empty() \\
delG(makeR(l, v, r)) &= \begin{cases} l & \text{si } isEmpty(r) \\ makeR(l, v, delG(r)) & \text{sinon} \end{cases} \\
delL(empty()) &=
\end{aligned}
$$

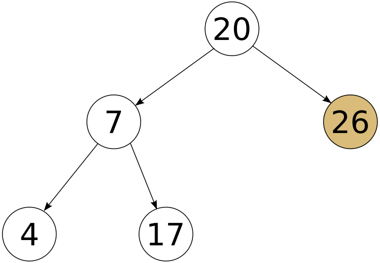
Axiomes avec *deleteGreatest* et *deleteLeast*



Exemple : *deleteGreatest* appliqué 2 fois.

$$\begin{aligned} delG(empty()) &= empty() \\ delG(makeR(l, v, r)) &= \begin{cases} l & \text{si } isEmpty(r) \\ makeR(l, v, delG(r)) & \text{sinon} \end{cases} \\ delL(empty()) &= empty() \end{aligned}$$

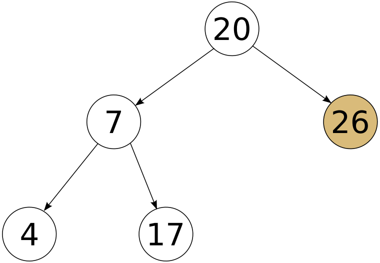
Axiomes avec *deleteGreatest* et *deleteLeast*



Exemple : *deleteGreatest* appliqué 2 fois.

$$
\begin{aligned}
delG(empty()) &= empty() \\
delG(makeR(l, v, r)) &= \begin{cases} l & \text{si } isEmpty(r) \\ makeR(l, v, delG(r)) & \text{sinon} \end{cases} \\
delL(empty()) &= empty() \\
delL(makeR(l, v, r)) &=
\end{aligned}
$$

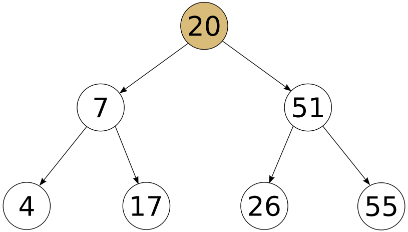
Axiomes avec *deleteGreatest* et *deleteLeast*



Exemple : *deleteGreatest* appliqué 2 fois.

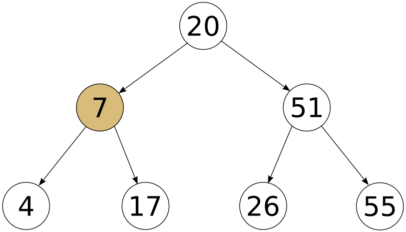
$$
\begin{array}{rcl}
delG(empty()) & = & empty() \\
delG(makeR(l, v, r)) & = & \begin{cases} l & \text{si } isEmpty(r) \\ makeR(l, v, delG(r)) & \text{sinon} \end{cases} \\
delL(empty()) & = & empty() \\
delL(makeR(l, v, r)) & = & \begin{cases} r & \text{si } isEmpty(l) \\ makeR(delL(l), v, r) & \text{sinon} \end{cases}
\end{array}
$$

Axiomes avec *delete*



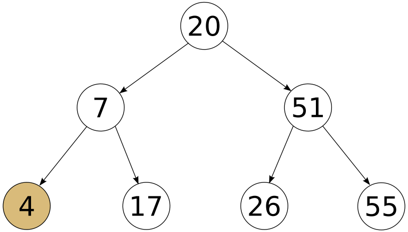
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



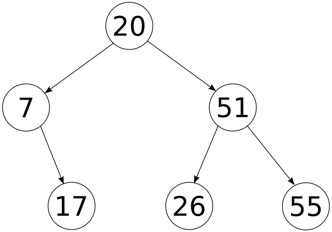
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



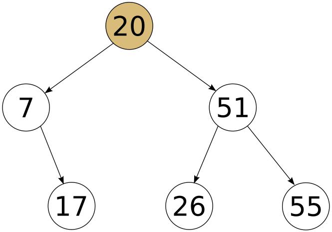
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



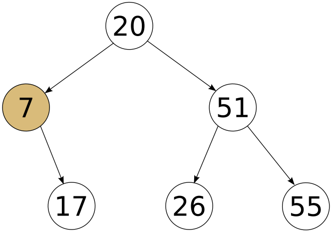
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



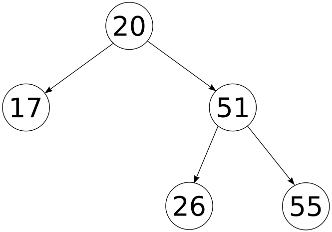
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



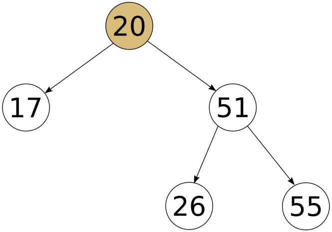
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



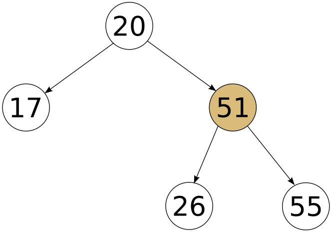
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



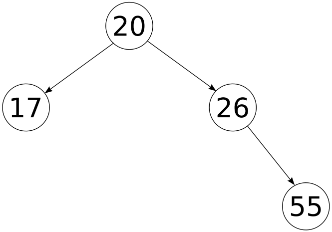
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



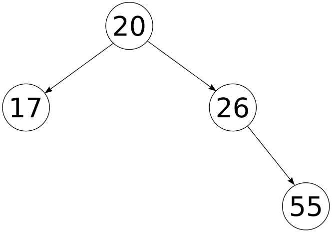
Exemple : *delete* appliqué à 4, 7 puis 51.

Axiomes avec *delete*



Exemple : *delete* appliqué à 4, 7 puis 51.

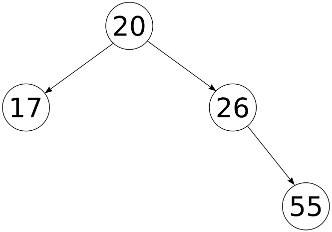
Axiomes avec *delete*



Exemple : *delete* appliqué à 4, 7 puis 51.

$$del(empty(), v) \quad =$$

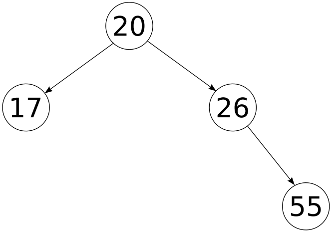
Axiomes avec *delete*



Exemple : *delete* appliqué à 4, 7 puis 51.

$$del(empty(), v) \quad = \quad empty()$$

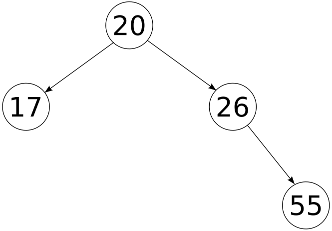
Axiomes avec *delete*



Exemple : *delete* appliqué à 4, 7 puis 51.

$$
\begin{aligned}
del(empty(), v) &= empty() \\
del(makeR(l, v_1, r), v_2) &=
\end{aligned}
$$

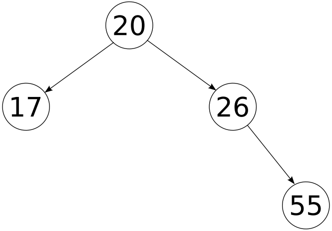
Axiomes avec *delete*



Exemple : *delete* appliqué à 4, 7 puis 51.

$$
\begin{aligned}
del(empty(), v) &= empty() \\
del(makeR(l, v_1, r), v_2) &= \begin{cases} makeR(del(l, v_2), v_1, r) & \text{si } v_2 < v_1 \\ makeR(l, v_1, del(r, v_2)) & \text{si } v_2 > v_1 \end{cases}
\end{aligned}
$$

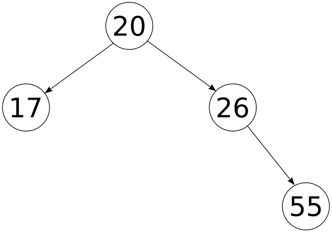
Axiomes avec *delete*



Exemple : *delete* appliqué à 4, 7 puis 51.

$$
\begin{aligned}
del(empty(), v) &= empty() \\
del(makeR(l, v_1, r), v_2) &= \begin{cases} makeR(del(l, v_2), v_1, r) & \text{si } v_2 < v_1 \\ makeR(l, v_1, del(r, v_2)) & \text{si } v_2 > v_1 \end{cases} \\
del(makeR(l, v, r), v) &=
\end{aligned}
$$

Axiomes avec *delete*



Exemple : *delete* appliqué à 4, 7 puis 51.

$$\begin{aligned}
del(empty(), v) &= empty() \\
del(makeR(l, v_1, r), v_2) &= \begin{cases} makeR(del(l, v_2), v_1, r) & \text{si } v_2 < v_1 \\ makeR(l, v_1, del(r, v_2)) & \text{si } v_2 > v_1 \end{cases} \\
del(makeR(l, v, r), v) &= \begin{cases} r \text{ si } isEmpty(l) \\ l \text{ si } isEmpty(r) \\ makeR(delG(l), greatest(l), r) & \text{sinon} \end{cases}
\end{aligned}$$

Plan

3.24pt

3.24pt

TELECOM nancy

Ingénieurs du numérique • Inspiring your digital future

Ajouts triés

L'opération *add* **ajoute** une valeur en tant que **feuille**.
Que se passe-t'il si on ajoute des valeurs déjà **ordonnées** ?

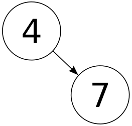
Exemple avec [4, 7, 17, 20, 26, 27, 32, 45, ...] :

4

Ajouts triés

L'opération *add* **ajoute** une valeur en tant que **feuille**.
Que se passe-t'il si on ajoute des valeurs déjà **ordonnées** ?

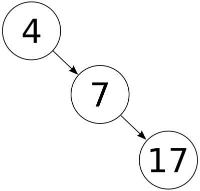
Exemple avec [4, 7, 17, 20, 26, 27, 32, 45, ...] :



Ajouts triés

L'opération *add* **ajoute** une valeur en tant que **feuille**.
Que se passe-t'il si on ajoute des valeurs déjà **ordonnées** ?

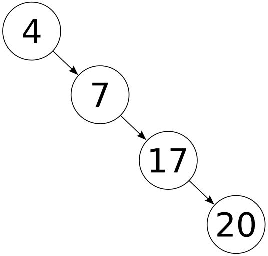
Exemple avec [4, 7, 17, 20, 26, 27, 32, 45, ...] :



Ajouts triés

L'opération *add* **ajoute** une valeur en tant que **feuille**.
Que se passe-t'il si on ajoute des valeurs déjà **ordonnées** ?

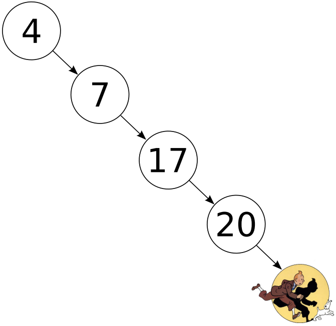
Exemple avec [4, 7, 17, 20, 26, 27, 32, 45, ...] :



Ajouts triés

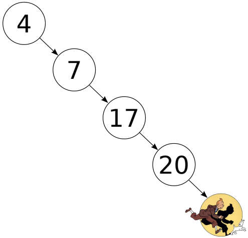
L'opération *add* **ajoute** une valeur en tant que **feuille**.
Que se passe-t'il si on ajoute des valeurs déjà **ordonnées** ?

Exemple avec [4, 7, 17, 20, 26, 27, 32, 45, ...] :



Rappel de l'objectif (lune)

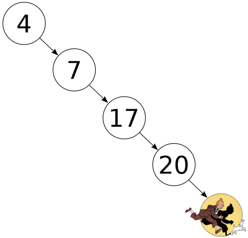
Une structure de données doit permettre d'optimiser :



Rappel de l'objectif (lune)

Une structure de données doit permettre d'optimiser :

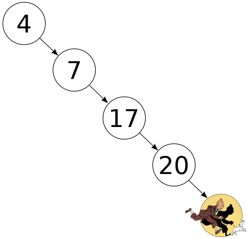
- soit la place occupée par les données



Rappel de l'objectif (lune)

Une structure de données doit permettre d'optimiser :

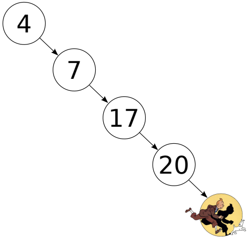
- soit la place occupée par les données
- soit le temps l'accès aux données



Rappel de l'objectif (lune)

Une structure de données doit permettre d'optimiser :

- soit la place occupée par les données
- soit le temps l'accès aux données

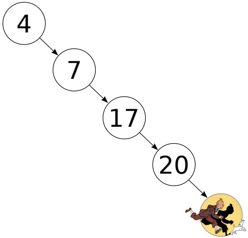


L'arbre est une structure qui a pour objectif d'optimiser

Rappel de l'objectif (lune)

Une structure de données doit permettre d'optimiser :

- soit la place occupée par les données
- soit le temps l'accès aux données

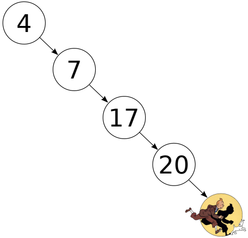


L'arbre est une structure qui a pour objectif d'optimiser **le temps d'accès aux données.**

Rappel de l'objectif (lune)

Une structure de données doit permettre d'optimiser :

- soit la place occupée par les données
- soit le temps l'accès aux données



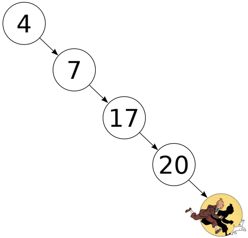
L'arbre est une structure qui a pour objectif d'optimiser **le temps d'accès aux données.**

Quel est le problème ici ?

Rappel de l'objectif (lune)

Une structure de données doit permettre d'optimiser :

- soit la place occupée par les données
- soit le temps l'accès aux données

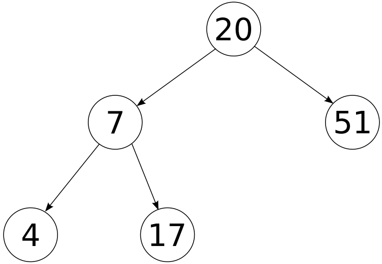


L'arbre est une structure qui a pour objectif d'optimiser **le temps d'accès aux données.**

Quel est le problème ici ?

Une solution ?

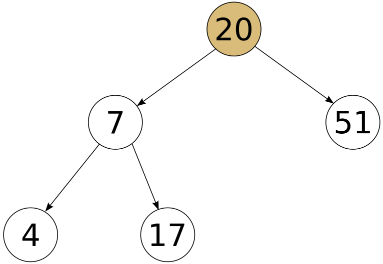
L'équilibrage



Rotation droite : $(4 < 7 < 17) < 20 < 51$

On peut **rééquilibrer** l'arbre grâce à des **rotations**.

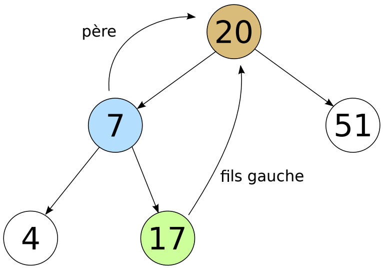
L'équilibrage



Rotation droite : $(4 < 7 < 17) < 20 < 51$

On peut **rééquilibrer** l'arbre grâce à des **rotations**.

L'équilibrage

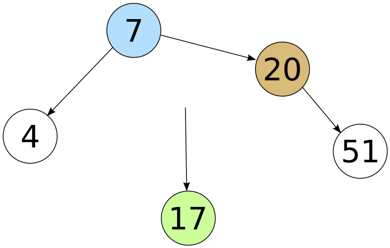


Rotation droite : $(4 < 7 < 17) < 20 < 51$

On peut **rééquilibrer** l'arbre grâce à des **rotations**.

- inversion d'une relation de descendance père-fils...

L'équilibrage

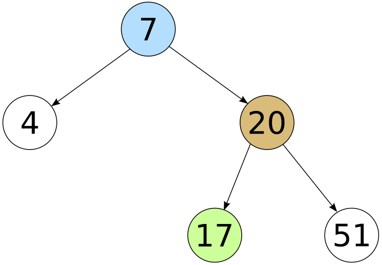


Rotation droite : $(4 < 7 < 17) < 20 < 51$

On peut **rééquilibrer** l'arbre grâce à des **rotations**.

- inversion d'une relation de descendance père-fils...

L'équilibrage



Rotation droite : $4 < 7 < (17 < 20 < 51)$

On peut **rééquilibrer** l'arbre grâce à des **rotations**.

- inversion d'une relation de descendance père-fils...
- ... tout en préservant l'ordre des éléments

Plan

3.24pt

3.24pt

Qu'est ce que c'est ?

Les arbres **AVL** (Georgii **A**delson-**V**elsky et Evguenii **L**andis) sont des **ABR équilibrés**.

Principe

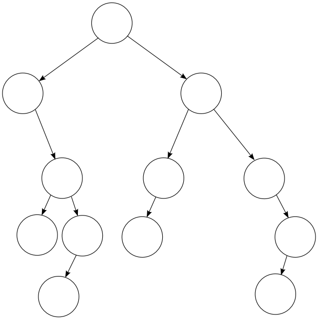
- On attribue un **poids** à chaque nœud
- On **équilibre** l'arbre (si besoin) après chaque **insertion** en utilisant des **rotations**

Définition

Un ABR est dit équilibré si tous ses nœuds ont un poid compris entre -1 et 1

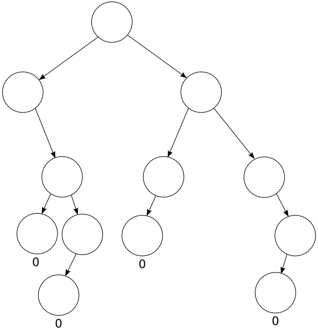
Calcul du poids

Le **poids** d'un nœud est égal à la **hauteur** de son **sous-arbre gauche moins** la hauteur de son sous-arbre **droit**.



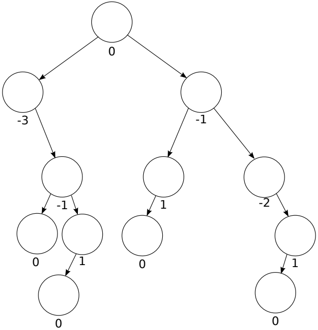
Calcul du poids

Le **poids** d'un nœud est égal à la **hauteur** de son **sous-arbre gauche moins** la hauteur de son sous-arbre **droit**.



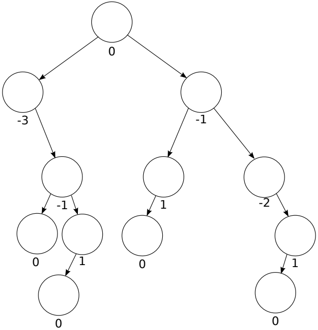
Calcul du poids

Le **poids** d'un nœud est égal à la **hauteur** de son **sous-arbre gauche moins** la hauteur de son sous-arbre **droit**.



Calcul du poids

Le **poids** d'un nœud est égal à la **hauteur** de son **sous-arbre gauche moins** la hauteur de son sous-arbre **droit**.

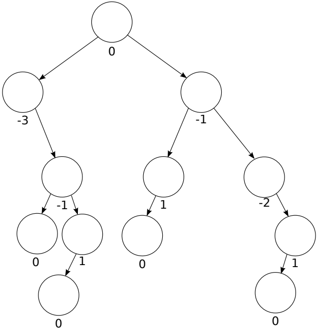


Definition axiomatique

$$balance(empty()) =$$
$$balance(makeRoot(l, v, r)) =$$

Calcul du poids

Le **poids** d'un nœud est égal à la **hauteur** de son **sous-arbre gauche moins** la hauteur de son sous-arbre **droit**.

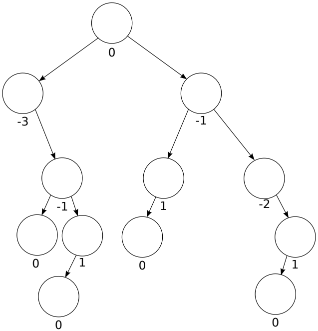


Definition axiomatique

$$balance(empty()) = 0$$
$$balance(makeRoot(l, v, r)) =$$

Calcul du poids

Le **poids** d'un nœud est égal à la **hauteur** de son **sous-arbre gauche moins** la hauteur de son sous-arbre **droit**.

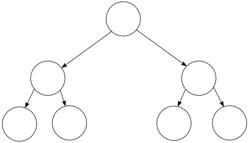


Definition axiomatique

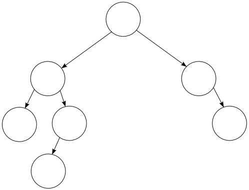
$$balance(empty()) = 0$$
$$balance(makeRoot(l, v, r)) = height(l) - height(r)$$

Exercice 4.F

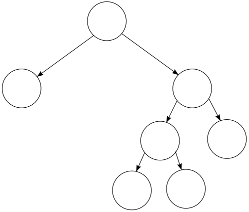
Calculez les facteurs d'équilibrage des arbres ci-dessous et dire s'ils sont équilibrés au sens des AVL.



(a) Équilibré ?



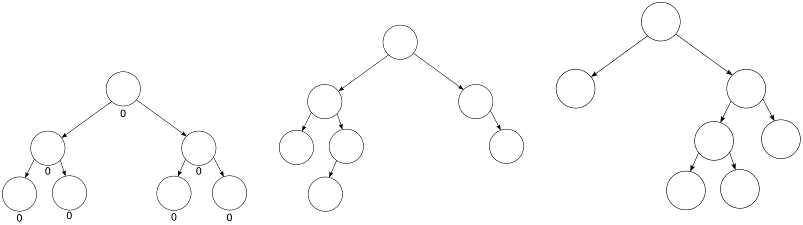
(b) Équilibré ?



(c) Équilibré ?

Exercice 4.F

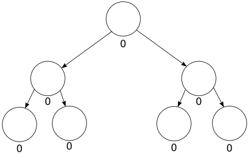
Calculez les facteurs d'équilibrage des arbres ci-dessous et dire s'ils sont équilibrés au sens des AVL.



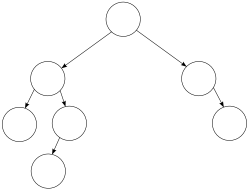
(a) Équilibré ? (b) Équilibré ? (c) Équilibré ?

Exercice 4.F

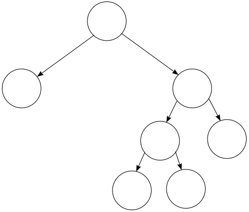
Calculez les facteurs d'équilibrage des arbres ci-dessous et dire s'ils sont équilibrés au sens des AVL.



(a) Oui



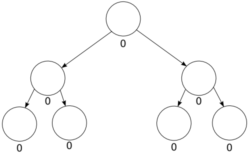
(b) Équilibré ?



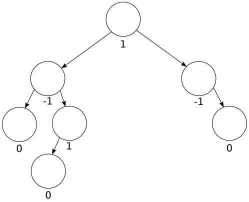
(c) Équilibré ?

Exercice 4.F

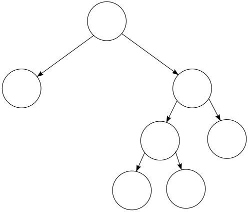
Calculez les facteurs d'équilibrage des arbres ci-dessous et dire s'ils sont équilibrés au sens des AVL.



(a) Oui



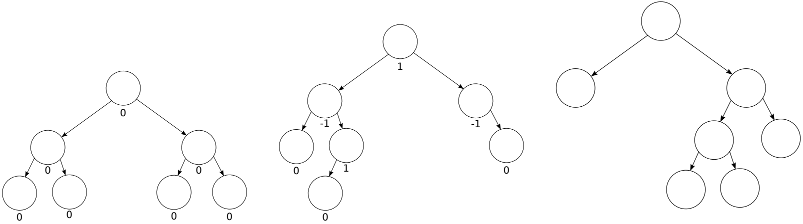
(b) Équilibré ?



(c) Équilibré ?

Exercice 4.F

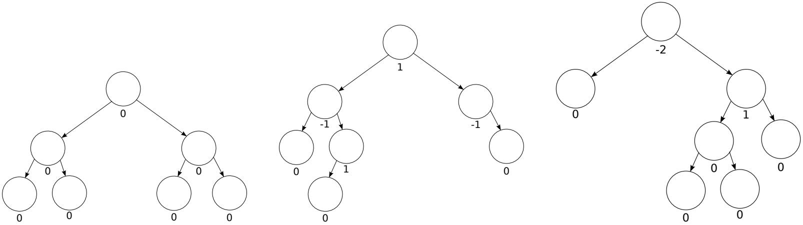
Calculez les facteurs d'équilibrage des arbres ci-dessous et dire s'ils sont équilibrés au sens des AVL.



(a) Oui (b) Oui (c) Équilibré ?

Exercice 4.F

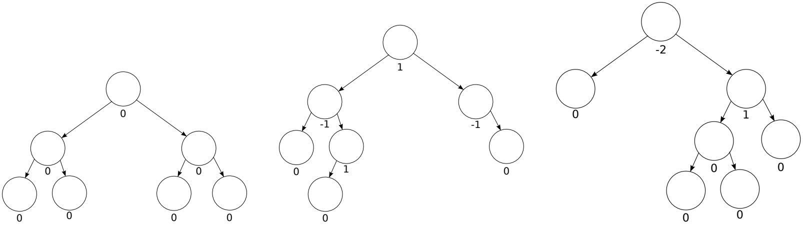
Calculez les facteurs d'équilibrage des arbres ci-dessous et dire s'ils sont équilibrés au sens des AVL.



(a) Oui (b) Oui (c) Équilibré ?

Exercice 4.F

Calculez les facteurs d'équilibrage des arbres ci-dessous et dire s'ils sont équilibrés au sens des AVL.



(a) Oui (b) Oui (c) Non (-2)

Comment ça marche ?

Une seule règle :

> Tous les nœuds d'un arbre AVL ont un poids de valeur absolue strictement inférieure à 2.

Un algorithme simple :

I) **Ajouter** le nœud N en tant que feuille de l'arbre comme dans un ABR

II) Remonter l'arbre depuis N et **mettre à jour** le poids de chaque nœud rencontré

III) **Équilibrer** le premier nœud rencontré dont le nouveau poids a une valeur absolue égale à 2

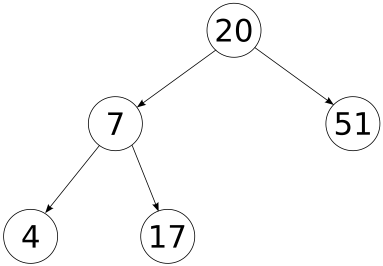
Algorithme d'équilibrage

Soit N_P le premier nœud visité de poids de valeur absolue égale à 2.
Soit N_F le nœud visité juste avant N_P.

On équilibre l'arbre AVL avec l'algorithme suivant :

1. Si $balance(N_P) = 2$ alors
   - 1.1 Si $balance(N_F) = -1$ alors $rotateLeft(N_F)$
   - 1.2 $rotateRight(N_P)$
2. Sinon si $balance(N_P) = -2$ alors
   - 2.1 Si $balance(N_F) = 1$ alors $rotateRight(N_F)$
   - 2.2 $rotateLeft(N_P)$

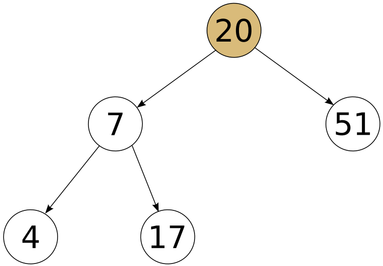
La rotation droite



Rotation droite : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

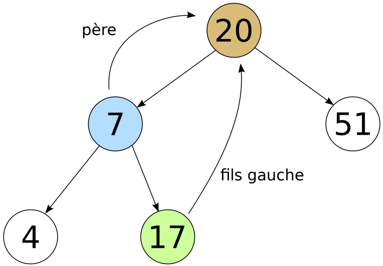
La rotation droite



Rotation droite : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

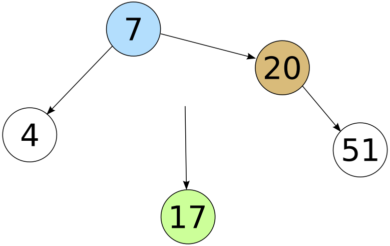
La rotation droite



Rotation droite : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

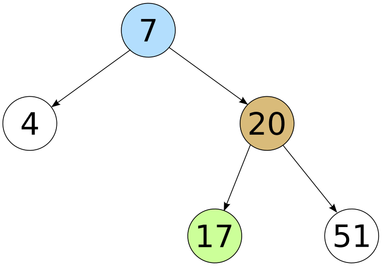
La rotation droite



Rotation droite : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

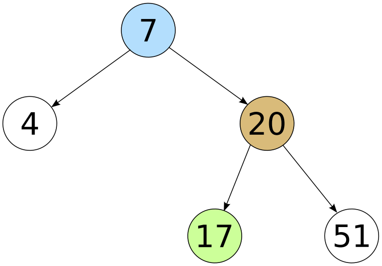
La rotation droite



Rotation droite : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

La rotation droite

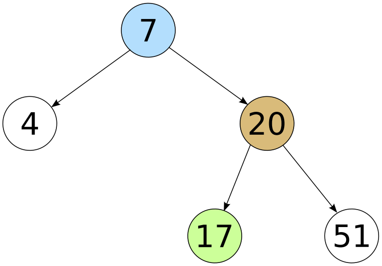


Rotation droite : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

Operation

La rotation droite

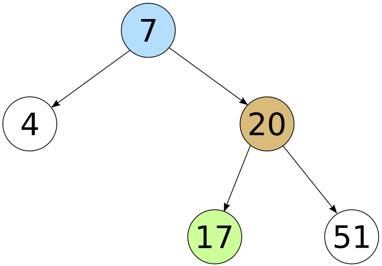


Rotation droite : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

Operation *rightRotate* : $BST{<}T{>} \rightarrow BST{<}T{>}$

La rotation droite



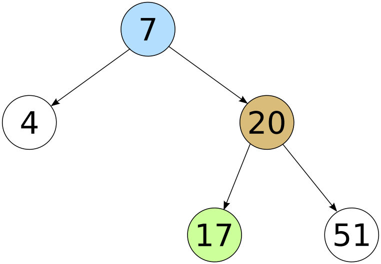
Rotation droite : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

Operation $rightRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition

La rotation droite



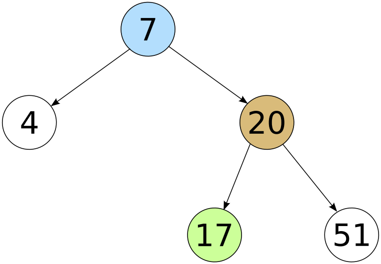
Rotation droite : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

Operation $rightRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition $rightRotate(b)$ défini ssi $hasLeft(b)$

La rotation droite



Rotation droite : $4 < 7 < (17 < 20 < 51)$

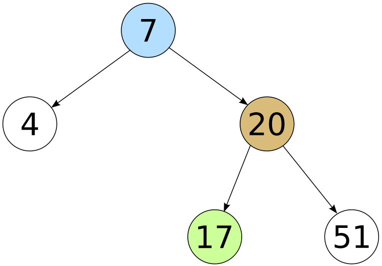
Définition axiomatique

Operation $rightRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition $rightRotate(b)$ défini ssi $hasLeft(b)$

Axiomes

La rotation droite



Rotation droite : $4 < 7 < (17 < 20 < 51)$

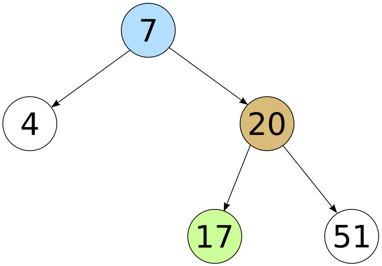
Définition axiomatique

Operation $rightRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition $rightRotate(b)$ défini ssi $hasLeft(b)$

Axiomes $rightRotate(makeRoot(l, v, r)) =$

La rotation droite



Rotation droite : $4 < 7 < (17 < 20 < 51)$

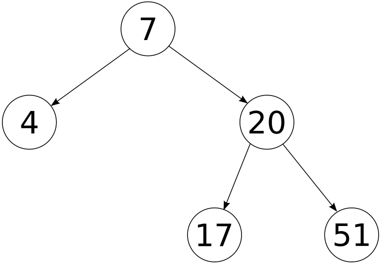
Définition axiomatique

Operation $rightRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition $rightRotate(b)$ défini ssi $hasLeft(b)$

Axiomes $rightRotate(makeRoot(l, v, r)) =$
$makeRoot(left(l), value(l), makeRoot(right(l), v, r))$

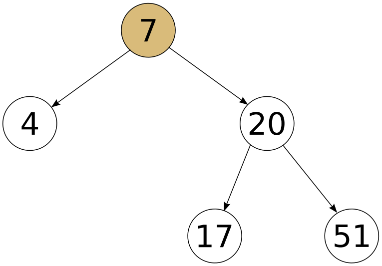
La rotation gauche



Rotation gauche : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

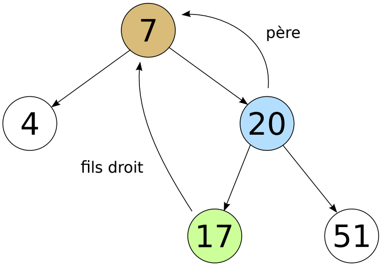
La rotation gauche



Rotation gauche : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

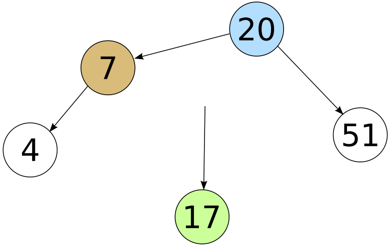
La rotation gauche



Rotation gauche : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

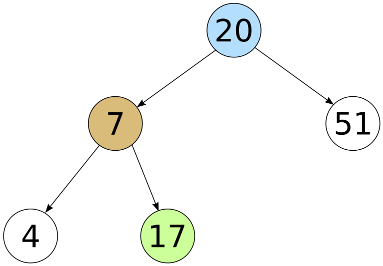
La rotation gauche



Rotation gauche : $4 < 7 < (17 < 20 < 51)$

Définition axiomatique

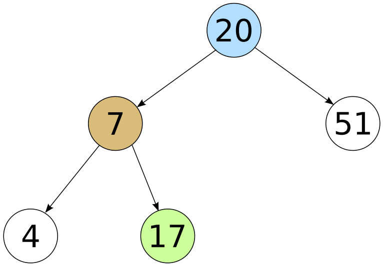
La rotation gauche



Rotation gauche : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

La rotation gauche

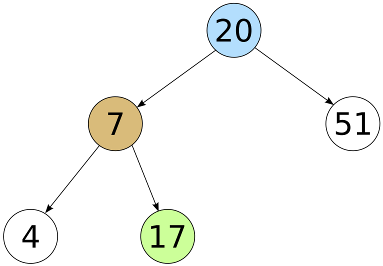


Rotation gauche : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

Operation

La rotation gauche

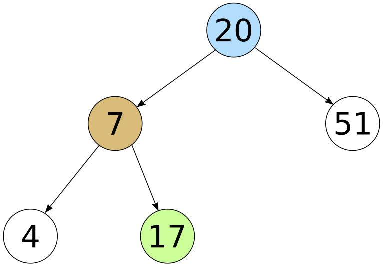


Rotation gauche : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

Operation $leftRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

La rotation gauche



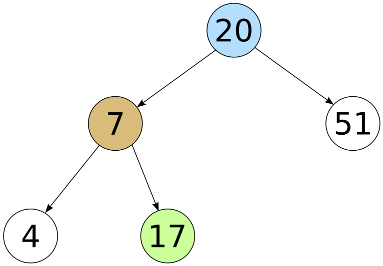
Rotation gauche : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

Operation $leftRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition

La rotation gauche



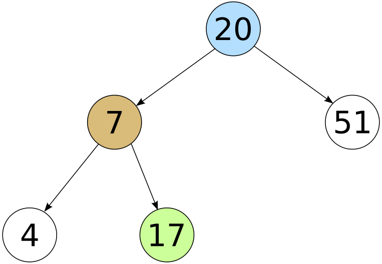
Rotation gauche : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

Operation $leftRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition $leftRotate(b)$ défini ssi $hasRight(b)$

La rotation gauche



Rotation gauche : $(4 < 7 < 17) < 20 < 51$

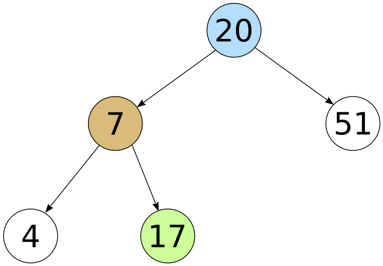
Définition axiomatique

Operation $leftRotate : BST{<}T{>} \to BST{<}T{>}$

Précondition $leftRotate(b)$ défini ssi $hasRight(b)$

Axiomes

La rotation gauche



Rotation gauche : $(4 < 7 < 17) < 20 < 51$

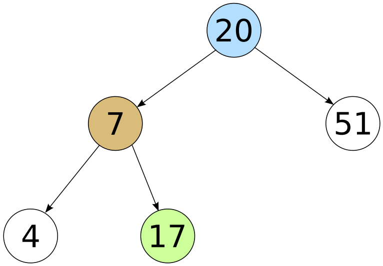
Définition axiomatique

Operation $leftRotate : BST{<}T{>} \rightarrow BST{<}T{>}$

Précondition $leftRotate(b)$ défini ssi $hasRight(b)$

Axiomes $leftRotate(makeRoot(l, v, r)) =$

La rotation gauche



Rotation gauche : $(4 < 7 < 17) < 20 < 51$

Définition axiomatique

Operation $leftRotate : BST\!<\!T\!> \rightarrow BST\!<\!T\!>$

Précondition $leftRotate(b)$ défini ssi $hasRight(b)$

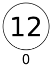
Axiomes $leftRotate(makeRoot(l, v, r)) =$
$makeRoot(makeRoot(l, v, left(r)), value(r), right(r))$

Enoncé (exercice 4G)

Construisez un arbre AVL en ajoutant dans l'ordre les éléments suivants :

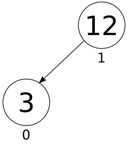
12, 3, 2, 5, 4, 7, 9, 11, 14, 10

Solution



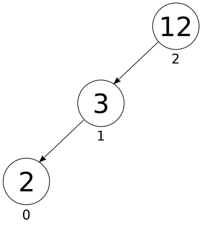
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



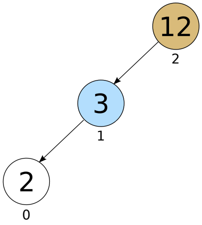
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



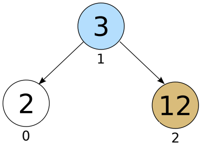
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



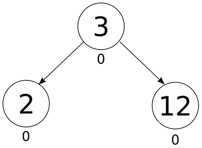
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



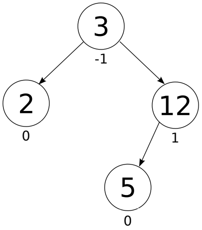
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



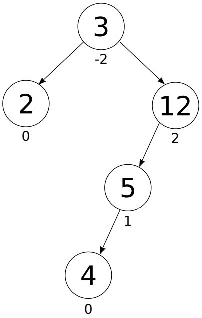
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



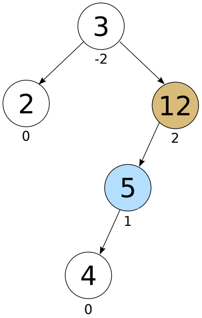
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



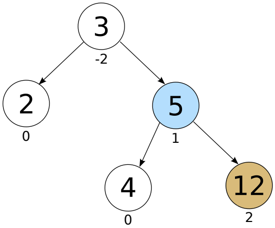
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



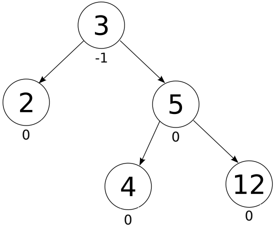
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



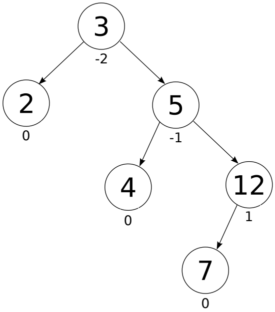
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



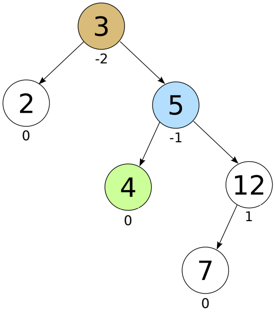
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



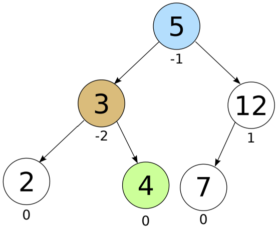
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



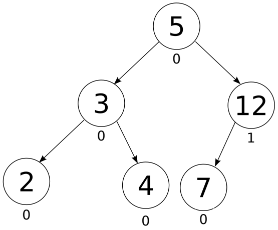
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



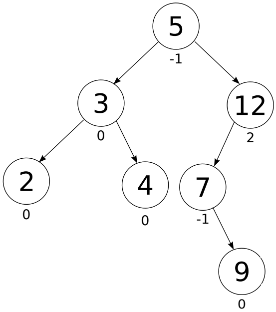
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



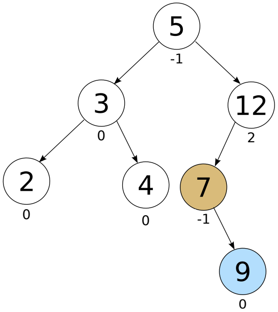
`http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html`

Solution



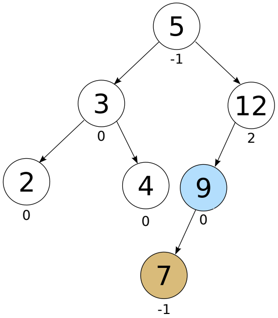
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



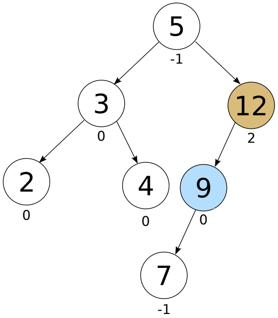
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



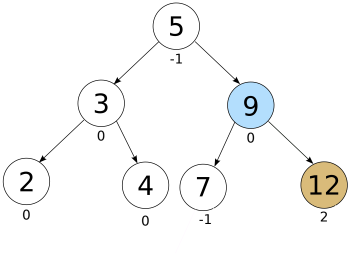
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



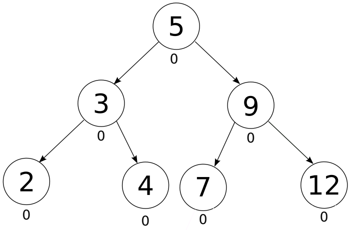
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



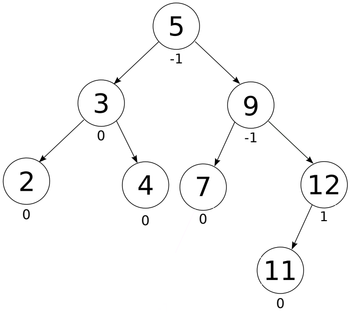
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



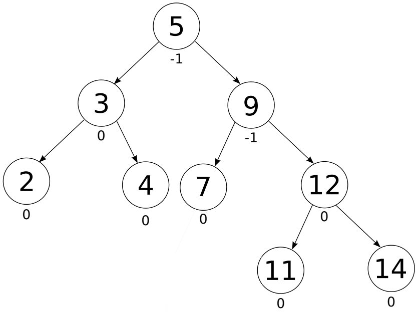
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



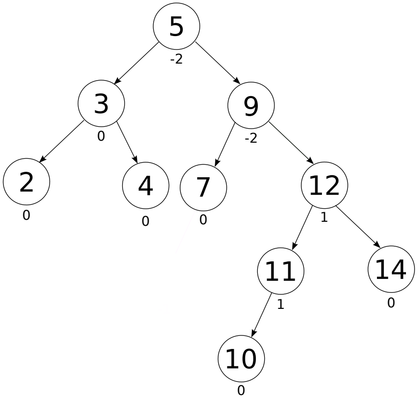
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



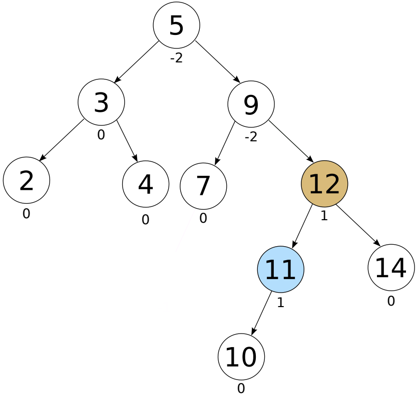
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



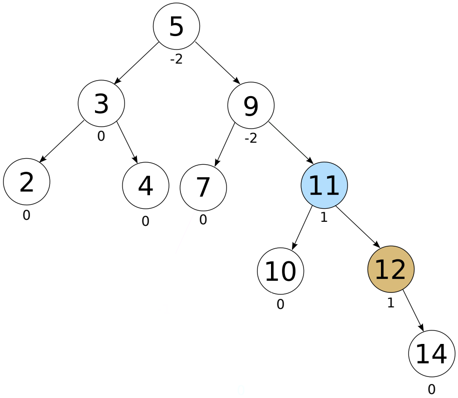
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



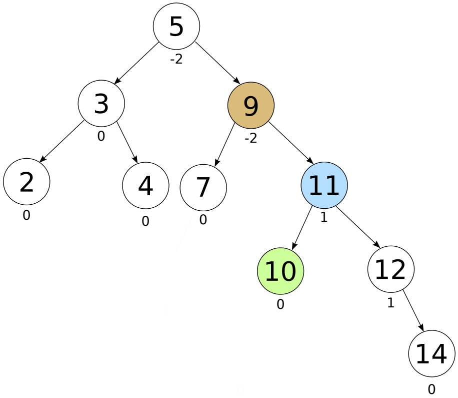
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



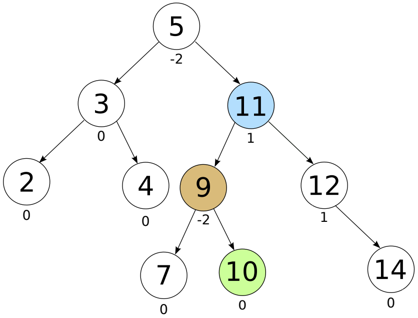
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



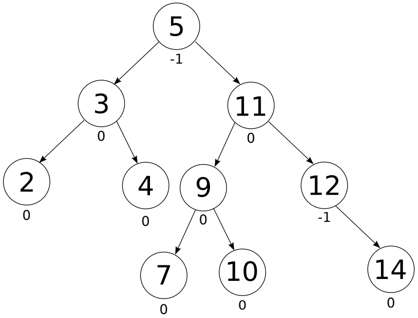
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Solution



http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

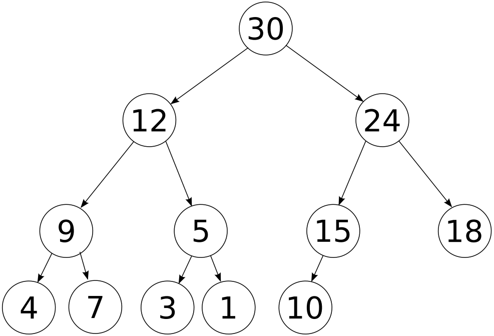
Solution



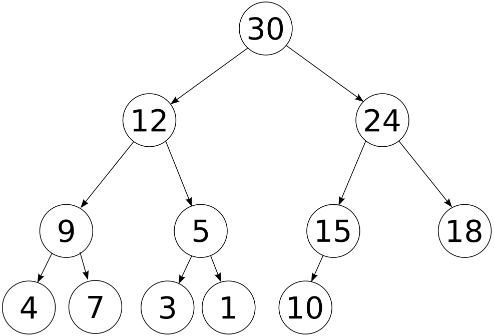
http://qmatica.com/DataStructures/Trees/AVL/AVLTree.html

Plan

En trois points

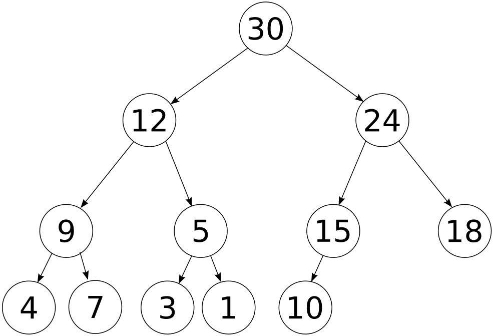


En trois points



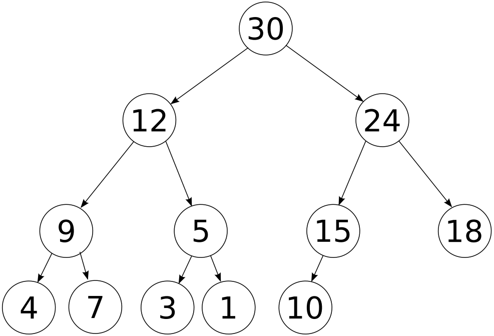
- C'est un **arbre binaire parfait** : tous les niveaux sont remplis

En trois points



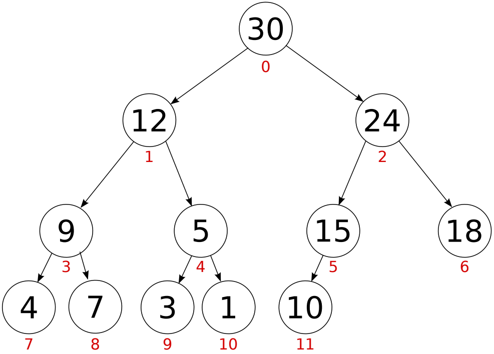
- C'est un **arbre binaire parfait** : tous les niveaux sont remplis
- La valeur de la **racine** est **supérieure** à la valeur de **tous ses descendants**.

En trois points



- C'est un **arbre binaire parfait** : tous les niveaux sont remplis
- La valeur de la **racine** est **supérieure** à la valeur de **tous ses descendants**.
- Et c'est tout !

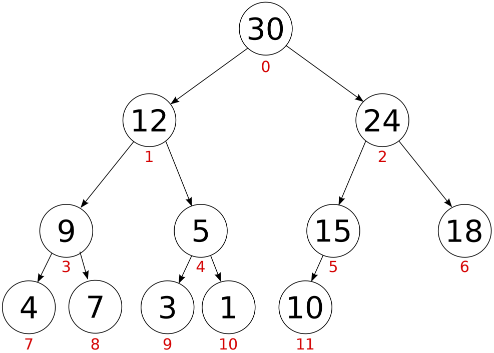
Numérotation canonique



À partir de la numérotation des nœuds ci-dessus :

- $\mathit{filsGauche}(i) =$
- $\mathit{filsDroit}(i) =$
- $\mathit{pere}(i) =$

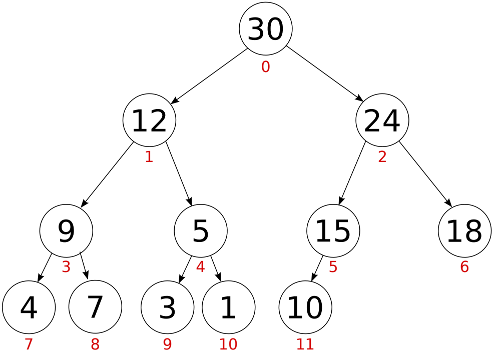
Numérotation canonique



À partir de la numérotation des nœuds ci-dessus :

- $filsGauche(i) = 2 \times i + 1$
- $filsDroit(i) =$
- $pere(i) =$

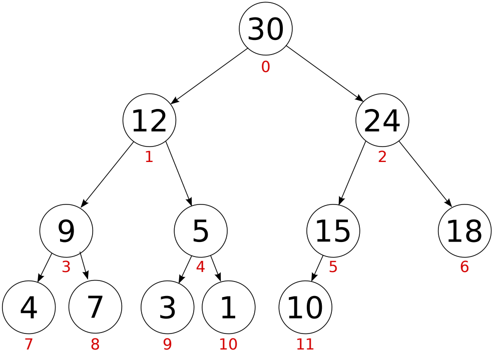
Numérotation canonique



À partir de la numérotation des nœuds ci-dessus :

- $filsGauche(i) = 2 \times i + 1$
- $filsDroit(i) = 2 \times i + 2$
- $pere(i) =$

Numérotation canonique



À partir de la numérotation des nœuds ci-dessus :

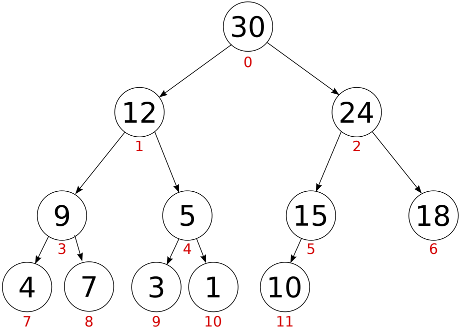
- $filsGauche(i) = 2 \times i + 1$
- $filsDroit(i) = 2 \times i + 2$
- $pere(i) = \lfloor (i-1)/2 \rfloor$

Utilisation

Pourquoi cette numérotation canonique ?

Utilisation

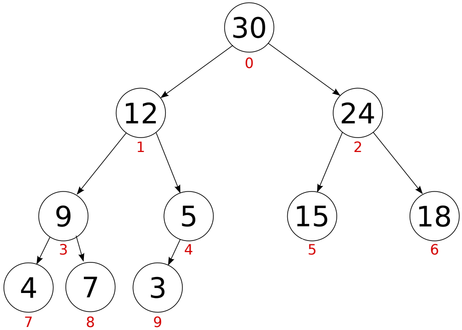
Pourquoi cette numérotation canonique ?



30	12	24	9	5	15	18	4	7	3	1	10
0	1	2	3	4	5	6	7	8	9	10	11

Pour une implémentation dans un tableau !

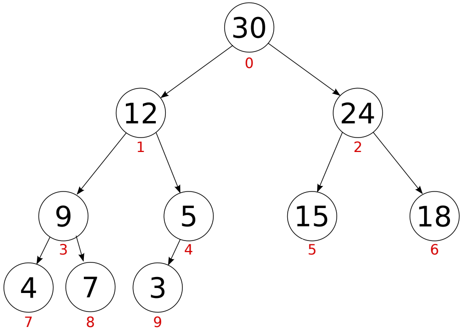
Algorithme et exemple



30	12	24	9	5	15	18	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** N à la première place libre

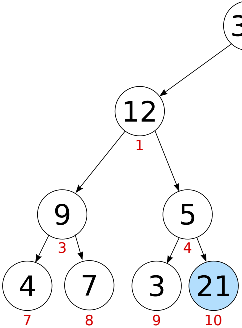
Algorithme et exemple



30	12	24	9	5	15	18	4	7	3	21	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** *N* à la première place libre

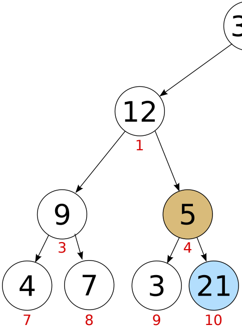
Algorithme et exemple



30	12	24	9	5	15	18	4	7	3	21	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** N à la première place libre

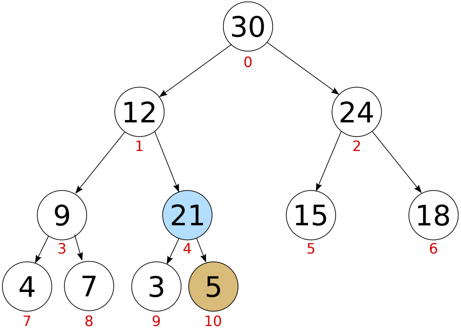
Algorithme et exemple



30	12	24	9	5	15	18	4	7	3	21	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** N à la première place libre
2. **Permuter** N avec son père N_P tant que $N > N_P$

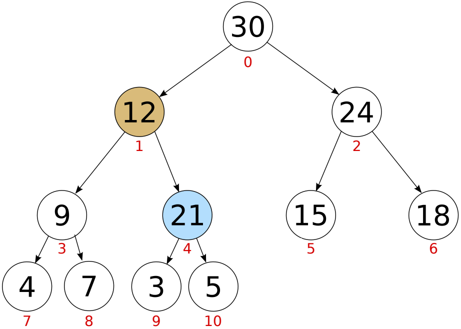
Algorithme et exemple



30	12	24	9	21	15	18	4	7	3	5	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** N à la première place libre
2. **Permuter** N avec son père N_P tant que $N > N_P$

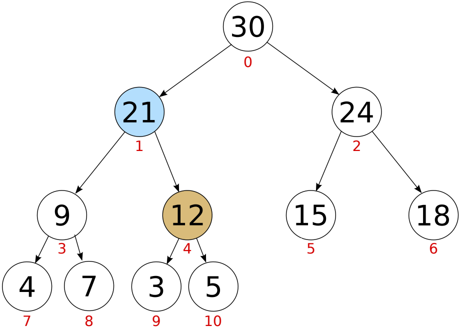
Algorithme et exemple



30	12	24	9	21	15	18	4	7	3	5	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** N à la première place libre
2. **Permuter** N avec son père N_P tant que $N > N_P$

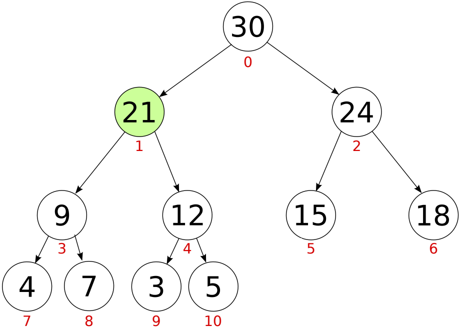
Algorithme et exemple



30	21	24	9	12	15	18	4	7	3	5	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** N à la première place libre
2. **Permuter** N avec son père N_P tant que $N > N_P$

Algorithme et exemple



30	21	24	9	12	15	18	4	7	3	5	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Ajouter** N à la première place libre
2. **Permuter** N avec son père N_P tant que $N > N_P$

Fonction add(int v)

Écrivez la fonction d'ajout ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void add( int v )
{



    // À compléter




}
```

Fonction add(int v)

Écrivez la fonction d'ajout ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void add( int v )
{
    int i = size;
    while ( i>0 && tab[(i-1)/2]<v )
    {
        tab[i] = tab[(i-1)/2];
        i = (i-1)/2;
    }
    tab[i] = v;
    size++;
}
```

Fonction add(int v)

Écrivez la fonction d'ajout ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void add( int v )
{
    int i = size;                    // i : indice nouveau
    while ( i>0 && tab[(i-1)/2]<v )
    {
        tab[i] = tab[(i-1)/2];
        i = (i-1)/2;
    }
    tab[i] = v;
    size++;
}
```

Fonction add(int v)

Écrivez la fonction d'ajout ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void add( int v )
{
    int i = size;                        // i : indice nouveau
    while ( i>0 && tab[(i-1)/2]<v )      // tant que v > père
    {
        tab[i] = tab[(i-1)/2];
        i = (i-1)/2;
    }
    tab[i] = v;
    size++;
}
```

Fonction add(int v)

Écrivez la fonction d'ajout ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void add( int v )
{
    int i = size;                        // i : indice nouveau
    while ( i>0 && tab[(i-1)/2]<v )      // tant que v > père
    {
        tab[i] = tab[(i-1)/2];           // fils <- père
        i = (i-1)/2;
    }
    tab[i] = v;
    size++;
}
```

Fonction add(int v)

Écrivez la fonction d'ajout ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void add( int v )
{
    int i = size;                      // i : indice nouveau
    while ( i>0 && tab[(i-1)/2]<v )    // tant que v > père
    {
        tab[i] = tab[(i-1)/2];         // fils <- père
        i = (i-1)/2;                   // i = indice père
    }
    tab[i] = v;
    size++;
}
```

Fonction add(int v)

Écrivez la fonction d'ajout ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void add( int v )
{
    int i = size;                          // i : indice nouveau
    while ( i>0 && tab[(i-1)/2]<v )    // tant que v > père
    {
        tab[i] = tab[(i-1)/2];         // fils <- père
        i = (i-1)/2;                   // i = indice père
    }
    tab[i] = v;                        // père <- v
    size++;
}
```

Fonction add(Comparable<T> v)

Et avec l'interface Comparable<T> ?

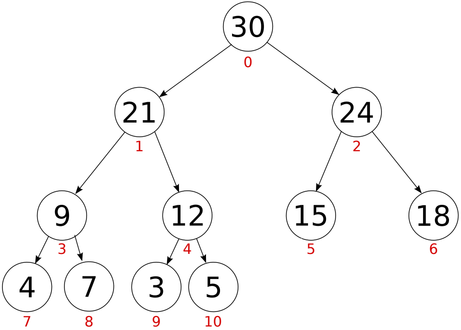
```
private Comparable<T>[] tab;
private int size;
public void add( Comparable<T> v )
{



    // À compléter




}
```

Fonction add(Comparable<T> v)

Et avec l'interface Comparable<T> ?

```
private Comparable<T>[] tab;
private int size;
public void add( Comparable<T> v )
{
    int i = size;
    while ( i>0 && tab[(i-1)/2].compareTo(v)<0 )
    {
        tab[i] = tab[(i-1)/2];
        i = (i-1)/2;
    }
    tab[i] = v;
    size++;
}
```

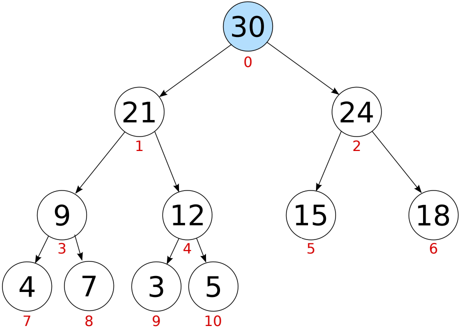
Algorithme et exemple



30	21	24	9	12	15	18	4	7	3	5	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine

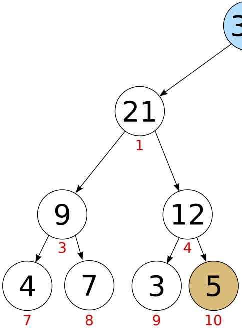
Algorithme et exemple



30	21	24	9	12	15	18	4	7	3	5	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine

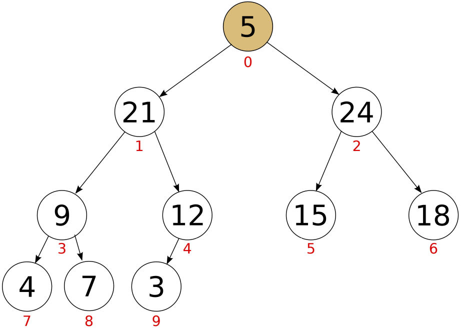
Algorithme et exemple



30	21	24	9	12	15	18	4	7	3	5	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine

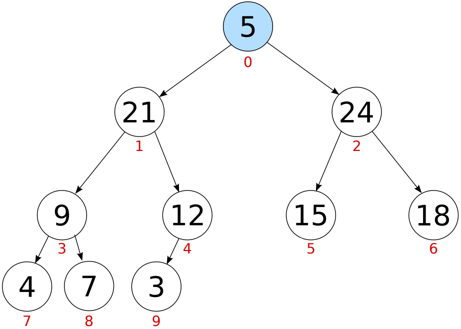
Algorithme et exemple



5	21	24	9	12	15	18	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine

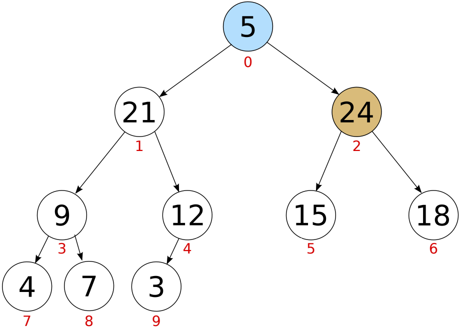
Algorithme et exemple



5	21	24	9	12	15	18	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine
2. **Permuter** N avec son plus grand fils N_F tant que $N < N_F$

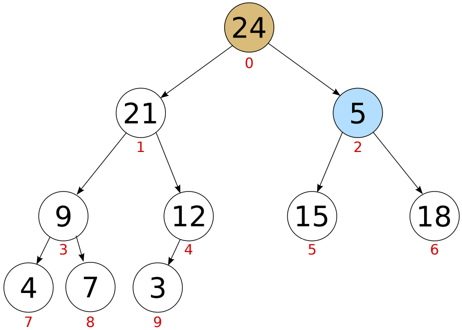
Algorithme et exemple



5	21	24	9	12	15	18	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine
2. **Permuter** N avec son plus grand fils N_F tant que $N < N_F$

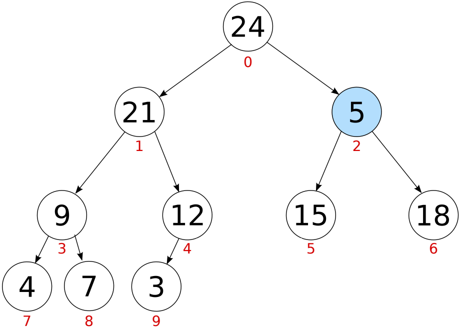
Algorithme et exemple



24	21	5	9	12	15	18	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine
2. **Permuter** N avec son plus grand fils N_F tant que $N < N_F$

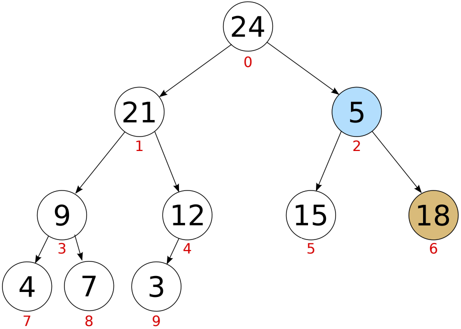
Algorithme et exemple



24	21	5	9	12	15	18	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine
2. **Permuter** N avec son plus grand fils N_F tant que $N < N_F$

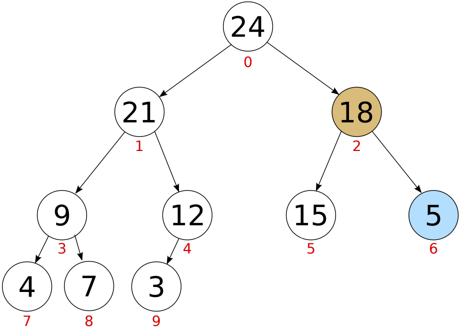
Algorithme et exemple



24	21	5	9	12	15	18	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine
2. **Permuter** N avec son plus grand fils N_F tant que $N < N_F$

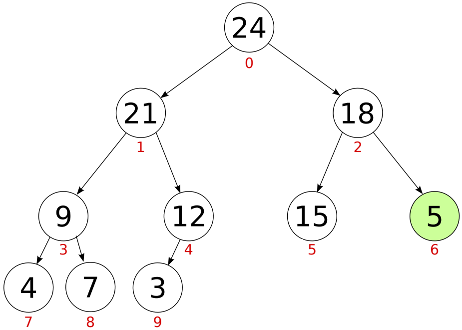
Algorithme et exemple



24	21	18	9	12	15	5	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine
2. **Permuter** N avec son plus grand fils N_F tant que $N < N_F$

Algorithme et exemple



24	21	18	9	12	15	5	4	7	3		
0	1	2	3	4	5	6	7	8	9	10	11

1. **Supprimer** le dernier nœud N et donner sa valeur à la racine
2. **Permuter** N avec son plus grand fils N_F tant que $N < N_F$

Fonction deleteGreatest()

Écrivez la fonction de suppression ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void deleteGreatest()
{





    // À compléter






}
```

Fonction deleteGreatest()

Écrivez la fonction de suppression ci-dessous qui ne considère que des entiers.

```
private int[] tab;
private int size;
public void deleteGreatest()
{
    int v, i, j;
    size--;
    v = tab[0] = tab[size];
    i = 0;
    while ( 2*i+1 < size ) {
        j = 2*i+1;
        if ( j+1 < size && tab[j] < tab[j+1] ) j++;
        if ( v >= tab[j] ) break;
        tab[i] = tab[j];
        i = j;
    }
    tab[i] = v;
}
```

Principe

Le **tri par tas** est un **algorithme de tri** qui utilise un **tas**.

Deux étapes :

Principe

Le **tri par tas** est un **algorithme de tri** qui utilise un **tas**.

Deux étapes :

1. **Créer** le tas en **ajoutant** une par une les données à trier

Principe

Le **tri par tas** est un **algorithme de tri** qui utilise un **tas**.

Deux étapes :

1. **Créer** le tas en **ajoutant** une par une les données à trier
   $\rightarrow$ utiliser `add(int v)`

Principe

Le **tri par tas** est un **algorithme de tri** qui utilise un **tas**.

Deux étapes :

1. **Créer** le tas en **ajoutant** une par une les données à trier
   $\rightarrow$ utiliser `add(int v)`
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

Principe

Le **tri par tas** est un **algorithme de tri** qui utilise un **tas**.

Deux étapes :

1. **Créer** le tas en **ajoutant** une par une les données à trier
   → utiliser `add(int v)`
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide
   → utiliser `deleteGreatest()`

Principe

Le **tri par tas** est un **algorithme de tri** qui utilise un **tas**.

Deux étapes :

1. **Créer** le tas en **ajoutant** une par une les données à trier
   → utiliser `add(int v)`
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide
   → utiliser `deleteGreatest()`

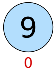
Les valeurs sont supprimées par ordre décroissant :
elles sont triées !

Algorithme et exemple

Triez l'ensemble de valeurs `E` en utilisant l'algorithme de tri par tas :

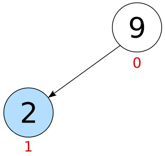
```
E = {9, 2, 11, 7, 4, 14, 3, 16, 8, 10, 15}
```

Algorithme et exemple



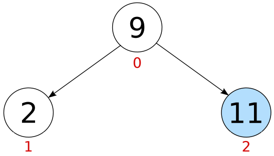
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



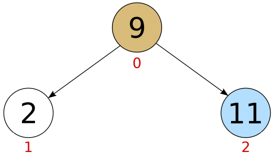
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



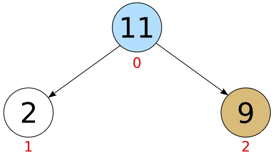
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



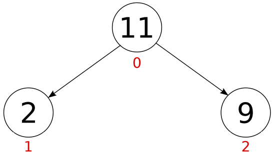
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



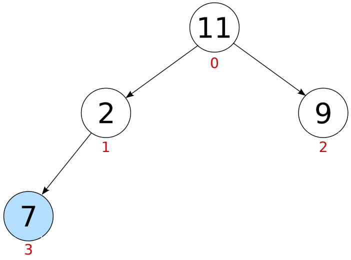
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



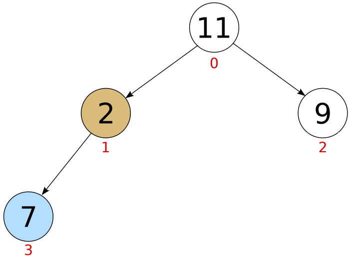
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



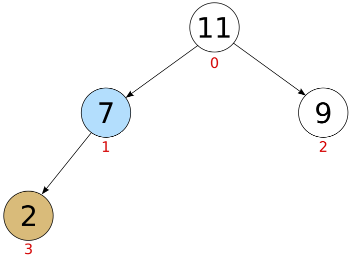
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



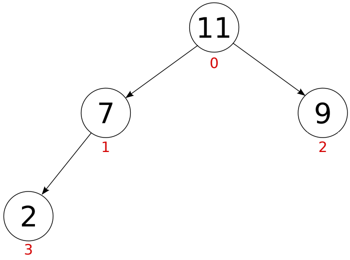
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



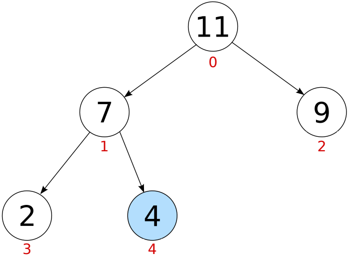
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



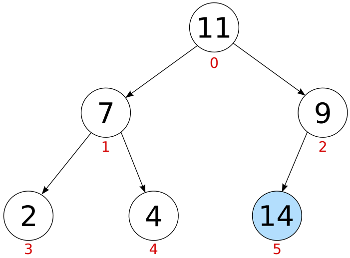
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



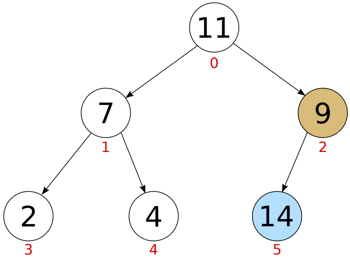
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



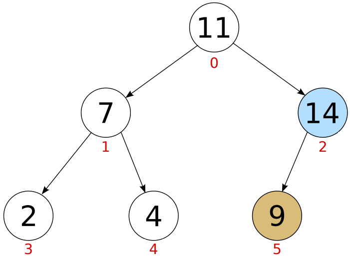
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



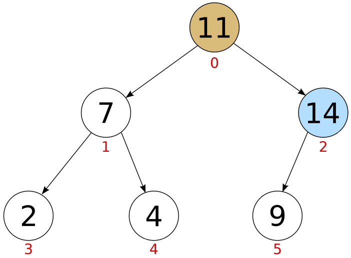
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



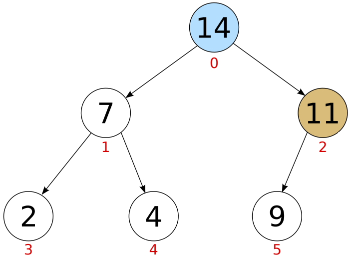
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



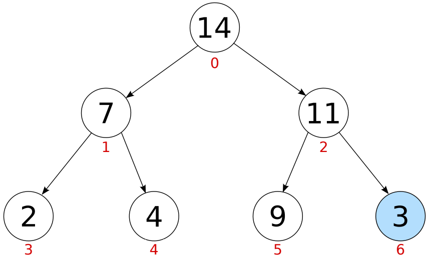
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



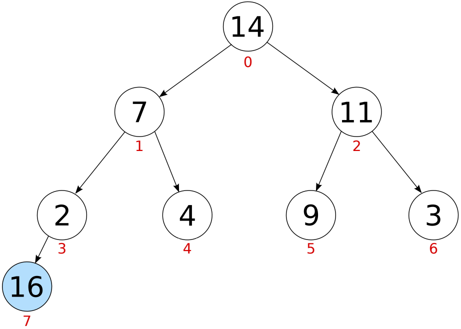
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



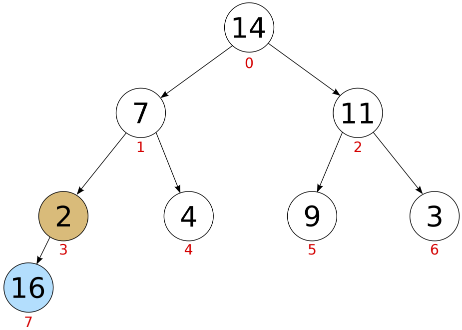
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



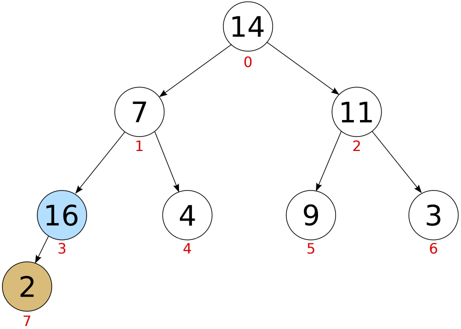
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



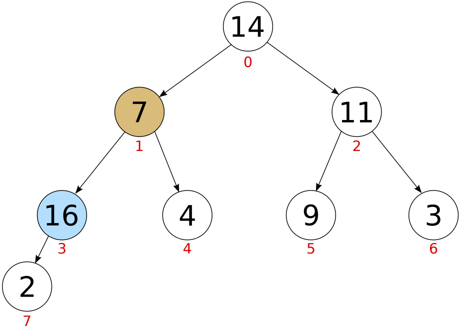
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



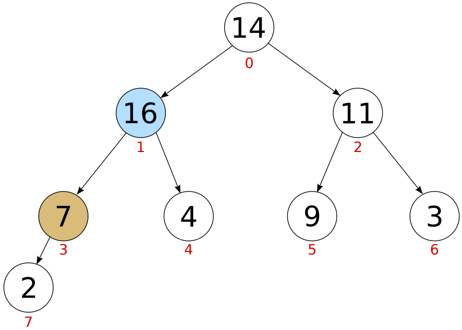
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



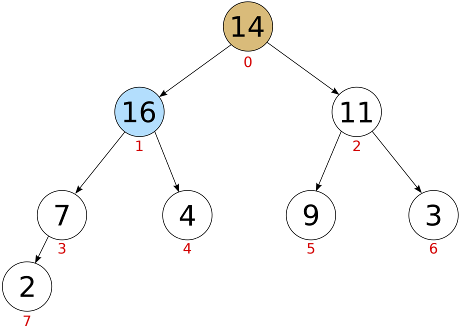
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



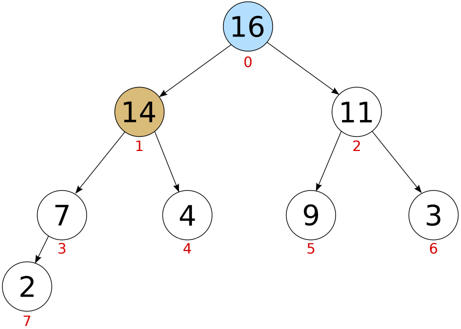
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



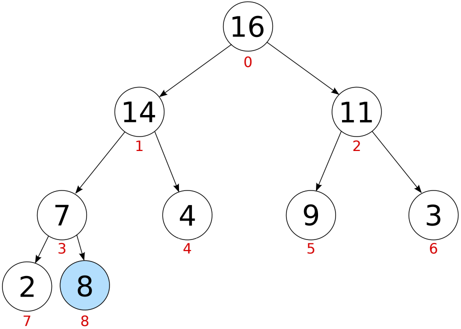
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



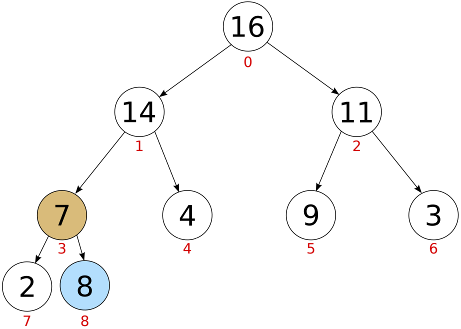
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



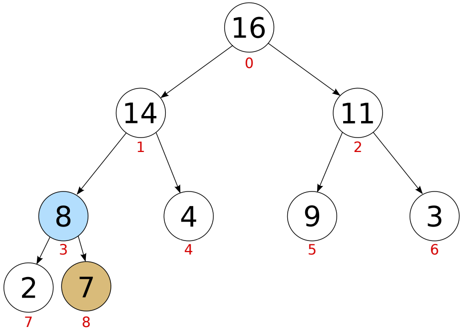
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



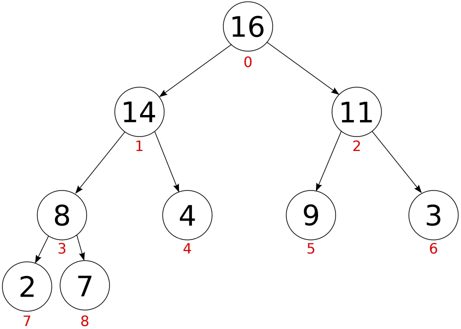
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



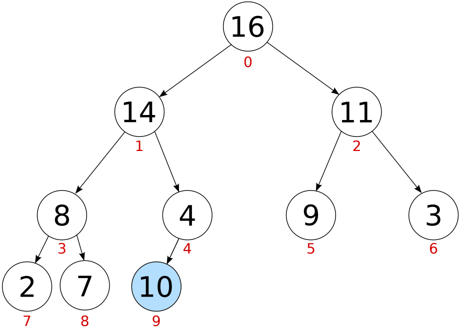
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



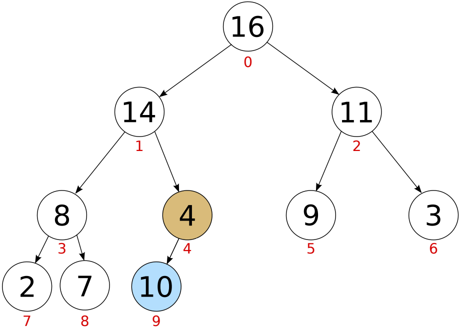
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



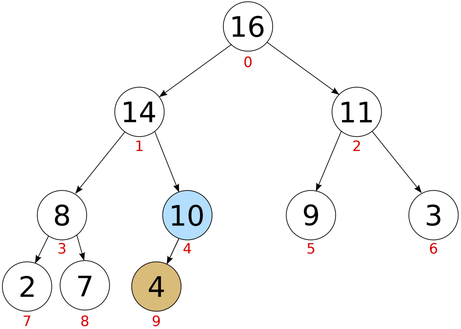
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



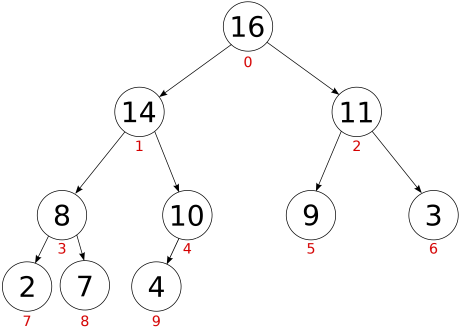
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



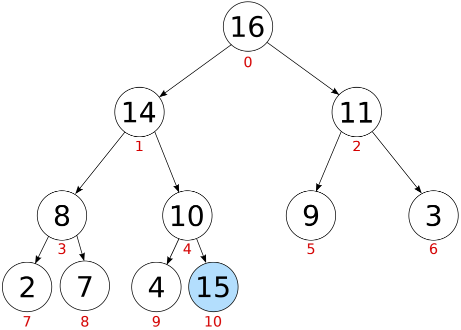
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



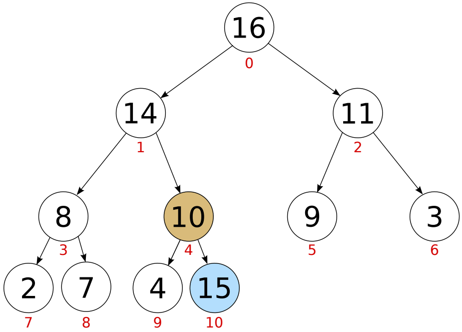
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



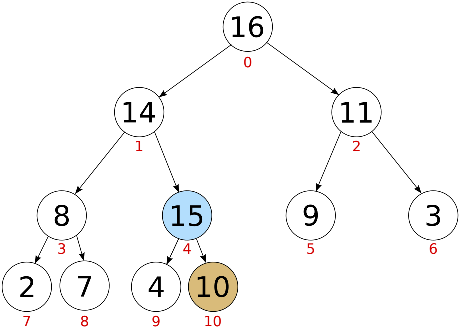
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



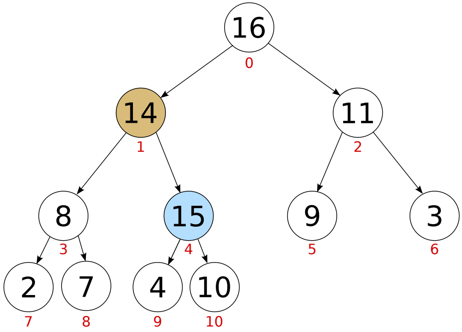
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



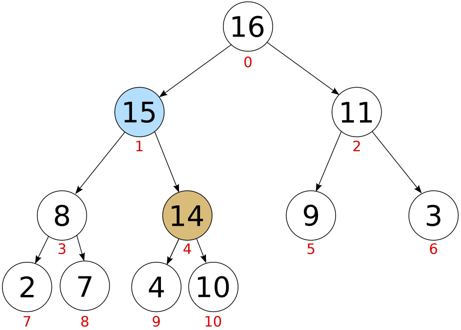
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



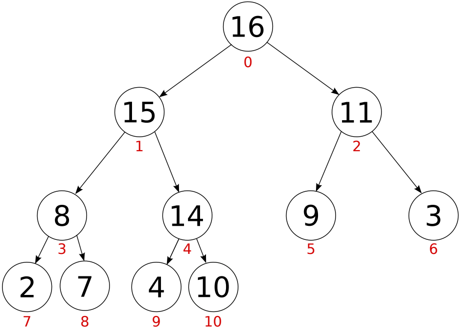
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



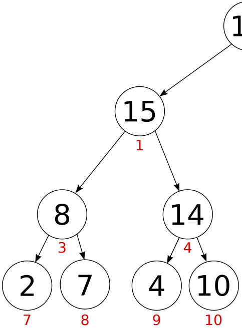
1. **Créer** le tas en **ajoutant** une par une les données à trier

Algorithme et exemple



1. **Créer** le tas en **ajoutant** une par une les données à trier

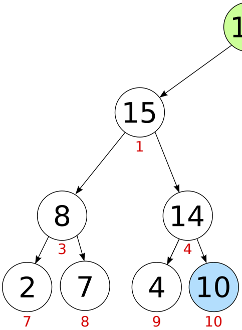
Algorithme et exemple



16	15	11	8	14	9	3	2	7	4	10	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

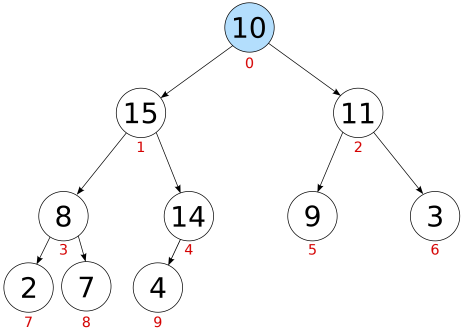
Algorithme et exemple



16	15	11	8	14	9	3	2	7	4	10	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

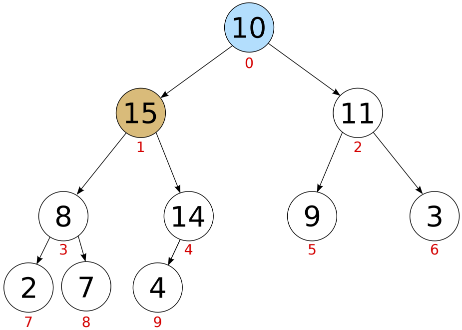
Algorithme et exemple



10	15	11	8	14	9	3	2	7	4	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

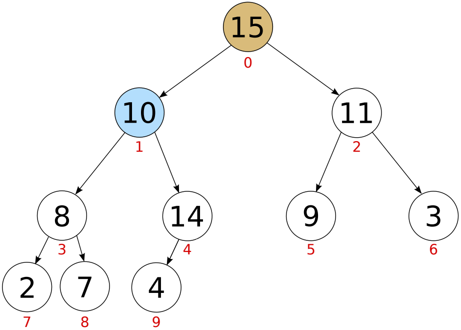
Algorithme et exemple



10	15	11	8	14	9	3	2	7	4	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

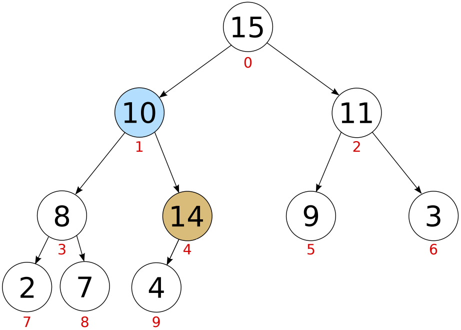
Algorithme et exemple



15	10	11	8	14	9	3	2	7	4	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

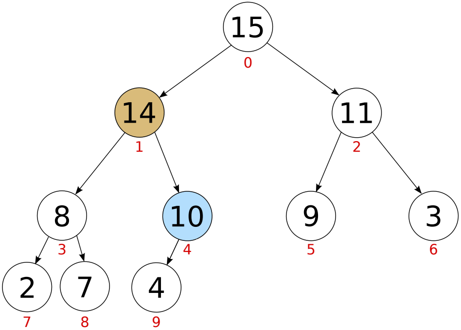
Algorithme et exemple



15	10	11	8	14	9	3	2	7	4	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

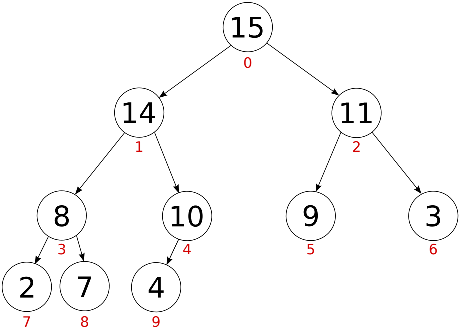
Algorithme et exemple



15	14	11	8	10	9	3	2	7	4	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

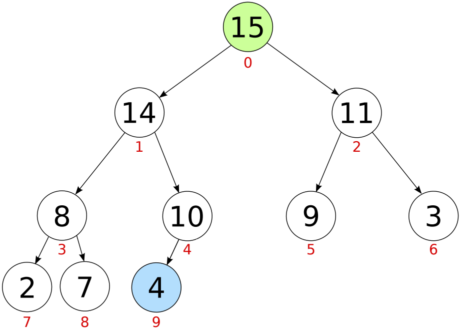
Algorithme et exemple



15	14	11	8	10	9	3	2	7	4	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

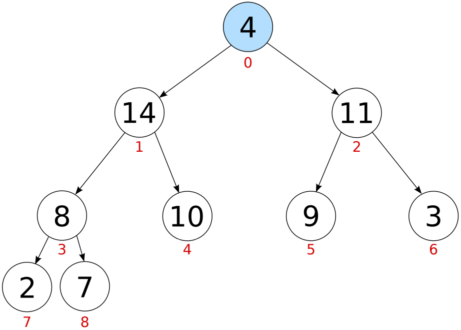
Algorithme et exemple



15	14	11	8	10	9	3	2	7	4	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

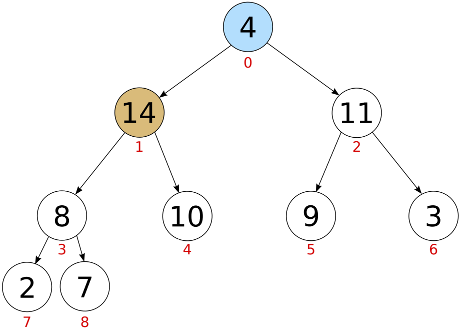
Algorithme et exemple



4	14	11	8	10	9	3	2	7	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

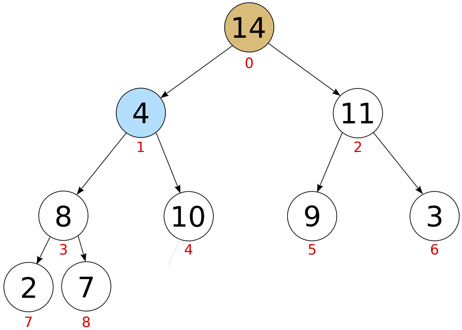
Algorithme et exemple



4	14	11	8	10	9	3	2	7	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

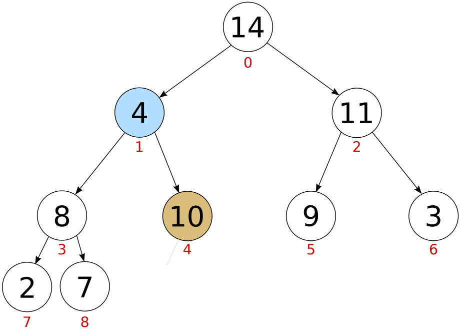
Algorithme et exemple



14	4	11	8	10	9	3	2	7	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

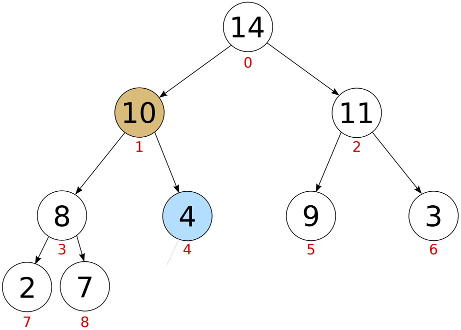
Algorithme et exemple



14	4	11	8	10	9	3	2	7	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

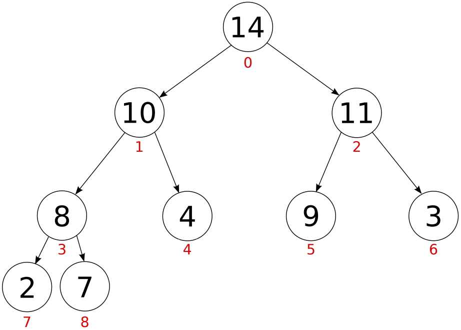
Algorithme et exemple



14	10	11	8	4	9	3	2	7	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

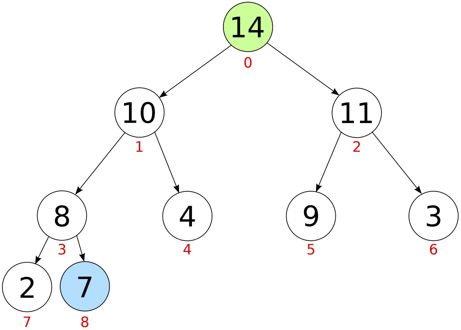
Algorithme et exemple



14	10	11	8	4	9	3	2	7	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

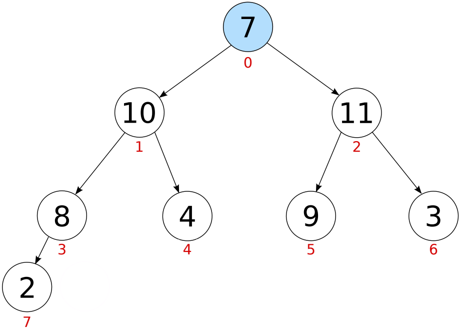
Algorithme et exemple



14	10	11	8	4	9	3	2	7	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

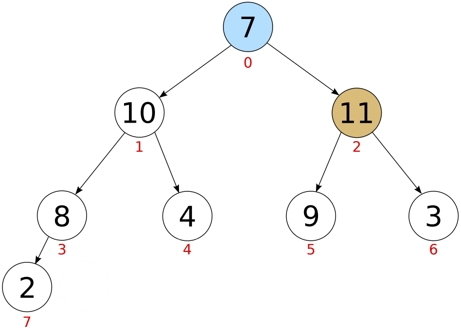
Algorithme et exemple



7	10	11	8	4	9	3	2	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

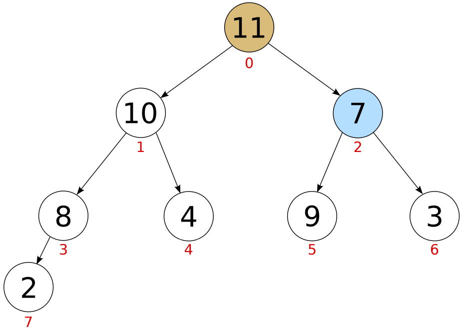
Algorithme et exemple



7	10	11	8	4	9	3	2	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

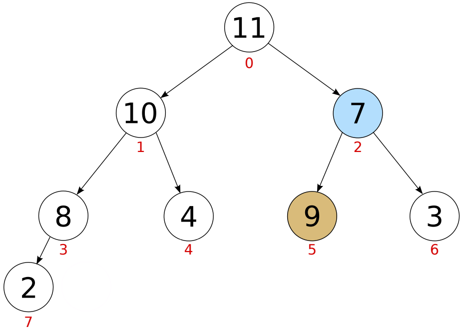
Algorithme et exemple



11	10	7	8	4	9	3	2	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

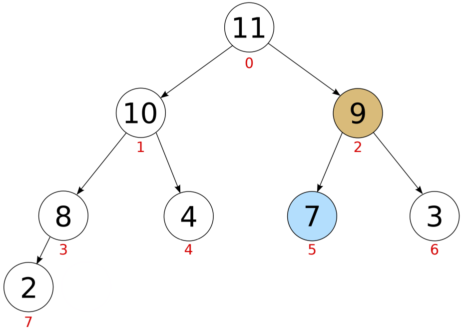
Algorithme et exemple



11	10	7	8	4	9	3	2	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

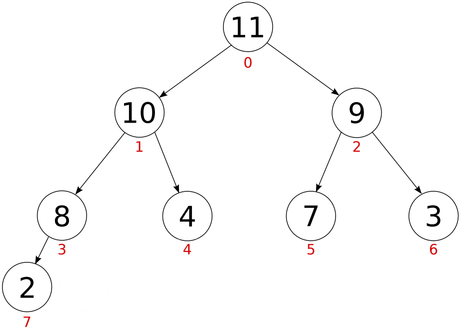
Algorithme et exemple



11	10	9	8	4	7	3	2	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

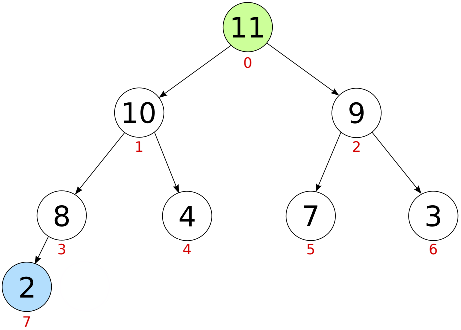
Algorithme et exemple



11	10	9	8	4	7	3	2	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

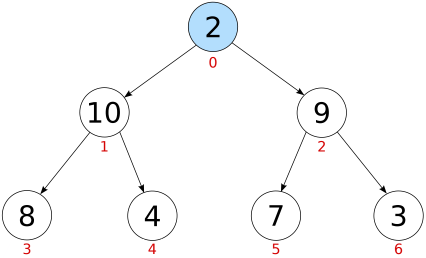
Algorithme et exemple



11	10	9	8	4	7	3	2	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

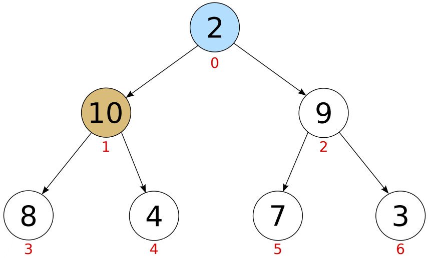
Algorithme et exemple



2	10	9	8	4	7	3	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

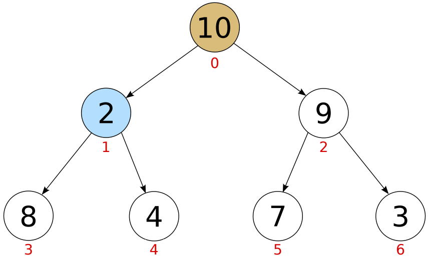
Algorithme et exemple



2	10	9	8	4	7	3	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

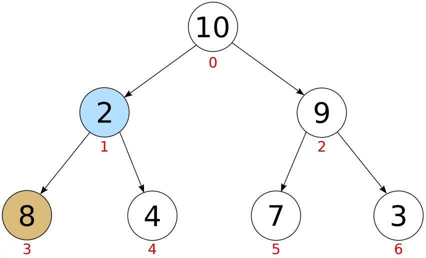
Algorithme et exemple



10	2	9	8	4	7	3	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

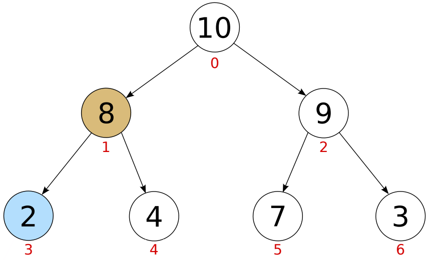
Algorithme et exemple



10	2	9	8	4	7	3	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

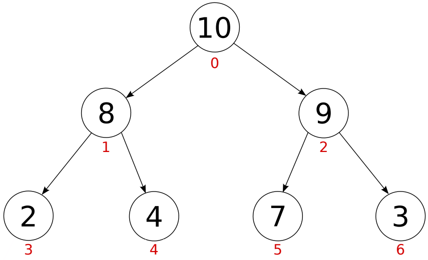
Algorithme et exemple



10	8	9	2	4	7	3	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

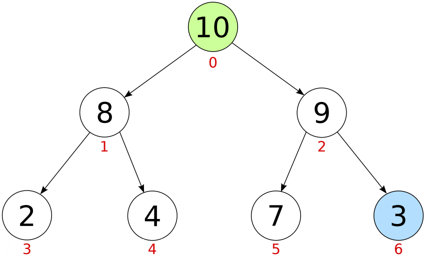
Algorithme et exemple



10	8	9	2	4	7	3	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

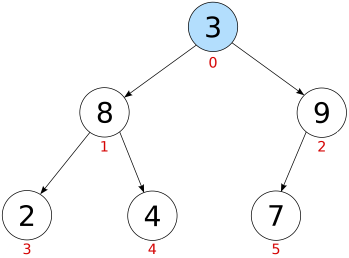
Algorithme et exemple



10	8	9	2	4	7	3	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

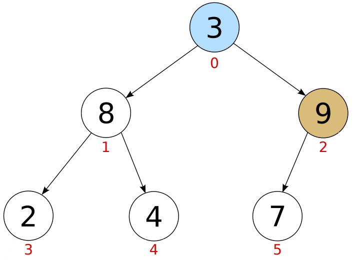
Algorithme et exemple



3	8	9	2	4	7	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

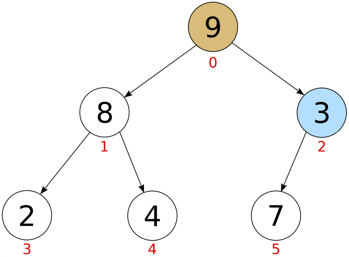
Algorithme et exemple



3	8	9	2	4	7	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

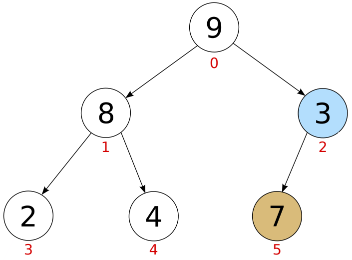
Algorithme et exemple



9	8	3	2	4	7	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

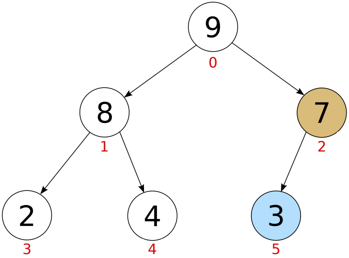
Algorithme et exemple



9	8	3	2	4	7	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

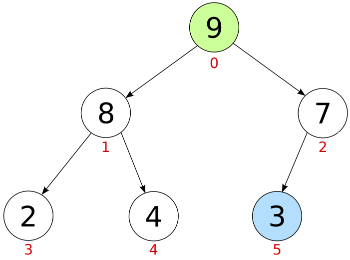
Algorithme et exemple



9	8	7	2	4	3	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

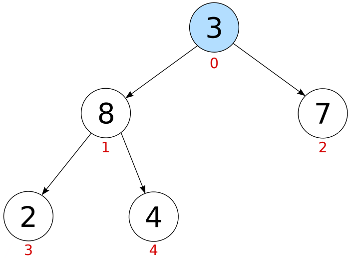
Algorithme et exemple



9	8	7	2	4	3	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

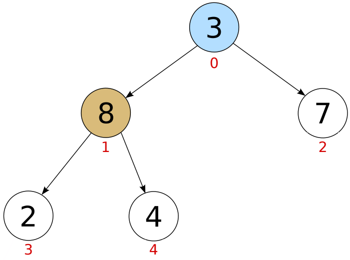
Algorithme et exemple



3	8	7	2	4	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

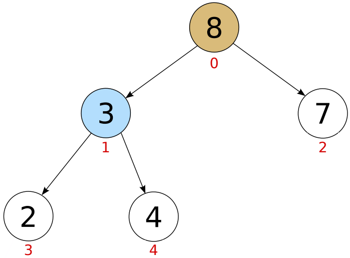
Algorithme et exemple



3	8	7	2	4	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

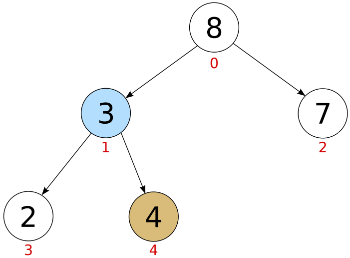
Algorithme et exemple



8	3	7	2	4	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

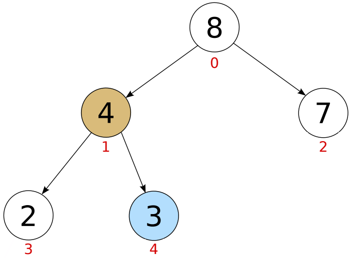
Algorithme et exemple



8	3	7	2	4	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

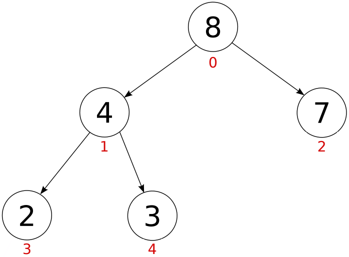
Algorithme et exemple



8	4	7	2	3	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

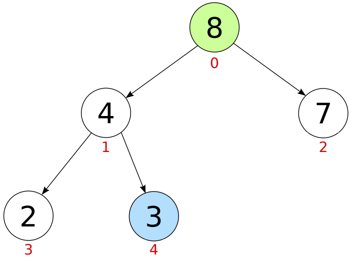
Algorithme et exemple



0	1	2	3	4	5	6	7	8	9	10	11
8	4	7	2	3	9	10	11	14	15	16	

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

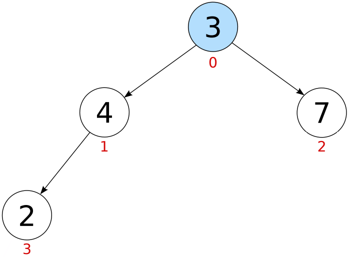
Algorithme et exemple



8	4	7	2	3	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

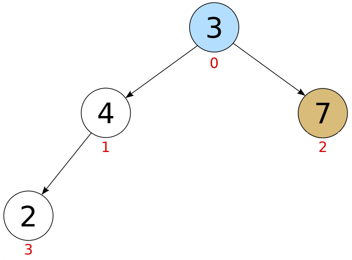
Algorithme et exemple



3	4	7	2	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

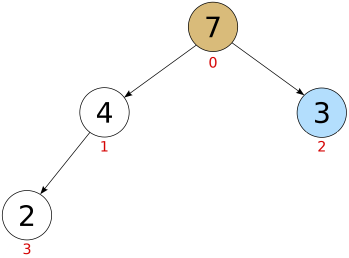
Algorithme et exemple



3	4	7	2	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

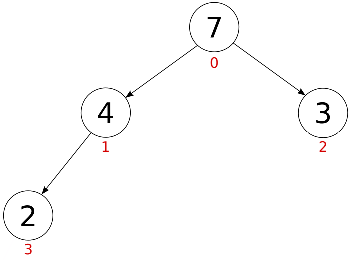
Algorithme et exemple



0	1	2	3	4	5	6	7	8	9	10	11
7	4	3	2	8	9	10	11	14	15	16	

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

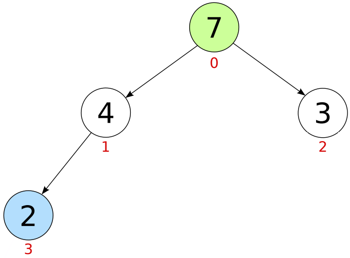
Algorithme et exemple



7	4	3	2	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

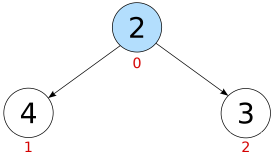
Algorithme et exemple



7	4	3	2	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

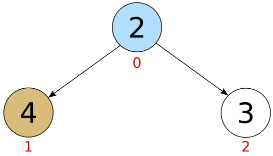
Algorithme et exemple



2	4	3	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

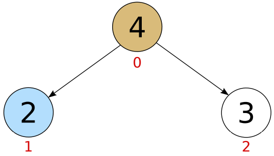
Algorithme et exemple



2	4	3	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

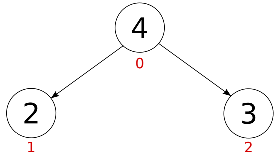
Algorithme et exemple



4	2	3	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

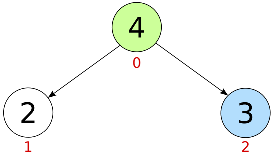
Algorithme et exemple



4	2	3	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

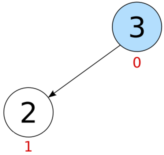
Algorithme et exemple



4	2	3	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

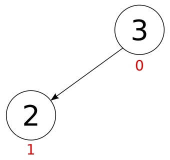
Algorithme et exemple



3	2	4	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

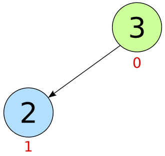
Algorithme et exemple



3	2	4	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

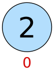
Algorithme et exemple



0	1	2	3	4	5	6	7	8	9	10	11
3	2	4	7	8	9	10	11	14	15	16	

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

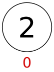
Algorithme et exemple



2	3	4	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

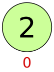
Algorithme et exemple



2	3	4	7	8	9	10	11	14	15	16	
0	1	2	3	4	5	6	7	8	9	10	11

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

Algorithme et exemple



0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	7	8	9	10	11	14	15	16	

1. **Créer** le tas en **ajoutant** une par une les données à trier
2. **Supprimer** la plus grande valeur tant que l'arbre n'est pas vide

Complexité

Le tri par tas a une complexité en $\Theta(n\log_2(n))$, avec n le nombre de données à trier.

Avantages

Inconvénients

Complexité

Le tri par tas a une complexité en $\Theta(n \log_2(n))$, avec n le nombre de données à trier.

Avantages

- Complexité **optimale**
- En place (**pas d'allocation mémoire**)
- **Pas de pire cas** en $\Theta(n^2)$ comme le tri rapide

Inconvénients

Complexité

Le tri par tas a une complexité en $\Theta(n \log_2(n))$, avec n le nombre de données à trier.

Avantages

- Complexité **optimale**
- En place (**pas d'allocation mémoire**)
- **Pas de pire cas** en $\Theta(n^2)$ comme le tri rapide

Inconvénients

- **Plus lent** en moyenne que le tri rapide

Complexité

Le tri par tas a une complexité en $\Theta(n \log_2(n))$, avec n le nombre de données à trier.

Avantages

- Complexité **optimale**
- En place (**pas d'allocation mémoire**)
- **Pas de pire cas** en $\Theta(n^2)$ comme le tri rapide

Inconvénients

- **Plus lent** en moyenne que le tri rapide

Pour aller plus loin

Le **tri introspectif** (Introsort) est une amélioration du tri rapide qui utilise le tri par tas.

Plan

3.24pt

1. Présentation des arbres
   - Vocabulaire
   - Les arbres binaires
   - Les ABR
2. Spécification algébrique des arbres binaires
   - Opérations
   - Axiomes
3. Spécification algébrique des ABR
   - Opérations
   - Axiomes
4. Limite des ABR
   - Illustration
   - Solution

3.24pt

5. Les arbres AVL
   - Présentation
   - Les poids
   - Insertion
   - Mise en pratique
6. Les tas
   - Présentation
   - Ajout
   - Suppression
   - Heapsort
7. Implantations Java
   - Arbre binaire complet
   - Arbre binaire quelconque
8. Complexités

Choix d'implémentation

FullBinaryTree<T>
#tab: T[]
#size: int

Un **arbre binaire complet** peut être représenté par un **tableau**.

Avantages

Inconvénients

Choix d'implémentation

FullBinaryTree<T>
`#tab: T[]`
`#size: int`

Un **arbre binaire complet** peut être représenté par un **tableau**.

Avantages

- Place mémoire
- Accès rapides

Inconvénients

- Ajouts plus complexes

Une seule classe

BinaryTree<T>

`#value: T`
`#left: BinaryTree<T>`
`#right: BinaryTree<T>`

Un **arbre binaire quelconque** peut être représenté par **une seule classe** “récursive”.

Avantages

Inconvénients

Une seule classe

BinaryTree<T>

`#value: T`
`#left: BinaryTree<T>`
`#right: BinaryTree<T>`

Un **arbre binaire quelconque** peut être représenté par **une seule classe** “récursive”.

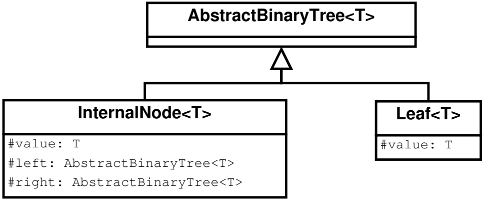
Avantages

- Place mémoire
- Ceux de la récursivité

Inconvénients

- Place mémoire
- Représentation de l’arbre vide

Plusieurs classes

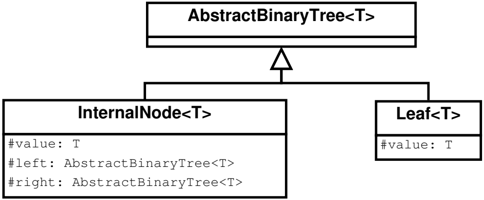


Un **arbre binaire quelconque** peut être représenté par **plusieurs classes** représentant les différents types de nœuds.

Avantages

Inconvénients

Plusieurs classes



Un **arbre binaire quelconque** peut être représenté par **plusieurs classes** représentant les différents types de nœuds.

Avantages

- Possibilité d'opérations spécifiques
- Utilisation de la liaison dynamique

Inconvénients

- Ajout/suppression avec retour de type abstrait
- Le type d'un nœud peut changer

Plan

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire

Accès, ajout, suppression au pire cas

en moyenne

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	au pire cas
	en moyenne

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(h(a))$ au pire cas
	en moyenne

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(h(a))$ au pire cas
	$\Theta(\log_2 N)$ en moyenne

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(h(a))$ au pire cas
	$\Theta(\log_2 N)$ en moyenne

avec $\log_2 N \leq h(a) \leq N - 1$

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(h(a))$ au pire cas
	$\Theta(\log_2 N)$ en moyenne

avec $\log_2 N \leq h(a) \leq N - 1$

AVL

Cout mémoire	
Accès, ajout, suppression	au pire cas
	en moyenne

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(h(a))$ au pire cas
	$\Theta(\log_2 N)$ en moyenne

avec $\log_2 N \leq h(a) \leq N - 1$

AVL

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	au pire cas
	en moyenne

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(h(a))$ au pire cas
	$\Theta(\log_2 N)$ en moyenne

avec $\log_2 N \leq h(a) \leq N - 1$

AVL

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(\log_2 N)$ au pire cas
	en moyenne

Complexité ABR vs. AVL

Soit un arbre a de N nœuds stockant des objets de taille T_N.

ABR

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(h(a))$ au pire cas
	$\Theta(\log_2 N)$ en moyenne

avec $\log_2 N \leq h(a) \leq N - 1$

AVL

Cout mémoire	$N \times (T_N + 2)$
Accès, ajout, suppression	$\Theta(\log_2 N)$ au pire cas
	$\Theta(\log_2 N)$ en moyenne