ITI 1521. Introduction à l'informatique II

Arbres binaires de recherche : méthodes

by

Marcel Turcotte

Version du 27 mars 2020

Préambule

Préambule

Aperçu

Aperçu

Arbres binaires de recherche : méthodes

Nous discutons des propriétés des arbres : arbre binaire plein, arbre complet, profondeur maximale d'un arbre balancé de taille n. Finalement, nous implémentons les méthodes pour ajouter et retirer un élément d'un arbre binaire de recherche.

Objectif général :

- Cette semaine, vous serez en mesure de concevoir et modifier des programmes informatiques reposant sur le concept d'arbre binaire de recherche.

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Discuter** l'efficacité du traitement récursif des arbres en Java, notamment par rapport à la consommation de mémoire.
- **Modifier** l'implémentation d'un arbre binaire de recherche afin d'y ajouter une nouvelle méthode.

Lectures :

- Pages 263, 265-268, 288-293 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

Résumé

Résumé

- Un arbre est une structure de données **hiérarchique**
- On peut implémenter un arbre à l'aide de noeuds **chaînés**
- Traitements **récursifs** et **itératifs** :

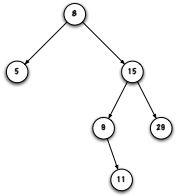
  Itératif : Une méthode qui suit **un et un seul chemin** dans l'arbre peut facilement être implémentée à l'aide d'une méthode itérative.

  Récursif : Une méthode qui doit **parcourir plus d'un sous-arbre pour un même noeud** est généralement implémentée plus facilement à l'aide d'une méthode récursive.

Définition

Un **arbre binaire de recherche** est un arbre binaire dont chaque noeud vérifie les deux propriétés suivantes :

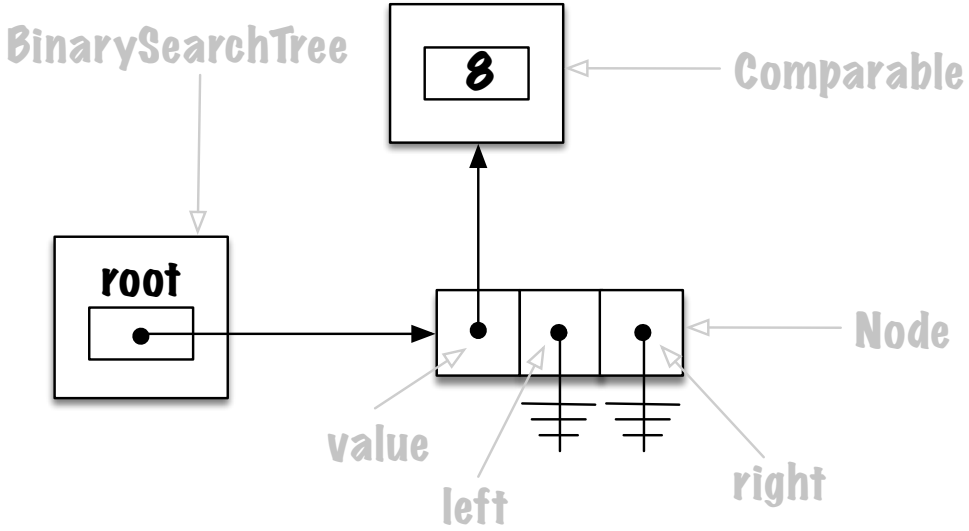
- Tous les noeuds de son sous-arbre **gauche** ont des valeurs **plus petites** que celle de ce noeud ou son sous-arbre gauche est vide ;
- Tous les noeuds de son sous-arbre **droit** ont des valeurs **plus grandes** que celle de ce noeud ou son sous-arbre droit est vide.

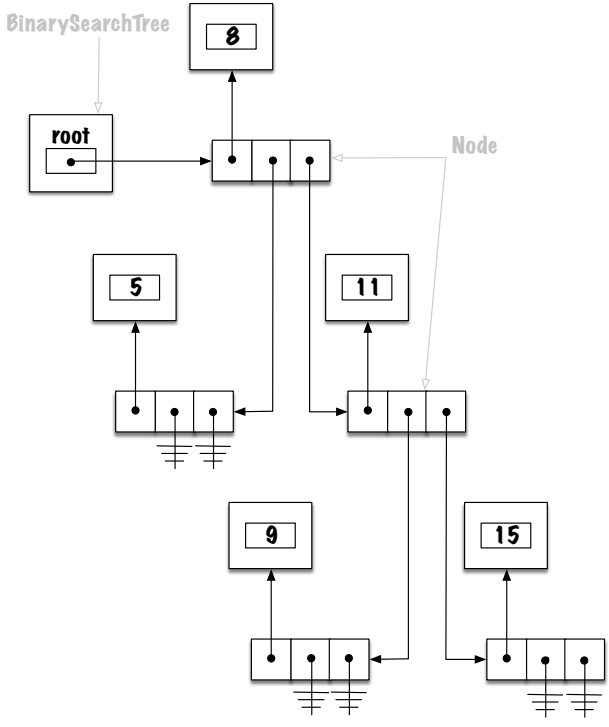


Implémentation des arbres binaires de recherche

```
public class BinarySearchTree<E extends Comparable<E>> {

    private static class Node<T> {
        private T value;
        private Node<T> left;
        private Node<T> right;
    }

    private Node<E> root;

}
```

Diagramme de mémoire



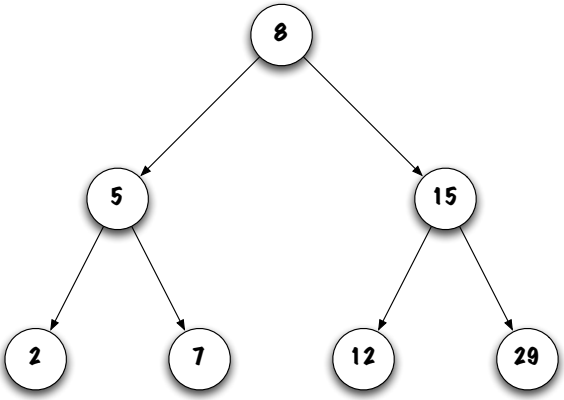

BinarySearchTree
8
root
Node
5
11
9
15

Définitions

Définitions

Arbre binaire plein

Arbre binaire plein



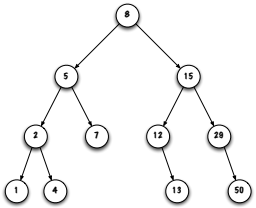
On dit qu'un arbre binaire est **plein** si tous ses noeuds ont exactement deux fils, à l'exception des feuilles.

Définitions

Arbre complet

Arbre complet

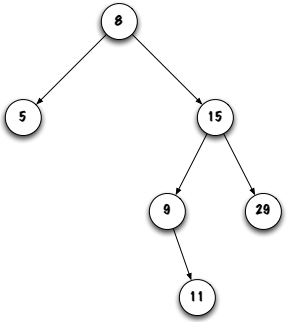
Un arbre binaire de profondeur d est **complet** si tous ses noeuds à profondeur moins de $d-1$ (donc dans l'intervalle $[0, 1 \ldots d-2]$) ont exactement deux fils.



- Est-ce que cet arbre **est complet** ?
  - **Oui**, la profondeur de l'arbre est $d = 3$, tous les noeuds aux profondeurs 0 et 1 ($\leq d-2$) ont exactement deux fils. Les noeuds à la profondeur 2 ont 0, 1 ou 2 fils. Tous les noeuds à profondeur 3 sont des feuilles.

Arbre complet

- Cet arbre **est-il complet** ?

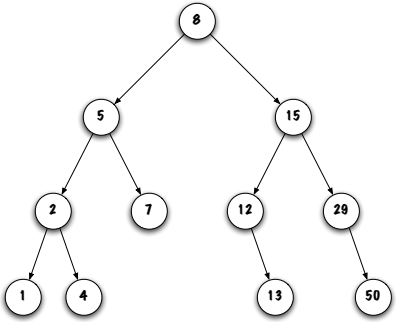


Non, la profondeur de l'arbre est $d = 3$, le noeud 5 à la profondeur 1 ($\leq d - 2$) n'a pas deux fils.

Définitions

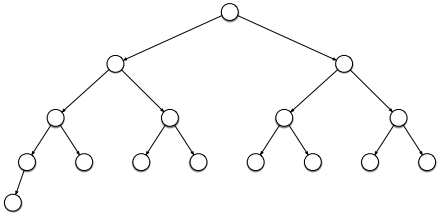
Profondeur maximale

Relation entre la profondeur et le nombre de noeuds

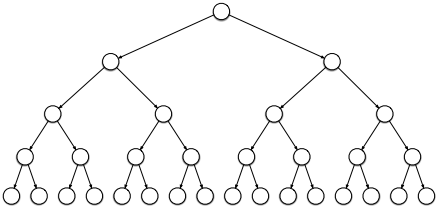


- Un arbre binaire complet de profondeur d a de 2^d à $2^{d+1} - 1$ noeuds ;
- La profondeur d'un arbre binaire complet de taille n est $\lfloor \log_2 n \rfloor$.

Relation entre la profondeur et le nombre de noeuds

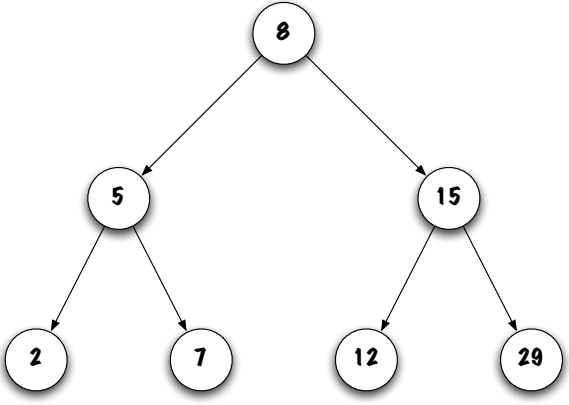


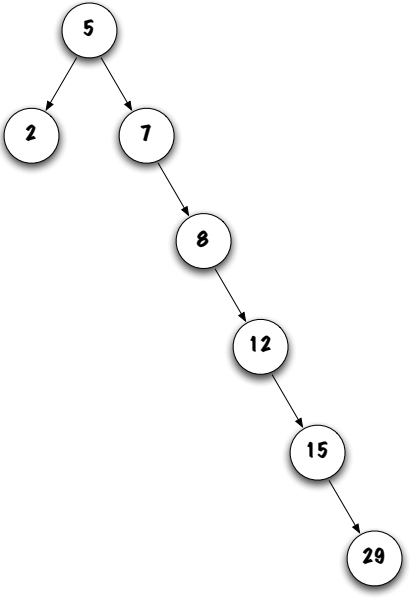
...

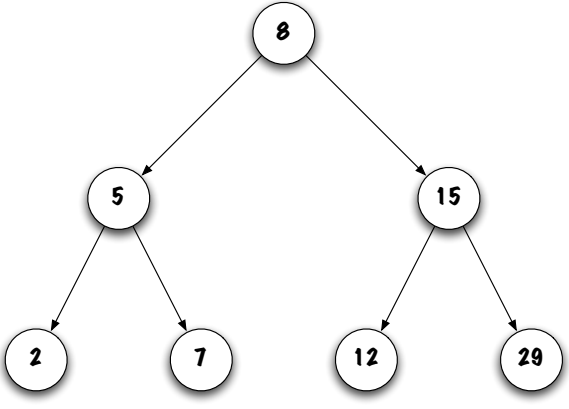


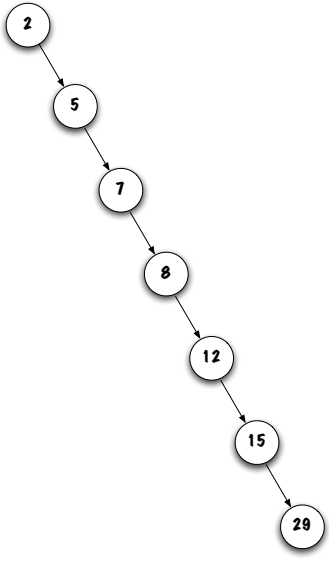
Discussion

- Quelle relation existe-t-il entre l'**efficacité** des méthodes et la **topologie** de l'arbre (complet ou pas).


8
5
15
2
7
12
29


5
2
7
8
12
15
29


8
5
15
2
7
12
29


2
5
7
8
12
15
29

Observations

- Lors de la recherche, **chaque comparaison élimine un sous-arbre** ;
- Le **nombre maximum de noeuds** visités dépend de la **profondeur** de l'arbre ;
- Ainsi, les arbres complets sont avantageux (puisque la profondeur de l'arbre est $\lfloor \log_2 n \rfloor$) *.

*Pour le cas extrême ou l'arbre est complètement débalancé, il faudrait traverser $n - 1$ liens.

n	$\lfloor \log_2 n \rfloor$
10	3
100	6
1,000	9
10,000	13
100,000	16
1,000,000	19
10,000,000	23
100,000,000	26
1,000,000,000	29
10,000,000,000	33
100,000,000,000	36
1,000,000,000,000	39
10,000,000,000,000	43
100,000,000,000,000	46
1,000,000,000,000,000	49

Préfixe du système international d'unités

Préfixe	n	$\lfloor \log_2 n \rfloor$
méga	10^6	19
giga	10^9	29
téra	10^{12}	39
péta	10^{15}	49
exa	10^{18}	59
zetta	10^{21}	69
yotta	10^{24}	79

- Consultez **How much data is generated each day?** par Jeff Desjardins dans *World Economic Forum* le 17 avril 2019.

Expérience

Machine :
- 64 coeurs = Intel(R) Xeon(R) CPU E7-4870 v2 2.30GHz
- RAM = 512 Giga-octets
- Système d'exploitation = Linux

Expérience :
- Temps moyen sur 5 exécutions de l'appel **add** sur un arbre contenant 1.000.000.000 (10^9) d'éléments
- 5,765 nono-secondes, 5.765 micro-secondes, 0.005765 milli-secondes

```
> java -Xmx256g TestBST 1000000000
```

La construction de l'arbre prend 3.1348975 heures.

add

boolean add(E element)

Exercice. À partir d'un arbre vide, ajoutez un à un les éléments suivants : «Lion», «Fox», «Rat», «Cat», «Pig», «Dog», «Tiger».

- Quelles **conclusions** tirez-vous ?

- Afin d'ajouter un élément, il faut **trouver l'endroit où l'insérer**.
- **Quelle méthode** permet de trouver un élément ?
  - C'est la méthode **contains**.
- Quels sont les **changements** à apporter ?

```
public boolean contains(E element) {
    boolean found = false;
    Node<E> current = root;
    while (! found && current != null) {
        int test = element.compareTo(current.value);
        if (test == 0) {
        found = true;
        } else if (test < 0) {
            current = current.left;
        } else {
            current = current.right;
        }
    }
    return found;
}
```

boolean add(E element)

- Y a-t-il un **cas spécial** à traiter ?
  - Les traitements impliquant un changement de la variable **root** sont des cas spéciaux, tout comme les changements de la variable **head** pour une liste chaînée.

```
if (current == null) {
    root = new Node<E>(element);
}
```

Sinon.

```
boolean done = false;
while (! done) {
    int test = element.compareTo(current.value);
    if (test == 0) {
        done = true;
    } else if (test < 0) {
        if (current.left == null) {
            current.left = new Node<E>(element);
            done = true;
        } else {
            current = current.left;
        }
    } else {
        if (current.right == null) {
            current.right = new Node<E>(element);
            done = true;
        } else {
            current = current.right;
        }
    }
}
```

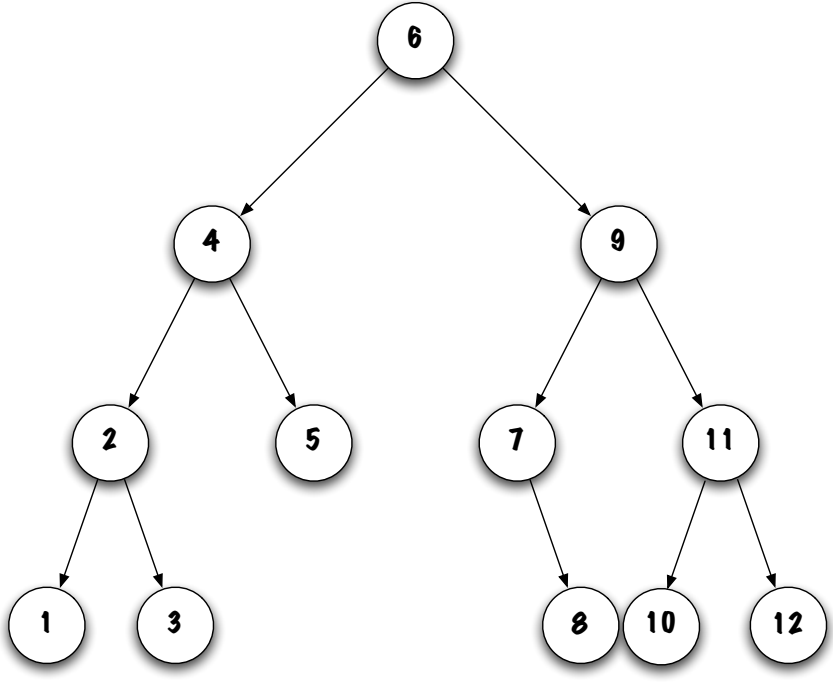
boolean add(E element)

- On remplace toujours une valeur **null** par un nouveau noeud ;
- La **structure** existante de l'arbre **ne change pas** ;
- La **topologie** de l'arbre dépend largement de l'**ordre dans lequel les éléments sont insérés**.

remove

boolean remove(E element)

- Les **retraits** entraîneront forcément des **changements de structure**.
- **Explorez** différentes stratégies à l'aide de l'arbre se trouvant à la page qui suit.
  - **Éliminez** chacun des 12 noeuds, un à un.


6
4
9
2
5
7
11
1
3
8
10
12

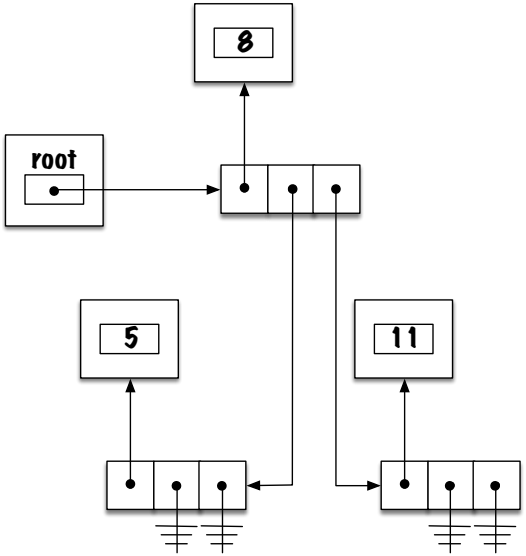
boolean remove(E element)

Considérez certains **cas spécifiques** :

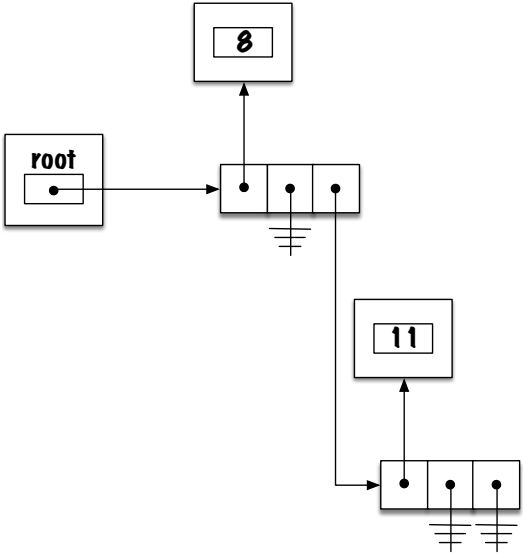
- Retirer le noeud le **plus à gauche**.
  - **Combien de sous-cas** y a-t-il et quels sont-ils ?
  - Il y a **deux** sous-cas :
    - Le noeud n'a **pas de sous-arbres** ;
    - Le noeud **1** du sous-arbre **6** est un exemple ;
    - Que fait-on ? **parent.left = null** ;
    - Le noeud a un **sous-arbre droit** ;
    - Le noeud **7** du sous-arbre **9** est un exemple ;
    - Que fait-on ? **parent.left = «sous arbre droit»** ;
    - **Le noeud ne peut avoir un sous-arbre gauche, sinon, ce n'est pas le noeud le plus à gauche !**

Cas 1 : retirer une feuille

Avant :

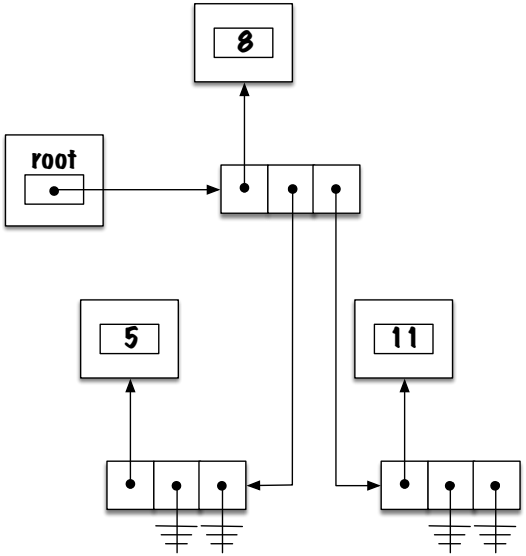


Après :

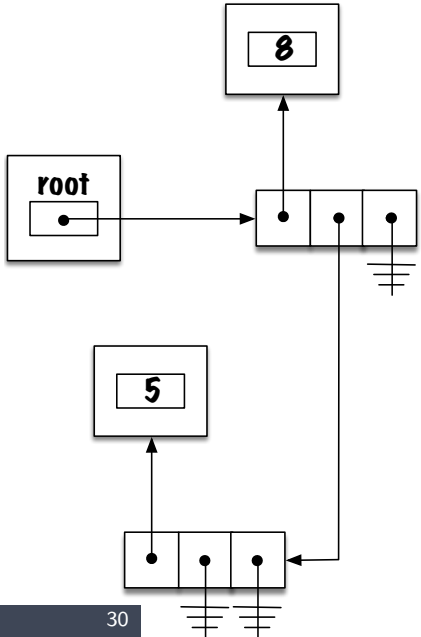


Cas 1 : retirer une feuille

Avant :

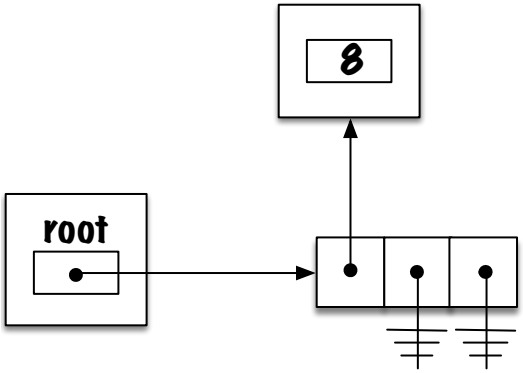


Après :

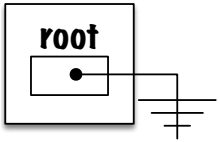


Cas 1 : retirer une feuille

Avant :

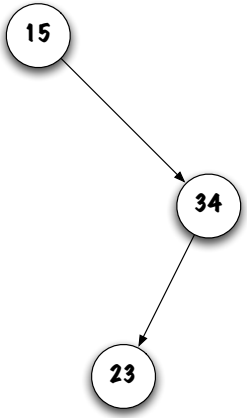


Après :

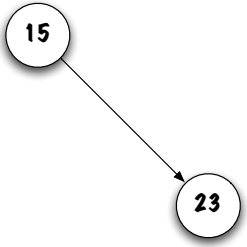


Cas 2 : t.remove(new Integer(34))

Avant :

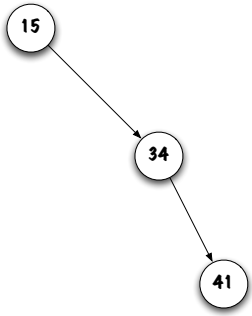


Après :

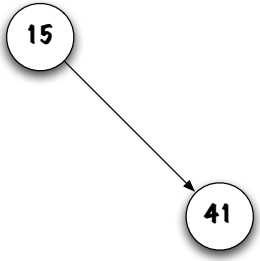


Cas 3 : t.remove(new Integer(34))

Avant :

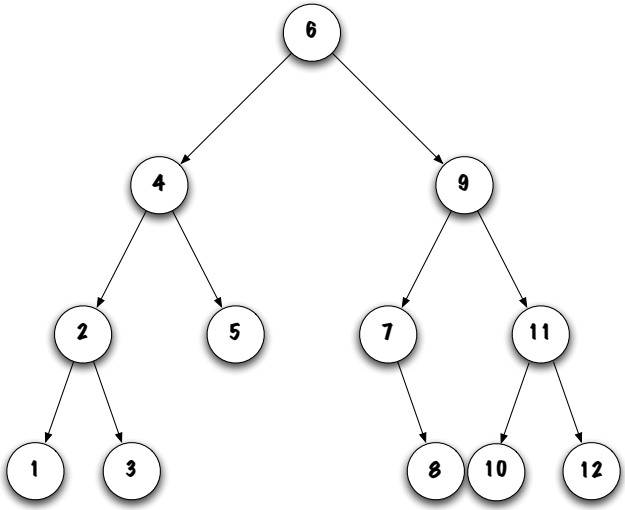


Après :



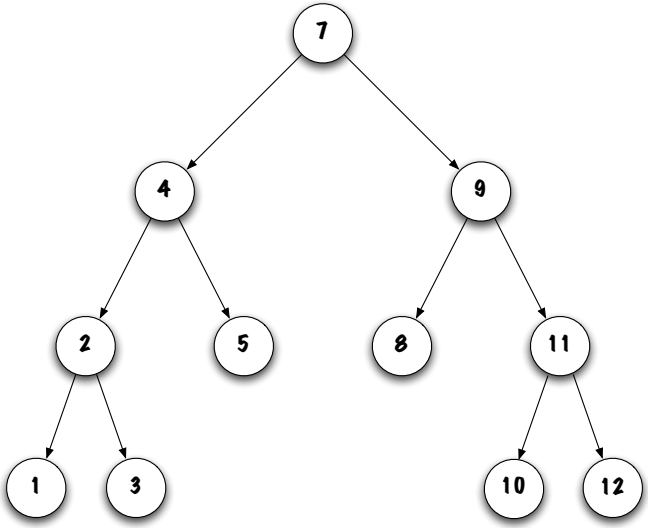
Cas 4 : t.remove(new Integer(6))

Avant :



Cas 4 : t.remove(new Integer(6))

Après :



Node<E> remove(E element)

```
// pre-condition:

if (element == null) {
    throw new NullPointerException;
}

if (root == null) {
    throw new NoSuchElementException();
}
```

Node<E> remove(E element)

Remplacer le noeud à la **racine** de l'arbre est un **cas spécial**.

```
if (element.compareTo(root.value) == 0) {

    root = removeTopMost(root);

}
```

Node<E> remove(E element)

element n'est pas à la racine de l'arbre

```
} else {

    Node<E> current, parent = root;

    if (element.compareTo(root.value) < 0) {
        current = root.left;
    } else {
        current = root.right;
    }

    // ...
```

```
    // ...
    while (current != null) {
        int test = element.compareTo( current.value );
        if (test == 0) {
            if (current == parent.left) {
                parent.left = removeTopMost(current);
            } else {
                parent.right = removeTopMost(current);
            }
            current = null; // stopping criteria
        } else {
            parent = current;
            if (test < 0) {
                current = parent.left;
            } else {
                current = parent.right;
            }
        }
    }
```

Node<E> removeTopMost(Node<E> current)

```
private Node<E> removeTopMost(Node<E> current) {

    Node<E> top;

    if (current.left == null) {
        top = current.right;
    } else if (current.right == null) {
        top = current.left;
    } else {
        current.value = getLeftMost(current.right);
        current.right = removeLeftMost(current.right);
        top = current;
    }

    return top;
}
```

E getLeftMost(Node<E> current)

```
private E getLeftMost(Node<E> current) {

    if (current == null) {
        throw new NullPointerException();
    }

    if (current.left == null) {
        return current.value;
    }

    return getLeftMost(current.left);
}
```

Node<E> removeLeftMost(Node<E> current)

```
private Node<E> removeLeftMost(Node<E> current) {

    if (current.left == null) {
        return current.right;
    }
    Node<E> top = current, parent = current;
    current = current.left;

    while (current.left != null) {
        parent = current;
        current = current.left;
    }

    parent.left = current.right;
    return top;
}
```

Implémentation récursive

```
public void remove(E element) {

    // pre-condition:
    if (element == null) {
        throw new NullPointerException();
    }

    root = remove(root, element);
}
```

```
private Node<E> remove(Node<E> current, E element) {
    Node<E> result = current;
    int test = element.compareTo(current.value);
    if (test == 0) {
        if (current.left == null) {
            result = current.right;
        } else if (current.right == null) {
            result = current.left;
        } else {
            current.value = getLeftMost(current.right);
            current.right = remove(current.right, current.value);
        }
    } else if (test < 0) {
        current.left = remove(current.left, element);
    } else {
        current.right = remove(current.right, element);
    }
    return result;
}
```

Observations

- Il existe une très grande variété d'arbres, dont les **arbres auto balancés** (AVL, Rouge-Noire, B).
- Un **arbre général** est un arbre dont les noeuds peuvent avoir plus de deux fils.

Résumé

- L'**arbre binaire de recherche** est un arbre binaire tel quel toutes les clés du sous-arbre **gauche** sont **inférieures** à celle du noeud courant et toutes les clés du sous-arbre **droit** sont **supérieures** ;
- Une telle structure de données permet des recherches efficaces.

Prologue

Résumé

- La **topologie** de l'arbre dépend de l'ordre dans lequel les éléments sont ajoutés.
- Si l'arbre est **complet** et qu'il contient **n** éléments, il faudra suivre au plus $\lfloor \log_2 n \rfloor$ liens pour trouver l'élément recherché.

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa