ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 26 mars 2011

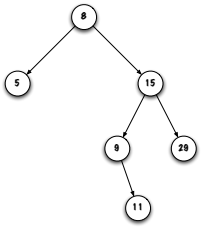
Résumé

– Arbre binaire de recherche (partie 2)

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Arbre binaire de recherche

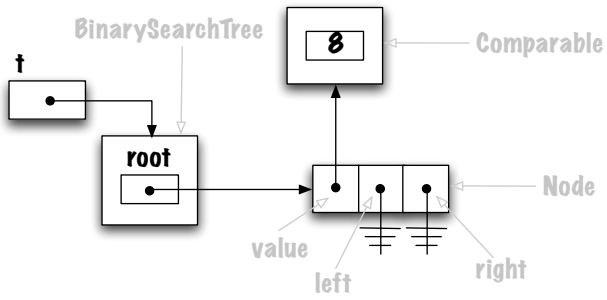
Un **arbre binaire de recherche** est un arbre binaire dont chaque noeud vérifie les deux propriétés suivantes :
– Tous les noeuds de son sous-arbre gauche ont des valeurs plus petites que celle de ce noeud ou son sous-arbre gauche est vide ;
– Tous les noeuds de son sous-arbre droit ont des valeurs plus grandes que celle de ce noeud ou son sous-arbre droit est vide.

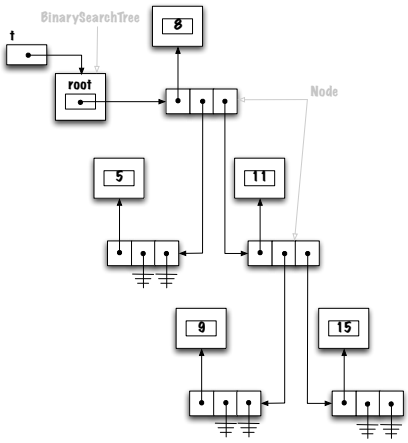


Implémentation d'un arbre binaire de recherche

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

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

    private Node<E> root;
```

Diagramme de mémoire





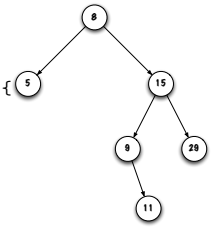
Traverser l'arbre

Pré-ordre : racine, gauche, droit ;
Symmétrique : gauche, racine, droit ;
Post-ordre : gauche, droit, racine.

Traverser l'arbre

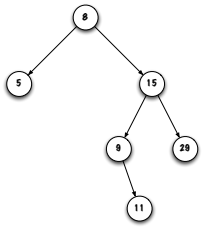
```
private void visit( Node<E> current ) {
    System.out.print( " " + current.value );
}


public void preOrder() {
    preOrder( root );
}

public void inOrder() {
    inOrder( root );
}

public void postOrder() {
    postOrder( root );
}
```

Préfixe



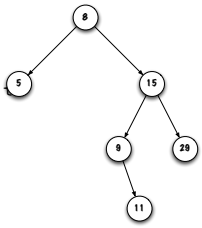

```
private void preOrder( Node<E> current ) {

    if ( current != null ) {

        visit( current );
        preOrder( current.left );
        preOrder( current.right );

    }

}

preOrder -> 8 5 15 9 11 29
```

Infixe

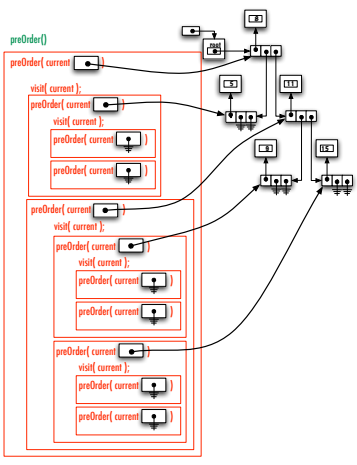



```
private void inOrder( Node<E> current ) {

    if ( current != null ) {

        inOrder( current.left );
        visit( current );
        inOrder( current.right );

    }

}

inOrder -> 5 8 9 11 15 29
```

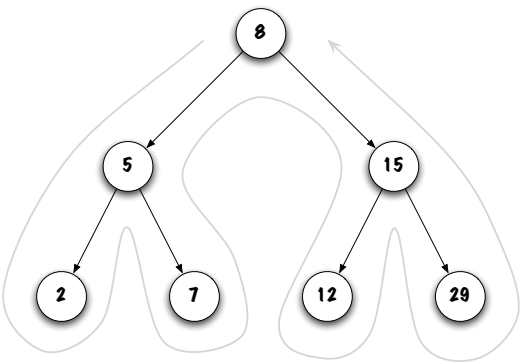
Postfixe



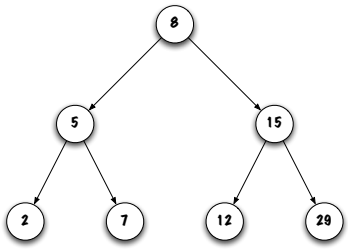

```
private void postOrder( Node<E> current ) {

    if ( current != null ) {

        postOrder( current.left );
        postOrder( current.right );
        visit( current );

    }

}

postOrder -> 5 11 9 29 15 8
```



Parcourt d'Euler



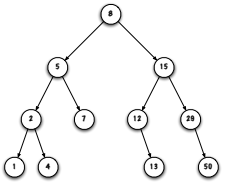
Arbre binaire plein



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

Arbre binaire

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

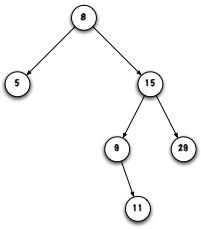


Cet arbre est-il balancé ?

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.

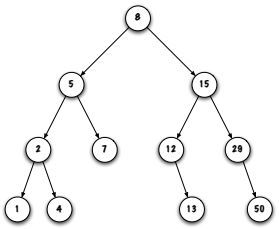
Arbre binaire

Cet arbre est-il balancé ?



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

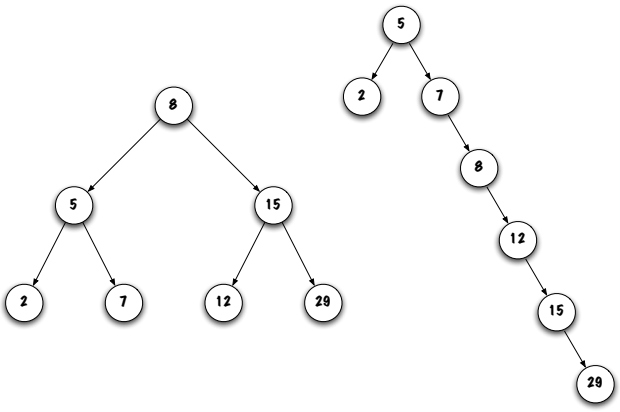
Arbre binaire

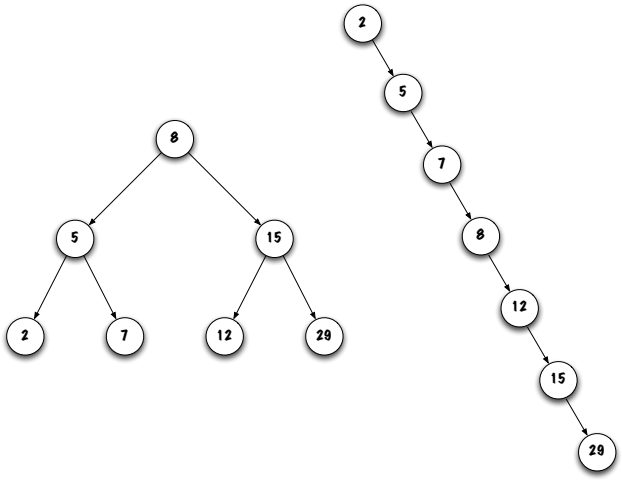


– Un arbre binaire balancé de profondeur d a de 2^d à $2^{d+1}-1$ noeuds ;
– La profondeur d'un arbre binaire balancé de taille n est $\lfloor \log_2 n \rfloor$.

Discussion

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





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 balancés sont avantageux (puisque la profondeur de l'arbre est $\lfloor \log_2 n \rfloor$) [1].

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

Observations

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

Observations

- Les méthodes qui parcourent un seul chemin, de la racine à une feuille par exemple, sont faciles à implémenter sans appel récursif, voir **contains** ;
- Les méthodes visitant plusieurs sous arbres sont souvent plus facilement implémenter à l'aide de la récursivité.

boolean add(E obj)

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

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

boolean add(E obj)

Cas spécial ? 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>( obj );
}
```

Sinon.

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

boolean add(E obj)

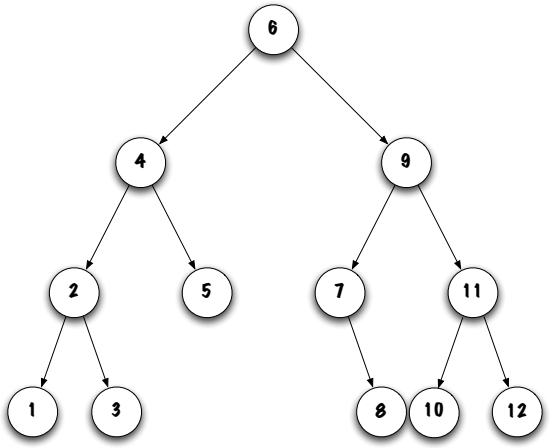
- On remplace toujours une valeur **null** par un nouveau noeud ;
- La structure existante de l'arbre ne change pas ;
- La topologie de l'arbre dépend largement de l'ordre dans lequel les éléments sont insérés.

boolean remove(E obj)

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.



boolean remove(E obj)

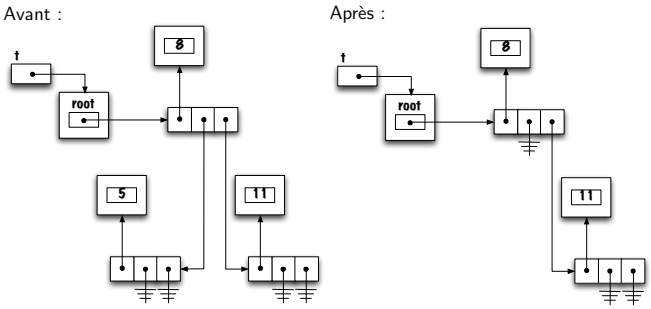
Considérez certains cas spécifiques :

- 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 !

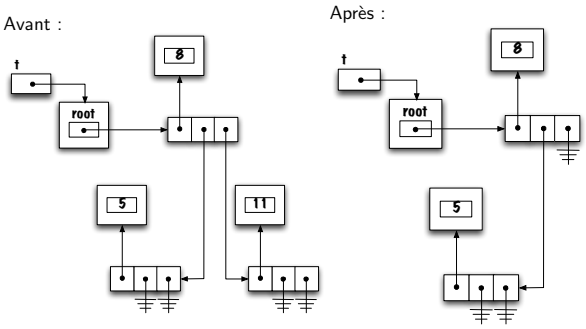
boolean remove(E obj)

Considérez certains cas spécifiques :
- Retirer la racine d'un sous-arbre.
  - Combien de sous-cas y-a-t-il et quels sont-ils ?
  - Il y a quatre sous-cas :
    1. Le sous-arbre n'a pas de sous-arbres ;
       Que fait-on ? Retire ce noeud ;
    2. Le sous-arbre possède uniquement un sous-arbre gauche ;
       Que fait-on ? Remplace le noeud par la racine de ce sous-arbre ;
    3. Le sous-arbre possède uniquement un sous-arbre droit ;
       Que fait-on ? Remplace le noeud par la racine de ce sous-arbre ;
    4. Le sous-arbre possède deux sous-arbres non-null ;
       Que fait-on ? Il y deux stratégies : 1) soit que l'on remplace ce noeud par celui qui le précède, donc l'élément le plus à droite du sous-arbre gauche, ou encore, 2) on remplace ce noeud par l'élément qui suit, donc l'élément plus à gauche du sous arbre droit.

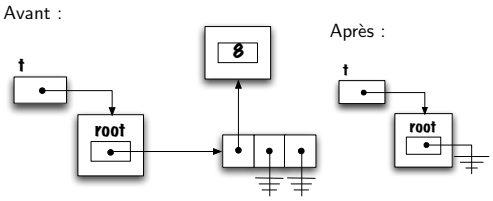
Cas 1 : retirer une feuille



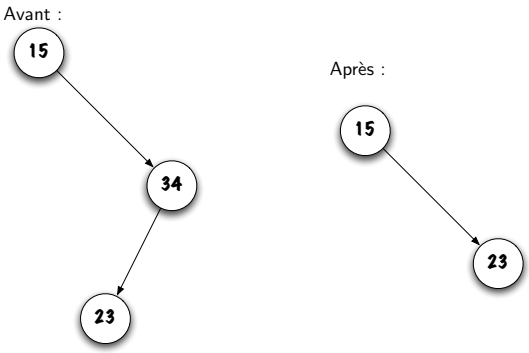
Cas 1 : retirer une feuille



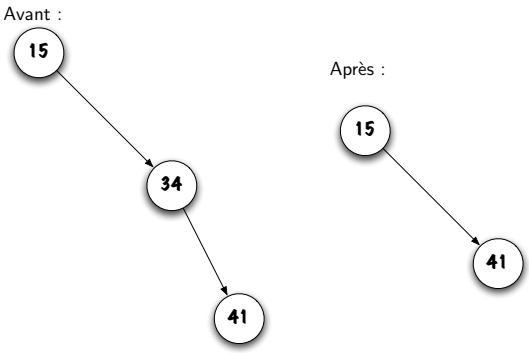
Cas 1 : retirer une feuille



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

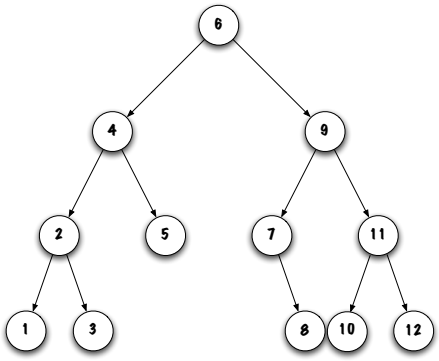


Cas 3 : t.remove(new Integer(34))



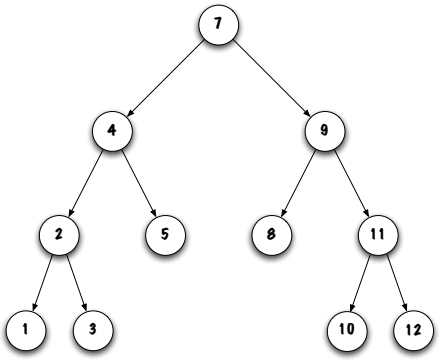
Cas 4 : t.remove(new Integer(6))

Avant :



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

Après :



Node remove(E obj)

```
// pre-condtion:

if ( obj == null ) {
    throw new IllegalArgumentException( "null" );
}

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

Node<E> remove(E obj)

```
// Remplacer le noeud à la racine de l'arbre (cas spécial)

if ( obj.compareTo( root.value ) == 0 ) {

    root = removeTopMost( root );

}
```

Node<E> remove(E obj)

```
} else { // obj n'est pas à la racine de l'arbre

    Node<E> current, parent = root;

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

    // ...
```

Node<E> remove(E obj)

```
// ...
while ( current != null ) {
    int test = obj.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 IllegalArgumentException( "null" );
    }

    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;
}
```

Alternative implementation

```
public void remove( E obj ) {
    Node<E> parent = null, current = root; boolean done = false;
    while ( current != null ) {
        int test = obj.compareTo( current.value );
        if ( test == 0 ) {
            Node<E> newTop = removeTopMost( current );
            if ( current == root ) {
                root = newTop;
            } else if ( current == parent.left ) {
                parent.left = newTop;
            } else {
                parent.right = newTop;
            }
            current = null;
        } else {
            parent = current;
            if ( test < 0 ) {
                current = parent.left;
            } else {
                current = parent.right;
            }
        }
    }
}
```

Remove (implémentation récursive)

```
public void remove( E obj ) {

    // pre-condtion:
    if ( obj == null ) {
        throw new IllegalArgumentException( "null" );
    }

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

```
private Node<E> remove( Node<E> current, E obj ) {
    Node<E> result = current;
    int test = obj.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, obj );
    } else {
        current.right = remove( current.right, obj );
    }
    return result;
}
```

Itérateurs

```
java.util.Iterator i = t.iterator();

while ( i.hasNext() ) {
    System.out.println( i.next() );
}
```

Traverser l'arbre dans quel ordre ?

Itérateurs

```
Iterator i = t.preOrderIterator();

while ( i.hasNext() ) {
    System.out.println( i.next() );
}
```

Itérateurs

Stratégie ?

Itérateurs

```
private class PreOrderIterator implements Iterator<E> {

    private Stack<E> trail;

    private PreOrderIterator() {
        trail = new LinkedStack<E>();
        if ( root != null ) {
           trail.push( root );
        }
    }
    // ...
```

Itérateurs

```
public boolean hasNext() {
    return ! trail.isEmpty();
}
```

Itérateurs

```
public E next() {

    if ( trail.isEmpty() ) {
        throw new NoSuchElementException();
    }

    Node<E> current = trail.pop();

    if ( current.right != null ) {
        trail.push( current.right );
    }
    if ( current.left != null ) {
        trail.push( current.left );
    }
    return current.value;
}
```

Observations

Il existe une très grande variété d'arbres, dont les arbres autobalancés (AVL, Rouge-Noire, B).

Un arbre général est un arbre dont les noeuds peuvent avoir plus de deux fils.