ITI 1121. Introduction to Computing II

Binary search trees: methods

by

Marcel Turcotte

Version March 27, 2020

Preamble

Preamble

Overview

Overview

Binary search trees: methods

We discuss the properties of trees: full binary tree, complete trees, maximum depth of a complete tree of size n. Finally, we implement methods for adding and removing an element to and from a binary search tree.

General objective:

- This week, you will be able to design and modify computer programs based on the concept of a binary search tree.

Preamble

Learning objectives

Learning objectives

- **Discuss** the efficiency of recursive tree processing in Java, especially in relation to memory consumption.
- **Modify** the implementation of a binary search tree to add a new method.

Readings:

- Pages 263, 265-268, 288-293 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

Summary

Summary

- A tree is a **hierarchical** data structure
- A tree can be implemented using **linked nodes**
- **Iterative** and **recursive** processing:

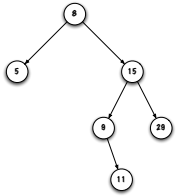
  Iterative: A method that follows **one and only one path** in the tree can easily be implemented using an iterative method.

  Recursive: A method that must **traverse more than one subtree for the same node** is usually implemented more easily using a recursive method.

Definition

A **binary search tree** is a binary tree where each node satisfies the following two properties:

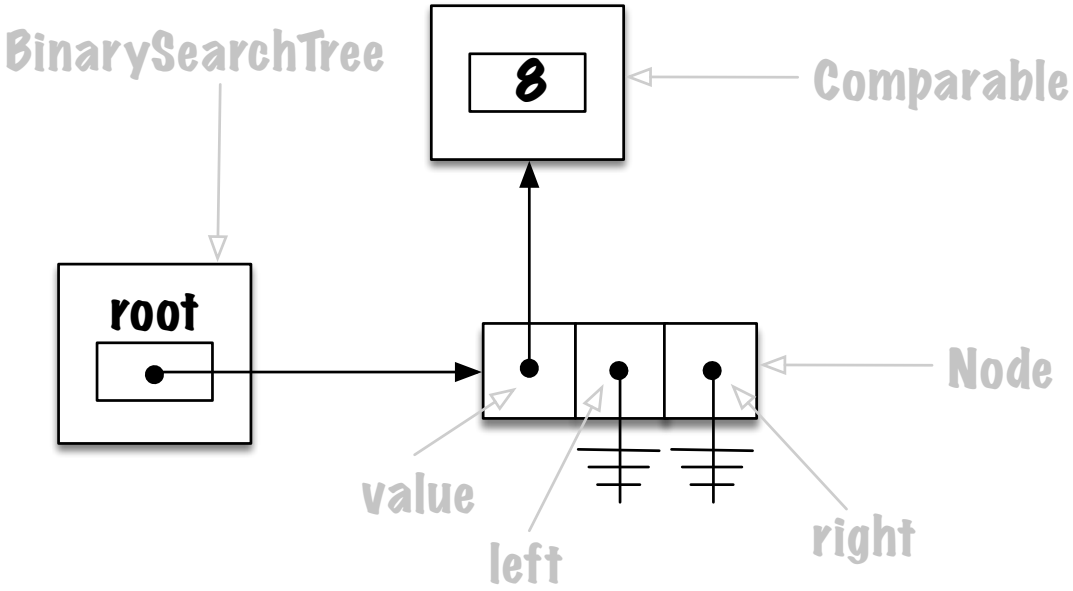
- All the nodes in its **left** subtree have **smaller** values than this node's or its left subtree is empty;
- All the nodes of its **right** subtree have **larger** values than this node's or its right subtree is empty.

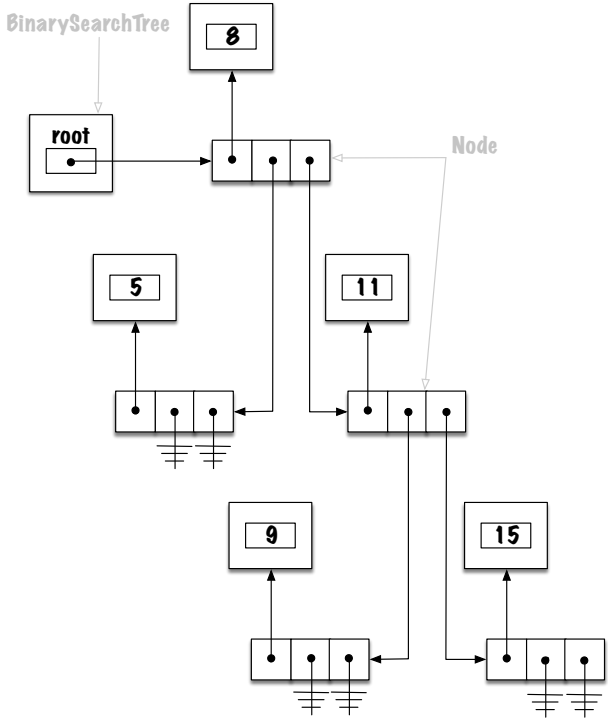


Implementation of binary research trees

```
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;

}
```

Memory diagram



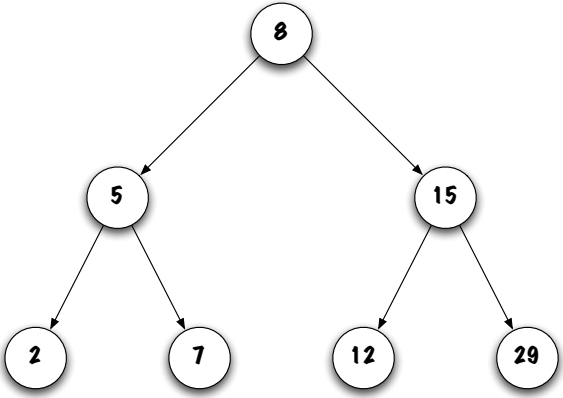

BinarySearchTree
8
root
Node
5
11
9
15

Definitions

Definitions

Full binary tree

Full binary tree



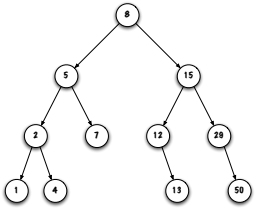
A binary tree is said to be **full** if all its nodes have exactly two children except for the leaves.

Definitions

Complete tree

Complete tree

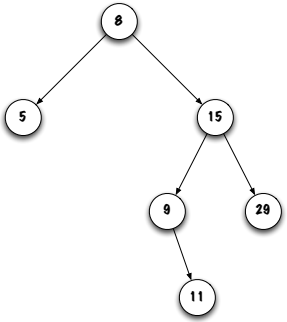
A binary tree of depth d is **complete** if all its nodes at depths less than $d-1$ (so in the interval $[0, 1 \ldots d-2]$) have exactly two children.



- Is this tree **complete**?
  - **Yes**, the depth of the tree is d =3, all the nodes at depths 0 and 1 ($\leq d-2$) have exactly two children. Nodes at depth 2 have 0, 1 or 2 children. All nodes at depth 3 are leaves.

Complete tree

- Is this tree **complete**?

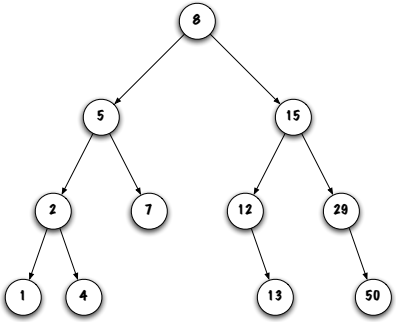


No, the depth of the tree is $d = 3$, node 5 at depth 1 ($\leq d - 2$) does not have two children.

Definitions

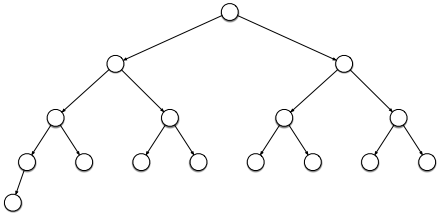
Maximum depth

Relationship between the depth and the number of nodes

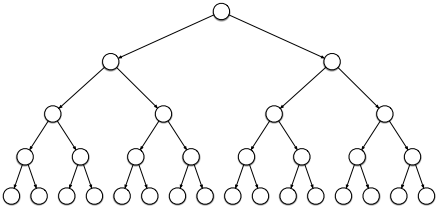


- A complete binary tree of depth d has from 2^d to $2^{d+1} - 1$ nodes;
- The depth of a complete binary tree of size n is $\lfloor \log_2 n \rfloor$.

Relationship between the depth and the number of nodes

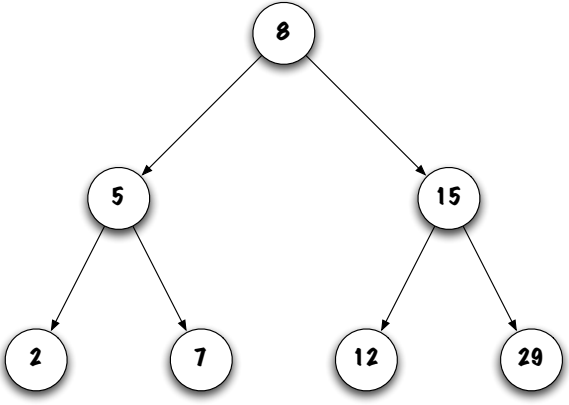


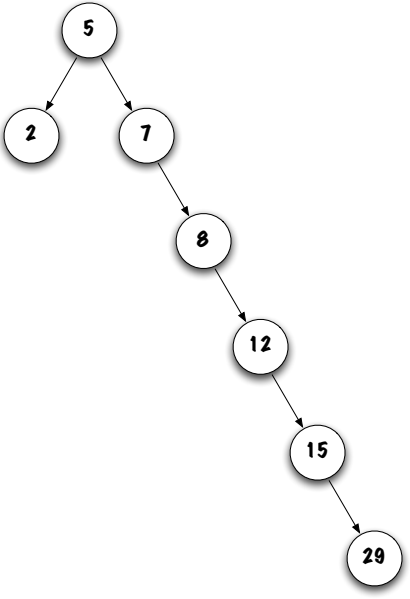
...

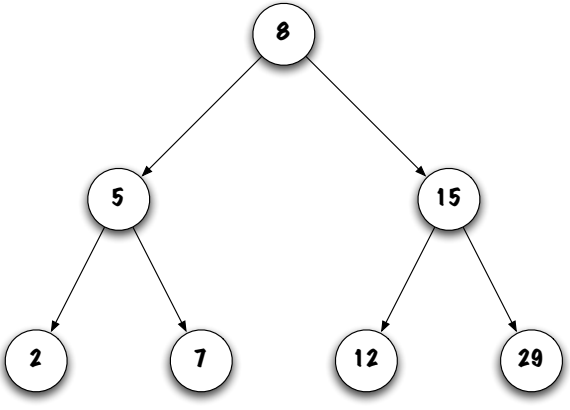


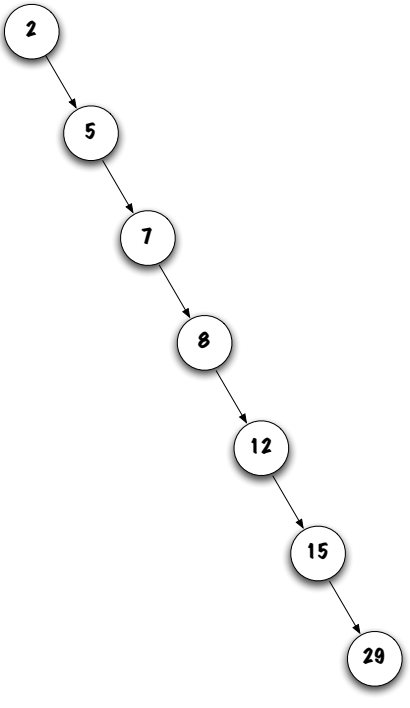
Discussion

- What relationship exists between the **efficiency** of the methods and the **topology** of the tree (complete or not).


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

- When searching, **each comparison eliminates a subtree**;
- The **maximum number of nodes visited** depends on the **depth** of the tree;
- Thus, complete trees are advantageous (since the depth of the tree is $\lfloor \log_2 n \rfloor$) *.

*In the extreme case where the tree is completely unbalanced, one would have to traverse $n - 1$ links.

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

Prefixes of the International System of Units

Prefix	n	$\lfloor \log_2 n \rfloor$
mega	10^6	19
giga	10^9	29
tera	10^{12}	39
peta	10^{15}	49
exa	10^{18}	59
zetta	10^{21}	69
yotta	10^{24}	79

- Consult **How much data is generated each day?** by Jeff Desjardins in *World Economic Forum* on 17 April 2019.

Experiment

Machine:
- 64 cores = Intel(R) Xeon(R) CPU E7-4870 v2 2.30GHz
- RAM = 512 Giga-bytes
- Operating system = Linux

Set-up:
- Average time over 5 runs calling **add** on a tree containing 1,000,000,000 (10^9) elements
- 5,765 nonoseconds, 5.765 microseconds, 0.005765 milliseconds

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

Building the tree takes 3.1348975 hours.

add

boolean add(E element)

Exercise. Starting from an empty tree, add one by one the following elements: «Lion», «Fox», «Rat», «Cat», «Pig», «Dog», «Tiger».

- What **conclusions** do you draw?

- In order to add an element, you must **find the place to insert it**.
- **Which method** is used to find an element?
  - It's the method**contains**.
- What are the **changes** to be made?

```
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)

- Is there a **special case** to deal with?
  - Operations involving a change to the variable **root** are special cases, as are changes to the variable **head** for a linked list.

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

Else.

```
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;
        }
    }
}
```

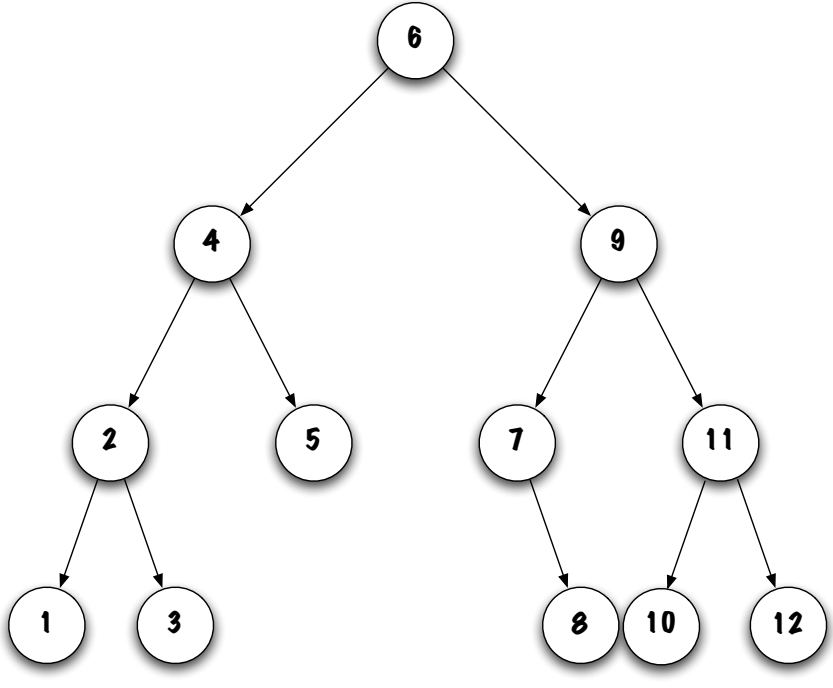
boolean add(E element)

- One always replaces a **null** value with a new node;
- The existing **structure** of the tree is **unchanged**;
- The **topology** of the tree depends largely on the **order in which the elements are inserted**.

remove

boolean remove(E element)

- **Removals** will inevitably result in **structural changes**.
- **Explore** different strategies using the tree on the next page.
  - **Remove** each of the 12 nodes, one by one.


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

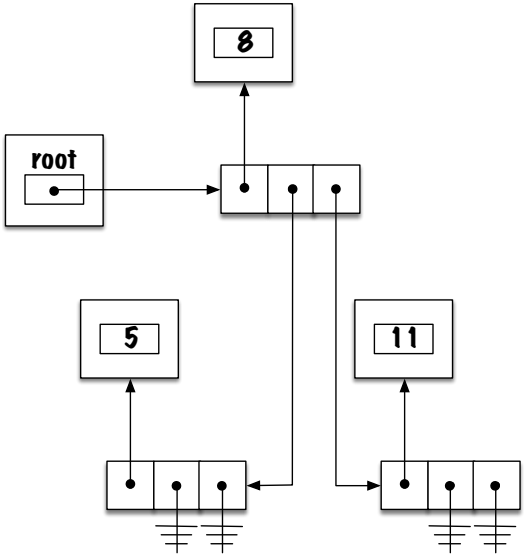
boolean remove(E element)

Consider some **specific cases**:

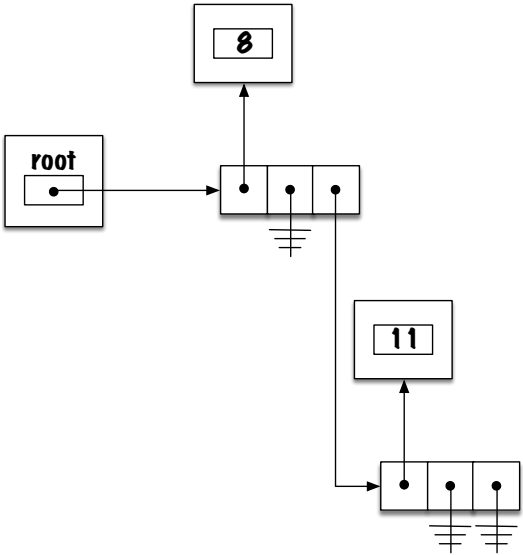
- Remove the **leftmost node**.
  - **How many** sub-cases are there and what are they?
  - There are **two** subcases:
    - The node **doesn't** have any subtrees;
    - Node **1** of the subtree **6** is an example;
    - What do we do? **parent.left = null**;
    - The node has a **right subtree**;
    - Node **7** of the subtree **9** is an example;
    - What do we do? **parent.left = "right subtree"**;
    - **The node cannot have a left subtree, otherwise it is not the leftmost node!**

Case 1: removing a leaf

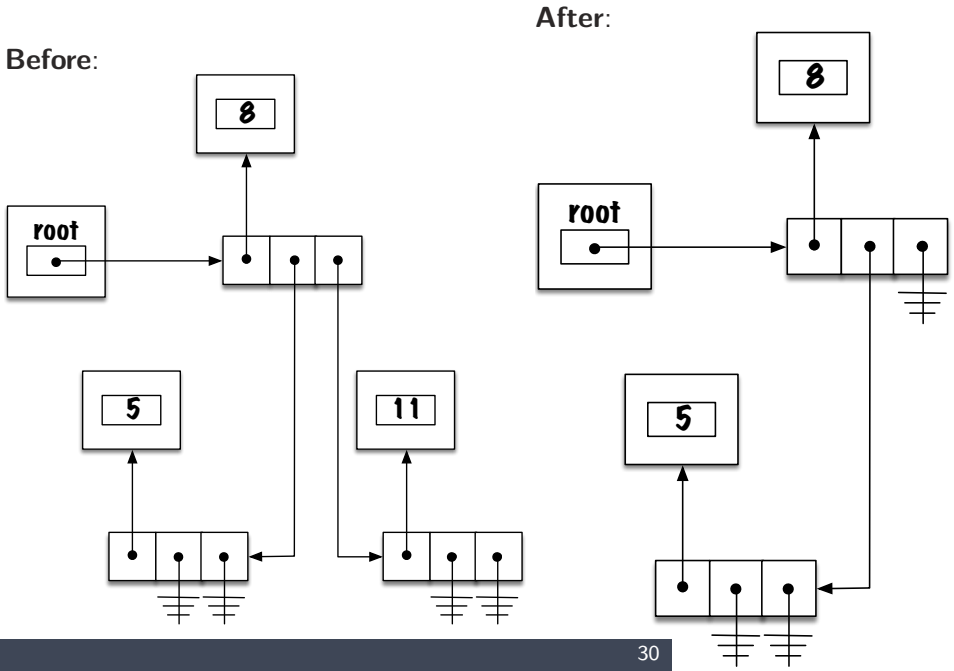
Before:



After:

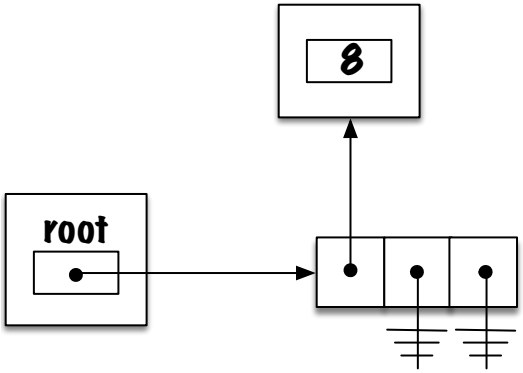


Case 1: removing a leaf

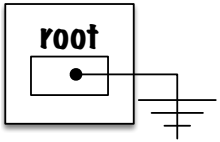


Case 1: Removing a leaf

Before:

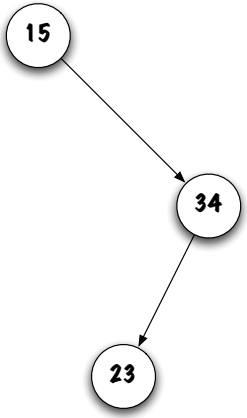


After:

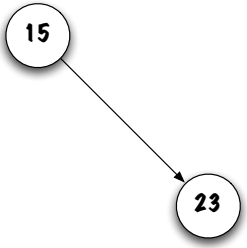


Case 2: t.remove(new Integer(34))

Before:

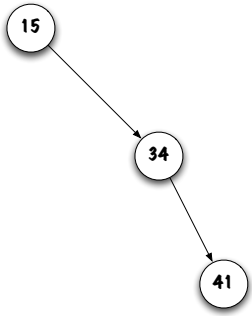


After:

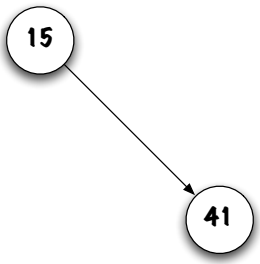


Case 3: t.remove(new Integer(34))

Before:

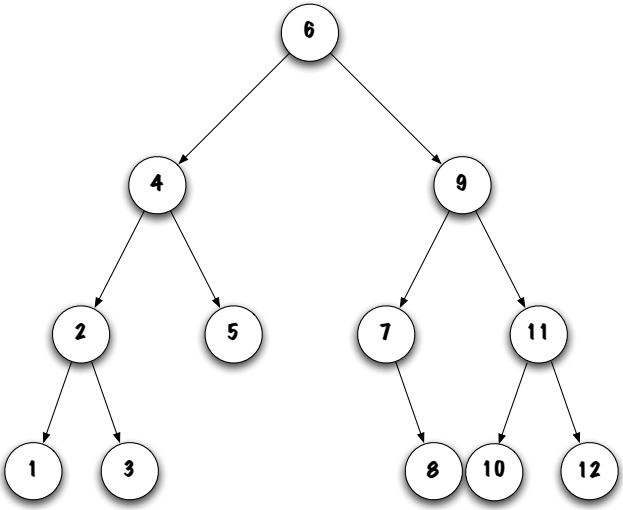


After:



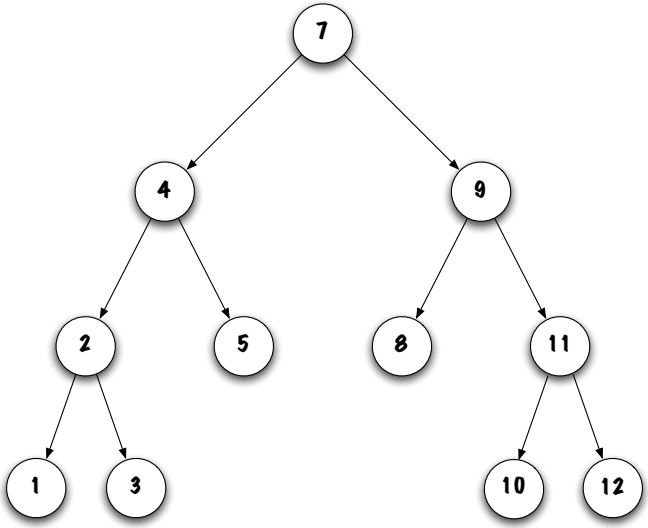
Case 4: t.remove(new Integer(6))

Before:



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

After:



Node<E> remove(E element)

```
// pre-condition:

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

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

Node<E> remove(E element)

Replacing the node at the root of the tree is a **special case**.

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

    root = removeTopMost(root);

}
```

Node<E> remove(E element)

element is not at the root of the tree

```
} 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;
}
```

Recursive implementation

```
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

- There is a very wide variety of trees, including the **self-balancing trees** (AVL, Red-Black, B).
- A **general tree** is a tree whose nodes may have more than two children.

Summary

- The **binary search tree** is a binary tree such that all the keys in the **left** subtree are **smaller** than that of the current node and all the keys in the **right** subtree are **larger**;
- Such a data structure allows for efficient searches.

Prologue

Summary

- The **topology** of the tree depends on the order in which the elements are added.
- If the tree is **complete** and contains **n** elements, you'll have to follow at most $\lfloor \log_2 n \rfloor$ links to find the element you are looking for.

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

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa