DATA STRUCTURES AND ALGORITHMS
Spring 2025

Tree Part II

Lecturer: Do Thuy Duong

Contents

- Tree implementations – part 2
  - Binary tree implementation
- More binary tree's operators
- Tree applications
  - Binary search tree
  - Arithmetic expression tree

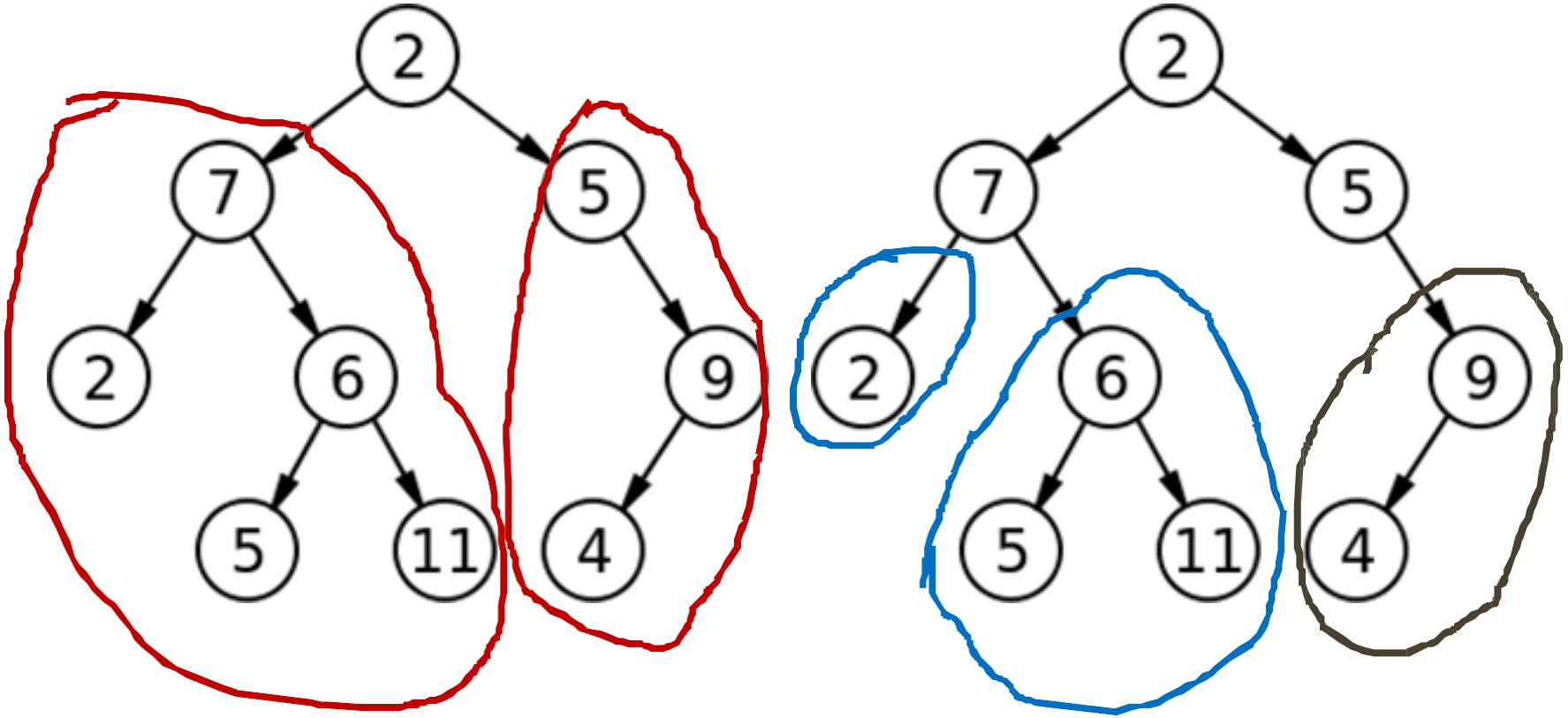
Tree Implementation

Binary tree implementation [1]

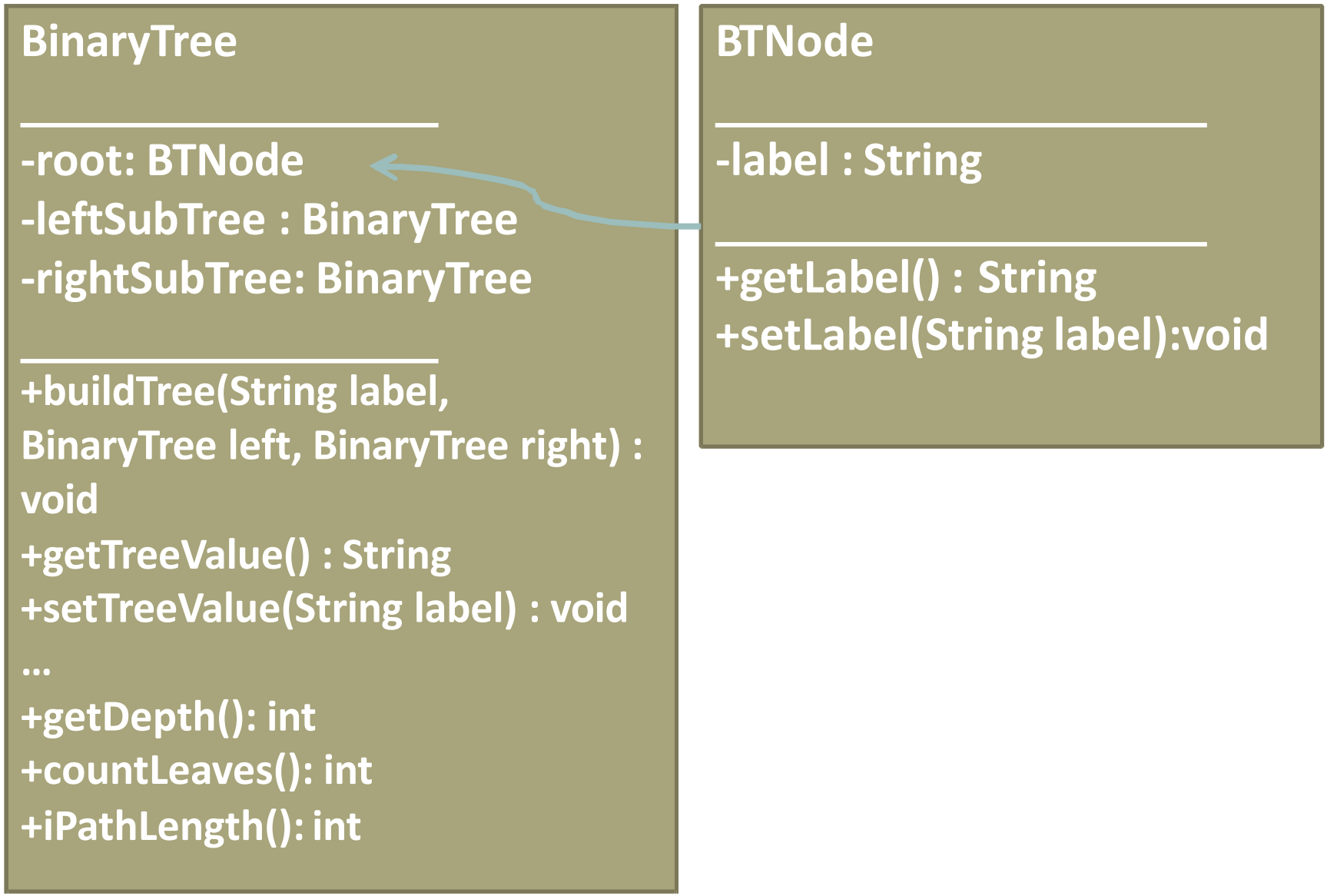
- **Recall recursive definition of a binary tree**
  - A binary tree contains
    - A root node: **root**.
    - A left sub-tree: **leftSubTree**.
    - A right sub-tree: **rightSubTree**.
    - root, leftSubTree and rightSubTree **can be empty**.
  - Tree's value:
    - **The value of a binary tree (using recursive definition) is the root's label.**

Binary tree implementation [2]

- **Recursive definition – Tree example**



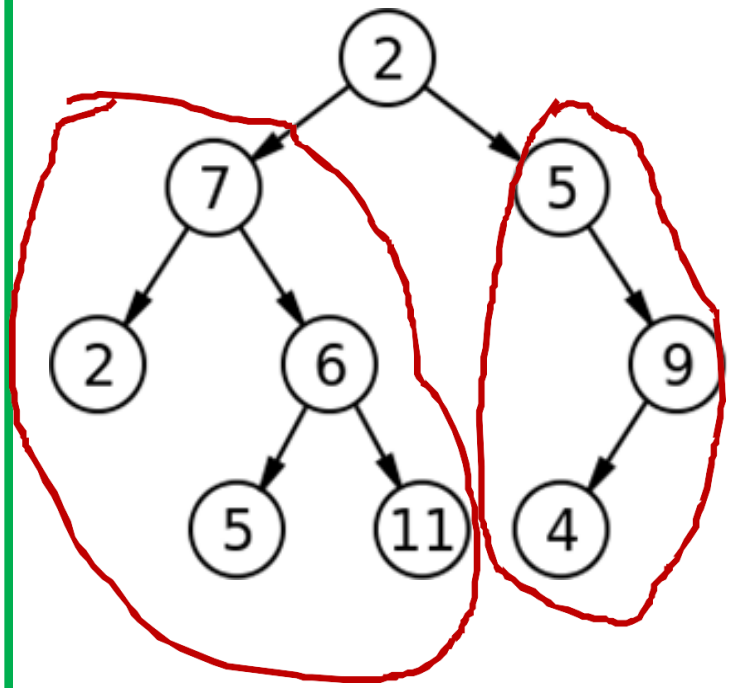
Binary tree implementation [3]



Binary tree implementation [4]

- **buildTreeByValue**(String rootValue, String leftValue, String rightValue);
- **buildTree**(String label, BinaryTree left, BinaryTree right)

Code




```
BinaryTree t3=new BinaryTree();
t3.buildTreeByValue("2",null,null);
BinaryTree t4=new BinaryTree();
t4.buildTreeByValue("6","5","11");
BinaryTree t5=new BinaryTree();
t5.buildTreeByValue("9","4",null);
BinaryTree t1=new BinaryTree();
t1.buildTree("7",t3,t4);
BinaryTree t2=new BinaryTree();
t2.buildTree("5",null,t5);
BinaryTree t= new BinaryTree();
t.buildTree("2",t1,t2);
```

Binary tree operators [1]

- **List of operators are presented in the Tutorial 10 instruction**

- **preOrderTravel**(BinaryTree t)

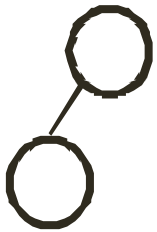
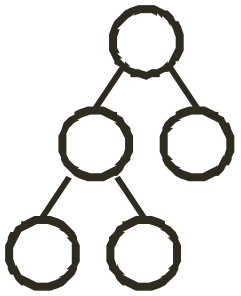
Code

```
public void preOrderTravel(BinaryTree t)
{
  if (t!=null)
  {
    System.out.print(t.getTreeValue() + " ");
    preOrderTravel(t.getLeftSubTree());
    preOrderTravel(t.getRightSubTree());
  }
}
```

Binary tree operators [2]

- **getDepth**(BinaryTree t)

Example:

Depth of a NULL binary tree is **0**	Depth of a tree with 1 node is **0**	 Depth = **1**	 Depth = **2**

depth (T) = 1 + max(depth(T.leftSubTree), depth(T.rightSubTree))

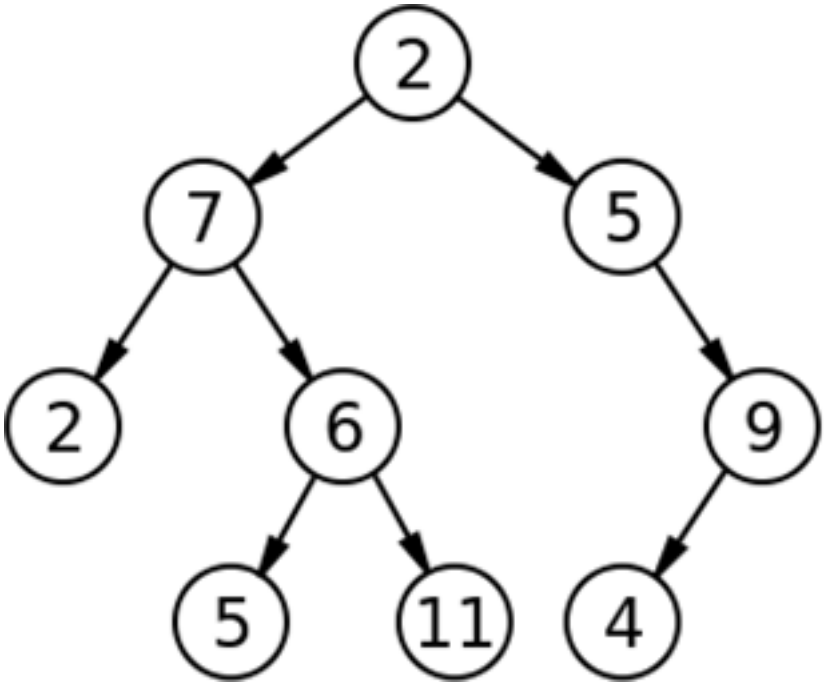
Binary tree operators [3]

- **getDepth**(BinaryTree t)

Code

```
public int getDepth(BinaryTree t)     {
  if (t==null)
     return 0;
  if (t.isLeaf())
     return 0;
  int leftSubTreeDepth=0;
  leftSubTreeDepth=getDepth(t.getLeftSubTree());
  int rightSubTreeDepth=0;
  rightSubTreeDepth=getDepth(t.getRightSubTree());
  if (leftSubTreeDepth > rightSubTreeDepth)
     return 1+leftSubTreeDepth;
  else
     return 1+rightSubTreeDepth;
}
```

Binary tree operators [4]



- **countLeaves**(BinaryTree t)

If T is null
return 0;
If T is a leaf node
return 1;
else
countLeaves (T) =
countLeaves(T.leftSubTree)+
countLeaves(T.rightSubTree)

- **iPathLength**(BinaryTree t)
  internal path length (IPL) is the total distance of all nodes to the root.
- Example:
  IPL=17

Binary tree operators [5]

- **iPathLength**(BinaryTree t, int height)
  internal path length is the total distance of all nodes to the root.

```
If T is null
    return 0;
else
    return iPathLength(T.getLeftSubTree,height+1)
            + iPathLength(T.getRightSubTree,height+1)+height
```

Tree Application

Finding duplicates problem [1]

- Considering a list of integers, **find all duplicate numbers** in the list
- Example: a={7,4,5,9,5,8,3,3} → 5 and 3 are duplicates.

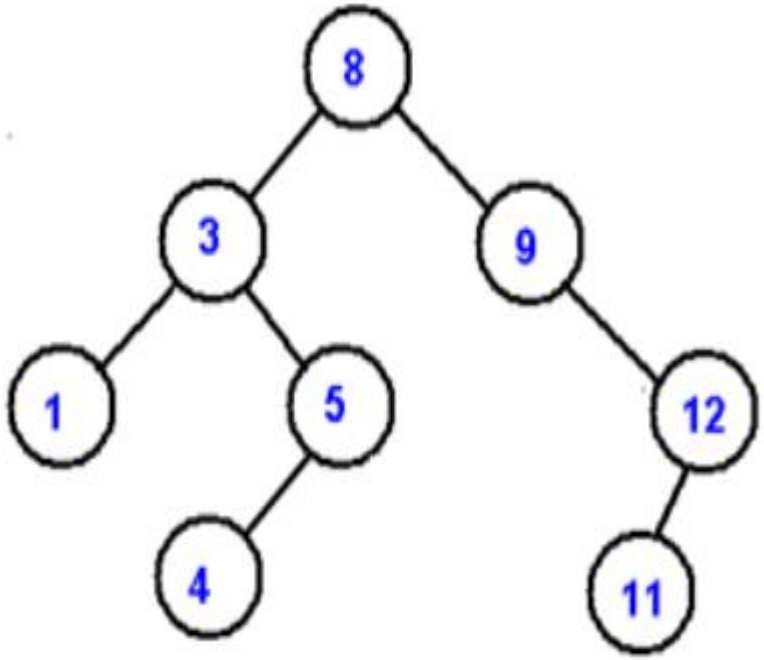
$O(n^2)$

Code

```
public void findDuplicates(int[] a)
{
  for (int i=0; i<a.length-1; i++)
    for (int j=i+1; j<a.length; j++)
      if (a[i]==a[j])
        System.out.println(a[j]+ "is a duplicate number");
}
```

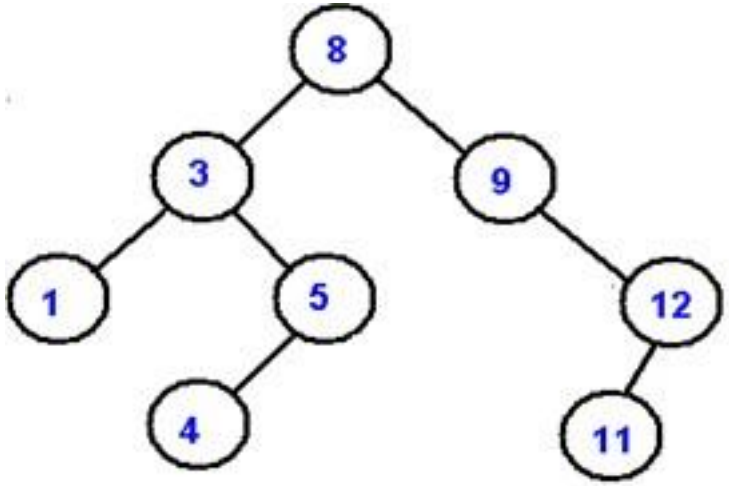
Finding duplicates problem [2]

- **Using a binary search tree (BST)**
  - A **binary tree with integer value** (nodes have integer label)
  - The value of the root is **bigger** than the value of all nodes in the **left sub-tree** and **smaller** and the value of all nodes in the **right sub-tree**.
  - The left sub-tree and the right sub-tree are also binary search trees.
  - Duplicates are not allowed

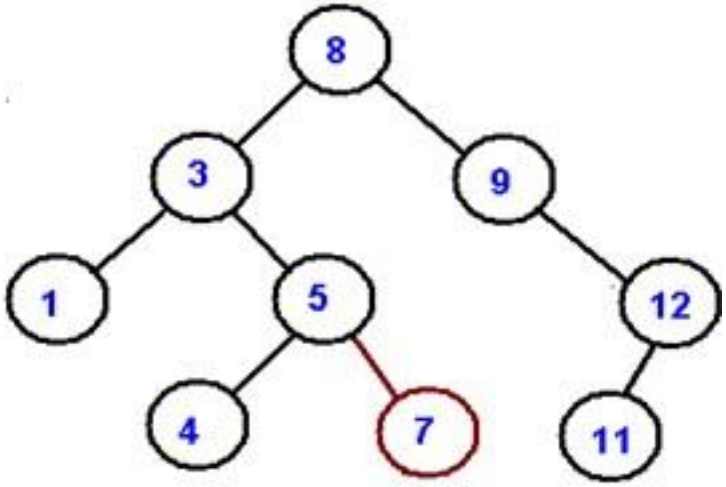


Finding duplicates problem [3]

- **BST operations**
  - **insert**(int **key**) : boolean
    Insert a new sub-tree with **key** value into the BST. **Returns false if the key is already in the tree**.
  - Example: **insert(7)**



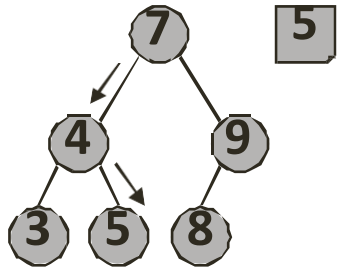
before insertion



after insertion

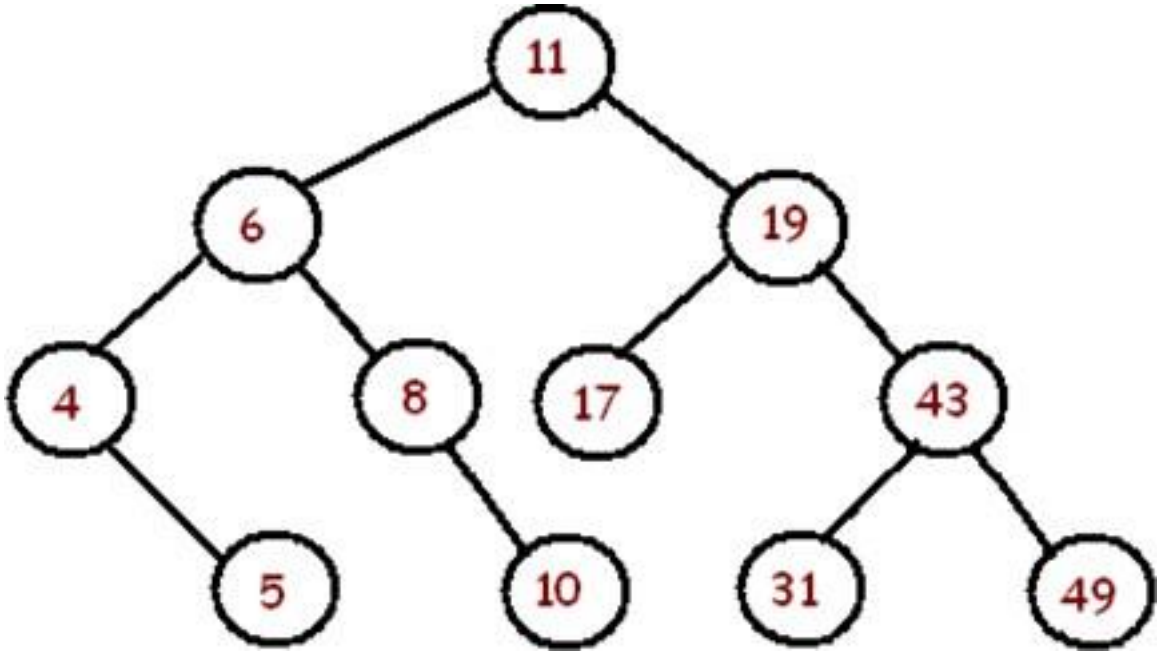
Finding duplicates problem [4]

- **BST operations analysis**
  - **insert algorithm**:
  - Start from root node, travel along the tree.
  - For each node reached in traversal:
    - **case1**: If the node is empty, add new node and return true.
    - **case 2**: If the node value is the same as the number, return false.
    - **case 3**: If the node content is larger (smaller) than input number, then travels to its left (right) sub-tree.
    - The traversal ends due to case 1 or 2.



Finding duplicates problem [5]

- **BST operations**
  - **search**(int **key**) : boolean
    searching for **key** value in the BST. **Returns false if the key does not exist in the tree**.
  - Example: **search(17) returns true**



Finding duplicates problem [6]

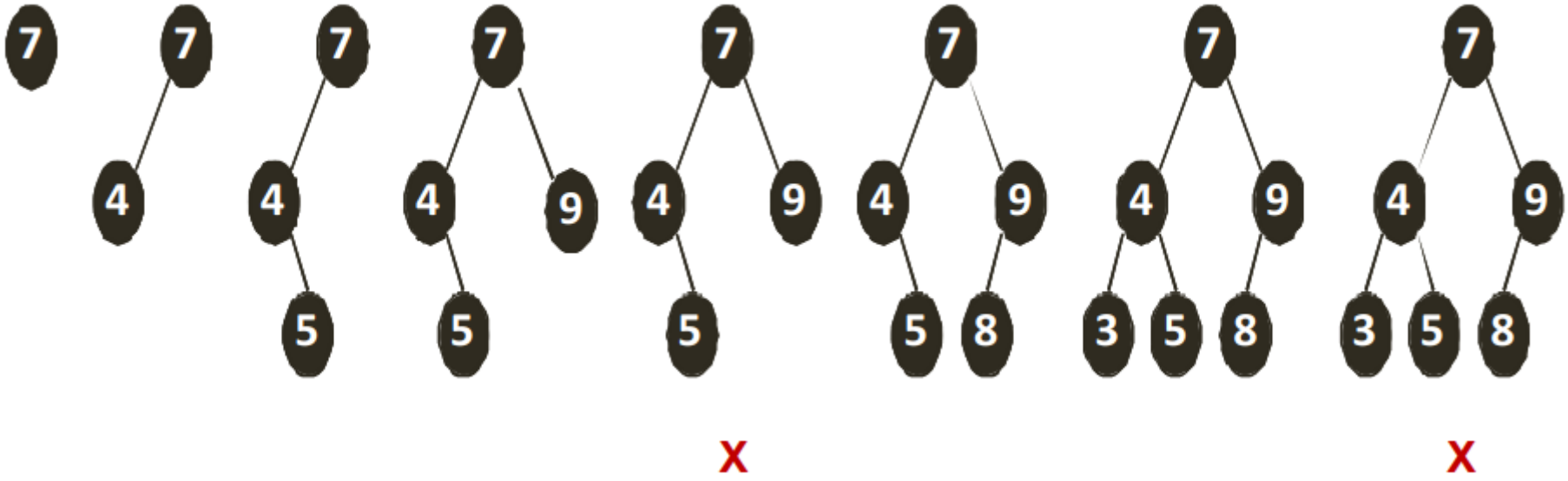
- **BST operations analysis**
  - If d is the depth of the BST then insert and search operations will take O(d) in the worst case.
  - A BST with N nodes has an average of logN level, then insert and search operations will take O(logN) in the average-case.
  - In the **degenerate situation**, a BST may have N level. In that case, insert and search will take O(N).

Finding duplicates problem [7]

- **Solving find duplicates problem with BST**
  - Create an empty BST, T
  - For each element in the array a, a[i]:
    - Insert a[i] into tree T
    - If the insert operation returns false, then a[i] is a duplicate number.
  - The time complexity of this algorithm is Nlog(N) in the average case, and $O(N^2)$ in the worst case.

Finding duplicates problem [8]

- **Example**
  - a={7,4,5,9,5,8,3,3}

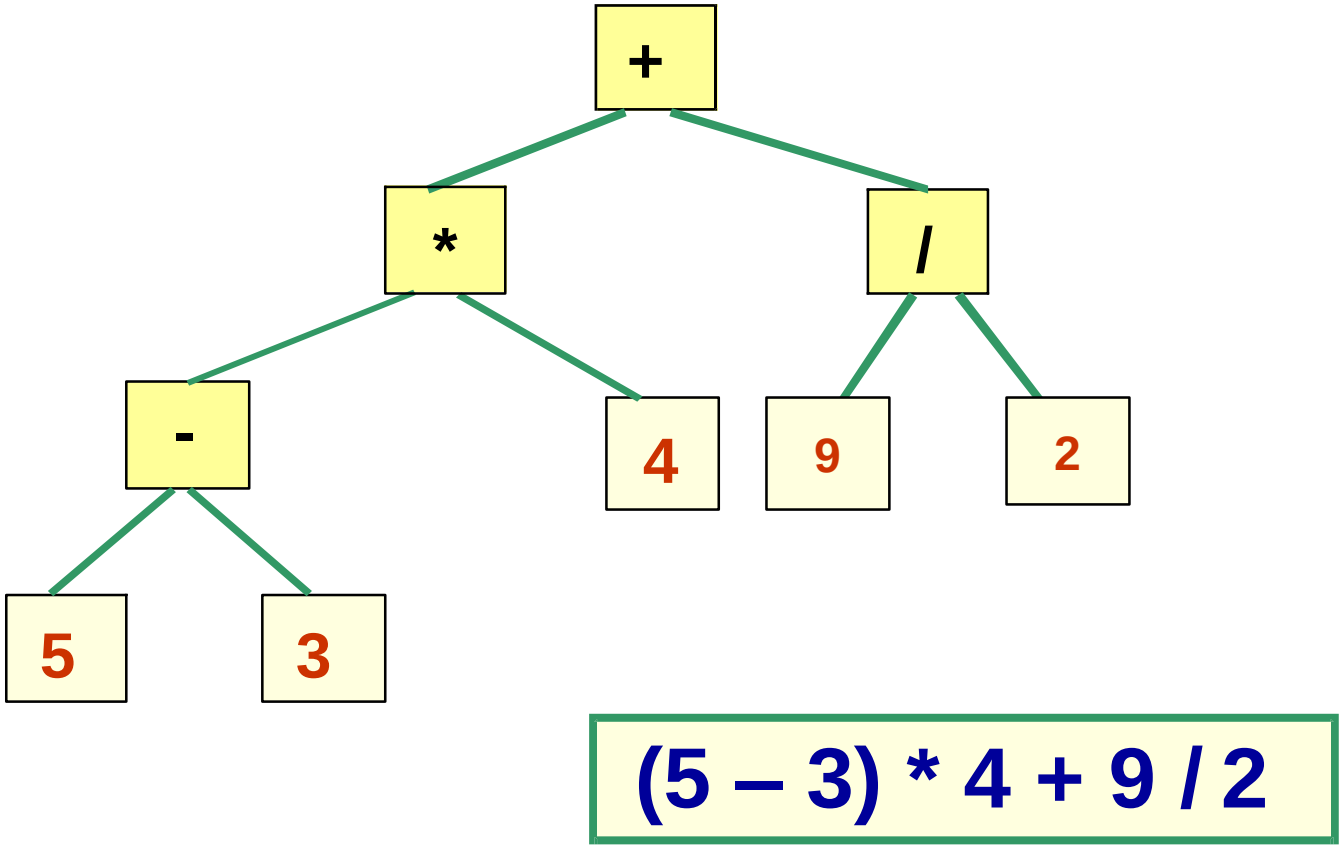


Expression tree [1]

- **Definition:** An **expression tree** for an arithmetic, relational, or logical expression is a **binary tree** in which:
  - The parentheses in the expression do not appear.
  - The **leaves** are the **operands** in the expression.
  - The **non-leaf nodes** are the **operators** in the expression.
  - A node for a binary operator has two non-empty sub-trees.

Expression tree [2]

- **Example**



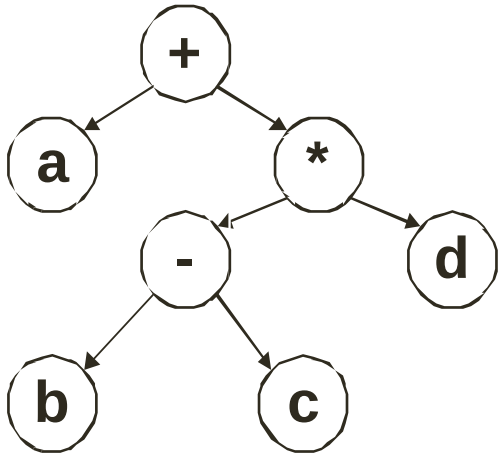
Expression tree [3]

- **Why expression trees?**
  - Expression trees impose a hierarchy on the operations in the expression.
    - Terms deeper in the tree get evaluated first.
    - This allows the establishment of the correct precedence of operations without using parentheses.
  - Expression trees can be very useful for:
    - Evaluation of the expression.
    - Generating correct compiler code to actually compute the expression's value at execution time.
    - Performing symbolic mathematical operations (such as differentiation) on the expression.

Expression tree [4]

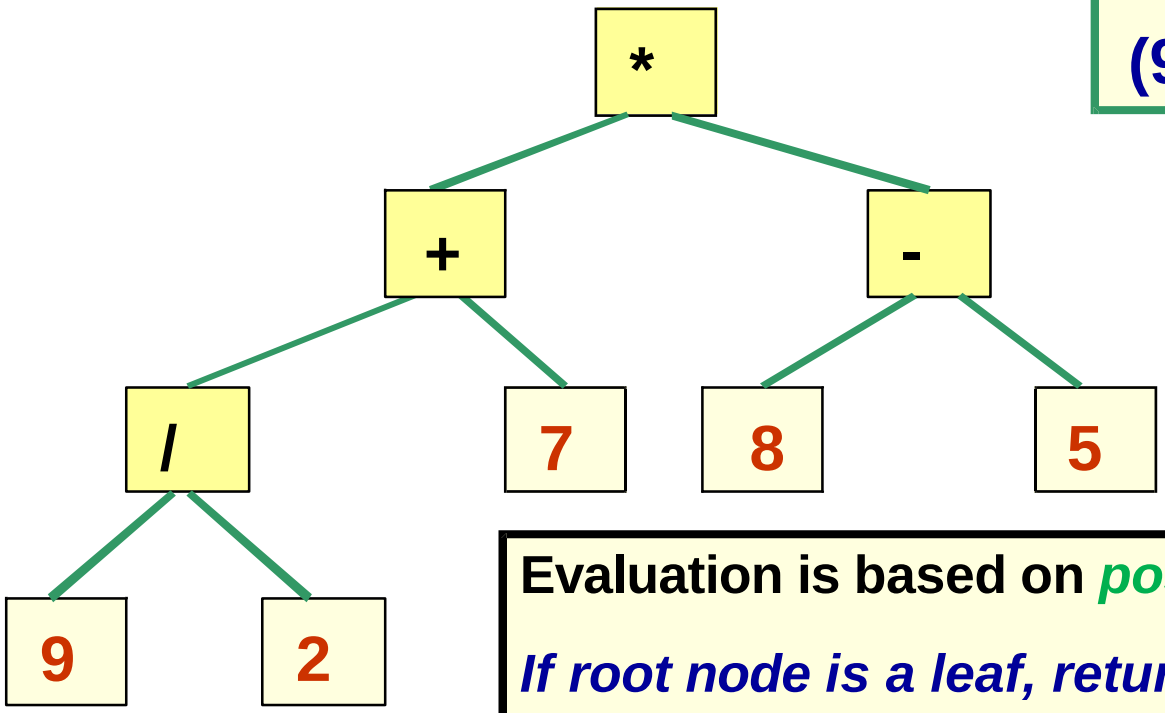
- **Expression trees and prefix, infix, postfix forms?**
  - A **pre-order** traversal of an expression tree yields the **prefix form** of the expression.
  - An **in-order** traversal of an expression tree yields the **infix form** of the expression.
  - A **post-order** traversal of an expression tree yields the **postfix form** of the expression.

Prefix form: + a * - b c d
Infix form: a + b - c * d
Postfix form: a b c - d * +



Expression tree [5]

- **Evaluating an expression tree**



This tree represents the expression

(9 / 2 + 7) * (8 – 5)

Evaluation is based on *post-order* traversal:

If root node is a leaf, return the associated value.

Recursively evaluate expression in left sub-tree.

Recursively evaluate expression in right sub-tree.

Perform operation in root node on these two values, and return result.

Expression tree [6]

- **Evaluating an expression tree**

Code

```
public int evaluate(ExpressionTree t)
{
 if(t.isLeaf())
   return    (int)t.getTreeValue();
   else {
    String   operator =  t.getTreeValue;
    int operand1   = evaluate(t.getLeftSubTree());
    int operand2   = evaluate(t.getRightSubTree());
    return(applyOperator(operand1,operator,operand2);
 }
}
```

Expression tree [7]

- **Building expression tree**
  - Built from **the postfix form** of the expression.
  - **Use a stack of ExpressionTree objects**
- **Process the postfix expression from left to right**
  - If it is an operand:
    - Create a new expression tree contains this operand
    - Push this new expression tree into stack
  - If it is an operator:
    - Pop two expression trees from the stack
    - Create a new expression tree contains this operator and two expression trees
    - Push this new expression tree into stack

Expression tree [8]

- **Example**

Build an expression tree from the postfix expression 5 3 - 4 * 9 +

- **Process the postfix expression from left to right**

Processing item **Action**

5

push(new ExpressionTree(5,null,null));

ExpressionTree Stack (top at right)

Expression tree [9]

- **Example**

Build an expression tree from the postfix expression 5 3 - 4 * 9 +

- **Process the postfix expression from left to right**

Processing item **Action**

3

```
push(new ExpressionTree(3,null,null));
```

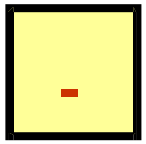
ExpressionTree Stack (top at right)

Expression tree [9]

- **Example**

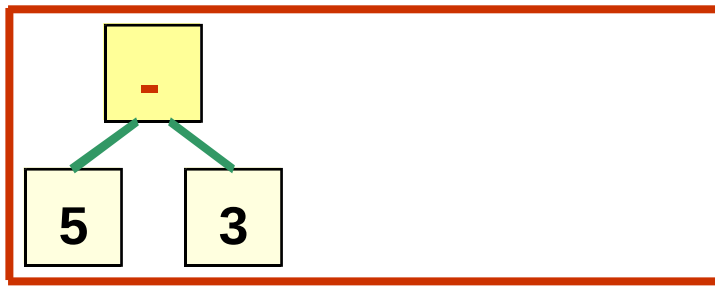
Build an expression tree from the postfix expression 5 3 - 4 * 9 +

Processing item **Action**




```
op2 = pop
op1 = pop
push(new ExpressionTree(-,op1,op2));
```

ExpressionTree Stack (top at right)



Expression tree [10]

- **Example**

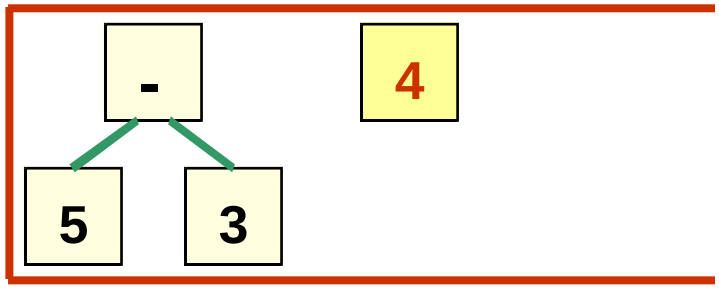
Build an expression tree from the postfix expression 5 3 - 4 * 9 +

Processing item **Action**

4

```
push(new ExpressionTree(4,null,null));
```

ExpressionTree Stack (top at right)

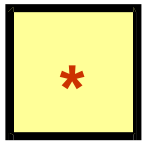


Expression tree [11]

- **Example**

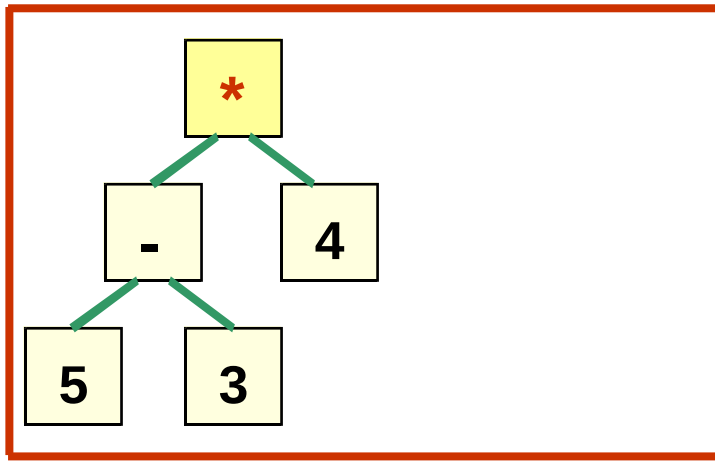
Build an expression tree from the postfix expression 5 3 - 4 * 9 +

Processing item **Action**




```
op2 = pop
op1 = pop
push(new ExpressionTree(*,op1,op2));
```

ExpressionTree Stack (top at right)



Expression tree [12]

- **Example**

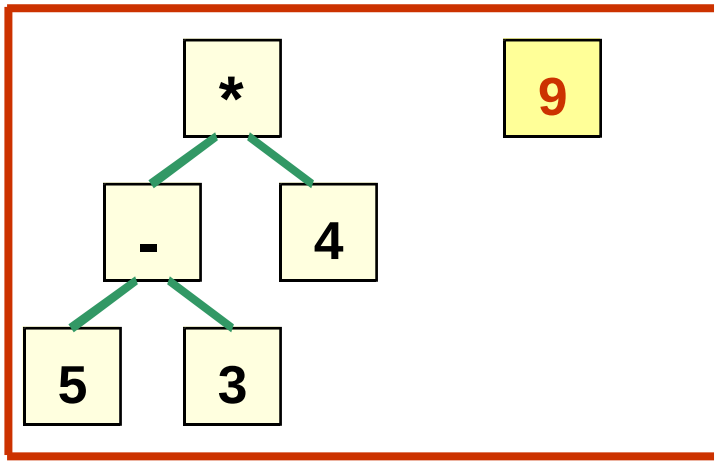
Build an expression tree from the postfix expression 5 3 - 4 * 9 +

Processing item **Action**

9

```
push(new ExpressionTree(9,null,null));
```

Expression Tree Stack (top at right)



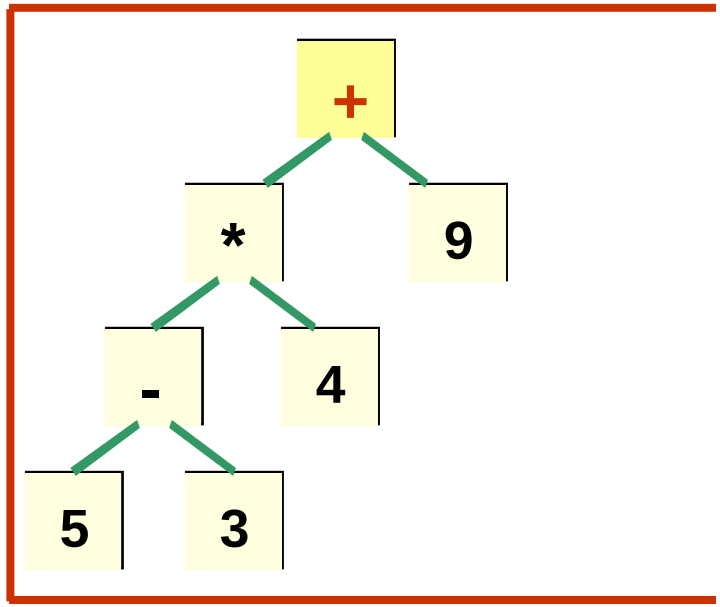
Build an expression tree from the postfix expression 5 3 - 4 * 9 +

Processing item **Action**

+

```
op2 = pop
op1 = pop
push(new ExpressionTree(+,op1,op2));
```

ExpressionTree Stack (top at right)



End of the expression has been reached, and the full expression tree is the only tree left on the stack

Tutorial & next topic

- **Preparing for the tutorial:**
  - Practice with examples and exercises in Tutorial 09
- **Preparing for next topic:**
  - Read textbook chapter 9 (9.1 – 9.3): Graph algorithms.
  - Read supplementary book chapter 22, 24 and chapter 25