UCL DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

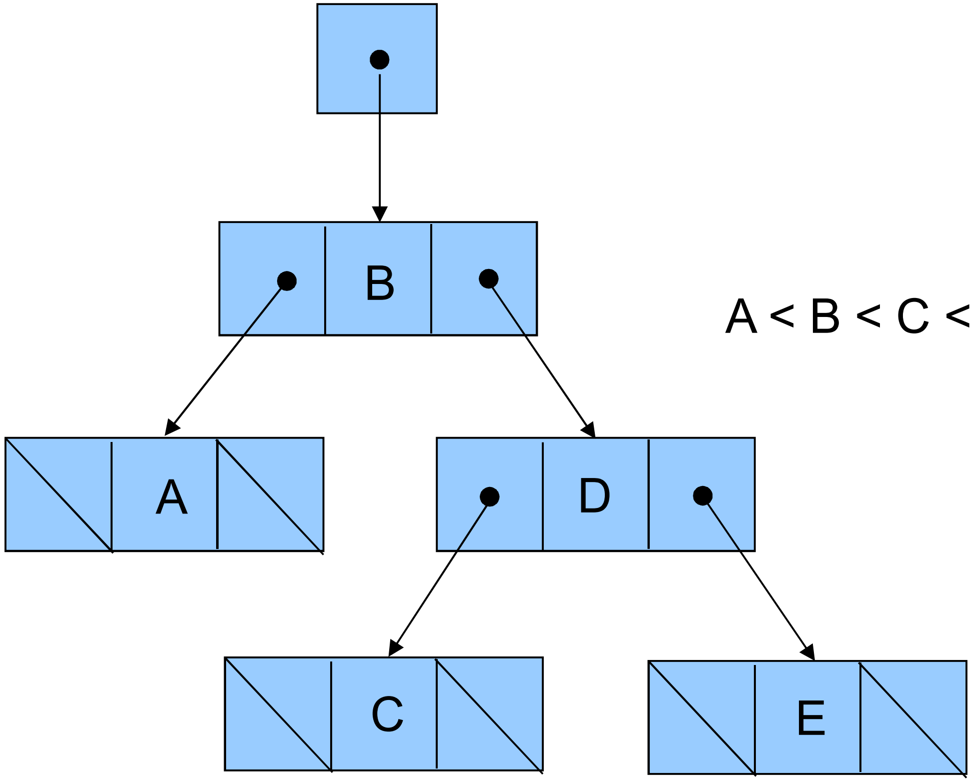
ELEC0021 - PROGRAMMING II
OBJECT-ORIENTED PROGRAMMING

DATA STRUCTURES 2 -
BINARY TREES

Prof. George Pavlou
Communication and Information Systems Group
Dept. of Electronic and Electrical Engineering
http://www.ee.ucl.ac.uk/~gpavlou/
g.pavlou@ee.ucl.ac.uk

Binary Trees

- Binary (Search) Trees: values in any left subtree are smaller and in any right subtree greater than the subtree's parent node
  - No duplicate values
  - Tree shape for a data set varies based on the order of insertion
    e.g. for the tree below order was B, A, D, C, E or B, D, E, C, A or …
  - Order of insertion A, B, C, D, E will result in a list-like very unbalanced tree!



A < B < C < D < E

TreeNode Self-Referential Class

- Binary trees use the self-referencial TreeNode class which represents a subtree
  - Includes *left* and *right* references to left and right subtrees
  - Includes also an integer *key* and a reference to a general *data* object
- For integer trees, only the key is set. For more general trees, e.g. employee records, the *data* object should contain the data
  - The *key* could be set to a unique part of the data, e.g. employeeID
- TreeNode methods use recursion as this is natural, the key methods and their signatures are the following:
  - public TreeNode find (int myKey)
  - public boolean insert (int newKey, Object newData)
  - public Object remove (int myKey, TreeNode parent)

    (for *remove* we need the *parent* in order to link its left/right subtree to a node below the removed one)

TreeNode

```
public class TreeNode {
    // public instance variables as there is nothing to "protect"
    public  int key;              // the node "key"
    public  Object data;          // the contained data object
    public  TreeNode left;        // the left subtree self-reference
    public  TreeNode right;       // the right subtree self-reference

    // constructor
    public TreeNode (int myKey, Object myData)  {
        key = myKey;
        data = myData;
        left = right = null;
    }

    // continued
```

TreeNode (cont'd) - *find*

```
// TreeNode class continued
// find a node in a tree, starting at this node, and return it

public TreeNode find (int myKey) {
    if (myKey < key) {            // smaller, search left subtree
        if (left != null)
             left.find(myKey);
        else
            return null;
    }
    else if (myKey > key) {      // bigger, search right subtree
        if (right != null) {
            right.find(myKey);
        else
            return null;
        }
    // equal, found it
    return this;
}
```

TreeNode *find* Pseudo Code

```
method find
    input: seachKey
    output: treeNode

    if searchKey is smaller than treeNode's key
        if left subtree exists
            perform find recursively on left subtree
        else
            return null / not found
    else  if searchKey is greater than treeNode's key
        if right subtree exists
            perform find recursively on right subtree
        else
            return null / not found
    return treeNode / found it
end find
```

TreeNode (cont'd) - *insert*

```
// TreeNode class continued
// insert a node in a tree, starting at this node

public boolean insert (int newKey, Object newData) {
    if (newKey < key) {            // smaller, search left subtree
        if (left != null)
            return left.insert(newKey, newData);
        else {
            left = new TreeNode(newKey, newData);
            return true;
    }
    else if (newKey > key) {    // bigger, search right subtree
        if (right != null)
            return right.insert(newKey, newData);
        else {
            right = new TreeNode(newKey, newData);
            return true;
    }
    // equal, node exists and duplicate nodes are not allowed
    return false;
}
```

TreeNode *insert* Pseudo Code

```
method insert
    input: newKey, newData
    output: true for insertion, false for failure/duplicate

    if newKey is smaller than treeNode's key
        if left subtree exists
            perform insert recursively on left subtree
        else
            create new left TreeNode with newKey / newData
            return true
    else  if newKey is greater than treeNode's key
        if right subtree exists
            perform insert recursively on right subtree
        else
            create new right TreeNode with newKey / newData
            return true
    node exists, return false
end insert
```

TreeNode (cont'd) - *remove*

```
// TreeNode class continued
// remove a node with particular key under a parent node -  not examinable
// we do not cover the case where the node has 2 subtrees as it is complex!
// see www.algolist.net/Data_structures/Binary_search_tree/Removal

public Object remove (int myKey, TreeNode parent) {
    if (myKey < key)                // smaller, search left subtree
        return left == null ? false : left.remove(myKey, this);
    else if (myKey > key) {         // bigger, search right subtree
        return right == null ? false : right.remove(myKey, this);
    else {                          // equal, found it so remove node
        if (left != null && right != null) {  // two subtrees, we do not cover it
            System.err.println("cannot remove node with two subtrees!");
            return null;
        }
        // for one subtree, we simply hang it directly from the parent
        else if (parent.left == this)
            parent.left = (left != null) ? left : right;
        else if (parent.right == this)
            parent.right = (left != null) ? left : right;
        return this.data;
    }
}
```

TreeNode *remove* Pseudo Code

```
method remove  -  not examinable
    input: searchKey, parentNode
    output: data object

    if searchKey is smaller than treeNode's key
        if left subtree exists
            perform remove recursively on left subtree
        else
            return null / not removed
    else  if searchKey is greater than treeNode's key
        if right subtree exists
            perform remove recursively on right subtree
        else
            return null / not removed
    else
        if node to remove has two subtrees
            return null / not removed – we do not cover this
        else if the parent left is this node
            hang the subtree from parent left
        else if the parent right is this node
            hang the subtree from parent right
        return node's data object
end remove
```

TreeNode (cont'd) - *toString*

```
// TreeNode class continued
@Override
public String toString () {
    String output = new String();
    if (this == null)
        return output;  // empty subtree/string

     //  preorder print traversal: print node then left subtree then right subtree
    output += this.key + ":" + this.data + " ";
    if (this.left != null)
        output += this.left.toString();
    if (this.right != null)
        output += this.right.toString();
    return output;
}

} // end class TreeNode
```

TreeNode and Tree

- The TreeNode class essentially represents a subtree
  - Its toString uses a preorder tree traversal to print its content, see Tree class which also uses postorder and inorder traversals

- The Tree class to be shown next uses the TreeNode class and simply includes a *root* instance variable
  - It also uses recursion like TreeNode as this is natural
- Its methods are simple as they use the equivalent TreeNode methods, the key methods and their signatures are:
  - public TreeNode findNode (int myKey)
  - public boolean insertNode (int newKey, Object newData)
  - public Object removeNode (int remKey)

Binary Tree (1/4) - *findNode*

```
public class Tree {
    private TreeNode root;          // the root

    public Tree () {   // constructor
        root = null;
    }

    public TreeNode findNode (int myKey) {   // finds if a node exists in the tree
        if (root == null)
            return null;  // empty tree

        if (myKey < root.key)
            return root.left.find(myKey);     // uses TreeNode.find
        else if (myKey > root.key)
            return root.right.find(myKey);   // uses TreeNode.find
        else // myKey == root.myKey
            return root;
    }

    // conitnues
```

Binary Tree (2/4) – *insertNode, removeNode*

```
public boolean insertNode (int newKey, Object newData) {  // inserts a node
    if (root == null) { // empty tree
        root = new TreeNode(newKey, newData);
        return true;
    }
    else
        root.insert(newKey, newData);       // uses TreeNode.insert
}

public Object removeNode (int remKey) {  // removes a node - not examinable
    if (root == null)
        return null;  // empty tree

    if (remKey < root.key)
        return root.left.remove(remKey);     // uses TreeNode.remove
    else if (remKey > root.key)
        return root.right.find(myKey);          //  ..
    // myKey == root.myKey, node to remove is the root
    Object remData = root.data;
    root = null;
    return remData;
}
```

Binary Tree (3/4) - *preorderTraversal*

```
//  preorder traversal processes/prints the value of a node as visited
//  after that it processes values in the left and in the right subtree

public void preorderTraversal () {        // not examinable
    preorderHelper(root);
}

// the preorderHelper (also the inorderHelper and postorderHelper later)
// enables us to start a traversal from any tree node and use it recursively
// all these methods are static as they do not use any instance variable

private static void preorderHelper (TreeNode node) {
    if (node == null)
        return;     // empty subtree, do nothing

    System.out.print(node.key + ":" + node.data + " "); // print node data
    preorderHelper(node.left);
    preorderHelper(node.right);
}

// continues
```

Binary Tree (4/4) - *inorderTraversal*

```
// inorder traversal processes/prints the value of a node only after
// it has processed values in the left subtree

// it effectively prints the tree values in ascending order, hence is a tree “sort”

 public void inorderTraversal () {          // not examinable
    inorderHelper(root);
}

private static void inorderHelper (TreeNode node) {
    if (node == null)
        return;     // empty subtree, do nothing

    inorderHelper(node.left);
    System.out.print(node.key + ":" + node.data + " "); // print node data
    inorderHelper(node.right);
}

// continues
```

Binary Tree (4/4) - *postorderTraversal*

```
// postorder traversal processes/prints the value of a node only after
// it has processed values in the left AND right subtree i.e. all the children

public void postorderTraversal () {       // not examinable
    postorderHelper(root);
}

private static void postorderHelper (TreeNode node) {
    if (node == null)
        return;     // empty subtree, do nothing

    postorderHelper(node.left);
    postorderHelper(node.right);
    System.out.print(node.key + ":" + node.data + " "); // print node data
}

@Override
public String toString () {     // uses TreeNode.toString (preorder traversal)
    return root != null ?  root.toString() : new String("“);
}
} // end class Tree
```

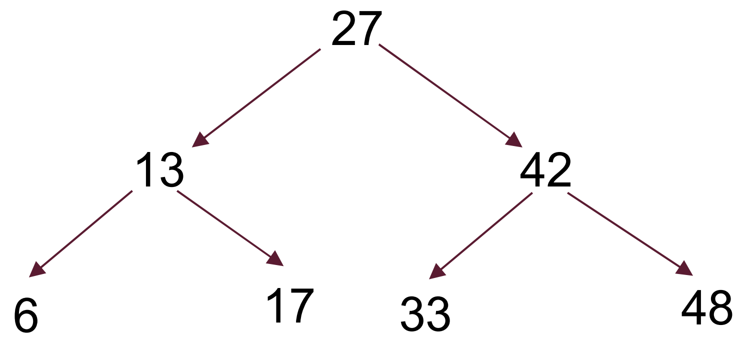
Example TreeTest Program

```
public class TreeTest {
    public static void main (String args[]) {
        Tree tree = new Tree();

        // adding 27, 13, 42, 6, 17, 17 (duplicate), 33, 48
        tree.insertNode(27, null); tree.insertNode(42, null);
        tree.insertNode(13, null); tree.insertNode(6, null);
        tree.insertNode(17, null); tree.insertNode(17, null);
        tree.insertNode(48, null); tree.insertNode(33, null);

        System.out.println("Preorder traversal");
        tree.preorderTraversal();

        System.out.println("\nInorder traversal");
        tree.inorderTraversal();

        System.out.println("\nPostorder traversal");
        tree.postorderTraversal();
        System.out.println();
    }
}
```

Balanced Binary Trees

- A binary tree is balanced (or packed) if every level contains roughly twice the elements of the level above
- A fully balanced tree with k levels has $1+2^1+2^2+2^4+\ldots+2^{k-1} = 2^k-1 = n$ nodes. Since $\log_2 n = \log_2 2^k = k$ levels, searching to find an item has O(logn) computational complexity
  - For example, for $2^{20} = 1048576 > 10^6$ nodes we need at most 20 comparisons ($n= 2^{20}$, $k=20$)
- A balanced binary tree is a much more efficient dynamic data structure than a linked list
  - Find/remove an element has O(logn) computational complexity in comparison to O(n) for the linked list
  - Having a balanced tree is easy when we know all the elements in advance (see next) but costly periodic balancing is required if the tree content is changing often

Balanced Binary Trees (cont'd)



- See above the balanced tree of the previous test program
  - Order of insertion was: 27, 42, 13, 6, 17, 48, 33 => balanced tree
  - Preorder print: 27, 13, 6, 17, 42, 33, 48
  - If a balanced tree is printed preorder and this order is used to re-create it, it results in exactly the same balanced tree e.g. 27, 13, 6, 17, 42, 33, 48 for the tree above
- We can create a balanced tree from an ordered list through the following algorithm:
  - Find the middle of the list and make it root
  - Do recursively the same for the left & right half, i.e. left & right subtrees