



DM537

Object-Oriented Programming

Peter Schneider-Kamp

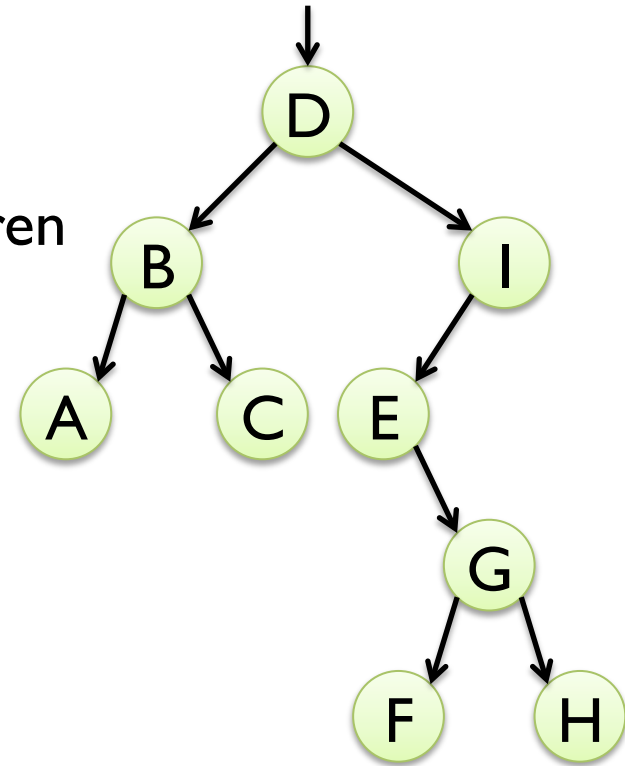
petersk@imada.sdu.dk

<http://imada.sdu.dk/~petersk/DM537/>

ABSTRACT DATA TYPES FOR (BINARY) TREES

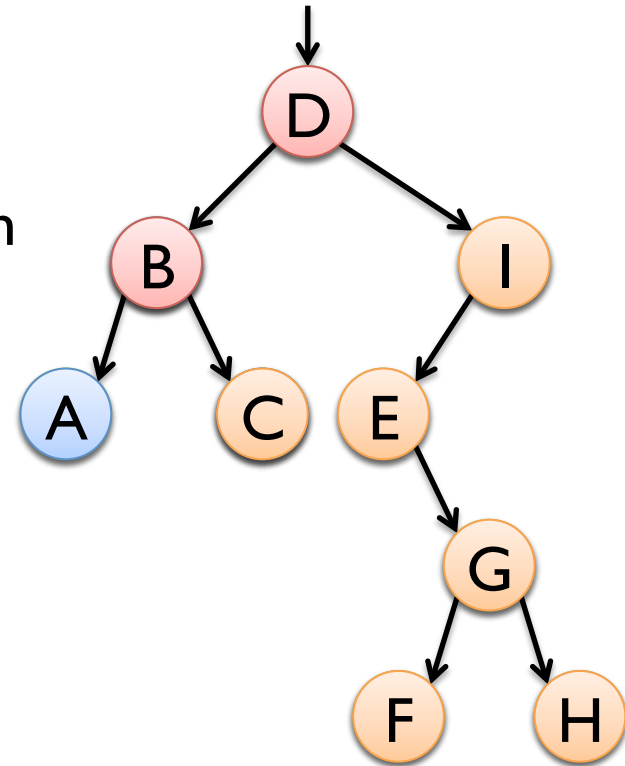
Trees

- trees store elements non-sequentially
- every node in a tree has 0 or more children
- imagine a tree with root in the air 😊
- many uses:
 - decision tree
 - binary sort trees
 - data base indices
 - ...
- no consensus on what basic binary tree operations are 😞
- set of operations depends on application
- **here:** keeping elements sorted, combinatorics



Binary Trees

- special case of general trees
- every node in a tree has 0, 1 or 2 children
- tree on the right is an example
- notation:
 - first node is called “**root**”
 - other nodes either in “**left subtree**”
 - ... or in “**right subtree**”
- every node is root in its own subtree!
- for example, look at node B
- node A is the “left child” of B
- node C is the “right child” of B
- node B is the “parent” of both A and C



BinTree ADT: Specification

- data are arbitrary objects of class E
- operations are defined by the following interface

```
public interface BinTree<E> {  
    public boolean isEmpty();           // is tree empty?  
    public int size();                 // number of elements  
    public int height();              // maximal depth  
    public List<E> preOrder();         // pre-order traversal  
    public List<E> inOrder();         // in-order traversal  
    public List<E> postOrder();       // post-order traversal  
}
```

BinTree ADT: Design & Implement. I

- Design I: use recursive data structure
 - based on representing tree nodes by `BinTreeNode<E>`
- Implementation I:

```
public class BinTreeNode<E> {  
    public E elem;  
    public BinTreeNode<E> left, right;  
    public BinTreeNode(E elem, BinTreeNode<E> left,  
                        BinTreeNode<E> right) {  
        this.elem = elem;  
        this.left = left; this.right = right;  
    }  
}
```

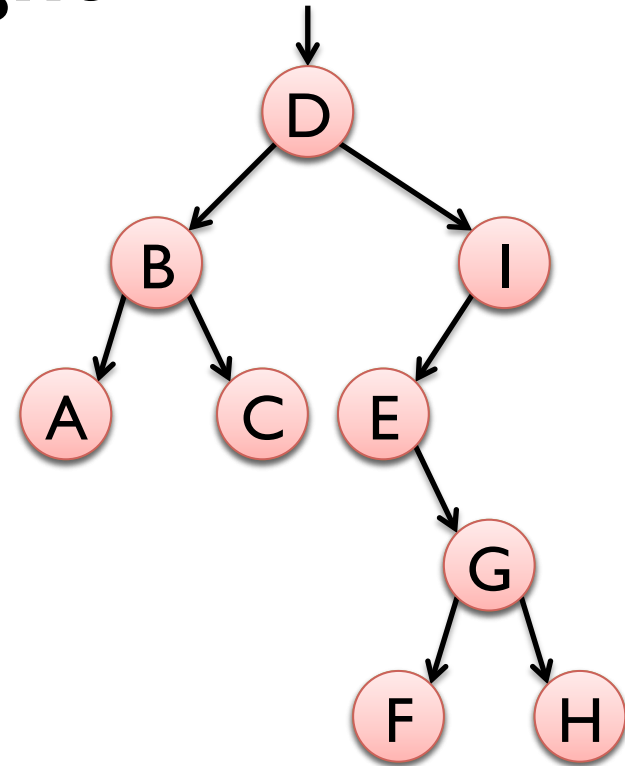
BinTree ADT: Implementation I

- Implementation I (continued):

```
public class RecursiveBinTree<E> {  
    private BinTreeNode<E> root = null;  
    public boolean isEmpty() { return this.root == null; }  
    public int size() { return size(this.root); }  
    private static <E> int size(BinTreeNode<E> node) {  
        if (node == null) { return 0; }  
        return 1 + size(node.left) + size(node.right);  
    }  
    ...  
}
```

Depth and Height

- depth of the root is 0
- depth of other nodes is $1 + \text{depth}(\text{parent})$
- Example:
 - 0
 - 1
 - 2
 - 3
 - 4
- height of a subtree is maximal depth of any of its nodes
- Example: height of tree (=subtree starting in D) is 4



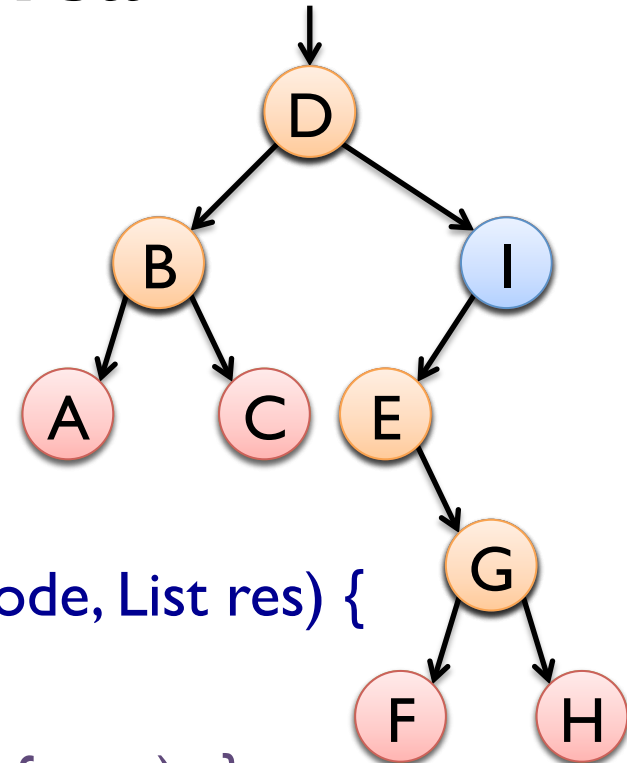
BinTree ADT: Implementation I

- Implementation I (continued):

```
public class RecursiveBinTree<E> {  
    private BinTreeNode<E> root = null;  
    ...  
    public int height() { return height(this.root); }  
    private static <E> int height(BinTreeNode<E> node) {  
        if (node == null) { return -1; }  
        return 1 + max(height(node.left), height(node.right));  
    }  
    private static int max(int a, int b) { return a > b ? a : b; }  
    ...  
}
```

Binary Tree Traversal

- traversal can be either
 - depth-first
 - breadth-first
- three standard depth-first traversals
 - pre-order

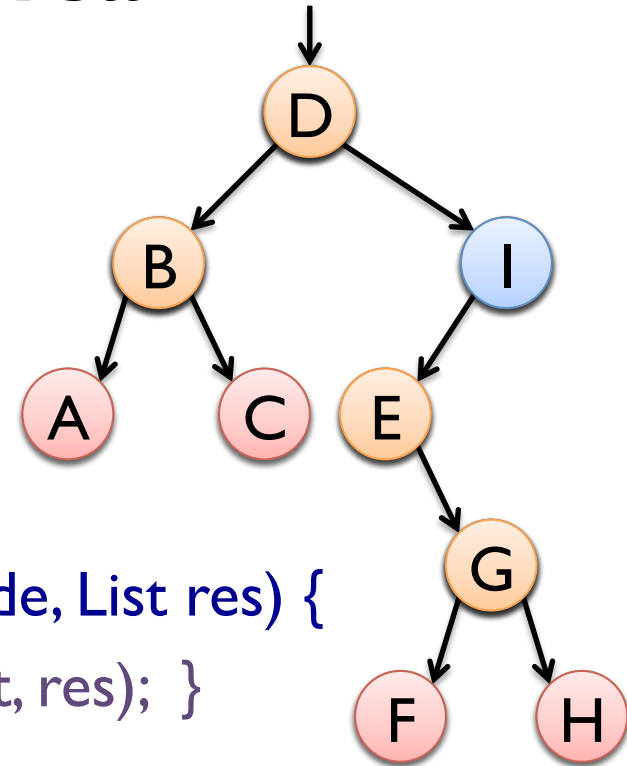


```
public static void preOrder(BinTreeNode node, List res) {  
    res.add(node.elem);  
    if (node.left != null) { preOrder(node.left, res); }  
    if (node.right != null) { preOrder(node.right, res); }  
}
```



Binary Tree Traversal

- traversal can be either
 - depth-first
 - breadth-first
- three standard depth-first traversals
 1. pre-order
 2. in-order
 3. post-order

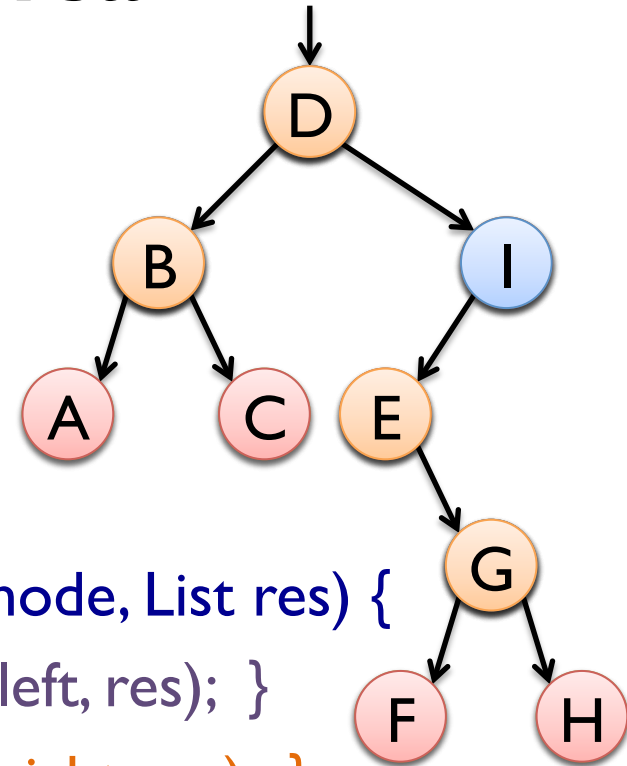


```
public static void inOrder(BinTreeNode node, List res) {  
    if (node.left != null) { inOrder(node.left, res); }  
    res.add(node.elem);  
    if (node.right != null) { inOrder(node.right, res); }  
}
```

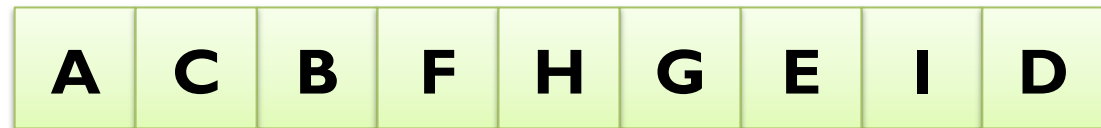


Binary Tree Traversal

- traversal can be either
 - depth-first
 - breadth-first
- three standard depth-first traversals
 - pre-order
 - in-order
 3. post-order



```
public static void postOrder(BinTreeNode node, List res) {  
    if (node.left != null) { postOrder(node.left, res); }  
    if (node.right != null) { postOrder(node.right, res); }  
    res.add(node.elem);  
}
```



BinTree ADT: Implementation I

- Implementation I (continued):

```
public class RecursiveBinTree<E> {  
    private BinTreeNode<E> root = null;  
    ...  
    public List<E> preOrder() {  
        List<E> res = new ArrayList<E>();  
        if (this.root != null) { preOrder(this.root, res); }  
        return res;  
    }  
    ... // the same for inOrder, postOrder  
}
```

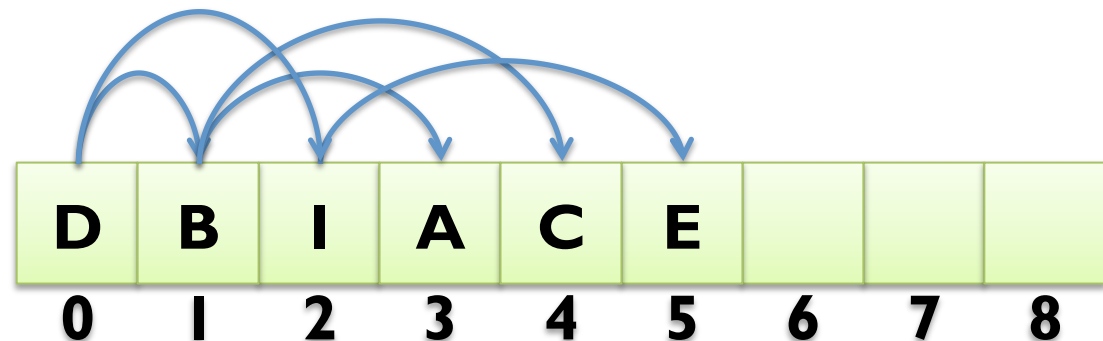
BinTree ADT: Design & Implement. 2

- Design 2: use array (list)
 - root stored at position 0
 - left child of position n stored at $2n+1$
 - right child of position n stored at $2n+2$
 - null where position stores no element
- Implementation 2:

```
public class ArrayBinTree<E> {  
    private List<E> data = new ArrayList<E>();
```

```
    ...
```

```
}
```



BinTree ADT: Implementation 2

- Implementation 2 (continued):

```
public class ArrayBinTree<E> {  
    private List<E> data = new ArrayList<E>();  
    ...  
    public boolean isEmpty() { return this.data.get(0) == null; }  
    public int size() {  
        int counter = 0;  
        for (int i = 0; i < this.data.size(); i++) {  
            if (this.data.get(i) != null) { counter++; }  
        }  
        return counter;  
    }  
    ... }  
}
```

BinTree ADT: Implementation 2

- Implementation 2 (continued):

```
public class ArrayBinTree<E> {  
    private List<E> data = new ArrayList<E>();  
    ...  
    public int height() { return height(0); }  
    private int height(int index) {  
        if (this.data.get(index) == null) { return -1; }  
        return 1 + max(height(2*index+1), height(2*index+2));  
    }  
    private static int max(int a, int b) { return a > b ? a : b; }  
    ...  
}
```


BinTree ADT: Implementation 2

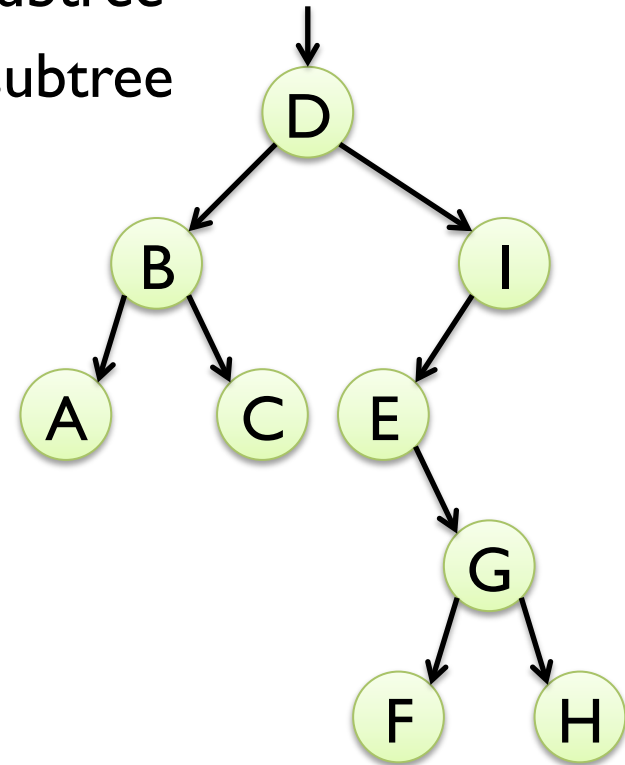
- Implementation 2 (continued):

```
public List<E> preOrder() {  
    return preOrder(0, new ArrayList<E>());  
}  
  
private List<E> preOrder(int index, java.util.List<E> res) {  
    E elem = this.data.get(index);  
    if (elem != null) {  
        res.add(elem);  
        preOrder(2*index+1, res);    preOrder(2*index+2, res);  
    }  
    return res;  
} ... /* same for inOrder, postOrder */ }
```

BINARY SORTING TREES

Binary Sort Tree

- special binary tree
- invariant for all subtrees:
 - all elements smaller than root in left subtree
 - all elements bigger than root in right subtree
- elements need to be comparable
- interface `Comparable`
- use method `compareTo`
- our example tree is a sort tree
- nodes are of class `Character`



SortTree ADT: Specification

- data are objects of some class E that extends Comparable
- operations are defined by the following interface

```
public interface SortTree<E extends Comparable<E>> {  
    public int size();           // number of elements  
    public boolean contains(E elem); // true, if elem in tree  
    public void add(E elem);     // add elem to tree  
    public void remove(E elem);  // remove elem from tree  
    public List<E> traverse();    // in-order traversal  
    public void balance();       // balance the tree  
}
```

SortTree ADT: Design & Implement.

- Design: use recursive data structure
 - reuse class `BinTreeNode<E>`
- Implementation I:

```
public class BinSortTree<E extends Comparable<E>>
    implements SortTree<E> {
    private BinTreeNode<E> root = null;
    public int size() { return size(this.root); }
    private static <E extends Comparable<E>> int
        size(BinTreeNode<E> node) {
        if (node == null) { return 0; }
        return 1 + size(node.left) + size(node.right);
    } ...
}
```

SortTree ADT: Implementation

- Implementation I (continued):

contains('G')

```
public boolean contains(E elem) {  
    return contains(this.root, elem);  
}
```

```
private static <E extends Comparable<E>> boolean
```

```
contains(BinTreeNode<E> node, E elem) {
```

```
    if (node == null) { return false; }
```

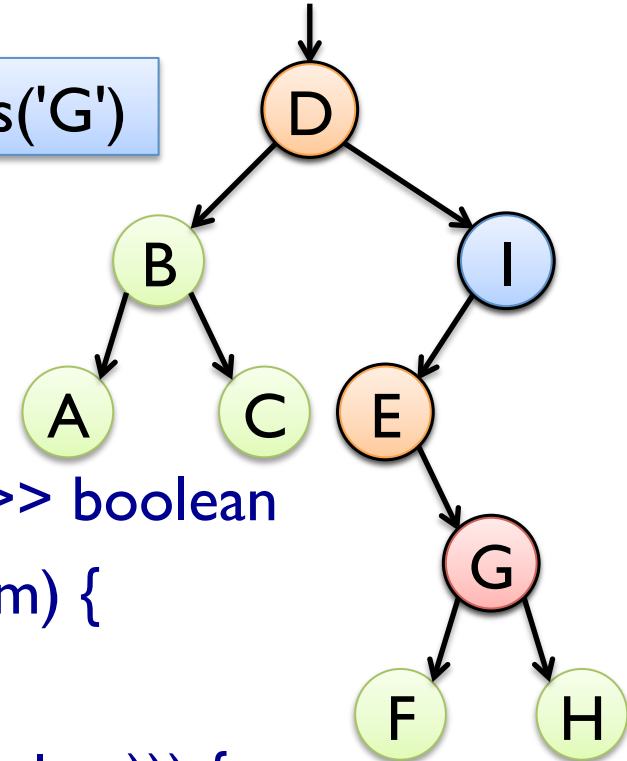
```
    switch (signum(elem.compareTo(node.elem))) {
```

```
        case -1: return contains(node.left, elem);
```

```
        case 0: return true;
```

```
        case 1: return contains(node.right, elem);
```

```
    } throw new RuntimeException("compareTo error"); } ...
```



SortTree ADT: Implementation

- Implementation I (continued):

```
public void add(E elem) { this.root = add(this.root, elem); }
private static int signum(int x) {
    return x == 0 ? 0 : (x > 0 ? 1 : -1);
}
private static <E extends Comparable<E>> BinTreeNode<E>
add(BinTreeNode<E> node, E elem) {
    if (node == null) {
        return new BinTreeNode<E>(elem, null, null);
    }
    switch (signum(elem.compareTo(node.elem))) {
        ...
    }
}
```

SortTree ADT: Implementation

- Implementation I (continued):

case -1:

```
node.left = add(node.left, elem);
```

```
return node;
```

case 0: return node;

case 1:

```
node.right = add(node.right, elem);
```

```
return node;
```

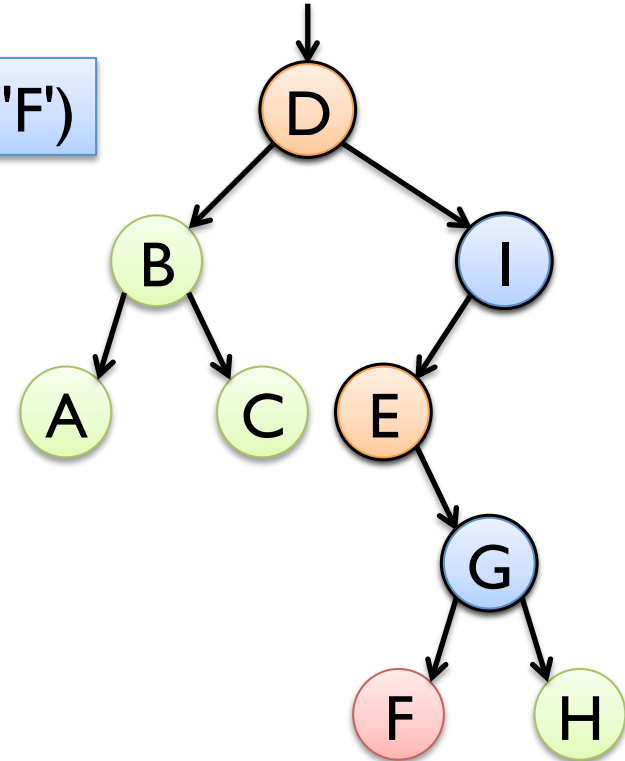
```
}
```

```
throw new RuntimeException("compareTo error");
```

```
}
```

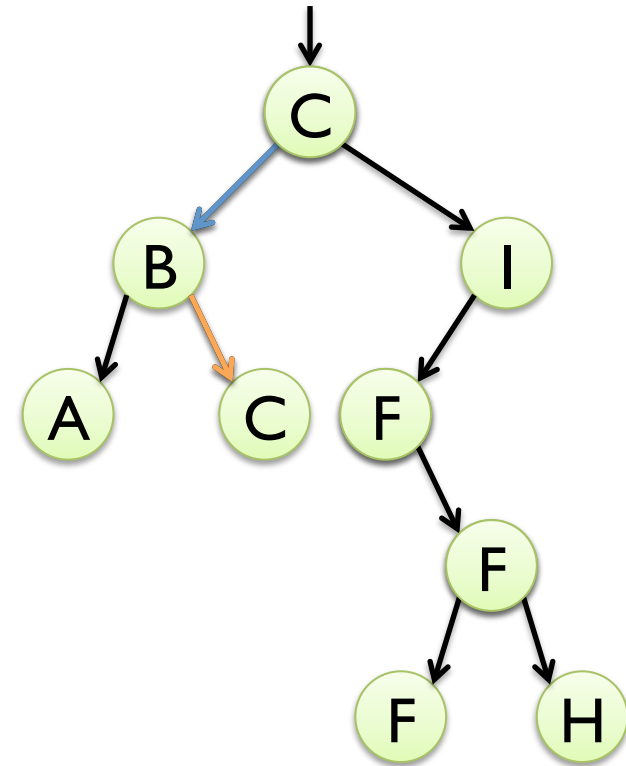
...

add('F')



Binary Sort Tree: Removal

- first, find the node (as contains or add)
- four cases for node to be removed
 1. no children `delete('H')`
 2. only a left child `delete('G')`
 3. only a right child `delete('E')`
 4. both left and right child `delete('D')`
- easy to handle 1-3:
 1. delete node `delete('D')`
 2. replace node by left child
 3. replace node by right child
 4. replace node by largest element in left subtree



SortTree ADT: Implementation

- Implementation I (continued):

```
public void remove(E elem) {
    this.root = remove(this.root, elem);
}

private static <E extends Comparable<E>> E
max(BinTreeNode<E> node) {
    return node.right == null ? node.elem : max(node.right);
}

private static <E extends Comparable<E>> BinTreeNode<E>
remove(BinTreeNode<E> node, E elem) {
    if (node == null) { return null; }
    switch (signum(elem.compareTo(node.elem))) { ...
```

SortTree ADT: Implementation

- Implementation I (continued):

```
case -1: return new BinTreeNode<E>(node.elem,  
    remove(node.left, elem), node.right);
```

```
case 0:  if (node.left == null) { return node.right; }
```

```
        if (node.right == null) { return node.left; }
```

```
        E max = max(node.left);
```

```
        return new BinTreeNode<E>(max,
```

```
            remove(node.left, max), node.right);
```

```
case 1:  return new BinTreeNode<E>(node.elem,
```

```
            node.left, remove(node.right, elem));
```

```
}
```

```
throw new RuntimeException("compareTo error"); } ...
```

SortTree ADT: Implementation

- Implementation I (continued):

```
public java.util.List<E> traverse() {  
    return traverse(this.root, new java.util.ArrayList<E>());  
}
```

```
private static <E extends Comparable<E>> List<E>  
traverse(BinTreeNode<E> node, List<E> result) {  
    if (node != null) {  
        traverse(node.left, result);  
        result.add(node.elem);  
        traverse(node.right, result);  
    }  
    return result; } ...
```

SortTree ADT: Implementation

- Implementation I (continued):

```
public void balance() { this.root = balance(this.root); }
```

```
private static <E extends Comparable<E>> E
```

```
    min(BinTreeNode<E> node) {
```

```
        return node.left == null ? node.elem : min(node.left);
```

```
    }
```

```
private static <E extends Comparable<E>> BinTreeNode<E>
```

```
    balance(BinTreeNode<E> node) {
```

```
        if (node == null) { return null; }
```

```
        int lSize = size(node.left);
```

```
        int rSize = size(node.right);
```

```
        ...
```

SortTree ADT: Implementation

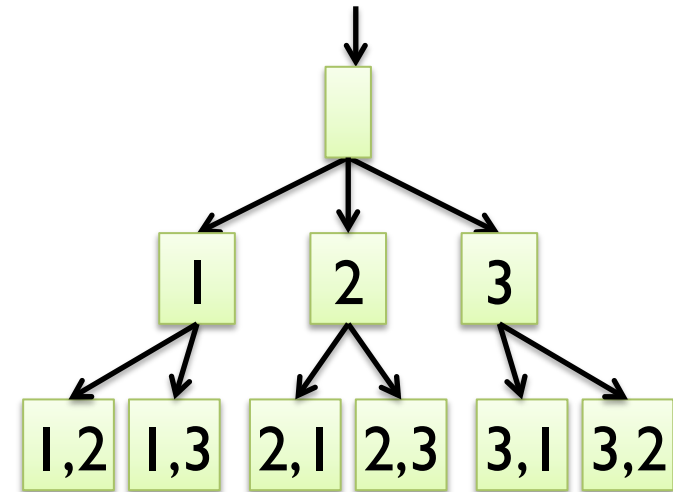
- Implementation I (continued):

```
while (lSize > rSize+1) {
    E max = max(node.left); lSize--; rSize++;
    node = new BinTreeNode<E>(max,
        remove(node.left, max), add(node.right, node.elem)); }
while (rSize > lSize+1) {
    E min = min(node.right); rSize--; lSize++;
    node = new BinTreeNode<E>(min,
        add(node.left, node.elem), remove(node.right, min)); }
return new BinTreeNode<E>(node.elem,
    balance(node.left), balance(node.right));
} } // DONE!
```

MULTIVARIATE TREES

Multivariate Trees

- general class of trees
- nodes can have any number of children
- our application:
 - k-permutations of n
 - all sequences without repetition
 - total of n elements
 - sequences of length $k < n$
- Example:
 - total of $n = 3$ elements
 - sequences of length $k = 2$



MTree ADT: Specification

- data are sequences of integers (`List<Integer>`)
- operations are defined by the following interface

```
public interface MTreeADT extends javax.swing.tree.TreeModel {  
    public Object getRoot();  
    public boolean isLeaf(Object node);  
    public int getChildCount(Object node);  
    public Object getChild(Object parent, int index);  
    public int getIndexOfChild(Object parent, Object child);  
}
```

- `TreeModel` specifies additional operations needed for `JTree`

MTree ADT: Design & Implement.

- Design: use recursive data structure
- Implementation:

```
public class MTreeNode {  
    private List<Integer> seq; // sequence of integers  
    private List<MTreeNode> children =  
        new ArrayList<MTreeNode>();  
    public MTreeNode(List<Integer> seq) { this.seq = seq; }  
    public List<Integer> getSeq() { return this.seq; }  
    public List<MTreeNode> getChildren() { return this.children; }  
    public String toString() { return this.seq.toString(); }  
}
```

MTree ADT: Implementation

- Implementation (continued):

```
public class MTree implements MTreeADT {
    private MTreeNode root = new MTreeNode(
        new ArrayList<Integer>());
    public Object getRoot() { return this.root; }
    public boolean isLeaf(Object node) {
        return this.getChildCount(node) == 0;
    }
    public int getChildCount(Object node) {
        return ((MTreeNode)node).getChildren().size();
    }
    ...
}
```

MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT { ...
    public Object getChild(Object parent, int index) {
        return ((MTreeNode)parent).getChildren().get(index);
    }
    public int getIndexOfChild(Object parent, Object child) {
        return ((MTreeNode)parent).getChildren().indexOf(child);
    }
    public void addTreeModelListener(TreeModelListener l) {}
    public void removeTreeModelListener(TreeModelListener l) {}
    public void valueForPathChanged(TreePath p, Object o) {}
    ...
}
```

MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT { ...
    private static MTree makeTree(String title) {
        MTree tree = new MTree();
        JScrollPane content = new JScrollPane(new JTree(tree));
        JFrame window = new JFrame(title);
        window.setContentPane(content);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setSize(400,400);
        window.setVisible(true);
        return tree;
    } ...
```

MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT { ...  
    public static void main(String[] args) {  
        int n = 4;  
        int k = 3;  
        MTree tree1 = makeTree("MTree queue");  
        tree1.expandQueue(n,k);  
        MTree tree2 = makeTree("MTree stack");  
        tree2.expandStack(n,k);  
        MTree tree3 = makeTree("MTree recursive");  
        tree3.expandRecursively(n,k);  
    } ...
```

MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT {
```

```
...
```

```
private void expandRecursively(int n, int k) {
```

```
    expandRecursively(this.root, n, k);
```

```
}
```

```
private List<Integer> copyAdd(List<Integer> seq, int i) {
```

```
    List<Integer> copySeq = new ArrayList<Integer>(seq);
```

```
    copySeq.add(i);
```

```
    return copySeq;
```

```
}
```

```
...
```

MTree ADT: Implementation

- Implementation (continued):

```
private void expandRecursively(MTreeNode node, int n, int k) {  
    List<Integer> seq = node.getSeq();  
    if (seq.size() < k) {  
        for (int i = 1; i <= n; i++) {  
            if (!seq.contains(i)) {  
                List<Integer> newSeq = copyAdd(seq, i);  
                MTreeNode child = new MTreeNode(newSeq);  
                node.getChildren().add(child);  
                expandRecursively(child, n, k);  
            }  
        }  
    }  
}
```

...

MTree ADT: Implementation

- Implementation (continued):

```
private void expandStack(int n, int k) {
    Stack<MTreeNode> stack = new LinkedStack<MTreeNode>();
    stack.push(this.root);
    while (!stack.isEmpty()) {
        MTreeNode node = stack.pop();
        List<Integer> seq = node.getSeq();
        if (seq.size()<k) { for (int i=1; i<=n; i++) { if (!seq.contains(i)) {
            MTreeNode child = new MTreeNode(copyAdd(seq, i));
            node.getChildren().add(child);
            stack.push(child);
        } } } ...
    }
}
```

MTree ADT: Implementation

- Implementation (continued):

```
private void expandQueue(int n, int k) {
    Queue<MTreeNode> queue = new LinkedList<MTreeNode>();
    queue.offer(this.root);
    while (!queue.isEmpty()) {
        MTreeNode node = queue.poll();
        List<Integer> seq = node.getSeq();
        if (seq.size()<k) { for (int i=1; i<=n; i++) { if (!seq.contains(i)) {
            MTreeNode child = new MTreeNode(copyAdd(seq, i));
            node.getChildren().add(child);
            queue.offer(child);
        } } } // DONE!
```

PROJECT PART 2

Organizational Details

- exam project consisting of 2 parts
- both parts have to be passed to pass the course
- projects must be done individually, so no co-operation
- you may talk about the problem and ideas how to solve them
- deliverables:
 - written 4 page report as specified in project description
 - handed in as a single PDF file
 - deadline: Friday, January 10, 2014, 23:59
- ENOUGH - now for the ABSTRACT part ...

Board Games: Tic Tac Toe & Co

- Task 0: Preparation
 - download and understand existing framework
 - integrate (and fix) **TTTBoard** and **Coordinate**
- Task 1: Implement ADT
 - design and implement **TTTGameTree** (and node class)
 - need to cooperate with **GameTreeDisplay** and **TTTExplorer**
- Task 2: Building the Game Tree
 - build a tree with one node representing the initial game
 - add successors of a game state as children
 - keep going until one player wins or a draw is reached
 - check you progress using **TTTExplorer**

Board Games: Tic Tac Toe & Co

- Task 3 (optional): Reducing the Size of the Game Tree
 - reuse nodes for identical game states
 - consider rotational symmetry?
 - consider mirroring?
- Task 4 (optional): Artificial Intelligence
 - fully expanded game tree useful to implement AI player
 - AI player tries to develop towards winning situations
 - AI player tries to avoid losing situations