DM550 / DM857
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

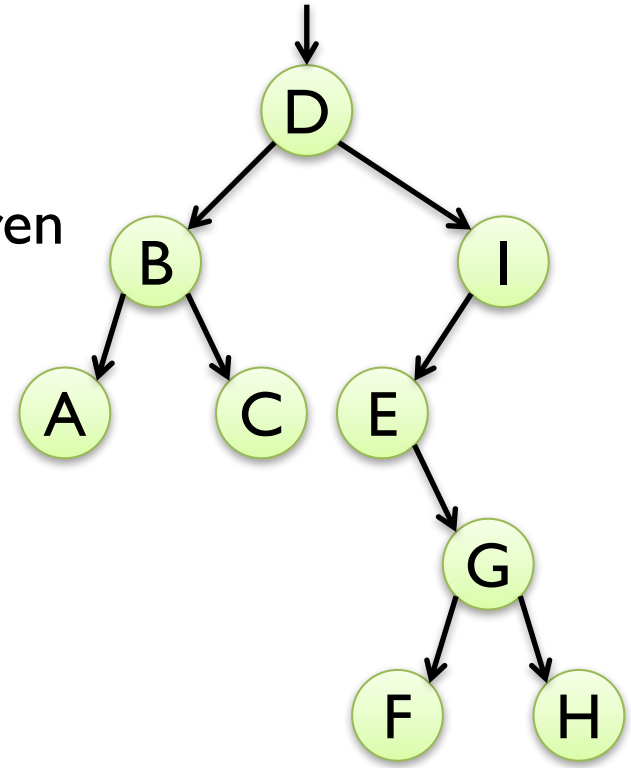
http://imada.sdu.dk/~petersk/DM550/

http://imada.sdu.dk/~petersk/DM857/

ABSTRACT DATA TYPES FOR (BINARY) TREES

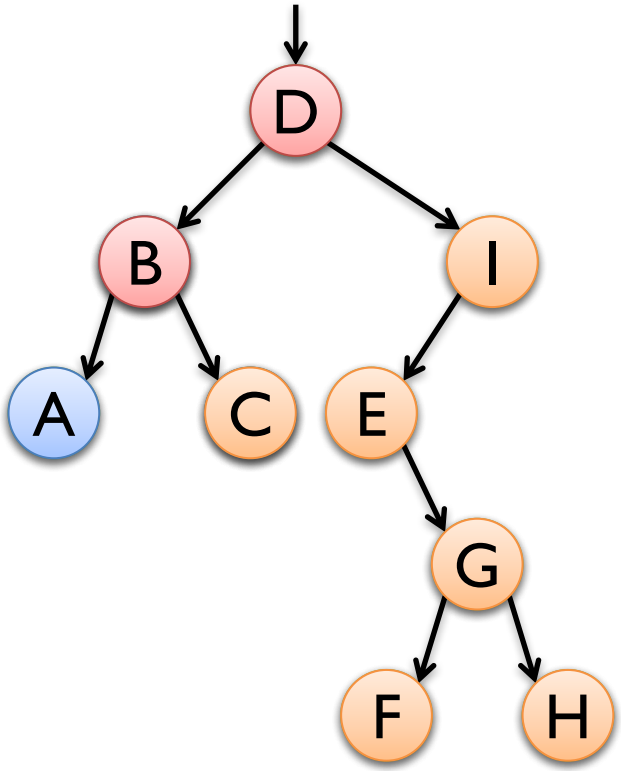
Trees

- 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 1: use recursive data structure
  - based on representing tree nodes by BinTreeNode<E>
- Implementation 1:

```
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;
    }
}
```

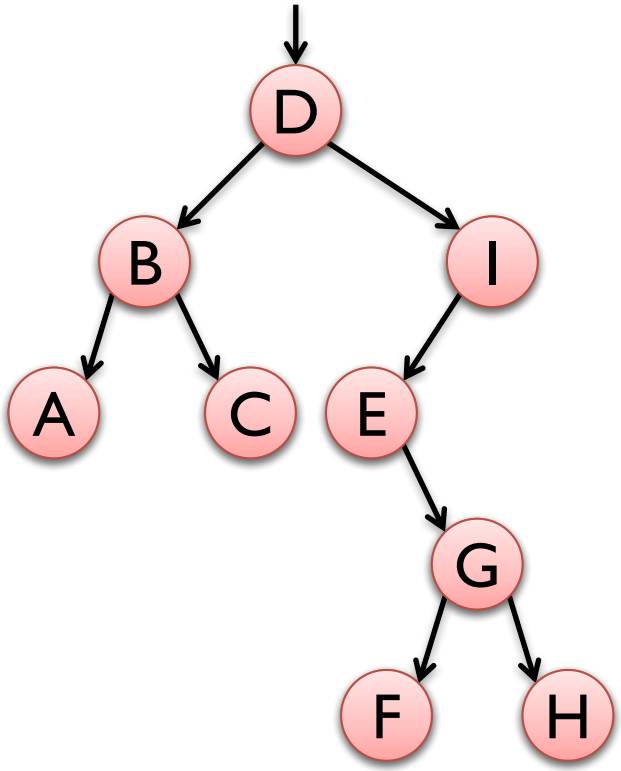
BinTree ADT: Implementation 1

- Implementation 1 (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+depth(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

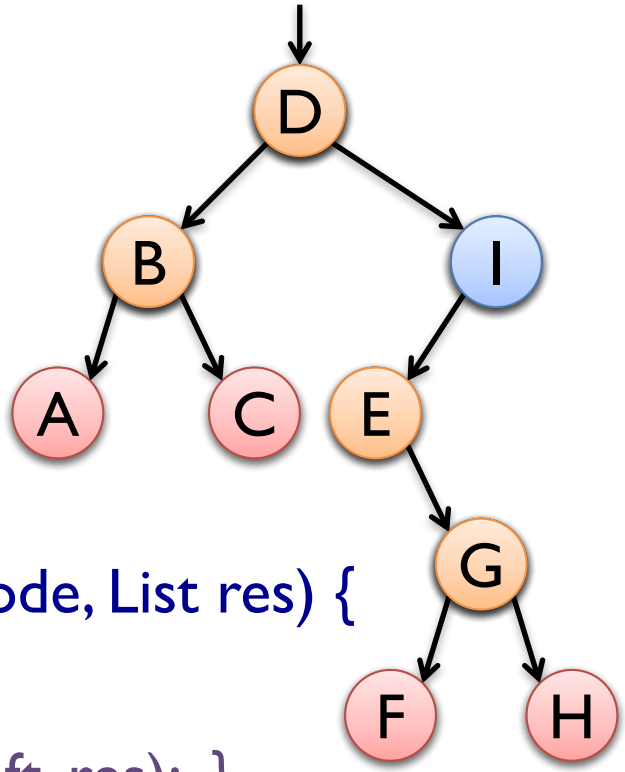
BinTree ADT: Implementation 1

- Implementation 1 (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
  1. pre-order

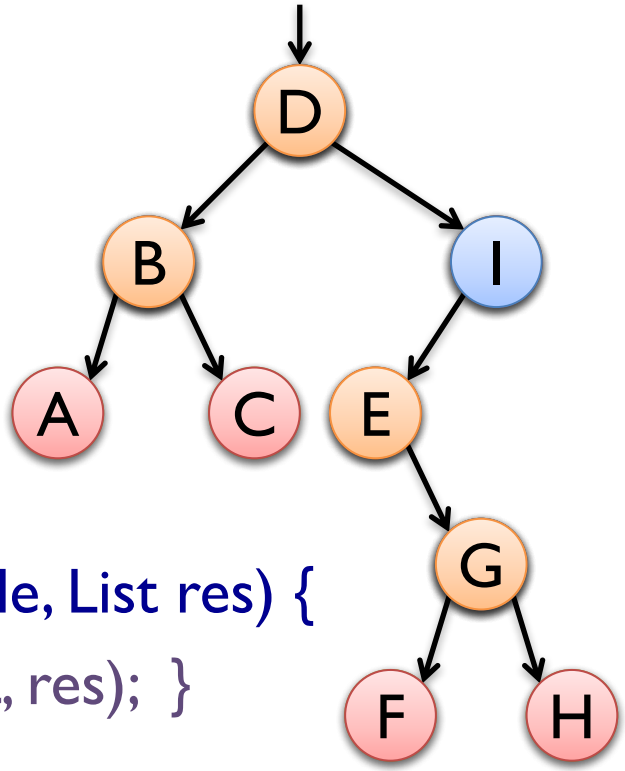



```
public static void preOrder(BinTreeNode node, List res) {
    res.add(node.elem);
    if (node.left != null)   {  preOrder(node.left, res);  }
    if (node.right != null) {  preOrder(node.right, res);  }
}
```

D	B	A	C	I	E	G	F	H

Binary Tree Traversal

- traversal can be either
  - depth-first
  - breadth-first
- three standard depth-first traversals
  2. in-order

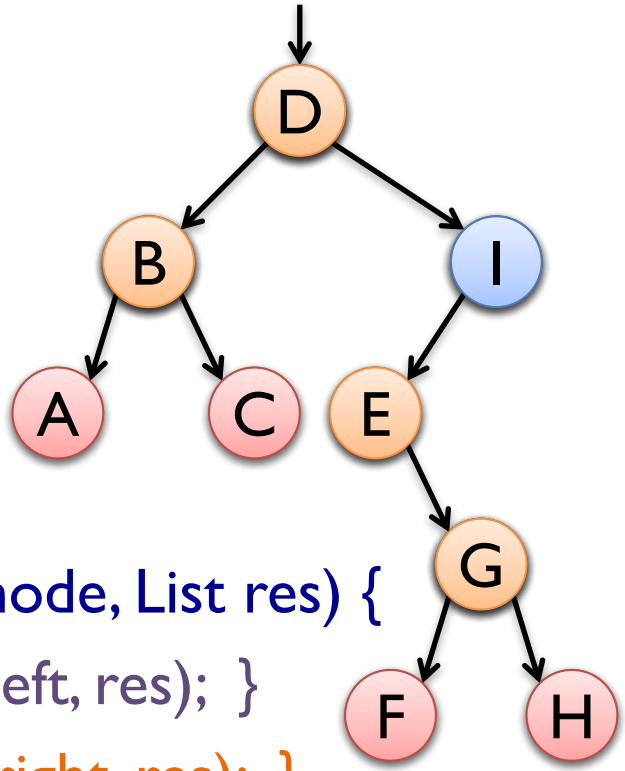



```
public static void inOrder(BinTreeNode node, List res) {
    if (node.left != null)   {  inOrder(node.left, res);  }
    res.add(node.elem);
    if (node.right != null) {  inOrder(node.right, res);  }
}
```

A	B	C	D	E	F	G	H	I

Binary Tree Traversal

- traversal can be either
  - depth-first
  - breadth-first
- three standard depth-first traversals
  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);
}
```

A	C	B	F	H	G	E	I	D

BinTree ADT: Implementation 1

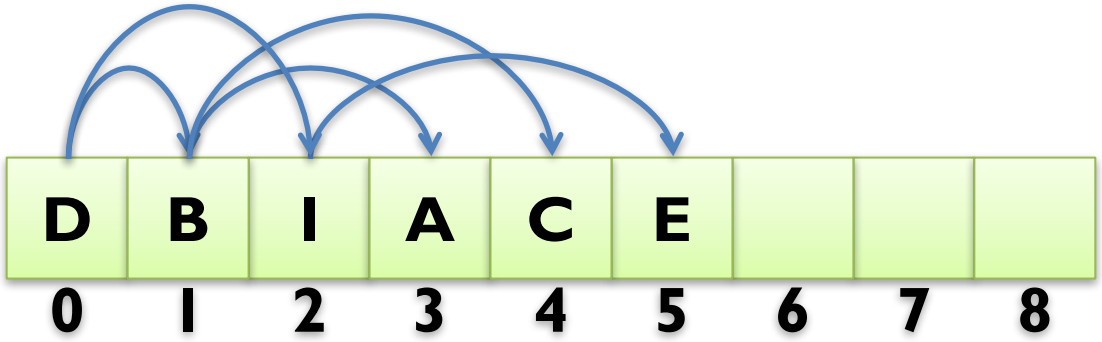
- Implementation 1 (continued):

```
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 in max(int a, int b) {  return a > b ? a : b;  }
    …
}
```

BinTree ADT: Implementation 2

- Implementation 2 (continued):

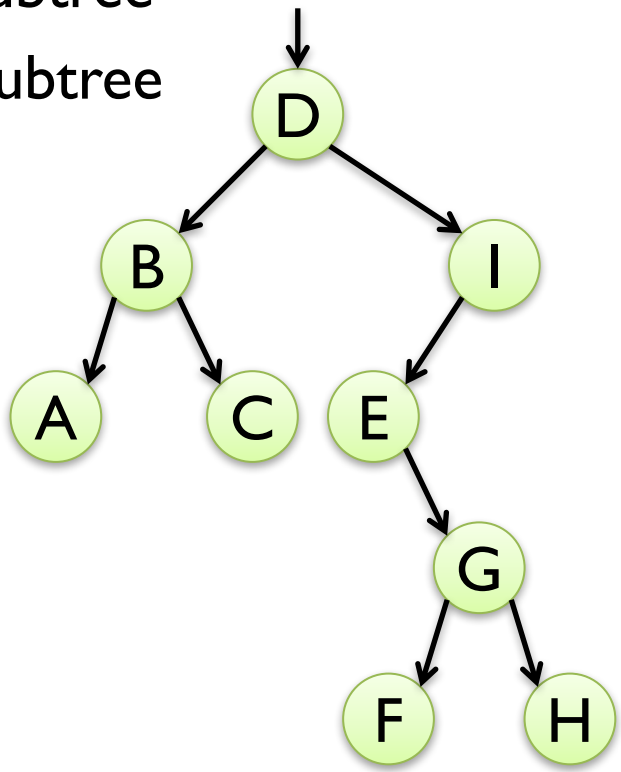
```
public List<E> preOrder() {
    return preOrder(0, new ArrayList<E>());
}
private List<E> preOrder(int index, java.util.List<E> res) {
    E elem = this.data.get(index);
    if (elem != null) {
        res.add(elem);
        preOrder(2*index+1, res);    preOrder(2*index+2, res);
    }
    return 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);         // tremove elem from tree
    public List<E> traverse();          // in-order traversal
    public void balance();              // balance the tree
}
```

SortTree ADT: Design & Implement.

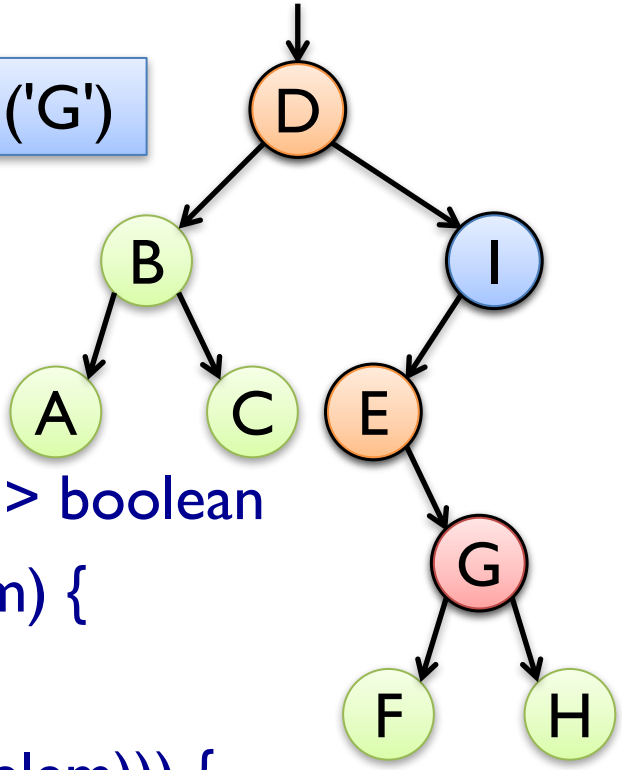
- Design: use recursive data structure
  - reuse class BinTreeNode<E>
- Implementation 1:

```
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 1 (continued):

```
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

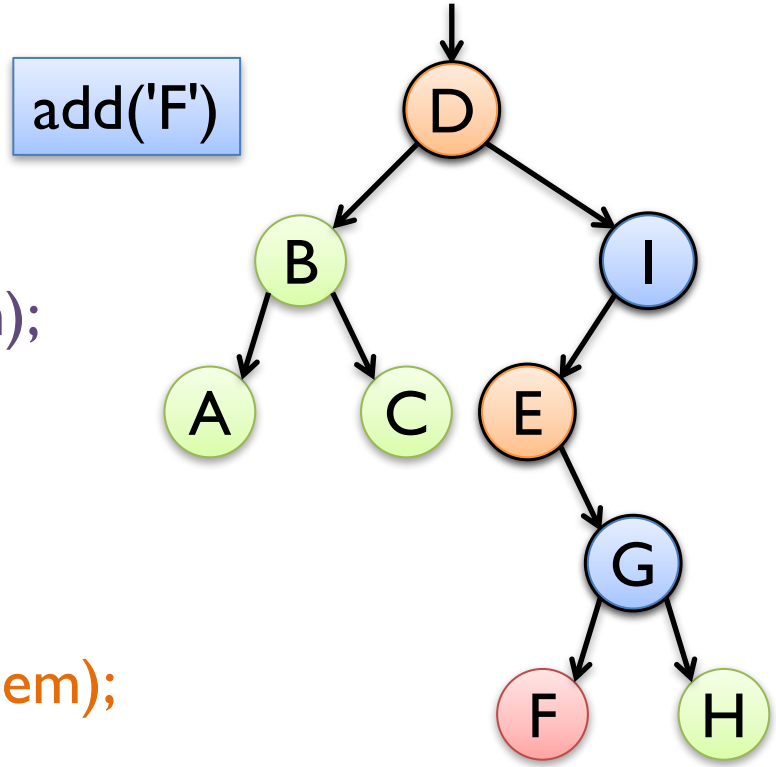
- Implementation 1 (continued):

```
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 1 (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");
    }
    …
```



Binary Sort Tree: Removal

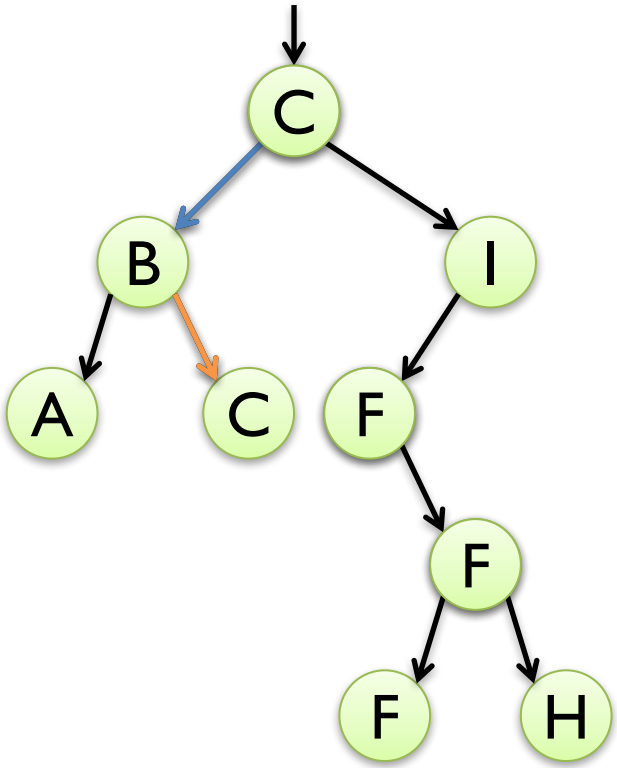
- first, find the node (as contains or add)
- four cases for node to be removed
  1. no children
  2. only a left child
  3. only a right child
  4. both left and right child
- easy to handle 1-3:
  1. delete node
  2. replace node by left child
  3. replace node by right child
  4. replace node by largest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

delete('H')

delete('G')

delete('E')

delete('D')



SortTree ADT: Implementation

- Implementation 1 (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 1 (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 1 (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 1 (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 1 (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!
```

NON-SEQUENTIAL COLLECTION CLASSES

Set ADT: Specification

- interface Set<E> extends Collection<E>
- unordered sequences of objects without duplicates
- no additional operations, as Collection<E> already specifies
  - isEmpty, size, contains, add, remove, …
- no index-based access to elements, as order undefined
- elements MUST implement equals and hashCode correctly:
  1. for two elements e1 and e2 that are equal, both e1.equals(e2) and e2.equals(e1) must return true
  2. for two elements e1 and e2 that are equal, we must have e1.hashCode() == e2.hashCode()
  3. for two elements e1 and e2 that are NOT equal, both e1.equals(e2) and e2.equals(e1) must return false

Set ADT: Example

- Example (intersecting two sets):

```
int[] n1 = new int[] {1, 2, 3, 5, 7, 11, 13};
Set<Integer> set1 = new HashSet<Integer>(Arrays.asList(n1));
int[] n2 = new int[] {1, 3, 5, 7, 9};
Set<Integer> set2 = new HashSet<Integer>(Arrays.asList(n2));
Set<Integer> set3 = new HashSet<Integer>(set1);
set3.retainAll(set2);
```

- retainAll modifies set3, thus we have (informally):
  - set1 == {1, 2, 3, 5, 7, 11, 13}
  - set2 == {1, 3, 5, 7, 9}
  - set3 == {1, 3, 5, 7}

Iterator ADT: Example 2

- Example (iterate over all elements of a HashSet):

```
Set<String> set = new HashSet<String>();
set.add("Hej");
set.add("hej");
set.add("Hej");
Iterator<String> iter = set.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
```

- prints the two strings in some undefined order

Interface Comparator

- allows to specify how to compare elements

```
public interface Comparator<E> {
    public int compare(E o1,  E o2);      // compare o1 and o2
    public boolean equals(Object obj);  // equals other Comparator?
}
```

- compare behaves like o1.compareTo(o2) from Comparable<E>
  - < 0 for o1 less than o2
  - ==0 for o1 equals o2
  - > 0 for o1 greater than o2
- Comparable defines *natural* ordering
- Comparator can define additional orderings

Set ADT: TreeSet Implementation

- TreeSet implements sets as special sort trees (Red-Black Trees)
- elements are compared to according to natural ordering
- Example:

```
public class Compi implements Comparator<Integer> {
    public int compare(Integer i1, Integer i2) {
    return i2.compareTo(i1); }
    public boolean equals(Object other) { return false; } } …
TreeSet<Integer> set1 = new TreeSet<Integer>();
set1.add(23); set1.add(42); set1.add(-3);
for (int n : set1) { System.out.print(" "+n); } // -3 23 42
TreeSet<Integer> set2 = new TreeSet<Integer>(new Compi());
set2.addAll(set1);
for (int n : set2) { System.out.print(" "+n); } // 42 23 -3
```

Set ADT: Implementations

- HashSet based on hash tables
  - very good choice if order really does not matter
- LinkedHashSet based on hash tables + linked list
  - in addition to hash table keeps track of insertion order
  - useful for keeping algorithms deterministic
- TreeSet based on special sort trees
  - implements the SortedSet<E> interface
  - useful for ordered sequences without duplicates
  - can use Comparators for different orderings
  - also useful when e.g. hash code not available

Map ADT: Specification

- maps work like dictionaries in Python
- interface Map<K,V> specifies standard operations
  - boolean isEmpty(); // true, if there are no mappings
  - int size(); // returns number of mappings
  - boolean containsKey(Object key); // is key mapped?
  - boolean containsValue(Object value); // is value mapped?
  - V get(Object key); // return mapped value or null
  - V put(K key,V value); // add mapping from key to value
  - Set<K> keySet(); // set of all keys
  - Collection<V> values(); // collection of all values
  - Set<Map.Entry<K,V>> entrySet(); // (key,value) pairs
  - clear, putAll, remove, …

Map ADT: Example

- Example (using and modifying a phone directory):

```
Map<String,Integer> dir = new HashMap<String,Integer>();
dir.put("petersk", 65502327);  dir.put("bwillis", 55555555);
for (String key : dir.keySet()) {
    System.out.println(key+" -> "+dir.get(key));
}
for (Map.Entry<String,Integer> entry : dir.entrySet()) {
    System.out.println(entry.getKey()+" -> "+entry.getValue());
    entry.setValue(12345678);
}
dir.keySet().remove("bwillis");
System.out.println(dir);      // only petersk is mapped
```

Hash Table

- a hash table uses the hashCode method to map objects to ints
- objects are stored in an array
- the position of the object is determined by its hash code modulo the length of the array
- Example: if o has hash code 10 and array has length 7,
  o is stored at position 10 % 7 == 3
- more in **DM507 Algorithms and Data Structures**
- efficient for get and put
- assuming that hashCode is implemented in a useful way
- if two or more objects have the same hash code, the array stores a list of objects in that position

Map ADT: Implementations

- HashMap based on hash tables
  - very good choice if order does not matter
- LinkedHashMap based on hash tables + linked list
  - in addition to hash table keeps track of insertion order
  - useful for keeping algorithms deterministic
- TreeMap based on special sort trees
  - implements the SortedMap<K,V> interface
  - useful for ordered mappings
  - can use Comparators for different orderings
  - also useful when e.g. hash code not available
- Hashtable based on hash tables
  - old implementation – only use for synchronization