DM537
Object-Oriented Programming

Peter Schneider-Kamp

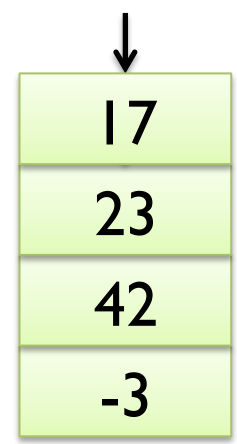
petersk@imada.sdu.dk

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

ABSTRACT DATA TYPES FOR STACKS & QUEUES

Stacks

- stacks are special sequences, where elements are only added and removed at one end
- imagine a stack of paper on a desk
- many uses:
  - postfix calculator
  - activation records
  - depth-first tree traversals
  - …
- basic stack operations are
  - looking at the top of the stack
  - removing the top-most element
  - adding an element to the top of the stack



Stack ADT: Specification

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

```
public interface Stack<E> {
    public boolean isEmpty();        // is stack empty?
    public E peek();                 // look at top element
    public E pop();                  // remove top element
    public void push(E elem);        // add top element
}
```

Stack ADT: Design 1

- Design 1: use dynamic array
  - the top of the stack is the end of the list
  - in other words, num specifies the top position
  - pushing corresponds to adding at the end
  - poping corresponds to removing at the end

Stack ADT: Implementation 1

- Implementation 1:

```
public class DynamicArrayStack<E> implements Stack<E> {
    private int limit;              // maximal number of elements
    private E[] data;               // elements of the list
    private int num = 0;            // current number of elements
    public DynamicArrayStack(int limit) {
        this.limit = limit;
        this.data = (E[]) new Object[limit];
    }
    public boolean isEmpty() {  return this.num == 0;  }
    …
}
```

Stack ADT: Implementation 1

- Implementation 1 (continued):

```
public class DynamicArrayStack<E> implements Stack<E> {   …
    public E peek() {
        if (this.isEmpty()) {  throw new RuntimeException("es");  }
        return this.data[this.num-1];
    }
    public E pop() {
        E result = this.peek();
        num--;
        return result;
    }   …
}
```

Stack ADT: Implementation 1

- Implementation 1 (continued):

```
public class DynamicArrayStack<E> implements Stack<E> {   …
    public void push(E elem) {
        if (this.num >= this.limit) {
            E[] newData = (E[]) new Object[2*this.limit];
            for (int j = 0; j < limit; j++) {  newData[j] = data[j];  }
            this.data = newData;
            this.limit *= 2;
        }
        this.data[num++] = elem;
    }
}
```

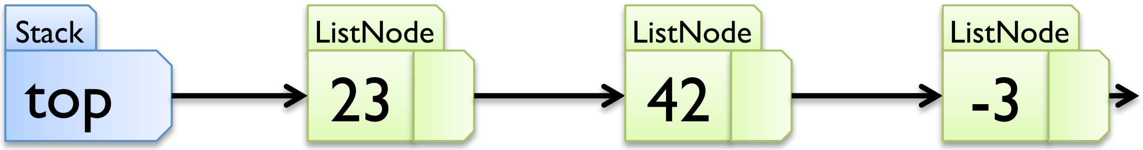
Stack ADT: Design & Implement. 2

- Design 2: reuse dynamic array list (ArrayList<E>)
- Implementation 2:

```
public class ArrayListStack<E> implements Stack<E> {
    private List<E> list = new ArrayList<E>();
    public boolean isEmpty() {  return this.list.isEmpty();  }
    public E peek() {  return this.list.get(this.list.size()-1);  }
    public E pop() {  return this.list.remove(this.list.size()-1);  }
    public void push(E elem) {  this.list.add(elem);  }
}
```

Stack ADT: Design 3

- Design 3: use recursive data structure
  - linked lists have cheap insert and remove operations
  - adding at the end requires running to the end
  - represent top as the beginning of the "list"
- reuse linked list node class (ListNode<E>)
- with dynamic arrays, sometimes need to copy full array
- with linked list, always constant time operations



Stack ADT: Implementation 3

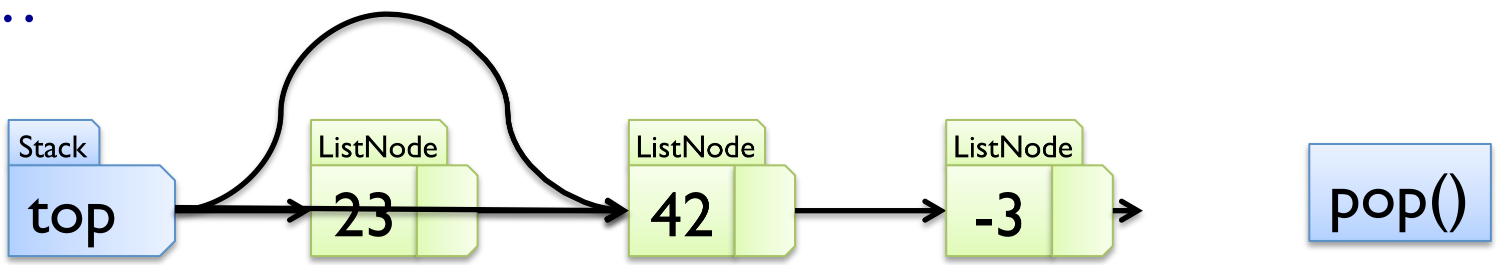
- Implementation 3:

```
public class LinkedStack<E> implements Stack<E> {
    private ListNode<E> top = null;    // top of the stack
    public boolean isEmpty() {  return this.top == null;  }
    public E peek() {
        if (this.isEmpty()) {  throw new RuntimeException("es");  }
        return this.top.get(0);
    }
    …
}
```

Stack ADT: Implementation 3

- Implementation 3 (continued):

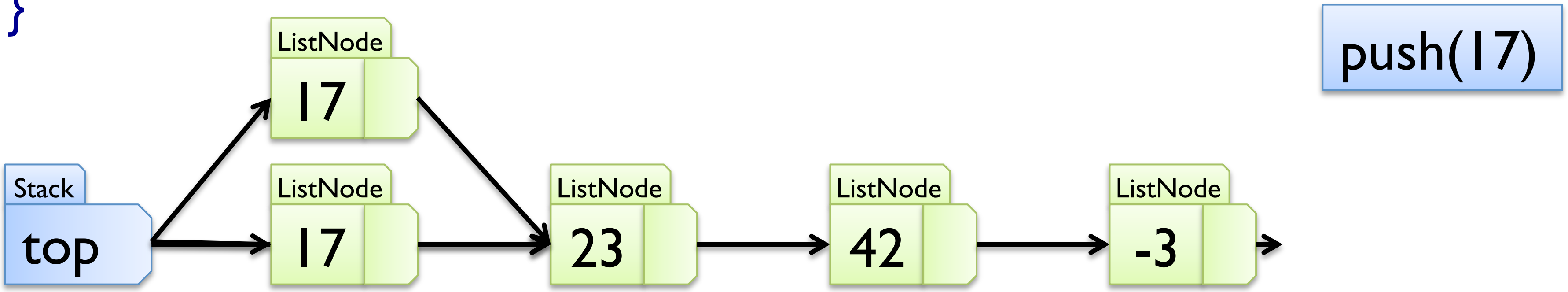
```
public class LinkedStack<E> implements Stack<E> {
    …
    public E pop() {
        E result = this.peek();
        this.top = this.top.getNext();
        return result;
    }
    …
}
```



Stack ADT: Implementation 3

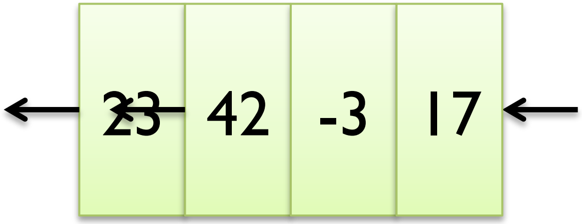
- Implementation 3 (continued):

```
public class LinkedStack<E> implements Stack<E> {
    private ListNode<E> top = null;    // top of the stack
    …
    public void push(E elem) {
        this.top = new ListNode<E>(elem, this.top);
    }
}
```



Queues

- queues are special sequences, where elements are added on one and removed at the other end
- imagine a waiting line in the supermarket
- many uses:
  - network send/receive buffers
  - process scheduling
  - breadth-first tree traversals
  - …
- basic queue operations are
  - looking at the beginning of the queue
  - removing the first element
  - adding an element to the end of the queue



Queue ADT: Specification

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

```
public interface Queue<E> {
    public boolean isEmpty();          // is queue empty?
    public E peek();                   // look at first element
    public E poll();                   // remove first element
    public boolean offer(E elem);      // true, if element added
        // at end of queue; false, if queue is full
}
```

Queue ADT: Design & Implement. I

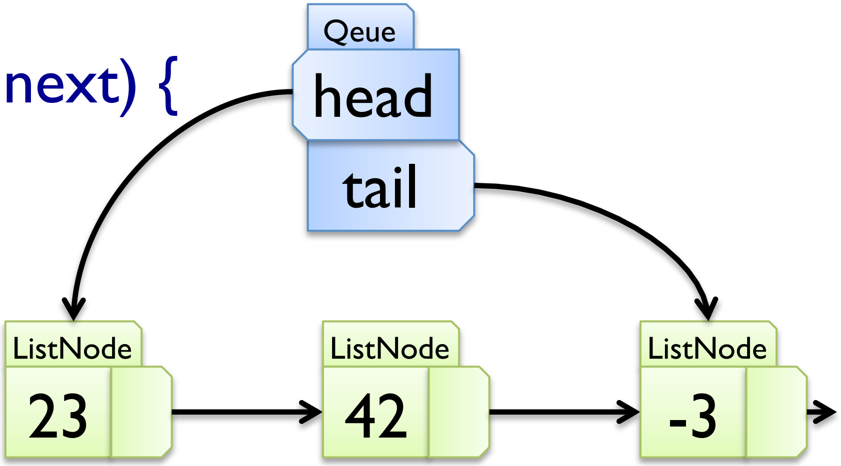
- Design I: reuse dynamic array list (ArrayList<E>)
- Implementation I:

```
public class ArrayListQueue<E> implements Queue<E> {
    private List<E> list = new ArrayList<E>();
    public boolean isEmpty() {  return this.list.isEmpty();  }
    public E peek() {  return this.list.get(0);  }
    public E poll() {  return this.list.remove(0);  }
    public boolean offer(E elem) {
        this.list.add(elem);
        return true;
    }
}
```

Queue ADT: Design & Implement. 2

- Design 2: use recursive data structure
  - use two references instead of one
  - one reference to end of queue
  - one reference to beginning of queue
- reuse & extend linked list node class (ListNode<E>)
- Implementation 2:

```
public class ListNode<E> {   …
    public void setNext(ListNode<E> next) {
        this.next = next;
    }
}
```



Queue ADT: Implementation 2

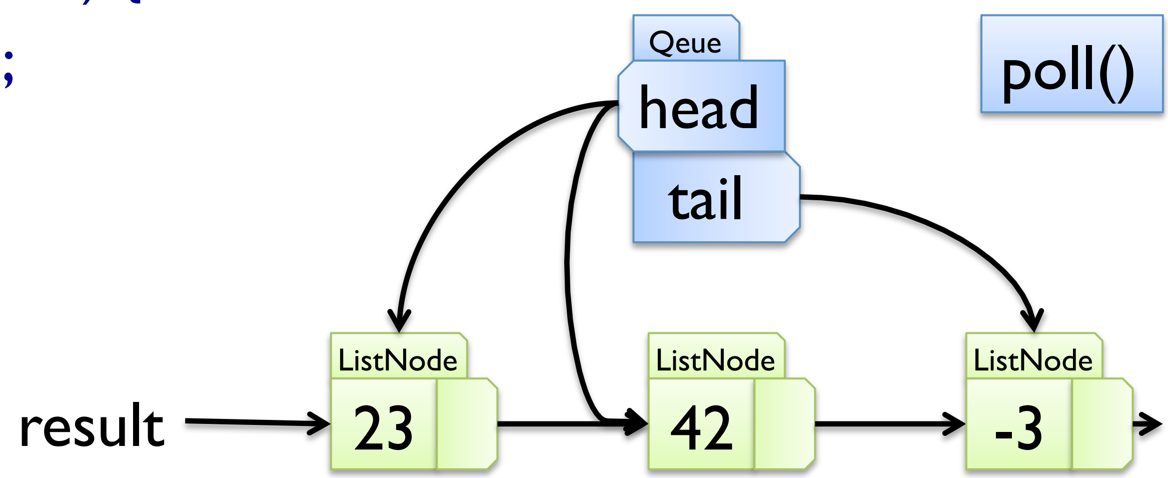
- Implementation 2 (continued):

```
public class LinkedQueue<E> implements Queue<E> {
    private ListNode<E> head = null;          // beginning
    private ListNode<E> tail = null;          // end
    public boolean isEmpty() {
        return this.head == null;
    }
    public E peek() {
        return this.head.get(0);
    }
    …
}
```

Queue ADT: Implementation 2

- Implementation 2 (continued):

```
public class LinkedQueue<E> implements Queue<E> {   …
    public E poll() {
        E result = this.peek();
        this.head = this.head.getNext();
        if (this.head == null) {
            this.tail = null;
        }
        return result;
    }
    …
}
```

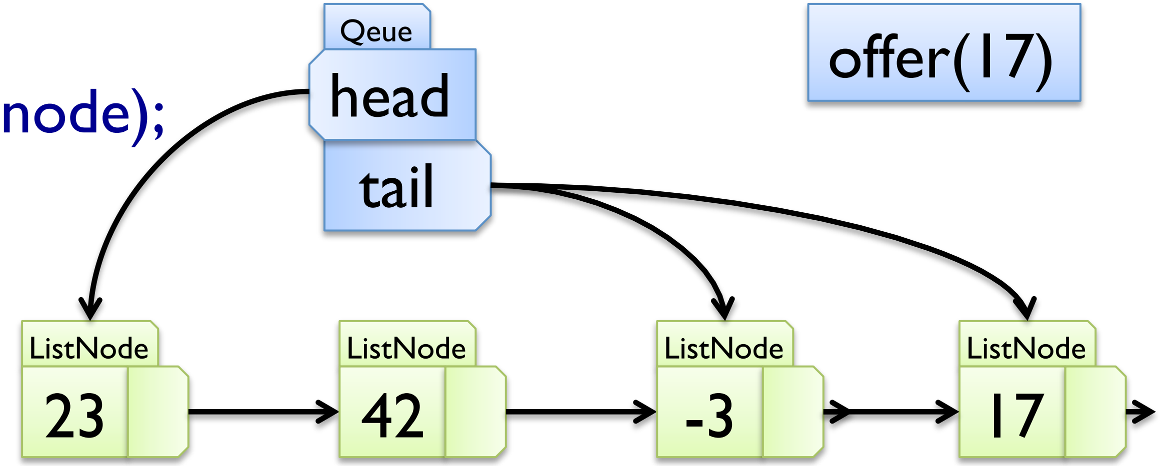


Queue ADT: Implementation 2

- Implementation 2 (continued):

```
public class LinkedQueue<E> implements Queue<E> {   …
    public boolean offer(E elem) {
        ListNode<E> node = new ListNode<E>(elem, null);
        if (this.head == null) {
            this.head = this.tail = node;
        } else {
            this.tail.setNext(node);
            this.tail = node;
        }
        return true;
    } }
```

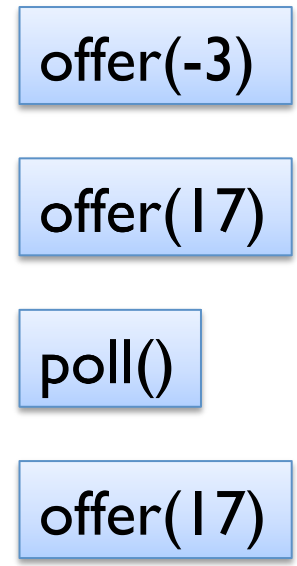
offer(17)

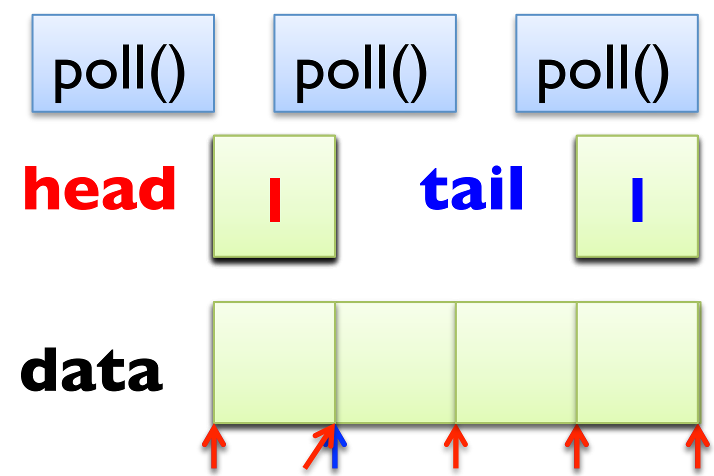


Queue ADT: Design & Implement. 3

- Design 3: use a fixed length array
  - use two indices denoting beginning and end
  - wrap around end of array
- very efficient (memory and runtime – no objects!)
- Implementation 3:

```
public class RingQueue<E> implements Queue<E> {
    private int limit;
    private int head = 0;        // beginning
    private int tail = 0;        // end
    private E[] data;
    …
}
```





Queue ADT: Implementation 3

- Implementation 3 (continued):

```
public class RingQueue<E> implements Queue<E> {   …
    private int head = 0;       // beginning
    private int tail = 0;       // end
    private E[] data;
    public boolean isEmpty() {  return this.head == this.tail;  }
    public E peek() {
        if (this.isEmpty()) {  throw new RuntimeException("eq");  }
        return this.data[this.head];
    }
    …
}
```

Queue ADT: Implementation 3

- Implementation 3 (continued):

```
public class RingQueue<E> implements Queue<E> {
    …
    public E poll() {
        E result = this.peek();
        this.head = (this.head+1) % this.limit;
        return result;
    }
    …
}
```

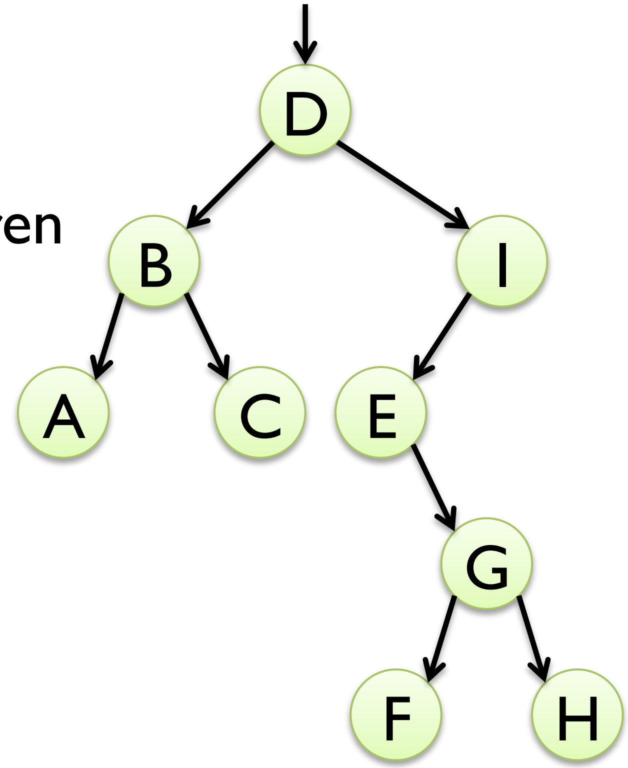
Queue ADT: Implementation 3

- Implementation 3 (continued):

```
public class RingQueue<E> implements Queue<E> {   …
   public boolean offer(E elem) {
      int newTail = (this.tail+1) % this.limit;
      if (newTail == this.head) {
         return false;               // full
      }
      this.data[this.tail] = elem;
      this.tail = newTail;
      return true;
   }   …
}
```

ABSTRACT DATA TYPES FOR (BINARY) TREES

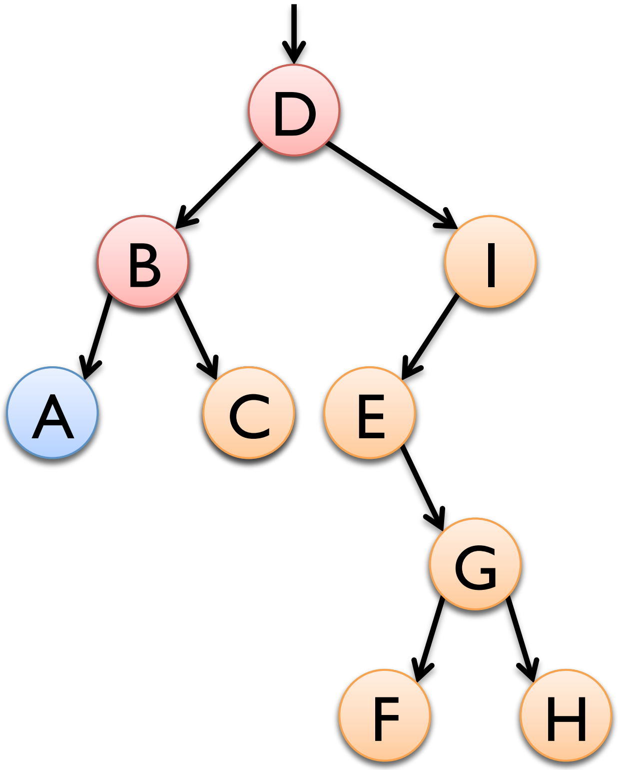
Trees



- 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:

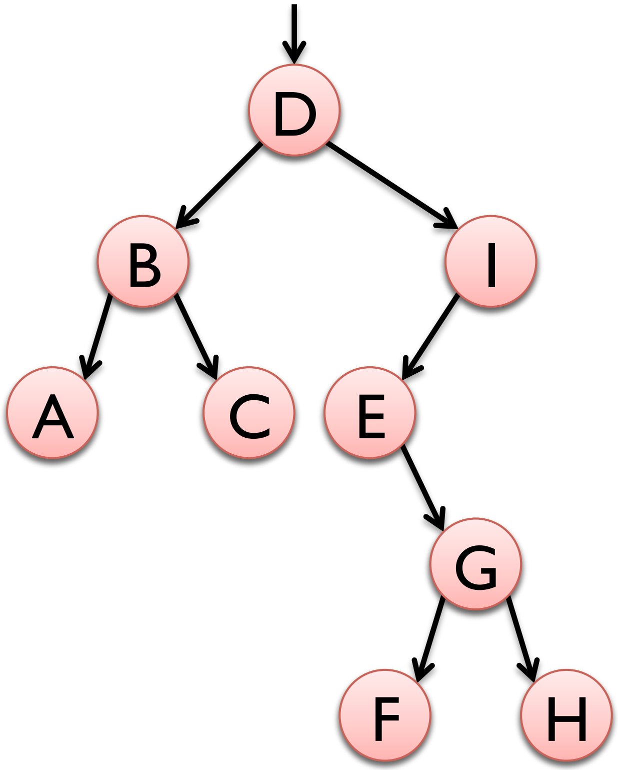
```
public class BinTreeNode<E> {
    public E elem;
    public BinTreeNode<E> left, right;
    public BinTreeNode(E elem, BinTreeNode<E> left,
                                        BinTreeNode<E> right) {
        this.elem = elem;
        this.left = left;  this.right = right;
    }
}
```

BinTree ADT: Implementation 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



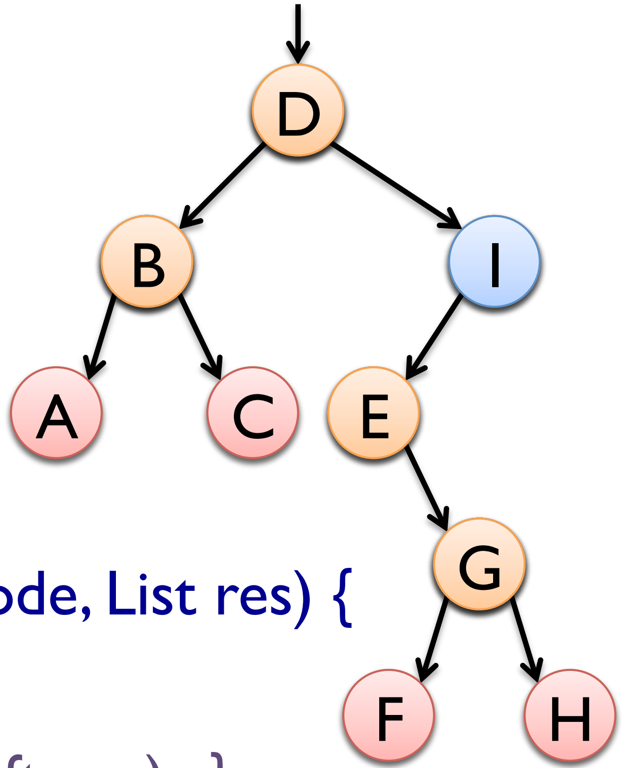
- depth of the root is 0
- depth of other nodes is 1+depth(parent)
- Example:
  - 0
  - 1
  - 2
  - 3
  - 4
- height of a subtree is maximal depth of any of its nodes
- Example: height of tree (=subtree starting in D) is 4

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

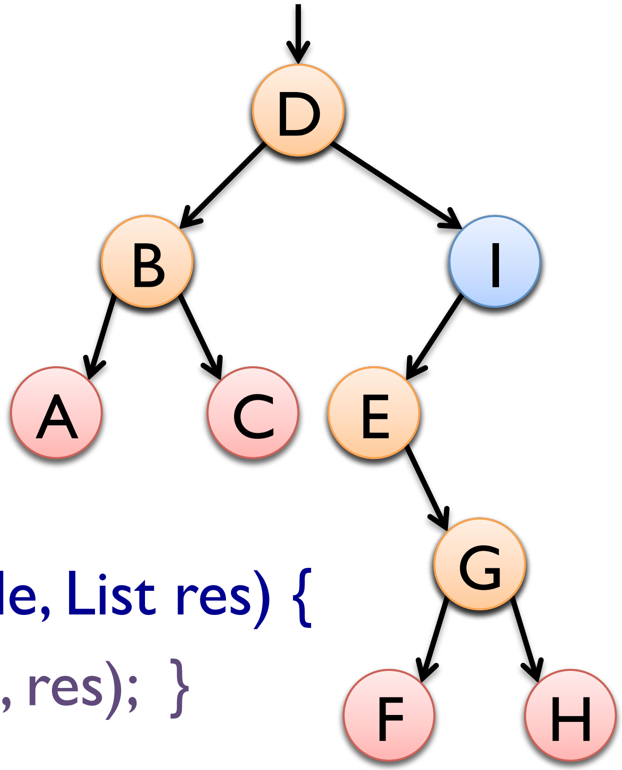


- traversal can be either
  - depth-first
  - breadth-first
- three standard depth-first traversals
  1. pre-order

```
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



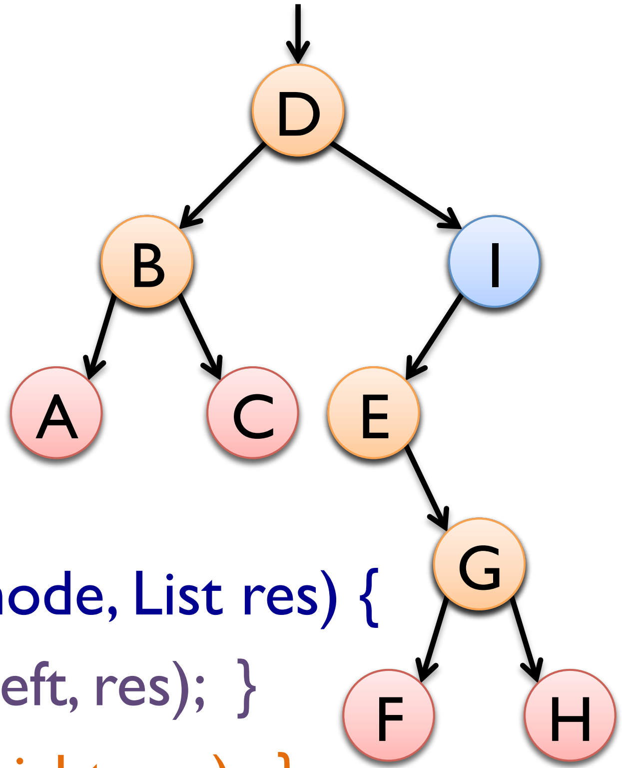
- traversal can be either
  - depth-first
  - breadth-first
- three standard depth-first traversals
  2. in-order

```
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

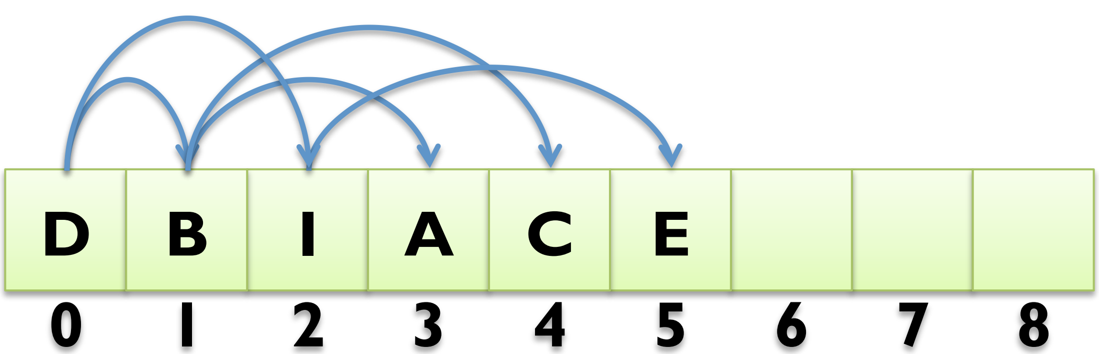
- Implementation 1 (continued):

```
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):

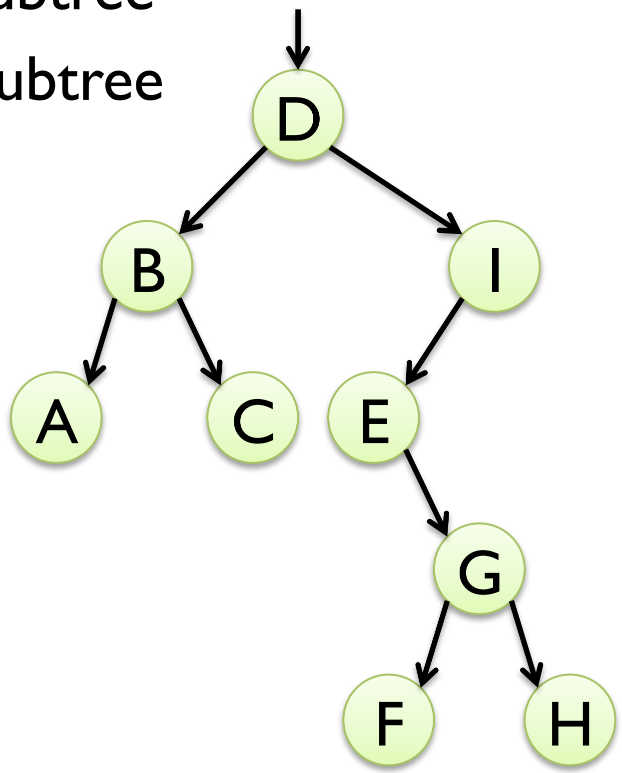
```
public List<E> preOrder() {
    return preOrder(0, new ArrayList<E>());
}
private List<E> preOrder(int index, java.util.List<E> res) {
    E elem = this.data.get(index);
    if (elem != null) {
        res.add(elem);
        preOrder(2*index+1, res);    preOrder(2*index+2, res);
    }
    return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

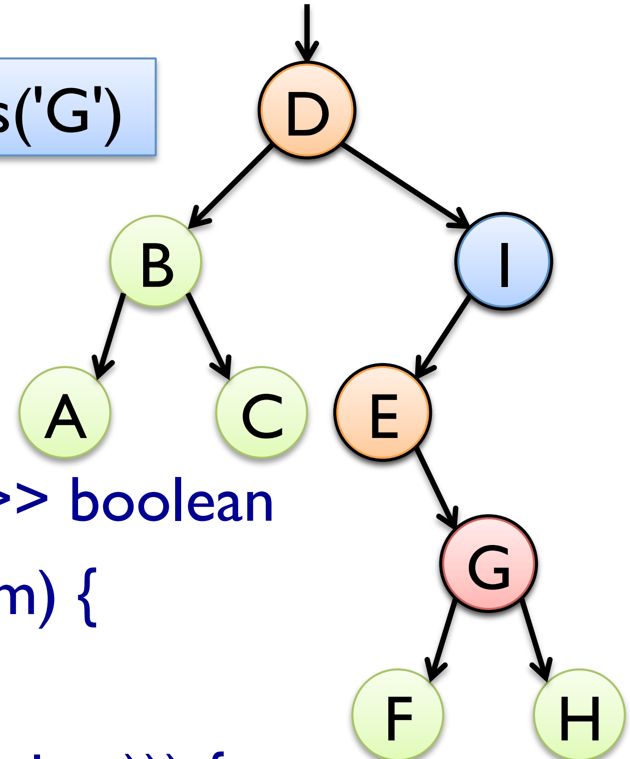
```
public class BinSortTree<E extends Comparable<E>>
    implements SortTree<E> {
    private BinTreeNode<E> root = null;
    public int size() {  return size(this.root);  }
    private static <E extends Comparable<E>> int
        size(BinTreeNode<E> node) {
        if (node == null) {  return 0;  }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

SortTree ADT: Implementation

- Implementation 1 (continued):

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

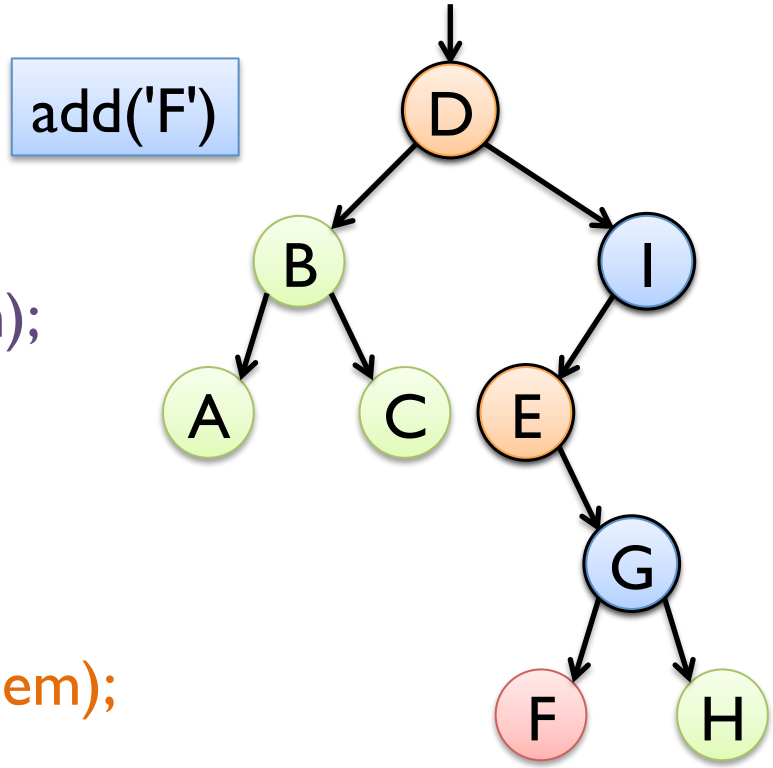
- Implementation 1 (continued):

```
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

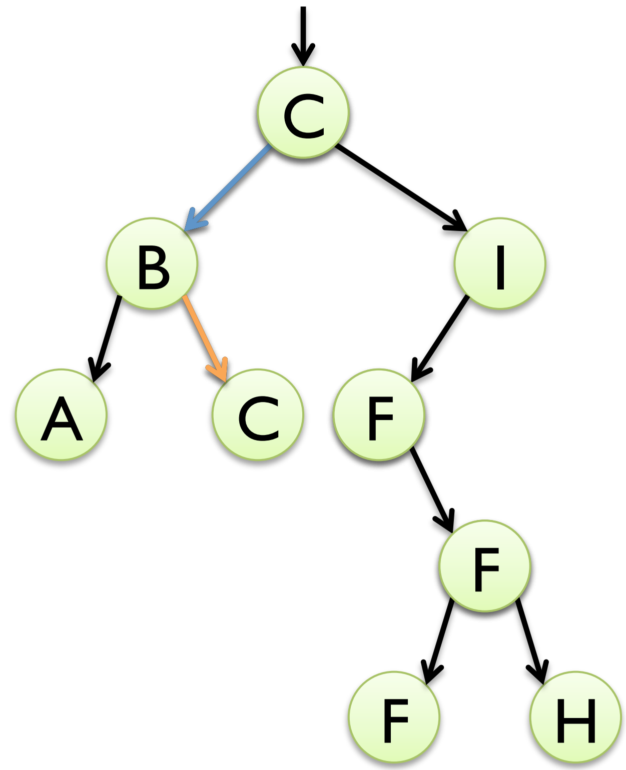
- first, find the node (as contains or add)
- four cases for node to be removed
  1. no children
  2. only a left child
  3. only a right child
  4. both left and right child
- easy to handle 1-3:
  1. delete node
  2. replace node by left child
  3. replace node by right child
  4. replace node by largest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