



# DM537

## Object-Oriented Programming

Peter Schneider-Kamp

[petersk@imada.sdu.dk](mailto:petersk@imada.sdu.dk)

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



# RECURSION (REVISITED)

# Recursion (Revisited)

- recursive function = a function that calls itself
- Example (meaningless):

```
public static void main(String[] args) {  
    if (args.length > 0) { main(new String[args.length-1]); }  
}
```

- base case = no recursive function call reached
- we say the function call *terminates*
- infinite recursion = no base case is reached
- also called *non-termination*
- Java recursion depth only limited by Java stack size

# Comparable Interface

- imposes *total order* on elements, i.e., all elements comparable
- works similar to `__cmp__(self, other)` method in Python
- classes need to implement interface **Comparable**:  
public interface Comparable<T> {  
    public int compareTo(T other);  
}
- has to return 0, if **this** and **other** are equal according to **equals**
- has to return 1, if **this** is greater than **other**
- has to return -1, if **this** is smaller than **other**

# Binary Search

- assume ordered list of elements implementing Comparable
- “Divide et Impera” gives us *binary search* by halving the list
- Example:

```
public int find(Comparable elem) {  
    return this.binarySearch(elem, 0, this.getSize()-1);  
}  
  
public int binarySearch(Comparable elem, int low, int high) {  
    if (low > high) { return -1; } // range is empty; not found  
    int middle = (low + high) / 2;  
    switch (elem.compareTo(this.get(middle))) { ... }  
}
```

# Binary Search

- Example (continued):

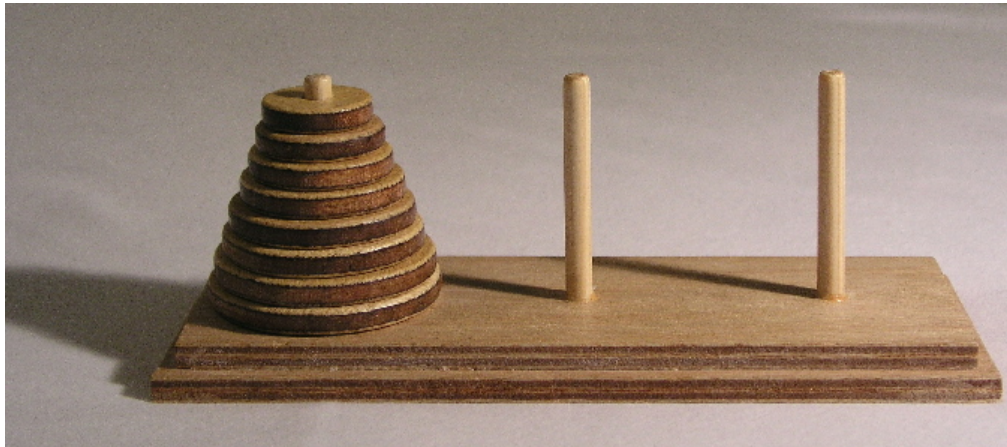
```
public int binarySearch(Comparable elem, int low, int high) {  
    if (low > high) { return -1; } // range is empty; not found  
    int middle = (low + high) / 2;  
    switch (elem.compareTo(this.get(middle))) {  
        case 0: return middle;  
        case 1: return binarySearch(elem, middle+1, high);  
        case -1: return binarySearch(elem, low, middle-1);  
    }  
    throw new RuntimeException("compareTo error");  
}
```



binarySearch(42, 4, 4)

# Towers of Hanoi

- game invented by Edouard Lucas in 1883
- three pins with discs of varying diameter



- **Goal:** move all discs from the left to the right pin
- **Rule 1:** only one disc may be moved at a time
- **Rule 2:** a bigger disc may not lie on top of a smaller one
- **Rule 3:** all discs except the one moved are on some pin



# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



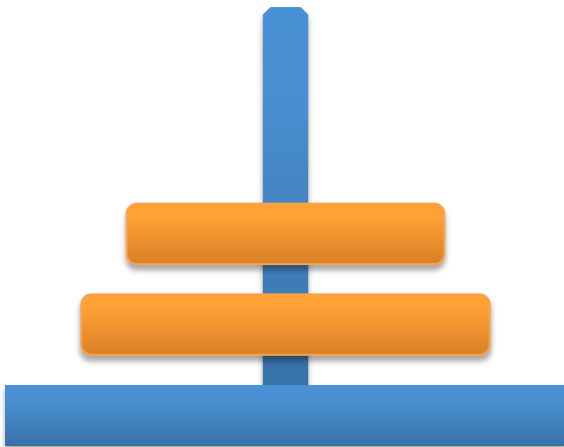
# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



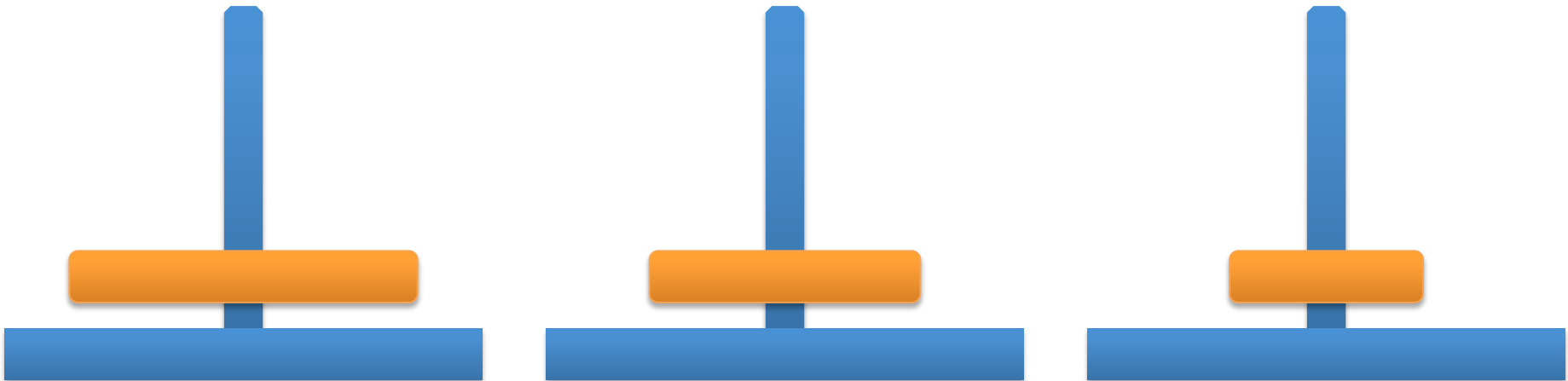
# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



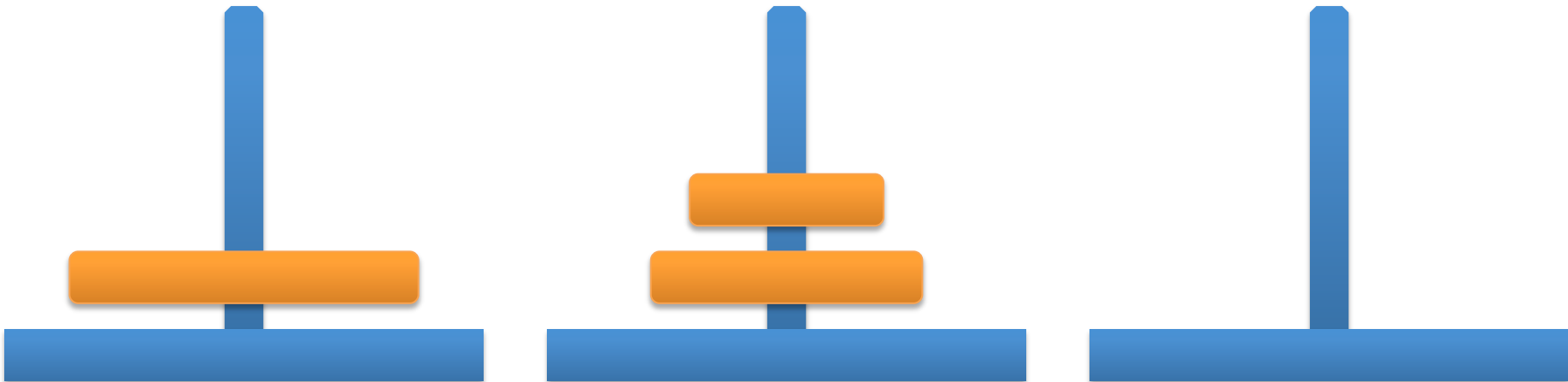
# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



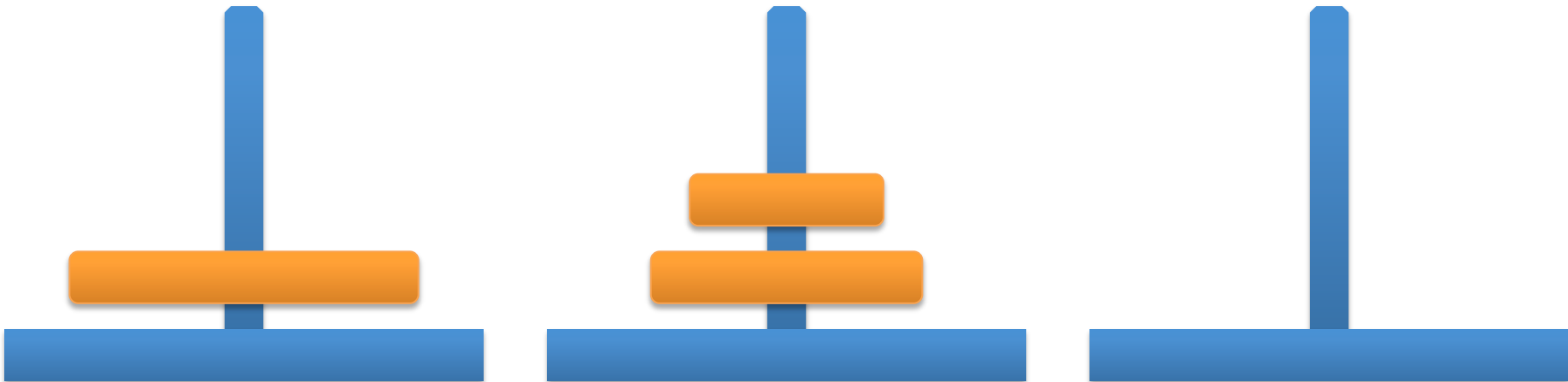
# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



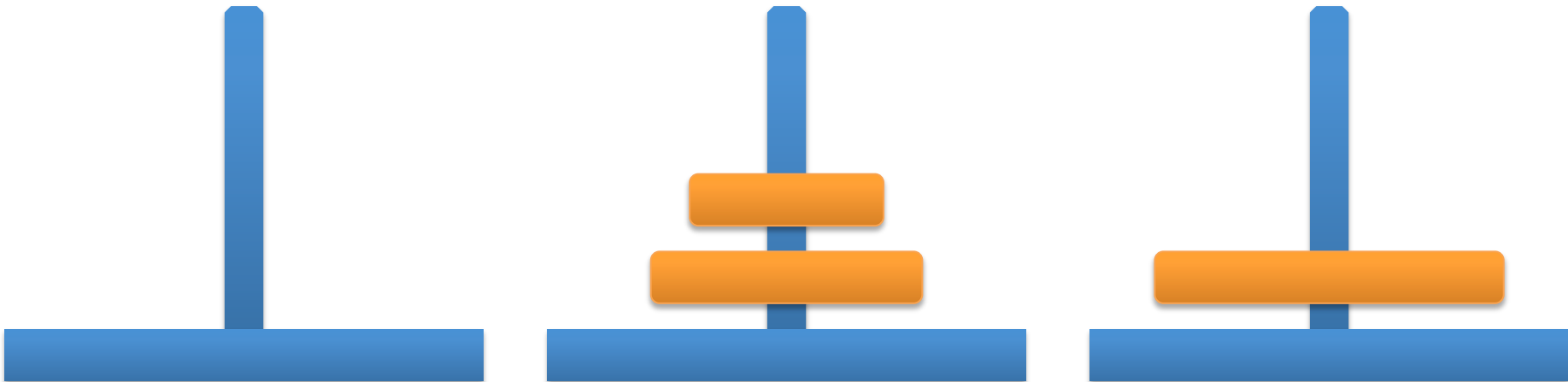
# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - **move the largest disc from left to right pin**
  - move all discs (except the largest) from middle pin to the right pin using the left pin



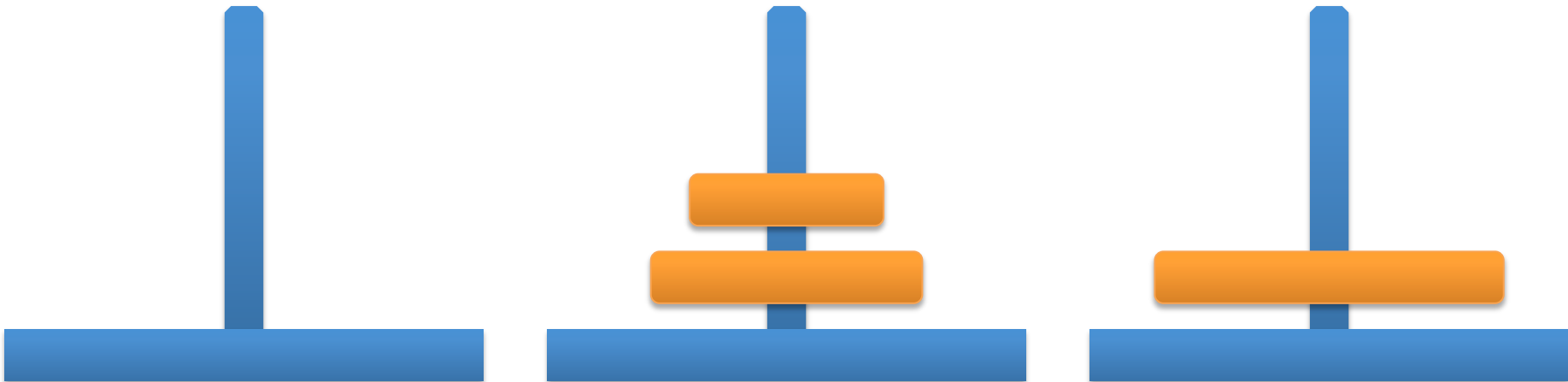
# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - **move the largest disc from left to right pin**
  - move all discs (except the largest) from middle pin to the right pin using the left pin



# Towers of Hanoi

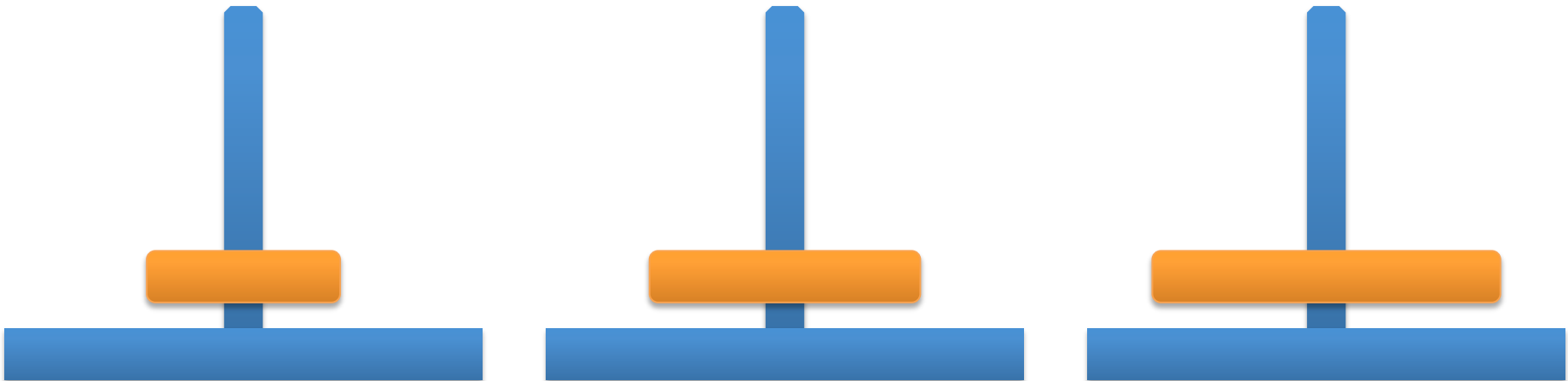
- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin





# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin



# Towers of Hanoi

- **Solution:** move discs from left to right using the middle pin
  - move all discs (except the largest) from left pin to the middle pin using the right pin
  - move the largest disc from left to right pin
  - move all discs (except the largest) from middle pin to the right pin using the left pin
- this solution is inherently recursive
- can be formulated very simple by specifying individual moves
- to move  $n$  discs from pin 1 to pin 3 using pin 2, do:
  - move  $n-1$  discs from pin 1 to pin 2 using pin 3
  - move disc from pin 1 to pin 3
  - move  $n-1$  discs from pin 2 to pin 3 using pin 1

# Towers of Hanoi

```
public class Move { public int from, to; // simply a pair of ints
    Move(int from, int to) { this.to = to; this.from = from; }
    public String toString() { return this.from+" -> "+this.to; }
}

public static List<Move> hanoi(int n, int from, int to, int using) {
    List<Move> moves = new ArrayList<Move>();
    if (n > 1) { moves.addAll(hanoi(n-1, from, using, to)); }
    moves.add(new Move(from, to));
    if (n > 1) { moves.addAll(hanoi(n-1, using, to, from)); }
    return moves;
} ...

System.out.println(hanoi(3, 1, 3, 2));
```

# **RECURSIVE DATA STRUCTURES**

# Recursive Data Structures

- like functions, data structures can be recursive, too
- recursive class = contains a member variable of same class
- Example:

```
public class Student {  
    String name;  
    Student tutor;  
    public String toString() {  
        return name + tutor == null ? "" : " (" + tutor.name + ")";  
    }  
}
```

- useful to implement linked lists, trees, ...

# List ADT: Specification (revisited)

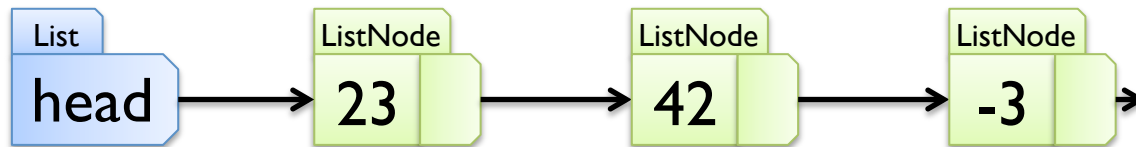
- data are arbitrary objects of class E
- operations are defined by the following interface

```
public interface List<E> {  
    public E get(int i);           // get i-th integer (0-based)  
    public void set(int i, E elem); // set i-th element  
    public int size();           // return length of list  
    public void add(E elem);     // add element at end  
    public void add(int i, E elem); // insert element at pos. i  
    public void remove(int i);   // remove i-th element  
}
```



# Linked Lists

- arrays require copying when inserting in the middle
- avoid copying by using links from one element to the next
- **Idea:**
  - use a recursive data structure `ListNode`
  - one instance of this class per element
  - every node (except the last) refers to next element



# List ADT: Design & Implementation 4

- Design 4: linked lists
- Implementation 4:

```
public class ListNode<E> {  
    private E elem;           // one list element  
    private ListNode<E> next; // next element  
    public ListNode(E elem, ListNode<E> next) {  
        this.elem = elem;  
        this.next = next;  
    }  
    ...  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued)

```
public class ListNode<E> {  
    private E elem;                // one list element  
    private ListNode<E> next;     // next element  
    ...  
    public E get(int i) {  
        if (i == 0) { return this.elem; }  
        if (this.next == null) { throw new Index...Exception(); }  
        return this.next.get(i-1);  
    }  
    ...  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued)

```
public class ListNode<E> {  
    private E elem;                // one list element  
    private ListNode<E> next;     // next element  
    ...  
    public void set(int i, E elem) {  
        if (i == 0) { this.elem = elem; } else {  
            if (this.next == null) { throw new Index...Exception(); }  
            this.next.set(i-1, elem);  
        }  
    }  
    ... }  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued)

```
public class ListNode<E> {  
    private E elem;                // one list element  
    private ListNode<E> next;     // next element  
    ...  
    public int size() {  
        int result = 1;  
        if (this.next != null) { result += this.next.size(); }  
        return result;  
    }  
    ...  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued)

```
public class ListNode<E> {  
    private E elem;                // one list element  
    private ListNode<E> next;     // next element  
    ...  
    public void add(E elem) {  
        if (this.next != null) {  
            this.next.add(elem);  
        } else {  
            this.next = new ListNode<E>(elem, null);  
        }  
    }  
    ... }  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued)

```
public void add(E elem) {  
    if (this.next != null) {  
        this.next.add(elem);  
    } else {  
        this.next = new ListNode<E>(elem, null);  
    }  
}
```

add(17)



# List ADT: Implementation 4

- Implementation 4 (continued)

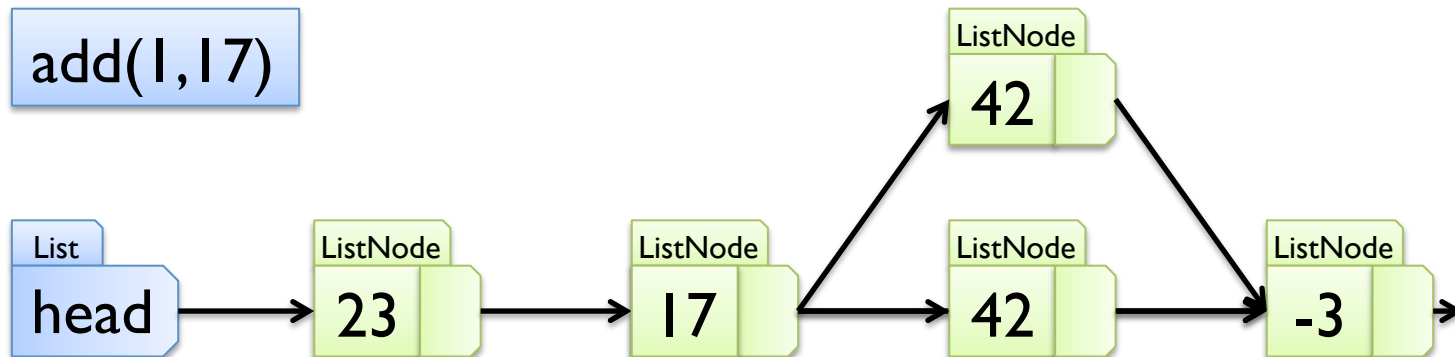
```
public void add(int i, E elem) {  
    if (i == 0) {  
        this.next = new ListNode<E>(this.elem, this.next);  
        this.elem = elem;  
    } else {  
        if (this.next == null) {  
            if (i == 1) { this.add(elem); } // end of list  
            else { throw new Index...Exception(); }  
        } else { this.next.add(i-1, elem); }  
    }  
}
```



# List ADT: Implementation 4

- Implementation 4 (continued)

```
public void add(int i, E elem) {  
    if (i == 0) {  
        this.next = new ListNode<E>(this.elem, this.next);  
        this.elem = elem;  
    } else {  
        ...  
    }  
}
```



# List ADT: Implementation 4

- Implementation 4 (continued)

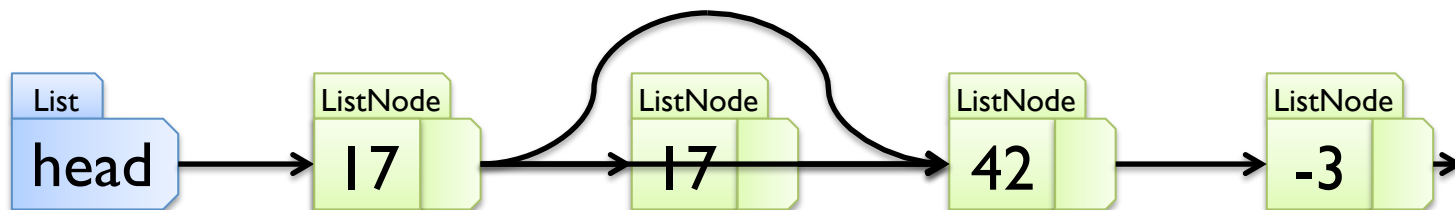
```
public void remove(int i) {  
    if (i == 0) {  
        if (this.next == null) { throw new RuntimeException(); }  
        this.elem = this.next.elem;  
        this.next = this.next.next;  
    } else {  
        if (this.next == null) { throw new Index...Exception(); }  
        this.next.remove(i-1);  
    }  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued)

```
public void remove(int i) {  
    if (i == 0) {  
        if (this.next == null) { throw new RuntimeException(); }  
        this.elem = this.next.elem;  
        this.next = this.next.next;  
    } ...  
}
```

remove(0)



# List ADT: Implementation 4

- Implementation 4 (continued):

```
public class LinkedList<E> implements List<E> {  
    private ListNode<E> head = null;  
    public E get(int i) {  
        if (i < 0) { throw new IllegalArgumentException(); }  
        if (head == null) { throw new Index...Exception(); }  
        return head.get(i);  
    }  
    ...  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued):

```
public class LinkedList<E> implements List<E> {  
    private ListNode<E> head = null; ...  
    public void set(int i, E elem) {  
        if (i < 0) { throw new IllegalArgumentException(); }  
        if (head == null) { throw new Index...Exception(); }  
        head.set(i, elem);  
    }  
    ...  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued):

```
public class LinkedList<E> implements List<E> {  
    private ListNode<E> head = null; ...  
    public int size() {  
        if (head == null) { return 0; }  
        return head.size();  
    }  
    ...  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued):

```
public class LinkedList<E> implements List<E> {  
    private ListNode<E> head = null; ...  
    public void add(E elem) {  
        if (head == null) {  
            head = new ListNode<E>(elem, null);  
        } else {  
            head.add(elem);  
        }  
    }  
    ...  
}
```

# List ADT: Implementation 4

- Implementation 4 (continued):

```
public class LinkedList<E> implements List<E> { ...
    public void add(int i, E elem) {
        if (i < 0) { throw new IllegalArgumentException(); }
        if (head == null) {
            if (i > 0) { throw new Index...Exception(); }
            head = new ListNode<E>(elem, null);
        } else {
            head.add(i, elem);
        }
    } ...
}
```



# List ADT: Implementation 4

- Implementation 4 (continued):

```
public class LinkedList<E> implements List<E> { ...
    public void remove(int i) {
        if (i < 0) { throw new IllegalArgumentException(); }
        if (head == null) { throw new Index...Exception(); }
        else if (head.getNext() == null) {
            if (i > 0) { throw new Index...Exception(); }
            head = null;
        } else {
            head.remove(i);
        }
    }
}
```

# **STATIC FUNCTIONS FOR RECURSIVE DATA STRUCTURES**

# List ADT: Implementation 5

- Implementation 5:

```
public class RecursiveList<E> implements List<E> {
    private ListNode<E> head = null;
    public E get(int i) {
        if (i < 0) { throw new IllegalArgumentException(); }
        return get(this.head, i);
    }
    public static <E> E get(ListNode<E> node, int i) {
        if (node == null) { throw new Index...Exception(); }
        if (i == 0) { return node.getElem(); }
        return get(node.getNext(), i-1);
    }
    ...
}
```

# List ADT: Implementation 5

- Implementation 5 (continued):

```
public class RecursiveList<E> implements List<E> { ...
    public void set(int i, E elem) {
        if (i < 0) { throw new IllegalArgumentException(); }
        set(this.head, i, elem);
    }
    public static <E> void set(ListNode<E> node, int i, E elem) {
        if (node == null) { throw new Index...Exception(); }
        if (i == 0) { node.setElem(elem); }
        else { set(node.getNext(), i-1, elem); }
    }
    ...
}
```

# List ADT: Implementation 5

- Implementation 5 (continued):

```
public class RecursiveList<E> implements List<E> {  
    ...  
    public int size() {  
        return size(this.head);  
    }  
    public static <E> int size(ListNode<E> node) {  
        if (node == null) { return 0; }  
        return 1+size(node.getNext());  
    }  
    ...  
}
```

# List ADT: Implementation 5

- Implementation 5 (continued):

```
public class RecursiveList<E> implements List<E> {  
    ...  
    public void add(E elem) {  
        this.head = add(this.head, elem);  
    }  
    public static <E> ListNode<E> add(ListNode<E> n, E e) {  
        if (n == null) { return new ListNode<E>(e, null); }  
        n.setNext(add(n.getNext(), e));  
        return n;  
    }  
    ...  
}
```

# List ADT: Implementation 5

- Implementation 5 (continued):

```
public class RecursiveList<E> implements List<E> { ...
    public void add(int i, E elem) {
        if (i < 0) { throw new IllegalArgumentException(); }
        this.head = add(this.head, i, elem);
    }
    public static <E> ListNode<E> add(ListNode<E> n, int i, E e) {
        if (i == 0) { return new ListNode<E>(e, n); }
        if (n == null) { throw new Index...Exception(); }
        n.setNext(add(n.getNext(), i-1, e));
        return n;
    } ...
}
```

# List ADT: Implementation 5

- Implementation 5 (continued):

```
public class RecursiveList<E> implements List<E> { ...
    public void remove(int i) {
        if (i < 0) { throw new IllegalArgumentException(); }
        this.head = remove(this.head, i);
    }
    public static <E> ListNode<E> remove(ListNode<E> n, int i) {
        if (n == null) { throw new Index...Exception(); }
        if (i == 0) { return n.getNext(); }
        n.setNext(remove(n.getNext(), i-1));
        return n;
    } } // DONE
```



# 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



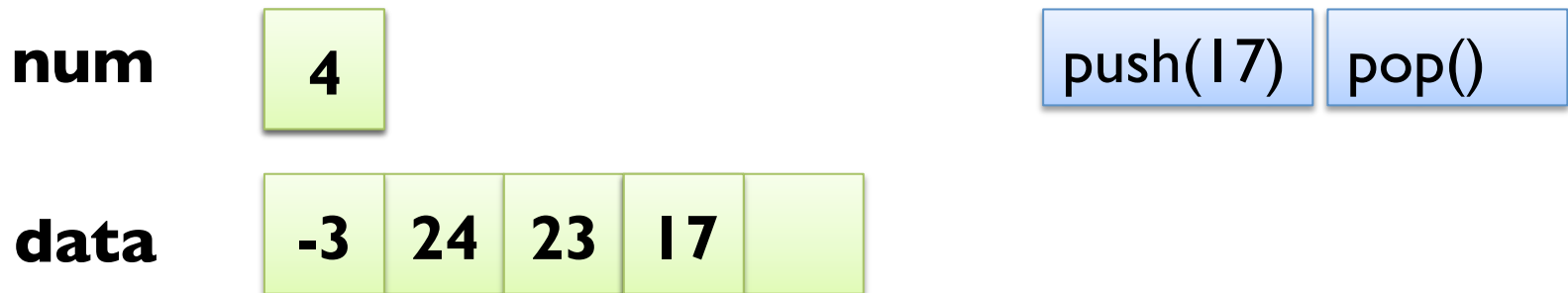
# Stack ADT: Specification

- 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 I

- Design I: 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
  - popping corresponds to removing at the end



# Stack ADT: Implementation I

- Implementation I:

```
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 I

- Implementation I (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 I

- Implementation I (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

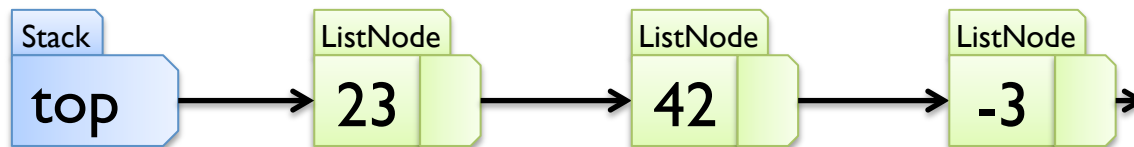
- Design 2: reuse dynamic array list (`ArrayList<E>`)
- Implementation 2:

```
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();
```

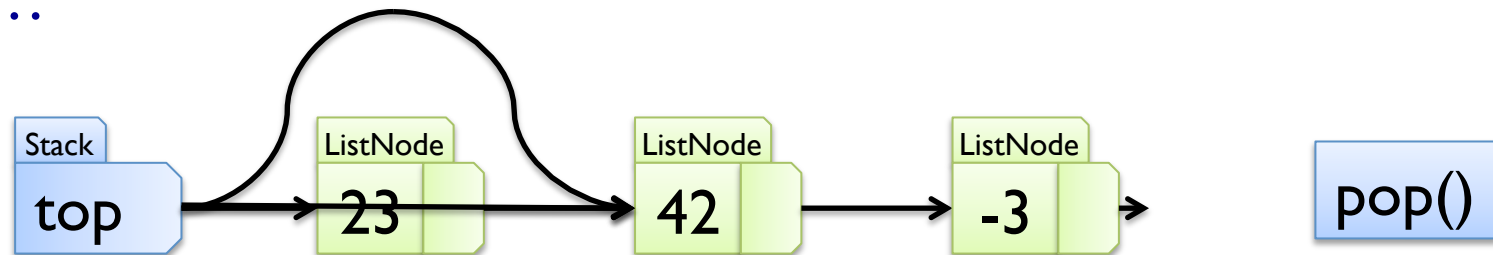
```
    this.top = this.top.getNext();
```

```
    return result;
```

```
}
```

```
...
```

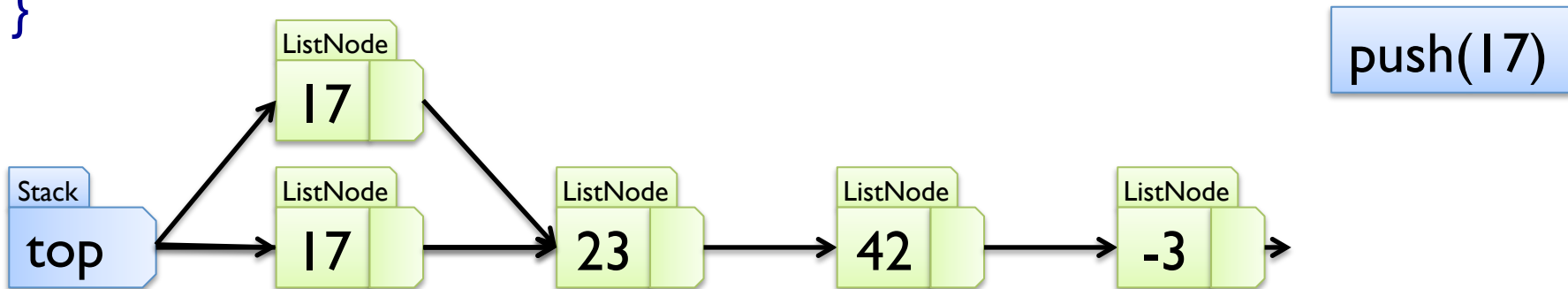
```
}
```



# Stack ADT: Implementation 3

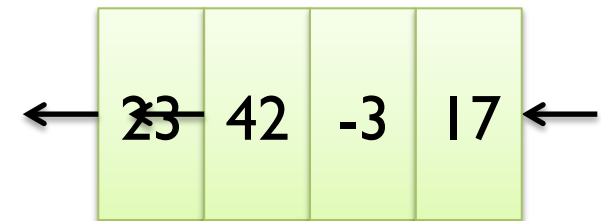
- Implementation 3 (continued):

```
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

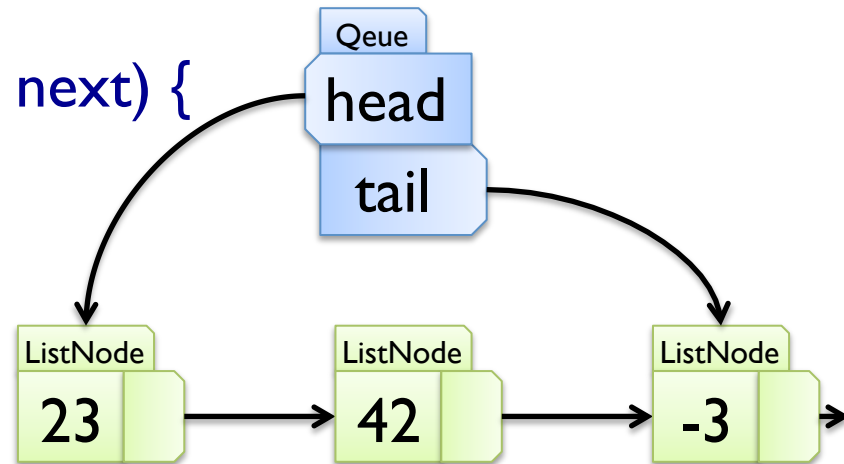
- Design I: reuse dynamic array list (`ArrayList<E>`)
- Implementation I:

```
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

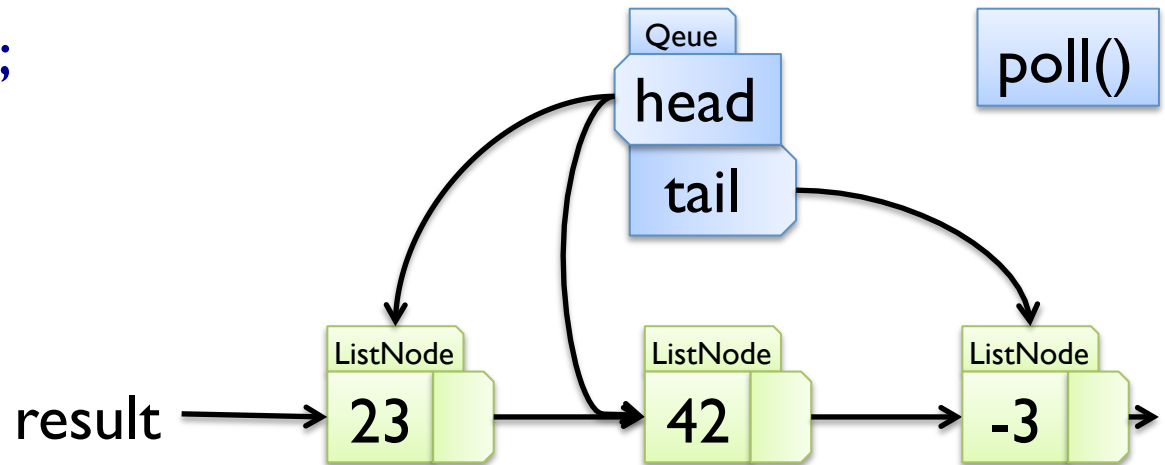
- Implementation 2 (continued):

```
public class LinkedListQueue<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):

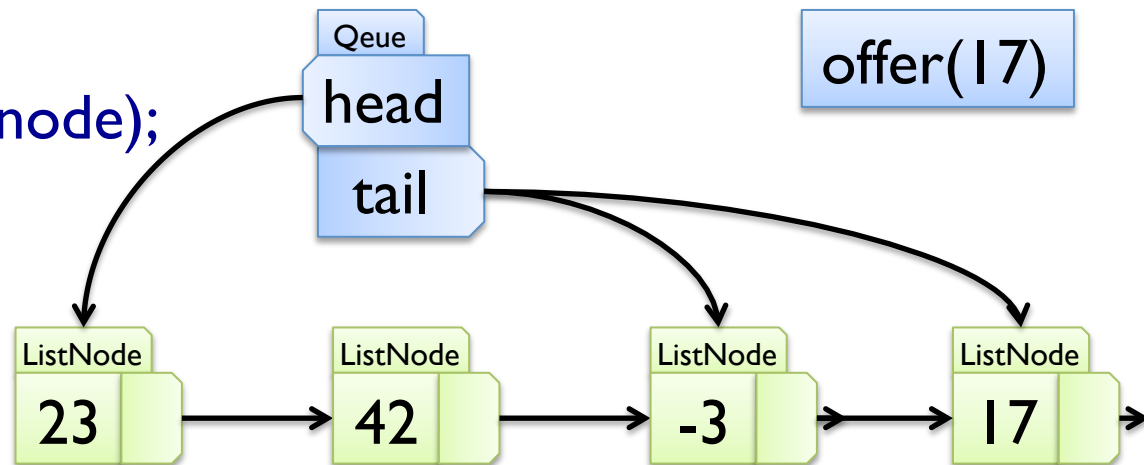
```
public class LinkedQueue<E> implements Queue<E> { ...  
    public E poll() {  
        E result = this.peek();  
        this.head = this.head.getNext();  
        if (this.head == null) {  
            this.tail = null;  
        }  
        return result;  
    }  
    ...  
}
```



# Queue ADT: Implementation 2

- Implementation 2 (continued):

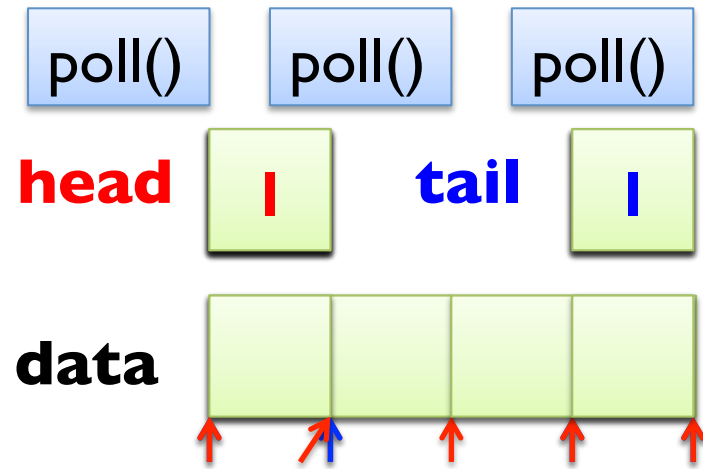
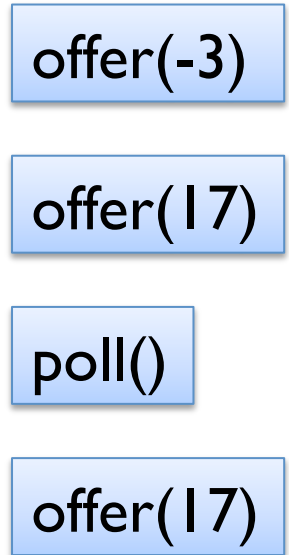
```
public class LinkedQueue<E> implements Queue<E> { ...  
    public boolean offer(E elem) {  
        ListNode<E> node = new ListNode<E>(elem, null);  
        if (this.head == null) {  
            this.head = this.tail = node;  
        } else {  
            this.tail.setNext(node);  
        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



# 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;
    } ...
}
```