# Chapter 7

## Objectives

- Introduce stacks and queues as concepts and implementations
- Stacks and queues in the JCF
- Introduce Markup Language (XML,...)
- Introduce the heap and priority queue

## The Stack

- Push-down LIFO structure
- Add data *only to* and take data *only from* the `top` of the stack
- Cafeteria tray dispensers, Pez dispensers
- `push` and `pop` methods required
- `peek` and `isEmpty` often useful

# UML for a Stack

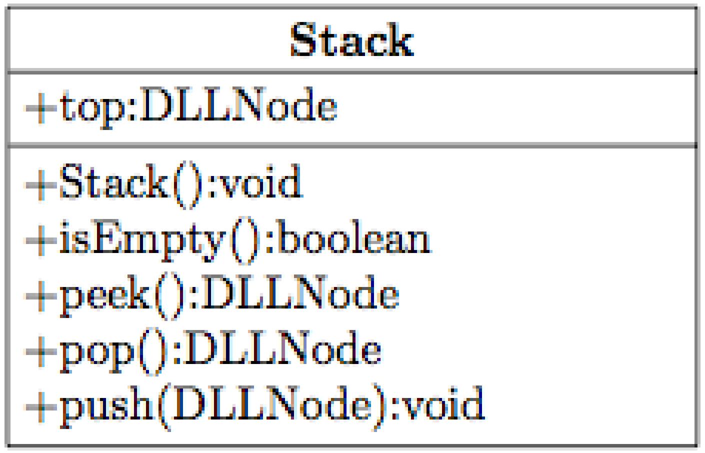
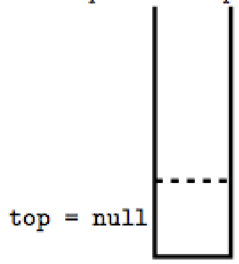
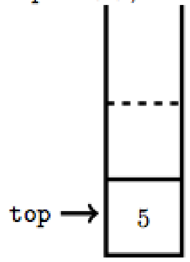


**Figure 7.1** A UML diagram for a Stack class

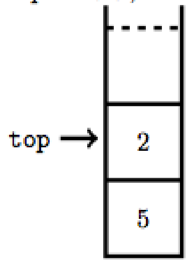
# A Stack in Action



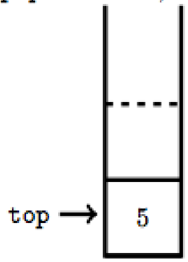
Step 1: An empty Stack



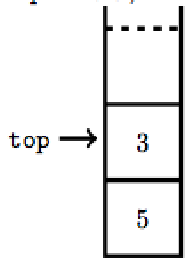
Step 2: push(5), and then we have



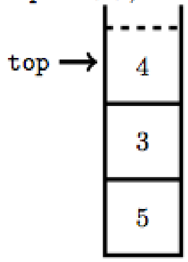
Step 3: push(2), and then we have



Step 4: pop returns 2, and then we have



Step 5: push(3), and then we have



Step 6: push(4), and then we have

# Stack Implementations

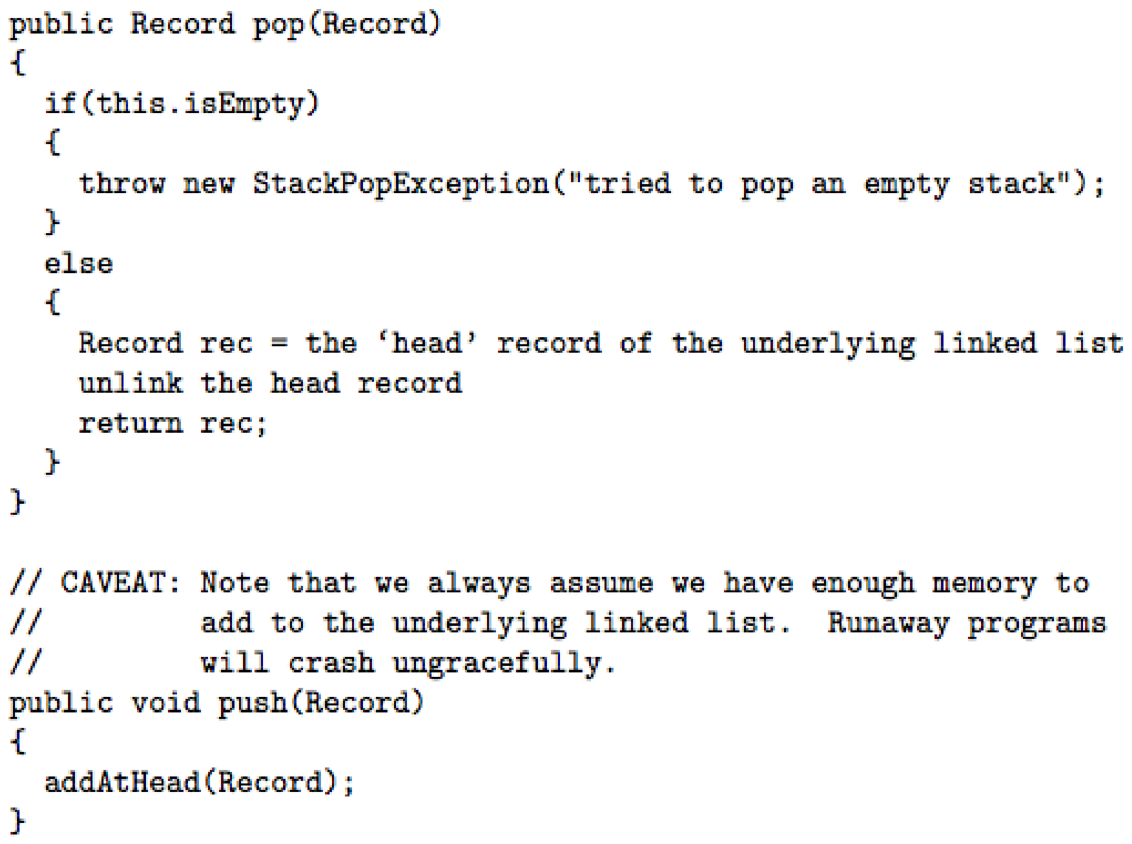
- Often done with linked lists
- `push` and `pop` just wrap LL methods
- Can be done with `ArrayList`
- `Stack` class exists in the JCF and is a standard stack (unlike, e.g., `Queue` which is not so standard)

## Stack Code Stubs

```
public boolean isEmpty()
{
  return ‘isEmpty’ of the underlying linked list
}

public Record peek()
{
  if(this.isEmpty)
  {
    return null;
  }
  else
  {
    Record rec = the ‘head’ record of the underlying linked list
    return rec;
  }
}
```

# Stack Code Stubs (2)



```
public Record pop(Record)
{
  if(this.isEmpty)
  {
    throw new StackPopException("tried to pop an empty stack");
  }
  else
  {
    Record rec = the 'head' record of the underlying linked list
    unlink the head record
    return rec;
  }
}

// CAVEAT: Note that we always assume we have enough memory to
//         add to the underlying linked list.  Runaway programs
//         will crash ungracefully.
public void push(Record)
{
  addAtHead(Record);
}
```

**Figure 7.2** Code stubs for a Stack class

## Throwing Exceptions

`isEmpty` “can't” really be in error

`push` … only by running out of memory?

`peek` and `pop` just as in LL

- Return a null, user must check?
- Throw exception and die?
- Either could be appropriate

# A Stack Using an Array

Step 1: Empty stack

top = *

| subscript | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| data | * | * | * | * | * | * | * | * | * | * |

Step 2: `push(5)`, and then we have

top = 0

| subscript | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| data | 5 | * | * | * | * | * | * | * | * | * |

Step 3: `push(2)`, and then we have

top = 1

| subscript | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| data | 5 | 2 | * | * | * | * | * | * | * | * |

Step 4: `pop` returns 2, and then we have

top = 0

| subscript | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| data | 5 | 2 | * | * | * | * | * | * | * | * |

Step 5: `push(3)`, and then we have

top = 1

| subscript | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| data | 5 | 3 | * | * | * | * | * | * | * | * |

Step 6: `push(4)`, and then we have

top = 2

| subscript | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| data | 5 | 3 | 4 | * | * | * | * | * | * | * |

**Figure 7.4** A stack implemented with an array

## Stack Processing – RPN

- RPN = Reverse Polish Notation, after Jan Łukasiewicz
- Used in HP calculators
- Permits writing arithmetic expressions without parentheses
- Used in several early computers because it is simple to build

# RPN (2)

$4 +_a (2 *_b 3) -_c ((5 +_d 7) *_e 8)$

Rewrite as $4\ 2\ 3 *_b +_a 5\ 7 +_d 8 *_e -_c$

Read L to R, push numbers onto the stack

When operator is read, pop two numbers, perform the op, and push the result back onto the stack

Requires an *accumulator* for the first of the two numbers popped, and op into accum

# RPN (3)

- [4 null] $4\ 2\ 3\ *_b\ +_a\ 5\ 7\ +_d\ 8\ *_e\ -_c$
- [2 4 null] $2\ 3\ *_b\ +_a\ 5\ 7\ +_d\ 8\ *_e\ -_c$
- [3 2 4 null] $3\ *_b\ +_a\ 5\ 7\ +_d\ 8\ *_e\ -_c$
- $*_b \rightarrow$ [6 4 null] $*_b\ +_a\ 5\ 7\ +_d\ 8\ *_e\ -_c$
- $+_a \rightarrow$ [10 null] $+_a\ 5\ 7\ +_d\ 8\ *_e\ -_c$

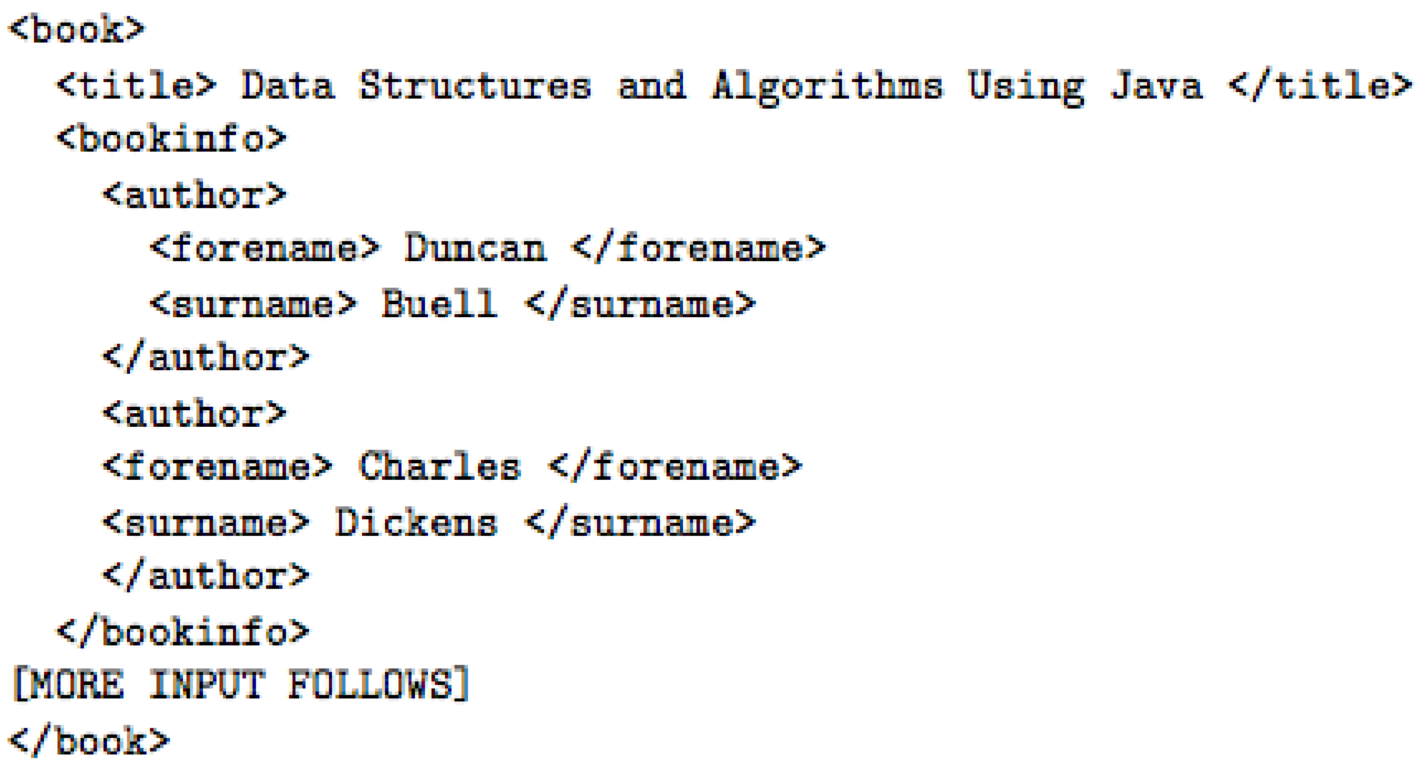
# RPN (3)

- [5 10 null]  5 7 $+_d$ 8 $*_e$ $-_c$
- [7 5 10 null] 7 $+_d$ 8 $*_e$ $-_c$
- $+_d \rightarrow$ [12 10 null] $+_d$ 8 $*_e$ $-_c$
- [8 12 10 null] 8 $*_e$ $-_c$
- $*_e \rightarrow$ [96 10 null] $*_e$ $-_c$
- $-_c \rightarrow$ [86 null] $-_c$

## Why Stacks?

- Stack processing allows us to parse left to right in one pass on “ordinary” expression and to handle the nesting of parentheses
- Or of HTML/XML/whateverML tags

# Parsing Well-Formed XML



```
<book>
  <title> Data Structures and Algorithms Using Java </title>
  <bookinfo>
    <author>
      <forename> Duncan </forename>
      <surname> Buell </surname>
    </author>
    <author>
    <forename> Charles </forename>
    <surname> Dickens </surname>
    </author>
  </bookinfo>
[MORE INPUT FOLLOWS]
</book>
```

**Figure 7.5** A sample XML-like document

# Good and Bad XML

## Good

```
<phonebook>
  <name>
    John Smith
  </name>
  <number>
    555.1212
  </number>
  <office>
    3A73
  </office>
</phonebook>
```

## Bad

```
<phonebook>
  <name>
    John Smith
    <number>
      555.1212
  </name>
    </number>
  <office>
    3A73
  </office>
</phonebook>
```

## Picture of XML Processing

## The Queue

FIFO processing

- Add data only at the tail
- Take data only from the head

# The Queue (2)

`enqueue` (wraps `addBeforeTail`?)

`dequeue` (wraps `removeAtHead`?)

`peek`

`isEmpty`

# Queue Implementations

- Often done with linked lists
- Queue methods wrap LL methods
- Can be done with `ArrayList`
- `Queue` class exists in the JCF but is much more general than this traditional definition of a queue

## Queueing

- Spelled in the old-fashioned way, instead of the newer way (“queuing”), is the only basic English word with five consecutive vowels.

# UML for the Queue

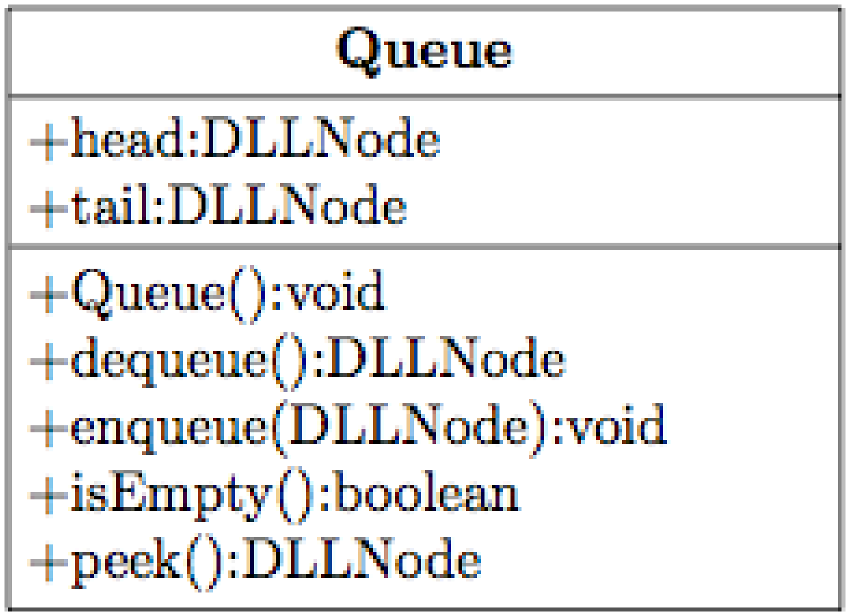
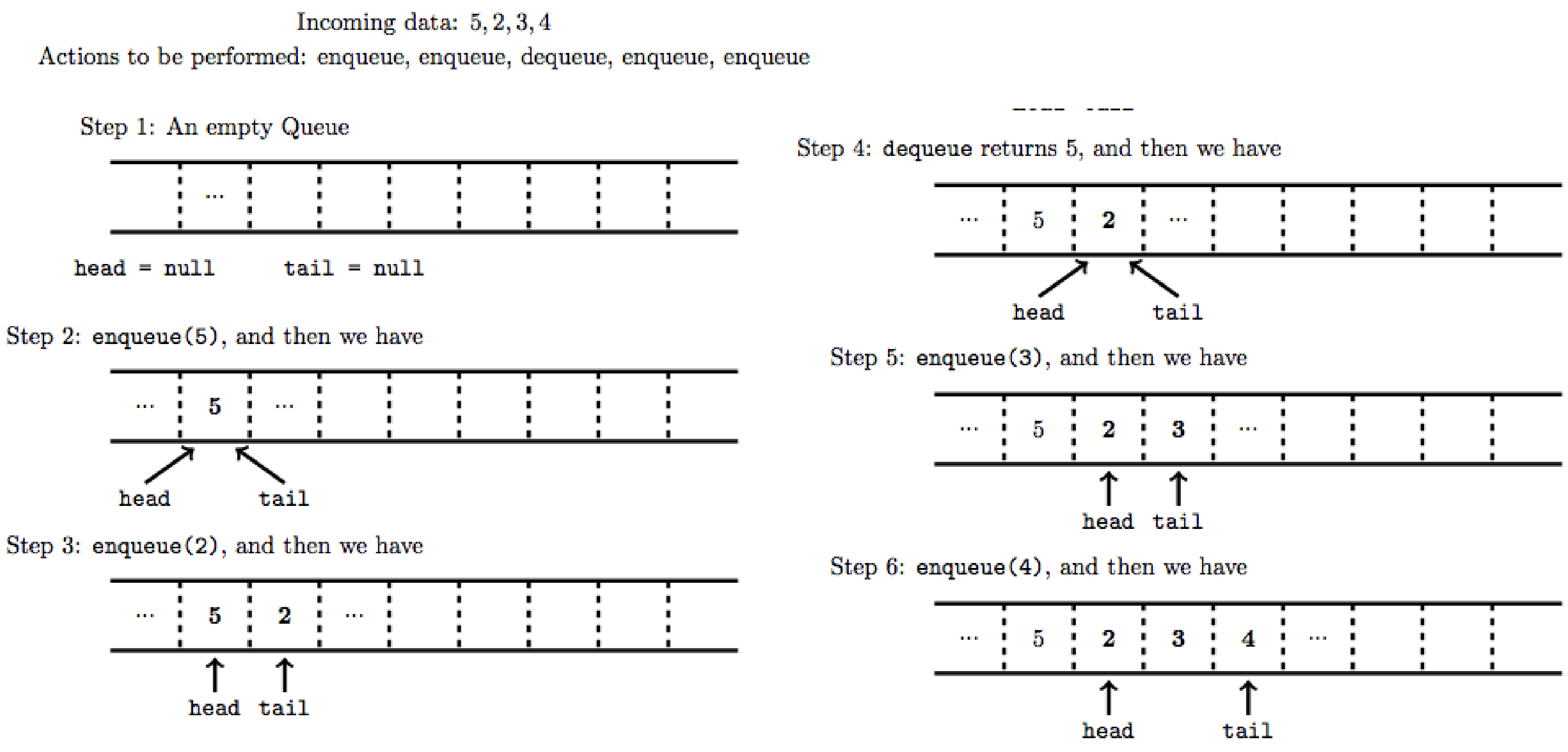


**Figure 7.8** The UML diagram for a `Queue` class

## Queues With Arrays

- A queue is a conceptually easy concept
- But it means we need an infinite array

# Queues With Arrays (2)


Incoming data: 5, 2, 3, 4

Actions to be performed: enqueue, enqueue, dequeue, enqueue, enqueue

Step 1: An empty Queue

`head = null` `tail = null`

Step 2: `enqueue(5)`, and then we have

Step 3: `enqueue(2)`, and then we have

Step 4: `dequeue` returns 5, and then we have

Step 5: `enqueue(3)`, and then we have

Step 6: `enqueue(4)`, and then we have

**Figure 7.10** A queue viewed as an array of infinite length

# Queues With Arrays (3)

- Infinite arrays are impossible to implement
- So we use a *circular* array
- Increment `top` modulo the length
- `top = (top+1)% array.length;`
- But the array must be pre-allocated

# Queues With Arrays (4)

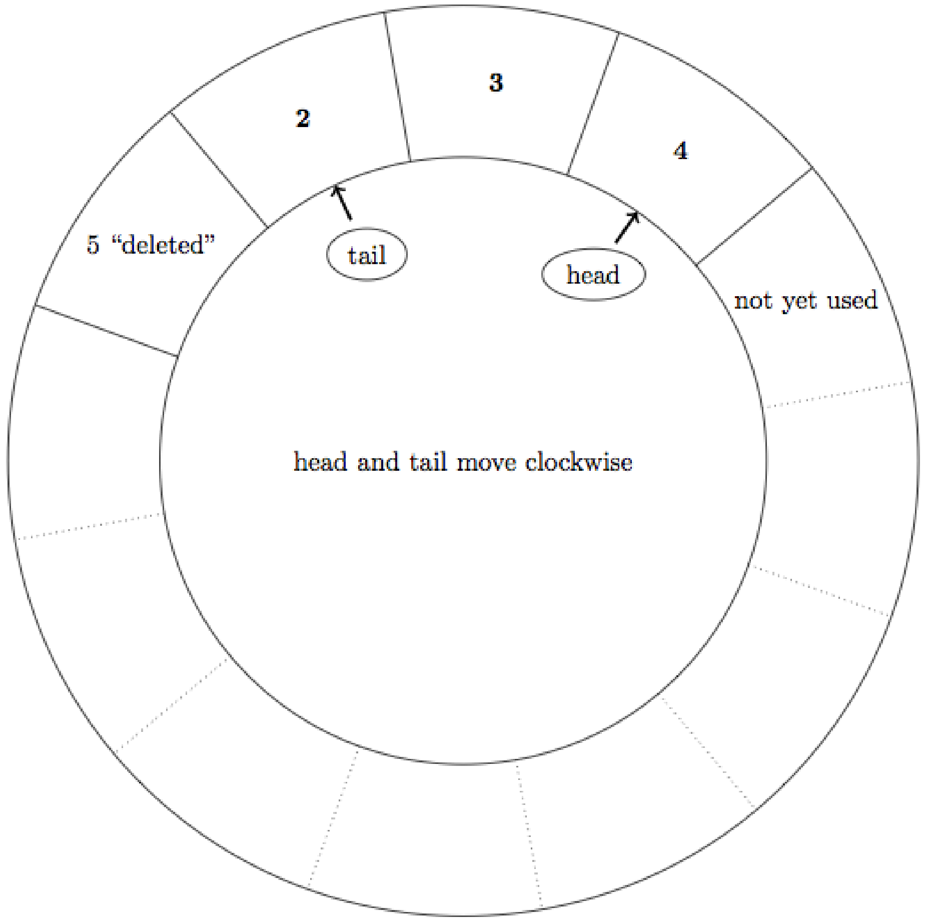


**Figure 7.11** A queue implemented as a circular array

# Java's Built-In JCF Classes

```
import java.util.Stack

import java.util.Queue
```

Stack

Queue

# Java's Built-In JCF Classes (2)

Note that `Stack` is a class, but `Queue` is an `Interface`

`Stack` is older

`Queue` is almost dangerous in its generality—most people when they think “queue” will not think of all the available options that exist in `Queue`

## Heaps and Priority Queues

- A *priority queue* (which is usually not a “queue”) is a structure of items such that the “top” element is the element of highest “priority” in the structure.
- A PQ need not be totally ordered (i.e., sorted). It is necessary only that the “top” item is the “next” item.
- Commonly used for dynamic data.

## Priority Queues

- We could implement a priority queue with a queue and an insertion sort.
- But that would be expensive—we would be storing more “structure content” than is necessary for the job at hand.
- We need a structure that does only what is necessary, and thus doesn't charge us for what we aren't going to use.

## Priority Queues (2)

- Assume we have data in a priority queue.
- We need to be able to
  - Re-build the PQ quickly after we remove the top element
  - Re-build the PQ quickly after we insert a new item

# A Bit of Magic—The Heap

An array `a[*]` has the *max-heap property* if, for all subscripts n, 2n, 2n+1 that are legal, we have

`a[n] >= a[2n]` and `a[n] >= a[2n+1]`.

A array is structured as a *max-heap* if it has the max-heap property.

(min-heap is defined analogously)

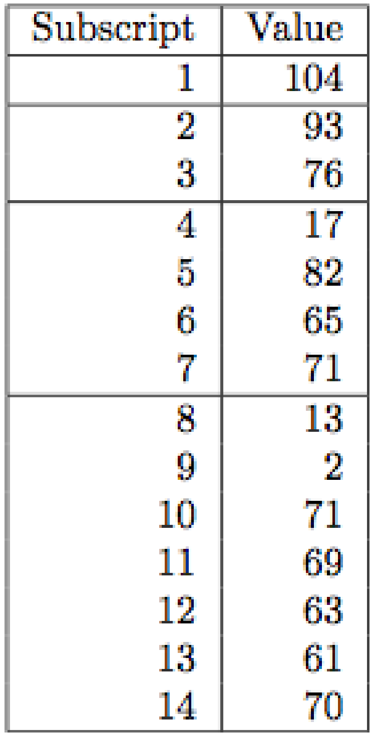
# A Heap

Here's an example

But it's not easy to "see"

that the heap property

is satisfied.



| Subscript | Value |
|---|---|
| 1 | 104 |
| 2 | 93 |
| 3 | 76 |
| 4 | 17 |
| 5 | 82 |
| 6 | 65 |
| 7 | 71 |
| 8 | 13 |
| 9 | 2 |
| 10 | 71 |
| 11 | 69 |
| 12 | 63 |
| 13 | 61 |
| 14 | 70 |

**Figure 7.12** An example of an array that is a max-heap

# A Picture is Worth a Thousand Subscripts

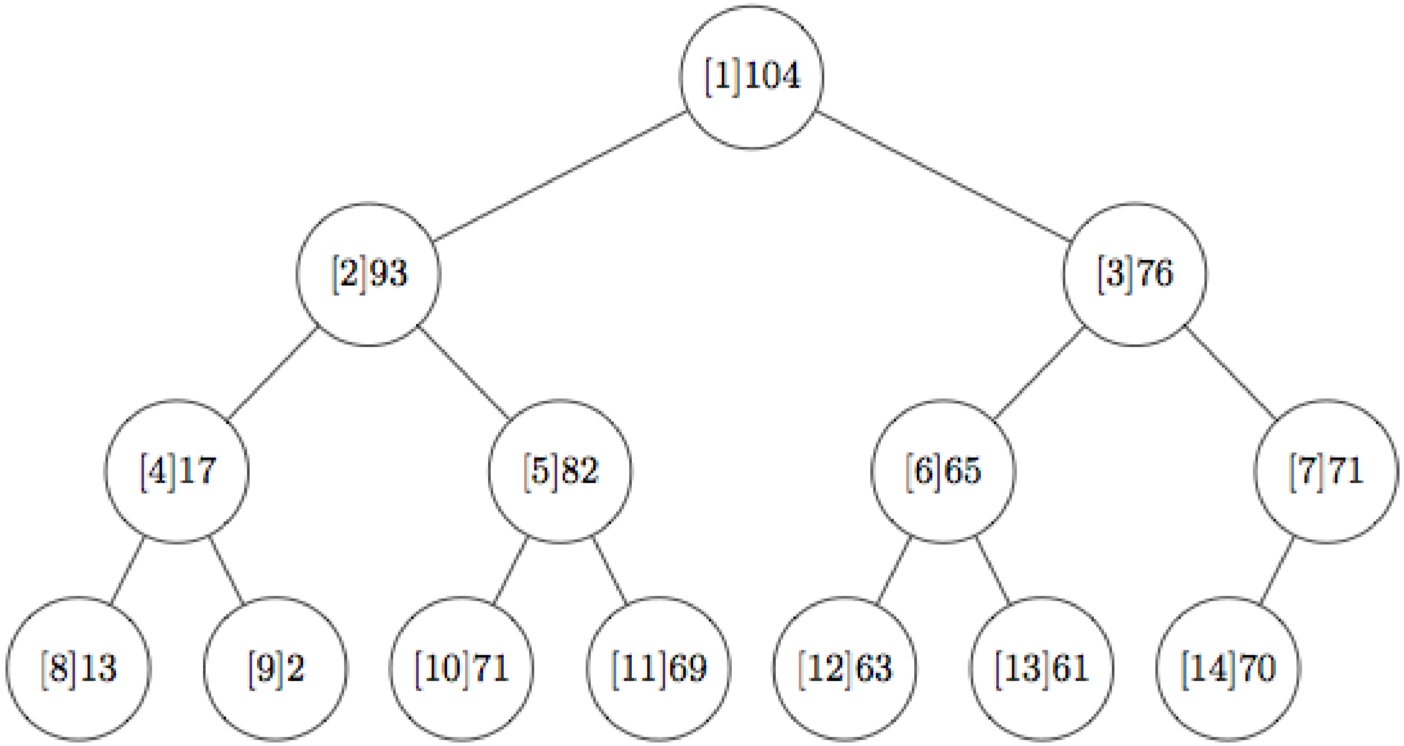


**Figure 7.13** A tree structure for a heap, with [*node*]*value* entries

# The Magic

Insertionsort on a PQ in steady state would take *O(N)* comparisons for each new entry.

Thm. 7.1: We can build a heap of *N* items in *O(N lg N)* comparisons.

Thm. 7.2: We can remove the top element and rebuild the heap in *O(lg N)* comparisons in steady state.

## Theorem 7.1—Building the Heap

We run the following loop (next slide).

*N* array entries, *N* loops, so total of

*N* * (*cost of* `fixHeapUp`) comparisons

We will prove `fixHeapUp` takes *lg N* comparisons, and that would prove the theorem.

# fixHeapUp

```
public void fixHeapUp(int insertSub)
{
  int parentSub;
  ProcessRecord insertRec;

  insertRec = this.theData.get(insertSub);
  parentSub = insertSub/2;
  while(1 < insertSub)
  {
    if(this.theData.get(parentSub).compareTo(insertRec) < 0)
    {
      this.swap(insertSub,parentSub);

      insertSub = parentSub;
      insertRec = this.theData.get(insertSub);
      parentSub = insertSub/2;
    }
    else
      break;
  }
} // public void fixHeapUp(int insertSub)
```

**Figure 7.16** The `fixHeapUp` code

# `fixHeapUp` (2)

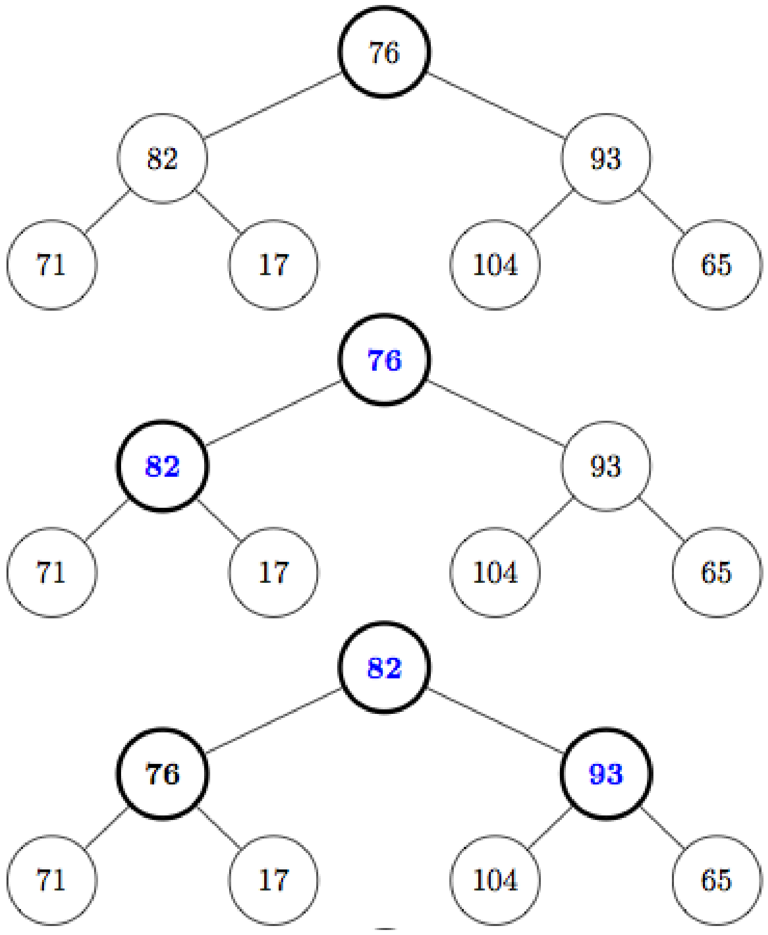
We subscript top to bottom, left to right.

We take the “last” element and bubble it into its proper place by comparing with its parent and swapping if necessary.

The outer loop creates a heap of 2, 3,... items, until done.

Only one test is needed—if we had a heap to start with we don't need to test both the parent and the possible other sibling.

# `fixHeapUp` (3)



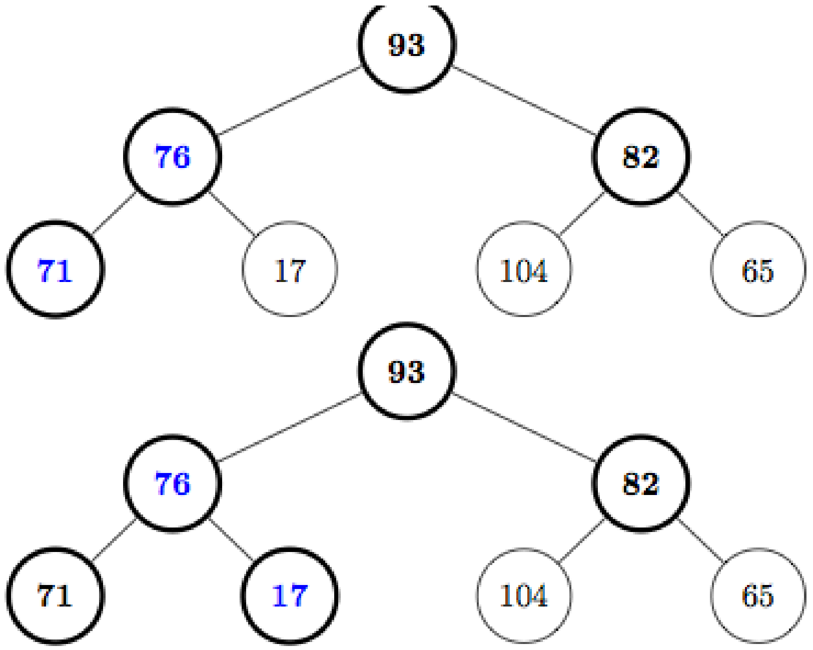
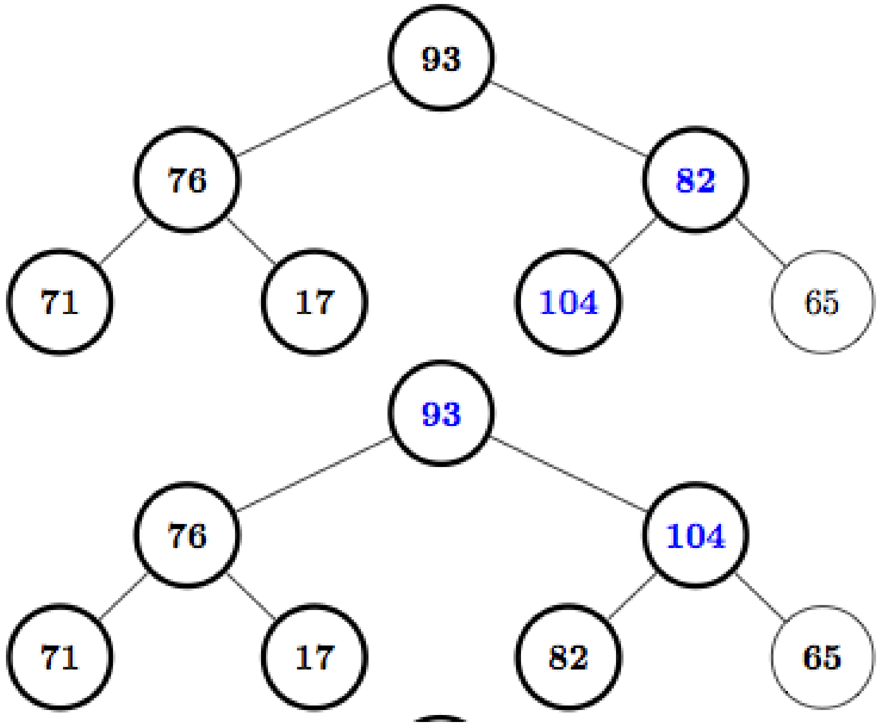


**Figure 7.14** Building a heap from an array, part 1

# `fixHeapUp` (4)



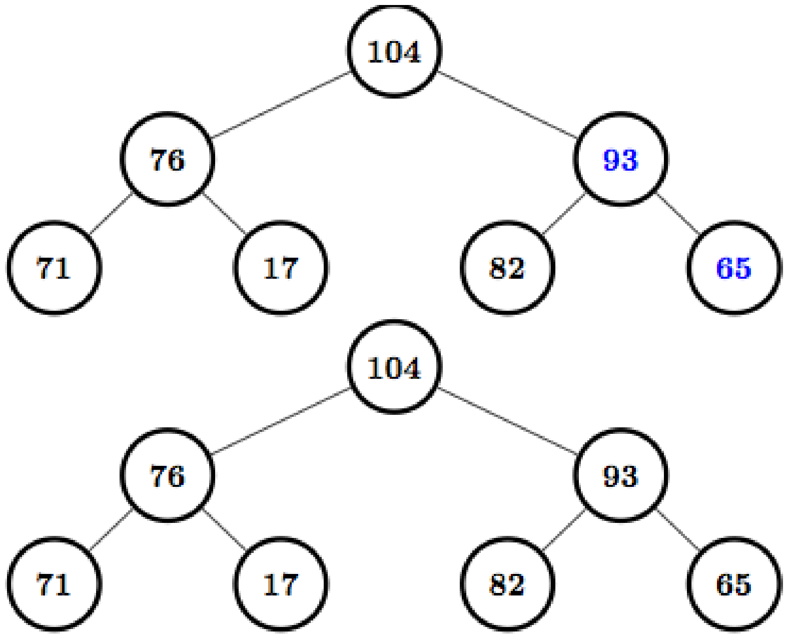


**Figure 7.15** Building a heap from an array, part 2

## Theorem 7.2—Maintaining the Heap

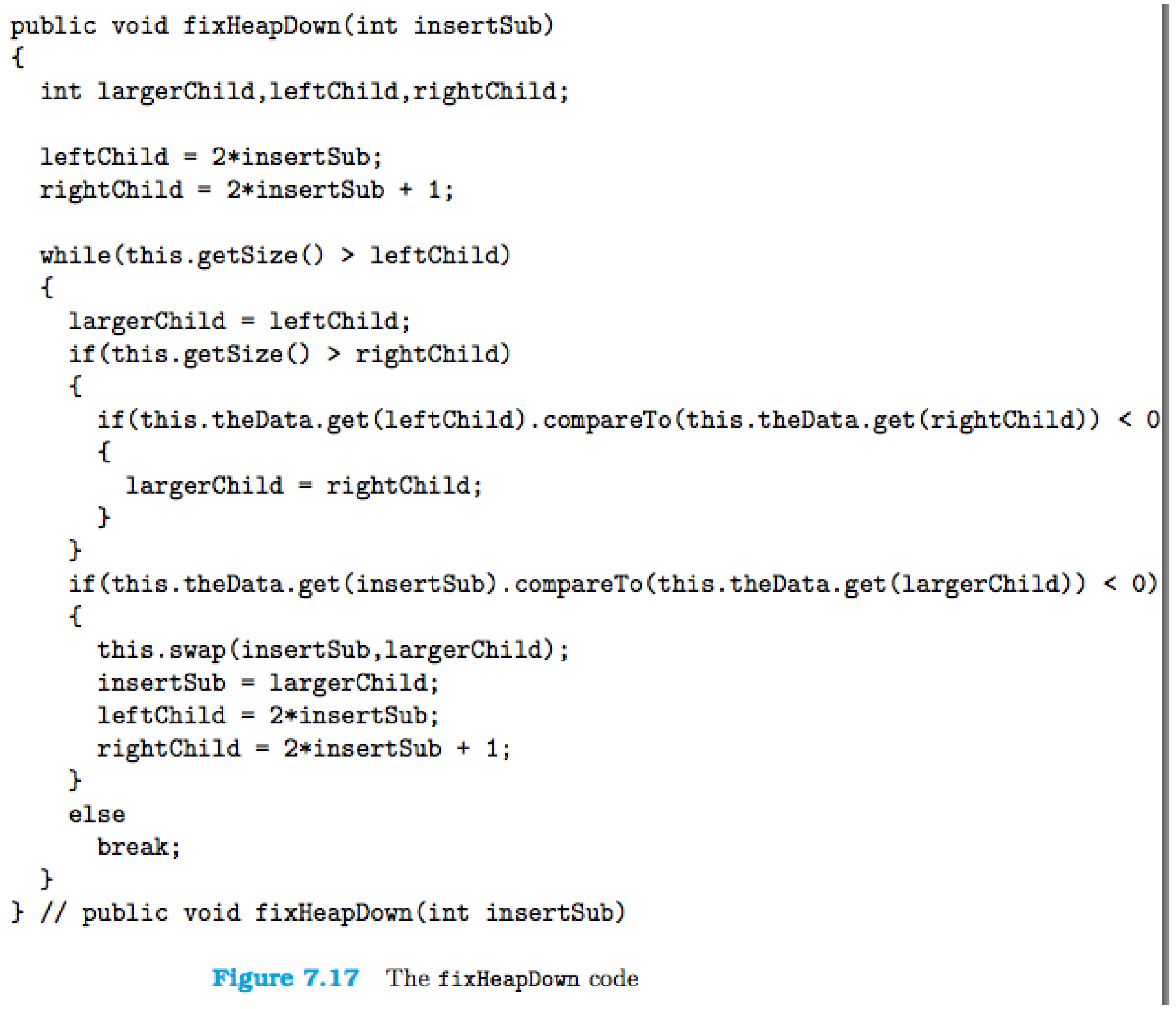
Take off the top entry. This leaves a hole.

Move the last entry to the hole at the top.

Run `fixHeapDown`, which we claim takes *O(lg N)* comparisons

Notice that this time we must compare against both left child and right child to get the max of the triplet into the parent location.

# fixHeapDown



```java
public void fixHeapDown(int insertSub)
{
  int largerChild,leftChild,rightChild;

  leftChild = 2*insertSub;
  rightChild = 2*insertSub + 1;

  while(this.getSize() > leftChild)
  {
    largerChild = leftChild;
    if(this.getSize() > rightChild)
    {
      if(this.theData.get(leftChild).compareTo(this.theData.get(rightChild)) < 0
      {
        largerChild = rightChild;
      }
    }
    if(this.theData.get(insertSub).compareTo(this.theData.get(largerChild)) < 0)
    {
      this.swap(insertSub,largerChild);
      insertSub = largerChild;
      leftChild = 2*insertSub;
      rightChild = 2*insertSub + 1;
    }
    else
      break;
  }
} // public void fixHeapDown(int insertSub)
```

**Figure 7.17** The `fixHeapDown` code

## The Magic

The magic is that all items satisfy a *local* condition (*N* versus *2N* and *2N+1*), but we don't insist on a *global* condition (the total order produced by a sort).

We need less than a total order for a PQ, so we only pay for what we need.

The concept of local vs. global conditions is ubiquitous in computing.

The End