Interface vs. Implementation

**Interface:** the operations of an ADT
- What you see on documentation web pages
- Method names and specifications
- Abstract from details: what to do, not how to do it
- Java syntax: `interface`

**Implementation:** the code for a data structure
- What you see in `source` files
- Fields and method bodies
- Provide the details: how to do operation
- Java syntax: `class`

Could be many implementations of an interface
- e.g. `List`: `ArrayList`, `LinkedList`

ADTs (interfaces)

<table>
<thead>
<tr>
<th>ADT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>Ordered collection (aka sequence)</td>
</tr>
<tr>
<td>Set</td>
<td>Unordered collection with no duplicates</td>
</tr>
<tr>
<td>Map</td>
<td>Collection of keys and values, like a dictionary</td>
</tr>
<tr>
<td>Stack</td>
<td>Last-in-first-out (LIFO) collection</td>
</tr>
<tr>
<td>Queue</td>
<td>First-in-first-out (FIFO) collection</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>Later this lecture!</td>
</tr>
</tbody>
</table>

Implementations of ADTs

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation (data structure)</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td><code>ArrayList</code>, <code>LinkedList</code></td>
</tr>
<tr>
<td>Set</td>
<td><code>HashSet</code>, <code>TreeSet</code></td>
</tr>
<tr>
<td>Map</td>
<td><code>HashMap</code>, <code>TreeMap</code></td>
</tr>
<tr>
<td>Stack</td>
<td>Can be done with a <code>LinkedList</code></td>
</tr>
<tr>
<td>Queue</td>
<td>Can be done with a <code>LinkedList</code></td>
</tr>
<tr>
<td>Priority Queue</td>
<td>Can be done with a heap — later this lecture!</td>
</tr>
</tbody>
</table>

Efficiency Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>Class</th>
<th><code>ArrayList</code></th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>prepend(val)</code></td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>get(i)</code></td>
<td>O(1)</td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>

Which implementation to choose depends on expected workload for application.
Priority Queues

Primary operation:
- Stack: remove newest element
- Queue: remove oldest element
- Priority queue: remove highest priority element

Priority:
- Additional information for each element
- Needs to be Comparable

<table>
<thead>
<tr>
<th>Priority</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Practice for swim test</td>
<td></td>
</tr>
<tr>
<td>Learn the Cornell Alma Mater</td>
<td></td>
</tr>
<tr>
<td>Study for 2110 prelim</td>
<td></td>
</tr>
<tr>
<td>Find Eric Andre ticket for sale</td>
<td></td>
</tr>
</tbody>
</table>

java.util.PriorityQueue<E>

```java
class PriorityQueue<E> {
    boolean add(E e); // insert e.
    E poll(); // remove & return min elem.
    E peek(); // return min elem.
    boolean contains(E e);
    boolean remove(E e);
    int size();
    ...
}
```

Implementations

<table>
<thead>
<tr>
<th>LinkedList</th>
<th>add()</th>
<th>put new element at front – O(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>poll()</td>
<td>must search the list – O(n)</td>
</tr>
<tr>
<td></td>
<td>peek()</td>
<td>must search the list – O(n)</td>
</tr>
</tbody>
</table>

Linked List that is always sorted

| add() | must search the list – O(n) |
| poll() | highest priority element at front – O(1) |
| peek() | same – O(1) |

Balanced BST

| add() | must search the tree & rebalance – O(log n) |
| poll() | same – O(log n) |
| peek() | same – O(log n) |

Can we do better?

Heaps
A Heap.

Is a binary tree satisfying 2 properties:

1) **Completeness.** Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.

Do not confuse with heap memory – different use of the word heap.

Completeness

Every level (except last) completely filled.
Nodes on bottom level are as far left as possible.

A Heap.

Is a binary tree satisfying 2 properties:

1) **Completeness.** Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.

2) **Heap-order.**
   
   **Max-Heap:** every element in tree is $\leq$ its parent
   
   **Min-Heap:** every element in tree is $\geq$ its parent

"max on top"
"min on top"

Heap-order (max-heap)

Every element is $\leq$ its parent

Note: Bigger elements can be deeper in the tree!

Piazza Poll #1
A Heap

Is a binary tree satisfying 2 properties

1) **Completeness.** Every level of the tree (except last) is completely filled. All holes in last level are all the way to the right.

2) **Heap-order.**

   Max-Heap: every element in tree is <= its parent

Primary operations:

1) **add(e):** add a new element to the heap
2) **poll():** delete the max element and return it
3) **peek():** return the max element

Priority queues

Heaps can implement priority queues

- Each heap node contains priority of a queue item
- (For values+priorities, see JavaHyperText)

Priority queues

Efficiency we will achieve:

- **add():** $O(\log n)$
- **poll():** $O(\log n)$
- **peek():** $O(1)$

- No linear time operations: better than lists
- **peek()** is constant time: better than balanced trees

Heap Algorithms

Heap: **add(e)**

1. Put in the new element in a new node (leftmost empty leaf)

Time is $O(\log n)$

Heap: **add(e)**

1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent
1. Save root element in a local variable

2. Assign last value to root, delete last node.

3. While less than a child, switch with bigger child (bubble down)

Time is $O(\log n)$

Heap: poll()

1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

Time is $O(1)$

Heap: peek()

1. Return root value

Tree implementation

```java
public class HeapNode<E> {
    private E value;
    private HeapNode left;
    private HeapNode right;
    ...
}
```

But since tree is complete, even more space-efficient implementation is possible...
Array implementation

```java
public class Heap<E> {  
    /* represent tree as array */
    private E[] heap;
    ...
}
```

Numbering tree nodes

- Number node starting at root
  row by row, left to right
- Same order as level-order traversal
  - Parent of node \( k \) is node \( \frac{k-1}{2} \)
  - Children of node \( k \) are nodes \( 2k+1 \) and \( 2k+2 \)

Represent tree with array

- Store node number \( i \) in index \( i \) of array \( b \)
- Children of \( b[k] \) are \( b[2k+1] \) and \( b[2k+2] \)
- Parent of \( b[k] \) is \( b[(k-1)/2] \)

```
class Heap<E> { 
    E[] b; // heap is b[0..n-1] 
    int n; 
    /** Create heap with max size */
    public Heap(int max) { 
        b = new E[max]; 
        // n == 0, so heap invariant holds 
        // (completeness & heap-order) 
    }
}
```

Constructor

```
class Heap<E> { 
    E[] b; // heap is b[0..n-1] 
    int n; 
    /** Create heap with max size */
    public Heap(int max) { 
        b = new E[max]; 
        // n == 0, so heap invariant holds 
        // (completeness & heap-order) 
    }
}
```

add() (assuming enough room in array)

```
class Heap<E> { 
    /** Add e to the heap */
    public void add(E e) { 
        b[n]= e; 
        n= n + 1; 
        bubbleUp(n - 1); // on next slide 
    }
}
```

add(). heap is in b[0..n-1]

```
class Heap<E> { 
    /** Bubble element #k up to its position. */ 
    * Pre: heap inv holds except maybe for k */
    private void bubbleUp(int k) { 
        int p= (k-1)/2; 
        // inv: p is parent of k and every element 
        // except perhaps k is <= its parent 
        while (k > 0 && b[k].compareTo(b[p]) > 0) {
            swap(b[k], b[p]); 
            k= p; 
            p= (k-1)/2; 
        }
    }
}
```
peek()

/** Return largest element 
 * (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    return b[0];  // largest value at root.
}

poll()

/** Remove and return the largest element 
 * (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    E v= b[0];  // largest value at root
    n= n – 1;  // move last
    b[0]= b[n];  // element to root
    bubbleDown(); // on next slide
    return v;
}

poll(). heap is in b[0..n-1]

/** Bubble root down to its heap position. 
Pre: b[0..n-1] is a heap except maybe b[0] */
private void bubbleDown() {
    int k= 0;
    int c= biggerChild(k);  // on next slide
    if (c < n && b[k] < b[c]) {
        swap(b[k], b[c]);
        k= c;
        c= biggerChild(k);
    }
}

poll()

/** Return index of bigger child of node k */
public int biggerChild(int k) {
    int c= 2*k + 2;  // k’s right child
    if (c >= n || b[c-1] > b[c])
        c= c-1;
    return c;
}

Efficiency

Class PriorityQueue<E> {
    boolean add(E e); //insert e.  log
    E poll(); //remove&return min elem. log
    E peek(); //return min elem.  constant
    boolean contains(E e);  linear
    boolean remove(E e);  linear
    int size();  constant
}

*IF implemented with a heap!
Heapsort

Goal: sort this array in place
Approach: turn the array into a heap and then poll repeatedly

// Make b[0..n-1] into a max-heap (in place)
// inv: b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted for (k = n-1; k > 0; k = k - 1)
// b[k] = poll - i.e., take max element out of heap.

// Make b[0..n-1] into a max-heap (in place)
// inv: b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted for (k = n-1; k > 0; k = k - 1)
// b[k] = poll - i.e., take max element out of heap.