Abstract vs concrete data structures

- **interface** List defines an "abstract data type".
  - It has methods: add, get, remove, ...
  - Various classes ("concrete data types") implement List:

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>O(n)</th>
<th>LinkedList</th>
<th>O(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(i, val)</td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add(0, val)</td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add(n, val)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get(i)</td>
<td>O(1)</td>
<td>O(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get(0)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Concrete Data Types

- Array
- LinkedList (singley-linked, doubly-linked)
- Trees (binary, general)
- Heaps

Heaps

- A heap is a binary tree with certain properties (it’s a concrete data structure)
  - Heap Order Invariant: every element in the tree is >= its parent
  - Complete Binary Tree: every level of the tree (except last) is completely filled, there are no holes

Do not confuse with heap memory, where the Java virtual machine allocates space for objects — different usage of the word heap.
Every element is $\geq$ its parent

Order Property

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

Completeness Property

Not a heap because it has two holes

Completeness Property

Not a heap because:
- missing a node on level 2
- bottom level nodes are not as far left as possible

Heaps

- A heap is a binary tree with certain properties (it's a concrete data structure)
- Heap Order Invariant: every element in the tree is $\geq$ its parent
- Complete Binary Tree: every level of the tree (except last) is completely filled, there are no holes
- A heap implements two key methods:
  - add(e): adds a new element to the heap
  - poll(): deletes the least element and returns it

add(e)

1. Put in the new element in a new node

add(e)
add(e) to a tree of size n

- Time is $O(\log n)$, since the tree is balanced
  - size of tree is exponential as a function of depth
  - depth of tree is logarithmic as a function of size

add(e)

- Add e at the leftmost empty leaf
- Bubble e up until it no longer violates heap order
- The heap invariant is maintained!

poll()
1. Save top element in a local variable.

2. Assign last value to the root, delete last value from heap.


• Save the least element (the root)
• Assign last element of the heap to the root.
• Remove last element of the heap.
• Bubble element down – always with smaller child, until heap invariant is true again.

   The heap invariant is maintained!
• Return the saved element.

Time is $O(\log n)$, since the tree is balanced.
Implementing Heaps

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

### Implementing Heaps

```java
public class Heap<E> {
    private E[] heap;
    ...
}
```

---

#### Numbering the nodes in a heap

Number node starting at root row by row, left to right

Level-order traversal

- Children of node \( k \) are nodes \( 2k + 1 \) and \( 2k + 2 \)
- Parent of node \( k \) is node \( \frac{k - 1}{2} \)

---

#### Store a heap in an array (or ArrayList) \( b \)!

- Heap nodes in \( b \) in order, going across each level from left to right, top to bottom
- Children of \( b[k] \) are \( b[2k + 1] \) and \( b[2k + 2] \)
- Parent of \( b[k] \) is \( b[k - 1]/2 \)

---

#### add() --assuming there is space

```java
/** An instance of a heap */
class Heap<E> {
    E[] b= new E[50]; // heap is b[0..n-1]
    int n= 0; // heap invariant is true

    /** Add e to the heap */
    public void add(E e) {
        b[n]= e;
        n= n + 1;
        bubbleUp(n - 1); // given on next slide
    }
}
```
poll(). Remember, heap is in b[0..n-1]

```java
/** Remove and return the smallest element 
 * (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    E v = b[0];     // smallest value at root.
    n = n - 1;      // move last
    b[0] = b[n];    // element to root
    bubbleDown(0);
    return v;
}
```

c’s smaller child

```java
/** Tree has n node. 
* Return index of smaller child of node k 
(2k+2 if k >= n) */
public int smallerChild(int k, int n) {
    int c = 2*k + 2;  // k’s right child
    if (c == n || b[c-1].compareTo(b[c]) < 0) {  
        c = c-1;
        return c;
    }
}
```

/** 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 = smallerChild(k, n);   // inv: b[0..n-1] is a heap except maybe b[k] AND  
    // b[c] is b[k]'s smallest child
    while (c < n && b[k].compareTo(b[c]) > 0) {
        swap(b[k], b[c]);
        k = c;
        c = smallerChild(k, n);
    }
}

Abstract Data Types

Some Abstract Data Types
- List
- Stack (LIFO) implemented using a List
- Queue (FIFO) implemented using a List
  - allows only add(0,val), remove(0) (push, pop)
  - allows only add(n,val), remove(0) (enqueue, dequeue)
    Both efficiently implementable using a singly linked list with head and tail
- PriorityQueue

Priority Queue
- Data structure in which data items are Comparable
- Smaller elements (determined by compareTo()) have higher priority
- remove() return the element with the highest priority = least element in the compareTo() ordering
- break ties arbitrarily
Many uses of priority queues (& heaps)

- Event-driven simulation: customers in a line
- Collision detection: "next time of contact" for colliding bodies
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- AI Path Planning: A* search
- Statistics: maintain largest M values in a sequence
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
- College: prioritizing assignments for multiple classes.

Priority queues as lists

- Maintain as a list
  - add() put new element at front – O(1)
  - poll() must search the list – O(n)
  - peek() must search the list – O(n)
- Maintain as an ordered list
  - add() must search the list – O(n)
  - poll() min element at front – O(1)
  - peek() O(1)

Can we do better?

Priority queues as heaps

- A heap is can be used to implement priority queues
- Gives better complexity than either ordered or unordered list implementation:
  - add(): O(log n) (n is the size of the heap)
  - peek(): O(1)
  - poll(): O(log n)

What if the priority is independent from the value?

Separate priority from value and do this:
  add(e, p); //add element e with priority p (a double)
THIS IS EASY!

Be able to change priority
change(e, p); //change priority of e to p
THIS IS HARD!

Big question: How do we find e in the heap?
Searching heap takes time proportional to its size! No good!
Once found, change priority and bubble up or down. OKAY

Assignment A6: implement this heap! Use a second data structure to make change-priority expected log n time

java.util.PriorityQueue<E>

interface PriorityQueue<E> {
  boolean add(E e) {...} //insert e.
  void clear() {...} //remove all elems.
  E peek() {...} //return min elem.
  E poll() {...} //remove/return min elem.
  boolean contains(E e)
  int size() {...}
  Iterator<E> iterator()
}

If implemented with a heap!