Abstract vs concrete data structures

- Abstract data structures are interfaces
  - specify only interface (method names and specs)
  - not implementation (method bodies, fields, …)
  - Have multiple possible implementations

- Concrete data structures are classes
  - These are the multiple possible implementations

Abstract data structures made concrete

<table>
<thead>
<tr>
<th>Interface</th>
<th>Class (implementation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>ArrayList, LinkedList</td>
</tr>
<tr>
<td>Set</td>
<td>HashSet, TreeSet</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap, TreeMap</td>
</tr>
<tr>
<td>Stack</td>
<td>can be done with a LinkedList</td>
</tr>
<tr>
<td>Queue</td>
<td>can be done with a LinkedList</td>
</tr>
</tbody>
</table>

2 classes that both implement List

- List is the interface ("abstract data type")
  - has methods: add, get, remove, ...

These 2 classes implement List ("concrete data types"):

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

Announcements

- A4 goes out today!
- Prelim 1:
  - regrades are open
  - a few rubrics have changed
- No Recitations next week (Fall Break Mon & Tue)
- We’ll spend Fall Break taking care of loose ends
Priority Queue

Unbounded queue with ordered elements
→ data items are Comparable (ties broken arbitrarily)

Priority order: smaller (determined by compareTo()) have higher priority

remove(): remove and return element with highest priority

Many uses of priority queues

Event-driven simulation: customers in a line
Collision detection: "next time of contact" for colliding bodies
Graph searching: Dijkstra’s algorithm, Prim’s algorithm
Art 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.

java.util.PriorityQueue<E>

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

Priority queues can be maintained 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)
- An ordered list
  - add(): must search the list – O(n)
  - poll(): min element at front – O(1)
  - peek(): O(1)
- A red-black tree (we’ll cover later)
  - add(): must search the tree & rebalance – O(log n)
  - poll(): must search the tree & rebalance – O(log n)
  - peek(): O(log n)

Can we do better?

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.

Completeness Property

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

Do not confuse with heap memory, where a process dynamically allocates space—different usage of the word heap.
Completeness Property

Not a heap because:
• missing a node on level 2
• bottom level nodes are not 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 Invariant. Max-Heap: every element in tree is \leq its parent
Min-Heap: every element in tree is \geq its parent

Order Property (max-heap)

Every element is \leq its parent

Heap Quiz #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 Invariant.
   Max-Heap: every element in tree is \leq its parent

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

Heap: add(e)

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

Time is $O(\log n)$

Heap: add(e)

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. Return root value

Time is $O(1)$

Heap: peek()

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

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

Level-order traversal

Parent of node $k$ is node $(k-1)/2$

Children of node $k$ are nodes $2k+1$ and $2k+2$

Storing a heap in an 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 $b[(k-1)/2]$

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 largest element */
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(0);
    return v;
}
```
/** Tree has n node. * Return index of bigger child of node k *(2k+2 if k \geq n) */
public int biggerChild(int k, int n) {
    int c = 2*k + 2; // k’s right child
    if (c >= n || b[c-1] > b[c])
        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 = biggerChild(k, n);
    // inv: b[0..n-1] is a heap except b[k] AND
    //      b[c] is b[k]'s biggest child
    while (c < n && b[k] < b[c])
    {
        swap(b[k], b[c]);
        k = c;
        c = biggerChild(k, n);
    }
}

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

Heap Quiz #2

// Make b[0..n-1] into a max-heap (in place)

HeapSort

Goal: sort this array in place
// 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--) {
// b[k]= poll -- i.e., take max element out of heap.
// }

Priority queues as heaps

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

java.util.PriorityQueue\(<E>\)

interface PriorityQueue\(<E>\) {
  boolean add\( (E e) \); // insert e. \text{ log }
  void clear(); // remove all elements.
  \( E \) peek(); // return min elem. \text{ constant }
  \( E \) poll(); // remove/return min elem. \text{ log }
  boolean contains\( (E e) \); \text{ linear }
  boolean remove\( (E e) \); \text{ linear }
  int size(); \text{ constant }
  Iterator\(<E>\) iterator(); \text{ *IF implemented with a heap! }
}

What if priority is independent from the value?

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

Be able to change priority
  \change(e, p); // change priority of e to p

This is easy!

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 A4: implement this heap! Use a second data structure to make change-priority expected \( \log n \) time