PRIORITY QUEUES AND HEAPS

Lecture 17
CS2110 Fall 2016
Read Chapter 26 “A Heap Implementation” to learn about heaps

Exercise: Salespeople often make matrices that show all the great features of their product that the competitor’s product lacks. Try this for a heap versus a BST. First, try and sell someone on a BST: List some desirable properties of a BST that a heap lacks. Now be the heap salesperson: List some good things about heaps that a BST lacks. Can you think of situations where you would favor one over the other?
Stacks and queues are restricted lists

- Stack (LIFO) implemented as list
  - `add()`, `remove()` from front of list (push and pop)
- Queue (FIFO) implemented as list
  - `add()` on back of list, `remove()` from front of list
- These operations are $O(1)$

Both efficiently implementable using a singly linked list with head and tail

![Linked List Diagram]
Interface Bag (not In Java Collections)

```java
interface Bag<E>
    implements Iterable {
    void add(E obj);
    boolean contains(E obj);
    boolean remove(E obj);
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
}
```

Also called multiset

Like a set except that a value can be in it more than once. Example: a bag of coins

Refinements of Bag: Stack, Queue, PriorityQueue
Priority queue

- **Bag** 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

Surface simplification [Garland and Heckbert 1997]
```java
import java.util.*;

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)
    boolean remove(E e)
    int size() {...}
    Iterator<E> iterator()
}
```
Priority queues as lists

• Maintain as unordered 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 ordered list
  – `add()` must search the list – O(n)
  – `poll()` wanted element at top – O(1)
  – `peek()` O(1)

Can we do better?
A heap is a concrete data structure that can be used to implement priority queues

Gives better complexity than either ordered or unordered list implementation:
- \texttt{add}(): \(O(\log n)\) (n is the size of the heap)
- \texttt{poll}(): \(O(\log n)\)

\(O(n \log n)\) to process \(n\) elements

Do not confuse with \textit{heap memory}, where the Java virtual machine allocates space for objects – different usage of the word \textit{heap}
Heap: first property

Every element is $\geq$ its parent

Note: 19, 20 < 35: Smaller elements can be deeper in the tree!
Heap: second property: is **complete**, has no holes

Every level (except last) completely filled.

Nodes on bottom level are as far left as possible.
Heap: Second property: has no “holes”

Not a heap because it has two holes

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

- Binary tree with data at each node
- Satisfies the *Heap Order Invariant*:

  1. Every element is $\geq$ its parent.

- Binary tree is **complete** (no holes)

  2. Every level (except last) completely filled. Nodes on bottom level are as far left as possible.
Numbering the nodes in a heap

Number node starting at root in breadth-first left-right order

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

Parent of node $k$ is node $(k-1)/2$
Can store a heap in an array $b$
(could also be ArrayList or Vector)

- 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 $b[(k - 1)/2]$

Tree structure is implicit.
No need for explicit links!
add(e)
add(e)

1. Put in the new element in a new node
2. Bubble new element up if less than parent
add ()

2. Bubble new element up if less than parent
add()
add()
add()  

2. Bubble new element up if less than parent
add()
add()

2. Bubble new element up if less than parent
add()
add(e)

- Add e at the end of the array
- Bubble e up until it no longer violates heap order
- The heap invariant is maintained!
add() 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
/** 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
    }
}
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 elmnt // 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;
        }
    }
}

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

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

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

3. Bubble root value down
poll()

3. Bubble root value down
poll()

3. Bubble root value down
poll()

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

2. Assign last value to the root, delete last value from heap
`poll()`

2. Assign last value to the root, delete last value from heap
4 5

3. Bubble root value down
$\text{poll()}$

3. Bubble root value down
poll()

3. Bubble root value down
poll()
poll()

3. Bubble root value down
poll()
/** 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);
    }
}
Change heap behaviour a bit

Separate priority from value and do this:

```cpp
add(e, p);    //add element e with priority p (a double)
```

THIS IS EASY!

Be able to change priority

```cpp
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
HeapSort(b, n) — Sort b[0..n-1]

Wet your appetite – use heap to get exactly \( n \log n \) in-place sorting algorithm. 2 steps, each is \( O(n \log n) \)

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

2. for \( (k = n-1; k > 0; k = k-1) \) {
   b[k] = poll — i.e. take max element out of heap.
}

A max-heap has max value at root