Priority Queues and Heaps

Lecture 17
CS2110 Fall 2011

Priority Queues

The Bag Interface

```
interface Bag<E> {
    void insert(E obj);
    E extract(); //extract some element
    boolean isEmpty();
}
```

Examples: Stack, Queue, PriorityQueue

Stacks and Queues as Lists

- Stack (LIFO) implemented as list
  - `insert()`, `extract()` from front of list
- Queue (FIFO) implemented as list
  - `insert()` on back of list, `extract()` from front of list
- All Bag operations are O(1)

Priority Queue

- A Bag in which data items are Comparable
- Lesser elements (as determined by `compareTo()`) have higher priority
- `extract()` returns the element with the highest priority = least in the `compareTo()` ordering
- break ties arbitrarily

Priority Queue Examples

- Scheduling jobs to run on a computer
  - default priority = arrival time
  - priority can be changed by operator
- Scheduling events to be processed by an event handler
  - priority = time of occurrence
- Airline check-in
  - first class, business class, coach
  - FIFO within each class

Java.util.PriorityQueue<E>

```
boolean add(E e) {...} //insert an element (insert)
void clear() {...} //remove all elements
E peek() {...} //return min element without removing
            //(null if empty)
E poll() {...} //remove min element (extract)
            //(null if empty)
int size() {...}
```
Priority Queues as Lists

- Maintain as unordered list
  - \texttt{insert()} puts new element at front – O(1)
  - \texttt{extract()} must search the list – O(n)

- Maintain as ordered list
  - \texttt{insert()} must search the list – O(n)
  - \texttt{extract()} gets element at front – O(1)

- In either case, \(O(n^2)\) to process \(n\) elements

Can we do better?

Important Special Case

- Fixed number of priority levels \(0, \ldots, p - 1\)
- FIFO within each level
- Example: airline check-in

- \texttt{insert()} – insert in appropriate queue – O(1)
- \texttt{extract()} – must find a nonempty queue – O(p)

Heaps

- 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{insert()}: O(log n)
  - \texttt{extract()}: O(log n)
- \(O(n \log n)\) to process \(n\) elements
- Do not confuse with heap memory, where the Java virtual machine allocates space for objects – different usage of the word heap

Examples of Heaps

- Ages of people in family tree
  - parent is always older than children, but you can have an uncle who is younger than you

- Salaries of employees of a company
  - bosses generally make more than subordinates, but a VP in one subdivision may make less than a Project Supervisor in a different subdivision
Balanced Heaps

Two restrictions:

1. Any node of depth < d – 1 has exactly 2 children, where d is the height of the tree
   – implies that any two maximal paths (path from a root to a leaf) are of length d or d – 1, and the tree has at least $2^d$ nodes

2. All maximal paths of length d are to the left of those of length d – 1

A Balanced Heap

Store in an ArrayList

- Elements of the heap are stored in the array in order, going across each level from left to right, top to bottom
- The children of the node at array index n are found at $2n + 1$ and $2n + 2$
- The parent of node n is found at $(n – 1)/2$

Store in an ArrayList or Vector

insert()

- Put the new element at the end of the array
- If this violates heap order because it is smaller than its parent, swap it with its parent
- Continue swapping it up until it finds its rightful place
- The heap invariant is maintained!
insert()

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

class PriorityQueue<E extends Comparable<E>> extends ArrayList<E> {

  public void put(E obj) {
    add(obj); // add new element to end of array
    rotateUp(size() - 1);
  }

  private void rotateUp(int index) {
    if (index == 0) return;
    int parent = (index - 1)/2;
    if (get(parent).compareTo(get(index)) <= 0) return;
    swap(index, parent);
    rotateUp(parent);
  }

insert()

insert()

insert()

extract()

• Remove the least element – it is at the root
• This leaves a hole at the root – fill it in with the last
element of the array
• If this violates heap order because the root element is
too big, swap it down with the smaller of its children
• Continue swapping it down until it finds its rightful
place
• The heap invariant is maintained!
**HeapSort**

Given a **Comparable** array of length n,
- Put all n elements into a heap – O(n log n)
- Repeatedly get the min – O(n log n)

```java
public static <E extends Comparable<E>> void heapSort(E[] a) {
    PriorityQueue<E> pq = new PriorityQueue<E>();
    for (E x : a) pq.put(x);
    for (int i = 0; i < a.length; i++) a[i] = pq.get();
}
```

**PQ Application: Simulation**

- Example: Probabilistic model of bank-customer arrival times and transaction times, how many tellers are needed?
  - Assume we have a way to generate random inter-arrival times
  - Assume we have a way to generate transaction times
  - Can simulate the bank to get some idea of how long customers must wait
- Time-Driven Simulation
  - Check at each tick to see if any event occurs
- Event-Driven Simulation
  - Advance clock to next event, skipping intervening ticks
  - This uses a PQ!