CMPSCI 187: Programming With Data Structures

Lecture #30: Priority Queues and Heaps
David Mix Barrington
21 November 2012

Priority Queues and Heaps

- The Definition of Priority Queues
- Possible Implementations
- The Idea of a Heap
- Implementing Heaps With Implicit Pointers
- Reheaping Down to Dequeue
- Reheaping Up to Enqueue
- An Additional Method: `newHole`

The Definition of Priority Queues

- A **priority queue** is a collection where the elements come from an ordered type, and where the removal operation gives us not the newest element (as in a stack) or the oldest element (as in a queue) but the *largest* element according to the order. We might call this BIFO for “best-in-first-out”, as opposed to FIFO for a queue and LIFO for a stack.

- There are many situations where the next item in a list to tackle is the most important according to some measure. Operating systems have a relative priority on processes and favor the ones with highest priority in timesharing.

```
public interface PriQueueInterface<T extends Comparable<T>> {
   boolean isEmpty( );
   boolean isFull( );
   void enqueue(T element); // throws PQOverflowException
   T dequeue( );} // throws PQUnderflowException
```

Possible Implementations

- If we used an unsorted list to implement a priority queue, we could enqueue in O(1) time but we would need O(N) to dequeue from a queue of size N.
- With an array-based sorted list, we could dequeue in O(1) time but would need O(N) to enqueue, because in the worst case we must move O(N) elements.
- With a reference-based sorted list, dequeuing is again O(1) time but now enqueueing requires O(N) time in the worst case to find the place to insert.
- In a binary search tree, both enqueueing and dequeueing require a trip from the root of the tree down to a leaf. This is O(log N) time if the tree is balanced, but could be as bad as O(N) if it is not.
- We'll achieve guaranteed O(log N) time by using a heap,a tree structure that is always balanced after each operation.

The Idea of a Heap

- A **heap** (in this case a **max-heap**) is a **complete** tree of comparable elements that satisfies the **heap property**: the element at every node is larger than both the elements at its children if there are any. Hence it must also be larger than the elements at any of its descendants. In particular, the largest element must be at the root. (In a min-heap, each node's element is *smaller* than the elements at its children.)

- Remember that a complete binary tree has its leaves on one level or on two adjacent levels -- in the latter case the leaves on the upper level all exist and those on the lower level are left-justified.

- A heap is "somewhat sorted". We can find the maximum element quickly, but the farther down we go the less we know about the relative order of elements. The key virtue of the heap is that we can insert a new element or remove the maximum element (the PQ operations) and restore the heap property quickly.

Implementing Heaps With Implicit Pointers

- As we mentioned in Lecture #28, we can implement a tree structure with an array by using **implicit pointers**: the left child of node i is node 2i + 1, and the right child is node 2i + 2. An array of length n corresponds to an n-node complete binary tree with this convention.

- We use the `java.util` class `ArrayList`, whose methods are described in the API and in DJW's Appendix E.

```
public class Heap<T extends Comparable<T>>
                    implements PriQueueInterface<T> {
   private ArrayList<T> elements;
   private int lastIndex, maxIndex;
   public Heap (int maxSize) {
      elements = new ArrayList<T> (maxSize);
      lastIndex = -1; maxIndex = maxSize - 1;}
   public boolean isEmpty( ) {return (lastIndex == -1);}
   public boolean isFull( ) {return (lastIndex == maxIndex);}
```

Reheaping Up to Enqueue

- When we enqueue an element, our heap becomes larger by one element. We know exactly where the new element must go, in the next available array slot, so we can add it there, but this could destroy the heap property.

- We will fix the property by **reheaping up**. We think of the place for the new element as a hole. If putting the new element in the hold would violate the heap property, we move the parent element down into the hole and repeat the process with a new hole in the parent's position. We continue this way until the new element may go in the hole, which might not happen until we reach the root. This means shifting up to O(log n) elements and takes O(log n) time.

```
public void enqueue (T element) throws PriQOverflowException {
   if (lastIndex == maxIndex)
      throw new PriQOverflowException (“Priority Queue is full”);
   lastIndex++;
   elements.add (lastIndex, element);
   reheapUp (element);}
```

Code for `reheapUp`

- The process of moving the hole upward looks like a good candidate for recursion, but we can manage it about as simply with a loop. We have a problem as long as the hole's parent has a element smaller than the element we are trying to place, unless we've moved the hole to the root, where no element is too big. So we use the basic programming paradigm of "while there is a problem, do something about it". Once we have a suitable place for the element, we just put it there.

```
private void reheapUp (T element) {
   int hole = lastIndex;
   while ((hole > 0) &&
          (element.compareTo(elements.get((hole - 1)/2)) > 0) {
      elements.set(hole, elements.get((hole - 1)/2));
      hole = (hole - 1)/2;}
   elements.set (hole, element);}
```

Reheaping Down to Dequeue

- We know that we want to return the element in the root, but we have to adjust the heap before we do that. The element from the former last location has to go into some remaining slot in the heap. If it may go into the root we are done, but otherwise (if it is smaller than one of the root's children) we must **reheap down**, promoting elements to move the hole down until we can fill it.

```
public T dequeue( ) throws PriQUnderflowException {
   T hold, toMove;
   if (lastIndex == -1)
      throw new PriQUnderflowException ("Priority queue is empty");
   else {
      hold = elements.get(0);
      toMove = elements.remove(lastIndex);
      lastIndex--;
      if (lastIndex != -1)
         reheapDown(toMove);
      return hold;}}
```

Code for `reheapDown`

- We want to move the hole downward until it no longer has children that are too big. The way we do this is to promote the element from the larger of the hole's children into the hole -- it may go there because it is larger than (or equal to) its new children.

- The comparisons to determine which element to promote are a bit tedious and DJW leave them to an auxiliary method for clarity. The `newHole` method returns the index of the larger child if that child's element is too big, and returns the index of the hole itself if it is not. (In the latter case we leave the loop.)

```
private void reheapDown (T element) {
   int hole = 0; newhole;
   newhole = newHole (hole, element);
   while (newhole != hole) {
      elements.set(hole, elements.get(newhole));
      hole = newhole;
      newhole = newHole (hole, element);}
   elements.set (hole, element);}
```

An Additional Method: `newHole`

- I've eliminated some of DJW's uses of `else` that are made redundant by the `return` statements. We have special cases for the hole having less than two children -- if it has two we promote the larger, if it is bigger than `element`.

```
private int newHole (int hole, T element) {
   int left = (hole * 2) + 1, right = (hole * 2) + 2;
   if (left > lastIndex) return hole; // hole has no children
   if (left == lastIndex)             // hole has one child
      if (element.compareTo(elements.get(left)) < 0)
         return left;
      else return hole;
   if (elements.get(left).compareTo(element.get(right)) < 0)
      if (elements.get(right).compareTo(element) <= 0)
         return hole;
      else return right;
   else if (elements.get(left).compareTo(element) <= 0)
      return hole;
   else return left;}
```