CMPSCI 187: Programming With Data Structures

Lecture #31: Using Heaps (Priority Queues, Heapsort)
30 November 2011

Using Heaps (Priority Queues, Heapsort)

- Priority Queues as a General Concept
- The Heap Implementation
- HeapSort
- Building a Heap from an Unsorted Array
- Comparing Sorting Algorithms
- Beyond Heaps?

Priority Queues as a General Concept

- We've seen the Java `PriorityQueue` class, and mentioned the simpler `PriorityQueue` generic class in L&C. They don't give an ADT, but if they did it would be the simple one below.

- The elements of our priority queues are pairs consisting of an object and its priority. We can assign a priority to an object when we add it, and when we reach for an object from the queue we get the one with the highest priority (or lowest -- we decide at the beginning). Because it's a queue, elements with equal priority are first-in-first-out.

- We saw priority queues in best-path search. They are also used in networking, operating systems, and for sorting (as in Project #7).

```
public interface PriorityQueueADT<T> {
   public void addElement (T object, int priority);
   public T removeNext ( );}
```

The Heap Implementation

- Last time we implemented heaps by linked structures and by arrays, and either implementation can be used for a priority queue. The simple code below hides a bit in the definition of `PQNode`, but essentially just gives new names to the heap operation.

- The class `PQNode` needs a compareTo method that will make the priority queue respect the first-in-first-out rule. Let's look at the code for this next.

```
public class PriorityQueue<T> extends ArrayHeap<PQNode<T>> {
   public PriorityQueue( ) {super( );}
   public void addElement (T object, int priority) {
      PQNode<T> node = new PQNode<T> (object, priority);
      super.addElement (node);}
   public T removeNext ( ) {
      PQNode<T> temp = (PQNode<T>) super.removeMin( );
      return temp.getElement( );}}
```

The **`PQNode`** Class

- The static variable marks each newly created `PQNode` object with its creation number, so that when two equal-priority nodes are compared, the first-created one wins the comparison. L&C are confused at times whether low or high priority numbers come out first, but the `compareTo` method will decide.

```
public class PQNode<T> implements Comparable<PQNode<T>> {
   private static int nextOrder = 0;
   private int priority, order; private T element;
   public PQNode (T obj, int prio) {
      element = obj; priority = prio;
      order = nextOrder; nextOrder++;}
   // public get methods for the three instance fields, toString
   public int compareTo (PQNode<T> obj) {
      int result;
      PQNode<T> temp = obj; // do I need the red <T>'s for this?
      if (priority > temp.getPriority( )) result = 1;
      else if (priority < temp.getPriority( )) result = -1;
      else if (order > temp.getOrder( )) result = 1;
      else result = -1;}}
```

HeapSort

- This code from L&C is the simplest form of HeapSort -- it loads the given segment of the input array into a heap, then pulls the elements out one by one in order and puts them back. Clearly any priority queue can be used to sort in this way. But do we need to have storage for two copies of the data?

- Not if we're clever -- we take the input array, swap enough elements to make it satisfy the maxheap property as an array, then repeatedly pull out the largest element and put it in its position, contracting the heap to leave it out. There's a nice animation of this in the Wikipedia article "Heapsort".

```
public class HeapSort<T> {
   public void HeapSort (T[ ] data, int min, int max) {
      ArrayHeap<T> temp = new ArrayHeap<T>( );
      for (int ct = min; ct <= max; ct++)
         temp.addElement (data[ct]);
      int count = min;
      while (!temp.isEmpty( )) {
         data[count] = temp.removeMin( );
         count++;}}}
```

Building a Heap From an Unsorted Array

- If I have an arbitrary array, I can think of element 0 as an ArrayHeap of size 1. I can insert element 1 into the heap, then element 2, and so on, so that I make an ArrayHeap of size n in time O(n log n). (Most inserts take O(log n) time.)

- I can actually do this more cleverly and form the unsorted array into a heap in O(n) time. This doesn't change the overall O(n log n) for heapsort but might reduce the leading constant.

- The trick is to work *downward* in the index number. I find the highest non-leaf, and with at most one swap make it the root of a little heap. I then do the same for the previous element, the one before, and so on until we have a heap.

- L&C **incorrectly** say that each of these O(n) operations is O(1) time. But the O(n) total is correct -- we have one swap for half the elements, two for n/4 of them, three for n/8, four for n/16, and so on. There are various ways to see that this sum is O(n) -- I'll show a simple one on the blackboard.

Comparing Sorting Algorithms

- We've now seen three O(n log n) comparison-based sorting algorithms: MergeSort, QuickSort, and HeapSort.
- From the argument we sketched in Discussion #9, we can't sort by comparisons alone using fewer than log (n!) = O(n log n) in the worst case.
- MergeSort can be applied without random access. But it needs O(n) additional storage to sort n items, while QuickSort and HeapSort **sort in place** or near enough -- they need only O(log n) space for a stack in QuickSort and only space for O(1) extra items in HeapSort.
- QuickSort is the fastest of the three on average for random permutations, but as we saw it has a bad worst-case behavior. HeapSort is nearly as fast (or maybe faster, in principle, in one variant) and is consistent for all inputs.

Beyond Heaps?

- Heaps are very nice data structures, carrying out the priority queue operations in O(log n) time in the worst case. We can easily do some other operations in the same time bound, such as removing an arbitrary element (once it is found), or changing the priority of a node. But some things are still O(n), like taking two heaps and merging them into a third.

- There has been a huge amount of research into better data structures, some of only theoretical interest and some practical. (You'll see some of this work in CMPSCI 311.)

- A structure called a **Fibonacci heap** gives O(1) **amortized** time for all but the two delete operations, and O(log n) amortized for them. It keeps the data in a list of trees obeying the heap property, and reorders items only on deletes.

- A structure called a **Brodal queue** gives O(log n) time in principle for the deletes and O(1) worst case in principle for the others, but it is not practical.