

CS231 Fall 2013 Lecture 25

Stephanie R Taylor

November 1, 2013

1 Administrative Topics

- We take the quiz.

2 Implementing PriorityQueues with heaps

A heap is a complete binary tree in which:

1. the elements are comparable, and
2. no child is greater than its parent (it may be equal).

A complete tree (as compared to a fully complete tree) is full on all levels except possibly the bottommost level. On that level, all leaves are as far to the left as possible.

[Give an example]

Note: This is different from the heap we talk about in CS232, where we are referring to that part of memory usually between the stack and the code where objects are created.

Note that if a tree is a heap, then every subtree is also a heap.

We will implement a PriorityQueue using a heap. To do so efficiently, we need to be able to quickly find the last node, add new or remove nodes in

the next available location in the bottom level of the tree, and swap data between parent and children. Therefore we will implement the heap using an array!

[Note: We implement PQSort with a PQ, we implement a PQ with a heap, we implement a heap with an array. Isn't this great!?!]

Because the heap is complete, there will be no unused slots in the array mixed in with the used slots. We can easily find the last item and add or remove items at the end and we can easily move between parents and children. As we add and delete nodes from a heap, we will always “reheapify” in such a way that the graph is made complete as well as a heap.

```
public class PQ {
    //instance variable
    private Comparable[] heap; //null means a nonexistent node
    private int size;

    //constructor
    public PQ() {
        heap = new Comparable[20]; //default initial size
        size = 0;
    }

    // the rest of the PQ methods
}
```

In-class exercises: Assume the array is long enough so that every non-empty node has two children in the array. 1) Write a void add(Comparable obj) method that adds a new item to the PQ. Note: The heap field must still have the heap property after the insertion! How do we restore heapness if it is absent? To answer that, we must figure out how the heap property is violated. If it is illegal, it will be illegal because the new element is “bigger” than its parent. Our strategy will be to work our way back up the tree, beginning with the newly added node, swapping the “current” element with its parent, if the current element is larger than its parent.

```
public void add(Comparable obj) {
    // first enlarge array if necessary
    if (this.size == heap.length) {
        Comparable[] nh = new Comparable[heap.length * 2];
        for (int i = 0; i < heap.length; i++) {
            nh[i] = heap[i];
        }
    }
}
```

```

        heap = nh;
    }
    //then add new value at end of array.
    heap[size] = obj;
    //now reheapify
    moveUp(size);
    size++;
}

// move the value data[index] up the heap to its proper position
private void moveUp(int index) {
    int pIdx = (index-1)/2;
    if ( index > 0 &&
        heap[index].compareTo(heap[pIdx]) > 0) {
        //swap the values in the two nodes
        Comparable temp = heap[index];
        heap[index] = heap[pIdx];
        heap[pIdx] = temp;
        //move up further
        moveUp(pIdx);
    }
}

```

2) Write an remove method that removes the highest priority item. The resulting heap field must still have the heap property. To reheapify in this case, we need to move the last item into the root position, then work our way down the tree, swapping items that violate the heap property. On the way down the tree, we make sure we promote the larger child to parenthood.

```

public Comparable remove() {
    if (size == 0)
        return null;
    Comparable valueToReturn = heap[0];
    //move last node in tree into head position
    heap[0] = heap[size-1];
    size--;
    //now reheapify
    moveDown(0);
    return valueToReturn;
}

// move the value heap[index] down the heap represented by
// the values in heap[0]..heap[end-1]
private void moveDown(int index)
{

```

```

int lkid = index*2+1;
int rkid = index*2+2;
if( lkid >= size )
    return; //heap[index] is a leaf so stop

int largerChildIndex = lkid;
if( rkid < size &&
    heap[largerChildIndex].compareTo(heap[rkid])<0 )
    largerChildIndex = rkid;
if(heap[largerChildIndex].compareTo(heap[index])>0 ) {
    //swap heap[index] with heap[largerChildIndex]
    Comparable temp = heap[largerChildIndex];
    heap[largerChildIndex] = heap[index];
    heap[index] = temp;
    //move down further
    moveDown(largerChildIndex);
}
}

```

What are the time complexities of each of these methods? They each need to examine one element at each level of the tree. A complete binary tree has $\log_2 n$ levels. So each method is $O(\log n)$.

2.1 Sorting

Using PQSort, the data in the initial array A are added to the heap array in the PQ and then are extracted from that array and put back in A.

What are the time and space complexities of this algo? [discuss this.] The time complexity is $\log n$ for each add and remove. So the total time is $O(n \log n)$. The space required is the original A plus the space used by the heap, which will be another array of at least the same size.

Can we do better in terms of space? It turns out that there is a way to reuse array A as the heaps array and so the only addition storage necessary is $O(1)$.

2.2 Heap sort in place

(This part will not be discussed in class, but I include it for your personal edification.)

If you reuse the same array instead of a new one inside a PQ object, you can simplify the process immensely. Think of what is happening in the PQs array and As array as you do PQSort. The items from A are gradually added to the PQs array [show this happening with a sample array, such as [3, 6, 2, 7, 1, 4, 8] where the part of A still needed does not overlap the part of the heap needed. Then show the evolution of the array as you add and remove items from the heap.]

How does this time complexity compare to other sorts we've seen? [The time complexity of this sort procedure is as good as any we've seen.] Question to think about: Are there any better sorting algorithms? [let them guess]

```
/*
 * File:   HeapSorter.java
 * Author: Dale Skrien
 * Course: CS231 Fall 2011
 */

public class HeapSorter
{
    /**
     * sorts A using a HeapPQ
     */
    public static void HeapSort(int [] A)
    {
        HeapPQ pq = new HeapPQ(A.length);

        for( int i = 0; i < A.length; i++ ) {
            pq.add(A[i]);
        }

        for( int i = A.length-1; i >= 0; i-- ) {
            A[i] = pq.remove();
        }
    }

    /**
     * sorts A in place by making it into a heap.
     */
}
```

```

public static void HeapSortInPlace(int [] A)
{
    for( int i = 1; i < A.length; i++ ) {
        moveUp(i, A);
    }
    for( int i = A.length-1; i >= 1; i-- ) {
        swap(A, 1, i);
        moveDown(1,i,A);
    }
    //now put A[0] in its proper place
    int i = 0;
    while(i < A.length-1 && A[i] > A[i+1]) {
        swap(A, i, i+1);
        i++;
    }
}

//move the value data[index] up the heap to its proper position
private static void moveUp(int index, int [] data)
{
    while( index > 1 && data[index] > data[index/2] ) {
        swap(data, index, index/2);
        //move up further
        index = index/2;
    }
}

//move the value data[index] down the heap represented by
//the values in data[1]..data[end-1]
private static void moveDown(int index, int end, int [] data)
{
    while( 2*index < end ) {
        int indexOfLargerChild = 2*index; // largest is left child
        if( 2*index+1 < end && //if there is a right child
            data[index*2+1] > data[index*2] ) //and it's greater
            indexOfLargerChild = index*2+1; //largest is right child
        if( data[indexOfLargerChild] <= data[index] )
            return; // we're done moving down
        else {
            swap(data, index, indexOfLargerChild);
            //move down further
            index = indexOfLargerChild;
        }
    }
}

```

```

private static void swap(int[] A, int i, int j)
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

public static void main(String[] args) {
    int[] A = {5,1,3,8,2,7,4,6};
    HeapSort(A);
    for(int x : A)
        System.out.print(x + " , ");
    System.out.println();
    A = new int[] {5,1,3,8,2,7,4,6};
    HeapSortInPlace(A);
    for(int x : A)
        System.out.print(x + " , ");
    System.out.println();
}
}

```