

SORTING WITH A COMPLETE HEAP BINARY TREE IMPLEMENTATION OF A PRIORITY QUEUE:

THE HEAP PRIORITY QUEUE SORT.

In a heap PQ sort, we insert into the priority queue, maintaining heap order, and then extract from the priority queue in key order.

Since both insertion and removal are $O(n \log_2 n)$, it follows that the heap sort algorithm must be $O(n \log_2 n)$ as well.

Compare with $O(n^2)$ for both the insertion and select sort earlier.

CONTAINERS FOR A COMPLETE HEAP BINARY TREE IMPLEMENTATION OF A PRIORITY.

There are two approaches to the data container for heap sorting:

1. Use a vector underlying a binary tree
2. Use linked nodes underlying a binary tree.

It is important to grasp that the $O(n \log_2 n)$ of the heap PQ sort does not depend on our choice of either vector or linked nodes as the container for the priority queue.

1. Vector implementation approaches.

(a) Use a vector class, and binary tree classes

Use an existing binary tree class built on top of a vector class. Then build the priority queue class on top of the binary tree class, i.e use a binary tree object as a data member, and use binary tree class methods internally in the five priority queue class methods. This is what is done in the text book.

The advantage is that we do a little less coding (maybe), and so gain from object-oriented programming techniques. Disadvantages are possibility of errors due to programming confusion over a plethora¹ of methods. There will likely be a little longer running time because of use of irrelevant code in the underlying binary tree and vector methods.

¹*plethora* -- over abundance of

(b) Specialty vector approach.

Only a few methods are needed for a priority queue, as we have seen, so it is no trouble to design a vector that is both specially designed for holding entries, and for keeping the collection of entries as a binary tree in heap order. These entries could be stored in an array where we never use slot zero. We can then use only the priority queue methods to manipulate the array directly. We will have a little more coding (maybe), but will have tighter code that is a lot easier to follow.

2. Linked node implementation approaches.

(a) Using a binary tree class built on linked nodes

Here we use an existing binary tree class, built on top of a linked node class. Then, in turn we build a priority queue class on top of the binary tree class. We do a little less coding (maybe), and so gain from object-oriented programming techniques, but, once more, probably lose a bit in efficiency because of extra unneeded code execution.

The programmer also needs to know all about the relevant binary tree methods, and may get lost in a maze of method and class names. In other words, while not difficult, the code may be hard to follow.

(b) Specialty linked node binary tree approach.

Once more, only a few methods are needed for a priority queue, as we have seen, so it is no trouble to design a node specially for entries, and then build a binary tree with these linked nodes, using essentially only the priority queue methods to manipulate the list. We will have a little more coding (maybe), but will have tighter code that is a lot easier to follow. This is the approach required in Assignment 4.

Implementation choices made

We will take the simplest possible approach to get to understand what this heap technique is all about. In contrast the textbook uses the most complex approach possible.

We thus use the *Specialty vector approach*. To be specific:

- (a) The PQ class will use an array as the container.
- (b) The array will hold just entries, each consisting of a key and a value. There will be a separate inner class for entries.
- (c) Slot zero is never used, and slot 1 holds the root. The slot with its index value equal to the size of the queue is the last slot of the heap.
- (d) The array will behave like a vector to the extent needed, i.e., the queue will double in capacity if it runs out of slots.
- (e) A user of the PQ class can supply either his/her own comparator, or fall back on a default comparator.
- (f) We closely follow the logic of the more complex multileveled approach of the text book on pages 333-335 [*more about the text book version later.*] A user should not notice any difference between this *simplest possible* implementation and the *most complex possible* (!) implementation in the textbook.

Need for an Entry class (as given in Lec 23)

Users of a priority queue instance P insert a pair of data items, the *value* and the *key*, into P when using it.

However, it is convenient for $\langle \textit{key}, \textit{value} \rangle$ to be handled together in a class of ultimate type *Entry*, whose two data members are *value* and the *key*.

This is particularly true if we are using a standard linked list as the container. The *single* element *e* stored in each linked list node will be of type *Object*, and it is then convenient to store an *Entry* instance as the element *e*.

If *Entry* is defined as an interface, the necessary class is:

```
public class MyEntry implements Entry {
    protected Object    k; //key component
    protected Object    v; //value component
//constructor
    public              MyEntry (Object key, Object value){
                                k = key; v = value;  }
//methods
    public Object       key() { return k;}
    public Object       value() { return v;} }
```

where

```
public interface E
```

The priority queue class

VectorCHBTPriorityQueue

(a vector-containerized, complete, heap-ordered binary-tree priority queue)

Note: *The class container is an array of entries, initial capacity 64. Capacity will double automatically, as needed. The container thus has all the vector capability needed.*

```
public class VectorCHBTPriorityQueue {
    protected Entry []      V //array of entries (k + v)
    protected int           PQSize;
    protected int           capacity = 64;
    protected Comparator   c;

    //constructor generates a PQ with user-defined comparator
    public VectorCHBTPriorityQueue (Comparator cptr) {
        V = new Entry [capacity];
        PQSize = 0;    c = cptr;
        V[0] = null; //never used}

    //constructor generates empty PQ with default comparator
    public VectorCHBTPriorityQueue() {
        V = new Entry [capacity];
        PQSize = 0;
        c = new DefaultComparator();
        V[0] = null; // never used }

    //Necessary DefaultComparator class is defined later.
    //An inner Entry class is also defined later
```

//The five standard PQ methods follow

```
public int          size() { return PQSize };

public boolean     isEmpty() {
    if (PQSize == 0) return true;
    else return false; }

public Entry      min() throws
    EmptyPQException {
    if (isEmpty) throw new
        EmptyPQException ("Empty PQ");
    return V[1];

public Entry      removeMin() throws
    EmptyPQException {
    if (isEmpty) throw new
        EmptyPQException ("Empty PQ");
    Entry temp = V[1];
    //now make last node the root, if there is a last
    if PQSize > 1
        V[1] = V[PQSize];
    PQSize--;
    //now bubble down
    downHeap();
    return temp;
}    //an Entry type is returned
```

```

protected void downHeap( ) {
    int currP ; //current parent rank
    currP = 1; //start with root as current parent
    while (2*currP <= PQsize) { //has children

        //First decide which is smaller child
        int s = 2*currP; //rank of smaller child
        if ((s + 1) <= PQSize) {//has right child
            if c.(compare (V[s].key(), V[s +1].key())
                                >=0)
                s++; } //right child is smallest

        //Now see if a swap is needed
        if (c.compare (V[s].key(), V[currP].key())
                < 0 {//swap needed
            Entry ctemp = V[s];
            V[s] = V[currP];
            V[currP] = ctemp; //swap complete
            currP = s; }//ready for next iteration
        else break; }
    }
}

```

```

public Entry    insert(Object kk, Object vv) throws
                InvalidKeyException {
    checkKey (kk); //can throw exception
    if ((PQSize +1) == capacity)
        doubleVectorCapacity();
    V[PQSize + 1] = new Entry (kk, vv);
    PQSize++;
    //now bubble up
    upHeap ();
    return V[PQSize]; }

protected void upHeap( ) {
    int currCh = PQSize; //current child
    int currP = currCh/2; //current parent

    while ((currP!= 0) { //while child has parent
        if (c.compare
            (V[currP].key(), V[currCh].key()))
            <=0) //no swap needed
                break;
        //otherwise swap parent/child nodes;
        Entry ctemp = V[currCh];
        V[currCh] = V[currP];
        V[currP] = ctemp; //swap complete
        currCh = currP; }//ready for next iteration
    }
}

```

```
public void      doubleVectorCapacity() {
    int tempcap = capacity;
    capacity = 2 * capacity;
    Entry[] B = new Entry [capacity];
    for (int j = 0; j < tempcap; j++)
        B[j] = V[j];
    V = B;      }
```

/ Note: This need to double the container capacity every so often will not be a part of a heap-order binary-tree implementation based on linked nodes, as in your Assignment 4. */*

/ We still need to specify inner classes for
DefaultComparator and Entry. */*

```
protected static class MyEntry implements Entry {  
    protected Object    k; //key component  
    protected Object    v; //value component  
//constructor  
    public              MyEntry (Object key, Object value){  
                            k = key; v = value; }  
  
//methods  
    public Object        key() { return k;}  
    public Object        value() { return v;} }
```

```
protected static class DefaultComparator  
    implements Comparator {  
    //constructor  
    public          DefaultComparator(){/*does nothing*/ }  
//only method  
    public int      compare (Object a, Object b)  
                    throws ClassCastException {  
        int j;  
        j = ((Comparable) a).compareTo (b);  
/* Cast Comparable converts a from Object  
to type Comparable, enabling compareTo() */  
        return j; }  
}
```

Textbook Implementation of a complete heap-ordered binary-tree priority queue

The textbook has a complex (in the sense of taking a lot of space to describe), and a multi-layered, implementation.

The textbook class is **HeapPriorityQueue**, on page 333, and this class has the specification of the five priority queue methods *size()*, *isEmpty()*, *min()*, *removeMin()* and *insert()*. Notice that on page 333-335, the logic of these methods is close to that given above for the same methods with the class **VectorCHBTPriorityQueue()**. *[Please study.]*

This class rests on an existing class **VectorBinaryTree**, from page 325, and this class specifies all the standard Binary Tree methods, such as *hasRight()*, *hasLeft()*, *left()*, *right()*, and so on, plus a few more, namely *add()* to the end of the heap, and *remove()* from the end of the heap.

VectorBinaryTree also uses a further inner class, called **VectorNode**, to enable an object (which will be an **Entry** class), combined with a *rank* value, to be entered into a binary tree node as an *element*.

VectorBinaryTree in turns rests on an existing **Vector** class, from page 206, which has the standard vector methods *insertAtRank()*, *removeAtRank()*, etc.

We will not go into the details of this implementation in class (no pun). It is straightforward, except for the long-windedness. *[Please study yourselves.]*