Priority Queues

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

04/12/2010
6.1 Model
A priority queue ADT is a data structure that includes at least the operations insert and deleteMin, which returns and removes the minimum element.

These are the equivalent of enqueue and dequeue for a queue. An example application of a priority queue is job scheduling, which is usually prioritized rather than first come first serve.

6.2 Simple Implementations
- **Linked list**: insertions are $O(1)$; removals are $O(N)$.
- **Sorted list**: insertions are $O(N)$; removals are $O(1)$. Since there are at least as many insertions as deletions, it is better to minimize the cost of insertion.
- **Binary Search Tree**: both insertions and removals are $O(\log N)$ on average if insertions are random.
- **Balanced BST**: both insertions and removals are $O(\log N)$ in the worst case.
Binary search trees are overkill because they support operations that are usually not required. We can get the $O(\log N)$ time without the overhead of links by using a binary heap.

### 6.3.1 Structure Property

A heap (binary heap) is a *complete binary tree* (one that is completely filled except at the bottom level which is filled from left to right). Height $h$ implies $2^h \leq N \leq 2^{h+1} - 1$ and $h = \lceil \log N \rceil$.

The regularity of a complete binary tree allows it to be represented by an array requiring no links. Array elements are ordered left-to-right by level starting with position 1 so that the element in position $i$ has left child in position $2i$, right child in position $2i + 1$, and parent in position $\lfloor i/2 \rfloor$. 
Figure 6.2 A complete binary tree

Figure 6.3 Array implementation of complete binary tree
Figure 6.4

```
1 template <typename Comparable>
2 class BinaryHeap
3 {
4     public:
5         explicit BinaryHeap( int capacity = 100 );
6         explicit BinaryHeap( const vector<Comparable> & items );
7
8         bool isEmpty( ) const;
9         const Comparable & findMin( ) const;
10
11         void insert( const Comparable & x );
12         void deleteMin( );
13         void deleteMin( Comparable & minItem );
14         void makeEmpty( );
15
16     private:
17         int currentSize; // Number of elements in heap
18         vector<Comparable> array; // The heap array
19
20         void buildHeap( );
21         void percolateDown( int hole );
22     };
```
6.3.2 Heap-Order Property
The key value in every node is less than or equal to those of its children (and hence its descendants). Thus, the minimum element is at the root, and every subtree is also a heap. For every node $X$ other than the root, the parent of $X$ has key less than or equal to that of $X$.

The above property defines a min-heap. We can define a max-heap analogously.

**Figure 6.5** Two complete trees (only the left tree is a heap)
To insert $x$, create a hole in the next available location, and then iterate on moving the hole up toward the root by moving the parent’s element into the hole if it is larger than $x$. The new element is *percolated up* the heap until the correct location is found.

An example is depicted in Figures 6.6 and 6.7, and the code is in Figure 6.8. Note that an implementation using swaps (three assignment statements) would require $3d$ assignments instead of $d + 1$ assignments for $d$ levels.

We could eliminate the test for $\text{hole} > 1$ in the for loop by storing $x$ into $\text{array}[0]$ before the loop.

The worst-case insert time is $O(\log N)$ but it has been shown that the average insertion requires 2.607 comparisons and moves the element 1.607 levels up. (More than half of the elements are in the bottom two levels.)
Figures 6.6-6.7

**Figure 6.6** Attempt to insert 14: creating the hole, and bubbling the hole up

**Figure 6.7** The remaining two steps to insert 14 in previous heap
Procedure to insert into a binary heap

```c
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size() - 1 )
        array.resize( array.size() * 2 );

    // Percolate up
    int hole = ++currentSize;
    for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```

Figure 6.8 Procedure to insert into a binary heap
Finding the minimum element is easy but deleting it is harder. It leaves a hole at the root which is filtered down (percolated down) by moving the smaller of the child elements into the hole until the hole can be filled by the element in the last position. The hole is expected to percolate to the bottom level, resulting in an average cost of $O(\log N)$.

**Figure 6.9** Creation of the hole at the root
Figures 6.10-6.11

Figure 6.10 Next two steps in deleteMin

Figure 6.11 Last two steps in deleteMin
/**
 * Remove the minimum item.
 * Throws UnderflowException if empty.
 */
void deleteMin()
{
    if( isEmpty() )
        throw UnderflowException();
    array[1] = array[ currentSize-- ];
    percolateDown(1);
}

/**
 * Remove the minimum item and place it in minItem.
 * Throws UnderflowException if empty.
 */
void deleteMin( Comparable & minItem )
{
    if( isEmpty() )
        throw UnderflowException();
    minItem = array[1];
    array[1] = array[ currentSize-- ];
    percolateDown(1);
}

/**
 * Internal method to percolate down in the heap.
 * hole is the index at which the percolate begins.
 */
void percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];
    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child <= currentSize && array[ child + 1 ] < array[ child ] )
            child++;
        if( array[ child ] < tmp )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}

**Figure 6.12** Method to perform deleteMin in a binary heap
A min-heap is of no help in finding the maximum element. The maximum element is at a leaf but there are \( \lceil N/2 \rceil \) leaves. If it is necessary to efficiently find the position (array index) associated with a key value, then another data structure, such as a hash table, is used. In that case, the following three operations have logarithmic worst-case times.

- **decreaseKey(p, \( \Delta \))**: decreases the value at position \( p \) by \( \Delta > 0 \). This requires a percolate up for the increased priority.

- **increaseKey(p, \( \Delta \))**: increases the value at position \( p \) by \( \Delta > 0 \). This uses percolate down to decrease the priority.

- **remove(p)**: removes the node at position \( p \) from the heap. This uses decreaseKey(\( p, \infty \)) followed by deleteMin(). It allows user termination of a process in a priority queue.
Figure 6.13 A very large complete binary tree
Like `percolateDown`, `buildHeap` is a private internal method. It is called by a constructor to build a heap from an arbitrarily-ordered sequence of `Comparable` items. Using \(N\) calls to `insert` would result in \(O(N)\) average and \(O(N \log N)\) worst-case run time. We can guarantee linear worst-case time by storing the elements in arbitrary order, and then calling `percolateDown(i)` for \(i = n/2, n/2 - 1, \ldots, 2, 1\), where \(n = \text{currentSize}\).

The code is shown in Figure 6.14, and Figures 6.15-6.18 depict an example with \(n = 15\). Note that, for \(i > n/2\), node \(i\) has no children and therefore need not be processed. Also, reversing the order would not work, as demonstrated by the sequence 2, 3, 4, 1. The following theorem can be proved by finite induction.

**Theorem** In function `buildHeap` the subtree anchored at \(i\) has the heap-order property after the call to `percolateDown(i)`.
Figure 6.14

```cpp
explicit BinaryHeap( const vector<Comparable> & items )
    : array( items.size() + 10 ), currentSize( items.size() )
{
    for( int i = 0; i < items.size(); i++ )
        array[ i + 1 ] = items[ i ];
    buildHeap();
}

/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
void buildHeap()
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

Figure 6.14 buildHeap and constructor
Figure 6.15 Left: initial heap; right: after percolateDown(7)

Figure 6.16 Left: after percolateDown(6); right: after percolateDown(5)
Figure 6.17 Left: after `percolateDown(4)`; right: after `percolateDown(3)`

Figure 6.18 Left: after `percolateDown(2)`; right: after `percolateDown(1)`
Theorem 6.1: In a perfect binary tree of height $h$ containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$. The cost of buildHeap is therefore $O(N)$.

proof: The number of nodes at height $h - i$ is $2^i$ for $i = 0$ to $h$. Denote by $s$ the sum of heights of all nodes:

$$s = \sum_{i=0}^{h} 2^i(h-i) = h + 2(h-1) + 4(h-2) + \ldots + 2^{h-1}(1) \Rightarrow$$

$$2s = 2h + 4(h-1) + \ldots + 2^{h-1}(2) + 2^h(1).$$

Subtracting, we obtain

$$s = -h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h$$

$$= -h + \sum_{i=1}^{h} 2^i = -(h+1) + \sum_{i=0}^{h} 2^i$$

$$= -(h + 1) + 2^{h+1} - 1. \square$$