Priority queues and heaps

Professors Clark F. Olson and Carol Zander

Priority queues

A common abstract data type (ADT) in computer science is the **priority queue**. As you might expect from the name, each item in the priority queue has a priority that determines the order in which the items leave the queue (not first-in first-out, as in the queues we have seen previously). There are many applications of priority queues. Scheduling print jobs and system processes are two simple examples. This is also the abstract data type that we need to implement Huffman coding. The primary operations that are necessary for a priority queue are:

insert (with priority)
findMin (or findMax)
deleteMin (or deleteMax)

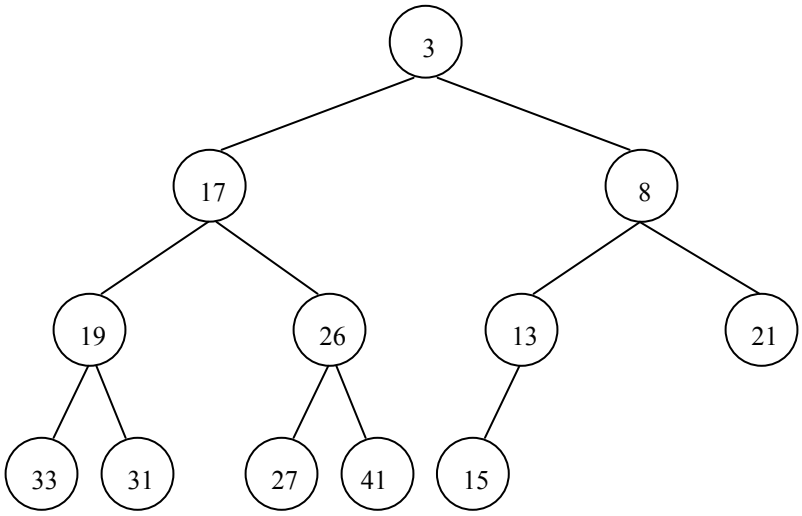
Of course, we want to be able to implement these three operations efficiently in terms of *n* (the number of items in the queue). This could be implemented in many ways, including several that we have already seen.

Data structure	insert	deleteMin
Unsorted collection	O(1)	O(n)
Sorted array	O(n)	O(1)
Sorted linked list	O(n)	O(1)
Binary search tree (balanced)	O(log n)	O(log n)

Only the balanced binary search tree is efficient (in the worst-case) unless the number of insertions is very small compared to the number of deleteMin operations. There are two types of data structure that can guarantee O(log n) time for insert and deleteMin in the worst-case. One is the balanced binary search tree; the other (simpler) data structure is a ***heap***. (Note that this is not related to the usage of heap to describe free memory.)

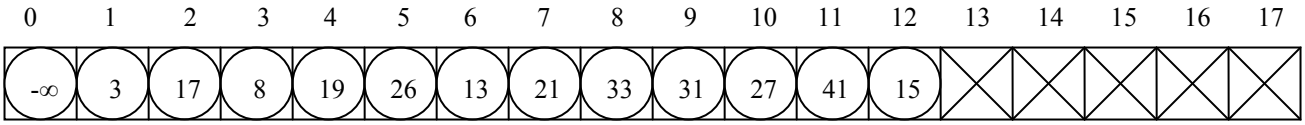
Heaps

Binary heaps are complete binary trees that satisfy the **heap-order property**, which states that each node (except the root) must have a key (or priority) that is no less than the key of its parent. (This is a **minheap**. A **maxheap** would be the reverse.) Complete binary tree means that all levels are completely full except possibly the last level, which is filled from left to right. With this definition, each subtree within the heap is also a heap and the minimum key must be at the root of the tree. Here is an example heap:



Note that there is no restriction on the ordering of the children of a node; no tree traversal will output the items in sorted order. We can also have smaller keys at lower levels of the tree than larger keys, i.e., there is no relationship between left and right subtrees other than the parent is smaller.

Usually, these are implemented with an array, since they are *always* complete trees.



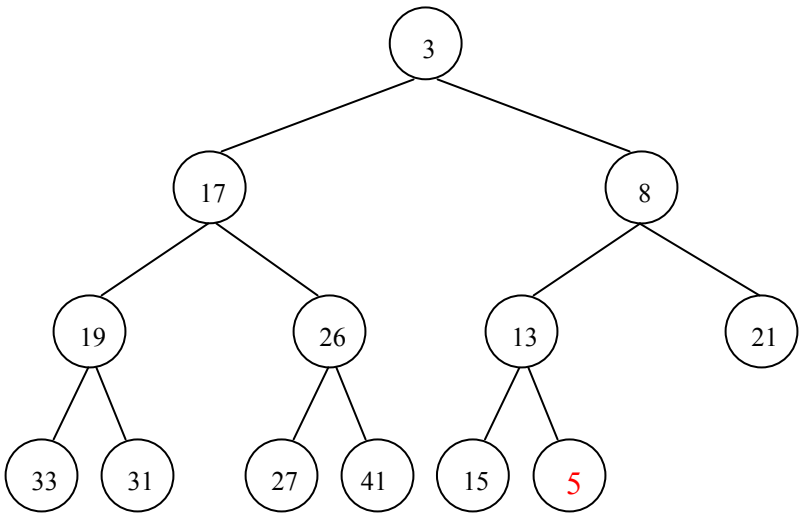
Note that the first node is often initialized to a value known to be smaller than all values in the heap in order to simplify the insertion operation. The array may be larger than the heap to allow insertion or because items have been deleted from the heap. When implemented, a heap keeps track of both how many items are in the heap and how much space has been allocated to the array. Even better is to use a vector, which keeps track of the allocation for us, so the declaration might look like this.

```
template <typename Comparable>
class Heap {
public:
      // Constructor, copy constructor, destructor, as usual…
      void insert(Comparable *c);
      const Comparable * findMin() const;
      bool deleteMin();
private:
      int size;
      vector<Comparable *> items;
};
```

We can perform a findMin operation in O(1) time on a minheap, since the minimum value is always at the root (*items[1]).

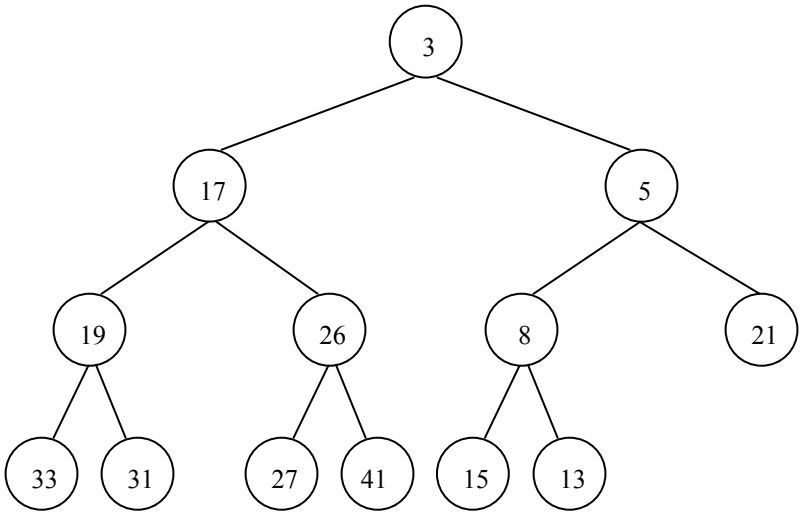
Heap insertion

To insert in a heap, you place the value in the next available spot and then "**percolate**" it upwards until it is no longer smaller than its parent is. Let's insert a 5 into our heap above. When we start the structure looks like this:



Often the item is not actually in the end array position yet. This way as we percolate the value up, it is one assignment (one operation) instead of one swap (three operations).

This isn't a heap, since the 5 is smaller than its parent. We "swap" it with the 13 to fix this. Since it still isn't smaller than its parent is, we then "swap" it with the 8 to get the following heap:



Try inserting 22, 2, and 19 (in that order) to get a new heap.

In practice, the insertion works on the array like this:

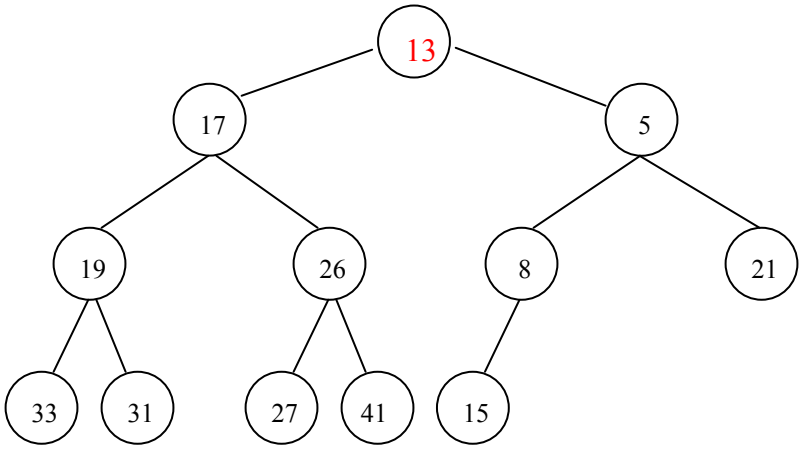
```
template <typename Comparable>
void Heap<Comparable>::insert(Comparable *c) {
   // add item in position 0 (dummy position) to prevent percolating up from root
   items[0] = c;

   // Ensure we have enough space
   size++;
   if (items.size() == size)
      items.push_back(NULL);

   // Percolate up
   int position = size;
   while (*items[0] < *items[position / 2]) {
      items[position] = items[position / 2];
      position = position / 2;
   }
   items[position] = items[0];
}
```

The code above uses a couple of optimizations. One is not to insert the item until the end. In this case, we just move the necessary elements until we find the final resting spot for the new item. Also, the value at position zero is initialized to the new item in this case. This is also a useful technique when using templates, since we can't create a dummy object for index 0 that we know has a key less than every object that will be placed in the heap. (Note that vector transparently increases the size of the internal dynamic array by more than one at a time.)

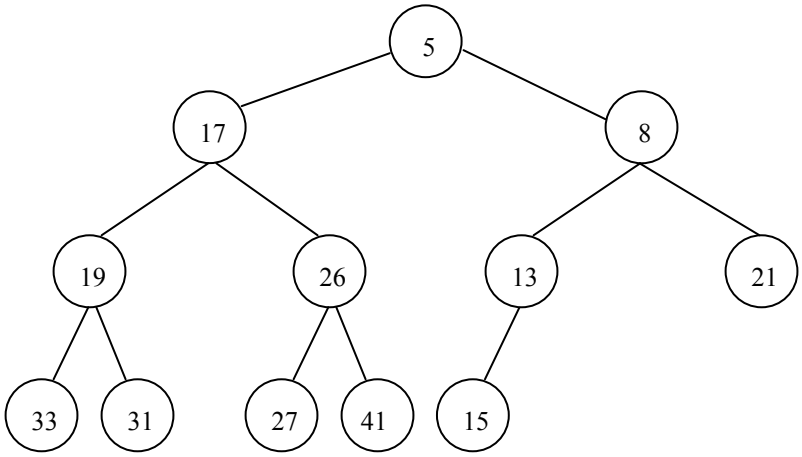
This operation is **O(log n) in the worst case,** but **O(1) in the average case** (assuming that the numbers are inserted in random order). The average number of levels that a new item needs to be percolated is only 1.6. Arguably, this is O(n) in the worst case, if we need to expand the size of the vector or array, but clever tricks can be used to insure that we do no more than O(log n), if necessary.

The deleteMin operation

When we want to do a deleteMin, we perform a similar operation in reverse. So that there are no holes in the array, we move the last element to the root and then percolate it down. Here is the state when we move the last element to the root:



Now we need to "reheapify" the tree by percolating the 13 down. We always replace it with the smaller of the two children (unless they are both larger than it is) and, then, repeat the process, yielding:



The code looks like:

```
template <typename Comparable>
bool Heap<Comparable>::deleteMin() {
      if (size == 0) return false;

      delete items[1];
      items[1] = items[size];
      size--;

      percolateDown(1);
      return true;
}

template <typename Comparable>
void Heap<Comparable>::percolateDown(int position) {
      int child = position * 2;
      if (child > size) return;
      if (child != size && *items[child] > *items[child + 1]) child++;
      if (*items[child] < *items[position]) {
            swap(items[child], items[position]);
            percolateDown(child);
      }
}
```

This operation requires O(log n) time in the average case and in the worst case, since the last item is usually relatively large and often needs to be percolated down almost to the bottom.

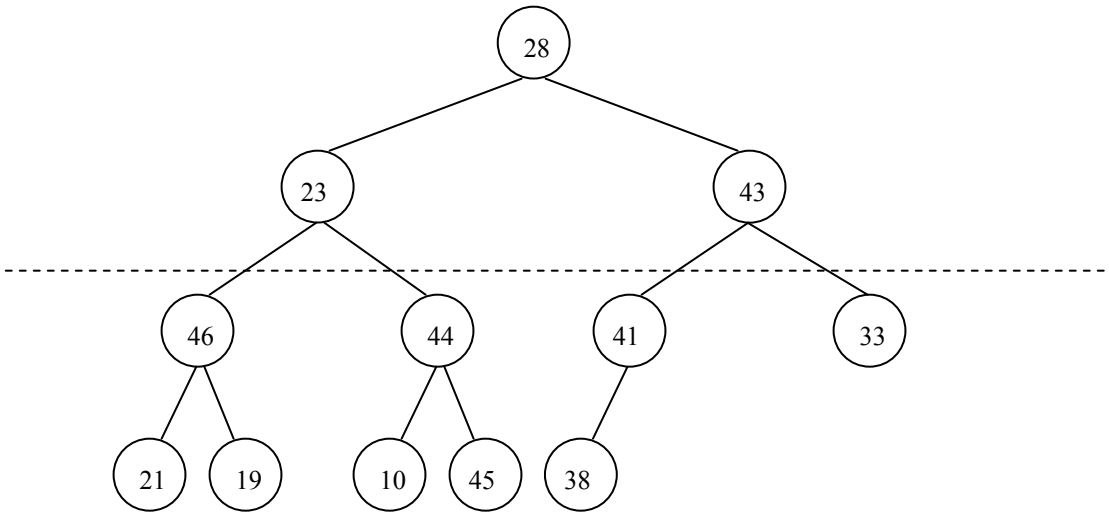
Linear time heap building

We can build a heap in linear expected time by using insert repeatedly, but this has O(n log n) worst-case efficiency. It turns out that we can build a heap in linear time in the worst-case if we have all of the elements when we start. The basic idea is straightforward. We call percolateDown for each non-leaf node starting with the last node and ending with the root.

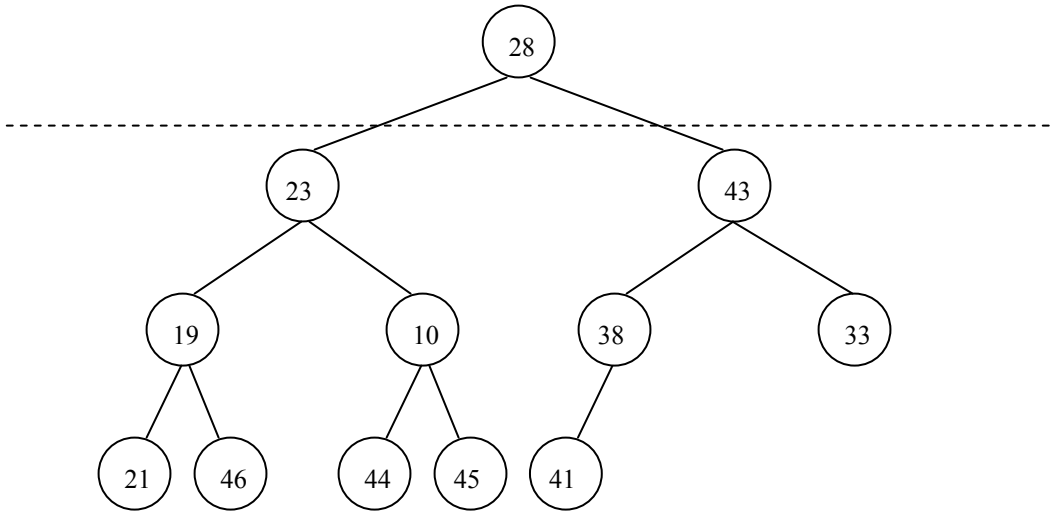
```
// the routine assumes that the objects and size have already
// been loaded into the heap data
template <typename Comparable>
void Heap<Comparable>::heapify() {
      for (int i = size / 2; i > 0; i--)
            percolateDown(i);
}
```

How do we know that size/2 is the right place to start? We can't have more than size/2 non-leaf nodes. Each non-leaf node has two children (with one possible exception – the last non-leaf node). If there are m non-leaves, then there are either 2m-1 or 2m children. Since m-1 of these children are other non-leaves (one non-leaf is the root), that leaves m or m+1 leaf nodes and there must be at least as many leaves as there are non-leaves.

Does this result in a heap when we're done? Let's try an example:

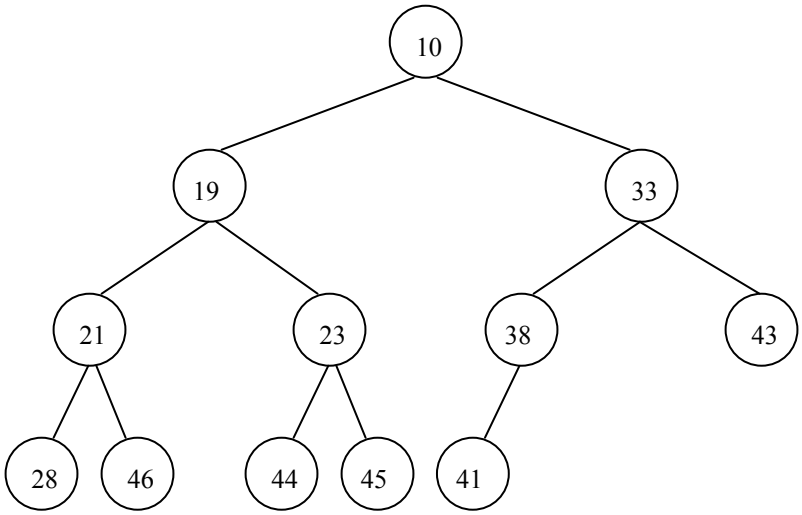


The 41 percolates down one spot, swapping with the 38.
The 44 swaps with the 10.
The 46 swaps with the 19.
Essentially we turn the bottom level of trees into heaps. This yields:



At the next level, the 43 swaps with the 33.
The 23 swaps with the 10. Now these subtrees are heaps.

Finally, at the root, the 28 swaps with the 10, and then the 19, and then the 21, yielding:



Two questions: Do we know that this always results in a heap? Does this really take linear time in the worst case?

For the first question: if we start with two heaps, join them together with any root and, then, percolate the root down, we end up with a heap. Induction using this idea proves that we get a heap in the end with **heapify**.

For the second question: what is the worst that can happen? At every non-leaf node, we could have to percolate it down to a leaf. This means that the maximum number of swaps is the same as the sum of the heights of each of the nodes in the tree. Proving the following theorem, proves heapify is O(N) where $N = 2^{(h+1)} - 1$. N is the number of nodes in a complete binary tree. (Assume h=0 for a tree with just a root.)

Theorem: For a complete binary tree of height h containing $2^{(h+1)} - 1$ nodes, the sum of the heights is $2^{(h+1)} -1 -(h+1)$.

(Note that since $N = 2^{(h+1)} - 1$, $2^{(h+1)} -1 -(h+1)$ is O(N) .)

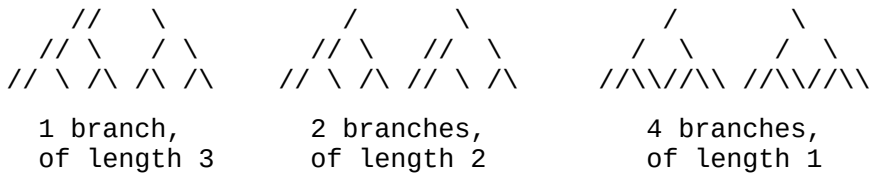
Proof: Sum the height of the tree. Here we use the definition that says an empty tree has height = -1; a tree with just a root is height zero.

For example, when h is 3, the sum of the heights is:

$1(3) + 2(2) + 4(1) + 8(0)$

Now generalize and write as a formula:

$$\begin{aligned} & 1(3) + 2(2) + 4(1) + 8(0) \\ = & \ 2^0(3-0) + 2^1(3-1) + 2^2(3-2) + 2^3(3-3) \\ = & \ 2^0(h-0) + 2^1(h-1) + 2^2(h-2) + 2^3(h-3) \end{aligned}$$



The double lines show the height count.

So the sum, S, is as follows:

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

We can use a neat trick and compute $S = 2S - S$ as follows:

$$\begin{aligned} 2S = & \quad 2^1(h) + 2^2(h-1) + 2^3(h-2) + \ldots + 2^{(h-2)}(3) + 2^{(h-1)}(2) + 2^h \\ -S = & -2^0(h) - 2^1(h-1) - 2^2(h-2) - 2^3(h-3) - \ldots - 2^{(h-2)}(2) - 2^{(h-1)}(1) - 0 \\ \hline S = & -h + 2^1 + 2^2 * 2 + 2^3 * 3 + \ldots + 2^{(h-2)}(3-2) + 2^{(2-1)} + 2^h \\ & \quad - 2^2 * 1 - 2^3 * 2 \\ & \quad = 2^2 \quad = 2^3 \quad \ldots \end{aligned}$$

Subtracting term by term, the terms with 'h' drop out and a power of 2 is left. Then subtract and add one, so our summation starts at zero:

$$2S - S = S = -h + 2 + 2^2 + 2^3 + \ldots + 2^h = -h + \sum_{i=1}^{h} 2^i = -h - 1 + \sum_{i=0}^{h} 2^i$$

Now, using a well-known formula:

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

proves the theorem:

$$S = -h - 1 + 2^{h+1} - 1 = O(N)$$

Heap sort

A final note that's worth mentioning is that we can easily sort using a heap, by building the heap and repeatedly deleting the minimum value. We use the delete operation, but store the "deleted" value at the end of the array. Interesting, this yields an algorithm that is O(n log n) in the worst case (unlike quicksort/hoaresort) and that doesn't require extra memory (unlike mergesort). Quicksort is still used most often in practice.