Heaps and Priority Queues

- The priority queue ADT
- Priority queue implementations
- The heap data structure

The Priority Queue ADT

A priority queue is similar to an ordinary queue, but does not enforce a FIFO policy.

Elements are dequeued from a priority queue by order of priority, instead of order of arrival.

If there are two items of equal priority in the queue, which is dequeued first is not defined. (Many priority queues are implemented in such a way that in the case of equal priority, items are dequeued in FIFO order.)

You could define an ordinary queue in terms of a priority queue where the priority of an item depends upon when it was entered in the queue, but that would be making extra work for yourself.
Priority Queue ADT operations:

- **INSERT (VALUE, PRIORITY)**
  
  also known as ENQUEUE

- **VALUE = DELETEMIN ()**
  
  also known as DEQUEUE

- Add an item with the given value and priority to the queue.

- Remove the item with the minimum priority from the queue, and return its value.

- Note that the item with the minimum priority is the next to be dequeued; this is somewhat counterintuitive, but standard notation. The definition can be (and often is) restated to use the maximum, if that convention is more convenient.

We can define a number of other ADT operations, such as IsEmpty, which might be quite useful, but for now we’ll focus on INSERT and DELETEMIN.

Implementing Priority Queues

We’ve seen many data structures that can be used to implement the priority queue ADT. Let’s examine linked lists and binary search trees for their suitability:
Linked Lists

• If we keep the list sorted:

**INSERT** - find the correct location in the list, and insert the new item there. **INSERT** is responsible for keeping the list in order by priority.

**DELETEMIN** - pluck the first item out of the list.

**DELETEMIN** is O(1), if we can delete the first element in the list in time O(1). **INSERT** must scan the list in order to figure out where to place the new item. For a linked list, this is O(n).

• If we do not keep the list sorted:

**INSERT** - place the new item wherever is most convenient (probably the start of the list).

**DELETEMIN** - scan the list and find the item with the smallest priority, and remove it.

In this situation, **INSERT** can be O(1) but **DELETEMIN** must now do the work of scanning the list.

• If we keep the list sorted, there are tricks we can use to reduce the expected time necessary to find the correct place to insert a new element to O(log n) (by using some particular data structures to implement the list). However, if we do not keep the list sorted, then there is no nice way to quickly find the minimum.
Binary Search Trees

• Unbalanced BST

  INSERT - using ordinary Tree INSERT, find the correct location in the tree, based on priority, and insert the new item there. INSERT is responsible for keeping the list in order by priority.

  DeleteMin - find the leftmost node, delete it and return it.

    Both INSERT and DELETEMIN are O(log N) expected, O(N) worst case. Unfortunately, the worst case is probably typical - a bunch of items added in order by priority.

• Balanced BST

  INSERT and DELETEMIN as before, but keep the tree balanced. This ensures O(log N) for both INSERT and DELETEMIN, but adds significant overhead and complexity.
The HEAP Data Structure

- The HEAP is a data structure specifically designed to implement the PRIORITY QUEUE ADT cleanly and efficiently.

- Heaps come in several varieties (which you can explore in the reading), but the only kind we will do much with is the BINARY HEAP.

- The binary heap has the same basic structure as a binary tree, with the following additional constraints:
  - The tree is always complete.
    
    *All leaves are at depth \( d \) or \( d-1\), and all leaves at depth \( d-1\) are to the right of all leaves at depth \( d\).*

  - The key stored in any node is smaller than the keys stored in its children. (By induction, the key of the root of any subheap is the smallest key in the subheap.)

- Heap Examples:

```
1
1 2
3

1
7 10
2
```
Which of the following trees are also heaps? If not, why not?
The Heap Properties are Crucial

- The heap order property means that the root is *always* the element with the smallest key in the heap. This means that we can always find the minimum key in $O(1)$ time. (This is useful all by itself, but we need to show more in order to show how much it costs to delete this key.)

- The "complete" tree property totally and unambiguously defines the "shape" of the tree
  - We know the height of the tree, and that it is complete.
  - If we add another node to the tree, then the shape of the resulting tree is known (same for deletion) - for each number of nodes $N$, there is only one complete tree "shape".
Complete Trees as Arrays

The complete tree property allows us to use a very neat trick: a complete tree containing N nodes can be represented (implemented) by an array of length N + 1, in a very simple and efficient manner:

• The root is at index 1.

• For each position X in the array:
  • the parent is at index $X/2$ (unless it is the root, of course)
  • the left child is at index $2X$
  • the right child is at index $2X + 1$

Note: some authors and implementations use a zero-based scheme. In that case, the tree requires N array positions, the root is at index 0, parent at $(X-1)/2$, left child at $2X + 1$, and right child at $2X + 2$.

Either convention will work properly, as long as you are consistent. For today, we will use the 1-based scheme. In a future application, we will use a 0-based scheme. Use whichever is most convenient.
Squashing a Tree into an Array
Heaps as Arrays

Since heaps are complete trees, and complete trees can be represented by arrays, heaps can be represented by arrays. This is very convenient:

- No need to deal with pointers at all
  - No need to store the pointers: saves space
  - No need to manipulate the pointers
  - Implicit "parent pointers" (because we can always find the parent of a node, if that node has a parent).
- The arithmetic for finding the children (or the parent) is fast.

We will often draw binary heaps as if they were implemented in the same manner as the ordinary binary trees we have seen previously, because this illustrates the relationships between the nodes more clearly. However, when we implement a binary heap, we will always use an array representation.
Heap Insert

• To add a new key $K$ to a heap containing $N$ nodes. (Note that $N$ can be zero.)

1. Place $K$ in position $N+1$ (for one-based heaps)
   
   This is the only place that we can add a node to the heap such that the shape of the heap does not violate the *complete* property.

   However, placing $K$ here might violate the heap-order property. The next step takes care of this situation.

2. As long as $K$ is not the root and $K < K$’s parent, exchange $K$ and its parent.
   
   This is called bubble-up: we start with the key in a leaf, and let it rise like a bubble in the heap until the heap-order property is restored.

   This ensures that the heap-order property holds.
**INSERT Preserves the Heap Order**

Imagine that we have the following heap:

```
       X
      / \
     Y   Z
```

By the heap property, we know that \( X \leq Y \).

Therefore, if we add a new node \( Z \):

```
       X
      /  \
     Y   Z
```

then one of the following conditions must hold:

a) \( X \leq Z \) The result is a heap.

b) \( Z < X \) We can restore the heap property by exchanging \( X \) and \( Z \). Since \( Z < X \) and \( X \leq Y \), \( Z < Y \), so the result is a heap.
Heap DELETEMIN

To delete the "minimum" value of a heap (which is the root) that contains N nodes:

1. Save a copy of the value (the key) of the root node.

2. Move the contents of node N (for 1-based heaps, this is the “last” node in the heap) to the root of the tree, and delete node N.

   Deleting the last node in the heap (after copying its value to the root) preserves the heap shape property-- the tree is still complete, although it has 1 less node.

3. If the new root is less than or equal to its children, we have a heap. Otherwise, percolate this key down by exchanging it with its smallest child. Repeat this process for the new position of this key until the key no longer needs to be moved or the key is placed into a leaf.
Example: "Percolate Down"

Always switch with the smaller child in order to preserve the heap order.
Analysis of \textbf{INSERT} and \textbf{DELETEmin}

\textbf{Theorem:} INSERT and DELETEmin are $O(\log N)$

\textbf{Proof:}

Adding the new key to the end of the heap (\textbf{INSERT}) or removing the root node and moving the last element to the root’s position (\textbf{DELETEmin}) are $O(1)$.

The bulk of the time is spent percolating up (\textbf{INSERT}) or down (\textbf{DELETEmin}) - but we can’t exchange more than $h$ times, where $h$ is the height of the heap.

\textit{Since the heap is a complete tree}, $h = \Theta(\log N)$, so the number of exchanges (and nodes we look at) is bounded by $O(\log N)$. 
Building a Heap in O(N)

Given a collection of N keys, we can construct a new heap containing the N keys in time O(N log N) by repeatedly using INSERT to insert each key, one at a time.

However, we can also build a heap containing N keys in O(N) time.

• Algorithm:
  
  • Throw the keys into a complete tree of size N. (Do not consider the order at all.)
  
  • Starting at the lowest level and working your up the root, percolate each element down.
  
  • The total number of steps required to "percolated down" the key in any node is AT MOST proportional to the height of the node, because each step in the process either ends the process or moves the key closer to a leaf.
  
  • So if we know the "total height" of a heap (the sum of the heights of every node) then we will have a bound on the amount of work heapify can perform.
Showing that Heapify is O(N)

- Let $h$ be the height of the root. Then we have

  1 node at height $h$,
  2 nodes at height $h-1$
  4 nodes at height $h-2$
  \[ 2^h \text{ nodes at height 0 (the leaf nodes)} \]

This is just the sum:

\[
\sum_{i=0}^{h-1} 2^i (h - i)
\]
Heapify is O(N) (cont’d)

\[
\sum_{i=0}^{h} 2^i(h-i) = \sum_{i=0}^{h} 2^i h - \sum_{i=0}^{h} 2^i i
\]

We’ll tackle each of these terms separately: let A be the first term of the subtraction, and B be the second term.

\[
A = \sum_{i=0}^{h} 2^i h = h \sum_{i=0}^{h} 2^i = h(1 + 2 + 4 + \ldots + 2^h)
\]

We’ve seen this series before:

A = h(2^{h+1} - 1) = h2^{h+1} - h

Now, let’s look at the B term

\[
B = \sum_{i=0}^{h} 2^i i = 0(1) + 1(2) + 2(4) + 3(8) + \ldots + h2^h
\]

\[
2B = 2 \sum_{i=0}^{h} 2^i i = 0 + (0)(2) + 1(4) + 2(8) + 3(16) + \ldots + (h-1)2^h + h2^{h+1}
\]
**Heapify is O(N) (cont’d)**

\[ B = 2B - B = - (1(2) + 1(4) + 1(8) + \ldots + 2^h) + h2^{h+1} \]

We’ve seen this series before. It looks like

\[ 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1 \]

but it’s missing the 1 in the front, so

\[ B = -(2^{h+1} - 2) + h2^{h+1} = h2^{h+1} - 2^{h+1} + 2 \]

Remember, we want \( A - B \), so we get

\[
A - B = [h2^{h+1} - h] - [h2^{h+1} - 2^{h+1} + 2] \\
= -h + 2^{h+1} - 2 = 2^{h+1} - h - 2
\]

Now we have a bound on the number of comparisons and exchanges required to heapify a perfect tree of height \( h \): \( O(2^{h+1}) \)

Recall that a perfect tree of height \( h \) contains \( 2^{h+1} - 1 \) nodes, so we can rewrite this in terms of \( N \):

\[
O(2^{h+1}) \quad O(N + 1) \quad O(N)
\]
Heapify is O(N) for complete trees

For complete (not perfect) trees, we can show that the same property holds. A complete tree of height $h$ has no more nodes than a perfect tree of the same height, and at least half as many nodes.

Therefore our bound (for a perfect tree) is perhaps twice as large as necessary - but it is certainly not smaller than necessary.