Heaps and Priority Queues

15.1 Priority Queue
The (min)-priority queue ADT supports:

- `insertItem(e)`: Insert new item e.
- `removeMin()`: Remove and return item with minimum key (Error if priority queue is empty).
- standard `isEmpty()` and `size`, maybe peeks.

Other possible methods include `decrease-key`, `increase-key`, and `delete`. Applications include selection, and the event queue in discrete-event simulation. There is also a version focusing on the maximum.

There are several inefficient implementations:

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>removeMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted linked list</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>sorted linked list or array</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>binary search tree</td>
<td>(O(n); \text{ average } O(\log n))</td>
<td></td>
</tr>
</tbody>
</table>

15.2 Heap
In *level numbering* in binary trees, the nodes are numbered such that:

for a node numbered \(x\), its children are \(2x+1\) and \(2x+2\)

Thus a node’s parent is at \((x-1)/2\) (rounded down), and the root is 0.

![Binary tree diagram]

One can store a binary tree in an array/vector by storing each value at the position given by level numbering. But this is wasteful storage, unless nearly balanced.

We can change the definition of *complete binary tree* as a binary tree where each level except the last is complete, and in the last level nodes are added left to right.

With this definition, a *min-heap* is a complete binary tree, normally stored as a vector, with values stored at nodes such that:
**heap-order** property: for each node, its value is smaller than or equal to its children's

So the minimum is on top. A heap is the standard implementation of a priority queue. Here is an example:

A **max-heap** can be defined similarly.

### 15.3 Min-Heap Operations

The idea for **insertion** is to *Add as last leaf, then bubble up value until heap-order property re-established*.

**Algorithm**: Insert(v)

1. add v as next leaf
2. while v < parent(v) {
   1. swapElements(v, parent(v))
   2. v = parent(v)
}

Use a “hole” to reduce data movement.

Here is an example of Insertion: inserting value 12:

The idea for **removeMin** is to *Replace with value from last leaf, delete last leaf, and bubble down value until heap-order property re-established*.
Algorithm: RemoveMin()

\[
\text{temp} = \text{value of root} \\
\text{swap root value with last leaf} \\
\text{delete last leaf} \\
v = \text{root} \\
\text{while } v > \text{any child}(v) \{ \\
    \text{swapElements}(v, \text{smaller child}(v)) \\
    v = \text{smaller child}(v) \\
\}\ \\
\text{return temp}
\]

Here is an example of RemoveMin:

\[
\begin{array}{c}
\text{7} \\
\text{24} \\
\text{25} \\
\text{29} \\
\text{29} \\
\text{19} \\
\text{40} \\
\text{40} \\
\text{68} \\
\text{58} \\
\text{56} \\
\text{31} \\
\end{array} \Rightarrow \begin{array}{c}
\text{19} \\
\text{24} \\
\text{25} \\
\text{29} \\
\text{29} \\
\text{68} \\
\text{58} \\
\text{56} \\
\text{40} \\
\text{68} \\
\text{31} \\
\end{array}
\]

Variations of heaps include

- \(d\)-heaps; each node has \(d\) children
- support of merge operation: leftist heaps, skew heaps, binomial queues

### 15.4 Heap Sort

Any priority queue can be used to sort:

- Insert all values into priority queue
- Repeatedly removeMin()

It is clear that inserting \(n\) values into a heap takes at most \(O(n \log n)\) time. Possibly surprising, is that we can create a heap in linear time. Here is one approach: work up the tree level by level, correcting as you go. That is, at each level, you push the value down until it is correct, swapping with the smaller child.

Analysis: Suppose the tree has depth \(k\) and \(n = 2^{k+1} - 1\) nodes. An item that starts at depth \(j\) percolates down at most \(k - j\) steps. So the total data movement is at most

\[
\sum_{j=0}^{k} 2^j(k - j),
\]

which is \(O(n)\), it turns out.
Thus we get **Heap-Sort**. Note that one can **re-use** the array/vector in which heap in stored: **removeMin** moves the minimum to end, and so repeated application produces sorted the list in the vector.

A Heap-Sort Example is:

1. \[
\begin{array}{ccccccc}
1 & 3 & 2 & 6 & 4 & 5 \\
\end{array}
\]
2. \[
\begin{array}{ccccccc}
2 & 3 & 5 & 6 & 4 & 1 \\
\end{array}
\]
3. \[
\begin{array}{ccccccc}
3 & 4 & 5 & 6 & 2 & 1 \\
\end{array}
\]
4. \[
\begin{array}{ccccccc}
4 & 6 & 5 & 3 & 2 & 1 \\
\end{array}
\]
5. \[
\begin{array}{ccccccc}
5 & 6 & 4 & 3 & 2 & 1 \\
\end{array}
\]
6. \[
\begin{array}{ccccccc}
6 & 5 & 4 & 3 & 2 & 1 \\
\end{array}
\]

### 15.5 Application: Huffman Coding

The standard binary encoding of a set of \(C\) characters takes \(\lceil \log_2 C \rceil\) bits for a character. In a variable-length code, the most frequent characters have the shortest representation. However, now we have to decode the encoded phrase: it is not clear where one character finishes and the next-one starts. In a **prefix-free code**, no code is the prefix of another code. This guarantees unambiguous decoding: indeed, the **greedy** decoding algorithm works:

\[
\text{traverse the string until the part you have covered so far is a valid code;}
\]
\[
\text{cut it off and continue.}
\]

Huffman’s algorithm constructs an optimal prefix-free code. The algorithm assumes we know the occurrence of each character:

Repeat

- merge two (of the) rarest characters into a mega-character
- whose occurrence is the combined
Until only one mega-character left
Assign mega-character the code EmptyString
Repeat

48
split a mega-character into its two parts assigning each of these 
the mega-character’s code with either 0 or 1

The information can be organized in a trie: this is a special type of tree in which 
the links are labeled and the leaf corresponds to the sequence of labels one follows to 
get there.

For example if 39 chars are A=13, B=4, C=6, D=5 and E=11, we get the coding 
A=10, B=000, C=01, D=001, E=11.

Note that a priority queue is used to keep track of the frequencies of the letters.

Sample Code

```
PriorityQ.h
Heap.h
Heap.cpp
```