Heaps, Priority Queues, Compression

- **Compression is a high-profile application**
  - .zip, .mp3, .jpg, .gif, .gz, ...
  - What property of MP3 was a significant factor in what made Napster work (why did Napster ultimately fail?)

- **What’s the difference between compression for .mp3 files and compression for .zip files? Between .gif and .jpg?**
  - What’s the source, what’s the destination?
  - What is lossy vs. lossless compression? Are the differences important?

- **Is it possible to compress (lossless compression rather than lossy) every file? Every file of a given size?**
  - What are repercussions?
Priority Queue

- Compression motivates the study of the ADT **priority queue**
  - Supports three basic operations
    - **insert** -- an element into the priority queue
    - **delete** -- the *minimal* element from the priority queue
    - **peek/getMin** -- find (don’t delete) minimal element
  - `getMin/delete` analagous: stack top/pop, queue enqueue/dequeue

- Simple sorting using priority queue (see pqdemo.cpp and simplepq.cpp), what is the complexity of this sorting method?

```cpp
string s; priority_queue pq;
while (cin >> s) pq.insert(s);
while (pq.size() > 0) {
    pq.deleteMin(s);
    cout << s << endl;
}
```
## Priority Queue implementations

- Implementing priority queues: average and worst case

<table>
<thead>
<tr>
<th></th>
<th>Insert average</th>
<th>Getmin (delete)</th>
<th>Insert worst</th>
<th>Getmin (delete)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted vector</strong></td>
<td>0(1)</td>
<td>0(n)</td>
<td>0(1)</td>
<td>0(n)</td>
</tr>
<tr>
<td><strong>Sorted vector</strong></td>
<td>0(n)</td>
<td>0(1)</td>
<td>0(n)</td>
<td>0(1)</td>
</tr>
<tr>
<td><strong>Search tree</strong></td>
<td>log n</td>
<td>log n</td>
<td>0(n)</td>
<td>0(n)</td>
</tr>
<tr>
<td><strong>Balanced tree</strong></td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>0(1)</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
</tbody>
</table>

- Heap has $O(1)$ find-min (no delete) and $O(n)$ build heap
Class `tpqueue<...>`, see `tpq.h`

- **Templated class like tstack, tqueue, tvector, tmap, ...**
  - If `deletemin` is supported, what properties must types put into `tpq` have, e.g., can we insert string? double? struct?
  - Can we change what minimal means (think about anaword and sorting)?
  - Implementation in `tpq.h`, `tpq.cpp` uses `heap`

- **If we use a compare function object for comparing entries we can make a min-heap act like a max-heap, see `pqdemo.cpp`**
  - Notice that `RevComp` inherits from `Comparer<Kind>`
  - Where is class `Comparer` declaration? How used?

- **STL standard C++ class `priority_queue`**
  - See `stlpq.cpp`, changing comparison requires template
void sort(tvector<string>& v)
// pre: v contains v.size() entries
// post: v is sorted
{
    tpqueue<string> pq;
    for(int k=0; k < v.size(); k++) pq.insert(v[k]);
    for(int k=0; k < v.size(); k++) pq.deletemin(v[k]);
}

• How does this work, regardless of tpqueue implementation?
• What is the complexity of this method?
  ➢ insert $O(1)$, deletemin $O(\log n)$? If insert $O(\log n)$?
  ➢ In practice heapsort uses the vector as the priority queue rather than separate pq.
  ➢ From a big-Oh perspective no difference: $O(n \log n)$
    • Is there a difference? What’s hidden with $O$ notation?
Priority Queue implementation

- The class `tpqueue` uses heaps, fast and reasonably simple
  - Why not use inheritance hierarchy as was used with `tmap`?
  - Trade-offs when using HMap and BSTMap:
    - Time, space
    - Ordering properties, e.g., what does BSTMap support?
- Changing method of comparison when calculating priority?
  - Create a function that replaces operator `<`
    - We want to pass the function, most general approach creates an object to hold the function
    - Also possible to pass function pointers, we avoid that
  - The function object replacing operator `<` must:
    - Compare two objects, so has two parameters
    - Returns -1, 0, +1 depending on <, ==, >
Creating Heaps

- **Heap** is an array-based implementation of a binary tree used for implementing priority queues, supports:
  - insert, findmin, deletemin: complexities?

- Using array minimizes storage (no explicit pointers), faster too --- children are located by index/position in array

- **Heap** is a binary tree with *shape* property, *heap/value* property
  - shape: tree filled at all levels (except perhaps last) and filled left-to-right (complete binary tree)
  - each node has value smaller than both children
Array-based heap

- Store “node values” in array beginning at index 1
- For node with index k
  - Left child: index $2k$
  - Right child: index $2k+1$

- Why is this conducive for maintaining heap shape?
- What about heap property?
- Is the heap a search tree?
- Where is minimal node?
- Where are nodes added? Deleted?
Thinking about heaps

- Where is minimal element?
  - Root, why?
- Where is maximal element?
  - Leaves, why?
- How many leaves are there in an N-node heap (big-Oh)?
  - O(n), but exact?
- What is complexity of find max in a min heap? Why?
  - O(n), but 1/2 N?
- Where is second smallest element? Why?
  - Near root?
Adding values to heap

- to maintain heap shape, must add new value in left-to-right order of last level
  - could violate heap property
  - move value “up” if too small

- change places with parent if heap property violated
  - stop when parent is smaller
  - stop when root is reached

- pull parent down, swapping isn’t necessary (optimization)
Adding values, details

```cpp
void pqueue::insert(int elt)
{
    // add elt to heap in myList
    myList.push_back(elt);
    int loc = myList.size();

    while (1 < loc &&
           elt < myList[loc/2])
    {
        myList[loc] = myList[loc/2];
        loc /= 2; // go to parent
    }
    // what's true here?
    myList[loc] = elt;
}
```

```c
tvector myList
```

```plaintext
<table>
<thead>
<tr>
<th>6 10 7 17 13 9 21 19 25</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10</td>
</tr>
</tbody>
</table>
```
Removing minimal element

- Where is minimal element?
  - If we remove it, what changes, shape/property?
- How can we maintain shape?
  - “last” element moves to root
  - What property is violated?
- After moving last element, subtrees of root are heaps, why?
  - Move root down (pull child up) does it matter where?
- When can we stop “re-heaping”?
  - Less than both children
  - Reach a leaf
Huffman codes and compression

- **Compression exploits redundancy**
  - Run-length encoding: 000111100101000
    - Coded as 3421113
    - Useful? Problems?
  - What about 1010101010101010101?

- **Encoding can be based on characters, chunks, ...**
  - Instead of using 8-bits for ‘A’, use 2-bits and 14 bits for ‘Z’
    - Why might this be advantageous?
  - Methods can exploit local information
    - abcabcabc is 3(abc) or is 111 111 111 for alphabet ‘abc’

- **Huffman coding is optimal** *per-character* coding method
Towards Compression

- Each ASCII character is represented by 8 bits, one byte
  - bit is a binary digit, byte is a binary term
  - compress text: use fewer bits for frequent characters (does this come free?)
- 256 character values, $2^8 = 256$, how many bits needed for 7 characters? for 38 characters? for 125 characters?

**go go gophers:** 8 different characters

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103 1100111 000</td>
</tr>
<tr>
<td>o</td>
<td>111 1101111 001</td>
</tr>
<tr>
<td>p</td>
<td>112 1110000 010</td>
</tr>
<tr>
<td>h</td>
<td>104 1101000 011</td>
</tr>
<tr>
<td>e</td>
<td>101 1100101 100</td>
</tr>
<tr>
<td>r</td>
<td>114 1110010 101</td>
</tr>
<tr>
<td>s</td>
<td>115 1110011 110</td>
</tr>
<tr>
<td>sp.</td>
<td>32 1000000 111</td>
</tr>
</tbody>
</table>

ASCII: $13 \times 8 = 104$ bits

3 bit code: $13 \times 3 = 39$ bits

compressed: ???
Huffman coding: *go go gophers*

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- **choose two smallest weights**
  - combine nodes + weights
  - Repeat
  - Priority queue?
- **Encoding uses tree:**
  - 0 left/1 right
  - How many bits?
Huffman coding: *go go gophers*

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- **Encoding uses tree:**
  - 0 left/1 right
  - How many bits? 37!!
  - Savings? Worth it?
Properties of Huffman code

- Prefix property, no code is prefix of another code
- Optimal per character compression

- Where do frequencies come from?

- Decode: need tree

```
1000111101001110100000110101111011110001
```

```
                          t
                      a
                    r  s  e
```

1000111101001110100000110101111011110001
Rodney Brooks

- *Flesh and Machines*: “We are machines, as are our spouses, our children, and our dogs... I believe myself and my children all to be mere machines. But this is not how I treat them. I treat them in a very special way, and I interact with them on an entirely different level. They have my unconditional love, the furthest one might be able to get from rational analysis.”

- Director of MIT AI Lab