Heaps, Priority Queues, Compression

- Compression is a high-profile application
  - .zip, .mp3, .jpg, .gif, .gz, ...
  - Why is compression important?

- 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?
  - Why does the difference make a difference?

- 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 two basic operations
    - insert — an element into the priority queue
    - delete — the minimal element from the priority queue
  - Implementations may allow getmin separate from delete
    - Analogous to top/pop, front/dequeue in stacks, queues

- Simple sorting using priority queue (see pqdemo.cpp and usepq.cpp)

```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 O(...)</th>
<th>Getmin O(...)</th>
<th>DeleteMin O(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted vector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted vector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked list (sorted?)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Balanced tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quick look at class tpq<...>

- 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)?

- 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>
  - How is Comparer accessed?

- How is this as a sorting method, suppose we store elements in vector?
Priority Queue implementation

- The class in tpq.h uses heaps, very 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

- Heap sort, for example, is always $O(n \log n)$, never $O(n^2)$ [worst case for quicksort], does this matter?
  - What’s done in practice?
  - What’s missing when using big-Oh expressions?

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 $2 \times k$
  - right child: index $2 \times k + 1$

  - 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?

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;
}
```

Huffman codes and compression

- Compression exploits redundancy
  - Run-length encoding: 00011100101000
    - 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
    - abacabc is 3(abc) or is 111 111 111 for alphabet 'abc'

- Huffman coding is optimal per-character coding method

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"?

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?

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>0101100</td>
</tr>
<tr>
<td>o</td>
<td>0110101</td>
</tr>
<tr>
<td>p</td>
<td>0111000</td>
</tr>
<tr>
<td>h</td>
<td>0101010</td>
</tr>
<tr>
<td>e</td>
<td>0110101</td>
</tr>
<tr>
<td>x</td>
<td>0111100</td>
</tr>
<tr>
<td>s</td>
<td>0111101</td>
</tr>
<tr>
<td>sp.</td>
<td>1010000</td>
</tr>
</tbody>
</table>

ASCII: 13 x 8 = 104 bits
3 bit code: 13 x 3 = 39 bits
compressed: ???
Huffman coding: go go gophers

- Choose two smallest weights
  - Combine nodes + weights
  - Repeat
  - Priority queue?
- Encoding uses tree:
  - 0 left/1 right
  - How many bits?

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

ASCII | 3 bits | Huffman
---|---|---
g | 103 | 1100111 | 000
o | 111 | 1101111 | 001
p | 112 | 1110000 | 010
h | 104 | 1101000 | 011
s | 114 | 1110010 | 100
r | 113 | 1110011 | 110
sp. 32 | 1000000 | 111

g 100111101001111010000110101111011110001

e 111011000000000011010000111011100011001