Data Compression

- Why do we care?
  - Secondary storage capacity doubles every year
  - However, disk space fills up quickly on every computer system
- More data to compress than ever before
- What’s the difference between compression for .mp3 files and compression for .zip files? Between .gif and .jpg?
- Must we exactly reconstruct the data?
  - Lossy methods
    - Generally fine for pictures, video, and audio (JPEG, MPEG, etc.)
  - Lossless methods
    - Run-length encoding
- Is it possible to compress (lossless compression prefer to than lossy) every file? Every file of a given size?

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/popt, 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();
    cout << s << endl;
}
```

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 Comparator<Kind>
  - Where is class Comparator declaration? How used?

- STL standard C++ class priority_queue
  - See stlpq.cpp, changing comparison requires template
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

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 <, ==, >

Array-based heap
- store "node values" in array beginning at index 1
- for node with index k
  - left child: index 2*k
  - right child: index 2*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?
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 finding max in a minheap? Why?
  - \( O(n) \), but _N?_
- Where is second smallest element? Why?
  - Near root?

```
CPS 100 9.9
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 finding max in a minheap? Why?O(n), but _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)

```
CPS 100 9.10
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

```
void pqueue::insert(int elt) {
    // add elt to heap in myList
    myList.push_back(elt);
    int loc = myList.size();
    while (loc > 0 &&
        elt < myList[loc/2]) {
        myList[loc] = myList[loc/2];
        loc /= 2; // go to parent
    }
    myList[loc] = elt;
}
```

```
CPS 100 9.11
Adding values, details
void pqueue::insert(int elt) {
    // add elt to heap in myList
    myList.push_back(elt);
    int loc = myList.size();
    while (loc > 0 &&
        elt < myList[loc/2]) {
        myList[loc] = myList[loc/2];
        loc /= 2; // go to parent
    }
    myList[loc] = elt;
}
```

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
Text Compression

- Input: String $S$
- Output: String $S'$
  - Shorter
  - $S$ can be reconstructed from $S'$

Huffman Coding

- D.A. Huffman in early 1950's
- Before compressing data, analyze the input stream
- Represent data using variable length codes
- Variable length codes though Prefix codes
  - Each letter is assigned a codeword
  - Codeword is for a given letter is produced by traversing the Huffman tree
  - Property: No codeword produced is the prefix of another
  - Letters appearing frequently have short codewords, while those that appear rarely have longer ones
- Huffman coding is optimal per-character coding method

Text Compression: Examples

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII</th>
<th>Fixed length</th>
<th>Var. length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>01100001</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>b</td>
<td>01100010</td>
<td>001</td>
<td>11</td>
</tr>
<tr>
<td>c</td>
<td>01100011</td>
<td>010</td>
<td>01</td>
</tr>
<tr>
<td>d</td>
<td>01100100</td>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>e</td>
<td>01100101</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

```
Input: String S
Output: String S'
```

```
S:  abc
   01100001 01100010 01100011
S': a b c
```

Building a tree

- Initial case: every character is a leaf/tree with the respective character counts → "the forest" of $n$ trees
  - $n$ is the size of your alphabet
- Base case: there is only one tree in the forest
- Reduction: Take the two trees with the smallest counts and combine them into a tree with count is equal to the sum of the two subtrees' counts → $n-1$ trees in our forest

```
Fixed: "0000101100100"
Var: "00001011111"
```
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

Encoding

1. Count occurrence of various characters in string \( O( ) \)
2. Build priority queue \( O( ) \)
3. Build Huffman tree \( O( ) \)
4. Write Huffman tree and coded data to file \( O( ) \)

Properties of Huffman coding

- Want to minimize weighted path length \( L(T) \) of tree \( T \)
- \( L(T) = \sum_{i \in \text{leaf}(T)} d_i w_i \)
  - \( w_i \) is the weight or count of each codeword \( i \)
  - \( d_i \) is the leaf corresponding to codeword \( i \)
- How do we calculate character (codeword) frequencies?
- Huffman coding creates pretty full bushy trees?
  - When would it produce a “bad” tree?
- How do we produce coded compressed data from input efficiently?
Decoding a message

0110000100001001101

Decoding

1. Read in tree data
   O ( )

2. Decode bit string with tree
   O ( )
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>001</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>010</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>011</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>011</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>101</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*

Encoding uses tree:
- 0 left/1 right
- How many bits? 37!!
- Savings? Worth it?

**Other methods**
- **Adaptive Huffman coding**
- **Lempel-Ziv algorithms**
  - Build the coding table on the fly while reading document
  - Coding table changes dynamically
  - Cool protocol between encoder and decoder so that everyone is always using the right coding scheme
  - Works darn well (*compress, gzip, etc.*)
- **More complicated methods**
  - Burrows-Wheeler (*bunzip2*)
  - PPM statistical methods
Questions

- How about ternary Huffman trees?
  - How would that affect the algorithm?
  - How about n-ary trees?
  - What would we gain?
- Are Huffman trees optimal?
  - What does that mean? (Hint: $L(T)$)
  - How can that be proven? (Hint: Induction will be your friend again)