Data 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?)

- Why do we care?
  - Secondary storage capacity doubles every year
  - Disk space fills up quickly on every computer system
  - More data to compress than ever before
More on Compression

- What’s the difference between compression techniques?
  - .mp3 files and .zip files?
  - .gif and .jpg?
  - Lossless and lossy
- Is it possible to compress (lossless) every file? Why?
- Lossy methods
  - Good for pictures, video, and audio (JPEG, MPEG, etc.)
- Lossless methods
  - Run-length encoding, Huffman, LZW, ...
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

- See `pqdemo.cpp` and `usepq.cpp`
  - code below sorts, complexity?

```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>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted vector</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>Search tree</strong></td>
<td>log n</td>
<td>log n</td>
<td>O(n)</td>
<td>O(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>O(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 \texttt{tpqueue<...>}, see \texttt{tpq.h}

- Templated class like \texttt{tstack, tqueue, tvector, tmap, ...}
  - If \texttt{deletemin} is supported, what properties must inserted objects have, e.g., can we insert \texttt{string}, \texttt{double}, \texttt{struct}?
  - Change what minimal means?
  - Implementation in \texttt{tpq.h, tpq.cpp} uses \texttt{heap}

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

- STL standard C++ class \texttt{priority_queue}
  - See \texttt{stlpq.cpp}, changing comparison requires template
Sorting with tapest्रypq.cpp, stlpq.cpp

```c++
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)?`
  - heapsort uses 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 minheap? Why?
  - O(n), but ½ 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

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;
}

vector myList
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 $S'$

CPS 100
Text Compression: Examples

Encodings
ASCII: 8 bits/character
Unicode: 16 bits/character

“abcde” in the different formats
ASCII:
011000010110001001100100110001101100100...
Fixed:
000001010011100
Var:
000110100110

<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>000</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>
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
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 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
Building a tree

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **How do we go from characters to codewords?**
  - Build a table as we build our tree
  - Keep links to leaf nodes and trace up the tree
- **Need way of writing bits out to file**
  - Platform dependent?
  - UNIX `read` and `write`
- **See bitops.h**
  - `obostream` and `ibstream`
  - Write bits from ints
- **How can differentiate between compressed files and random data from some file?**
  - Store a *magic* number
Decoding a message

$01100000100001001101$

CPS 100
Decoding a message

1100000100001001101
Decoding a message

100000100001001101
Decoding a message

00000100001001101

CPS 100
Decoding a message

0000100001001101

CPS 100

G
Decoding a message

000100001001101

CPS 100 13.44
Decoding a message

00100001001101
Decoding a message

0100001001101

CPS 100
Decoding a message

100001001101

CPS 100 13.47
Decoding a message

00001001101

CPS 100
Decoding a message

0001001101

GO
Decoding a message

001001101

CPS 100 13.50
Decoding a message

01001101

CPS 100

GO
Decoding a message

1001101

GO
Decoding a message

001101
Decoding a message

01101
Decoding a message

1101

GOO
Decoding a message
Decoding a message

01

GOO
Decoding a message

 GOOD
Decoding a message

0110000010000100001001101

GOOD
Decoding

1. Read in tree data \( O(\quad) \)

2. Decode bit string with tree \( O(\quad) \)
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?
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ⇔ "101011010010001010011100111000000000"
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ⇔ "10101101001000101001110011100000"

```
  69
   /\  \
  97 33
 /   \   /
21/\  16/\16
 |   /
10/\6
|   |  |
5/\6/\6
E/\N/\S
|/\|/\|/\|/|
O/\T/\G/\A
 C/\F/\D/\B
```
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ⇔ "10101101001000101001110011100000"

```
C
  |
  |
  O
  |
  |
  F
  |
  |
  P
  |
  |
  U

A
  |
  |
  B
```
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ⇔ "10101101001000101001110011100000"

![Huffman Tree Diagram]
Huffman Tree 2

- “A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
  - E.g. “A SIMPLE” ⇔ “10101101001000100101001110011100000”
Huffman Tree 2

- “A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
  - E.g. “ A SIMPLE” ⇔ “10101101001000101001110011100000”
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ⇔ "10101101001000101001110011100000"

```
  30
  / \  
22 (9) 23 (2) 12 (6) 11 (6)
  /   /   /   /
 5 (10)10 (11)16 (16)5 (6)
  /   /   /   /   /   /
E (6) 6 (6) 8 (8) 4 (4) 2 (4)
  /   /   /   /   /   /
O (3)N (6)S (6)M (4)G (2)
  /   /   /   /   /   /
C (3)F (3)T (1)D (2)R (2)
  /   /
P (1)U (2)
```
Huffman Tree 2

- “A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”
  
  E.g. “A SIMPLE” ⇔ “10101101001000101001110011100000”
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ⇔ "10101101001000101001110011100000"

---

"A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"

- E.g. "A SIMPLE" ⇔ "10101101001000101001110011100000"
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