Plan for the Course!

- Understand Huffman Coding
  - Data compression
  - Priority Queues
  - Bits and Bytes
  - Greedy Algorithms

- Algorithms + Data Structures = Programs
  - What does this mean and who said it?

- Graphs & the Oracle of Bacon

Scoreboard

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Array List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Table/Map</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What else might we want to do with a data structure?

Text Compression

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

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>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>

“abced” in the different formats

ASCII: 01100001011000100110001101100100...
Fixed: 000001010011100

Unicode: 16 or 32 bits/character
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 1100111 000 00</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>111 1101111 001 01</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>112 1110000 010 1100</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>104 1101000 011 1101</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>101 1100101 100 1110</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>114 1110010 101 1111</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>115 1110011 110 101</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>32 1000000 111 101</td>
<td></td>
</tr>
</tbody>
</table>

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

**Building a Huffman tree**

- Begin with a forest of single-node trees (leaves)
  - Each node/tree/leaf is weighted with character count
  - Node stores two values: character and count
  - There are $n$ nodes in forest, $n$ is size of alphabet?

- Repeat until there is only one node left: root of tree
  - Remove two minimally weighted trees from forest
  - Create new tree with minimal trees as children,
    - New tree root’s weight: sum of children (character ignored)

- Does this process terminate? How do we get minimal trees?
  - Remove minimal trees, need to order based on what?

**Priority Queue**

- **Stacks: Last-in First-Out**
  - java.util.Stack, java.util.Deque

- **Queues: First-in First-out**
  - java.util.LinkedList, java.util.Deque

- **Priority Queues: Highest-priority first-out**
  - java.util.PriorityQueue
  - Supports two basic operations
    - insert -- an element into the priority queue
    - delete -- the minimal element from the priority queue
  - Code below sorts. Complexity?

```java
public static void sort(ArrayList<String> a){
    PriorityQueue<String> pq = new PriorityQueue<String>();
    pq.addAll(a);
    for(int k=0; k < a.size(); k++)
        a.set(k, pq.remove());
}
```
Priority Queues

- Basic operations
  - Insert
  - Remove extremal
- What properties must the data have?
- Applications
  - Event-driven simulation: Colliding particles
  - AI: A* - Best-first search
  - Operating systems: Load balancing & scheduling
  - Statistics: Maintain largest \( m \) values
  - Graph searching: Dijkstra's algorithm
  - Data Compression: Huffman coding
  - Physics: Molecular dynamics simulation

PriorityQueue.java

- What about objects inserted into pq?
  - If deletemin is supported, what properties must inserted objects have, e.g., insert non-comparable?
  - Change what minimal means?
  - Implementation uses heap
- If we use a Comparator for comparing entries we can make a min-heap act like a max-heap, see PQDemo
  - Where is class Comparator declaration? How used?
  - What's a static inner class? A non-static inner class?
- In Java 5/6 there is a Queue interface and PriorityQueue class
  - The PriorityQueue class also uses a heap

Heap implements PriorityQueue

- 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 \( 2*k \)
  - 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 find max in a minheap? Why?
- 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 (pseudocode)

```java
void add(Object elt) {
    // add elt to heap in myList
    myList.add(elt);
    int loc = myList.size();

    while (1 < loc &&
        elt.compareTo(myList[loc/2]) < 0) {
        myList[loc] = myList[loc/2];
        loc = loc/2; // go to parent
    }
    // what's true here?
    myList.set(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
Priority Queue implementations

- Implementing priority queues: average and worst case

<table>
<thead>
<tr>
<th>Method</th>
<th>Insert</th>
<th>Getmin</th>
<th>Insert</th>
<th>Getmin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted vector</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted vector</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Balanced binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>search tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Heap has O(1) find-min (no delete) and O(n) build heap

How do we create Huffman Tree/Trie?

- Insert weighted values into priority queue
  - What are initial weights? Why?
  - Remove minimal nodes, weight by sums, re-insert
    - Total number of nodes?

```java
PriorityQueue<TreeNode> pq = new PriorityQueue<TreeNode>();
for (int k = 0; k < freq.length; k++) {
    pq.add(new TreeNode(k, freq[k], null, null));
}
while (pq.size() > 1) {
    TreeNode left = pq.remove();
    TreeNode right = pq.remove();
    pq.add(new TreeNode(0, left.weight + right.weight, left, right));
}
TreeNode root = pq.remove();
```

Building a tree

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

```
I 11 23 10 12 13 18 17 16 5 6 15 19 20 21 22 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
```

```
```

Building a tree

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

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Encoding

1. Count occurrence of all occurring character \( O(\quad) \)
2. Build priority queue \( O(\quad) \)
3. Build Huffman tree \( O(\quad) \)
4. Create Table of codes from tree \( O(\quad) \)
5. 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 encodings?
  - Build Huffman tree
  - Root-to-leaf path generates encoding
- Need way of writing bits out to file
  - Platform dependent?
  - Complicated to write bits and read in same ordering
- See BitInputStream and BitOutputStream classes
  - Depend on each other, bit ordering preserved
- How do we know bits come from compressed file?
  - Store a magic number

Creating compressed file

- Once we have new encodings, read every character
  - Write encoding, not the character, to compressed file
  - Why does this save bits?
  - What other information needed in compressed file?
- How do we uncompress?
  - How do we know foo.hf represents compressed file?
  - Is suffix sufficient? Alternatives?
- Why is Huffman coding a two-pass method?
  - Alternatives?
Uncompression with Huffman

- We need the trie to uncompress
  - 000100100010011001101111
- As we read a bit, what do we do?
  - Go left on 0, go right on 1
  - When do we stop? What to do?

- How do we get the trie?
  - How did we get it originally? Store 256 int/counts
    - How do we read counts?
  - How do we store a trie? Traversal order?
    - Reading a trie? Leaf indicator? Node values?

Decoding a message

Decoding

1. Read in tree data \( O( ) \)
2. Decode bit string with tree \( O( ) \)

Other Huffman Issues

- What do we need to decode?
  - How did we encode? How will we decode?
  - What information needed for decoding?

- Reading and writing bits: chunks and stopping
  - Can you write 3 bits? Why not? Why?
  - PSEUDO_EOF
  - BitInputStream and BitOutputStream: API

- What should happen when the file won't compress?
  - Silently compress bigger? Warn user? Alternatives?
Good CompSci 100(e) Assignment?

- Array of character/chunk counts, or is this a map?
  - Map character/chunk to count, why array?
- Priority Queue for generating tree/trie
  - Do we need a heap implementation? Why?
- Tree traversals for code generation, uncompression
  - One recursive, one not, why and which?
- Deal with bits and chunks rather than ints and chars
  - The good, the bad, the ugly
- Create a working compression program
  - How would we deploy it? Make it better?
- Benchmark for analysis
  - What’s a corpus?

Other methods

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

Data Compression

<table>
<thead>
<tr>
<th>Year</th>
<th>Scheme</th>
<th>Bit/Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>1967</td>
<td>ASCII</td>
<td>7.00</td>
</tr>
<tr>
<td>1950</td>
<td>Huffman</td>
<td>4.70</td>
</tr>
<tr>
<td>1977</td>
<td>Lempel-Ziv (LZ77)</td>
<td>3.94</td>
</tr>
<tr>
<td>1984</td>
<td>Lempel-Ziv-Welch (LZW) - Unix compress</td>
<td>3.32</td>
</tr>
<tr>
<td>1987</td>
<td>(LZH) used by zip and unzip</td>
<td>3.30</td>
</tr>
<tr>
<td>1987</td>
<td>Move-to-front</td>
<td>3.24</td>
</tr>
<tr>
<td>1987</td>
<td>gzip</td>
<td>2.71</td>
</tr>
<tr>
<td>1995</td>
<td>Burrows-Wheeler</td>
<td>2.29</td>
</tr>
<tr>
<td>1997</td>
<td>BOA (statistical data compression)</td>
<td>1.99</td>
</tr>
</tbody>
</table>

- Why is data compression important?
- How well can you compress files losslessly?
  - Is there a limit?
  - How to compare?
- How do you measure how much information?