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>Data Structure</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Vector/array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted vector/array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Maps</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>

“abcde” in the different formats

ASCII: 01100001011000100110001101100100...

Fixed: 0000001010011100

Unicode: 16 or 32 bits/character

Encodings

ASCII: 8 bits/character
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</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 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 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

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>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 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

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

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**
    - 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 E N S M A B O T G D L R U P F C
```

Building a tree

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

```
  F C
```

Building a tree

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

```
  L R
```

Building a tree

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

```
  2 2
```

Building a tree

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

```
  2 2
```
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”

Diagram of a tree with nodes labeled from 1 to 11, illustrating the encoding process.

Building a tree

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

Diagram showing the construction of a tree with nodes labeled from 1 to 11, demonstrating the encoding technique.
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 all occurring character $O( )$
2. Build priority queue $O( )$
3. Build Huffman tree $O( )$
4. Create Table of codes from tree $O( )$
5. Write Huffman tree and coded data to file $O( )$
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?
Decoding a message

110000100001001101

Decoding a message

10000100001001101

Decoding a message

00000100001001101

Decoding a message

0000100001001101
Decoding a message

000010001001101

Decoding a message

100001001101
Decoding a message

```
1001101
```

GO

Decoding a message

```
001101
```

GOO

Decoding a message

```
01101
```

GOO

Decoding a message

```
1101
```

GOO
Decoding a message

01

GOO

Decoding a message

01

GOO

Decoding a message

01100000100001001101

GOOD

Decoding a message

01100000100001001101

GOOD
Decoding

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

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</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?

Views of programming

- Writing code from the method/function view is pretty similar across languages
  - Organizing methods is different, organizing code is different, not all languages have classes,
  - Loops, arrays, arithmetic, ...

- Program using abstractions and high level concepts
  - Do we need to understand 32-bit two's-complement storage to understand x = x+1?
  - Do we need to understand how arrays map to contiguous memory to use ArrayLists?
  - Top-down vs. bottom-up?

From bit to byte to char to int to long

- Ultimately everything is stored as either a 0 or 1
  - Bit is binary digit, a byte is a binary term (8 bits)
  - We should be grateful we can deal with Strings rather than sequences of 0's and 1's.
  - We should be grateful we can deal with an int rather than the 32 bits that comprise an int

- If we have 255 values for R, G, B, how can we pack this into an int?
  - Why should we care, can’t we use one int per color?
  - How do we do the packing and unpacking?

Signed, unsigned, and why we care

- Some applications require attention to memory-use
  - Differences: one-million bytes, chars, and int
    - First requires a megabyte, last requires four megabytes
    - When do we care about these differences?
  - int values are stored as two's complement numbers with 32 bits, for 64 bits use the type long, a char is 16 bits
  - Java byte, int, long are signed values, char unsigned
  - Java signed byte: -128..127, # bits?
    - What if we only want 0-255? (Huff, pixels, ...)
    - Convert negative values or use char, trade-offs?
  - Java char unsigned: 0..65,536 # bits?
    - Why is char unsigned? Why not as in C++/C?
More details about bits

- How is 13 represented?
  - ...0 0 1 1 0 1
  - Total is 8+4+1 = 13
- What is bit representation of 32? Of 15? Of 1023?
  - What is bit-representation of 2^n - 1?
  - What is bit-representation of 0? Of -1?
    * Study later, but -1 is all 1’s, left-most bit determines < 0
- Determining what bits are on? How many on?
  - Understanding, problem-solving

How are data stored?

- To facilitate Huffman coding we need to read/write one bit
  - Why do we need to read one bit?
  - Why do we need to write one bit?
  - When do we read 8 bits at a time? 32 bits?
- We can’t actually write one bit-at-a-time. We can’t really write one char at a time either.
  - Output and input are buffered, minimize memory accesses and disk accesses
  - Why do we care about this when we talk about data structures and algorithms?
    * Where does data come from?

How do we buffer char output?

- Done for us as part of InputStream and Reader classes
  - InputStreams are for reading bytes
  - Readers are for reading char values
  - Why do we have both and how do they interact?
    Reader r = new InputStreamReader(System.in);
  - Do we need to flush our buffers?
- In the past Java IO has been notoriously slow
  - Do we care about I? About O?
  - This is changing, and the java.nio classes help
    * Map a file to a region in memory in one operation

Buffer bit output

- To buffer bits we store bits in a buffer (duh)
  - When the buffer is full, we write it.
  - The buffer might overflow, e.g., in process of writing 10 bits to 32-bit capacity buffer that has 29 bits in it
  - How do we access bits, add to buffer, etc.?
- We need to use bit operations
  - Mask bits -- access individual bits
  - Shift bits -- to the left or to the right
  - Bitwise and/or/negate bits
Representing pixels

- Pixel typically stores RGB and alpha/transparency values
  - Each RGB is a value in the range 0 to 255
  - The alpha value is also in range 0 to 255

  ```java
  Pixel red = new Pixel(255, 0, 0, 0);
  Pixel white = new Pixel(255, 255, 255, 0);
  ```

- A picture is simply an array of int values

  ```java
  void process(int pixel){
    int blue = pixel & 0xff;
    int green = (pixel >> 8) & 0xff;
    int red = (pixel >> 16) & 0xff;
  }
  ```

Bit masks and shifts

- Hexadecimal number: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
  - f is 15, in binary this is 1111, one less than 1000
  - The hex number 0xff is an 8 bit number, all ones

- Bitwise & operator creates an 8 bit value, 0—255
  - Must use an int/char, what happens with byte?
    - 1&1 == 1, otherwise we get 0 like logical and
    - Similarly we have |, bitwise or

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*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 \( \frac{1}{2} \) 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 (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
    }
}
```

**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