PFTWBTh

● **Review Priority Queues and O(log N)**
  ➢ Binary search, binary trees, binary heaps

● **Review for Midterm**
  ➢ Source code provided
  ➢ Handouts/What you bring
  ➢ What you bring, how you work on exam, lessons learned from Midterm I (both sides)

● **Toward Huffman Coding**
  ➢ Priority Queues and Data compressions
WOTO


More review possible
Priority Queue top-M sorting

• What if we have *lots and lots and lots of data*
  ➢ code below sorts top-M elements, complexity?

    Scanner s = ... // initialize;
    PriorityQueue<String> pq =
        new PriorityQueue<String>();
    while (s.hasNext()) {
        pq.add(s.next());
        if (pq.size() > M) pq.remove();
    }

• What's advantageous about this code?
  ➢ Store everything and sort everything?
  ➢ Store everything, sort first M?
  ➢ What is complexity of *sort*: $O(n \log n)$
Priority Queue implementations

● **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 list</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted list</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(n) build heap from n elements**
Use TreeSet (balanced Search Tree)

- `tree.subSet(4,12)`
  - [Link](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html#subSet-E-boolean-E-boolean-)

![TreeSet Diagram](image-url)
Trie

● reTRIEval structure supporting very efficient lookup, $O(w)$ where $w$ is length of query, regardless of number of entries in structure!

  ➢ 26-way branching
  ➢ N-way branching

● Map if sparse branching
Trie, Trie, and Trie again

- [https://git.cs.duke.edu/201fall16/set-examples/blob/master/src/TrieSet.java](https://git.cs.duke.edu/201fall16/set-examples/blob/master/src/TrieSet.java)

- **Method `.contains` is similar to others**
  - What does `Node` class look like?

```java
public boolean contains(String s) {
    Node t = myRoot;
    for (int k = 0; k < s.length(); k++) {
        char ch = s.charAt(k);
        t = t.children.get(ch);
        if (t == null)
            return false; // no path below? done
    }
    return t.isWord; // was this marked as a word?
}
```
Priority Queue implementation

- Heap data structure is fast and reasonably simple
  - Uses array, contiguous memory, good performance with cache and more

- Changing comparison when calculating priority?
  - Create object to replace, or in lieu of `compareTo`
    - Comparable interface compares `this` to passed object
    - Comparator interface compares two passed objects
  - Comparisons: `compareTo()` and `compare()`
    - Compare two objects (parameters or self and parameter)
    - Returns -1, 0, +1 depending on <, ==, >
Creating Heaps

- **Heap**: array-based implementation of binary tree used for implementing priority queues:
  - add/insert, peek/getmin, remove/deletemin, O(???)

- Array minimizes storage (no explicit pointers), faster too, contiguous (cache) and indexing

- Heap has *shape* property and *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
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 twos-complement storage to understand $x = x + 1$?
  - Do we need to understand how arrays map to contiguous memory to use ArrayLists?
Quicksort Partition (easy but slow)

**what we want**

<table>
<thead>
<tr>
<th>&lt;= pivot</th>
<th>&gt; pivot</th>
</tr>
</thead>
</table>

**what we have**

<table>
<thead>
<tr>
<th>????????????????</th>
</tr>
</thead>
</table>

**invariant**

<table>
<thead>
<tr>
<th>&lt;=</th>
<th>&gt;</th>
<th>???</th>
</tr>
</thead>
</table>

- **Easy to develop partition**

```java
int partition(String[] a, int left, int right) {
    string pivot = a[left];
    int k, pIndex = left;
    for(k=left+1, k <= right; k++) {
        if (a[k].compareTo(pivot) <= 0){
            pIndex++;
            swap(a,k,pIndex);
        }
    }
    swap(a,left,pIndex);
    return pIndex;
}
```

- **loop invariant:**
  - statement true each time loop test is evaluated, used to verify correctness of loop
- **Can swap into a[left] before loop**
  - Nearly sorted data still ok
Developing Loops

- The Science of Programming, David Gries
- The Discipline of Programming, Edsger Dijkstra
From goal to invariant to code

- Establish the invariant before loop, so true initially
- Re-establish the invariant in the loop as index increases (which could make invariant false)
- Two skills
  - Developing the invariant
  - Using the invariant to develop code
- Also have *class invariants* for development
what is search?

binary search
Why write the binary search method?

● After all, there is Collections.binarySearch
  ➢ Which of several equal keys found? 1,1,2,2,2,3,3,3
  ➢ Why does this matter?

● Look up the code online!?!#?!

● Why did you take Compsci 201?
  ➢ How to write/develop? How to randomly permute and hope? Search skills?
Coding Interlude: Reason about code

```java
public class Looper {
    public static void main(String[] args) {
        int x = 0;
        while (x < x + 1) {
            x = x + 1;
        }
        System.out.println("value of x = "+x);
    }
}
```
What does this code do?

```java
int x = 0;
while (x < x + 1) {
    x = x + 1;
}
System.out.println(x);
```

A. Runs Forever

B. Runs until memory exhausted (a few seconds with 8 Gb)

C. Runs for a second, prints about 2 billion

D. Runs for a second, prints about -2 billion
public static int binarySearch(int[] elements, int target) {
    int left = 0;
    int right = elements.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (target < elements[mid])
            right = mid - 1;
        else if (target > elements[middle])
            left = mid + 1;
        else return mid;
    }
    return -1;
}
public static int binarySearch(int[] elements, int target) {
    int left = 0;
    int right = elements.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (target < elements[mid])
            right = mid - 1;
        else if (target > elements[mid])
            left = mid + 1;
        else return mid;
    }
    return -1;
}
What should you remember?

$2^{10} = 1,024$

$2^{31}$ is about 2 billion

Store that many values in memory?
Don't know much about algebra

\[
\frac{\text{left} + \text{right}}{2} \\
\frac{\text{right} - \text{left}}{2} + \text{left} \\
\frac{\text{right}}{2} - \frac{\text{left}}{2} + \text{left} \\
\frac{\text{right}}{2} - \frac{\text{left}}{2} + 2\frac{\text{left}}{2} \\
\frac{\text{right}}{2} + \frac{\text{left}}{2} \\
\frac{\text{right} + \text{left}}{2}
\]
Huffman Coding

- Understand Huffman Coding
  - Data compression
  - Priority Queue
  - Bits and Bytes
  - Greedy Algorithm

- Many compression algorithms
  - Huffman is optimal, per-character compression
  - Still used, e.g., basis of Burrows-Wheeler
  - Other compression 'better', sometimes slower?
  - LZW, GZIP, BW, ...
Compression and Coding

- **What gets compressed?**
  - Save on storage, why is this a good idea?
  - Save on data transmission, how and why?

- **What is information, how is it compressible?**
  - Exploit redundancy, without that, hard to compress

- **Represent information: code (Morse cf. Huffman)**
  - Dots and dashes or 0's and 1's
  - How to construct code?
PQ Application: 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 there is more data to compress than ever before
  ➢ Ever need to stop worrying about storage?
More on Compression

- Different compression techniques
  - .mp3 files and .zip files?
  - .gif and .jpg?
- Impossible to compress/lossless everything: Why?
- Lossy methods
  - pictures, video, and audio (JPEG, MPEG, etc.)
- Lossless methods
  - Run-length encoding, Huffman

11 3 5 3 2 6 2 6 5 3 5 3 5 3 10
Coding/Compression/Concepts

● For ASCII we use 8 bits, for Unicode 16 bits
  ➢ Minimum number of bits to represent N values?
  ➢ Representation of genomic data (a, c, g, t)?
  ➢ What about noisy genomic data?

● We can use a variable-length encoding, e.g., Huffman
  ➢ How do we decide on lengths? How do we decode?
  ➢ Values for Morse code encodings, why?
  ➢ … - - - …
Huffman Coding

- D.A Huffman in early 1950’s: story of invention
  - Analyze and process data before compression
  - Not developed to compress data “on-the-fly”
- Represent data using variable length codes
  - Each letter/chunk assigned a codeword/bitstring
  - Codeword for letter/chunk is produced by traversing the Huffman tree
  - Property: No codeword produced is the prefix of another
  - Frequent letters/chunk have short encoding, while those that appear rarely have longer ones
- Huffman coding is optimal per-character coding method
Mary Shaw

- Software engineering and software architecture
  - Tools for constructing large software systems
  - Development is a small piece of total cost, maintenance is larger, depends on well-designed and developed techniques

- Interested in computer science, programming, curricula, and canoeing, health-care costs

- ACM Fellow, Alan Perlis Professor of Compsci at CMU
Huffman coding: *go go gophers*

### ASCII 3 bits

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Weight</th>
<th>Code</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
<td>3</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
<td>3</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
<td>3</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
<td>3</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
<td>3</td>
</tr>
<tr>
<td>sp</td>
<td>32</td>
<td>1000000</td>
<td>3</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 tree diagram]
Huffman coding: *go go gophers*

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>000</th>
<th>00</th>
<th>001</th>
<th>01</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
<th>1100000</th>
<th>111</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
<td>000</td>
<td>00</td>
<td>001</td>
<td>01</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>1100000</td>
<td>111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
<td>001</td>
<td>01</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>1100000</td>
<td>111</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
<td>010</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>1100000</td>
<td>111</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
<td>011</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>1100000</td>
<td>111</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
<td>100</td>
<td>1110</td>
<td>1111</td>
<td>1100000</td>
<td>111</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
<td>101</td>
<td>1111</td>
<td>1100000</td>
<td>111</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
<td>110</td>
<td>100</td>
<td>1110000</td>
<td>111</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000</td>
<td>111</td>
<td>101</td>
<td>1110000</td>
<td>111</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Encoding uses tree/trie:**
  - 0 left/1 right
  - How many bits? 37!!
  - Savings? Worth it?
Building a Huffman tree

- Begin with a forest of single-node trees/tries (leaves)
  - Each node/tree/leaf is weighted with character count
  - Node stores two values: character and count

- Repeat until there is only one node left: root of tree
  - Remove two minimally weighted trees from forest
  - Create new tree/internal node with minimal trees as children,
    - Weight is sum of children’s weight (no char)
- How does process terminate? Finding minimum?
  - Remove minimal trees, hummm……
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();
```
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? 20 Questions relevance?
    - Reading a trie? Leaf indicator? Node values?
Decoding a message

00000100001001101

Compsci 201, Fall 2016
Decoding a message

0000100001001101

G
Decoding a message

000100001001101
Decoding a message

00100001001101

Compsci 201, Fall 2016
Decoding a message

0100001001101

G
Decoding a message

100001001101
Decoding a message

00001001101

GO
Decoding a message

0001001101

GO
Decoding a message

001001101

GO
Decoding a message

01001101

GO
Decoding a message

1001101
Decoding a message

001101

GOO
Decoding a message

01101

G O O
Decoding a message

1101
Decoding a message

101

GOO
Decoding a message

G O O
Decoding a message

GOOD
Decoding a message

01100000100001001101

G O O D
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?
Huffman Complexities

- **How do we measure? Size of input file, size of alphabet**
  - Which is typically bigger?

- **Accumulating character counts: ______**
  - How can we do this in O(1) time, though not really

- **Building the heap/priority queue from counts ____**
  - Initializing heap guaranteed

- **Building Huffman tree ____**
  - Why?

- **Create table of encodings from tree ____**
  - Why?

- **Write tree and compressed file ____**