Scoreboard

<table>
<thead>
<tr>
<th>Algorithm</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>
<tr>
<td>Binary search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Balanced BST</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Priority Queues

- Basic operations
  - Insert
  - Remove extremal
- What properties must the data have?
- Applications
  - Event-driven simulation: Colliding particles
  - AI
  - Operating systems
  - Statistics
  - Graph searching
  - Data Compression: Huffman coding

Data Compression

- Compression is a high-profile application
  - .zip, .mp3, .aac, .jpg, .gif, .gz, .mpg, ...
  - 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

- Data compression facilitated by priority queue
  - All-time best assignment in a CompSci 100(e) course?
    - Subject to debate, of course

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, ...

11 3 5 3 2 6 2 6 5 3 5 3 5 3 10
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
  - 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 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>Search tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Balanced tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</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 (Java 5)

- 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 there is a Queue interface and PriorityQueue class
  - The PriorityQueue class also uses a heap

Sorting w/o Collections.sort(...)

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

- How does this work, regardless of pqueue implementation?
- What is the complexity of this method?
  - add O(1), remove O(log n)? If add O(log n)?)
  - heapsort uses array as the priority queue rather than separate pq object.
  - From a big-Oh perspective no difference: O(n log n)
    - Is there a difference? What’s hidden with O notation?
Priority Queue implementation

- **PriorityQueue** uses heaps, fast and reasonably simple
  - Why not use inheritance hierarchy as was used with Map?
  - Trade-offs when using HashMap and TreeMap:
    - Time, space
    - Ordering properties, e.g., what does TreeMap support?
- Changing method of 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
  - Both comparison methods: `compareTo()` and `compare()`
    - Compare two objects (parameters or self and parameter)
    - 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 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?
  - O(n), but 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)

void add(Object elt)
{
    // Add elt to heap in myList
    myList.add(elt);
    int loc = myList.size();
    while (1 < loc &&
        elt < myList[loc/2])
    {
        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

Text Compression

- Input: String S
- Output: String S'
  - Shorter
  - S can be reconstructed from S'
Text Compression: Examples

“abcde” in the different formats

<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>

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

Huffman coding: go go gophers

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Huffman</th>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>11001111</td>
<td>000</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>11011111</td>
<td>001</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>11100000</td>
<td>010</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>11010000</td>
<td>011</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
<td>100</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>11100101</td>
<td>101</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
<td>110</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>10000000</td>
<td>111</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, hummm....
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 $O( )$

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
Decoding a message

0110000100001001101

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

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?