Building Java Programs

Priority Queues, Huffman Encoding
Prioritization problems

- **Emergency room**
  - Gunshot victim should be treated before guy with a sore neck
  - Treat urgent cases first

- **Printing**
  - Print faculty jobs before student jobs
  - Print grad student jobs before undergrad jobs

- **Homework**
  - Work on things that are due soonest, even if given more recently

- What would be the runtime of solutions to these problems using the data structures we know (list, sorted list, map, set, BST, etc.)?
Inefficient structures

- **List**
  - Remove min/max by searching \( \mathcal{O}(N) \)
  - Problem: expensive to search

- **Sorted list**
  - Binary search it in \( \mathcal{O}(\log N) \) time
  - Problem: expensive to add/remove \( \mathcal{O}(N) \)

- **Binary search tree**
  - Go right for max in \( \mathcal{O}(\log N) \)
  - Problem: tree becomes unbalanced
Priority queue ADT

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
- Useful when we want to deal with things unequally

- Works like a queue: priority queue operations:
  - **add**: adds in order; \(O(\log N)\) worst
  - **peek**: returns *minimum* value; \(O(1)\) always
  - **remove**: removes/returns *minimum* value; \(O(\log N)\) worst
  - **isEmpty**, **clear**, **size**, **iterator**: \(O(1)\) always
Java's **PriorityQueue** class

```java
public class PriorityQueue<E> implements Queue<E>
```

<table>
<thead>
<tr>
<th>Method/Constructor</th>
<th>Description</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue&lt;E&gt;()</td>
<td>constructs new empty queue</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(E value)</td>
<td>adds value in sorted order</td>
<td>O(log N )</td>
</tr>
<tr>
<td>clear()</td>
<td>removes all elements</td>
<td>O(1)</td>
</tr>
<tr>
<td>iterator()</td>
<td>returns iterator over elements</td>
<td>O(1)</td>
</tr>
<tr>
<td>peek()</td>
<td>returns minimum element</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove()</td>
<td>removes/returns min element</td>
<td>O(log N )</td>
</tr>
<tr>
<td>size()</td>
<td>number of elements in queue</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

```java
Queue<String> pq = new PriorityQueue<String>();
pq.add("Helene");
pq.add("Melissa");
...
```
Inside a priority queue

- Usually implemented as a **heap**, a kind of binary tree.

- Instead of sorted left → right, it's sorted top → bottom
  - guarantee: each child is greater (lower priority) than its ancestors
  - add/remove causes elements to "bubble" up/down the tree
  - (take CSE 332 or 373 to learn about implementing heaps!)

Another use for this? **Heap sort**
Exercise: Fire the TAs

- We have decided that novice TAs should all be fired.
  - Write a class `TAManager` that reads a list of TAs from a file.
  - Find all with $\leq 2$ quarters experience and fire them.
  - Print the final list of TAs to the console, sorted by experience.
Priority queue ordering

- For a priority queue to work, elements must have an ordering
  - in Java, this means implementing the `Comparable` interface

- Reminder:

```java
public class Foo implements Comparable<Foo> {
    ...
    public int compareTo(Foo other) {
        // Return positive, zero, or negative integer
    }
}
```
Homework 8
(Huffman Coding)
File compression

- **compression**: Process of encoding information in fewer bits.
  - But isn't disk space cheap?

- Compression applies to many things:
  - store photos without exhausting disk space
  - reduce the size of an e-mail attachment
  - make web pages smaller so they load faster
  - reduce media sizes (MP3, DVD, Blu-Ray)
  - make voice calls over a low-bandwidth connection (cell, Skype)

- Common compression programs:
  - WinZip or WinRAR for Windows
  - Stuffit Expander for Mac
ASCII encoding

- **ASCII**: Mapping from characters to integers (binary bits).
  - Maps every possible character to a number (’A’ → 65)
  - uses one *byte* (8 *bits*) for each character
  - most text files on your computer are in ASCII format

<table>
<thead>
<tr>
<th>Char</th>
<th>ASCII value</th>
<th>ASCII (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>'z'</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>
Huffman encoding

- **Huffman encoding**: Uses variable lengths for different characters to take advantage of their relative frequencies.
  - Some characters occur more often than others. If those characters use < 8 bits each, the file will be smaller.
  - Other characters need > 8, but that's OK; they're rare.

<table>
<thead>
<tr>
<th>Char</th>
<th>ASCII value</th>
<th>ASCII (binary)</th>
<th>Hypothetical Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>32</td>
<td>00100000</td>
<td>10</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
<td>01100001</td>
<td>0001</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
<td>01100010</td>
<td>01110100</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
<td>01100011</td>
<td>001100</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
<td>01100101</td>
<td>1100</td>
</tr>
<tr>
<td>'z'</td>
<td>122</td>
<td>01111010</td>
<td>00100011110</td>
</tr>
</tbody>
</table>
Compressing text

- Key insight: characters occur unevenly
  - Common characters should use fewer bits
  - Uncommon characters should use more bits
  - Then average length of a file would decrease

- How can we come up with these encodings?
  - Hint: for each character we make a sequence of choices (0 or 1), kind of like “yes” or “no” answers in 20 Questions.
Huffman's algorithm

- **The idea:** Create a "Huffman Tree" that will tell us a good binary representation for each character.
  - Left means 0, right means 1.
    - example: 'b' is 10
  - More frequent characters will be "higher" in the tree (have a shorter binary value).

- To build this tree, we must do a few steps first:
  - **Count occurrences** of each unique character in the file.
  - Use a **priority queue** to order them from least to most frequent.
What you will write

• HuffmanNode
  • Binary tree node (à la 20 Questions)
  • Each storing a character and a count of its occurrences

• HuffmanTree
  • Two ways to build a Huffman-based tree
  • Output the Huffman codes to a file
  • Decode a sequence of bits into characters
Huffman compression

1. Count the occurrences of each character in file
   ' ' = 2, 'a' = 3, 'b' = 3, 'c' = 1, EOF = 1

2. Place characters and counts into priority queue

   ![Priority Queue]

3. Use priority queue to create Huffman tree

   ![Huffman Tree]

4. Traverse tree to find (char → binary) map

   { ' ' = 00, 'a' = 11, 'b' = 10, 'c' = 010, EOF = 011 }
Make code: “a dad cab”

\[
i \quad 0 \quad 1 \quad 2 \quad 32 \quad 97 \quad 98 \quad 99 \quad 100 \quad 255
\]

counts \[0, 0, 0, \ldots, 2, \ldots, 3, 1, 1, 2, \ldots, 0, 0] \]
Output encoding

```
+-----+
 | 9   |
+-----+
     /     \
   /       \\       \
 /         \\       \\      \\      \\      \\     \
| 4 |     | 5 |    | 2 |     | 2 |     | 3 |
+-----+    +-----+    +-----+    +-----+    +-----+    +-----+
     /       \\       \\      \\      \\      \\     \
   /         \\       \\      \\      \\      \\     \
  +-----+     +-----+     +-----+     +-----+     +-----+     +-----+  
 | 2 |      | 2 |      | 2 |      | 3 |      | 1 |      | 1 |      | 1 |
+-----+     +-----+     +-----+     +-----+     +-----+     +-----+     +-----+  
    \       \       \       \       \       \       \       \       \       \\
     d       / \       a
   / \      /   \\      /   \\    /   \\    /   \\    /   \\  
  +-----+  +-----+  +-----+  +-----+  +-----+  +-----+  +-----+  +-----+  
 | 1 |    | 1 |    | 1 |    | 1 |    | 1 |    | 1 |    | 1 |    | 1 |
+-----+  +-----+  +-----+  +-----+  +-----+  +-----+  +-----+  +-----+  +-----+
    b    c
```
Encoding

- For each character in the file
  - Determine its binary Huffman encoding
  - Output the bits to an output file
  - *Already implemented for you*

- Problem: how does one read and write bits?
Bit I/O streams

- Java's input/output streams read/write 1 byte (8 bits) at a time.
  - We want to read/write one single bit at a time.

BitInputStream: Reads one bit at a time from input.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public BitInputStream(String file)</td>
<td>Creates stream to read bits from given file</td>
</tr>
<tr>
<td>public int readBit()</td>
<td>Reads a single 1 or 0</td>
</tr>
<tr>
<td>public void close()</td>
<td>Stops reading from the stream</td>
</tr>
</tbody>
</table>

BitOutputStream: Writes one bit at a time to output.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public BitOutputStream(String file)</td>
<td>Creates stream to write bits to given file</td>
</tr>
<tr>
<td>public void writeBit(int bit)</td>
<td>Writes a single bit</td>
</tr>
<tr>
<td>public void close()</td>
<td>Stops reading from the stream</td>
</tr>
</tbody>
</table>
Encode (you don’t do this)

\[ \text{a d a d c a b} \]

32 (' ')
00
100 (d)
01
98 (b)
100
99 (c)
101
97 (a)
11
EOF

- We need a special character to say “STOP”
  - Otherwise, we may read extra characters (can only write whole bytes – 8 bits – at a time)
- We call this the EOF – end-of-file character
- Add to the tree manually when we construct from the int[] counts

- What value will it have?
  - Can’t represent any existing character

```
pq --> | 1 | | 1 | | 1 | | 2 | | 2 | | 3 |
      +----- +----- +----- +----- +----- +----- +-----
      b     c     eof     ''     d     a
```
Tree with EOF

+----+
| 10 |
+----+

0 / \ 1

+----+
| 4 |
+----+

0 / \ 1

+----+   +----+
| 2 |     | 2 |
+----+   +----+

\ \ d 0 / \ 1  a

+----+   +----+
| 1 |     | 2 |
+----+   +----+

eof 0 / \ 1

+----+   +----+
| 1 |     | 1 |
+----+   +----+

b c

32 (' ')

00

100 (d)

01

256 (eof)

100

98 (b)

1010

99 (c)

1011

97 (a)

11
Remaking the tree

32 (``)
00
100 (d)
01
256 (eof)
100
98 (b)
1011
99 (c)
1010
97 (a)
11
Decompressing

How do we decompress a file of Huffman-compressed bits?

• useful "prefix property"
  • No encoding A is the prefix of another encoding B
  • I.e. never will have $x \rightarrow 011$ and $y \rightarrow 011\_00110$

• the algorithm:
  • Read each bit one at a time from the input.
  • If the bit is 0, go left in the tree; if it is 1, go right.
  • If you reach a leaf node, output the character at that leaf and go back to the tree root.
Decoding

```
+-----+ 11000111 01001011 11101010 00000000
 | 10 |
+-----+
 0 / \ 1
 /  \
+-----+      +-----+
 | 4 |      | 6 |
+-----+      +-----+
 0 / \ 1      0 / \ 1
+-----+      +-----+      +-----+      +-----+
| 2 |      | 2 |      | 3 |      | 3 |
+-----+      +-----+      +-----+      +-----+
 d        0 / \ 1d      a
 /  \
+-----+      +-----+
 | 1 |      | 2 |
+-----+      +-----+
eof      0 / \ 1
+-----+      +-----+
 | 1 |      | 1 |
+-----+      +-----+
 b       c
```
Public methods to write

- **public `HuffmanTree`**(int[] counts)
  - Given character counts for a file, create Huffman tree

- **public void `write`**(PrintStream output)
  - Write the character-encoding pairs to the output file

- **public `HuffmanTree`**(Scanner input)
  - Reconstruct the Huffman tree from a code file

- **public void `decode`**(BitInputStream input, PrintStream output, int eof)
  - Use Huffman tree to decompress the input into the given output