Priority Queues & Huffman Encoding
Prioritization problems

- **ER scheduling:** You are in charge of scheduling patients for treatment in the ER. A gunshot victim should probably get treatment sooner than that one guy with a sore neck, regardless of arrival time. How do we always choose the most urgent case when new patients continue to arrive?
Structure Options

- **list**: store people in a list; remove min/max by searching \( O(N) \)
  - problem: expensive to search

- **sorted list**: store in sorted list; remove in \( O(1) \) time
  - problem: expensive to add \( O(N) \)

- **binary search tree**: store in BST, go right for min in \( O(\log N) \)
  - problem: tree becomes unbalanced
Java's `PriorityQueue` class

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element

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

public class PriorityQueue<String> pq = new PriorityQueue<String>();
pq.add("Rasika");
pq.add("Radu");
...```

<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>

Queue<String> pq = new PriorityQueue<String>();
pq.add("Rasika");
pq.add("Radu");
...
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!)
Homework 11
(Huffman Coding)
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>
aaaaabbbbbbaaaaaabbbbbbaaaaaaabbbbbbaaaaaabbbbbbaaaaaabbbbbbaaaaaaabbbbbbaaaaaabbbbbbaaaaaaabbbbbbaaaaaaabbbbbbaaaaaaabbbbbbaaaaaaabbbbbbaaaaaaabbbbbbaaaaaaabbbbbbaaaaaaabbbbb
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>001000000</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>
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

- Example: 0001010
  'cb'

'c' 'b' 'a' 'e'
Huffman compression

1. **Count** the occurrences of each character in file
   
   `\{ ' ': 2, 'a': 3, 'b': 3, 'c': 1, 'e': 1 \}`

2. Place characters and counts into **priority queue**

   ![Priority Queue Diagram]

3. Use priority queue to create **Huffman tree** →

4. **Traverse** tree to find (char → binary) map
   
   `\{ ' ': 00, 'a': 11, 'b': 10, 'c': 010, 'e': 011 \}`

5. For each char in file, **convert** to compressed binary version
   
   `a b a b c a b e 11 10 00 11 10 00 010 11 10 011`
1) Count characters

- **step 1:** count occurrences of characters into a map
- example input file contents:

  ab ab ab cab

<table>
<thead>
<tr>
<th>byte</th>
<th>count</th>
<th>ASCII</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'a'</td>
<td>2</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>'b'</td>
<td>3</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>' '</td>
<td>32</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>'a'</td>
<td>2</td>
<td>97</td>
<td>01100011</td>
</tr>
<tr>
<td>'b'</td>
<td>3</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>'c'</td>
<td>1</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>'a'</td>
<td>2</td>
<td>97</td>
<td>01100010</td>
</tr>
<tr>
<td>'b'</td>
<td>1</td>
<td>98</td>
<td>01100010</td>
</tr>
</tbody>
</table>

counts array:

- (in HW11, we do this part for you)
2) Create priority queue

- **step 2:** place characters and counts into a priority queue
  - store a single character and its count as a **Huffman node** object
  - the priority queue will organize them into ascending order
3) Build Huffman tree

- **step 2**: create "Huffman tree" from the node counts

  **algorithm:**
  - Put all node counts into a **priority queue**.
  - **while** P.Q. size > 1:
    - Remove two rarest characters.
    - Combine into a single node with these two as its children.
Build tree example
4) Tree to binary encodings

- The Huffman tree tells you the binary encodings to use.
  - left means 0, right means 1
  - example: 'b' is 10

- What are the binary encodings of:

  ' ', 'c', 'a'?
5) compress the actual file

- Based on the preceding tree, we have the following encodings:
  { ' ' = 00, 'a' = 11, 'b' = 10, 'c' = 010, 'e' = 011 }

- Using this map, we can encode the file into a shorter binary representation. The text ab ab cab would be encoded as:

<table>
<thead>
<tr>
<th>char</th>
<th>'a'</th>
<th>'b'</th>
<th>' '</th>
<th>'a'</th>
<th>'b'</th>
<th>' '</th>
<th>'c'</th>
<th>'a'</th>
<th>'b'</th>
<th>'e'</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>010</td>
<td>11</td>
<td>10</td>
<td>011</td>
</tr>
</tbody>
</table>

- Overall: 1110001110000101110011, (22 bits, ~3 bytes)
Compression example

Compressed binary:

```
dabdccadadad
```

```
0 1 1 1 0 0 0 1 0 1 1 0 1 1 0 1 1 0
```
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 $y \rightarrow 011100110$

● 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 root.
Decompressing

- Use the tree to decompress a compressed file with these bits:
  1011010001101011011

  - Read each bit one at a time.
  - If it is 0, go left; if 1, go right.
  - If you reach a leaf, output the character there and go back to the tree root.

- Output:
  bac aca
Public methods to write

- public HuffmanCode(int[] frequencies)
  - Given character frequencies for a file, create Huffman code (Steps 2-3)

- public void save(PrintStream output)
  - Write mappings between characters and binary to a output stream (Step 4)

- public HuffmanCode(Scanner input)
  - Reconstruct the tree from a .code file

- public void translate(BitInputStream input, PrintStream output)
  - Use the Huffman code to decode characters
Bit input stream

- Java's input stream reads 1 byte (8 bits) at a time.
  - We want to read 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 hasNextBit()</td>
<td>Checks to see if stream still has input</td>
</tr>
</tbody>
</table>
That's it!