Building Java Programs

Priority Queues, Huffman Encoding

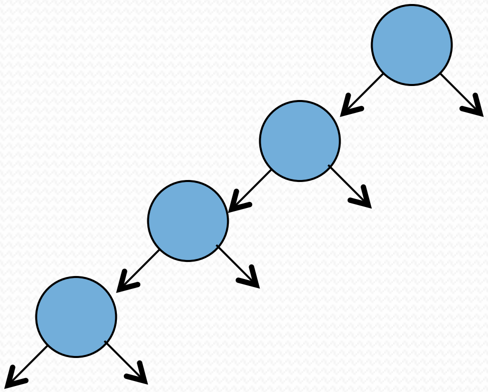
Prioritization problems

- **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 (O(*N*))
  - Problem: expensive to search
- *Sorted list*
  - Binary search it in O(log *N*) time
  - Problem: expensive to add/remove (O(*N*))
- *Binary search tree*
  - Go right for max in 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

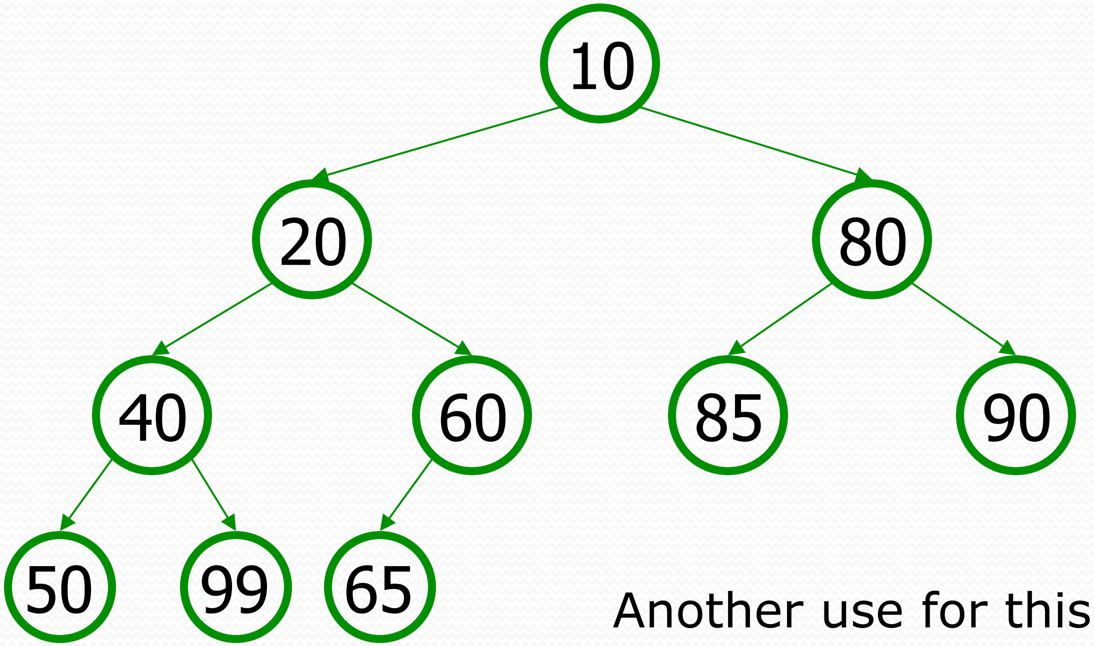
`public class PriorityQueue<`**`E`**`> implements Queue<`**`E`**`>`

Method/Constructor	Description	Runtime
`PriorityQueue<`**`E`**`>()`	constructs new empty queue	O(1)
`add(`**`E`**` value)`	adds value in sorted order	O(log *N*)
`clear()`	removes all elements	O(1)
`iterator()`	returns iterator over elements	O(1)
`peek()`	returns minimum element	O(1)
`remove()`	removes/returns min element	O(log *N*)
`size()`	number of elements in queue	O(1)

```
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 ≤ 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:

```
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

Char	ASCII value	ASCII (binary)
`' '`	32	00100000
`'a'`	97	01100001
`'b'`	98	01100010
`'c'`	99	01100011
`'e'`	101	01100101
`'z'`	122	01111010

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.

Char	ASCII value	ASCII (binary)	Hypothetical Huffman
' '	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	01110100
'c'	99	01100011	001100
'e'	101	01100101	1100
'z'	122	01111010	00100011110

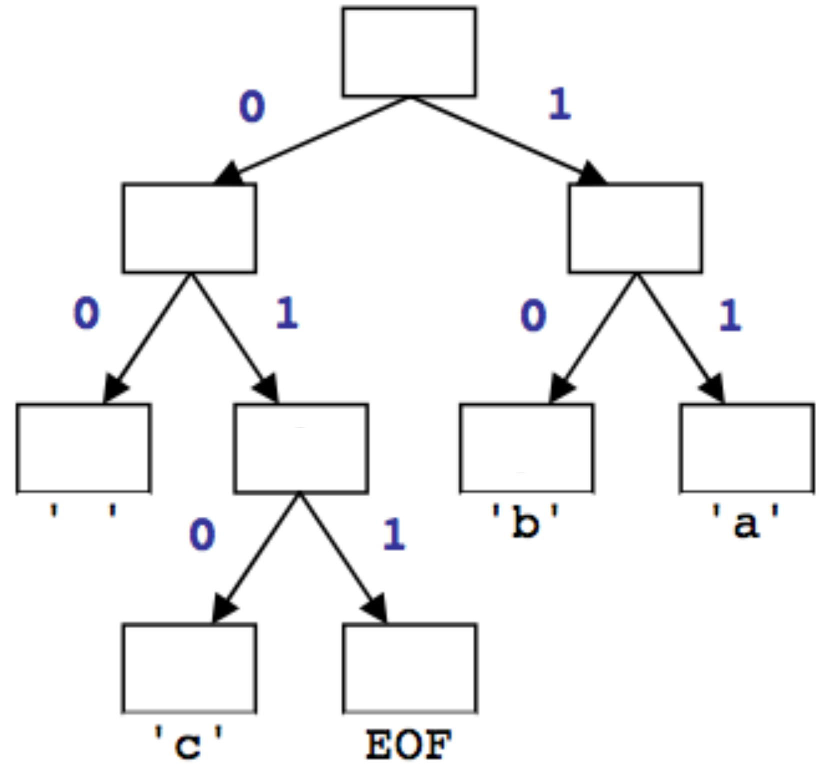
Compressing text

- 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

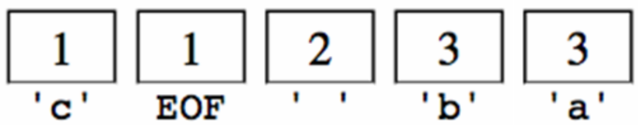
- HuffmanTree
  - Two ways to build a Huffman-based tree
  - Output the Huffman codes to a file
  - Decode a sequence of bits into characters

Huffman compression

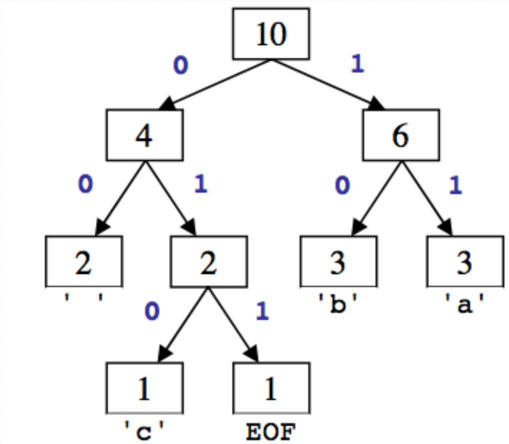
1. Count the occurrences of each character in file

`' '=2, 'a'=3, 'b'=3, 'c'=1, EOF=1`

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



3. Use priority queue to create **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          0 1 2      32     97 98 99 100           255
counts [0,0,0,...,2,..., 3, 1, 1, 2,..., 0, 0]
```

Output encoding

```
                +----+
                |  9 |
                +----+
               /      \
           /              \
     +----+                 +----+
     |  4 |                 |  5 |
     +----+                 +----+
     /    \                 /    \
+----+    +----+       +----+    +----+
|  2 |    |  2 |       |  2 |    |  3 |
+----+    +----+       +----+    +----+
 ' '        d           / \        a
                   +----+ +----+
                   |  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.

`public `**`BitInputStream`**`(String file)`	Creates stream to read bits from given file
`public int `**`readBit`**`()`	Reads a single 1 or 0
`public void `**`close`**`()`	Stops reading from the stream

- `BitOutputStream`: Writes one bit at a time to output.

`public `**`BitOutputStream`**`(String file)`	Creates stream to write bits to given file
`public void `**`writeBit`**`(int bit)`	Writes a single bit
`public void `**`close`**`()`	Stops reading from the stream

Encode (you don't do this)

```
                    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 |                     |  6 |
     +----+                     +----+
  0 /    \ 1                0 /    \ 1
+----+    +----+          +----+     +----+
|  2 |    |  2 |          |  3 |     |  3 |
+----+    +----+          +----+     +----+
 ' '        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`**`100110`

- 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 /     \ 1    a
                +----+    +----+
                |  1 |    |  2 |
                +----+    +----+
                 eof     0 / \ 1
                       +----+ +----+
                       |  1 | |  1 |
                       +----+ +----+
                          b      c
```

Public methods to write

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

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

- `public HuffanTree(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
  ```