

CSE 143

Lecture 24

Priority Queues and Huffman Encoding

slides created by Marty Stepp and Daniel Otero

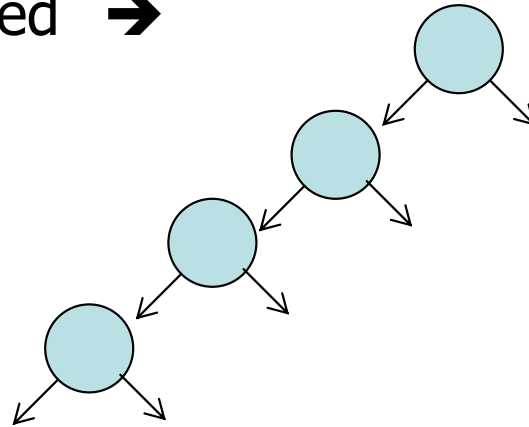
<http://www.cs.washington.edu/143/>

Prioritization problems

- The CSE lab printers constantly accept and complete jobs from all over the building. Suppose we want them to print faculty jobs before staff before student jobs, and grad students before undergraduate students, etc.?
- 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?
- Why can't we solve these problems efficiently with the data structures we have (list, sorted list, map, set, BST, etc.)?

Some poor choices

- *list* : store customers/jobs in a list; remove min/max by searching ($O(M)$)
 - problem: expensive to search
- *sorted list* : store in sorted list; binary search it in $O(\log M)$ time
 - problem: expensive to add/remove
- *binary search tree* : store in BST, search in $O(\log M)$ time for min element
 - problem: tree could be unbalanced →



- *auto-balancing BST*
 - problem: extra work must be done to constantly re-balance the tree

Priority queue ADT

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
 - a mix between a queue and a BST
 - usually implemented using a tree structure called a *heap*
- **priority queue operations**:
 - add adds in order; $O(\log N)$ worst
 - peek returns **minimum** element; $O(1)$
 - remove removes/returns **minimum** element; $O(\log N)$ worst
 - isEmpty,
clear,
size,
iterator $O(1)$

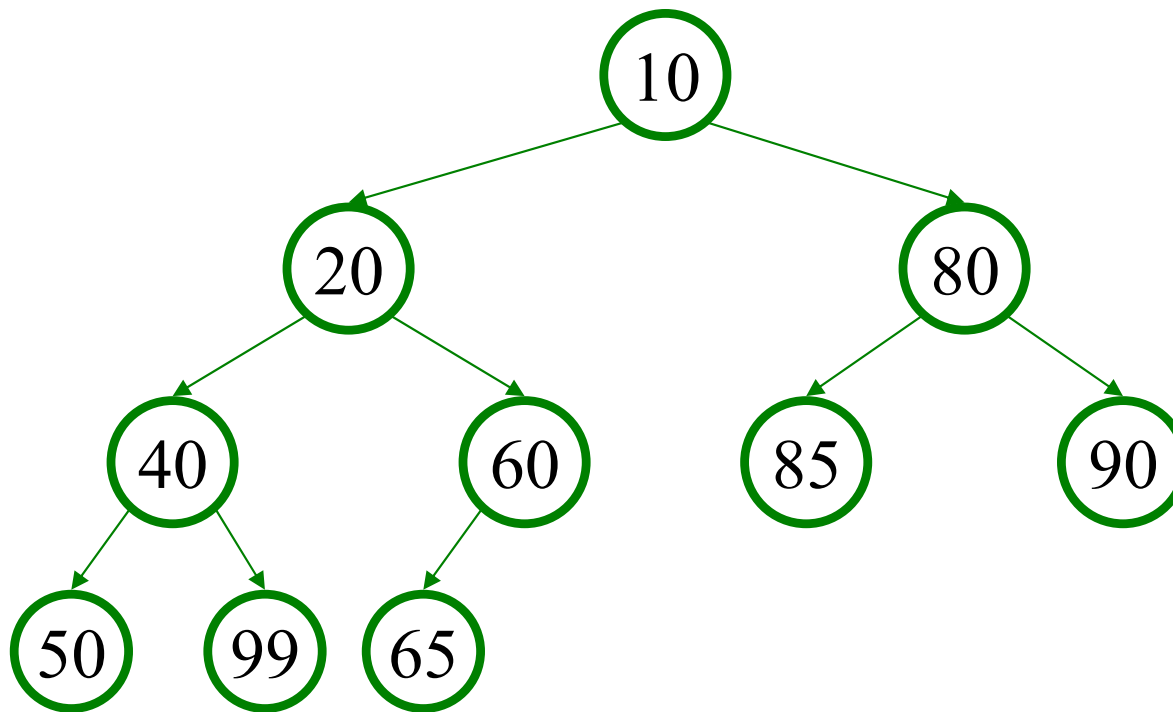
Java's PriorityQueue class

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

Method/Constructor	Description	Avg. Runtime
<code>public PriorityQueue<E>()</code>	constructs new empty queue	$O(1)$
<code>public void add(E value)</code>	adds value in sorted order	$O(\log N)$
<code>public void clear()</code>	removes all elements	$O(1)$
<code>public Iterator<E> iterator()</code>	returns iterator over elements	$O(1)$
<code>public E peek()</code>	returns minimum element	$O(1)$
<code>public E remove()</code>	removes/returns min element	$O(\log N)$

Inside a priority queue

- Usually implemented as a "heap": a kind of binary tree.
- Instead of sorted left \rightarrow right, it's sorted top \rightarrow bottom
 - guarantee: each child is greater (lower priority) than its ancestors



Exercise: Firing Squad

- We have decided that TA performance is unacceptably low.
 - We must fire all TAs with ≤ 2 quarters of experience.
- Write a class `FiringSquad`.
 - Its `main` method should read a list of TAs from a file, find all with sub-par experience, and replace them.
 - Print the final list of TAs to the console, sorted by experience.
 - Input format:

name	quarters	Lisa	0
name	quarters	Kasey	5
name	quarters	Stephanie	2

The caveat: ordering

- For a priority queue to work, elements must have an ordering
- aoeu
- Reminder:

```
public class Foo implements Comparable<Foo> {  
    ...  
    public int compareTo(Foo other) {  
        // Return positive, zero, or negative number...  
    }  
}
```


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 number...  
    }  
}
```

Homework 8

(Huffman Tree)

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



Homework Assignment 8

- You're going to make a WinZip clone except
 - without a GUI (graphical user interface)
 - it only works with a weird proprietary format (not ".zip")
- Your program should be able to compress/decompress files
 - "Compression" refers to size (bytes); compressed files are smaller



ASCII encoding

- At the machine level, everything is binary (1s and 0s)
 - Somehow, we must "encode" all other data as binary
 - **ASCII**: One of the most common character encodings.
 - Maps every possible character to a number ('A' → 65)
 - ASCII uses one *byte* (or eight *bits*) for each character:

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's idea: Why use the same length for every letter?
 - Some letters occur more often than others.
 - If those letters used fewer than 8 bits, the file would be smaller.
 - Other characters might need to use more than 8, but that's okay.
 - Idea: use fewer bits for high-frequency characters.

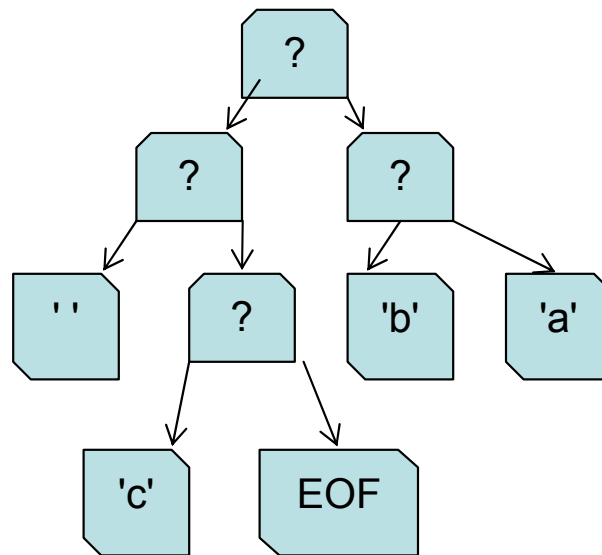
Char	ASCII value	ASCII (binary)	Hypothetical Huffman
' '	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	01110100
'c'	99	01100011	001100
'e'	101	01100101	1100
'z'	122	01111010	00100011110

Compressing a file

- Count occurrences of each character in the file.
 - Using: ?
- Place each character into priority queue using frequency comparison.
 - Using: a priority queue
- Convert priority queue to another binary tree via mystery algorithm X.
 - Using: a binary tree
- Traverse the tree to generate binary encodings of each character.
 - Using: ?
- Iterate over the source file again, outputting one of our binary encodings for each character we find.

"Mystery Algorithm X"

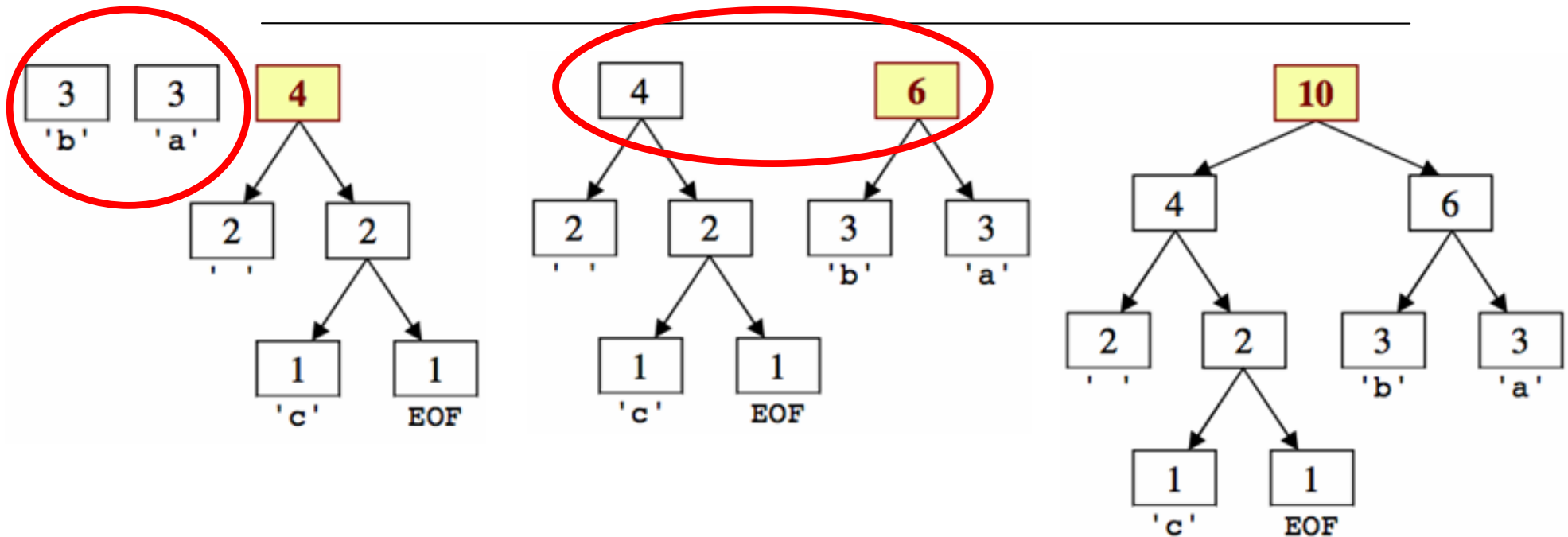
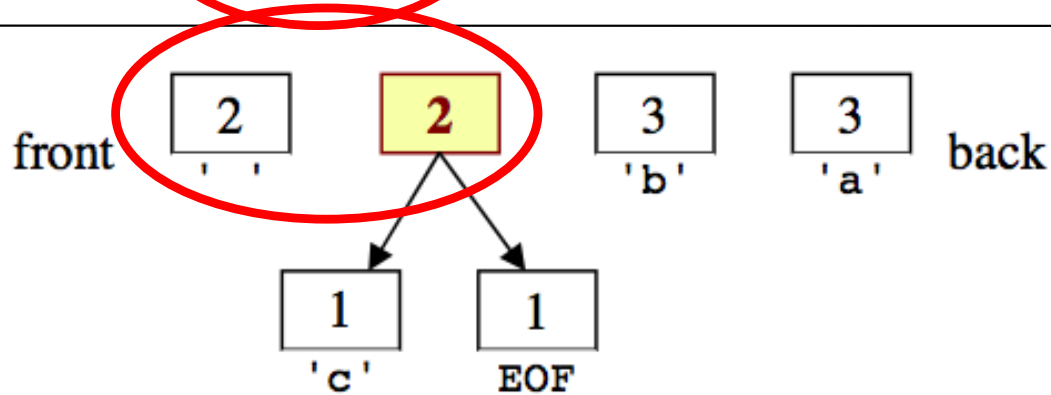
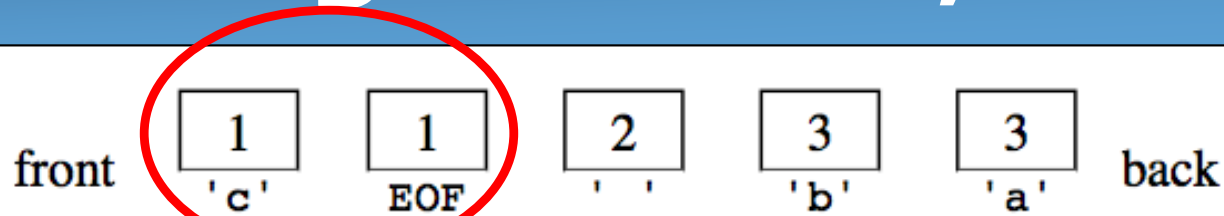
- The secret:
 - We'll build a tree with common chars on top
 - It takes fewer links to get to a common char
 - If we represent each link (left or right) with one bit (0 or 1), we automagically use fewer bits for common characters
- Tree for the example file containing text "ab ab cab":



Building the Huffman tree

- Create a binary tree node for each character containing:
 - The character
 - # of occurrences of that character
- Put all of the tree nodes into a priority queue.
- Until the queue has just one element:
 - Remove the two smallest nodes from the priority queue.
 - Join them together by making them children of a new node.
 - Set the new node's frequency to be the sum of its children's.
 - Reinsert the new node into the priority queue.
- Each iteration reduces the size of the queue by 1.

Building the tree, cont'd



HuffmanTree: Part I

- Class for HW8 is called `HuffmanTree`
 - Does both compression and decompression

- Compression:

```
public HuffmanTree(Map<Character, Integer> counts)
```

- Given a Map containing counts per character in an file, create its Huffman tree.

```
public Map<Character, String> createEncodings()
```

- Traverse your Huffman tree and produce a mapping from each character in the tree to its encoded binary representation as a String. For the previous example, the map is the following: `{' '=010, 'a'=11, 'b'=00, 'd'=011, 'n'=10}`

```
public void compress(InputStream in, BitOutputStream out)  
    throws IOException
```

- Read the text data from the given input file stream and use your Huffman encodings to write a Huffman-compressed version of this data to the given output file stream

Bit Input/Output Streams

- Filesystems have a lowest size denomination of 1 byte.
 - We want to read/write one *bit* at a time (1/8th of a byte)
- `BitInputStream`: like any other stream, but allows you to read one bit at a time from input until it is exhausted.

<code>public BitInputStream(InputStream in)</code>	Creates stream to read bits from given input
<code>public int readBit()</code>	Reads a single 1 or 0; returns -1 at end of file
<code>public boolean hasNextBit()</code>	Returns <code>true</code> iff another bit can be read
<code>public void close()</code>	Stops reading from the stream

- `BitOutputStream`: same, but allows you to *write* one bit at a time.

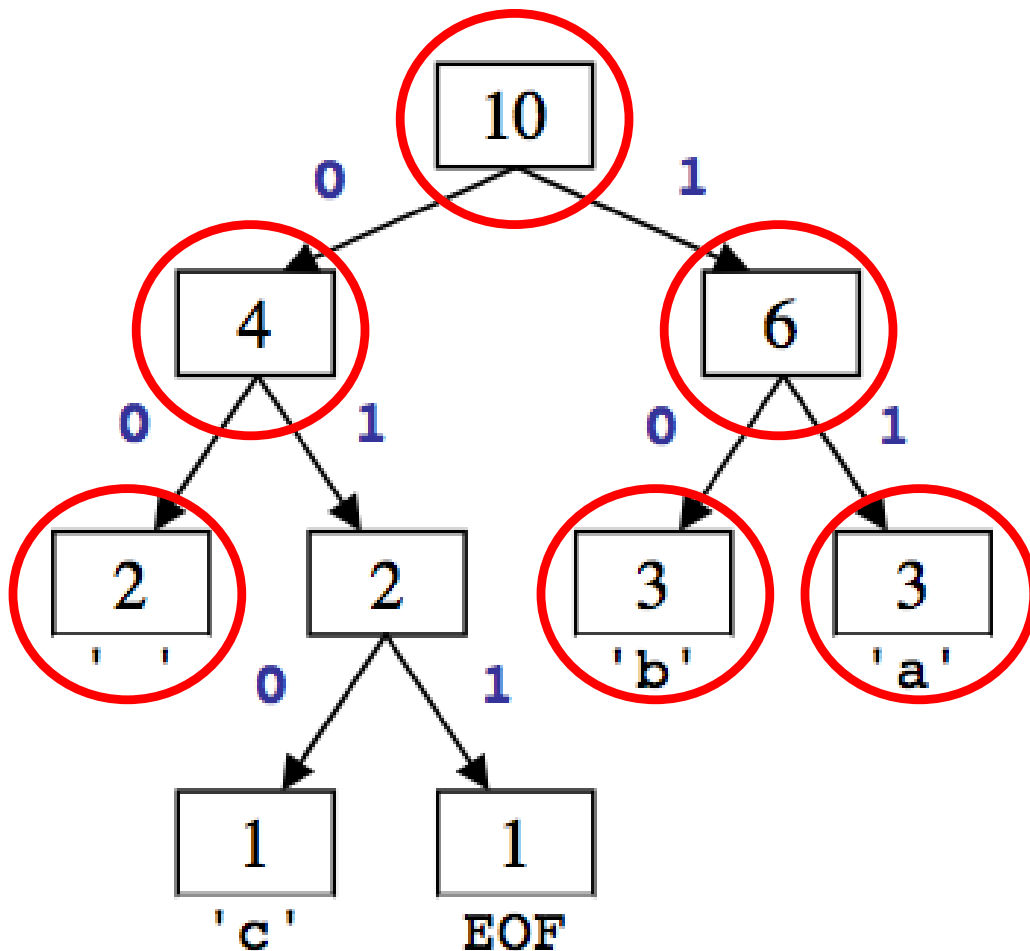
<code>public BitOutputStream(OutputStream out)</code>	Creates stream to write bits to given output
<code>public void writeBit(int bit)</code>	Writes a single bit
<code>public void writeBits(String bits)</code>	Treats each character of the given string as a bit ('0' or '1') and writes each of those bits to the output
<code>public void close()</code>	Stops reading from the stream

HuffmanTree: Part II

- Given a bunch of bits, how do we decompress them?
- Hint: HuffmanTrees have an encoding "prefix property."
 - No encoding A is the prefix of another encoding B
 - I.e. never will $x \rightarrow 011$ and $y \rightarrow 011100110$ be true for any two characters x and y
- Tree structure tells how many bits represent "next" character
- While there are more bits in the input stream:
 - Read a bit
 - If zero, go left in the tree; if one, go right
 - If at a leaf node, output the character at that leaf and go back to the tree root

HuffmanTree: Part II cont'd.

HuffmanTree for "ab ab cab"



Sample encoding

111000...

→ "ab "

HuffmanTree: Part II cont'd.

- The decompression functionality of `HuffmanTree` is handled by a single method:

```
public void decompress(BitInputStream in, OutputStream out)
    throws IOException
```

- Reads the compressed binary data from the given input file stream and use your Huffman tree to write a decompressed text version of this data to the given output file stream.
- You may assume that all characters in the input file were represented in the map of counts passed to your tree's constructor.

EOF?

- When reading from files, end is marked by special character: EOF ("End Of File")
 - NOT an ASCII character
 - Special code used by each particular OS / language / runtime

- Do you need to worry about it?
 - No, it doesn't affect you at all.
 - You may however notice it in your character maps, so don't get confused or worried.
 - FYI: EOF prints as a ? on the console or in jGRASP. (binary 256)