Topics:

- Continue discussion of **Trees**
  - So far we’ve studied several types of Binary Trees:
    - Binary Heaps (Priority Queue)
    - Binary Search Trees/BSTs (Map)
    - We also heard about some variants and cousins of the BST: red-black trees, splay trees, B-Trees
- Today we’re going to be talking about **Huffman trees**
Last couple thoughts on Tree Traversals
Traversals a very commonly-used tool in your CS toolkit

```c
void traverse(Node* node) {
    if (node != NULL) {
        traverse(node->left);
        // "do something"
        traverse(node->right);
    }
}
```

- Customize and move the “do something,” and that’s the basis for dozens of algorithms and applications

Stanford University
Stanford Library Map

- Remember how when you iterate over the Stanford library Map you get the keys in sorted order?
  - (we used this for the word occurrence counting code example in class)
- Now you know why it can do that in $O(N)$ time!
  - **Stanford library Map is a BST**
  - **In-order traversal on BST!**
Applications of the traversals

- You are writing the **destructor** for a BST class. Given a pointer to the root, it needs to free each node. Which traversal would form the foundation of your destructor algorithm?

A. Pre-order  
B. In-order  
C. Post-order  
D. Breadth-first
Applications of the traversals

- You are writing the **destructor** for a BST class. Given a pointer to the root, it needs to free each node. Which traversal would form the foundation of your destructor algorithm?

  A. Pre-order
  B. In-order
  C. Post-order
  D. Breadth-first

```cpp
void bstDestructorRecursiveHelper(Node *node) {
    if (node != NULL) {
        bstDestructorRecursiveHelper(node->left);
        bstDestructorRecursiveHelper(node->right);
        delete node; // post-order
    }
}
```
Breadth-First Tree Traversal

A somewhat different kind of traversal
How can we get code to print top-to-bottom, left-to-right order?

```cpp
void traverse(Node* node) {
    if (node != NULL) {
        cout << node->key << " ";
        traverse(node->left);
        traverse(node->right);
    }
}
```

You can’t do it by using this code and moving around the cout—we already tried moving the cout to all 3 possible places and it didn’t print in order

- You can but you use a **queue** instead of recursion
- “Breadth-first” search
- *Again we see this key theme of BFS (queue) vs DFS (stack/recursion)!*
Getting Started on Huffman
Encoding with Huffman Trees:

- Today we’re going to be talking about your next assignment: **Huffman coding**
  - It’s a compression algorithm
  - It’s provably optimal (take that, Pied Piper)
  - It involves binary tree data structures, yay!

- But before we talk about the tree structure and algorithm, let’s set the scene a bit and talk about BINARY
In a computer, everything is numbers!

Specifically, everything is binary

- Images (gif, jpg, png):
- Integers (int):
- Non-integer real numbers (double):
- Letters and words (ASCII, Unicode):
- Music (mp3):
- Movies (streaming):
- Doge pictures (🐶):
- Email messages:

Encodings are what tell us how to translate

- “if we interpret these binary digits as an image, it would look like this”
- “if we interpret these binary digits as music, it would sound like this”
In a computer, everything is numbers!

- Recall we represent variables as boxes:

  - What is contained in each box—whether it be an `int` or an `int*` or a `string` or anything else—is always some number of binary digits (bits).
  
  - We can’t know by looking at the bits whether they are being stored with the intention to be an `int` or an `int*` or a `string` or something else—just looks like bits.

111001101110011011111010

Number (int): 15132410

Color (RBG):
ASCII is an old-school encoding for characters

- The “char” type in C++ is based on ASCII
- Leftover from C in the 1970’s
- Doesn’t play well with other languages, and today’s software can’t afford to be so America-centric, so Unicode is more common
- ASCII is simple so we use it for this assignment
### ASCII Table

Notice each symbol is encoded as 8 binary digits (8 bits).

There are 256 unique sequences of 8 bits, so numbers 0-255 each correspond to one character. (*this only shows 32-74*)

00111110 = ‘<’
<table>
<thead>
<tr>
<th>char</th>
<th>ASCII</th>
<th>bit pattern (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>104</td>
<td>01101000</td>
</tr>
<tr>
<td>a</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>01110000</td>
</tr>
<tr>
<td>y</td>
<td>121</td>
<td>01111001</td>
</tr>
<tr>
<td>i</td>
<td>105</td>
<td>01101001</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>01101111</td>
</tr>
<tr>
<td>space</td>
<td>32</td>
<td>00100000</td>
</tr>
</tbody>
</table>

“happy hip hop” =
104 97 112 112 121 32 104 105 (decimal)

Or this in binary:

<table>
<thead>
<tr>
<th>01101000</th>
<th>01100001</th>
<th>01110000</th>
<th>01110000</th>
<th>01111001</th>
<th>00100000</th>
<th>01101000</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>01110000</td>
<td>00100000</td>
<td>01101000</td>
<td>01101111</td>
<td>01110000</td>
<td></td>
</tr>
</tbody>
</table>

FAQ: Why does 104 = ‘h’?
Answer: it’s arbitrary, like most encodings. Some people in the 1970s just decided to make it that way.
[Aside] Unplugged programming: The Binary Necklace

- Choose one color to represent 0’s and another color to represent 1’s
- Write your name in beads by looking up each letter’s ASCII encoding
- For extra bling factor, this one uses glow-in-the-dark beads as delimiters between letters

<table>
<thead>
<tr>
<th>DEC</th>
<th>OCT</th>
<th>HEX</th>
<th>BIN</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>101</td>
<td>41</td>
<td>01000001</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>102</td>
<td>42</td>
<td>01000010</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>103</td>
<td>43</td>
<td>01000011</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>104</td>
<td>44</td>
<td>01000100</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>105</td>
<td>45</td>
<td>01000101</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>106</td>
<td>46</td>
<td>01000110</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>107</td>
<td>47</td>
<td>01000111</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>108</td>
<td>48</td>
<td>01001000</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>109</td>
<td>49</td>
<td>01001001</td>
<td>I</td>
</tr>
</tbody>
</table>

Web tool to help you translate words to bead patterns: https://web.stanford.edu/~cbl/binary_bead_design.html
Non-ASCII (variable-length) encoding example

“happy hip hop” =

```
01 000 10 10 1111 110 01 001 10 110 01 1110 10
```

The variable-length encoding scheme makes a MUCH more space-efficient message than ASCII:

```
01101000 01100001 01110000 01110000 01111001 00100000 01101000
01101001 01110000 00100000 01101000 01101111 01110000
```
Huffman encoding

- Huffman encoding is a way of choosing which characters are encoded which ways, *customized to the specific file you are using*.
- Example: character ‘#’
  - Rarely used in Shakespeare (code could be longer, say ~10 bits)
  - If you wanted to encode a Twitter feed, you’d see # often (maybe only ~4 bits) #contextmatters #thankshuffman

- We store the code translation as a tree:
Your turn

What would be the binary encoding of “hippo” using this Huffman encoding tree?

A. 11000
B. 0101101010
C. 0100110101110
D. 0100010101111
E. Other/none/more than one
Okay, so how do we make the tree?

1. Read your file and count how many times each character occurs
2. Make a collection of tree nodes, each having a key = # of occurrences and a value = the character
   › Example: “c aaa bbb”
   › For now, tree nodes are not in a tree shape
   › We actually store them in a Priority Queue (yay!!) based on highest priority = LOWEST # of occurrences
   › Next:
     • Dequeue two nodes and make them the two children of a new node, with no character and # of occurrences is the sum,
     • Enqueue this new node
     • Repeat until PQ.size() == 1
Your turn

If we start with the Priority Queue above, and execute one more step, what do we get?

(A)  

(B)  

(C)
Last two steps

Stanford University
Now assign codes

We interpret the tree as:
- Left child = 0
- Right child = 1

What is the code for “c”?
A. 00
B. 010
C. 101
D. Other/none

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>010</td>
<td>10</td>
</tr>
</tbody>
</table>
Key question: How do we know when one character’s bits end and another’s begin?

Your turn:

**TRUE OR FALSE:** Huffman needs delimiters (like the glow-in-the-dark beads), unlike ASCII, which is always 8 bits (and didn’t really need the beads).

A. TRUE  
B. FALSE

Discuss/prove it: why or why not?

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>010</td>
<td>10</td>
</tr>
</tbody>
</table>