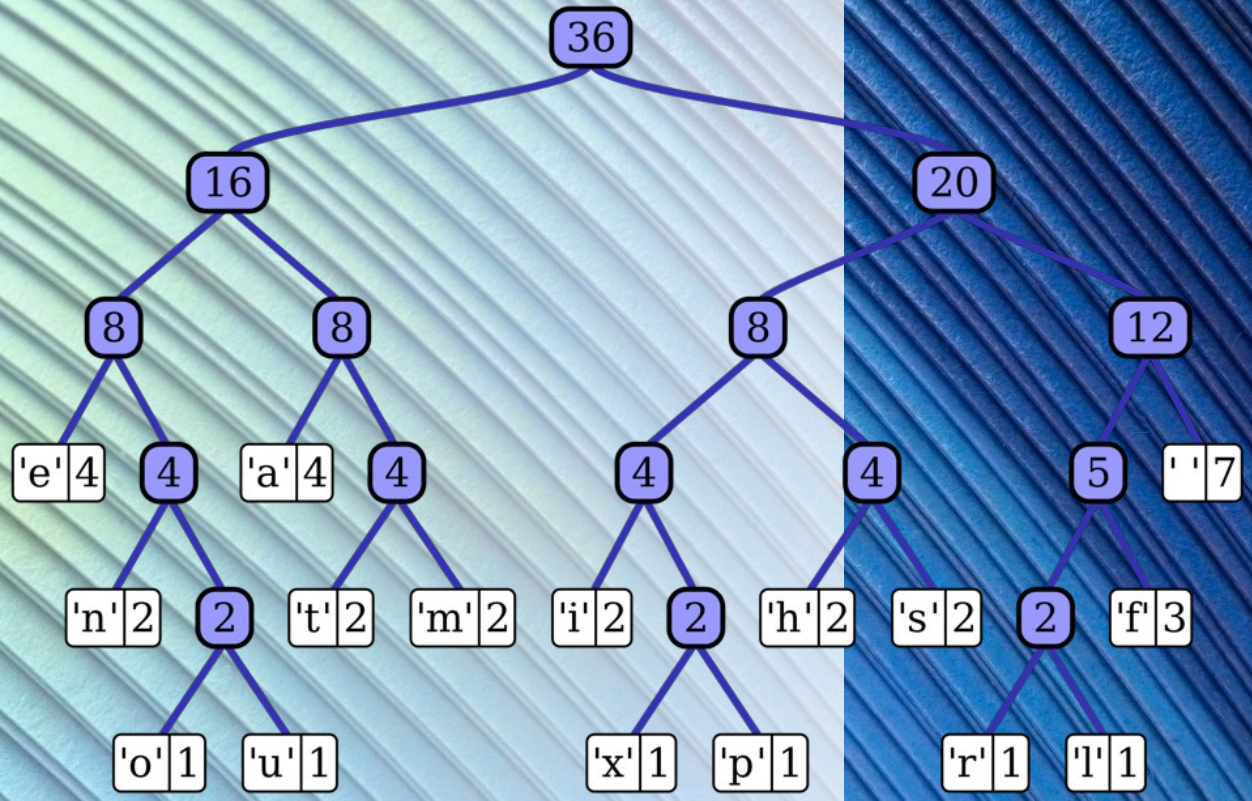
YEAH A6

Huffman Encoding



Press 1101 to pay respects. For the last YEAH :(

Today’s Topic

- This assignment is all about Huffman Encoding Trees! For the sake of time, I will not be reviewing the 8/5 lecture on Huffman Coding, but I highly recommend checking it out if you’d like extra review :)
- As you saw in that lecture, we can use **Huffman Encoding Trees** to **compress** massive amounts of data! In this final assignment, you’ll learn how to harness their power to write **your own compression/ decompression algorithm!**

Game Plan

- Part 1: Huffman Warmups
- Part 2: Huffman Routines
  - Only two parts :o

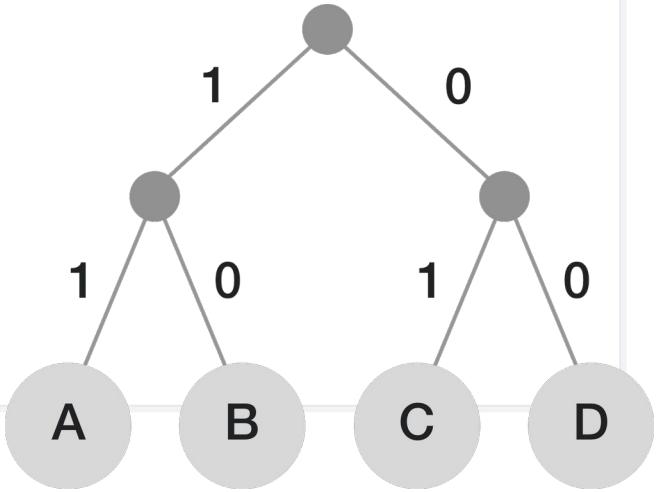
Part 1: The Warmups

- These warmups are **specifically** designed to help you complete the remainder of the assignment!
  - You'll be implementing key helper functions that you will use to debug / write tests for your code!
- The warmups are in two fundamental parts: short answer and coding!

Huffman Nodes

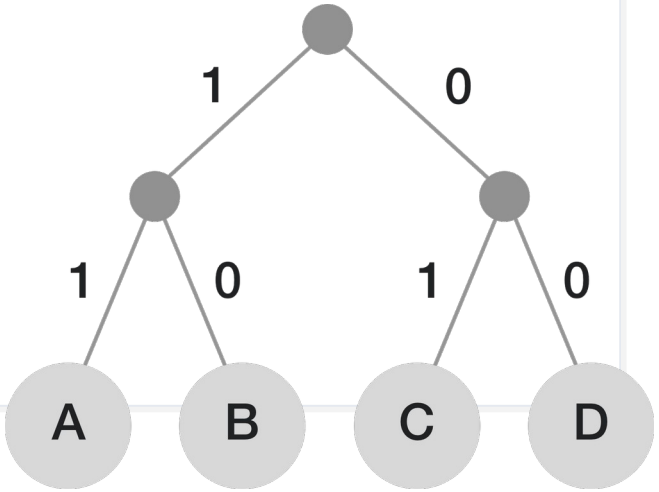
```
struct EncodingTreeNode {

    char ch;
    EncodingTreeNode* zero;
    EncodingTreeNode* one;

    EncodingTreeNode(char c) {  // use this constructor for new leaf node
        ch = c;
        zero = one = nullptr;
    }

    EncodingTreeNode(EncodingTreeNode* z, EncodingTreeNode* o) { // use this constructor for new interior node
        zero = z;
        one = o;
    }

    TRACK_ALLOCATIONS_OF(EncodingTreeNode); //  SimpleTest allocationg tracking
};
```

Part 1: The Warmups



- The first half of the warmups is primarily short-answer. In it, you will give examples of:
  1. How to encode / decode with a simple tree, like the one above.
  2. Flattening and unflattening a tree from a sequence of 1's and 0's (we'll talk about this one more.)
  3. Building optimal Huffman trees. (not all Huffman Trees are created equal!)

Flattening a Huffman Tree



- You can denote an entire **Huffman Encoding Tree** as a series of 1's and 0's! According to the handout:
  - If the root of the tree is a leaf node, it's represented by the bit 0.
  - If the root of the tree is not a leaf node, it's represented by a 1 bit, followed by the encoding of its zero (left) subtree, followed by the encoding of its one (right) subtree.
- For example, the above tree's flattened encoding would be 1100100.
  - Think about how you might write a recursive flattening algorithm… we'll return to this.

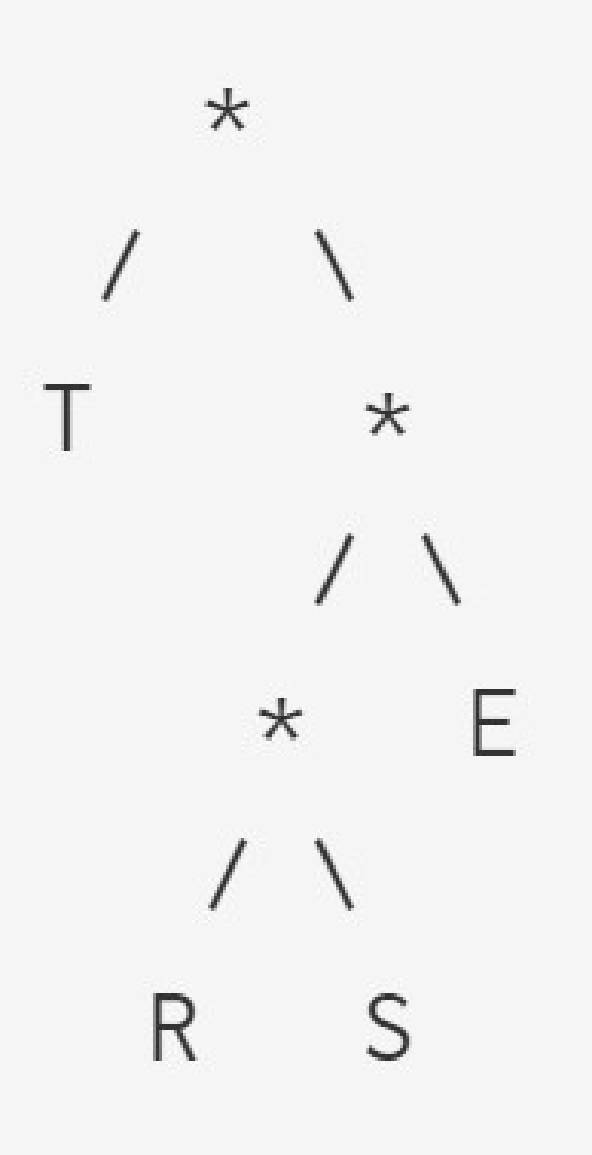
Oops. The zeroes and ones in this diagram are the opposite of what you'll see in A7!

Part 1: The Warmups

- For the second part of the warmups, you'll need to implement a few functions:
  - **createExampleTree()**: Hand-craft a small Huffman Tree!
  - **deallocateTree()**: Given a Huffman Tree node, free all associated nodes in the tree!
  - **areEqual**: Given two Huffman Tree nodes, can you verify that their corresponding trees are equal?

createExampleTree()

- You are tasked with creating the following Huffman Tree:
- There are two helpful EncodingTreeNode **constructors** you should consider using:
  1. **EncodingTreeNode**(EncodingTreeNode* z, EncodingTreeNode* o) – Allocate an interior tree node.
  2. **EncodingTreeNode**(char c) – Allocate a leaf node.
- Feel free to wire this tree up manually. You can verify your tree is correct by inspecting it in the debugger.



deallocateTree()

- Given an EncodingTreeNode*, delete all nodes associated with this tree.
  - To do this, think about which nodes you'll want to delete first – is there a specific tree traversal that lets us delete those nodes first?

areEqual()

- In this final warmup, write a function that takes in two EncodingTreeNode*'s and returns whether the trees that both represent are equal, meaning they have the same shape and values
  - **Important side note:** To compare two nodes, you'll probably want to compare their **ch** values – DO NOT do this for 'internal' nodes (i.e. leaves that not nodes.)
    - What should you do if you've determined your current node is an internal node?

Questions about the warmups?

Part II: Huffman Encoding

- It’s time for the grand finale! In this part, you will implement functions that let you **construct, flatten** and **unflatten** trees, and functions that let you **encode, decode, compress** and **decompress** user data!
- Let’s cover a few more loose ends before we jump in!

Part II: Huffman Encoding

- Before you start implementing, familiarize yourself with an class we've provided for you: the **Bit** class!
  - You can use a **Bit** much like an **int**, except a **Bit** can only be **1 or 0**, and an error will be raised if you try and set it to something else.
    - This is to help you! You'll work with these bits for encoding / decoding data, and you'll want to know if your bit values are not 0 or 1!
- Remember your friend the **priority queue?** You'll be using them on this assignment too, to help with tree construction!

decodeText()

- The first thing we want you to write is the function

```
string decodeText(EncodingTreeNode* tree, Queue<Bit>& messageBits)
```

- The **Queue<Bit>** given represents a series of 1’s and 0’s that are the Huffman Encoded data. You’re also given an **EncodingTreeNode*** that points the corresponding Huffman Tree.
- Your job is to translate the data, bit by bit, into a string by traversing the Huffman Tree!

decodeText()

```
string decodeText(EncodingTreeNode* tree, Queue<Bit>& messageBits)
```

Some notes about this problem:

- To traverse this tree, think about how you'd normally decode a sequence of bits into a string – if you begin at the base root of a tree, what does encountering a 0 do to your traversal vs. encountering a 1?
  - In a similar vein, how do you know when you've found a character?
- If you solve this problem with a 'curr' pointer to do your tree traversal, remember to reset it equal to root when you find a character! You need to repeat the top-down traversal process for every character in the decoded string!
- I'd recommend implementing this one iteratively. I don't think the iterative implementation is that bulky.

Questions about decodeText()?

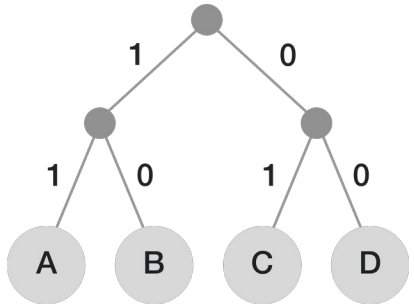
unflattenTree()

- In this second part, you'll need to implement the following:

```
EncodingTreeNode* unflattenTree(Queue<Bit>& treeBits, Queue<char>& treeLeaves)
```

- Remember our discussion of **compressing** a Huffman Tree into bits? In this function, you'll be converting a **Queue<Bit>** and a **Queue<char>** representing a Huffman Tree into a real node-based Huffman Tree, returning the pointer to the root of said tree.

unflattenTree()




```
EncodingTreeNode* unflattenTree(Queue<Bit>& treeBits, Queue<char>& treeLeaves)
```

- This one is trickier so let's examine it in the context of how to procedurally unflatten a tree (from the handout)
    - If the root of the tree is a leaf node, it's represented by the bit 0.
    - If the root of the tree is not a leaf node, it's represented by a 1 bit, followed by the encoding of its zero (left) subtree, followed by the encoding of its one (right) subtree.
- With these points in mind, let's think about how to unflatten a tree:
    - Take a bit from the **Queue<Bit>**
        - What should you do if this bit is a 0? Where can you find the necessary data for this?
        - Remember that this function returns an **EncodingTreeNode*** -- you need to return these too in your cases!
        - What should you do if this bit is a 1? You need to somehow get the tree corresponding to your child... Can recursion do that for you?
    - You can assume the queues are formatted as such to support this construction of a tree. You can assume the first bit in the bit queue represents the root of the tree.

Questions about unflattenTree()?

decompress()

```
string decompress(EncodedData& data)
```

- Time to combine your first two parts into a decompression routine!
  - To do so, you'll be working with an **EncodedData** struct that looks like this:

```
struct EncodedData {
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;
    Queue<Bit>  messageBits;
};
```

  - Given this struct, you'll be responsible for returning the correctly decoded string!

decompress()

```
struct EncodedData {
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;
    Queue<Bit>  messageBits;
};
```

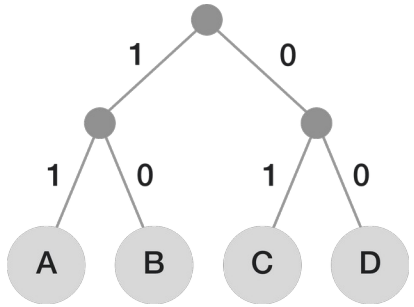
```
string decompress(EncodedData& data)
```

- To do so, you'll first want to create the Huffman Tree from the treeBits queue and the treeLeaves queue.
- Using that tree you'll then want to decode the messageBits into a readable string message that you will return to the caller!
- **Remember to free memory**! When you created the Huffman Tree, you allocated new nodes. Don't forget to free them.

Questions about decompress()?

Winrar is a popular file compression/decompression service for windows users (hence .rar files). It’s famous for ~~nobody paying for it~~ being an excellent data compression service.

encodeText()

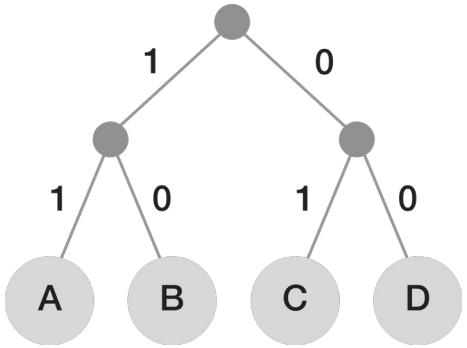



```
Queue<Bit> encodeText(EncodingTreeNode* tree, string text)
```

Congrats! Now it’s on to your final leg, encoding!

- In this part, you’ll be writing a function that takes in a string of text to be encoded, along with a valid Huffman Tree. It returns a **Queue<Bit>** representing the encoded message.
- The best way to create this queue is to first create a **map** that pairs **characters** to **bit sequences**. That way, you won’t need to do repeat-work for duplicate letters in the text string.

encodeText()




```
Queue<Bit> encodeText(EncodingTreeNode* tree, string text)
```

- You should write a helper function to create this map. To do this, consider making an empty **map** (one that pairs char's and sequences (**Vector<Bit>** is great!), and then try doing a **traversal** of the provided Huffman Tree.
  - As you make your way through the tree, be sure to keep track of the path that you've followed from the root (every time you explore a new location, tack it onto your path variable!)
  - When you encounter a leaf node, put this entry into your map, noting the ch value and the current part from root to node that you've kept track of!
  - Do this for the entirety of the tree!

encodeText()

```
Queue<Bit> encodeText(EncodingTreeNode* tree, string text)
```

- Once you've created this tree, your way forward should be more clear!
  - For each character in **text**, retrieve the proper sequence of encoding chars from your newly-made **map**. Enqueue these bits into a queue that you'll return!
- Once again, decomposition comes to the rescue!

Questions about encodeText()?

flattenTree()

```
void flattenTree(EncodingTreeNode* tree, Queue<Bit>& treeBits, Queue<char>& treeLeaves)
```

- As you might have guessed, in this function, you're going to reverse the work you did in **unflattenTree().** Specifically, you're going to turn the passed-in Huffman Tree into **two queues**: a queue of bits representing character encodings and a queue of chars representing the values at the leaves in the tree. (below is the flattening logic)
    - If the root of the tree is a leaf node, it's represented by the bit 0.
    - If the root of the tree is not a leaf node, it's represented by a 1 bit, followed by the encoding of its zero (left) subtree, followed by the encoding of its one (right) subtree.
- You will need to do a traversal of the tree. In this traversal, think about these points:
    - Because you are traversing the tree node-by-node, think about this as a bit-by-bit process. Each "decision" you make will add one piece of data to your queue(s)
    - If you reach a leaf node, what data do you need to put in the queues?
    - If you are on an interior node, what data do you need to put into the queues? Do you need to enter data in both queues? Are you done searching, or should you continue your traversal for your children?
    - Does a top-down or bottom-up traversal make more sense here? Remember that for unflatten(), the bit representing the root was the first element you dequeued!

Questions about flattenTree()?

buildHuffmanTree()

```
EncodingTreeNode* buildHuffmanTree(string text)
```

- It’s time to build the base for your Huffman Tree. Given a string of user data, it’s your job to build the **Optimal Huffman Tree.**
  - Don’t worry, this isn’t as bad as it sounds! Let’s go through it step by step!

buildHuffmanTree()

```
EncodingTreeNode* buildHuffmanTree(string text)
```

1. You're going to want to build a **frequency map** of characters in the text string.
   - Write a helper function that loops through the string, keeping a frequency count of all unique characters. Map<char, int> does this work just fine.
2. Once you have this map, create a **priority queue** of **EncodingTreeNode***'s. This will look like a `PriorityQueue<EncodingTreeNode*>`. With this queue in hand, for each character in your **frequency map, enqueue()** a new **EncodingTreeNode*** node, with the character as its value. For the priority of the element, you'll use the char's frequency in the map!
   - In case you're wondering, this pq is a min queue!

buildHuffmanTree()

```
EncodingTreeNode* buildHuffmanTree(string text)
```

3. Once you've filled this pq, you'll begin a **combination routine**, combining **pairs** on nodes repeatedly until only a single super node exists!
   - In a loop, pull off two elements from the queue. Then create a parent node! The first node you dequeue()'d will be the **zero** child of this parent, and the second node will be the **one** child.
   - Take this node trifecta and re-enqueue it to the pq, with a new priority – the sum of the priorities of both nodes you just dequeued!
     - You can get these priorities with the **peekPriority**() method before you dq the elements!

- Once there's only **one** node left, you have a complete Huffman Tree! Return it!

Questions about buildHuffmanTree() ?

- If you’re at all confused, the 5/29 lecture has some fantastic resources / videos about creating this tree!

compress()

```
struct EncodedData {
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;
    Queue<Bit>  messageBits;
};
```

```
EncodedData compress(string messageText)
```

- This is it! The last task – and it's not so bad! Much like **decompress()**, your job here is to put the pieces together.
  - Given a **string** messageText, you'll need to create a corresponding **Huffman Tree**, **encode** the text using that tree and the original message, **flatten** it, and then put your newly-acquired data into an **EncodedData** struct, **deleting** the tree before returning the struct.

Questions about compress()?

General notes:

- Be sure to test these functions as you go! It is **very common** on Huffman to only notice bugs once you try compressing/decompressing. Save yourself that frustration by testing early and testing often!
- Once you’re comfortable with your functions and believe they work, we’ve provided some larger files for you (pictures / sound files) to try compressing and decompressing. If you can get those to work, you’ll be in good shape.
- There are a lot of parts to this, and many of them can be tricky! My advice is to **not be afraid to restart a part** if you feel your solution is getting too complicated – these functions shouldn’t be too long / complex, and chances are, if you’re writing what you think is a herculean program, it may be too much logic.

Congratulations!

- You did it! You’re now ready to tackle the **final** CS106B assignment!
- Think about where you were at the start of the quarter – are you surprised at how much you’ve learned?