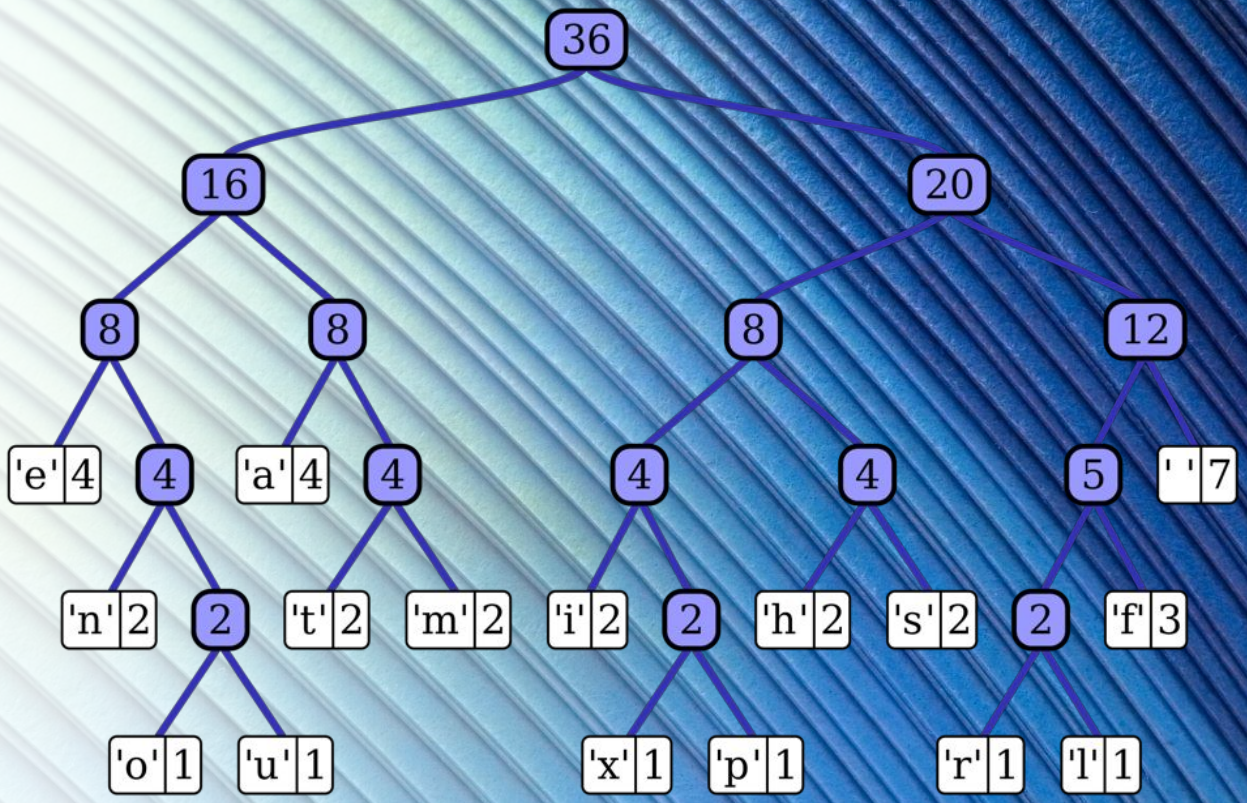
YEAH A9

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, we will not be reviewing lecture 24 on Huffman Coding, but we highly recommend checking it out if you'd like extra review 😊

Today's Topic

- This assignment is all about Huffman Encoding Trees! For the sake of time, we will not be reviewing lecture 24 on Huffman Coding, but we 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

- 5 Milestones, each designed to walk you through the Huffman algorithm!

Game Plan

- 5 Milestones, each designed to walk you through the Huffman algorithm!
  - Build the Huffman Tree

Game Plan

- 5 Milestones, each designed to walk you through the Huffman algorithm!
  - Build the Huffman Tree
  - Text encoding / decoding

Game Plan

- 5 Milestones, each designed to walk you through the Huffman algorithm!
  - Build the Huffman Tree
  - Text encoding / decoding
  - Tree encoding / decoding

Game Plan

- 5 Milestones, each designed to walk you through the Huffman algorithm!
  - Build the Huffman Tree
  - Text encoding / decoding
  - Tree encoding / decoding
  - Compress / decompress

Game Plan

- 5 Milestones, each designed to walk you through the Huffman algorithm!
  - Build the Huffman Tree
  - Text encoding / decoding
  - Tree encoding / decoding
  - Compress / decompress
  - Bask in the fruits of your labor!

Huffman Nodes

```
struct EncodingTreeNode {
    char ch;                    // Which character is stored here.
    EncodingTreeNode* zero; // Child tree labeled 0.
    EncodingTreeNode* one;  // Child tree labeled 1.
};
```

Something worth noting: if an EncodingTreeNode is NOT a leaf node, should you be examining the ch value?

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

- 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.

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

- 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* huffmanTreeFor(const string& str);
```

1. You're going to want to build a **frequency map** of characters in the text string.

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

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.

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

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!

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

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* huffmanTreeFor(const string& str);
```

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!

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

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.

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

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!

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

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!

buildHuffmanTree()

```
EncodingTreeNode* huffmanTreeFor(const string& str);
```

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!

A question to think about: What is the intuition behind doing this?

Questions about buildHuffmanTree() ?

- If you're at all confused, lecture 24 has some fantastic resources about creating this tree!

The Bit class

- Before you start implementing more, familiarize yourself with an class we've provided for you: the Bit class!

The Bit class

- Before you start implementing more, 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 (similar to a boolean in that it only has two possible values), and an error will be raised if you try and set it to something else.

The Bit class

- Before you start implementing more, 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 (similar to a boolean in that it only has two possible values), 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!

decodeText()

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

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

decodeText()

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

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

- 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.

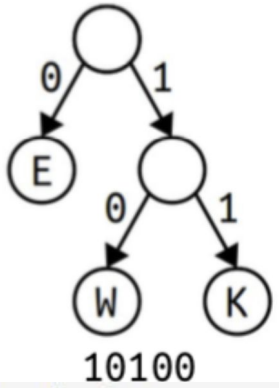
decodeText()

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

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

- 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!

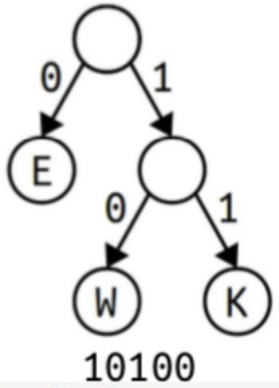
decodeText()




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

Some notes about this problem:

decodeText()

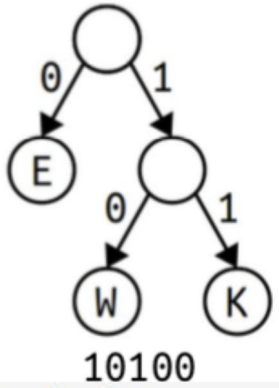



```
string decodeText(Queue<Bit>& bits,  EncodingTreeNode* tree);
```

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?

decodeText()

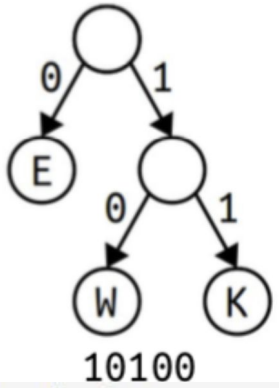



```
string decodeText(Queue<Bit>& bits,  EncodingTreeNode* tree);
```

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!

decodeText()

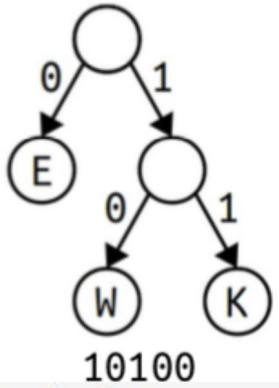



```
string decodeText(Queue<Bit>& bits,  EncodingTreeNode* tree);
```

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.

decodeText()




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

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.
- Check out lecture 24 starting at slide 55 to see an example walkthrough!

encodeText()

```
Queue<Bit> encodeText(const string& str, EncodingTreeNode* tree);
```

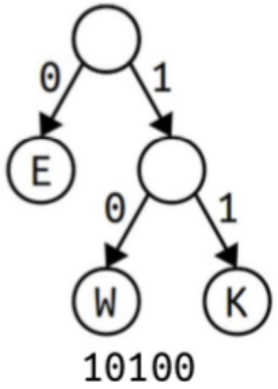
- In this part, you'll be writing a function that does the opposite of decodeText(). It takes in a string of text to be encoded, along with a valid Huffman Tree. It returns a Queue<Bit> representing the encoded message.

encodeText()

```
Queue<Bit> encodeText(const string& str, EncodingTreeNode* tree);
```

- In this part, you'll be writing a function that does the opposite of decodeText(). It 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 (and likely the only 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.

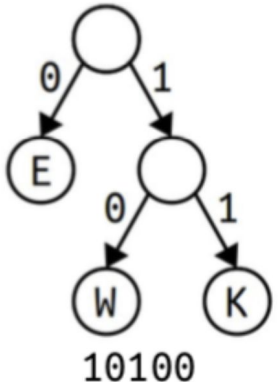
encodeText()




```
Queue<Bit> encodeText(const string& str, EncodingTreeNode* tree);
```

- 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.

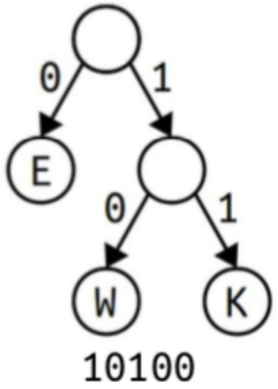
encodeText()




```
Queue<Bit> encodeText(const string& str, EncodingTreeNode* tree);
```

- 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!)

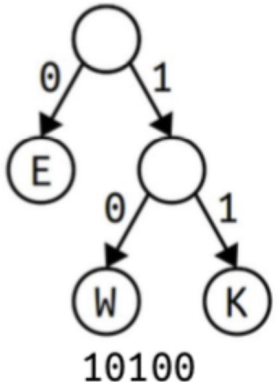
encodeText()




```
Queue<Bit> encodeText(const string& str, EncodingTreeNode* tree);
```

- 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!

encodeText()




```
Queue<Bit> encodeText(const string& str, EncodingTreeNode* tree);
```

- 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(const string& str, EncodingTreeNode* tree);
```

- 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!

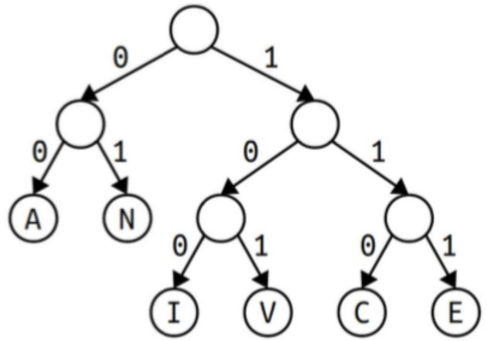
encodeText()

```
Queue<Bit> encodeText(const string& str, EncodingTreeNode* tree);
```

- 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 decode / encodeText()?

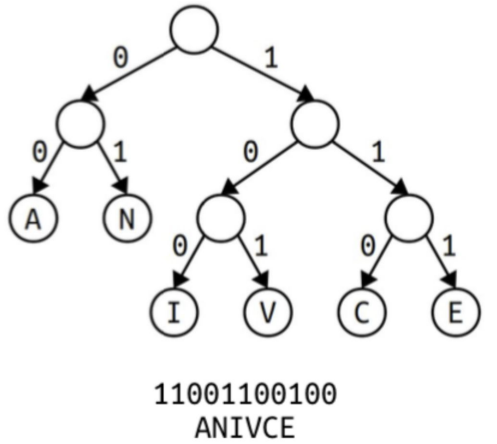
Encoding a Huffman Tree



11001100100
ANIVCE

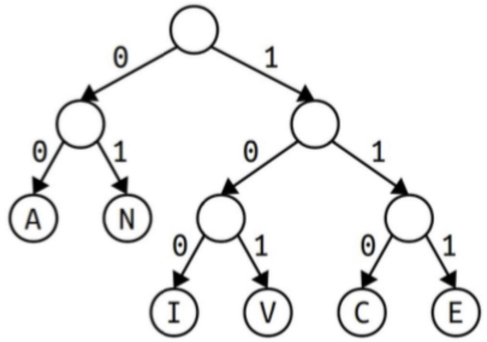
- You can encode an entire Huffman Encoding Tree as a series of 1's and 0's! According to the handout:

Encoding a Huffman Tree



- You can encode 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.

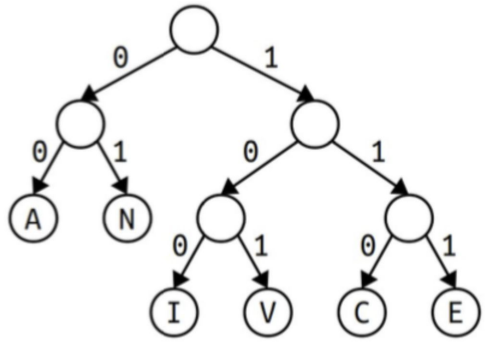
Encoding a Huffman Tree



11001100100
ANIVCE

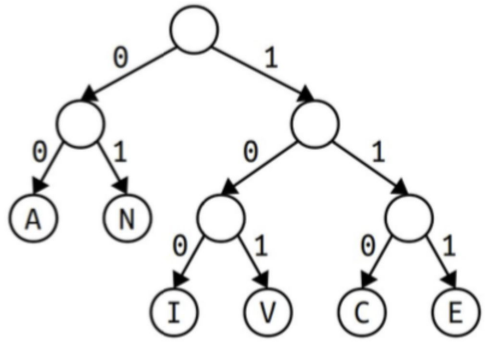
- You can encode an entire Huffman Encoding Tree as a series of 1's and 0's! According to the handout:
  - If the root of the tree is a leaf node, it's represented by the bit 0.
  - If the root of the tree is not a leaf node, it's represented by a 1 bit, followed by the encoding of its zero (left) subtree, followed by the encoding of its one (right) subtree.

Encoding a Huffman Tree



- You can encode 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 11001100100.

Encoding a Huffman Tree



- You can encode 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 11001100100.
  - Think about how you might write a recursive flattening algorithm… we'll return to this.

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- Here's where it gets more involved. Your task is 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)

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- Here's where it gets more involved. Your task is 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.

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- Here's where it gets more involved. Your task is 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.

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- You will need to do a traversal of the tree. In this traversal, think about these points:

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- 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).

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- 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?

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- 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?

encodeTree()

```
void encodeTree(EncodingTreeNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
```

- 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 encodeText(), the bit representing the root was the first element you dequeued!

Questions about encodeTree()?

decodeTree()

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

```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

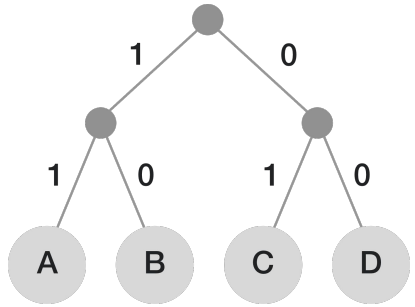
decodeTree()

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

```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- 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.

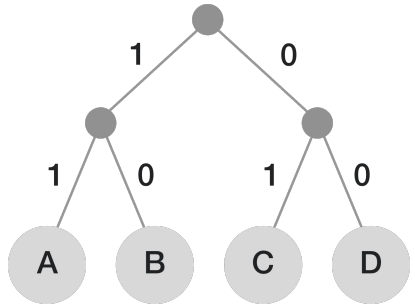
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- This one is trickier so let's examine it in the context of how to procedurally unflatten a tree (from the handout)

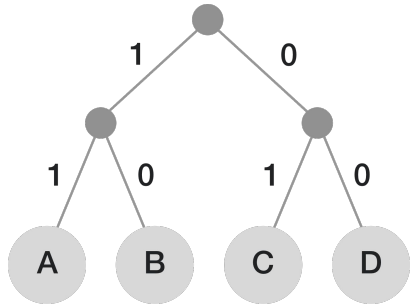
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- 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.

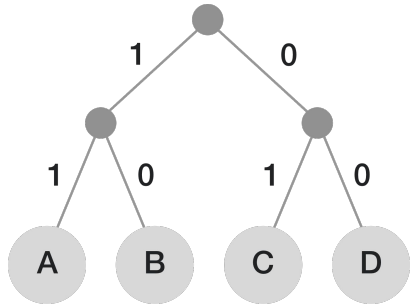
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- 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.

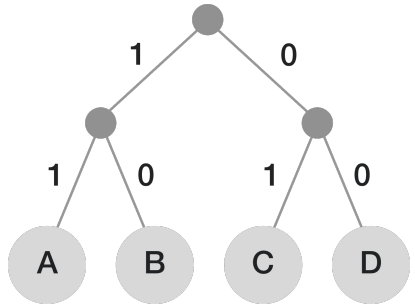
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- With these points in mind, let's think about how to unflatten a tree:

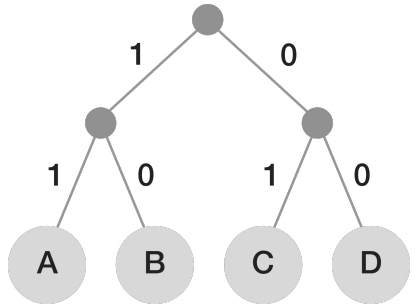
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- With these points in mind, let's think about how to unflatten a tree:
  - Take a bit from the Queue<Bit>

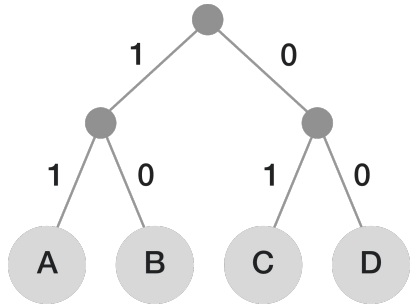
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- 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?

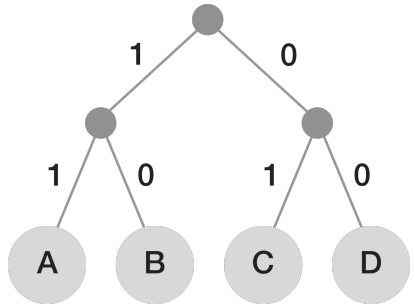
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- 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!

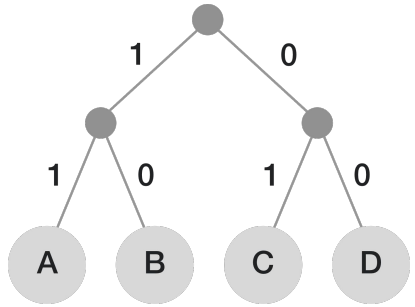
decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- 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?

decodeTree()




```
EncodingTreeNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

- 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 decodeTree()?

compress()

```
HuffmanResult compress(const string& text);
```

- You're almost done! 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 **HuffmanResult** struct, **deleting** the tree before returning the struct.

```
struct HuffmanResult {
    /* Encoded version of the Huffman tree. */
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;

    /* Encoded version of the message. */
    Queue<Bit> messageBits;
};
```

decompress()

```
string decompress(HuffmanResult& file);
```

```
struct HuffmanResult {
    /* Encoded version of the Huffman tree. */
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;

    /* Encoded version of the message. */
    Queue<Bit> messageBits;
};
```

- It's time to wrap things up with a decompression routine!

decompress()

```
string decompress(HuffmanResult& file);
```

```
struct HuffmanResult {
    /* Encoded version of the Huffman tree. */
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;

    /* Encoded version of the message. */
    Queue<Bit> messageBits;
};
```

- It's time to wrap things up with a decompression routine!
- To do so, you'll first want to create the Huffman Tree from the treeBits queue and the treeLeaves queue.

decompress()

```
string decompress(HuffmanResult& file);
```

```
struct HuffmanResult {
    /* Encoded version of the Huffman tree. */
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;

    /* Encoded version of the message. */
    Queue<Bit> messageBits;
};
```

- It's time to wrap things up with a decompression routine!
- 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!

decompress()

```
string decompress(HuffmanResult& file);
```

```
struct HuffmanResult {
    /* Encoded version of the Huffman tree. */
    Queue<Bit>  treeBits;
    Queue<char> treeLeaves;

    /* Encoded version of the message. */
    Queue<Bit> messageBits;
};
```

- It's time to wrap things up with a decompression routine!
- 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.

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!

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!
- I think an isLeaf() function would be a great helper :)

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!
- I think an isLeaf() function would be a great helper :)
- 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.

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!
- I think an isLeaf() function would be a great helper :)
- 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?