YEAH A6: Huffman Coding

CS106B Summer '21

Jin-Hee Lee, Grant Bishko, Lauren Saue-Fletcher


106Bers, I have a feeling we're not in soundex anymore.

Here is a picture of us, getting our C++ legs back in June. V cute.

Us, recursing.

Using everything we've learned in 106B to code up a final, amazing assignment that shows how far we've come.

We are so close to the finish line and so proud of everything you've done this quarter!

Our last YEAH logistics slide :(

- Assignment 6 is due on Wed, August 11 at 11:59 pm PDT
- *** There is no grace period for this assignment ***
    - We have some unmovable deadlines from the university with the end of the Summer Session, so *please please please* submit by the deadline.

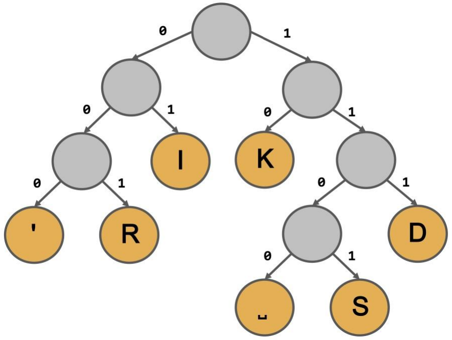
Huffman Coding

- Warmup
- Decode / decompress
- Encode / compress

Huffman Coding

- **Warmup**
- Decode / decompress
- Encode / compress

Let's talk Huffman.



- Given some message -> comes up with an efficient way of storing that message
- More frequent letters are represented by shorter sequences
- Our job is to implement encode and decode
- Along the way: lots of practice with trees :)

What are traversal orders??

Pre-Order: **root**, go left, go right

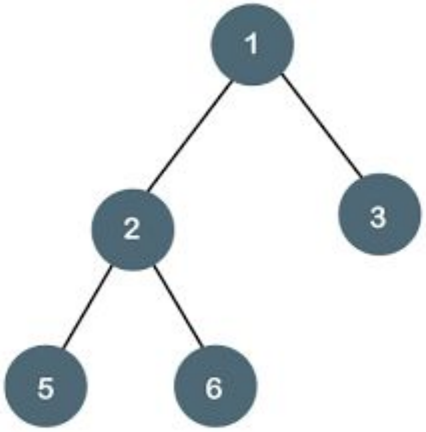
- 12563

In-Order: go left, **root**, go right

- 52613

Post-Order: go left, go right, **root**

- 56231

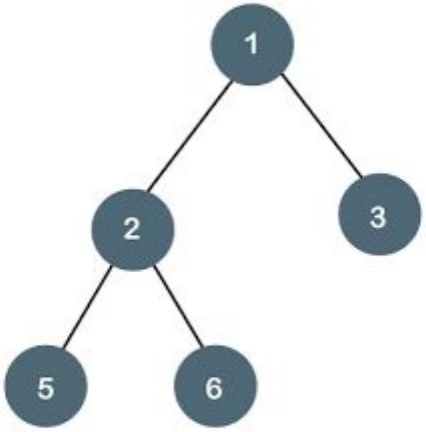


How do I remember them all?

Pre-Order: **root**, go left, go right

In-Order: go left, **root**, go right

Post-Order: go left, go right, **root**



1) Recursively going left is always before recursively going right
2) pre/in/post tells us where **root** goes

When do I use each? **sorta/kinda**

Pre-Order: **root**, go left, go right

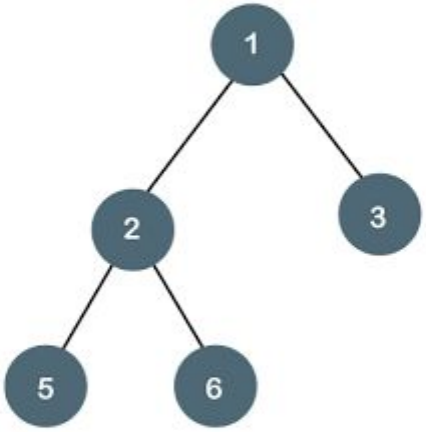
- Creation

In-Order: go left, **root**, go right

- Go through leaves "in order"

Post-Order: go left, go right, **root**

- Deletion



Encoding and decoding using an encoding tree (Q1-Q3)

```
     *
    / \
   *   O
  / \
 N   *
    / \
   M   S
```

Q1: decode a sequence

Q2: encode a string

- Check out lecture examples

Q3: Why prefix property?

- What would be true of the two characters if one was a prefix of another? Draw out the tree!

Flattening and unflattening an encoding tree (Q4-Q5)

Q4: Flatten a tree

Q5: Unflatten some sequences

```
    *                      *                      *
   / \                    / \                    / \
  E   *                  A   *                  *   O
     / \                    / \                / \
    W   K                  *   N              N   *
                          / \                    / \
                         D   B                  M   S

10100                  1011000
EWK                    ADBN
```

What traversals are we using?

- There’s a reason we give you these warm ups :)

Important aside: Bit

A new variable type that can only hold a 1 or 0

```
Queue<Bit> q;
Bit b = 0;
q.enqueue(0);
if (q.dequeue() == 1) { }
```

Helps avoid annoying bugs :)

Generating an optimal Huffman tree (Q6-Q8)

Q6: Construct a Huffman tree for "BOOKKEEPER"

- Walk through the algorithm in lecture :)

Q7: Why can't a tree have 1 non-null child?

- Construct one this way! Can you simplify that tree?

Q8: Huffman tree shape that leads to little compression or lots of?

- Why do we assign different sequence sizes to some characters but not others? How does this relate to the tree shape?

Encoding tree node

```
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;
        // note: ch not used for interior node
    }

    bool isLeaf() {
        return zero == nullptr && one == nullptr;
    }

    char getChar() {
        if (!isLeaf()) {
            error("Interior (non-leaf) node does not have assigned character!");
        }
        return ch;
    }

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

Char that's here (if it's a leaf node), a pointer to the left tree, a pointer to right tree

This creates a leaf node

This creates an interior node

You can call this on a node to get whether it's a leaf node

Safe way to grab the char at a lead node

Utility Functions

`EncodingTreeNode* createExampleTree()`

`void deallocateTree(EncodingTreeNode* t)`

`bool areEqual(EncodingTreeNode* a, EncodingTreeNode* b)`

- Hint: when do you want to compare characters in the nodes? (not always)

As always: test thoroughly :)

Questions about Huffman warmups?

Huffman Coding

- Warmup
- **Decode / decompress**
- Encode / compress

The process to decompress...

The process to decompress...

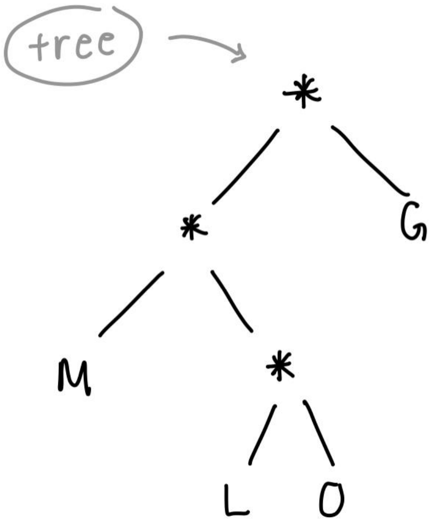
We'll be writing the following functions:

1. `decodeText()`
2. `unflattenTree()`
3. `decompress()`

We recommend you write them in this order.

Decode Text

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

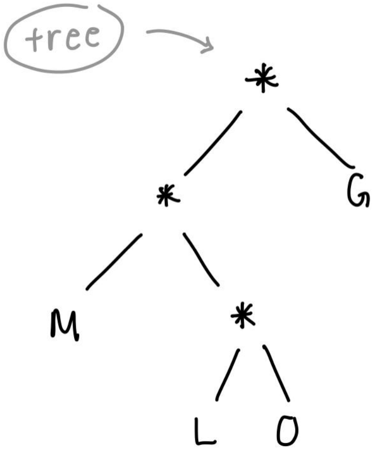


messageBits:

{0, 1, 1, 0, 0, 1}

We want to decode the messageBits into a string → return!

Decode Text



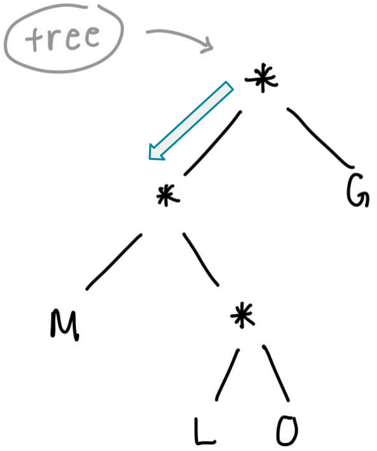
messageBits:

{ $\boxed{0}$, 1, 1, 0, 0, 1 }

We want to decode the message Bits into a string → return!

- Look at message bits one at a time

Decode Text



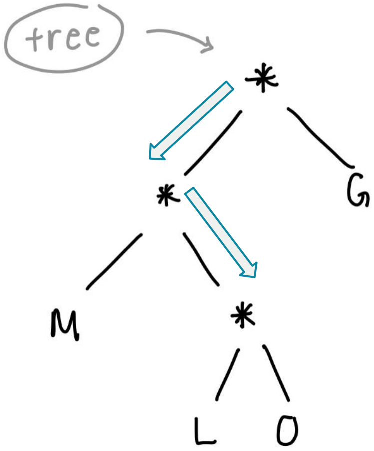
messageBits:

$\{\boxed{0}, 1, 1, 0, 0, 1\}$

We want to decode the message Bits into a string → return!

- Look at message bits one at a time
- Depending on the bit we read, we either…
  - Go down the left
  - Go down the right

Decode Text

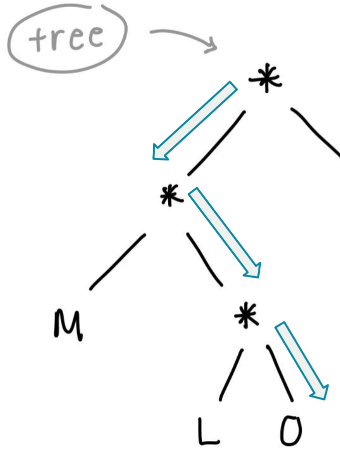


messageBits:

{ 0, $\boxed{1}$, 1, 0, 0, 1 }

We want to decode the message Bits into a string → return!

- Look at message bits one at a time
- Depending on the bit we read, we either…
  - Go down the left
  - Go down the right



messageBits:

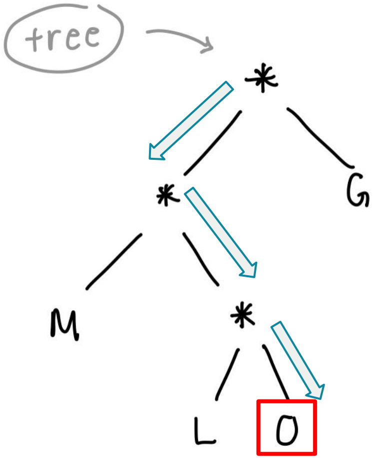
{0, 1, $\boxed{1}$, 0, 0, 1}

We want to decode the message Bits into a string → return!

Decode Text

- Look at message bits one at a time
- Depending on the bit we read, we either…
  - Go down the left
  - Go down the right

Decode Text



messageBits:
{0, 1, $\boxed{1}$, 0, 0, 1}

We want to decode the message Bits into a string → return!

- Look at message bits one at a time
- Depending on the bit we read, we either...
  - Go down the left
  - Go down the right
- If we reach a leaf node, add its character to our result

Decode Text

- Just repeat this strategy until you've looked at all Bits in your Queue of `messageBits`.
- At this point, you should have a resulting string that you can return!

- Note: look through the `EncodingTreeNode` struct to see which method(s) might be able to help us in this function!

Unflatten Tree

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

Given the shape of our tree and every leaf character, return a pointer to the root node of the full tree.

treeShape : { 1, 0, 1, 0, 0 }

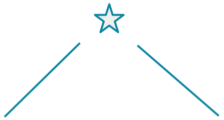
treeLeaves : { 'W', 'A', 'H' }

treeShape : { 1, 0, 1, 0, 0 }

treeLeaves : { 'W', 'A', 'H' }

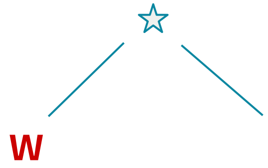
treeShape : $\{\boxed{1}, 0, 1, 0, 0\}$

treeLeaves : {'W', 'A', 'H'}



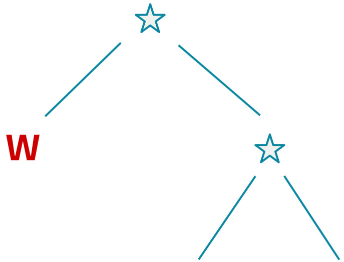
treeShape : { $\boxed{0}$, 1, 0, 0 }

treeLeaves : { '~~W~~', 'A', 'H' }



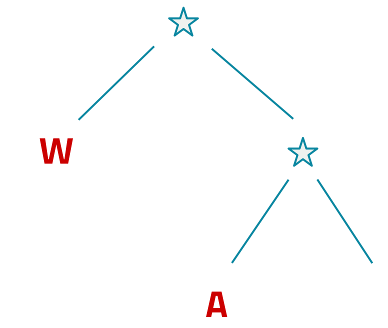
treeShape : { $\boxed{1,}$ 0, 0 }

treeLeaves : { '~~W~~', 'A', 'H' }



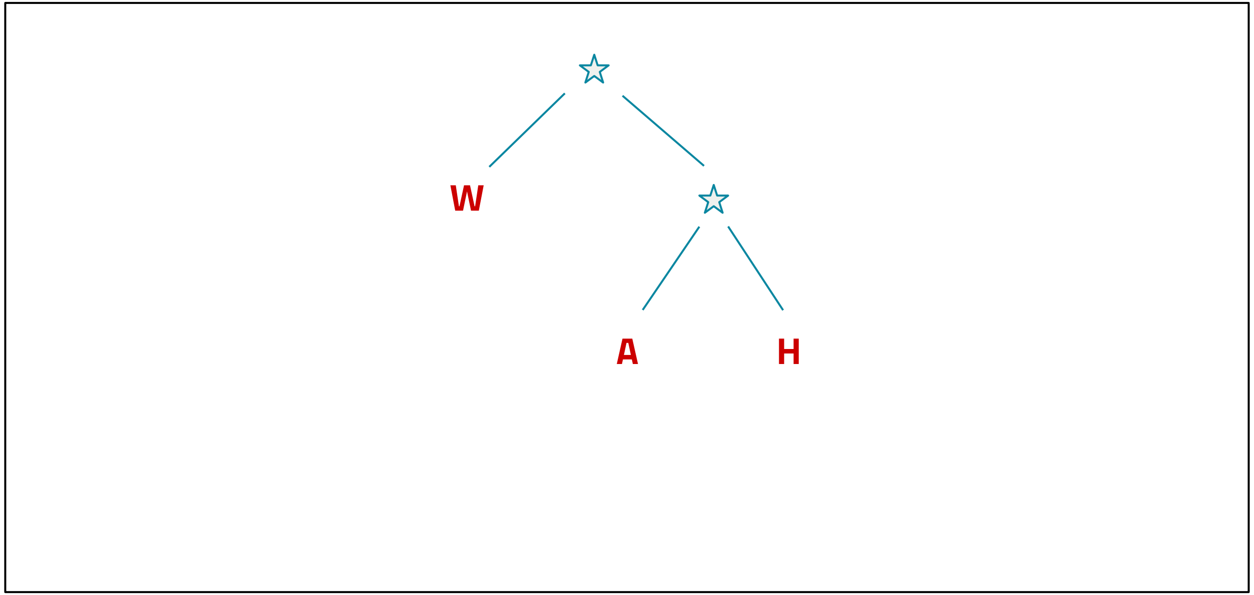
treeShape : { $\boxed{0}$, 0 }

treeLeaves : { ~~'W'~~, ~~'A'~~, 'H' }



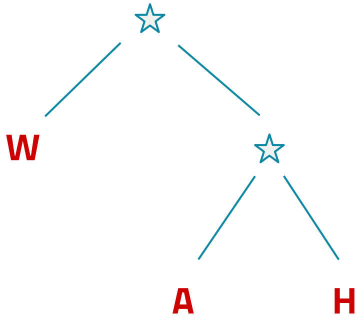
treeShape : { $\boxed{0}$ }

treeLeaves : { ~~'W'~~, ~~'A'~~, ~~'H'~~ }



treeShape : { $\boxed{0}$ }

treeLeaves : { ~~'W'~~, ~~'A'~~, ~~'H'~~ }



Unflatten Tree

Think about how the nature of this task is recursive.

- *At what point can I stop building my tree?*
- *Otherwise, how do I keep branching off my tree?*

This function also involves pondering:

- What do I do when I dequeue a 0?
- What do I do when I dequeue a 1?

Unflatten Tree

This function relies heavily on both constructors in the `EncodingTreeNode` struct.

```
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;
        // note: ch not used for interior node
    }

    bool isLeaf() {
        return zero == nullptr && one == nullptr;
    }

    char getChar() {
        if (!isLeaf()) {
            error("Interior (non-leaf) node does not have assigned character!");
        }
        return ch;
    }

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

Don't you forget about me! ♫

Decompress

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

Takes in encoded data and decompresses it to return the original text!

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

Decompress

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

treeShape : { 1, 0, 1, 0, 0 }
treeLeaves : { 'W', 'A', 'H' }
messageBits : { 0, 1, 0, 1, 1 }

Wow, these Queues look so familiar... I should ~~probably~~ use my helper functions that I've already written to handle these different elements!

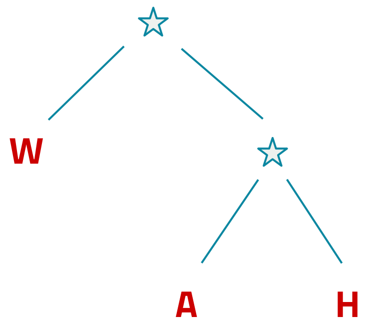
- Just be sure to handle them in the appropriate order!
- && don't forget to clean up when you're done :)

Decompress

treeShape : { 1, 0, 1, 0, 0 }

treeLeaves : { 'W', 'A', 'H' }

messageBits : { 0, 1, 0, 1, 1 }

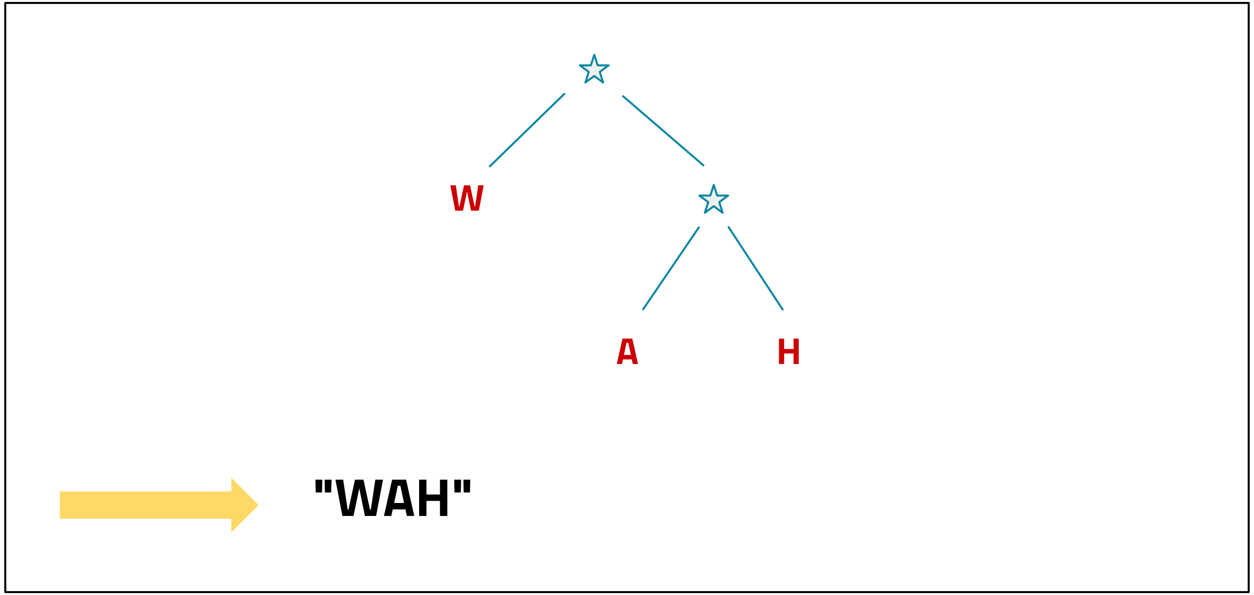


Decompress

treeShape : { 1, 0, 1, 0, 0 }

treeLeaves : { 'W', 'A', 'H' }

messageBits : { 0, 1, 0, 1, 1 }

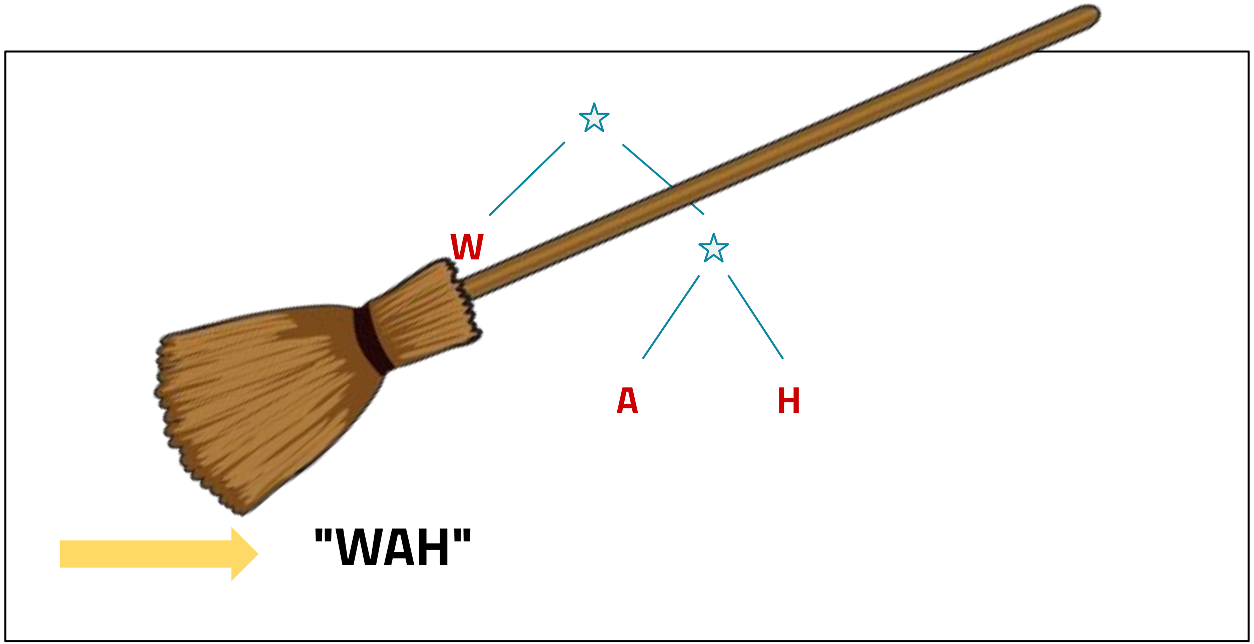


Decompress

treeShape : { 1, 0, 1, 0, 0 }

treeLeaves : { 'W', 'A', 'H' }

messageBits : { 0, 1, 0, 1, 1 }



Questions about decode → decompress?

Huffman Coding

- Warmup
- Decode / decompress
- **Encode / compress**

The process to compress...

Do these functions (in order!)

1. `encodeText()`
2. `flattenTree()`
3. `buildHuffmanTree()`
4. `compress()`

Encode Text

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

Takes in a string and a huffman tree

Returns a Queue<Bit> loaded with the sequence of bits to represent the inputted string

Encode Text

Here, the **Bits mean branches**

1. Traverse the tree (recursively) building a map of char->BitSequence
2. Each time you reach a node, stop:
    a. Recurse on both the zero and one child, adding a 0 or 1 to the sequence respectively
3. Go through given text and using your map, add the bit sequence to your queue for each char!

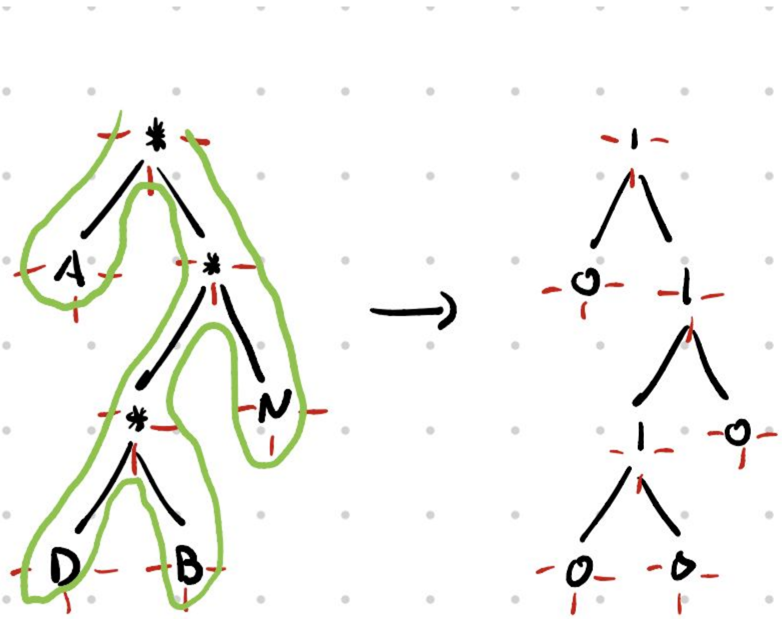
Flatten Tree

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

Given a tree, this produces a pair of `Queue<Bit>` (tree shape) and `Queue<char>` (tree leaves) representing the flattened Huffman tree

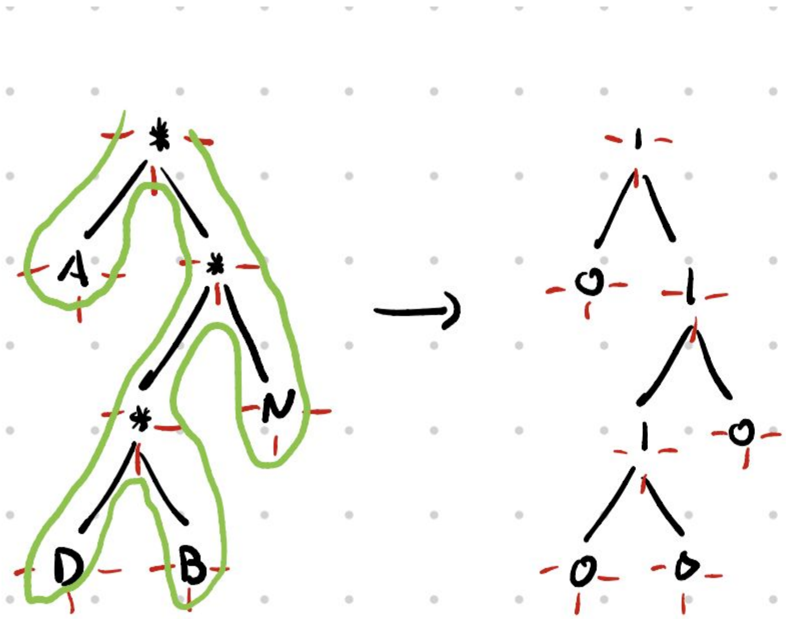
Void function! Doesn't return anything!

Flatten Tree



Traverse your tree using pre-order (left stick) traversal and enqueue to your shape/leaves queue as you go!

Flatten Tree



After this function is done:

Queue<Bit> = 1011000

Queue<char> = ABDN

Build Huffman Tree

Takes in a string, and returns the root of a huffman tree representing that string! Super cool.

Build Huffman Tree

1. Build a ***frequency map*** of the chars in the given string (helper function?)

Build Huffman Tree

1. Build a ***frequency map*** of the chars in the given string (helper function?)
2. For each char in the map:
    a. Enqueue it as a EncodingTreeNode into a ***priority queue*** with that char’s freq as the priority weight

Build Huffman Tree

1. Build a ***frequency map*** of the chars in the given string (helper function?)
2. For each char in the map:
    a. Enqueue it as a EncodingTreeNode into a ***priority queue*** with that char's freq as the priority weight
3. Rebuild the pqueue with *just parents* until it is just one *mega parent*

Build Huffman Tree

Step 3 explained:

- Put all the nodes in a priority queue by frequency
- While there is more than one node in the queue:
  - Dequeue the first two nodes
  - Create a **new** node with the sum of their frequencies
  - Reinsert the new node into the pqueue

Compress

Putting everything together!!

Keep in mind what each of your functions do:

- `buildHuffmanTree()` takes in a string, returns the tree
- `encodeText()` takes in a string and a tree, and returns a Queue<Bits>
- `flattenTree()` takes in the tree and fills the two empty queues

Don't forget to deallocate your tree when you no longer need it!

Compress

Takes in a `string messageText`, returns `EncodedData` for that string!

Strategy:

1. Build the huffman tree for messageText
2. Encode the messageText based on the Huffman tree you just made, and store the result in a queue of Bits
3. Flatten the Huffman tree you built and store the result in a queue of chars and a queue of Bits

Questions about encode → compress?

Encode a message for your SL!

To wrap up our final assignment of the quarter, compress a **secret message** to send to your SL!

Use your program that you just spent hours making!! It works!! And it's awesome!!

Final questions?


You
CS106B Huffman