Data Structures Lab

Coding in a Huff
Assignment 3 Recap

● Assignment 3 grades returned
  ○ Improvement over assignment 2
  ○ Average grade: 80s

● Some issues with balancing
  ○ Lots of balance cases to check

● Some issues with efficiency
  ○ Computing height recursively

● Nice job on a hard assignment
Assignment 3 Recap

- Implementing destructor is encouraged
  - `BalancedTree::~BalancedTree()`

- Calling destructor explicitly is not
  - `tree.~BalancedTree()`

- Destructor will be called automatically
  - When you delete an object
  - When an object goes out of scope
Assignment 4

- Due Friday, March 9
  - Tomorrow night

- Two components
  - Implement Templated Heap
  - Implement Huffman encoding

- Components should be standalone (mostly)
  - Heap should not know about huffmanNodes
  - Huffman code should not be accessing heap innards
Assignment 4 - Pitfalls

- Your heaps will be templated with a pointer value
  - `Heap<int*>`
  - `Heap<Node*>`

- Elements in your array will be pointers
  - `T* heapArray`

- Dereference your pointers before you use them
  - `( *heapArray[i] < *heapArray[j] )`

- There's a subtle syntactic bug here...
Assignment 4 - Pitfalls

- How does C++ parse `* heapArray [ i ]`?
  - `(* heapArray)[i]`
  - `*(heapArray[i])`

- These are both syntactically correct
  - What does each do?
  - Which one do we want?
Any Heap Questions?
Huffman Encoding Examined

**Frequency Map**
- a => 37
- b => 28
- c => 13
...

**Huffman Nodes**
- a, 37
- b, 28
- c, 13

**Heap<Node*>**

**Huffman Tree**

**Code Map**
- a => 000
- b => 010
- c => 101
...

Huffman Encoding Examined

- What structures do we need?
  - Heap - to perform algorithm
  - Map - to store frequencies and codes
  - Nodes - to build Huffman Tree

- You only need to implement two of these
  - But you're fully capable of implementing all three...
Huffman Encoding Examined

- We've talked a lot about heaps
  - Let's look at Huffman Nodes...

- What should they store?

- What methods should they support?
Huffman Encoding Examined

- We've talked a lot about heaps
  - Let's look at Huffman Nodes...

- What should they store?
  - character
  - frequency
  - left/right Node pointers

- What methods should they support?
  - merge
  - comparison
C++ Spotlight - Operator Overloading

● What happens when we use an operator in C++?

● It depends on context
  ○ 0<<1
  ○ cout<<1

● Under the surface, operators are just function calls
  ○ 0.operator<<(1)
  ○ cout.operator<<(1)

● Each class can redefine its operators
  ○ Foo Foo::operator<<(Foo& other)
C++ Spotlight - Operator Overloading

- C++ translates `a+b` into "a.operator+(b)"
  - We can write a operator+ method for a's class
  - `T T::operator+(T& other)`

- Similar for other operators
  - `bool T::operator==(T&)` //equality operator
  - `T T::operator++()` //pre-increment operator
  - `T T::operator new(size_t)` //new operator??

- Want to know what else you can overload?
C++ Spotlight - Operator Overloading

- How might we compare two nodes?

```cpp
class Node{
    int data;

public:
    bool operator<(Node& other){
        bool return data < other.data;
    }
};
```
Huffman Encoding Examined

Frequency Map

- a => 37
- b => 28
- c => 13
- ...

Huffman Nodes

- a, 37
- b, 28
- c, 13

Heap<Node*>

Huffman Tree

Code Map

- a => 000
- b => 010
- c => 101
- ...

Heap<Node*>
Huffman Encoding Examined

- So we've produced a Huffman Tree
  - Now what?

- Need to traverse through tree to produce codes
  - What will that look like?
Huffman Encoding Examined

- So we've produced a HuffmanTree
  - Now what?

- Need to traverse through tree to produce codes
  - What will that look like?

- `getCodes(Node* curr, string code, map& codes)`
  - `getCodes(curr->left, code+'0', codes)`
  - `getCodes(curr->right, code+'1', codes)`

- Don't forget your base cases!
We can use a code map to encode our message.

- How do we decode it?
Huffman Encoding Examined

- We can use a code map to encode our message
  - How do we decode it?

- Use the Huffman Tree!
  - Traverse as we read bits
  - Characters are at leaves!
Huffman Encoding Examined

• Input
  ○ A string of characters ending in '.'

• Output
  ○ The Huffman encoded string (as a binary string)
  ○ The decoded string (should look a lot like the original)
  ○ The amount of space saved
Huffman Encoding Examined

- Input
  - Four score and seven.

- Output
  - 01000101001011000111010101011100110...
  - Four score and seven.
  - 0.4375
Huffman Encoding Examined

● Input
  ○ Four score and seven.

● Output
  ○ 0100010100101100001110101011100110...
  ○ Four score and seven.
  ○ 0.4375

● But wait, that output looks a lot longer than the input!
  ○ Where does our compression come from?
Huffman Encoding Examined

- Chars take up more space than bits
  - encoded in 8 bits (1 byte)
  - Strings take up 8 times more space than bits!

- How can we use this to our advantage?
  - Break up binary string into 8 bit chunks
  - Cast chunks to chars
  - Write those chars out to file

- File will read like gibberish
  - Binary file
  - But we can decode it
Huffman Encoding Examined

A = 0
B = 110
C = 10
D = 111

A A A A B C C C D
0 0 0 0 110 10 10 10 111
00001101 01010111
13 87
Huffman Encoding Examined

● What if our binary string length isn't divisible by 8?
  ○ We could pad the string...
  ○ But then we wouldn't know where to stop decoding

● Use some unique end of string character
  ○ For this assignment, all inputs end with a period
  ○ You could pick your own, though
Huffman Code Standardization

- One set of character frequencies
  - Many possible Huffman encodings...

- Extra rules for standardizing codes
  - When merging two nodes, smaller node goes to the left
  - Store smaller node's char in newly merged node
  - If two nodes have the same frequency, order by char

- Everyone should end up with the same codes
  - Same string encodings
  - Check against my examples
Assignment 4 Tips

● Extra Credit - Implementing file compression
  ○ Includes all of the above

● Come up with a convenient dictionary encoding
  ○ Could store character frequencies instead
  ○ Easy to recreate Huffman Tree

● Make sure you know where your file ends
  ○ Special EOF character

● Write up some minimal documentation
  ○ Let me know how to run your code!