Project Four: Binary Tree

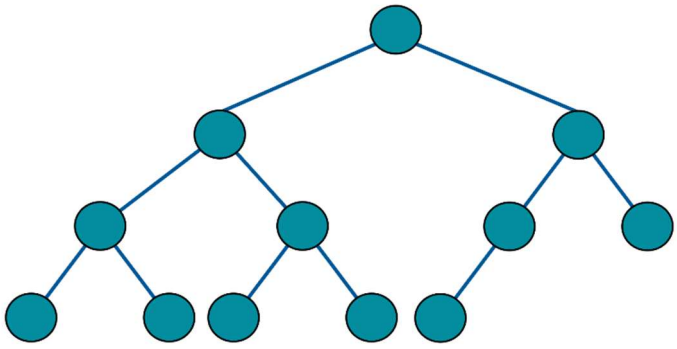
Out: July 7th, 2020; Due: July 21st, 2020

I. Motivation

This project will give you experience in writing recursive functions that operate on a recursively-constructed data structure — binary tree. Also, you will practice using file streams, ADTs and dynamic memory. Besides, you will learn about Huffman coding, which is an interesting application of binary trees to file compression.

II. Introduction

Binary tree is a data structure that is commonly used in computer science. A typical binary tree structure is shown in the picture below:



https://en.wikipedia.org/wiki/Binary_tree

As the picture shows, each circle in the tree is a "tree node". A tree node contains its own information, as well as "links" (pointers) to its **child nodes** (nodes linked below it). As the name of "binary tree" suggests, one tree node can have **at most** two children, namely, the left child and the right child. Notice that a node does not contain any information or link to its parent node (node linked above it). Therefore, the only way to search for a node from a binary tree is to visit nodes recursively from the **root node** (the top node in the tree).

Binary trees are often useful for searching and sorting, and you are going to learn more about it in Ve281. In this project, you will get familiar with binary trees by implementing some basic

operations on them. Also, you will explore a more interesting application of binary tree -- the Huffman Coding.

III. Programming Assignment

This project is divided into two parts. The first part is to implement various member functions of the given *binaryTree* class using recursive methods, and the second part is to apply what you have implemented in part one to simulate file compression and decompression using the Huffman Coding method. The header file of the *binaryTree* class and the *huffmanTree* class is provided on Canvas. Please **do not modify them in any way**. For this project, the penalty for code that does not compile will be **severe**, regardless of the reason. Also, please pay attention to the memory management. If your program leads to memory leak in the test cases, the deduction will be severe.

Part I. Binary Tree

In this part, you will need to implement various functions specified in `binaryTree.h`. Write your code in a file named `binaryTree.cpp`.

Here we have a more general binary tree type. Each tree node is defined as a class as follows:

```
class Node {
    // A node in a binary tree
    std::string str;
    int num;
    Node *left;         // A pointer to the left child of this node.
                        // If it does not exist, left should be NULL
    Node *right;        // A pointer to the right child of this node.
                        // If it does not exist, right should be NULL
public:
    Node(const std::string &str, int num, Node *left = nullptr, Node
*right = nullptr);
    // REQUIRES: The input left and right each points to a dynamically
    //           allocated node object, if not being NULL.
    // MODIFIES: this
    // EFFECTS:  Constructs a node with given input values.

    Node *leftSubtree() const;
    // EFFECTS:  Returns the pointer to the left child of the node.

    void setleft(Node *n);
    // MODIFIES: this
    // EFFECTS: Sets the left child of the node to be n.

    Node *rightSubtree() const;
```

```
    // EFFECTS:  Return the pointer to the right child of the node.

    void setright(Node *n);
    // MODIFIES: this
    // EFFECTS: Sets the right child of the node to be n.

    std::string getstr() const;
    // EFFECTS:  Returns the "str" component of the node.

    int getnum() const;
    // EFFECTS:  Returns the "num" component of the node.

    void incnum();
    // MODIFIES: this
    // EFFECTS: Increments num by 1


    Node *mergeNode(Node *leftNode, Node *rightNode)
    // REQUIRES: leftNode and rightNode points to dynamically
    //           allocated node objects.
    // EFFECTS: Returns a pointer to a node with "str" being the
    //          result of appending leftNode->str and rightNode->str,
    //          and "num" being leftNode->num + rightNode->num. Also,
    //          please allocate memory for this returned node object.
    //
    //          For example, if leftNode->str = "a", and
    //          rightNode->str = "b", then the "str" of the merged
    //          node is "ab".

};
```

As shown in the above code, a *Node* contains its own information (a string and an integer), as well as two pointers to its child nodes. You need to implement all the methods of the *Node* class .

Make sure that **all the nodes are dynamically allocated**.

Then you are going to implement the methods of the *binaryTree* class using *Node* and the above functions. Before you do it, you may want to know some concepts:

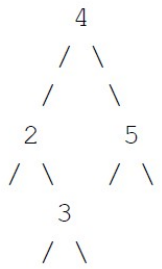
- Traversal: Visit all the nodes in a binary tree. In this project, you will implement 3 kinds of traversal methods. When visiting each node, you are required to print the "num" or "str" component of it (see comments in the given functions).
  a) Pre-order traversal:
     1. Access the data part of the current node.
     2. Traverse the left subtree by recursively calling the pre-order function.
     3. Traverse the right subtree by recursively calling the pre-order function.

b) In-order traversal:
   1. Traverse the left subtree by recursively calling the in-order function.
   2. Access the data part of the current node.
   3. Traverse the right subtree by recursively calling the in-order function.

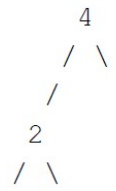
c) Post-order traversal:
   1. Traverse the left subtree by recursively calling the post-order function.
   2. Traverse the right subtree by recursively calling the post-order function.
   3. Access the data part of the current node.

- Covered by: We can define a special relation between trees "is covered by" as follows:
  a) An empty tree is covered by all trees.
  b) The empty tree covers only other empty trees.
  c) For any two non-empty trees, A and B, A is covered by B if and only if the root's "num" component (we only consider "num" for simplicity) of A and B are equal, the left subtree of A is covered by the left subtree of B, and the right subtree of A is covered by the right subtree of B.

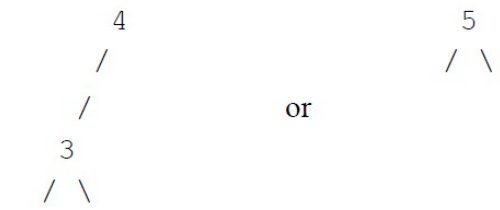
  For example, the tree (only "num" components are shown):



  covers the tree



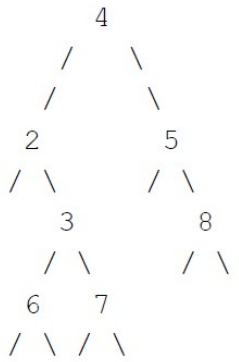
  but not the trees



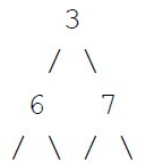
- Contained by: With the definition of "covered by", we can define a relation "contained by". A tree A is contained by a tree B if

a) A is covered by B, or,
b) A is covered by a subtree of B.

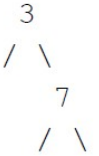
Note that in the above definitions, a **subtree** of a tree T is an empty tree or a non-empty tree composed of a node S in T together with all downstream nodes of the node S in T (called the descendants of S in T). For example, for the tree T (only "num" components are shown)



the tree



is a subtree of T. However, the tree



is not.

Following is the class you need to implement:

```
class BinaryTree {
    // A binary tree object

    Node *root;          // Root node of the binary tree
```

```
public:
    BinaryTree(Node *rootNode = nullptr);
    // REQUIRES: The input rootNode points to a dynamically allocated
    // node object, if not being NULL.
    // MODIFIES: this
    // EFFECTS: Constructs a binary tree with a root node.

    ~BinaryTree();
    // MODIFIES: this
    // EFFECTS: Free all the memory allocated in this binary tree.

    std::string findPath(const std::string &s) const;
    // EFFECTS: Returns the path from the root node to the node with
    //          the string s. The path is encoded by a string only
    //          containing '0' and '1'. Each character, from left to
    //          right, shows whether going left (encoded by '0') or
    //          right (encoded by '1') from a node can lead to the
    //          target node. For example, we want to find "g"
    //          in the following tree (only the "str" components are
    //          shown):
    //
    //
    //
    //                          "a"
    //                         /   \
    //                        /     \
    //                      "b"     "c"
    //                      / \     / \
    //                        "d"     "e"
    //                        / \     / \
    //                      "f" "g"
    //                      / \ / \
    //
    //          The returned string should be "011". (Go left from
    //          "a", then go right from "b", and finally go right
    //          from "d" can lead us to "g".)
    //
    //          If s is in root node, then return an empty string.
    //          If s is not in the tree, then return "-1".
    //          You can assume that the "str" components of all the
    //          nodes are unique in a tree.

    int sum() const;
    // EFFECTS: Returns the sum of "num" values of all nodes in the
    //          tree. If the tree is empty, return 0.

    int depth() const;
    // EFFECTS: Returns the depth of the tree, which equals the number
    //          of layers of nodes in the tree. Returns zero if the
    //          tree is empty.
    //
    // For example, the tree
```

```
//
//                            a
//                          /   \
//                         /     \
//                        b       c
//                       / \     / \
//                          d       e
//                         / \     / \
//                        f   g
//                       / \ / \
//
// has depth 4.
// The node a is on the first layer.
// The nodes b and c are on the second layer.
// The nodes d and e are on the third layer.
// The nodes f and g are on the fourth layer.

void preorder_num() const;
// EFFECTS: Prints the "num" component of each node using a pre-
//          order traversal. Separate each "num" with a space.
//          A pre-order traversal prints the current node first,
//          then recursively visits its left subtree, and then
//          recursively visits its right subtree and so on, until
//          the right-most element is printed.
//
//          For any node, all the elements of its left subtree
//          are considered as on the left of that node and all
//          the elements of its right subtree are considered as
//          on the right of that node.
//
//          For example, the tree (only "num" components are
//          shown):
//
//                            4
//                          /   \
//                         /     \
//                        2       5
//                       / \
//                      7   3
//                         / \
//                        8   9
//
//          would print 4 2 7 3 8 9 5
//
//          An empty tree would print nothing.

void inorder_str() const;
// EFFECTS: Prints the "str" component of each node using an in-
//          order traversal. Separate each "str" with a space. An
//          in-order traversal prints the "left most" element
//          first, then the second-left-most, and so on, until
//          the right-most element is printed.
```

```
//
//          For any node, all the elements of its left subtree
//          are considered as on the left of that node and
//          all the elements of its right subtree are considered
//          as on the right of that node.
//
//          For example, the tree (only "str" components are
//          shown):
//
//                          "a"
//                         /   \
//                        /     \
//                     "bb"    "ddd"
//                     /  \
//                  "e"   "c"
//
//          would print e bb c a ddd
//
//          An empty tree would print nothing.

void postorder_num() const;
// EFFECTS: Prints the "num" component of each node using a post-
//          order traversal. Separate each "num" with a space.
//          A post-order traversal recursively visits its left
//          subtree, and then recursively visits its right subtree
//          and then print the current node.
//
//          For any node, all the elements of its left subtree
//          are considered as on the left of that node and
//          all the elements of its right subtree are considered
//          as on the right of that node.
//
//          For example, the tree (only "num" components are
//           shown):
//
//                            4
//                           /   \
//                          /     \
//                         2       5
//                        / \
//                       7   3
//                          / \
//                         8   9
//
//          would print 7 8 9 3 2 5 4
//
//          An empty tree would print nothing.

bool allPathSumGreater(int sum) const;
// REQUIRES: The tree is not empty
//
// EFFECTS: Returns true if and only if for each root-to-leaf path
```

```
    //          of the tree, the sum of "num" of all nodes along the
    //          path is greater than "sum".
    //
    //          A root-to-leaf path is a sequence of nodes in a tree
    //          starting with the root element and proceeding downward
    //          to a leaf (an element with no children).
    //
    //
    // For example, the tree (only the "num" components are shown):
    //
    //                            4
    //                          /   \
    //                         /     \
    //                        1       5
    //                       / \     / \
    //                      3   6
    //                     / \ / \
    //
    // has three root-to-leaf paths: 4->1->3, 4->1->6 and 4->5.
    // Given the input sum = 9, the path 4->5 has the sum 9, so the
    // function should return false. If the input sum = 7, since all
    // paths have their sums greater than 7, the function should
    // return true.

    bool covered_by(const BinaryTree &tree) const;
    // EFFECTS: Returns true if this tree is covered by the input
    //          binary tree "tree" (only consider the "num"
    //          components).

    bool contained_by(const BinaryTree &tree) const;
    // EFFECTS: Returns true if this tree is contained by the input
    //          binary tree "tree" (only consider the "num"
    //          components).

    BinaryTree copy() const;
    // EFFECTS: Returns a copy of this tree. Hint: use deep copy.
};
```

We will test all those methods implemented by you via Online Judge. You are encouraged to write test files and play with the binary tree on your own.

Important: Each function definition in your `binaryTree.cpp` should be **no longer than 10 lines** (function body)**.** You are required to use recursion to write all the functions, except

1. Functions within the Node class.
2. `BinaryTree(Node *rootNode = nullptr);`

If you use recursion for the implementation, the code can be very short. So if you write functions longer than 10 lines, there will be points deducted. You are welcomed to use helper function for the recursion.

Part II. Huffman Coding

Huffman Coding is a commonly used method to compress data. In this part, you will use some data types and functions inherited from `binary.h` to build a Huffman tree, and use that tree to convert each character in a file to the corresponding binary code. We present next how the Huffman coding works:

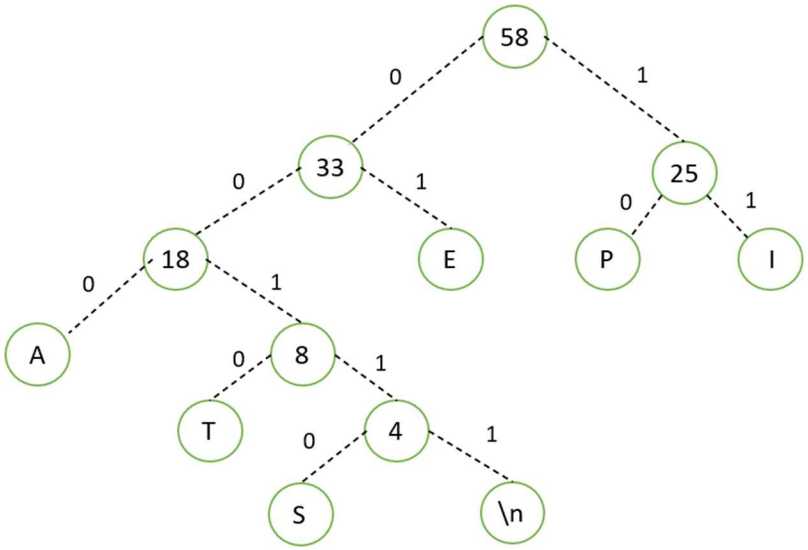
What is Huffman coding:

Huffman coding is a method to compress files. As a simple example, suppose that you have a text file containing only the characters in the following table:

Character	Code	Frequency	Total bits
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	12
T	100	4	12
P	101	13	39
\n	110	1	3

Suppose your computer uses a 3-bit code to encode each character, and the number of occurrences of each character in the file is shown in the "Frequency" column. The total bits needed for storing each character is just 3 times its frequency. Therefore, the whole file will occupy 174 bits (sum the number of bits for each character).

However, we know it is common that some characters appear very often in a file, while some characters only appear a few times. With that property, we can build a binary tree called Huffman tree, to re-encode the characters based on their frequencies (we explain later how to build a Huffman tree):



Now the new code of A is obtained by finding the path from the root node to 'A' in the Huffman tree. As done in Part I, going left is encoded by '0' and right by '1'. In the example, the path for 'A' is encoded by '000', which provides its new code. After searching for all the characters, we can obtain a new table:

Character	Code	Frequency	Total bits
A	000	10	30
E	01	15	30
I	11	12	24
S	00110	3	15
T	0010	4	16
P	10	13	26
\n	00111	1	5

Ideally, characters with higher frequency are assigned shorter codes, while characters with lower frequency are assigned longer codes. If we sum up the total bits for each character again, we can see that the new size of the file is only 146 bits. So now the file has been successfully compressed.

The example is adapted from https://www.youtube.com/watch?v=dM6us854Jk0 according to our building rules, which will be elaborated later. You can have a look to obtain a better understanding.

Steps to build a Huffman tree:

1. Read a file. Note that for simplicity, you can assume that all test files only contain

- lower case letters ('a' ~ 'z'),
- empty spaces (' '),
- end of line ('\n'), and there will always be one '\n' in the end of the file.

2. Count the frequency for each character (how many times this character appears in the file).
3. Build a node for each character and add all nodes to an array. The "str" component should contain the character and the "num" component should be the frequency.
4. While there are more than one node in the array:
   - Remove two nodes with smallest frequency, which is calculated in step 2.
     - If there are more than two candidates, choose those whose first character has **smaller ASCII value**. For example, if "a", "b", "c" all have the frequency 2, and 2 is the smallest frequency, then choose nodes whose "str" components are "a" and "b" in this case.
   - Merge the two nodes into one node, called a parent node.
     - The node with smaller frequency should be put as the right child of the parent node. The node with larger frequency should be put as the left child of the parent node. If their frequencies are the same, the node whose first character of the "str" component has smaller ASCII value should be put on the right. For example, if both "a" and "b" have a frequency of 2, then choose the node corresponding to "a" as the right child.
     - "Merge" means the "str" component of the parent node is the concatenation of the "str" components of its two children, with the value of the left child first. And the "num" component of the parent node is the sum of the "num" component of its two children.
   - Put the parent node back into the array.
5. After the while loop, there will be only one node left in the array. This is the root node of our final Huffman Tree.

The HuffmanTree type:

We have defined a derived class `HuffmanTree` from the class `BinaryTree` for you in `HuffmanTree.h` as well as the implementation in `HuffmanTree.cpp`. You can use the functions declared in `HuffmanTree.h` directly in your code. What they do is mainly constructing a Huffman tree from a file and printing a Huffman tree. You do not need to understand the details of the implementation of these functions.

```
class HuffmanTree : public BinaryTree {
    // Huffman tree
```

```
public:

    HuffmanTree(Node *rootNode = nullptr);
    // REQUIRES: The input rootNode points to a dynamically allocated
    //           node object, if not, it is NULL.
    // MODIFIES: this
    // EFFECTS: Constructs a huffman tree with a root node.

    HuffmanTree(const std::string &treefile);
    // MODIFIES: this
    // EFFECTS: Constructs a huffman tree from the treefile. (treefile
    //          corresponds to the file name.) The treefile saves all
    //          the node information needed for a huffman tree. You do
    //          not need to understand the details of the
    //          implementation of this function.
    //
    // In a huffman tree, all the leaf nodes contains information of
    // a character. You can find the encoding for each character just
    // by the path from root node to it. So only the "str" components
    // of leaf nodes need to be save in the treefile (we do not care
    // about its "num" when doing compression or decompression). For
    // nodes other than leaf nodes, they contain information about
    // frequencies. So we only care about the "num" components of
    // those nodes. Thus a huffman tree can be saved in a file like
    // the following example:
    //
    // 8,
    // a,4,
    // -,-,b,c,
    //
    // which represents the tree
    //
    //           8
    //         /   \
    //        a     4
    //       / \   / \
    //           b   c
    //          / \ / \
    //
    // Each '-' in the file means there is no node present in that
    // position.
    //

    void printTree();
    // EFFECTS: Prints the huffman tree following the format explained
    //          above. You do not need to understand the details of
    //          the implementation of this function.

};
```

Compress a file:

Write your implementation for compressing a file in `compress.cpp`. First, you will be given a file name in a program argument (the arguments will always be valid and `argc` will either be 2 or 3) so that you can read the file. The command will be either

```
./compress <filename>
```

or

```
./compress -tree <filename>
```

Then, use the procedure shown previously to build a Huffman tree. Once you have the Huffman tree, you should be able to find the binary encoding of each character using the methods of *BinaryTree* (inherited to *HuffmanTree*) you implemented. Then you have a table of the character encoding.

Finally, you can process the original file again and map each character to the corresponding binary code.

Requirements:

1. If the command is

```
./compress -tree <filename>
```

then you should **only** print the tree to stdout using the `printTree()` function from the `HuffmanTree` class.

2. If the command is

```
./compress <filename>
```

you should **only output (using `cout`) the binary code of each character in the same sequence as the original file, separated by one space.**

For example, the original file contains “ab\n”, and you encode ‘a’ as 00, ‘b’ as 01, ‘\n’ as 10, then your output should be

```
00 01 10
```

Note that the spaces for separating the binary codes and the spaces in the original file are different. You need to encode the spaces in the original file as well.

You can test your code using a sample file `textfile.txt` in the starter files.

Decompress a file:

Decompressing is very similar to compressing. You should build a Huffman Tree from the given tree file, which will have the same format as the output of the **`printTree()`** function .You can use the constructor of the *HuffmanTree* with `string` as the input (name of tree file) to construct the tree. Write your code in `decompress.cpp`.

You will be given two file names as argument (the argument will always be valid, and `argc` will always be 3), e.g.

```
./decompress <treefilename> <binaryfile>
```

You should use the file whose name is the first argument after "`./decompress`" (the `<treefilename>`) to build the Huffman Tree. Then, use the tree that you just built to translate the binary file (the second argument after "`./decompress`"). **Output (using `cout`) the translated characters in the same sequence as the binary code. No extra space separation should be added between the translated characters.**

For example, the binary file contains "00 01 10", and you encode 'a' as 00, 'b' as 01, '\n' as 10, then your output should be

```
ab
```

Note that '\n' is not displayed as `\n` since it is a new-line symbol. Instead, the output changes to a new line.

You can use a sample file `tree.txt` (for building the Huffman tree) and a sample binary file `binary.txt` in the starter files to test your code. The output should be the same as `textfile.txt`.

Also note that you don't need to worry about invalid files, which means if you built your tree successfully, then each binary code in binaryfile will always have its corresponding character from the tree.

Hint:

1. When building the Huffman Tree following the steps, you can consider using `std::sort` (http://www.cplusplus.com/reference/algorithm/sort/) and you should `#include <algorithm>`. Note that the third argument of `std::sort` is a comparator, you can consider function pointer there.
2. If you use the `std::sort` mentioned above, don't forget to resort the array after you put the new **parent node** back into the array.
3. If you do not want to use array to store the nodes when building Huffman tree, you may use `vector` instead.

IV. Implementation Requirements and Restrictions

1. When writing your code, you may use the following standard header files: `<iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<cstdlib>`, `<vector>` and `<algorithm>`. No other header files can be included.
2. Please do not modify `binaryTree.h`, `huffmanTree.h and huffmanTree.cpp`.
3. All required output should be sent to the standard output stream; none to the standard error stream.

V. Source Code Files and Compiling

To compile, you should have `binaryTree.h`, `binaryTree.cpp`, `huffmanTree.cpp`, `huffmanTree.h`, `compress.cpp` and `decompress.cpp` in your directory. Use the following Linux command to compile:

```
g++ --std=c++17 -o compress compress.cpp binaryTree.cpp
huffmanTree.cpp

g++ --std=c++17 -o decompress decompress.cpp binaryTree.cpp
huffmanTree.cpp
```

In order to guarantee that the TAs can compile your program successfully, you should name you source code files exactly like how they are specified above. For this project, the penalty for code that does not compile will be **severe**, regardless of the reason.

VI. Testing

We have provided you three files: `textfile.txt`, `tree.txt` and `binary.txt`. If you compress the `textfile.txt`, the output should be the same as what's shown in `binary.txt`. And if you build the Huffman tree using `tree.txt` and try to decompress `binary.txt`, the output should be the same as `textfile.txt`. One thing to remember is that the outputs are sent using `cout` rather than writing them into a file.

Memory leak is going to be tested for this project. You can use `valgrind` to check your memory usage when running the program.

VII. Submitting and Due Date

You should submit three source code files `binaryTree.cpp`, `compress.cpp`, and `decompress.cpp`. (in one compressed file) via Online Judge. The due time is 11:59 pm on July 21st, 2020.

VIII. Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions.