Project Four: Binary Tree

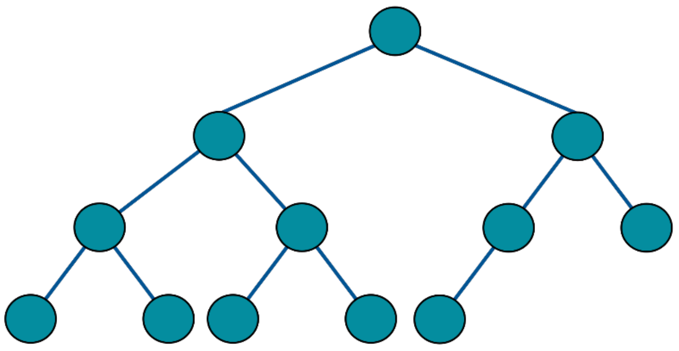
Out: July 9th, 2021; Due: July 20th, 2021

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 compression, which is an interesting application of binary trees.

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 an interesting application of binary tree: dictionary-based compression.

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 dictionary-based compression method. The header file of the *BinaryTree* class is provided on Canvas. Please **do not modify it 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 in **BinaryTree.h**:

You need to implement most of the methods there. We have highlighted such methods in **BinaryTree.h** with a comment like this:

```
// todo: implement this
```

A *Node* has an integer value, 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 its methods. 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 its value.
    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, called "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 value of A and that of 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:

```
    4
   / \
  /   \
 2     5
/ \   / \
   3
  / \
```

covers the tree

```
    4
   / \
  /
 2
/ \
```

but not the trees

```
    4
   /
  /
 3
/ \
```

or

```
 5
/ \
```

- 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

```
     4
   /   \
  /     \
 2       5
/ \     / \
   3       8
  / \     / \
 6   7
/ \ / \
```

the tree

```
   3
  / \
 6   7
/ \ / \
```

is a subtree of T. However, the tree

```
 3
/ \
   7
  / \
```

is not.

You need to implement most of the methods. We have highlighted such methods in **BinaryTree.h** with a comment like this:

```
// todo: implement this
```

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

<u>Important</u>: 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. Constructor of the BinaryTree class.

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.

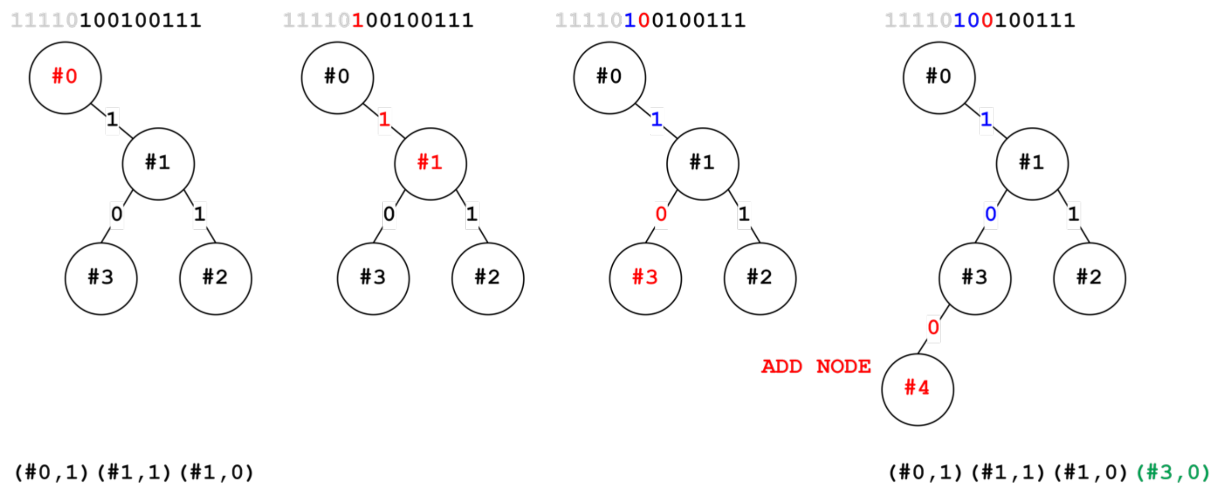
Part II. Dictionary-Based Compression

Information theory developed by Shannon provides the theoretical bound for any lossless compression algorithm. Also, Shannon created a simple coding algorithm called Shannon code which was proved to meet the theoretical bound in terms of mathematical expectation. However, researchers wanted to find an algorithm that could compress and decompress quickly in practice. In 1978, Ziv and Lempel developed an easy but powerful algorithm based on dynamic dictionary and its variation is still used in zip files nowadays. In Part II, we will use our **binary tree** to implement this algorithm.

In this project, for simplicity, we only consider strings with 0s and 1s, such as “`11110100100111`”. To **compress** a string, a binary tree to encode it is built iteratively. A path from the root node in this tree corresponds to a substring: moving to the left child corresponds to 0 while moving to the right child corresponds to 1. See the next figure for an example. The compression algorithm that builds this binary tree and the decompression algorithm are described next.

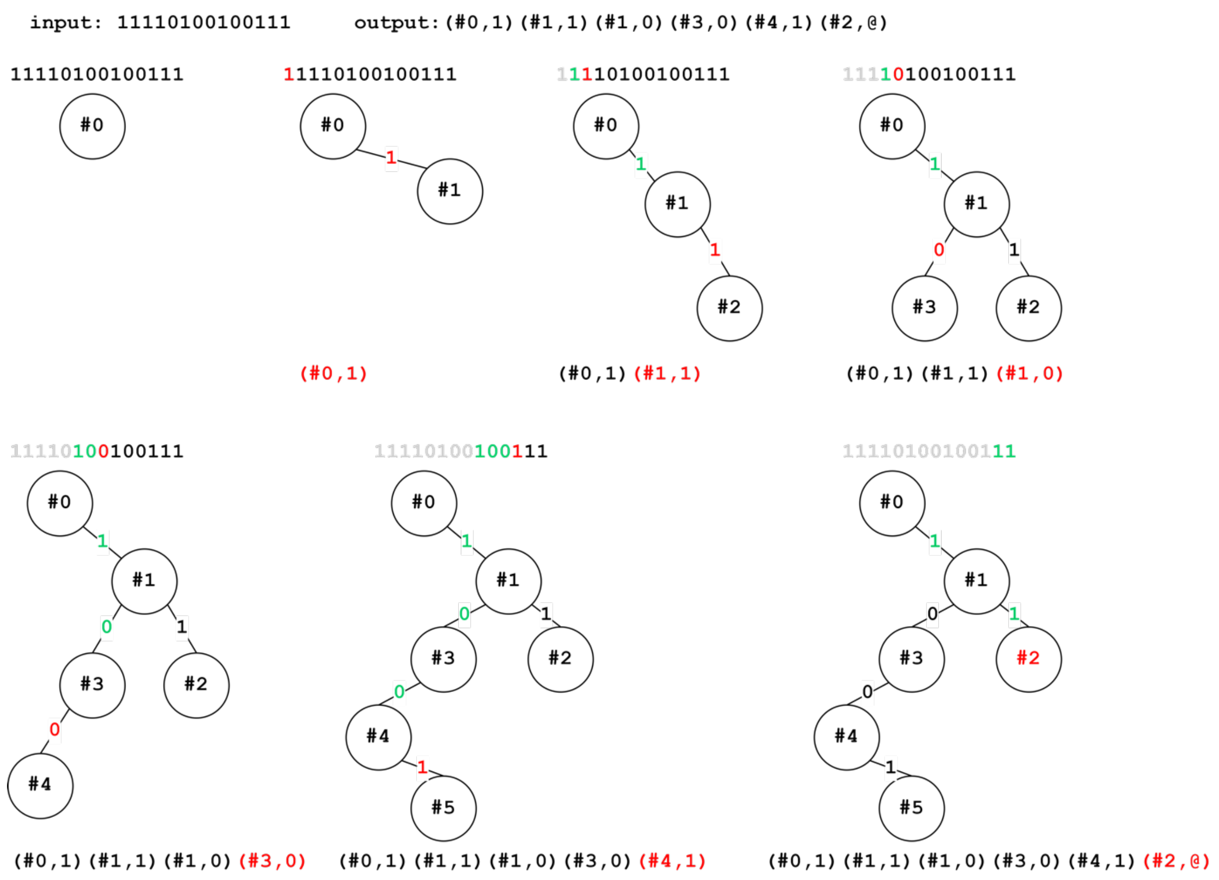
Compression

The compression algorithm starts from a binary tree with only a root node labeled by `#0`. Initially, the whole string to be compressed is marked as unprocessed. At each step, given the current binary tree and the unprocessed part of the string, we first **find the longest substring** that appears in the tree and then update the binary tree with the next extra character from the unprocessed string, if any. See the nextfigure showing an example with input string “`11110100100111`”.



The substring in gray corresponding to "`11110`" is the processed part and we have built the current binary tree (leftmost tree of figure) in the previous steps using that processed part. First, as shown in the figure above, we find **the longest substring that occurs in the tree**by reading the characters of the unprocessed part one by one. The longest substring is `10` marked in blue. Second, we add a new node into the tree by reading the next extra character from the unprocessed part. In this example, this character is `0` marked in red. Therefore, here, we add a left child. Please note that all the nodes are labeled according to the order in which they are added. This node is the fourth node that is added. So, we label it as `#4`. In addition to building this binary tree, the compression algorithm outputs the compressed string. This string is expanded every time a new node is added to the binary tree with a pair, `(node_index, char)`, where `node_index` is the label of the new node's parent and `char` determines whether the node is the left or right child. In this example, the pair is `(#3, 0)` (0 means left and 1 means right).

The figure below gives a complete example illustrating the algorithm for the input string "`11110100100111`". For each step, the green string is the longest substring and the red one is the extra character. For the last part, the remaining string may not be long enough to create a new node. So, we can directly output the node's label corresponding to the remaining string. In this example, it is `(#2,@)`. Here, @ represents empty. If there is no remaining string, just output `(#0, @)`.

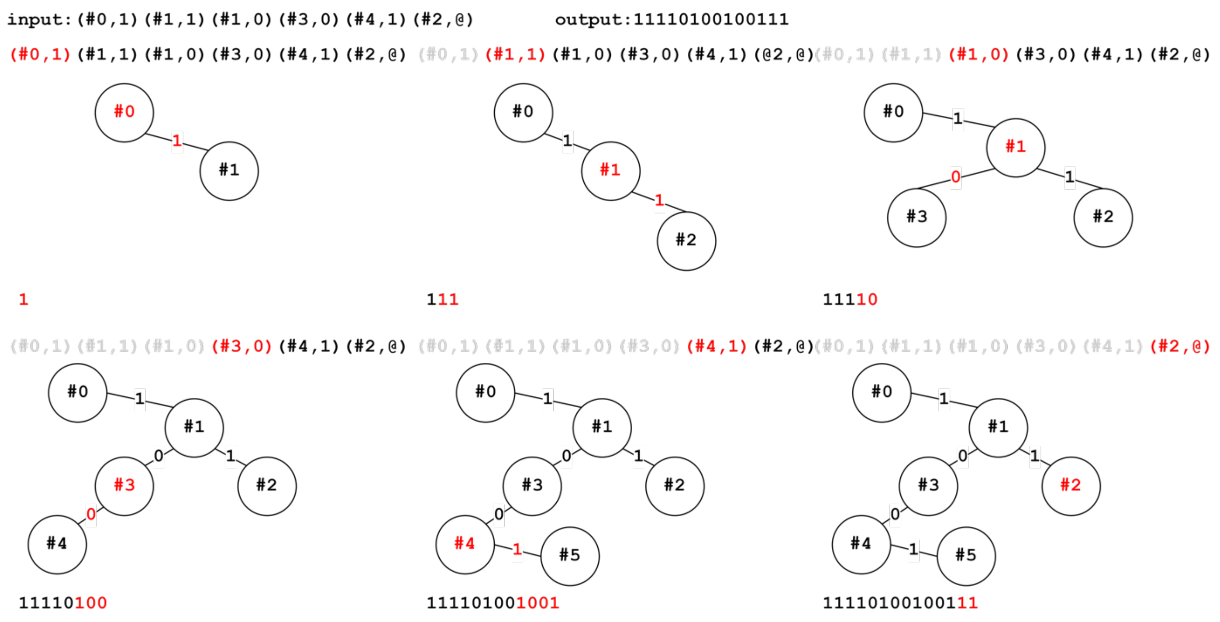


The pseudo code of the compression algorithm is given below:

```
Build a tree with node #0
w ← “”
while (read a character c) {
    w’ ← w + c
    if w’ exists in the tree
        w ← w’
    else
        n ← node corresponding to w
        add a child for n and output the child node information
        w ← “”
}
Output the node corresponding to w
```

Decompression

The procedure for decompression is just the inverse of compression. We need to build the tree according to the pairs `(node_index, char)`. Noting that the pairs are outputted in order, we only need to retrieve the original string by printing the path of the new node in order. The detail is again shown on an example illustrated in the next figure.



The pseudo code of the decompression algorithm is given below:

```
Build a tree with node #0
while (read a pair (node_index, c)) {
   n ← node labeled by parent;
   add a child for n according to c (c is 0 or 1 or @)
   print the path from root to this child;
}
```

<u>Requirement:</u>

Please first read the provided files and try to implement this algorithm. Write your implementation in a file `dbc.cpp`. First, when the option is `-x`, you need to compress the provided file. The command will be

```
./dbc -x <filename>
```

then you should **only** print the output of the compression algorithm to `stdout`.

If the option is `-d`, you need to decompress the provided file:

```
./dbc -d <filename>
```

then you should only print the output of the decompression algorithm to `stdout`.

Hint:

A struct defined in `NodeInfo.h` provides the output and input for this part.Its definition is given below:

```
struct NodeInfo {
    int node_index;
    char c;
}
```

Here `node_index` is the label `node_index` described above. We can use it directly with the standard input stream or output stream. For example, when you want to print the pair (#4, 1), you can write:

```
NodeInfo ni(4, '1');

std::cout << ni;
```

When the last node is (#2, @), you can write:

```
NodeInfo ni(2, '@');

std::cout << ni;
```

As for the input, the method is similar. You can use `input_stream >> ni;` then get information through `ni.node_index` and `ni.c.`

Assumptions:

You can assume all the inputs are correct. There is no empty input. For compression, the characters are only from '0', '1' or '@'. The input of decompression can be recognized by the normal input stream operator provided in `NodeInfo.h`. Also, the program will always receive correct arguments. So, you don't need to check for incorrect arguments.

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`.
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, dbc.cpp, NodeInfo.h` in your directory. Use the following Linux command to compile:

```
g++ --std=c++17 -o dbc dbc.cpp BinaryTree.cpp -Wall -Werror
```

We use some features of C++ 17 to implement debug functions. Be sure to use `--std=c++17` when compiling your program.

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 one file for part one: `test.cpp`. You can test your binary tree with this source file or with methods written in this file. Be aware that this test is not the final test. You need to write your own test cases.

We have provided you two files for part two: `textfile.txt` and `textfile.dbc`. If you compress the `textfile.txt`, the output should be the same as what is shown in `textfile.dbc`. And if you build the dictionary tree using `textfile.dbc` and try to decompress, 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 two source code files `BinaryTree.cpp`, `dbc.cpp` (in one compressed file) via Online Judge. The due time is 11:59 pm on July 20th, 2021.

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.