Analysis of Algorithms II

Homework 1

Oguzhan Karabacak 150170021

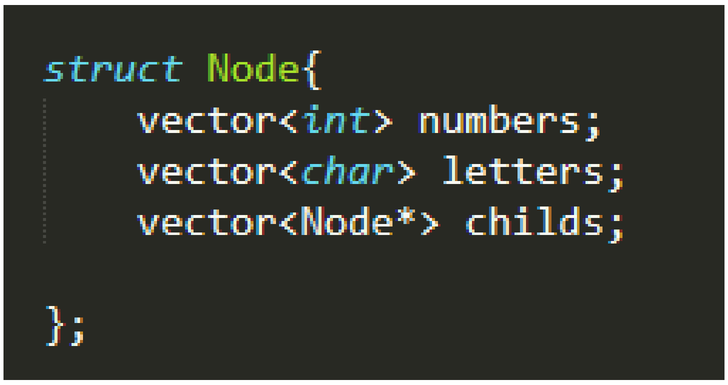
Just command ./hwl TWO TWO FOUR result1 while running the code.

The program adds the .txt extension itself.

1. Report Problem Formulation

a. Node and Assignment Representations in Detail

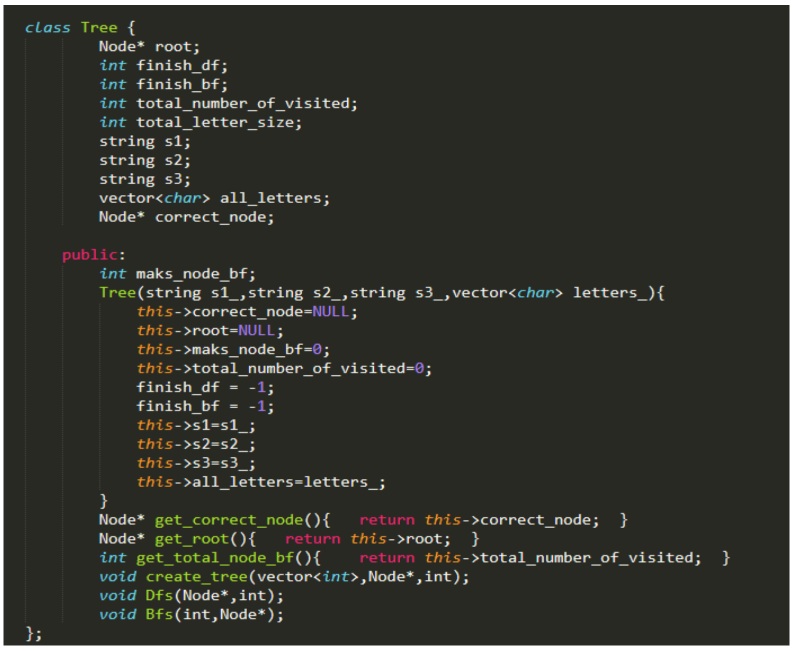
Node Structure



```
struct Node{
    vector<int> numbers;
    vector<char> letters;
    vector<Node*> childs;

};
```

In the Node part, I could define Matrix as a 2-dimensional array.
But since the 2D array is slower in Tree Creation, Breadth First Search and Depth First Search, and the size of these arrays changes in each layer (1x1,2x2,3x3), I used two separate vectors. 'numbers' vector is to store number in each layer and letters is to store letter in each layer.
The 'childs' variable is used to keep the nodes on the next layer.

Tree Class



```
class Tree {
        Node* root;
        int finish_df;
        int finish_bf;
        int total_number_of_visited;
        int total_letter_size;
        string s1;
        string s2;
        string s3;
        vector<char> all_letters;
        Node* correct_node;

    public:
        int maks_node_bf;
        Tree(string s1_,string s2_,string s3_,vector<char> letters_){
            this->correct_node=NULL;
            this->root=NULL;
            this->maks_node_bf=0;
            this->total_number_of_visited=0;
            finish_df = -1;
            finish_bf = -1;
            this->s1=s1_;
            this->s2=s2_;
            this->s3=s3_;
            this->all_letters=letters_;
        }
        Node* get_correct_node(){   return this->correct_node;  }
        Node* get_root(){   return this->root;  }
        int get_total_node_bf(){    return this->total_number_of_visited;  }
        void create_tree(vector<int>,Node*,int);
        void Dfs(Node*,int);
        void Bfs(int,Node*);
};
```

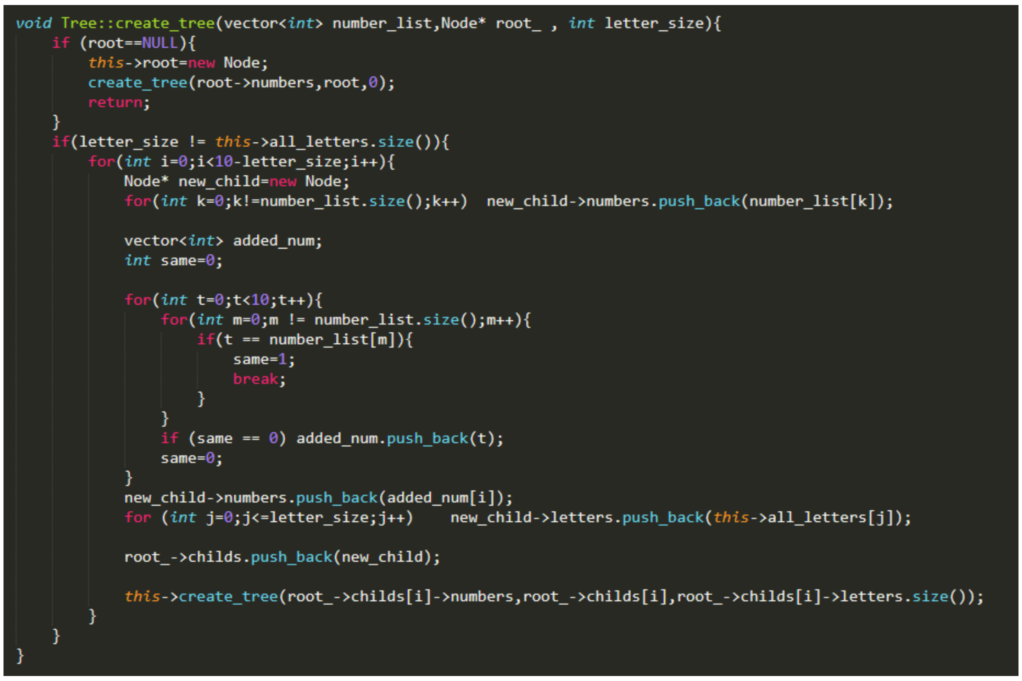
The tree class forms the structure of the entire tree.

root holds the first node. The variables s1, s2, s3 are used for the encrypted words (two, two, four) the tree is built on. All_letters contains all the letters of the encrypted words ('t', 'w', 'o', 'f', 'u', 'r').

The ‘correct_node’ points the node with the correct numbers.

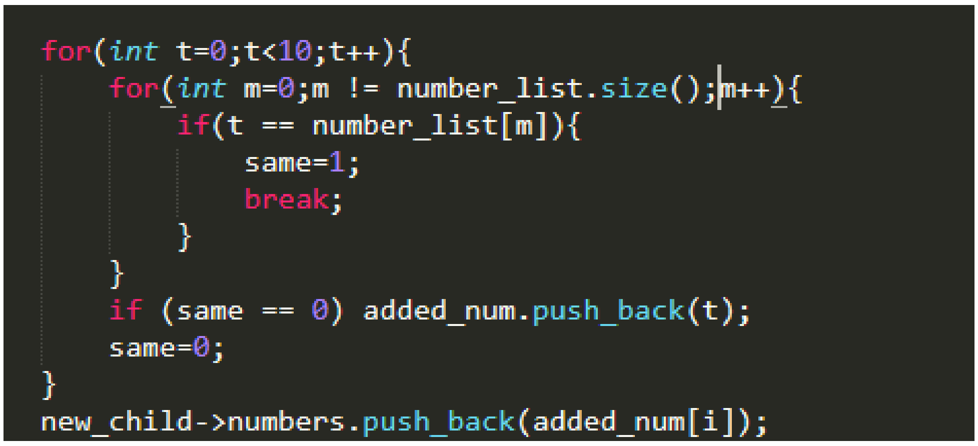
What other variables are used for will be explained while explaining the functions.

create_tree() function



```cpp
void Tree::create_tree(vector<int> number_list,Node* root_ , int letter_size){
    if (root==NULL){
        this->root=new Node;
        create_tree(root->numbers,root,0);
        return;
    }
    if(letter_size != this->all_letters.size()){
        for(int i=0;i<10-letter_size;i++){
            Node* new_child=new Node;
            for(int k=0;k!=number_list.size();k++)  new_child->numbers.push_back(number_list[k]);

            vector<int> added_num;
            int same=0;

            for(int t=0;t<10;t++){
                for(int m=0;m != number_list.size();m++){
                    if(t == number_list[m]){
                        same=1;
                        break;
                    }
                }
                if (same == 0) added_num.push_back(t);
                same=0;
            }
            new_child->numbers.push_back(added_num[i]);
            for (int j=0;j<=letter_size;j++)    new_child->letters.push_back(this->all_letters[j]);

            root_->childs.push_back(new_child);

            this->create_tree(root_->childs[i]->numbers,root_->childs[i],root_->childs[i]->letters.size());
        }
    }
}
```

The create_tree function is the function from which the tree's structure is created and the most complex part of the assignment.
create_tree has 3 parameters. The first parameter is number_list, this parameter is used to inherit the numbers from the parent of the node.
The root_ variable, on the other hand, points the subtrees in every recursive function.
letter_size gives you the number of the current layer, how many letters have been mapped so far.
First, it is checked whether root is null. If root is null, it shows that this tree has not been created yet. So first we create layer 0.
In the line if (letter_size! = this-> all_letters.size ()), we check if the tree has reached the maximum layer level.
letter_size gives the current number of layers and all_letters.size () gives the total number of layers.
Then we use the for loop to determine how many children each node will have based on the tree's level.
Here we define new_child in the for loop for each node.
Then with the for loop we add the numbers to a new child from the parent for each child. For example, if there is (7,5,4) in the parent node, these numbers are added to the list of numbers for the child.



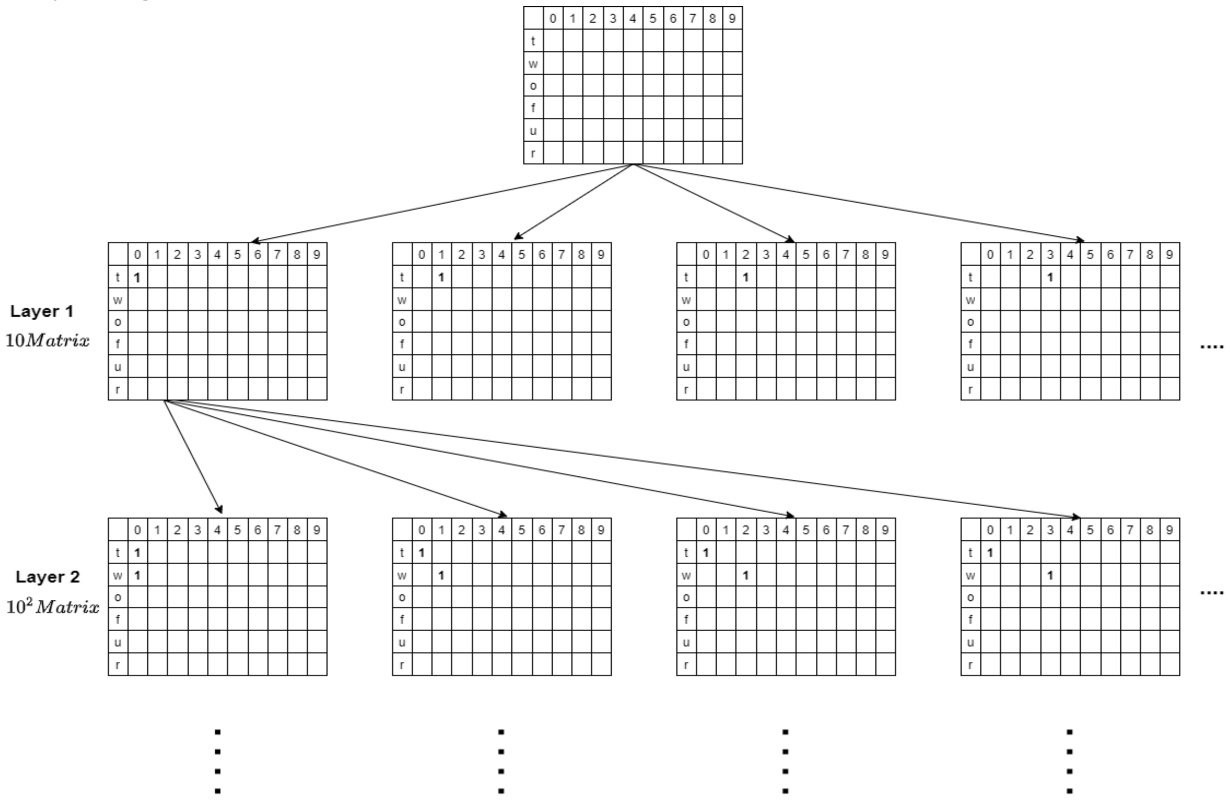
```
for(int t=0;t<10;t++){
    for(int m=0;m != number_list.size();m++){
        if(t == number_list[m]){
            same=1;
            break;
        }
    }
    if (same == 0) added_num.push_back(t);
    same=0;
}
new_child->numbers.push_back(added_num[i]);
```

The operation we do here is to add the new number to the child, but we use a small algorithm to not add the numbers in the parent while adding.
For example, if there are 7,5,4 numbers in the parent, we can add only 0,1,2,6,8,9 to the children's number_list.

for (int j = 0; j <= letter_size; j ++) new_child-> letters.push_back (this-> all_letters [j]);
In this part, we add as many letters to the child's letters array, which layer the child is in, from the all_letters array.
Then we add the child to the parent's childs array and call the create_tree function again for the subtrees.

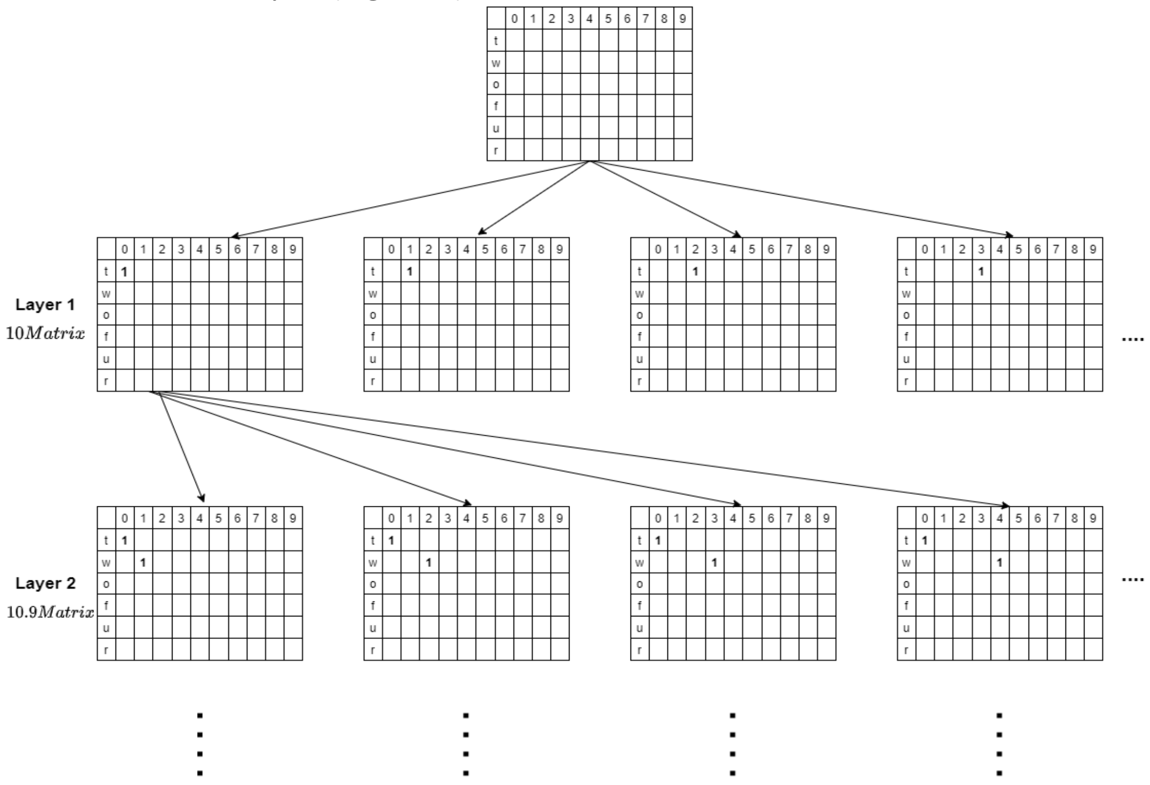
Important Note

This tree structure does not exactly supply the tree structure wanted in the assignment. Each node is required to have 10 children in the assignment. For example, 10 * 10 nodes are expected to be created in 1.layer, 10 * 10 in 2.layer, and 10 * 10 * 10 in 3.layer.(Figure 1)



(Figure 1)

But in this case, when the number of tree letters is too much, it uses too much memory and cannot perform BFS,DFS searches.
For example, for a tree with 8 layers, we need to use 10 ^ 8 nodes in the last layer.
Also, in this method, the same number can match different letters in matrix nodes, which violates the constraint.
Therefore, in this code, the number of children of each node decreases as the layer level increases, so that the same number does not match with different letters.
For example, it is expected that 10 nodes will be created in 1.layer, 10 * 9 in 2.layer, 10 * 9 * 8 in 3.layer. Thus, in a tree with 8 layers, only 1.814.400 nodes are formed instead of 10 ^ 8 in the last layer. (Figure 2)

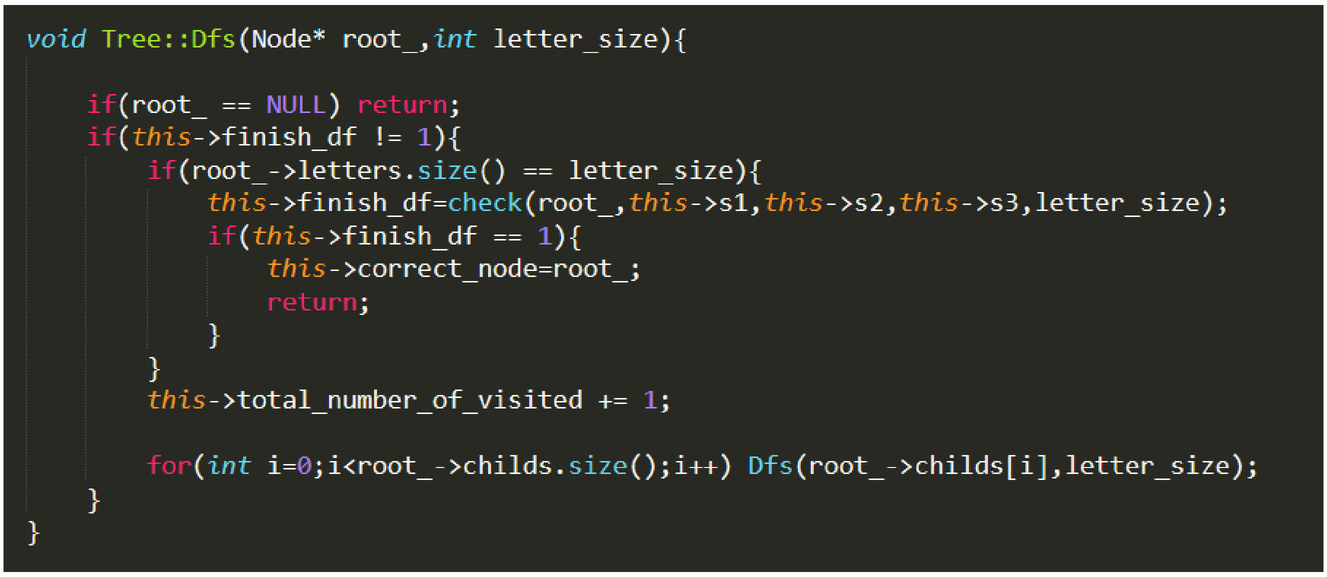


(Figure 2)

Pseudo-Code

```
create_tree(number_list, root_ , letter_size)
    if root == null
            root=Node
            create_tree(root.numbers,root,0)
            return
    if letter_size == all_letters.size
            for (i=0 to 10-letter_size)
                    new_child=Node
                    for(k=0 to number_list.size)
                            new_child.numbers.add(number_list[k])
                    added_num
                    same=0
                    for(t=0 to 10)
                            for(m=0 to number_list.size)
                                    if t == number_list[m]
                                            same=1
            break
        if same == 0
            added_num.add(t)
        same=0
new_child.numbers.add(added_num[i])
for(j=0 to letter_size )
    new_child.letters.add(all_letters[j])
root_.childs.add(new_child)
create_tree(root.childs[i].numbers, root.childs[i]., root.childs[i].letters.size)
```

Dfs() function



```cpp
void Tree::Dfs(Node* root_,int letter_size){

    if(root_ == NULL) return;
    if(this->finish_df != 1){
        if(root_->letters.size() == letter_size){
            this->finish_df=check(root_,this->s1,this->s2,this->s3,letter_size);
            if(this->finish_df == 1){
                this->correct_node=root_;
                return;
            }
        }
        this->total_number_of_visited += 1;

        for(int i=0;i<root_->childs.size();i++) Dfs(root_->childs[i],letter_size);
    }
}
```

Dfs function performs Depth First Search on Tree.

It takes two parameters. The root_ parameter is used to give subtrees parameters in the recursive function. The letter_size shows the total number of letters.

First we check whether root_ is null or not. If it is null, we return without doing anything.

If it is not null, we check whether the variable finish_df value is 1. The purpose of this check is if the algorithm finds the node where the encrypted letters match the correct numbers, finish_df value is 1 and the function ends.

If finish_df value is not 1, we first check if letter_size and letters.size() values are equal. The purpose of this is to check if root_ is on the last layer. If it is not on the last layer, it means there are letters that do not match any numbers. In this case, we cannot decrypt and check root_ is correct.

If root_ is on the last layer, we call the check function to do the decrypt operation. The check function checks if the root_ node is the correct node, and also checks 2 constraints.

If root_ is the correct node, check function returns 1. And finish_df value assign 1, if finish_df value is 1, root_ is assigned to correct_node and the function ends.

If root_ is not the correct node, the variable total_number_of_visited value is incremented by 1 and the Dfs function is called recursive for each child of root, ie the root's subtrees.

Important

In general, Dfs algorithm uses stack data structure but in this function , stack is not used because tree has not cycle and used recursive function so visited nodes is not needed to store in stack because the recursive function does not return for any node.

Pseudo-Code and The Complexity

```
Dfs(root_ , letter_size )
if root_ == NULL
        Return
If finish_df != 1
        If root_.letters.size == letter_size
                finish_df= check(root_, s1,s2,s3,letter_size)
                if finish_df == 1
                        correct_node = root_
                        return
        total_number_of_visited += 1
        for(i=0 to root_.childs.size)
                Dfs(root_.childs[i], letter_size)
```

The Complexity

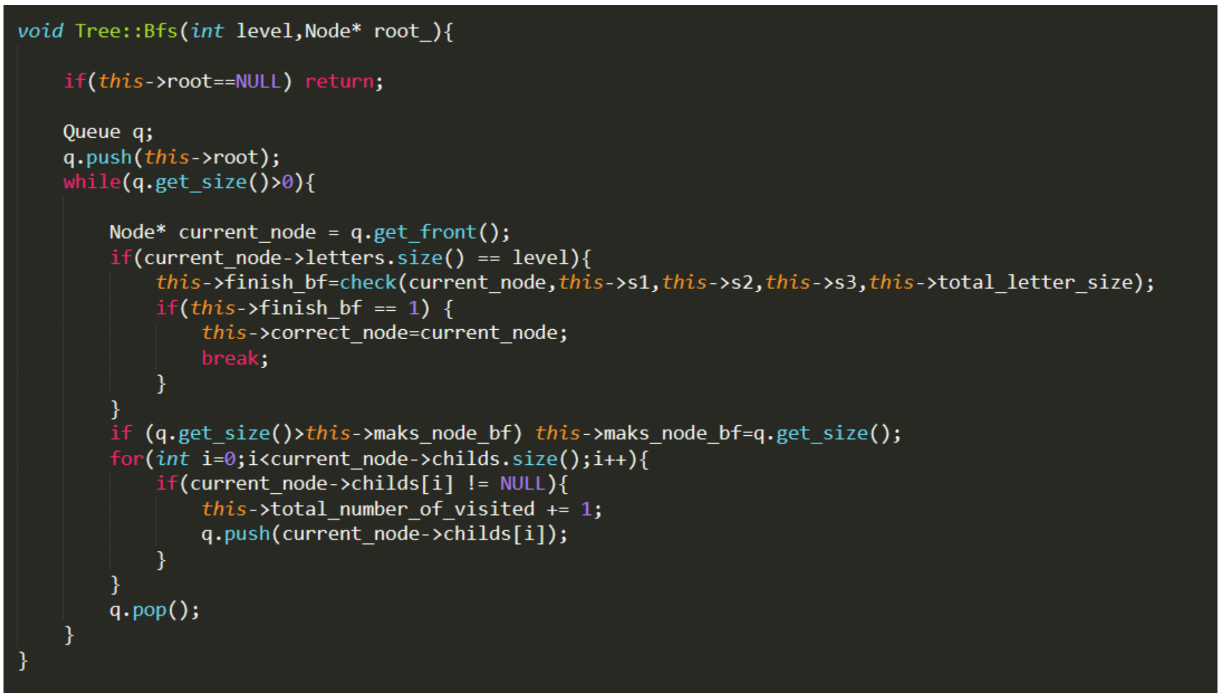
If we examine time complexity according to Pseudo-Code. Checking if root is null is a simple operation, so it is O (1). It is a simple task to check if finish_df is 1 and therefore it is O (1).

The if root_.letter_size == letter_Size operation and this if block are still simple operations. The check function can get confusing here, but as you will see in the code section, the check function consists of for loops with a limited number of loops and if-elses. The total_number_of_visited + = 1 operation is also a simple operation.

The part that determines the time complexity of DFS is the last part of the function. Here we are looking at the for loop first. In i = 0 to root_.childs.size, this is not decisive in time complexity since the number of children will decrease as root_'s layer increases and the maximum layer number will be 10. However, we call the Recursive process each time for each node and its subtrees, so we do as many recursive functions as the number of nodes.

Therefore, if we say n to the total number of nodes, **time complexity is O(n).**

Bfs() function



```
void Tree::Bfs(int level,Node* root_){

    if(this->root==NULL) return;

    Queue q;
    q.push(this->root);
    while(q.get_size()>0){

        Node* current_node = q.get_front();
        if(current_node->letters.size() == level){
            this->finish_bf=check(current_node,this->s1,this->s2,this->s3,this->total_letter_size);
            if(this->finish_bf == 1) {
                this->correct_node=current_node;
                break;
            }
        }
        if (q.get_size()>this->maks_node_bf) this->maks_node_bf=q.get_size();
        for(int i=0;i<current_node->childs.size();i++){
            if(current_node->childs[i] != NULL){
                this->total_number_of_visited += 1;
                q.push(current_node->childs[i]);
            }
        }
        q.pop();
    }
}
```

Bfs function performs Breadth First Search on Tree.

It takes two parameters. The root_ parameter is used to give subtrees parameters in the recursive function. The level shows the total number of letters.

First we check whether root_ is null or not. If it is null, we return without doing anything.

If it is not null, Queue q is defined. (Queue is not STL. Queue and its functions are defined in the first lines of the .cpp file in the assignment.)

Then we add the root of the tree to this queue. Then it enters the while loop until the queue is empty. First, we assign the first element of the queue to the current_node variable. Then the number of letters of current_node and the level is checked if it is not equal, current_node is not on the last layer and there are still letters that do not mapping any numbers and in this case decrypt operation cannot be done.

If it is equal, we use the check function to check if current_node is the correct node. If it is the correct node, check returns 1, if not, it returns -1 and the return value is assigned to finish_bf.

If the value of finish_bf is 1, current_node is assigned to the correct_node variable and the function ends.

If current_node is not in the last layer, the current size of the queue is checked. If the queue size is greater than max_node_bf, the new max_node_bf value is assigned the queue size.

Then we traverse the children of current_node with the for loop. In each loop loop, the number of total_number_of_visited increases by 1 first and add the children of current_node to the end of the queue.

At the end of the while loop, current_node in front of the queue is pop from the queue.

Pseudo-Code and The Complexity

```
Bfs(level, root_):
If (root == NULL )
    Return
Queue q
q.add(root);
while q.size to 0
    current_node = q.front
    if current_node.letter.size == level
        finish_bf=check(current_node,s1,s2,s3,total_letter_size)
        if  finish_bf == 1
            correct_node = current_node
            break
    if q.size > maks_node
        maks_node= q.size
    for (i=0 to current_node.childs.size)
        if current_node.childs[i] != NULL
            q.add(current_node.childs[i])


    q.pop()
```

The Complexity

If we examine the time complexity of BFS according to pseudo-code. first we check if root is null. This is a simple operation.

Then we create a queue object to store discovered nodes, which is a simple operation.

The part that determines the time complexity of Bfs is the while loop. Because here we are adding all the nodes in the tree to the queue in the worst case. Assuming there are n nodes, the time complexity of the while loop is O (n).

Let's look at the while block. First we get current_node the first element in the queue, this is a simple operation, so O (1).

Then we check if the number of letters in root equals level, which is a simple operation. The check function is called . I explained in BFS that the check function is a simple function.

Checking if finish_df is 1 and assigning root_ to correct_node is a simple operation, ie their time complexity is O (1).

Then we compare the max_node number with the size of the queue, it's a simple operation in this operation. Then we add the children of current_node to the queue and check whether the children are null or not.

The for loop here is not a simple operation, but since the number of children of current_node will be limited, this does not affect time complexity.

In this case, the while loop and the queue are the main factors that determine the time complexity of the Bfs function.

In other words, **the time complexity of the Bfs function is O(n).**

2. Analyze and compare the algorithm results

a. The number of visited nodes

For TWO TWO FOUR input the results:
BFS: 147.660
DFS: 138.197
For SEND MORE MONEY input the results:
BFS : 2.540.331
DFS : 2.511.446

The number of nodes visited is less in DFS. This is because the correct node is always on the last layer and Since DFS searches deep priority, it looks at the nodes in the last layer before visiting all the nodes in the upper layers, so it finds the correct numbers by visiting fewer nodes.
BFS, on the other hand, examines all the nodes in the upper layers before going down to the nodes on the last layer.
because BFS is doing breadth priority searches.
That's why he has to visit more nodes than DFS.

b. The maximum number of nodes kept in the memory

For TWO TWO FOUR input the results:
BFS: 151.200
DFS: 187.301
For SEND MORE MONEY input the results:
BFS : 1.814.400
DFS : 2.606.501

The maximum number of nodes kept in memory is too high in DFS.
Actually, this situation is a little odd because logically there should have been more in BFS.

This is because I didn't use stacks in DFS since there are no cycles in the tree structure. Instead, the total number of nodes in the tree is assigned to the maximum number of nodes kept in memory for DFS.
In BFS, the nodes on the higher layer are always kept in the Queue and since most nodes are on the last layer, the maximum number of nodes kept in memory in BFS is equal to the number of nodes on the last layer.

c. The running time (in ITU SSH)

For TWO TWO FOUR input the results:
BFS: 0.15 Second
DFS: 0.13 Second
For SEND MORE MONEY input the results:
BFS : 3.44 Second
DFS : 3.09 Second

These times do not include the time of creating tree. It is just the time of the search.
There is a relationship between the running time and the number of nodes visited.
As I explained above, the runnig time is also longer because the number of nodes visited is higher in BFS.

3. Why we should maintain a list of discovered nodes? How this affects the outcome of the algorithms?

Since we created a tree structure in this assignment, and since there is no cycle, we could do DFS without keeping discovered nodes in a list.
In BFS, we used a queue structure because we had to keep the nodes in a lower layer because we were doing spread-priority searches. If we didn't use queues here and kept the nodes we visited. Probably we could not go to the sibling node and visit its children, in this case this algorithm would not be BFS.

Speaking for graphs, if we did not keep the nodes we visited in a list, we would constantly visit the same nodes as there are cycles and the search would never end, so the algorithm would not yield a result.
Keeping the nodes we visit affects the algorithm, first of all, if we cannot establish a good tree structure, it uses too much memory and the algorithm stops working after a while it also slows down the algorithm.