

**Marmara University Faculty of Engineering**

CSE2225 – DATA STRUCTURES

**Project 2**

|  |  |  |
| --- | --- | --- |
| Department | Student Id Number | Name & Surname |
| CSE | 150122051 | Devrim Polat |
| CSE | 150123992 | Nuraddin Abbasov |

# Introduction

We are looking for a data structure where, even though some worst case (O(n)) accesses may be possible, m consecutive tree operations starting from an empty tree (inserts, finds and/or removals) take O(m\*log2n). The main idea is to assume that O(n) accesses are not bad if they occur relatively infrequently. Hence, we are looking for modifications of a BST per tree operation that attempts to minimize O(n) accesses by converting the node on which the operations are performed mostly into a root.

The code will provide two different algorithms: Splay and Mod-Splay.

In Splay algorithm, as we insert a new node into the tree, we will make it root by performing splay rotations.

In Mod-Splay algorithm we will make insert operations besides, we will only splay the node if the node is being accessed more frequently than the others.

The code inserts nodes to the tree by reading input.txt file. In output file it shows:

* Pre-order traversal of Splay tree
* Cost of constructing Splay tree
* Pre-order traversal of Mod-Splay tree
* Cost of constructing Mod-Splay tree

Before implementing our methods, we construct a StNode struct which holds data, frequency, parent node, left child and right child. The implementation has shown below:

# 

# Methods and Algorithm Overview

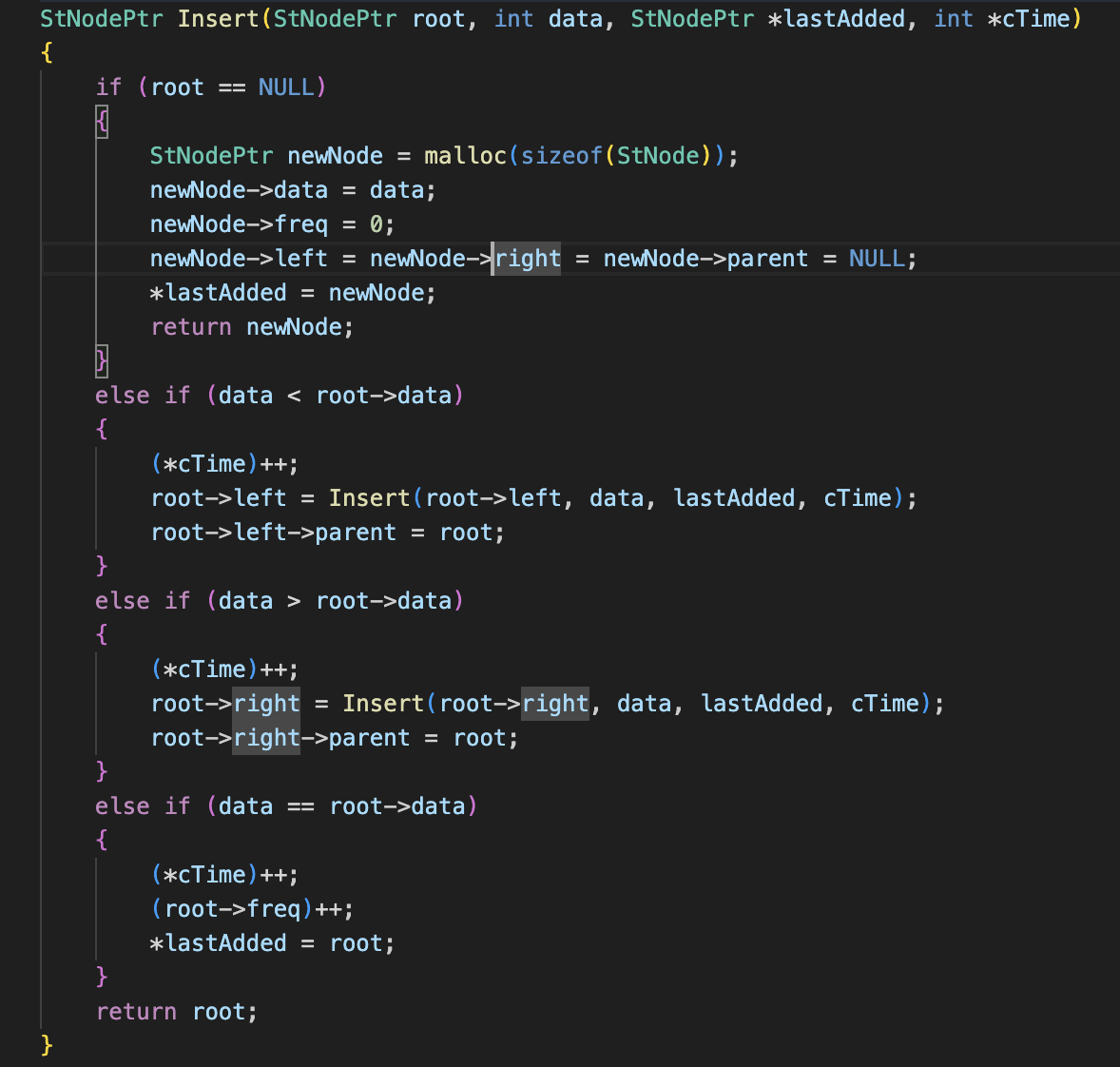
We implied two important methods to construct the algorithm: insert and Splay.

StNodePtr Insert(StNodePtr root, int data, StNodePtr \*lastAdded, int \*cTime)

Insert method searches for a node that contains data parameter as its key. It performs comparison operation starting from the root than iterate through the tree by iterating to left or right child with recursive manner. The method terminates in two conditions. If last called Insert method gets “NULL” parameter as the root it means that there is no key exists in the tree so we should insert the key as a node to the place that we meet with NULL parameter. Second condition is that, if there is a node which has the key same as the data parameter. It increases the frequency of that node then is being terminated.

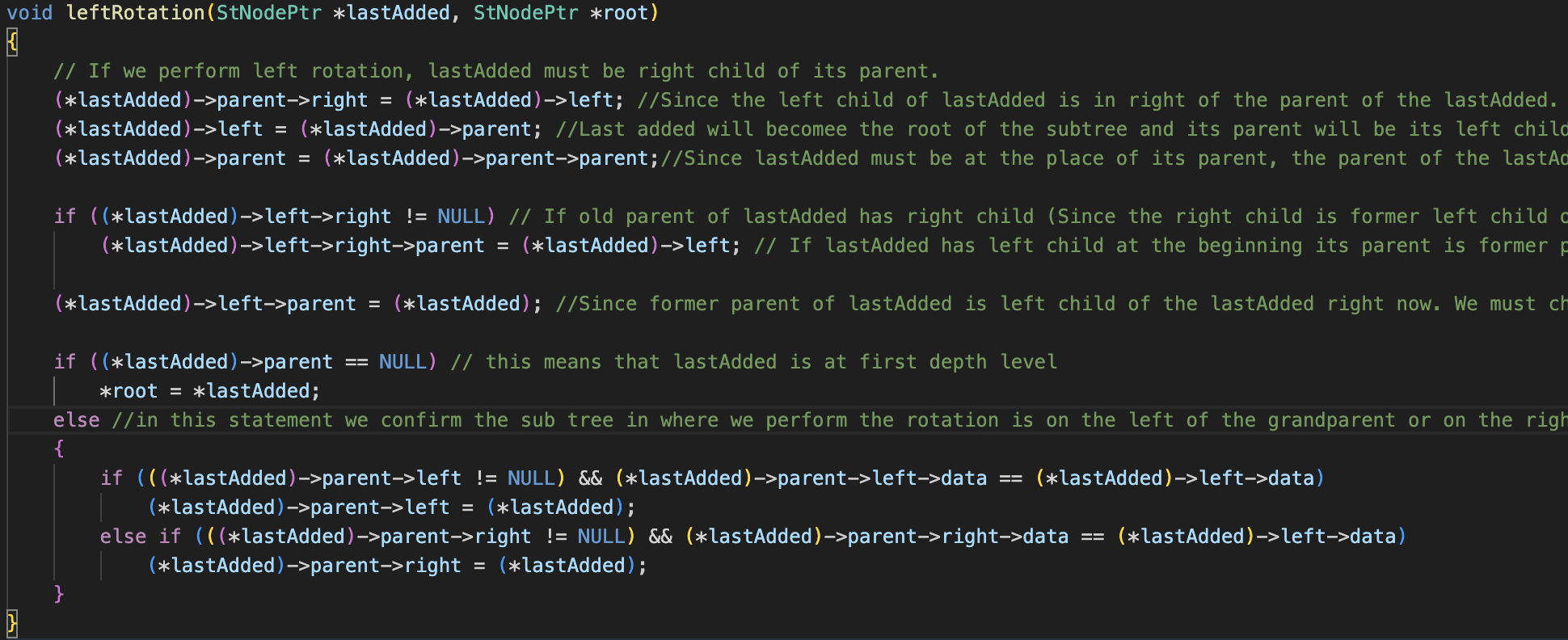
cTime parameter points to cost time of the tree. It increases by 1 when a comparison operation is completed.

lastAdded pointer points to last added node or the node whose frequency is increased recently. We use this parameter to perform splay operations to the node without implementing find method. This reduces the cost.



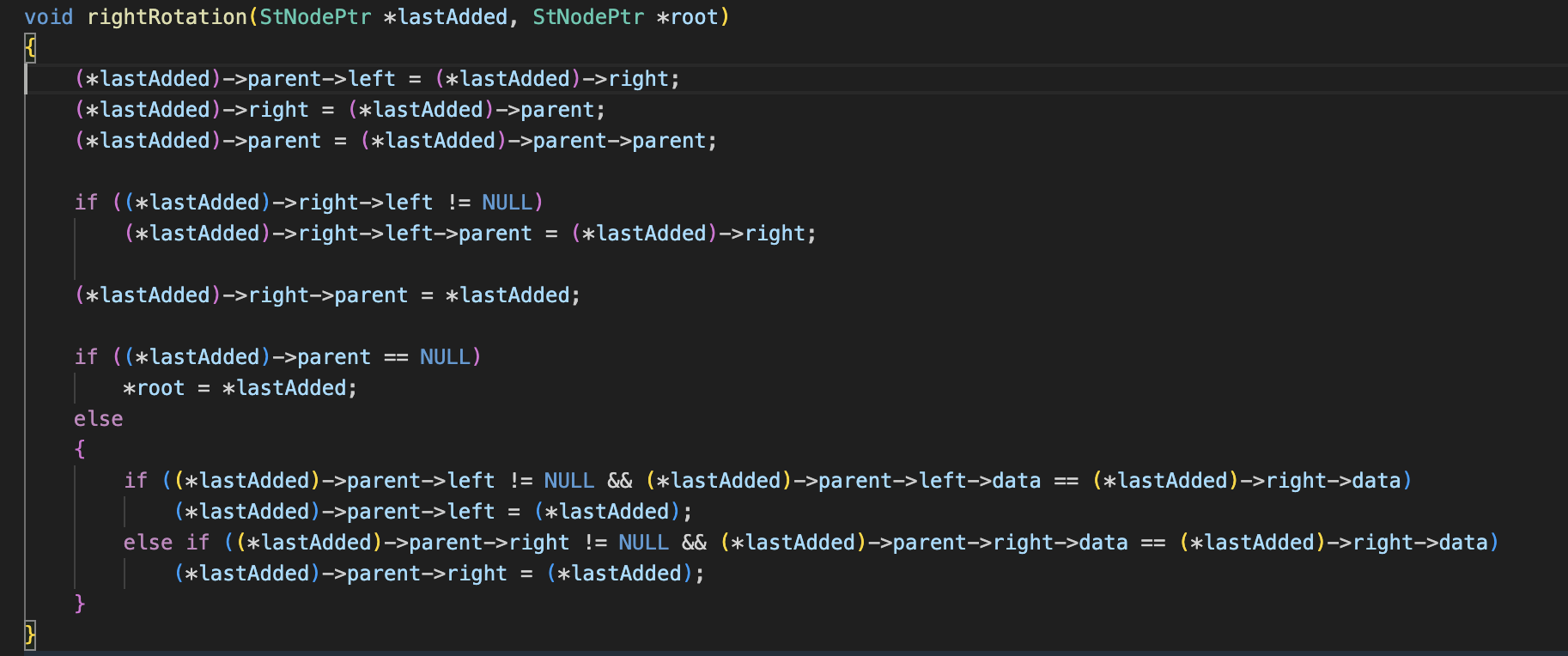
Before mentioning Splay method, we should introduce leftRotation and rightRotation methods to explain Splay method clearly.

void leftRotation(StNodePtr \*lastAdded, StNodePtr \*root)



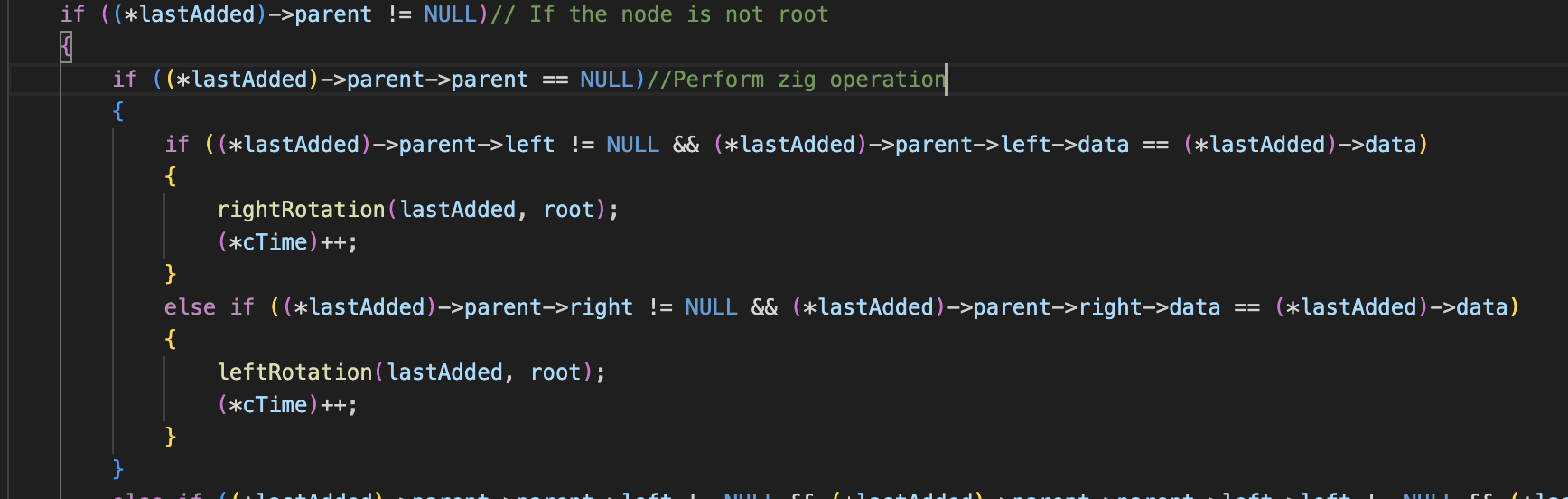
When we perform rotation, lastAdded must be put on one higher depth level. Its right child must be on the right of the former parent of lastAdded. The former parent must be the left child of lastAdded. Since lastAdded must be at the depth level of its former parent, we set parent of lastAdded as the parent of its former parent. If new parent of lastAdded is NULL we can say that lastAdded is the root. Finally, we check the place of our new subtree by comparing data of the children of the former grandparent of the lastAdded. The code implements all these things above successively as mentioned.

void rightRotation(StNodePtr \*lastAdded, StNodePtr \*root)



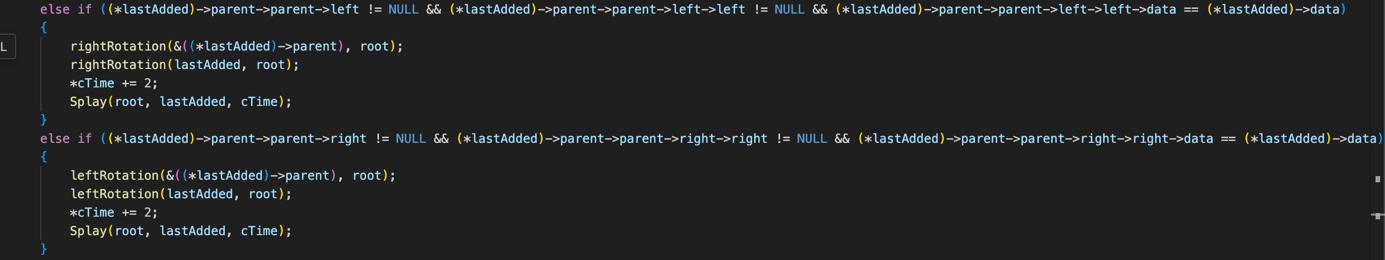
We perform similar operations but this time we consider opposite directions of nodes.

void Splay(StNodePtr \*root, StNodePtr \*lastAdded, int \*cTime)

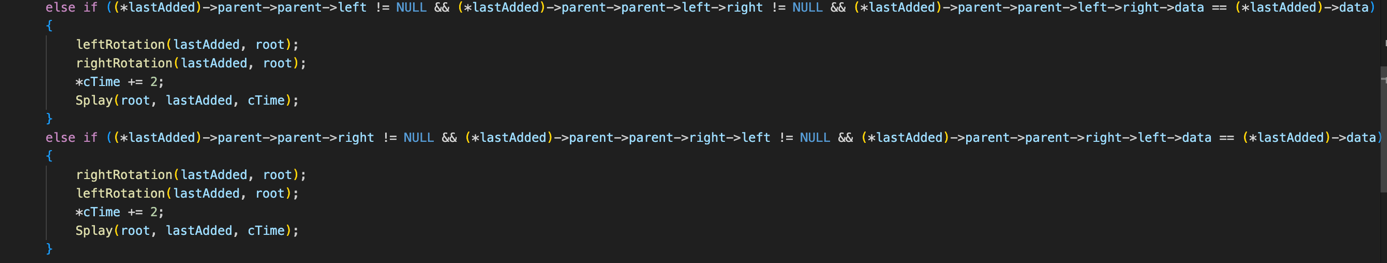


cTime is same cost time as we mentioned in the insert method.

The image above we check if the node that we want to splay is the child of the root. If it is then we perform zig rotation since it is single rotation we increment \*cTime by one.

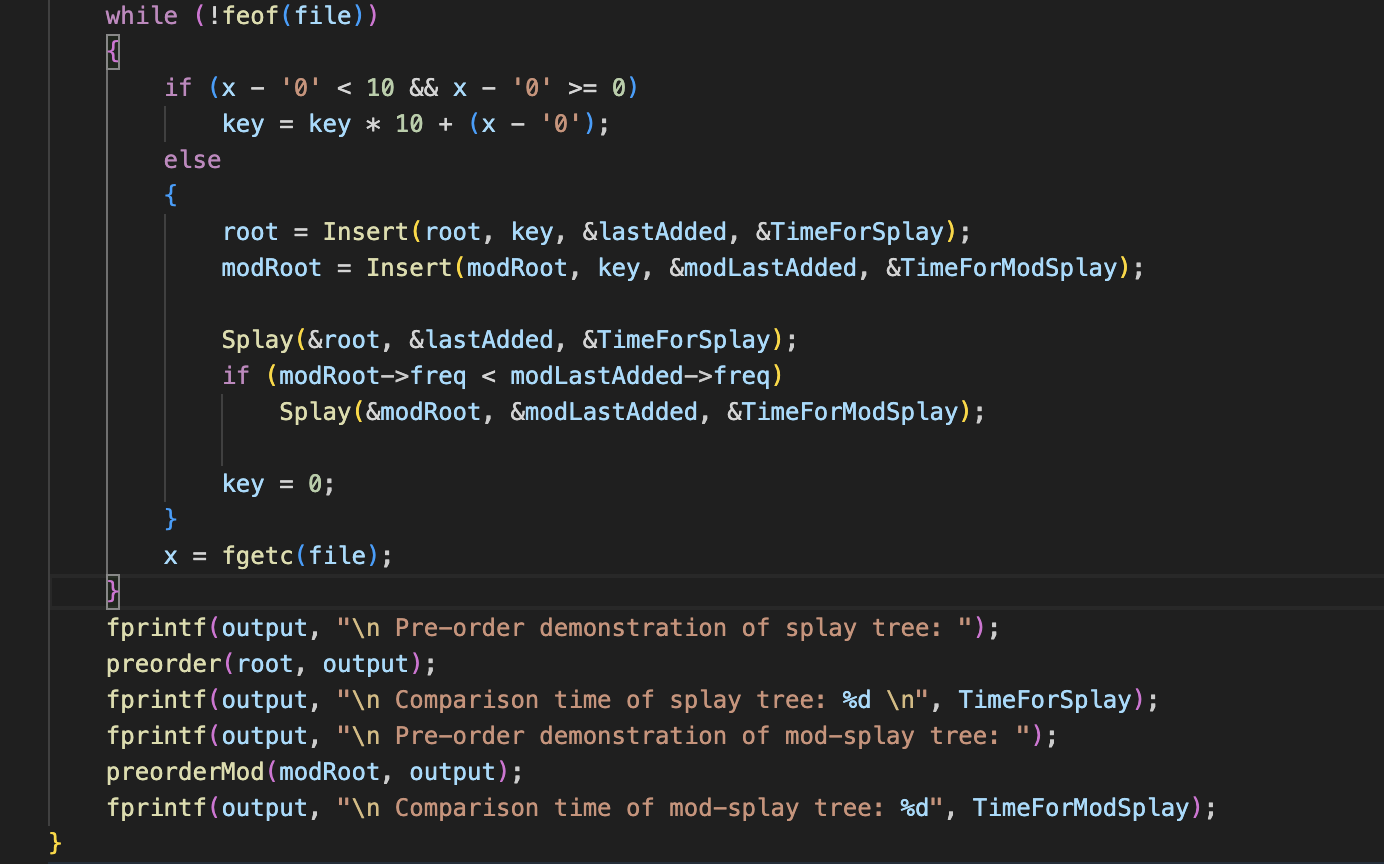


Next two conditions, we check that if the node, its parent, and grandparent placed like a line together in the tree structure. In this case we perform zigzag operations by calling the same rotation function twice.



After we check if it suits for zigzag operation, we perform two rotations. After the zigzag rotation the parent and grandparent of the lastAdded becomes new children of the lastAdded and lastAdded rises two depth levels above. The order of leftRotation and rightRotation depend on the formation of the zigzag shape. The zigzag shape also affects which one of the parent and grandparent will become which child of the lastAdded.

# Using methods for the algorithm

****

When we insert new key to tree, we immediately perform splay operations to that node. To perform splay operations to a node in mod-splay tree, it must have higher frequency than the root since the root was the highest frequent node before.

# Conclusion

metin, yazı tipi, ekran görüntüsü, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi, siyah içeren bir resim

Açıklama otomatik olarak oluşturuldu

Finally, we observe that mod-splay tree has lower cost than normal Splay tree since we perform splay operations on mod-splay tree only if it is necessary which means if we want to reach specific node frequently. This is the reason why mod-splay tree might be useful while we implement different algorithms.