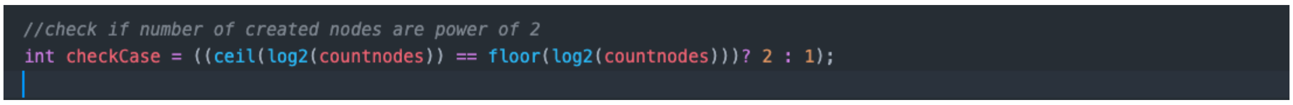
Berkay Mengunogul 1501199934

CSE2225 Data Structures PROJECT #2 Report

In this report, first I will explain my algorithm with illustrations then I will present how I implemented it. At last, I will show how to use it and I will share the results.

First, the input file is read and the nodes of the tree are created one by one. While creating the nodes each node is checked if they are less than or equal to zero and if they are replicated. If one of those two conditions is true then that node is skipped with displaying an error message. In the end, after nodes are created, it is checked that if there are 16 or more nodes created. If it is less than 16, the program will produce an error message and exit. Otherwise created nodes will be sorted with insertion sort in descending order by their value. Each node has value, height, position in the depth level, depth level, right and left child attributes.

Second, considering the number of nodes created, It is checked that if the number of nodes is power of 2. It is checked by taking the celling of log base 2 and the floor of log base 2 as it is shown in Code 1. If the results are equal then it means it is power of two. The reason is, as it is explained in the class there are two cases that need to be implemented to construct BST. If the number of nodes is not power of 2 then case 1 will be applied. If nodes are equal to any power of 2 then case 2 will be applied. Cases will be explained with their solutions.



```
//check if number of created nodes are power of 2
int checkCase = ((ceil(log2(countnodes)) == floor(log2(countnodes)))? 2 : 1);
```

Code 1: Deciding on Case 1 or 2.

For both cases, simply there are 3 sections which are the complete BST section, the excessive nodes section which comes right after the complete BST, and the remaining nodes section.

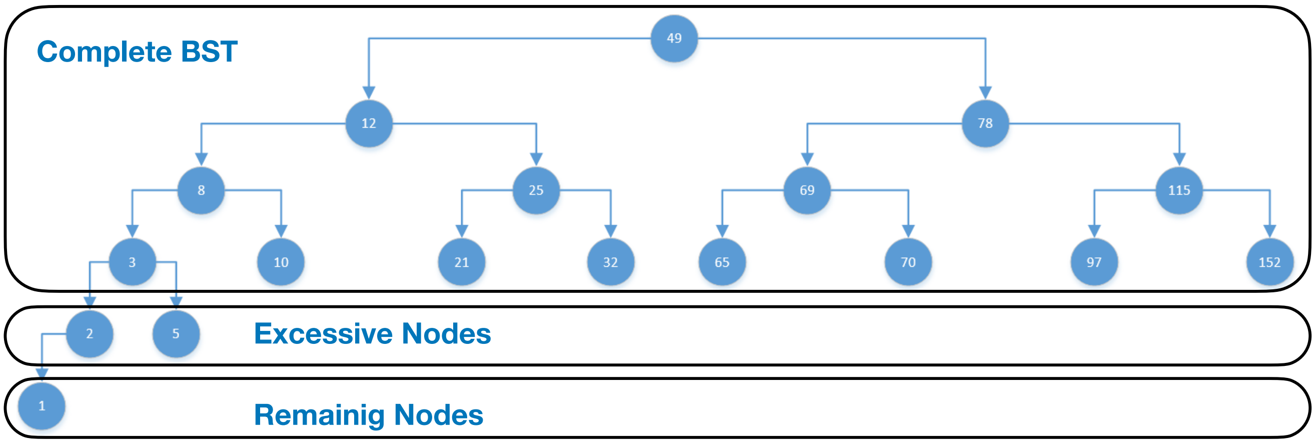


Figure 1: Illustration of Sections.

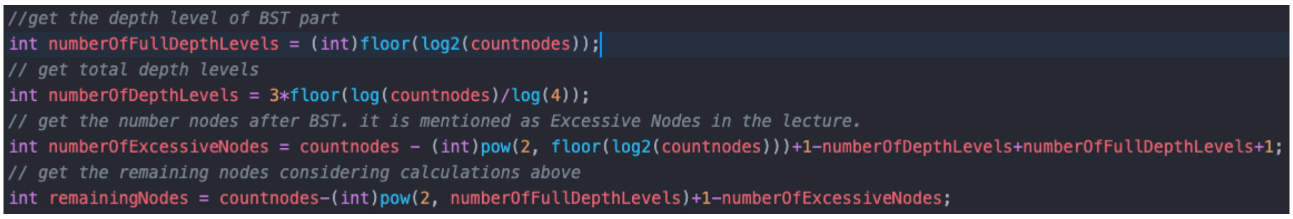
In Figure 1, the sections mentioned above are shown in clusters.

Considering that nodes are distributed over $3 \lfloor \log_4 n \rfloor$ depth levels with n number of nodes:

- Complete BST has:
    - Depth levels: $\lfloor \log_2(n) \rfloor$
    - Number of Nodes: $2^{\lfloor \log_2(n) \rfloor} - 1$

- Excessive Nodes:
    - Depth levels: 1
    - Number of Nodes: $n - 2^{\lfloor \log_2(n) \rfloor} + 1 - 3\lfloor \log_4(n) \rfloor + \lfloor \log_2(n) \rfloor + 1$

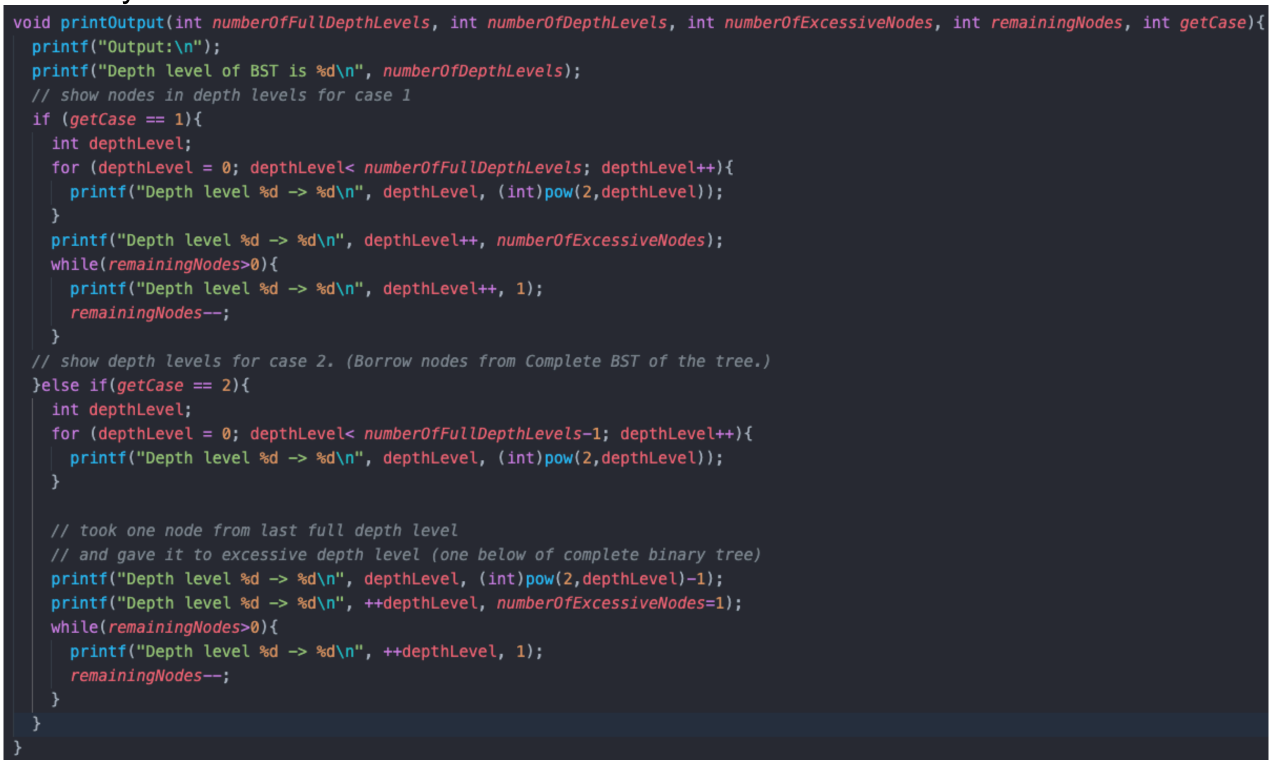
- Remaining Nodes:
    - Depth levels: same as the number of nodes
    - Number of Nodes: $n - 2^{\lfloor \log_2(n) \rfloor} + 1 -$ number of excessive nodes.



```c
//get the depth level of BST part
int numberOfFullDepthLevels = (int)floor(log2(countnodes));
// get total depth levels
int numberOfDepthLevels = 3*floor(log(countnodes)/log(4));
// get the number nodes after BST. it is mentioned as Excessive Nodes in the lecture.
int numberOfExcessiveNodes = countnodes - (int)pow(2, floor(log2(countnodes)))+1-numberOfDepthLevels+numberOfFullDepthLevels+1;
// get the remaining nodes considering calculations above
int remainingNodes = countnodes-(int)pow(2, numberOfFullDepthLevels)+1-numberOfExcessiveNodes;
```

Code 2: Calculation of Complete BST, Excessive Nodes and Remaining Nodes.

In Code 2, calculations are expressed in code. In terms of Case 1 and Case 2, calculations are slightly different. In Case 1, results can be used as it is but for case 2 since the number of nodes is power of 2, excessive nodes end up as 0. So, it is necessary to take nodes from complete BST to fill up the depth levels. Therefore, it is only needed to take one node and carry it down to the excessive node section.

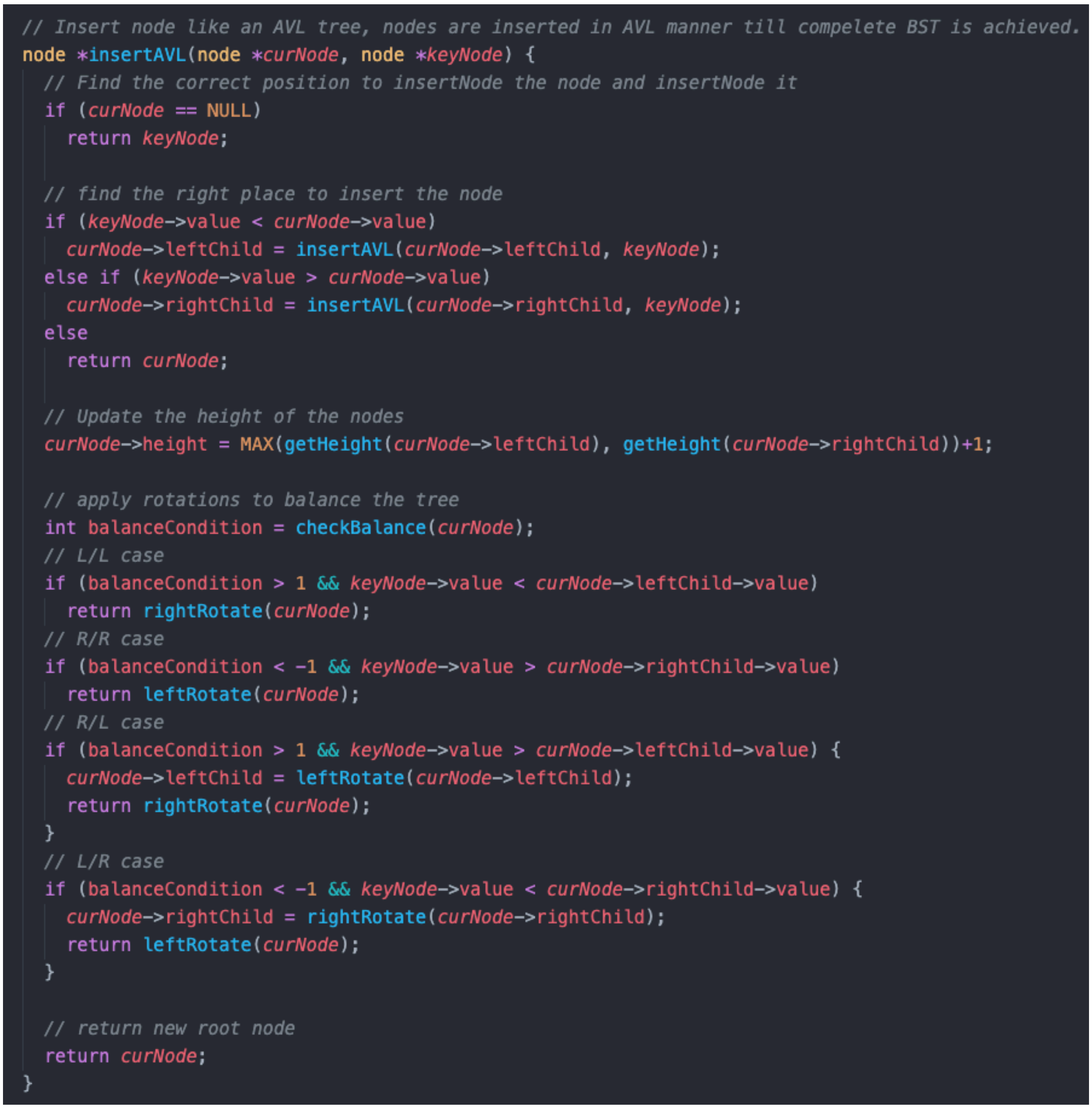


```c
void printOutput(int numberOfFullDepthLevels, int numberOfDepthLevels, int numberOfExcessiveNodes, int remainingNodes, int getCase){
  printf("Output:\n");
  printf("Depth level of BST is %d\n", numberOfDepthLevels);
  // show nodes in depth levels for case 1
  if (getCase == 1){
    int depthLevel;
    for (depthLevel = 0; depthLevel< numberOfFullDepthLevels; depthLevel++){
      printf("Depth level %d -> %d\n", depthLevel, (int)pow(2,depthLevel));
    }
    printf("Depth level %d -> %d\n", depthLevel++, numberOfExcessiveNodes);
    while(remainingNodes>0){
      printf("Depth level %d -> %d\n", depthLevel++, 1);
      remainingNodes--;
    }
  // show depth levels for case 2. (Borrow nodes from Complete BST of the tree.)
  }else if(getCase == 2){
    int depthLevel;
    for (depthLevel = 0; depthLevel< numberOfFullDepthLevels-1; depthLevel++){
      printf("Depth level %d -> %d\n", depthLevel, (int)pow(2,depthLevel));
    }

    // took one node from last full depth level
    // and gave it to excessive depth level (one below of complete binary tree)
    printf("Depth level %d -> %d\n", depthLevel, (int)pow(2,depthLevel)-1);
    printf("Depth level %d -> %d\n", ++depthLevel, numberOfExcessiveNodes=1);
    while(remainingNodes>0){
      printf("Depth level %d -> %d\n", ++depthLevel, 1);
      remainingNodes--;
    }
  }
}
```

Code 3: Printing number of nodes at Depth Levels.

In Code 3, the calculations above gave as parameters to the printOutput() function. Depending on cases, if it is Case 2 one node is taken from complete BST and carried down. The number of nodes in each depth level is displayed on the console.

When it comes to the construction of BST, simply Complete BST and Excessive Nodes sections are inserted one by one with AVL Tree manners. Since nodes are inserted from an array that is in descending order, the remaining nodes have the lowest values. That makes them align in the most left part of the BST, as represented in Figure 1. After Creating the Balanced BST, the remaining nodes are inserted in BST manner, remaining nodes are inserted with the descending order to the system so that they can build a structure that each node goes down to a single depth level. For example, if there are 3 nodes in the excessive nodes section then there will be 3 depth levels in the excessive nodes section.



```
// Insert node like an AVL tree, nodes are inserted in AVL manner till compelete BST is achieved.
node *insertAVL(node *curNode, node *keyNode) {
  // Find the correct position to insertNode the node and insertNode it
  if (curNode == NULL)
    return keyNode;

  // find the right place to insert the node
  if (keyNode->value < curNode->value)
    curNode->leftChild = insertAVL(curNode->leftChild, keyNode);
  else if (keyNode->value > curNode->value)
    curNode->rightChild = insertAVL(curNode->rightChild, keyNode);
  else
    return curNode;

  // Update the height of the nodes
  curNode->height = MAX(getHeight(curNode->leftChild), getHeight(curNode->rightChild))+1;

  // apply rotations to balance the tree
  int balanceCondition = checkBalance(curNode);
  // L/L case
  if (balanceCondition > 1 && keyNode->value < curNode->leftChild->value)
    return rightRotate(curNode);
  // R/R case
  if (balanceCondition < -1 && keyNode->value > curNode->rightChild->value)
    return leftRotate(curNode);
  // R/L case
  if (balanceCondition > 1 && keyNode->value > curNode->leftChild->value) {
    curNode->leftChild = leftRotate(curNode->leftChild);
    return rightRotate(curNode);
  }
  // L/R case
  if (balanceCondition < -1 && keyNode->value < curNode->rightChild->value) {
    curNode->rightChild = rightRotate(curNode->rightChild);
    return leftRotate(curNode);
  }

  // return new root node
  return curNode;
}
```

Code 4: Inserting node in AVL Tree manner.

In Code 4, AVL tree insertion is expressed. Firstly, the right position of the node is searched on the existing tree then it is inserted into that place. Then heights of the nodes are updated. After updating the nodes, the height of their left and right subtrees are compared. If the difference is bigger than 1 then that node becomes an alpha node. So, appropriate rotations can be applied considering the heights. From the lecture notes, the right rotation is shown in Figure 2 and the left rotation is shown in Figure 3.

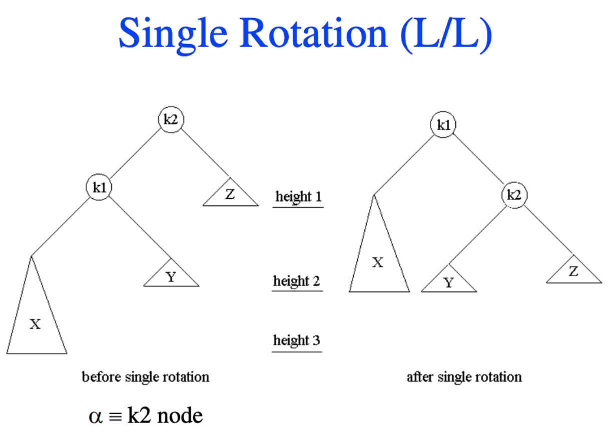


Figure 2: Single Rotation to Right (L/L case)

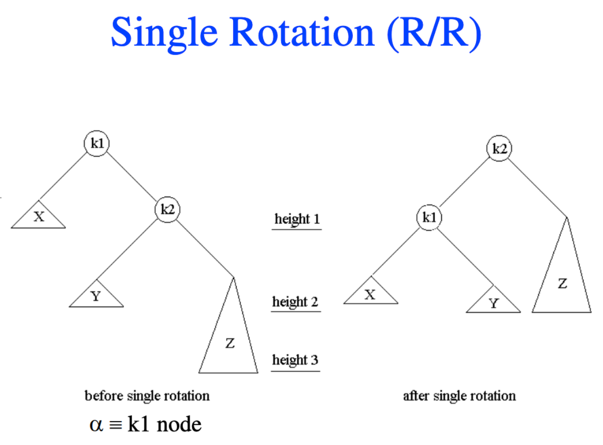
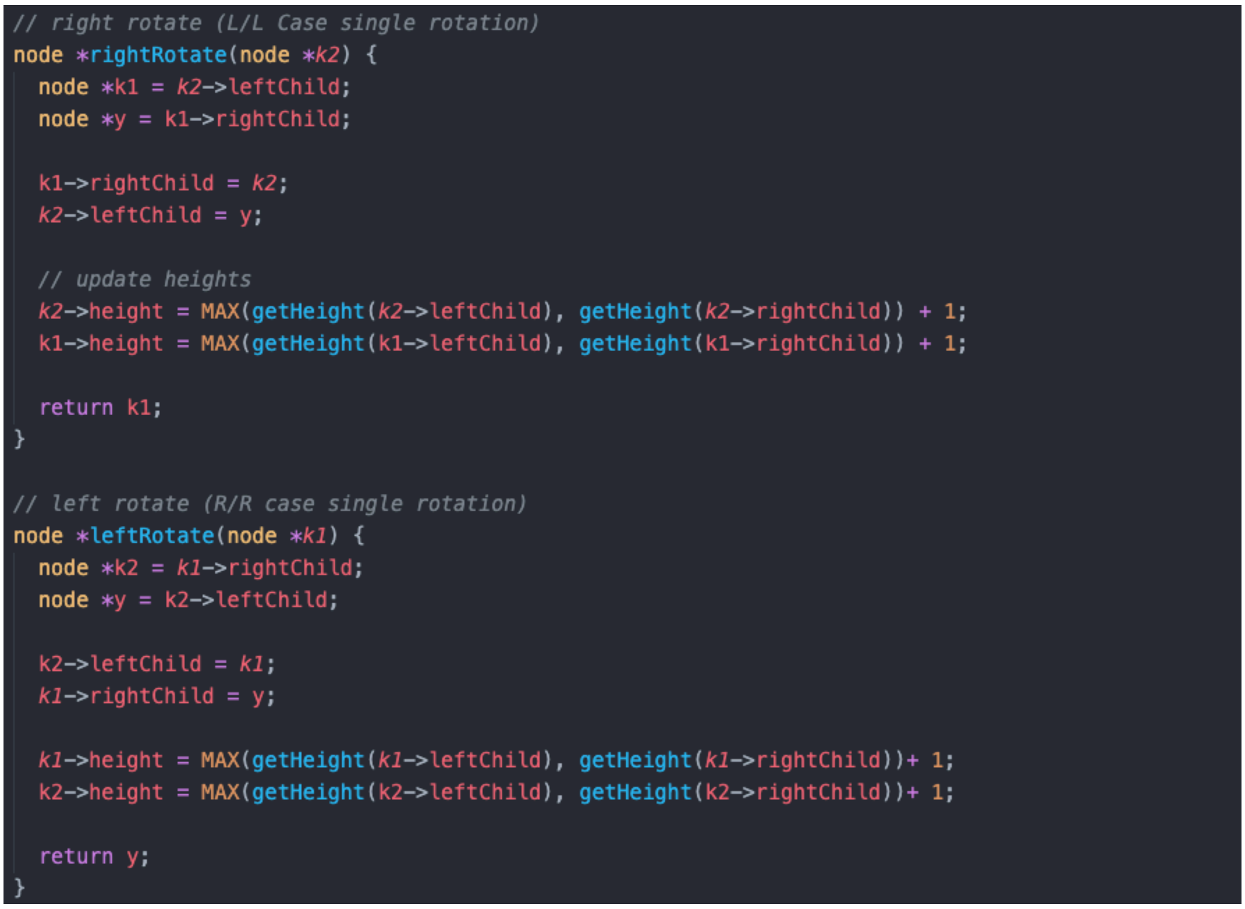


Figure 3: Single Rotation to Left (R/R case)

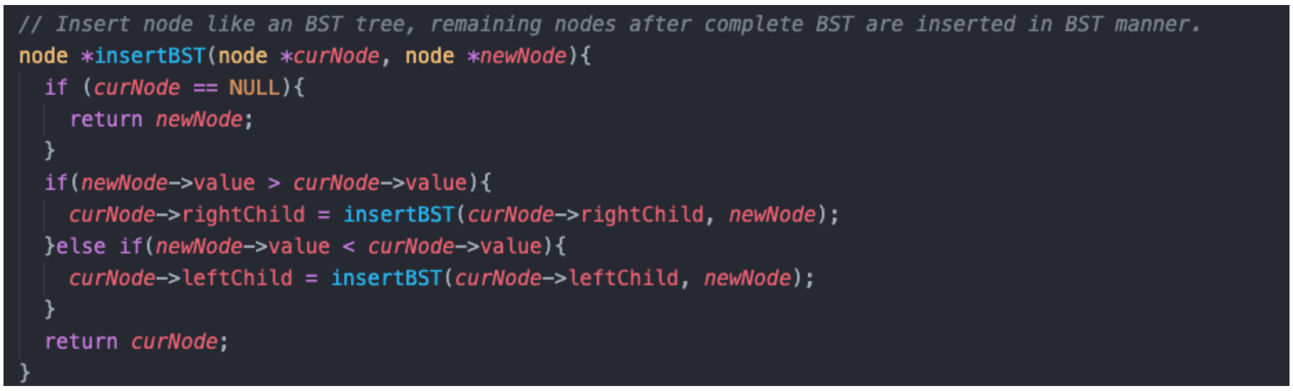
Since double rotations are a combination of single rotations, they are called sequentially for the R/L and L/R cases.



```
// right rotate (L/L Case single rotation)
node *rightRotate(node *k2) {
  node *k1 = k2->leftChild;
  node *y = k1->rightChild;

  k1->rightChild = k2;
  k2->leftChild = y;

  // update heights
  k2->height = MAX(getHeight(k2->leftChild), getHeight(k2->rightChild)) + 1;
  k1->height = MAX(getHeight(k1->leftChild), getHeight(k1->rightChild)) + 1;

  return k1;
}

// left rotate (R/R case single rotation)
node *leftRotate(node *k1) {
  node *k2 = k1->rightChild;
  node *y = k2->leftChild;

  k2->leftChild = k1;
  k1->rightChild = y;

  k1->height = MAX(getHeight(k1->leftChild), getHeight(k1->rightChild))+ 1;
  k2->height = MAX(getHeight(k2->leftChild), getHeight(k2->rightChild))+ 1;

  return y;
}
```

Code 5: Right Rotation (L/L case) and Left Rotation (R/R case) expressed.

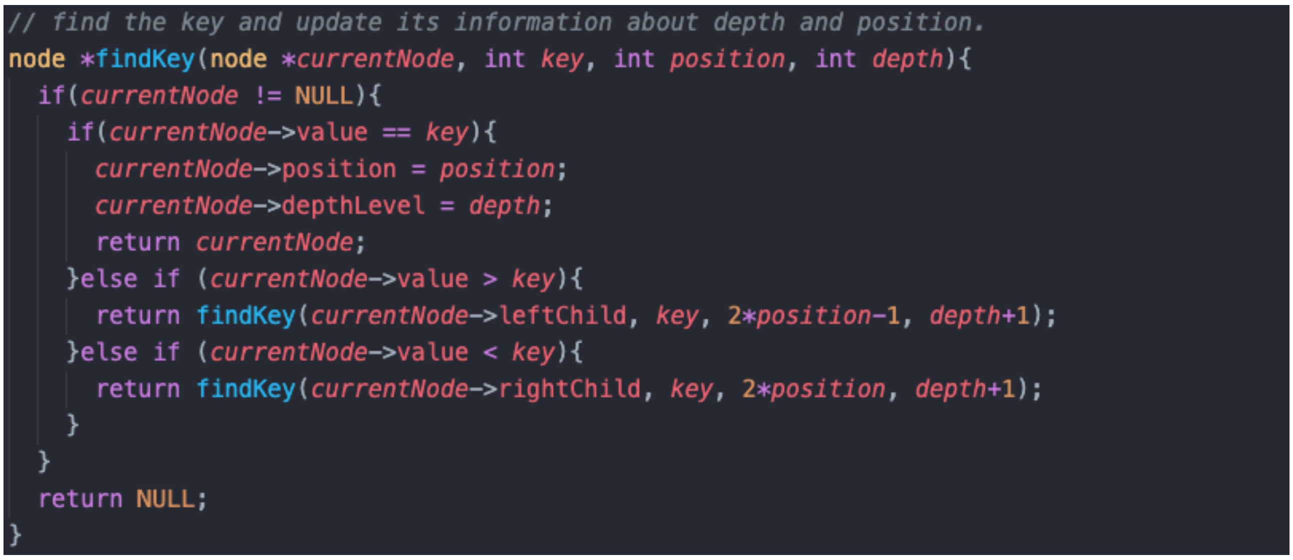
In code 5, considering Figure 2 and Figure 3, rotations are coded. After rotating the nodes their heights are also updated.



```c
// Insert node like an BST tree, remaining nodes after complete BST are inserted in BST manner.
node *insertBST(node *curNode, node *newNode){
  if (curNode == NULL){
    return newNode;
  }
  if(newNode->value > curNode->value){
    curNode->rightChild = insertBST(curNode->rightChild, newNode);
  }else if(newNode->value < curNode->value){
    curNode->leftChild = insertBST(curNode->leftChild, newNode);
  }
  return curNode;
}
```

Code 6: inserting a node in BST manner.

In Code 6, the insertion function is implemented for BST. Simply, the node which is going to be inserted finds its position by comparing itself with the nodes of the tree starting from the root. If it is smaller than the node it compares the left child of the compared node, if it is bigger than the node then it compares itself with the right child of the node. If the compared node is an empty node then it takes its place.

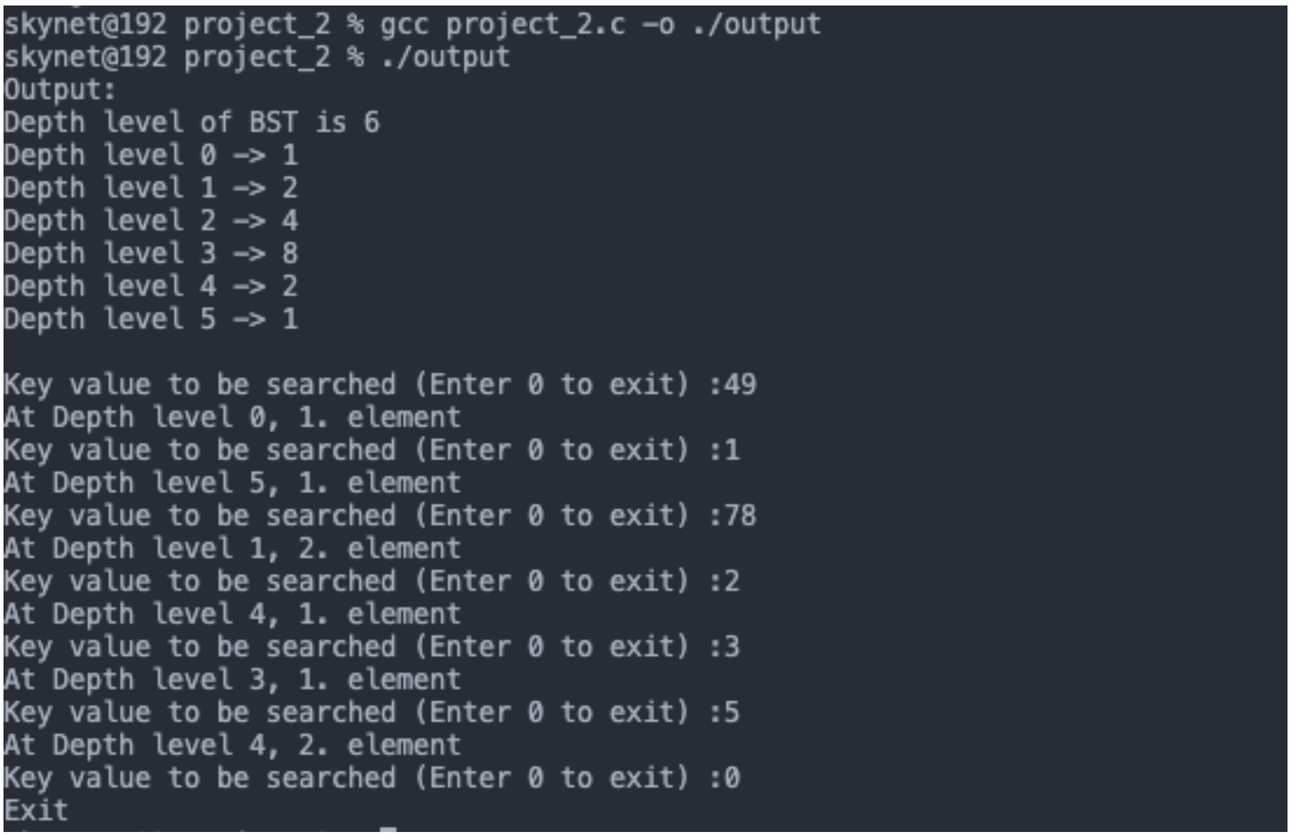


```c
// find the key and update its information about depth and position.
node *findKey(node *currentNode, int key, int position, int depth){
  if(currentNode != NULL){
    if(currentNode->value == key){
      currentNode->position = position;
      currentNode->depthLevel = depth;
      return currentNode;
    }else if (currentNode->value > key){
      return findKey(currentNode->leftChild, key, 2*position-1, depth+1);
    }else if (currentNode->value < key){
      return findKey(currentNode->rightChild, key, 2*position, depth+1);
    }
  }
  return NULL;
}
```

Code 7: Function for Finding a Key.

After building the tree, the user will search nodes in the tree. To be able to do that, a search function is implemented to the system as the findKey() function. The given input from the user is given to the function as key, this function also takes nodes, their position, and depth as its parameters. It compares the key with the value of the node. If it matches, it returns the node by updating its position and depth. If it doesn't match, it compares the key with the node. If the key is smaller than the node, it compares itself with the left child of the node. If it is bigger than the key, it compares itself with the right child of the node. It makes these comparisons, initially starting from the root which has the position of 1 and depth level of 0. When findKey() goes to the left child, it increases the depth level by one, also it multiplies the current position by 2 and subtracts 1 from it. When it goes to the right child it also increases the depth level by one and multiplies the current position by 2. In this way, it will be possible to find matching nodes' positions and depth levels.

To be able to run the program, the input.txt file must be in the same directory. Simply just compile the program and run it.



```
skynet@192 project_2 % gcc project_2.c -o ./output
skynet@192 project_2 % ./output
Output:
Depth level of BST is 6
Depth level 0 -> 1
Depth level 1 -> 2
Depth level 2 -> 4
Depth level 3 -> 8
Depth level 4 -> 2
Depth level 5 -> 1

Key value to be searched (Enter 0 to exit) :49
At Depth level 0, 1. element
Key value to be searched (Enter 0 to exit) :1
At Depth level 5, 1. element
Key value to be searched (Enter 0 to exit) :78
At Depth level 1, 2. element
Key value to be searched (Enter 0 to exit) :2
At Depth level 4, 1. element
Key value to be searched (Enter 0 to exit) :3
At Depth level 3, 1. element
Key value to be searched (Enter 0 to exit) :5
At Depth level 4, 2. element
Key value to be searched (Enter 0 to exit) :0
Exit
```

Figure 4: Example Run of the program.

In Figure 4, the program is running with example inputs. The exact mechanism is implemented in Figure 1.