Istanbul Technical University

Analysis of Algorithm I

BLG 335E

Red Black Tree

Homework 3

Tevfik Özgü

150180082

19 December 2020

CONTENT OF TABLE

FIGURES

EQUATIONS

CODE SNIPPETS

1 INTRODUCTION

In this homework it will be tried to explain red black trees that is a type of binary search tree. In this homework, used dataset that was used is taken from euroleague. In this dataset there are users with their name, team, rebound, assist and point values at the specified season. In the first part red black tree codes will be given and it will be explained line by line. Red black tree will be created by using player names.

In the first part it will be tried to explain the codes as stated before. In this part code sections will be divided and will be explained accordingly.

In the second part initially complexity of the red black tree for insertion and search operations will be explained. After that Red Black Tree will be compared with Binary Search Tree. And finally, there will be tried to implement augmented data structure that keeps both name and position of the player. As an extra data, number of players of each position at the node's childs and itself will be kept and i^{th} player name according to selected position will be found. These procedures will be explained briefly. Pseudocodes will be given for each position searching function.

2 PART 1

In this part, the codes will be explained accordingly. Before that it must be good to explain that red black tree is a type of binary search tree and this RBT is balanced tree. There are some properties that it must hold. These properties are given below.

- Property 1: Every node is either red or black.
- Property 2: Root must be black.
- Property 3: All leaves are always black.
- Property 4: Red nodes child must be black.
- Property 5: All paths from any node to leaves have the same number of black nodes.

2.1 Data Structures of Red Black Tree

Initially file name is read from the terminal and put into the tree by allocating from memory dynamically. Selected data structure is the binary tree which collects left, right and the parent nodes address as a pointer. After that database class which keeps the address of the root, max pointed user, max assisted user and finally max rebounded user. These class codes are given at Code Snippet 1.

```
struct Node{

    Node* left;
    Node* right;
    Node* parent;
    string name;
    bool color;
    int point;
    int rebound;
    int assist;
};

class BasketballDB{
    public:
    Node* root;
    Node* max_point;
    Node* max_assists;
    Node* max_rebs;
    BasketballDB(){
        root = NULL;
        max_point = NULL;
        max_assists = NULL;
        max_rebs = NULL;
    }
};
```

Code Snippet 1. Codes of the Node and the BasketballDB

2.2 Accessing Near Nodes Functions

After creating data structures, it is wanted to implement access methods to parent, grandparent, sibling and uncle.

2.2.1 Get Parent Function

Get parent function which returns address of the parent is given at Code Snippet 2.

```
1 Node* GetParent(Node* &n){
2     if (n->parent){
3         return n->parent;
4     } else {
5         return NULL;
6     }
7 }
```

Code Snippet 2. Get Parent Function

2.2.2 Get Grandparent Function

Get grandparent function which returns address of the grandparent is given at Code Snippet 3.

```
1 Node* GetGrandParent(Node* &n){
2     Node* parent = GetParent(n);
3     return GetParent(parent);
4 }
```

Code Snippet 3. Get Grandparent Function

2.2.3 Get Sibling Function

Get sibling function which returns address of the sibling is given at Code Snippet 4.

```
 1 Node* GetSibling(Node* &n){
 2     Node* parent = GetParent(n);
 3     if (parent != NULL){
 4         if (n == parent->left){
 5             return parent->right;
 6         } else {
 7             return parent->left;
 8         }
 9     } else {
10         return NULL;
11     }
12 }
```

Code Snippet 4. Get Sibling Function

2.2.4 Get Uncle Function

Get uncle function which returns address of the uncle is given at Code Snippet 5.

```
1 Node* GetUncle(Node* &n){
2     Node* parent = GetParent(n);
3     return GetSibling(parent);
4 }
```

Code Snippet 5. Get Uncle Function

2.3 Adding Player to the Tree

In this part, adding function will be explained. This function is called from the main part of the code in the while loop. This function adds node to tree if there is no player called node name. If there is same player in the tree its assist, point and rebound amounts are updated. This function is searching tree by

using recursive calling. Initially it finds the appropriate position according to player name and adds node to there. When new node is added, tree is repaired by using UpdateTree function to fit tree as a Red Black Tree which will be explained next. Adding to the tree function is given at Code Snippet 6.

```
void addPlayer(Node* &root, Node* newNode){

    if (root == NULL){
        root = newNode;
        UpdateTree(newNode);
    } else {
        if (newNode->name != root->name) {
            if (newNode->name < root->name){
                if(root->left != NULL){
                    addPlayer(root->left, newNode);
                    return;
                } else {
                    root->left = newNode;
                    newNode->parent = root;
                    UpdateTree(newNode);
                }
            } else{
                if(root->right != NULL){
                    addPlayer(root->right, newNode);
                    return;
                } else {
                    root->right = newNode;
                    newNode->parent = root;
                    UpdateTree(newNode);
                }
            }
        } else {
            root->assist += newNode->assist;
            root->point += newNode->point;
            root->rebound += newNode->rebound;
            delete newNode;
        }
    }
}
```

Code Snippet 6. Add Player Function

2.4 Updating Tree for Red Black Tree When New Node Is Added

In this function, as explained before when new node is added to the tree, tree must be updated to fit into the red black tree which is a balanced binary search tree. In this approach there are 4 different cases.

2.4.1 Case 1

One of the cases is that if added node is the first node, it becomes black since root must be black.

2.4.2 Case 2

Another case is that if added nodes' parent is black, there will be no operation.

2.4.3 Case 3

Third case is that if new node has an uncle and nodes' uncle is red and parent is red, parent and uncle is painted as black. After that grandparent is painted as red. When this is done there is a problem with the property 2 that says root must be black. So, same update function is called with the grandparent node.

2.4.4 Case 4

As a final case the most complex case will be explained. In this case, parent is red and uncle is black. When this situtation is handled, there will be some rotating operations. Our purpose is that we are want to rotate new node to the grandparent position. But there will be some problems if new node is inside of the subtree. This means that new node is left child of right child of G or right child of left child of grandparent. When this is happened, new node comes the parent of its parent. These operations code are given at Code Snippet 7.

```
void InsertCase4(Node* n) {
  Node* p = GetParent(n);
  Node* g = GetGrandParent(n);

  if (n == p->right && p == g->left) {
    RotateLeft(p);
    n = n->left;
  } else if (n == p->left && p == g->right) {
    RotateRight(p);
    n = n->right;
  }
  InsertCase4Step2(n);
}
```

Code Snippet 7. Case 4 Insert Function

Then after this operation, tree is rotated to left or right. Left rotate and right rotate are given at Code Snippet 8 and Code Snippet 9 respectively.

```
void RotateLeft(Node* n) {
    Node* new_node = n->right;
    Node* parent_node = GetParent(n);
    if (new_node == NULL){
        exit(1);
    }

    n->right = new_node->left;
    new_node->left = n;
    n->parent = new_node;

    if (n->right != NULL) {
        n->right->parent = n;
    }

    if (parent_node != NULL) {
        if (n == parent_node->left) {
            parent_node->left = new_node;
        } else if (n == parent_node->right) {
            parent_node->right = new_node;
        }
    }
    new_node->parent = parent_node;
}
```

Code Snippet 8. Rotating Left Function

```
void RotateRight(Node* n) {
    Node* new_node = n->left;
    Node* p = GetParent(n);

    if (new_node == NULL){
        exit(1);
    }
    n->left = new_node->right;
    new_node->right = n;
    n->parent = new_node;

    if (n->left != NULL) {
        n->left->parent = n;
    }
    if (p != NULL) {
        if (n == p->left) {
            p->left = new_node;
        } else if (n == p->right) {
            p->right = new_node;
        }
    }
    new_node->parent = p;
}
```

Code Snippet 9. Rotating Right Function

2.5 Printing Tree (Preorder)

First season of the dataset is printed in preorder. For this purpose, recursive is used. Other seasons are not printed. Also, while traversing tree, max pointed, assisted and rebounded user is found and kept in a pointer to print at the end. These function codes are given at Code Snippet 10.

```
1   void printPreorder(Node* node, int depth=0)
2   {
3       if (node == NULL)
4           return;
5       if(first_season == 1){
6           string color;
7           for(int i=0; i<depth; i++)
8               cout <<"-";
9           depth++;
10          if (node->color == 0){
11              color = "(BLACK)";
12          } else {
13              color = "(RED)";
14          }
15          cout << color << "  " << node->name << endl;
16      }
17      if (max_point == NULL){
18          max_point = node;
19          max_assists = node;
20          max_rebs = node;
21      }
22
23      if(node->assist > max_assists->assist){
24          max_assists = node;
25      }
26
27      if(node->point > max_point->point){
28          max_point = node;
29      }
30
31      if(node->rebound > max_rebs->rebound){
32          max_rebs = node;
33      }
34
35      printPreorder(node->left, depth);
36
37      printPreorder(node->right, depth);
38  }
39
```

Code Snippet 10. Printing and Finding Max Players Function

3 PART 2

In this part of the homework, complexity of the red black tree, difference between the red black tree and binary search tree will be given. And finally, augmenting data structures will be tried to explain.

3.1 Complexity

In the red black tree, it is said that tree has a height of O(logn) when there are n internal nodes. Now this will be explained briefly.

3.1.1 Searching Complexity

Suppose H(v) is the height of the subtree from the root v and BH(v) is the number of black nodes from v to leaf.

It is said that $2^{BH(v)} - 1$ is the lowest number of internal nodes from root b. Now to prove this induction will be used.

When there is only 1 element which is NULL, H(v) = 0 so $2^0 - 1 = 0$. This is the base step.

If root is v such that H(v) is k has at least $2^{BH(v)} - 1$ internal nodes. This implies that node x such that H(x) = k+1 has at least $2^{BH(x)} - 1$ internal nodes. If child is red, its child will have black height of BH(x) and if child is black it will have black height of BH(x) - 1. Since we are looking for the lowest bound at the proof, it will be supposed that all children are black. In this time all children will have at least $2^{BH(x)-1} - 1$ internal nodes. So, node x has at least addition of its left and right child and itself. This addition is given at Equation 1.

$$2^{BH(x)-1} - 1 + 2^{BH(x)-1} - 1 + 1 = 2^{BH(x)} - 1$$

Equation 1. Total Number of Internal Nodes at Root

Since this equality is hold, we can continue. Since at most half the leaves on any path are red, black height of tree is at least h(root)/2 = BH(root). This equation is given at Equation 2.

$$n \geq 2^{h/2} - 1 \leftrightarrow log(n+1) \geq \frac{h(root)}{2} \leftrightarrow h(root) \leq 2log(n+1)$$

Equation 2. Proving Lemma

From this equation it is clear that height of the root is O(logn). So, when trying to find the node, there is a searching at the upper bound of O(logn). This means upper bound of searching is **O(logn)** for worst case and average case.

3.1.2 Inserting Complexity

When inserting new node, initially it will be searched so O(logn) will be taken. When it is added, coloring will take O(1) time. Then there will be reconstructing whether its properties are violated or not. It takes also O(1) since it involves at max 5 pointer changing. But when this is done there would become 2 red black situations. In the worst-case recoloring will be done from leaf to root that is logn. So, these operations will take O(1*logn) = O(logn). As a result, inserting complexity of worst and average case is O(logn+logn) which comes from searching so it is equal to **O(logn)**.

3.2 RBT vs BST

As explained many times, red black tree is one of the balanced binary search tree. Now let's think that dataset which is 5,4,3,2,1 is given respectively. When this is given and binary search tree is constructed it will seem like given at Figure 1.

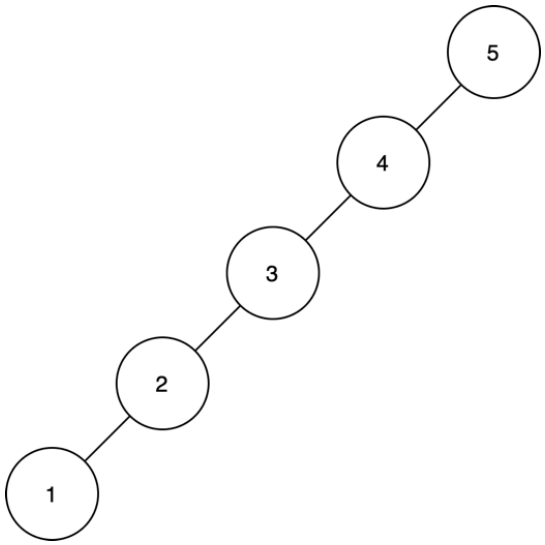

Figure 1. Unbalanced Binary Search Tree

When red black tree structure is constructed, tree will seem like given at Figure 2.

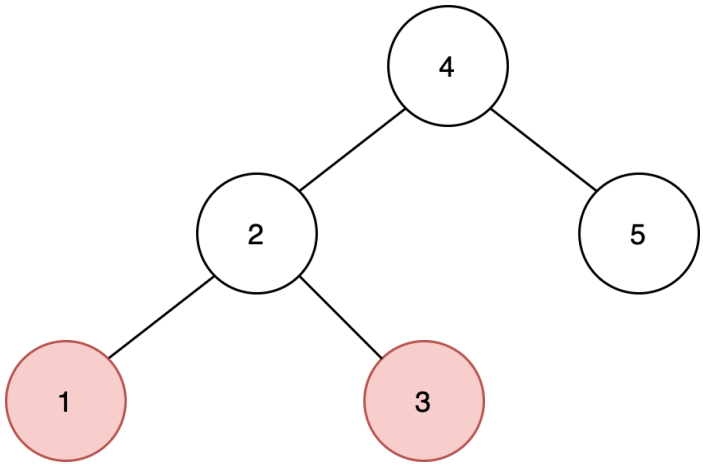

Figure 2. Red Black Tree

It is clear that when tree is constructed as red black tree, structure will be more balanced and it is guaranteed that upper bound of searching is O(logn). But it is not sure that in the binary search tree will be like that. So red black tree is safer and more balanced tree.

3.3 Augmenting Data Structures

In this question it was said that position of the player (Point Guard PG, Shooting Guard SG, Small Forward SF, Power Forward PF or Center C) is given. Then it was asked to find i^{th} PG, SG, SF, PF and Center players. Also, it is said that these positions are augmented with my Red-Black Tree.

When these datas are augmented, I would add num_of_pg, num_of_sg, num_of_sf, num_of_pf and num_of_c data which is integer to my nodes. In these values of each node, I would keep the number of Point Guard, Shooting Guard, Small Forward, Power Forward and Center players that it has under it and added to 1 which is itself. So, I mean that if there are 3 Point Guard players at its under level, I would store 3 or 4 in its num_of_pg whether it is Point Guard player or not. Like this example I would keep number of Shooting Guard, Small Forward, Power Forward and Center players in the num_of_sg, num_of_sf, num_of_pf and num_of_c variables of each node.

When these datas are stored it is easy to implement search function. Pseudocodes of each function is given below consequtively. In these functions only difference is that selected data variable is changed. When Point Guard is searched, r became left childs num_of_pg and additionally 1 if its type is Point Guard. Others are repeated according to its type.

```
    OS-SELECT_PG(x,i)
1   r = x.left.num_of_pg
2   if x.left == "PG"
3      r += 1
4   if i == r
5      return x.name
6   elseif i < r
7      return OS-SELECT_PG(x.left,i)
8   else return OS-SELECT(x.right,i)
```

Pseudocode 1. Finding Point Guard

```
    OS-SELECT_SG(x,i)
1   r = x.left.num_of_sg
2   if x.left == "SG"
3      r += 1
4   if i == r
5      return x.name
6   elseif i < r
7      return OS-SELECT_SG(x.left,i)
8   else return OS-SELECT(x.right,i)
```

Pseudocode 2. Finding Shooting Guard

```
    OS-SELECT_SF(x,i)
1   r = x.left.num_of_sf
2   if x.left == "SF"
3      r += 1
4   if i == r
5      return x.name
6   elseif i < r
7      return OS-SELECT_SF(x.left,i)
8   else return OS-SELECT(x.right,i)
```

Pseudocode 3. Finding Small Forward

```
    OS-SELECT_PF(x,i)
1   r = x.left.num_of_pf
2   if x.left == "PF"
3      r += 1
4   if i == r
5      return x.name
6   elseif i < r
7      return OS-SELECT_PF(x.left,i)
8   else return OS-SELECT(x.right,i)
```

Pseudocode 4. Finding Power Forward

```
    OS-SELECT_C(x,i)
1   r = x.left.num_of_c
2   if x.left == "C"
3      r += 1
4   if i == r
5      return x.name
6   elseif i < r
7      return OS-SELECT_C(x.left,i)
8   else return OS-SELECT(x.right,i)
```

Pseudocode 5. Finding Center

In these pseudocodes, as explained earlier, initially how many players which are selected type position is found. And then checked whether itself is that position player or not and added 1 if it is. In the fourth lines, checked whether player found or not. In the sixth line if i is lower it is understood that there are more player than its selection so position moved to left else it is moved to right. As explained, in the fifth line selected user's name is returned.

NOTE: In my design, different positions are handled from different functions. There is a possibility to handle in same function but I thought that it will be more suitable to do like this since there won't be redundant if else statement and also sending position recursively again and again.