Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

BLG 335E
Analysis of Algorithms I Homework 3

Hasan Fatih Durkaya - 150200074

December 28nd, 2022

Contents

1 Introduction

In this homework, we are asked to create completely fair scheduler with using some sort of red-black tree implementation. First we should define what is a completely fair scheduler. A completely fair scheduler is a method used by computer operating systems to fairly distribute CPU resources to processes, allowing all processes to have equal access to the CPU. The scheduler assigns each process a ”virtual runtime” which determines their priority and how much CPU time they are allocated. The virtual runtime is adjusted based on the actual CPU usage of each process to ensure a fair distribution of resources over time. The main benefit of using a completely fair scheduler is that it helps prevent a single process from using a large amount of CPU resources and negatively impacting the performance of other processes. Instead, it ensures that all processes have a fair opportunity to utilize the CPU and improves the overall performance of the system. To implement a completely fair scheduler we have created classes and used some data structures.

2 Description of Code

2.1 Data Structurs

In this homework I have used mainly vectors and classes that i have created.

2.1.1 Vectors

A vector in C++ is a container class that is part of the Standard Template Library (STL). It allows for the storage of elements in a contiguous block of memory and provides functions for adding and removing elements, accessing and modifying elements, and performing other common operations. Vectors are useful for storing and manipulating sequences of elements in a dynamic way. They can be accessed and modified as needed and allow for fast element access and efficient memory allocation.
In this homework I have created 2 vectors one for storing the processes which is a Node* vector (process_list) and one for the storing the finished process which is a string vector (list_finished_process).

Vectors used

```
vector<Node*> process_list;
vector<string> list_finished_process;
```

process_list holds all the process and when their arrival time has come they will be sended to tree with their allocated time , name and arrival time values. list_finished_process vector holds the names of the process that is finished.

2.2 Implementation

In this part, I am going to analyze the implementation by going over the files; RBTree.h , RBTree.cpp and last but not least main.cpp .

2.2.1 RBTree.h

RBTree.h is the header file that my classes has been defined. There are 2 types of classes one for the represent nodes of red black tree which is called Node and the other to represent red-black tree itself with its member functions and getters, setters.

Node Class

```
class Node{

    private:

        string name;
        int arrival_time;
        int virtual_run_time;
        int allocated_time;
        string colour;

        Node* right_child;
        Node* left_child;
        Node* parent;

    public:

        Node(string name, int arrival_time ,int allocated_time);
        void set_alocated_time(int alocated_time);
        void increase_vrun_time();
        void set_vrun_time(int virtual_run_time);
        void set_colour(string colour);
        int get_alocated_time();
        int get_arrival_time();
        string get_name();
        int get_virtual_run_time();
        string get_colour();
        void set_left_child(Node* node);
        void set_right_child(Node* node);
        void set_parent(Node* node);
        Node* get_right_child();
        Node* get_left_child();
        Node* get_parent();
};
```

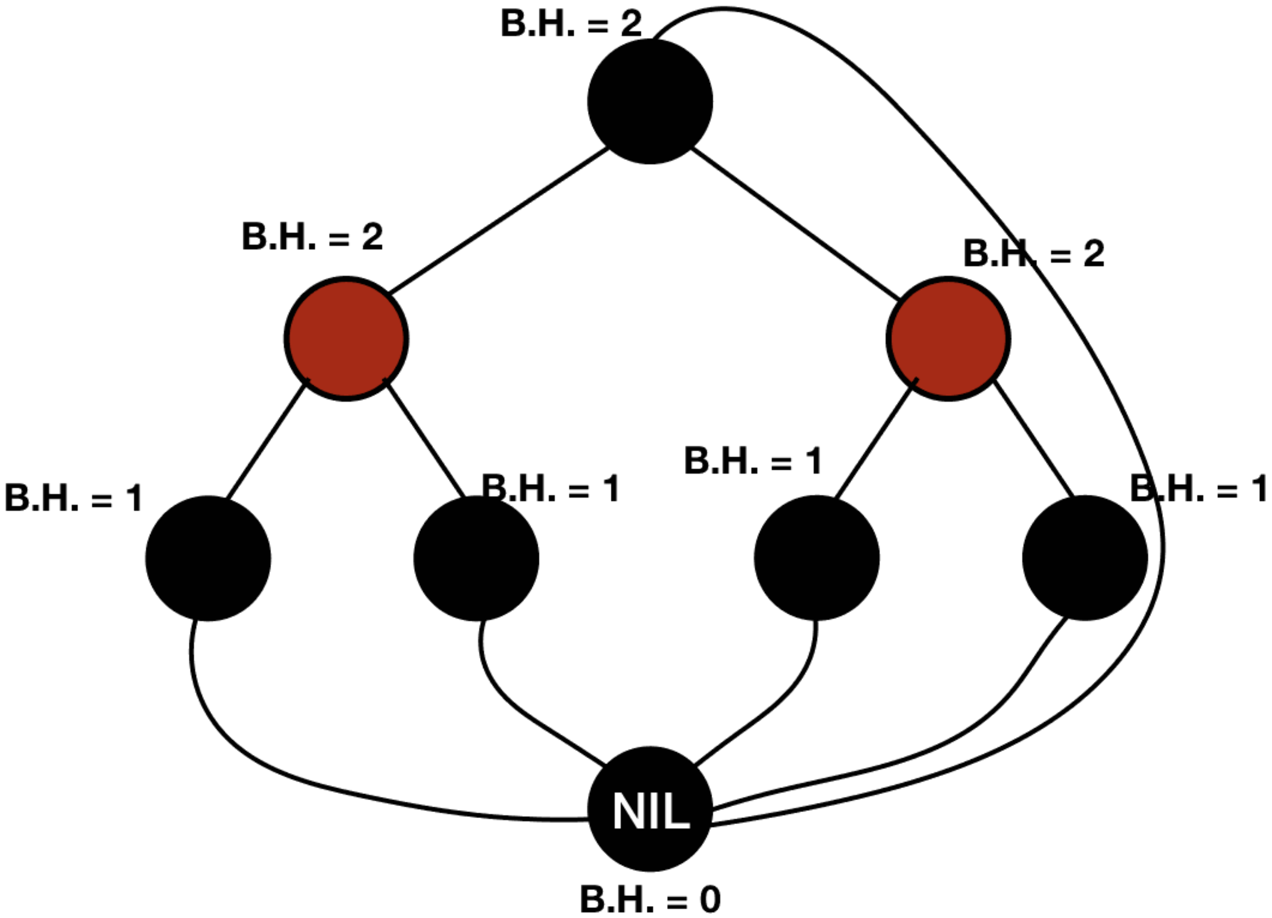
In private part I am creating the data variables that we should store to implement red black tree and features. In the public part, there are getters and setters for data attributes and pointers for object oriented approach. Besides getter and setters there are 2 member functions that is important.

First one is the increase_vrun_time this function is only responsible for increasing the virtual run time of the selected nodes by one. Other one is constructor. Constructor will be explained in a more detailed way in RBTree.cpp .
Other class definiton in RBTree.h is RBTree class which represents the red black tree.

RBTree Class

```
class RBTree{

    private:
        Node* root;
        Node* NIL;

    public:

        RBTree();
        Node* get_root();
        Node* get_nil();
        void set_root(Node* new_root);

        void rotate_left(Node* secondary_node);
        void rotate_right(Node* secondary_node);
        void reorder_insert(Node* secondary_node);
        void reorder_delete(Node* secondary_node);

        void RBinsert(Node* new_node);
        void RBDelete(Node* target_node);
        void transplant(Node* node1, Node* node2);
        void inorder_traversal(ofstream &myfile , Node* root);
        Node* get_left_most(Node* root);
        Node* get_right_most(Node* root);

};
```

In private part, root and nill nodes have been created. Root will be the first and nil will be the last element for each path. A nil node is a placeholder for leafs. It is sued for ease of simplification for implementations. Nil nodes are always black. Nil node has no data attributes or mainly an unusued value will be used. They are responsible for representing the end of a path in a tree. As can be seen in the below figure all paths are starting with root and ending with nil.



In public there aresome setters and getters to access root and nils and a constructor which will be covered lated. Important part for this section is the insertidon, deletion, reordering and rotation functions which is define here.

2.2.2 RBTree.cpp

In this part, I am going to cover the functions that have been defined above and what they do. On the other hand, I am going to analyze their complexity for individual functions.
Node constructor is getting 3 inputs for name, arrival time and allocated time and setting these values. Virtual run time is set to 0 by default because every node when its created has 0 processed time. Childs and parent are set to NULL and colour will be red when first created because after the creation of a node there will be an insertion and nodes that will be inserted must be red after insertion there will be reordering. The complexity of this function is O(1) because all operations have constant time and there are not any loops.

Node Constructor

```
Node::Node(string name, int arrival_time ,int allocated_time){
    this->virtual_run_time = 0;
    this->name = name;
    this->arrival_time = arrival_time;
    this->alocated_time = alocated_time;
    this->set_left_child(NULL);
    this->set_right_child(NULL);
```

```
    this->set_parent(NULL);
    this->colour = "Red";
}
```

There are some getters and setters to access private elements of classes. Example code block can be seen below. The complexity of these functions is O(1) because all operations have constant time and there are not any loops.

Example of Getters and Setters

```
int Node::get_virtual_run_time(){
    return this->virtual_run_time;
}

string Node::get_colour(){
    return this->colour;
}

void Node::set_right_child(Node* node){
    this->right_child = node;
}

void Node::set_left_child(Node* node){
    this->left_child = node;
}
```

Red black tree constructor called RBTree is creating a node with values (”nil”, 0, 0) because this will be the respresentation of nil nodes and nil nodes colour is set to black by definiton. NIL node of the tree will have the nill values that have created in contructor and root will be set to nil for creation of tree. The complexity of this function is O(1) because all operations have constant time and there are not any loops.

Red Black Tree Constructor

```
RBTree::RBTree(){

    Node* NIL_node = new Node("nil",0,0);
    NIL_node->set_colour("Black");

    this->NIL = NIL_node;
    this->root = this->NIL;
}
```

Insertion operation for red black trees are different from the normal trees because normal insertion will cause some violations of red black tree properity because of that we have to reorder if there are any violations after insertion.

Algorithm 1 RBTree Insertion

```
function RBINSERT(new_node)
     node = nil
     temp = root
    while temp ≠ nil do
        node ← temp
        if new_node.data > temp.data then
            temp ← temp.left
        else if new_node.data < temp.data then
            temp ← temp.right
        else if new_node.data == temp.data then
            if new_node.arrival > temp.arrival then
                temp ← temp.left
            else
                temp ← temp.right
            end if
        end if
    end while
    new_node.parent ← node
    if node == nil then
        root ← new_node
    else if new_node.data < node.data then
        node.left ← new_node
    else if new_node.data > node.data then
        node.right ← new_node
    else if new_node.data == node.data then
        if new_node.arrival > node.arrival then
            node.left ← new_node
        else
            node.right ← new_node
        end if
    end if
    new_node.left ← nil
    new_node.right ← nil
    function REORDER_INSERT(new_node)
    end function
```

This is the RBTree's modified insert function, data refers to virtual run time for our case. First of all we are creating two temporary variables temo and node. Since this is a member function of RBTree we can directly access to nil and root we do not have to take tree as input. Node is set to nil and temp is set to root. While temp is not equal nil, we are deciding which path that we should insert .Therefore we are checking with if statements that new_node's virtual run time is greater than root if it is we are going right if not we are going left. When we reach nil with temp we are moving to next step , setting new_node's parent as node after that we are checking where we should insert the new node according to it's parent. After that we are setting it's left and right child as nil. This inserting operation may cause violation of red-black tree properities therefore we should call reordering functions. When we check for the complexity we will see that complexity is O(logn) because height of the tree will be upper bounded by logn and this insert operation contains only one loop. All of the other cases that does not goes in to loop are constant time, therefore best case may be better but thw worst case analysis of insert is O(logn). (Below backslash character is used to represent line continuation.)

Red Black Tree Insertion

```
void RBTree::RBinsert(Node* new_node){

    Node* temp_node = this->get_root();
    Node* node = this->get_nil();

    while(temp_node != this->get_nil()){

        node = temp_node;

        if(new_node->get_virtual_run_time() < temp_node->\
            get_virtual_run_time()){
            temp_node = temp_node->get_left_child();
        }
        else if (new_node->get_virtual_run_time() > temp_node->\
            get_virtual_run_time()){
            temp_node = temp_node->get_right_child();
        }
        else if(new_node->get_virtual_run_time() == temp_node->\
            get_virtual_run_time()){

            if(new_node->get_arrival_time() < temp_node->\
                get_arrival_time()){
                temp_node = temp_node->get_left_child();
            }
            else{
                temp_node = temp_node->get_right_child();
            }
        }
    }
    new_node->set_parent(node);

    if(node == this->get_nil()){
        this->set_root(new_node);
    }
    else if(new_node->get_virtual_run_time() < node->\
        get_virtual_run_time()){
        node->set_left_child(new_node);
    }
    else if(new_node->get_virtual_run_time() > node->\
        get_virtual_run_time()){
        node->set_right_child(new_node);
```

```
    }
    else if(new_node->get_virtual_run_time() == node->\
        get_virtual_run_time()){

        if(new_node->get_arrival_time() < node->get_arrival_time()){
            node->set_left_child(new_node);
        }
        else{
            node->set_right_child(new_node);
        }
    }

    new_node->set_left_child(this->get_nil());
    new_node->set_right_child(this->get_nil());

    this->reorder_insert(new_node);

}
```

Algorithm 2 Red-Black Tree Reorder Insertion

```
function REORDER_INSERT(new_node)
    while new_node.parent.color = Red do
        if new_node.parent = new_node.parent.parent.left then
            temp_node ← new_node.parent.parent.right
            if temp_node.color = Red then
                new_node.parent.color ← Black
                temp_node.color ← Black
                new_node.parent.parent.color ← Red
                new_node ← new_node.parent.parent
            else
                if new_node = new_node.parent.right then
                    new_node ← new_node.parent
                    rotate_left(new_node)
                end if
                new_node.parent.color ← Black
                new_node.parent.parent.color ← Red
                rotate_right(new_node.parent.parent)
            end if
        else
            temp_node ← new_node.parent.parent.left
            if temp_node.color = Red then
                new_node.parent.color ← Black
                temp_node.color ← Black
                new_node.parent.parent.color ← Red
                new_node ← new_node.parent.parent
            else
                if new_node = new_node.parent.left then
                    new_node ← new_node.parent
                    rotate_right(new_node)
                end if
                new_node.parent.color ← Black
                new_node.parent.parent.color ← Red
                rotate_left(new_node.parent.parent)
            end if
        end if
    end while
    root.color ← Black
end function=0
```

Reorder_insert function is a set of steps that are followed after inserting a new node into a red-black tree in order to maintain the balance of the tree. These steps may include rotating the tree and adjusting the colours of nodes to meet the red-black tree properties, which include the requirement that the root is black, leaves be black, and red nodes have black children. The purpose of the reorder_insert function is to maintain the balance of the tree, which allows for efficient search, insertion, and deletion operations. Time complexity of reorder_insert function is O(log n), where n is the number of nodes in the tree. This is because the reorder function may involve a series of rotations and color changes, but the number of these operations is limited by the height of the tree. In a red-black tree, the height is at most 2*log(n+1), so the time complexity is O(log n).

Inserting a new node's complexity was O(logn) and reordering after insertion is O(logn) to so overall insertion complexity is O(logn).

Red Black Tree Reorder Insertion

```
void RBTree::reorder_insert(Node* new_node){

    while(new_node->get_parent()->get_colour() == "Red"){

        if(new_node->get_parent() == new_node->get_parent()->\
            get_parent()->get_left_child()){

            Node* temp_node = new_node->get_parent()->get_parent()->\
                get_right_child();

            if(temp_node->get_colour() == "Red"){
                new_node->get_parent()->set_colour("Black");
                temp_node->set_colour("Black");
                new_node->get_parent()->get_parent()->set_colour("Red");
                new_node = new_node->get_parent()->get_parent();
            }
            else{
                if(new_node == new_node->get_parent()->get_right_child()){
                    new_node = new_node->get_parent();
                    this->rotate_left(new_node);
                }
                new_node->get_parent()->set_colour("Black");
                new_node->get_parent()->get_parent()->set_colour("Red");
                this->rotate_right(new_node->get_parent()->get_parent());
            }
        }
        else{

            Node* temp_node = new_node->get_parent()->get_parent()->\
                get_left_child();

            if(temp_node->get_colour() == "Red"){
                new_node->get_parent()->set_colour("Black");
                temp_node->set_colour("Black");
                new_node->get_parent()->get_parent()->set_colour("Red");
                new_node = new_node->get_parent()->get_parent();
            }
            else{
                if(new_node == new_node->get_parent()->get_left_child()){
                    new_node = new_node->get_parent();
                    this->rotate_right(new_node);
                }
                new_node->get_parent()->set_colour("Black");
                new_node->get_parent()->get_parent()->set_colour("Red");
                this->rotate_left(new_node->get_parent()->get_parent());
            }
```

```
        }
    }

    this->get_root()->set_colour("Black");

}
```

Algorithm 3 Left Rotation

```
function ROTATE_LEFT(node)
    temp_node ← node.right
    node.right ← temp_node.left
    if temp_node.left == nil then
        temp_node.left.parent = node
    end if
    temp_node.parent = node.parent
    if node.parent == nil then
        root = temp_node
    else if node == node.parent.left then
        node.parent.left = temp_node
    else
        node.parent.right = temp_node
    end if
    temp_node.left = node
    node.parent = temp_node
end function
```

Left rotation is an operation that is used to balance the tree and maintain its properties. It involves rotating the tree around a specific node, called the pivot node, in a counterclockwise direction. This has the effect of moving the pivot node and its right child to the left and shifting its left child to the right. The time complexity of left rotation is O(1) because it is constant time do not contain any loops just updating some pointers.

Left Rotation

```
void RBTree::rotate_left(Node* secondary_node){

    Node* node = secondary_node->get_right_child();
    secondary_node->set_right_child(node->get_left_child());

    if(node->get_left_child() != this->get_nil()){
        node->get_left_child()->set_parent(secondary_node);
    }
    node->set_parent(secondary_node->get_parent());

    if(secondary_node->get_parent() == this->get_nil()){
        this->set_root(node);
    }
    else if(secondary_node == secondary_node->get_parent()->\
        get_left_child()){
```

```
        secondary_node->get_parent()->set_left_child(node);
    }
    else{
        secondary_node->get_parent()->set_right_child(node);
    }
    node->set_left_child(secondary_node);
    secondary_node->set_parent(node);

}
```

Algorithm 4 Right Rotation

function ROTATE_RIGHT($node$)
    $temp\_node \leftarrow node.left$
    $node.left \leftarrow temp\_node.right$
    if $temp\_node.right == nil$ **then**
        $temp\_node.right.parent = node$
    end if
    $temp\_node.parent = node.parent$
    if $node.parent == nil$ **then**
        $root = temp\_node$
    else if $node == node.parent.right$ **then**
        $node.parent.right = temp\_node$
    else
        $node.parent.left = temp\_node$
    end if
    $temp\_node.right = node$
    $node.parent = temp\_node$
end function

Right rotation operation is same as left rotation only difference is now rotation is in clockwise direction.Time complexity is again O(1).

Right Rotation

```
void RBTree::rotate_right(Node* secondary_node){

    Node* node = secondary_node->get_left_child();
    secondary_node->set_left_child(node->get_right_child());

    if(node->get_right_child() != this->get_nil()){
        node->get_right_child()->set_parent(secondary_node);
    }
    node->set_parent(secondary_node->get_parent());

    if(secondary_node->get_parent() == this->get_nil()){
        this->set_root(node);
    }
    else if(secondary_node == secondary_node->get_parent()->\
        get_right_child()){
        secondary_node->get_parent()->set_right_child(node);
```

```
    }
    else{
        secondary_node->get_parent()->set_left_child(node);
    }
    node->set_right_child(secondary_node);
    secondary_node->set_parent(node);

}
```

Algorithm 5 Red-Black Tree Deletion

```
function RB-DELETE(target_node)
    temp ← target_node
    tempOriginalColor ← temp.color
    temp_node ← nil
    if target_node.left = nil then
        temp_node ← target_node.right
        transplant(target_node, target_node.right)
    else if target_node.right = nil then
        temp_node ← target_node.left
        transplant(target_node, target_node.left)
    else
        temp ← get_left_most(target_node.right)
        tempOriginalColor ← y.color
        temp_node ← temp.right
        if temp.parent = target_node then
            temp_node.parent ← temp
        else
            transplant(temp, temp.right)
            temp.right ← target_node.right
            temp.right.parent ← temp
        end if
        transplant(target_node, temp)
        temp.left ← target_node.left
        temp.left.parent ← temp
        temp.color ← target_node.color
    end if
    if tempOriginalColor = Black then
        reorder_delete(temp_node)
    end if
end function
```

The deletion operation is a process used to remove a node from a red-black tree while maintaining the balance of the tree. The operation involves finding the node to be deleted and replacing it with another node and then adjusting the tree's structure and colors to maintain the red-black tree properties. The time complexity of this operation is logarithmic, meaning that it increases at a slower rate as the number of nodes in the tree increases. This makes the red-black tree deletion operation an efficient way to delete a node from the tree. The time complexity of this functions is O(logn).

Red Black Tree Deletion

```
void RBTree::RBDelete(Node* target_node){

    Node* temp = target_node;
    Node* temp_node;
    std::string temp_color = temp->get_colour();

    if(target_node->get_left_child() == this->get_nil()){

        temp_node = target_node->get_right_child();
        transplant(target_node, target_node->get_right_child());

    }
    else if(target_node->get_right_child() == this->get_nil()){

        temp_node = target_node->get_left_child();
        transplant(target_node, target_node->get_left_child());
    }
    else{
        temp = get_left_most(target_node->get_right_child());
        temp_color = temp->get_colour();
        temp_node = temp->get_right_child();

        if(temp->get_parent() == target_node){
            temp_node->set_parent(target_node);
        }
        else{
            transplant(temp, temp->get_right_child());
            temp->set_right_child(target_node->get_right_child());
            temp->get_right_child()->set_parent(temp);
        }
        transplant(target_node, temp);
        temp->set_left_child(target_node->get_left_child());
        temp->get_left_child()->set_parent(temp);
        temp->set_colour(target_node->get_colour());
    }
    if(temp_color == "Black")
        reorder_delete(temp_node);

}
```

Algorithm 6 Red-Black Tree Reorder Delete

```
function REORDER_DELETE(node)
    while node ≠ root and node.color = BLACK do
        if node = node.parent.left then
            new_node ← node.parent.right
            if new_node.color = RED then
                new_node.color ← BLACK
                node.parent.color ← RED
                rotate_left(node.parent)
                new_node ← node.parent.right
            end if
            if new_node.left.color = BLACK and new_node.right.color = BLACK then
                new_node.color ← RED
                node ← node.parent
            else
                if new_node.right.color = BLACK then
                    new_node.left.color ← BLACK
                    new_node.color ← RED
                    rotate_right(w)
                    new_node ← node.parent.right
                end if
                new_node.color ← node.parent.color
                node.parent.color ← BLACK
                new_node.right.color ← BLACK
                leftRotate(node.parent)
                node ← T.root
            end if
        else
            if node = node.parent.right then
                new_node ← node.parent.left
                if new_node.color = RED then
                    new_node.color ← BLACK
                    node.parent.color ← RED
                    rotate_right(node.parent)
                    new_node ← node.parent.left
                end if
                if new_node.right.color = BLACK and new_node.left.color = BLACK then
                    new_node.color ← RED
                    node ← node.parent
                else
                    if new_node.left.color = BLACK then
                        new_node.right.color ← BLACK
                        new_node.color ← RED
                        rotate_left(new_node)
                        new_node ← node.parent.left
                    end if
                    new_node.color ← node.parent.color
                    node.parent.color ← BLACK
                    new_node.left.color ← BLACK
                    rotate_right(node.parent)
                    node ← T.root
                end if
            end if
```

The reorder_delete is used to restore the balance of the tree after a node is deleted. When a node is removed from the tree, it can cause the tree to become unbalanced, violating one or more of the red-black tree properties. The reordering is used to restore these properties and ensure that the tree remains balanced.

reordering starts at the replacement node (which is the node that took the place of the deleted node) and works its way up to the root of the tree, adjusting the colors of the nodes and performing rotations as needed to maintain the red-black tree balance.

The time complexity of the reordering in red-black tree deletion is O(logn), where n is the number of nodes in the tree. This is because the reordering involves traversing up the tree from the replacement node to the root, and the height of a red-black tree is O(logn).

Red Black Tree Reorder Deletion

```
void RBTree::reorder_delete(Node* secondary_node){

    while(secondary_node != this->get_root() && secondary_node->\
        get_colour() =="Black"){

        if(secondary_node == secondary_node->get_parent()->\
            get_left_child()){

            Node* new_node = secondary_node->get_parent()->\
                get_right_child();

            if(new_node->get_colour() == "Red"){
                new_node->set_colour("Black");
                secondary_node->get_parent()->set_colour("Red");
                rotate_left(secondary_node->get_parent());
                new_node = secondary_node->get_parent()->get_right_child();
            }
            if(new_node->get_left_child()->get_colour() == "Black"\
                && new_node->get_right_child()->get_colour() == "Black"){
                new_node->set_colour("Red");
                secondary_node = secondary_node->get_parent();
            }
            else{
                if(new_node->get_right_child()->get_colour() == "Black"){
                    new_node->get_left_child()->set_colour("Black");
                    new_node->set_colour("Red");
                    rotate_right(new_node);
                    new_node = secondary_node->get_parent()->\
                        get_right_child();
                }
                new_node->set_colour(secondary_node->get_parent()\
                    ->get_colour());
                secondary_node->get_parent()->set_colour("Black");
                new_node->get_right_child()->set_colour("Black");
                rotate_left(secondary_node->get_parent());
                secondary_node = this->get_root();
```

```
            }
        }
        else{

            Node* new_node = secondary_node->get_parent()->get_left_child()

            if(new_node->get_colour() == "Red"){

                new_node->set_colour("Black");
                secondary_node->get_parent()->set_colour("Red");
                rotate_right(secondary_node->get_parent());
                new_node = secondary_node->get_parent()->get_left_child();

            }

            if(new_node->get_right_child()->get_colour() == "Black"\
                && new_node->get_left_child()->get_colour() == "Black"){
                new_node->set_colour("Red");
                secondary_node = secondary_node->get_parent();
            }
            else{
                if(new_node->get_left_child()->get_colour() == "Black"){
                    new_node->get_right_child()->set_colour("Black");
                    new_node->set_colour("Red");
                    rotate_left(new_node);
                    new_node = secondary_node->get_parent()->\
                        get_left_child();
                }
                new_node->set_colour(secondary_node->get_parent()\
                    ->get_colour());
                secondary_node->get_parent()->set_colour("Black");
                new_node->get_left_child()->set_colour("Black");
                rotate_right(secondary_node->get_parent());
                secondary_node = this->get_root();
            }

        }
    }
    secondary_node->set_colour("Black");

}
```

Algorithm 7 Red-Black Tree Transplant

function TRANSPLANT($node1, node2$)
    if $node1.parent = NIL$ **then**
        $root \leftarrow node2$
    else if $node1 = node1.parent.left$ **then**
        $node1.parent.left \leftarrow node2$
    else
        $node1.parent.right \leftarrow node2$
    end if
    if $node2 \neq NIL$ **then**
        $node2.parent \leftarrow node1.parent$
    end if
end function=0

The transplant operation is used to replace one subtree with another. It is often used in conjunction with other operations, such as deletion or insertion, to reorganize the tree and maintain its balance. The time complexity of the transplant operation in a red-black tree is O(1), since it only involves a constant number of pointer assignments and does not involve any traversals or comparisons.

Red Black Tree Transplant

```
void RBTree::transplant(Node* node1, Node* node2){

        if(node1->get_parent() == this->get_nil()){
            this->set_root(node2);
        }
        else if(node1 == node1->get_parent()->get_left_child()){
            node1->get_parent()->set_left_child(node2);
        }
        else{
            node1->get_parent()->set_right_child(node2);
        }
        node2->set_parent(node1->get_parent());
}
```

Listed above the most crucial functions of this implementation but they are not all, there are some helper functions that we have created.
First helper function is the inorder traversal function, this helps implementation to see what the values of tree are in a inordered way. Inorder traversal has O(n) complexity because it has to traverse every node.

Inorder Traversal

```
void RBTree::inorder_traversal(ofstream &myfile, Node *root){

    if(root != this->get_nil()){
        inorder_traversal(myfile,root->get_left_child());

        cout<<root->get_name()<<":"<<root->get_virtual_run_time()\
```

```
            <<"-"<<root->get_colour();
        myfile<<root->get_name()<<":"<<root->get_virtual_run_time()\
            <<"-"<<root->get_colour();
        if(root != this->get_right_most(this->get_root())){
            cout<<";";
            myfile<<";";
        }
        inorder_traversal(myfile,root->get_right_child());
    }

}
```

There are get_left_most and get_right_most functions that have been declared. Their purpose is to reach the edge nodes of tree and complexity is proportional to height which is O(logn).

Get Right Most

```
Node* RBTree::get_right_most(Node* root){

    while(root->get_right_child() != this->get_nil())
        root = root->get_right_child();

    return root;
}
```

Get Left Most

```
Node* RBTree::get_left_most(Node* root){

    while(root->get_left_child() != this->get_nil())
        root = root->get_left_child();

    return root;
}
```

2.2.3 main.cpp

Main file is the part that I have made reading from the file and writing to file parts.And also inserting new nodes, deleting finished processes and deleting and inserting the same node. Main can be analyzed top to bottom as follows.

Reading From File

```
    getline(fin, number_of_process_str,'␣');
    getline(fin, time_str);
    number_of_process = stoi(number_of_process_str);
    time = stoi(time_str);

    for(int i = 0; i < number_of_process; i++){
        getline(fin, name,'␣');
```

```
        getline(fin, arrival_time_str,'␣');
        getline(fin, alocated_time_str);

        arrival_time = stoi(arrival_time_str);
        alocated_time = stoi(alocated_time_str);

        Node* new_node = new Node(name, arrival_time, alocated_time);

        process_list.push_back(new_node);
    }
```

This is the part where reading from file and pushing into a vector to store processes. These processes will be inserted to tree when their arrival time has come.

Time loop

```
    for(int i = 0; i < time; i++){

        if(is_complete == true)
            break;

        cout<<i;
        myfile << i;
        for(int k = 0; k < number_of_process; k++){
            if(process_list[k]->get_arrival_time() == i){
                tree.RBinsert(process_list[k]);
            }
        }
```

Above loop is the general structure of time line, checking the process vector to find if there are any nodes that must be inserted to tree. I have defined a special node called CPU Node to use it as node that has been working in GPU but this node will not be in tree .Values of this node will simultaneously change and according to values a new node created from this node will be inserted or not. Therefore this node is only will be used for checking if the conditions are satisfied or not.

Checking CPU Node

```
if(tree.get_root() != tree.get_nil() && CPU_node == NULL){

    if(tree.get_left_most(tree.get_root())->get_right_child() ==\
        tree.get_nil()){

        CPU_node = tree.get_left_most(tree.get_root());
```

This is the start of the main process, main is divided into 2 parts . First one is the reading and second one is the processing. Processing is divided into 2 parts as can be seen from the second if .We are checking the left most element which has the highest priority will be processed and after that process according to its neighbours values we may reprocess the same node again or we may delete the node from the tree and insert it again to go right subtree or we may just finish that process. Procedure is always same but the operations will may change. Complexity

analysis of main part will be considered as sum of all operations. On the other hand mainly all operations are constant time in main except some of the special ones . Main file does not contain any loop inside the loop structures and biggest loop is the main loop that contains the time line. It's complexity is directly proportional to time amount. Complexity analysis of all implementation will be covered lately but above individual complexities have been analyzed.
Checking the Virtual Run Time

```
if(CPU_node->get_virtual_run_time() < CPU_node->get_alocated_time()){

    if(CPU_node == tree.get_root()){
        tree.set_root(CPU_node);
    }

    else if(CPU_node->get_virtual_run_time() > CPU_node->\
        get_parent()->get_virtual_run_time()){

        Node* new_node = new Node(CPU_node->get_name(), CPU_node->\
        get_arrival_time(), CPU_node->get_alocated_time());
        new_node->set_vrun_time(CPU_node->get_virtual_run_time());
        tree.RBDelete(tree.get_left_most(tree.get_root()));
        tree.RBinsert(new_node);
    }

        CPU_node = NULL;
}
else if(CPU_node->get_virtual_run_time() == CPU_node->get_alocated_time()){

    tree.RBDelete(tree.get_left_most(tree.get_root()));

    list_finished_process.push_back(CPU_node->get_name());
    finished_process++;
    CPU_node = NULL;
}
```

There are 2 cases which we are interested in. First is virtual run time of the currently processing node is not greater than the allocated time and in this case we either just increase the virtual run time and do not any delete insert operation because it's not greater than it's parent or right child. Other case is the currently processing node's virtual run time is greater than parent or right child and in this case we must delete and using the nodes values we must insert a new node with updated values. General topics have been covered but code will be displayed in a more detailed way. For the sake o simplcity comments in the code blocks have been removed in report.

2.3 Complexity Analysis

This part will be more short because complexity analysis of individual functions have already been covered in description of code section. In this part we are going to analyze all code blocks as a whole.
We are going to seperate into the different parts and analyze the complexity because in the

main part there will or will not be any inseriton or deletion depending on the input.

2.3.1 Complexity of Insertion

The time it takes to insert an element into a red-black tree is typically logarithmic in the number of elements already in the tree. This is because red-black trees are self-balancing, meaning that they are designed to maintain a logarithmic height regardless of the order in which elements are inserted. Insertion into a red-black tree involves adding a new leaf node and, occasionally, adjusting the colors of nodes and performing rotations to maintain balance. These adjustments and rotations take time, but they are relatively quick and occur infrequently, so the overall time complexity of red-black tree insertion is O(log n). As a result insertion operation has the time complexity O(logn) and after that it may be necessary to make reordering (fixing the violations) which has a time complexity O(logn) so overall time complexity for insertion is O(logn).

2.3.2 Complexity of Deletion

For deletion operaiton specifically in this homework we are only deleting from the left most node and accesing to that node is proportional do height of the tree which is O(logn). As same as inseriton red black trees being balanced are a key part for deletion to. Deletion from a red-black tree may involve restructuring the tree in order to maintain balance, which can include adjusting the colors of nodes and performing rotations. These adjustments and rotations take time, but they are relatively quick and occur infrequently, so the overall time complexity of red-black tree deletion is O(log n).

2.3.3 Complexity of Getting Left - Right Most

Same cases are true for this function to , in a balanced tree with purpose of reaching the edge nodes we are going to left until we reach a nil node and that is proportional to height of the tree whihc is O(logn).

2.3.4 Complexity of Inorder Traversal

Generally if we are talking about a traversal function , since it has to reach to every element on a tree it's complexity must be proportional to O(n).

2.3.5 Overall Complexity of Implementation

When we consider all functions and operations excluding the operations that have constant time. In main, there are a main loop which will be processed as musch as time that has been given in the input file. We can call this time h. In this loop in the worst case we may delete, insert and traverse the tree in a one cycle but it is the worst case. Therefore in the worst case we will have time complexity O(h(n + logn)), h comes from the length of the loop and n comes from the traversal and logn is coming from the deletion , insertion operations. On the other hand in the best case there won't be any insertion and deletion then our complexity will be just $o(h(n))$.

2.4 Food For Thought

2.4.1 Advantages of Red-Black Trees

There are many advantages using a red-black tree as a underlying data structure.One of the most important advantages is inseriton and deletion operations are quick which has a time complexity $O(logn)$ and this makes them faster than other data structures. Another advantage is Red-Black trees are balanced trees which means that height will be always proportional to logn. Height does not depend on the which element or values is inserted because of the balance maintaining functions like reorderin(fixup) and rotations. This allows faster search, deletion and insertion. On the other hand red-black trees are memory efficient to. They have only 2 colors which makes the difference but data structures like avl or 234 trees need more memory to store. Implementation is quite simple to if we compare them with other balancing trees.

2.4.2 Completely Fair Schdeduler

CFS, or Completely Fair Scheduler, is a scheduling algorithm used in the Linux kernel to allocate CPU time to processes and threads in a fair way, based on their priority and the amount of CPU time they have already received. It is widely used in many real-world systems that use the Linux kernel, including servers, desktop computers, and embedded devices. CFS has also been implemented in other operating systems and in the Xen hypervisor for scheduling virtual machines. It has been the default scheduler in the Linux kernel since version 2.6.23, released in 2007, and is known for its effectiveness and efficiency.

2.4.3 Maximum Height of a RBTree with N Proces

The maximum height of a tree with n elements is $O(\log n)$. This can be proven using a simple argument based on the number of nodes at each level of the tree. Lets consider a Red Black Tree which has a height of h where h is an integer.In the first level there are only 1 node.In the second layer there will be 2 nodes at most. In the next layer there will be 4 nodes .Nodes represent the process. This will continue like this , so number of nodes at hth level there will be at most 2*(h-1) nodes. Sum of all nodes in layers will be 1 + 2 + 4 + 8... 2*(h-1) which is equal to $2\hat{h}$-1 and this is equal to n which is the number of process. After that we can rearrange this equation in the form of h = log_2(n+1) This means that the maximum height is upper bounded by $O(logn)$. This is because of reordering mechanisms when we violate the balancing properity. When balance is broken we will make rotations and recoloring to maintain it again and as we know in a balanced search tree the height is logn. As a result a tree with n process will have the height which is proportional to logn.

2.5 Results

```
0,-,-,-,-,-,-
1,P1,0,0,P1:0-Black,Incomplete
2,P2,0,0,P2:0-Red;P3:0-Black;P1:1-Red,Incomplete
3,P3,0,0,P3:0-Red;P1:1-Black;P2:1-Red,Incomplete
4,P3,1,1,P3:1-Red;P1:1-Black;P2:1-Red,Completed
5,P1,1,1,P1:1-Black;P2:1-Red,Completed
6,P2,1,1,P2:1-Black,Completed

Scheduling finished in 145 ms.
3 of 3 processes are completed.
The order of completion of the tasks: P3-P1-P2
```

This is the output for the inputs (3 7);(P1 1 2);(P2 2 2); (P3 2 2)

```
0,-,-,-,-,-,-
1,P1,0,0,P1:0-Black,Incomplete
2,P2,0,0,P2:0-Red;P3:0-Black;P1:1-Red,Incomplete
3,P3,0,0,P3:0-Red;P1:1-Black;P2:1-Red,Incomplete
4,P3,1,1,P3:1-Red;P1:1-Black;P2:1-Red,Completed

Scheduling finished in 127 ms.
1 of 3 processes are completed.
The order of completion of the tasks: P3
```

This is the output for the inputs (3 5);(P1 1 2);(P2 2 2); (P3 2 2)
Additional test cases has been created and tested to with them.