COP5536 Advanced Data Structures
Fall 2019

Programming Project

Name: Saheel Ravindra Sawant
UFID: 1164-7923
Email ID: sawant.s@ufl.edu

1. Project Description:

In this project, a program is devised for a certain number of buildings that are being constructed in a new city, with the help of a certain parameters like **buildingNum** (unique integer identifier for each building), **executed_time** (total days spent on the building construction) and **total_time** (number of days needed to complete the construction of the building).

To maintain these parameters, two data structures are used: Min Heap and Red-Black Tree. It uses a min heap to store all these three parameters together such that the element at the root node must have the lowest executed_time and subsequent nodes will have higher execution times than root, but any child node should not have execution time lesser than its parent node or it violates the min heap functionality. The Red Black tree contains the record ordered by the building number and mainly used for printing operations. The same record is stored in both the data structures in order to ensure the changes made in one are reflected in another.

Once a node or building is inserted into the heap, it checks whether there are other buildings who have lower executed_time. If there is, then that building, with the lowest executed_time is run first for five days. If that building completes its work, it is removed or else it's executed time is updated and again the comparison is made until a new building with lower executed_time enters the heap. In such case the old building is removed and the work starts on the new one.

2. Input / Output:

The input is given through a text file to the this program. This file is read by the program through args[0] and it mainly contains a specific commands which are as follows:

1. Print (buildingNum) which is expected to print the three parameters for the given building number.
2. Print (buildingNum1, buildingNum2) which is expected to print all the buildings between the range of buildingNum1 and buildingNum2.
3. Insert (buildingNum,total_time) which is expected to insert the record into the heap with executed_time initialised to zero.

After performing the above operations, the output is reflected in to another text file named output.txt

Note: For the provided 'input' file, the obtained output is as follows:

```
(15,0,25)(50,15,20)
(15,4,25)(50,15,20)
(50,60)
(15,65)
(30,75)
```

Here, while printing the output, firstly the operation is performed and then the updation takes place for the executed_time, so that is the reason why here *the executed_time has one day less* in the PrintBuilding command.

3. Implementation:

1) risingCity.java

Firstly, the input from the file is read and all three parameters are stored in the 'val' array for every line and the val[3] values are stored in the format one or zero to distinguish between insert and print commands.

```
for (int i = 0; i < i_input.size(); i++) {
    val = new int[4];
    s = i_input.get(i);

    int a =Integer.parseInt(s.substring( i: 0, s.indexOf(":")));
    int b =Integer.parseInt(s.substring( i: s.indexOf("(") + 1, s.indexOf(",")) );
    int c =Integer.parseInt(s.substring( i: s.indexOf(",") + 1,s.indexOf(")")) );
    if(s.contains("Insert")) {
        //Insert value in val[3] as per Insert or Print Building
        val[0] = a;
        val[1] = b;
        val[2] = c;
        val[3] = 1;
        bldg_val.add(val);

    }
    else if(s.contains("PrintBuilding") && s.contains(",")){
        val[0] = a;
        val[1] = b;
        val[2] = c;
        val[3] = 0;
        bldg_val.add(val);
    }
    else{
        val[0] = a;
        val[1] = Integer.parseInt(s.substring( i: s.indexOf("(") + 1, s.indexOf(")")) );
        val[2] = -1;
        val[3] = 0;
        bldg_val.add(val);
    }
}
```

In the snapshot attached below, if all the conditions of the if part satisfies then insert command is executed by creating an object and passing it to both min heap and red black tree. In the print command, if all else-if conditions satisfy, then it checks the number of parameters.

If the printBuilding command, has one parameter i.e. val[2] < 0, then accordingly it constructs a RBT and displays an output which is defined in the class Construct_RedBlackTree. Similarly, the output is displayed for the printBuilding command with two parameters, which particularly defines a range.

```
while( i_input.size() > line_count || bHeap.Building_array[0] != null || five_days_counter > 0){

    if(bldg_val.size() > 0 && bldg_val.get(0)[3] == 1 && global_clock == bldg_val.get(0)[0])
    {
        val = bldg_val.remove( i: 0);
        Bldg_details record = new Bldg_details(val[1], val[2]);
        //Insert the same record into Heap and Red-Black Tree
        bHeap.insert(record);
        redBlackTree.add(record);
        //Increment line counter
        line_count++;
    }
    else if(bldg_val.size() > 0 && bldg_val.get(0)[3] == 0 && global_clock == bldg_val.get(0)[0])
    {
        val = bldg_val.remove( i: 0);
        if(val[2] < 0){
            single_output = Construct_RedBlackTree.search(Construct_RedBlackTree.root, val[1]);
            System.out.println(single_output);

        }
        else {
            empty = new ArrayList<>();
            rbt_output = Construct_RedBlackTree.searchRange(empty, Construct_RedBlackTree.root, val[1], val[2]);
            System.out.println(rbt_output);
        }
        line_count++;
    }
```

Here, the five days condition is checked. If a building ‘current_bldg’ has completed five days and it’s executed days matches its required days then the building is printed and removed or else it is put in the heap again with its updated executed time. The five_days_counter is always reset after 5 days whereas the global counter is incremented by one every time the main while loop in the previous snapshot is implemented .

```
//Check whether the current_bldg is executed
if( five_days_counter != 0) {
    if( five_days_counter == 5) {
        if(current_bldg.get_executed_time() + 1 == current_bldg.get_total_time())
        {
            System.out.println( "("+ current_bldg.get_bldg_num() +","+ global_clock + ")" );
            redBlackTree.remove(current_bldg); //remove record from Red-Black Tree
        }
        else {
            current_bldg.set_executed_time(current_bldg.get_executed_time() + 1);
            bHeap.insert(current_bldg);
        }
        five_days_counter = 0; // Reset counter to zero
    }
    else if(current_bldg.get_executed_time() + 1 == current_bldg.get_total_time()) {
        System.out.println( "("+ current_bldg.get_bldg_num() +","+ global_clock + ")" );
        redBlackTree.remove(current_bldg);
        five_days_counter = 0;
    }
    else {
        current_bldg.set_executed_time(current_bldg.get_executed_time() + 1);
        five_days_counter++;
    }
}

if(five_days_counter == 0 && bHeap.Building_array[0] != null) {
    current_bldg = bHeap.removeRoot();
    five_days_counter++;
}
global_clock++; //Increment global counter after every step
```

2) Construct_Heap.java

Here, a building is inserted into the heap. The heap size is first incremented, then the compare_funct is called and checked with respect to the executed_time. For the current building if the executed_time is lower, then it returns true and allows to insert the node into the heap. The swap function is being called if the conditions hold, and the node with the lower executed_time becomes current node.

```
public boolean compare_funct(Bldg_details bR, int index){
    if(Building_array[index].get_executed_time() > bR.get_executed_time())
        return true;
    if(Building_array[index].get_executed_time() < bR.get_executed_time() )
        return false;
    if(Building_array[index].get_bldg_num() < bR.get_bldg_num() )
        return false;
    return true;
}

public void insert(Bldg_details record){

    Building_array[++size_value] = record;
    int current = size_value;

    while( current>0 && compare_funct(Building_array[current], pNode(current) ) )  {
        {
            swap(current, pNode(current));
            current = pNode(current);
        }
    }
}
```

```
private void swap(int index1, int index2){
    Bldg_details tmp = Building_array[index1];
    Building_array[index1] = Building_array[index2];
    Building_array[index2] = tmp;
}
```

In this, a node is removed from the tree, which is a root node. After the node is removed, the size of the heap is decremented and the heapify function is called, which ensures every parent node of the min heap has value lesser than all its child nodes. This function takes the value of the first existing node which is present after the removal operation.

```
public Bldg_details removeRoot()
{
    Bldg_details abc = null;
    if(Building_array[0] != null)
    {
        abc = Building_array[Front_value];
        Building_array[Front_value] = Building_array[size_value];
        Building_array[size_value--] = null;
        Heapify(Front_value);
    }
    return abc;
}
```

```
public void Heapify(int index) {
    if (!isLeaf(index)) {
        if ((Building_array[lNode(index)] != null &&
                compare_funct(Building_array[lNode(index)], index))
                || (Building_array[rNode(index)] != null &&
                compare_funct(Building_array[rNode(index)], index))
        ) {
            if (Building_array[lNode(index)] == null) {
                swap(index, rNode(index));
                Heapify(rNode(index));
            } else if (Building_array[rNode(index)] == null) {
                swap(index, lNode(index));
                Heapify(lNode(index));
            } else {
                if (compare_funct(Building_array[lNode(index)], rNode(index))) {
                    swap(index, lNode(index));
                    Heapify(lNode(index));
                } else {
                    swap(index, rNode(index));
                    Heapify(rNode(index));
                }
            }
        }
    }
}
```

3) Bldg_details.java

In this, the buildingNum, executed_time and total_time are stored in methods or constructors which are used to fetch the values as and when required. The first Bldg_details takes two parameters in buildingNum and total_time and here executed time is set to zero. It is used for insertion of building when it appears for the first time in the file.

```
public class Bldg_details {
    private int buildingNum;
    private int executed_time;
    private int total_time;

    public Bldg_details(int buildingNum, int total_time) {
        this.buildingNum = buildingNum;
        this.executed_time = 0;
        this.total_time = total_time;
    }

    public int get_bldg_num() {
        return buildingNum;
    }

    public int get_executed_time() {
        return executed_time; }

    public int get_total_time() {
        return total_time; }

    public void setBuildingNum(int buildingNum) {
        this.buildingNum = buildingNum; }

    public void set_executed_time(int executed_time) {
        this.executed_time = executed_time; }

    @Override
    public String toString() {
        return "("+ buildingNum + "," + (executed_time) + "," + total_time + ")";
    }
}
```

4) Construct_RedBlackTree.java

In Red-Black Tree, the following function is used to print the building details. It takes the ArrayList as input along with the node and the starting and ending nodes. The building details are checked as per the starting building number and ending building number and passed back as an output.

```
public static Bldg_details search(Node node, int buildingNumber) {
    if(node == null || node.bldg_data_value == null) {
        return null;
    }
    if(node.bldg_data_value.get_bldg_num()==buildingNumber) {
        return node.bldg_data_value;
    }else if(node.bldg_data_value.get_bldg_num()<buildingNumber) {
        return search(node.r_child, buildingNumber);
    }else if(node.bldg_data_value.get_bldg_num()>buildingNumber) {
        return search(node.l_child, buildingNumber);
    }
    return null;
}

public static ArrayList<Bldg_details> searchRange(ArrayList<Bldg_details> list, Node node, int startBuilding, int endBuilding){
    if(node==null || node.bldg_data_value == null) {
        return list;
    }
    if(inRange(node.bldg_data_value.get_bldg_num(), startBuilding, endBuilding)) {

        list.add(node.bldg_data_value);
    }if(node.bldg_data_value.get_bldg_num() > startBuilding) {

        searchRange(list, node.l_child, startBuilding, endBuilding);
    }if(node.bldg_data_value.get_bldg_num() < endBuilding) {

        searchRange(list, node.r_child, startBuilding, endBuilding);
    }
    return list;
}
```

Everytime a node is to be added to an RBT, first it is assigned a color red and all its child nodes are initialised with black with the values stored in them as null.

```
public class Construct_RedBlackTree {
    static class Node {
        // Data value
        Bldg_details bldg_data_value;
        // Left child pointer
        Node l_child;
        // Right child pointer
        Node r_child;
        // Parent Node pointer
        Node parent_node;
        String color; // color of the node.

        public Node(Bldg_details data, String color) {
            this.bldg_data_value = data;
            this.l_child = null;
            this.r_child = null;
            this.parent_node = null;
            this.color = color;
        }
    }
    //Root Node
    static Node root;

    // Create a new Node.
    public Node makeNode(Bldg_details bR) {
        Node node = new Node(bR,  color: "R");
        node.l_child = new Node( data: null,  color: "B");
        node.r_child = new Node( data: null,  color: "B");
        return node;
    }
```