COP 5536: Advanced Data Structures, Fall 2019
Programming Project Report – Rising City

Student: Minh N. Vu
ID: 9194-1842
minhvu@ufl.edu

November 21, 2019

1 Running the program

Run the makefile included in the zip file on its directory. Then run the main "risingCity" file with the argument `file_name` as the name of the input file:

```
\$ make
\$ java risingCity file_name
```

The program will generate an output text file named "output_file.txt", which contains the results of the run. The output file will be overwritten for each run of the program.

2 Contents of the submission

- makefile: Make file that compiles the java files and generates the java classes.
- risingCity.java: The main program which runs the global counter and (1) process the data structure in time, (2) read from the input and call the corresponding functions and (3) write the report to output file.
- RedBlackTree.java: The RedBlackTree java class, which stores the inserted data from the input file in form of the Red-Black Tree and provides the operations on the Red-Black Tree. The order is based on the building number.
- minHeap.java: The minHeap java class, which stores the inserted data from the input file in form of min-Heap and provides the operations on the min-Heap. The order is based on the executed time of the buildings.
- report.pdf: This report, which provide the descriptions and structure of this project.

3 Code Structure and Code Description

This section contains the code structure and code descriptions of this project.

3.1 risingCity.java

Main program: The main program is in the risingCity.java. Figure 1 is the code structure of the main function of the class risingCity. First, it initializes the data structure which is a combination of a Red-Black Tree and a min-Heap. Then, it starts to read from the input file. While there are commands from the input, the program first computed the `number_of_days`, which is number of days between the current day of the program and the day that the command will be executed. Then the program calls `Update(number_of_days)` iteratively which computes the state of the data structure after `number_of_days` days. The details of the `Update` function is discussed in next subsection. After that, the program checks the content of the command and process it accordingly. Note that, in order to print the building that finish at the moment of the printing command, the program waits after finishing the printing then removes the building from the data structure if necessary.

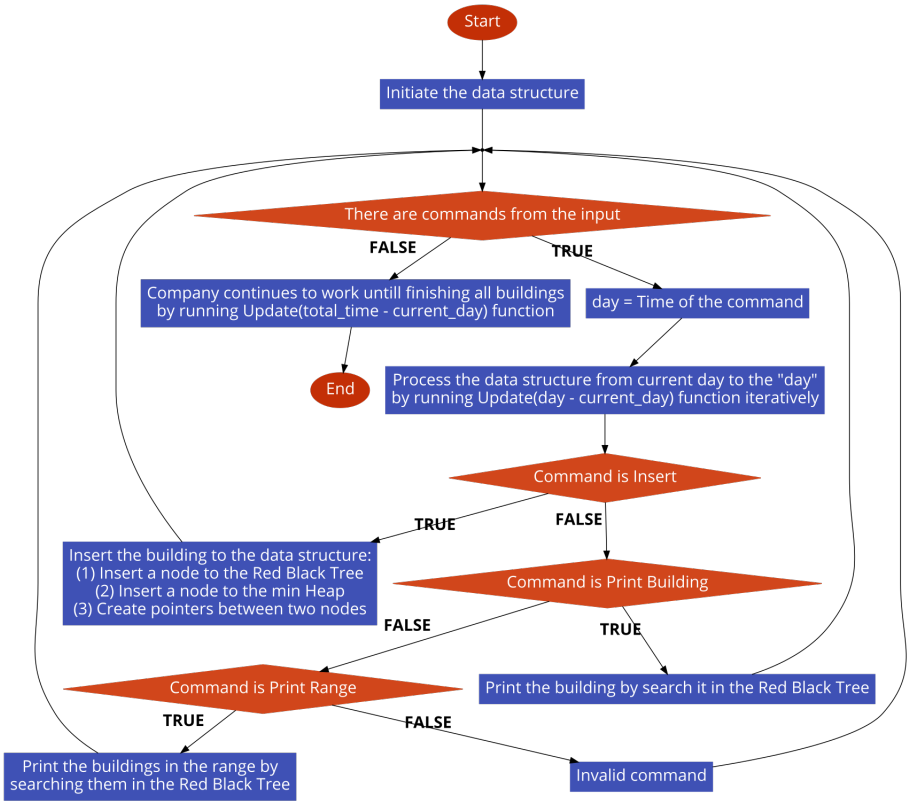

Figure 1: Code diagram for the main function in risingCity.java

Update: The `Update` function computes the state of the data structure after `number_of_days` days without any insertion. Figure 2 is the code structure of the `Update` function of the class risingCity. The function receives a `number_of_days` as an input. For each call of the `Update`, the `number_of_days` will decrease until it reach zero. Specifically, there are four cases:

- The company is not working on any building and there is no more building to work on: The program counts the number of days that the company stay idles and return 0. This value is assigned to `number_of_days` of next iteration.
- The company is not working on any building and there are buildings to work on: We assign the building with least executed time for the company to work on and return `number_of_days`. This value is assigned to `number_of_days` of next iteration.

- The company is working on a building and the duration until finishing the task is smaller than `number_of_days`: The company continues to work on the task until finishing it. The building is removed from the data structure if it is finished. The function then returns `number_of_days - number_of_days_to_finish_the_task`, which is the remaining time that the company can spend for other task. This value is assigned to `number_of_days` of next iteration.
- The company is working on a building and the duration until finishing the task is greater than or equals `number_of_days`: The company continues to work on the task. The function return 0. This value is assigned to `number_of_days` of next iteration.

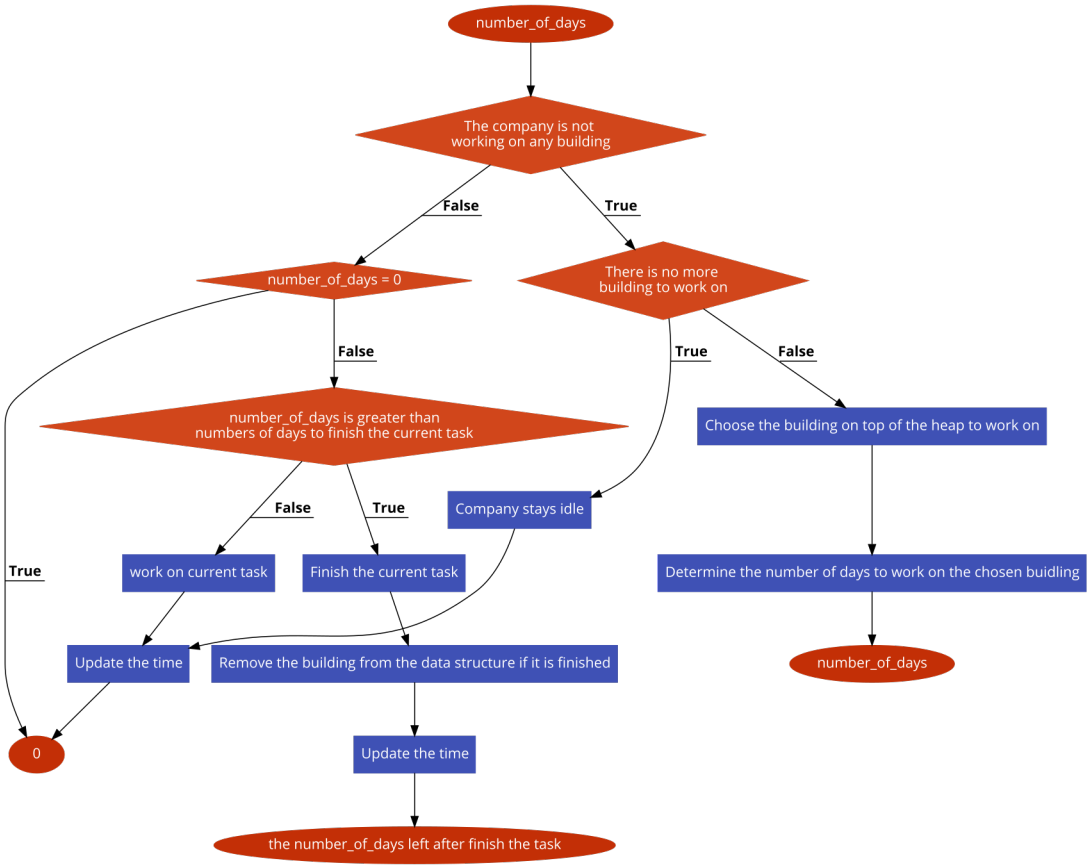

Figure 2: Code diagram for the Update function in risingCity.java

Note that the `outputWriter` is also passed to the `Update` function for generating output purpose.

Other methods: The risingCity.java also contains the several other methods listed as follows:

- `Insert(int buildingNum, int total_time)`: insert the building to the data structure. The function creates a Red-Black Tree node and a heapNode, establishes the pointers between them and inserts them into the corresponding data structure.
- `Remove(int buildingNum)`: remove the node with the `buildingNum` building number from both the Red-Black Tree and the min-Heap. The function just call the corresponding remove functions of each data structure.
- `RemoveAtTime(FileWriter outputWriter, minHeap.HeapNode node)`: remove the `node` from the heap, and then the Red-Black Tree if the building is finished. Finally, the removal event is recorded into the `outputWriter`.

- `Print(FileWriter outputWriter, int buildingNum)` and `Print(FileWriter outputWriter, int buildingNum1, int buildingNum2)`: these function call the search and range search functions of the Red-Black Tree and recorded the results into the output file.

3.2 RedBlackTree.java

The RedBlackTree.java contain the `class RedBlackTree` and the `class Node`, which is the node of the tree. In the followings, I will discuss about these two classes.

`class RedBlackTree`

- Description: the Red-Black Tree data structure of the inserted buildings. The order is based on the building number.
- Parameters:
  - `final static int RED, BLACK, LEFT, RIGHT`: constants defining the colors and the sides of the nodes in the tree.
  - `static boolean first`: variable determines the first found of a node in a range search. `True` indicates that no node has been found. This variable is used for printing.
  - `Node root`: the root of the tree
  - `class Node`: The node of the Red-Black Tree. The parameters of this class is discussed below.
- Methods:
  - `public static Node initNode(int buildingNum, int total_time)`: Initialize node by the building number and time for completion.
  - `public static Node findNode(int buildingNum)`: find the node in the tree by building number.
  - `private static Node findNextNode(Node node)`: find the node with degree 0 or 1 whose key is the smallest key greater than or equal to the key of the input node. If the input node has degree 0 or 1, the function just returns the input node. This function is used for removing node.
  - `private static int countRedChild(Node node)`: count the number of red children of the node.
  - `private static void swapData(Node node1, Node node2)`: swap the data of two nodes in the tree, which includes the pointers to the corresponding node in the heap.
  - `public static void insert(Node node)`: insert the node into the tree.
  - `public static void insert(int buildingNum, int time)`: create a node with given building number and time till completion and then insert it to the tree.
  - `private static void remedy(Node node)`: fix the node in the insertion of the tree.
  - `public static void remove(int buildingNum)`: search for the node by the building number and then remove it from the tree.
  - `private static void fix(Node node, int side)`: fix the node in removal of the tree.
  - `private static void rightRotate(Node node)`: the right rotate move.
  - `private static void leftRotate(Node node)`: the left rotate move.
  - `private static void rangeQuery(FileWriter outputWriter, Node node, int k1, int k2)`: function to search for the nodes in the range of `(k1,k2)` in the tree.

There are other methods supporting the printing and verifying the correctness of the Red-Black Tree.

`class Node`

- Description: the node of the Red-Black Tree.
- Parameters:
  - `int key`: the key which determines the order of the node in the tree. In this case, it is always equal to `buildingNum` (the building number).
  - `Node left, right, parent`: the pointers to the left child, the right child and the parent of the node.
  - `int color`: the color of the node.
  - `int buildingNum, executed_time, total_time`: the building number, the executed time and the total time of the building.
  - `minHeap.HeapNode heapNode`: the pointer to the corresponding node in the heap.
- The class is equipped with the constructor `public Node(int key, int color)` which initializes the `key, buildingNum` and `color` of the node. All others parameters are set to `null` or `0`.

3.3 minHeap.java

Similar to the Red-Black Tree, the min-Heap is also implemented by two classes, which are the `class minHeap` and its nodes `class HeapNode`.

`class minHeap`

- Description: the min-Heap data structure. The order is based on the executed time of the building.
- Parameters:
  - `private static HeapNode[] Heap`: the array of heap nodes, which determines the positions of the nodes in the heap.
  - `public static int HeapSize`: the actual size of the heap.
  - `private static int maxSize`: the max size of the heap (which is assumed to be 2000).
  - `static class HeapNode`: the heap node class.
- Methods:
  - `public static Node initNode(int buildingNum, int total_time)`: Initialize node by the building number and time for completion.
  - `private static boolean smaller (HeapNode node1, HeapNode node2)` : Determine whether `node1` is smaller than `node2` in the heap. The smaller is based on executed time of the node. In case the executed time are equal, the node with smaller building number is smaller.
  - `private static void swap(HeapNode node1, HeapNode node2)`: swap positions of two nodes on the heap.
  - `private static void heapify(HeapNode node)`: heapify the node, i.e if the node is larger than its children then swap it with the smaller children. The heapify continues until the node is larger than its children or it has became a leaf.
  - `public static void insert (HeapNode heapNode)`: insert the node to the heap.

- `public static void insert(int buildingNumber, int time)`: create a node with given building number and time till completion and then insert it to the heap.
- `public static HeapNode getTop()`: get the node on top of the heap.
- `public static void increaseExecutedTime(HeapNode node, int time)`: increase the executed time of the `node` by `time` and then `heapify` the node. The executed time of the corresponding Red-Black Tree node is increased accordingly. The tree node is found by searching the tree using the `node.buildingNum`.
- `public static HeapNode removeMin()`: remove the top of the heap and return the removed node.
- `public static void remove(HeapNode node)`: remove the node from the heap by replacing it with the last node. If the replaced node is smaller than its parent then swap them, else heapify the node.
- `private static int parentIndex(int index), leftIndex(int index),rightIndex(int index)`: return the positions of the parent, the left child and the right child of the node with given `index` location in the heap.
- `public static boolean isLeaf (HeapNode node)`: check if the `node` is a leaf.
- `public static void printHeap()`: print the heap.

`class HeapNode`

- Description: the node of the min-Heap.
- Parameters:
  - `int key`: the key which determines the order of the node in the heap.
  - `int buildingNum, executed_time, total_time`: the building number, the executed time and the total time of the building.
- The class is equipped with the constructor `public HeapNode(int buildingNum)` which initializes the `buildingNum` of the node. All others parameters are set to `null` or `0`.