Advanced Data Structures (COP 5536)

Fall 2019 Project Report

Amit Asish Bhadra
UF ID: 6649-4087
amitbhadra@ufl.edu

1. DESCRIPTION:

Wayne Enterprises is developing a new city. They are constructing many buildings and plan to use software to keep track of all buildings under construction in this new city. A building record has the following fields:
buildingNum: unique integer identifier for each building.
executed_time: total number of days spent so far on this building.
total_time: the total number of days needed to complete the construction of the building.

The needed operations are:
1. Print (buildingNum) prints the triplet buildingNume,executed_time,total_time.
2. Print (buildingNum1, buildingNum2) prints all triplets bn, executed_tims, total_time for which buildingNum1 <= bn <= buildingNum2.
3. Insert (buildingNum,total_time) where buildingNum is different from existing building numbers and executed_time = 0.

Wayne Construction works on one building at a time. When it is time to select a building to work on, the building with the lowest executed_time (ties are broken by selecting the building with the lowest buildingNum) is selected. The selected building is worked on until complete or for 5 days, whichever happens first. If the building completes during this period its number and day of completion is output and it is removed from the data structures. Otherwise, the building's executed_time is updated. In both cases, Wayne Construction selects the next building to work on using the selection rule. When no building remains, the completion date of the new city is output.

2. IMPLEMENTATION:

- **MinHeap**: We store the **Buildings** sorted by **Executed Time** here, so shortest executed times will be higher up. If we have 2 Buildings with the same executed time then the one with the smaller **Building Num** gets the preference. In Minheap, we get O(log n) time for **Insert**, **Delete** and **Remove Min**. Even **Heapify** takes O(log n) time.

- **Red Black Tree**: A Red Black Tree is a **Balanced Binary Search Tree** with a color constraint. Each Node can be either Red or Black. No 2 simultaneous nodes can be red in color. The root is always black. The advantage of AVL Trees to Red Black Trees is that the rotate step doesn't propagate to root.

3. EXECUTION:

This is how we execute the project:

- Unzip the project
- Paste the input file
- make
- java risingCity <InputFile.txt>
- View the output in output_file.txt

4. STRUCTURE OF THE PROJECT:

I have the following 5 classes:

- **Building**:
  This is the framework for the smallest unit of the project. **Minheap** and **RedBlackTree** uses **Building** as a class object.

- **risingCity**:
  This is the main class. Here we call **WorkTillEnd** which works till all the buildings are worked upon. In **WorkTillEnd**, we run a loop while the input is completely read. Inside the loop we wait for the next input day number. We loop inside while the global counter is not equal to the input day number. When we loop inside in the previous statement, we keep calling **WorkOneDay**. Calling this amounts to 1 unit of work.

  How it works is, initially, we remove min and store the node to work in **currentNode**. We remove this out of the minheap and **heapify** when we do this so that we are always able to maintain the heap property. If we do this it costs us O(log n) time. If we don't heapify at this time, then later on, if we have multiple inputs then Heapify won't work. We will have

to **BuildMinheap** which takes O(n) time which is a costlier function. **WorkOneday** will increase the executed time of the current node which is the current building to work by 1. It also checks if the building's total time is sane as executed time. That means the building is finished building. In that case we must remove the building from Red black tree. We don't need to remove from **MinHeap** because we had already extracted it out of min heap when we started working on it. We simply do not need to re-insert it into the min heap anymore.

We also have a variable here called **tempCount**. This is used to store the temporary count of the number of days a building is spent in building. We start with **tempCount**=0 and increase it by +1 every time **WorkOneDay** is called. If **tempCount**=5 then we have finished working 5 days on a building. So we make **tempCount**=0. Also if a building finishes building, we make **tempCount**=0.

Now in **WorkTillEnd**, when we the day number is same as global counter then we execute the command. We first check if it's an Insert or Print command. For insert, we first insert into minheap and RBT and then call **WorkOneDay**. For Print, we first work one day and then we print. For print, we call **WorkOneDay** with an extra String which stores the print command. When the work is done we first print if the building finishes execution and then print the RBT if Print command is issued.

In the main, once the input file is read completely, we have another loop that runs till all the buildings are built. This also works the same way as before by calling **WorkOneDay**.

- **Minheap**:
  This is the class that will implement **Minheap**. There is another class **Node** which will be each element of the minheap Node array. Class Node's key is the number of executed days of the Building. Here we also have a pointer to the RB Node so that when we need to delete it, we can easily find point to the RedBlack Tree Node and then proceed with deletion from the structure.

  Minheap will be an array of the Nodes because the way that minheap is done is that we can get parent of node n by finding the node at position **(n-1)/2** and we can get children of parent n as **left child as 2*n+1** and **right child as 2*n+2**. This class contains the ingredients to insert into minheap, remove min, heapify which is also **heapifydown**, and percolate up or **heapify up**.

- **Simulation**:
  This is the Driver class of my project. **risingCity** calls **Simulation** and Simulation delegates the work between **Minheap** and **Redblack** Tree. So there is no direct access from main to the 2 fundamental data structures. This is done so that the data structure is kept away from the functioning of the central code. Simulation has functions which will preprocess code before sending to either minheap or RedBlack, and vice versa. This also

contains the **printWriter** object which prints the output to output.txt. At the end, we close the **printWriter** function.

- **RedBlackTree**:
  This contains a class called **RBNode** which will be each element or node of the Red black tree. It contains details as mentioned in the code. The **key** is the **Building number** because we create the red black tree based on the building number.

  The main **redBlack** class is the one which is the implementation of the red black tree. It contains the variable root which will eventually be used to point to children RB nodes. Inside this class, we have functions which are used to perform the basic RB functions such as **Insert**, **Delete**. For each, we might need to **rotate** and **transplant** the tree to maintain the height and color constraint that is required for a red black tree. We also have the check to see if a building number is already present or not before inserting. If the building is new then we insert it, if it has already been inserted, we throw **Error**. We also have functions to Print the **Node** and **Print** between **Indexes** as required for the project.

5. DOCUMENTATION:

- **Building.java:**

Class Variables:

Name	Type	Description
buildingNum	int	The number of the building
executed_time	int	How many days has the building been worked on
total_time	int	Maximum number of days we need to work on a building

Member Functions:

Name	Return Type	Description
Building		Parameterized constructor
setExecutedTime	void	Setter Function for Executed Time
getExecutedTime	int	Getter Function for Executed Time
getBuildingNum	int	Getter Function for Building Number
getTotalTime	int	Getter Function for Total time of a building
changeExecutedTime	void	If we need to change the executed Time when we increment it

- **risingCity.java:**

Class Variables:

Name	Type	Description
currentNode	Node	This node contains the building we are currently working on
finished	boolean	If the current building has finished working

tempCount	int	The number of days current Building has been worked upon

Member Functions:

Name	Return Type	Description
main	void	Static Function where code starts execution from
WorkTillEnd	void	Called from main, this function mainly works on finishing all buildings
workOneDay	int	Returns tempCount. This is the function to work one day it gets called from WorkTillEnd
identifyCommand	Int	This is just to identify the input command, 1 for Insert and 2 for Print

- **Simulation.java:**

Class members:

Name	Type	Description
minHeap	minHeap	Object of class Minheap
redblack	redBlack	Object of class redBlack
printWriter	PrintWriter	Object of PrintWriter used to write to file

Member Functions:

Name	Return Type	Description
Simulation		Default Constructor
setOutputFile	void	Set the printwriter to output file
removeMin	Node	Driver class that calls remove min of minheap

buildBuilding	boolean	Driver class that works on one building by increasing its executed time
printFinishedBuilding	void	If a building finishes work we output this to file
insert	void	Whenever a new Building is inserted first we insert to RedBlack so that we get the pointer to RBNode and use it to initialise the minheap with the pointer. After imserting we heapifyUp to maintain the heap property
printIndex	void	Driver class that calls the RedBlack Print function
printBetweenIndexes	void	Driver class that calls printbetween of redBlack
printBuildings	void	Preprocessing for print function
insertBuilding	void	Preprocessing for Insert function
reInsertPreviousMin	void	Driver class that calls MinHeap to reinsert the building we had worked on before this
deleteRBNode	void	Driver function that calls thefunction to delete from RedBlack Tree

- **MinHeap.java:**

1. Node.class:

Class Members:

Name	Type	Description
key	int	Executed Time will be used as key
building	Building	Building of each Node
rbnode	RBNode	The Red Black node this will point to.

Member Functions:

Name	Return Type	Description
Node		Parameterized Constructor
setBuilding	void	Setter function for Building
setKey	void	Setter function for building
getBuilding	Building	Getter function for Building
getKey	int	getter function for key
setRBNode	void	Set the RedBlack Pointer Node
getRBNode	RBNode	Get the red black pointer node

2. MinHeap.class:

Class Members:

Name	Type	Description
totalBuildings	int	totaBuildings stores the current no of buildings and the array of nodes
head	Node[]	The list of nodes in minheap data structure

Member Functions:

Name	Return Type	Description
MinHeap		Default Constructor
MinHeap		Parameterized Constructor
heapifyUp	void	Function to Heapify Up or percolate up
heapifyDown	void	Function to Heapify Down
removeMin	Node	At the start of 5 days or when a a building finishes, we call this function to get the new minimum
insert	void	Insert new Buildings
insert	void	This is overloaded Insert for inserting old Buildings which have been worked for 5 days

- **redBlack.java:**

1. RBNode.class:

Class Members:

Name	Type	Description
building	Building	building object
data	Int	holds the building id
parent	RBNode	this is pointing to the parent
left	RBNode	this is pointing to left child
right	RBNode	this is pointing to right child
color	color	This is 1 for Red and 0 for Black

Member Functions:

Name	Return Type	Description
RBNode		Default Constructor
RBNode		Parameterized Constructor number 1
RBNode		Parameterized Constructor number 2

2. redBlack.class:

Class Variables:

Name	Type	Description
root	RBNode	Store the root of RBT
LeafNode	RBNode	This is special dummy leaf nodes

Member Functions:

Name	Return Type	Description
redBlack		Default Constructor
searchTreeHelper	RBNode	Search the Node that contains the key from the nodeToSearch onwards
fixRedBlackDelete	void	Once a delete action takes place the RedBlack Tree needs to be fixed. This is because RBT is a balanced binary tree
deleteRedBlackNodeHelper	void	This is the helper class for red black tree delete. This calls the transplant function
fixRedBlackInsert	void	Once a new RB Node is inserted, we need to fix the Red black tree. If it is no more height or color balanced
searchTree	RBNode	Search the RedBlack tree for a value or key
searchTreeRange	List<RBNode>	Search the Tree in a particular Range. This calls the print range helper function
searchTreeRangeHelper	void	Given a particular node and a range, it does In order traversal and adds the value to the List result
findMinimumNode	RBNode	Find the smallest RBNode
findMaximumNode	RBNode	Find the largest RBNode
findSuccessorNode	RBNode	Find the RB Successor of a RB Node
findPredecessorNode	RBNode	Find the RB Predecessor of a RB Node
leftRotateTree	void	Left Rotate the RB Tree starting from a node
rightRotateTree	void	Right Rotate the RB Tree starting from a node
insert	RBNode	This function gets called from the Driver class to insert a new building
getRoot	RBNode	Returns the RB Root

deleteRedBlackNode	void	Called from Driver class. This calls the helper class which performs the function
displayBuilding	String	If the input is Print(x) returns the correct print string and Driver will output the string to output.txt
displayBuildlingRange	String	If the input is Print(x,y) returns the correct print string and Driver will output the string to output.txt

6. REFERENCES:

- Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.