REPORT FOR PROJECT
Wayne Enterprises

Name: VEJAY MITUN VENKATACHALAM JAYAGOPAL
UF-ID: 3106-5997
UF Email: vejaymit.venkata@ufl.edu

OBJECTIVE:

To construct Wayne Enterprise by using a min-heap data structure based on execution time to prioritize which the building to taken into construction at the time of execution. To perform print operation of the buildings from the red-black tree data structure, such that all the operations take time lesser than or equal to O(log N) running time.

WORKING AND PROTOTYPES USED:

For building the Wayne Enterprise, every building inserted executes for five days or till the execution of the building is completed within those five days. Then the next building is taken in for construction based on min priority heap. The print function is done using a red-black binary search tree data structure.

Three classes are:

1. risingCity
2. MinHeapRisingCity
3. RedBlackTreeRisingCity

These three java classes perform all the essential operations required to build the Wayne Enterprise city by assigning a global counter which keeps track of the number of days and execute building each day. Inputs will be executed when the global counter matches the input time of the command.

❖ risingCity.java

Main class which contains all the attributes of the building. Such as the building number, execution time, and the total time taken to complete the buildings.
RisingCity class is the controller class which performs all the operations required to build the Wayne Enterprise. The input file name is read from the command. Each command from the input file is executed one by one. The Min heap and the red-black tree data structure is initialized max size of min-heap is given as 2000 as per requirement mentioned. The global counter is initialized to zero. The first command is read at zero input time. When the insert operation is obtained from the input file, an object is created of the building with information given in the command, the building number, the total time, and the execution time is set to zero.

Variables:

GlobalCounter is maintained to run for the total number of days.
buildtimeAllocator is used to allocate a 5-day cycle for each building.

Functions:

inputData

- Reads the input file name from the command line argument and retrieves the input from the input file
- First while loop runs to calculate the total number days the global counter execution that must take place.
- Reads the input from the input file and executes each building, when the input time matches the global counter then the command is executed , till the building are constructed for a cycle of 5 days then the next building with the smallest input time is chosen.

executeBld

- This function takes care of the building construction the building is built on the bases of the increase in global counter. Every day each building is executed at least for one day
- The heap size is checked for zero, when it is not zero the function executes the existing building for a day and a min heap function is called after every five-day cycle. This min heap function chooses the next building for execution.
- When the building reaches its completion time the building is printed out and it is removed from both the data structures.

execOperations

- When the input time matches the global counter time the exec operation is executed.
- Form the input file the command is read and executed based on print or insert.
- When insert command is encountered the object of the building is created and the instance of the building is added to the minheap data structure and the same instance pointer is added to the red-black tree as well.
- When the print command is executed the two types of print takes place a single building print and a range print. This print uses the red-black tree data structure to do the building. The single print searches for the building number in the red black tree and returns the building along with execution number and total time. Range print query does a in order traversal in the red-black tree and returns the list of building which fall in the range and is printed out. If the return for both the searches are empty the command is printed out as (0,0,0).

❖ MinHeapRisingCity

Min-Heap class of Wayne Enterprise.

Variables

risingCity[] riseHeap; - array that holds the minheap which is initialized with 2000 as per requirement.

Function

parent, leftChild,rightChild

- These functions return the respective position of the parent, left child and right child node position of the particular node

isLeafNode

- This node returns true or false if the given node is leaf or not
-

minHeapify

- This function is used to prioritize the next building for execution.
- When the cycle of 5 days is reached the min-heap chooses the next building, using this function based on execution time the least executed building is picked up. When the buildings have the same execution time then the tie is broken and the building with smaller building number is picked for execution.

Construct

- This is the insert function of the min-heap. Each time an insert is encountered this function adds the object to the min-heap array.
- Inserted array is based on its execution time the object is moved up to the child of the root node. As the newly inserted node will have the least execution time and tie with the execution is broken using building number.

removeMinRisingCiti

- Once the building reaches its completion the object has to be removed from the data structure.
- Thus, the remove function deletes the node and replaces it with the last node and calls the min-heapify function to maintain the heap property of the array.

❖ RedBlackRisingCity

Variable

Red and black: are assigned as final and 0 and 1 values to show the represent the color of the node

class RBNode

This the structure that is used as link list with left, right, parent pointers and color attribute to represent the color of the node.

Functions:

grandparent

- This function returns the grand parent node of the given node.

Insert

- This insert function used to insert a node into the data structure.
- Using the while a binary search traversal is done. By comparing each node with the new node that is being inserted and the moves left if the new node is small or moves right if the new node is larger than the existing nodes.
- When the null node is encountered the new node is inserted at the space.
- Each node inserted is initially of red color.
- When there are two consecutive objects with red color. An imbalance is created.
- The repair RBT function is called to restore the red-black tree property.

repairRBT

- In the repairRBT function the imbalance is fixed by calculating the nodes parents' sibling node or the uncle node.
- The uncle node is retrieved and checked if it is black or red.
- If the uncle node is black only a color change operation of the uncle and parent node takes place which will fix the imbalance.
- If the uncle node is black the respective rotation needs to be done and color change.
- If the uncle node is on the left then the rotation is done in anti-clockwise direction, else it done the other way by checking the nodes child node and respective rotation are done to restore the imbalance caused.

Delete

- This performs the delete operation where the building when completed the building number is sent to the function.
- Building is searched for in the tree and the building object is deleted by calling the delete replacement function.
- Then there is an imbalance is created then the imbalance is resolved by calling the delete fixup function.

deleteReplacement

- The delete replace function finds close non null node.
- Replaces the child not null node to the node which is to be deleted.
- This searches for smallest node in right subtree if, the node deleted has 2 children.

- If the node is leaf node then delete is done and imbalance is resolved by rotation or color change depending on the node deleted

deleteFixup

- This function fixes the imbalance caused in the delete function and all twelve cases of imbalance are handled in this function.
- When the root node is deleted the smallest node form the right subtree is replaced.
- If the child node or the node is red, the child node is replaced, and respective color change is done.
- If the deleted node is red leaf node then the delete is simple deleted of the node and no imbalance is created.
- If the node deleted has black child node this creates, double black node imbalance where cases are Lr0,Lr2,Rr1,Rb1... all the cases are handled.
- This restores the red-black tree property and maintains balanced binary search tree.

searchRBNode

- This search binary search traversal is done. Object returned if the object is found in the tree.
- When the object is node found the search returns a null value.

searchInRange

- For the search an in-order traversal is done. The result is stored in a list and is sent to main function.
- The main function gets the files and print the value on to the output file.

WORKING OF THE PROGRAM

The Wayne Enterprise is built, by initializing the attributes of the buildings and setting the Global counter to zero. Each command is executed when the input time

mentioned in the command matches the time of the global counter. The input file name is got from command line argument, and the commands are read, the first command is read at input time 0, matching the global counter and respective command is executed.

First building is inserted into the Data Structures. While the insert command is executed, all the building's information is extracted from the command, and the building object is created with this information and assigning the execution to be zero. This instance that is created is added to the min-heap data structure, and the same instance pointer is added to the red-black tree.

The execution of the buildings starts at day 0, each day of execution carried out the global counter increase by one. The building-time allocator is present to take care of days of execution of each building does not exceed five days at each cycle of execution. Once a building is executed for five consecutive days. The min heapify function is called from the min-heap data structure, and the next building with the lowest execution time is chosen. If the buildings have the same execution time, then the tie is broken with the building number, and the building executed. This cycle is repeated. Each update made here is automatically updated in the red-black tree as the same pointer objects are used.

The print operation is got from the command and uses the red-black tree data structure to retrieve the building information at log n time. According to the type of print command, the respective search is done. Data is retrieved from the red-black tree and is sent back to the main function to be printed out to the file.

The global counter runs till the day of command by executing the building, when the day and input time matches. The execution of building for that day is completed, and then the command is executed.
When the print command and the completion of the building happens on the same day, then the building is printed out first, then the print command is executed.

Once all the commands are completed, the remaining buildings are completed by doing the same cycle and functionality, and all the buildings are printed out along with their respective finish dates.

CONCLUSION

The Key observation made while using a red-black tree and min heap as data structure is that search, insert and delete from the tree. The traversing happens only till the height of the tree at the worst-case scenarios, and thus all the operations are performed in O(log N) time. Appropriate implementation of the respective data structures covering all the cases of the data structure has been included and executed. Thus all the buildings of the Wayne rising city are built, with all its operations running at O(log N).

Command Execution: **java risingCity input.txt**

INPUT :

```
0: Insert(50,100)
45: Insert(15,200)
46: PrintBuilding(15)
90: PrintBuilding(0,100)
92: PrintBuilding(0,100)
93: Insert(30,50)
95: PrintBuilding(0,100)
96: PrintBuilding(0,100)
100: PrintBuilding(0,100)
135: PrintBuilding(0,100)
140: Insert(40,60)
185: PrintBuilding(0,100)
190: PrintBuilding(0,100)
195: PrintBuilding(0,100)
200: PrintBuilding(0,100)
209: PrintBuilding(0,100)
211: PrintBuilding(0,100)
```

OUTPUT :

```
(15,1,200)
(15,45,200),(50,45,100)
(15,47,200),(50,45,100)
(15,50,200),(30,0,50),(50,45,100)
(15,50,200),(30,1,50),(50,45,100)
(15,50,200),(30,5,50),(50,45,100)
(15,50,200),(30,40,50),(50,45,100)
(15,50,200),(30,45,50),(40,45,60),(50,45,100)
(30,190)
(15,50,200),(40,45,60),(50,45,100)
(15,50,200),(40,50,60),(50,45,100)
(15,50,200),(40,50,60),(50,50,100)
(15,55,200),(40,54,60),(50,50,100)
(15,55,200),(40,55,60),(50,51,100)
(40,225)
(50,310)
(15,410)
```