ADS PROJECT REPORT

UFID: 6751-2967
NAME:MOHAMMED HAROON RASHEED K
MAIL ID : mkalilurrahman@ufl.edu

Classes

1) **RisingCity.Java**
   - **Variables:**
     - Consists of the main class to run the program
     - Global counter is maintained in this class to read the input
     - Enum is defined for type of operation
       - Insert
       - Print

   Functions:
   - **Main function**
     - Total time needed to run the buildings is calculated in the first while loop
     - Main while loop where the construction occurs is run until the number of days needed by the buildings to finish the construction
     - In each iteration of while loop
       - Input is read from the file
       - Input time is read and is compared with the global counter
       - If the global counter is equal to input time then input is read otherwise global counter is incremented until input time is reached. In each increment min heap will be executed and building construction will occur for each increment of the while loop. Each increment is equivalent to a day of construction.
     - After input time has been reached depending on the type of operation if statement is executed and any one of the operations are performed
       - **Insert**
         - Building number and total time required is read. Building instance is created. If not created, it is a duplicate building and is displayed as (0,0,0)
         - Building is inserted into red black tree (rbt) and into min heap and is ready for execution
       - **Print**
         - **Print range**
           - If print range is the operation , then the range of the buildings is read from the input
           - A search operation is done in RBT for the input and buildings that are found are displayed along with their building number, executed time and total time.

        - If none of the buildings are present then it is displayed as 0,0,0
    - **Print Building**
        - Building is obtained from RBT by doing a traversal in RBT.
        - Building number and executed time is displayed.
        - If building is not present in RBT then construction is already over and 0,0,0 is displayed.
  - Execution for the day happens here. If the building has been inserted on this day then it will be considered for the execution the day it has been inserted itself.
- **Order of operations**
  - **Insert or Print**
  - **Execution of the building**
  - **When a print occurs Print is executed first, therefore print will have time executed until the previous day. Similarily for insert Building is first inserted and then execution happens for the day with the inserted building also in consideration.**

2) **Min heap.java (Min heap implementation)**

<u>Functions:</u>

- **Getparent(elem) –** used to get the parent of the element
- **Getleftchild(elem) –** Used to get the left child
- **Get rightchild(elem) –** get right child
- **isLeaf(elem) –** Check if the element is leaf or not
- **buildingInsert(building b)**
    - Insert the building in heap.
    - Percolate the newly inserted building up by comparing it with the parent's execution time and if it is lesser than parent's execution time. If tie occurs break it with the building number.
- **Execute(RBTBuilding b)**
    - Execution time of the root building is incremented by each day.
    - If execution time is equal to 5 or if the totaltime = executedtime. Then call heapify. If totaltime = executedtime remove the building from RBT and swap the root with the last element and call heapify.
- **removeMinOnCompletion(RBTBuilding rbt)**
    - if the building has finished construction as total time is equal to executed time remove it from RBT and swap the root with the last element.
- **Heapify()**
    - heapify function is used to select the building after 5 days or when a building finishes construction.

- check if the left child execution time is greater than right child if yes select right child as smaller child else left child. If execution time is equal break the tie by comparing the building numbers.
- once smallest of the left and right child is taken, now compare it with parents execution time and again if tie occurs break it with building number.
- swap the parent with the smallest child and keep comparing the parent with the children and pushing it down if necessary.

3) **RBTBuilding.java (Red Black Tree implementation)**
   - **Variables**
     - **Red = 0**
     - **Black = 1**
     - **Node**
       1. **Builiding** (to be inserted in the node)
       2. **Node color (black default)**
       3. **Parent node**
       4. **Left child and Right Child**
   - **Functions**
     - **Insert (Building)**
       1. If root node is nil and there are no nodes in Red Black Tree then create a root node and insert the building in the root node
       2. Else
          a. Create a node and insert the building with red color
          b. If new building number is smaller than current building number then go to left and check for empty. if not empty assign left node as the current node and keep traversing else if left node is empty, insert node and break.
          c. If new building number is greater than current building number then go to right and check for empty. if not empty assign right node as the current node and keep traversing else if right node is empty, insert node and break.
          d. Call balanceRBT function to balance the tree with the newly inserted node
     - **balanceRBT(newnode)**
       1. Get the uncle node.
       2. If uncle node is red, no rotation is needed only color change is enough. Change parent's and uncle's node color as black and grandparent's node color as red. (**Only color flip)**
       3. Else if uncle node is black, rotation is required
          a. if inserted node is right node perform rr rotation
          b. else if inserted node is left node perform ll rotation
          c. Set parent's node color as black

d. Set Grandpa's node color as red
e. if uncle is right perform ll rotation with grandparent node
f. else if uncle is left perform rr rotation with grandparent node

- **doColorChange(newNode, unclenode)**
  1. change the uncle color and parent color to black
  2. change the grandparent color to red
- **rotateLeft(RBTNode newNode)**
  1. call rotate rootleft method if node to be performed ll rotation is root
  2. Else
     a. if node is a left child set node's right child as parents left child.
     b. else set node's right child as parent's right child.
     c. set node's parent as parent of node's right child
     d. set node's right child as node's parent
- **rotateRight(RBTNode newNode)**
  1. call rotate rootright method if node to be performed rr rotation is root
  2. Else
     a. if node is a left child set node's left child as parents left child.
     b. else set node's left child as parent's right child.
     c. set node's parent as parent of node's left child
     d. set node's left child as node's parent.
- **rotateRootLeft()**
  1. set right node of root as temp variable
  2. set root as parent of temp variable's leftchild
  3. set temp variable as root's parent
  4. set root as leftchild of temp right child
  5. set temp variable parent as nil since temp is the new root
- **rotateRootRight()**
  1. set left node of root as temp variable
  2. set right child of root's left child as root's left child
  3. set root as parent of temp's right child
  4. set temp variable as root's parent
  5. set root as temp's right child
  6. set temp's parent as nil since temp is new root
- **findRBT(int findNodeBN, RBTNode parentNode)**
  1. if node is greater than parent, node is in right side of parent
  2. else if node is smaller than parent, node is in left side of parent
  3. else element is present in the current location

- **deleteBuildingFromRBT(int deleteBuildingNum)**
    1. if deleted node's left child is leaf, replace node with right child.
    2. if deleted node's right child is leaf, replace node with left child
    3. if node has both right and left child.
    4. set temp node as delete node's right child. Traverse left node until u reach a node with no left child.
    5. replace temp node with the temp variable's right node.
    6. replace delete node and temp node.
- **searchInOrder(RBTNode beginNode,List<Building> buildingPrint, int begin, int finish)**
    1. In order traversal to get the buildings and print it
    2. If beginNode is greater than begin, then call SearchInorder recursively with beginNode's leftchild as begin node.
    3. If beginNode is greater than or equal to begin and finish is greater than equal to beginNode building numberadd building to the list as it is in the range of search.
    4. If beginNode is smaller than finish, then call SearchInorder recursivelywith beginNode's rightchild as begin node.