ADS Project Report

Name: Rachana Nitinkumar Gugale
UFID: 6353-2454
Email: rgugale@ufl.edu

Red Black Trees

- The height of a regular binary search tree (BST) can be ***n*** in the worst case if the tree is completely skewed.
- This means that the time complexity of operations like insert, delete and search in a BST is O(n). We can achieve better time complexity if the tree is balanced.
- Red Black Trees are a type of balanced binary search trees where the height of the tree lies between $\log_2(n+1)$ and $2 * \log_2(n+1)$.
- AVL trees are another type of balanced BSTs with $\log_2(n)$ height. The problem with AVL trees is that in the case of deletion from an AVL tree, the tree might have to do $\log_2(n)$ rotations in the worst case to balance itself. Red Black trees avoid this and guarantee a maximum of only 1 rotation for any operation.
- The nodes in a red black tree have to satisfy the following properties to achieve a maximum tree height of $2 * \log_2(n+1)$:
    - Every node is colored either red or black.
    - The root and external nodes are always black.
    - The number of black nodes on any path from the root to any external node is the same.
    - No two red nodes appear on a path consecutively.
- The following operations can be done on a red black tree:
    - **Insert** –
        - Insert a new node in the tree.
        - The new node is always inserted as a red node.
        - If there is a red-red conflict, the tree is fixed through a series of rotations and recolorings.
        - There can be multiple cases involved in balancing the tree, depending on the color of the current node, its parent and uncle node.
        - Time complexity of insert = $O(height) = O(log_2(n))$
    - **Delete** –
        - Delete a node from the tree.
        - This operation involves searching for the node, replacing it with its inorder successor or predecessor till the node to be deleted doesn't have any children or has only a red child.
        - After deleting the node, the tree is balanced through a series of rotations and recolorings.
        - Here too, there can be multiple cases involved depending on the color of the current node, its parent, its sibling and sibling's children.
        - Time complexity of delete = $O(height) = O(log_2(n))$

- **Search** –
  - Since red-black trees are BSTs and follow the basic BST property of the left child < parent < right child, search in red-black trees is exactly the same as that in BSTs.
  - Time complexity of search = $O(height) = O(log_2(n))$

Min Heaps

- Min heaps are complete binary trees.
- Each node has two children and the values of both children are greater than the parent.
- The root of the min heap is the smallest node in the tree.
- Min heaps are used as priority queues where the item with the next high priority node can be obtained in O(1) time (as it is the root node).
- As min heaps are complete binary trees, their height is $O(log_2(n))$.
- Min heaps support the following operations:
  - **Insert**
    - Insert a new node in the heap.
    - The new node is inserted as the last node in the heap and then bubbled up to its right place using a heapifyUpwards operation which compares the parent node with the current node and swaps the node with its parent if the parent is bigger.
    - In the worst case, if the newly inserted element is the smallest in the heap, it will take $log_2(n)$ swaps to reach its right place i.e. the top of the heap.
    - Time complexity of insert = $O(height) = O(log_2(n))$
  - **FindMin**
    - Returns the root node.
    - Time complexity of findMin = $O(1)$
  - **DeleteMin**
    - Delete the smallest element i.e. the root node.
    - The root is replaced with the last heap element.
    - The root is detached from the tree and the new root is now bubbled down through a heapifyDown operation till it reaches its correct position in the heap.
    - In the worst case, the heapifyDown operation might have to do $log_2(n)$ swaps.
    - Time complexity of deleteMin = $O(height) = O(log_2(n))$

Structure of the program

My program consists of the following files:

1. gatorTaxi.py
2. red_black_tree.py
3. min_heap.py
4. output_file.txt – Stores the output

gatorTaxi.py

This file contains the driver program. It contains a class called CabService which implements the functions defined in the problem statement with the help of functions from RBTree and MinHeap classes. This file also contains code to parse the input file and write the output to the output file.

a. **CabService.print(self, rideNumber):**
   i. Searches for the ride with rideNumber in red black tree and prints it. If such a ride is not found, prints (0,0,0).
   ii. **Time complexity:** $O(log_2(n))$ as search in a red black tree can be done in $O(log_2(n))$.
   iii. **Space complexity:** $O(1)$ as no extra space is needed by this operation.

b. **CabService.printRange(self, rideNumber1, rideNumber2):**
   i. Searches for all the rides with ride numbers between rideNumber1 and rideNumber2 in the red black tree and prints all such rides in ascending order of ride numbers.
   ii. **Time complexity:** $O(log_2(n) + S)$ where S is the number of rides in the range. This function first finds the lower bound node in the red black tree and then does an inorder traversal till the upper bound node.
   iii. **Space complexity:** $O(S)$ as I have used a list to store all the nodes in range to sort them before displaying.

c. **CabService.insert(self, rideNumber, rideCost, tripDuration):**
   i. Inserts a new ride in the red black tree and the min heap.
   ii. If a ride with the same rideNumber is already present in the system, an error message is printed and the program exits.
   iii. **Time complexity:** $O(log_2(n))$ as insert in red black tree and min heap can both be done in that time.
   iv. **Space complexity:** $O(1)$. Constant amount of space needed to create new nodes in red black tree and min heap to insert the new ride.

d. **CabService.getNextRide(self):**
   i. Gets the next ride with the lowest rideCost.
   ii. It accomplishes this by getting the root of the min heap and deleting it. Also deletes the corresponding red black tree node by using the pointer to it stored in the min heap node.
   iii. If no active rides are present, an error message is printed.
   iv. **Time complexity:** $O(log_2(n))$ time as deleteMin in min heap takes $O(log_2(n))$ time and delete in a red black tree also takes the same time.
   v. **Space complexity:** $O(1)$. No extra space needed.

e. **CabService.cancelRide(self, rideNumber):**
   i. Deletes nodes with the given rideNumber from red black tree and min heap. The function does this by first searching for the node with the given rideNumber from the red black tree and deleting it.
   ii. The corresponding min heap node is also deleted by using the pointer to it from the red black tree node.

iii. **Time complexity:** $O(log_2(n))$ as deletion from red black trees takes $O(log_2(n))$ and heapify takes $O(log_2(n))$.
iv. **Space complexity:** $O(1)$. No extra space needed.

f. **CabService.updateTrip(self, rideNumber, newTripDuration):**
i. Updates the tripDuration for the ride.
ii. The trip cost is also updated based on the newTripDuration.
iii. If newTripDuration > oldTripDuration but less than twice the oldTripDuration, then rideCost is increased by 10.
iv. If the newTripDuration is more than twice the oldTripDuration, the ride is deleted. The function implements the updation operation by first deleting the old trip from the data structures and inserting a new trip with the same rideNumber but updated cost and duration in the system.
v. **Time complexity:** $O(log_2(n))$ as it is a combination of cancelRide and insert operations which both take $O(log_2(n))$.
vi. **Space complexity:** $O(1)$. Constant amount of space needed to create new nodes in red black tree and min heap to insert the updated ride.

min_heap.py

This file contains 2 classes – MinHeapNode and MinHeap.

1. **MinHeapNode class** – defines the structure of every min heap node. This class doesn't contain any functions.
   - *MinHeapNode structure:*
     - rideNumber
     - rideCost
     - tripDuration
     - listIdx – The index at which the current node is stored in the heap array.
     - RBTNode – Pointer to the corresponding RedBlackTree node

2. **MinHeap class** – defines the functions used to create and manipulate a heap of MinHeapNode objects. It contains a *heapNodeList* class variable to store the heap nodes and a *currentHeapSize* variable to store the heap size. The class contains the following min heap functions:
   a. **MinHeap.insert(self, rideNumber, rideCost, tripDuration, RBTNode)**:
      i. Inserts a node with the given parameters in the min heap as its last element. Bubbles this node up using heapifyUpwards() till the new node reaches its correct position in the heap.
      ii. **Time complexity:** $O(log_2(n))$ as heapifyUpwards() takes that long.
      iii. **Space complexity:** $O(1)$. Constant space needed to create new node.
   b. **MinHeap.deleteMin(self):**
      i. Deletes the root of the heap and returns it.
      ii. It first swaps the last heap element with the root and then deletes the root.
      iii. The new root is then moved to its correct position by heapifyDownwards().
      iv. **Time complexity:** $O(log_2(n))$ as heapifyDownwards() takes that long.
      v. **Space complexity:** $O(1)$. No extra space needed.

c. **MinHeap.deleteArbitraryByIdx(self, idx):**
   i. Deletes the node present at index idx in the heap list.
   ii. Swaps the node at idx with the last heap node.
   iii. Depending on whether the last node is smaller or greater than the old node, heapifyUpwards or heapifyDownwards is called.
   iv. **Time complexity:** $O(log_2(n))$ as heapifyUpwards() or heapifyDownwards() takes that long.
   v. **Space complexity:** $O(1)$. No extra space needed.

d. **MinHeap.heapifyUpwards(self, i):**
   i. Takes the node at index i in the heap list and bubbles it to the top by doing swaps with the parent.
   ii. Swaps are done till the node is smaller than the parent.
   iii. **Time complexity:** $O(height) = O(log_2(n))$ as maximum those many swaps might be needed.
   iv. **Space complexity:** $O(1)$. No extra space needed.

e. **MinHeap.heapifyDownwards(self, i):**
   i. Takes the node at index i in the heap list and bubbles it down till it is greater than its children by doing swaps with the minimum child.
   ii. **Time complexity:** $O(height) = O(log_2(n))$ as maximum those many swaps might be needed.
   iii. **Space complexity:** $O(1)$. No extra space needed.

f. **getMinChildIdx(self, i):**
   i. Utility function used by heapifyDownwards to find the minimum child.
   ii. **Time complexity:** $O(1)$
   iii. **Space complexity:** $O(1)$. No extra space needed.

red_black_tree.py

This file contains 2 classes RedBlackTreeNode and RedBlackTree.

1. **RedBlackTreeNode class** – defines the structure of every red black tree node. This class doesn't contain any functions.
   - *RedBlackTreeNode structure:*
     - rideNumber
     - rideCost
     - tripDuration
     - parent – Pointer to the parent node
     - left – pointer to the left child
     - right – pointer to the right child
     - color – color of the node. Either red or black.
     - minHeapNode – Pointer to the corresponding MinHeap node

2. **RedBlackTree class** – defines the functions used to create and manipulate a red black tree of RedBlackTreeNode objects. It contains a class level *root* variable to maintain the red black tree's root. Another class variable *TNULL* is declared for pointing to external nodes to avoid any Null Pointer Exceptions. The class contains the following red black tree functions:
    a. **insert(self, rideNumber, rideCost, tripDuration):**
        i. Inserts a node in the red black tree.
        ii. The new node is always inserted as a red node.
        iii. If there is a red-red conflict, it is fixed by a calling balanceAfterInsert().
        iv. **Time complexity:** $O(log_2(n))$. Searching for the location to insert and fixing the tree after insert can take logarithmic time.
        v. **Space complexity:** $O(1)$. Constant space needed to create new node.
    b. **search(self, key, node):**
        i. Search works just the way it does in a BST.
        ii. **Time complexity:** $O(log_2(n))$
        iii. **Space complexity:** $O(1)$. No extra space needed.
    c. **printRange(self, node, lowerBound, upperBound, res):**
        i. Prints all the nodes lying between lower and upper bound.
        ii. Works by first searching for the lowerBound node and then doing an inorder traversal till the upperBound node.
        iii. **Time complexity:** $O(log_2(n) + S)$. S is the number of nodes in range. A factor of S is added because of inorder traversal.
        iv. **Space complexity:** $O(S)$ as I have used a list to store all the nodes in range to sort them before displaying.
    d. **deleteNode(self, node1):**
        i. Deletes the node from the red black tree.
        ii. If direct deletion is not possible, replaces the node with its inorder successor.
        iii. Node is deleted after calls to rbRotate() and balanceAfterDelete(). There can be multiple orders in which these functions are called depending on the existence, color and position of the parent, sibling and sibling's children nodes.
        iv. **Time complexity:** $O(log_2(n))$. To find the node to be deleted and to fix the tree after deletion.
        v. **Space complexity:** $O(1)$. No extra space needed.
    e. **balanceAfterInsert(self, currNode):**
        i. Balances the tree if there is a red-red conflict after insertion by performing a series of recolorings and atmost 1 rotation.
        ii. The order of recolorings and type of rotation depends on which case the insert falls under. There can be various cases depending on the color of the parent and uncle nodes.
        iii. **Time complexity:** $O(log_2(n))$. These many recolorings might be required in the worst case.
        iv. **Space complexity:** $O(1)$. No extra space needed.
    f. **balanceAfterDelete(self, currNode):**
        i. Balances the tree after a delete operation series of recolorings and atmost 1 rotation.

ii. **Time complexity:** $O(log_2(n))$. These many recolorings might be required in the worst case.
iii. **Space complexity:** $O(1)$. No extra space needed.

g. **rotateLeft(self, node1):**
   i. Performs a left rotation to balance the tree.
   ii. **Time complexity**: $O(1)$.
   iii. **Space complexity:** $O(1)$. No extra space needed.

h. **rotateRight(self, node1):**
   i. Performs a right rotation to balance the tree.
   ii. **Time complexity**: $O(1)$.
   iii. **Space complexity:** $O(1)$. No extra space needed.

i. **rbRotate(self, node1, node2):**
   i. Performs a rotation to balance the tree. Used by deleteNode().
   ii. **Time complexity**: $O(1)$.
   iii. **Space complexity:** $O(1)$. No extra space needed.

j. **getSmallestNodeInTree(self, currNode):**
   i. Utility function used to get the inorder successor.
   ii. Used by deleteNode().
   iii. **Time complexity:** $O(log_2(n))$. As there could be "height" number of nodes traversed in the worst case.
   iv. **Space complexity:** $O(1)$. No extra space needed.

Steps to run the program

- The program can be run with the following command:

  python3 gatorTaxi.py <input_filename>