COP 5536 Spring 2023
Programming Project Report

(Ratna Prabha Bhairagond, 8827-4983, r.bhairagond@ufl.edu)

Code Structure

Class: ***minHeapRide***

The *minHeapRide* class represents a ride object that can be stored in a min heap data structure.

- Data Members:
  - **public int rideNumber:** An integer variable that represents the ride number.
  - **public int rideCost:** An integer variable that represents the ride cost.
  - **public int tripDuration:** An integer variable that represents the trip duration in minutes.
  - **public redBlackTreeNode<redBlackTreeRide> redBlackTreeNodePointer:** A reference to a *redBlackTreeNode* object, which is used for linking the ride object to a red-black tree node.
- Constructor:
  - **public minHeapRide(int rideNumber, int rideCost, int tripDuration):** A constructor that takes the ride number, ride cost, and trip duration as arguments and initializes the corresponding data members of the *minHeapRide* object.
- Methods:
  - **@Override int compareTo(Object object):** A method that compares the *minHeapRide* object with another object. It compares the ride cost and trip duration of the two objects and returns -1, 0, or 1 based on the comparison result.
  - **@Override String toString():** A method that returns a string representation of the *minHeapRide* object. It returns a string containing the ride number, ride cost, and trip duration of the object in a formatted manner.

Class: ***minHeapNode***

The *minHeapNode* class represents a node in the *minHeap* data structure implemented in Java.

- Data Members:
  - **minHeapRide data:** An object of type *minHeapRide* that represents the data to be stored in the node of the heap.
  - **int positionIndex:** An integer variable that represents the position index of the node in the binary heap. It is used to track the position of the node in the heap array.
- Constructor:
  - **public minHeapNode(minHeapRide data, int positionIndex):** A constructor that takes the data to be stored in the node and its position index as arguments and initializes the corresponding data members of the *minHeapNode* object.

Class: ***minHeap***

The *minHeap* class represents a minimum heap data structure implemented in Java. It is used to store objects of type *minHeapRide* in a binary heap format, where the object with the smallest value (as determined by its *compareTo()* method) is always at the root of the heap.

- Data Members:
  - **private final minHeapNode[] Heap:** An array of *minHeapNode* objects that represents the binary heap. The binary heap is implemented as an array.
  - **private int size:** An integer variable that represents the number of elements currently stored in the heap.
  - **private final int maximumSize:** An integer variable that represents the maximum number of elements that the heap can store.
  - **private static final int startPositionIndex:** A constant integer variable that represents the starting position index of the heap, which is always set to 1. It is used to access the root of the heap.
- Constructor:
  - **public minHeap(int maximumSize):** This is a constructor for the *minHeap* class which initializes the min heap with a specified *maximumSize*. It creates an array to store the min heap structure, sets the current size to 0, and creates a sentinel node at position 0 in the array.
- Methods:
  - **private int getParentPositionIndex(int positionIndex):** This function calculates the index of the parent node in the min heap, given the index of a node.
  - **private int getLeftChildPositionIndex(int positionIndex):** This function calculates the index of the left child node in the min heap, given the index of a node.
  - **private int getRightChildPositionIndex(int positionIndex):** This function calculates the index of the right child node in the min heap, given the index of a node.
  - **private boolean isLeaf(int positionIndex**): This function checks if a node is a leaf node in the min heap, given its index.
  - **private void swap(int positionIndex1, int positionIndex2):** This function swaps two nodes in the min heap, given their indices.
  - **private void minHeapify(int positionIndex):** This function performs the min heapify operation on a node in the min heap, given its index. It takes an integer *positionIndex* as input, which represents the index of the node to be heapified, and recursively moves the node down the min heap until the min heap property is restored.
  - **public minHeapNode insert(minHeapRide minHeapRide):** This function inserts a new node with a *minHeapRide* object as its data into the min heap.
  - **public minHeapRide deleteMin():** This function deletes the minimum element (root) from the min heap and returns it. It removes the root node from the min heap and restores the min heap property by performing the min heapify operation.
  - **public void arbitraryDelete(minHeapNode minHeapNode):** This function deletes a node from the min heap, given its *minHeapNode* object. It takes a *minHeapNode* object as input, representing the node to be deleted from the min heap, and removes the node from the min heap and restores the min heap property by performing the min heapify operation.

Class: ***redBlackTree***

A generic class that implements a red-black tree data structure.

- Data Members:
  - **private redBlackTreeNode<T> root:** A private field of type *redBlackTreeNode<T>* that represents the root node of the red-black tree.
  - **private final redBlackTreeNode<T> externalRedBlackTreeNode:** A private field of type *redBlackTreeNode<T>* that represents an external node used in the red-black tree as a sentinel.
- Constructor:
  - **public redBlackTree():** Constructor for the *redBlackTree* class. It initializes the *externalRedBlackTreeNode* and sets it as the root of the tree.
- Methods:
  - **public T find(Object value):** A public method that finds a node with the specified value in the red-black tree by calling the private *findHelper* method.
  - **private T findHelper(redBlackTreeNode<T> redBlackTreeNode, Object value):** A private helper method that recursively finds a node with the specified value in the red-black tree.
  - **public int printInRange(Object value1, Object value2, T[] results):** A public method that prints all values within the specified range [value1, value2] in the red-black tree by calling the private *printInRangeHelper* method.
  - **private void printInRangeHelper(redBlackTreeNode<T> redBlackTreeNode, Object value1, Object value2, T[] results):** A private helper method that recursively prints all values within the specified range [value1, value2] in the red-black tree.
  - **private void rotateToLeft(redBlackTreeNode<T> redBlackTreeNode):** A private method that performs a left rotation on the specified node in the red-black tree.
  - **private void rotateToRight(redBlackTreeNode<T> redBlackTreeNode):** A private method that performs a right rotation on the specified node in the red-black tree.
  - **public redBlackTreeNode<T> insert(T data):** This method inserts a new node with the specified data into the red-black tree. It follows the red-black tree insertion algorithm to maintain the properties of the red-black tree.
  - **private void refactorInsert(redBlackTreeNode<T> newRedBlackTreeNode):** A private method that performs the necessary rotations and recolouring after inserting a new node into the red-black tree to maintain the red-black tree properties.
  - **public T deleteData(Object data):** This method deletes a node with the specified data from the red-black tree. It first searches for the node with the specified data in the tree, and then removes it while maintaining the properties of the red-black tree.
  - **private redBlackTreeNode<T> minimumOfSubTree(redBlackTreeNode<T> redBlackTreeNode):** This is a private helper method that finds and returns the minimum node (leftmost node) in the subtree rooted at the specified red-black tree node.
  - **public void deleteNode(redBlackTreeNode<T> deleteRedBlackTreeNode):** This method deletes the specified red-black tree node from the red-black tree. It first finds the child and replacement nodes for the node to be deleted, and then replaces the node with its child while maintaining the properties of the red-black tree.
  - **private void refactorDelete(redBlackTreeNode<T> redBlackTreeNode**): This is a private helper method that performs the necessary adjustments to maintain the properties of the red-black tree after a deletion operation. It takes a *redBlackTreeNode* as input and performs the necessary rotations and recolouring's to restore the red-black tree properties after deleting a node.

Class: ***redBlackTreeNode<T extends Comparable<Object>>***

The redBlackTreeNode class represents a node in a Red-Black Tree.

- Data Members:
  - **public T data:** A generic type T representing the data stored in the node.
  - **public redBlackTreeNode<T> parent:** A reference to the parent node of the current node.
  - **public redBlackTreeNode<T> leftChild:** A reference to the left child node of the current node.
  - **public redBlackTreeNode<T> rightChild:** A reference to the right child node of the current node.
  - **public colour colour:** An enum colour representing the colour of the node, which can be either Red or Black.
- Constructor:
  - **public redBlackTreeNode(T data, redBlackTreeNode<T> parent, redBlackTreeNode<T> leftChild, redBlackTreeNode<T> rightChild, colour colour):** Constructor that takes in the data, parent node, left child node, right child node, and color of the node as parameters and initializes the corresponding data members.

Class: ***redBlackTreeRide***

The *redBlackTreeRide* class represents a ride in a Red-Black Tree data structure.

- Data Members:
  - **public int rideNumber:** An integer representing the unique number assigned to the ride.
  - **public int rideCost:** An integer representing the cost of the ride.
  - **public int tripDuration:** An integer representing the duration of the ride.
  - **public minHeapNode minHeapNodePointer:** A reference to a *minHeapNode* object, which is a node in a Min Heap data structure.
- Constructor:
  - **public redBlackTreeRide(int rideNumber, int rideCost, int tripDuration):** Constructor that takes in the ride number, ride cost, and trip duration as parameters and initializes the corresponding data members.
- Methods:
  - **@Override public int compareTo(Object object):** Compares the ride number of this object with another object.
  - **@Override public String toString():** Method to provide a custom string representation of the object, which includes the ride number, ride cost, and trip duration concatenated as a string in a specific format.

Class: ***gatorTaxi***

This class represents a taxi service that manages rides using a red-black tree and a min-heap data structure.

- Data Members:
  - **public static redBlackTree<redBlackTreeRide> redBlackTree:** A static instance of *redBlackTree* class, which is a red-black tree implementation that stores *redBlackTreeRide* objects.
  - **public static minHeap minHeap:** A static instance of *minHeap* class, which is a min-heap implementation that stores *minHeapRide* objects.
- Methods:
  - **public static String insert(int rideNumber, int rideCost, int tripDuration):** Inserts a new ride with the given ride number, ride cost, and trip duration into both the red-black tree and the min-heap. Adds red-black node reference to minheap node and vice-versa. Returns *"Duplicate RideNumber"* if the ride number already exists in the red-black tree.
  - **public static String getRidesInRange(int rideNumber1, int rideNumber2):** Retrieves a list of rides with ride numbers in the range of *rideNumber1* and *rideNumber2* (inclusive) from the red-black tree. Returns a string representation of the rides in the format *"(rideNumber,rideCost,tripDuration),(rideNumber,rideCost, tripDuration),..."* if there are rides in the range, or *"(0,0,0)"* if there are no rides in the range.
  - **public static String getRidesInRange(int rideNumber):** Retrieves the ride with the given ride number from the red-black tree. Returns a string representation of the ride in the format *"(rideNumber, rideCost, tripDuration)"* if the ride exists, or *"(0,0,0)"* if the ride does not exist.
  - **public static String getNextRide():** Retrieves the ride with the lowest ride cost from the min-heap, and removes it from both the min-heap and the red-black tree. Returns a string representation of the ride in the format *"(rideNumber, rideCost, tripDuration)"* if there is a ride in the min-heap, or *"No active ride requests"* if there are no rides in the min-heap.
  - **public static void cancelRide(int rideNumber):** Deletes the ride with the given ride number from both the red-black tree and the min-heap, if it exists.
  - **public static void updateTrip(int rideNumber, int new_tripDuration):** Updates the trip duration of the ride with the given ride number in both the red-black tree and the min-heap, if it exists. If the new trip duration is less than or equal to the current trip duration, only the trip duration is updated. If the new trip duration is between the current trip duration and twice the current trip duration, the ride is cancelled and reinserted with an updated ride cost. If the new trip duration is greater than twice the current trip duration, the ride is cancelled.
  - **public static void main(String[] args):** The entry point of the program. Reads input from a file specified in the command line arguments, calls the appropriate methods based on the input, and writes the output to a file named "output_file.txt".

Space and Time Complexities

1. **Print(rideNumber)**

   Time Complexity: O(log n), where n is the number of elements in the red-black tree. Method would start at the root of the tree and compare the target *rideNumber* with the value of the current node. If the target *rideNumber* is smaller, the search would continue in the left subtree, and if it is larger, the search would continue in the right subtree. At each step of the search, the number of nodes that need to be visited is reduced by half, resulting in a logarithmic time complexity of O(log n), where n is the number of nodes in the tree. This is because the height of a balanced Red-Black Tree is logarithmic with respect to the number of nodes in the tree.

   Space Complexity: O(1), as the additional space used by this method does not depend on the size of the data structure or input.

2. **Print(rideNumber1, rideNumber2)**

   Time Complexity: O(log(n) + S), where n is the number of active rides in the Red-Black Tree and S is the number of rides printed.

   - Search for rideNumber1 in the Red-Black Tree: The first step is to search for *rideNumber1* in the Red-Black Tree, which has a time complexity of O(log n), where n is the number of nodes in the tree.
   - Search for rideNumber2 in the Red-Black Tree: The second step is to search for *rideNumber2* in the Red-Black Tree, which also has a time complexity of O(log n), where n is the number of nodes in the tree.
   - Traverse the subtree to print rides in range: Once *rideNumber1* and *rideNumber2* are found in the Red-Black Tree, the method needs to traverse the subtree rooted at the node containing rideNumber1 to print all the rides in the range between *rideNumber1* and *rideNumber2*. The time complexity of this step would depend on the number of rides, S, that fall within the range. If S rides need to be printed, the time complexity of this step would be O(S).
   - Overall time complexity: Adding up the time complexities of the above steps, we get O(log n + log n + S) = O(2 log n + S). Since the constant factor 2 is not relevant in big-O notation, we can simplify it to O(log n + S).

   Space Complexity: O(S), as an array is used to store the S ride elements to be printed.

3. **Insert(rideNumber, rideCost, tripDuration)**

   Time Complexity: O(log n)

   - In a Red-Black Tree, the insert operation involves finding the appropriate position for the new node based on its *rideNumber*, inserting the new node as a leaf node, and then performing any necessary rotations and colour adjustments to maintain the Red-Black Tree properties. However, these steps take constant time and do not depend on the size of the tree. Since the height of a balanced Red-Black Tree is at most log n, finding the appropriate position takes at most log n, therefore the insertion operation in a Red-Black Tree takes O(log n) time in the worst case.

- In a Min Heap, the insertion operation involves adding a new element at the bottom of the heap and then "swapping up" the element until the heap property is restored. Since the height of a Min Heap is log n, the insertion operation in a Min Heap takes O(log n) time in the worst case.
- Overall time complexity: O(log n)

Space Complexity: O(1), as the additional space used by this method does not depend on the size of the data structure or input.

4. GetNextRide()

Time Complexity: O(log n)

- In Min Heap, the deleteMin() operation involves removing the root element (the minimum element) from the heap, which leaves an empty spot at the root. To maintain the heap property, the last element (i.e., the element at the last leaf node of the last level) is moved to the root, and then the element is moved down by repeatedly swapping it with its smallest child until the heap property is restored. The "swapping down" process takes at most O(log n) time, as the element being moved down can move down the entire height of the heap.
- In Red Black Tree,
  - Search for the node to be deleted: As from the deleteMin node of minheap the pointer to the redBlackTree node can be accessed. The time complexity of this step will be O(1).
  - Delete the node: Once the node to be deleted is found, the actual deletion operation involves removing the node from the tree and potentially perform rotations and recoloring to restore the properties. These operations take constant time O(1) each, and the number of fix-up operations performed is typically limited to a constant number of operations. Therefore, the time complexity of this step is also O(1).
  - Overall time complexity: O(1)
- Overall time complexity: O(log n)

Space Complexity: O(1), as the additional space used by this method does not depend on the size of the data structure or input.

5. CancelRide(rideNumber)

Time Complexity: O(log n)

- In Red Black Tree,
  - Search for the node to be deleted: This step involves traversing the red-black tree from the root to the node to be deleted, which takes O(log n) time, where n is the number of nodes in the tree.
  - Delete the node: Once the node to be deleted is found, the actual deletion operation involves removing the node from the tree and potentially perform rotations and recoloring to restore the properties. These operations take constant time O(1) each, and the number of fix-up operations performed is typically limited to a constant number of operations. Therefore, the time complexity of this step is also O(1).

  - Overall time complexity: O(log n)
- In Min Heap,
  - Search for the node to be deleted: Since each the deleted *redBlackTree* node has a direct pointer to its corresponding minheap node, finding the node to be deleted can be done in O(1) time.
  - Delete the node: Once the node to be deleted is found, the actual deletion operation involves removing the node from the heap. This step can be done in O(1) time since you have a direct pointer to the node.
  - Fix the heap property: After the node is deleted, the min heap property may be violated, and it may be necessary to perform heapify operations to restore the property. The worst-case time complexity of heapify operations is O(log n), where n is the number of nodes in the heap.
  - Overall time complexity: O(log n)
- Overall time complexity: O(log n)

Space Complexity: O(1), as the additional space used by this method does not depend on the size of the data structure or input.

6. UpdateTrip(rideNumber, new_tripDuration)

Time Complexity: O(log n)

- Search for the node to be update: This step involves traversing the red-black tree from the root to the node to be deleted, which takes O(log n) time, where n is the number of nodes in the tree.
- Cancel the existing node: *CancelRide(rideNumber),* takes O(log n) as shown above.
- Insert a new node: *Insert(rideNumber, rideCost, tripDuration)*, takes O(log n) as shown above.
- Overall time complexity: O(log n)

Space Complexity: O(1), as the additional space used by this method does not depend on the size of the data structure or input.

Red-Black Tree Data Structure

Overall Space Complexity: The space complexity of *redBlackTree* is O(n), where n is the number of active rides in the tree. This is because each node requires a constant amount of space to store its value (e.g., ride number, trip duration, colour, etc.), pointers to its parent, left child, and right child, and additional fields for maintaining the red-black tree properties. Therefore, the total space used by the tree would be proportional to the number of nodes, which is n in this case.

Min Heap Data Structure

Overall Space Complexity: The space complexity of *minHeap* would be O(n), where n is the number of active rides in the heap. As an array of size 2000 is used to store the heap, but it can contain up to n elements. Implementation of a binary min heap, each node requires a constant amount of space to store its value (e.g., ride number and trip duration). Therefore, the total space used by the heap would be proportional to the number of nodes, which is n in this case.