Assignment 4

The assignment 4 comprises of 4 different components namely Red Black Trees, Tries, Priority Queues and Project Scheduler. The implementation details for each of them are as follows:

1. **Tries:**
   - The **class Person** has 2 private fields namely **n(String)** and **ph(String)** which are provided to the constructor during the formation of the object. The method **getName()** returns the **name** of the person(**String n**) and the method **toString()** returns the details of the person as **"[Name: "+n+", Phone="+ph+"]".**

   - The **class TrieNode<T>** contains the private fields:
     - ➢ **children(TrieNode<T> [ ])-** Array of size **size(int)** whose slots correspond to the **ASCII value** of the **character** which is the **key** of the **TrieNode** whose reference is stored in the array.
     - ➢ **parent(TrieNode<T>)** is the reference to the parent of the TrieNode<T>.
     - ➢ **count(int)** contains the number of slots in the array **children(TrieNode<T> [ ])** which aren't empty.
     - ➢ **key(char)** is the **key** (**which can be any character whose ASCII value lies in 32-126**) of the TrieNode. This is passed to the constructor while creating a new TrieNode<T> object.
     - ➢ **value(T)**, represents the **value** stored. The **value** is stored at the **TrieNode** whose **key** is the **last character** of the **name** (**where endOfword is true**).
     - ➢ **size(int)** represents the size of the array required (**95 in this case**). This field is passed to the constructor while creating a new TrieNode<T> object.
     - ➢ **level(int)** represents the level of the **TrieNode<T>** in the Trie. **It is 1 + level of the TrieNode<T> parent.**
     - ➢ **endOfword(boolean)** is a **boolean** which is true for the TrieNode<T> corresponding to the **last character** of the **name**. Also the **TrieNodes** for which **endOfWord is true contains the value stored (Person in this case)**.
     - ➢ There are **getter** and **setter methods** for all the parameters except **children(TrieNode<T> [ ])**.

   - The **class Trie<T> implements the TrieInterface<T>**

Param Khakhar
2018CS10362

Assignment 4

- The fields are **size(int)** which represents the size of the array corresponding to the possible ASCII values(32-126) of the keys, **root(TrieNode<T>** is the TrieNode at the 0th level and whose key is ‘ ’.
- The method **ascii(char c)** returns the **ascii value of the character - 32** as the keys lie from 32-126.
- The method **search(String word)** returns the **TrieNode** corresponding to the **last character of the name if** the **name** is **present** and **null if** the **name** is **absent**. The string name is first converted to an array of its individual characters by using the method **toCharArray().** A for loop is used where the **number of iterations** is **equal to** the **length of the word to be searched**. The **search starts from the root** and **in the (i+1)th iteration (i>=0) of the loop a check is made for the presence of the character(TrieNode with key as character) at the ith index in the name, in the children of the TrieNode whose key is the (i-1)th index of the name**. **If any of the intermediate TrieNodes are null or the endOfword for the last character is False, null is returned otherwise the TrieNode corresponding to the last character is returned.**

The **complexity** of the above algorithm is **O(length of the word to be searched) which is evident from the number of iterations of the for loop in the worst case and the other operations being constant time operations.**

- The method **insert(String word, Object value)** uses an algorithm similar to the method **search()** but in this extra constant time operations are present such as creation of new TrieNodes at each level of the Trie if they aren’t present . The boolean variable **inserted** represents the final result. If in the trie, there exists a TrieNode corresponding to the last character of the name with the endOfword variable true, this indicates a duplicate input and the **inserted** is set as false for this case otherwise inserted is true.The value is added at the TrieNode corresponding to the last character of the name to be inserted and the endOfword is set to be true.

The **complexity of the above method would be O(length of the word to be inserted as well)** since the method uses an algorithm similar to the search method.

Param Khakhar
2018CS10362

Assignment 4

- ➢ The method **startsWith(String prefix)** again uses the same algorithm used by the method **search** and returns the **TrieNode** corresponding to the **last character of the String prefix.** The **complexity** of the method is **O(length of the string prefix).**

- ➢ The method **printLevel(int level)** uses the **Breadth First Traversal** for printing the elements of the particular level. A **queue** is maintained as an auxiliary data structure. The algorithm terminates when the queue becomes empty. **Only those elements are inserted which are at a level less than the elements of the required level. For the elements of the last level they are firstly inserted in an ArrayList and then they are sorted.** The above algorithm actually results in the traversal of all the characters of the total number of strings as the characters of all the strings at a particular index are inserted in the queue unless the characters are of the required level hence the number of characters processed would be **O(n*k where n is the total number of distinct strings and k is the length of the longest string). Since the characters of the required level are further sorted as well** the number of steps for sorting would be **O(nlogn as there can be at most n different characters which need to be sorted). Therefore the total complexity would be O(nlogn + k*n).**

- ➢ The method **print()** uses the same **Breadth First Traversal** algorithm used in the method **printLevel(),** but in this method elements at each level are printed in the sorted order. Again an ArrayList is used to maintain the characters and then sort them. So the number of elements processed would be **O(n*k where n is the total number of strings and k is the length of the longest string). Since sorting is done at each level the total steps required would be O(k*nlogn) as maximum number of elements at each level would be n. So the overall complexity would be O(k*nlogn).**

- ➢ The method **printTrie(TrieNode trienode)** again uses the **Breadth First Traversal** algorithm used in the method **printLevel()** to print all the words with the same prefix. In this case the values of the trieNode for which the **boolean endOfword** is **true,** are stored in an ArrayList and then it is sorted. The number of steps required for processing would be **O(n*k where n is the total number of different strings and k is the length of the longest string). Assuming comparison of strings to take place in constant time**, the

Param Khakhar
2018CS10362

complexity for sorting would be **O(nlogn). Therefore the overall complexity would be O(n*k + nlogn).**

- The method **delete(String word)** first searches the word in the trie. **If the word is not present i.e. the result of the search is null, then false is returned.** The method **search()** is used for detecting the presence of the word in the trie. If the word is present, the **boolean endOfword is set to be false for the TrieNode corresponding to the last character of the word. Depending on whether the last node has children or not(i.e. The variable count for the last node is zero or not, different procedures are performed.** If the count for the last node is zero then it's reference from the parent node is removed and it's count is decremented by 1. However, if the count is non-zero, then nothing extra needs to be done. Then, a similar procedure is performed for the parent of the TrieNode and depending on the count it's reference from the parent node is deleted or not. This is done until we find a node on the path from the last node to the root whose count is greater than 1. Since effectively traversal is done twice across the height, the complexity for the method delete would be **O(length of the word to be deleted).**

- The methods **CheapSort(ArrayList<Character> arr) and SheapSort(ArrayList<String>)** uses heap to sort ArrayLists of characters and strings respectively and returns a new ArrayList of character and string. The standard heap sort algorithm is used in which the elements are inserted into the heap and then they are extracted. Since, the heap implemented was a max Heap, an auxiliary Stack is used in order to reverse the elements from descending order to ascending order. The complexity is **O(nlogn) where n is the number of elements in the ArrayList.**

2. **Red Black Tree:**
   - The **class RedBlackNode<T extends Comparable, E>** implements the **RBNodeInterface<E>.** The different **private fields** of this class are:
     - **color(String)** represents the **color** of the node which can be **RED or BLACK or EXT(in case of external nodes).**
     - **left (RedBlackNode<T,E>)** is a reference to the left child of the node.
     - **right(RedBlackNode<T,E>)** is a reference to the right child of the node.
     - **parent(RedBlackNode<T,E>)** is a reference to the parent of the node.

- ➢ **value(E)** corresponds to the value stored by this node.
- ➢ **key(T)** corresponds to the key of the node.
- ➢ **values(List <E>)** corresponds to the collection of values in case a particular key has more than 1 value.
- ➢ The fields **color(String), key(T) and value(E)** are passed to the constructor and a new ArrayList **values** is initialized as well.
- ➢ There are **getter and setter** methods for each of the above mentioned parameters are implemented. The method **addValue(E v) adds a value to the ArrayList values** and the method **compareTo(RedBlackNode<T,E> b)** compares the **toString() representations of the current node and the node in the argument.**

- The class **RBTree** has a field **root(RedBlackNode<T,E>)** which is a reference to the root of the tree and it is initialized as null in the constructor. The following methods are implemented:
  - ➢ **search(T key)** searches for the key in the red black tree. The reference curr(RedBlackNode<T,E>) refers to the current node which is compared with the key. The algorithm starts from the root and comparisons are made with the key of curr (Initially curr is assigned to be root). If the key which is to be searched is greater than the root, then curr is assigned to its right child and if the key is lesser than the key of the curr, then curr is assigned to its left child. If the keys are equal then curr is returned. The termination condition of the loop is when curr refers to an external node(i.e. Node with color "EXT") and if the keys of the curr and the key in the argument match. Since in this algorithm, we are traversing the height of the key and therefore the complexity of the above algorithm would be **O(height of the tree).**

  - ➢ The method **insert(T key, E value)** inserts a new Node whose color is defaulted to **"RED"**. Firstly, the location of the new node which needs to be inserted is found, and then the insertion is made at the node by replacing the external nodes by the new nodes. If the node is inserted as a child of a "BLACK" node, than no restructuring is required. If it's inserted as a child of a "RED" node, then there would be a double red problem. The parent of the "RED" node at which insertion is made would be guaranteed to be "BLACK" and if the other child of this "BLACK" node is a "BLACK" colored node, than a rotation would fix the issue and the insertion would terminate and if the other child of the "BLACK" grandparent is “RED” than a recoloring needs to be done between the parent and the grand parent of the node which is inserted. This may result in a double red problem at an above level and therefore we need to check this at a higher level. If we end up recoloring the root while traversing upwards, then we would again recolor the root to be “BLACK” thereby increasing the black height of all the external nodes by 1. In the worst case deletion of the root at the leaf would be done and we’d end up recoloring the root . Doing this would lead to traversal along the height of the root twice and therefore the complexity of the method would be **O(height of the tree).**

➢ The method **Rotation(RedBlackNode<T,E> x,RedBlackNode<T,E> y,RedBlackNode<T,E> z)** takes in nodes which the node y is the parent of node z and node x is the parent of the node. There are 4 different configurations in which they can exist and accordingly pointers are changed so that the new order obtained of the inorder traversal remains consistent with the original order and given the structural modifications which are node. Since finite number of pointers are changed the complexity of the method would be **O(1).**

3.Priority Queue (Max heap implementation):

- The class Student has **private** fields **marks(int)** and **name(String).** Both **marks and name** these are passed to the constructor. There are **getter methods for name as well as marks** while a setter method for **marks.** The **compareTo(Student student)** returns **-1 if the marks of the student** in the argument is greater than that of the **object; 0 if the marks are equal and 1 if the marks of the object are greater than the student in the argument of the method compareTo().** The **toString()** method returns a modified string representing the information of the **student as "Student{name='"+name+"', marks="+marks+"}".**

- The **class Auxiliary<T extends Comparable<T>>** is an auxiliary class used to implement the **FIFO** order of the priority queue. It consists of two fields **first(int)** and **second(T) which are private.** There are **getter and setter** methods for both of these fields. Instead of an object of T type, an object of type Auxiliary<T> is entered in the heap. Where the integer first denotes the order of entry of the object T into the heap. The compareTo() method is changed and the comparison would be first made by comparing the element T and if the comparison of the two T objects results in 0, then the comparison is made on the basis of the order in which they are entered which would be guaranteedly be different for all the objects in the heap.

- The class **MaxHeap** implements the **Priority Queue** using an **ArrayList (ArrayList<T> heap).** The field **last** contains the index at which the new element would have to be inserted. In the constructor, the ArrayList **heap** is initialized and **null** is added at the 0th index which acts as a **sentinel**. There is also an **integer variable trick** which stores the maximum possible size of the at a particular instant. The variable trick is used for imposing the **FIFO** order.The following methods are implemented within the class MaxHeap:

  ➢ The method **size()** returns last - 1 which corresponds to the **number of elements added** at any moment.

  ➢ The method **parent(int i)** returns **[i/2]** which corresponds to the parent node for the node at the index **i of the ArrayList.**

  ➢ The method **left(int i)** returns **2*i,** which corresponds to the left child of the node at the index **i of the ArrayList.**

  ➢ The method **right(int i) returns 2*i + 1,** which corresponds to the right child of the node at the **index i of the ArrayList.**

  ➢ The method **insert(T element)** inserts an Auxiliary<T> object which is constructed and the integer parameter is set equal to the value **trick,** in the **heap(ArrayList<T>).** Firstly, the object(Auxiliary<T>) is inserted at the end of the **i.e. at the index last of the ArrayList. Also the variable trick is calculated which would be the maximum of the current size(last) and (the value trick of the last element + 1).** Doing this ensures that the **structural property of the heap is not violated.** The heap property also needs to be satisfied which is, "**priority of every node should be greater than both its children.**" In order to do this, a check needs to be made at the parent where the new node is inserted. If the parent is smaller, than the newly inserted node and the parent node are **swapped. This is repeated for all the nodes in the path from the newly inserted node and the root whose priority is lesser than the newly inserted node.** In this algorithm, we are traversing up the **height of the heap** and therefore the complexity of this method would be **O(height of the tree)** which would be **O(logn where n is the number of elements inserted) and in this case it would be O(log(last-1)).**

- ➢ The method **swap(ArrayList<T> l,int i,int j)** swaps the elements present at the indexes i, j in the ArrayList.

- ➢ The method **insert2(Auxiliary<T> aux)** inserts an Auxiliary object into the priority queue and the algorithm used is similar to the method insert. In the method insert, a new Auxiliary<T> object is created but for insert2(), the Auxiliary<T> object to be inserted is already provided as an argument. Again the complexity of the above method would be **O(logn) where n is the number of elements inserted.**

- ➢ The method **heapify(int i)** is an auxiliary method used for the restructuring of the heap. It restructures the heap in such that the subheap at index i again becomes a heap if it is not. The algorithm assumes that both the left and right subheaps of the element at index i are heaps. If the element(Auxiliary<T>) at index i is larger than both its children, then we're done. If not then the element swaps with the child with higher priority. Here priority is firstly decided by comparing the two Auxiliary<T> objects and the higher priority Auxiliary<T> object is swapped with the element at index i. The procedure is recursively repeated for the sub heap in which the swap is made. In the worst case, we would have to traverse the entire height of the heap and therefore the complexity of the heapify would be **O(height of the heap, which would be logn where n denotes the elements inserted in the heap).**

- ➢ The method **search(String key) searches for an element with the required key in the ArrayList. Linear search is used to implement the above requirement and therefore the complexity is O(n where n denotes the number of elements inserted).**

4. Project Management

- The class **Job** is used to implement the job object. It contains the following fields:
  - ➢ **user(String),** which is the name of the user to which the job is assigned(passed to the constructor)
  - ➢ **name(String),** which is the name of the job (passed to the constructor)

- ➢ **runTime(int),** which is the running time of the job (passed to the constructor)
- ➢ **project(String),** which is the name of the project to which the job is assigned (passed to the constructor)
- ➢ **Status(String),** which is the current status of the job(initialized as "NOT FINISHED" in the constructor).
- ➢ **Priority(int),** which is the priority of the job.
- ➢ **completedTime(int),** which is the time of completion of the job.
- ➢ There are **getter and setter** methods for each of the fields. The **compareTo() method is implemeted by comparing the respective priorities of the job which essentially the priority of the projects to which the jobs belong. Accordingly the values 1,-1 and 0 are returned.**

- The class **Project** is used for the creation of the project object. It contains three fields **name(String), priority(int) and budget(int)** each of which are passed to the constructor. There are getter and setter methods for these fields.

- The file **Scheduler_Driver.java** contains methods which are implemented in order to achieve the desired functionality. Auxiliary fields defined are as follows:
  - ➢ **trie(Trie<Project),** which stores all the projects in a trie.
  - ➢ **jobs(MaxHeap<Job>),** which stores all the jobs which are created by query JOB
  - ➢ **jobs2(RBTree<String,Job>),** which stores all the jobs in a red black tree and is used for the query QUERY to retrieve the status of a particular job.
  - ➢ **users(RBTree<String,User>) ,** which stores all the users in a red black tree and is used to check the existence of a particular user while creating a new job.
  - ➢ **globalTime(int),** which acts as a time storing counter and it is incremented every time a job gets executed.
  - ➢ **completed(ArrayList<Job>),** which is an ArrayList storing the jobs which have been executed i.e. whose status is FINISHED.
  - ➢ **notCompleted(ArrayList<Job>),** which is an ArrayList storing jobs whose status is REQUESTED.
  - ➢ **notReady(LinkedList<Job>),** which stores the jobs which haven't been executed due to the insufficient budget of the project to which they belong. Jobs from this linked list are

again inserted in the priority queue jobs when the budget of a particular project is increased.

- ➢ **budgPro(String),** which contains the name of the project whose budget is increased.

The following methods are overridden:

- ➢ The method **run_to_completion()** is called when there aren't any queries to be processed. In this method all the jobs currently present in the heap are checked whether they can be executed or not. If they get executed, they are moved to the ArrayList completed and if not they are moved to the linked list. After removing all the jobs from the heap, the remaining jobs are moved to the ArrayList<Job> notCompleted.

- ➢ The method **handle_project()** is responsible for creation a new project object and inserting that into the Trie trie.

- ➢ The method **handle_job()** is responsible for the creation of a new job for the specified inputs. The membership of the project and users are checked initially and a job is only created if there exists a user and the project to which the job is assigned. The priority of the job is assigned by fetching the priority while searching the project. The job is then inserted into the MaxHeap jobs and RBTree jobs2.

- ➢ The method **handle_user()** creates a new user object and inserts the user object into the red black tree user.

- ➢ The method **handle_query()** processes the query QUERY by searching the RBTree for the queried job and prints it's status if the job is present otherwise prints "NO SUCH JOB".

- ➢ The method **handle_add()** processes the query ADD by searching the project in the trie and updating the budget if the project exists followed by promotion of jobs of the project in the LinkedList NotReady to the heap jobs. If the project does not exist, than "No such project exists." + project_name is printed.

Assignment 4

- ➢ The method **print_stats()** is responsible for printing the statistics. Iterations are performed over the ArrayLists completed and notCompleted and the respective details are printed accordingly.

- ➢ The method **schedule()** is executed when an end of line is detected in the queries. The job of maximum priority is removed from the heap jobs and it’s project is checked for the available budget. If enough budget is available, then the job is executued i.e. the global time is increased along with decrement of the budget of the budget of the project. If the budget is not sufficient, than the job is added to the linked list NotReady. The status of the job is updated as well and also other relevant details need to be printed.

Param Khakhar
2018CS10362