COL106
Assignment 5 README

ROHIT AGARWAL
2018EE10494

In this assignment, we implemented three types of data structures - Red-Black Trees, Tries and Priority Queue using MaxHeap. Then we made a project management system using all three of them.

TRIES-
TrieNode-
It contains a boolean value isEnd indicating this node corresponds to the end of a word.
It also has an array of nodes where it stores its children.
It maintains a count of the number of children in an int value called children.
It also stores a value of parametric type T.

Person-
It has phone no and name stored in it. Also has a toString method which returns all the info as a string in the required format for printing a person.

insert() -
It takes key and value to be added then looks if the child corresponding to the first character of the key in the current node's children array. The current node is initially set to the root node. Now it is set as the child. If the child was non-existent then it adds a new node at that index and if it exists, then sets the current node as the child. This repeats (length-of-key) times. In the end, if the final node already existed and isEnd is already marked true, then it means that the insert query was duplicate one, so false is returned. Else isEnd is marked as true and True is returned.
It also updates the children int of parent if need.
Time complexity is O(key length) as it goes down only key-length levels.

search()-
It takes key as value and then goes through the trie in the same manner as described in insert(). Upon reaching a null node, it returns null as it means that the key is not present; otherwise, it keeps on going forward. At the end (after length-of-key steps) if it reaches a node that is not marked as isEnd, then it means that the key is not present and it returns null. If isEnd is true, then it returns this node. Time complexity is O(key length) as it goes down only key-length levels.

delete()-
It searches for the key given in the same way as above. If not found, it returns false.

Along the way, if it encounters a node that has only one child and is not the end of a word, it stores that node by storing its parent and the index where it is present in the parent's child array so that parent.children[index] = null will trim the trie branch having that node as root. Let us call the variables where these are stored as holder variables. This node remains stored until the function encounters a node which has more than one child or is the end of a word. Upon reaching such a node, it resets the holder variables to hold dummy value. Then again sets them if it finds a node with only one child and isEnd = false. If it reaches a leaf, then it stores it.
Time complexity is O(key length) as it goes down only key-length levels.

In the end, if the key is found, if the end is a leaf then it executes parent.children[index] = null if holder variables do not hold dummy values to cut the branch of trie which was exclusive of this key only as this no longer needed after key deletion. It also updates the children int of node from which branch is cut, if it is.

startswith()-
This is same as search, but it does not do a isEnd check. If it reaches the final node without reaching a null one, it returns that node, otherwise null node. Time complexity is O(input length) as it goes down only input-length levels.

printTrie()-
It first checks whether the node passed to it has isEnd true or not. If true, then prints the toString of the node.
Then it goes ahead to iterate through its children array. It again calls the printTrie function for each non-null child.
This is results in a DFS like traversal of the subtrie with the given node as its head. Since it is DFS, it automatically ensures that the printing will be in lexicographical order. So there is no need to first store results in an array for eg. and then sort and print it.
Time complexity is O(n) where n is the number of nodes in the subtrie with given node as root since it is a DFS.

printlevel()-
A ArrayList of characters is used to first store the results. It is cleared at the start of the execution of the function.
A new recursive function printlevel_() was made and is called for this which takes an int for level and a trienode as parameters. Its base case is when level == 1, add all the characters whose corresponding trie node child is not null, in the global ArrayList.
In other cases it calls the printlevel_ function for each non null child of the trienode with level as level-1.
The initial call to this has root as the node and the level to be printed as the parameters.
This is a like a DFS traversal till the level required. And at the final level, it stores the characters in the ArrayList. This list is then sorted and printed. The array sorting and printing is wrapped in a new function called sortAndPrint which takes list and level as parameters.
Time complexity = O(n) as it is a DFS till a level, where n is number of nodes till that level.

print()-
A new recursive print_() is made and called which does a BFS traversal of trie and store all the characters found in a level in a list and passes it along with the level to the sortAndPrint function.
This function takes int level and a node list as parameters and then for each element of node list, store its children node in a new node list and the corresponding characters in a character list. Then passes the character list to sortAndprint function along with level. If the new node list is not empty, it again calls the print_ on this list with level = level+1.
The initial call made to it in print() has a list with only element as root and 1 as the level as parameters.
Time complexity = O(log n) where n is the number of nodes in the trie as it is a BFS.

PRIORITY QUEUE-
It maintains a list which stores all the entries, instead of storing left and right child in nodes. This works because a heap is always a left complete tree.
To get left child - 2*index+1
To get right child - 2*index+2
To get parent - (index - 1) / 2

pqNode-
This is what is actually stored in our priority queue. It contains to value to stored and an int called sno which basically stores the number at which this node was added. Each node contains a unique sno.
This implements comparable, and the compareTo method returns the compareTo of the values stored in the compared nodes, but it the priority of the value stored is same, then the node with smaller sno is given higher priority and a corresponding value is returned. This maintains FIFO if the value priority is the same. All the comparisons are done in insert or extract max functions are done by calling this compareTo.

insert()-
Adds the element to end of the list. Then it keeps on comparing it to its parent and swapping places with it until it reaches the top or gets a parent that has higher priority than it.
A global int named sno is maintained which keep a count of how many elements have been added till now. When adding an element this sno is passed to the new pqNode along with the value and then this value is incremented for the next insertion. The worst case occurs when the element goes all the way to top, the time complexity = O(height). Since maxheap is a complete tree h = O(log n), thus insertion = O(log n).

Extractmax()-
Swaps the first and last element and the deletes the new last element and stores it to return it at the end. Then it comes to the first element and keeps on comparing it with its children, if they exist, and swaps it with the higher priority child until it reaches a position where it has no child or

both children have lower priorities. Here the worst case it the top element reaching all the way to the bottom, so here too, time complexity = O(log n).

makeHeap()-
This restructures the heap from the list. It starts from n/2 -1 th element and heapifies down each element from n/2 -1 to 0 th index. This reconstructs the heap in O(n).

RB TREE
RedBlack Node-
Stores key and a list of values, a boolean b to store colour (true means black), and references to left child, right child and parent. In the constructor, if key passed is not null, it sets key and adds an entry to values list as given in parameters, but also sets its left and right child as new Node having null key and value and then set their parents as itself.

search()-
This is a normal BST search where it compared value to the current node and goes left or right accordingly until it reaches null or finds the key and returns the found node or null correspondingly. The time complexity will be O(height) and as Red-Black tree has height = O(log n), time complexity = O(log n)

insert()-
It first does a normal BST insert by finding the location where the value is to be added in the same way as search works. If the key is already existing, it simply adds the given value to the values list of the node. If the key is not found and a node with key = null is reached, that node is then reassigned as a new Node and its colour is set as red (i.e. b = false).
In the first case, the insertion ends, but in the second case, we need to balance the tree. Balancing can be done by restructuring or recolouring. Recolouring leads to recursive calls to balance function, while restructuring leads to rotation/s. Balance function determines what case it is and then calls the appropriate functions.
BST insertion is O(height), which here is O(log n). Balancing worst case occurs when we need to recolour all the way to top, thus balancing is O(log n). Thus the overall complexity is O(log n)

PROJECT MANAGEMENT
It stores all projects in a trie with name as the key.
It stores all users in a trie with name as the key.
Jobs ready to be executed are stored in a MaxHeap.
The jobs which were not executed due to an unsufficient budget are stored in a RB tree with project name as the key.
All the jobs which are completed are stored in an ArrayList.
All the created jobs are stored in a trie.
All users are stored in an arraylist.
All projects are stored in an arraylist.

A global time and a serial numbers maintaing number of jobs, projects, users created is maintained.

JOB-
It contains name, project, user, runtime, completion time, arrival time, boolean indicating if job completed or not and an int value sno which stores at the number at which that job was created. The compareTo of job return value according to priority of projects, but if they are same, if same then according to sno of jobs. This is for FIFO in case of same project priority.

PROJECT-
Stores name, priority and budget, sno. It also stores an arraylists of jobs it has.

USER-
Stores name, comsumption, latest completion time, sno and compareTo return according to consumption, if same then latest completion time, then sno. It also stores an arraylists of jobs it has.

handle_project()-
Creates a new Project and adds it to the projects trie. Time complexity - O(project-name-length)

handle_job()-
Searches projects and users trie to get the Project and User objects associated with the job. If not found, then aborts job creation and prints the error. Otherwise creates a new job, update its arrival time and adds it to the jobs MaxHeap and also adds to the jobs trie. Then adds it to its user's job list and project's jobs list. Adds job to all jobs list. Time complexity - O(log n + project-name-length + user-name-length + job-name-length), where n is the number of jobs already in heap.

handle_user()-
Creates a new user and adds it to users trie and all users list. Time complexity - O(user-name-length)

handle_query()-
Finds the job is trie. If not found then print not found, else print the status-completed or not. Time complexity - O(job-name-length).

handle_add()-
Searches projects in trie, if not found then aborts and prints error. Else adds given value to the budget. Then searches the project in the RB tree holding uncomplete jobs, and adds any completed jobs present to the jobs MaxHeap. Since the jobs hold their creation serial no, FIFO will not break. Time complexity - O(project-name-length + project-name-length*log(p) + j*log(j)). Where p is total number of projects and j is the total numbers of jobs. Project-name-length

comes from finding project in project heap, then project-name-length*log(p) comes from finding project in rb tree, then j*log(j) comes from adding j jobs to jobs Heap.

execute_a_job()-
Tries to execute a job until a job with sufficient budget is executed or no jobs are left. If job is completed then updates the completion time, global time, status, project budget, users consumption and latest completion time and adds to it to completed jobs arraylist. If job is not completed then add it to jobs RBtree. It also prints all the corresponding statements required by the assignment specs. Time Complexity - O(j*log(j) + j*log(p)) where j is the total number of jobs and p is the total number of projects. j*log(j) comes from getting j jobs from the job heap till sufficient job budget is found. Executing a job with suffient budget is constant time operation. j*log(p) comes from adding jobs with an insufficient budget to rb tree.

handle_empty_line()-
Calls schedule() and then prints "Execution cycle completed". Time Complexity - inherited from schedule().

schedule()-
Prints the remaining number of jobs in the heap and then calls execute_a_job(). Time Complexity - inherited from execute_a_job().

run_to_completion()-
Calls schedule() and then prints “System execution completed” till there are still jobs remaining in the jobs MaxHeap. Complexity is same as schedule() as that is according to worst-case and in that, all jobs will be extracted from Heap.

print_stats()-
Prints the finished jobs (by calling toString of Jobs) from the ArrayList. This will print completed jobs in order of completion. Then calls getUnfinishedJobs() to get a maxHeap of uncompleted jobs then extracts and prints all the remaining jobs. Time complexity = O(j + j*log(j) + j) = O(j*log(j)). j comes from printing all completed jobs from arraylist, then first j*log(j) is inherited from getUnfinishedJobs and the last j comes from extracting j jobs from returned ArrayList and printing them.

getUnfinishedJobs()-
Does BFS of rb tree having uncompleted jobs and add each to an ArrayList after required sorting i.e. first project priority then project fifo then job fifo. Then return it.
Time Complexity = O(j*log(j)), where j is the number of uncompleted jobs. Where j comes from traversing to each job in rb tree and log(j) comes from adding each job to a Heap. Complexity of sort is O(j*logj) too.

Timed_handle_user, timed_handle_job, timed_handle_project, timed_run_to_completion -
These are same as their untimed counterparts but do not print anything.

NEW_PROJECT-
Find project from project trie, search its jobs, they will already be in arrival time order. To search first find the index of job having arrival time t1 or just bigger using binary search. Then iterating the list till arrival time becomes > t2 and adding its report to an arraylist. Return that arraylist. Complexity = O(project-name-length + log j + j), where j is the number of jobs in that project

NEW_USER-
Find user from user trie, search its jobs, they will already be in arrival time order. To search first find the index of job having arrival time t1 or just bigger using binary search. Then iterating the list till arrival time becomes > t2 and adding its report to an arraylist. Return that arraylist. Complexity = O(user-name-length + log j + j), where j is the number of jobs of that user.

NEW_PROJECTUSER-
Find project from project trie, search its jobs, they will already be in arrival time order. To search first find the index of job having arrival time t1 or just bigger using binary search. Then iterating the list till arrival time becomes > t2 and adding its report to an arraylist if user matches. Return that arraylist after doing required sorting. Complexity = O(project-name-length + log j + j*user_name_length), where j is the number of jobs in that project.

NEW_PRIORITY-
Sort list of all jobs. Start iterating through the list and add each unfinished job to a new arraylist till we find a job with priority less that the given one. Return the new arraylist. Complexity - O(j*log(j) + j) = O(j*logj) where j is the number of jobs.

TIMED_TOP_CONSUMER
Sort the list of users, Make and add UserRepoerts of top <top> users to an arraylist and return it. Complexity - O(u*log(u) + top) where u is the number of users.

TIMED_FLUSH-
Make a new Job MaxHeap.
Transverse the arraylist of the jobs trie, when we find a job that has enough budget and is long waiting, swap it with the last element and remove from here and add to list of the new MaxHeap. (O(j)) Now call makeheap() of new Heap (O(j)). Now repeatly call extract max on new heap to get a job, if enough budget is there then execute it else add it to end of original heap's list. Do this till new heap is empty. (O(j*logj)). Now call makeheap() on original heap. (O(j)) Complexity - O(j+j+j*log(j)+ j) = O(j*logj) where j is the number of jobs.