README

Assignment 4:

I have implemented following important classes:

1. MaxHeap<T> : It has only following functions-

- **insert (T element):** inserts a new element at the leftmost end of tree, that is end of the arraylist, and calls bubbleUp function. Every element is a node **(PQNode)** which stores the element and list of elements if there are more than one elements having same priority.
- **bubbleUp (int currentIndex):** It recursively maintains the maxHeap. If the parent is smaller than the currentIndex's element, it swaps them.
- **extractMax():** It removes the topmost element from heap (that is, maximum), shifts the last element of arrayList to the top and calls bubbleDown. If there is a element whose node has list of elements having same priority. The oldest element is removed and is replaced by second oldest one in the list.
- **BubbleDown(int parent**): It also recursively maintains maxHeap. If the child of node is larger or equal to it, it swaps them.
- **size():** it returns the size of maxHeap at any instant.
- **Search (String str):** it searches for the key in maxHeap and returns it.

2. RedBlackTreeNode<T, E>: Here T's object serves as key and E's object is the object to be stored in RB Tree. It has pointers to the leftChild, rightChild, and parent of the currentNode. It maintains a list of objects of E since same key may be used more than once for more than one objects. It also has markers for marking the node red or black.

3. RBTree: It has only following functions-

- **insert (T key, E value):** inserts a new key in the same fashion as BST, and stores the value in the node. If the key already exists, it simply appends the new Object to the existing list corresponding to that key. It then calls FixUpTree(node), where node is the newly inserted node.
- **fixUpTree (RedBlackTree<T, E> Node):**

 It first checks whether uncle is red. If so, then both parent and uncle are set to black and the grandparent is set red, then fixUpTree is recursively called on grandparent until it reaches the root or there is no imbalance.

 Else, uncle is black or null. In both cases, I have identified the type of imbalance and then called following separate function which rotate and recolour the tree to fix the imbalance.

  - ✓ leftLeftFixup (TrieNode<T,E>)
  - ✓ rightRightFixup (TrieNode<T,E>)
  - ✓ leftRightFixup (TrieNode<T,E>)
  - ✓ rightLeftFixup (TrieNode<T,E>)

 In all these fix-Ups, I have used standard rotation and recolouring of nodes required. One thing that troubled me, and I found it worth mentioning is that if grandparent is the root, and it needs to be replaced by some node during rotation, the new node must be set as the root. Oherwise, the search will fail.

- **search (T key):** Simple and standard BST search.

4. TrieNode<T>: It is basically an array which can store characters of ASCII values 32-126 in different indexes. Each time a character is stored in an index, a new array is generated for storing the next character and so on. *Boolean isLeaf* marks the end of the word, and the object is stored at this end.

5. Trie: It has following functions:

- **insert (String word, Object value):** It inserts on characters per trienode, that is, array. Next character of each character is stored in subarray at next level. . *Boolean isLeaf* marks the end of the word, and the object is stored at this end. It returns false if the word is already present in the Trie.
- **search (String word):** It follows the path of word same as insert. If at any level, it encounters a null child Array, or at the end of word, *isLeaf* is false, it returns null. If *isLeaf* is true, it returns the node of the leaf.
- **startsWith (String prefix):** Almost similar to search except it doesn't care if at the end, the Boolean *isLeaf* of the node is false or true. It only returns the node of the last character of the prefix if it is not null.
- **printTrie (Trienode trienode):** It goes to each childNode starting from the given trienode. At each occurrence of *isLeaf==true* it prints the object stored at the node. Hence it prints all the objects which have common prefix.
- **delete (String word):** It traverses to the end of the word. If *isLeaf* is false, it means that word is not present in trie and returns false. If *isLeaf* is true, it makes it off and checks if the node has any children. If it has no children, it simply deletes the node and checks same for its parent recusrively. Otherwise, it traverses back without deleting anything
- **printLevel (int level):** It traverses the array of level and if the currentLevel of traversing matches with the levelToBePrinted, it prints all the characters of the level in lexicographical order for which I have used an arrayList and *sort()* function.
- **print():** It prints the trie level-wise, same as in printLevel.

6. Scheduler_Driver: It combines all the three data structures.

It has Following functions:

- **handle_project(String[] cmd):** It simply allocates the name, priority and budget of a project to a project object and stores projects in both trie and RBTree.
- **handle_job (String[] cmd):** It searches for the project and user related to the job and then assigns it, along with name and runtime to Job object which are initially stored in maxHeap.

  If it can't find the associated project or user, it returns an error.
- **handle_user (String name):** It allocates the name of user to the User object and stores this object in a RBTree of user (so that job may search it during initialization)
- **handle_query (String key):** It first searches for the key in the maxHeap, then in the list of nonCompletedJobs and finally in the list of completed jobs. If it is found in first two structures, it means that the job is *NOT COMPLETED*. If it is found in list of CompltedJobs, it means that the job has been *COMPLETED.* If it is found in none, it means that *JOB DOES NOT EXIST.*
- **handle_empty_line():** I have extracted the job from Priority queue of jobs and checked if the runtime of job is more than budget. If yes, it gets added to the *listOfNonCompletedJobs*, otherwise it gets added to the *listOfCompletedJobs,* it's status is changed to completed.
- **handle_add(String[] cmd):** Check the elements of *listOfCompletedJobs* and if the project name is same as project's name to which budget is added, remove it from list and add it to Priority queue of jobs. Also, temporarily store existing elements of PQ in a list so that insertion order remains same.
- **print_stats():** Print the elements of *listOfNonCompletedJobs and listOfCompletedJobs* with the running and endtime of jobs.
- **run_to_completion:** Executes all jobs in the PQ until they get added to *listOfCompletedJobs* or to *listOfNonCompletedJobs* due to budget issue.

Time Complexities:

1. Red Black Tree: Both insert and search functions are O(log N), N = number of nodes. This is because the height of a red black tree is O(log N). Since insert and search takes O(h) time in worst case, it is O(log N) operations.

2. Trie: Insert, search and delete are O(word length) functions because they go to next levels up to the length word. The print function may be $O(n_2)$ in worst- case since we may have to traverse through complete arrays in worst-case. Here, n= maximum size of array or maximum number of unique characters in trie.

3. MaxHeap: search is always O(log N) since the height of heap is always O(log N) since it is a complete tree always. Insert and delete may be O(1) in best-case when there is no need of bubbleUp/ bubbleDown. However, in worst case, we may want to bubbleUp/ bubbleDown through entire height if tree, and hence worst-case complexities are O(log N) where N is the number of nodes in maxHeap.