README

Name : Aakash Garg Entry No: 2018MT60776

In this assignment , we have implemented the 3 advanced data structures namely Trie, RedBlackTree and maxheap. The data structures are **generic** and hence can be used for any application. These data structures process queries in O(log(n)) time complexity and hence are used for optimization at large scales containing **large datasets**.

TRIE

This data structure is a search tree which has optimal time complexities of length of key which is searched. In this data structure we have used the condition that the number of characters are fixed and hence we can create an array of fixed size of characters .In this we have implemented the functions :

Insert :-

In this function , we have inserted the element by picking each character and then making arrays at different levels. The time depends on the number of elements in the array of each key at different levels and also on the length of the key.

Time Complexity - **O(key.length)**

StartsWith :-

In this function , the node is returned on which the prefix end.The time depends on the number of elements in an array of each key at different levels and also on the length of the key.

Time Complexity - **O(key.length)**

Search :-

In this function , the node is returned on which the total key end if there is a key present there. The presence of the key is checked by a boolean variable which is true when there is an end of word.The time depends on the number of elements in the array of each key at different levels.

Time Complexity - **O(key.length)**

Delete :-

In this function , the node to be deleted is searched and if found it is then deleted. There are 3 cases namely- it is a substring of another key or it contains a substring or neither. All these are handled properly. The time will be the same as that of search.

Time Complexity - **O(key.length)**

PrintLevel :-

In this function , we have printed all the elements in the lexicographic order of the level asked. There is a recursive code to print the level in any order and then bubblesorted it to have in lexicographic order.

Time Complexity - **$O(n^2)$** where **n** is no. of key. It is because the number of elements at any level depends upon the number of keys and bubblesort is of **n^2** order.

Print :-

In this function , we have called the print level for the number of levels or the height of the trie which is calculated with the help of a recursive method.The time complexity will be equal to that of (Printlevel complexity) * key.length.

Time Complexity - **$O(key.length * n^2)$**

The trie is implemented with the help of the class TrieNode and all the functions are correctly implementing the code and showing correct output for the given input file.We have overridden the ToString and ToCompare functions to show the output.

REDBLACKTREE

This is a balanced binary tree in which red and black nodes are present with some restrictions of black height and double red due to which some restructuring is required which results in the balancing of the tree.The height of a red black tree is of the order of log(n) and hence all the time complexities are **O(log(n))** as at max we have to traverse upto the height of the tree.

Insert:-

In this function, we have added in the Red Black Tree as follows:-

Satisfy the redblack property by restructuring and recolouring which are O(log(n)) and O(1) time complexity functions and hence the

Time Complexity - O(log(n))

Worst Case Complexity - O(n)

Search:-

In this function , simple BST search is done which has a O(log(n)) complexity.

Time Complexity - O(log(n)) (n is the number of nodes in the tree)

Worst Case Complexity - O(n) (when all objects have the same key so all objects will get in one arraylist of size n)

All the functions of RedBlackTree are correctly implementing the code and showing correct output on the given testcase.The tree is balanced and the height is also of the order **O(log(n))**. We have updated the ToString and ToCompare functions to show the output and use the RedBlackNode as the basic building unit.

PRIORITY QUEUE USING MAX HEAP

The priority queue data structure is implemented with the help of a Max heap using the array with left child, right child and parent at specific locations defined by the max heap.The functions used are:-

Insert :-

In this function, the element is appended at the end of the array and then the location of this element is found by comparing parent and replacing until its parent priority is higher than its child.

Time Complexity - O(log(n)) (n is the number of elements)

ExtractMax :-

In this function, the first element of the array is extracted since it is of highest priority and the last element of array is placed at 0 index and then the replacing of elements takes place till the heap is made correct.

Time Complexity - O(log(n)) (n is the number of elements)

The elements are compared first on the comparison of the compareTo function of T objects and then for the same priority objects , a variable which stores the time of insert is compared and hence priority is decided. **This ensures the FIFO order of heap.** This is done by making a new class **Pair** which stores the **T object and time of adding**.The compareTo function of Pair is overridden which gives the **FIFO** output.

The heap is correctly implementing all the code and showing correct output on the given file. The FIFO is also implemented correctly.

All the data structures are completely **generic and no type casting is used anywhere**.

INTERESTING FINDINGS

1. Two search efficient data structures are implemented whose complexities are optimal and are most efficient at different times. For large no of input , Trie is efficient while in other cases Tree is efficient.
2. A maxHeap can be created which gives the highest priority element and also the FIFO order for the same priority in O(log(n)) time complexity.
3. The RedBlacktree is balanced by just satisfying the basic properties of the red black tree such as black height and double red problem.