**Analysis of Sorting Algorithms**

**Algorithms Implemented:**

1. Insertion Sort
2. Merge Sort
3. Heap Sort
4. Quick Sort

**Code Structure and Execution:**

All source code is written in python. Every sorting algorithm has a standalone executable file (eg: insertion\_sort.py) which takes an input file and outputs the sorted instances into an output file (eg: insertion\_sort\_output.txt).

There is also a program file “input\_generator.py” which is used for generating the random instances as mentioned in the writeup. Every time you execute this will generate new instances in to input file (“input\_sort.txt”).

**Execution steps:**

**Input file generation:**

python3 input\_generator.py

**Sort Algorithms:**

Every sort algorithm prompts to give an input file to be sorted.

1. python3 insertion\_sort.py

input\_sort.txt

1. python3 merge\_sort.py

input\_sort.txt

1. python3 heap\_sort.py

input\_sort.txt

1. python3 quick\_sort.py

input\_sort.txt

**Data Structures and Implementation:**

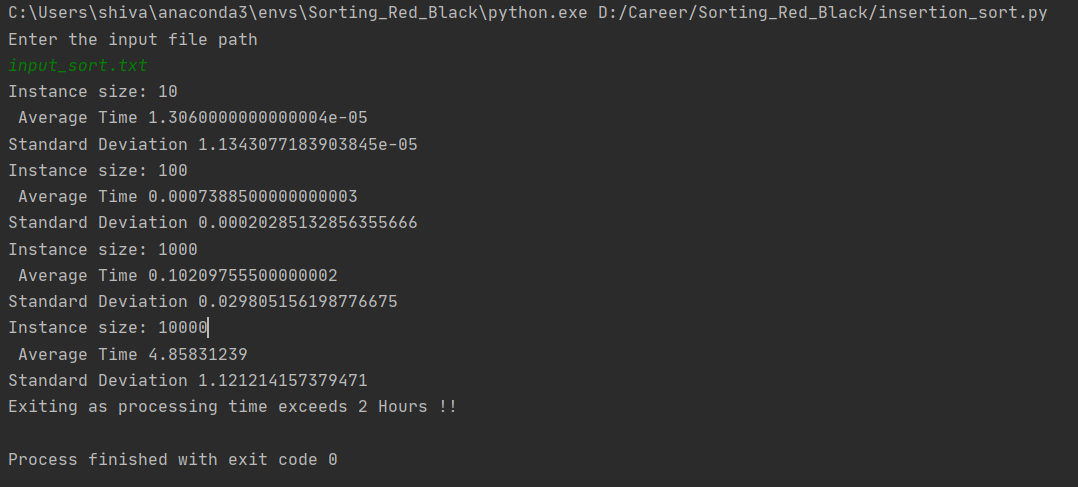
All the sorting algorithms are implemented based on the pseudocode mentioned in the textbook. Since there is no array data structure in python, implementation uses a list data structure to only store data while reading instances from input file. No inbuilt sort methods of “list” are used to implement any sorting algorithm.

* All the sort functions to in place sorting, i.e., when a sort call is made on list “L” the sort is performed on “L” itself. No new lists are created to perform sort operations, therefore no need to return anything from sort function since sort is performed on the same instance itself.
* Since we are analyzing the time for sorting on each instance and have a time clause not to exceed 2hours of execution time writing into the file is done after all sort or some sorts(timeout) are completed. For this purpose, the sorted instances are stored in a list and then called to write to a file.

For Analysis of mean and standard deviation of time on different instances of same sizes I used a python3 inbuilt library “timeit” to get the start and end times of sorting each instances. Also used another timer object to check the overall time on while sorting algorithm, this is to check if a takes more than 2 hours and breaks out of the sorting and exits safely.

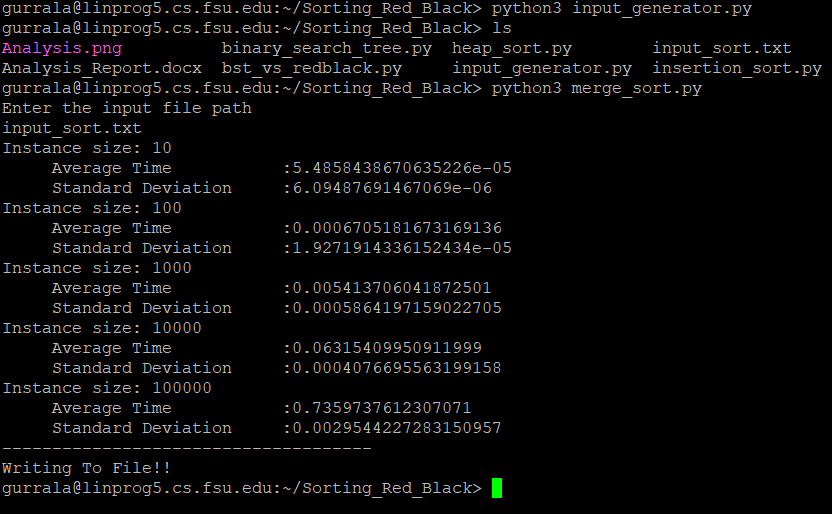
Sample output :

**Insertion Sort:**



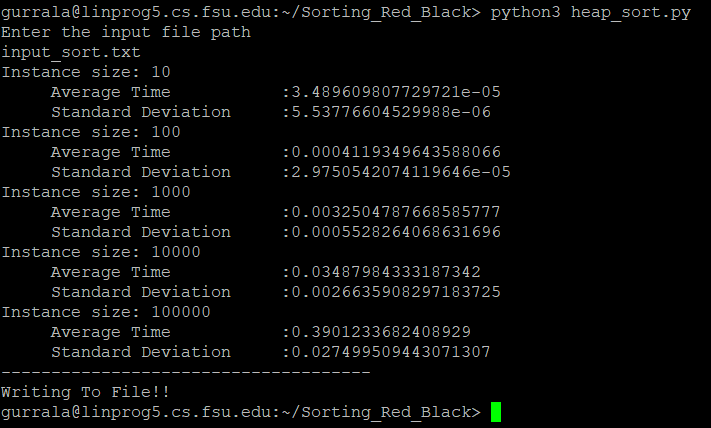
As the insertion sort time complexity is more average is θ(n2) it took more than 2hours to do the sorting and got exited before completing the 105 instances. We could see how the Average time changed upon increase in the instance size.

**Merge Sort:**



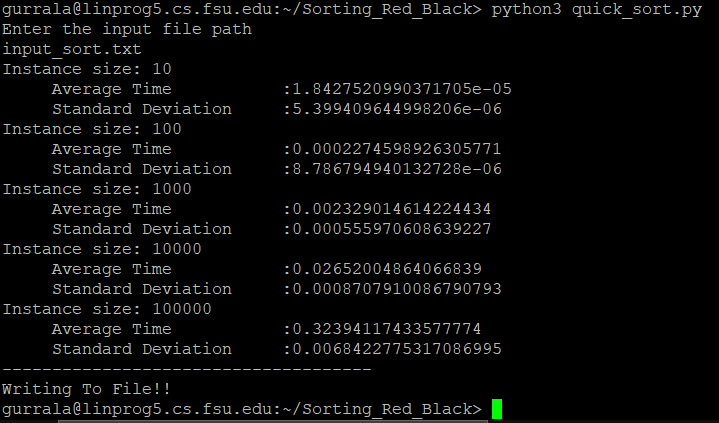
Since merge sort uses divide and conquer technique and time complexity as θ ( n log(n) ), though the input instance sizes increased to 105 it took only 0.7 secs on an average to complete the sorting.

**Heap Sort:**



Even though both merge sort and heap sort have same time complexity we could see there is a difference in the execution times, as the constant factors of both the algorithms are different and we should also consider the no of memory instructions each algorithm takes.

**Quick Sort:**



As we know even though worst case of quick sort is O(n2) the expected time is O(n log(n) ) and Quick sort gave better execution times than all other sorts analyzed.

Analysis of BST and Red Black Tree

Binary Search Tree:

Code Structure and Execution:

Created a class “BinarySearchTree” that has a member variables required to be a node in according to a binary search tree.

Member functions implemented:

**Insert() :** Takes a BST node as an input and performs insertion on the BST object.

Example :

bst = BinarySearchTree()  
node = BinarySearchTree()  
node.key = 8

bst.insert(node) - This inserts “node” in “bst” object.

**Search() :**Takes a value/key as input and returns the node whose value match with the given input.

Example:

bst.search(bst.root, 5) //returns the node if there is an element match or returns a None type object.

**inorder\_walk():** Takes root object and traverses tree in inorder fashion and displays the elements.

Example:

bst.inorder\_walk(bst.root)

**height():** Takes root object/ any node and gives height respectively.

Example:

bst.height(bst.root)

**delete():** Takes a node in tree as input and deletes.

Example:

bst.delete(bst.root.left.left) // Deletes the left->left node from root if present.

There are various other functions like minimum(), maximum(), successor(), iterative\_search() and transplant implemented please look at the commented code in binary\_search\_tree.py for how to call those.

Red Black Tree:

For Red Black tree I have created a node class and a Red Black Tree class, a node class has variables left, right, parent, key, color. An enumerator color class is written to store Red and Black colors.

RedBlack class basically has two types of member variables :

1. Tree.NIL // NIL node for representing the edge/Null nodes.
2. Tree.root // Root node

Initially when a new Red Black tree instance is created a Nil node is assigned to root. Upon adding a new node, the algorithm kicks in and works as a regular BST insertion, once insertion is done it calls insert\_fixup() method with tree root and node. Every new node is added with red color.

Insert\_fixup() takes care of the Red Black tree properties by making appropriate rotations and changes in the colors of the node. Finally changes the root color to black, if in case the color is modified while doing the operations.

Example:

rb = RedBlack() // red black tree object creation

rb.insert(rb, 5) // inserting a node with value as 5

**height():**

outputs the height of the tree, this does not include the leaf/Null nodes.

Example :

print(rb.height(rb.root))

**Deletion():**

Implemented deletion for Red black tree, takes two parameters the RedBlack object and the node that to be deleted.

Once deletion is done we call the delete\_fixup() that takes care of the Red Black tree properties.

Example:

rb.delete(rb, rb.root.left) // deletes the node at root->left if present.

A sample deletion code is written in the “bst\_vs\_redblack.py”.

Also implemented inorder traversal, this just prints the value a node and its color.

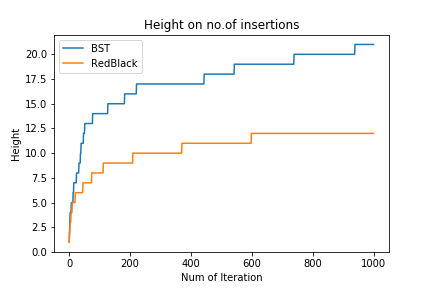
**Execution :**

There is an utility program “bst\_vs\_redblack.py” this imports both BST and Red\_Black tree and inserts 1000 values randomly picked between 1 to 1000. And each time an insert is made the heights of both the trees is stored in lists. These lists are used for analyzing the change of height over the insertion by plotting a graph.

To run:

python3 bst\_vs\_redblack.py

**Height Plot for BST vs Red Black Tree:**



For 1000 iterations every time a random value between 1 to 1000 is picked and added to both the trees. Once insertion is done we call the height functions of each of them and store the height in lists and finally these are used to plot the graph using matplot lib.

I commented out the matplot lib part since lingprog does not have a matplot library installed.

Outcomes:

As we could see Red Black has lesser height on no of insertions when compared to Binary Search tree. Lesser height leads to less memory to store. Also, less execution time to search an element.