How the code is organized into different files. Instructions for how to compile your code.

Ans. : The codebase has two folders which are “sort” and “tree”. The sort folder contains all the sorting scripts and tree folder contains binary tree and red black tree implementations.

sort

-Insertion_sort.py (Insertion Sort)

-merge_sort.py (Merge Sort)

-heap_sort.py (Heap Sort)

-quick_sort.py (Quick Sort)

tree

-bst.py (Binary Search Tree)[Both insertion and deletion implemented]

-rb_tree.py (Red Black Tree)[Both insertion and deletion implemented]

- input_tree (input file for running the bst.py and rb_tree.py)

Required packages:

Python 2.7.5

numpy

Instructions to compile:

```
python <sort/tree file> <input file>
```

How data is represented internally. For sorting algorithms, discuss if each implementation of sorts in-place or not. For the red-black tree, discuss how nodes in the tree are represented.

Ans. : In insertion_sort.py data has been represented using single array.

In merge_sort.py, two temporary arrays have been used to store the “right” and “left” split of the input array “merge_list”.

In heap_sort.py a single array “arr” has been used for sorting. The array is represented as a tree here where 2*i+1 is the left child of i and 2*i+2 is the right child of the node i. Temporary variables have been used for swapping.

In quick_sort.py also a single array has been used to represent the data.

In bst.py a Node class has been used to represent every node in the tree. The “Node” class consists of left child, right child and value to store the value of the node.

Insertion sort- in-place

Merge sort -not in-place

Heap sort- in-place

Quick sort- in-place

For red-black tree each node object has 5 components which are: value, color, left child, right child and parent.

Redblack tree is constructed using the class RedBlackTree. The class is initialized with the root node object.

Describe any utility procedures implemented and their parameters.

Ans.:

insertion_sort.py- insertion_sort(insertion_list): This is the procedure which takes an array(insertion_list) as parameter and performs an in-place insertion sort algorithm on the input array.

merge_sort.py-mergeSort(merge_list): This is the recursive procedure which takes an array(merge_list) as parameter and performs an merge sort algorithm on the input array with the help of two auxiliary arrays left and right.

heap_sort.py-heapify(arr, i,n):This is a recursive procedure which maintains the max heap property in the array which is basically a tree representation. Here arr is the input array, i is the index of the parent node and n the length of the array.

quick_sort.py- partition(arr,low,high): This procedure takes the input array (arr), leftmost index and rightmost index of the partition as input and places the pivot element at its correct position in the array. The rest of the elements are swapped in a way such that all smaller values are at left of pivot and all greater elements are at right.

bst.py-height(root)- This is recursive procedure which takes the root of the tree as input and returns the height of the tree.

deletion(root,key)- This is a recursive function which takes the root of the tree(root) and node to be deleted(key) as input and recursively searches for the node. If the node is found it is deleted and replaced with the in order successor such that binary search tree property is conserved.

Insert(root, key)- This is a recursive procedure which takes root of the tree (root) and the value (key) to be inserted as input and recusively searches for the correct position to insert the key.

rb_tree.py- levelOrder(self): This procedure recurvely traverses the tree level by level. This procedure is basically used for showing the tree after insert and deletion.

get_height(self): This procedure recursively computes the height of tree.

transplant(self, u, v): This procedure replaces the subtree rooted at node u with the subtree rooted at v.

insert(self, node_value): This procedure takes a value (node_value) as input and inserts it in a position such that the red-black tree properties are conserved.

left_rotate(self, x): This procedure turns another node y’s left subtree into x’s right subtree.

Right_rotate(self,x): This procedure turns another node y’s right subtree into x’s left subtree.

delete_node(self.root, value): This procedure takes the root and the value as input and deletes the value from the tree and fixes the tree such that the red-black tree property is conserved.

Evaluation of Sorting Algorithms:

The average and sample standard deviation of the wall-clock runtime (in seconds) of each sorting algorithms has been calculated on 20 randomly permuted instances of each size n.

Evaluation for Insertion Sort:

No. of elements	Average	Standard deviation
10	5.686e-06	2.071e-06
10^2	0.000278	2.194e-05
10^3	0.0267	0.0005
10^4	2.734	0.0342
10^5	555.724	8.813

Evaluation for Merge Sort:

No. of elements	Average	Standard deviation
10	1.22427e-05	3.06001e-06
10^2	0.00016	7.04098e-06
10^3	0.00207	2.70709e-05
10^4	0.02587	0.00051
10^5	0.33043	0.02373

Evaluation for Heap Sort:

No. of Elements	Average	Standard Deviation
10	1.58309e-05,	4.04699e-06
10^2	0.00023	7.01067e-06
10^3	0.00366	0.00014
10^4	0.05482	0.00789
10^5	0.66655	0.02102

Evaluation for Quick Sort:

No. of elements	Average	Standard Deviation
10	8.51154e-06	5.84162e-06
10^2	0.00011	6.88773e-06
10^3	0.00151	5.55464e-05
10^4	0.02360	0.01054
10^5	0.25922	0.00527

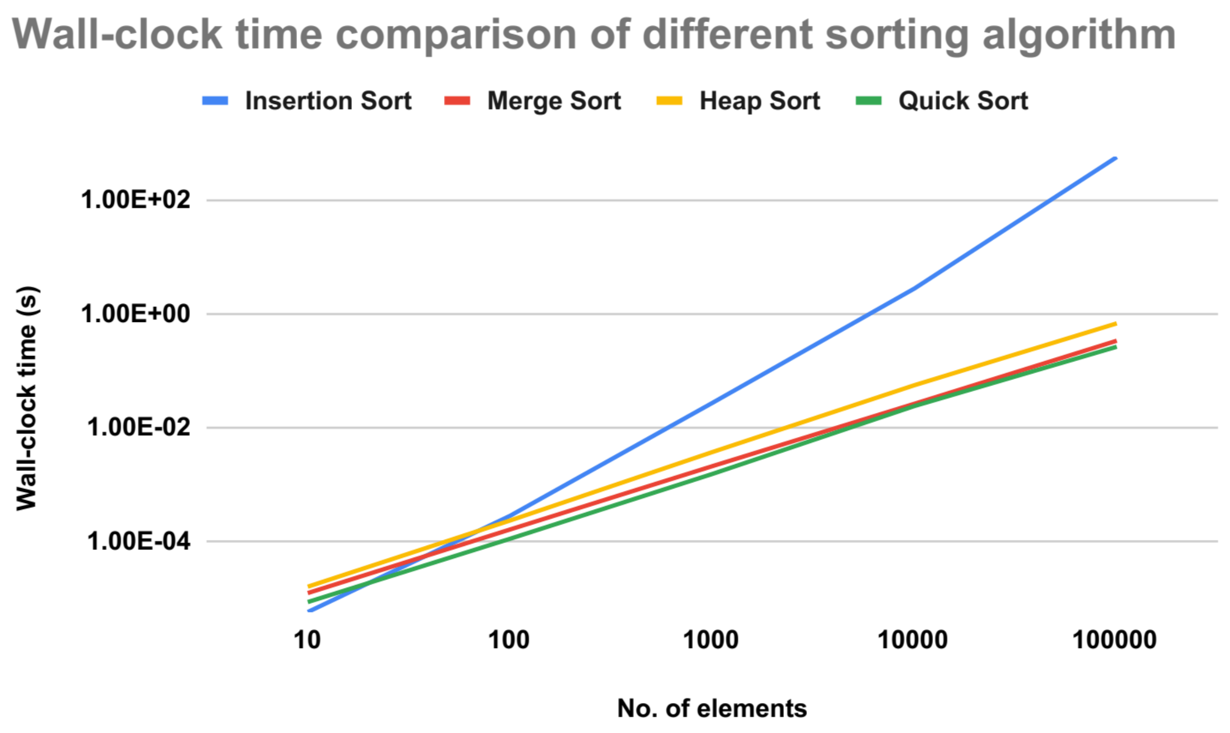
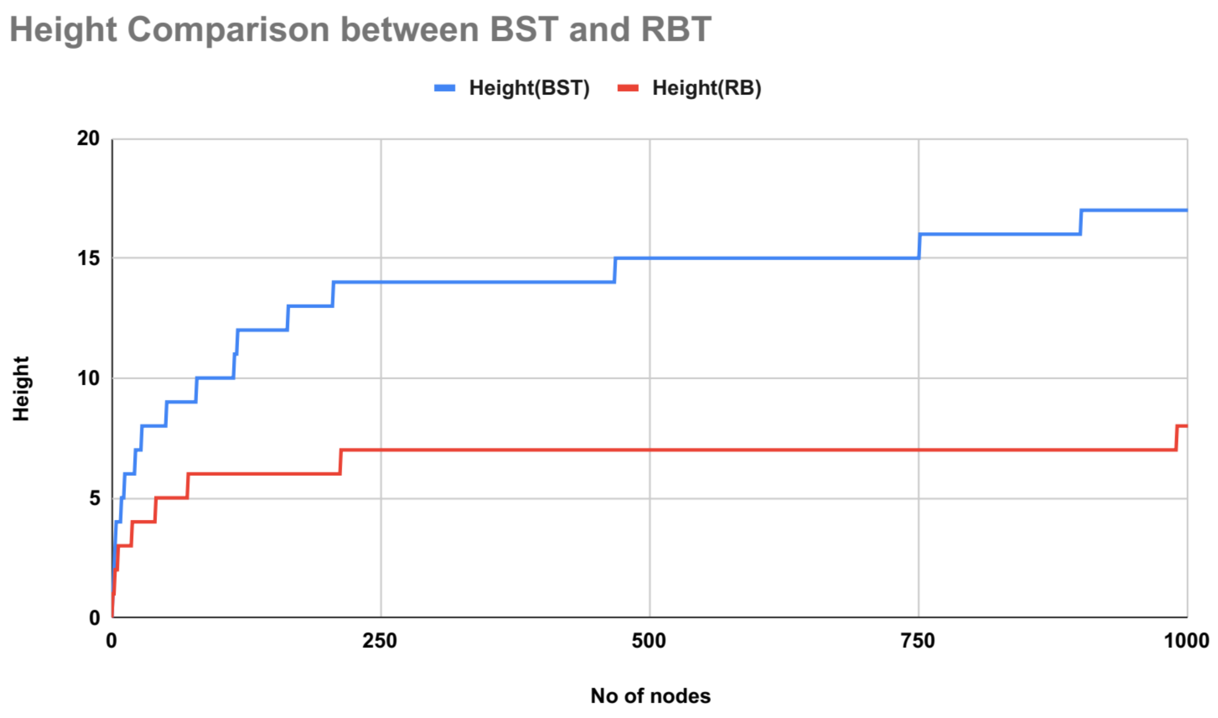


Figure-1: Wall-clock time comparison for different sorting algorithm

From Figure-1, we see that Insertion sort shows worst scalalbility as far as the clock time is concerned. Merge sort and Quick sort shows almost similar results up to 10^5 nodes. The Heap sort performs relatively worse compared to Merge and Quick sort. The values (wall-clock time) in y-axis are in logarithmic scale. Therefore in my evaluation, Quick sort performs the best. All the tests have been conducted in linprog.

Evaluation of Binary tree and Red black tree insertion:



For this evaluation we have inserted 1000 nodes in both the binary tree and Red black tree. From my evaluation, Red-black tree (RB) has performed significantly better compared to Binary Search Tree (BST) as far as the height is concerned. For 1000 node insertion, Red-black tree has a height(Height(RB)) of 8 whereas for the same number of nodes Binary Search Tree has a height (Height(BST)) of 8.