Name: Naga Venkata V Vinnakota
Net ID: nvv13
Course: ECE 16:332:573 DSA

Assignment-3

Q1 Develop an implementation of the basic symbol-table API that uses 2-3 trees that are not necessarily balanced as the underlying data structure. Allow 3-nodes to lean either way. Hook the new node onto the bottom with a black link when inserting into a 3-node at the bottom.

Solution:

Since there is no ideal methodology for implementing an unbalancing 2-3 trees, i.e. Insertion can be done in various ways, I have used the procedure that was discussed in the class, i.e. if a node has a single key, the new key should be added to the node. If the node has two keys, then the new key to be added is added to a new node that hooks onto the bottom, i.e. a new node gets added at the bottom, since it is not necessarily balanced.

I have implemented an unbalanced 2-3 tree which upon insertion checks if there is only one value in the node. If there is only one node, the new value to be inserted will be added to the same node following the below rules:

i. If the key to be inserted is smaller than the node, then the key is saved as the left value of the node.
ii. If the key to be inserted is greater than the value in the node, then the key to be inserted is saved as the right value of the node.

If the node already contains two values, then the insertion follows the procedure:

i. If key is smaller than the left value, the key is added to the left subtree if present. If there is no left subtree for the node, A left child is created for the node and this key is saved in the left child.
ii. If key is greater than the right value, the key is added to the right subtree if present. If there is no right subtree for the node, A right child is created for the node and this key is saved in the right child.
iii. If key is smaller than the right value and greater than the left value, the key is added to the center subtree if present. If there is no center subtree for the node, A center child is created for the node and this key is saved in the center child.

The above rules are checked for every node insertion.

Similarly, while deletion, if there is a single key node the deletion is same as Binary tree, but if the node has two keys and one of the value is the key to be deleted, It is replaced with the minimum value in the down subtree.

I have inserted few random values, printed the tree and deleted a value and printed the tree. Since it will be not feasible to verify the same for larger values, I just used few values. The search and print functions use Inorder traversal.

I have added few values/keys for insertion and deleted a key in the function main. You can run the code to understand how the functions insertion, deletion, print and search works in my code. Note use the functions with the syntax root.function(), where root is the object of the tree.

Q2. Run experiments to develop a hypothesis estimating the average path length in a tree built from (i) N-random insertions. (ii) N-sorted insertions?

Solution:

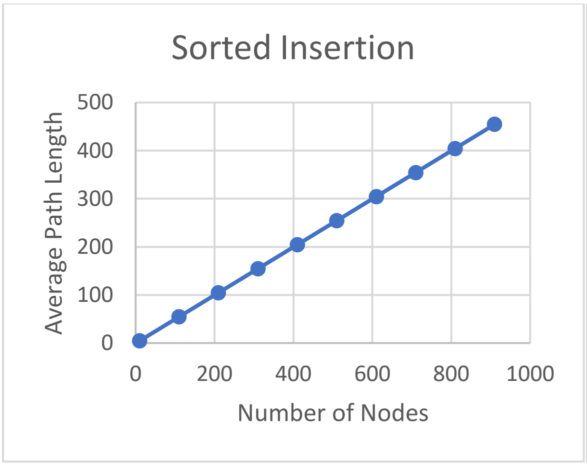
I have implemented a BST for the problem to find the average path length of the tree for N-random insertions and N-sorted insertions as there was no specification on the tree.

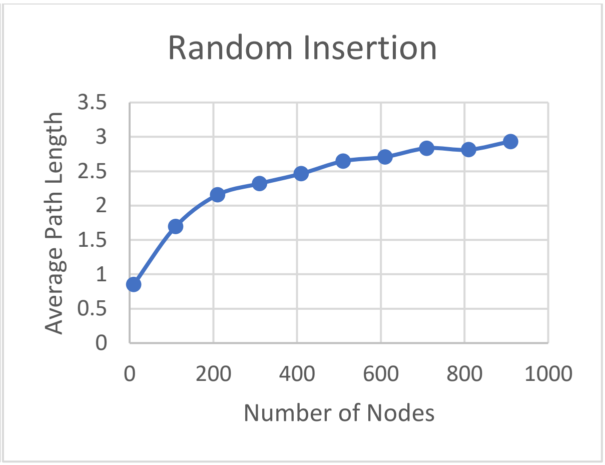
As a part of experiment of calculating Average path Length, I have run 100 trials on shuffled insertion and sorted insertion and calculated the mean of the path lengths for a given number of Nodes. This is done since random insertion won't give a constant Average path length so an average of 100 trials is used to estimate the same.

We already know that N-sorted insertions would give the worst-case average path length, as the tree is linear from smaller to larger node.

The below table gives the average path length found for N-random insertions and N-Sorted insertions.

Size	Sorted	Random
10	4.5	0.855
110	54.5	1.6958
210	104.5	2.156
310	154.5	2.32
410	204.5	2.464
510	254.5	2.646
610	304.5	2.70765
710	354.5	2.83528
810	404.5	2.8129
910	454.5	2.93325





From the experiments run, we can observe that the average path length increases linearly as the number of nodes increases for sorted insertions. So, we can argue that the Average path length for sorted insertions is of order O(N), while the same for Random insertions is of order O(logN).

We can observe that the worst case average path length is almost half the number of nodes i.e. is of order O(N) where N is the Number of nodes, i.e linear for N-sorted insertions and based on the values above, we can see that the worst for finding an element is of order O(N).

For N-Random insertions we can observe that the average path length for random insertions is almost log(N) and based on the above values, we can observe that finding an element takes nearly average case time i.e O(logN).

Conclusion:

For a Binary search Tree, the average length when the insertions are random are given by ~log(N) which is the average length of a Binary search tree.

When the insertions are made with a sorted data, then the (maximum) height of the tree will be ‘N’ and the average path length will be 0.5N.

Q3. Write a program that computes the percentage of red nodes in a given red-black tree. Test program by running at least 100 trials of the experiment of increasing N random keys into an initially empty tree for N=10^4, 10^5 and 10^6 and formulate a hypothesis.

Solution:

A Red-Black tree was implemented but only the functions Insertion, is_red(), red_count(), rotate functions and flip are used so far as the main goal here is to calculate the average percentage of the nodes present in a Red Black tree for N random insertions.

Total of three sizes i.e three different number of nodes, N were considered where N= 10^4, 10^5 and 10^6. A total of 100 runs were executed to compute the percentage of the red nodes present in the tree.

A function count() is used to count the nodes. This function returns the total number of nodes present in the Red-Black tree formed, while red_count() is used for counting red nodes.

The percentage of red nodes is calculated using the ratio of the results from red_count() and count().

Observations:

For the tree with nodes, N=10000, the percentage of RED nodes in average is 25.4331%

For the tree with N=10^5 Nodes, the percentage of RED nodes on average is 25.3947%

For the tree with N=10^6 Nodes, the percentage of RED nodes on average is 25.3981%

Conclusion:

As we can see, the total of the red nodes will cover almost 25% of the entire red-black tree irrespective of the number of nodes inserted. This means that we can find a red node for every 4 nodes as we traverse along a Red-Black tree.

Therefore, we can conclude that the average percentage of the red nodes present in a RED_BLACK tree will be nearly 25-25.5% independent of the number of nodes present in the tree.

Q4. Run empirical studies to compute the average and std deviation of the average length of a path to a random node (internal path length divided by tree size) in a red-black BST built by insertion of N random keys into an initially empty tree, for N from 1 to 10,000. Do at least 1,000 trials for each size.

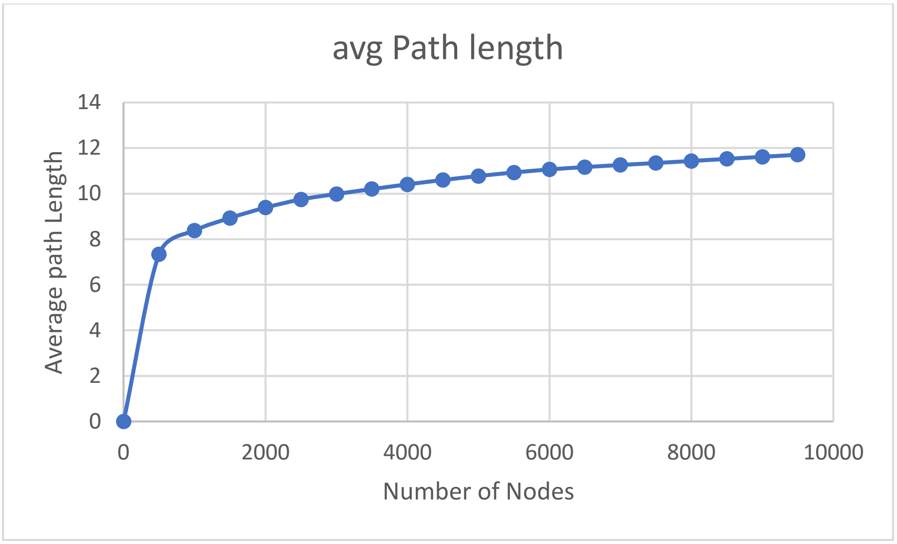
Solution:

```
For Size 1 Average Path Length is 0.0 with Standard Deviation 0.0
For Size 500 Average Path Length is 7.334445 with Standard Deviation 0.10495460276774654
For Size 1000 Average Path Length is 8.37359 with Standard Deviation 0.08289603829170882
For Size 1500 Average Path Length is 8.92904 with Standard Deviation 0.06389065191772221
For Size 2000 Average Path Length is 9.38820 with Standard Deviation 0.06609297366818158
For Size 2500 Average Path Length is 9.745495 with Standard Deviation 0.0972734004710179
For Size 3000 Average Path Length is 9.98837 with Standard Deviation 0.09687301856466562
For Size 3500 Average Path Length is 10.1917 with Standard Deviation 0.05741883165277517
For Size 4000 Average Path Length is 10.4098 with Standard Deviation 0.059391549579523625
For Size 4500 Average Path Length is 10.5967 with Standard Deviation 0.06424243060032173
For Size 5000 Average Path Length is 10.7689 with Standard Deviation 0.07259003617173933
For Size 5500 Average Path Length is 10.9314 with Standard Deviation 0.08951949172076663
For Size 6000 Average Path Length is 11.0587 with Standard Deviation 0.10657030512116994
For Size 6500 Average Path Length is 11.1655 with Standard Deviation 0.11121960872788635
For Size 7000 Average Path Length is 11.2503 with Standard Deviation 0.09733960471662811
For Size 7500 Average Path Length is 11.3454 with Standard Deviation 0.07826570809672004
For Size 8000 Average Path Length is 11.4362 with Standard Deviation 0.06288156439814264
For Size 8500 Average Path Length is 11.5248 with Standard Deviation 0.05481881006469087
For Size 9000 Average Path Length is 11.6172 with Standard Deviation 0.05477999995236781
For Size 9500 Average Path Length is 11.7057 with Standard Deviation 0.059866797309929304
For Size 10000 Average Path Length is 11. 78581     with Standard Deviation 0.06179410418
```

From the above data we can understand that as the number of nodes increases, the average path length increases, but slightly.

Using these values, I have plotted the average path length vs the number of nodes, which is a shown below.

We can clearly understand that the average path length computed from the program for a given number of Nodes, N, is nearly equal to c*log(N).



From the plot we can observe that the average path length is following the logarithmic trend as the number of nodes increases.

The average path Length though increases with increasing nodes, this increase is following the logarithmic scale i.e. ~c*log(N). Using this observation, we can conclude that the average path length of a tree for a given number of nodes is always a function of log(N).

This clearly gives us an idea that searching a key in a RED-Black tree for average case (Random Insertions) is of order log(N).

Conclusion:
The average path length of a red-black tree will be of order O(logN) for N number of nodes present in it provided the insertion is done in random order.

Q5. Implement the rank() and select() ordered operations for a BST. Use data set linked below. (i) What is the value of select(7) for the data set? (ii) What is the value of rank(7) for the data set? [10 points]

Solution:
The rank and select operations are implemented by using a list which stores the values as we traversals along the tree.
The best approach to store the keys into the list is by INORDER traversal as it always starts with the smallest key and ends with the largest key.

Whenever the rank(key) or select(rank) is called, an empty list is passed through inorder function which stores all the keys in ascending order.

The select directly uses the values in the list for a given key and returns the same. Whereas the rank returns the index of the key passed.

Result for the data set provided:

rank(7) is 6
select(7) is 8

References:

- https://www.tutorialspoint.com/python_data_structure/python_binary_tree.htm

- Referred few slides from the class for rotation, Red-Black trees insertion and flip functions.