SPLAY TREE VS RED BLACK TREE

Team Members: K YASWANTH(2018202011), SVL SARAT(2018202013)

Deliverables:

- Menu driven c++ code for splay and red-black trees.
- Python script for test case analysis.

Technologies used:

C++ (for building project) and python (for visualization).

Online resources:

- https://en.wikipedia.org/wiki/Splay_tree
- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
- http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf

Repository where work is being committed:

- https://github.com/yaswanthkoravi/Apsproject.git

Splay tree:

A splay tree is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality. A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation. Importantly, splay trees offer amortized O(log n) performance; a sequence of M operations on an n-node splay tree takes O(M log n) time.A splay tree is a binary search tree. It has one interesting difference,whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant.If the next lookup request is for the same element it can be returned immediately.

Red Black Tree:

Red-Black trees are a popular tree structure for real time applications. A Red Black Tree is a type of self-balancing binary search tree, in which every node is colored with eithe red or black. The red black tree satisfies all the

properties of the binary search tree but there are some additional properties which were added in a Red Black Tree. The height of a Red-Black tree is O(Logn) where (n is the number of nodes in the tree).

Properties of Red Black Tree:

- The root node should always be black in color.
- Every null child of a node is black in red black tree.
- There are no two adjacent red nodes i.e a red node cannot have a red parent or a red child.
- Every simple path from the root node to the (downward) leaf node contains the same number of black nodes.

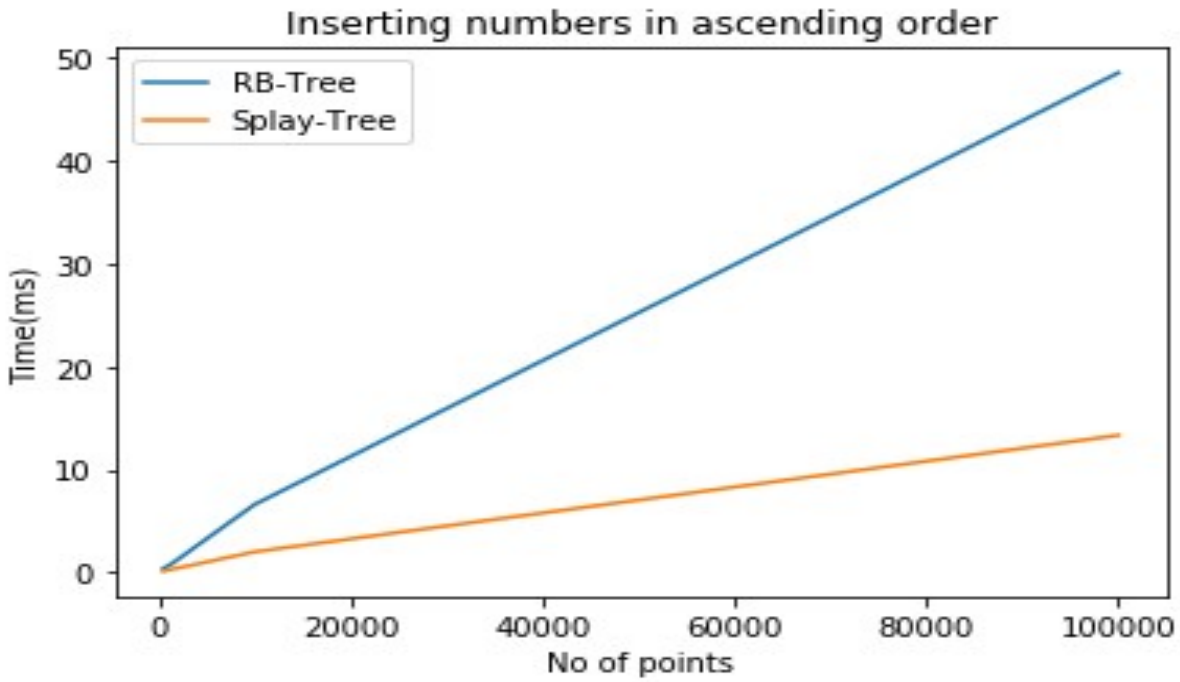
Performance analysis:

We will compare this two trees based on the following operations

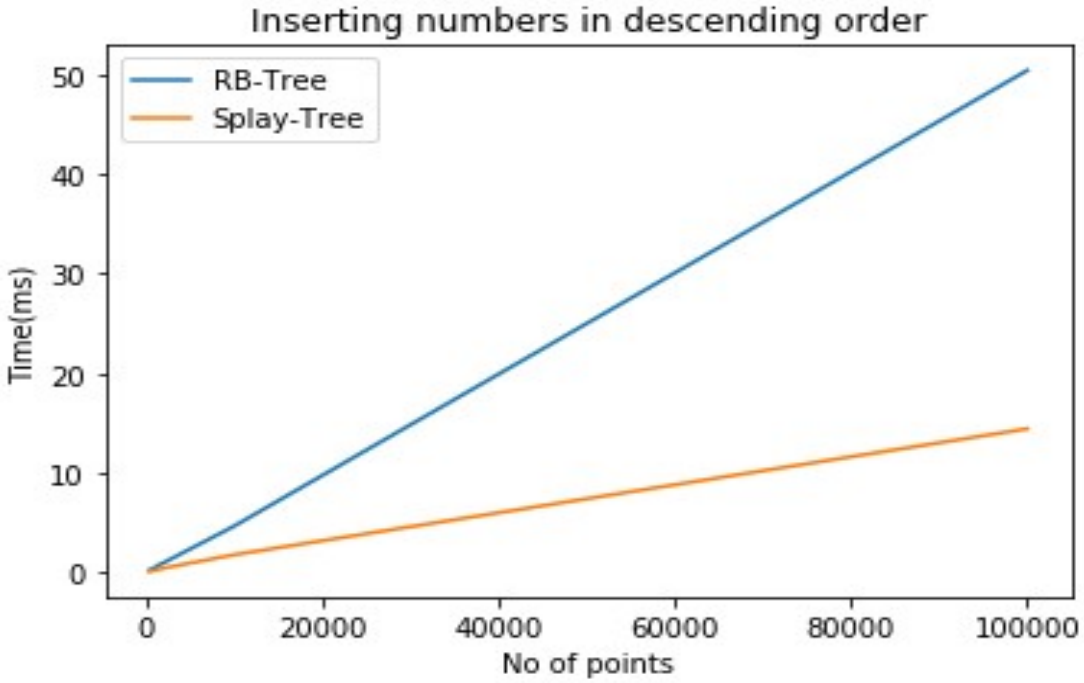
- Insertion.
- Searching.
- Deletion.

Insertion:

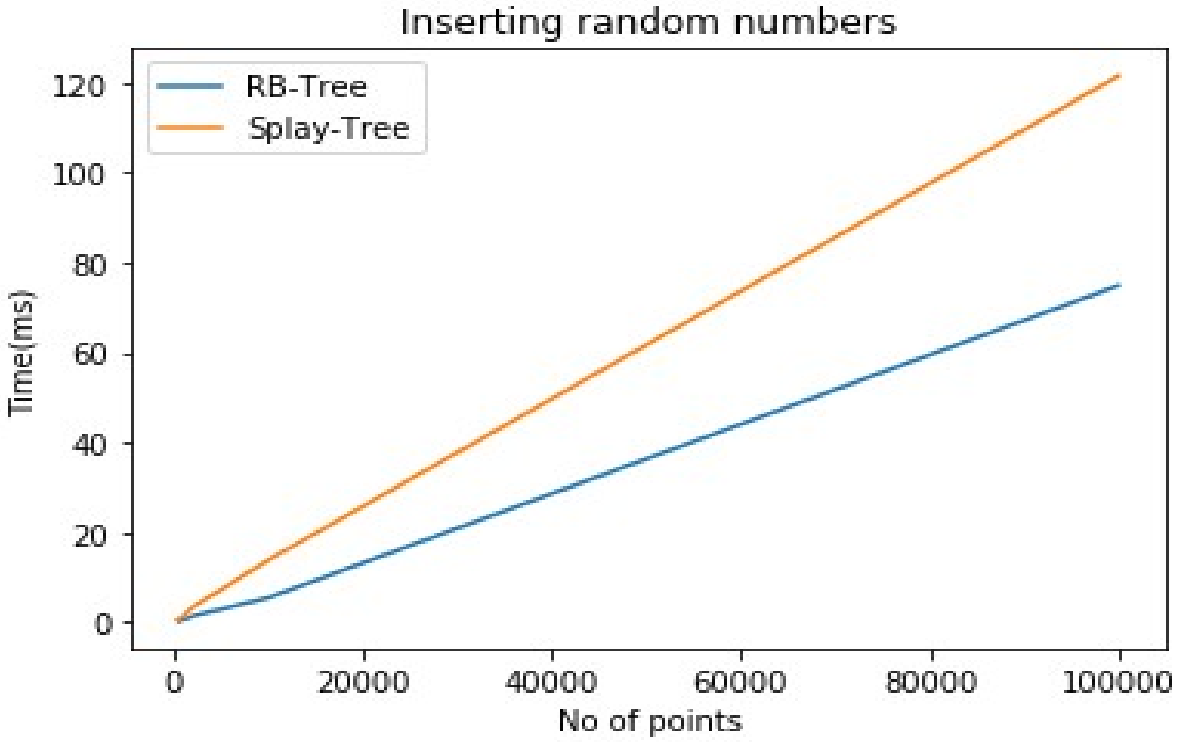
Consider three different cases while inserting elements into the tree
1. Elements inserted in ascending order.



2. Elements inserted in descending order.



3. Elements inserted in random order.



Observation for inserting elements:

Case 1&2:

Skew trees will be formed in splay tree because of performing splaying operation after each insertion of element and moving that to the root, whereas in red-black tree there will be multiple rotations inorder to maintain the property of red-black trees so inserting elements either in ascending order or in descending order will take more time in red-black tree when compared to splaytree.

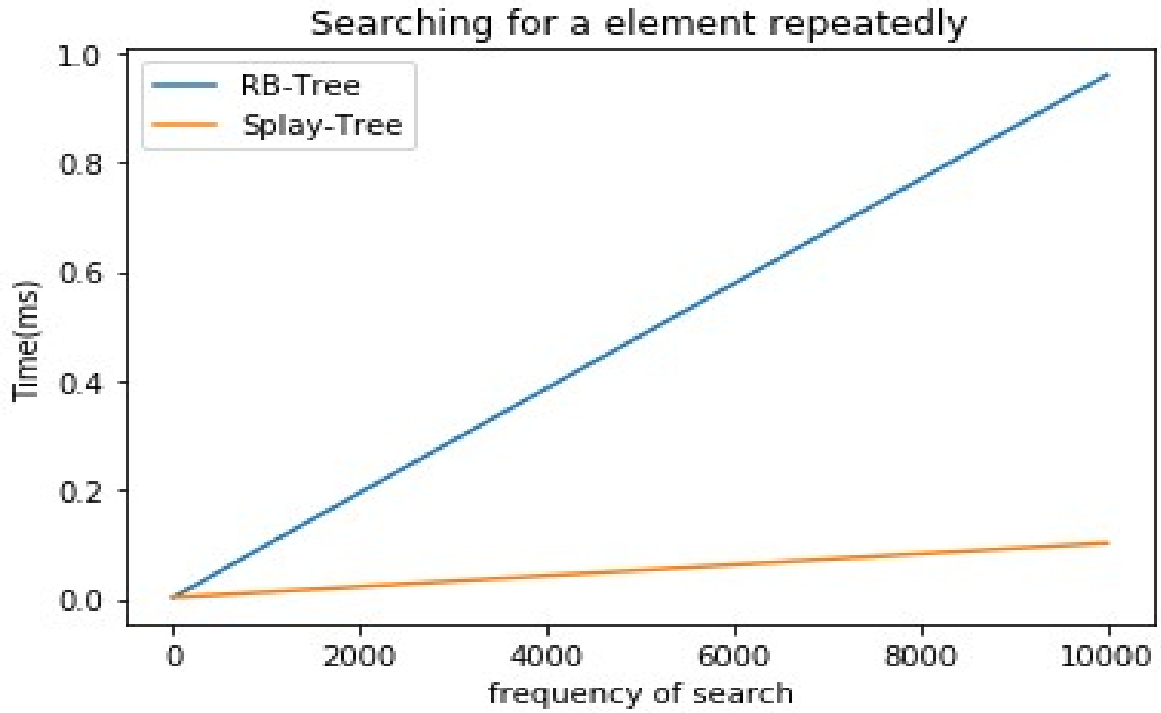
Case 3:

While inserting elements in random order splay tree will take more time when compared to red-black tree because splaying will bring the newly inserted element to the rooot after each insertion but red-black tree maintains its property while inserting elements into the tree by performing appropraite roatations.

Search:

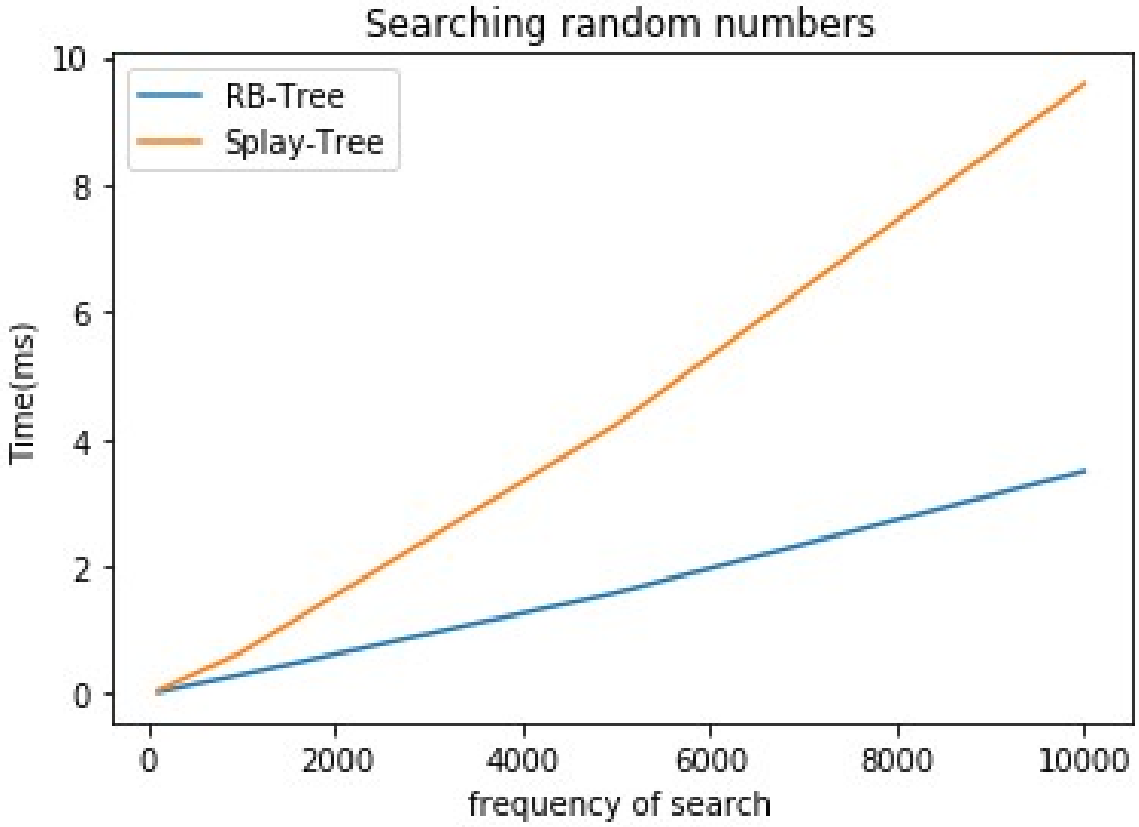
To find the performance of search operation in both trees we wil insert 100000 random elements in both the trees.

Observation for searching elements:



Case 1:

We are performing search operation of constant element for multiple times say(1,100,1000,10000) due to splaying property of splay tree there will be huge time gap between this two trees,in splay tree after first search operation it will move that constant searched element into the root and for next search operations it will take constant time to access that element,whereas in red black trees it takes the same amount of time for every of search operation of that constant element.

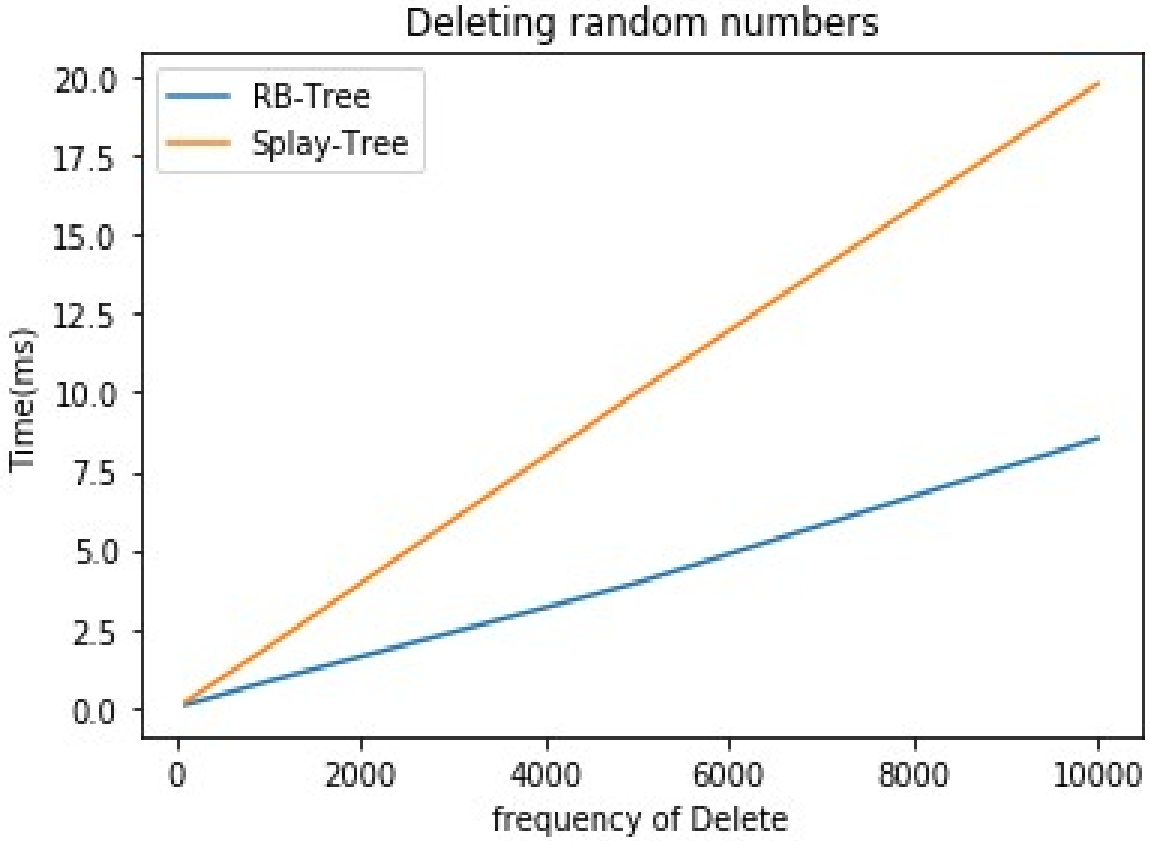


Case 2:

The search algorithm will perform same as the BST search algorithm. In splay tree if a element is not found, then the last node where search operation is ended will be moved to the root.Whereas in red black tree, search will terminate by reaching null nodes. In this test case,we are searching random elements multiple times say(1,100,1000,10000). In this case red-black trees performs better than splay trees because after every search opeation the splay structure will be changed whereas the structure of red-black tree remains constant.

Delete:

To find the performance of delete operation in both trees we wil insert 100000 random elements in both the trees.



Observation for deleting elements:

Here we deleting a random number formultiple times say (100,1000,5000). Splay tree will take more time when compared to red-black tree because while deleting element we will first search that element and bring that element to the root and delete that element and make inorder succesor as the root whereas in red-black tree we will search that element and delete it and we will perform approprate rotations to maintain the properties of the red black tree.

Conclusion:

- If our application has more number of insert operation than search operation prefer **red-black tree.**
- If our application has repeated number of search operation for the same element prefer **splay tree**(cache implementation).

End User Documentation:

We have implemented menu driven program for both splay and red black trees. For test case analysis, we included python script files in tests folder.

For running splay and red-black trees:

- Run *g++ -o splay splay.cpp* and *g++ -o redblack redblacktree.cpp* command through terminal.
- *.out* files will be generated for both the codes respectively.
- Execute this files using ./splay and ./redblack commands respectively .

The following menu will be displayed for red black tree

- Press 1 to insert: Enter the value to be inserted into the red black tree.
- Press 2 to delete: Enter value to be deleted from the tree, it will first search for the given value and then deletes the value if it is found in the tree, else program will return *not found* message.
- Press 3 to search: Enter value to be searched from the tree,it will follow the normal BST search algo and displays whether the element is found in the tree or not.
- Press 4 to display: Inorder travesel of nodes will be displayed along with the colors of the nodes.
- Press 5 to exit: It will exit from the main program.

The following menu will be displayed for splay tree

- Press 1 to insert: Enter the value to be inserted into the splay tree.
- Press 2 to delete: Enter value to be deleted from the tree, it will first search for that value and move it to the root and then delete that element if that value is found in the tree, else it return *value not found* message along with that last element where search ended will be moved to the root.
- Press 3 to search: Enter value to be searched from the tree,it will follow the normal BST search algo and moves that particular element to the root, if that element is not found it will move the last element where the search ended to the root.
- Press 4 to display:Inorder travesel of nodes will be displayed.
- Press 5 to exit: It will exit from the main program.

For running python script:

Open tests folder and run *python3 file-name.py* for various scripts present in that folder to get test case analysis. The files inlcude

- insertion1.py : Inserting numbers in ascending order in both trees for different frequencies and measuring time intervals for each cases.

- insertion2.py : Inserting numbers in descending order in both trees for different frequencies and measuring time intervals for each cases.
- insertion3.py : Inserting numbers in random order in both trees for different frequencies and measuring time intervals for each cases.
- search1.py : searching for a constant element for different frequencies in both the trees and measuring time interval for each cases.
- search2.py : searching for a random elements for different frequencies in both the trees and measuring time interval for each cases.
- delete.py : deleting random elements from both the trees and measuring time interval for each cases.