Nicole Vachon
Ryan Lockett

Red-Black Trees

Introduction

Red-black trees are a type of binary search trees. They are subject to the requirements imposed on binary search trees, specifically the order property where the left child of each node has a key value less than the key value of its parent and the right child of each node has a key value greater than the key value of the parent. Red-black trees are also subject to other requirements such as (1) each node must either be red or black and (2) the root must be black. (3) All leaf nodes are null pointers and must also be black. Also (4) both children of a red node must be black, ie. no two red nodes can be parent/child of each other. And finally, (5) every path from the root node to any null leaf must have the same number of black nodes. These restrictions allow the user to know with certainty that the path from the root node to the farthest null leaf is no more than twice as long as the root node the the nearest leaf node. The proof for this is quite simple as the path between the root node to the nearest leaf node has only x black nodes and 0 red nodes while the path from the root node to the farthest leaf must also have x black nodes, to fulfill requirement 5, and x red nodes as only one red node can go between each black node, to fulfill requirement 4. This leaves the length of the path from the root node to the nearest null leaf x and the length of the path from the root node to the farthest null leaf $2x$ which is twice the length of the path from the root node to the nearest null leaf.

This data structure guarantees worst-case runtimes for the functions insertion, search, and delete, all $O(\log n)$. This is helpful for real-time applications and functional programming including storing arrays or sets for previous versions before mutating the function.

In addition to the three standard functions of most data structures, insertion, search, and delete, it is also possible to call parent, children, maximum value, minimum value, and a fix function for both insert and delete to correct any violations of properties 2 or 4 to maintain the color property. The insertion function follows a binary search tree insertion function where is checks the value of the key with the root to determine which side of the tree to insert, and recursively checks until a null leaf node is found, at which point the node to be inserted replaces the null leaf as a red node and creates two new null leaf nodes as the new node's children. After the insertion, a check must be done to see if any of the red-black tree properties have been violated, and if so, must be fixed with the rotation/color change function. The search function is identical the the binary search tree search function that moves along the tree in the same way as insertion, checking every key value against the search key, only returning if it found the node, returning true, or found a null leaf node, returning false. The delete function uses the search function to find the appropriate node, just like in a standard binary search tree, then deleting the

node which returns from the search function, unless it is a null leaf, in which the delete function does not delete anything. After a delete is performed, a check must be done again to ensure the red-black trees do not have any properties that are violated, and if one is, must be fixed with the fix function. The parent function utilizes part of the search function by starting at the root then searching for the node which was given to determine its parent and returning the last node checked before finding the node in the search. The children function returns both left and right child of the node given, as they are stored with pointers. The maximum value function returns the rightmost node by starting with the root and following the right child of each successive node until a null leaf is found as the right child, signifying the last right child node with a value. Because the key value of the right child of a node is always larger than the node's key value, the rightmost child has the largest value in the tree. The minimum value function returns the leftmost node by starting with the root and following the left child of each successive node until a null leaf is found as the left child, signifying the last left child node with a value. Because the key value of the left child of a node is always smaller than the node's key value, the leftmost child has the smallest value in the tree.

The fix functions are the most vital functions of the red-black tree data structure. Both the fix functions first checks if there is a red node that has a red child and if so, will change the color of the child to black. Then the insert fix function checks if the node inserted was red then goes left then if the next node is also red, pushed the inserted red node right, then checking if the new parent was black then checks if the nodes need rotating based on property 5. If the inserted node is black, the same process is done except switching the colors then once again checking if a rotation is needed. The rotation functions are simply AVL rotations. The delete fix function starts by checking if it was the root node that was deleted, and if so, changing the color of the replacing node to be black if needed. After any node is deleted, if both children off the deleted node are black, then a right rotation is done into a red node. If both children off the deleted node are red, then a left rotation is done into a black node.

Correctness and Running Time

Starting with the simplest function, the search function is correct as it identical to a binary search tree search function with has been proven in class to be correct. The search function then also has the same running time for both red-black trees and binary search trees, that being O($log\ n$) as the search cuts the number of nodes to search through in half each time in hits another node. This is true as it an easily be found if the node being searched for is in the right subtree or left subtree of the current node by comparing the key values of the current node and the node being searched for.

Nicole Vachon
Ryan Lockett

Let the following function be insertion. The correctness of insertion is dependant on the correctness of the insertion of a binary search tree as well as the correctness of the fix function. The latter will be analyzed later in this section, showing it is correct. The former has been proven correct in class, therefore the insertion function is correct. This function has a running time of O(*log n*) as it calls the search function which has a running time of O(*log n*) and then calls the fix function which either will do O(*1*) color changes or O(*log n*) rotations, which will be shown later in this section. Therefore the running time of both the search and the fix together in the insertion function is O(*log n*).

The next analysis will be on the delete function. The correctness of the delete function is dependant on the correctness of the delete function of a binary search tree as well as the correctness of the fix function. The latter will be analyzed later in this section, as stated before, showing it is correct. The former has been proven correct in class, therefore the delete function is correct. This function has a running time of O(*log n*) as it calls the search function which has a running time of O(*log n*) and then may call the fix function depending on if the delete has violated any of the red-black tree properties. Therefore the running time of both the search and the fix together in the delete function is O(*log n*).

The minimum and maximum functions are identical and opposite to each other therefore they will have the same analysis. Both these functions are correct because of the structure of binary search tree where the smallest number is the farthest left node and the largest number is the farthest right node. These functions find the farthest left node and farthest right node respectively, therefore they are correct. Their running times are also identical. Because the red-black tree has the property that the length of the path to the farthest leaf is no more than twice the length of the path to the nearest leaf, the running time if the max/min is the farthest leaf is O(*log n*) and the running time if the max/min is the nearest leaf is O((*log n)/2*) therefore the running time of both these functions is O(*log n*).

The children function is correct as it stores the children as a property of the node. This makes the running time O(*1*) as it is only a call to the property.

The parent function is very similar to the search function, only returning the last node previously found before the child node given, which is the parent of the given child, therefore this function is correct. The running time of this function is the same as the running time of the search function as it is the same process, therefore its running time is O(*log n*).

Both the insert fix function and the delete fix function check then, if necessary, correct any violation of properties 2, 4, and 5 by doing a simple check then doing a color change or an appropriate rotation when necessary. They will do at most 2 color changes to correct for the root

not being black and the red parent having 2 red children, therefore it will do O(*1*) color changes. This function will also do at most O(*log n*) rotations to correct property 5, therefore worst case, both these functions have a running time of O(*log n*).

Experiments and Results

To calculate the running time of insertion, delete, and search, we will run each operations multiple times, using crono_highresolution_clock to output the time in nanoseconds, then take an average of the data outputted to ensure we get the most precise final running time for each operation. All three functions also had an average space requirement of 10mb.

Insertion

100,000:	**1,000,000**	**10,000,000:**	**50,000,000:**
93695000	1087739000	11716235000	61865602000
93715000	1032009000	11051485000	62327676000
78092000	1016752000	11180035000	65506102000
78064000	1055641000	11103911000	60493281000
93698000	972483000	11102984000	60654056000
Avg: 87452800ns	Avg:1032924800ns	Avg:11230930000ns	Avg:62169343400ns

Search

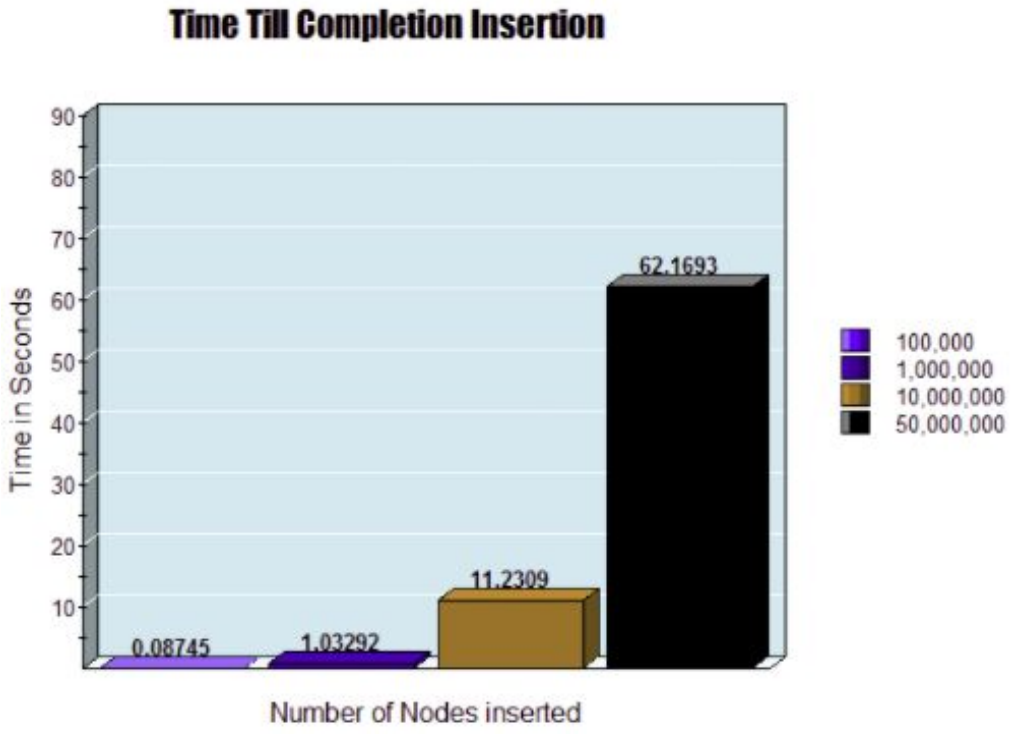
100,000	**1,000,000**	**10,000,000**	**50,000,000**
86454000	978322000	9881625000	57562102000
91234000	938975000	9953244000	60832536000
78324000	968349000	10082356000	55636324000
81322000	984033000	9853245000	58442383000
81293000	990234000	9973296000	59256325000
Avg:83725400ns	Avg:971982600ns	Avg: 9948753200ns	Avg:58345934000ns

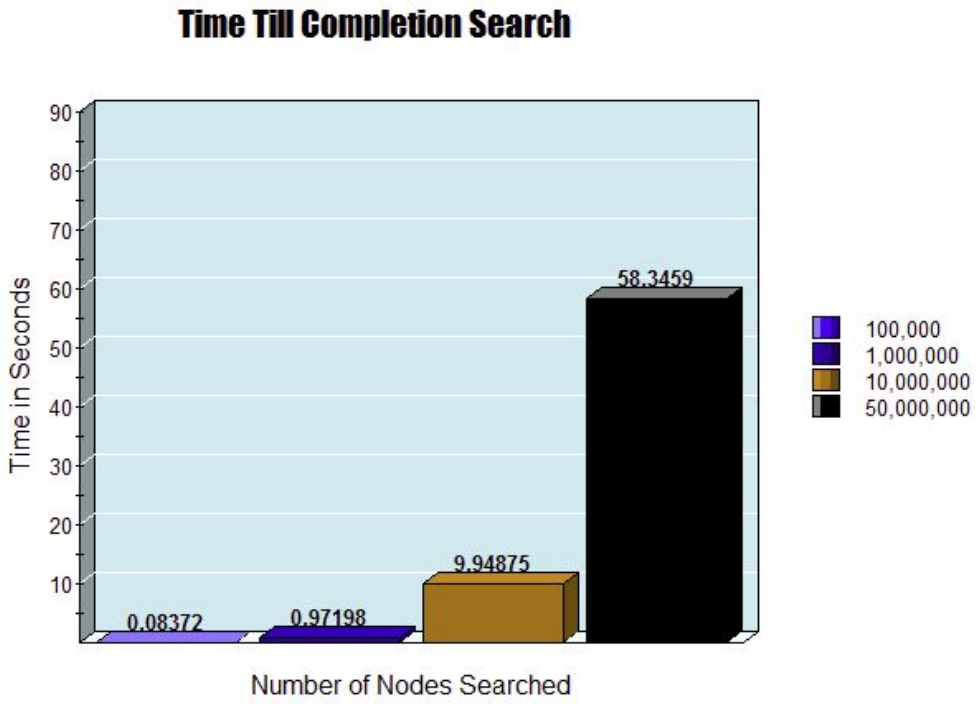
Deletion

100,000	**1,000,000**	**10,000,000**	**50,000,000**
79347000	959371000	9745635000	59174211000
88968000	929862000	9637831000	58344367000
83632000	938721000	9932116000	54834129000
85341000	947432000	9763239000	55883289000
81975000	961238000	9843843000	57656327000
Avg:83852600ns	Avg:947324800ns	Avg:9784532800ns	Avg:57178464600ns

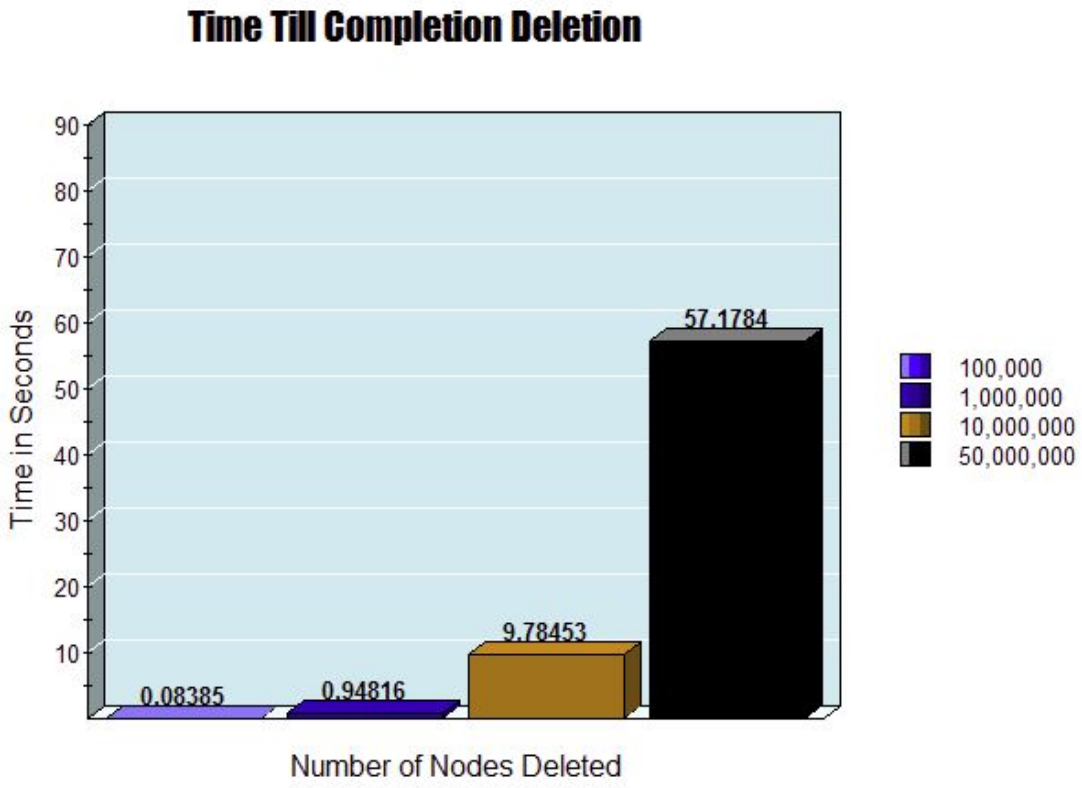
Nicole Vachon
Ryan Lockett

Graphs





Nicole Vachon
Ryan Lockett



Conclusions

This data shows that our implementation of insert, search, and delete all have very similar running times. We can also conclude that these functions have a linear running time based on the ratio between time in seconds and number of nodes being equal as the number of nodes increased. From the graphs, we can see that as we multiply the number of nodes by 10, the time it takes in seconds is also 10 times greater. Between 10,000,000 nodes and 50,000,000 nodes, the number of nodes is 5 times more as well as the time taken is roughly 5 times more, from around 10 seconds to around 50-60 seconds.

Comparisons

The calculated running time in section Correctness and Running Time for insert, search, and delete functions were all O(*log n*). If our functions were running at O(*log n*), we would see a very small difference in running time between each of our data sets, rather than the large gaps shown in our results. Based on our conclusion of our implementation, our functions are running at O(*n*) therefore our implementation is not the most efficient implementation of these functions.