COMPSCI 2XB3 L04 Lab Report #6

Mahad Aziz - azizm17 - 400250379 - azizm17@mcmaster.ca
Talha Amjad - amjadt1 - 400203592 - amjadt1@mcmaster.ca
Logan Brown - brownl33 - 400263889 - brownl33@mcmaster.ca
McMaster University

COMPSCI/SFWRENG 2XB3: Binding Theory To Practice

Dr. Vincent Maccio

TA: Amir Afzali

March 5, 2021

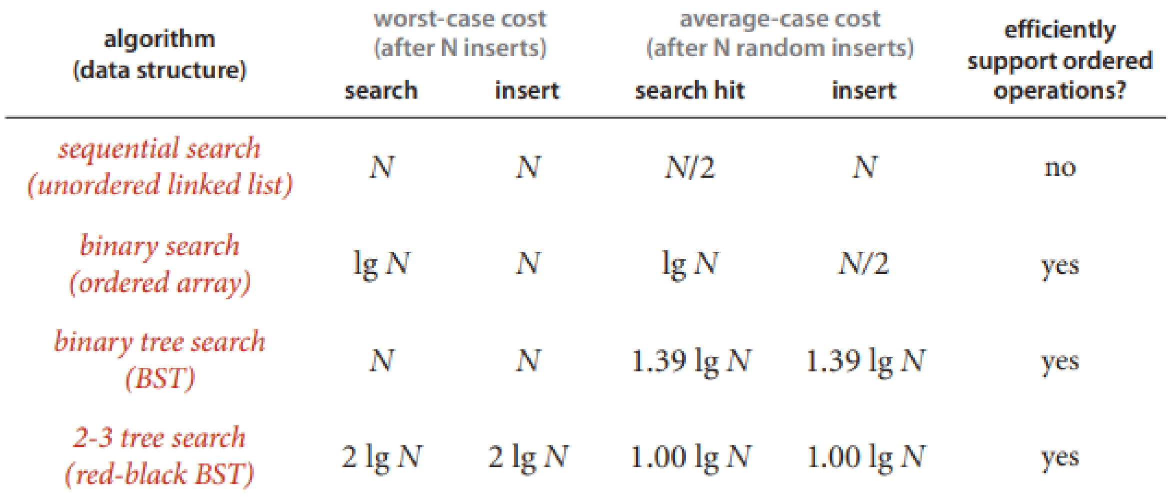
Red-Black Tree Height:

Scenario:

We believe that there is not an issue with our insert method. We believe the reason the height becomes smaller after inserting the values from 1 to 10,000 three times is because of the self-balancing property of RBTs. One explanation may be for the first round of inserts, the RBT is close to its worst-case height which is given by $2\log_2(10000)$. In the next round of inserts, the RBT may have gotten more of a chance to rebalance itself which is why it stays at a height of 24. In the final round of inserts, the specific input may allow the RBT to rebalance itself in a more efficient way which leads to a decreased height of 16.

Should we always use RBTs over BSTs?

For nearly every situation, an RBT is superior to BST. When it comes to inserting values into a tree, an RBT is much better because of its self-balancing property. Regardless of the value itself, when a value is inserted various rotations are used throughout the tree to ensure that it is balanced even if that means resetting the root of the tree to another value. For the case of the BST, inserting can lead to very bad trees since the height can become very large based on the values being inserted. BSTs have no concept of self-balancing which could potentially lead to very unbalanced trees with very large heights. Looking at the image below, taken from *Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne,* it's clear that RBTs outperform BSTs. However, if you are trying to minimize storage usage for your code and minimize the difficulty of implementation, then a BST may be the better option.

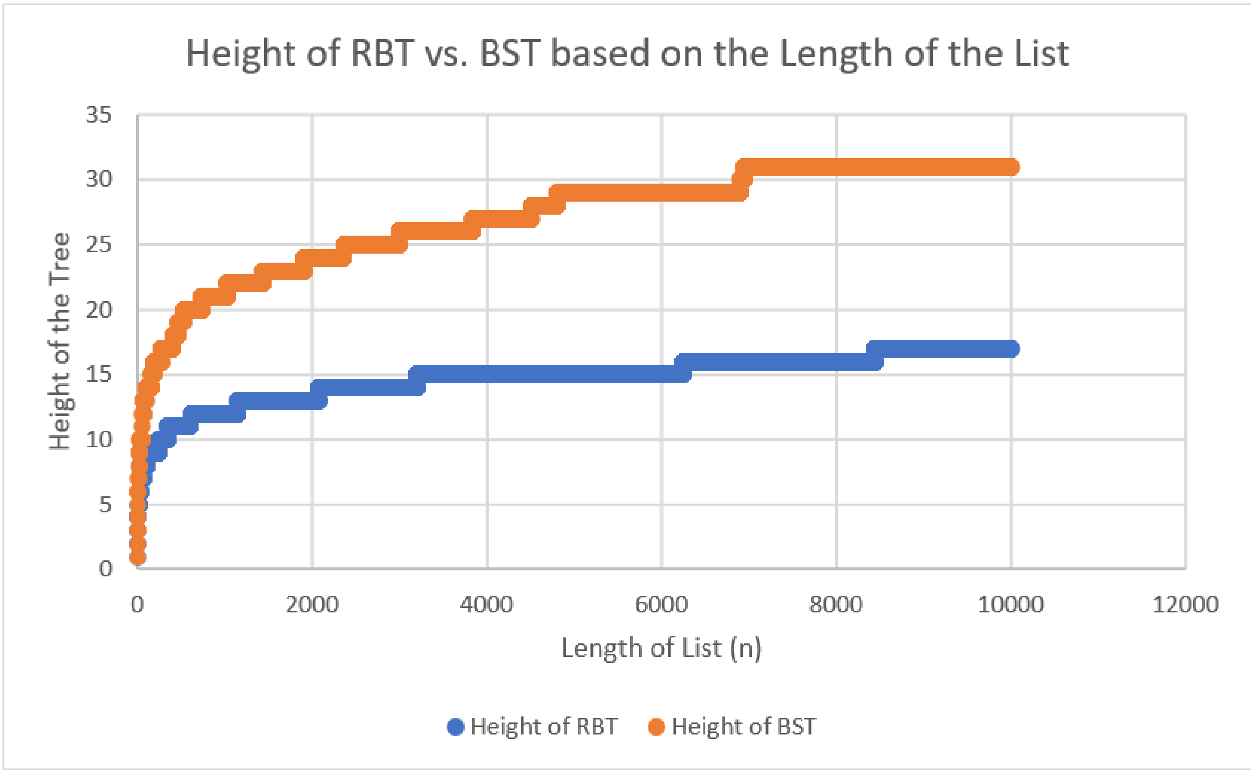


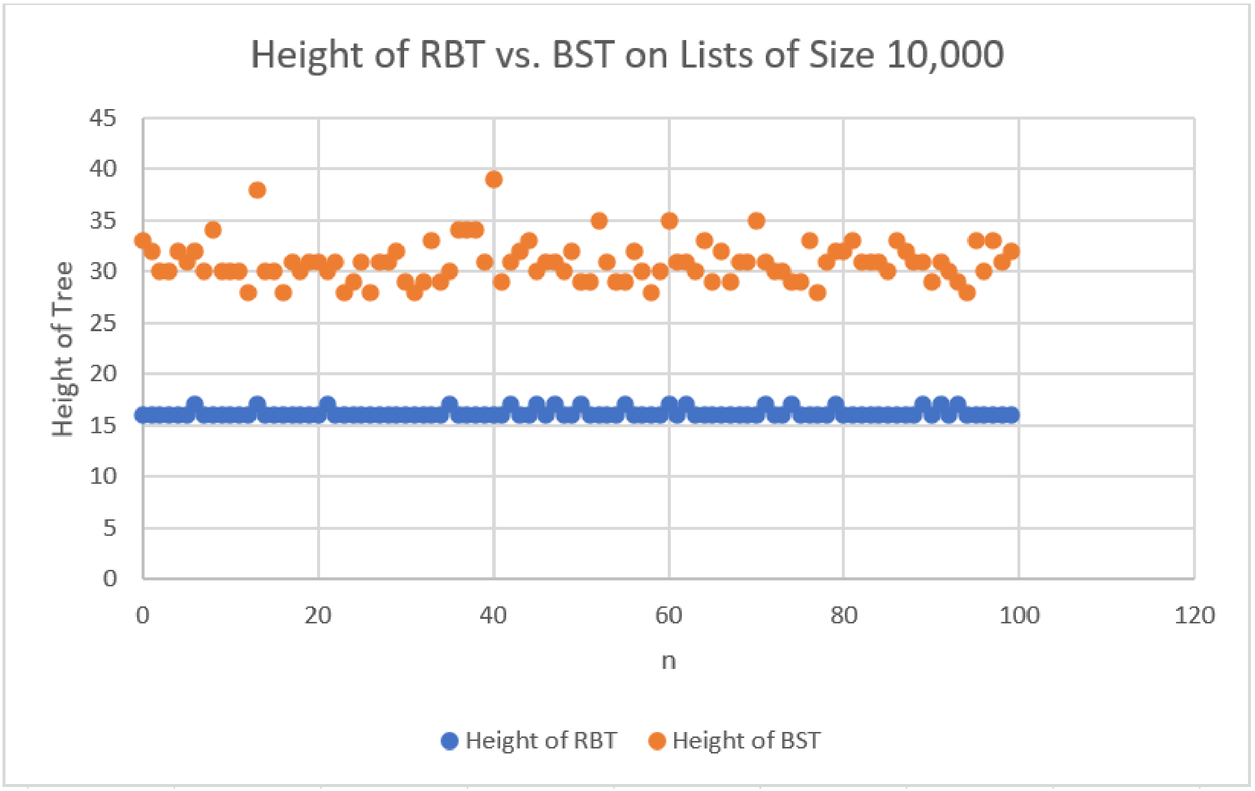
algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$	yes
binary tree search (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$	yes
2-3 tree search (red-black BST)	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes

Cost summary for symbol-table implementations (updated)

What is the average difference between the height of RBT and BST?

The average difference between the height of an RBT and BST when used on lists with a length of 10,000 is about 15. From the data, we discovered that the average height of an RBT when it holds 10,000 random values is about 16 and the average height of a BST when it holds 10,000 random values is about 31. Subtracting the average heights we can see that the average difference between the heights is 15 which shows that the RBT is shorter by a height of 15 on average when storing 10,000 random values. This makes sense because RBTs have the self-balancing property where they rotate based on the colour of the uncle, so the root does change to keep both sides of the tree balanced. On the other hand, it's does not have this feature so the height cannot be reduced after a value has been inserted. This is why RBTs have lower heights when they store the same amount of values as a BST.

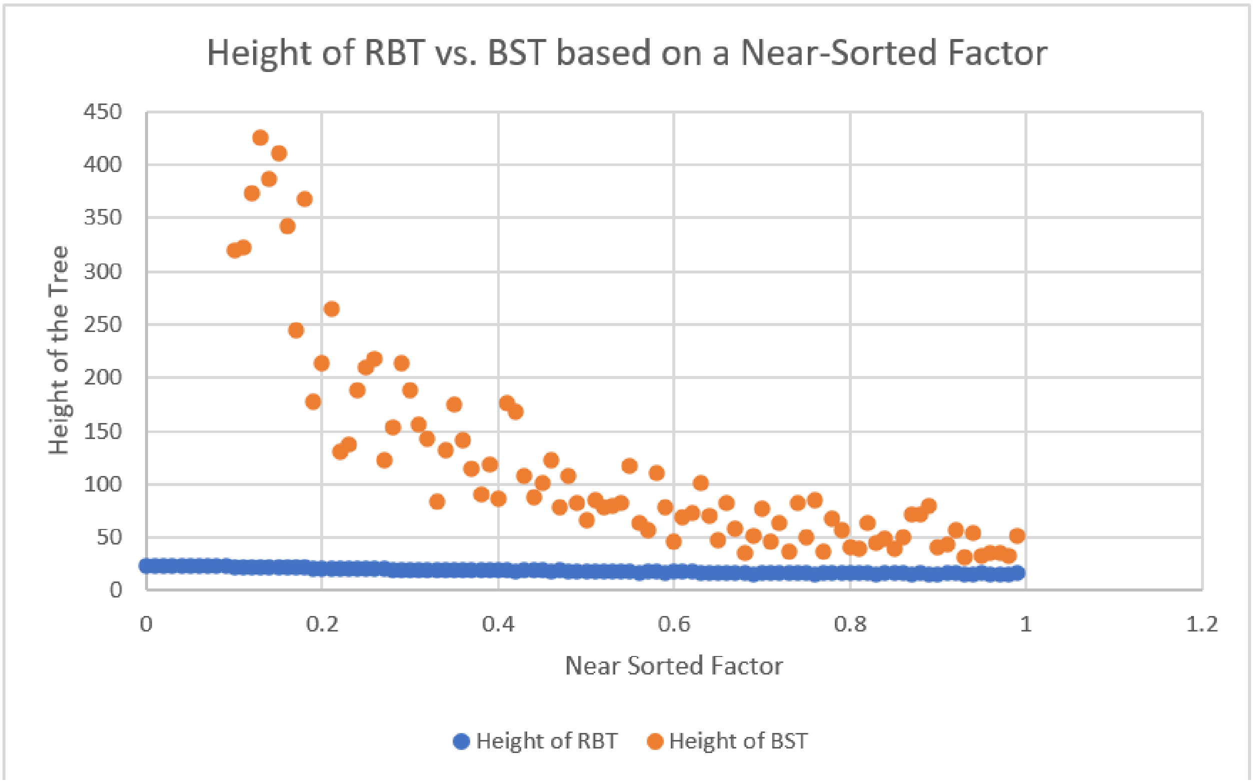

Height of RBT vs. BST based on the Length of the List
Height of the Tree
Length of List (n)
0
5
10
15
20
25
30
35
0
2000
4000
6000
8000
10000
12000
Height of RBT
Height of BST


Height of RBT vs. BST on Lists of Size 10,000
Height of Tree
n
0
5
10
15
20
25
30
35
40
45
0
20
40
60
80
100
120
Height of RBT
Height of BST

What is your instinct on the performance of a BST insert to a RBT insert for trees of similar heights?

When the length of the lists is small, the BST insert performs much faster. You can see this result in the first graph above where BST insert outperforms RBT insert for small lists where both inserts have similar heights. By researching we discovered that the insert function of a BST insert has an O(log n) average runtime and the RBT insert also has an average runtime of O(log n). However, we believe that the runtime of the RBT insert is faster than the insert of BST even though both of them are O(log n) because RBTs are more balanced than BSTs. This leads to a more efficient insert.

Consider near sorted lists based on a factor (like we have seen previously). For lists of length 10,000. Graph the height of a BST and RBT vs the near sorted factor of the list. What do you observe?



We observed from our graph that when comparing the height of an RBT vs. BST based on a near-sorted factor on lists of 10,000 was that RBT trees were much better. When the factor was really small, the height of the BST was very large whereas the height of the RBT was around 25. We noticed from our graph that the height of both RBT and BST improved as the near-sorted factor got larger which makes sense since there is a smaller chance that the values in the lists will be near sorted. For the graph of the RBT, we saw a small improvement in the height of the tree as the nearly sorted factor grew since the height of the tree starts at 24 when the factor is 0 and the height becomes 16 when the factor gets to values very close to 1. On the other hand, for the graph of the BST, we saw a large improvement in the height of the tree as the values started with being over 300 when the near-sorted factor was close to 0.1 and the height decreased to a point where the values were about 30-50 when the factor gets to values very close to 1. This shape of the graph makes sense because RBTs are self-balancing so whether the list is close to being sorted or not the RBT can still maintain a relatively small height. Whenever a value gets inserted into the tree, the tree checks whether there needs to be a colour change and based on the order of the colours the tree can then be rotated to balance both sides of the tree out. When the nearly sorted factor is larger, the RBT height doesn't vary too much since the RBT is self-balancing the root does change when a rotation occurs within the tree which shows that RBT does balance both sides of the tree. This shows that the nearly sorted factor does not change the height of the RBT when the length of the list is 10,000. Whereas the BST is not self-balancing so when we have a list that is near sorted the majority of the values get inserted into one side of the tree. This makes the height of the BST very large since there is no concept of rotation or colour changing in a BST. As the nearly sorted factor gets larger there is a larger chance for the values to be less than or greater than the root. There is a better chance that both sides of the tree will be balanced on sides causing the height of the tree to decrease when the nearly sorted factor is larger.