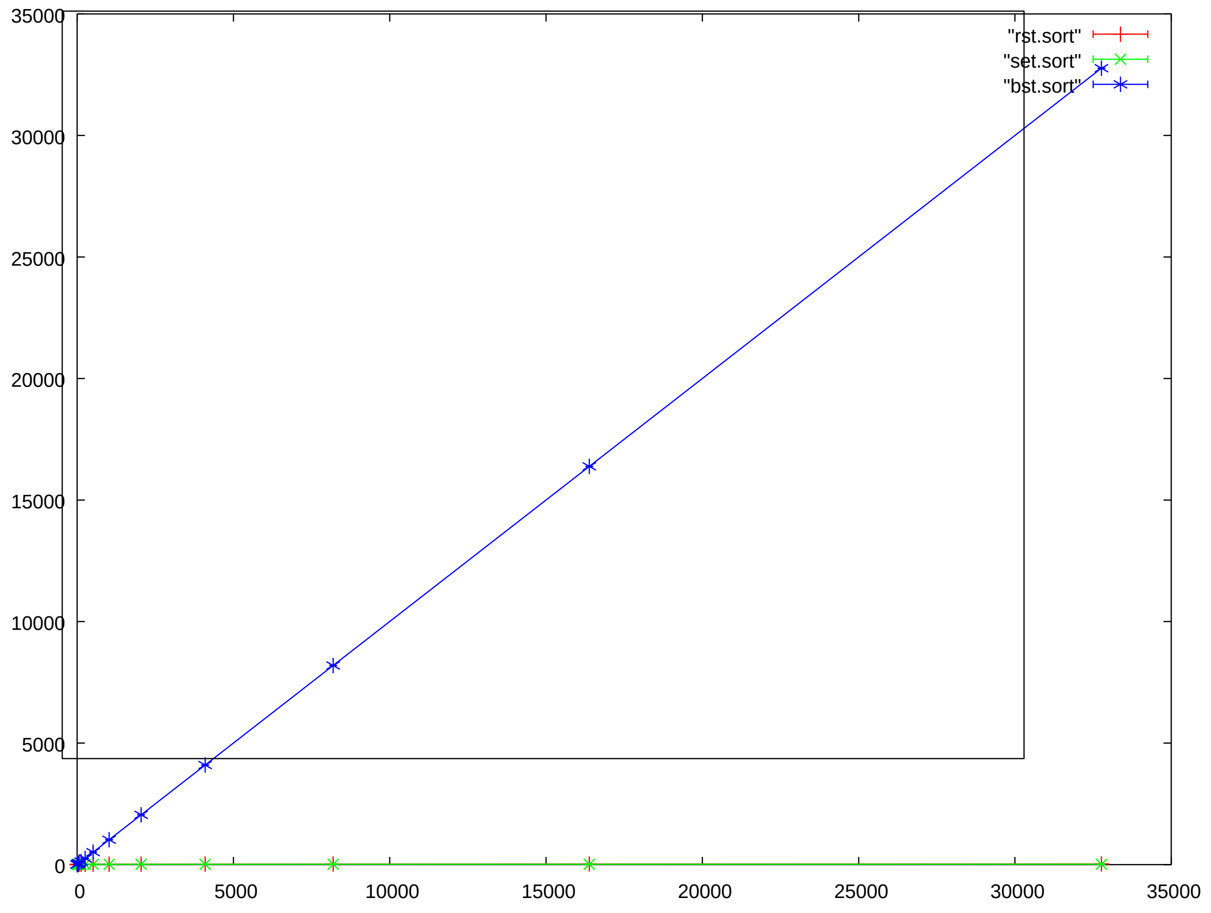
Benchtree.pdf
Dominic Kuang
Pa2
cs100wdy

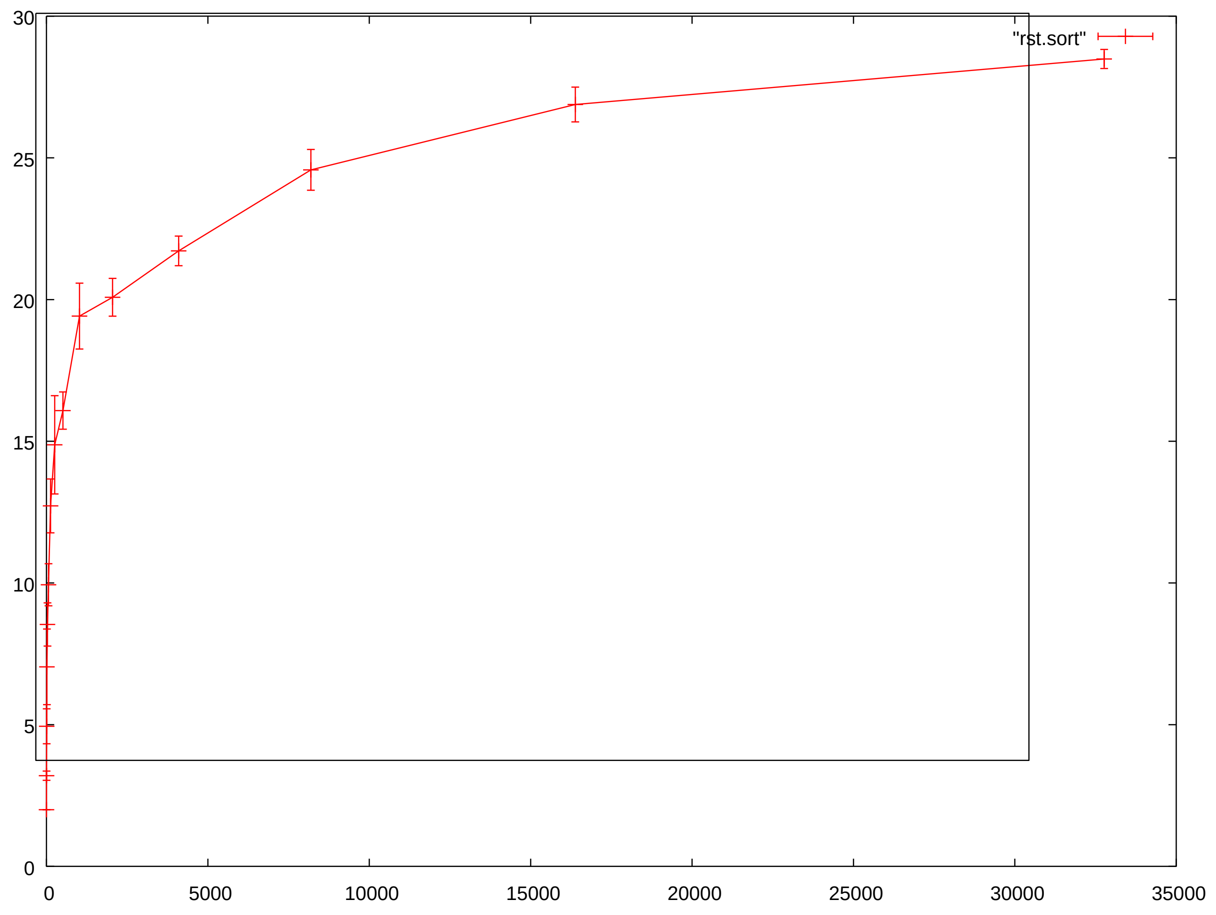
In a BST, the time to access an item is determined by the depth of the node in the tree. In a sorted BST, we can see a linear line due to the increasing amount of comparisons for insert and find as the tree's size increases. Therefore this pattern is consistent with O(N) in worst time. For a RST, the comparisons for insert and find is logarithmic since comparisons are cut in half as seen in Figure 2. The sorted RST does prove to have Olog(N). Furthermore, the difference between the RST and BST is apparent since the RST blends in with the x-axis in Figure 1 due to such small comparisons compared to the BST. The red black tree (SET) appeared to be most efficient; however, the RST gave comparable result on large data sets. They also have a O(log N) worst case performance similar to RST's. The RST does have more erratic behavior every time which can be seen in the standard deviation since the structure of the tree is different every time. The BST's and SET's had a constant 0 stand deviation due to their fixed structure. The red black tree has a fixed structure due to its efficient rotations and balanced nodes.

For a BST, the sorting performance would be O(N). By performing an inorder traversal throughout the BST, we can reach all the nodes efficiently. For a hash table, we could copy all of the values into an array and run a quicksort on the array. On average, this could take O(N log(N) + N). It would take O(N) to go down the array and copy each element into a new array and O(N log N) to do the quick sort on the new array. Sorting a hash table would defeat the purpose of the mapping hash function. The BST inorder traversal would be faster in terms of performance and time complexity. The BST inorder traversal would be a linear line. The quicksort would probably take a logarithmic form.

Please ignore the box on top of diagrams. I'm not sure why it comes out like that saved in pdfs.

Figure 1


"rst.sort"
"set.sort"
"bst.sort"
35000
30000
25000
20000
15000
10000
5000
0
0
5000
10000
15000
20000
25000
30000
35000



"rst.sort"
0
5
10
15
20
25
30
0
5000
10000
15000
20000
25000
30000
35000

Figure 2