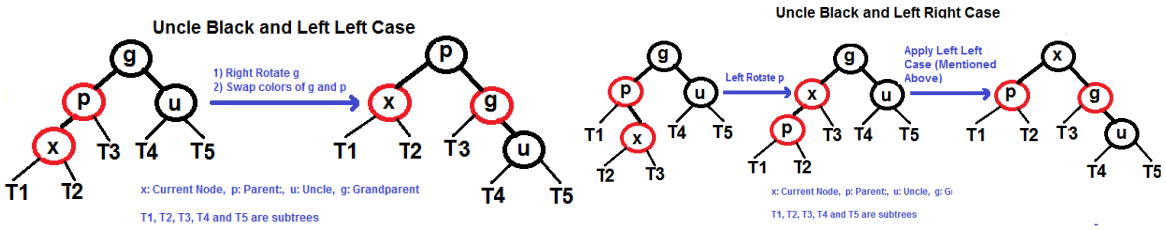
Hrytsyna Anastasiia-Solomiia
ID 105323

RED_BLACK TREE

(my implementation)

For implementing Red-Black tree I used standard methods: rotation and recoloring. Recoloring is the simple one: I just changed colors to the opposite ones when it was needed (except the root&nulls – they always stays black). For the rotation I used the next schemes:



…and two more mirror cases for the Right Right or Right Left cases. The height of a Red-Black tree is always logn where n is the number of nodes in the tree. As Red-Black tree is BST - it has the same properties as BST, so its' insert&search operations have O(logn), worst case is O(n).

AVL TREE

AVL tree is a BST, where the difference between heights of left and right subtrees cannot be more than one for all nodes. If this situation occurs – we have to rotate the tree, so it saves its' functionality. Scheme of the rotations is next:

```
T1, T2, T3 and T4 are subtrees.
        z                                      y
       / \                                   /   \
      y   T4      Right Rotate (z)          x      z
     / \          - - - - - - - - ->       /  \    /  \
    x   T3                                T1  T2  T3  T4
   / \
  T1   T2
```

```
     z                                z                                x
    / \                              /   \                            /  \
   y   T4  Left Rotate (y)          x    T4  Right Rotate(z)         y      z
  / \      - - - - - - - - ->      /  \      - - - - - - - - ->    / \    / \
T1   x                           y    T3                        T1  T2 T3  T4
    / \                         / \
   T2   T3                    T1   T2
```

…and two more mirror cases for the Right Right or Right Left cases. The same as RB tree, AVL tree is BST and owns all its' properties. The rotation routines are all themselves O(1), so they don't significantly impact on the insert operation complexity, which is still O(k) where k is the height of the tree. But as noted before, this height is log N, so insertion into an AVL tree has a worst case O(log N).

SOURCE OF THE CODE: https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)AvlTree.html

Hrytsyna Anastasiia-Solomiia
ID 105323

HASHING (OPEN ADDRESSING)

(my implementation)

In this implementation I used quadratic probing (as it has medium clustering & cache performance) and resizing methods of solving collisions. Probing function is next: (x^2+x)/2. Also, the size of the table has to be the power of 2 during the whole program runtime (so all of the future resized tables are always twice bigger than the previous one). Table will be resized when the amount of allocated space in the list is greater than 0,4*(whole size of the list). With this combination I tried to provide the best variant for inserting data and avoid looping of the index after probing.

Hash tables are O(1) average, however it suffers from O(n) worst case time complexity.

HASHING (CHAINING)

In compare to the previous method this one uses linked list of data. Chaining also uses resizing (it double the size of the table, just as in my implementation) for solving collisions, but there is one difference – resizing of the table starts when at least one of the lists' heights is greater than 5.

SOURCE OF THE CODE: https://github.com/sahuankita2203/hashing-separatechaining-in-C/blob/master/hashing_program.c

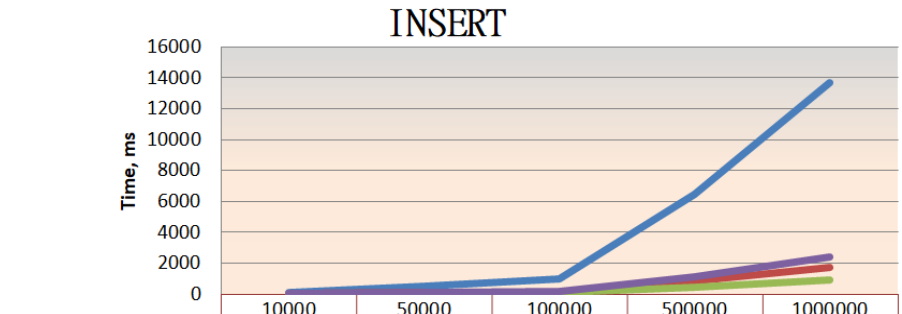
---

TESTS & COMPARISON

For testing my implementations I decided to make a review of two functions of mine (insert & search) and check how much time I need to perform each function.

That is why I tried to create & implement trees of different size (XS(10000 numbers), S(50000 numbers), M(100000 numbers), L(500000 numbers), XL(1000000 numbers)). As I wanted to better my results and get more accurate output, to each type of a tree/structure I made 10 insertions(searching) of the same sized equal arrays (but with different sequence of numbers) and then took an average time (measured at ms). Testing of the search function I made on the similar way: inserting different arrays and looking for elements from the one exact array (numbers). All of the numbers in arrays are random, without repeating.

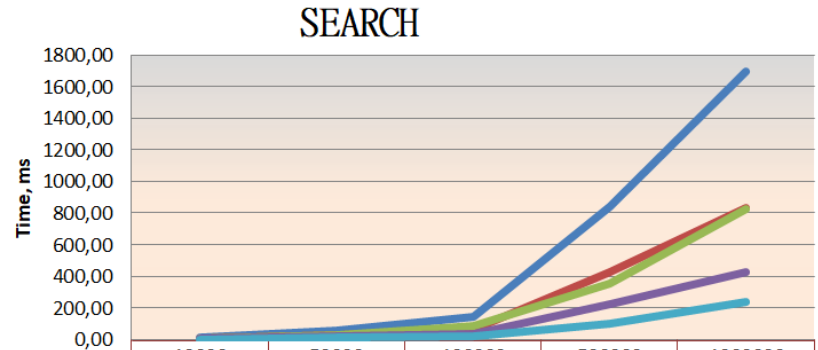
One version of my outputs (rounded) is shown under the graphs. Also, a few more notes to the graphs: the horizontal axis expresses the size of the insertion/searching, vertical – average time. Also, in the table there are rounded values of the function duration.



	10000	50000	100000	500000	1000000
AVL_TREE	94	511	1003	6445	13720
RED_BLACK_TREE	14	81	126	863	1720
HASH_CHAIN	9	47	86	470	925
HASH_ADR	19	86	169	1138	2399

Hrytsyna Anastasiia-Solomiia
ID 105323

As we can see from the graph, AVL tree need more time for the insertion big amount of data than Red-Black tree (so it is worse option). It is because AVL tree is more balanced one, so it need more time for its' rotations. The least time consuming is hashing(chaining) method. Hashing(open addressing) is a bit worse, as it demands more time for looking for a free space in a big array (skipping all bad variants), when in chaining method we simply add data to the list and resize table, when it is necessary.



	10000	50000	100000	500000	1000000
AVL_TREE_RECUR	10,90	56,60	146,00	837,90	1694,80
AVL_TREE_NORM	3,60	30,10	61,30	425,80	833,10
RED_BLACK_TREE	2,40	30,50	82,30	357,00	825,50
HASH_CHAIN	2,00	19,50	32,70	225,90	423,20
HASH_ADR	1,10	11,90	17,50	100,30	236,20

On the other hand, search function performs faster in addressing method rather than in chaining. Probably it is because of lower clustering in quadratic probing, so we don't have to check a big amount of data in the array/linked list.

Also, I noticed that search function in AVL tree is slower that in the Red-Black tree. Usually it doesn't have to be like that (as AVL tree is more balanced, so it's easier to find the necessary information there). The only explanation that I found for this situation is that we use recursive function for searching, which complicates the situation. That is why I decided to add one more function (search_AVL), where I perform searching without recursion (my result is the red line on the graph). We can notice that it is much better, as it need less time than the old one. Unfortunately, it is still slower than searching in Red-Black tree, as in the structure was used array child[ ] with the pointers, not the exact pointers to its' children. That is why we waste time for getting to the array data and after that to the child we want. As it wasn't my implementation I can't change the structure, so I can't improve the search function in AVL more. That is why Red Black tree 'won' in both cases.

P.S. I didn't change nothing in not-mine implementations, just commented unnecessary functions in the program.