NAME: SALEHIN RAHMAN KHAN

ID : 110016396

Acknowledgement: I confirm that I will keep the content of this assignment confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this assignment. I acknowledge that a mark of 0 may be assigned for copied work."

Within a Java class, write a method that creates n random strings of length 10 and inserts them in a hash table. The method should compute the average time for each insertion.

Location For Code: Assignment1\src

Java File Name: CuckooHashTable.Java, GenerateStrings.java, test.java, QuadraticProbingHashTable.java, SeparateChainingHashTable.java

Method Names:

- task1();
- task11();
- task111();
- getAlphaNumericString();
- insert();

Reference Of The Code: From Advance Computing Lab Sessions.

2. Write another method that finds n random strings in the hash table. The method should delete the string if found. It should also compute the average time of each search.

Location For Code: Assignment1\src

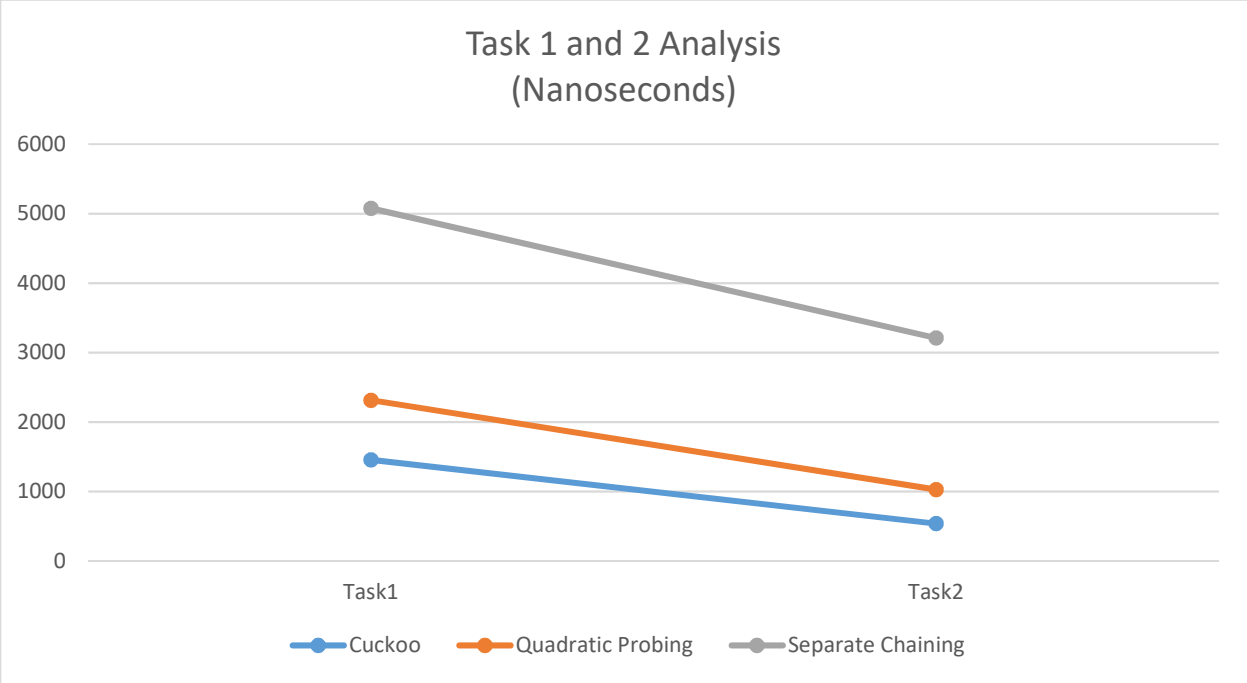
Java File Name: CuckooHashTable.Java, GenerateStrings.java, test.java, QuadraticProbingHashTable.java, SeparateChainingHashTable.java

Method Names:

- task2();
- task3();
- task22();
- task33();
- task222();
- task333();
- getAlphaNumericString();
- contains();
- remove();

Reference Of The Code: From Advance Computing Lab Sessions.

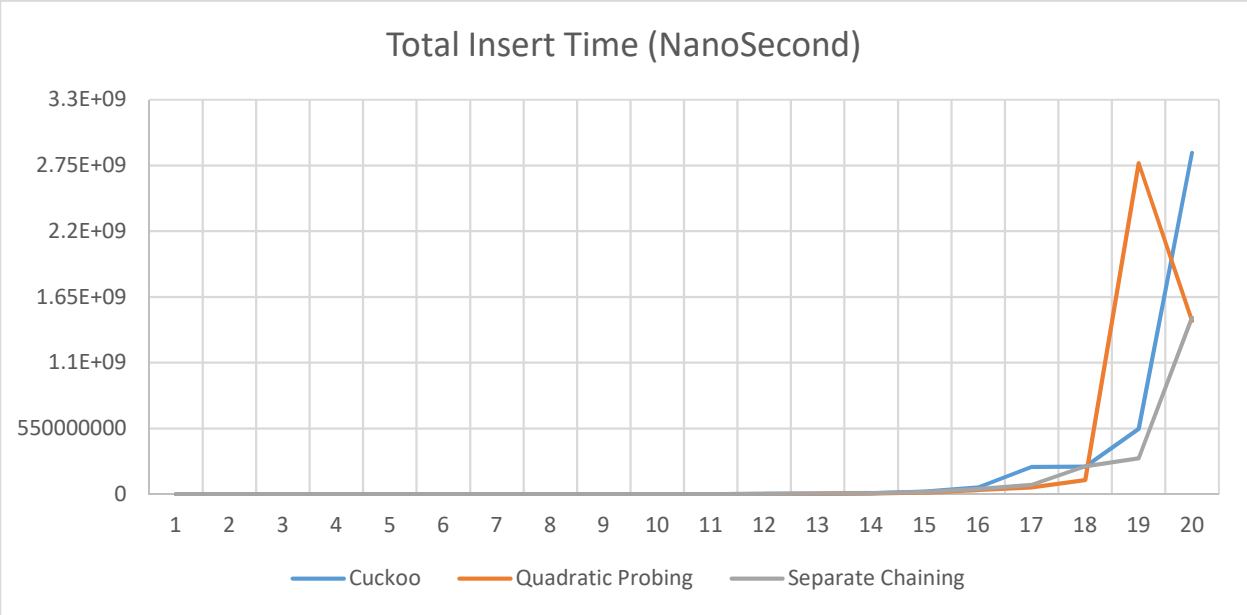
	Cuckoo (Nanoseconds)	Quadratic Probing (Nanoseconds)	Separate Chaining (Nanoseconds)
Task1	1457	858	2762
Task2	538.942	490.8997	2180.986



3. Repeat #1 and #2 with n = 2i , i = 1, …, 20. Place the numbers in a table and compare the results for Cuckoo, QuadraticProbing and SeparateChaining. Comment on the times obtained and compare them with the complexities as discussed in class

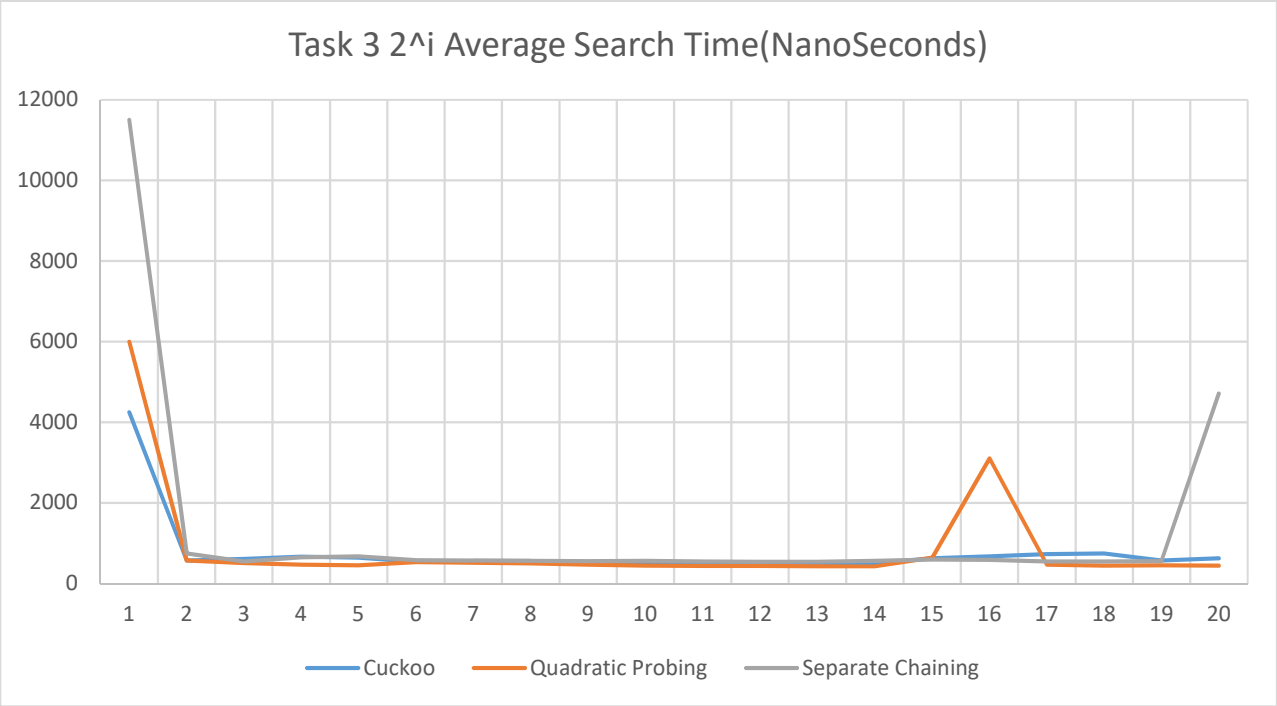
	Cuckoo (NanoSecond)	Quadratic Probing (NanoSecond)	Separate Chaining (NanoSecond)	Cuckoo Average Time (NanoSecond)	Quadratic Probing Average Time (NanoSecond)	Separate Chaining Average Time (NanoSecond)
2^1	249200	71800	31400	0.1246	0.0359	0.0157
2^2	32300	20200	12600	0.01615	0.0101	0.0063
2^3	29200	24300	8900	0.0146	0.01215	0.00445
2^4	61200	22000	18100	0.0306	0.011	0.00905
2^5	96600	30000	27300	0.0483	0.015	0.01365
2^6	66100	43400	54100	0.03305	0.0217	0.02705
2^7	103200	114700	79400	0.0516	0.05735	0.0397
2^8	166100	162000	152900	0.08305	0.081	0.07645
2^9	313100	309500	293700	0.15655	0.15475	0.14685
2^10	581000	523000	589100	0.2905	0.2615	0.29455
2^11	1126300	908400	1159400	0.56315	0.4542	0.5797
2^12	2165800	1851600	2281300	1.0829	0.9258	1.14065

2^13	4371400	3588100	4619500	2.1857	1.79405	2.30975
2^14	8842800	7003300	8839600	4.4214	3.50165	4.4198
2^15	22916800	14349000	18687200	11.4584	7.1745	9.3436
2^16	55774400	35048300	42098500	27.8872	17.52415	21.04925
2^17	2.27E+08	57091100	78486300	113.7149	28.54555	39.24315
2^18	2.32E+08	1.19E+08	2.33E+08	116.0317	59.4154	116.3486
2^19	5.46E+08	2.77E+09	3E+08	272.9081	1384.974	150.0215
2^20	2.86E+09	1.45E+09	1.48E+09	1427.915	724.2656	738.7748



	Cuckoo average (NanoSeconds)	Quadratic Probing average (NanoSeconds)	Separate Chaining average (NanoSeconds)
2^1	4250	6000	11500
2^2	575	575	750
2^3	612.5	512.5	550
2^4	668.75	468.75	650
2^5	643.75	450	678.125
2^6	543.75	535.9375	578.125
2^7	550	518.75	573.4375
2^8	547.2656	500.7813	564.0625
2^9	529.1016	472.0703	556.8359
2^10	529.0039	449.5117	562.8906
2^11	511.7188	436.4746	549.707
2^12	517.0898	435.1318	543.6523
2^13	515.7959	427.6489	543.4937
2^14	499.6887	426.7517	562.0544
2^15	631.0333	640.9515	594.5282
2^16	677.3743	3105.957	587.5916
2^17	729.8264	470.3491	549.1554
2^18	750.1427	448.9258	551.1711

2^19	569.1717	452.9987	560.4155
2^20	632.04	444.3118	4717.208



Analysis: On comparing the 3 algorithms (Cuckoo hasing, Quadratic Probing, Separate Chaining) overall, cuckoo hashing takes the least amount of time to insert the random strings in the hash table. So, cuckoo hashing has the best performance. the Time Complexity for this algorithm is big O (n).

4. Use the Java classes BinarySearchTree, AVLTree, RedBlackBST, SplayTree given in class. For each tree: a. Insert 100,000 integer keys, from 1 to 100,000 (in that order). Find the average time for each insertion. Note: you can add the following VM arguments to your project: -Xss16m. This will help increase the size of the recursion stack. b. Do 100,000 searches of random integer keys between 1 and 100,000. Find the average time of each search. c. Delete all the keys in the trees, starting from 100,000 down to 1 (in that order). Find the average time of each deletion.

Location For Code: \lab01\src\lab01

Java File Name: AVLTree.Java, GenerateInt.java, testB.java, BinarySearchTree.java, RedBlackBST.java, RedBlackBST.java, SplayTree.java

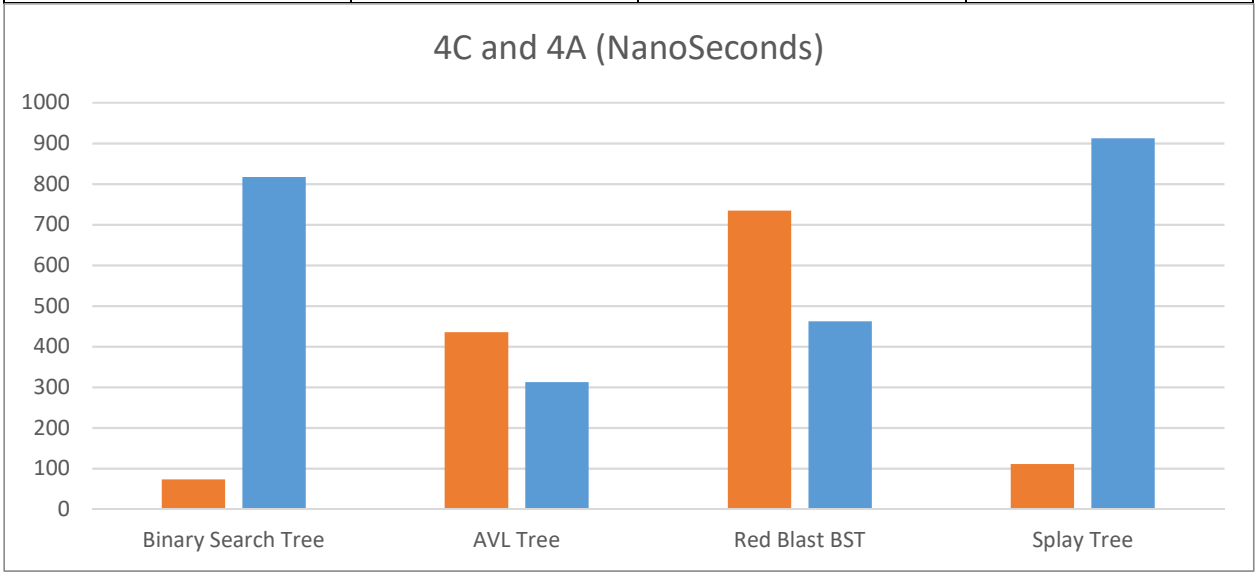
Method Names:

- task1(); //4A
- task2(); //4B

- task3(); //4C
- task11(); //4A
- task22(); //4B
- task03(); //4C
- task111();//4A
- task222(); //4B
- task033(); //4C
- task1111(); //4A
- task2222(); //4B
- task03333(); //4C
- put();
- insert();
- contain;
- get();
- remove();
- delete();

Reference Of The Code: From Advance Computing Lab Sessions.

Program Name/ Problem Number	4A (NanoSeconds)	4B (NanoSeconds)	4C (NanoSeconds)
Binary Search Tree	817	765657.3	73
AVL Tree	312	354393.7	435
Red Blast BST	462	328157.8	734
Splay Tree	912	323890.9	111



5. For each tree: a. Insert 100,000 keys between 1 and 100,000. Find the average time of each insertion. b. Repeat #4.b. c. Repeat #4.c but with random keys between 1 and 100,000. Note that not all the keys may be found in the tree.

Location For Code: \lab01\src\lab01

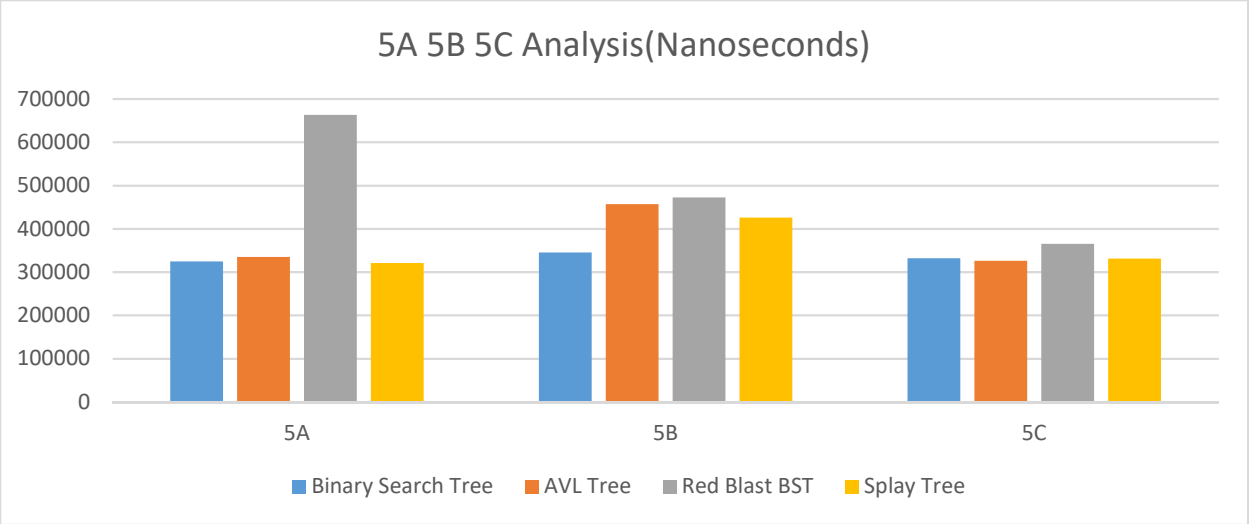
Java File Name: AVLTree.Java, GenerateInt.java, testB.java, BinarySearchTree.java, RedBlackBST.java, RedBlackBST.java, SplayTree.java

Method Names:

- task01(); //5A
- task2(); //5B
- task4(); //5C
- task011(); //5A
- task22(); //5B
- task33(); //5C
- task0111(); //5A
- task222(); //5B
- task333(); //5C
- task01111(); //5A
- task2222(); //5B
- task3333(); //5C
- put();
- insert();
- contain;
- get();
- remove();
- delete();

Reference Of The Code: From Advance Computing Lab Sessions.

Program Name/ Problem Number	5A (Nanoseconds)	5B (Nanoseconds)	5C (Nanoseconds)
Binary Search Tree	324239	345568.8	332061.9
AVL Tree	334684	457093.6	325963
Red Blast BST	663235	472728.2	365030.4
Splay Tree	320968	426079.8	331364.3



Analysis: For task 4C binary search tree did the best among all and for task 4b AVL tree did the best. In general, time complexity is O(h) where h is height of BST. searching in binary search tree has worst case complexity of O(n). An AVL tree is balanced, so its height is O(log N) where N is the number of nodes. For task 5A SplayTree did the best. For 5B Binary search tree did the best. For task 5C AVL tree did the best. SplayTree performs basic operations such as insertion, look-up and removal in O(log n) amortized time.