JAYADITYA KAKARALA

ID: 110026622

"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." JAYADITYA KAKARALA, 110026622

1. **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 of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java files: test.java, CuckooHashTable.java, QuadraticProbingHashTable.java, SeperateChaining.java
   Method Name: randomStringgenerator (long l, int m);
   Variables used: long avgTime, sumTime, startTime, endTime;
   int size, i, j, a;
   String generatedString;
   Methods used: insert();

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 of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java files: test.java, CuckooHashTable.java, QuadraticProbingHashTable.java, SeperateChaining.java
   Method Name: randomStringgenerator2 (long l, int m);
   Variables used: long avgTime, sumTime, startTime, endTime;
   int size, i, j, a;
   String generatedString1;
   Methods used: contains(), remove();

3. **Repeat #1 and #2 with n = 2^i, 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.**
   Location of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java files: test.java, CuckooHashTable.java, QuadraticProbingHashTable.java, SeperateChaining.java
   Method Name: randomStringgenerator3(int m);
   Variables used: int n, i;
   Methods used: randomStringgenertor(), randomStringgenerator2();

On Observing Table1.1, Table 1.2, Graph 1.1 and Graph1.2, the time determined in nanoseconds for insertion and search of strings on all the hashing methods, for example, Cuckoo Hashing, Quadratic Probing, and Separate Chaining, we can undoubtedly say that Quadratic Probing sets aside less effort for both insertion and search of the generated strings.

Table 1.1: Average time taken for INSERTION of n number of strings

Number of Strings	Cuckoo Hashing	Quadratic Probing	Separate Chaining
2^1	15110	47200	47500
2^2	6100	1200	1275
2^3	3462	1725	1087
2^4	3631	593	706
2^5	5500	1390	659
2^6	5753	2164	9181
2^7	2983	2566	3383
2^8	1657	1148	6071
2^9	3276	856	2956
2^{10}	1839	1062	2068
2^{11}	818	1063	1319
2^{12}	1722	1121	1413
2^{13}	1201	860	1394
2^{14}	1259	795	960
2^{15}	609	597	652
2^{16}	1265	569	1613
2^{17}	1821	963	1103
2^{18}	2335	1055	1345
2^{19}	1417	909	1696
2^{20}	2596	1377	1727

Graph 1.1: Average Time of Inserting Strings into the Hash table

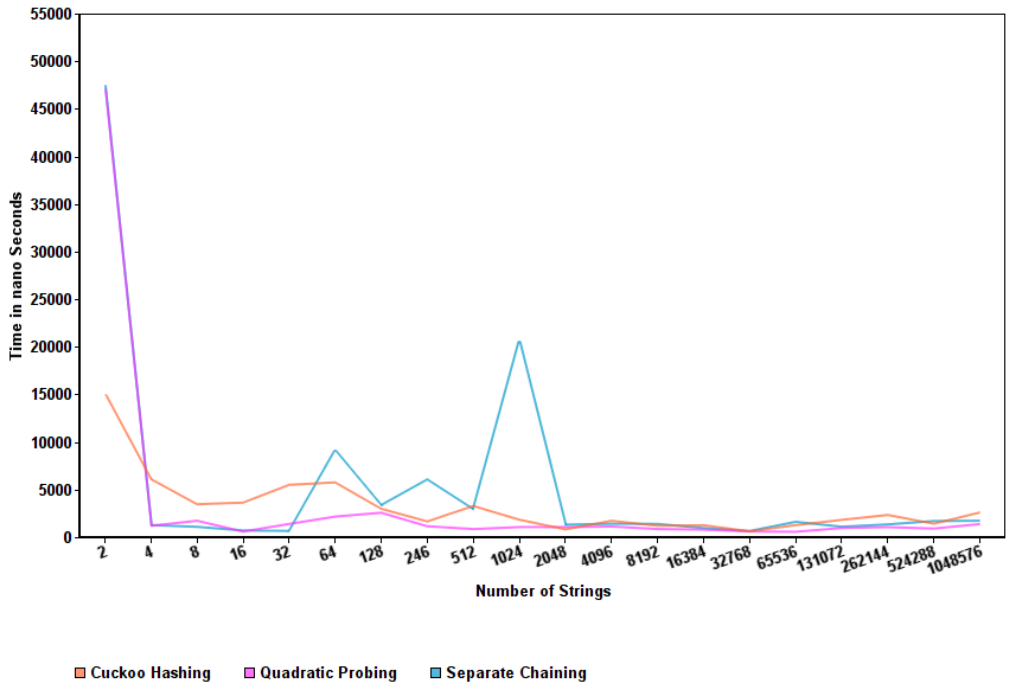
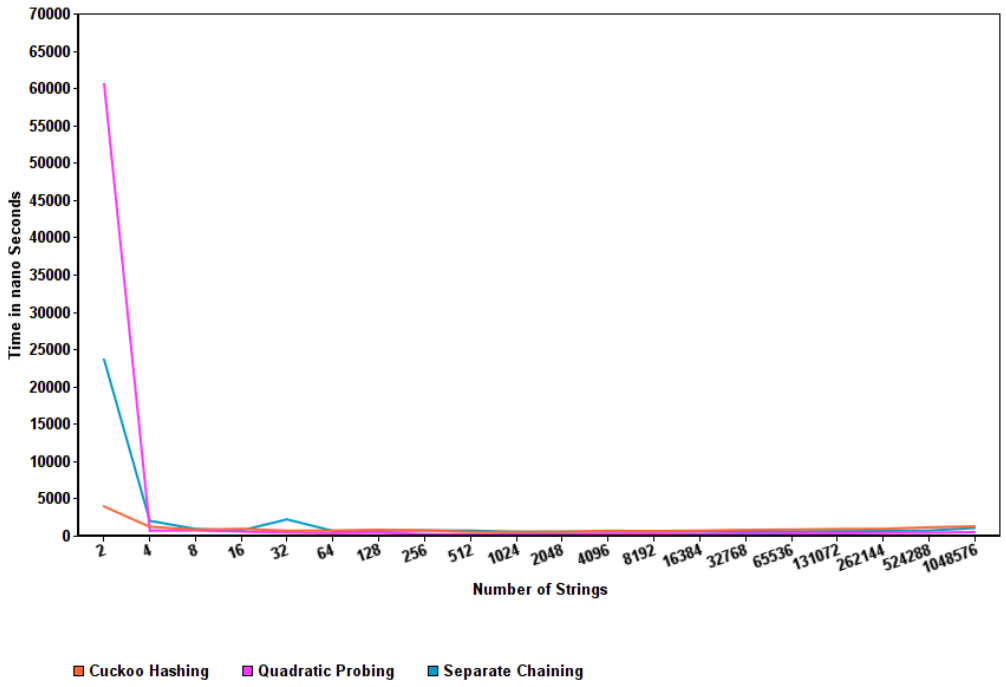


Table 1.2: Average time taken for SEARCH of n number of strings

Number of Strings	Cuckoo Hashing	Quadratic Probing	Separate Chaining
2^1	3950	60700	23700
2^2	1200	675	2000
2^3	800	700	912
2^4	918	550	693
2^5	675	421	2181
2^6	687	342	645
2^7	796	391	653
2^8	717	216	707
2^9	516	187	678
2^{10}	514	205	525
2^{11}	524	205	535
2^{12}	657	269	556
2^{13}	624	280	549
2^{14}	679	260	587
2^{15}	781	297	522
2^{16}	826	287	546
2^{17}	894	306	604
2^{18}	915	368	639
2^{19}	1124	397	685
2^{20}	1257	489	1033

Graph 1.2: Avg Time of Searching Strings in the Hash Table



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.

   Location of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java files: test.java, BinarySearchTree.java, AVLTree.java, RedBlackBST.java, SplayTree.java
   Method Name: insertkey (int m, int z);
   Variables used: long sumTime, avgTime, startTime, endTime;
   int max, min, x, i;
   Methods used: insert(), put();

   Note: Since, **Recursive method** is taking more time in BinarySearchTree, I used 2nd method which is **Non-Recursive method** in the BST Source code. By using this method only, all the data collected and noted in BinarySearchTree Column of tables generated below.

   b. Do 100,000 searches of random integer keys between 1 and 100,000. Find the average time of each search.

   Location of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java file name: test.java, BinarySearchTree.java, AVLTree.java, RedBlackBST.java, SplayTree.java
   Method Name: searchkey (int m);
   Variables used: long sumTime, avgTime, startTime, endTime;
   int max, min, x, i;
   Methods used: contains(), get();

   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 of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java files: test.java, BinarySearchTree.java, AVLTree.java, RedBlackBST.java, SplayTree.java
   Method Name: deletekey (int m, int z);
   Variables used: long sumTime, avgTime, startTime, endTime;
   int max, min, x, i;
   Methods used: remove(), delete();

5. **For each tree:**

   a. Insert 100,000 keys between 1 and 100,000. Find the average time of each insertion.

   Location of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java files: test.java, BinarySearchTree.java, AVLTree.java, RedBlackBST.java, SplayTree.java
   Method Name: insertkey (int m, int z);
   Variables used: long sumTime, avgTime, startTime, endTime;
   int max, min, x, i;
   Methods used: insert(), put();

b. Repeat 4.b.

Location of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
Reference of the code: Advanced Computing Concepts Lab Source Code
Java files: test.java, BinarySearchTree.java, AVLTree.java, RedBlackBST.java, SplayTree.java
Method Name: serachkey (int m);
Variables used: long sumTime, avgTime, startTime, endTime;
int max, min, x, i;
Methods used: contains(), get();

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.

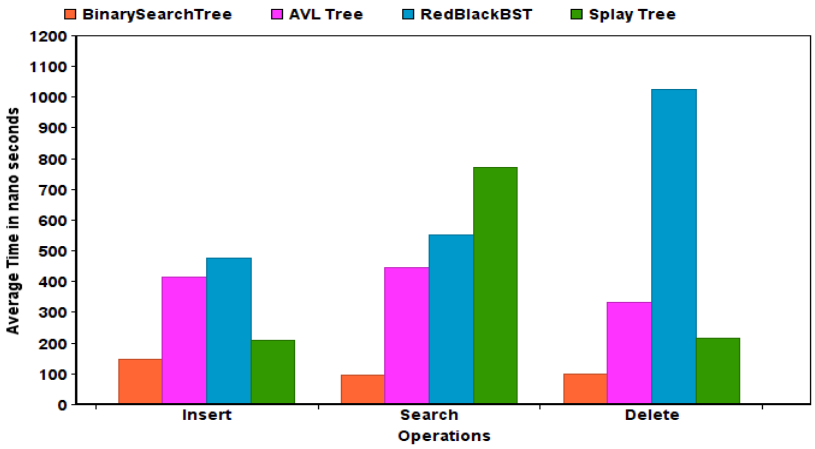
Location of the code: C:\Users\jayad\eclipse-workspace\Assignment1_110026622\src
Reference of the code: Advanced Computing Concepts Lab Source Code
Java files: test.java, BinarySearchTree.java, AVLTree.java, RedBlackBST.java, SplayTree.java
Method Name: deletekey (int m, int z);
Variables used: long sumTime, avgTime, startTime, endTime;
int max, min, x, i;
Methods used: remove(), delete();

6. **Draw a table that contains all the average times found in #4 and #5. Comment on the results obtained and compare them with the worst-case and average-case running times of each operation for each tree. Which tree will you use in your implementations for real problems? Note: you decide on the format of the table (use your creativity to present the results in the best possible way).**

Table 2.1: Average Time taken to Insert, Search and Delete the strings from 1 to 100000 in ORDER

Operations	BinarySearchTree	AVL Tree	RedBlackBST	Splay Tree
Insert	147	414	475	208
Search	95	447	552	770
Delete	98	334	1026	217

Graph 2.1: Average Time taken to Insert,Search and Delete the strings from 1 to 100000 in ORDER



On Observing the above Table 2.1 and Graph 2.1, we can clearly say that BinarySearchTree is faster than the AVLTree, RedBlackBST and SplayTree.

Comparison of Time taken by Trees on performing **INSERT** Operation in ORDER:
BinarySearchTree < SplayTree < AVLTree < RedBlackBST

Comparison of Time taken by Trees on performing **SEARCH** Operation in ORDER:
BinarySearchTree < AVLTree < RedBlackBST < SplayTree

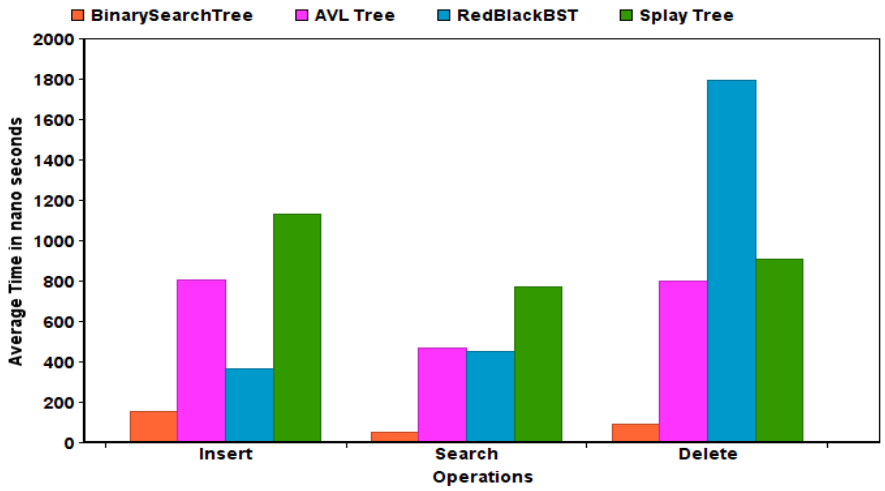
Comparison of Time taken by Trees on performing **DELETE** Operation in ORDER:
BinarySearchTree < SplayTree < AVLTree < RedBlackBST

BinarySearchTree was the best Tree to perform any Operation in less amount of time.

Table 2.2: Average Time taken to Insert, Search and Delete the strings from 1 to 100000 in RANDOM

Operation	BinarySearchTree	AVL Tree	RedBlackBST	SplayTree
Insert	152	808	366	1133
Search	51	467	454	774
Delete	94	800	1795	909

Graph 2.2: Average Time taken to Insert,Search and Delete the strings from 1 to 100000 in RANDOM



On Observing the above Table 2.2 and Graph 2.2, we can clearly say that BinarySearchTree is faster than the AVLTree, RedBlackBST and SplayTree. Average-case for insert, search and delete for all the trees BST, AVL, RedBlackBST and Splay Trees are same i.e. O(log n) where as log n denotes the depth or height of the tree. Worst-case for insert, search or delete for the BinarySearchTree is 0(n).

Comparison of Time taken by Trees on performing **INSERT** Operation in RANDOM:
BinarySearchTree < RedBlackBST < AVLTree < SplayTree

Comparison of Time taken by Trees on performing **SEARCH** Operation in RANDOM:
BinarySearchTree < RedBlackBST < AVLTree < SplayTree

Comparison of Time taken by Trees on performing **DELETE** Operation in RANDOM:
BinarySearchTree < AVLTree < SplayTree < RedBlackBST

Since, BinarySearchTree was the best tree to perform any Operation like insert, search or delete in an average-case i.e. less amount of time. So, I prefer better to implement **BinarySearchTree** than any other trees for the real problems.