**60-654 Advanced Computing Concepts**

**Winter 2020 Lab**

**Assignment 1**

**“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.”**

**Name : Vishal Solanki**

**Student ID : 110060480**

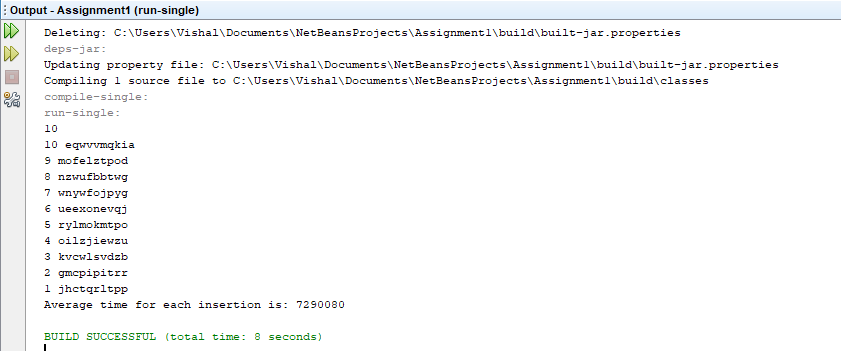
**Tasks**

**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.**

The code is in the package Assignment1 and class Task1.java.

The Program requires input from user to decide the number of random Strings to be inserted in the hash table and calculates the nano seconds required for the operation.

7290080ns

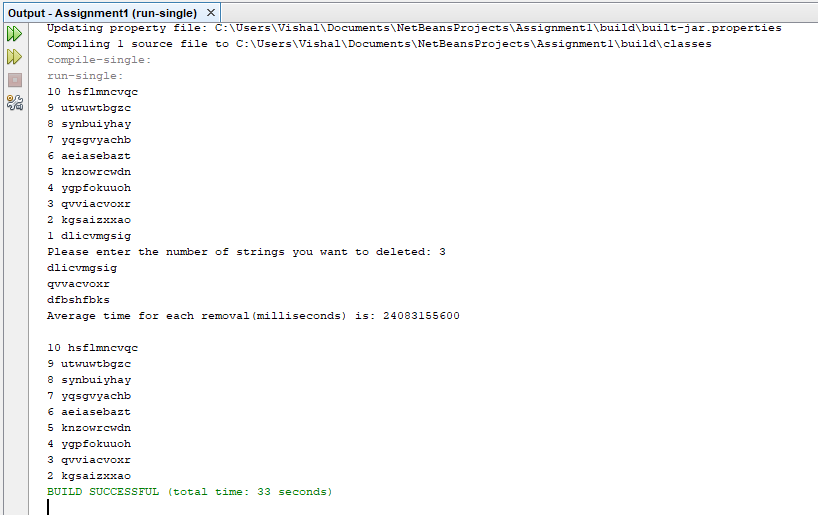


**2. Write another method that finds n random strings in the hash table. The method should delete the string if found. It should also compute the average time of each search.**

The code is in the package Assignment1 and class Task2.java.

The program takes n random Strings from user and searches each of them in the HashTable. Once found it will delete the entry and calculate the Average time for the operations.

24083155600ns

.

**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.**

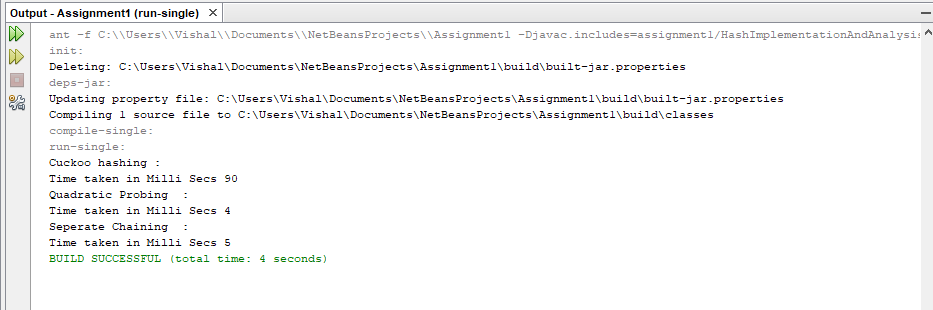
The code is in the package Assignment1 and class HashImplementationAndAnalysis.java**.**

The program calls functions from 3 different classes namingly CuckooHashingOperations, QuadraticProbingOperations and SeperateChainingOperations which calls operateOnHashingFunctions from claases provided in the resources. Hence impletmenting the code reuse Approach and calculating the average time for each of the insertion while doing so.

This task was repeated with n = 2^i, i = 1, …, 20. The outputs were recorded into the excel sheet : Time Report.xlsx

For Instance the below row is for n=2^1 and the time is recorded in miliseconds:

**Cuckoo**- 69ms , **QuadraticProbing**- 85ms, **SeperateChaining**-84ms



1. **Complexities of a Hash Table/Map with Cuckoo Hashing**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Worst-Case Time Complexity | Average-Case Time Complexity | Best-Case Time Complexity |
| Find | O(1) | O(1) | O(1) |
| Insert | O(n) | O(1) | O(1) |
| Remove | O(1) | O(1) | O(1) |

O(n) — Hash Tables typically have a capacity that is at most some constant multiplied by n (the constant is predetermined)

1. **Complexities of a Hash Table/Map with Separate Chaining**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Worst-Case Time Complexity | Average-Case Time Complexity | Best-Case Time Complexity |
| Find | O(n) | O(1) | O(1) |
| Insert | O(n) | O(1) | O(1) |
| Remove | O(n) | O(1) | O(1) |

O(n) — Hash Tables typically have a capacity that is at most some constant multiplied by n (the constant is predetermined), and each of our n nodes occupies O(1) space

1. **Complexities of a Hash Table/Map with Quadratic Probing**

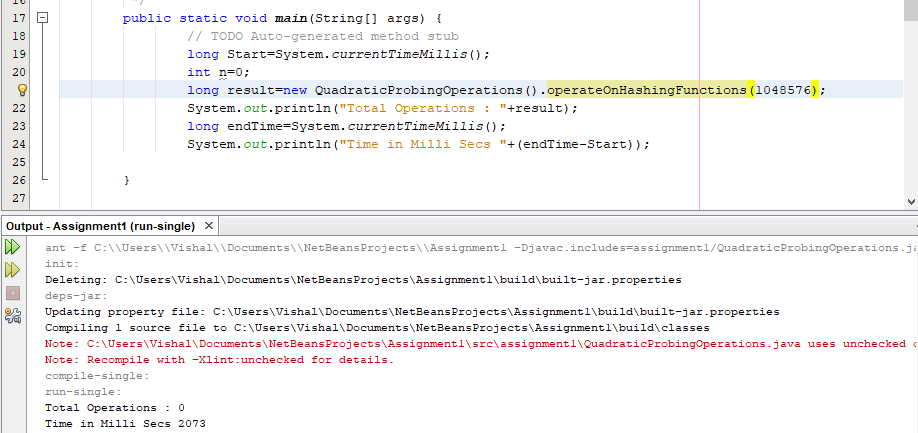
|  |  |  |  |
| --- | --- | --- | --- |
|  | Worst-Case Time Complexity | Average-Case Time Complexity | Best-Case Time Complexity |
| Find | O(1) | O(1) | O(1) |
| Insert | O(nlogn) | O(1) | O(1) |
| Remove | O(1) | O(1) | O(1) |

Complexity is O(nlogn)— Hash Tables typically have a capacity that is at most some constant multiplied by n (the constant is predetermined)

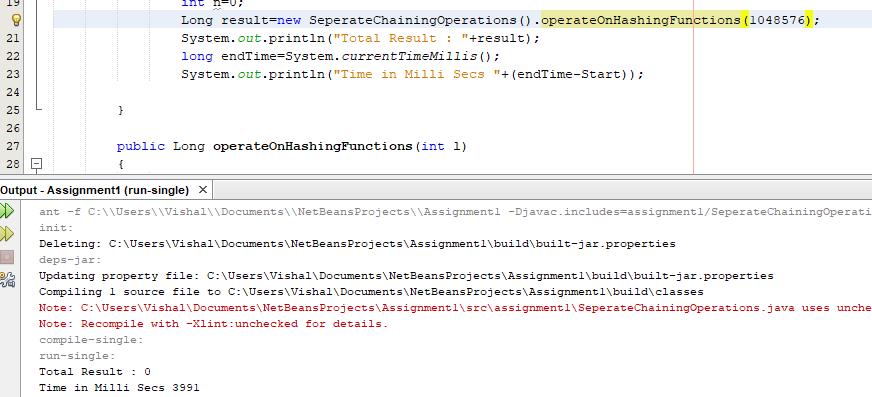
CUCKOO



QUADRATINC PROBING



SEPARATE CHAINING



Cuckoo Hashing, in worst case scenario, for every insert there can be n number of locations are looked up before any insertion. Hence, the Time Complexity for this algorithm is big O (n). But the probability of occurrence of this situation is least as compared to other algorithms.

**4. Use the Java classes BinarySearchTree, AVLTree, RedBlackBST, SplayTree given in class. For each tree:**

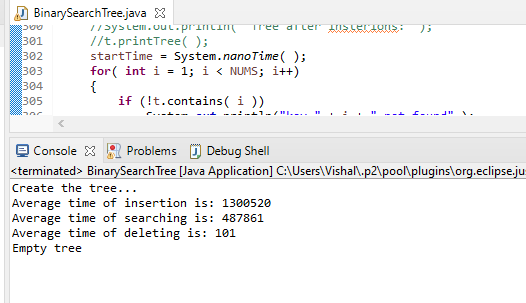
**a. Insert 100,000 integer keys, from 1 to 100,000 (in that order). Find the average timefor each insertion. Note: you can add the following VM arguments to your project: -Xss16m. This will help increase the size of the recursion stack.**

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

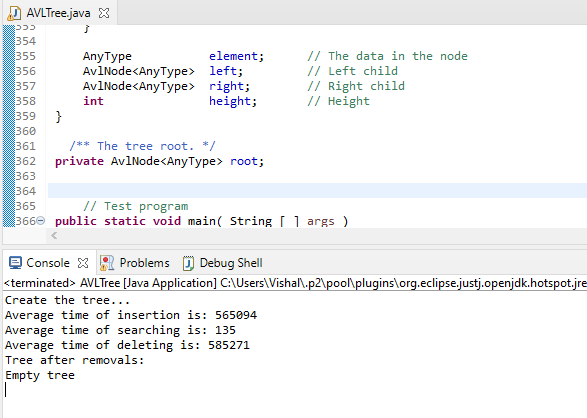
**c. Delete all the keys in the trees, starting from 100,000 down to 1 (in that order). Find the average time of each deletion.**

The code is in the package Assignment1 and classes BinarySearchTree.java, AVLTree.java, RedBlackBST.java and SplayTree.java**.**

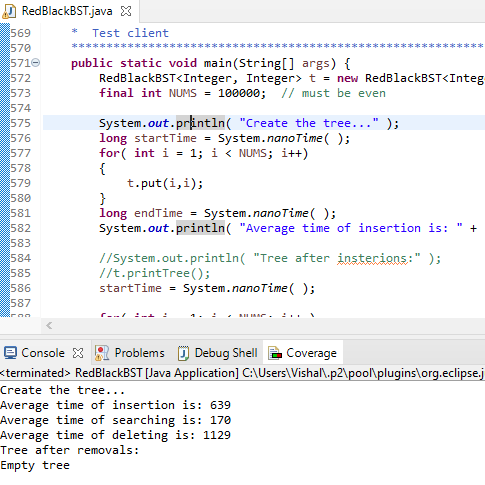
**\*Binary Search Tree**



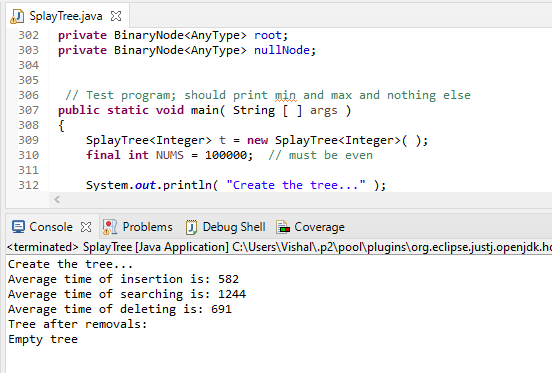
**\*AVL Tree**



**\*RedBlackBST**



**\*SplayTree**



**5. For each tree:**

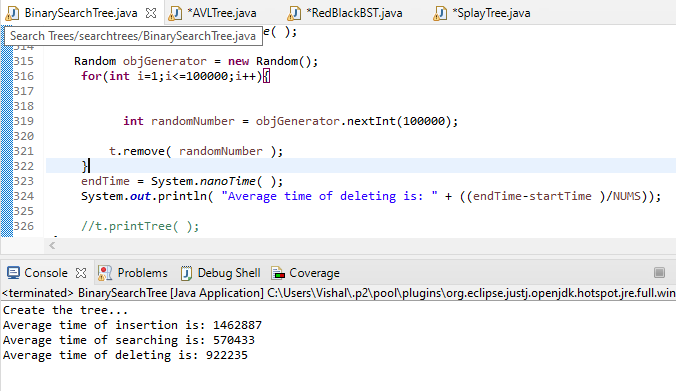
**a. Insert 100,000 keys between 1 and 100,000. Find the average time of each insertion.**

**b. Repeat #4.b.**

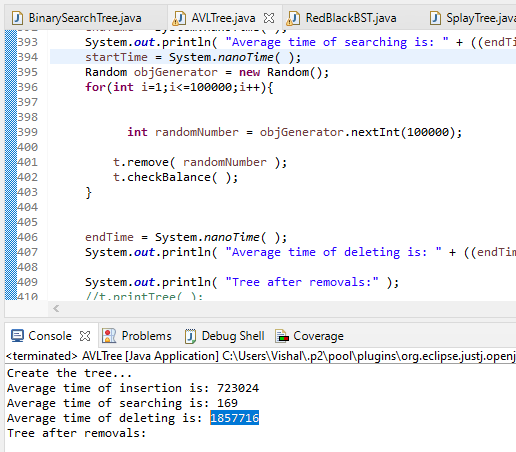
**c. Repeat #4.c but with random keys between 1 and 100,000. Note that not all the keys may be found in the tree.**

The code is in the package Assignment1 and classes BinarySearchTree2.java, AVLTree2.java, RedBlackBST2.java and SplayTree2.java**.**

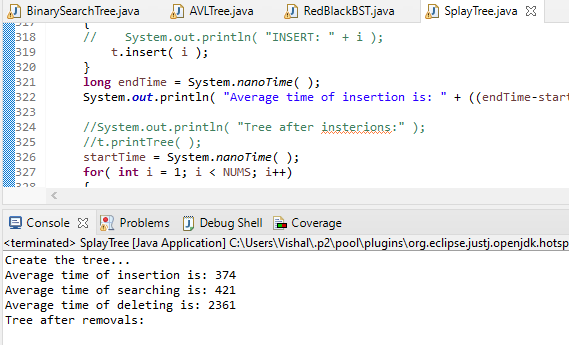
**BST**



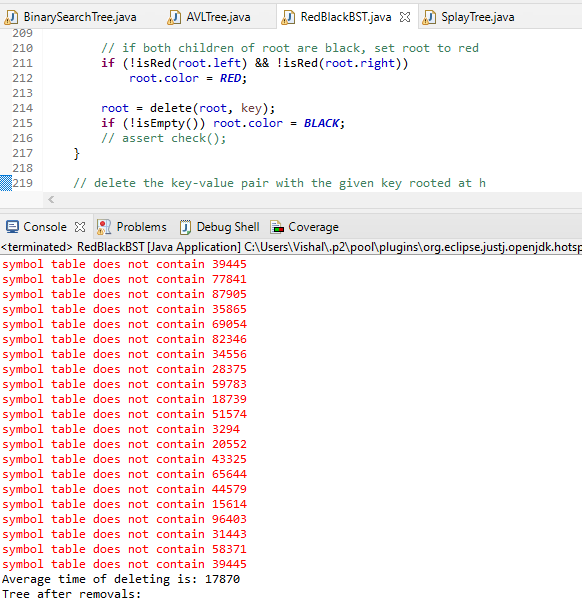
**AVL**



**Splay Tree**



**RedBlackBST**



**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?**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Search Trees(ns)** | **BinarySearchTree** | **AVLTree** | **RedBlackBST** | **SplayTree** |
| **Insert** | **1300520** | **565094** | **639** | **582** |
| **Search** | **487861** | **135** | **170** | **1244** |
| **Delete** | **101** | **585271** | **1129** | **691** |
| **Delete**  **Randomly** | **922235** | **1857716** | **17870** | **2361** |

| **Search Tree** | **Average Case** | | |
| --- | --- | --- | --- |
|  | **Insert** | **Delete** | **Search** |
| Binary Search Tree | O(log n) | O(log n) | O(log n) |
| AVL tree | O(log2 n) | O(log2 n) | O(log2 n) |
| Red-Black Tree | O(log n) | O(log n) | O(log n) |
| Splay Tree | O(log2 n) | O(log2 n) | O(log2 n) |

| **Search Tree** | **Worst Case** | | |
| --- | --- | --- | --- |
|  | **Insert** | **Delete** | **Search** |
| Binary Search Tree | O(n) | O(n) | O(n) |
| AVL tree | O(log2 n) | O(log2 n) | O(log2 n) |
| Red-Black Tree | O(log n) | O(log n) | O(log n) |
| Splay Tree | O(log2 n) | O(log2 n) | O(log2 n) |

**Conclusion:**

Each Table has its unque use cases for instance AVL tree is best technique for searching and BinarySearch Tree is quickest at deletion.

After implementing each of the above table I will consider Splay tree implementation for real problems as it has the best time complexity.