“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.” + Nishesh Kalakheti + 110017507

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.

Ans: Methods used:

**public** **static** String random\_string\_generate() //Generate random string of length 10

**public** **static** **void** insert\_into\_hash\_table(Object obj) // Adds the above generated random string to the hash table and computes average time for each insertion.

*insert\_into\_hash\_table*(*CH*); // Adds the random string to Cuckoo Hash Table

*insert\_into\_hash\_table*(*SH*); // Adds the random string to SequenceChaining Hash Table

*insert\_into\_hash\_table*(*QH*); // Adds the random string to QuadraticProbing Hash Table

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

Ans: Methods used:

**public** **static** String random\_string\_generate() //Generate random string of length 10

**public** **static** **void** find\_random\_strings(Object obj)// Searches for the above generated random string in the hash table and computes average time for each search.

*find\_random\_strings*(*CH*); // Adds the random string to Cuckoo Hash Table

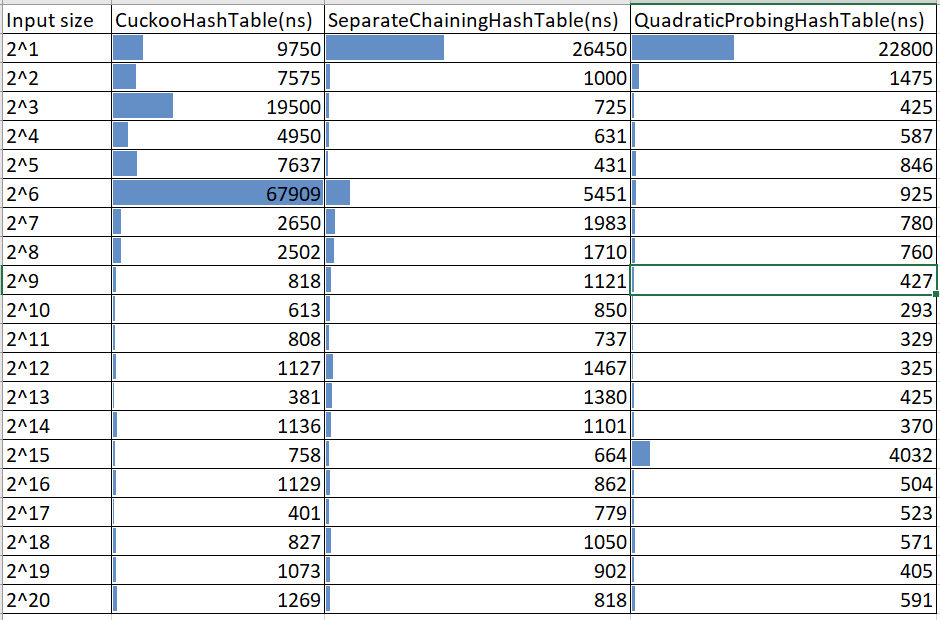
*find\_random\_strings*(*SH*); // Adds the random string to SequenceChaining Hash Table

*find\_random\_strings*(*QH*); // Adds the random string to QuadraticProbing Hash Table

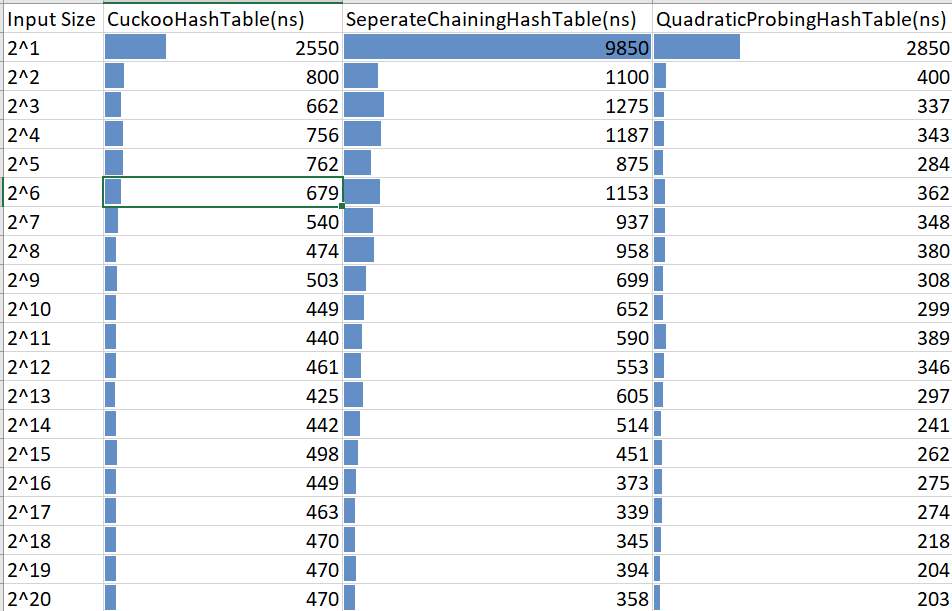
1. Repeat #1 and #2 with n = 2^i, i = 1, …, 20. Place the numbers in a table and compare the results for Cuckoo, QuadraticProbing and SeparateChaining. Comment on the times obtained and compare them with the complexities as discussed in class.

Ans: Random String Insertion using different hashing algorithm:

Average Insert Comparison:



Average Search Comparison:



Comments:

For the insertion of strings

* Cuckoo Hash Table seems to have comparatively more average insertion time than Separate Chaining Hash Table and Quadratic Probing Hash Table. The probable reason for cuckoo hash table for taking more time for insertion is it might end up in an infinite loop and rehashing algorithm is applied. Rehash requires O(n) insertions.

However a successfully insertion will have time complexity O(1).

* There seems to be a competition between Separate Chaining Hash Table and Qudratic Probing Hash Table. However, Quadratic Probing Hash Table has better average time. The reason should be because of the less clustering suffered by Quadratic Probing Hash.
* The insertion time complexity of insertion of Separate Chaining is O(1+α)

1 for applying hash function and get the slot

Α to traverse through the linked list.

α = Average keys per slot(load factor)

* The insertion time complexity for Quadratic Probing is O(1/(1- α)) time

α < 1 since Hash table size is greater than number of keys.

For the searching of strings:

* Separate Chaining Hash Table has taken more average time to search the strings than Cuckoo Hash Table and Quadratic Probing Hash table.
* Separate Chaining Hash Table maintains a linked list, on the worse case the search time can be O(n).
* There seems to be a competition between Cuckoo Hash Table and Qudratic Probing Hash Table. However, Quadatic Probing Hash Table has better average time.
* The search time complexity for Quadratic Probing is O(1/(1- α)) time

where α = (Number of elements present in the hash table / Hash Table size)

The benefit of using Quadratic Probing Hash Table is it causes less clustering. Hence the average search time for an element is better.

* For Cuckoo Hashing, if rehashing has to be done frequently, then the search time complexity increases. Otherwise for successful insertion of item, the search time complexity is O(1).

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.

Ans: Methods used:

a)

**public** **static** **void** insert\_elements(BinarySearchTree obj\_bin, AVLTree obj\_avl, RedBlackBST obj\_redbl, SplayTree obj\_spl, **boolean** random\_flag)

* This method calls the method insert\_tree() shown below. The random\_flag is set to false which will insert the elements in the order.

**public** **static** **long** insert\_tree(BinarySearchTree t, **int** i)

* This method will add the elements to Binary Search Tree.

**public** **static** **long** insert\_tree(AVLTree t, **int** i)

* This method will add the elements to AVL Tree.

**public** **static** **long** insert\_tree(RedBlackBST t, **int** i)

* This method will add the elements to Red Black Binary Search Tree.

**public** **static** **long** insert\_tree(SplayTree t, **int** i)

* This method will add the elements to Splay Tree.

b)

**public** **static** **int** generate\_random\_numbers();

* This method generates the random number between 1 and 100000.

**public** **static** **void** search\_tree(BinarySearchTree obj\_bin, AVLTree obj\_avl, RedBlackBST obj\_redbl, SplayTree obj\_spl)

* This method calls the method search\_tree() shown below.

**public** **static** **long** search\_tree(RedBlackBST t, **int** random\_num)

* This method will search the elements in Red Black Binary Search Tree.

**public** **static** **long** search\_tree(SplayTree t, **int** random\_num)

* This method will search the elements in Splay Tree.

**public** **static** **long** search\_tree(BinarySearchTree t, **int** random\_num)

* This method will search the elements in Binary Search Tree.

**public** **static** **long** search\_tree(AVLTree t, **int** random\_num)

* This method will search the elements in AVL Tree.

c)

**public** **static** **void** delete\_tree(BinarySearchTree obj\_bin, AVLTree obj\_avl, RedBlackBST obj\_redbl, SplayTree obj\_spl, **boolean** random\_flag)

* This method calls the method delete\_tree() shown below

**public** **static** **long** delete\_tree(AVLTree t, **int** elem)

* This method will delete the elements in AVL Tree.

**public** **static** **long** delete\_tree(BinarySearchTree t, **int** elem)

* This method will delete the elements in Binary Search Tree.

**public** **static** **long** delete\_tree(SplayTree t, **int** elem)

* This method will delete the elements in Splay Tree.

**public** **static** **long** delete\_tree(RedBlackBST t, **int** elem)

* This method will delete the elements in Red Black BS Tree.

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.

Ans: Methods used:

a)

**public** **static** **void** insert\_elements(BinarySearchTree obj\_bin, AVLTree obj\_avl, RedBlackBST obj\_redbl, SplayTree obj\_spl, **boolean** random\_flag)

* This method calls the method insert\_tree() shown below. The random\_flag is set to true which will insert the elements in the random order.

**public** **static** **long** insert\_tree(BinarySearchTree t, **int** i)

* This method will add the random elements to Binary Search Tree.

**public** **static** **long** insert\_tree(AVLTree t, **int** i)

* This method will add the elements to AVL Tree.

**public** **static** **long** insert\_tree(RedBlackBST t, **int** i)

* This method will add the elements to Red Black Binary Search Tree.

**public** **static** **long** insert\_tree(SplayTree t, **int** i)

* This method will add the elements to Splay Tree.

b)

**public** **static** **int** generate\_random\_numbers();

* This method generates the random number between 1 and 100000.

**public** **static** **void** search\_tree(BinarySearchTree obj\_bin, AVLTree obj\_avl, RedBlackBST obj\_redbl, SplayTree obj\_spl)

* This method calls the method search\_tree() shown below.

**public** **static** **long** search\_tree(RedBlackBST t, **int** random\_num)

* This method will search the elements in Red Black Binary Search Tree.

**public** **static** **long** search\_tree(SplayTree t, **int** random\_num)

* This method will search the elements in Splay Tree.

**public** **static** **long** search\_tree(BinarySearchTree t, **int** random\_num)

* This method will search the elements in Binary Search Tree.

**public** **static** **long** search\_tree(AVLTree t, **int** random\_num)

* This method will search the elements in AVL Tree.

c)

**public** **static** **void** delete\_tree(BinarySearchTree obj\_bin, AVLTree obj\_avl, RedBlackBST obj\_redbl, SplayTree obj\_spl, **boolean** random\_flag)

* This method calls the method delete\_tree() shown below. The random flag is set to true which is used to generate random values between 1 and 10000

**public** **static** **long** delete\_tree(AVLTree t, **int** elem)

* This method will delete the random elements in AVL Tree.

**public** **static** **long** delete\_tree(BinarySearchTree t, **int** elem)

* This method will delete the random elements in Binary Search Tree.

**public** **static** **long** delete\_tree(SplayTree t, **int** elem)

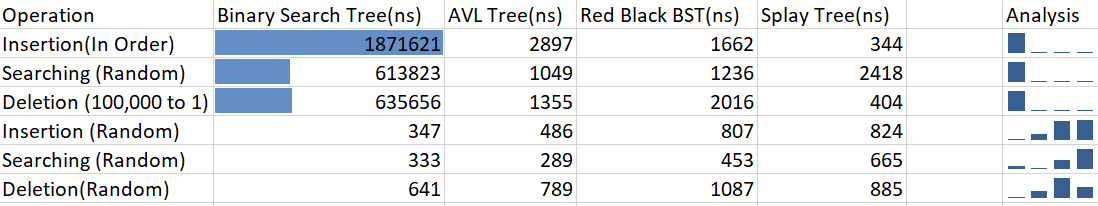
* This method will delete the random elements in Splay Tree.

**public** **static** **long** delete\_tree(RedBlackBST t, **int** elem)

* This method will delete the random elements in Red Black BS Tree

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

Ans:



Comments:

For insertion of elements on sequencial order from 1 to 10000:

* BST is going to be the worst option because the tree is skewed. Hence the time complexity to enter every new node is O(n).
* Splay tree insertion seems to be faster than other algorithms which is very obvious. Because we are inserting the elements in a sequential order, the new element will be the only one right child of the parent node. And a single rotation is required for that new node to be the parent node.
* AVL tree and Red Black BST has average time complexity O(logn).

For searching random elements between 1 and 10000:

* BST is going to take longer time since the tree is skewed. Hence its just like searching in a linked list. The worse case would be O(h).
* AVL tree is going to have the better time complexity than others since the tree is balanced. And the average time complexity would be O(logn).
* Splay tree is also skewed, hence searching random value takes more average time.
* Red Black BST will have average time complexity O(logn)

For deleting elements from 10000 and 1:

* Since the tree is skewed and its in the ascending order, removing elements in the opposite order will have the worse time complexity which is O(n).
* Splay tree will have the best case because the root element is the most recently inserted element. So the deletion from 10000 to 1 has the best time complexity which is O(1).
* AVL tree and Red Black BST will have average time complexity O(logn)

For inserting, removing and searching random elements:

* Splay trees seems to have more average time for each operations than others.
* AVL tree seems to be the consistent one.
* All the trees for each operations will have same average time complexity O(logn).

Which trees would I use in the practical applications?

* Splay tree when the data has to be entered in an order and when I have to frequently access the recently inserted element.
* AVL trees when data is inserted in an order and when the data has to be fetched randomly.
* Binary Search Tree could be used if input data is not in ascending or descending order. The data should be random.
* Since Red Black Binary Search Tree is self balancing binary search tree, it is an option over AVL tree. I would implement both of these and check the average operation time and choose the best one.