

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.” + Arun Reddy Nalla + 110088379

Created a java file for both TASK 1 AND TASK 2

FILE NAME Task1And2.java

Three methods are used in this file

public static String generaterandomstring ():

This method is used for generating random strings of length 10

public static HashMap<Integer, String>avginsertiontime (int n, HashMap<Integer, String>hashtable):

public static Map<Integer, String> avgsearchtime (int n, HashMap<Integer, String> hashtable):

TASK1

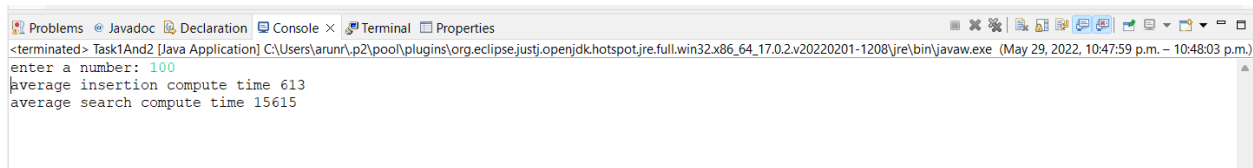
Within a Java class, write a method that creates n random strings of length 10 and inserts them in a hash table. The method should compute the average time for each insertion.

Output :

public static HashMap<Integer, String>avginsertiontime (int n, HashMap<Integer, String>hashtable):

This method is used for inserting n random string which is generated from the function generaterandomstring(). This method also computes the average time taken for insertion

```
enter a number: 100
average insertion compute time 613 nanoseconds
```

A screenshot of an IDE terminal window. The terminal shows the output of a Java application. It starts with a prompt "enter a number:" followed by the user input "100". Below this, it displays "average insertion compute time 613" and "average search compute time 15615". The window title bar indicates it's a Java application running in Eclipse. The status bar at the bottom shows the date and time as May 29, 2022, 10:47:59 p.m. - 10:48:03 p.m.

Task1And2.java output

Note : the time is calculated in nano seconds

TASK 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

Output :

```
enter a number: 100
average search compute time 15615 nanoseconds
```

public static Map<Integer, String> avgsearchtime (int n, HashMap<Integer, String> hashtable):

In this method, again generaterandomstring() function is used to generate a random string and search if the string is found in the hash table. if the string is found, it is deleted. This method also calculates the average time taken for searching the string

TASK 3

FILE LOCATION : src\Assignment\Task3.java

Repeat #1 and #2 with $n = 2^i$, $i = 1, \dots, 20$. 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.

public static String generaterandomstring():

This method is used for generating random strings of length 10

public static void avginsertiontime()

This method is used for inserting n random string which is generated from the function generaterandomstring().

This method insert n random string in three different methods: CuckooHashTable, quadraticHashTable, separateChainHashTable where $n = 2^i$, $i = 1, \dots, 20$.

This method also computes the average time taken for insertion

public static void avgsearchtime():

In this method again, generaterandomstring() function is used to generate a random string and search if the string is present in the table.

The search is done on all the three hashtable and string is removed if found.

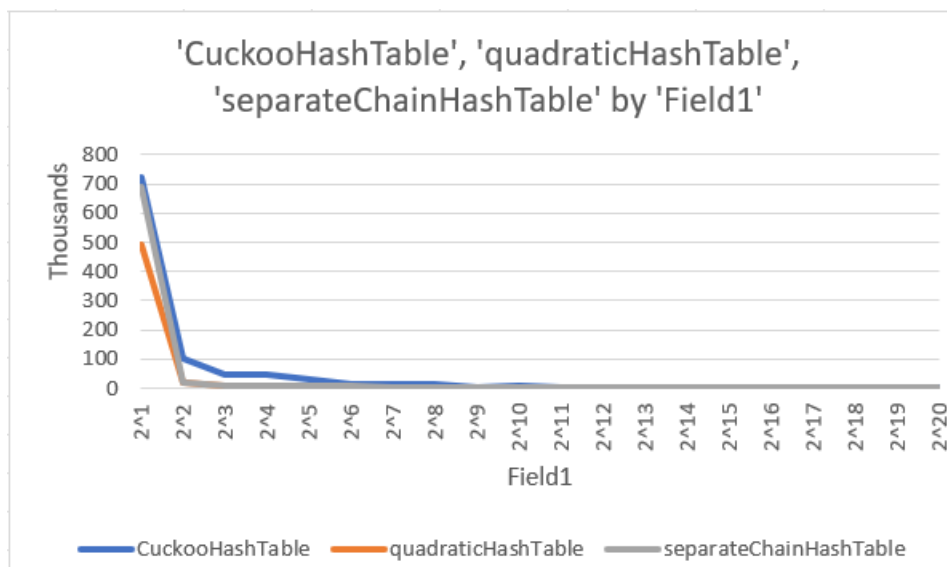
This method also calculates the average time taken for searching the string

Note: Time is measured in Nanoseconds

Output for insertion:

	CuckooHashTable	quadraticHashTable	separateChainHashTable
2^1	724300	494400	692300
2^2	102200	20850	22400
2^3	44700	8662.5	11337.5
2^4	44450	5375	8681.25
2^5	30903.125	4806.25	8478.125
2^6	15870.3125	3606.25	6642.1875
2^7	14365.625	1725.78125	2996.09375
2^8	11707.42188	3580.078125	2947.65625
2^9	5232.8125	1462.695313	2563.085938
2^10	6384.277344	1091.894531	1273.632813
2^11	2624.511719	562.9394531	3299.658203
2^12	2843.896484	520.4101563	1878.710938
2^13	2709.020996	259.7412109	817.6635742
2^14	2364.044189	222.9797363	763.1835938
2^15	2341.461182	400.6164551	679.4555664
2^16	2130.181885	188.1744385	531.1340332
2^17	1915.39917	190.9606934	494.4229126
2^18	1916.215897	192.8611755	500.3799438
2^19	1910.628128	190.2812958	493.1629181
2^20	1909.50222	191.7541504	514.1279221

0

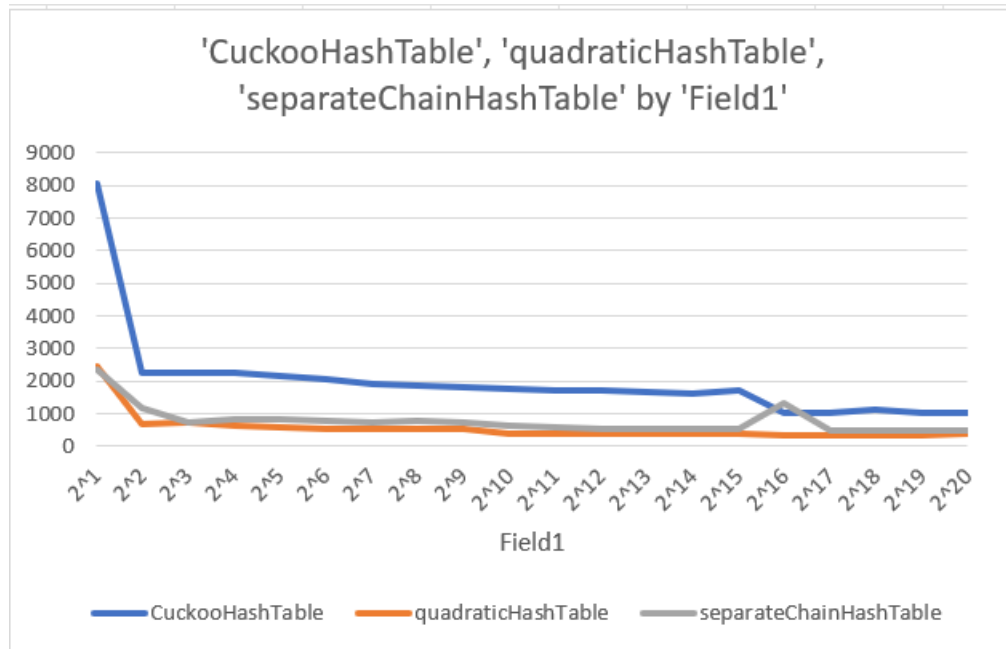


- From the insertion graph, we can observe that there is a sudden drop at 2^2 in the insertion in all the three techniques. For cuckoo hash table there was a steady decrease up to 2^8 then it became constant. For quadratic probing and separate chaining, the graph became steady right next to 2^2 .
- Also we can observe that quadratic is better than both cuckoo hash table and separate chaining.

Output for search and deletion:

	CuckooHashTable	quadraticHashTable	separateChainHashTable
2^1	8050	2450	2350
2^2	2275	700	1150
2^3	2262.5	725	712.5
2^4	2262.5	631.25	812.5
2^5	2171.875	596.875	818.75
2^6	2071.875	537.5	765.625
2^7	1888.28125	539.84375	725
2^8	1866.796875	543.359375	758.984375
2^9	1816.015625	522.265625	715.8203125
2^{10}	1780.175781	406.1523438	624.0234375
2^{11}	1692.382813	400.8300781	591.796875
2^{12}	1691.186523	382.7880859	550.7324219
2^{13}	1648.974609	370.1904297	527.2827148
2^{14}	1630.13916	363.6474609	520.8068848
2^{15}	1726.782227	364.0289307	518.7713623
2^{16}	1037.304688	357.4539185	1334.208679
2^{17}	1004.233551	341.0934448	480.3840637
2^{18}	1127.365875	335.6529236	478.3435822
2^{19}	1011.042786	338.3094788	487.8606796
2^{20}	1046.811295	399.2645264	490.3558731

Note: Time is measured in Nanoseconds



- From the deletion graph, we can observe that the cuckoo hash table is performance worst than quadratic and separate chaining hash table.
- Also we can observe that there is a sudden drop at 2^2 in the insertion in all the three techniques. . For cuckoo hash table there was a steady decrease till end. For other two techniques there was a slight decrease
- From both the tables and graphs we can notice that Separate Chaining and Quadratic Probing. is performing better than Cuckoo hash table.

TASK 4

FILE LOCATION : src\Assignment\Task4.java

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

- 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.
- Do 100,000 searches of random integer keys between 1 and 100,000. Find the average time of each search.
- Delete all the keys in the trees, starting from 100,000 down to 1 (in that order). Find the average time of each deletion.

FOUR methods are used in this file

```
public static void AVLTASK():  
public static void BinarySearchTask()  
public static void RedBlackBSTTASK()  
public static void SplayTreeTASK()
```

in all the four methods, insertion values from 1 to 100,000 done.

keys are searched from between 1 and 100,000 which are generated randomly

And the keys are removed from 100,000 to 1 with respective methods

```
Create the AVL tree...  
Average time to insert 100000 keys: 250 nanoseconds  
Average time for searching 100000 random strings: 288 nanoseconds  
Average time to delete 100000 keys: 182 nanoseconds  
  
Create the binary tree...  
Average time to insert 100000 keys: 573102 nanoseconds  
Average time for searching 100000 random strings: 201853 nanoseconds  
Average time to delete 100000 keys: 548601 nanoseconds  
  
Create the RedBlackBST tree...  
Average time to insert 100000 keys: 410 nanoseconds  
99999keys found  
Average time for searching 100000 random strings: 283 nanoseconds  
Average time to delete 100000 keys: 472 nanoseconds  
  
Create the Splay tree...  
Average time to insert 100000 keys: 97 nanoseconds  
Average time for searching 100000 random strings: 480 nanoseconds  
Average time to delete 100000 keys: 60 nanoseconds
```

TASK 5

FILE LOCATION : src\Assignment\Task5.java

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.

FOUR methods are used in this file

```
public static void AVLTASK():  
public static void BinarySearchTask()  
public static void RedBlackBSTTASK()  
public static void SplayTreeTASK()
```

in all the four methods, insertion values between 1 to 100,000 are done randomly .
keys are searched from between 1 and 100,000 which are generated randomly
And the keys are removed from 1 to 100000 randomly with respective methods

```
Create the AVL tree...  
Average time to insert 100000 keys: 714 nanoseconds  
Average time for searching 100000 random strings: 343 nanoseconds  
Average time to delete 100000 keys: 740 nanoseconds  
  
Create the binary tree...  
Average time to insert 100000 keys: 318 nanoseconds|  
Average time for searching 100000 random strings: 221 nanoseconds  
Average time to delete 100000 keys: 332 nanoseconds  
  
Create the RedBlackBST tree...  
Average time to insert 100000 keys: 280 nanoseconds  
Average time for searching 100000 random strings: 468 nanoseconds  
Average time to delete 100000 keys: 1155 nanoseconds  
  
Create the Splay tree...  
Average time to insert 100000 keys: 478 nanoseconds  
Average time for searching 100000 random strings: 338 nanoseconds  
Average time to delete 100000 keys: 384 nanoseconds
```

TASK 6:

2. 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).

	TASK 4			TASK 5		
	Insertion	Search	Deletion	insertion	search	deletion
AVL tree	536	895	322	714	343	740
binary tree	609107	220618	542760	318	221	332
RedBlackBST tree	329	288	497	280	468	1155
Splay tree	97	489	52	478	338	384

NOTE: Time is measured in nanoseconds

We can observe that the splay tree taken less time for insertion in sequential order where as RedBlackBST tree less time in random insertion. But except binary sequential insertion each insertion method took almost same time.

The same applies to search and deletion also. except binary sequential insertion each insertion method took almost same time.

So we can we say that we shouldn't use binary tree for sequential insertion. For some insertion applications, Splay tree would be an efficient choice. But that we couldn't decide any other thing because all avg time are almost same. It only can be decide based on the real life situations.