

***An Experiment with Hashtables:  
Cuckoo, Quadratic Probing and Separate Chaining***

## ***Introduction***

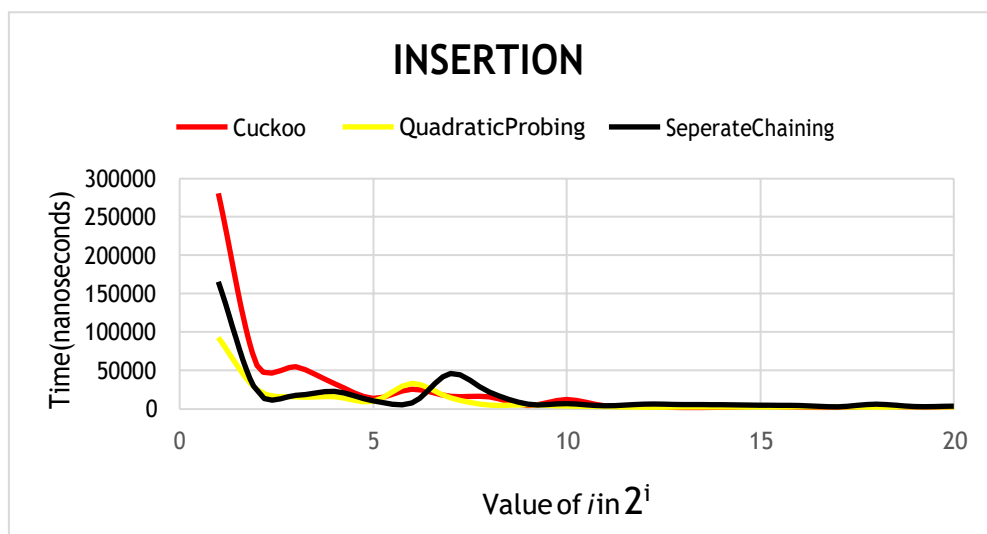
### **(Java Project: Linear Data Structure)**

- I have created a class file named **HashTask.java** in hashTable package. This class has a method called **RunHashCuckoo(int num, int j)**, which creates n random strings i.e, value passed in integer num and **inserts** them in Cuckoo hash table using insert() method.
- The same method creates n random strings & then **searches** them in Cuckoo hash table by using contains() method. The contains() method is placed in if condition, so if string is found it gets **deleted** by using delete() method.
- The main function of HashTask.java has a for loop which controls the value of n. It calls three functions named **RunHashCuckoo(int num, int j)**, **RunQuadraticProbing(int num, int j)** and **RunSeperateChaining(int num, int j)**, which calculates the average time of each insertion, search and deletion for all three hash table.
- **Table 1** contains the average time of each insertion for different values of  $n=2^i$ , where i has values as:  $i=1, \dots, 20$ . **Figure 1.1** shows the comparison of insertion time of the three hash table used. The x-axis represents value of i and y-axis represents time in nanoseconds.
- As the numbers to be inserted increases, the time of insertion decreases for each hash table. From Figure 1.1, I can say that a stable trend is observed when values of i is between 10 to 20 (Figure 1.2). It can be concluded that the average insertion time complexity of each hash table is  $O(1)$ . The data does not affects the time taken, it takes constant time for insertion when size of data is large.

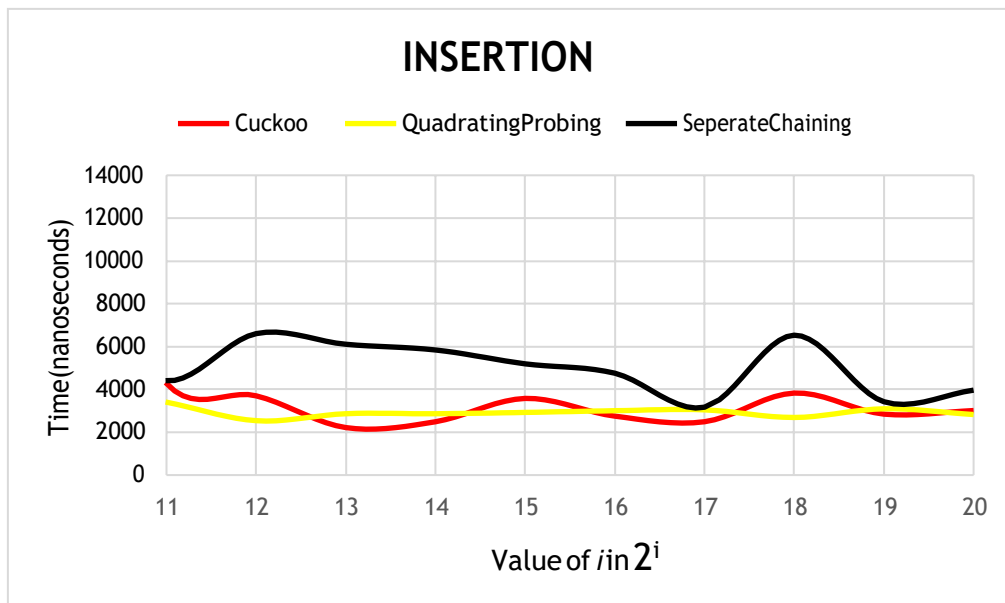
## *Results and discussion*

n	Average time of each insertion in nanoseconds		
	Cuckoo	Quadratic Probing	Seperate Chaining
$2^1$	280970	93341	165888
$2^2$	56832	25521	22331
$2^3$	55414	16069	17959
$2^4$	33319	16275	23187
$2^5$	14104	10146	10973
$2^6$	26016	33577	8049
$2^7$	16770	14680	46519
$2^8$	16105	5328	22290
$2^9$	4828	5807	6510
$2^{10}$	12600	4152	7218
$2^{11}$	4268	3408	4404
$2^{12}$	3700	2543	6611
$2^{13}$	2216	2870	6112
$2^{14}$	2492	2869	5846
$2^{15}$	3579	2929	5201
$2^{16}$	2750	3011	4761
$2^{17}$	2489	3052	3176
$2^{18}$	3830	2695	6539
$2^{19}$	2848	3100	3430
$2^{20}$	3011	2832	3972

**Table 1: Average time of each insertions**



**Figure 1.1: Average time of each insertion**



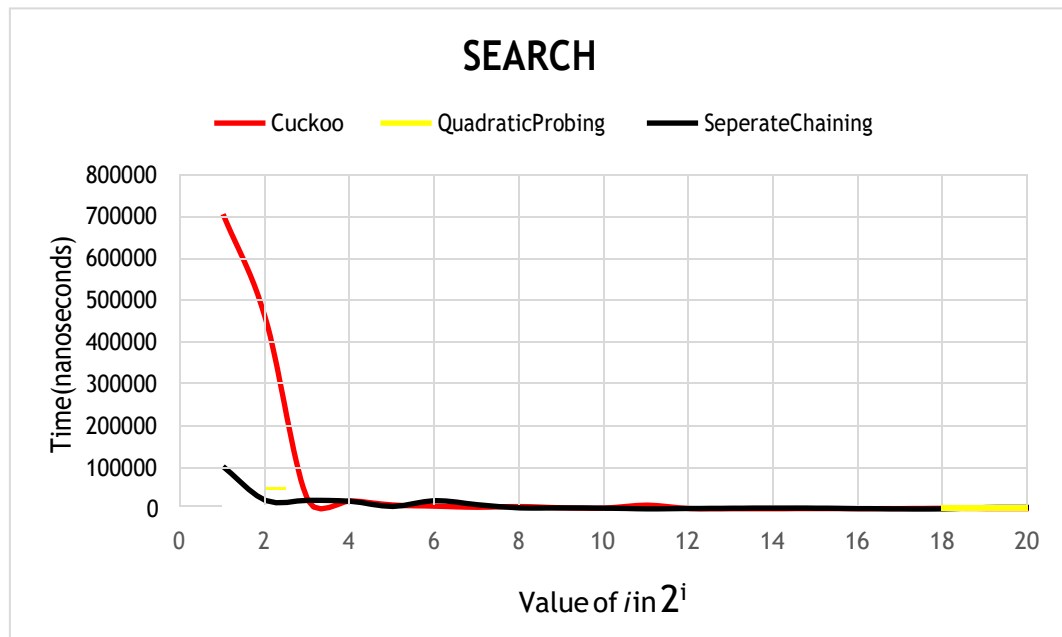
**Figure 1.2: Average time of each insertion when  $i=11$  to 20**

- Figure 1.2 shows that among the three hash table, Separate chaining hash table is slowest in insertion. Though time taken by Cuckoo & Quadratic Probing is varying.
- Table 2 contains the average time of each **search** for different values of  $n=2^i$ , where  $i$  has values as:  $i=1, \dots, 20$ . Figure 2.1 shows the comparison of **search** time of the three hash table used. The x-axis represents value of  $i$  and y-axis represents time in nanoseconds.
- From figure 2.1, it can be concluded that average search time complexity for all three hash table is  $O(1)$  again. Here a stable trend is values of  $i$  is between 6 to 20 (figure 2.2)
- As the size of data increases, the hash tables takes less time for searching. Moreover, the average time taken is fluctuating steadily, when compared with each other as shown in Figure 2.2.

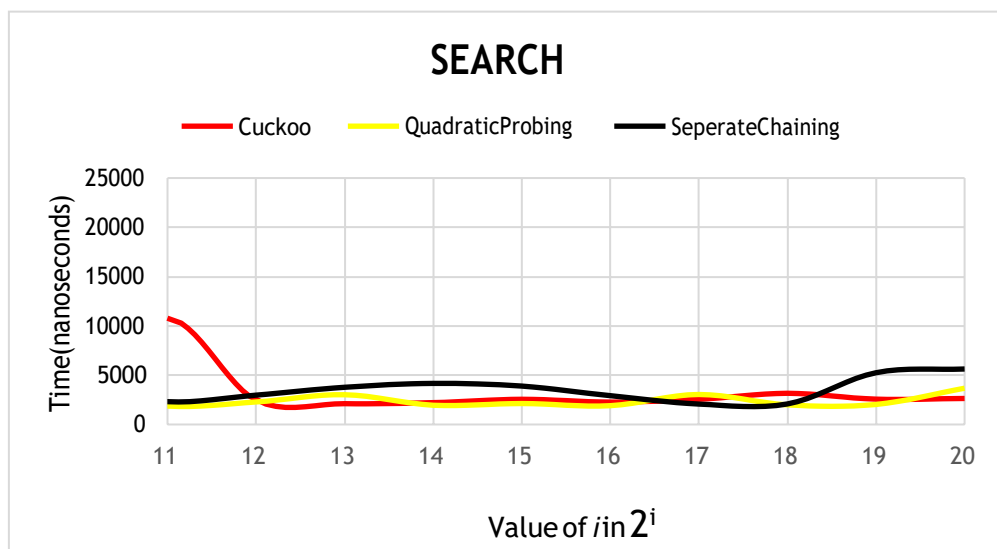
n	Cuckoo	Quadratic Probing	Separate Chaining
$2^1$	706087	43008	103502
$2^2$	456901	45489	22331
$2^3$	22153	15419	22685
$2^4$	21297	31694	20381
$2^5$	11121	12893	8004
$2^6$	8211	6734	21947
$2^7$	5723	10076	12118
$2^8$	7434	6428	4541
$2^9$	4362	7292	4425
$2^{10}$	3676	3312	3909
$2^{11}$	10793	1857	2319
$2^{12}$	2459	2275	2982
$2^{13}$	2118	3013	3777
$2^{14}$	2220	1942	4175

$2^{15}$	2557	2114	3894
$2^{16}$	2276	1895	2923
$2^{17}$	2557	3034	2077
$2^{18}$	3170	1979	2084
$2^{19}$	2571	2025	5256
$2^{20}$	2646	3677	5629

**Table 2: Average time of each search**



**Figure 2.1 Average time of each search**



**Figure 2.2: Average time of each insertion when  $i=11$  to 20**

***An Experiment with Search Trees:***  
***BinarySearchTree, AVLTree, RedBlackBST, SplayTree***

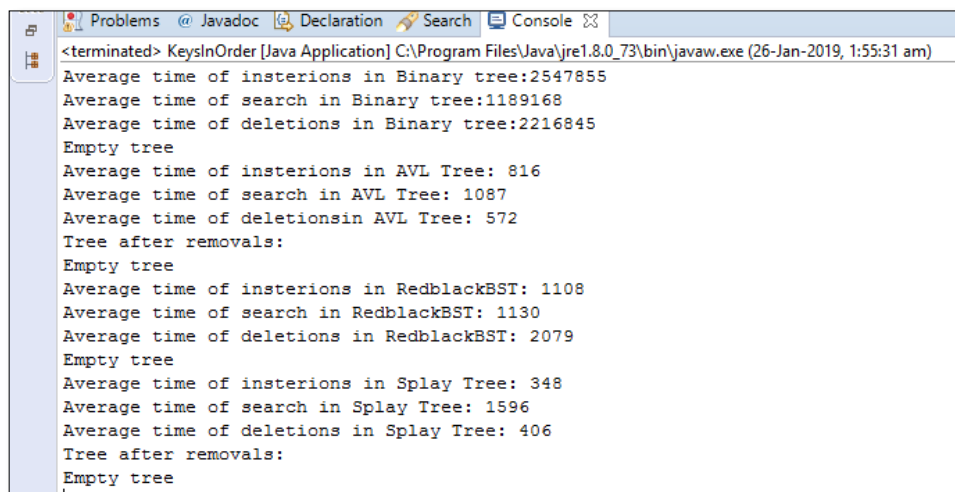
## Introduction

### (Java Project: Search Trees)

#### ❖ Ordered Keys

- For all the trees, **contains()** method is used to perform search. Random integer between 1 to 100,000 is passed as argument to these method 100,000 times (in for loop), which performs 100,000 searches.
- Keys are deleted by **remove(i)** method in Binary Search Tree, AVL Tree and Splay Tree and by **delete(i)** method in Red Black BST. The initial value of i is 100,000 which decrements by a for loop.
- I have created a class file named **KeysInOrder.java** in searchtrees package. For each tree a method is created which performs insertion, search and deletion for that tree when called from the main function. Those methods are: **runBinaryTree(num)**, **runAVLTree(num)**, **runRedBlackTree(num)** and **runSplayTree(num)**. The value of num is 100,000.

#### Output:

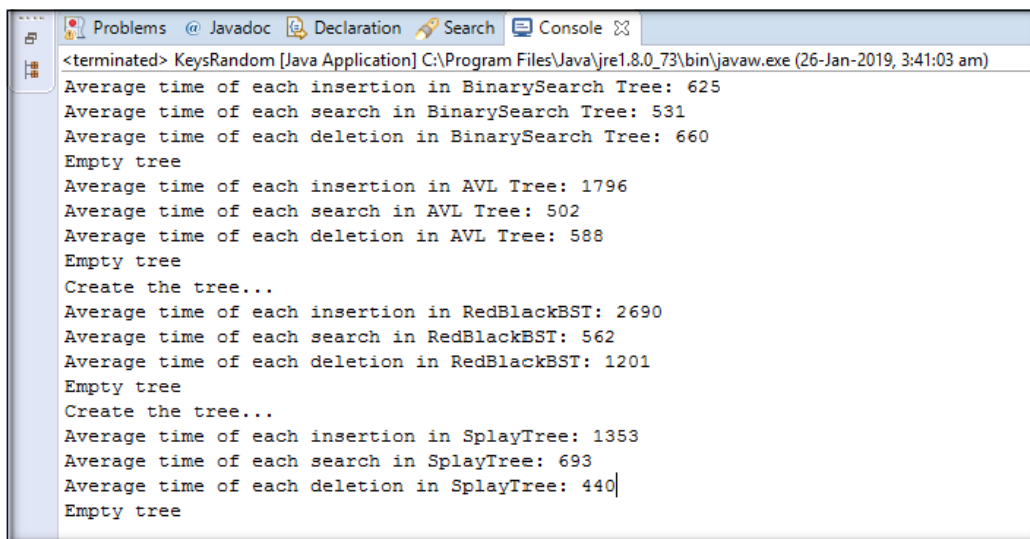


```
<terminated> KeysInOrder [Java Application] C:\Program Files\Java\jre1.8.0_73\bin\javaw.exe (26-Jan-2019, 1:55:31 am)
Average time of insterions in Binary tree:2547855
Average time of search in Binary tree:1189168
Average time of deletions in Binary tree:2216845
Empty tree
Average time of insterions in AVL Tree: 816
Average time of search in AVL Tree: 1087
Average time of deletionsin AVL Tree: 572
Tree after removals:
Empty tree
Average time of insterions in RedblackBST: 1108
Average time of search in RedblackBST: 1130
Average time of deletions in RedblackBST: 2079
Empty tree
Average time of insterions in Splay Tree: 348
Average time of search in Splay Tree: 1596
Average time of deletions in Splay Tree: 406
Tree after removals:
Empty tree
```

### ❖ *Random Keys*

- An array `insert[]` of length 100,000, stores 100,000 integers between 1 to 100,000 randomly. The values are inserted by same methods as above, though the argument passed is `insert[i]`, where `i` is controlled by for loop to access `insert[]` array.
- Another array named `delete[]`, contain random keys between 1 and 100,000. For all trees, keys are deleted in the same manner as before. As argument, element of `search[]` array is passed.
- Here, I have created a class file named **KeysRandom.java** in `searchtrees` package. For each tree a method is created which performs insertion, search and deletion for that tree when called from the main function. Those methods are: **`runBinarySearch(insert, search, delete, num)`**, **`runAVL( insert, search, delete, num)`**, **`runRedBlackBST(insert, search, delete, num)`** and **`runSplayTree(insert, search, delete, num)`**. The value of `num` is 100,000.

### Output:



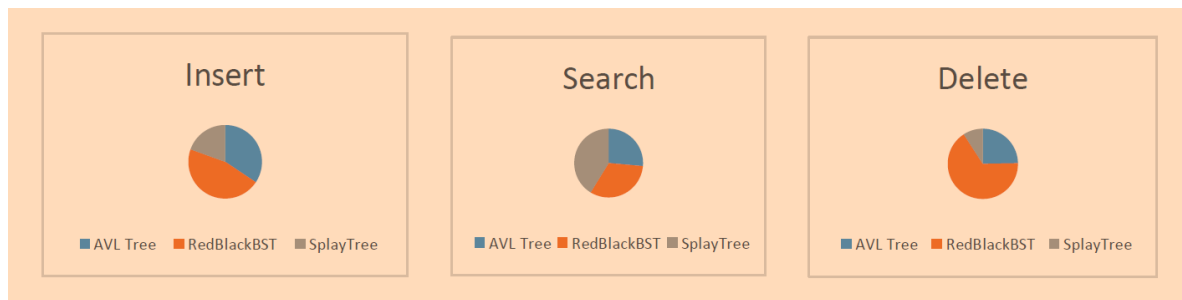
```
<terminated> KeysRandom [Java Application] C:\Program Files\Java\jre1.8.0_73\bin\javaw.exe (26-Jan-2019, 3:41:03 am)
Average time of each insertion in BinarySearch Tree: 625
Average time of each search in BinarySearch Tree: 531
Average time of each deletion in BinarySearch Tree: 660
Empty tree
Average time of each insertion in AVL Tree: 1796
Average time of each search in AVL Tree: 502
Average time of each deletion in AVL Tree: 588
Empty tree
Create the tree...
Average time of each insertion in RedBlackBST: 2690
Average time of each search in RedBlackBST: 562
Average time of each deletion in RedBlackBST: 1201
Empty tree
Create the tree...
Average time of each insertion in SplayTree: 1353
Average time of each search in SplayTree: 693
Average time of each deletion in SplayTree: 440
Empty tree
```



## Results and discussion

		Average time			
Key	Operation	BinarySearch Tree	AVL Tree	RedBlackBST	SplayTree
100,000 keys (In order)	Insert	2648435	787	1056	446
	Search	1543866	995	1219	1553
	Delete	3344740	765	2045	282
100,000 keys (Random)	Insert	625	1796	2690	1353
	Search	531	502	562	693
	Delete	660	588	1201	440

**Table 3: Average running times of each operation for each tree**

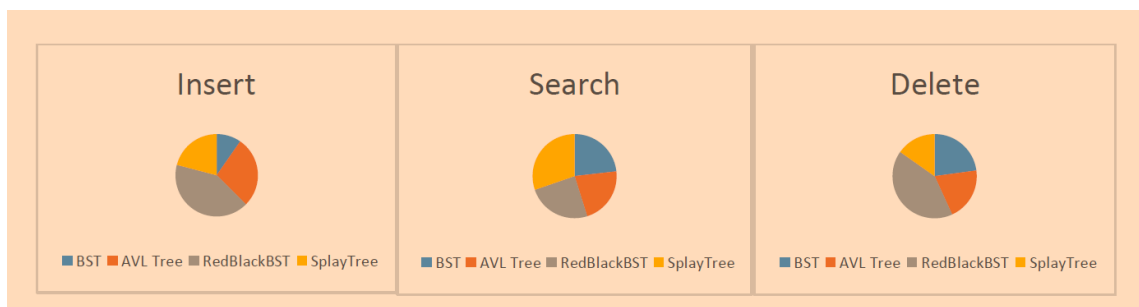


**Figure 3.1 Comparison when keys are in order**

**When keys are ordered:**

**For insertion & deletion:** Splay Tree is fast and Binary search tree is slow.

**For search:** AVL Tree is fast and Binary search tree is slow.



**Figure 3.2 Comparison when keys are random**

**When keys are random:**

**For insertion:** Binary search tree is fast and RedBlackBST is slow.

**For search:** AVL Tree is fast and Splay Tree is slow.

**For deletion:** Splay Tree is fast and redBlackBST is slow.

Tree	Keys for which Tree are faster		
	Insertion	Search	Deletion
BinarySearch Tree	Random	Random	Random
AVL Tree	In order	Random	Random
RedBlackBST	In order	Random	Random
SplayTree	In order	Random	In order

**Table 4: Comparison of when tree operations run faster**

- Table 4 shows for which type of data i.e. ordered or random, operation is fast. For example, binary search tree performs insertion faster when keys to be inserted are random. Similarly, Splay tree performs deletion faster when keys are in order and so on. Moreover, for **search and deletion**, AVL Tree and Splay Tree are best respectively regardless of type of data.
- For real applications, I would use **AVL Tree** because it's worst case and average case time complexity is  $O(\log n)$  for all the three operations. Same is the case with RedBlackBST, but compared to AVL Tree, RedBlackBST is slower from results in Table 3 (Figure 3.2). Worst case time complexity of Binary Search Tree and Splay tree for all operations is  $O(n)$ , which makes them slower.
- Complexity of  $O(\log n)$  means it has to traverse upwards and downwards through the height of tree to perform operation whereas complexity of  $O(n)$  means all the nodes needs to be visited, which takes longer time. Thus AVL Tree is fastest. For instance, worst case occur in binary search tree when ordered keys are to be inserted ( $O(n)$ ), but when random keys are to be inserted it inserts faster ( $O(\log n)$ ) as in Table 3.