**AVL Tree and Cuckoo Hashing Report**

# Cuckoo Hashing

1. Design Decision

* In this implementation, Cuckoo hashing table is modeled by an array that have two different hash functions. It can handle only take one when inserting duplicated items. This implementation does not require the inserted items to implement Comparable interface
* The table size of the hash table will always be a prime number
* The array *MULTIPLIERSARR* will hold the different random values of length 2 *(numHashFunctions* variable*)* which is used to calculate the hash functions.
* Method *generateNewHashFunctions()* create new hash functions.
* Method *myhash*() compute the hash code for the each element by multiply the appropriate hash function with the hash code of that element. It will place that hash code into the table by modulus and adding the table length if hash value is negative.
* The table size is always a prime number and its default size is 101.
* The maximum load factor for the table is 0.4 so the number of collisions is not large.
* Method *contains* checks to see if the hash functions will return the position of the item in the hash table that equals the compared item.
* Method *remove* simply find the position of the compared item, remove it and reduce the size
* *Insert* method will check to see if the item contained in the table if it is then return, if not then check to see if the current size exceeds the max load factor, expand if need to.
* It will attempt to insert the item using the first hash function, if the spot is filled then try again with second hash function. If it still doesn’t find the empty spot, an item in the first or second position (chosen randomly) will be evicted and is replaced by the new item. The evicted item will be reinserted using same technique.
* Variable *lasPos* is used to avoid cycles. It keeps track of the last position is evicted. If the random position is the same as *lasPos*, select a new random item. Also, a limited number of probing random items is five since two hash functions may probe the same evicted location and go on forever.
* Limit for evicting items is 100 times (hold by *COUNT\_LIMIT*). After 100 times of eviction but there is no empty spot, *rehash* variable will be increment.
* If *rehash* variable is less than or equals to *ALLOWED\_REHASHES,* *rehash* method will be called which will compute two different hash functions and copy every element over using these hash functions on the same hash table
* if *rehash* variable is greater than *ALLOWED\_REHASHES, expand* method is called, which will create a new different hash table with its size is 2.5 \* the old size, and call *rehash* function.
* The number of calling rehash method (*ALLOWED\_REHASHES*) is at most one time, which is specified by variable *rehashes*. Allowed rehash number is kept low because it can slow down the program by reinserting every item in the table, if the value of *rehash* is more than one then *expand* method is called.
* *Expand* will extend the size by 1/0.4 = 2.5 times the old table size then call *rehash*.
* After *expand* and *rehash* methods, *insert* method keeps reinserting the item until an empty spot is found

1. Runtime Analysis (Expectation)

* *Insert* method should run in **O(N)** time in the worst case because it will eventually need to expand or rehash the hash table, which will require to copy every item over.
* *Contains* method should run in **O(1)** time since I only check two position returned by two hash functions.
* *Remove* method should in **O(1)** time since I only check two position returned by two hash functions and remove one.

# AVL Tree

1. Design Decision

* AVL tree is represented as a binary tree that only keeps track of the root and has a balanced condition (the height of left and right subtree differs at most by 1).
* Items is in ordered so it needs to implement Comparable interface
* This AVL tree will ignore the duplicate items of the inserted
* To balance the tree, rotations are used, implemented in four methods which accounts for four cases:

1. An insertion into the left subtree of the left child of *α* (*rotateWithLeftChild*)

2. An insertion into the right subtree of the left child of *α* (*doubleWithLeftChild*)

3. An insertion into the left subtree of the right child of *α* (*doubleWithRightChild*)

4. An insertion into the right subtree of the right child of *α* (*rotateWithRightChild*)

* To insert an item to the tree, starting from the root and place it in appropriate spot by comparing it to the node. Perform balance after each insertion.
* Method *contains* will compare the item will the root and continue down until it reach the node that has the same item or return false if there is no such node
* Method *remove* will first find the item x in the tree, then remove it right away if it is a leaf, if not then it will replace that empty spot by the minimum value of the its right subtree. *Balance* method is called at the end.

1. Runtime Analysis (Expectation)

* *Insert* method should run in **O(logN)** time in the worst case since it takes O(logN) to get down to the appropriate node and possibly O(logN) to rebalance the tree from that node up to the root (rotations are performed in constant time).
* *Contains* method should run in **O(logN)** time since each comparison will reduce the input size in half (the tree is always balanced), because I can either check the right or left subtree of the node.
* *Remove* method should in **O(logN)** time since it takes O(logN) time to get down to the proper node and maybe replace it with the smallest of the right child O(logN) and rebalance the node which takes O(logN) time.

# Actual Runtime Data

1. Testing Runtime Design Note

* Runtime analysis is performed with AVL Tree and Cuckoo Hashing in the same manner
* Array *testSize* specify the test case to run methods method *insert, contains, and remove* of two classes. It ranges from 500000 to 32000000, with the next test size is double the previous test size
* The runtime will be measured in millisecond, using System.*currentTimeMillis*() method. To compute the *totalTime* variable, I use several for loops
* One outer loop will run through every item in the testSize array (7 runs). I use *indexTest* variable to specify the index of testSize array
* Three inner loops, which are placed sequentially, will run three methods: insert, contains, remove, inside them. They will run with the running number of the value from the *totalTime[indexTest]*.
* Before insert loop, specify *start* variable to hold the current millisecond. In the *insert* method, a random number generator is used for insertion, the number can range from 0 to number of test size. I will randomly insert the items into the array. Then I save the runtime of the loop and substract *start* from that runtime and assign it to *totalTime* variable.
* With that same array, after insertion and before checking contains, reassign *start* variable to hold the current millisecond. In the method *contains,* continue to pick a random number (range from 0 to number of test size) and pass it to *contains* method. Then I save the runtime of the loop and substract *start* from that runtime and assign it to *totalTime* variable.
* With that same array, after checking *contains* and before calling *remove* method, reassign *start* variable to hold the current millisecond. In the *remove* method*,* pick a random number (from 0 to number of test size) and pass it to *remove* method. Then I save the runtime of the loop and substract *start* from that runtime and assign it to *totalTime* variable.
* The array of AVL tree’s initial size is its default size (101). In Cuckoo Hashing, the array size of the hash table is set to the size of input.

1. Actual Runtime Data and Analysis
   1. **AVL Tree**

|  |  |  |
| --- | --- | --- |
| Testing AVL Tree Insertion | | |
| Case | Test Size | Total Time (ms) |
| 0 | 500000 | 724 |
| 1 | 1000000 | 1017 |
| 2 | 2000000 | 2072 |
| 3 | 4000000 | 5630 |
| 4 | 8000000 | 14217 |
| 5 | 16000000 | 28815 |
| 6 | 32000000 | 64370 |

|  |  |  |
| --- | --- | --- |
| Testing AVL Tree Contains | | |
| Case | Test Size | Total Time (ms) |
| 0 | 500000 | 503 |
| 1 | 1000000 | 643 |
| 2 | 2000000 | 2085 |
| 3 | 4000000 | 3442 |
| 4 | 8000000 | 8731 |
| 5 | 16000000 | 20352 |
| 6 | 32000000 | 49406 |

|  |  |  |
| --- | --- | --- |
| Testing AVL Tree Remove | | |
| Case | Test Size | Total Time (ms) |
| 0 | 500000 | 861 |
| 1 | 1000000 | 891 |
| 2 | 2000000 | 2135 |
| 3 | 4000000 | 5668 |
| 4 | 8000000 | 13784 |
| 5 | 16000000 | 28642 |
| 6 | 32000000 | 66728 |

* Based on the data, the runtime of randomly insert, contains, and remove items AVL tree grow linearly as the input increases. This happens because these methods are called in a for loop that runs from 0 to testSize-1 times, which increases the runtime of the AVL structure to **O(NlogN).**
* Method insertion and remove appear to have identical runtime and much larger than contains method (approximately 1.5 times). This is reasonable since after each insert and remove item, balance method is called.
  1. **Cuckoo hashing**

|  |  |  |
| --- | --- | --- |
| Testing CuckooHash Insertion | | |
| Case | Test Size | Total Time (ms) |
| 0 | 500000 | 297 |
| 1 | 1000000 | 347 |
| 2 | 2000000 | 889 |
| 3 | 4000000 | 1346 |
| 4 | 8000000 | 3858 |
| 5 | 16000000 | 6553 |
| 6 | 32000000 | 25551 |

|  |  |  |
| --- | --- | --- |
| Testing CuckooHash Contains | | |
| Case | Test Size | Total Time (ms) |
| 0 | 500000 | 162 |
| 1 | 1000000 | 216 |
| 2 | 2000000 | 254 |
| 3 | 4000000 | 632 |
| 4 | 8000000 | 1387 |
| 5 | 16000000 | 3467 |
| 6 | 32000000 | 6551 |

|  |  |  |
| --- | --- | --- |
| Testing CuckooHash Remove | | |
| Case | Test Size | Total Time (ms) |
| 0 | 500000 | 132 |
| 1 | 1000000 | 171 |
| 2 | 2000000 | 247 |
| 3 | 4000000 | 686 |
| 4 | 8000000 | 1373 |
| 5 | 16000000 | 2788 |
| 6 | 32000000 | 5231 |

* The data suggests that the runtime is not much affected by the increment of the input size.
* Based on the graph, *Insertion* method, with the growth rate of O(N), takes more time to run than *contains* and *remove,* with growth rate is O(1)*.* This behavior is not surprising because when the items inserted cause the hash table to be rehashed or expanded.
* The runtimes of both methods *contains* and *remove* appear to be idential
  1. **AVL Tree and Cuckoo Hashing Comparison**
* Overall, the runtime of both data structures increases as the input size increases. The runtime of AVL Tree is much larger (around 4 to 6 times) and grows at higher rate than the runtime of Cuckoo Hashing
* The runtime of *insert* methods in both structures grows faster than the runtime of *contains* method.
* The runtime of *remove* method in AVL Tree grows faster than the runtime of its *contains,* while *remove* in Cuckoo Hashing increases at the same rate as its *contains.*

# Summary

* The data suggests that the runtime of *AVL Tree* Tester is O(Nlog N). As explained, since there are N times calling to AVL Tree methods after applying deductions the runtime of AVL Tree methods (insert, contains, and remove) should be O(logN)
* Insertion and removal in AVL Tree run slower than its contains method
* it clearly shows that Cuckoo Hashing’s runtime is much faster than AVL Tree runtime for all three operations: insert, contains, and remove.
* Insertion method in Cuckoo Hashing is O(N); contains and remove method are O(1);