HashMap Implementation using variants of Binary Search Trees

Abhinandan Sampathkumar
School Of Informatics and Computing
Indiana University Bloomington
Bloomington, Indiana 47405
Email: asampath@indiana.edu

Raghuveer Raavi
School Of Informatics and Computing
Indiana University Bloomington
Bloomington, Indiana 47405
Email: rraavi@indiana.edu

Supreeth Shivanand
School Of Informatics and Computing
Indiana University Bloomington
Bloomington, Indiana 47405
Email: sushivan@indiana.edu

***Abstract*—A hash table uses a hashing function to get a value which is nothing but an index into a bucket, these buckets/array hold the original values. HashMap and HashSet are Java Implementations of Hash Table for Map and Set Interfaces. These Hash Table Implementations use a list to store items in a bucket. When we come across a hash collision, the slots will store data in a list. This list has O(n) insertion, deletion and search.**
The idea is to use a Binary Search Tree variant like AVL tree or Red Black Tree which can reduce these operations to O(log n). We are planning to do performance analysis for different initial capacity and load factors and see how they influence the time complexity and space complexity. We are also looking to address the implementation ease/difficulty of this method over available implementations. Furthermore, we would be discussing the pros and cons of implementation of hash table with trees over a list.

I. Introduction

Consider a scenario where we wish to design a system to store student records of a university keyed by using their student IDs and can perform various operations such as Insertion, Deletion and Searching. There are various data structures to maintain this information such as Array, Linked List, Binary Search Tree and Direct Address Table. With arrays and linked lists, searching is done in a linear fashion and is very costly in practice. The search operation can take up to $\mathcal{O}(\log n)$ time using Binary Search and the insert and delete operations would be much costlier.
With direct address tables, all operations would take $\mathcal{O}(1)$ time and this solution is best among all of them. In this implementation we use an array in which each position or slot corresponds to a key, or in our case, it is student ID. We shall also assume that no two students have the same student ID. This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if student ID is of n digits, we need $\mathcal{O}(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to the above mentioned limitations direct address tables cannot always be used.Therefore now we look at Hashing. Hashing is the solution that can be used and performs extremely well when compared to arrays, linked lists, binary search tree or direct address tables.

II. Implementation

A Hash map, also commonly referred to as Hash table, is a data structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. With hashing we can get $\mathcal{O}(1)$ time to perform search operation.A hash function is a function that can be used to map data of arbitrary size to data of fixed size. The values returned by this function are called hash values, hash codes or simply hashes.

Ideally, the hash function will assign each key to a unique bucket.The entry in the hash table will be NIL if no existing record has hash function value equal to the index for the entry. It is possible that two keys will generate an identical hash causing both keys to point to the same bucket. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and is to be handled using some collision handling technique. Following are the ways to handle collisions:

- Chaining: The main idea of this implementation is to have the new values which collide in a linked list. The implementation is really simple but requires additional memory outside the table.
- Open Addressing: In open addressing, all elements are stored in the hash table itself. Whenever collision occurs we scan linearly within the table and use the bucket that is empty. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Collision resolution by chaining:

Hash collisions are unavoidable even when hashing a random subset of a very large set of keys. For example, if 5000 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem the chances of them being hashed to same bucket is high, approximately 95%.

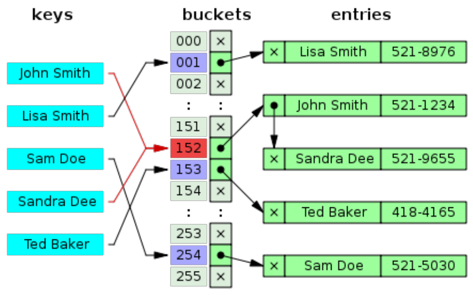


Fig. 1. [1]Collision resolution by Chaining.

Therefore, we need some collision handling strategies for hash implementations to handle such events. One of the most used strategy, chaining using linked lists is described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

1) Separate chaining with linked lists: Chained hash tables with linked lists are probably one of the easiest implementations and are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods. For separate-chaining, the worst-case scenario is when all entries collide onto the same bucket, in which case we add them to a linked list of the same bucket, in which case the hash table is ineffective and the cost is that of searching procedure may have to scan all its entries, so the worst-case cost is proportional to the number n of entries in the linked list.

2) Collision resolution with self balanced binary search tree: As we have seen before, in the worst case scenario the cost is directly proportional to the number of entries n in the linked list, which is $\mathcal{O}(n)$. Therefore our main idea is to bring this worst case cost scenario from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ by resolving the collision using self balanced binary search tree.

A self balanced binary search tree is any such binary tree which can automatically balance and keep it's height to the minimum whenever we insert or delete an element. The examples of such self balancing binary search trees are Red Black tree and AVL tree.

Red-Black tree: Red-black tree is a kind of self balancing tree in which each node of the binary search tree has one extra bit, representing the color of the binary search tree. This color attribute of the binary search tree helps us to ensure that the height is always balanced. The worst case cost of a binary search tree is whenever the tree is either right skewed or left skewed, in which case the cost will be of $\mathcal{O}(n)$. But in case of Red Black tree it is always $\mathcal{O}(\log n)$ since we ensure that the tree is always balanced. The insertion and deletion operations are also performed in $\mathcal{O}(\log n)$ time where n denotes the number of nodes in the tree.

AVL tree: An AVL tree (Georgy Adelson-Velsky and Evgenii Landis' tree, named after the inventors) is another such self-balancing binary search tree. At any point of time, heights of left and right subtrees differ by at most one. If at any point during, if it violates this property it re-balances the tree by performing either right rotation or left rotation. The AVL trees are more balanced than the red-black trees but they may cause more rotations during insertion or deletion. So if number of insertions and deletion operations are significantly higher, red-black trees are preferred otherwise, AVL trees are preferred over red-black trees [2].

III. Results

As we discussed in the previous sections, the costs related to insertion and deletion operations are significantly higher for AVL trees when compared red-black tree or linked lists. This is because, the tree heights in AVL are more balanced than red black trees. Therefore inserting more number of records into AVL causes the tree to rebalance more often by performing rotate operations which inturn affects costs.

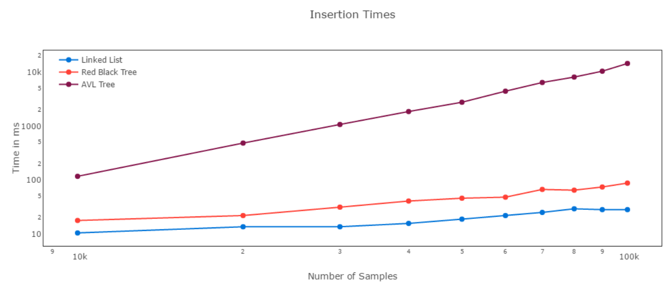


Fig. 2. Figure showing insertion times for HashMaps with various collision handling techniques with samples ranging from 10,000 to 100,000 and bucket size 16. Notice that the insertion times for linear chaining with linked lists is clearly less than red-black or AVL trees.

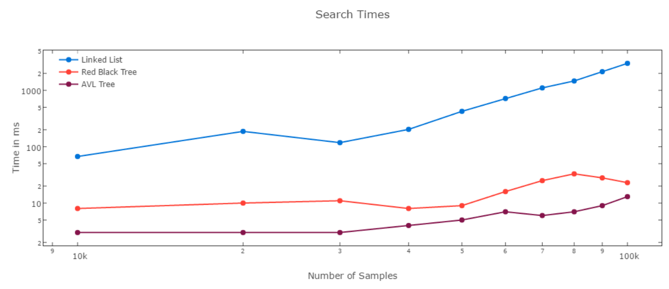
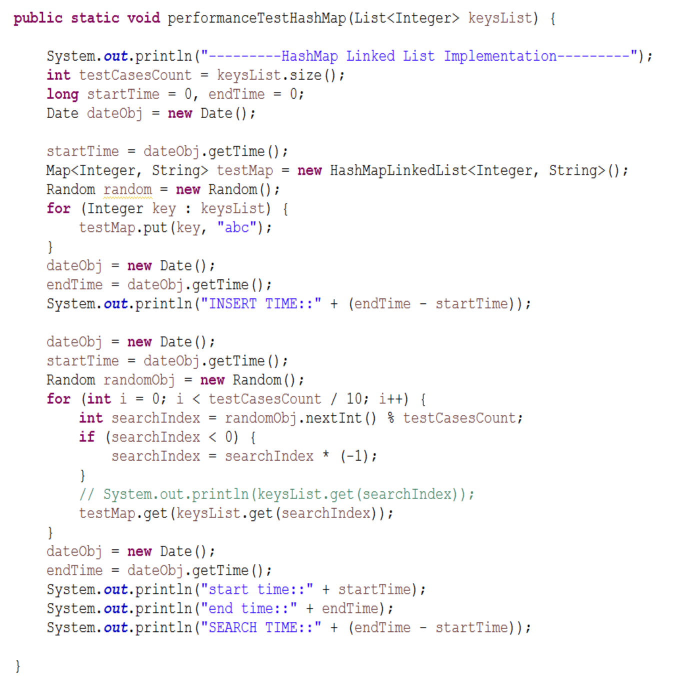


Fig. 3. Figure showing search times for HashMaps with various collision handling techniques with samples ranging from 10,000 to 100,000. Notice that the search times for associated with AVL tree is significantly much lower than the usual linear chaining using linked list technique.

The original implementation of HashMaps used linear chaining to handle collisions. This implementation effects costs when the data set is huge. We may have to search linearly through the linked list for corresponding bucket for which collision occurs. This takes about $\mathcal{O}(n)$ time. The main idea of this paper is to provide a way to bring these costs down to $\mathcal{O}(\log n)$ by using balanced binary search trees. The figure above corroborates exactly to this idea.

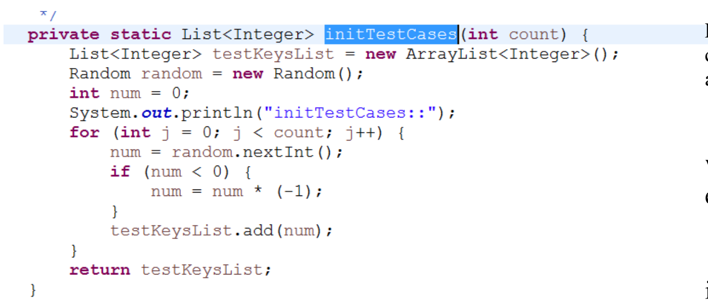
Below module shows how we are evaluate and measure our performance for linked list implementation.



```java
public static void performanceTestHashMap(List<Integer> keysList) {

    System.out.println("---------HashMap Linked List Implementation---------");
    int testCasesCount = keysList.size();
    long startTime = 0, endTime = 0;
    Date dateObj = new Date();

    startTime = dateObj.getTime();
    Map<Integer, String> testMap = new HashMapLinkedList<Integer, String>();
    Random random = new Random();
    for (Integer key : keysList) {
        testMap.put(key, "abc");
    }
    dateObj = new Date();
    endTime = dateObj.getTime();
    System.out.println("INSERT TIME::" + (endTime - startTime));

    dateObj = new Date();
    startTime = dateObj.getTime();
    Random randomObj = new Random();
    for (int i = 0; i < testCasesCount / 10; i++) {
        int searchIndex = randomObj.nextInt() % testCasesCount;
        if (searchIndex < 0) {
            searchIndex = searchIndex * (-1);
        }
        // System.out.println(keysList.get(searchIndex));
        testMap.get(keysList.get(searchIndex));
    }
    dateObj = new Date();
    endTime = dateObj.getTime();
    System.out.println("start time::" + startTime);
    System.out.println("end time::" + endTime);
    System.out.println("SEARCH TIME::" + (endTime - startTime));

}
```

Fig. 4. Figure showing how we are testing the performance for Linked List. The same module is used for Red Black tree and AVL tree implementations as well.

In order to evaluate the performance we need to give test our implementation on some data, preferably in large amounts. In order to that precisely we use the module shown below. Using this module we generate test cases data.
For our implementation purposes, the number of elements we search for will be the total number of test cases divided by 10.



```java
 */
private static List<Integer> initTestCases(int count) {
    List<Integer> testKeysList = new ArrayList<Integer>();
    Random random = new Random();
    int num = 0;
    System.out.println("initTestCases::");
    for (int j = 0; j < count; j++) {
        num = random.nextInt();
        if (num < 0) {
            num = num * (-1);
        }
        testKeysList.add(num);
    }
    return testKeysList;
}
```

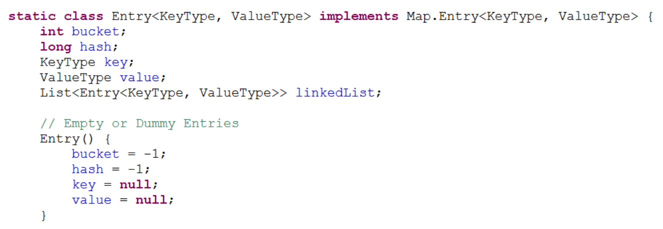
Fig. 5. Figure showing how we generate test cases for test our implementation.

IV. Analysis

1) There is a clear trade off between insertion time and search time when considering self balancing binary search trees and linked lists.
2) The search costs for AVL tree is significantly much better than the red black tree and linked list.

V. Individual Work

I have implemented collision resolution by linear chaining by using Linked lists to resolve the collisions occurring at a particular bucket. Whenever there is a collision, we simply add the element to the linked list of the same bucket for which it collides with.
The below is the module that shows the implementation of Linked List.



```java
static class Entry<KeyType, ValueType> implements Map.Entry<KeyType, ValueType> {
    int bucket;
    long hash;
    KeyType key;
    ValueType value;
    List<Entry<KeyType, ValueType>> linkedList;

    // Empty or Dummy Entries
    Entry() {
        bucket = -1;
        hash = -1;
        key = null;
        value = null;
    }
```

Fig. 6. Figure showing the strucutre of Linked List.

The number of collisions occurring in our hashmap hugely depends on the size of the bucket we have. If the size is less, then more number of collisions will occur and if the size is less, then relatively less or few number of collisions will occur. Below two figures exactly show that by plotting a graph with varying bucket sizes.

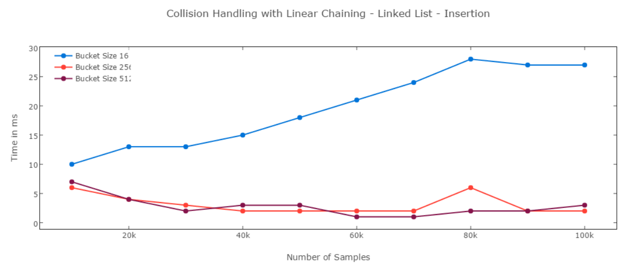


Fig. 7. Figure showing insertion times for HashMaps with linear chain collision handling techniques using linked list ranging from 10,000 to 100,000 and varying bucket sizes.

Here we can clearly notice that for the implementation with less bucket size, the time taken to insert and search the element varies quite significantly.

Below are the tables that shows us time taken for insertion and searching operations of varying bucket sizes for collision resolution by linear chaining using linked lists.

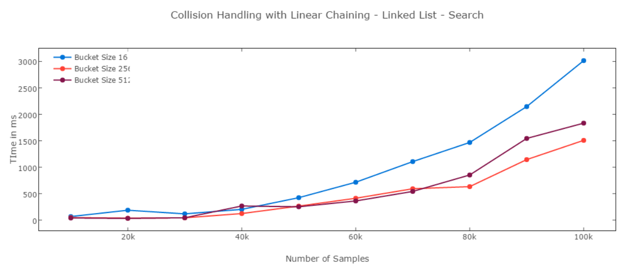


Fig. 8. Figure showing search times for HashMaps with linear chain collision handling techniques using linked list trees ranging from 10,000 to 100,000 and varying bucket sizes.

TABLE I
INSERTION AND SEARCH TIMES
FOR LINEAR CHAINING USING LINKED LIST
WITH BUCKET SIZE 16 AND TIME IN MS.

Number of samples	Insertion time	Search time
10000	10	67
20000	13	187
30000	13	118
40000	15	203
50000	18	424
60000	21	716
70000	24	1106
80000	28	1468
90000	27	2146
100000	27	3015

TABLE II
INSERTION AND SEARCH TIMES
FOR LINEAR CHAINING USING LINKED LIST
WITH BUCKET SIZE 256 AND TIME IN MS.

Number of samples	Insertion time	Search time
10000	6	48
20000	4	37
30000	3	43
40000	2	124
50000	2	265
60000	2	413
70000	2	593
80000	6	633
90000	2	1144
100000	2	1508

TABLE III
INSERTION AND SEARCH TIMES
FOR LINEAR CHAINING USING LINKED LIST
WITH BUCKET SIZE 512 AND TIME IN MS.

Number of samples	Insertion time	Search time
10000	7	40
20000	4	33
30000	2	44
40000	3	267
50000	3	252
60000	1	362
70000	1	542
80000	2	853
90000	2	1544
100000	3	1833

VI. CONCLUSION

From the above figures and tables shown in the section - III, it is clearly noticeable that using Linear chaining with AVL trees, the cost associated with search operation is less when compared to red black and linked list.

Using Linear chaining with RBT, the cost associated with insertion operation is less when compared to AVL Trees.

ACKNOWLEDGMENT

The authors would like to thank Professor *Andy Somogyi* for his knowledge and help, without him we wouldn't be able to implement this project.

REFERENCES

[1] Hash Table - Wikipedia
[2] Red black trees - Set - 1, *www.geeksforgeeks.org* and *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.