International Journal of Hybrid Information Technology
Vol.8, No.3 (2015), pp.185-194
http://dx.doi.org/10.14257/ijhit.2015.8.3.18

A Hybrid Chaining Model with AVL and Binary Search Tree to Enhance Search Speed in Hashing

[1]Akshay Saxena, [2]Harsh Anand, [3]Tribikram Pradhan and [4]Satya Ranjan Mishra

[1, 2, 3] Department Of Information and Communication Technology (ICT)
Manipal University, Manipal - 576014, Karnataka, India
[4]School of Information Technology and Engineering (SITE)
VIT University, Vellore, Tamilnadu, India
[1] aky.saxena@gmail.com, [2]harshanand007@yahoo.co.in,
[3]tribikram.pradhan@manipal.edu, smsatya1@gmail.com

Abstract

The main idea behind any hash function is to find a one to one correspondence between a key value and an index in the hash table where the key value can be placed. In closed hashing, it is very difficult to handle the situation of table overflow in a satisfactory manner. Key values are haphazardly placed and generally majority of the keys are placed far away from their hash location. Thus, the number of probes is greatly increased which degrades the overall performance. To resolve this problem another hashing method known as open hashing (separate chaining) is employed. But this type of hashing is still not efficient in case of searching because here all the elements that have the same hash function are inserted in a sequential order. Due to this, traversal of all the previously inserted elements is required when we are searching for the last element. So the overall complexity will be O (n). In this paper, we propose a hybrid chaining model which is a combination of Binary Search Tree and AVL Tree to achieve a complexity of O (log n).

Keywords: *Hashing, AVL Tree, Binary Search Tree (BST), Separate Chaining Method, Hybrid Chaining Model*

1. Introduction

Hashing is a searching technique where we have to search and insert the values in a Table which is called the hash Table. A function which transforms or maps a key into a Table index is called the hash function. If x is the key and m is the hash Table size and h is the hash function then h(x) is called the hash of m and is the index where the record with key x should be placed. For example h(x) = key mod 1000. The values produced by h should cover the entire set of indices in the Table. For example the function x mod 1000 can produce any integer between 0-999, depending upon the value of x. The implementation of hash Table is called hashing. Generally, mapping defined by hash function will be many to one mapping *i.e.,* there will exist many pairs of distinct keys x and y, such that, their h(x) =h(y). In other words, two or more record keys are mapped to the same array index. This situation is called collision. There are so many existing approaches to deal with collisions such as Separate chaining method, Linear probing, Quadratic probing, double hashing or re hashing *etc.*

1.2 Properties of a Good hash Function

1. It avoids collisions. In practice unless we know something about the keys chosen we can't guarantee that there will not be collision. However, in certain applications we have some specific knowledge about the keys that we can use to reduce the likelihood of collision.

ISSN: 1738-9968 IJHIT

2. A good hash function tends to spread keys evenly in the array. The hash values computed by the function are uniformly distributed. An equal number of keys should map into each array position.
3. A good hash function is easy to compute. It means that the running time of hash function is O (1).

1.3 Avl Trees

An AVL tree is another balanced Binary Search Tree. Named after their inventors, AdelsonVelskii and Landis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an $O(\log n)$ search time. Addition and deletion operations also take $O(\log n)$ time.

An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

A binary search tree is said to be a height balanced tree (BST) if all its nodes have a balance factor of 1, 0, and -1.

1.4 Binary Search Tree

A binary search tree is a binary tree that is either empty or in which every node contains a key and satisfies the conditions.

1. The key in the left child of a node (if it exists) is less than the key in its parent node.
2. The key in the right child of a node (if it exists) is greater than the key in its parent node.
3. The left and right sub trees of the root are again binary search trees.

The first two properties describe the ordering relative to the key in the root node and the third property extends them to all the nodes in the tree. Hence, we can continue to use the recursive structure of the binary tree. Generally the movement is either from left sub tree or the right sub tree from the root. And that sub tree should always follow BST properties. This definition assumes that no duplicate keys are permitted.

One of the most important operations in data structure is searching. Here we have to follow one procedure to search a particular target key throughout a Binary Search Tree. To search for any target key, first we make a comparison with a key. If it is the same, then we are done. If it is not the same we have to move either to the left sub tree or right depending on the values of the search keys. We have to repeat this process for the entire sub tree until we find the keys.

For example, if we want to search the node “19” in the BST of Figure 1. We first compare “19” with “43”, since “43” is greater than “19” we have to move to the left sub tree of “43”. Then again we compare “19” with “32”. Once again “32” is greater than “19”, so we have to move to the left sub-tree.

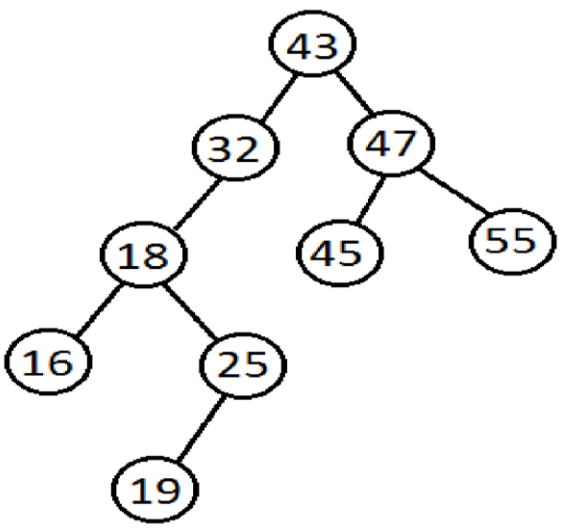

Figure 1. Binary Search Tree

Now the next node is "18" but "18" is less than the key to be searched, *i.e.*, "19" so we move to the right sub-tree. Then the node "25" is reached, so we move to the left sub-tree. Now, the current key ("19") matches with the key to be searched "19". So, we stop at this step as the key is found. If we find the key, then searching terminates successfully. If not, we have to continue the searching procedure until we reach an empty sub tree (*i.e.*, where a null pointer is encountered).

2. Existing System

2.1 Collision Resolution

Collisions are treated differently in different methods. A data set with key S is called a colliding element if bucket B h(s) is already taken by another data set. What can we do with colliding elements?

2.2 Chaining

Implement the buckets as linked lists. Colliding elements are stored in these lists.

2.3 Open Addressing

Colliding elements are stored in other vacant buckets. During storage and lookup, these are found through so-called probing.

2.4 Problem: Obviously, a mapping from a potentially huge set of strings to a small set of integers will not be unique. The hash function maps keys into indices in many-to-one fashion. Having a second key into a previously used slot is called a collision.

2.5 Separate Chaining: In this approach, we install a linked list at each index in the hash Table. A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index. Other items that hash to the same index are simply added to the linked list; there is no need to search for empty cells in the primary array. This method is called separate chaining, because items that collide are chained together in separate linked lists. If there are many items, access time is reduced because access to a specified item requires searching through an average of half the items on the list. Finding the initial cell takes fast O (1) time, but searching through a list takes time proportional to the number of items on the list; O (M) time. Thus we do not want the list to become too full. To perform a

search, we use the hash function to determine which list to traverse. We then search this list. To insert an item, we check the appropriate list and insert it at the front of the list.

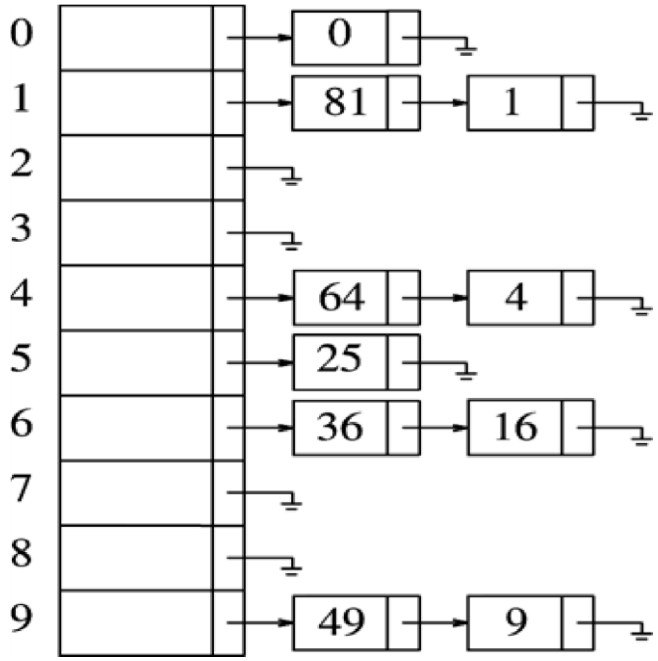

Figure 2. Example of Separate Chaining Method

2.6 Complexity of Separate Chaining

The time to compute the index of a given key is a constant. Then we have to search in a list for the record. Therefore the time depends on the length of the lists. It has been shown empirically that on average the list length is N/M (the load factor L), provided M is prime and we use a function that gives good distribution. Unsuccessful searches go to the end of some list, hence we have L comparisons. Successful searches are expected to go half the way down some list. On average the number of comparisons in successful search is L/2. Therefore we can say that runtime complexity of separate chaining is O (L).

2.7 How to Choose M in Separate Chaining?

Since the method is used in cases when we cannot predict the number of records in advance, the choice of M basically depends on other factors such as available memory. Typically M is chosen relatively small so as not to use up a large area of contiguous memory, but enough large so that the lists are short for more efficient sequential search. Recommendations in the literature vary from M to be about one tenth of N- the number of the records to M to be equal (or close to) N.

2.8 Other Methods of Chaining:

1. Keep the lists ordered: useful if there are more searches than inserts and if most of the searches are unsuccessful.
2. Represent the chains as binary search tree. Extra effort needed – not efficient.

Advantages of Separate Chaining– used when memory is of concern, easily implemented.
Disadvantages –unevenly distributed keys long lists and many empty spaces in the Table.

3. Proposed System

In this paper, we have proposed a hybrid model which is a combination of Binary Search Tree (BST) and AVL tree. First, we implement the hash Table by an AVL tree.

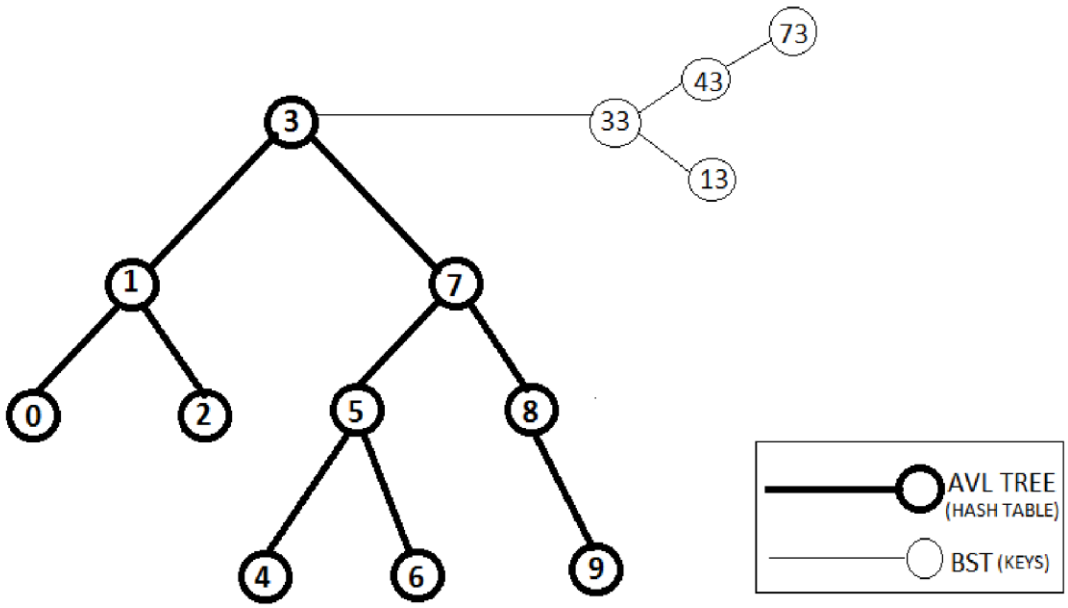

Figure 3. Hybrid Chaining Model to Enhance Search Speed in Hashing

Then we add the keys to be stored in the hash Table inside a BST originating at that location in the hash Table (implement in the form of an AVL Tree). There is no usage of linked lists (as in the case of separate chaining) originating from each hash location in the hash Table, as BSTs are used instead of linked lists. In the Separate Chaining method, search operation is not so efficient because whenever there is a collision, we store the key value in the data part of a new node and then join previous key value's node to the new key value's node. This process continues for every collision that occurs at the same location in the hash table. Suppose there are 7 elements to be inserted in a hash Table: 40, 20, 10, 30, 22, 50 and 60. According to normal separate chaining method we have to insert 40, 20, 10, 30, 50 and 60 in a single chain sequentially because all of these have same hash value. To search the key 60, we have to traverse each key in a sequential manner. So we need a time complexity of O (n), where n is the number of elements traversed. To avoid this situation we propose a hybrid model which is called hybrid chaining (HC).

This model is a combination of AVL trees and BSTs. The hash Table has to be implemented by AVL tree method and the collided nodes by BSTs. For example, in the above case, hash Table is formed in the form of an AVL tree and the values 40, 20, 10, 30, 50 and 60 (say they have the same hash function value) after collision will form a BST originating from AVL tree. And now if we wanted to search 60 then this would take time complexity of O (log n). The worst case of this proposed model is O (n) when the element inserted at that location after collision is either in increasing or decreasing order.

4. Methodology

In this paper we have replaced the Hash Table with AVL Tree methodology because suppose there are 10 nodes in the hash Table then according to normal separate chaining, if we want to insert a value at position determined by hash function, we have to traverse all the previous positions sequentially which is a time taking process.

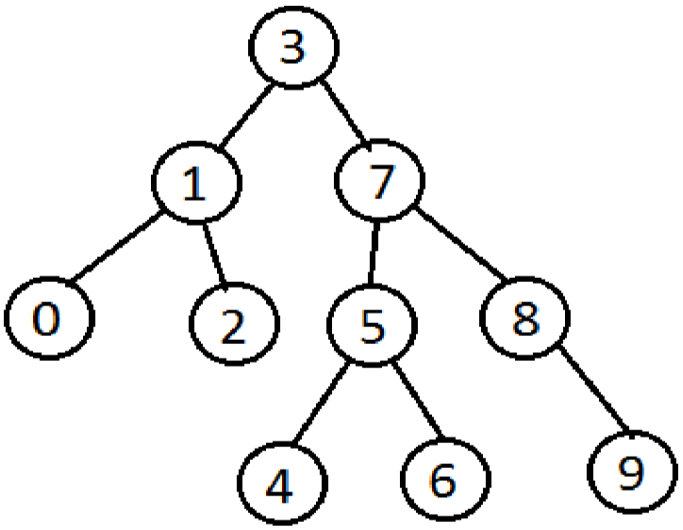

Figure 4. AVL Tree Formation

Here, this situation will not occur because we have inserted the nodes according to AVL Properties. For *e.g.,* to create a hash Table of size 10, we follow the AVL tree properties to create the hash Table (tree).
The Insertion of the following data 0,1,2,3,4,5,6,7,8,9 will be in the following way:
Step1: Input is taken from the user
Step2: Input is inserted in the hash-table (implemented in the form of AVL-tree).
Step3: AVL tree is reconfigured after adding new node with input value.
Step4: steps 1, 2, 3 are repeated until no more hash table entries are required.
Step5: Hash Table is created.

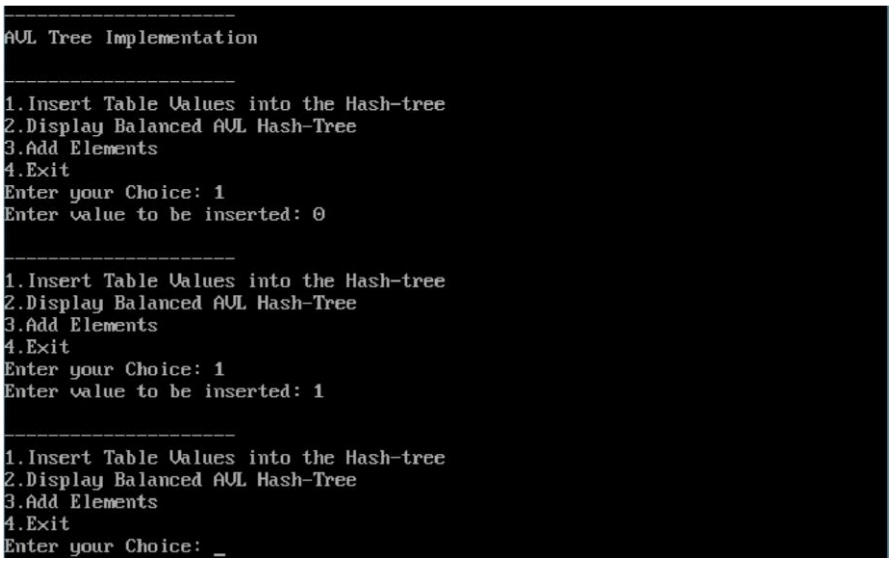

Figure 5. Initial AVL Tree Implementation Model

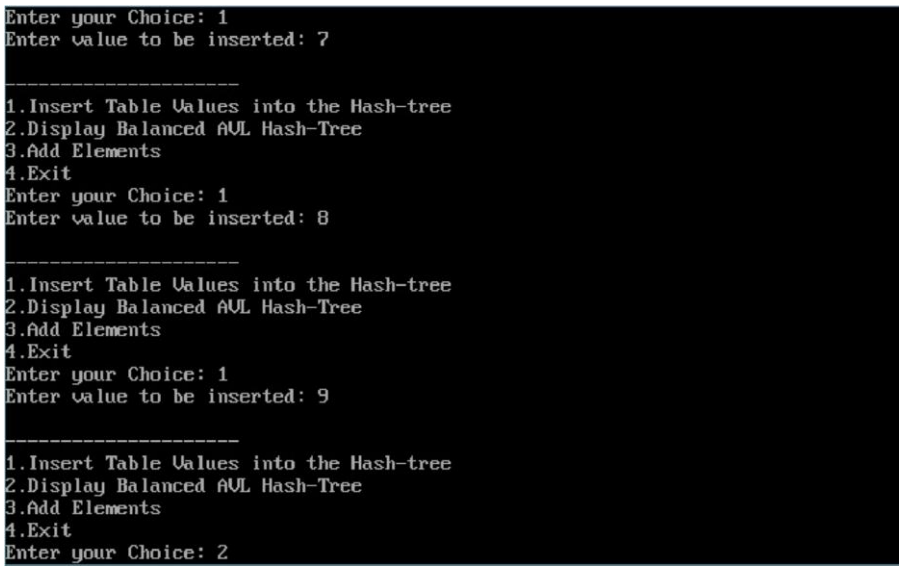

Figure 6. Insertion Process in AVL Tree Method

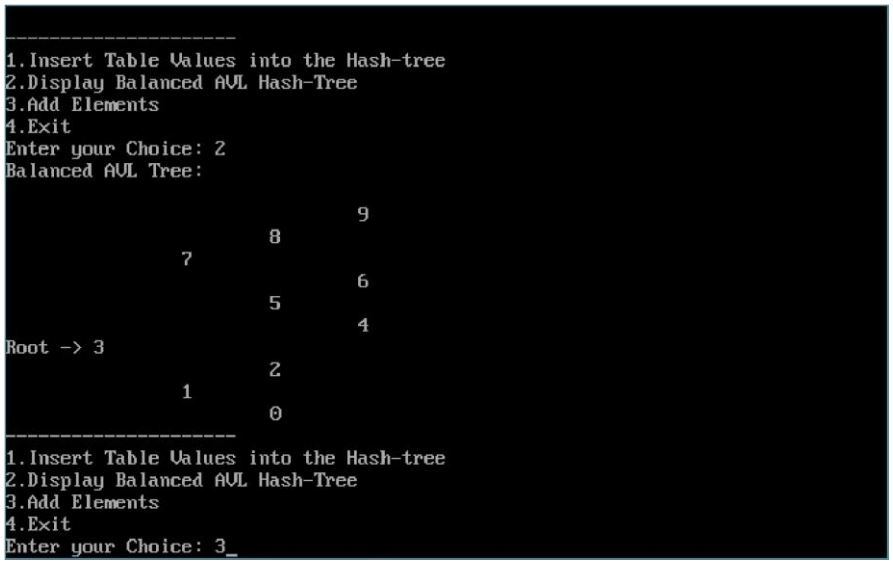

Figure 7. Final AVL Tree Based hah Table Formation

Now, suppose we have to insert the values 33, 13, 43, 73 into the hash Table, which is of size 10, we have to perform hashing function: h(x) will be x % m, where x is the key value which is to be inserted and m is the hash table size (in this example m=10). So according to the above formula, we will get a remainder 3 for each of the key values. So now we have to perform insertion by using hybrid chaining method and based on that we get the following result:

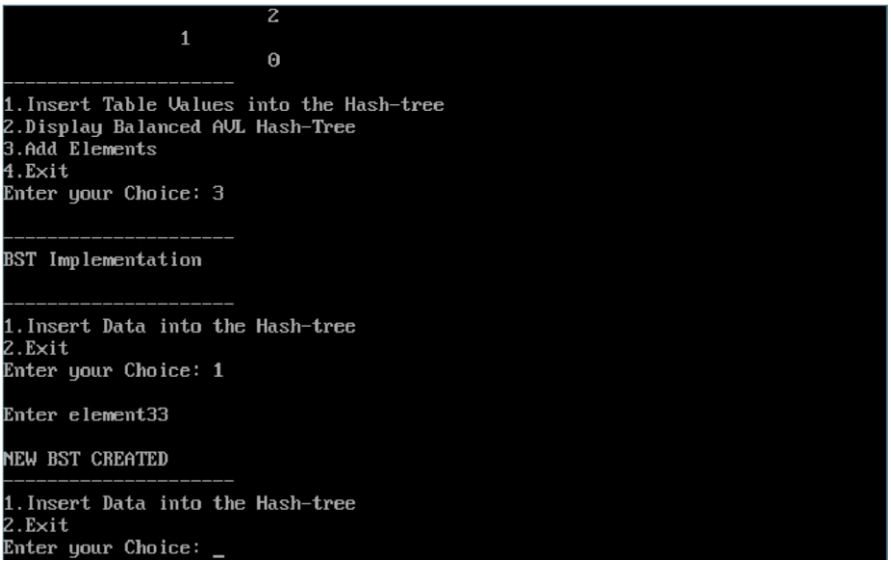

Figure 8. Insertion of Keys 33 into Hash Table

After giving the input 33 in location 3 we have to consider 33 as the root node. Suppose now we want to insert 13, which is less than 33.so we have to insert it on the left sub tree of the root node 33.

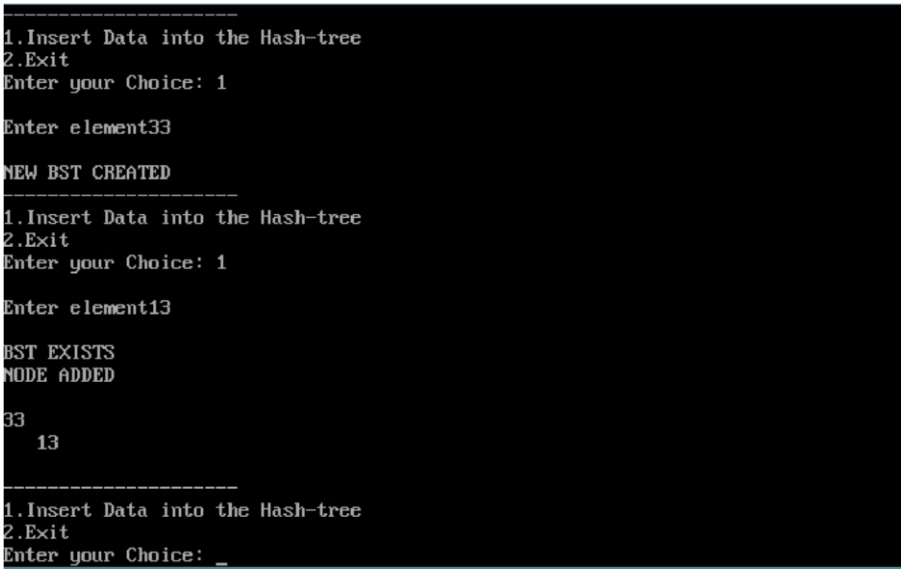

Figure 9. Insertion of Keys 13 into Hash Table

After inserting key value 13, suppose we want to insert 43 then we have to compare with root node 33, which is greater than 33. So insert it on the right sub tree of the root node 33.

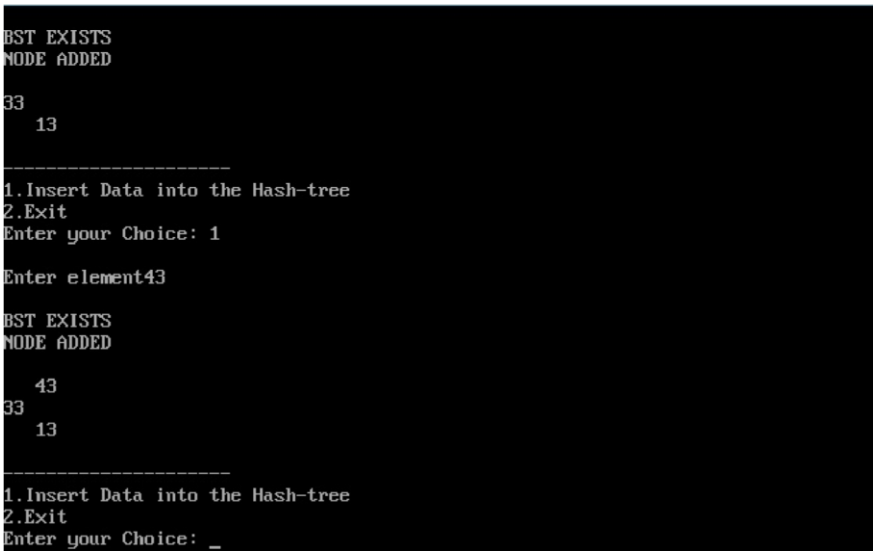

Figure 10. Insertion of Keys 43 into Hash Table

After the insertion of node 43 the implementation structure will be like Figure 6. Finally take the key values 73 and try to insert it in the hash table. The key value 73 is greater than the root so insert it on the right hand side of the root. And the overall insertion result is depicted in Figure

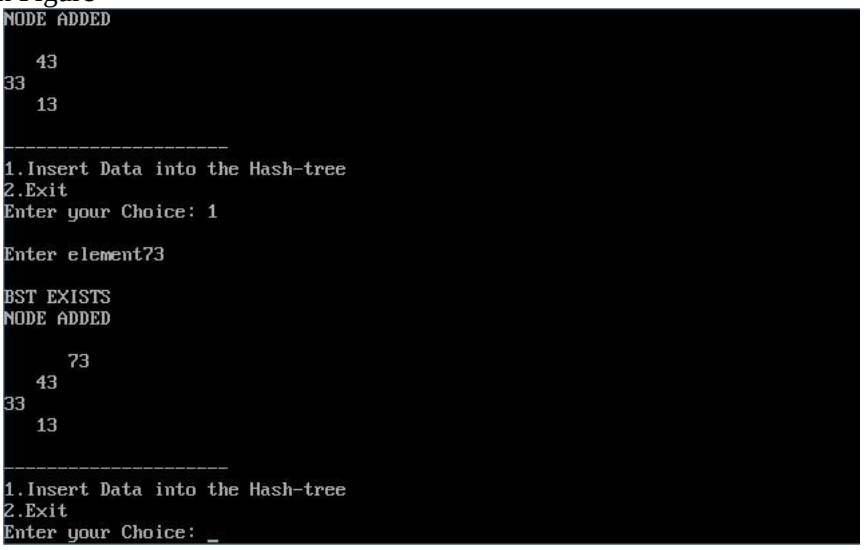

Figure 11. Insertion of Keys 73 into Hash Table with Final Implementation Details

References

[1] D. Knuth, "The Art of Computer Programming", Reading, Mass.: Addison-Wesley, vol. 3, **(1973)**.
[2] M. V. Ramakrishna, "Hashing in Practice, Analysis of Hashing and Universal Hashing," Proc. ACM SIGMOD Conf., **(1988)**, pp. 191-199**.**
[3] L. Carter and M. Wegman, "Universal Classes of Hashing Functions," J. Computer and System Sciences, vol. 18, no. 2, **(1979)**, pp. 143-154**.** [CrossRef]
[4] M. Benhase, "Resetting Storage Unit Directories," IBM Technical Disclosure Bulletin, vol. 25, no. 7B, **(1982)**, pp. 3,760-3,761.
[5] H. Robinson and G. Taylor, "Hashing Addresses to a Cache on DASD," IBM Technical Disclosure Bulletin, vol. 24, no. 11A, **(1982)**, pp. 5,354-5,356.
[6] R. Bryant, "Extendible Hashing for Line-Oriented Paging Stores," IBM Technical Disclosure Bulletin, vol. 26, no. 11, **(1984)**, pp. 6,046-6,049**.**

[7] K. Ramamoganarao and R. Sacks-Davis, "Hardware Address Translation for Machines with a Large Virtual Memory," Information Processing Letters, vol. 13, no. 1, **(1981),** pp. 23-29**.** [CrossRef]
[8] M. Houdek and G. Mitchell, "Translating a Large Virtual Address," IBM Systems/38 Technical Developments, **(1978)**, pp. 22-24.
[9] J. Cocke and W. Worley, "Virtual to Real Address Translation Using Hashing," IBM Technical Disclosure Bulletin, vol. 24, no. 6, **(1981)**, pp. 2,724-2,726.
[10] A. Chang and M. Mergen, "801 Storage: Architecture and Programming," ACM Trans. Computer Systems, vol. 6, no. 1, **(1988)**, pp. 28-50.
[11] J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machine," Proc. 20th Int',l Symp. Computer Architecture, **(1993)** May, pp. 39-50.
[12] G. Gonnet, "Expected Length of the Longest Probe Sequence in Hash Code Searching," J. ACM, vol. 28, no. 2, **(1981)**, pp. 289-304.
[13] P. Larson, "Expected Worst-Case Performance of Hash Files," The Computer J., vol. 25, no. 3, **(1982)**, pp. 347-352.
[14] M. V. Ramakrishna, E. Fu and E. Bahcekapili, "Efficient Hardware Hashing Design for High Performance Computers," Technical Report TR-96-13, Computer Science Dept., RMIT, Melbourne, Australia, **(1996)**.
[15] D. Knuth, “The Art of Computer Programming”, Reading, Mass.: Addison-Wesley, vol. 2, **(1969)**.

