IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 3, Ver. VIII (May-Jun. 2014), PP 47-51
www.iosrjournals.org

Modulo Search Trees (Plain and Z-Overlapped)

Prasad V Chaugule
(Former Student, Dr. Babasaheb Ambedkar Technological University, India)

Abstract: *This paper exploits the modulo division operation to construct a tree which is termed as Modulo search tree with two versions out of which one is coined as Plain Modulo Search Tree and the other is termed as Z-Overlapped Modulo search Tree . The use of the term Modulo search tree implicitly implies the plain version of the tree. Searching Data in Modulo search tree will have better complexity as compared to the Binary Search tree. But the plain Modulo search tree structure requires more storage space to implement. Albeit, in contrast to the plain version of the Modulo search tree, Z-Overlapped version addresses the issue of extra storage and minimizes the storage requirement. And hence provides a flexibility (with the value of Z) to the implementer to choose a mid path between search time and storage space as and when required.*
Keywords: *Modulo search tree, Overlapping constant, Z-Overlapped Modulo search tree*

I. INTRODUCTION

Searching and Sorting are the most frequently encountered operations in various domains of Computer Science and Engineering. The most obvious and intuitive searching is linear but is not a popular choice of many because of its linear time complexity. Many better alternatives for linear searching are evolved over the years. A famous alternative is Binary Search Tree with Binary logarithmic time complexity which is widely used to store and search data. A better data structure for efficient storing and searching of data is presented here which is called as Modulo search tree. The concept used in constructing this data structure is really easy and intuitive. Before understanding the concept of Modulo search tree, we will first exploit a single and very intuitive property of Modulo division. The property says that whenever we divide a positive non zero integer number M by any natural number N, the remainder is always a member of a finite set containing natural numbers from zero to (N-1) [1]. In short, any integer when divided by 2 will always yield a remainder which is either zero or one. Similarly, any integer when divided by 6 will always yield a remainder which is an element from the set {0, 1, 2, 3, 4, 5}[1]. This is the crux in the formation of Modulo search trees. We will use this property to define both Plain Modulo search tree and Z-Overlapped Modulo search tree.

II. DEFINITION OF MODULO SEARCH TREE (PLAIN)

Before formally defining Modulo search tree, we will define the node structure of a Modulo search tree. Unlike Binary search trees, where each node has three fields, viz. data field, left pointer field and right pointer field [2], in Modulo search tree, the data field is always present in every node but the number of link fields in the node are dynamic in nature and it solely depends on the data value which is put into the node. A node in Modulo search tree cannot be defined without considering the data value it is holding. Let us assume that a node in Modulo search tree is holding a non zero positive integer K, then there are K link fields in this node and each link field is capable of pointing to another Modulo search tree just like in case of Binary search tree[1][2]. Fig.1 represents the node structure of Modulo search tree.

After discussing the node structure of a Modulo Search tree, let us now formally define Modulo search tree. A modulo Search tree T is a tree having no nodes at all (NULL tree) or has a special root node designated as R holding data value K and having K link fields designated as (0, 1, 2 , …, K-1), such that all of these K link fields again points to some Modulo search sub trees (may be NULL) with respect to the root node R such that the sub tree pointed by 0^{th} link field of the root node R contain elements all of the form Kx where x is any whole number, sub tree pointed by 1^{st} link field contain elements all of the form (Kx+1) where x is any whole number and so on.

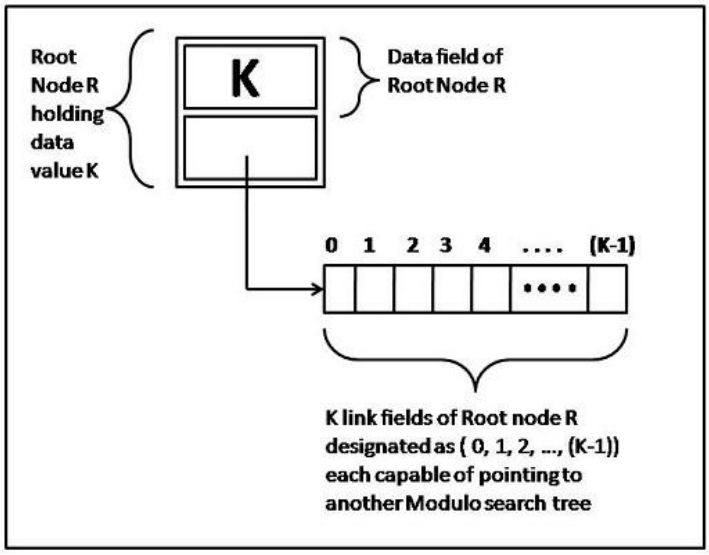

Fig 1: Structure of Root Node R Modulo search tree

In other words, the 0^{th} link of the root node R points a Modulo search tree whose all elements when divided by root value K yields a zero remainder. The 1^{st} link of the root node R points a tree whose all elements when divided by root value K always yields a unit remainder, the 2^{nd} link of the root node R points a tree whose all elements when divided by root value K always yields a remainder of 2, and so on. In general, the L^{th} link of the root node R points a Modulo search tree whose all elements when divided by root value K always yields a remainder of L. Hereafter, Modulo search tree is also referred as MST. It is easy to find out that, searching an element is efficient in MST as compared to that of Binary Search Tree. The most important point to note here is that the number of link fields of each node in Modulo search tree is not uniform and hence, the structure of such trees could not be compared with the structure of any one of m-ary trees like Binary, Ternary or so on . The structure of MST is a mixture of structures of various m-ary trees. The search operation in the Modulo search tree is really efficient and fast. Informally, we can argue that since the number of links in each node of the Modulo search tree are large, the resulting structure will be somewhat fat and short rather than thin and tall and hence, the number of comparisons required to search any element in this structure will be less as compared to any other searching methods including Binary search. Of course, like any other data structure, the search time in Modulo search tree also primarily depends on the sequence of the data values that are fed into the MST. The only but crucial drawback of Modulo search tree is its storage requirement. Let us assume that a node N of Modulo search Tree T is holding a very large data value V and hence will have V number of link fields. Also, the probability of using all of these link fields is very less. Perhaps, most of the link fields out of V will be holding NULL values for most of the time resulting in wastage of crucial storage space in the machine. One could always use Modulo search trees when ample storage requirement is available and efficient searching has higher priority than the efficient storage utilization. The ideas of MST could be easily used in situations where the data once stored remains static and the number of data items are very large but their data values are not so large. Fig 2 represents an example of Modulo search tree.

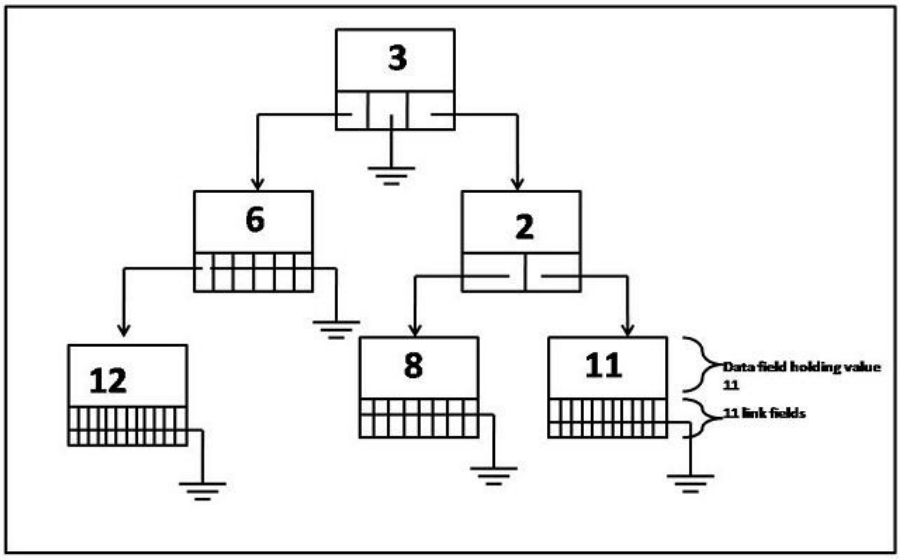

Fig 2: An example of Modulo search tree

III. DEFINITION OF Z-OVERLAPPED MODULO SEARCH TREE

A slight modification to the above definition of Modulo search tree defines a different version of Modulo search tree which is called as Z-Overlapped Modulo search tree (Z-ary Modulo search tree). This modification not only overcomes the drawback of excessive space requirement by Modulo search tree but also provides an opportunity to the implementer to find a mid path between the two extremes which are excessive space requirement for implementation and efficient searching. In Z-Overlapped Modulo search tree, the data field is always present in the node but the number of link fields in the node are static in nature and it solely depends on the value of Z chosen by the implementer. Z is constant for all the nodes in any Z-Overlapped Modulo Search Tree. Z could be any positive integer greater than or equal to one and is called as Overlapping Constant. Fig 3 represents the node structure of a Z-Overlapped modulo search tree. A Z-Overlapped modulo Search tree T is a tree having no nodes at all (NULL tree) or has a special root node designated as R holding any non zero positive integer value K but has exactly Z link fields (irrespective of the data value that the root node R is holding) designated as (0, 1, 2, …, Z-1), such that all of these Z link fields again points to some Z-Overlapped Modulo search sub trees (may be NULL) with respect to node R such that the Z-Overlapped Modulo search sub tree pointed by 0^{th} link field of the root node R contains elements all of the form (Kx + (Zy)) where x and y are some whole numbers and , Z-Overlapped Modulo search sub tree pointed by 1^{st} link field contains elements all of the form (Kx+ (Zy+1)) and so on.

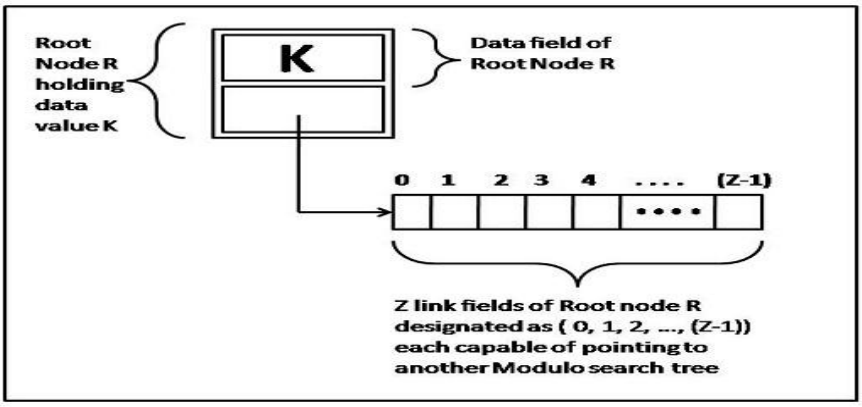

Fig 3: Structure of Root Node R of Z-Overlapped Modulo search tree

In other words, the 0^{th} link of the root node R points a Z-Overlapped Modulo search tree whose all elements when divided by root node value K yields an intermediate remainder which when divided by Z yields a zero remainder. The 1^{st} link of the root node points a Z-Overlapped sub tree whose all elements when divided by root node value K always yields an intermediate remainder which when divided by Z yields a Unit remainder, the 2^{nd} link of the root node R points a tree whose all elements when divided by root node value K always yields an intermediate remainder which when divided by Z yields a remainder of 2, and so on. In general, the L^{th} link

of the root node R points a Z-Modulo search tree whose all elements when divided by root value K always yields an intermediate remainder which when divided by Z yields a remainder of L. A overlapping constant Z when taken as one generates a 1-Overlapped Modulo Search tree whose structure will be exactly same as a linked list. Likewise, a Z-Overlapped MST with the value of Z as 2 is 2-Overlapped MST or 2-ary MST and so on. The structure of Z-Overlapped MST is exactly same as that of Z-ary search tree except that of the node structure. That is, the structure of 3-Overlapped MST will be same as that of Ternary search tree except that of the node structure. The most important feature of Z-Overlapped MST is the flexibility provided to the implementer by the value of Z. The implementer gets a power of changing the complete structure of the tree by merely changing the value of the variable Z . Choosing a small value for Z will reduce the space requirement for implementing the MST but on the cost of search efficiency. On the contrary, a very large value of Z will improve the search complexity but on the cost of extra space requirement. Many other data structures like 2-3 trees[3], B trees[3], AVL trees[3], m-way search trees[3] are used for efficient searching. But as we know that the node structure of each of these trees is very complex with many node values and links. The decision for traversing the correct link from any node requires many comparison operations. The Z-Overlapped modulo search tree is a perfect blend of such efficient trees with the essence of "single node- single data" feature of the Binary Search tree hence making the implementation of Z-Overlapped MST very easy and fast. It imports the thought of making many links to flow out of a single node for improving search complexity but without increasing the node complexity and decision complexity. Less Overlap constant Z reduces the width of the tree and increases its height hence reducing the size requirement for implementing the tree but on the cost of degraded search performance. On the other hand, a bigger overlap constant Z increases the width of the tree and reduces its height hence upgrading the search performance but with a penalty of large storage space for each node. Hence, a overlap constant should be neither a very large nor a very small number. One interesting thing to note here is that a overlap constant of 2 will make the tree structure exactly as that of Binary Search tree. But the search complexity of 2-Overlapped MST is more than that of a Binary search tree if two modulo division operations are considered costly than a comparison operator. Fig 4 represents an example of 4-Overlapped Modulo search tree.

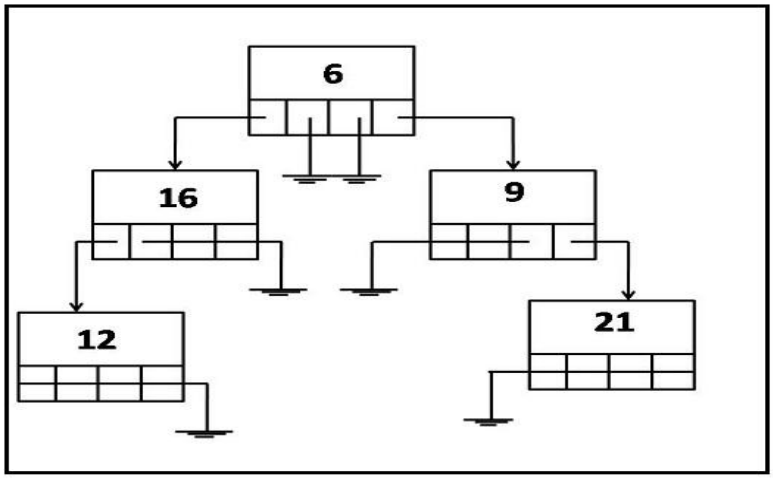

Fig 4 An example of 4-Overlapped Modulo search tree

IV. INSERTION ALGORITHM FOR Z-OVERLAPPED MODULO SEARCH TREE

The following insertion algorithm is presented in a pseudo language having C/C++ like syntax. Let us understand the syntax used in the following insertion algorithm. Here, we are inserting an element item in a non empty Z-ary MST with the address of root node of the tree stored in pointer variable Root. The token mod is used to represent modulus operator. The token Root.link[i] represent the i^{th} link of the node Root.

```
VOID INSERT (Root, item){
Remainder  = (item) mod (value in Data field of Root Node);
Remainder =(Remainder) mod Z;
i = Remainder;
if( Root.link[i] == NULL)
{
Create a New node N;
Put item in data field of node N;
Set Root.link[i] equal to address of N;
```

```
Create Z number of links for node N;
Set all the links of node N to NULL;
}
else INSERT(Root.link[Remainder], item);
}
```

V. SEARCHING ALGORITHM FOR Z-OVERLAPPED MODULO SEARCH TREE

The following search algorithm is represented in a pseudo language having C/C++ like syntax. The syntax used in the following searching algorithm is same as discussed in insertion algorithm . Here, we are searching an element item in a non empty Z-ary MST with the address of Root node of the tree stored in pointer variable Root. The token mod is used to represent modulus operator. The token Root.link[i] represent the i^{th} link of the node Root. The routine returns a TRUE value if the element item is found in the tree otherwise returns FALSE.

```
BOOLEAN SEARCH(Root, item){
if(Data field of Root Node is equal to item)
return TRUE;
else
{
Remainder  = (item) mod (value in Data field of Root Node);
Remainder =(Remainder) mod Z;
i = Remainder;
if(Root.link[i] == NULL)
return FALSE;
else
return( SEARCH(Root.link[Remainder], item) );
}
}
```

VI. TRAVERSING ALGORITHM FOR Z-OVERLAPPED MODULO SEARCH TREE

As we know that traversing a data structure essentially implies visiting its each data item exactly once in some defined fashion. We are going to formally define four different algorithms for traversing Modulo Search tree. The names of the four traversing methods are Root First Left to Right Traversing (hereafter also referred as RFLR), Root First Right to Left Traversing (hereafter also referred as RFRL), Root Last Left to Right Traversing (hereafter also referred as RLLR) and Root Last Right to Left Traversing (hereafter also referred as RLRL). The algorithms for these traversing methods can be easily written by their nomenclature itself. The RFLR traversal visits the Root node of the modulo search tree first and then traverse its Sub-trees from left to right again in RFLR recursively. The RFRL traversal visits the Root node of the modulo search tree first and then traverse its sub-trees from right to left again in RFRL recursively. The RLLR traversal traverse the sub-trees of the root first from left to right in RLLR recursively and then prints the root node. The RLRL traversal traverse the sub-trees of the root first from right to left again in RLLR recursively and then prints the root node. We can easily claim that the RFLR traversal of 4-Overlapped MST in fig. 2 is 3, 6, 12, 2, 8, 11. The RLLR traversal of the same tree will be 12, 6, 8, 11, 2, 3. These four traversing algorithms could also be extended for traversing Z-Overlapped Modulo search tree.

VII. CONCLUSION

A new data structure Z-Overlapped Modulo search tree is defined which can be used to store and search data efficiently. Recursive algorithms for insertion, searching and traversing Z-Overlapped Modulo search tree are presented. Various Deletion schemes could be proposed for this new data structure. Various modifications could be proposed to this same idea to improve the search time further.

REFERENCES

Books:

[1]. Kenneth H Rosen, Discrete Mathematics and its Application with Combinatorics and Graph Theory (India, Tata McGraw-Hill Publishing Company Limited New Delhi, 2007).
[2]. C. L. Liu, Elements of Discrete Mathematics (McGraw Hill International Edition, 1985).
[3]. Seymour Lipschutz, G A Vijaylakshmi Pai, Data Structures (India, Tata McGraw-Hill Education Private Limited New Delhi, 2006)