The TREE LIST – Introducing a Data Structure

Sibin James, Pranav Prakash, R Nandakumar

***Abstract:** The array and the linked list are two classic data structures. The array allows constant time random access (achieved in C language with the [] operator) but suffers from its fixed size and relative inflexibility (the latter becomes an issue while performing deletions). The linked list, on the other hand allows dynamic allocation of memory leading to greater flexibility (manifested in easy insertions and deletions) but suffers from slow search speed - O(N) . We describe a complete binary tree-based data structure which we call TREE LIST. It allows dynamic allocation of memory (hence free from many of the fixed size issues of the array) and provides random access with O(log N) complexity - which is an improvement over the linked list although not as fast as the array. We also discuss some other aspects of this data structure – in particular how it supports some of the classic sorting algorithms.*

***Index Terms**: Tree list, Linked list, Array, Algorithm, Binary tree, Complexity.*

I. INTRODUCTION – DESCRIPTION OF THE TREE LIST

The Linked List is a linear data structure which gains its flexibility from dynamic allocation of data. Its main drawback is lack of random access ie. if want to access the Nth element in a linked list, we can do so only by visiting every previous node in this list – this leads to O(N) [1] complexity.

The array stores its data in a pre-fixed block of memory in contiguous locations. This enables very rapid – constant time - random access and leads to algorithms that depend on random access to run very efficiently. However, the array suffers from lack of flexibility due to its fixed size. The size of the array is predetermined so there is always the possibility of a large portion of its memory never getting utilized – and the danger of the memory proving insufficient to hold the data to be stored. This paper examines a data structure that attempts to strike a balance between the linked list and array – we call it the Tree List ; the data structure is a combination of linked list and binary tree. Recall that in a binary tree, each node holds two pointers to its left and right child and the data.

Revised Manuscript Received on 30 March 2019.

***** Correspondence Author

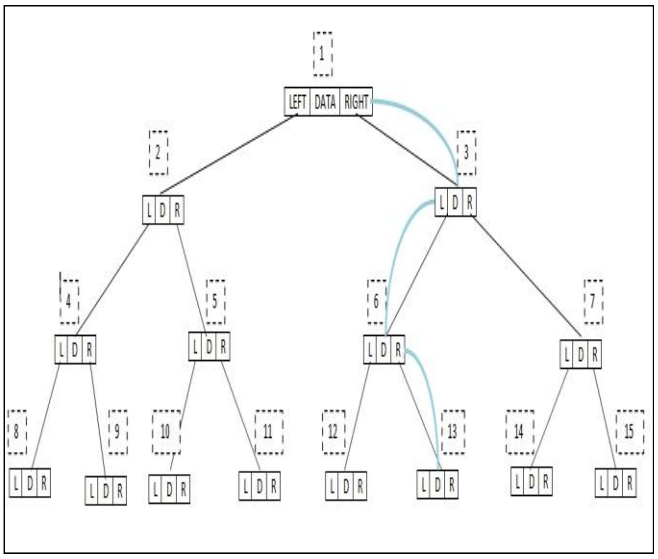
Sibin James*, Department of Computer Science and IT, Amrita School of Arts & Sciences, Kochi, Amrita Vishwa Vidyapeetham, India.

Pranav Prakash, Department of Computer Science and IT, Amrita School of Arts & Sciences, Kochi, Amrita Vishwa Vidyapeetham, India.

R Nandakumar, Department of Computer Science and IT, Amrita School of Arts & Sciences, Kochi, Amrita Vishwa Vidyapeetham, India.

A leaf is a node that has no children. A full binary tree is a binary tree in which every node other than the leaves has two children. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible [2]. We begin with the first data item stored as the root and progressively build the data structure by inserting each succeeding value in that position which maintains the structure as a complete binary tree. For example, the incoming sequence of data (here integers). 5,1,12,7,3,11,7,23,9,14 forms the complete binary tree (Tree list) shown in fig 1. It should be kept in mind that a Tree List cannot be used as binary search tree.

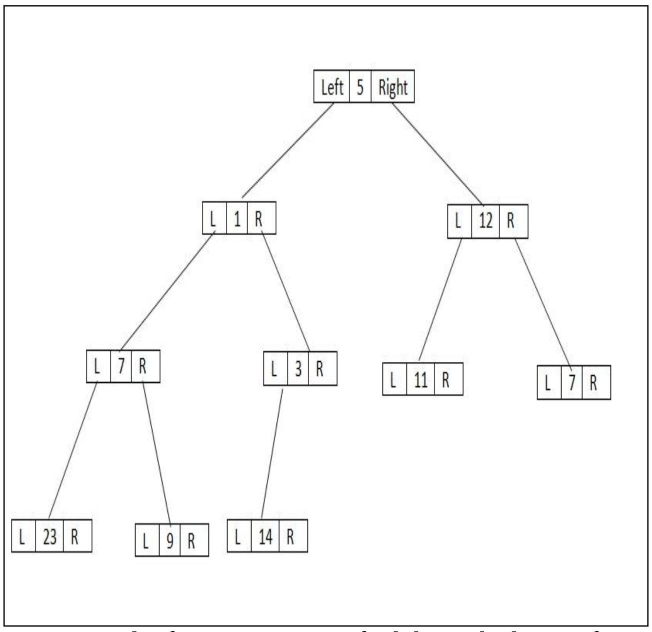
Note: To facilitate appending successive elements to the tree list, we also maintain the total number of elements added so far to the data structure.



(Fig 1)

II. RANDOM ACCESS OPERATIONS IN THE TREE LIST

We first demonstrate the principal property of the Tree List: Given any N, we can find the Nth element in the tree list with O(log N) complexity (note that this is a substantial improvement over the linked list which takes (O(N) time)).



Consider fig 2. We want to find the 13th element for

(Fig 2)

which a linked list would take 13 steps. Note that D indicates the value (variable) of the data stored in a node.

We convert the number 13 into its binary value. This binary value, viewed as a string of 1s and 0s, yields a path to reach the 13th element in the data structure: the first 1 in the binary number string corresponds to the root from where the path begins. Then, as each successive bit is read,

(1) if the bit is 1, we move to the right child of the present element.

(2) if the bit is 0, move to the left child of the present element.

The element we reach when the string is fully read is the element we need.

For 13th element in below tree, the binary representation is 1101. So, our path to the 13th element is beginning at the root, Right-Left-Right and the path has been marked in fig 2. We have reached the 13th element via a path that visits only 4 elements in the data structure as opposed to a linked list where 13 elements would have been visited.

Complexity: For finding the Nth element, we note that the binary representation of a value N has length of only $\log_2 N$. And since this sequence of bits is guaranteed to take us to the element we are trying to teach, we get the complexity of O(log N) for random access in the tree list.

III. SEARCHING IN A TREE LIST

Case 1: We first consider binary search on sorted data. Recall that on a sorted array with N elements, a binary search is very efficient and takes only O(log N) time and also that due to lack of random access, a linear linked list fails to support binary search even if the entries are sorted.

Since the tree list does provide random access with each access taking O(log N) time as described above, we infer that if the data items stored in a tree list are sorted, a binary search can progress with $O(\log^2 N)$ complexity.

Case 2: If the data items stored are not sorted, searching for a particular key value can take O(N) time in both array and linked list. For the tree list, if we access elements in the ascending order of insertion (first, second, third and so on…) this can result in a higher O(N log N) complexity; indeed, reaching the ith element takes log(i) time. However, we can manage to do this in O(N) by running a depth first search(DFS) on the tree. Recall that depth first search has O(V+E) complexity in a graph with V vertices and E edges ([3],[4],[6]). Crucially, for any tree, number of edges E is only V-1. The DFS algorithm has only O(N) complexity for the tree list so there is no issue of performance degradation.

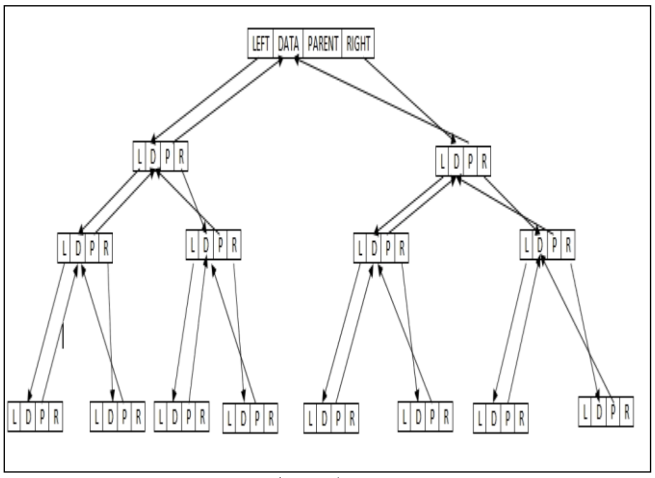
Note: During DFS, the elements in a tree list are not visited in the order of insertion but we do know that the children of the ith element are 2*i and (2*i)+1. So if we are to transfer the contents onto a display array, we only need to allocate an array equal to the size of the total data in the tree list and populate this array using the fact that after ith element, we add the (2*i)th and ((2*i)+1)th element to the array and not the (i+1)th. So, if we are to display or write out the entire contents stored, the tree list does not perform any worse than an array of linked list – all are O(N).

IV. THE TREE LIST AND SORTING

We recall that the linked list, due to lack of random access, does not allow popular sorting algorithms such as quicksort and heapsort which depend crucially on random access to be performed naturally. These algorithms work best on arrays. Quick sort has the average complexity of O(N logN) and heapsort, the worst case complexity of O(N logN) when applied to arrays [3].

A. Quicksort

Since the tree list allows random access with time complexity O(logN) per access, we infer that each comparison of quicksort goes through with time complexity O(logN) – as opposed to constant time if we are working with an array – and so quick sort, that takes O(N logN) comparisons between elements on an average, when done on a tree list takes $O(N \log^2 N)$ time. This is admittedly inferior to the performance of an array but a gain over a linked list.



(Fig 3)

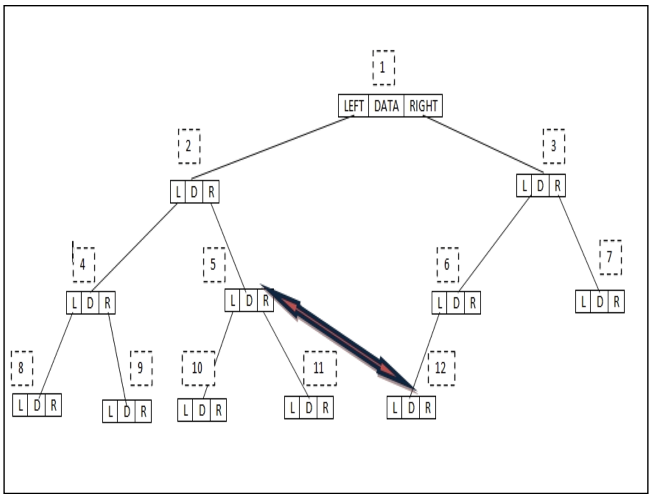
B. Heapsort

The heap is structured as a complete binary tree [5]. So is the tree list as we defined it above. So the heapsort algorithm can be seen to work naturally on the tree list with same complexity (O (N logN)) as on an array.

Note: The principal heap operations are percolating a data value up and down the complete binary tree that constitutes the heap. The way we have defined the tree list (with each node storing pointers to children and data) does not efficiently allow upward percolation. This can be enabled by each node also storing a pointer to its parent. Such an enhanced tree list is in Fig 3. Without the parent pointer stored, the complexity of heap sort will rise to O(N $\log^2$ N) for a tree list. Also note that in the classical heap sort [3], finding the parent to a particular data item is straightforward since the heap is held by an array and no pointers are used. The tree list, to achieve flexible size and dynamic allocation, needs pointers and hence, the explicit need to store parent pointers in each node if heap sort has to go through efficiently

C. Merge sort

Merge sort works naturally on anarray and can be implemented efficiently on a linked list as well (O(N logN) complexity) but is somewhat difficult to execute on a tree list due to elements with successive indices not being adjacent on the structure.



(Fig 4)

V. INSERTION AND DELETION OF VALUES

The linked list is a highly flexible structure that allows a particular node to be inserted anywhere in the data structure or deleted with just a constant number of pointer redirections. But both operations on an array are more difficult – we will need all succeeding elements to be pushed forward of slid back one step and that takes O(N) time. In a tree list, if a particular i^{th} element is to be deleted or a value is to be inserted after the i^{th} element, if we further require that the insertion order of all remaining elements needs to be maintained, we will need rearrangement of the entire data structure following it and that takes O(N logN) time. However, if the order of the rest of the elements can be for deletion, we could exchange the data items of the i^{th} node and the last node in the tree list and delete the last node – the latter deletion is straightforward and needs only constant number of pointer changes.

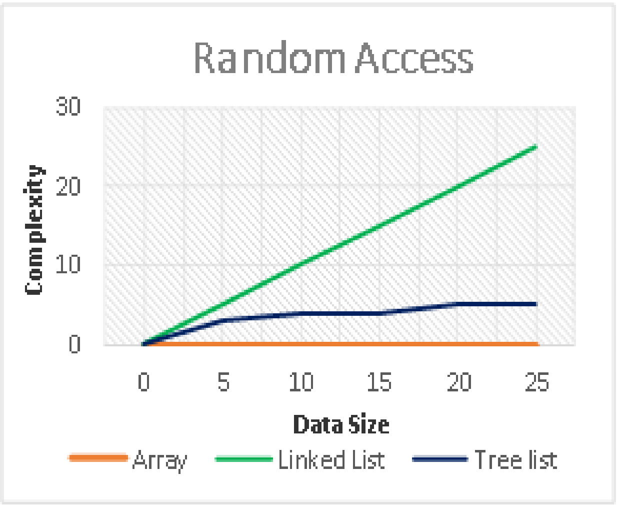
Fig 4 shows how the 5^{th} element in the tree list is to be deleted. This will 'promote' the 12^{th} element to be the new 5^{th} element (Note that this approach works for an array too).

Remark: If deletions need to be performed and the insertion order preserved, one can avoid doing an actual deletion by marking particular nodes as 'deleted' by means of an additional binary flag variable stored in each node. This can save processing time if the number of nodes deleted is not too high. A 'grand-slam' rebuild of a fresh tree list can be done once a certain fraction of nodes have got deleted.

VI. ANALYSIS OF RESUTLTS

In this section, we summarize the results of experiments done with Tree Lists. We focus on comparisons with arrays and linked lists when the basic operations of random access and searching are carried out.

A. Comparison of Data Structures - Random access



(Chart 1)

Data set size	Time (Array)	Time(Linked List)	Time(Tree list)
0	0	0	0
5	1	5	3
10	1	10	4
15	1	15	4
20	1	20	5
25	1	25	5

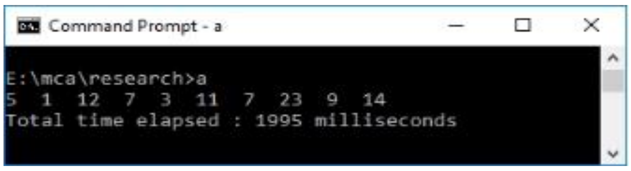
(Table 1)

The table 1 is formed based the result analysis. The chart1 shows, when in an array the complexity is 1 (instant access). But for a linked list, the complexity is linear. The Tree list performs much better than the linked list and the gains become more significant as data size increases.

B. Comparison of Data structures – Searching or Display

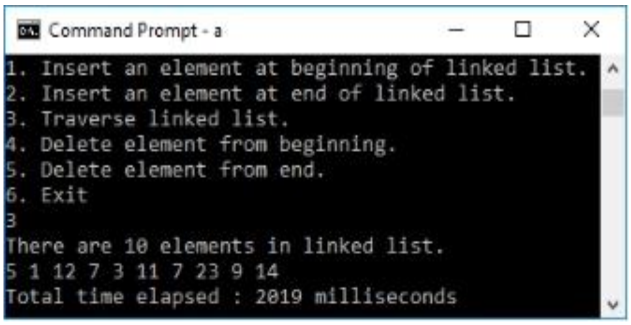
The results shown below demonstrate that the three data structures we considered, the array, linked list all show O(N) complexity in searching if the numbers stored are not sorted. and tree list is implemented based on the fig 1 data and while displaying, all the three shows same clock time. so we prove that three of them have same complexity (we already know that, while displaying the data the array and linked list has the O(N) complexity).to achieve O(N) Complexity in the tree list based on the recursive function(DFS).

Searching an Array:



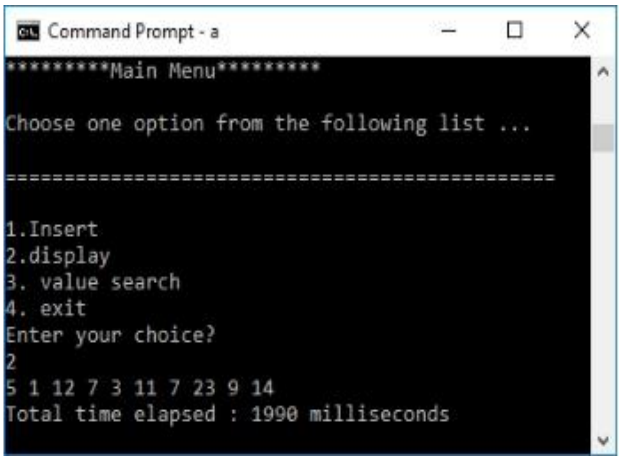
```
E:\mca\research>a
5  1  12  7  3  11  7  23  9  14
Total time elapsed : 1995 milliseconds
```

Searching an Linked list:



```
1. Insert an element at beginning of linked list.
2. Insert an element at end of linked list.
3. Traverse linked list.
4. Delete element from beginning.
5. Delete element from end.
6. Exit
3
There are 10 elements in linked list.
5 1 12 7 3 11 7 23 9 14
Total time elapsed : 2019 milliseconds
```

Searching a Tree List:



```
*********Main Menu*********

Choose one option from the following list ...

==============================================

1.Insert
2.display
3. value search
4. exit
Enter your choice?
2
5 1 12 7 3 11 7 23 9 14
Total time elapsed : 1990 milliseconds
```

VI. CONCLUSION

POSSIBLE APPLICATIONS OF THE TREE LIST

There are many situations where a variable large number of insertions need to be made and the number of deletions are much fewer or even close to zero. For example, the number of students in a batch can vary widely and few students if any drop out during the duration of the course. Likewise, the list of employee in an organization will keep on growing and even if an employee leaves, his or her record is only frozen and not deleted and his/her employee code is never assigned to a new recruit. An even larger example is the collection of aadhar card holders. In storing such data with frequent additions but very sparing deletions, the tree list can be of use. It is flexible enough to allow need-based growth and allows random access albeit at a price (which still remains low in comparison to a flexible data structure such as a linked list).

As we showed above, the tree list also allows many of the classical algorithms for sorting and searching to be performed without too much reworking. The use of 2 or 3 pointers per node does constitute an overhead but there certainly are situations where it is not too high a price to pay. Overall, we conclude that the tree list data structure incorporates several of the attractive features of both the classical linked list and array.

REFERENCES

1. Linked list complexity - https://www.geeksforgeeks.org/nth-node-from-the-end-of-a-linked-list/
2. Complete binary tree - https://en.wikipedia.org/wiki/Binary_tree#complete
3. David mount- https://www.cs.umd.edu/class/fall2013/cmsc451/Lects/cmsc451-fall13-lects.pdf
4. Depth first search- https://en.wikipedia.org/wiki/Depth-first_search
5. Heap sort-https://en.wikipedia.org/wiki/Heap_(data_structure)
6. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS.

AUTHORS PROFILE

Sibin James, PG Student, Master of omputer Application, Department of Computer Science & IT, Amrita School of Arts & Sciences, Kochi, Amrita Vishwa Vidyapeetham, India.

Pranav Prakash, PG Student, Master of Computer Application, Department of Computer Science & IT, Amrita School of Arts & Sciences, Kochi, Amrita Vishwa Vidyapeetham, India.

R Nandakumar, Assistant Professor, M.Tech.(CS), M.Sc. (Physics), Department of Computer Science & IT, Amrita School of Arts & Sciences, Kochi, Amrita Vishwa Vidyapeetham, India.