ISSN:0975-9646

Arijit Bhattacharya et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 6 (6) , 2015, 5558-5560

Yet Another nlogn Sorting

Arijit Bhattacharya [#1], Satrajit Ghosh [*2]

[#] *Assistant Professor, Department of Computer Science and Application, Gour Mahavidyalaya, Malda, West Bengal, India*

[*] *Associate Professor, Department of Computer Science, Acharya Prafulla Chandra College, New Barrackpore, West Bengal, India*

Abstract— **Sorting of Data is classical problem in Data Structure. Best known sorting algorithm can sort a sequence of n Records at the expense of log(n !) key comparisons. This work suggests a new data structure that achieves this theoretical minimum. Interestingly, pre-order traversal performed in such a structure can be readily mapped onto an in-order sequence.**

Keywords - ***Sorting, In-order, Pre-order, Height Balanced tree, Divide and Conquer, NLC.***

I. Introduction

Classification of data or sorting, depending upon their salient features, is a classical problem of human civilization and its socio economic activities. This is an important traditional problem in business data processing that creates the interest of the theoreticians as well. Sorting of an arbitrary sequence of data and retrieval of the desired ones from the sorted sequence is a classical problem in computational science. Interestingly, sorting of a sequence of *n* records requires $[\log_2(n!)]$ key comparisons or more as shown in Merge insertion method. Searching a particular record from such a sorted sequence is possible at the expense of $[\log_2 n]$ key comparisons. The problem of sorting can be solved at the expense of constant storage if one uses ternary heap-sort technique with complexity $1.47n \log_2 n$. A balanced binary tree structure is an elegant way to ensure optimal result.

II. Preliminaries

The simplest algorithms usually take $O(n^2)$ time to sort n objects and are only useful for sorting small numbers. One of the most popular sorting algorithms is quicksort, which takes $O(n\log_2 n)$ time on average. Quicksort, a divide and conquer based algorithm which works well for most common applications, although, in the worst case, it can take $O(n^2)$ time. There are other methods, such as heapsort and mergesort, that take $O(n\log_2 n)$ time in the worst case, although their average case behaviour may not be quite as good as that of quicksort.

III. Proposed Work

The proposed work, uses a divide and conquer technique, which will take a height balanced binary tree as an input and will provide the sorted sequence as an output. The sequence will be stored in an array.

The following algorithm follows the node structure.

<table>
<tr><td>Balance</td><td>Left</td><td>Right</td></tr>
<tr><td colspan="2">Key</td><td>NLC</td></tr>
</table>

Balance(p) is the difference of the height of the right subtree of p and the height of the left subtree of p. This is a two bit field that contains the values in the range -1, 0 and 1, here p is the pointer of the root of the height balanced tree. Left(p) is the pointer of the left subtree of the node p. Right(p) is the pointer of the right subtree of the node p. Key(p) is the data value stored in the node p. NLC(p) is the number of left child in the node p,. Low is a integer variable, which is initialized to 0. NLC denotes the number of left child of p. This is a $\lceil \log_2 n \rceil$ bit field, where n is the number of nodes in the tree.

The main idea of the proposed work is to place the data, stored in the AVL tree, into the relative portion of the array in one by one order, such that the full array is sorted. The Height balanced binary search tree ensures that all the key values of the left subtree of p is less than the key value of p, and all the key values of the right subtree of p is greater than the key value of p. So, the relative order of a node depends on the number of left child it has. If number of left child of a node is known, then we can place the node easily into the proper position of the array. Now divide the array into two parts and use the same technique for the left and right subtree recursively. From that position, the array is broken into two parts, one left part which is for all the nodes in the left subtree of p and the right part which is for all the nodes in the right subtree of p.

The Recursive Algorithm

Input: A height balanced binary search tree with n number of nodes, an array with size n.

Output: The sorted sequence of the data represented in the given tree, stored in the array.

Data Structure: A modified height balanced binary tree.

The following is the recursive implementation of the above said process.

```
rbtreesort ( ptr, arr, low )
begin
        if ( ptr ) then
        begin
                p← low + NLC[ptr]
                arr[p] ← key[ptr]
                rbtreesort ( left[ptr], arr, low )
                rbtreesort ( right[ptr], arr, p+1 )
        end if
end rbtreesort
```

Here, ptr is the pointer of the root of the height balanced binary search tree, low is the lower index of the array arr.

THE NON-RECURSIVE ALGORITHM

Input: A height balanced binary search tree with n number of nodes, an array with size n.
Output: The sorted sequence of the data represented in the given tree, stored in the array.
Data Structure: A modified height balanced binary tree.

```
nrbtreesort ( ptr, arr, low)
begin
    repeat
        while ptr do
            t ← low + NLC[ptr]
            arr[t] ← ptr
            if ( right[ptr] ) then
                Push ( right[ptr], t+1)
            endif
            ptr ← left[ptr]
        endwhile
        ptr ← Pop()
    until ptr= NULL
end nrbtreesort
```

IV. ILLUSTRATION

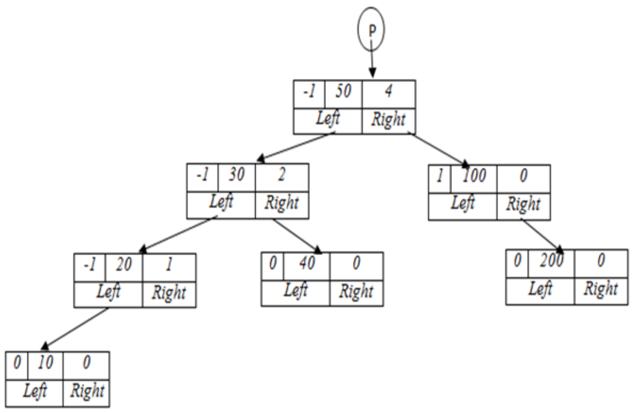

Fig.1 Example of btreesort

In this example, first the root node is placed properly. Number of left child in the root node is four, so the relative order of the node in the array is low index plus number of left child, which is zero plus four, is also four. From now on we apply the same technique for the left and right subtree recursively. The following table shows the steps how the array is filled.

TABLE I

	0	1	2	3	4	5	6
Step 1					50		
Step 2			30		50		
Step 3		20	30		50		
Step 4	10	20	30		50		
Step 5	10	20	30	40	50		
Step 6	10	20	30	40	50	100	
Step 7	10	20	30	40	50	100	200

The technique is applicable for any binary search tree with extra information of number of left child. The complexity of the procedure will rise as the height of a binary search tree. For a height balanced binary search tree with n internal nodes, it is easy to show that the height always lies between $\log_2(n+1)$ and $1.4404\log_2(n+2) - 0.3277$. So the time complexity of the proposed algorithm will not be more than $O(n\log_2 n)$.

Expression for minimum number of nodes in an AVL tree of height h:

$N_o = 1 \quad N_1 = 2$
$N_h = 1 + N_{h-1} + N_{h-2} = f_{h+3} - 1 = [\varphi^{h+3}/\surd 5] - 1$
More precisely, minimum number of nodes in the heavier sub-tree is given by $N_{h-1} = [\varphi^{h+2}/\surd 5] - 1$
Therefore, $(N_{h-1} + 1)/(N_h + 1) = 1/\varphi$
Therefore, the number of bits needed to store
$N_{h-1} \leq \lceil log_2(N_h+1) - log_2(\varphi) \rceil = l$, say
Interestingly, the number of bits required is $\geq \lceil log_2(N_{h-1}+1) \rceil - 1$
if $l - 1 + log_2(\varphi) < log_2(N_h) \leq l + log_2(\varphi)$
or $2^{l-1+log_2(\varphi)} < N_h + 1 \leq 2^{l+log_2(\varphi)}$
or $\varphi.2^{l-1} \leq N_h < \varphi.2^{l} + 1$

Expression for a maximally skewed AVL tree of height h:

The left sub-tree (the heavier one) is a complete binary tree of height h-1 and the right sub-tree is an AVL tree of height h-2 that is constructed with minimum number of nodes.
$N_o = 1 \quad N_1 = 2$
$N_{h-2} = 1 + N_{h-3} + N_{h-4} = f_{h+1} - 1 = [\varphi^{h+1}/\surd 5] - 1$
$N = N_{HEAVIER} + N_{LIGHTER} + 1$
$= 2^h - 1 + [\varphi^{h+1}/\surd 5] - 1 + 1$
$= 2^h - 1 + [\varphi^{h+1}/\surd 5]$

TABLE III

Tree of N nodes having height h	$(N_{HEAVIER}+1) / (N_{HEAVIER} + N_{LIGHTER} + 1)$
Complete binary tree	$2^h / 2^{h+1} = ½$
AVL tree with minimum nodes	$\varphi^{h+2} / \varphi^{h+3} = 1/\varphi$
Maximally skewed tree	$2^h / (2^h - 1 + [\varphi^{h+1}/\surd 5])$ $= 1/(1 - 2^{-h} + (\varphi/2)^h \varphi/\surd 5)$ $= 1/(1 - 2^{-h} + (\varphi/\surd 5) * \cos^h(\pi/5))$

V. CONCLUSION

The proposed sorting methodology ensures optimal result for a binary search tree with static structure. Further refinement of this data structure is needed to extend this optimal result for a tree with dynamic structure. Interestingly, this structure has a stricking feature, One can traverse such a *tree in pre-order fashion* to extract its in-order sequence.

REFERENCES

[1] Adel'son-Velskii, G. M., and Y. M. Landis [1962]. An algorithm for the organization of information, Dokl. *Akad. Nauk SSSR 146*, pp. 263-266, *English translation in Soviet Math. Dokl. 3*, pp. 1259-1262.

[2] Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. The Design and Analysis of Computer Algorithms, *Addison-Wesley, Reading, Mass*.

[3] Fischer, M. J. [1972]. Efficiency of equivalence algorithms, *Complexity of Computer Computations (R. E. Miller and J. W. Thatcher, Eds*.) pp. 153-168.

[4] Hoare, C. A. R. [1962]. "Quicksort," *Computer J. 5:1*, pp. 10-15.

[5] Knuth, D. E. [1973]. The Art of Computer Programming Vol. III: Sorting and Searching, *Addison-Wesley, Reading, Mass*.

[6] Nievergelt, J. [1974]. Binary search trees and file organization, *Computer Surveys 6:3, pp. 195-207*.

[7] Singleton, R. C. [1969]. "Algorithm 347: an algorithm for sorting with minimal storage," *Comm. ACM 12:3*, pp. 185-187.

[8] Wirth, N. [1976]. Algorithms + Data Structures = Programs, Prentice-Hall, *Englewood Cliffs, N. J.*

[9] J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," SIAM Journal on Computing 2 (1973), 33-43.

[10] M. O. Albertson and J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, New York, 1988.

[11] W. H. Burge, *Recursive Programming Techniques*, Addison-Wesley, Reading, Mass., 1975.

[12] D. D. Sleator and R. E. Tarjan, "Self-adjusting Binary Search Trees," *Journal of ACM* 32 (1985), 652-686.

[13] K. Melhorn, "A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions," SIAM Journal of Computing 11 (1982), 748-760.

[14] T. H. Hibbard, "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting," Journal of the ACM 9 (1962), 13-28.

[15] Heger, Dominique A. (2004), "A Disquisition on The Performance Behavior of Binary Search Tree Data Structures" , European Journal for the Informatics Professional **5** (5): 67–75.

[16] S. Baase. “Computer Algorithms, Introduction to Design and Analysis”, 3rd ed., Addison-Wesley, 2000.