International Journal of Computer Applications (0975 – 8887)
Volume 24– No.3, June 2011

Implementation and Performance Analysis of Exponential Tree Sorting

Ajit Singh
Department of computer Science and Engineering,
Thapar University, Patiala, Punjab, India

Dr. Deepak Garg
Assistance Professor,
Department of computer Science and Engineering,
Thapar University, Patiala, Punjab, India

ABSTRACT
The traditional algorithm for sorting gives a bound of $O(n\ log\ n)$ expected time without randomization and $O(n)$ with randomization. Recent researches have optimized lower bound for deterministic algorithms for integer sorting [1-3]. Andersson has given the idea of Exponential tree which can be used for sorting [4]. Andersson, Hagerup, Nilson and Raman have given an algorithm which sorts n integers in $O(n\ log\ log\ n)$ expected time but uses $O(m)$ space [4, 5]. Andersson has given improved algorithm which sort n integers in $O(n\ log\ log\ n)$ expected time and linear space but uses randomization [2, 4]. Yijie Han has improved further to sort n integers in $O(n\ log\ log\ n)$ expected time and linear space but passes integers in a batch i.e. all integers at a time [6]. These algorithms are very complex to implement. In this paper we discussed a way to implement the exponential tree sorting and later compare results with traditional sorting technique.

General Terms
Algorithms, Data Structure.

Keywords
Deterministic Algorithms; Sorting; Integer Sorting; Complexity; Space Requirement, Exponential Tree.

1. INTRODUCTION
Sorting is a classical problem which has been studied by many researchers. In modern computer world most of the problem is being solved by sorting. The traditional algorithms for sorting give a clear picture for complexity and these are best implemented and used widely. Although the complexity for comparison sorting is now well understood, the picture for integer sorting is still not clear. The only known lower bound for integer sorting is the trivial $\Omega(n)$ bound. Many continuous research efforts have been made by many researchers on integer sorting [2-16]. Recent advances in the design of algorithms for integers sorting have resulted in fast algorithms [2-3, 6, 9-13, 15-16]. However, many of these ideas use randomization or super-linear space. These algorithms still to be implemented.

For sorting integers in [0, m – 1] range $O(m)$ space is used in many algorithms. When m is large, the space used is excessive which makes implementation a tough task. Thus integer sorting using linear space is more important and therefore extensively studied by researchers.

Fredman and Willard showed that n integers can be sorted in $O(n\ log\ n\ /\ log\ log\ n)$ time in linear space [12]. Raman showed that sorting can be done in $O\left(n\sqrt{\log n \log\log n}\right)$ time in linear space [15]. Later Andersson improved the time bound to $O\left(n\sqrt{\log n}\right)$ [8]. Then Thorup improved the time bound to $O(n(loglogn)^2)$ [16]. Later Yijei Han showed $O(n\ log\ log\ n\ log\ log\ log\ n)$ time for deterministic linear space integer sorting [9]. Yijei Han again showed improved result with $O(n\ log\ log\ n)$ time and linear space [6]. In these ideas expected time is achieved by using Andersson's exponential tree [8]. The height of such a tree is $O(log\ log\ n)$. Also the balancing of the tree will take only $O(log\ log\ n)$ time. Balancing does not take much time. The major time is taken by the insertion of integers as proved by the Andresson [8]. These algorithms require work on implementation to make them effective. The implementation must provide expected run time bound with linear space.

Andersson has shown that if we pass down integers in exponential tree one by one than the insertion takes $O\left(\sqrt{\log n}\right)$ for each integer i.e. total complexity will be $O\left(n\sqrt{\log n}\right)$ [8]. This is improvement over the result of Raman which takes $O\left(n\sqrt{\log n \log\log n}\right)$ expected time [15]. And this idea seems to be easy to implement as it is best way to handle one integer at a time.

Yijie Han has given an idea which reduces the complexity to $O(n\ log\ log\ n)$ expected time in linear space [6]. The technique used by him is coordinated pass down of integers on the Andersson's exponential search tree [8] and the linear time multi-dividing of the bits of integers. Instead of inserting integer one at a time into the exponential search tree, he passed down all integers one level of the exponential search tree at a time. Such coordinated passing down provides the chance of performing multi-dividing in linear time and therefore speeding up the algorithm. This idea may provide speed up, but in practical implementation it is very difficult to handle integers in batches.

This paper will present a way to implement the exponential tree sorting. We will be using modified concept of exponential tree to implement the sorting as it is very difficult to handle the pointers in actual Andersson's exponential tree [8]. We will pass down integers one at a time. The use of Binary search will enhance the performance.

2. PRELIMINARY

First we need to understand the implementation of exponential tree. As discussed above we will use modified concept of exponential tree. The exponential tree was first introduced by Andersson in his research for fast deterministic algorithms for integer sorting [8]. In such a tree the number of children increases exponentially. An exponential tree is almost identical to a binary search tree, with the exception that the dimension of the tree is not the same at all levels. In a normal binary search tree, each node has a dimension (d) of 1, and has 2^d children. In an exponential tree, the dimension equals the depth of the node, with the root node having a $d = 1$. So the second level can hold two nodes, the third can hold eight nodes, the fourth 64 nodes, and so on. This shows that number of children at each level increased by a multiplicative factor of 2 i.e. exponential increases in number of children at each level.

The modified concept says that a tree with properties of binary search tree will be the exponential tree if it has following properties:

1. Each node at level k will hold k number of keys (or integers in our case) i.e. at depth k the number of key in any node will be k keeping root at level 1.
2. Each node at level k will be having $k+1$ children i.e. at depth k the number of children will be $k+1$.
3. All the keys in any node must be sorted.
4. An integer in child i must be greater than key $i-1$ and less than or equal to key i.

Table-1: Total Number of Integers up to levels

Level	1	2	3	4	5	6	7	8	9
No of Integers	1	5	23	119	719	5039	40319	362879	3628799

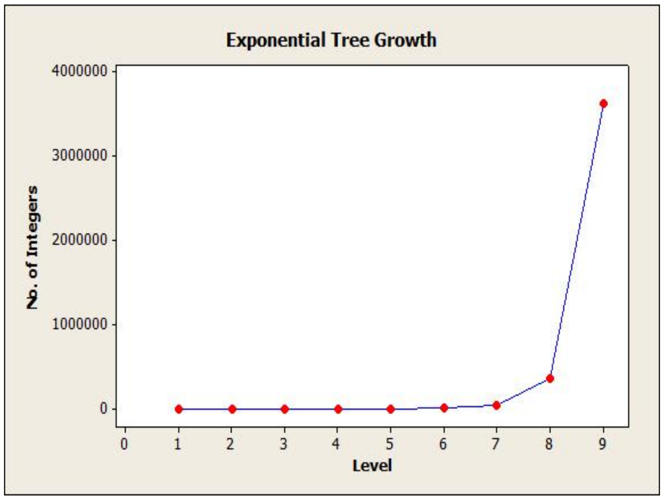

Figure-1: Plot of Growth of Exponential Tree

Thus it is clear that the new concept focus on the number of integers held by each level of exponential tree. The height of the tree will be $O(\log \log n)$.

The total number of integers hold by the tree up to level k will be given by the following formula:

$$N_k = N_{k-1} + k * k!$$

where N_k is total number of integers up to level k. $N_1 = 1$ as root is at level 1 and holds only 1 integer.

k is kth level and $k!$ denotes factorial of k.

The figure-1 which is the growth plot of the tree clearly shows that the tree has an exponential growth.

3. SORTING ALGORITHM

This section will provide the sorting algorithm with implementation. In order to do that, first we must create exponential node. Here is a skeleton for exponential node:

```
struct Node{
        int level;
        int count;
        Node **child;
        int data[];
}
```

Here *level* will holds the level number of the node.

count will holds the number of integer currently present in the node.

child is an array of pointers to *level+1* children of the node.

data is an array of integers to hold the integers present in that node.

Now before inserting any integer in the exponential tree, first we must know the exact node in which it is to inserted and position in that node. Hence we require searching method.

As we know that all integers presented in a node of exponential tree must be sorted thus we can exploit this property to enhance the performance. The modified binary search is as under.

```
BinarySearch(Node *ptr,int element)
Step1: If element > ptr->data[count-1] then return ptr->count.
Step2: Set start=0, end=ptr->count-1, mid= (start + end)/2.
Step3: Repeat step 4 & 5 while start < end.
Step4: If element > ptr->data[mid] then start=mid+1
            else end=mid
Step5: Set mid= (start + end)/2.
Step6: return mid.
```

Now we only require the insertion method in which we will call the binary search. The insertion method is as follows:

```
Insert(Node *root,int element)
Step1: Set *ptr=root, *parent=NULL, i=0.
Step2: Repeat step 3 to 6 while ptr <> NULL.
```

```
Step3: Set level=ptr->level, count=ptr->count.
Step4: Call i=BinarySearch(ptr, element).
Step5: If count<level then Repeat For j=count to i-1 by -1
                ptr->data[j]=ptr->data[j-1]
        Set ptr->data[i]=element
        Set ptr->count=count+1
        return
Step6: Set parent=ptr, ptr=ptr->child[i].
Step7: Create a new Exponential Node at ith child of parent and
        insert element in that.
Step8: Return.
```

The above discussed function insert() gives a complexity of $O(\log\log n \log\log\log n)$. When all the integers will be inserted in the tree, we will call in-order trace of the tree which will give us the sorted sequence. The in-order trace will be:

```
In-Order-Tace(Node *r)
Step1: Set count=r->count
Step2: Repeat For i=0 to count by 1
Step3: If r->child[i] <> NULL call In-Order-Trace(r->child[i]).
Step4: Print r->data[i].
Step5: If r->child[count] <> NULL then
        call In-Order-Trace(r->child[count]).
Step6: Return.
```

The overall combined complexity of above algorithms will be $O(n \log\log n \log\log\log n)$.

4. PERFORMANCE ANALYSIS

In this section we will compare the performance of the exponential tree sorting with binary tree sorting and quick sort. It is obvious to think here that we must compare the exponential sorting with Quick sort only; but as we know that quick sort mainly used for integers stored at consecutive memory location, here the exponential tree sorting is implemented on non-consecutive memory location. The comparison includes CPU running time and Memory requirement. The input integers are generated by using random function and stored in a file.

The implementation of algorithms is done in VC++ on Visual Studio 2008 using Object Oriented Approach. The platform used is Intel 64-bit with Core 2 Duo processor having a frequency of 2.0 GHz with Windows 7 64-bit Enterprises Edition running on it. The system had a RAM of 4GB. While measuring the performance i.e. collecting the details all other extra processes were terminated.

4.1 Runtime Comparison

The running time of algorithm is measured as CPU cycles and later converted to seconds. This is done using clock() function from clock_t in time.h and then divided by CLOCKS_PER_SECOND. Same process is followed for both other algorithms. The running time includes the reading inputs from input file, creating tree and writing output to output file. The best out of three runs is noted.

Table-2: CPU Running Time

N	Log N	Exp Tree	Binary Tree	Quick Sort
1024	10	0	0	0
2048	11	0.007	0.008	0.007
4096	12	0.016	0.016	0.016
8192	13	0.031	0.047	0.047
16384	14	0.046	0.078	0.141
32768	15	0.141	0.172	0.234
65536	16	0.202	0.343	0.359
131072	17	0.328	0.686	0.796
262144	18	0.687	1.341	1.326
524288	19	1.31	2.449	2.246
1048576	20	2.496	4.446	4.508
2097152	21	5.803	9.937	7.691
4194304	22	23.603	62.26	14.945
8388608	23	59.077	261.816	38.392
16777216	24	156.531	855.022	109.574

The figure-2 shows the CPU running time for the Exponential tree sorting, which clearly depicts that the slope of graph is linear.

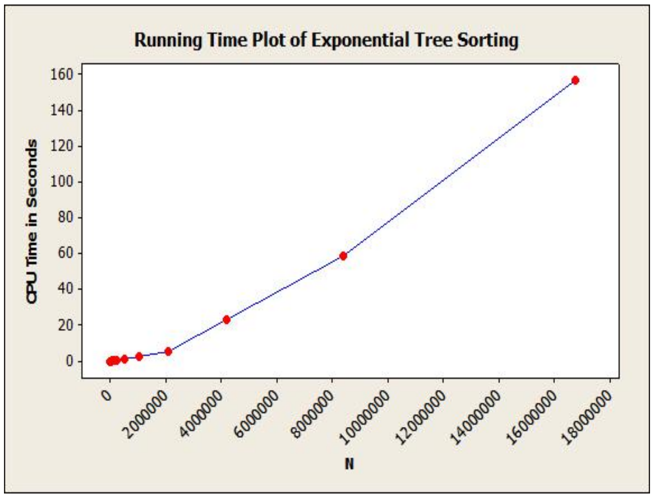

Figure-2: CPU Running Time Plot for Exponential Tree Sorting

The figure-3 shows the comparison between CPU running times for the exponential tree sorting and binary tree sorting which depicts that the exponential tree sorting takes less CPU time for same number of integers. As the number of integers increases the CPU time for exponential tree sorting increase with very smaller factor than binary tree sorting. The run time of Exponential tree sorting is also comparable to Quick Sort.

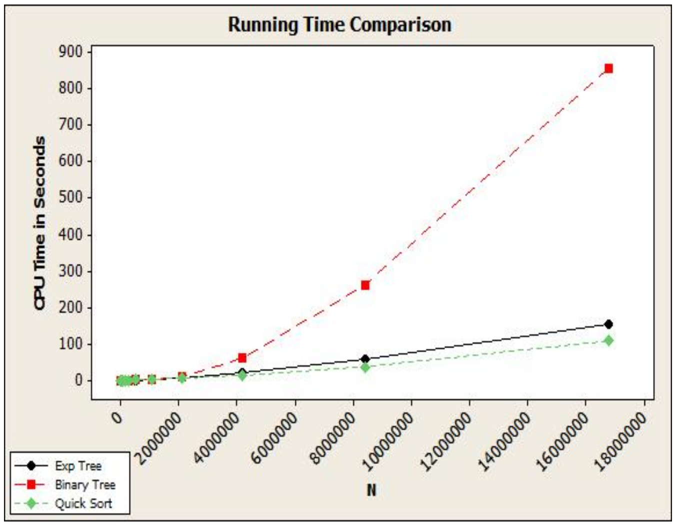

Figure-3: CPU Running Time Comparison

4.2 Memory Requirement

The memory used by the algorithm is measured by the Windows Task Manager. The algorithm is executed and the memory used is monitored and the maximum memory used by the algorithm during entire run is taken. Three runs are given and the maximum memory used is noted. The memory requirements are measured in KB (Kilo Bytes).

Table-3: Memory Requirement

N	Log N	Binary Tree	Exp Tree
1024	10	492	500
2048	11	568	572
4096	12	712	736
8192	13	996	1032
16384	14	1572	1668
32768	15	2724	2852
65536	16	5036	5008
131072	17	9656	8528
262144	18	18892	12860
524288	19	37368	20404
1048576	20	74312	34320
2097152	21	148208	59268
4194304	22	296012	102956
8388608	23	591596	176260
16777216	24	1182784	302908

The figure-4 shows the memory requirement plot for the exponential tree which depicts that the graph has a linear slope. The memory requirement increases directly proportionally to number of integers to be sorted.

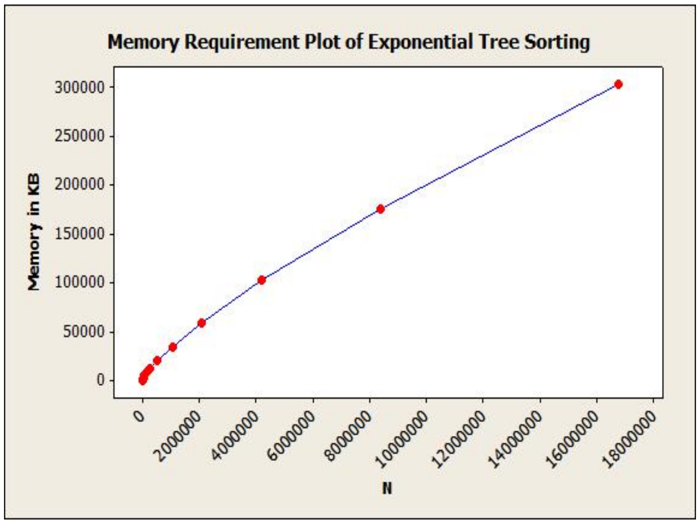

Figure-4: Memory Requirement Plot for Exponential Tree Sorting.

The figure-5 shows the comparison of memory requirements for exponential tree sorting and binary tree sorting which depicts that the exponential tree uses very less memory as compared to binary tree. The reason for less memory used by exponential tree is that binary tree uses two pointers with each node or each integer as there is always on integer present in binary tree node, whereas exponential tree uses *m+1* pointers with *m* integers as there is *m* integers are presented in exponential tree node.

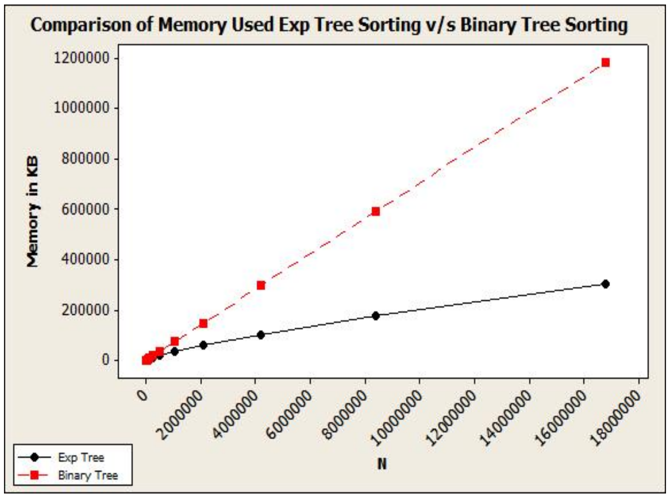

Figure-5: Memory Requirement Comparison

5. CONCLUSION

In this paper, we have given the implementation of the exponential tree with expected time bound of $O(n \log\log n \log\log\log n)$ in linear space. The way of implementation of exponential tree sorting discussed in this paper is one out of many different ways. The comparison of exponential tree sorting with binary tree sorting shows that the exponential tree sorting is far better than binary tree sorting. The performance is very competitive to quick sort. Thus, this must give better performance for CPU running time as well as memory requirements if implemented on set of integers stored in continuous memory location as array.

6. REFERENCES

[1] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest-pair problem, J. Algorithms 25, 1997.

[2] M. Thorup, Randomized sorting in *O(n log log n)* time and linear space using addition, shift, and bit-wise boolean operations, in: Proc. 8th ACM–SIAM Symposium on Discrete Algorithms (SODA'97), 1997.

[3] Y. Han, M. Thorup, Sorting integers in $O\left(n\sqrt{\log\log n}\right)$ expected time and linear space, IEEE Symposium on Foundations of Computer Science (FOCS'02), 2002.

[4] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in linear time?, Symposium on Theory of Computing, 1995.

[5] Y. Han, X. Shen, Conservative algorithms for parallel and sequential integer sorting, International Computing and Combinatorics Conference, 1995.

[6] Y. Han, Deterministic sorting in *O(n log log n)* time and linear space, 34th STOC, 2002.

[7] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10, 1977.

[8] A. Andersson, Fast deterministic sorting and searching in linear space, IEEE Symposium on Foundations of Computer Science, 1996.

[9] Y. Han, Improved fast integer sorting in linear space, Inform. and Comput., 2001.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms*, Second Edition,The MIT Press and McGraw-Hill Book Company, 2001.

[11] S. Albers, T. Hagerup, Improved parallel integer sorting without concurrent writing, Inform. and Comput., 1997.

[12] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. System Sci., 1994.

[13] T. Hagerup, H. Shen, Improved nonconservative sequential and parallel integer sorting, Inform. Process. Lett., 1990.

[14] D. Kirkpatrick, S. Reisch, Upper bounds for sorting integers on random access machines, Theoret. Comput. Sci., 1984.

[15] R. Raman, Priority queues: small, monotone and trans-dichotomous, European Symp. on Algorithms, 1996.

[16] M. Thorup, Fast deterministic sorting and priority queues in linear space, ACM–SIAM Symp. on Discrete Algorithms (SODA'98), 1998.