

IMPROVED EXPONENTIAL TREE INTEGER SORTING ALGORITHM USING NODE GROWTH

*Thesis submitted in partial fulfillment of the requirements for the award
of degree of*

**Master of Engineering
in
Computer Science and Engineering**

Submitted By
Ajit Singh
(Roll No. 800932001)

Under the supervision of:
Dr. Deepak Garg
Assistant Professor



**COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004**

June 2011

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Improved Exponential Tree Integer Sorting Algorithm Using Node Growth*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Deepak Garg* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.



(Ajit Singh)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.


(Dr. Deepak Garg)

Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by


(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala


(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

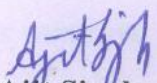
First of all, I am thankful to God for his blessings and for showing me the right direction. With His mercy; it has been made possible for me to reach so far.

It is a great privilege to express my gratitude and admiration toward my respected supervisor **Dr. Deepak Garg**. He has been an esteemed guide and a great support behind achieving the task. Without his able guidance, kind efforts and encouragement, the work wouldn't have been what it is. I am truly grateful to him for extending his total co-operation and understanding whenever I needed help and guidance from him.

I wish to express my heartiest thanks to **Dr. Maninder Singh**, Head, Department of Computer Science and engineering, Thapar University, Patiala for providing me the opportunity and all necessary facilities to accomplish this thesis successfully.

I would also like thank to my colleagues who were always there at the need of the hour and provided with the help for the completion of my thesis work.

I am grateful to my parents who soulfully provided me their constant support, and encouraging attitude to undertake the challenge of this proportion. They believed in me before I believed in myself. To them I owe my wonderful today and dream filled future.


Ajit Singh

(800932001)

The traditional algorithm for integer sorting gives a lower 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. This thesis will present an idea to achieve the complexity of deterministic algorithm for integer sorting in $O(n \log \log n \log \log \log n)$ expected time and linear space which is easy to implement and very simple enough. The idea will use Andersson's exponential tree to perform the sorting. The exponential tree can't be used as it is for this idea, so the modification in the exponential tree is necessary. Integers will be passed down to exponential tree one at a time but limit the comparison required at each level. The total number of comparison for any integer will be $O(\log \log n \log \log n)$ i.e. total time taken for all integers insertion will be $O(n \log \log n \log \log n)$.

The algorithm presented in this thesis can be compared with the result of Fredman and Willard that sorts n integers in $O(n \log n / \log \log n)$ time in linear space. It can also be compared with result of Raman that sorts n integers in $O(n \sqrt{\log n \log \log n})$ time in linear space and also with result of Andersson's time bound of $O(n \sqrt{\log n})$. The algorithm can also be compared with Yijei Han's result of $O(n \log \log n \log \log \log n)$ expected time for deterministic linear space integer sorting.

The implementation of the algorithm is also given and later performance is compared to traditional deterministic algorithms.

Table of Contents

Certificate	(i)
Acknowledgement	(ii)
Abstract	(iii)
Table of Contents	(iv)
List of Figures	(vi)
List of Tables	(vii)
Chapter-1: Introduction	1
1.1. Exponential Tree	2
1.1.1. Growth	3
1.2. Complexity of Exponential Tree	4
1.2.1. Insertion	4
1.2.2. Balancing	5
1.3. Sorting Using Exponential Tree	5
1.4. Deterministic Algorithms	6
1.5. Conservative and Non-Conservative Algorithms	6
1.6. Structure of Thesis	7
Chapter-2: Literature Review	8
2.1. Fusion Tree	10
2.2. Sorting in $O(n \sqrt{\log n})$ Expected Time	10
2.3. Sorting in $O(n(\log \log n)^2)$ Expected Time	12
2.4. Other Fast Deterministic Algorithms	16
2.4.1. Yijie Han's Concept of Sorting Integers in $(n \log \log n)$	16
Chapter-3: Problem Statement	18

Chapter-4: Improved Exponential Tree Integer Sorting Algorithm Using Node Growth	19
4.1. Modified Exponential Tree	19
4.1.1. Growth	22
4.1.2. Balancing	23
4.2. Insertion	24
4.3. Binary Search	26
4.4. Trace	27
Chapter-5: Testing and Results	29
5.1. Testing	29
5.1.1. Running Time	30
5.1.1.1. Analysis of Exponential Tree Sorting Running Time	31
5.1.1.2. Exponential Tree Sorting v/s Binary Tree Sorting	31
5.1.1.3. Exponential Tree Sorting v/s Quick Sort	32
5.1.1.4. Running Time Results	33
5.1.2. Space Requirement	34
5.1.2.1. Analysis of Exponential Tree Sorting Space Requirement	35
5.1.2.2. Exponential Tree Sorting v/s Binary Tree Sorting	36
5.2. Results	37
Chapter-6: Conclusion and Future Scope	39
6.1. Conclusion	39
6.2. Future Scope	40
References	41
List of Publications	44

List of Figures

Figure-1.1: Exponential Tree	2
Figure-1.2: Growth Plot of Exponential Tree	4
Figure-4.1: Root node of modified exponential tree	20
Figure-4.2: A node at i th level of modified exponential tree	21
Figure-4.3: A balanced and fully filled modified exponential tree	22
Figure-4.4: Growth plot of modified exponential tree	23
Figure-5.1: CPU running time plot for exponential tree sorting	31
Figure-5.2: Exponential tree sorting v/s binary tree sorting running time	32
Figure-5.3: Exponential tree sorting v/s quick sort running time	33
Figure-5.4: CPU running time comparison	34
Figure-5.5: Memory requirement plot for exponential tree sorting	36
Figure-5.6: Memory requirement comparison	37
Figure-5.7: Output	38

List of Tables

Table-1.1: Number of nodes in exponential tree	3
Table-2.1: Comparison of traditional sorting algorithms	8
Table-4.1: Total number of integers in modified exponential tree	23
Table-5.1: CPU running time	30
Table-5.2: Memory requirement	34

Chapter-1

Introduction

In modern computer world most of the problems are being solved by use of sorting. It is a classical problem which has been studied by many researchers. The traditional algorithms for sorting give a clear picture for complexity. 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. Recent advances in the design of algorithms for integers sorting have resulted in fast algorithms. However, many of these algorithms use randomization or super-linear space.

For sorting integers in $[0, m - 1]$ range $O(m^\epsilon)$ space is used in many algorithms. When m is large, the space used is excessive. 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 [1]. Raman showed that sorting can be done in $O(n \sqrt{\log n \log \log n})$ time in linear space [2]. Later Andersson improved the time bound to $O(n \sqrt{\log n})$ [3]. Then Thorup improved the time bound to $O(n(\log \log n)^2)$ [4]. Yijei Han also proved the same result [5]. Later Yijei Han showed $O(n \log \log n \log \log \log n)$ time for deterministic linear space integer sorting [6]. Yijei Han again showed improved result with $O(n \log \log n)$ time and linear space [7].

In most of these algorithms expected time is achieved by using Andersson's exponential tree [3]. The height of such a tree is $O(\log \log n)$. The exponential tree plays an important role in all these concepts.

1.1. Exponential Tree

The exponential tree was first introduced by Andersson in his research for fast deterministic algorithms for integer sorting [3]. The idea of exponential tree varies from one researcher to other [3, 7, 8]. But the basic idea of embedding nodes in tree such a way that the increase of nodes with respect to height or depth behaves exponentially. In such a tree the number of children increases exponentially as the number of level increase i.e. depth of tree increases.

One definition of exponential tree is: it 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 $2d$ children. In an exponential tree, the dimension equals the depth of the node, with the root node having $d = 1$. So the second level can hold two nodes, the third can hold eight nodes i.e. four children of each node at second level, the fourth can hold 64 nodes i.e. eight children of each node at third level, and so on [7].

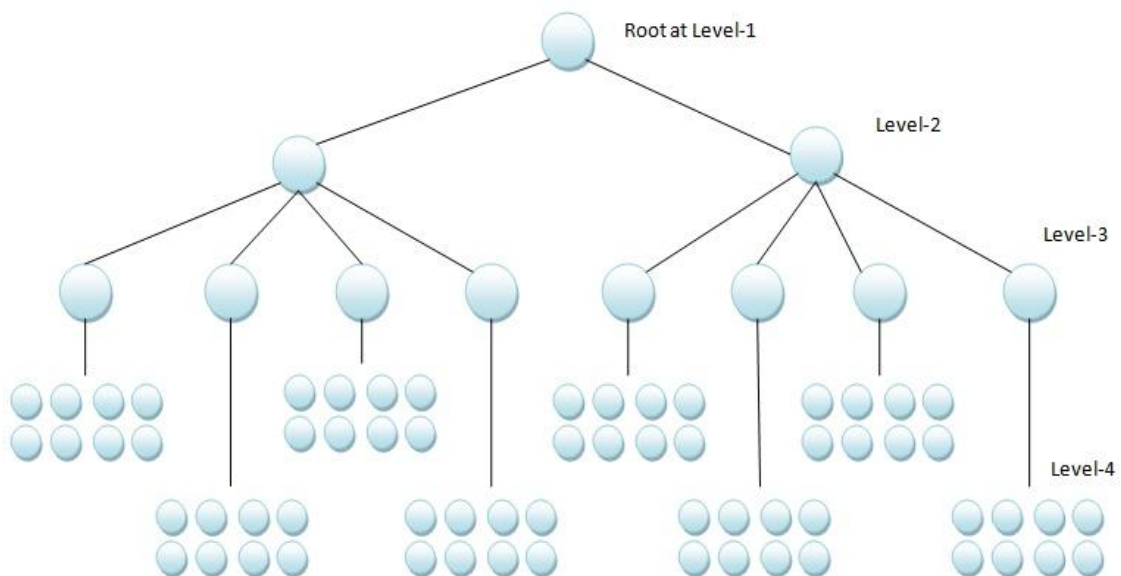


Figure-1.1: Exponential tree

The height or depth of such tree is $O(\log \log n)$ as proved by Andersson [3]. This exponential tree is very difficult to handle in implementation as the pointer to children at nodes increases exponentially. With each increasing level, number of children of a node becomes doubles to its parent. The complexity associated with this tree makes it

very difficult to use in practical development. This is not the only one definition available for exponential tree. There have been many other structures for exponential tree proposed by researchers. But it resembles nearly to traditional tree structure, thus easy to understand.

1.1.1. Growth

The growth of exponential tree is very important to understand in order to understand the complexity associated with this kind of data structure. As discussed above, with increasing level or depth the number of children doubled for every child of parent each time. This gives the exponential growth to the tree. The table-1.1 shows relation that how the total number of node present in the tree increases exponentially with the increment in the level or depth.

Table-1.1: Number of nodes in exponential tree

Level	Number of Nodes at Level	Total number of Nodes up to Level
1	1	1
2	2	3
3	8	11
4	64	75
5	1024	1099
6	32768	33867
7	2097152	2131019

The figure-1.2 clearly shows that the tree has exponential growth. Thus it becomes very tidy to handle the growth of the tree in practical as the number of level increases. Because of this exponential growth it is named as exponential tree. It is to be noted here that each node has single key i.e. integer. This implies that the comparison to the node values of children nodes is very difficult task as there is enormous number of pointers associated with each node.

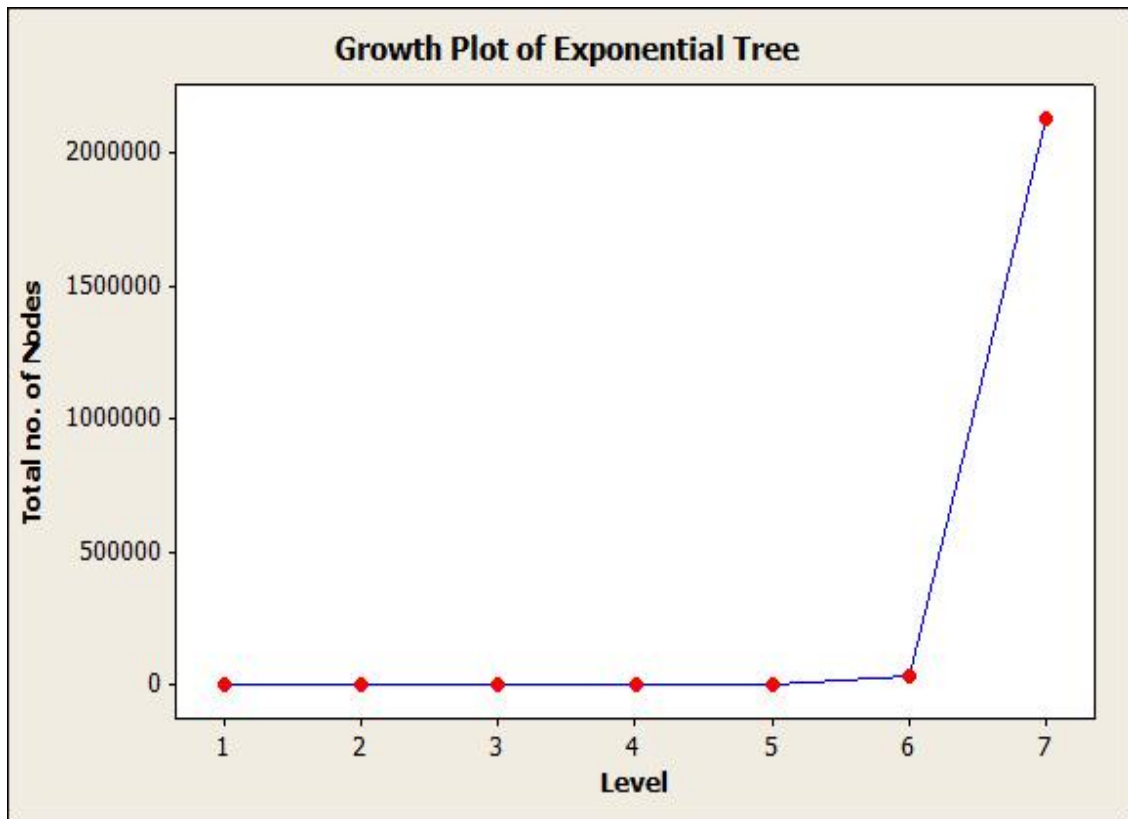


Figure-1.2: Growth plot of exponential tree.

1.2. Complexity of Exponential Tree

There are mainly two type of complexity associated with any tree. First complexity is of insertion in tree and the second complexity is of balancing of the tree. All other operation may include tracing the tree, deletion from tree and many more. But the major tasks are only of insertion and balancing, as all other operations always take less or negligible time as compared to these two techniques.

1.2.1. Insertion

Andersson has proved that the major time is taken by the insertion in case of exponential tree [3]. Andersson has shown that if we pass down integers in exponential tree one by one than the insertion takes $O(\sqrt{\log n})$ for each integer i.e. total complexity will be $O(n\sqrt{\log n})$ [3]. The expected time for insertion can be reduced further. One of the methods is to insert integers in batches [7]. The insertion of integers in batches improves the expected time required by insertion. Other ideas include multi-dividing of input integers. The multi-dividing technique improves the

insertion complexity very much. There are many other ideas proposed by researchers to improve the expected time for insertion.

1.2.2. Balancing

Andersson has shown that the balancing of the tree will take only $O(\log \log n)$ time [3]. Balancing does not take more time as compared to insertion. There is not as such method to balance the exponential tree. The balancing techniques for exponential tree vary from one idea to another. The self balancing tree technique can be used for the balancing of exponential tree. In this technique, the track of height or depth of each sub-tree must be kept so that the balancing of exponential tree can be done appropriately and in less time.

1.3. Sorting Using Exponential Tree

There are many ideas proposed for integer sorting using the exponential tree. These ideas do sorting by inserting integers in the tree and later tracing the tree in-order to get the desired sequence i.e. sorted sequence. These ideas either pass integers one by one or pass integers in batches. In this section, two important ideas of sorting using exponential tree is being discussed.

Andersson has shown that if we pass down integers in exponential tree one by one than the insertion takes $O(\sqrt{\log n})$ for each integer i.e. total complexity for n integers will be $O(n\sqrt{\log n})$ [3].

Yijie Han has given an idea which reduces the complexity to $O(n \log \log n)$ expected time in linear space [7]. The technique used by him is coordinated pass down of integers on the Andersson's exponential search tree 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.

1.4. Deterministic Algorithms

A deterministic algorithm is an algorithm which, in informal terms, behaves predictably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states. Deterministic algorithms are by far the most studied and familiar kind of algorithms, as well as one of the most practical, since they can be run on real machines efficiently.

Formally, a deterministic algorithm computes a mathematical function; a function has a unique value for any given input, and the algorithm is a process that produces this particular value as output.

A deterministic algorithm never goes out of order. It always follows the predefined set of steps in an ordered sequence. Thus, the deterministic algorithm is very important characteristic of a sorting algorithm. Recently developed fast algorithm is deterministic in nature.

The deterministic algorithms are very common and widely studied. It is always preferable to know that how an algorithm will behave in advance. The deterministic algorithm always has this nature.

1.5. Conservative and Non-Conservative Algorithms

An integer sorting algorithm which can sort n integers from set $\{0, 1 \dots m - 1\}$ is called a conservative algorithm if the word length (the number of bits in a word) used in the algorithm is $O(\log(m + n))$. There are many benefits of an algorithm to be conservative in nature. One benefit of this nature is that the integer overflow error can be avoided as the number of bits in a word is always under control and less than or equal to $O(\log(m + n))$. This will also reduce the extra operations required to handle the extra bit if integer is out of range.

An integer sorting algorithm which can sort n integers from set $\{0, 1 \dots m - 1\}$ is called a non-conservative algorithm if the word length used is larger than $O(\log(m + n))$. The non-conservative nature of an algorithm is always an extra overhead. The larger number of bits in a word may yield error in result. The

extra operations are required to handle the extra bit if integer is out of range. Also the actual word length of platform is always a constant, thus if word length increase beyond a limit it becomes impossible to handle integer in a single word.

Recent algorithms developed with the use of exponential tree are of both natures i.e. conservative as well as non-conservative. The algorithm with conservative nature is always preferred due to its many good yielding characteristics. The algorithm discussed in this thesis also has conservative advantage.

1.6. Structure of the Thesis

The rest of thesis is organized in the following order:

Chapter-2: This chapter will provide the overview of all recent work done in area of fast deterministic integer sorting.

Chapter-3: This chapter gives the problem statement and methodology used to solve the problem.

Chapter-4: This chapter provides the solution to the problem discussed in chapter-3. This chapter also gives the modified data structure and the deterministic algorithm for integer sorting.

Chapter-5: This chapter explains the implementation, testing and result of algorithm given in chapter-4.

Chapter-6: This chapter gives the conclusion of the thesis with the future scope of topic.

Sorting and searching in deterministic linear space is certainly a fundamental topic and many researchers have tried to find new improved bounds since the fusion trees [1] were first presented and even before. In modern computer world most of the problems are being solved by sorting.

It is a classical problem which has been studied by many researchers. The traditional algorithms for sorting give a clear picture for complexity [9]. 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. The table-2.1 shows the comparative analysis of various traditional algorithms [9]:

Table-2.1: Comparison of traditional sorting algorithms

Name	Best	Average	Worst	Memory
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n)$	<i>Depends</i>	$O(n(\log n)^2)$	$O(n)$
Binary tree sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Depends
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Randomized Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Signature sort	$O(n)$	$O(n)$	$O(n)$	Linear

Many continuous research efforts have been made by many researchers on integer sorting [1-10]. Recent advances in the design of algorithms for integers sorting have resulted in fast algorithms. However, many of these algorithms use randomization or super-linear space. For sorting integers in $[0, m - 1]$ range $O(m^\epsilon)$ space is used in many algorithms. When m is large, the space used is excessive. Thus integer sorting using linear space is more important and therefore extensively studied by researchers.

Some of the recent results include: Fredman and Willard showed that n integers can be sorted in $O(n \log n / \log \log n)$ time in linear space using fusion tree [1]. Raman showed that sorting can be done in $O(n \sqrt{\log n \log \log n})$ time in linear space [2]. Later Andersson improved the time bound to $O(n \sqrt{\log n})$ which is discussed later in this chapter [3]. Then Thorup improved the time bound to $O(n(\log \log n)^2)$ which is also discussed later in this chapter [4, 10]. Later Yijei Han showed $O(n \log \log n \log \log \log n)$ time for deterministic linear space integer sorting [6]. Yijei Han again showed improved result with $O(n \log \log n)$ time and linear space [7]. Thorup also achieved a time bound of $O(n \log \log n)$ using randomization [10].

Many researchers have also worked on parallel integer sorting techniques [11-24]. Arrays, trees and hyper-cubes are used to achieve better expected time for parallel integer sorting [18]. Radix sort have been optimized for parallel integer sorting [21]. Priority queues have also been used to achieve better results [25, 26]. Many ideas are proposed which has non-conservative nature [15, 27]. Bucket sort is also enhanced to measure better performance [15]. Insertion sort can also be used to achieve a time bound of $O(n \log n)$ [28]. Many ideas proposed by researchers use randomization to achieve better time bound for integer sorting [2, 27, 29]. But randomization does not always give better result. The idea of sorting integers using word comparison is also enhanced [8, 10, 27]. The performance is one of the key issues with word comparison sorting technique. Researchers have been trying to improve its performance. The better selection algorithms and work done on problems related to random access machines played an important role in improvement of time bound for integer sorting [30, 31].

Andersson's signature sort was the pioneer for word comparison sorting technique [2]. It has been used by many researchers to achieve better time bound for integer sorting. But still the performance and implementation issues need to be addressed.

2.1. Fusion Tree [1]

Fusion tree [1] can be obtained by increasing the degree of the binary trees to $\log n$ so that the height of the tree is reduced to $O(\log n / \log \log n)$. A fusion tree is essentially a B-tree with branching factor $B = (\log n)^{1/5}$. If h is the height of the B-tree, then $n = B^h$, so

$$n = ((\log n)^{1/5})^h$$

$$\Rightarrow \log n = \frac{h}{5} \log \log n$$

$$\Rightarrow h = \theta(\log n / \log \log n)$$

Search operation at a node of fusion tree can be done in a time bound of $O(\log \log n)$. Thus overall worst case time cost of sorting n integers is $O(n \log n / \log \log n)$. This is the improvement over past result shown by Raman and traditional algorithms [1].

2.2. Sorting in $O(n \sqrt{\log n})$ Expected Time [3]

Arne Andersson has proved that as an application, keys can be sorted in linear time of $O(n \sqrt{\log n})$ worst-case cost [3]. The best previous method for deterministic sorting and searching in linear space before this has been the fusion trees which supports updates and queries in $O(\log n / \log \log n)$ amortized time and sorting in $O(n \log n / \log \log n)$ worst-case time [1]. This is accomplished with the help of a new data structure named exponential search tree.

The data structure introduced by Andersson has the following properties:

1. Its root has degree $\theta(n^{1/5})$.

2. The keys of the root are stored in a local (static) data structure, with the properties stated above. During a search, the local data structure is used to determine in which sub-tree the search is to be continued.

3. The sub-trees are exponential search trees of size $\theta(n^{4/5})$.

Andersson has also driven following results for this data structure:

1. For given n sorted keys, an exponential search tree can be constructed in linear time and space. Since the cost of constructing a node of d degree is $O(d^4)$ then the total construction cost $C(n)$ is given by:

$$C(n) = O\left(\left(n^{\frac{1}{5}}\right)^4\right) + n^{\frac{1}{5}} \cdot C\left(n^{\frac{4}{5}}\right)$$

$$\Rightarrow C(n) = O(n)$$

2. The space required by the data structure is $O(n)$.

3. The search cost $T(n)$ is given by,

$$T(n) = O\left(S\left(n^{\frac{1}{5}}\right)\right) + T(n^{4/5})$$

4. Each time some sub-trees are reconstructed, the degree of the root will change and the root must be reconstructed. The cost of this reconstruction is $O\left(\left(n^{\frac{1}{5}}\right)^4\right)$.

Again, this is linear in the size of a sub-tree. Hence, the amortized cost of reconstructing the root is $O(1)$. This gives the following equation for the amortized restructuring cost $R(n)$:

$$R(n) = O(1) + R\left(n^{\frac{4}{5}}\right)$$

$$\Rightarrow R(n) = O(\log \log n)$$

5. The cost of searching is also $O(\log \log n)$.

Using the exponential search tree data structure Andersson has proven the following:

$$T(n) = O\left(\min.\left(1 + \frac{\log n}{\log w}, \log w\right)\right) + T(n^{4/5})$$

Balancing the two parts of the min-expression gives:

$$T(n) = O(\sqrt{\log n}) + T(n^{4/5})$$

And hence,

$$T(n) = O(\sqrt{\log n})$$

This shows that n integers can be sorted in $O(\sqrt{\log n})$ expected time in linear space [3]. This result is optimized over previous traditional results and integer sorting using fusion tree [1].

2.3. Sorting in $O(n(\log \log n)^2)$ Expected Time [5, 6]

The whole process of sorting in $O(n(\log \log n)^2)$ consists of mainly three functions discussed below [5]. This was first proved by Thorup [4, 10] and later given by Yijie Han [5]. This time complexity is achieved using word comparison. The parallelism of uniprocessor environment is exploited in order to achieve this. The bits of integers are divided into smaller size and a set is made and the sorting is applied to this set. The process is as follows [5]:

Sort (level, a_0, a_1, \dots, a_t) [5].

/* a_i 's are the input integers in a set to be sorted, level is the recursion level. */

Step-1: If level = 1 then examine the size of the set (i.e. t). If the size of the set is less than or equal to \sqrt{n} , then return. Otherwise use the current block to divide the set into at most three sets. For the set all of its elements are equal to the median eliminate the current block and note the next block to become the current block. Create a label which is the set number (0, 1, or 2 because the set is divided into at most three sets) for each integer. Then reverse the computation to route the labels for each integer back to the position where the integer located in the input to the procedure call. Also

route a number (a 2-bit number) for each integer indicating the current block back to the location of the integer.

Return.

Step-2: Cut the bits in each integer a_i into equal two segments a_i^{High} (high order bits) and a_i^{Low} (low order bits). Pack a_i^{High} 's into half the number of words.

Call Sort (level-1, a_0^{High} , a_1^{High} , ..., a_t^{High}).

/* when the algorithm returns from this recursive call the label for each integer indicating the set the integer belongs is already routed back to the position where the integer locates in the input of the procedure call. A number having at most the number of bits in a_i indicating the current block in a_i is also routed back to a_i . */

Step-3: For each integer a_i extract out a_i^{Low} which has half the number of bits as in a_i and is a continuous segment with the most significant block being the current block of a_i . Pack a_i^{Low} 's into half the number of words as in the input.

Step-4: For each set $S = \{a_{i_0}, a_{i_1}, \dots, a_{i_s}\}$ call Sort(level - 1, $a_{i_0}^{Low}$, $a_{i_1}^{Low}$, ..., $a_{i_s}^{Low}$).

Step-5: Route the label which is the set number for each integer back to the position where the integer located in the input to the procedure call. Also route a number (a $2(level + 1)$) bit number for each integer indicating the current block back to the location of the integer. This step is the reverse of the routing in Step 3.

IterateSort() [5].

Call Sort ($\log((\log n)/4)$, a_0 , a_1 , ..., a_{n-1});

For $j = 1$ to 5 do

Begin

Move a_i to its set by bucket sorting because there are only about \sqrt{n} sets;

For each set $S = \{a_{i_0}, a_{i_1}, \dots, a_{i_t}\}$

If $t > \sqrt{n}$ then call Sort ($\log((\log n)/4)$, $a_{i_0}, a_{i_1}, \dots, a_{i_t}$);

End.

Then $(3/2) \log n$ calls to the *level 1* procedure are executed. Blocks can be eliminated at most $\log n$ times. The other $(1/2) \log n$ calls are sufficient to partition the input set of size n into sets of size no larger than \sqrt{n} .

At *level j*, only $n/2^{\log((\log n)/4)-j}$ words are used to store small integers. Each call to the Sort procedure involves a sorting on labels and a transposition of packed integers and therefore involves a factor of $\log \log n$ in time complexity. Thus the time complexity of algorithm is:

$$T(\text{level}) = 2T(\text{level} - 1) + cn \log \log n / 2^{\log((\log n)/4)-\text{level}}$$

$$T(0) = 0$$

Where c is a constant. Thus, $T(\log((\log n)/4)) = O(n(\log \log n)^2)$. Algorithm IterateSort only sorts sets into sizes less than \sqrt{n} . Another recursion is needed to sort sets of size less than \sqrt{n} . This recursion has $O(\log \log n)$ levels. Thus the time complexity to have the input integers sorted is $O(n(\log \log n)^3)$.

The sorting process is not stable. Since the sorting is done for arbitrarily large integers, the address bits to each input integer can be appended to stabilize the sorting. Although this requires that each word contains $\log m + \log n$ bits, when $m \geq n$ the number of bits for each word can be kept at $\log m$ by using the idea of radix sorting, namely sorting $\log m + \log n$ bits in each pass.

The space used for each next level of recursion in Sort uses half the size of the space. After recursion returns the space can be reclaimed. Thus the space used is linear i.e. $O(n)$.

The following technique is applied to further improve the time complexity of the algorithm [5, 6]:

The $\log m$ bits of an integer are divided into $\log(\text{setsize}) \log \log n$ blocks with each block containing $(\log m)/(\log(\text{setsize}) \log \log n)$ bits, where setsize is the size of the set where integers are to sort into. Initially, $\text{setsize} = \sqrt{n}$. The algorithm $\text{SpeedSort}()$ is executed with $\text{setsize} = \sqrt{n}$:

$\text{SpeedSort}(\text{setsize})$ [5, 6].

While there is a set S which has $\text{size} > \text{setsize}$ do begin

Step-1: For each integer $a_i \in S$ extract out a' which contains $\log(\text{setsize})$ continuous blocks of a_i with the most significant block being the current block, put all a_i 's in S' ;

Step-2: Call IterateSort on set S' ;

End.

The whole sorting process consists of $\log \log n$ levels of calling SpeedSort : $\text{SpeedSort}(n^{1/2})$, $\text{SpeedSort}(n^{1/4})$, $\text{SpeedSort}(n^{1/8})$,, $\text{SpeedSort}(n^{1/2^i})$. In $\text{SpeedSort}(n^{1/2^i})$ each word stores $\log(\text{setsize}) = \log n/2^i$ blocks and each block contains $\log m/(\log(\text{setsize}) \log \log n) = 2^i \log m/(\log n \log \log n)$ bits. Therefore during the sorting process each word stores no more than $\log m/ \log \log n$ bits of integer data. One iteration of the while loop in any of the SpeedSort 's takes $O(\log \log n)$ time for each integer. The account is taken for the time for each integer in the whole sorting process by two variables D and E . If an integer a has gone through g_i iterations of the while loop of $\text{SpeedSort}(n^{1/2^i})$ then $(g_i - 1) \log m/ \log \log n$ bits of a have been eliminated in $\text{SpeedSort}(n^{1/2^i})$. The value $O((g_i - 1) \log \log n)$ is added to variable E indicating that much time has been expended to eliminate $(g_i - 1) \log m/ \log \log n$ bits. The value $O(\log \log n)$ is also added to variable D indicating that much time has been expended to divide the set in $\text{SpeedSort}(n^{1/2^i})$. Because of elimination of at most $\log m$ bits, the value of E is up bounded by $\sum_i g_i \log \log n = O((\log \log n)^2)$ throughout all levels of $\text{SpeedSort}()$ invocations. The value of variable D is also up bounded by $O((\log \log n)^2)$ because

there are $\log \log n$ levels of SpeedSort() invocations. Therefore, n integers can be sorted in linear space in $O(n(\log \log n)^2)$ time [5, 6].

2.4. Other Fast Deterministic Sorting Algorithms

The recent research in area of integer sorting has given many new ideas [1-24]. These ideas are either based on exponential tree or packed sorting technique or a combination of both. Fredman and Willard's concept of sorting n integers in $O(n \log n / \log \log n)$ expected time and linear space was based on fusion tree [1]. Raman showed that sorting can be done in $O(n \sqrt{\log n \log \log n})$ time in linear space [2].

Yijie Han showed $O(n \log \log n \log \log \log n)$ time for deterministic linear space integer sorting using a variant of exponential tree [6]. Yijie Han later show that n integers can be sorted in $O(n \log \log n)$ expected time in linear space using multi-dividing in exponential tree [7]. He used signature sorting to accomplish multi-dividing. This is one of the most important results in area of integer sorting, so it is discussed in detail in following sub-section. Yijie Han's one more result show that n integers can be sorted in $O(n \sqrt{\log \log n})$ expected time [27].

2.4.1. Yijie Han's Concept of Sorting Integers in $O(n \log \log n)$ [7]

For sorting n integers in the range $\{0, 1, 2, \dots, m - 1\}$ in $O(n \log \log n)$ expected time, it is assumed that the word length used in this conservative algorithm is $O(\log(m + n))$. In integer sorting, small integers are often packed into one word. It is also assumed that all the integers packed in a word use the same number of bits.

The basic idea of Andersson's exponential search tree is used here: a balanced search tree is constructed where the degree of a node with m descendents is $\Theta(m^{1/5})$, hence where each child has $\Theta(m^{4/5})$ descendents [3].

In Andersson's exponential search tree, integers are inserted (passed down) into the tree one at a time [3]. Thorup suggested to pass down d integers at a time, where d is the number of children of the node in the tree where integers are to be passed down [5]. This sorting algorithm will stick with this scheme, namely passing down d^2

integers at a time. The d^2 integers are not passed down all the way down the tree. Instead, the d^2 integers are passed down at a time to one level of tree until all integers are passed down one level. Thus at the root $n^{2/5}$ integers are passed down to the next level at a time. After all integers are passed down to the next level, integers essentially partitioned into $t_1 = n^{1/5}$ sets S_1, S_2, \dots, S_{t_1} with each S_i containing $n^{4/5}$ integers and $S_i < S_j$ if $i < j$. Then $n^{\left(\frac{4}{5}\right) \cdot \left(\frac{2}{5}\right)}$ integers are taken from each S_i at a time and coordinated to be passed down to the next level of the exponential search tree. This is repeated until all integers are passed down to the next level. At this time integers have been partitioned into $t_2 = n^{1/5} \cdot n^{4/25} = n^{9/25}$ sets T_1, T_2, \dots, T_{t_2} with each set containing $n^{16/25}$ integers and $T_i < T_j$ if $i < j$. Now integers are ready to be passed down to the next level in the exponential search tree. The levels of the exponential search tree are numbered top down order so that root is at level 0 [7].

The pass down can be viewed as sorting q integers in each set together with the p integers a_1, a_2, \dots, a_p in the exponential search tree so that these q integers are partitioned into $p + 1$ sets S_0, S_1, \dots, S_p such that $S_0 < \{a_1\} < S_1 < \{a_2\} < \dots < \{a_p\} < S_p$. Since the q integers don't require to be totally sorted and $q = p^2$. The linear timed multi-dividing technique is used to accomplish this. Signature sorting can be used to accomplish multi-dividing [7].

This technique of sorting integers gives a time bound of $O(n \log \log n)$ in linear space [7].

Problem Statement

The optimization of the deterministic integer sorting is still a challenging task. Many researchers have come forward with different idea. These ideas are very attractive in form of expected time taken by them and the space requirements. But the methodology used in these ideas is very difficult to understand and impractical to implement. Thus the first task is to simplify the idea in order to make the concept useful for real environment. The data structure used in recent deterministic algorithm is very complex not only in design but also for implementation. The exponential tree described by researchers is of different kind [3, 7, 8]. It has many layouts and designs. Many layouts of them are very complex as there are a lot of pointers to handle. These designs have one key per node. Thus these designs are very impractical for any value of input integers as there is more memory used for pointers as compared to integers. Hence, the second task is to re-define or re-design the layout of exponential tree in such a way that it becomes optimized for memory requirements and give comparatively good expected time for insertion and other operations.

The third and final task is to achieve optimized expected time using the simplified idea and new design of exponential tree. Achieving the optimized expected time is not only task to be done but it should also give good performance in real time. This means that the implementation should be minded while developing the algorithm. All these three tasks are very important in order to propose an algorithm which is not only optimized but also practical in nature.

Improved Exponential Tree Integer Sorting Algorithm Using Node Growth

In this section, solution to the problem discussed in above section is provided. The solution can be broadly categorized in two parts. The first part is to modify exponential tree so that it can be used and implemented properly. The second part includes designing the algorithm with logics for insertion, modifying binary search and logic for in-order tracing. This section will also provide the pseudo code for sorting algorithm with implementation.

4.1. Modified Exponential Tree

The exponential tree was first introduced by Andersson in his research for fast deterministic algorithm for integer sorting [3]. 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 $d = 1$. So the second level can hold two nodes, the third can hold eight nodes, the fourth 64 nodes, and so on [7]. This shows that number of children at each level increased by a multiplicative factor of 2 i.e. exponential increase in number of children at each level [7].

The tree itself is very complex to handle as number of integer increases. Also there is a need to handle more pointers at each level on each node. Thus the exponential tree needs to be modified. The modified concept of exponential tree will provide a convenient way for integer sorting. Instead of focusing number of nodes present in

tree it is profitable to focus on number of integers present in the tree as the concern is about how to use it for integer sorting.

A tree with properties of binary search tree will be called 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 key i .

The total number of integers hold by the tree up to level k will be the addition of total number of integers up to level $k - 1$ and integers present on that level. This will be given by the following formula:

$$N_k = N_{k-1} + k * k! \quad \dots\dots\dots (1)$$

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 if k th level and $k!$ denotes factorial of k .

Thus the node at root will look like as:

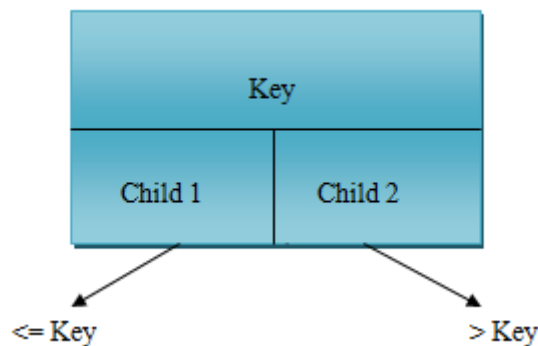


Figure-4.1: Root node of modified exponential tree.

And the node at *ith* level or depth will look like as:

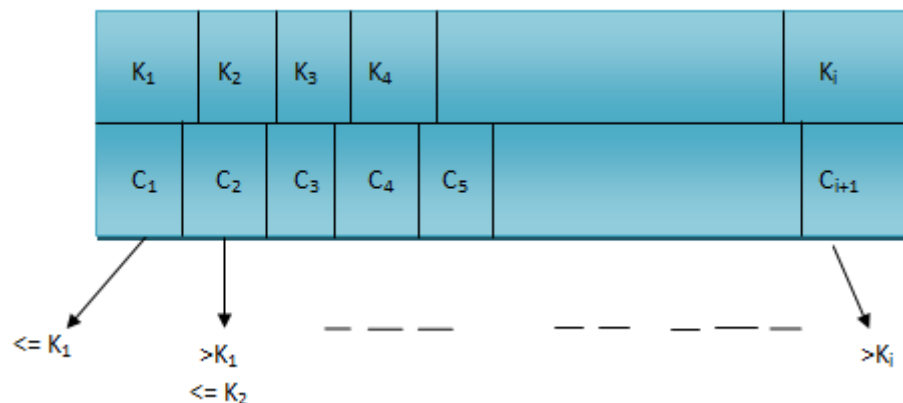


Figure-4.2: A node of at *ith* level of modified exponential tree.

The height of the tree will remain $O(\log \log n)$ which can be proved by use of induction. The modification will not only reduce the complexity of exponential tree involved in implementing it but also improves the balancing method as well as sorting technique. Integer sorting will be more convenient and fast with this modification.

The implementation of exponential tree requires creation of the exponential tree node. Here is a skeleton for exponential node which is used in implementation for integer sorting:

```
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.

It must be noted here that while creating a new child node for a parent, the variables held by that node must be initialized with default values. This will help to avoid error introduced by the garbage values present on those memory locations which are assigned to the variables of the new node.

4.1.1. Growth

The growth of this modified exponential tree is also exponential in nature. But this modified design provides an easy implementation. The concern is about to handle as much integers as possible up to a particular level. This is because the data structure is particularly designed for integer sorting. The figure-4.3 shows a fully filled and balanced exponential tree.

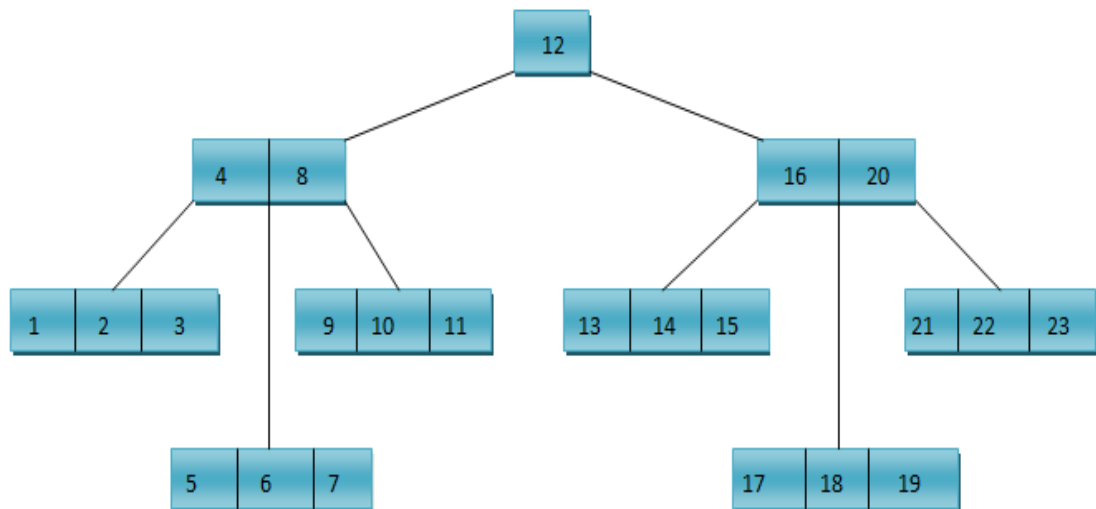


Figure-4.3: A balanced and fully filled modified exponential tree

It is important to see here that total number of integers become nearly ten times after each level up to level 9. After that the number of integers increases more than expectations.

Table-4.1: Total number of integers in modified exponential tree

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

The figure-4.4 shows that the modified tree also has exponential growth with respect to total number of integers present in the tree.

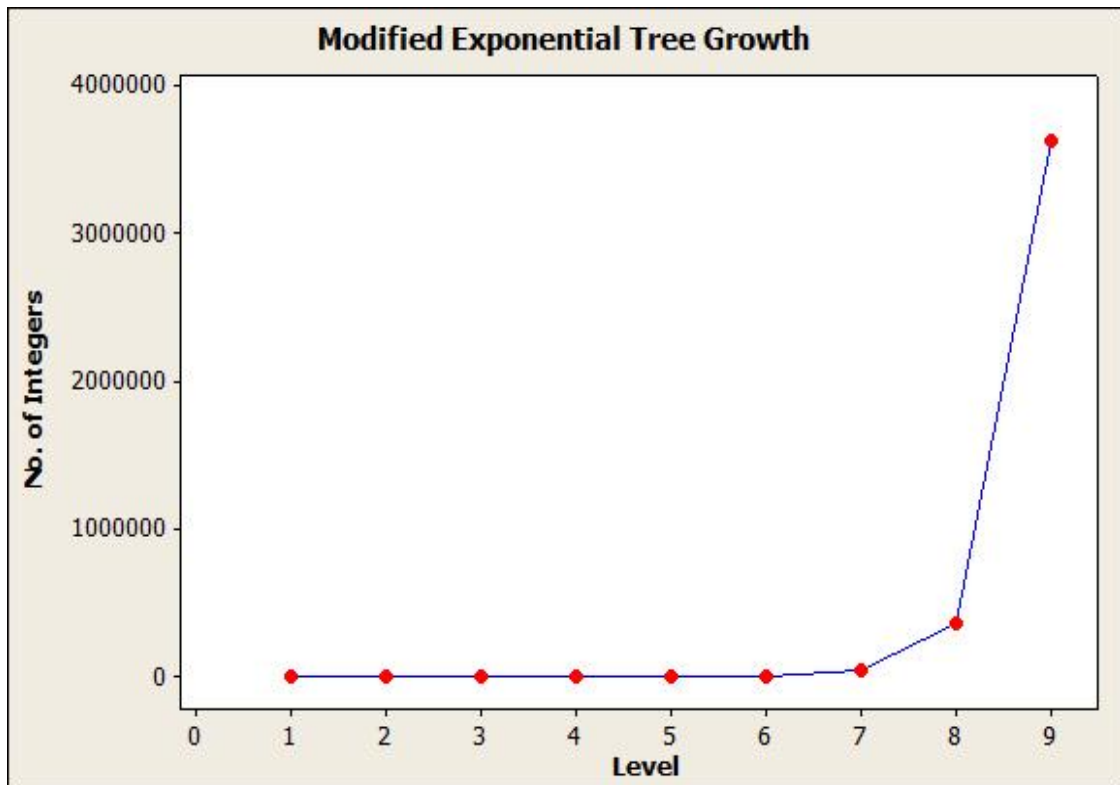


Figure-4.4: Growth plot of modified exponential tree

4.1.2. Balancing

The balancing of exponential tree is one of the most important and necessary task in order to achieve the $O(n \log \log n)$ expected time in linear space. If the tree is not

balanced then the worst case might happen and the expected running time will increase to undesired level. The balancing will guarantee that the depth of tree will remain $O(\log \log n)$ which will lead us to our target. The tree will be balanced if the difference between depths (or heights) of both sided children of a key will not exceed by 2. The balancing will happen on all the children of the node. Each key will play a role of a node like in binary search tree in balancing. In order to balance the tree, the track of number of keys passed through a particular node can be kept. This track will help when to call the balancing procedure. The difference between depths of two children can be calculated by checking how many keys have been passed to those children. If this difference will increase the maximum number of keys the node can hold then the tree need to be balanced.

The strictly balanced tree concept can enhance the use of tree with a little overhead of processing. The tree will be called strictly balanced if the difference between heights (or depths) of both sided children of a key will not exceed by 1. This balancing will help to achieve a true bound of $O(n \log \log n)$ expected time. The self balancing technique which is already in existence in self balancing binary tree can be used while implementing the balancing for exponential tree. It is up to the developer which balancing technique is used.

4.2. Insertion

The sorting of integers happens with inserting them into exponential tree. After inserting the integers into tree, trace the tree in in-order to get the sorted list as output. As proved by Andersson, insertion into exponential tree takes most of the time, thus the focus is to achieve the insertion in lesser expected time [3].

As discussed above the exponential tree is a binary search tree itself, thus the insertion requires finding the appropriate place of the integer. This will happen by comparing integer with keys presented in any node. There are at most k number of keys in any node at depth or level k , which means there will be at most k number of comparisons for passing the integer through that level.

Andersson has proved that insertion can be done in $O(n\sqrt{\log n})$ expected time by passing down integer to exponential tree one by one, which does not seem well

enough optimized [3]. But in the best case scenario the insertion will take $O(n \log \log n)$ time as only one comparison will be required at each level. The best case happens rarely, thus the best case scenario is not reliable. Hence, the further modification in concept is required to achieve better result.

As discussed passing down integers one at a time does not give well enough expected time. Yijie Han came up with the idea of passing down the integers in group [7]. He used multi-dividing technique to split the integers into smaller lists. He suggested that instead of inserting integer one at a time into the exponential search tree one can pass down all integers to 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 technique gives an expected time of $O(n \log \log n)$ in linear space [7]. But it is very difficult to pass all integers at once if the number of integers is very large. Thus the insertion of integers in groups does not seem very good in terms of implementation and complexity associated in passing down all integers at once.

The algorithm discussed in this thesis will pass down integers one by one in modified design. The algorithm will give a complexity of $O(n \log \log n \log \log \log n)$ by reducing the number of comparison required at each level.

The insertion method in which the modified binary search is used is described below and later modified binary search. The insertion method is as follows:

Insert(Node *root,int element)

Step-1: Set *ptr=root, *parent=NULL, i=0.

Step-2: Repeat step 3 to 6 while ptr <> NULL.

Step-3: Set level=ptr->level, count=ptr->count.

Step-4: Call i=BinarySearch(ptr, element).

Step-5: 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.

Step-6: Set parent=ptr, ptr=ptr->child[i].

Step-7: Create a new Exponential Node at *ith* child of parent and insert element in that.

Step-8: Return.

The tree has a depth or height of $O(\log \log n)$. Thus each integer requires to be passed down maximum of $O(\log \log n)$ levels. Considering that at each level only $O(1)$ expected number of comparisons happens, this will give overall time bound of $O(\log \log n)$ for insertion of each integer. To sort a sequence of n integers, all integers have to be passed down. Thus the total expected time will be $O(n \log \log n)$. But this is not the case in this algorithm. The number of comparison at each level is not $O(1)$. The number of comparisons at each level is given by the modified binary search. The traditional binary search takes $O(\log m)$ expected time; same is the case with binary search discussed below. This implies that at each level there will be at most $O(\log k)$ comparisons where k is the keys hold by the node i.e. depth of height of the node. Hence for each integer to be passed down to the tree there will be at most $O(\log \log n \log k)$ expected number of comparisons. For a sequence of n integers the expected time of the algorithm will be $O(n \log \log n \log k)$. The maximum value of k could be $O(\log \log n)$ i.e. the height of the tree. Thus the overall complexity of the algorithm will be $O(n \log \log n \log \log \log n)$ in linear space.

4.3. Binary Search

As discussed in the definition of the exponential tree, all integers presented in a node of exponential tree must be sorted thus this property can be exploited to enhance the

performance. In order to search the position in a sorted list, binary search can be used with little modification. The binary search takes $O(\log n)$ expected time in searching. Hence the performance will be enhanced by the use of binary search instead of linear search.

It must be noted here that the binary search is only used inside node of the exponential tree. It will give the position at which the element should be inserted. If there is space for more elements in that node only then the element will be inserted otherwise the algorithm moves to next level of exponential tree i.e. the predecessor child of that particular node. The modified binary search is as under:

BinarySearch(Node *ptr,int element)

Step-1: If element > ptr->data[count-1] then return ptr->count.

Step-2: Set start=0, end=ptr->count-1, mid= (start + end)/2.

Step-3: Repeat step 4 & 5 while start < end.

Step-4: If element > ptr->data[mid] then start=mid+1

else end=mid

Step-5: Set mid= (start + end)/2.

Step-6: return mid.

4.4. Trace

When all the integers will be inserted in the tree, then the in-order trace of the tree will give us the desired sorted sequence. The in-order trace for the exponential tree is same as for binary tree. The only difference is that first the predecessor for a key is traced before tracing the key and when all keys of a node is traced then the rightmost child is traced. The in-order trace will be:

In-Order-Tace(Node *r)

Step-1: Set count=r->count

Step-2: Repeat For $i=0$ to count by 1

Step-3: If $r \rightarrow \text{child}[i] \neq \text{NULL}$ call $\text{In-Order-Trace}(r \rightarrow \text{child}[i])$.

Step-4: Print $r \rightarrow \text{data}[i]$.

Step-5: If $r \rightarrow \text{child}[\text{count}] \neq \text{NULL}$ call $\text{In-Order-Trace}(r \rightarrow \text{child}[\text{count}])$.

Step-6: Return.

The tracing of the tree will take $O(n)$ expected time which is linear. This is because each integer is accessed only one time.

The overall combined complexity of above algorithms will be $O(n \log \log n \log \log \log n)$. This shows that the desired result has been achieved using new design of exponential tree and modified binary search. The algorithm runs in linear space.

5.1. Testing

This section will show the comparison of the performance of the exponential tree sorting with binary tree sorting and quick sort. It is obvious to think here that the comparison of the exponential tree sorting must be done with Quick sort only which is best known and widely used sorting technique; but as quick sort is mainly used for integers stored at consecutive memory location i.e. array, here the exponential tree sorting works on non-consecutive memory location. Hence binary tree sorting is also considered for comparison. The comparison includes CPU running time and Memory requirement.

The quick sort technique and binary tree sorting is used to compare the running time while only binary tree sorting technique is used for comparing the space requirement. This is because the quick sort is applied on array data structure while the binary tree sorting is applied on binary tree which is a tree structure relevant to the exponential tree data structure.

The input integers are generated by using random function and stored in a file. Later that file is used as an input for all sorting algorithms. The output of each sorting technique is also stored in a file to avoid the loss of output sequence which happens when printing the sequence on console. Other benefit of doing so is that the output of each algorithm later can be compared to ensure the desired result.

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. This helped to

measure running time accurately and very near to actual time taken by algorithm. This data is the result obtained immediately after running defragmentation of the memory to avoid any variation due to different of the job being stored at non-continuous location far away from each other.

5.1.1. Running Time

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 all 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. The running-time measurement is started with N=1024 and in each step the value of n is doubled to see the effect of increase in number of input integers. It is noted that the running time increases linearly with increase in number of input integers.

Table-5.1: 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

5.1.1.1. Analysis of Exponential Tree Sorting Running Time

The figure-5.1 shows the CPU running time for the Exponential tree sorting, which clearly depicts that the slope of graph is linear.

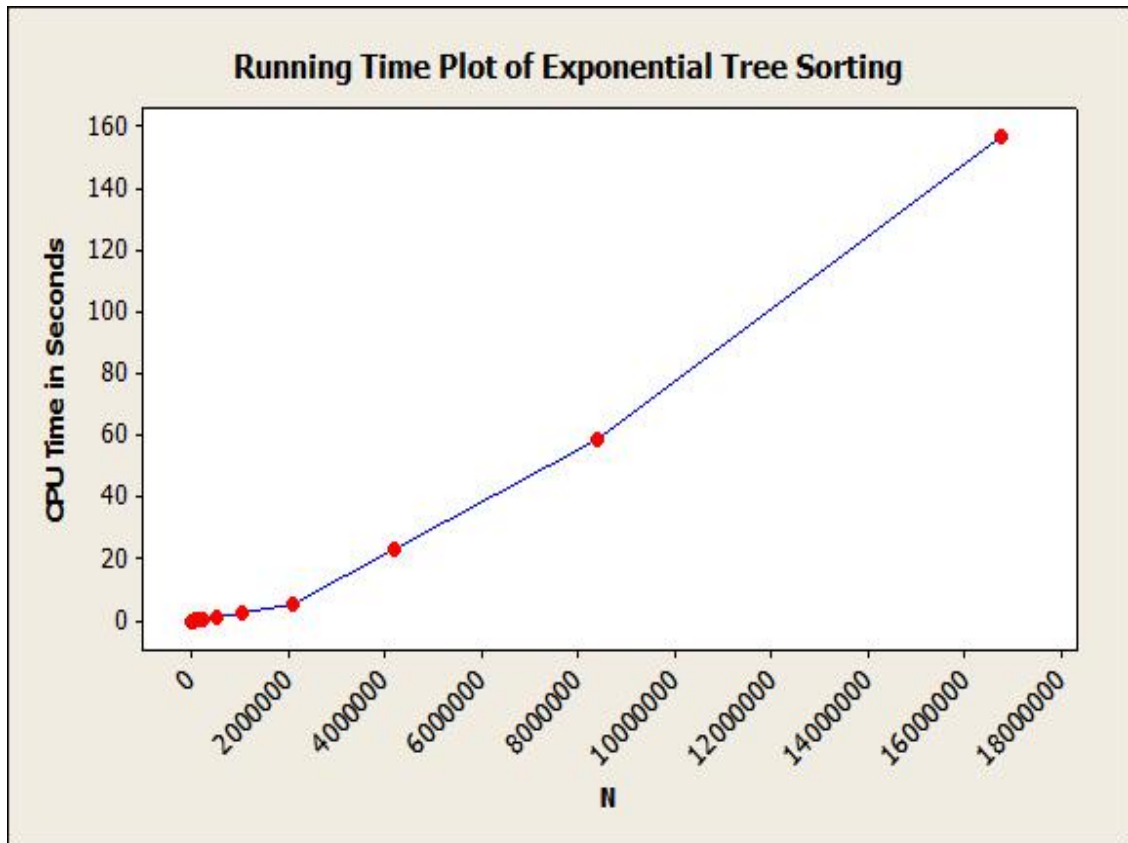


Figure-5.1: CPU running time plot for exponential tree sorting

The increase in running time of algorithm is directly proportional to number of input integers. The slope clearly shows that whenever number of input integers is increased then the running time is increased proportionally. This implies that the performance of the exponential tree sorting is very good.

5.1.1.2. Exponential Tree Sorting v/s Binary Tree Sorting

The figure-5.2 shows the comparison between CPU running times for the exponential tree sorting and binary tree sorting which depicts that the exponential tree sorting takes relatively very less CPU time for same number of integers than taken by binary tree sorting. As the number of integers increases the CPU time for exponential tree sorting increase with very small factor than binary tree sorting.

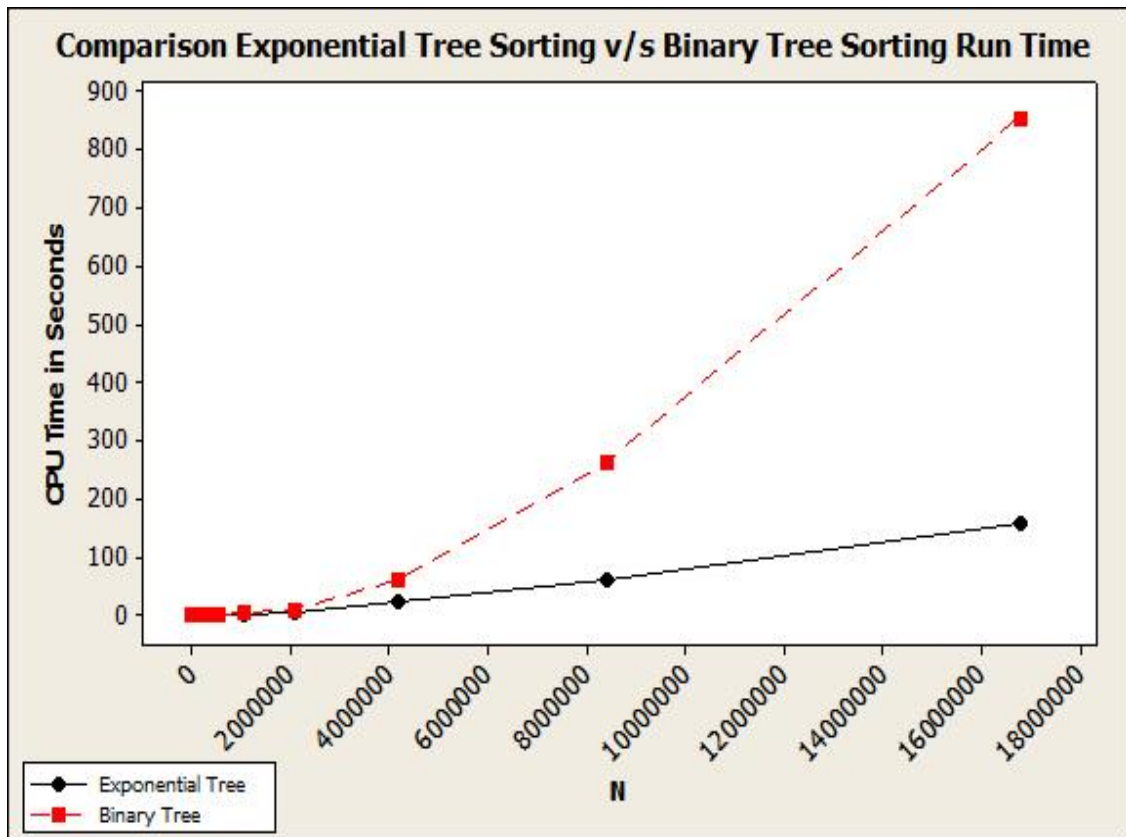


Figure-5.2: Exponential tree sorting v/s binary tree sorting running time

5.1.1.3. Exponential Tree Sorting v/s Quick Sort

The figure-5.3 shows the graph plotted between exponential tree sorting run time and quick sort run time. The line plot for both algorithms has a linear slope. The exponential tree gives better running time than quick sort for smaller number of input integers i.e. N . But after a particular value of N the running time of exponential tree is more than the quick sort. The reason behind this change is that as the number of integer increases new nodes in exponential tree are created. The creation of new node requires the use of new operators and also assigning of default values to node's variables. The running time of exponential tree also includes the running time taken by the increment of counter at each node and shifting of integers to create space for new integer. Thus after a particular value of N the running time for exponential tree sorting is more than quick sort.

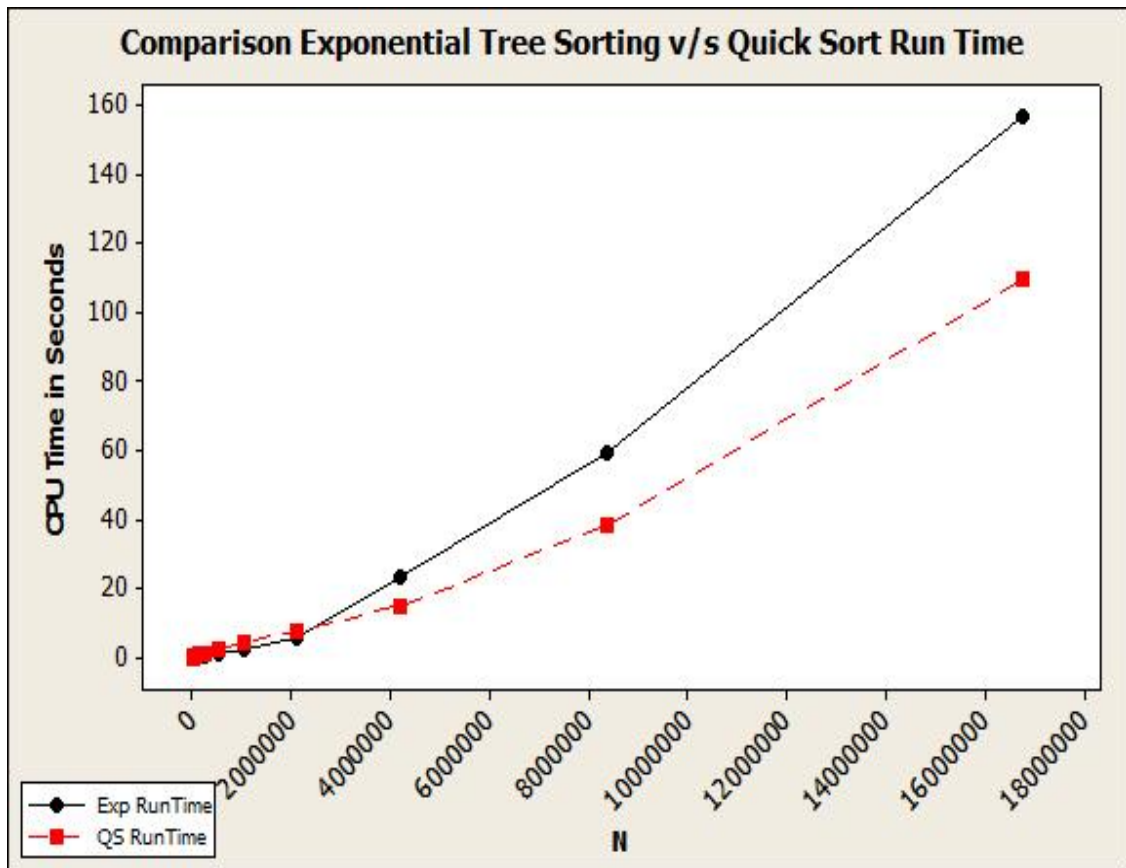


Figure-5.3: Exponential tree sorting v/s quick sort running time

5.1.1.4. Running Time Result

The figure-5.4 shows the overall comparison between all three algorithms. The graph clearly shows that the binary tree sorting gives worst running time among all. The exponential tree sorting gives a running time near to quick sort. The running time of exponential tree sorting is far better than the binary tree sorting. When the number of input integers is smaller, then the performance of quick sort and exponential tree is nearly same while the binary tree has a very bad performance even for small number of input integers. In the case of very large number of input integers, the running time for quick sort is slightly less than the exponential tree sorting. This is because of extra operation required while creation of nodes. The performance of binary tree is even worst for very large number of input integers.

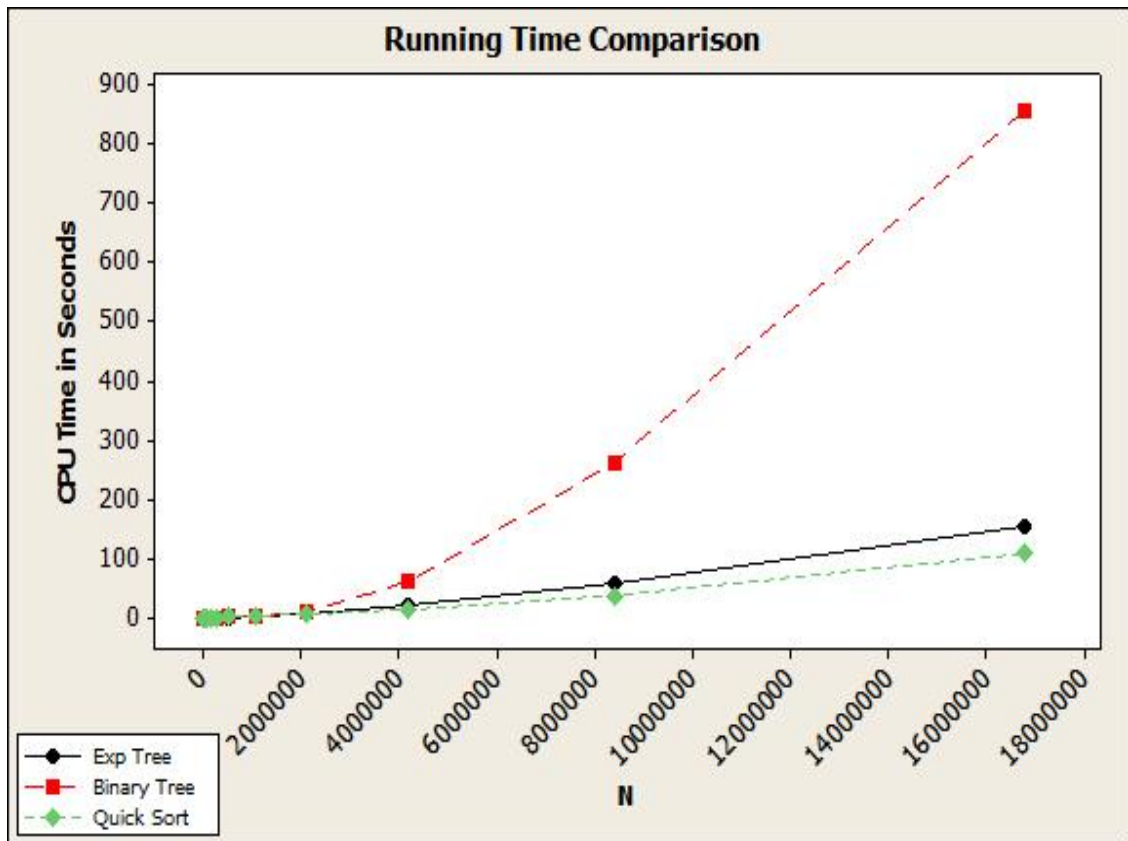


Figure-5.4: CPU running time comparison

5.1.2. Space 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-5.2: 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

5.1.2.1. Analysis of Exponential Tree Sorting Space Requirement

The figure-5.5 shows the memory requirement plot for the exponential tree which depicts that the graph has a linear slope. The memory requirement increases directly proportional to number of integers to be sorted. The memory used by exponential tree includes the memory for pointers created for children, the track of number of integers present in the node, depth or height of the node and the array used for storing the key values of the node. The memory requirement for the exponential tree is comparatively low.

It can be seen from the plot that the memory used by the exponential tree sort is linear as the slope of plot is linear. This proves the result that the modified design does integer sorting in linear space. The linear space requirement is best in nature as not more than a specific memory is required to perform the sorting operation. Thus, the algorithm has a good and optimized space requirement.

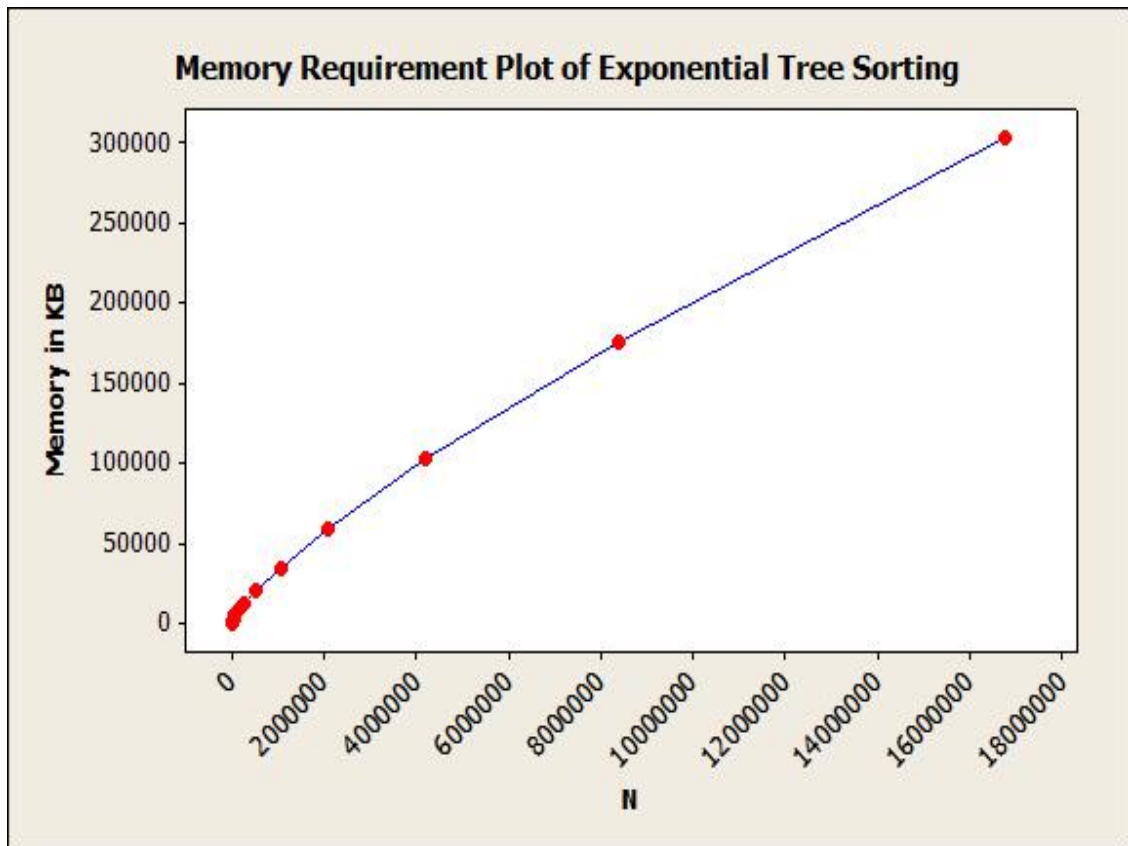


Figure-5.5: Memory requirement plot for exponential tree sorting.

5.1.2.2. Exponential Tree Sorting v/s Binary 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 an integer present in binary tree node, whereas exponential tree uses $m + 1$ pointers with m integers as there are m integers presented in exponential tree node. The difference of memory used by exponential tree as compared to the binary tree is very high even for very large number of input integers. This shows that the space requirement of exponential tree sorting is far better than the binary tree sorting. Hence the optimization in space requirement is achieved relatively.

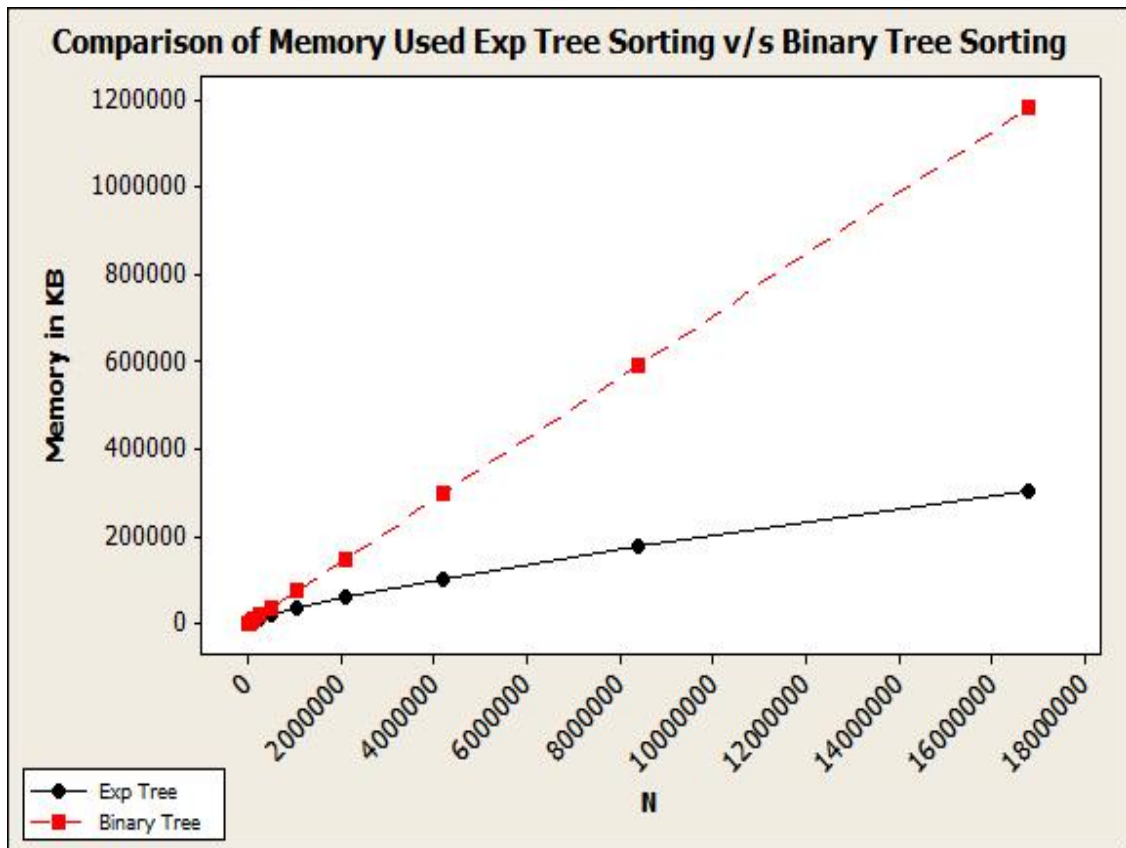


Figure-5.6: Memory requirement comparison

5.2. Results

The figure-5.7 shows the trial run of the algorithm. First the number of input integers is scanned from the file. After that all input integers are scanned one at a time and inserted into the modified exponential tree. This will reduce the requirement of maintaining list of input integers as integers are directly inserted into the tree. The passing of integers one at a time provides this feature.

After all integers are inserted, due to properties of modified exponential tree all integers will be in sorted sequence. The in-order trace of the tree will give the desired sorted output. This is also shown in figure-5.7. Hence the algorithm successfully runs and provides the integer sorting.

```
Number of input integers:: 256

Original Sequence::
3856 21874 10596 11160 15345 6751 26362 25669 29726 14433 17136 3155 25954 2651
17110 24732 20659 13188 15207 1329 9809 1111 1428 29967 3838 2411 2380 2497 1212
3 8842 23393 13734 9848 16289 22759 23637 4978 3721 13063 15099 22891 8770 17825
25465 18542 10594 13790 832 20994 26051 22601 22538 10194 6275 17844 5156 13419
4970 7032 22025 16558 4768 30516 15710 23100 23588 3010 26367 4074 31854 15996
23760 2552 18481 28193 18209 11882 31327 3923 14847 4814 6271 8320 16594 18888 1
3274 26172 9182 29143 2658 5374 9042 15030 15429 16532 9211 9267 10261 27091 272
44 17376 9248 9060 12423 30136 32241 1404 24767 7319 20477 4041 28058 22281 1180
1 20039 3230 741 27126 16084 12046 10921 28250 21163 29100 16172 18360 14528 90
22696 16014 6493 18844 6571 20961 26939 11965 28439 3504 11533 23285 23479 5647
2129 23203 16065 24469 30632 27687 12953 16132 12217 15977 3286 33 9229 14679 15
763 26970 16423 31607 2965 27743 23751 5239 25010 19310 9866 25265 27050 27120 3
0295 28644 29907 24366 23787 14032 28182 6759 1290 12856 2622 22479 4248 10137 1
1150 16235 10004 17465 15680 16306 26954 19298 2021 24120 25897 4442 3075 19789
14018 2231 25914 4092 4387 7566 18827 5174 11542 30487 24962 31712 17773 9518 24
01 20978 15107 30961 13662 7599 19688 13767 18464 19444 28289 16463 29465 28313
13522 26477 29143 12615 7532 17939 29514 29443 5954 29331 20026 14312 4761 14505
22626 30959 12298 32237 9300 13420 14802 3116 18712 15557 31028 10554 30025 450
4 17760 283

Sorted Sequence::
33 90 283 741 832 1111 1290 1329 1404 1428 2021 2129 2231 2380 2401 2411 2497 25
52 2622 2651 2658 2965 3010 3075 3116 3155 3230 3286 3504 3721 3838 3856 3923 40
41 4074 4092 4248 4387 4442 4504 4761 4768 4814 4970 4978 5156 5174 5239 5374 56
47 5954 6271 6275 6493 6571 6751 6759 7032 7319 7532 7566 7599 8320 8770 8842 90
42 9060 9182 9211 9229 9248 9267 9300 9518 9809 9848 9866 10004 10137 10194 1026
1 10554 10594 10596 10921 11150 11160 11533 11542 11801 11882 11965 12046 12123
12217 12298 12423 12615 12856 12953 13063 13188 13274 13419 13420 13522 13662 13
734 13767 13790 14018 14032 14312 14433 14505 14528 14679 14802 14847 15030 1509
9 15107 15207 15345 15429 15557 15680 15710 15763 15977 15996 16014 16065 16084
16132 16172 16235 16289 16306 16423 16463 16532 16558 16594 17110 17136 17376 17
465 17760 17773 17825 17844 17939 18209 18360 18464 18481 18542 18712 18827 1884
4 18888 19298 19310 19444 19688 19789 20026 20039 20477 20659 20961 20978 20994
21163 21874 22025 22281 22479 22538 22601 22626 22696 22759 22891 23100 23203 23
285 23393 23479 23588 23637 23751 23760 23787 24120 24366 24469 24732 24767 2496
2 25010 25265 25465 25669 25897 25914 25954 26051 26172 26362 26367 26477 26939
26954 26970 27050 27091 27120 27126 27244 27687 27743 28058 28182 28193 28250 28
289 28313 28439 28644 29100 29143 29143 29331 29443 29465 29514 29726 29907 2996
7 30025 30136 30295 30487 30516 30632 30959 30961 31028 31327 31607 31712 31854
32237 32241
```

Figure-5.7: Output

Conclusion and Future Scope

6.1. Conclusion

This thesis has presented a different idea to achieve the complexity of deterministic algorithm for integer sorting in $O(n \log \log n \log \log \log n)$ expected time and linear space with conservative advantage. This algorithm is easy to implement and simple. In order to achieve this complexity, the data structure used is exponential tree which has been modified to simplify the design and implementation. The idea has been inherited from Andersson's exponential tree.

The exponential tree presented in this thesis has a simple layout. The modified design is not only easy to understand but also simple to implement. The main focus of modified design is on to handle as many integers as possible with simple layout. The new design has achieved the optimized height and complexity associated with insertion. The height of the new data structure is $O(\log \log n)$ which is very fascinating. It is also proved that the design is also optimized for memory usage. It requires less memory than its counterparts.

The integer sorting is performed in $O(n \log \log n \log \log n)$ expected time in linear space using exponential tree. In order to achieve this expected time, the algorithm has used binary search with little modification with exponential tree. The basic concept of binary search is not modified but it is used according to requirements. Instead of searching the position of a key it searches for the successor of the key so that key can be inserted before it.

The implementation has shown that the algorithm has a very good performance on both criteria of running time as well as memory requirements. It has competing performance with quick sort for running time and far better than the binary tree sorting. The memory requirements of exponential tree sorting is also very less as

compared to binary tree sorting. It is needless to say that the exponential tree sorting is more preferable to binary tree sorting.

6.2. Future Scope

The true lower bound for integer sorting is not $O(n \log \log n)$ but it is still to be proved. Work can be done to further optimized complexity to achieve lower bound for integer sorting. This will require lots of efforts and may be achieved using randomized algorithms. The implementation of new concept with better performance or with expected performance is also a challenging task which is required to be worked upon.

This complexity is theoretically achieved using exponential tree. This might be achieved using any other data structure that can provide better time complexity than exponential tree. The new data structure, which may be proposed, can be having simple layout and design. It is also a challenging task to produce new data structure especially for integer sorting purpose.

References

- [1] Fredman M. L., and Willard D. E., Surpassing the information theoretic bound with fusion trees, *J. Comput. System Sci.*, vol. 47, pp. 424-436, 1994.
- [2] Andersson A., Hagerup T., Nilsson S., and Raman R., Sorting in linear time?, *J. Comput. Syst. Sci.*, vol. 57, no. 1, pp. 74-93, 1998.
- [3] Andersson A., Fast deterministic sorting and searching in linear space, in "Proc. 1996 IEEE Symp. on Foundations of Computer Science," pp. 135-141, 1996.
- [4] Thorup M., Fast deterministic sorting and priority queues in linear space, in "Proc. 1998 ACM-SIAM Symp. on Discrete Algorithms (SODA'98)," pp. 550-555, 1998.
- [5] Han Y., Fast integer sorting in linear space in "Proc. Symp. Theoretical Aspects of Computing (STACS'2000), February 2000," Lecture Notes in Computer Science, vol. 1170, pp. 242-253, 2000.
- [6] Y. Han, Improved fast integer sorting in linear space, *Inform. and Comput.*, vol. 170, no.1, pp. 81-94, 2001.
- [7] Y. Han, Deterministic sorting in $O(n \log \log n)$ time and linear space, *Journal of Algorithms*, vol. 50, no. 1, January 2004, pp. 96-105, 2004.
- [8] Michael A. Bender , Richard Cole , Rajeev Raman, Exponential Structures for Efficient Cache-Oblivious Algorithms, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pp. 195-207, July 08-13, 2002.
- [9] 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.
- [10] Thorup M., 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 Symp. on Discrete Algorithms (SODA'97)," pp. 352-359, 1997.
- [11] Albers S., and Hagerup T., Improved parallel integer sorting without concurrent writing, *Information and Comput.*, vol. 136, pp. 25-51, 1997.

- [12] Hagerup T., and Shen H., Improved nonconservative sequential and parallel integer sorting, *Inform. Process. Lett.*, vol. 36, pp. 57-63, 1990.
- [13] Han Y., and Shen X., Conservative algorithms for parallel and sequential integer sorting, in "Proc. 1995 International Computing and Combinatorics Conference, August 1995," *Lecture Notes in Computer Science*, vol. 959, pp. 324-333, 1995.
- [14] Bhatt P. C. P., Diks K., Hagerup T., Prasad V. C., Radzik T., and Saxena S., Improved deterministic parallel integer sorting, *Inform. and Comput.*, vol. 94, pp. 29-47, 1991.
- [15] Hagerup T., Towards optimal parallel bucket sorting, *Inform. and Comput.*, vol. 75, pp. 39-51, 1987.
- [16] Han Y., and Shen X., Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs, in "Proc. 1999 Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), Baltimore, Maryland, January 1999," pp. 419-428, 1999.
- [17] Kruskal C. P., Rudolph L., and Snir M., A complexity theory of efficient parallel algorithms, *Theoret. Comput. Sci.*, vol. 71, pp. 95-132, 1990.
- [18] Leighton F. T., "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes," Morgan Kaufmann, San Mateo, CA, 1992.
- [19] Rajasekaran S., and Reif J., Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J. Comput.*, vol. 18, pp. 594-607, 1989.
- [20] Rajasekaran S., and Sen S., On Parallel integer sorting, *Acta Inform.*, vol. 29, pp. 1-15, 1992.
- [21] Vaidyanathan R., Hartmann C. R. P., and Varshney P. K., Towards optimal parallel radix sorting, in "Proc. 7th International Parallel Processing Symposium," pp. 193-197, 1993.
- [22] Wagner R. A., and Han Y., Parallel algorithms for bucket sorting and the data dependent prefix problem, in "Proc. 1986 International Conf. on Parallel Processing," pp. 924-930, 1986.
- [23] Dessmark A., and Lingas A., Improved Bounds for Integer Sorting in the EREW PRAM Model, *J. Parallel Distrib. Comput.*, vol. 48, pp. 64-70, 1998.
- [24] Kirkpatrick D., and Reisch S., Upper bounds for sorting integers on random access machines, *Theoret. Comput. Sci.*, vol. 28, pp. 263-276, 1984.

- [25] Raman R., Priority queues: small, monotone and trans-dichotomous, in “Proc. 1996 European Symp. on Algorithms,” Lecture Notes in Computer Science, vol. 1136, pp. 121-137, Springer-Verlag, Berlin/New York, 1996.
- [26] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory, vol. 10, pp. 99-127, 1977.
- [27] Y. Han, M. Thorup, Sorting integers in $O(n \sqrt{\log \log n})$ expected time and linear space, IEEE Symposium on Foundations of Computer Science (FOCS’02), pp. 135–144, 2002.
- [28] Bender M. A., Farach-Colton M., and Mosteiro M. A., Insertion sort is $O(n \log n)$. Theory Comput Syst vol. 39 no. 3, pp. 391–397, 2006.
- [29] Dietzfelbinger M., Hagerup T., Katajainen J., and Penttonen M., A reliable randomized algorithm for the closest-pair problem, J. Algorithms, vol. 25, pp. 19-51, 1997.
- [30] Cole R., An optimally efficient selection algorithm, Inform. Process. Lett., vol. 26, pp. 295-299, 1987/88.
- [31] Miltersen P. B., Lower bounds for union-split-find related problems on random access machines, in “Proc. 26th STOC,” pp. 625-634, 1994.

List of Publications

- [1] A. Singh, and D. Garg, Optimizing Integer Sorting in $O(n \log \log n)$ Expected Time in Linear Space, International Journal of Information and Computing Technology, ISTAR, vol. 2, no. 1, pp. 40-44, 2011.
- [2] A. Singh, and D. Garg, Exponential tree sorting: Implementation and performance analysis (Communicated).