Improving Exponential Tree Integer Sorting Algorithm Using Node Growth

Bachelor of Information System
in
Engineering and Natural Science

Submitted by
Sugirbay Margulan 170103053
Nauryzbayeva Anel 170103056

Abstract

The integer sorting algorithm gives the lower bound of the expected time $O\ (n \log n)$. We present exponential search trees as a new method for transforming search structures of a static polynomial space for ordered sets into fully dynamic linear space data structures. But in order to realize our idea of achieving the complexity of the integer sorting algorithm, we must make changes to the exponential tree. In the current implementation, integers will be transmitted along the exponential tree one at a time, but limit the comparison required at each level. The total number of comparisons for any integer will be $\mathrm{O}\ (n \log\log n \log\log n)$, the total time taken to insert all integers will be $O\ (n \log\log n \log\log n)$. The algorithm presented in this report can be compared with the results of traditional techniques on times in linear space. It can also compare with the result of Raman [2], which sorts n integers into $O\sqrt{n \log n \log\log n}\ \ n$ by time in linear space, as well as with the result of Andersson's time-binding $O\left(n\sqrt{\log n \log\log n}\right)$. The algorithm can also be compared with the result of Ijey Khan's [5] expected determine time $\mathrm{O}\ (n \log\log n \log\log n\ n)$ for a deterministic linear spatial integer sort.

In this report, we discussed how to implement exponential sorting of trees, and then compared the results with traditional sorting techniques.

Before presenting our worst-case exponential search trees, we present here a simpler cushioned version, that converts static data structures into fully dynamic cushioned search structures. The basic definitions and concepts of a cushioned structure will be adopted for a more technical design in the worst case. Since this version of exponential search trees is much easier to describe than the worst-case version, it should be of great importance to the interested reader, because, we hope, it will provide a good understanding of the main ideas.

To sort integers in the [0, m - 1] range $O\ (m\varepsilon)$, space is used in many algorithms. When m is large, the used space is excessive. Thus, integer sorting using linear space is more important and therefore is widely studied by researchers. Fredman and Willard showed that n numbers can be sorted by time in linear space [1]. Raman showed that sorting can be done in $n\ \ O\left(n\sqrt{\log n \log\log n}\right)$ time in linear space [2]. Andersson improved time later attached to $n\ n\ \log n$ [3]. Thorup then improved the time tied to $O\left(\ n\ (\log\log n)^2\right)$ [4]. Ijey Khan also proved the same result [5]. Later Ijey Khan showed $O\ (n\ log\ log\ n\ log\ log\ n)$ time for a determinate linear spatial integer sorting [6]. Ijei Khan again showed an improved result with $O\ (n\ log\ log\ n)$ time and linear space [7].

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

1.1. Exponential Tree

The exponential search tree is a leaf-oriented, multi-threaded search tree, where the degrees of the nodes are reduced by half exponentially down the tree. By leaf orientation, we mean that all keys are stored in tree leaves. Moreover, for each node we store a separator for navigation: if the key reaches the node, a search locally among the separators of the child nodes determines which child it belongs to. Thus, if a child of v has a delimiter s and its descendant has a delimiter s ′, the key y belongs to v if

$y \in [s, s')$. We require that the delimiter of the inner node be equal to the delimiter of its leftmost descendant.

We also maintain a doubly linked list of stored keys by providing successor and predecessor pointers, as well as maximum and minimum values. A search in the exponential search tree can lead us to the successor of the desired key, but if the key found is too large, we simply return its predecessor.

In our exponential search trees, a local search in each internal node is performed using a static local search structure called an S-structure. We assume that the S-structure by keys d can be constructed in time $O\left(d^{k-1}\right)$ and that it supports the search in time S (d). We define an exponential search tree by n keys recursively:

• The root has degree $\Theta\left(n^{1/k}\right)$.

• Separators of root children are stored in a local S-structure with

properties specified above.

- Subtrees are trees of exponential search by keys $\Theta\left(n^{1-1/k}\right)$.

It immediately follows that searches are supported over time.

$$\begin{aligned} T(n) &= O\left(S\left(O(n^{1/k})\right)\right) + T\left(O(n^{1-1/k})\right) \\ &= O\left(S\left(O(n^{1/k})\right)\right) + O\left(S\left(O(n^{(1-1/k)/k})\right)\right) + T\left(O(n^{(1-1/k)^2})\right) \\ &= O\left(S\left(n\right)\right) + T\left(n^{1-1/k}\right). \end{aligned}$$

One of the definitions of an exponential tree: it is almost identical to the binary search tree, except that the dimension of the tree is not the same at all levels. In a regular binary search tree, each node has a dimension (d) of 1 and has 2d children. In an exponential tree, the dimension is equal to the depth of the node, with the root node having $d = 1$. Thus, the second level can contain two nodes, the third can contain eight nodes, that is, four children of each node at the second level, the fourth can contain 64 nodes, t .e. eight child nodes of each node at the third level, etc. [7].

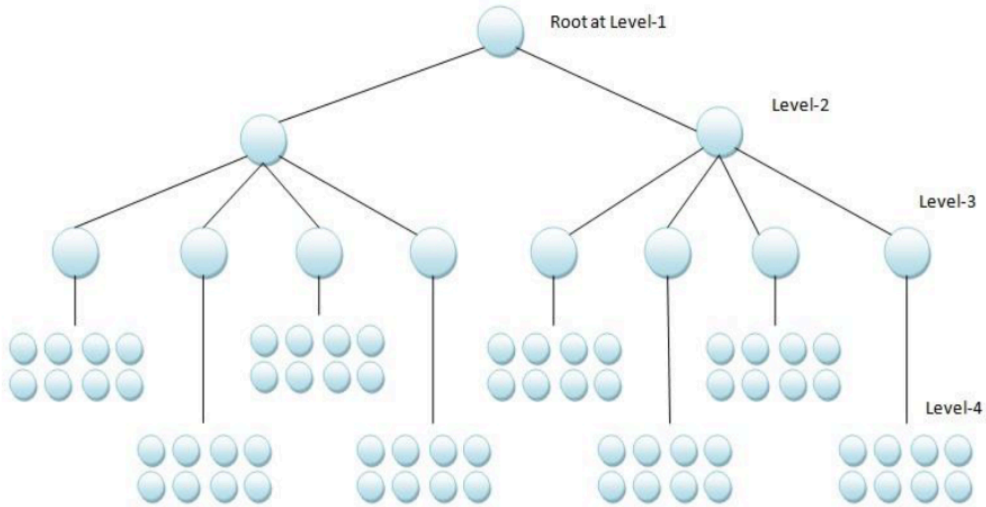


Figure-1.1: Exponential tree

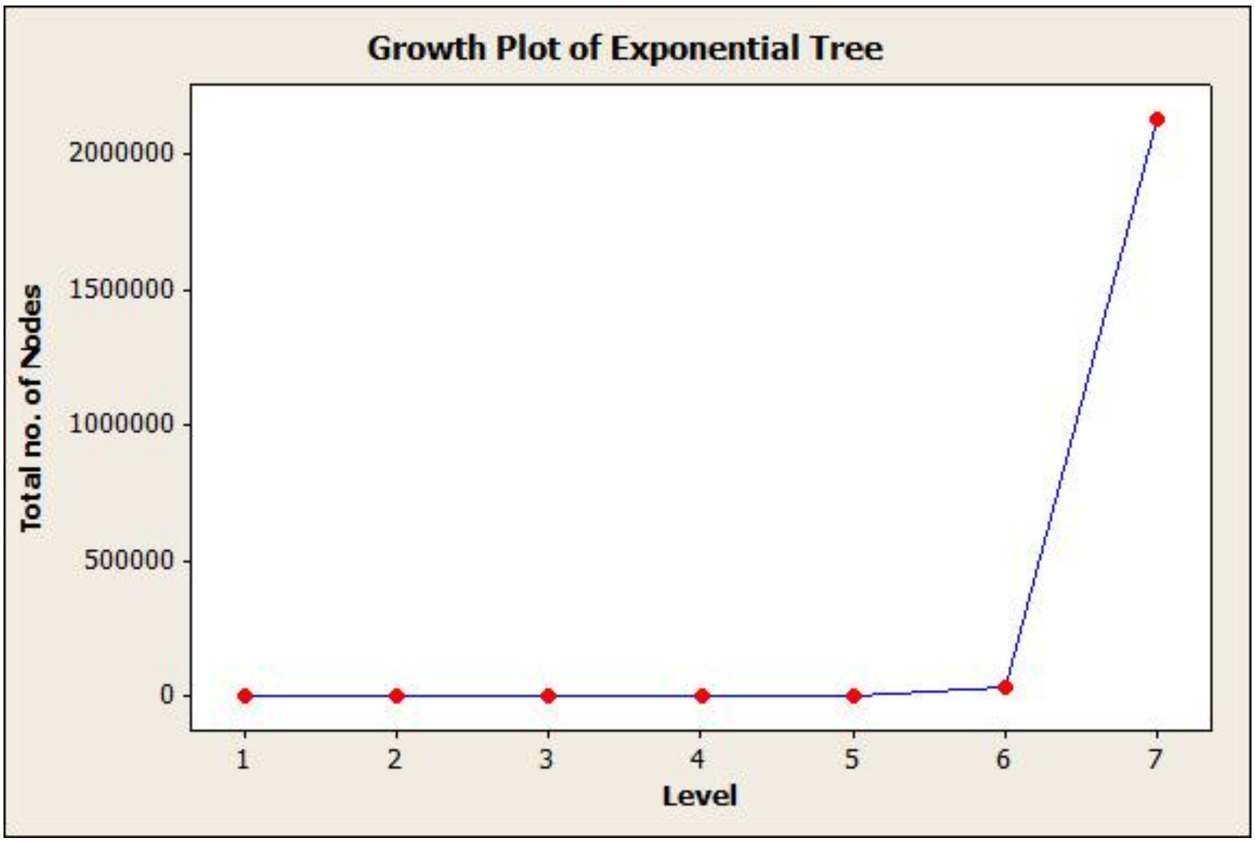
1.2.Growth

The growth of an exponential tree is very important to understand in order to understand the complexity associated with this kind of data structure. As discussed above, with an increasing level of depth, the number of children doubles for each child of the parent each time. This gives exponential tree growth. Table 1.1 shows a relationship showing how the total number of nodes present in a tree increases exponentially with an increasing level of depth.

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

Table-1.1: Number of nodes in exponential tree

This graph clearly shows that the tree has exponential growth. Thus, it becomes practical to control the growth of the tree in practice as the number of levels increases. Because of this exponential growth, it is called an exponential tree. It should be noted here that each node has one key, i.e. an integer. This implies that comparing with the values of the nodes of the child nodes is a very difficult task since a huge number of pointers are associated with each node.



1.2. The 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 operations may include tracing the tree, deletion from the 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.

1.3. Exponential Tree Sort

There are many ideas for integer sorting using an exponential tree. These ideas sort by inserting integers into the tree and then tracing the tree to get the desired sequence, i.e. a sorted sequence. These ideas either pass integers one after another or pass integers in batches. This section discusses two important ideas for exponential tree sorting.

Andersson showed that if we pass integers in order in an exponential tree, then the insertion takes “log” for each integer, that is, the total complexity for “integers” will be equal to “log” [3].

Yijie Khan gave an idea that reduces complexity to the expected time in linear space 7 (*n log log n*) [7]. The technique he uses is the coordinated transmission of integers in the Andersson exponential search tree and linear time division of integer bits. Instead of inserting an integer one at a time into the exponential search tree, it passed all integers one level at a time to the exponential search tree. Such a coordinated passage makes it possible to perform multiple divisions in linear time and, therefore, accelerate the algorithm.

In this section, the solution to the problem discussed in the above section is provided. The solution can be broadly categorized into two parts. The first part is to modify the 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 the sorting algorithm with implementation.

2.1. Modified Exponential Tree

The exponential tree was first introduced by Andersson in his research for a 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 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, the fourth 64 nodes, and so on [7]. This shows that the number of children at each level increased by a multiplicative factor of 2 i.e. exponential increase in the number of children at each level [7].

The tree itself is very difficult to process an integer increasing. It is also necessary to process more pointers at each level of each node. Thus, the exponential tree needs to be changed. A modified exponential tree concept provides a convenient way to integer sort. Instead of focusing on the number of nodes present in the tree, it is beneficial to focus on the number of integers present in the tree, since the task is how to use it for integer sorting.

A tree with the properties of a binary search tree will be called an exponential tree if it has the following properties:

1. Each node at the hold level will contain k the number of keys (or integers in our case), that is, at a depth k the number of keys in any node will k keep the root at level 1.

2. Each node at the having level will have $k + 1$ children, that is, at a depth of k the number of children will be $k + 1$.

3. All keys in any node must be sorted.

4. The integer in the child i must be greater than the key key - 1, and less than the key key.

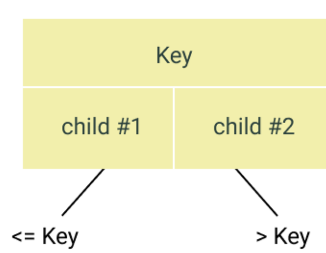
The total number of integers held by the tree to level k will be the addition of the total number of integers to level k - 1 and the integers present at this level. This will be given by the following formula:

$Nk = Nk - 1 + k * k!$ .... (1)

where Nk is the total number of integers up to the level of k. $N1 = 1$, since the root is at level 1 and contains only 1 integer

k if levels level and $k!$ denotes the factorial k.

Thus, the node in the root will look like this:



The height of the tree remains O ($log\ log\ n$), which can be proved by induction. The modification will not only reduce the complexity of the exponential tree involved in its implementation but also improve the balancing method, as well as the sorting method. Integer sorting will be more convenient and faster with this modification.
Implementing an exponential tree requires creating an exponential tree node. Here is the skeleton for the exponential node that is used in the implementation for integer sorting:

```
struct Node{
    int level;

    int count;

    Node **child;

    int data[];

}
```

Here the level will contain the node level number.

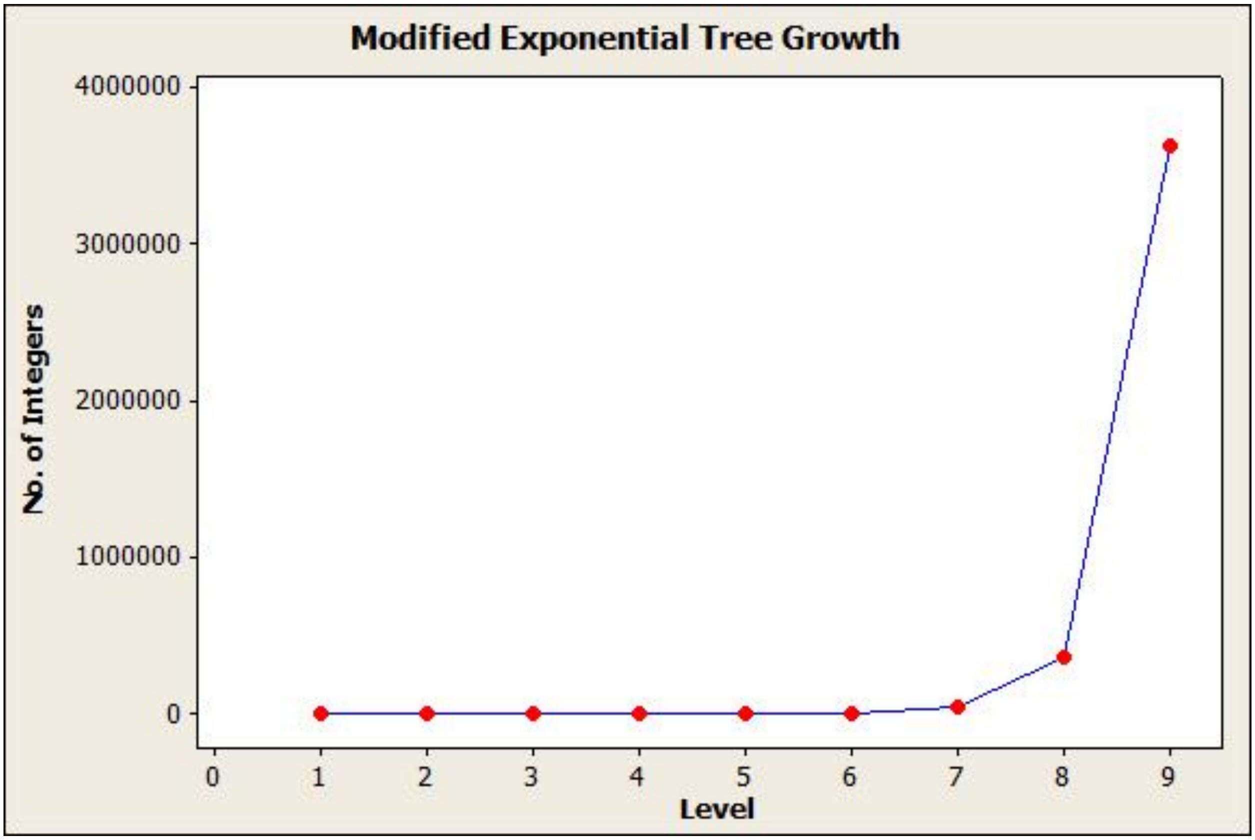
the count will contain the integer number currently present in the node.

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

data is an array of integers to store the integers present in this node.

2.3 Growth

The growth of this modified exponential tree is also exponential. But this redesigned design provides an easy implementation. The challenge is to process as many integers as possible to a certain level.



2.4 Insertion

Andersson proved that insertion can be done at “expected time” by passing an integer to the exponential tree in turn, which doesn’t look very good quite optimized [3]. But in the best case, inserting will require *O* (*n log log* времени) time, since only one comparison is required at each level. The best case is rare, so the best scenario is not reliable. Therefore, to achieve a better result, further modification of the concept is necessary.

As already mentioned, passing integers one at a time does not give enough expected time. Yijie Han came up with the idea of giving out integers in a group [7]. He used the multiple division method to break integers into smaller lists. He suggested that instead of inserting an integer one at a time into the exponential search tree, all integers can be simultaneously transferred to one level of the exponential search tree. Such a coordinated passage makes it possible to perform multiple divisions in linear time and, therefore, accelerate the algorithm. This method gives the expected time $O\ (n\ log\ log\ n)$ in linear space [7]. But it is very difficult to transfer all integers at once if the number of integers is very large. Thus, inserting integers into groups does not seem very good in terms of implementation and the complexity associated with transferring all integers at once.

The algorithm discussed in this thesis will transmit integers one after another in a modified scheme. The algorithm will give complexity $O\ (n\ log\ log\ n\ log\ log\ log\ n)$ by reducing the number of comparisons required at each level.

An insertion method using a modified binary search is described below, and then a 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 *it*h child of parent and insert element in that.

Step-8: Return.

The tree has a depth or height of $O(log\ log\ n)$.

2.5 Binary Search

As discussed in the definition of an exponential tree, all integers represented in the node of the exponential tree must be sorted, so this property can be used to improve

performance. To search for a position in a sorted list, you can use the binary search with minor changes. The binary search takes *n log n* the expected search time. Consequently, performance will be improved by using a binary search instead of linear search.

It should be noted here that a binary search is used only inside the node of the exponential tree. This will give the position in which the element should be inserted. If there is only room for more elements in this node, then this element will be inserted, otherwise, the algorithm moves to the next level of the exponential tree, i.e., the predecessor-descendant of this particular node. A modified binary search looks like this:

BinarySearch(Node *ptr,int element)
Step-1: If element > ptr->data[count-1] then return ptr->count.

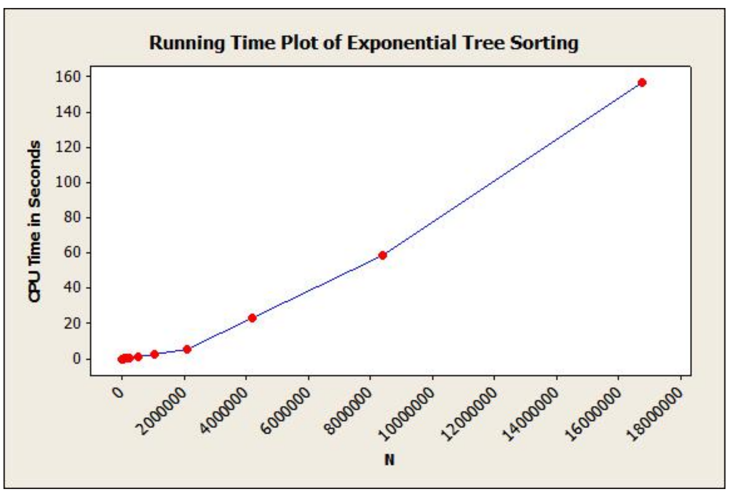
Step-2: Set start=0, end=ptr->count-1, mid= (start + end)/2.

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.

In this section, we will compare the performance of exponential tree sorting with binary tree sorting and quick sort. It is obvious here to think that comparing exponential sorting of trees should only be done using quick sorting, which is the most well-known and widely used sorting technique; but since the quick sort is mainly used for integers stored in a sequential memory cell, i.e. in an array, exponential tree sorting works here in an inconsistent memory cell. Therefore, binary tree sorting is also considered for comparison. The comparison includes processor time and memory requirements.

3.1. Exponential tree sorting time analysis

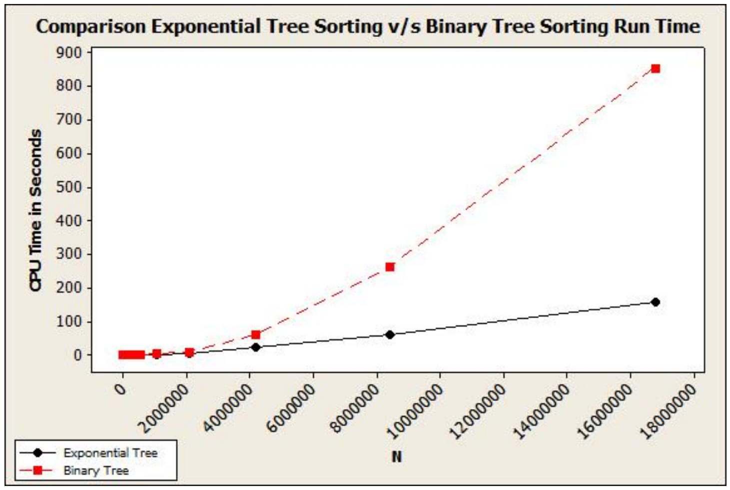
This graph shows the CPU time for exponential tree sorting, which clearly shows that the slope of the graph is linear.



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

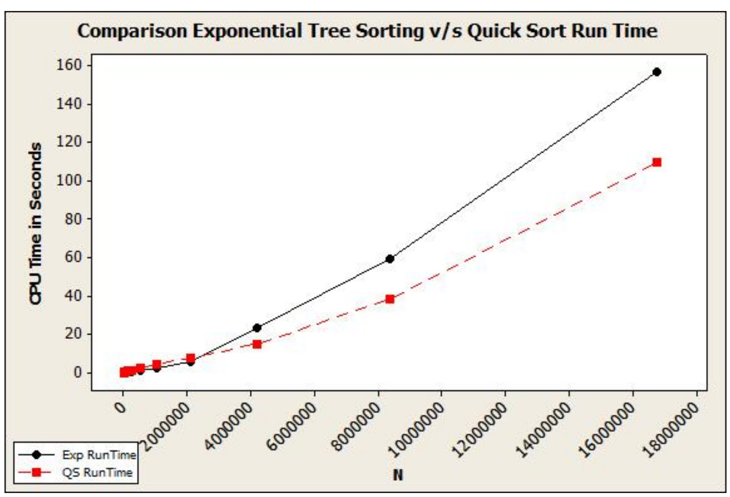
3.2 Exponential tree sorting v/s Binary tree sorting

This graph shows a comparison of the CPU time for exponential tree sorting and binary sorting, which shows that exponential tree sorting requires relatively much less CPU time for the same number of integers than for sorting a binary tree. As the number of integers increases, the CPU time for exponential tree sorting increases with a very small coefficient than binary tree sorting.



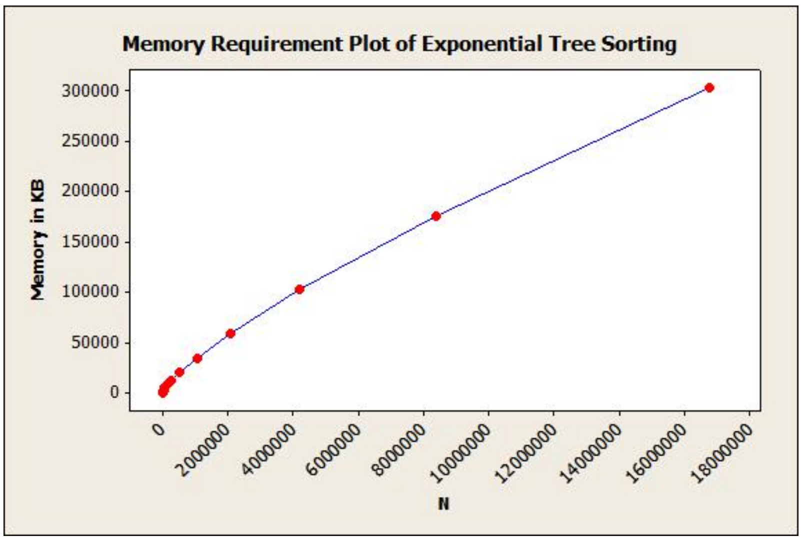
3.3 Exponential tree sorting v / s Quick Sort

This graph shows a graph showing the time of exponential sorting of a tree and the time of quick sorting. The line graph for both algorithms has a linear slope. An exponential tree gives better run time than quick sort for fewer input integers, i.e. N.



3.4 Analysis of Exponential Tree Sorting Space Requirement

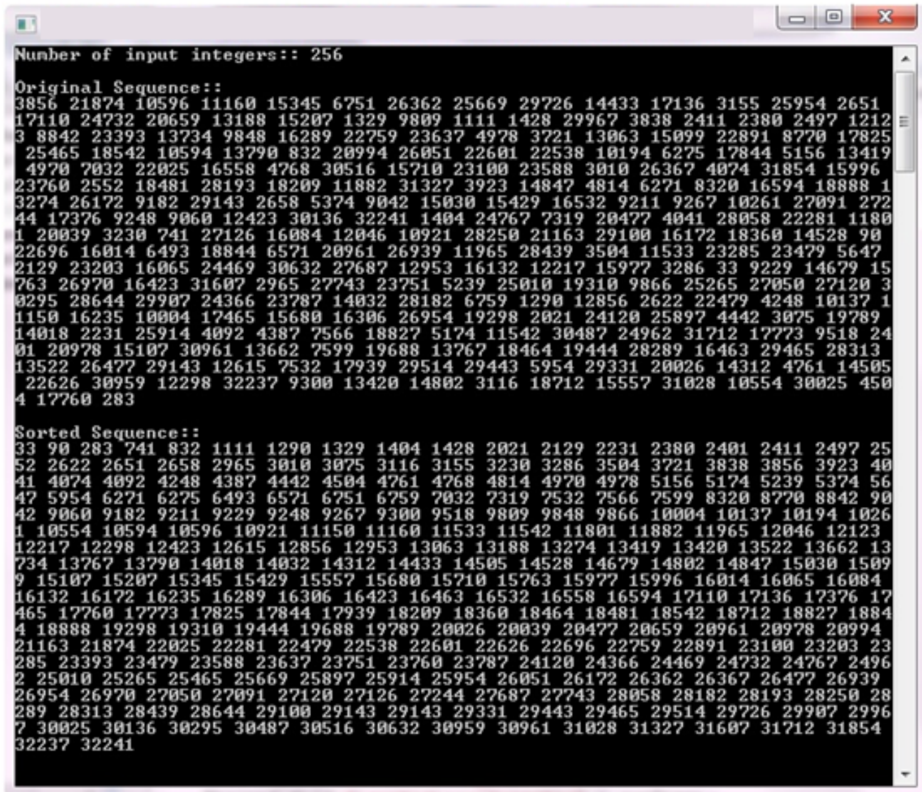
This graph shows a graph of the memory requirements for an exponential tree, which shows that the graph has a linear slope. The memory requirement increases in direct proportion to the number of integers to be sorted. The memory used by the exponential tree includes memory for pointers created for children, a track for the number of integers present in the node, the depth or height of the node, and an array used to store the key values of the node. The memory requirements for the exponential tree are relatively small.



3.5 Results

This terminal output shows a test run of the algorithm. First, the number of input integers from the file is checked. After that, all input integers are scanned one at a time and inserted into the modified exponential tree. This will reduce the need for a list of input integers since integers are directly inserted into the tree. Passing integers one at a time provides this function.

After all, integers are inserted, due to the properties of the modified exponential tree, all integers will be in a sorted sequence. Tracing in tree order will give the desired sorted output. This is also shown in the terminal output. Therefore, the algorithm works successfully and provides integer sorting.



Conclusion

This report presented another idea for achieving the complexity of the deterministic integer sorting algorithm in O ($n\ log\ log\ n$ log log n) expected time and linear space with a conservative advantage. This algorithm is simple to implement and simple. To achieve this complexity, the data structure used is an exponential tree that has been modified to simplify design and implementation. The idea was inherited from the Andersson exponential tree.

The exponential tree presented in this thesis has a simple layout. The modified design is not only easy to understand, but also easy to implement. The main emphasis in the modified design is to process as many integers as possible with a simple layout. The new design has reached the optimized height and complexity associated with the insert. The height of the new data structure is $O\ (log\ log\ n)$, which is very exciting. It is also proven that the design is also optimized for memory usage. This requires less memory than its counterparts.

Integer sorting is performed at the expected time in linear space using an exponential tree. To achieve this expected time, the algorithm used a binary search with a slight modification with an exponential tree. The basic concept of binary search does not change but is used in accordance with the requirements. Instead of looking for the key position, he searches for the successor of the key so that the key can be inserted before it.

Implementation has shown that the algorithm has a very good performance both in terms of runtime and memory requirements. It has a competitive performance with quick runtime sorting and is much better than binary tree sorting. The memory requirements for exponential tree sorting are also very less compared to sorting a binary tree. Needless to say, exponential tree sorting is preferable to binary tree sorting.

- [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.