Indexed Tree Sort: An Approach to Sort Huge Data with Improved Time Complexity

Prateek Agrawal
Department of Computer Science & Engineering
Lovely Professional University
India

Harjeet Kaur
Department of Computer Science & Engineering
Lovely Professional University
India

Gurpreet Singh
Department of Computer Science & Engineering
Lovely Professional University
India

ABSTRACT
Sorting has been found to be an integral part in many computer based systems and applications. Efficiency of sorting algorithms is a big issue to be considered. This paper presents the efficient use of Indexing with Binary Search Trees (BST) to model a new improved sorting technique, Indexed Tree (IT)-Sort, capable of working with huge data. Along with design and implementation details, major emphasis has been placed on complexity, to prove the effectiveness of new algorithm. Complexity comparison of IT-Sort with other available sorting algorithm has also been carried out to ascertain its competence in worst case also. In this paper, we describe the formatting guidelines for IJCA Journal Submission.

Keywords
Computing, Sorting Algorithm, Complexity, Huge Data Set, Binary Search Tree (BST), Indexing

1. INTRODUCTION
There are number of traditional algorithms used to find ordering of unordered data sets. Each algorithm has its own pros and cons and a specific methodology to arrange the data like merging divide and conquer, partitioning, recursive methods etc [1, 2]. Different sorting algorithms are analyzed and compared according to their complexity [3, 6, and 7]. The analysis of algorithms is the area of computer science that provides tools for contrasting the efficiency of different methods of solution. Although the efficient use of both time and space is important, inexpensive memory has reduced the significance of space efficiency [4]. Thus, focus of researcher has been restricted to primarily on time efficiency only. Time complexity of an algorithm is a function of the size of the input to the problem and quantifies the amount of time taken by an algorithm to execute. Designing of suitable sorting algorithm as per application is a continuous process. Lots of work is being carried out in this field with single objective to reduce time complexity of proposed algorithm. Existences of large number of data values have significant impact on computational complexity of sorting. Since, sorting large datasets may slowdown the overall execution, schemes to speedup sorting operations are needed [8].

Sorting algorithms are classified according to computational complexity, number of swaps, stability, memory requirements, recursive nature, number of comparisons etc. Most of the times aalgorithms are analysed for best, worst and average cases according to size of input data. In most of the cases, all efforts are laid on improving the average case complexity. Present work is related, yet different from existing works on efficient practical algorithms for sorting. Proposed algorithm concentrate on reducing time complexity to a great extend if sorting is carried out with huge data sets even in worst case. In this paper, an attempt has been made to present improved approach for finding more efficient solution, requiring less execution time, of sorting using Indexing and BSTs.

The rest of the paper is organized as follows. Next section describes the methodology followed in designing IT-Sort. Main emphasis in section 3 has been placed on presenting the design and implementation details of new algorithm. Section 4 supports the whole discussion with experimental results to prove the effectiveness of proposed algorithm and finally section 5 concludes the paper with future enhancements.

2. IT-SORT
2.1 Enhanced Algorithm for Sorting
Sort information is inherent in many applications, making sorting a fundamental problem in the study of algorithms. Efficient sorting is important to optimize the use of algorithms requiring sorted lists to work correctly. In proposed work a list of indexes is created. Each index value further points to a BST. The idea is to place all values starting from one particular digit to BST of the index whose value is equal to that digit. Elements of the unordered list are arranged in BST of corresponding starting digit. In simple words complete data is organized as a list of BSTs. Elements are arranged in such a manner that indexes are pointed directly instead of traversing of list. Every element in the list is placed in such a position which best suits for sorting. Element is added to its corresponding index position and in order to locate these indexes, direct pointers are used. Mechanism to directly calculate the address of appropriate position of elements has been implemented. Direct pointer straight forwardly cuts off the time to traverse the index list. This makes both searching and sorting operations on large data set, fast. It's only the arrangement of elements in a way that when we start reading the elements the read value comes out to be sorted. Elements are put into appropriate position after all input values and sorting process is just traversal of proposed organization. The pseudo code given below has two different procedures, one is to read the different values with appropriate organization and another is to give the sorted list.

2.2 Data Structure Used
The computation performed by the proposed algorithm requires dynamic manipulations of data, such as index and BST. The data can be of any length. Particularly, IT-Sort is designed to meet the requirements of applications working on large data set. It is therefore necessary that values are stored in flexible and dynamic data structures. In order to avoid the non-determinism of solutions, data is stored in a list of BSTs. List, which is serving as an index for BSTs, is again designed

to be an array of pointers. In order to create an index of BSTs, two special kinds of structures are created. First one is to access the indexed items and another to represent the node in a BST. As shown in Figure 1(a and b), we use a pointer-based linear list structure to implement Indexed BSTs. Each node of the list contains a link to one BST with index value equal to the node number and a reference down pointer to point the next node of the list.

An array of pointers to BSTs, *index_bst* has been used to represent an element in Index List. Pointer arrays offer a particularly convenient method for representing these index values. An *n* element array can point to *n* different BSTs. Each individual BST can be accessed by referring to its corresponding pointer. An advantage to this scheme is that a fixed block of memory needs not to be reserved in advance, as is done when initializing a conventional array. Moreover, size of the index list varies according to values in the list to be sorted. Maximum value from the unordered list is taken to determine the size of the list. If *max_value* represents the biggest value in unsorted list, then size is calculated as per following formula

INDEX_SIZE = sqrt(MAX_VALUE)

Memory is dynamically allocated to index list according to calculated *array_size* as

INDEX_BST = (int *) malloc (INDEX_SIZE * sizeof (int))

Since an array name is actually a pointer to the first element within the array, it is more convenient to define array as a pointer variable. In order to calculate the address of any element, one must specify only the array name and number of elements beyond the first. An important advantage of dynamic memory allocation is the ability to reserve as much memory as may be required during program execution and then release this memory when it is no longer needed.

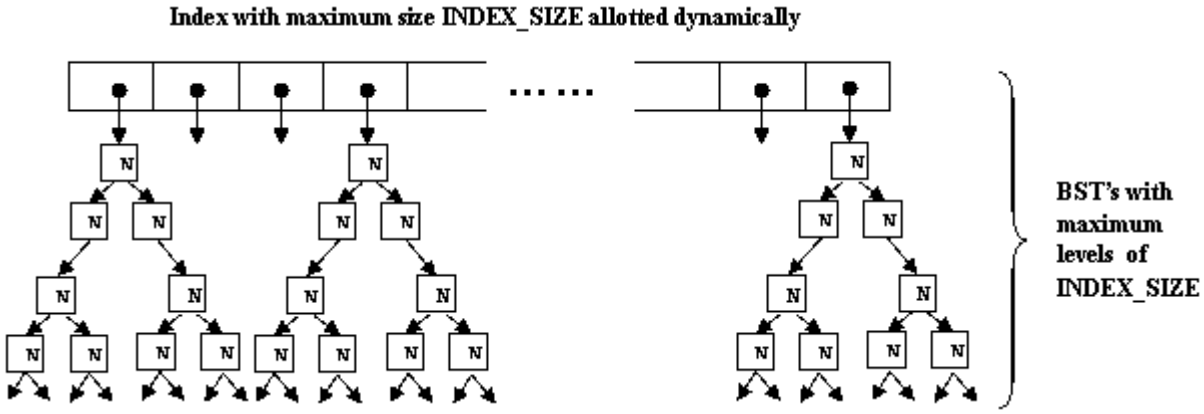

Figure 1(a): Pointer based dynamic linear list structure for Indexed BST

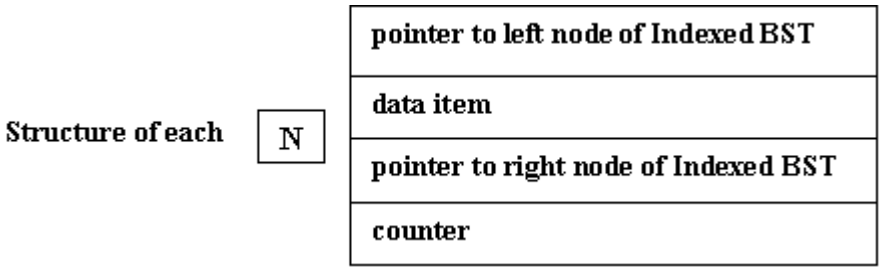

Figure 1 (b): Structure of each Node N in Indexed Tree

Apart from array_bst, one another significant structure is BST_Node defined as

```
typedef struct BST_Node
{
BST_Node*l_node;
int data;
BST_ node*r_node;
}
```

Each node of binary search tree will contain l_node to point to left BST, r_node to point to right BST and an integer data to hold value. In order to deal with duplicate values in data set, a separate counter has been associated with each data item. The major advantage of using binary search trees over other data structures is that the related traversal is very efficient. An in order tree walk will produce arranged data in sorted order.

3. DESIGN OF PROPOSED TECHNIQUE

3.1 Efficient Solution

The algorithm for IT Sort is presented below in Figure 2. The complete algorithm has been divided into two different modules. Before the application of IT Sort, data items are required to be placed in appropriate locations. Second module CREATE_INDEX() is responsible for making the indexed tree of data items. This function creates indexed tree of a list LIST with N items and returns starting address of the index INDEX_BST and size of the index INDEX_SIZE. INDEX_BST is a pointer indicating the start of the index with INDEX_SIZE entries. Value of N is supposed to be very large. Function FIND_MAX() calculates the largest value in the LIST, MAX_VALUE and later on this MAX_VALUE is utilized in calculating the size of the index INDEX_SIZE. Once the size is known, function malloc() dynamically allocates INDEX_SIZE blocks of memory for index in IT Sort. The starting address of the allocation is assigned to INDEX_BST. Step 4 in the algorithm arranges the data items in LIST in a binary search tree at corresponding index entry. LIST is read till the end, for every read data item LIST_ITEM, CALCULATE_INDEX_POINTER() is called to find its relevant index pointer. The address returned by this function, ROOT serve as root address of its corresponding BST. INSERT_BST() adds the LIST_ITEM in a BST at ROOT after finding its appropriate placement in BST.

Function IT_SORT() includes two basic steps of creating indexed tree and in order reading of BSTs. In first step function CREATE_INDEX is called which returns the beginning of the indexes INDEX_BST and number of indexes to be processed INDEX_SIZE. For every index starting from INDEX_BST to INDEX_SIZE, in order traversal of binary search tree INORDER() is made. A simple iterative loop has been applied to in order traverse the different binary search trees in sequence. After the execution of this module the printed items are found to be in sorted order. The psuedocode for the complete procedure is presented below. The complexity details are given in next section to prove the enhanced efficiency and effectiveness of the algorithm.

IT_SORT (LIST , N)

// This algorithm sorts LIST with N items.

Step 1: CREATE_INDEX (LIST , N , INDEX_BST , INDEX_SIZE)

Step 2: Repeat for I = 1 to INDEX_SIZE

- **a) Print INORDER (INDEX_BST)**
- **b) INDEX_BST = INDEX_BST + 1**

Step 3: Exit

CREATE_INDEX (LIST , N , INDEX_BST , INDEX_SIZE)

// This algorithm creates index tree of LIST with N items and returns the starting address of index INDEX_BST and size of index INDEX_SIZE

Step 1 : MAX_VALUE = FIND_MAX (LIST)

Step 2 : INDEX_SIZE = sqrt (MAX_VALUE)

Step 3 : INDEX_BST = malloc (INDEX_SIZE)

Step 4 : While Not (End of LIST) Repeat

- **(a) Read LIST_ITEM**
- **(b) ROOT = CALCULATE_INDEX_POINTER (LIST _ITEM)**
- **(c) INSERT_BST (LIST_ITEM , ROOT)**

Step 5 : Exit

Figure 2: Psuedocode for IT_SORT

3.2 Implementation Details

The implementations and executions during testing was carried out on Intel (R) Core(TM) i3 processor with M350 @ 2.27 GHz 2.27 GHz and 3 GB RAM on Windows 7 Ultimate Service Pack 1, 64 bit Operating System. Compiler used is Bloodshed Dev C++, Version-4.9.9.2. The choice of this compiler was because of the facts this being freeware, compatibility with Windows 7 and its capability to handle large volume of data. In order to generate a list of large set of numbers, worst case data arrangements have been used. List is produced in descending order. One of the main strengths of C is that it combines universality and portability across various computer architectures while retaining most of the control of the hardware provided by assembly language. One powerful reason is memory allocation. Unlike most computer languages, C allows the programmer to write directly to memory. Key constructs in C such as structs, pointers and arrays are designed to structure, and manipulate memory in an efficient, machine-independent fashion. In particular, C gives control over the memory layout of data structures. Moreover dynamic memory allocation is under the control of the programme. Whenever it comes to performance (speed of execution), C is unbeatable [5]. C provides you access to the basic elements of the computer. It gives its users direct access to memory through pointers. It is easy to manipulate and play with bits and bytes.

4. TESTS AND RESULTS

4.1 Examples

Figure and table on next page demonstrate a complete example solved on a small item set of 25. Unordered set of these 25 values include one, two and three digit number. The problem can be extended to *n* numbers with maximum digit width as per the capacity of processing environment. Table I gives the complete representation of data items and their corresponding calculated index values as per formulas described. Same data and index values have been represented in the form of complete Indexed Tree in next picture, Figure 3.

4.2 Complexity Analysis

Let T(n) be the time to execute IT_Sort on an array of size n. Examination of present algorithm leads to the following formulation for run time.

$$T_{IT_{Sort}}(n) = T_{max}(n) + T_{build_{index}}(n) + \sum_{k=1}^{m} T_{index_{read}}(k)$$

Eq.… 1

where *n* is the number of items and m is the index size such that

$$m = \lceil \sqrt{n} \rceil \quad or \quad m^2 \geq n$$

Eq. … 1(a)

$T_{max}(n)$ and $T_{build\_index}(n)$ refers to the execution time taken to calculate maximum element of the list and to create the indexed tree of each element of the list. Since approach used here divides the list into smaller *k* parts with each part to be read separately, time $T_{index\_read}(k)$ is calculated by summing up the individual time taken by *m* different indexes. In isolation these three time complexity functions for both average and worst case can be summarized as calculated in following equations Eq 2 – Eq 5.

A simple linear algorithm to find the maximum element of the list yields time complexity of *n* in both average and worst case so complexity remains in order of *n* only

$$T_{max}(n) = n \quad , \quad O(n)$$

Eq. … 2

In average case time complexity for building an indexed tree of size *m* comes out to be m log(m) while for worst case it is m^2.

$$T_{build\_index}(n) = m \ln m$$

Eq. …3(a)

$$T_{build\_index}(n) = m^2$$

Eq. …3(b)

Last stage of the complexity calculation deals with the simple accessing of the indexed tree. For this, the results vary with average and worst case. Since present work is concentrated on superior time complexity in worst case, discussion will continue on this track only. As per Eq. 1(a), in maximum cases it comes out to be far less than *n* which is the only key to successful implementation of the algorithm in worst case also. In initial analysis *m* appears to be very large and deceives with higher complexity issues but even the larger value of *m* is leading to fewer *k*'s in Eq 1. Many of the *k* values are unutilized and not required to be read as per the programming techniques used. Many of the null indexes will remain untouched in actual sort procedure. A demonstration for this has been given in the example considered in previous part (Figure 3). In order to substantiate the fact, Table II illustrates the difference between the *n* and m^2 as number of null pointers. This huge difference in these random values can portrait any maximum *n* and corresponding *m* value.

There will be one m_i associated with each $T_{index\_read}(k)$ for *k* varying from 1 to *m* whose value can be any integer greater than or equal to 0 which will be deducted every time $T_{index\_read}(k)$ is calculated. Above revealed Tabular results are for worst case only where all maximum 100000 values are there in the list in decreasing order. Number of null pointers will increase drastically with average cases where data set will have a normal range of random numbers. This time extraction λ is responsible for decreasing the complexity in worst case with very huge data set and large values.

Formally, λ can be defined as total number of null pointers available in the entire indexed tree.

$$\lambda = \sum_{i=1}^{k} m_i$$

Eq …. 4

Using Eq 3(a), and Eq. 3(b), for both worst case and average case

$$T_{index_{read}}(n) = m^2 - \lambda = n$$

Eq. … 5

Complexity comes out to be O(n)

Putting the results of Eq 2, 3(b) and 5 for worst case together in Eq. 1, Overall Time Complexity:

$$T(n) = O(n) + O(m^2) + O(n)$$

Or $$T(n) = O(m^2)$$

Similarly, combining the results of Eq 2, 3(a) and 5 for average case together in Eq. 1,

$$T(n) = O(n) + O(m \ln m) + O(n)$$

Or $$T(n) = O(m \ln m)$$

Table III summarizes the actual time taken by few popular algorithms along with IT_SORT executed individually on the same environment as specified in previous section.

Table I: Index value of randomly selected 25 data items

Item N	Value	Index
1	30	2
2	122	11
3	96	8
4	64	5
5	41	3
6	43	3
7	121	11
8	32	2
9	11	0
10	5	0
11	61	5
12	40	3
13	6	0
14	38	3
15	102	8
16	45	3
17	10	0
18	98	8
19	63	5
20	35	2
21	8	0
22	42	3
23	1	0
24	4	0
25	31	2

MAX_VALUE = 121 INDEX_SIZE = int (SQRT(121)) = 12

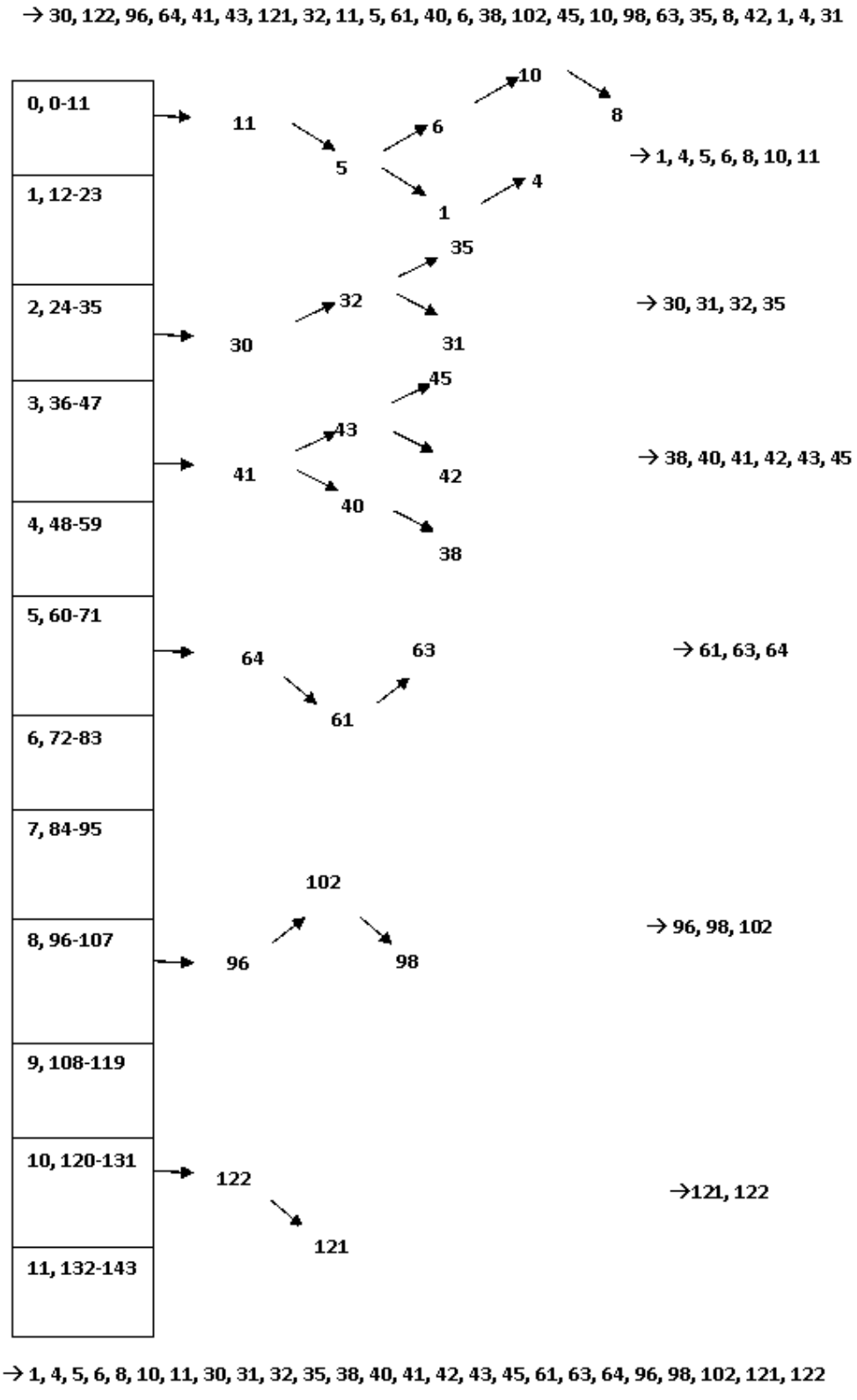

Figure 3: Demonstration of IT Sort

Table II: No of k values not to be read for few large n items (all worst cases)

S.No	No. of Items (n)	m	Null pointers (m_i)
1	10000	100	0
2	20000	142	164
3	30000	174	275
4	40000	200	0
5	50000	224	447
6	60000	245	25
7	70000	265	224
8	80000	283	89
9	90000	300	0
10	100000	317	489

Table III: Execution Time of popular sorting algorithm

	Execution Time (in sec)				
Data Set in thousands → / Sorting Algorithm ↓	10	20	30	40	50
Bubble	1	3	8	16	23
Insertion	0	2	6	10	16
Selection	1	4	8	14	24
Quick	0	0	2	4	6
IT Sort	0	1	2	4	6

5. CONCLUSION AND FUTURE WORK

The above discussion and experimental results contribute in making a conclusion that IT Sort perform much better in comparisons to most of the widely used popular sorting algorithms. Time complexity can be reduced with grouping and arranging data in separate Indexed Trees. The same experiment can be extended to parallel environment with multi processor architectures. All different Indexed Trees can be created on separate processor and manipulations can be done simultaneously in each processor. It will further reduce execution time. Our next paper in this series would be exploring and analyzing the parallel implementation of Extended IT-Sort on multi core system.

6. REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D.Ullman. The Design and Analysis of Computer Algorithms. Addison Wesley, 1974.

[2] Cormen T., Leiserson C., Rivest R. and Stein C., Introduction to Algorithms, McGraw Hill, 2001.

[3] Liu C. L., Analysis of sorting algorithms, Proceedings of Switching and Automata Theory, 12th Annual Symposium, 1971, East Lansing, MI, USA, pp 207-215

[4] Zulkarnain Md. Ali, Reduce Computation Steps Can Increase the Efficiency of Computation Algorithm, Journal of Computer Science 6, 1203-1207, 2010

[5] Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language

[6] Box R. and Lacey S., A Fast Easy Sort, Computer Journal of Byte Magazine, vol. 16,no. 4, pp. 315-315, 1991.

[7] G. Franceschini and V. Geffert, An In-place Sorting with O (n logn) comparisons and O(n) moves, In Proc. 44th Annual IEEE Symposium on Foundations of Computer Science, pages 242-250, 2003.

[8] Hari Krishna Gurram & Gera Jaideep, Index Sort, International Journal of Experimental Algorithms (IJEA), Volume 2: Issue (2) : pp:55-62, 2011