




N-Dimensional Structure: A Data Structure with Fast Access and Sorting of Integers

Harshit Gupta¹, Shruti Gupta¹ (✉) , and Akash Punhani²

¹ ABES Engineering College, Ghaziabad, India
shrutigupta.it@gmail.com

² SRM Institute of Science and Technology, Modinagar, Ghaziabad, U.P., India

Abstract. Various Data Structures have been introduced till date that aim at either to improve them in terms of time or space complexity or they are particularly designed for some special applications. This work aims at developing a new data structure that takes the idea of plotting each digit to another dimension and thereby reducing the access time for keys stored in it. In order to make it practical to implement, a tree-based approach has been used to store it in the memory. The asymptotic time complexity is $O(nu)$ for extracting the keys in sorted order where u is the size of universe. In practical testing, the procedure took much less time to complete as compared to Merge Sort which is the best Comparison based Sorting Technique in terms of worst-case time complexity. This work discusses the concept, implementation and algorithms associated with the data structure. Then it evaluates it against various test scenarios with other sorting techniques.

Keywords: Data structure · Algorithm · Sorting · Efficiency · Fast access · Time complexity · Optimization

1 Introduction

Data structures have been in use since the beginning of the era of computers. There have been various advancements in the field since then which continue till date. The aim is either to improve them in terms of time or space complexity or they are particularly designed for some special applications with pre implemented methods that makes it easy to use and prevents the user from getting deep inside the complexities of implementation of methods. Unlike most algorithmic issues, “linear time” is too slow; we can’t afford to scan the complete data structure to answer a single query—and the goal is usually logarithmic or even constant query time. Also, because the data corpus is usually huge, space becomes a pressing concern, and you don’t generally want a data structure that is larger than the data. However, these days, storage is becoming cheaper on the other hand we want to answer queries at a fast rate.

This work proposes a new data structure which can be used to sort numbers or retrieve sorted list of numbers from it in less or at least similar asymptotic time as compared to pre-existing sorting techniques including algorithms and data structures with pre implemented extract procedures and search time which is independent of number of

elements present in the data structure in terms of time complexity which is definitely much less than “linear time” in practical implementation.

This paper is organized into the following manner:

First, it discusses the concept and idea behind this data structure and plotting of numbers in a space and the basic idea to store it in memory using a direct intuitive array based implementation of the data structure.

We propose the tree base implementation of the data structure along with all the methods associated with it for insertion, deletion and search of elements, and then finally the extract sorted method for sorting numbers using the data structure.

Then, we evaluate our methods against benchmarks. Experimental Results show that proposed structure outperforms many pre-existing algorithms for the purpose of searching.

2 Literature Review

Tries were proposed in 1960 [1] that are used for storing alphabetical words in a corpus with an efficient search complexity. Bitwise Tries have also been used since a long time. Individual bits are utilized to traverse what basically becomes a sort of binary tree in bitwise attempts, which are similar to character-based trie. Although this procedure appears to be slow, it is cache-local and extremely parallelizable due to the lack of register dependencies, and as a result, it performs quite well on recent out-of-order execution CPUs. In [2], Huang et al. proposed a multi-block bitwise trie structure for exact r-neighbour search in hamming space. Bucket sort is another sorting technique being used for optimal sorting but it is not suitable everywhere. Bucket sort divides the data pieces to be sorted into buckets, which are subsequently sorted individually using any other sorting approach or by recursive application of the bucket sort technique itself. The difficulty of a bucket sort is determined by the number of buckets utilised, the technique used to sort each bucket, and the uniformity of the data items’ distribution. A stable sorting technique was introduced in [3] which a novel technique for sorting large scale data. Its primary aim was to check sorted big numbers however it performed better than quick sort and similar sorting techniques for sorted numbers. Apart from Merge Sort itself, tim sort was introduced which has been analysed in [4] which improves over merge sort modifying its performance in terms of time complexity in some scenarios.

Keshav Bajpai in [5] proposed an efficient version of counting sort that reduced the sorting time to half from counting sort for an array of 10,000 integers. In 2002, a new randomized sorting algorithm [6] was proposed sorting n integers in $O(n/\text{spl radic}/(\log \log n))$ expected time and linear space. Another algorithm for integer sorting was proposed in 2001, which claimed to sort n integers in linear space in $O(n \log \log n \log \log \log n)$ time. Under certain conditions, it claimed to have a time complexity of $O(n \log \log n)$ [7].

Apart from the various proposed algorithms, only few get used frequently considering there practical limitations and performance. Timsort is a modified version of merge sort that is used internally by Java and Python for there default sorting algorithm. It has the best case complexity of $O(n)$ and a worst case time complexity of $O(\log n)$. Its complexity analysis has been done in this paper [8].

In [9], Kumar et al. proposed a new sorting algorithm called recombination Sort which is derived from the recombination of cardinal principles from a number of different sorting methods. Radix sort's ability to deal with each digit of a number separately, counting sort's concept of counting the number of occurrences of elements, bucketing from bucket sort, and hashing a number to a multidimensional space are all combined to form a single sorting algorithm that outperforms its parent algorithms. For the best, average, and worst situations, the time complexity of the proposed Recombinant Sort was expected to be $O(n + k)$. In the worst-case situation, the k in $O(n + k)$ will become n , but n 's order will never approach two, i.e., k will never approach n^2 .

Moving on from sorting algorithms and looking at some data structures, Burst tries [10] were proposed in 2002 which claimed to be fast and efficient data structure for string keys. However, integers can be converted to strings and stored in the same. Self balancing trees like AVL, Red Black Tree, B Trees [11] have been used in database systems for indexing keys which have search complexity in the order of $O(\log n)$ and insert complexity $O(\log n)$. Numbers can be retrieved from the data structure in $O(n)$. So, they have an $O(\log n)$ time complexity for sorting numbers. There detailed evaluation has been done in this article [12]. A workshop proceeding [13] proposed various efficient data structures for storing partitions of n integer, however the best time complexity for them was $O(n^2)$. A dynamic integer set [14] was proposed in 2014 which had $O(n)$ space complexity and $O(\log n)$ time complexity for main set operations like insertion, deletion, predecessor/In 2019, FASTSET [15] was proposed which claimed to have optimal performance for most of the commonly used set operations claiming that previously existing sets had at least one non-optimal operation. The author also, compared it with the inbuilt set of Java collections implementation. In 2021, new data structures were proposed for orthogonal range reporting and range minimum queries in [16]. It proposed a data structure that can be used for 2-D orthogonal range minima queries in $O(n)$ and $O(\log \epsilon n)$ time, where number of points is represented by n in the data structure and ϵ is an arbitrarily small positive constant. A sorting algorithm was also proposed in 2021 in [17] which proposed a parallel multi-deque sorting algorithm. In 2017, A paper introduced an implementation and statistical comparison of different edge detection techniques like Prewitt, Sobel and Robert [18]. It was developed on the top of Multi-Stack Sort enhancing the performance while getting rid of the weakness of MSP Sort. It was primarily for multi-core CPUs and GPUs. In 2021, an energy efficient enhancement for prediction-base scheduling was proposed which helps in the network lifetime improvement in WSNs [19]. In 2019, another efficient sorting algorithm was introduced for non-volatile memory which claimed to outperform many other sorting algorithms in terms of execution time and non-volatile memory writes [20].

There have been various sorting algorithms introduced till date that have various properties associated with themselves namely Bubble Sort, Quick Sort, Heap Sort, Insertion Sort, Merge Sort, etc. Though, this paper doesn't propose exactly a sorting algorithm, but a data structure that can be used to sort numbers. Just like heap sort (analysed in the cited paper) [21] uses heap data structure to extract numbers in a sorted manner.

3 Methodology

3.1 U-Dimensional Space

The idea behind this new data structure is to plot numbers in a u dimensional space which we like to call a Universe. For a u -dimensional universe, we can plot numbers from 0 to 10^{u-1} . We can think of it as a cartesian u dimensional space where each integer can be represented as a point in the space.

For example, if we take a 3-dimensional space, we can plot numbers from 0–99. We reserve one dimension for storing value 0. Which means each number has value zero in the zeroth dimension. We can say 0 is 000, 1 is 001, 2 is 002, 10 is 010, and 99 is 099. We won't be able to plot 100 in this space because we want the zeroth dimension to be zero always. This might not make sense now but will help us in the implementation phase. Let us now take this 3-d space and plot some numbers. We will only look at the non-zero dimensions here, i.e. first and second dimension here. Let us take some numbers. We will then break them down to dimensions and plot them as given in Table 1.

These numbers plotted in a 3D space while only looking at the non zero dimensions looks like the space as shown in Fig. 1.

3.2 Array Based Implementation

If we try to implement this in a machine, we can use a 2-D array to do this, which is very intuitive. However, this approach is just for a better understating of how the tree-based implementation was derived and to get the intuition behind it. It will look like a table given below. In the array, the first dimension will represent the first non-zero dimension and the second similarly will represent the second non-zero dimension. Hence, we can locate each number in the array using these two dimensions.

Table 1. 3-D equivalent for numbers

Serial number	Integer	Equivalent	Point in 3-dimensions
1	4	004	(0, 0, 4)
2	7	007	(0, 0, 7)
3	12	012	(0, 1, 2)
4	34	034	(0, 3, 4)
5	68	068	(0, 6, 8)
6	91	091	(0, 9, 1)

A similar approach can be used to represent any set of numbers and we can generate the array for the same. If we wish to perform various operations in this array say search, insert, delete by key, one can easily do that by accessing the particular dimensions in the increasing order according to the values it should hold. If we want to get a sorted list of all the numbers, we can do so by starting with an empty array, iterating through the dimensions of the u-d array one by one and keep adding to the array whenever we find non zero value in the u-d array.

3.3 Tree Based Implementation

While exploring any non-zero dimension n , the value can be only be an integer in the range of 0 to 9. And if we access next dimension $n + 1$ for any of the value from 0–9, there might be elements only in some of the values for the specific dimension.

For example, in the previous example, there were no elements present for 1st non zero dimension at [2][y], [4][y], [5][y], [7][y], [8][y]. So, we don't need the $n + 1$ and following dimensions where $n = 2, 4, 5, 7, 8$. This fact can be used in saving storage. Though, we will have to change the implementation and use an abstract way to represent the u-d array.

If we treat each dimension as an array of 0–9 of itself and at each of these values, we point them to the next dimension as presented in Tables 3, 4, 5, 6 and 7, then we only need to keep those arrays that have points present in the dimension. Let's say for first dimension we have a 1-d array as given below. At each position in the array we only store one address of the array for the next dimension for the current value of current dimension.

Table 3. Array for first non-zero dimension

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
add0	add1	NULL	add3	NULL	NULL	add6	NULL	NULL	add9

Table 4. Array at add0 for value 1 in second non-zero dimension

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	0	0	0	1	0	0	1	0	0

Table 5. Array at add3 for value 3 in second non-zero dimension

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	0	0	0	1	0	0	0	0	0

Table 6. Array at add6 for value 6 in second non-zero dimension

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	0	0	0	0	0	0	0	1	0

Table 7. Array at add9 for value 9 in second non-zero dimension

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	1	0	0	0	0	0	0	0	0

This way, we didn't have to store empty arrays for 2, 4, 5, 7, 8.

In a similar way, it can be implemented for any number of dimensions. This will ensure memory efficiency and thereby reduce the time complexity of extracting sorted elements from the data structure.

If we look at it, then the structure it represents is sort of a tree, where each tree can have at max 10 children. Because each element of the array points to another array of the next dimension. Each array used in this representation is a node of the tree.

There will be two types of arrays, the ones which are not of the last dimension, and the ones of last dimension. The intermediate dimension nodes will have values of addresses. The terminal or last dimension nodes will store the count of elements present for that number in the particular space or universe.

3.4 Algorithms

In the tree implementation, an object-oriented approach is used. So, in place of directly considering nodes as arrays, objects will be created which will along with the address or count array, will also store the current dimension number and current dimension value. The class diagram for the representation of nodes is given below. Only Universe is the class which is exposed to the user. One can initialize the data structure by initializing the Universe class by selecting the suitable number of dimensions he/she wants. The other two classes are internally initialized by the universe class itself directly or indirectly by Dimension class. The user has access to AddToUniverse(), SearchInUniverse(), DeleteFromUniverse() and ExtractSorted() methods of the Universe class (Fig. 2).

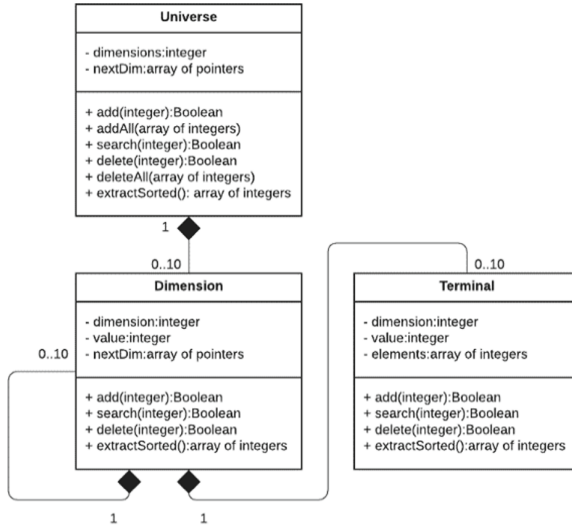


Fig. 2. Class diagram for tree based implementation

Algorithm 1: Add number to Universe

function ADDTOUNIVERSE (**universe**, **num**):

1. $numString \leftarrow string(num)$
 2. $numLen \leftarrow length(numString)$
 3. **if** $numLen > universe.dimensions - 1$ **do**
 4. **return** False
 5. Add to $numString$ in the beginning a string with every character '0' of size $(universe.dimensions - numLen - 1)$
 6. $numValue \leftarrow numString$
 7. $value \leftarrow int(numValue[0])$
 8. **if** $universe.nextDim[value] = NULL$ **do**
 9. $universe.nextDim[value] \leftarrow Dimension(1, value)$
 10. **if** $length(numValue[1:]) = 1$ **do**
 11. **return**
 ADDTOTERMINAL($universe.nextDim[value], numString, numValue[1:]$)
 12. **else do**
 13. **return**
 ADDTODIMENSION($universe.nextDim[value], numString, numValue[1:]$)
-

The above Algorithm 1 will be exposed as a member function of the Universe class (the only class exposed to the user). It will take up a number to be inserted into the universe, convert it into a string, add zeroes in the starting to make the length equal to size of the universe. Then, it will extract the first digit from the left and find the next dimension or terminal where it needs to be inserted and give it a function call for addition.

Algorithm 2: Search number in Universe

function SEARCHINUNIVERSE(universe, num):

1. $numString \leftarrow string(num)$
 2. $numLen \leftarrow length(numString)$
 3. **if** $numLen > universe.dimensions-1$ **do**
 4. **return** False
 5. Add to $numString$ in the beginning a string with every character '0' of size $(universe.dimensions - numLen - 1)$
 6. $numValue \leftarrow numString$
 7. $value \leftarrow int(numValue[0])$
 8. **if** $universe.nextDim[value]=NULL$ **do**
 9. **return** False
 10. **if** $length(numValue[1:]) = 1$ **do**
 11. **return**
 $SEARCHINTERMINAL(universe.nextDim[value], numString, numValue[1:])$
 12. **else do**
 13. **return**
 $SEARCHINDIMENSION(universe.nextDim[value], numString, numValue[1:])$
-

The Algorithm 2 will be exposed as a member function of the Universe class. It will take up a number to be inserted into the universe, convert it into a string, add zeroes in the starting to make the length equal to size of the universe. Then, it will extract the first digit from the left and find the next dimension or terminal where it needs to be searched and give it a function call for search.

Algorithm 3: Delete number from Universe

function DELETEFROMUNIVERSE(universe, num):

```

1. numString ← string(num)
2. numLen ← length(numString)
3. if numLen > universe.dimensions-1 do
4.     return False
5. Add to numString in the beginning a string with every character '0' of size (universe.dimensions- numLen-1)
6. numValue ← numString
7. value = int(numValue[0])
8. if universe.nextDim[value]=NULL do
9.     return False
10. if length(numValue[1:]) = 1 do
11.     return
        DELETEFROMTERMINAL(universe.nextDim[value],numString,numValue[1:])
12. else do
13.     return
        DELETEFROMDIMENSION(universe.nextDim[value],numString,numValue[1:])

```

The Algorithm 3 will be exposed as a member function of the Universe class. It will take up a number to be inserted into the universe, convert it into a string, add zeroes in the starting to make the length equal to size of the universe. Then, it will extract the first digit from the left and find the next dimension or terminal from where it needs to be deleted and give it a function call for delete.

Algorithm 4: Extract Sorted Numbers from Universe or Dimension

function EXTRACTSORTED (a):

```

1. extractSorted ← empty array
2. for each dimension in a.nextDim do
3.     if dim not = NULL do
4.         extractSorted.append(dim.extractSorted())
5. return extractSorted

```

Algorithm 4 will be exposed as a member function of the Universe class. It will iteratively give call to all the next dimensions [0..9] and will keep on appending the results to an array which be initialized beforehand. Finally, this array will be returned.

Algorithm 5: Add number to Dimension

function ADDTODIMENSION(*dimension*, *numString*, *numValue*):

```

1. value ← int(numValue[0])
2. if dimension.nextDim[value] = NULL do
3.     if length(numValue) not = 1 do
4.         dimension.nextDim[value] = Dimension
           dimension.dimension+1,value)
5.     else do
6.         dimension.nextDim[value] = Terminal(dimension.dimension+1,value)
7.     if length(numValue[1:]) = 1 do
8.         return
           ADDTODIMENSION(universe.nextDim[value],numString,numValue[1:])
9.     else do
10.        return
           ADDTODIMENSION(universe.nextDim[value],numString,numValue[1:])

```

In Algorithm 5 function extracts the first digit from the left from the number and iteratively calls itself for the next dimension for that digit and finally, when only one digit is left, it calls the AddToTerminal function. If there is no next dimension present for some digit, it initializes one.

Algorithm 6: Search number in Dimension

function SEARCHINDIMENSION(*dimension*, *numString*, *numValue*):

```

1. value ← int(numValue[0])
2. if dimension.nextDim[value] = NULL do
3.     return False
4. if length(numValue[1:]) = 1 do
5.     return
           SEARCHINTERMINAL(universe.nextDim[value],numString,numValue[1:])
6. else do
7.     return
           SEARCHINDIMENSION(universe.nextDim[value],numString,numValue[1:])

```

In Algorithm 6 the function extracts the first digit from the left from the number and iteratively calls itself for the next dimension for that digit and finally, when only one digit is left, it calls the SearchInTerminal function.

Algorithm 7: Delete number from Dimension

```

function DELETEFROMDIMENSION(dimension, numString, numValue):
1. value ← int(numValue[0])
2. if dimension.nextDim[value] = NULL:
3.     return False
4. if length(numValue[1:]) = 1 do
5.     return
        DELETEFROMTERMINAL(universe.nextDim[value],numString,numValue[1:])
6. else do
7.     return
        DELETEFROMDIMENSION(universe.nextDim[value],numString,numValue[1:])

```

In Algorithm 7, the function extracts the first digit from the left from the number and iteratively calls itself for the next dimension for that digit and finally, when only one digit is left, it calls the DeleteFromTerminal function.

Algorithm 8: Add number to terminal

```

function ADDTOTERMINAL(terminal, numString, numValue):
1. terminal.elements.append(int(numString))
2. return True

```

In Algorithm 8, the function simply appends the integer value of the number to the member list of the terminal.

Algorithm 9: Search number in terminal

```

function SEARCHINTERMINAL(terminal, numString, numValue):
1. if int(numString) in terminal.elements do
2.     return self.elements.search(int(numString))
3. else
4.     return False

```

In Algorithm 9, the described function returns true if the integer value of the number is present in the member list of the terminal.

Algorithm 10: Delete number from terminal

```

function DELETEFROMTERMINAL(terminal, numString, numValue):
1. if int(numString) in terminal.elements do
2.     return self.elements.delete(int(numString))
3. else
4.     return False
    
```

In Algorithm 10, the function deletes and returns true if the integer value of the number is present in the member list of the terminal otherwise it returns false.

Algorithm 11: Extract sorted numbers from terminal

```

function EXTRACTSORTEDFROMTERMINAL(terminal):
1. return terminal.elements
    
```

In Algorithm 11, the function returns the array of member elements present in the terminal.

3.5 Theoretical Analysis

The time complexity analysis of various operations performed on the data structure are given in Table 8.

Table 8. Asymptotic time complexity analysis for various operations supported by the proposed data structure

S.No.	Operation	Best case	Average case	Worst case
1	Insert	$\Omega(u)$	$\theta(u)$	$O(u)$
2	Delete	$\Omega(u)$	$\theta(u)$	$O(u)$
3	Search	$\Omega(u)$	$\theta(u)$	$O(u)$
4	Extract sorted numbers	$\Omega(nu)$	$\theta(nu)$	$O(nu)$

The theoretical time complexity analysis of the data structure has been compared for various operations in this section. The X-axis denotes the number of elements present in the data structure. As we move with the X-Axis, the size of the data structure increases. The Y-Axis is a theoretical order denotation of the worst-case time complexity for the specific operation as presented in Fig. 3.

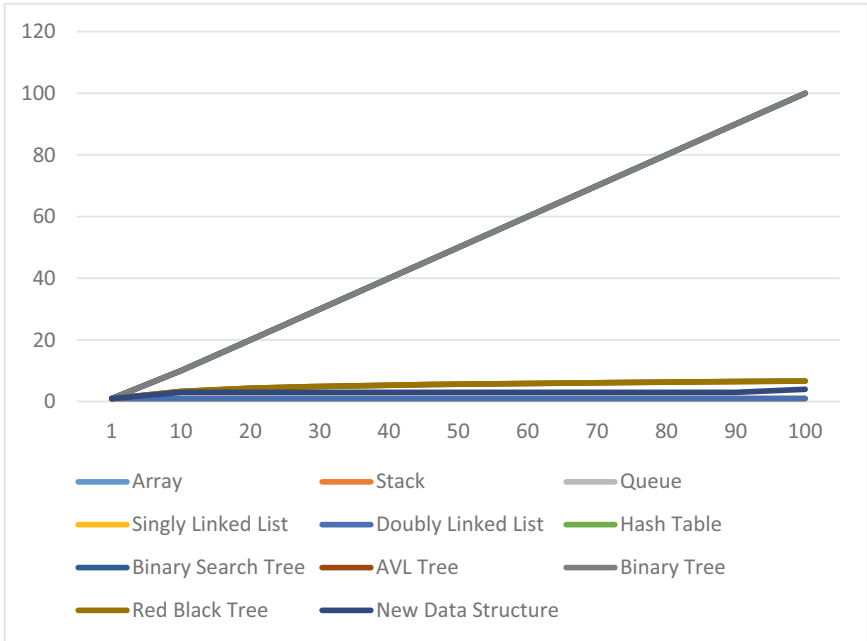


Fig. 3. Time complexity comparison of various data structure for insert operation

The graph in Fig. 4 shows the time complexity analysis for various commonly used data structures for the insertion operation. The Y-Axis here represents the value in which the order of time it will take to execute. As we can see, for some of the data structures, the insertion time complexity goes in linear time while for some, it is very less and we

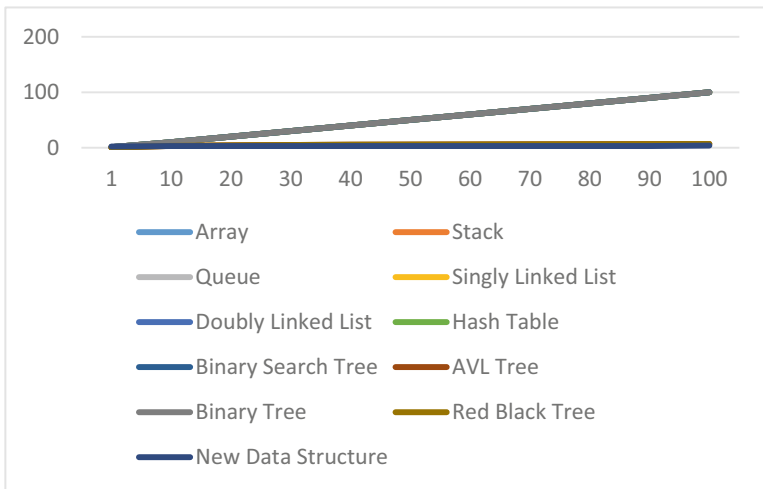


Fig. 4. Time complexity comparison of various data structure for search operation

can see that for the proposed data structure, it lies in the lower level. The size has been kept up to 100 to ensure that the reader can distinguish between the lines in the graph.

Figure 4 presented the time complexity analysis for various commonly used data structures for the search operation. The Y-Axis here represents the value in which the order of time it will take to execute. One can notice that here, that the proposed data structure has the best performance in terms of worst-case search complexity whereas most of the data structures take linear time to search for an element in the data structure. However few like AVL Tree and Red Black Tree have search complexity near to the proposed data structure. A separate graph for deletion of a particular element from the data structure has not been presented in this paper as it is same as to the search complexity for all the mentioned data structures.

Sorting is just one of the applications of this data structure. For testing the data structure to sort numbers or extracted sorted numbers from the data structure, the following test cases were used.

- All elements zero
- All elements sorted from 0 to $n - 1$
- All elements reverse sorted from $n - 1$ to 0.

Since, it is a data structure and not a sorting algorithm, a combination of insertAll() and extractSorted() was used to do comparison with other sorting algorithms in terms of time complexity and space complexity. The asymptomatic complexity of both the algorithms is presented in Table 9. Here n is the number of elements present and u is the number of dimensions for the data structure.

Table 9. Asymptotic time complexity analysis for sorting algorithms

S.No.	Algorithm	Best case	Average case	Worst case
1	Merge Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n \log n)$
2	Quick Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n^2)$
3	Bubble Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$
4	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
5	Insertion Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$
6	Counting Sort	$\Omega(n + k)$	$\Omega(n + k)$	$\Omega(n + k)$
7	NDS Sort	$\Omega(nu)$	$\theta(nu)$	$O(nu)$

If we take numbers up to 10,000,000 the value of u will be 8. Hence for time complexity, NDS Sort in practical scenario will perform better than Merge Sort. This is because NDS Sort is not a comparison-based sorting algorithm. Table 10 shows the space complexity for various sorting algorithms and the u -dimensional data structure.

Table 10. Asymptotic space complexity analysis for sorting algorithms

S.No.	Algorithm	Best case	Average case	Worst case
1	Merge Sort	$\Omega(n)$	$\theta(n)$	$O(n)$
2	Quick Sort	$\Omega(\log n)$	$\Omega(\log n)$	$O(n)$
3	Bubble Sort	$O(1)$	$O(1)$	$O(1)$
4	Selection Sort	$O(1)$	$O(1)$	$O(1)$
5	Insertion Sort	$O(1)$	$O(1)$	$O(1)$
6	Counting Sort	$\Omega(k)$	$\Omega(k)$	$\Omega(k)$
7	NDS Sort	$\Omega(n)$	$\theta(nu)$	$O(nu)$

In the best case, all the elements will belong to only one dimension, let’s say, there is only one element repeated. In that case, the space complexity will be linear.

In the worst case, if elements belong to all different dimensions at all level, in the last dimension, there will be at max n children, reducing by the factor of 10 in each dimension as we move up the tree implementation. Since, the dimensions are u the space complexity will be in the order of nu .

4 Results and Discussion

The testing of sorting numbers with the proposed data structure and Merge Sort was done on Google Colab which uses a Virtual Machine with Intel(R) Xeon(R) CPU @ 2.20 GHz, RAM of 13 GB and a HDD Size of 110 GB approx. The observed time taken by the algorithms is given in Table 11. The column names specify the number of elements taken. The range taken was 0 to $n - 1$, $n - 1$ to 0 and all elements 0. Each cell represents the time taken by algorithm in seconds. Time Complexity is presented in Figs. 5, 6 and 7.

Table 11. Time taken by system in sec to execute NDS Sort and other sorting algorithms using Python on Google Colab

Algorithm	Input	10	100	1000	10,000	100,000	1,000,000	10,000,000
Merge Sort	0 to n-1	0.0001	0.0003	0.004	0.0514	0.6884	7.6915	89.4555
	n-1 to 0	4.1962E-05	0.000387	0.00605011	0.052137	0.636338	7.418624	86.438826
	All 0	6.84E-05	0.000546	0.00457239	0.057035	0.737706	7.427241	85.912622
Quick Sort	0 to n-1	0.00174332	0.001148	0.05484915	5.037439	Crashed	Crashed	Crashed
	n-1 to 0	4.3869E-05	0.001741	0.07280445	8.614189	Crashed	Crashed	Crashed
	All 0	3.5286E-05	0.001638	0.21693325	23.6059	Crashed	Crashed	Crashed
Bubble Sort	0 to n-1	0.00016713	0.000635	0.06683016	6.313508	Crashed	Crashed	Crashed
	n-1 to 0	3.2902E-05	0.00242	0.15939116	17.38576	Crashed	Crashed	Crashed
	All 0	2.3603E-05	0.000927	0.06036782	6.286249	Crashed	Crashed	Crashed
Selection Sort	0 to n-1	8.4639E-05	0.000704	0.05371523	4.918	489.1333	Crashed	Crashed
	n-1 to 0	2.5749E-05	0.000838	0.05344391	5.174935	Crashed	Crashed	Crashed
	All 0	1.4544E-05	0.000751	0.04826069	4.75796	Crashed	Crashed	Crashed
Insertion Sort	0 to n-1	0.0000083	0.000056	0.00052929	0.005517	0.033569	0.335155	3.4667594
	n-1 to 0	1.7166E-05	0.00092	0.10612702	10.77562	Crashed	Crashed	Crashed
	All 0	8.3447E-06	3.46E-05	0.00031519	0.0029	0.03819	0.305743	3.0626819
Counting Sort	0 to n-1	2.3603E-05	0.000134	0.00172091	0.008509	0.086685	0.867572	8.5488007
	n-1 to 0	2.718E-05	0.000135	0.00148439	0.008563	0.082525	0.85849	8.4852989
	All 0	2.5511E-05	0.000132	0.00106096	0.011068	0.0807	0.83579	6.6027911
NDS Sort	0 to n-1	0.0001	0.0003	0.0029	0.0365	0.4191	4.7976	55.8676
	n-1 to 0	7.1287E-05	0.000255	0.00345135	0.033077	0.401883	4.661522	53.242062
	All 0	0.00017571	0.000271	0.00270486	0.034662	0.400066	4.643523	51.733629

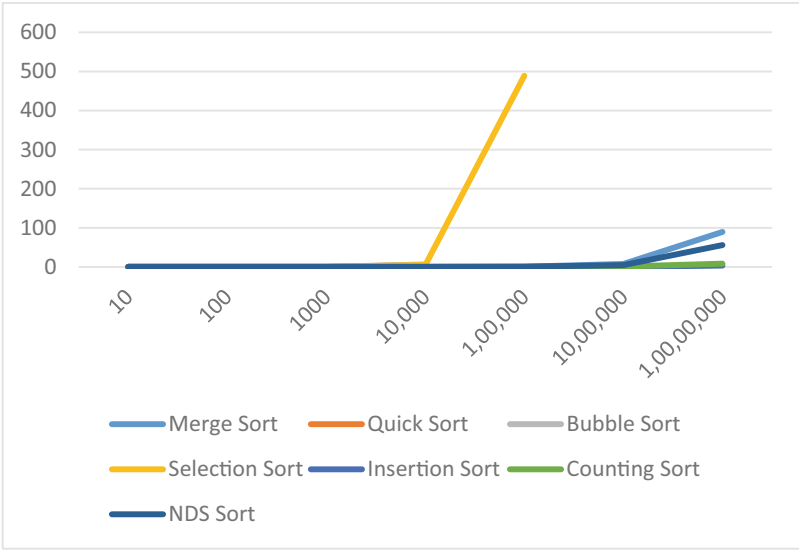


Fig. 5. Time complexity comparison of sorting algorithms for numbers 0 to $n - 1$

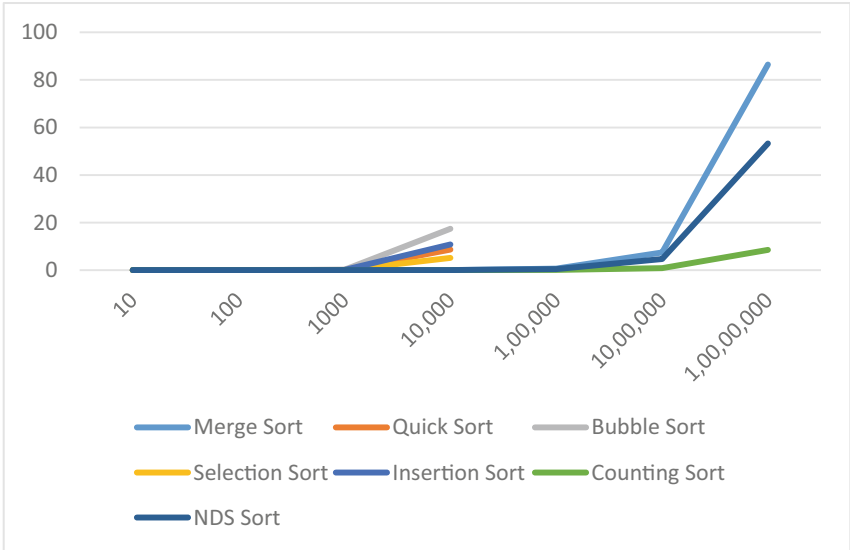


Fig. 6. Time complexity comparison of sorting algorithms for numbers from $n - 1$ to 0 (reverse sorted order)

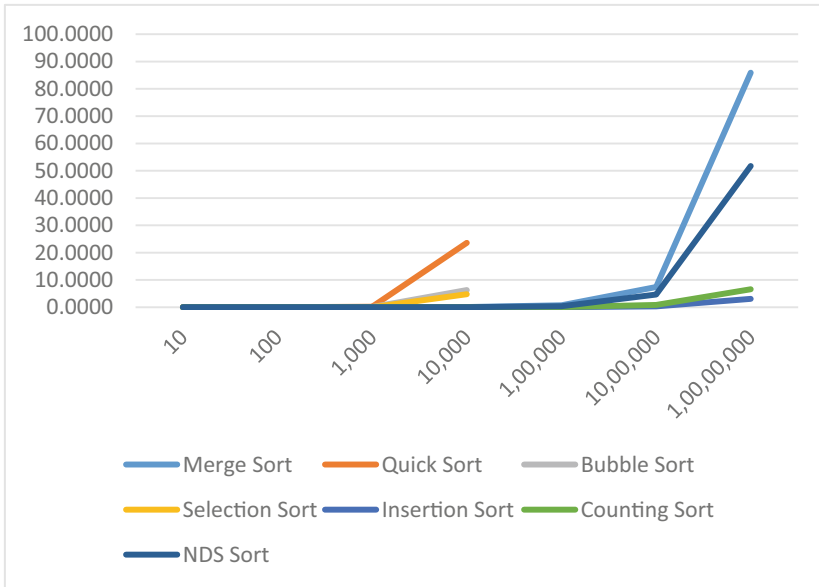


Fig. 7. Time complexity comparison of sorting algorithms for all elements 0

5 Conclusion

The data structure has efficient methods for adding, searching and deletion of numbers in the data structure with time complexity of each operation independent from the number of elements present in the data structure. The NDS Sort, proposed in the paper which is nothing but utilizing the ExtractSorted Method of the data structure. The NDS Sort, is more efficient than many sorting algorithms for positive integers. Extensive evaluation shows that it significantly outperforms Merge Sort and other sorting techniques over several testing scenarios mentioned in the paper in previous sections.

References

1. Fredkin, E.: Trie memory. *Commun. ACM* **3**, 490–499 (1960). <https://doi.org/10.1145/367390.367400>
2. Huang, Y., Duan, L.-Y., Wang, Z., Lin, J., Chandrasekhar, V., Huang, T.: A multi-block N-ary trie structure for exact r-neighbour search in hamming space. In: 2017 IEEE International Conference on Image Processing (ICIP) (2017). <https://doi.org/10.1109/icip.2017.8296455>
3. Shabaz, M., Kumar, A.: SA sorting: a novel sorting technique for large-scale data. *J. Comput. Netw. Commun.* **2019**, 1–7 (2019). <https://doi.org/10.1155/2019/3027578>
4. Nicolas, A., Cyril, N., Carine, P.: Merge Strategies: from Merge Sort to TimSort (2015). [hal-01212839v2](https://arxiv.org/abs/1512.08392)
5. Bajpai, K., Kots, A.: Implementing and analyzing an efficient version of counting sort (E-counting sort). *Int. J. Comput. Appl.* **98**, 1–2 (2014). <https://doi.org/10.5120/17208-7427>
6. Han, Y., Thorup, M.: Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (2002). <https://doi.org/10.1109/sfcs.2002.1181890>

7. Han, Y.: Improved fast integer sorting in linear space. *Inf. Comput.* **170**, 81–94 (2001). <https://doi.org/10.1006/inco.2001.3053>
8. Nicolas, A., Vincent, J., Cyril, N., Carine, P.: On the worst-case complexity of TimSort. In: 26th Annual European Symposium on Algorithms (ESA 2018), August 2018, Helsinki, Finland, pp. 4:1–4:13 (2018). <https://doi.org/10.4230/LIPIcs.ESA.2018.4>. (hal-01798381)
9. Kumar, P., Gangal, A., Kumari, S.: Recombinant sort: N-dimensional cartesian spaced algorithm designed from synergetic combination of hashing, bucket, counting and radix sort. *Ingénierie des systèmes d information* **25**, 655–668 (2020). <https://doi.org/10.18280/isi.250513>
10. Heinz, S., Zobel, J., Williams, H.E.: Burst tries. *ACM Trans. Inf. Syst.* **20**, 192–223 (2002). <https://doi.org/10.1145/506309.506312>
11. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET 1970 (1970). <https://doi.org/10.1145/1734663.1734671>
12. Bell, J., Gupta, G.: An evaluation of self-adjusting binary search tree techniques. *Softw.: Pract. Exp.* **23**, 369–382 (1993). <https://doi.org/10.1002/spe.4380230403>
13. Lin, R.-B.: Efficient data structures for storing the partitions of integers (2005). <https://www.yumpu.com/en/document/read/52473115/efficient-data-structures-for-storing-partitions-of-integers>
14. Patrascu, M., Thorup, M.: Dynamic integer sets with optimal rank, select, and predecessor search. In: 2014 IEEE 55th Annual Symposium on Foundations of Computer Science (2014). <https://doi.org/10.1109/focs.2014.26>
15. Lancia, G., Dalpasso, M.: FASTSET: a fast data structure for the representation of sets of integers. *Algorithms* **12**, 91 (2019). <https://doi.org/10.3390/a12050091>
16. Nekrich, Y.: New data structures for orthogonal range reporting and range minima queries. In: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1191–1205 (2021). <https://doi.org/10.1137/1.9781611976465.73>
17. Kittitornkun, S., Rattanatanurak, A.: A parallel multi-deque sorting algorithm. In: 2021 25th International Computer Science and Engineering Conference (ICSEC) (2021). <https://doi.org/10.1109/icsec53205.2021.9684620>
18. Srivastava, D., Kohli, R., Gupta, S.: Implementation and statistical comparison of different edge detection techniques. In: Bhatia, S.K., Mishra, K.K., Tiwari, S., Singh, V.K. (eds.) *Advances in Computer and Computational Sciences*. AISC, vol. 553, pp. 211–228. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-3770-2_20
19. Mohiddin, Md.K., Kohli, R., Dutt, V.B.S.S.I., Dixit, P., Michal, G.: Energy-efficient enhancement for the prediction-based scheduling algorithm for the improvement of network lifetime in WSNs. *Wirel. Commun. Mob. Comput.* **2021**, 1–12 (2021). <https://doi.org/10.1155/2021/9601078>
20. Chu, Z., Luo, Y., Jin, P.: An efficient sorting algorithm for non-volatile memory. *Int. J. Softw. Eng. Knowl. Eng.* **31**, 1603–1621 (2021). <https://doi.org/10.1142/s0218194021400143>
21. Schaffer, R., Sedgewick, R.: The analysis of heapsort. *J. Algorithms* **15**, 76–100 (1993). <https://doi.org/10.1006/jagm.1993.1031>