Predictive Sort

Dr. Anupam Shukla and Rahul Kala,

Indian Institute of Information Technology and Management Gwalior, Gwalior, Madhya Pradesh, INDIA

Summary
We know the various sorting algorithms available today. Sorting has become one of the most essential parts of the artificial intelligence algorithms these days. We have so many algorithms like Quick Sort, Hash Sort, Bucket Sort, Radix Sort, Insertion Sort, etc. All these are applied to various problems in their own way. In this paper we present a new sorting algorithm. This algorithm works on the principle of calculating the position of each and every node and then placing it onto that position. The algorithm does not start sorting on an extreme end and march towards the other end, as the other algorithms. Also it does not use a divide and conquer approach. Because in both these approaches, after half time of sorting, one half is completely sorted, but the other half is completely unsorted. Here we consider the case where a huge data is to be sorted. Our algorithm sorts this data evenly with respect to time. Hence after half of the time of algorithmic run, we can predict the appropriate position of any of the node. The basic applicability of such an algorithm lies in situations where there is no time to sort the entire data, but we need approximate positions of node in the final answer. Such an algorithm can be used for weather forecasts where usually the data is very big, or for sorting the raw data generated by GPS for fast data processing. The algorithm was implemented and tested in a random set of data. The results clearly proved the working of the algorithm. The accuracy of the algorithm improved very rapidly at each iteration. This further emphasized on the efficiency of the algorithm. This means that with a small increase in number of iterations, we may be able to get a high gain in performance.

Key Words:
Sorting, algorithms, partial sorting.

1. Introduction

Sorting is definitely the most basic operations which is widely used for data processing and data analysis in various ways. We have numerous sorting algorithms that have been developed over the period. All of these have their own philosophy of design, working, time and space complexities. All of these algorithms solve the problem of sorting data in their own way. Various algorithms exist like Selection Sort, Hashing, Quick Sort, Heap Sort, Bucket Sort etc. As a result the user has to choose between these algorithms, depending on input specifications. Quick sort with an average time complexity of $\theta(n \lg n)$ happens to be quite efficient. Radix sort is another algorithm useful in its own way.

These algorithms work well on limited data where the result comes in finite time, but if the total sorting time is infinite, any algorithm will fail. Hence it is not possible to fully sort the data. But the problem may only require us to know the approximate places where given input elements may be located in the whole sequence. Even though the total sorting time might be infinite, it is possible to find these in finite amount of time. This is with the help of partial sorting.

The purpose of this algorithm is to introduce a new approach that would sort the given huge amount of data. The algorithm can be stopped any moment to get the partially sorted data. This partially sorted data is uniformly distributed throughout the array. Hence we can get a fair idea about the position of any element any given point of time.

The algorithm is named as predictive sort, as using this algorithm, we may be able to predict the position of an element in sequence, without even fully sorting the sequence. The more time we give for sorting algorithm, the better or closer would the result be. If left uninterrupted, the input sequence would be sorted.

In section 2 we have a look at the present algorithms and their strengths and weaknesses. In Section 3 we would discuss the general properties, the algorithm and the various factors affecting the algorithm. Section 4 gives the results. Section 5 gives the conclusion.

2. Present Algorithms

Presently we have many algorithms being used for sorting. All these algorithms take the input sequence as the input and sort the data to generate the output. In the subsequent sections we take a brief look over few of these algorithms.

2.1 Insertion Sort

This algorithm selects elements, one by one. It inserts the selected element into the sorted sequence, so that at the end of all the elements being selected, the final list is sorted[8][10][13].

Strengths: Easy to implement. Good if the sequence is already sorted or almost sorted

Weakness: Large time complexity $\theta(n^2)$

Manuscript received June 5, 2008.
Manuscript revised June 20, 2008.

2.2 Quick Sort

This algorithm uses divide and conquer approach to sort. The problem is divided into two problems and then this algorithm is applied recursively. The results are combined to solve the problem[1][11]
Strengths: Good average time complexity θ(n lg n)
Weakness: In case the sequence is already sorted, time complexity is $O(n^{2)}$.

2.3 Hash Sort

The algorithm uses the principle of hashing to sort the given input sequence. The hash function maps the inputs to the keys. Sorting is done by iterating these keys[5][9].
Strengths: Good average time complexity θ(n)
Weakness: Very poor memory complexity

2.4 Radix Sort

The algorithm iterates through the digits from one end to the other to sort the given input data[3][6][7][12].
Strengths: Good average time complexity θ(n) (Assuming constant number of digits)
Weakness: Complexity high for large number of digits, large number of elements with same digit.

2.5 Heap Sort

The algorithm generates a max-heap or a min-heap out of the given input and then solves the sorting problem[2][14]
Strengths: Good average time complexity θ(n lg n)
Weakness: Provides the sorted sequence from start to end.

3. Predictive Sort

In this section we will discuss the algorithm and the way we implement it. We know that the task is to implement partial sorting, at the same time take care of the time and memory complexities. Here we design an efficient algorithm that sorts the input sequence.

3.1 General Properties

The algorithm works in any given input sequence. The following are the main properties of the algorithm

- The algorithm is a stable sort algorithm
- The performance of the algorithm depends on the division factor (α). The division factor is defined as the number of sub problems, the original problem is divided into.
- For α=2, the time average complexity is θ(n lg n). The worst case time complexity is $O(n^2)$ and the best case time complexity is ώ(n lg n). Hence the asymptotic behavior is similar to that of quick sort
- For α=2, the memory complexity is O(n). This is also same as quick sort, but it requires 2 times more memory as compared to quick sort.

3.2 Algorithm Description

The algorithm works by constantly dividing the problems and solving a unit step of every sub problem. The following are the main concepts of the algorithm

3.2.1 Priority Queue

The basic idea of this algorithm is to maintain a priority queue. The given problem is divided into a set of sub problems. These sub problems are inserted into the priority queue with a priority of the problem size. At any step the highest priority sub problem is selected for further solving. Hence the main loops runs to find a sub problem in the queue. If more than one sub problems are found, the largest one is chosen.

3.2.3 Answer Vector

The answer vector stores the answer after every stage. This vector is a collection of NULLs and numbers. If a number is stored, it means that this is the final number that would be stored in that location, when the sorting ends. If NULL is found, it means that the algorithm still doesn't know the final element that would occupy this position. At any time the user can interrupt the algorithm to see this vector, which would give an idea of how all the elements would occupy the positions.

3.2.3 Sub Problem

A sub problem is defined by <SQ,SS,SE,C>. Here SQ is the sequence of numbers it has to 'sort'. Here sorting means to find the right index of any input element. Every sub problem has a cost C which is the associated size of the sub problem. The elements of sub problem always occupy continuous positions in the final answer (SS to SE). This means that the sub problem is a sequence of input numbers such that all the numbers in it lie in a closed range [MIN,MAX]. It also means that if x is a member of any sub problem and x lies in the interval [MIN,MAX], then x must be in SQ. Here SS is the start index (sequence start) from where the sub problem starts filling its answers and SE is the last index (sequence end) in the final answer vector.

3.2.4 Method of sub division of problem

The whole problem <SQ,SS,SE,C> is to be subdivided into α sub problems (<SQ_1,SS_1,SE_1,C_1>, <SQ_2,SS_2,SE_2,C_2>…..<SQ_α,SS_α,SE_α,C_α >). Hence we need to find relations between the corresponding quantities.
Since we have the input sequence SQ, we may divide this range into α sub ranges, starting from the lowest to the highest. Hence if the range of input sequence SQ be MIN to MAX, we will have the range of various sequences of the sub problems as:

Sub Problem 1: min to min+(1*(max-min+1))/α -1
Sub Problem 2: min+(1*(max-min+1))/α to min+(2*(max-min+1))/α -1
……
Sub Problem i: min+((i-1)*(max-min+1))/α to min+(i*(max-min+1))/α -1
……..
Sub Problem α: min+((i-1)*(max-min+1))/α to min+(i*(max-min+1))/α -1

This means that if we get any input element 'x', then it must belong to the Sub Problem i,
Where i= ((x-min)*α)/(max-min+1)+1
Also we know that the original problem had SS to SE as the start and end of its scope in the final answer. Hence the start and end of the scope of any of the sub problems would be as follows

Sub Problem 1: SS_1=SS, SE_1=SS_1 + number of elements in SQ_1
Sub Problem 2: SS_2=SE_1+1, SE_2=SS_2 + number of elements in SQ_2
……
Sub Problem i: SS_i=SE_{i-1}+1, SE_i=SS_i + number of elements in SQ_i
…….
Sub Problem α: SS_α=$SE_{\alpha-1}$+1, SE_α=SS_α + number of elements in SQ_α

This method works for subdividing a problem into sub problems. But while implementing, we always process the greatest and smallest element of every sub problem before adding it onto the queue. In other words, we know that the minimum most element of every sub problem i, will occupy the position SS_i. Similarly the maximum most element of any sub problem will always occupy the last position, SE_i. Hence we store these two elements in their corresponding positions in the final answer vector. Since these have been solved, the following transformations are applied for every sub problem i.
$SQ_i \leftarrow SQ_i$ – {maximum most element of SQ_i, minimum most element of SQ_i}
$SS_i \leftarrow SS_i$ +1
$SE_i \leftarrow SE_i$ -1

To predict the position of the element at any general time, we first find out the range between which the element is sure to happen. This may be found by iterating through the final answer array. We say that this element lies between indices start and end. The position of any element p may be calculated by assuming that all elements between these are equally distributed. The formula for the index j of the element in such a case is

```
j = start (if start = end)(end-start) /
   (finalAnswer[end]-finalAnswer[start]) *
   (p-finalAnswer[start]) + start(in all other cases)
```

3.3 Algorithm

Step 1: q ← priority queue
Step 2: add a node s in q such that SQ(s) = input sequence, SS(s)=1, SE(s)=n(input size) with priority n (input size)
Step 3: while q is not empty
Do
Step 4: remove node s from q with the maximum cost
Step 5: if SS(s)>SE(s) continue
Step 6: find sub problems s_1,s_2,s_3,s_4….s_α
Step 7: for all sub problems s_i,
Step 8: if SQ_i contains at least 1 element
Step 9: finalAnswer[SS_i]= Minimum most element of SQ_i
Step 10: if SQ_i contains at least 2 elements
Step 11: finalAnswer[SE_i]= Maximum most element of SQ_i
Step 12: $SQ_i \leftarrow SQ_i$ – {maximum most element of SQ_i, minimum most element of SQ_i}
Step 13: $SS_i \leftarrow SS_i$ +1
Step 14: $SE_i \leftarrow SE_i$ -1
Step 15: add sub problem to queue

3.4 Loop Invariants of the algorithm

In order to ensure that the algorithm works correctly, we must ensure that it obeys the conditions of initialization, maintenance and termination

3.4.1 Initialization

Here we insert one sub problem of input size. The whole SQ is the complete input sequence. The SS and SE and the entire dimensions of the final answer vector. At the first run of the loop, this is extracted and is divided into sub problems. The sub problems and generated and added to queue.

3.4.2 Maintenance

Whenever a sub problem is chosen, we divide its entire range into α sub ranges. Since we have covered the entire range, all elements of the problem must be members of any one of the sub problem. Also using the above equations, we may say that these are closed ranges. Also SS and SE are correctly calculated using the above equations. Hence the generation of sub problems is valid.

3.4.3 Termination

Whenever the size of the input sequence SQ in any of the sub problems falls to a level such that there are no elements in it, the condition $SS_i > SE_i$ becomes true and the loop simply extracts the element from queue without further processing. At every iteration, the size of sub problem is first reduced by division, and then by 2 units of replacement. Hence every sub problem finally meets this condition and gets extracted from queue. This terminates the loop at last.

4. Results

The algorithm was implemented and tested. Various test cases were given to the algorithm. The condition of the final answer vector was seen at each iteration. It can be clearly seen that the algorithm works as desired. As the loop proceeded, the positions of elements started becoming clearer and clearer. We could see that the loop evenly filled the entries of the final answer vector.
A total of 100 test cases were randomly generated and the answer was seen at frequent iterations. The input size of these test cases was varied from 100 to 5000. The final answer was made to print on the screen. We also sorted these numbers using standard sorting technique (selection sort). We found out that for all inputs the final result of the standard algorithm and our algorithm matched. Hence the algorithm worked for all inputs.

The following is one of the input given which clearly illustrates the point:
{61,78,33,24,13,65,80,58,91,76,4,9,25,28,99,82,31,94,68,17,66,3,75,54,88,43,63,51,60,9,2,76,19,20}

The condition of the final answer vector at various iterations is as follows:

Iteration 1
{2, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 43, 51, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 99}

Iteration 2
{2, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 43, 51, 54, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 75, 76, NULL, NULL, NULL, NULL, NULL, NULL, 94, 99}

Iteration 3
{2, 3, NULL, NULL, NULL, NULL, NULL, NULL, 20, 24, NULL, NULL, NULL, 33, 43, 51, 54, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 75, 76, NULL, NULL, NULL, NULL, NULL, NULL, 94, 99}

Iteration 4
{2, 3, NULL, NULL, NULL, NULL, NULL, NULL, 20, 24, NULL, NULL, NULL, 33, 43, 51, 54, 58, NULL, NULL, 63, 65, NULL, 68, 75, 76, NULL, NULL, NULL, NULL, NULL, NULL, 94, 99}

Iteration 5
{2, 3, NULL, NULL, NULL, NULL, NULL, NULL, 20, 24, NULL, NULL, NULL, 33, 43, 51, 54, 58, NULL, NULL, 63, 65, NULL, 68, 75, 76, 76, NULL, NULL, 82, 88, 91, 94, 99}

Iteration 6
{2, 3, 4, NULL, 9, 13, NULL, 19, 20, 24, NULL, NULL, NULL, 33, 43, 51, 54, 58, NULL, NULL, 63, 65, NULL, 68, 75, 76, 76, NULL, NULL, 82, 88, 91, 94, 99}

Iteration 7
{2, 3, 4, NULL, 9, 13, NULL, 19, 20, 24, 25, 28, 31, 33, 43, 51, 54, 58, NULL, NULL, 63, 65, NULL, 68, 75, 76, 76, NULL, NULL, 82, 88, 91, 94, 99}

Iteration 8
{2, 3, 4, NULL, 9, 13, NULL, 19, 20, 24, 25, 28, 31, 33, 43, 51, 54, 58, 60, 61, 63, 65, NULL, 68, 75, 76, 76, NULL, NULL, 82, 88, 91, 94, 99}

Iteration 9
{2, 3, 4, NULL, 9, 13, NULL, 19, 20, 24, 25, 28, 31, 33, 43, 51, 54, 58, 60, 61, 63, 65, NULL, 68, 75, 76, 76, 78, 80, 82, 88, 91, 94, 99}

Iteration 10
{2, 3, 4, NULL, 9, 13, NULL, 19, 20, 24, 25, 28, 31, 33, 43, 51, 54, 58, 60, 61, 63, 65, 66, 68, 75, 76, 76, 78, 80, 82, 88, 91, 94, 99}

Iteration 11
{2, 3, 4, 9, 9, 13, NULL, 19, 20, 24, 25, 28, 31, 33, 43, 51, 54, 58, 60, 61, 63, 65, 66, 68, 75, 76, 76, 78, 80, 82, 88, 91, 94, 99}

Iteration 12
{2, 3, 4, 9, 9, 13, 17, 19, 20, 24, 25, 28, 31, 33, 43, 51, 54, 58, 60, 61, 63, 65, 66, 68, 75, 76, 76, 78, 80, 82, 88, 91, 94, 99}

ANSWER
{2, 3, 4, 9, 9, 13, 17, 19, 20, 24, 25, 28, 31, 33, 43, 51, 54, 58, 60, 61, 63, 65, 66, 68, 75, 76, 76, 78, 80, 82, 88, 91, 94, 99}

As we know that the basis of this algorithm is to predict the position of any element, when its final position is not determined. In order to test this capability of the algorithm we conducted another simple test. We randomly generated 1000 elements and sorted them using our algorithm. At each iteration we calculated the Root Mean Square Error (RMS Error).

The RMS Error was defined as:

$$\text{RMS Error} = \text{square root of } \frac{\sum (Pred_i - Act_i)^2}{N}$$

Here $Pred_i$ is the predicted position of the element
Act_i is the actual position in the sorted sequence
N is the number of elements in the sequence
The summation is performed over all elements. This means for all elements we predict the position and add it to the RMS error.

The results of two consecutive runs are given by the Fig 1(a) and Fig 1(b)

Fig. 1 : The RMS Error v/s Number of Iterations

We can clearly see that as the number of iterations increase, the RMS Error drastically decreases. This proves the efficiency of the algorithm. This also proves the fact that by increasing the number of iterations by small margin, the accuracy can be increased a lot.

We also know the fact that many people are interested in finding the exact position of the element, rather than the approximate value.

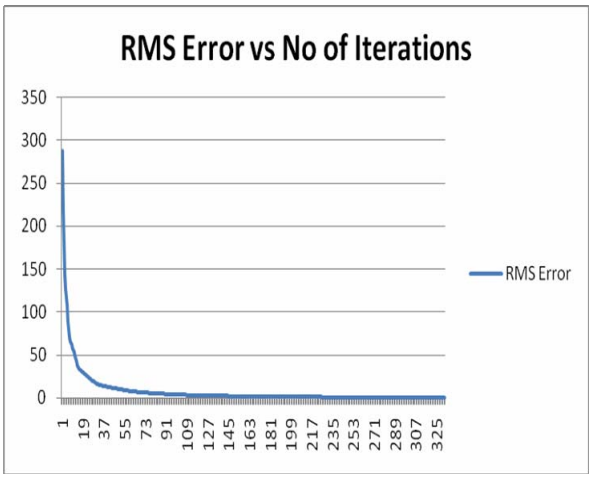

Fig. 1(a) RMS Error for Input 1.

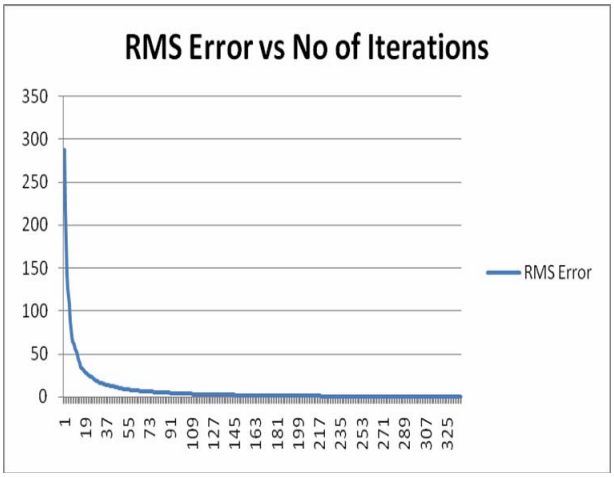

Fig. 1 (b) RMS Error for Input 2.

For this we also calculated the percentage of elements whose positions were predicted precisely by the algorithm. The Fig 2(a) and Fig 2(b) shows this percentage for various iterations for 2 consecutive runs.

Fig. 2 : The Performance of the Algorithm v/s The Number of Iterations

It can again be seen that the efficiency keeps on improving. We get an accuracy of more than 80% in about 60% of the number of iterations.

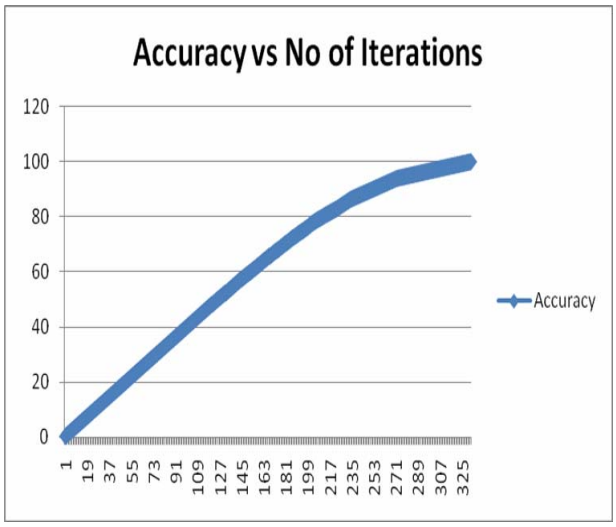

Fig. 2 (a) The accuracy for Input 1.

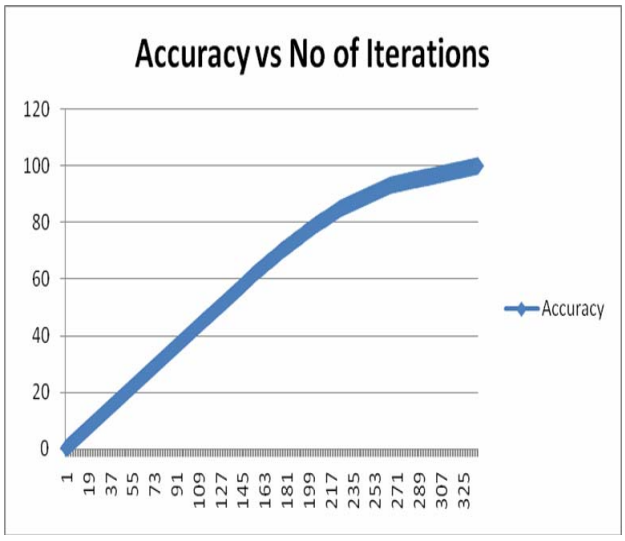

Fig. 2 (b) The accuracy for Input 2.

It may be noted that these only includes the exact results. The RMS Error method of finding accuracy is more practical in implementation, as we expect that the user wants to know the approximate positions. An index up or down would not make a difference.

5. Conclusion

We saw that using this algorithm we were able to sort the given input sequence. The algorithm uniformly filled the elements in the final answer vector. These were filled at their correct places to give the correct final answer. We also observed that the filling of the numbers in the final answer made it possible for us to predict the appropriate position for any of the elements. Hence if we needed to know approximate position of all the elements, complete sorting was not needed. Hence, this algorithm can be used as an effective partial sorting algorithm.

The algorithm used a priority queue to implement the approach. The given problem was divided into sub problems. The maximum and minimum most element of each of these sub problems was placed at the division itself. This process continued to give us the final answer.

We know that α controls the working of the algorithm. If we only want to reduce the final total time, we may set it to 2. But if we want to control the number of elements, in a given time span, it may take different values. The exact relation of this is yet to be studied.

References

[1] Khreisat Laila, "QuickSort A Historical Perspective and Empirical Study", IJCSNS International Journal of Computer Science and Network Security, VOL.7 No.12, December 2007

[2] Sharma Vandana, Singh Satwinder and Kahlon K. S.,"Performance Study of Improved Heap Sort Algorithm and Other Sorting Algorithms on Different Platforms", IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.4, April 2008

[3] Amer Al-Badarneh, Fouad El-Aker, "Efficient Adaptive In-Place Radix Sorting", Informatica, 2004, Vol. 15, No. 3, 295–302, 2004 Institute of Mathematics and Informatics, Vilnius

[4] Islam Tarique Mesbaul, Kaykobad M., "Worst-case analysis of generalized heapsort algorithm revisited", International Journal of Computer Mathematics, Vol. 83, No. 1, January 2006, 59–67

[5] Bratbergsengen Kjell, "Hashing Methods And Relational Algebra Operations"

[6] Khurana Udayan, "Decision Sort and its Parallel Formulation"

[7] Maus Arne, "Sorting by generating the sorting permutation, and the effect of caching on sorting"

[8] Astrachan Owen, "Bubble Sort: An Archaeological Algorithmic Analysis"

[9] Shakhnarovich Gregory, Viola Paul, Darrell Trevor, "Fast Pose Estimation with Parameter Sensitive Hashing"

[10] Nevalainen Olli, Raita Timo, Thimbleby Harold, "An improved insert sort algorithm"

[11] Chen Jing-Chao, "Quick Sort on SCMPDS", Journal Of Formalized Mathematics, Volume 12, Released 2000, Published 2003, Inst. of Computer Science, Univ. of Białystok

[12] Franceschini1Gianni, Muthukrishnan S., Patrascu Mihai, "Radix Sorting With No Extra Space"

[13] Michael A., Martın Farach-Colton, Miguel Mosteiro, "INSERTION SORT is O(n log n)"
[14] Li-Jen Mao, Sheau-Dong Lang, "An Empirical Study of Heap Traversal and Its Applications"

Dr. Anupam Shukla is an Associate Professor in the ICT Department of the Indian Institute of Information Technology and Management Gwalior. He has 19 years of teaching experience. His research interest includes Speech processing, Artificial Intelligence, Soft Computing and Bioinformatics. He has published around 62 papers in various national and international journals/conferences. He is referee for 4 international journals and in the Editorial board of International journal of AI and Soft Computing. He received Young Scientist Award from Madhya Pradesh Government and Gold Medal from Jadavpur University.

Rahul Kala is a student of 3rd Year Integrated Post Graduate Course (BTech + MTech in Information Communication Technology) in Indian Institute of Information Technology and Management Gwalior. His fields of research are robotics, design and analysis of algorithms, artificial intelligence and soft computing. He secured 7^{th} position in the ACM International Collegiate Programming Contest, Kanpur Regionals. He is a student member of ACM. He also secured Al India 8^{th} position in Graduates Aptitude Test in Engineeging-2008 with a percentile of 99.84.