INTRODUCTION

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

- The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order).

- The output is a permutation (reordering) of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2006). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as Big-O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

PROJECT DESCRIPTION

This is a Comparative Study Over Sorting Algorithm to be developed in C. The client should be presented with a well defined welcome page where he/she will be allowed to give predefined username and password to get along with the system. There should be a proper verification process otherwise the user should be redirected to the welcome page. The user can register in this way also. If the end user is a new one then there must be signup process to register him. Once the login is successful there will be options like details of a particular known algorithm, also some of the newly invented algorithm, comparison of different algorithms and lots more. The project may be enhanced by improving the user interface and inventing other algorithms.

FEASIBILITY ANALYSIS

Whatever we think need not be feasible. It is wise to think about the feasibility of any problem we undertake. Feasibility is the study of impact, which happens in the organization by the development of a system. The impact can be either positive or negative. When the positive dominates the negatives then the system is considered is feasible. Here the feasible study can be performed in two ways such as Technical Feasibility and Economical Feasibility.

Technical Feasibility:

We can strongly say that it is technically feasible, since there will not be much difficulty in getting required resources for the development and maintaining the system as well. All the resources needed for the development of the software as well as the same is available in the organization itself.

Economical Feasibility:

Development of this application is highly economically feasible. The organization needed not spend much money for the development of the system already available. The only thing is to be done is making an environment for the development with an effective supervision. If we are doing so, we can attain the maximum usability of the corresponding resources. Even after the development, the organization will not be in a condition to invest more in the organization. Therefore, the system is economically feasible.

SYSTEM REQUIREMENTS

This application should be used in Windows XP based environment. The system must be running Windows XP and must meet the following hardware & software requirements.

Minimum Hardware Requirements:

- Hard Disk (minimum 80 MB free space)
- RAM (128 MB or more)
- Processor (Intel, AMD minimum 1 GHz speed)
- Video memory(minimum 16MB)

Software Requirements:

- Turbo C++

Scheduling Criteria:

1. Proto type model are used in this project

CLASSIFICATION

Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the list (*n*). For typical sorting algorithms good behavior is O(*n* log *n*) and bad behavior is O(n^2). (See Big O notation.) Ideal behavior for a sort is O(*n*), but this is not possible in the average case. Comparison-based sorting algorithms, which evaluate the elements of the list via an abstract key comparison operation, need at least O(*n* log *n*) comparisons for most inputs.
- Computational complexity of swaps (for "in place" algorithms).
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place". Strictly, an in place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(*n*)) additional memory is considered "in place".
- Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, *etc.*. Exchange sorts include bubble sort and quick sort. Selection sorts include shaker sort and heapsort.

Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

STABILITY

Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list. When equal elements are indistinguishable, such as with integers or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first component:

(4, 2) (3, 7) (3, 1) (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

(3, 7) (3, 1) (4, 2) (5, 6) (Order maintained)
(3, 1) (3, 7) (4, 2) (5, 6) (Order changed)

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional computational cost.

Sorting based on a primary, secondary, tertiary, etc. sort key can be done by any sorting method, taking all sort keys into account in comparisons (in other words, using a single composite sort key). If a sorting method is stable, it is also possible to sort multiple times, each time with one sort key. In that case the keys need to be applied in order of increasing priority.

Example: sorting pairs of numbers as above by second, then first component:
(4, 2) (3, 7) (3, 1) (5, 6) (Original)
(3, 1) (4, 2) (5, 6) (3, 7) (After sorting by second component)
(3, 1) (3, 7) (4, 2) (5, 6) (After sorting by first component)

On the other hand:
(3, 7) (3, 1) (4, 2) (5, 6) (After sorting by first component)

(3, 1) (4, 2) (5, 6) (3, 7) (After sorting by second component, order by first component is disrupted).

THEORITICAL BACKGROUND

BUBBLE SORT:

An alternate way of putting the largest element at the highest index in the array uses an algorithm called *bubble sort*. While this method is neither as efficient, nor as straightforward, as selection sort, it is popularly used to illustrate sorting. We include it here as an alternate method.

Like selection sort, the idea of bubble sort is to repeatedly move the largest element to the highest index position of the array. As in selection sort, each iteration reduces the effective size of the array. The two algorithms differ in how this is done. Rather than search the entire effective array to find the largest element, bubble sort focuses on successive adjacent pairs of elements in the array, compares them, and either swaps them or not. In either case, after such a step, the larger of the two elements will be in the higher index position. The focus then moves to the next higher position, and the process is repeated. When the focus reaches the end of the effective array, the largest element will have ``bubbled'' from whatever its original position to the highest index position in the effective array.

For example, consider the array:

45, 67, 12, 34, 25, 39

In the first step, the focus is on the first two elements which are compared and swapped, if necessary. In this case, since the element at index 1 is larger than the one at index 0, no swap takes place.

45, 67, 12, 34, 25, 36

Then the focus moves to the elements at index 1 and 2 which are compared and swapped, if necessary. In our example, 67 is larger than 12 so the two elements are swapped. The result is that the largest of the first three elements is now at index 2.

45, 12, 67, 34, 25, 39

The process is repeated until the focus moves to the end of the array, at which point the largest of all the elements ends up at the highest possible index. The remaining steps and result are:

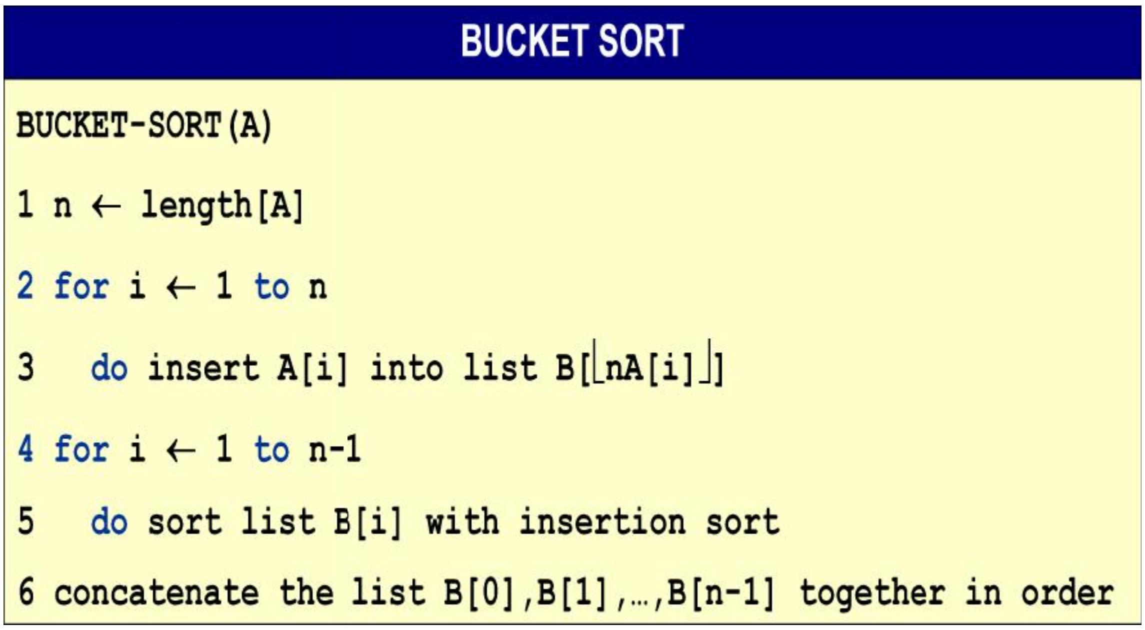
45, 12, 34, 67, 25, 39

45, 12, 34, 25, 67, 39

45, 12, 34, 25, 39,67

BUCKET SORT:

The idea behind bucket sort is that if we know the range of our elements to be sorted, we can set up buckets for each possible element, and just toss elements into their corresponding buckets. We then empty the buckets in order, and the result is a sorted list. In implementing this algorithm, we can easily use an array to represent our buckets, where the value at each array index will represent the number of elements in the corresponding bucket. If we have integers on the range [0..max], then we set up an array of (max + 1) integers and initialize all the values to zero. We then proceed sequentially through the unsorted array, reading the value of each element, going to the corresponding index in the buckets array, and incrementing the value there.



BUCKET SORT

```
BUCKET-SORT(A)
n ← length[A]
for i ← 1 to n
    do insert A[i] into list B[$\lfloor nA[i] \rfloor$]
for i ← 1 to n-1
    do sort list B[i] with insertion sort
concatenate the list B[0],B[1],…,B[n-1] together in order
```

COUNTING SORT:

A sorting technique that is used when the range of keys is relatively small and there are duplicate keys. Counting sorts differ from sorts that compare data in multiple passes. They work by creating an array of counters the size of the largest integer in the list; therefore, the keys must be integers or data that can be readily converted to integers.

Counting Sort Vs. Rapid Sort:

Space equal to the original unordered list is allocated in memory, and the second pass of the counting sort scans the original list and uses the data in the counters to move each record from the original list into the sorted list.

A "rapid sort" is similar to a counting sort, except that it is used to only count the number of occurrences of key fields with no additional data. Instead of using the data in the counters to move records into a new sequence, the counter data in a rapid sort are used to print each key field along with its corresponding count.

ALGORITHM OF COUNTING SORT:

```
for i←1 to k
do C[i]←0
for j←1 to n
do C[A[ j]] ++
/* After this step, C[i] contains num of elements
whose key =i */
for i←2 to k
do C[i]← C[i] + C[i–1]
/* After this step C [i] = |{key ≤ i}| */
/*C indicates where to put the next A[j] when copying B← A */
for j←n downto 1
do x ← A[ j ]
B[C[x]] ←x
C[x] –
```

HEAP SORT:

An Example of Heap sort:

Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort it in ascending order using heap sort.

Steps:

1. Consider the values of the elements as priorities and build the heap tree.
2. Start deleteMin operations, storing each deleted element at the end of the heap array.

After performing step 2, the order of the elements will be opposite to the order in the heap tree.
Hence, if we want the elements to be sorted in ascending order, we need to build the heap tree
in descending order - the greatest element will have the highest priority.

Note that we use only one array , treating its parts differently:

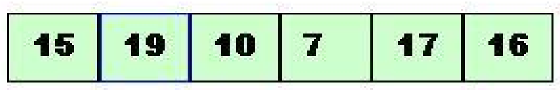
a) when building the heap tree, part of the array will be considered as the heap, and the rest part - the original array.
b) when sorting, part of the array will be the heap, and the rest part - the sorted array.

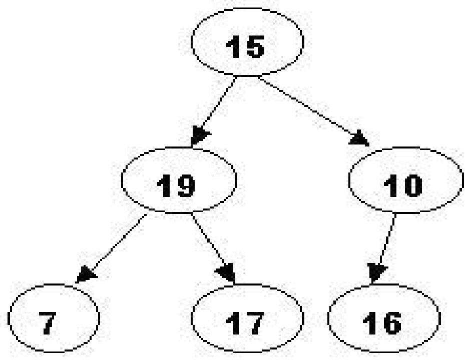
This will be indicated by colors: white for the original array, blue for the heap and red for the sorted array

Here is the array: 15, 19, 10, 7, 17, 6

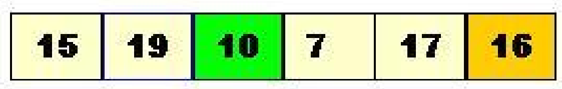
A. Building the heap tree:

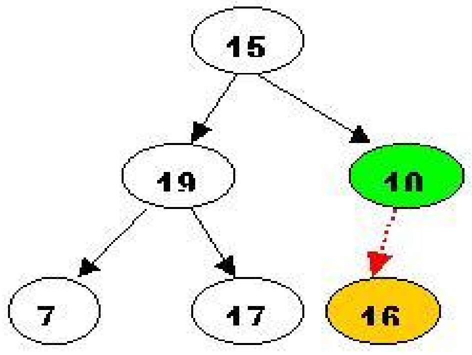
The array represented as a tree, complete but not ordered.



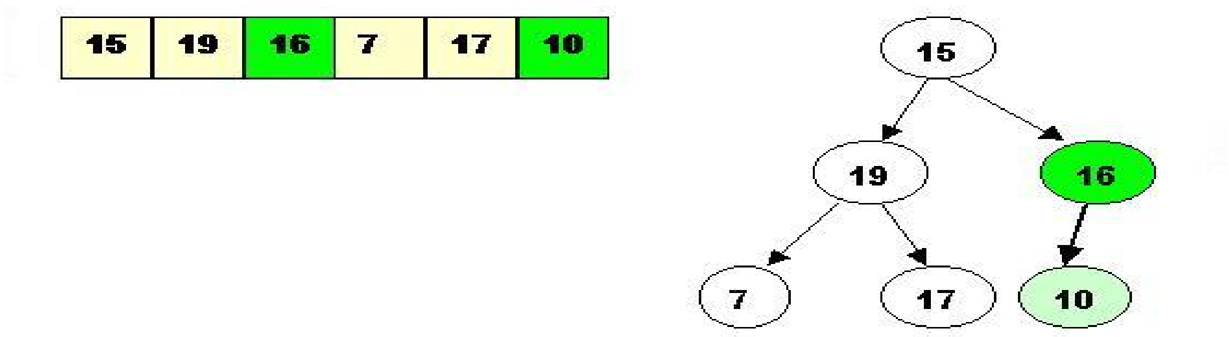


Start with the rightmost node at height 1 - the node at position 3 = Size/2. It has one greater child and has to be percolated down:

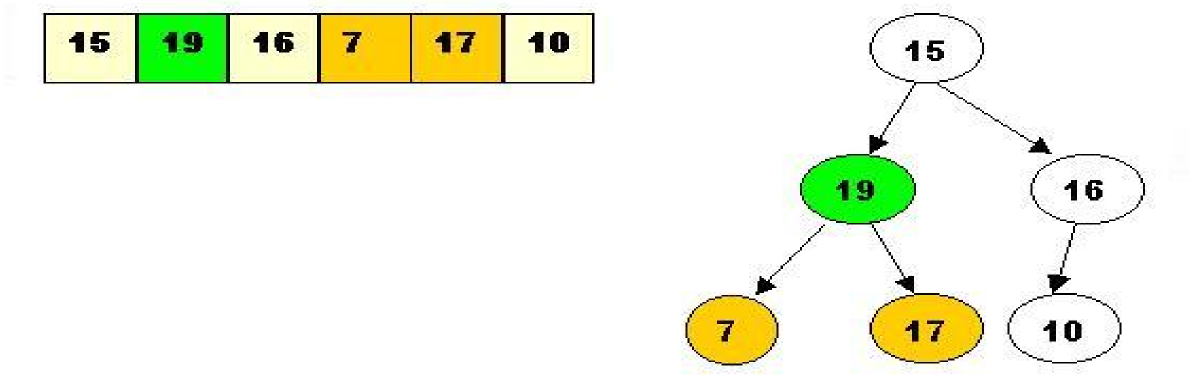




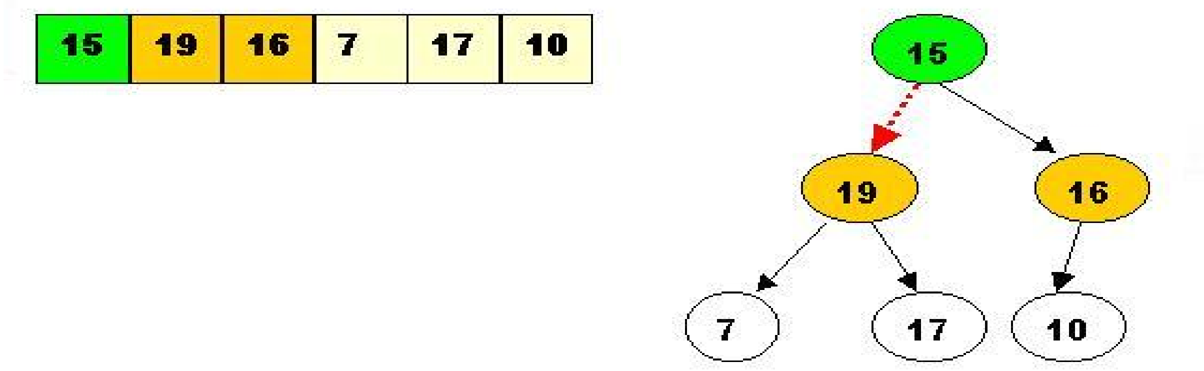
After processing array[3] the situation



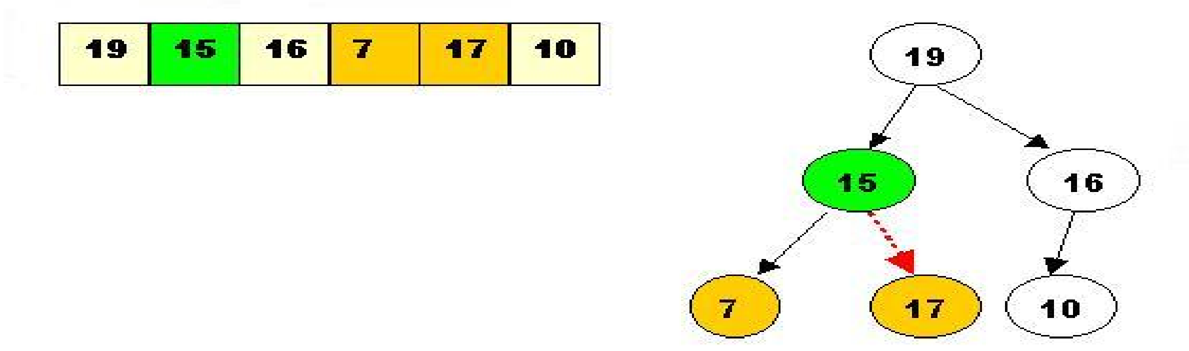
Next comes array[2]. Its children are smaller, so no percolation is needed.



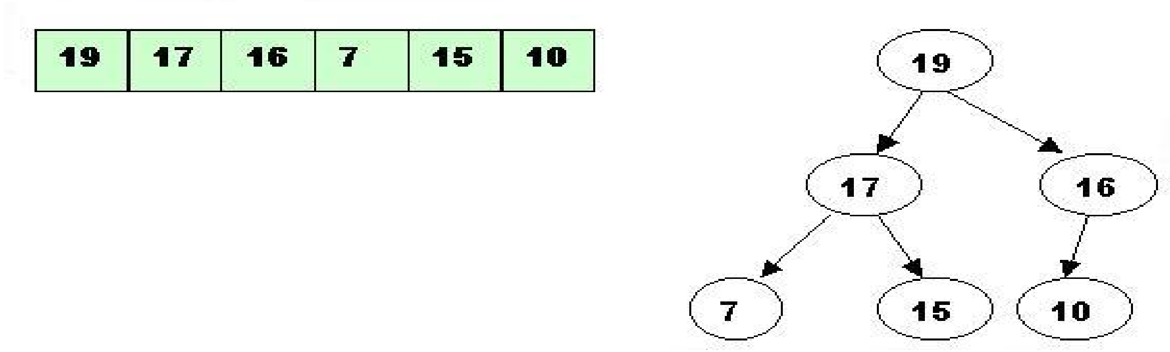
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array[1] has to be percolated down to the left, swapped with array[2].



As a result the situation is:



The children of array[2] are greater, and item 15 has to be moved down further, swapped with array[5].

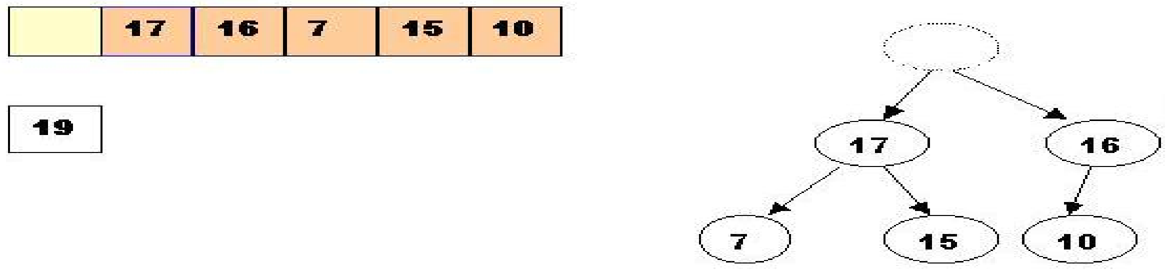


Now the tree is ordered, and the binary heap is built.

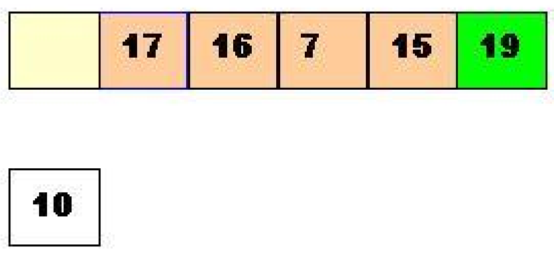
B. <u>Sorting - performing deleteMax operations:</u>

1. Delete the top element 19.

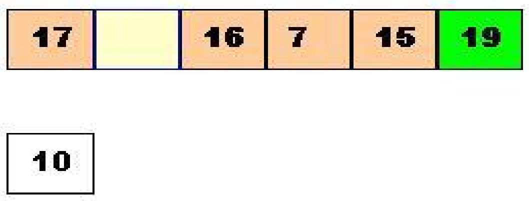
1.1. Store 19 in a temporary place. A hole is created at the top

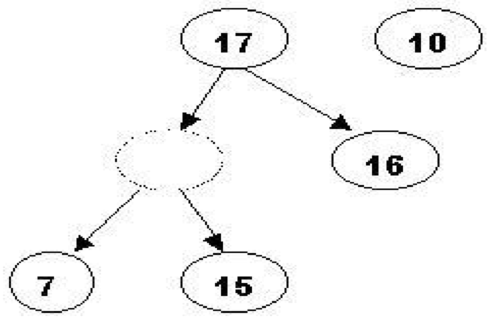


1.2.Swap 19 with the last element of the heap. As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array.

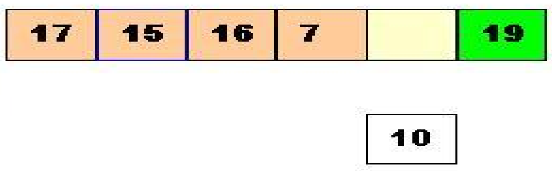


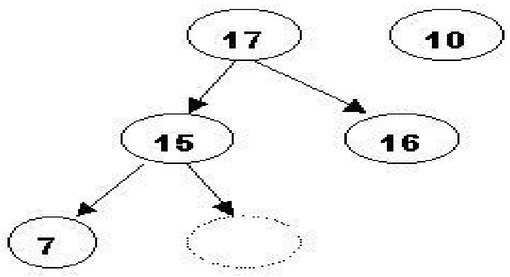
1.3. Percolate down the hole



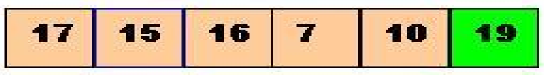


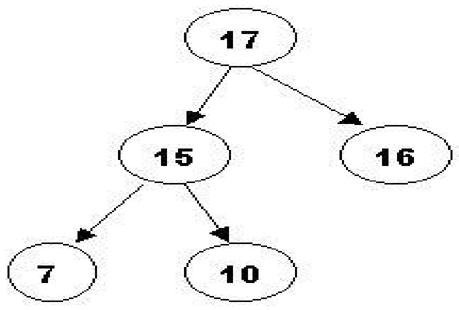
1.3.Percolate once more (10 is less that 15, so it cannot be inserted in the previous hole)





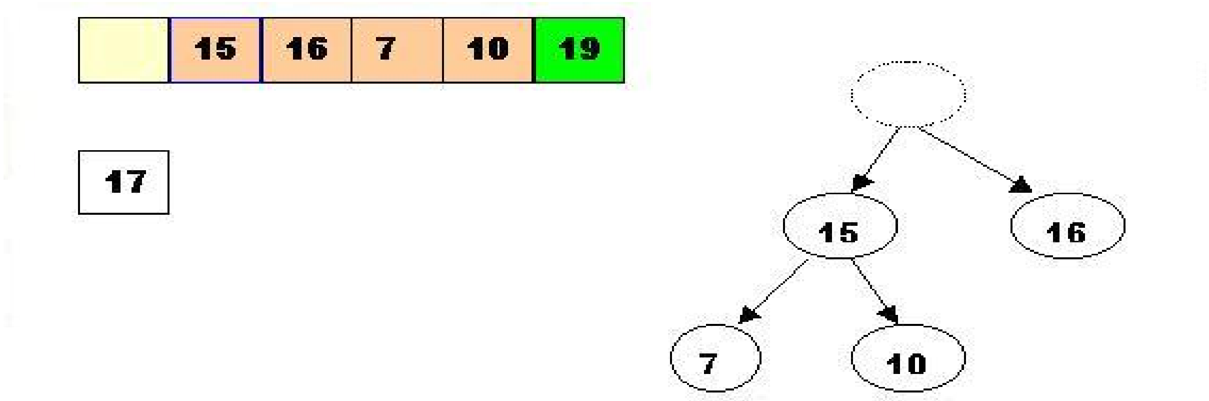
Now 10 can be inserted in the hole



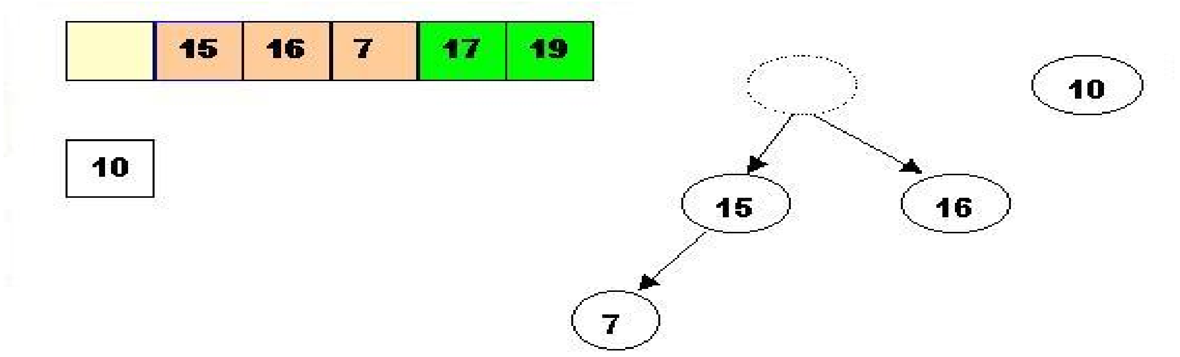


2. DeleteMax the top element 17

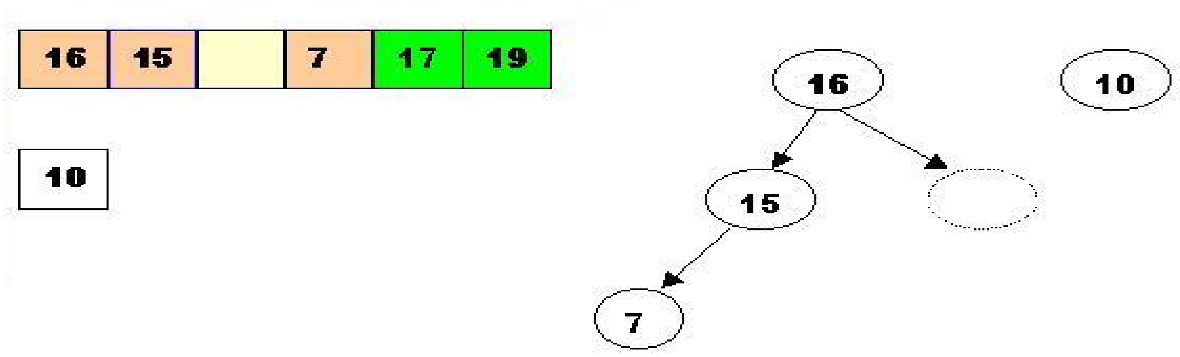
2.1. Store 17 in a temporary place. A hole is created at the top



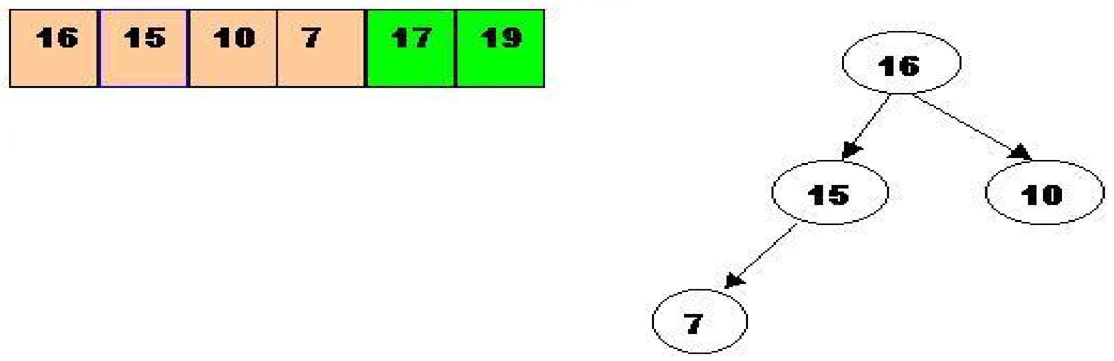
2.2. Swap 17 with the last element of the heap. As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



2.3. The element 10 is less than the children of the hole, and we percolate the hole down.

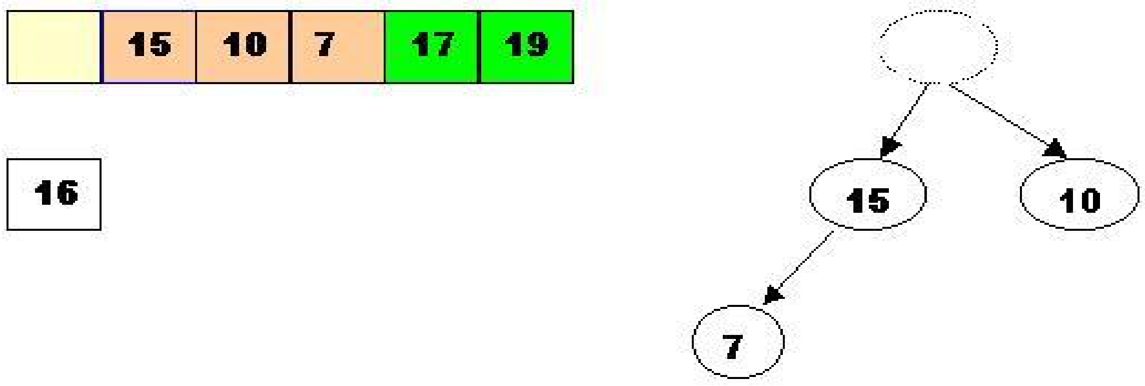


2.4. Insert 10 in the hole

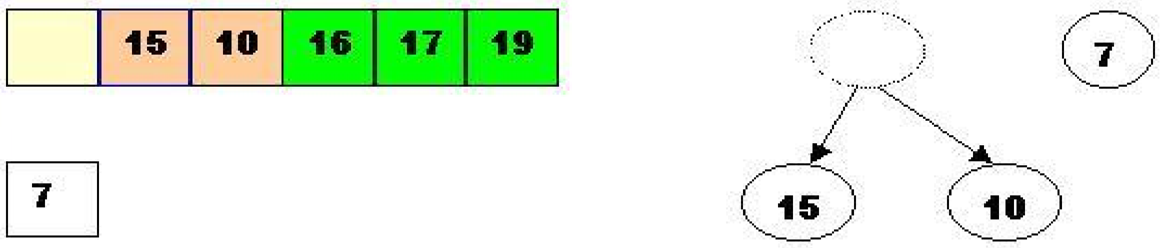


3. DeleteMax 16

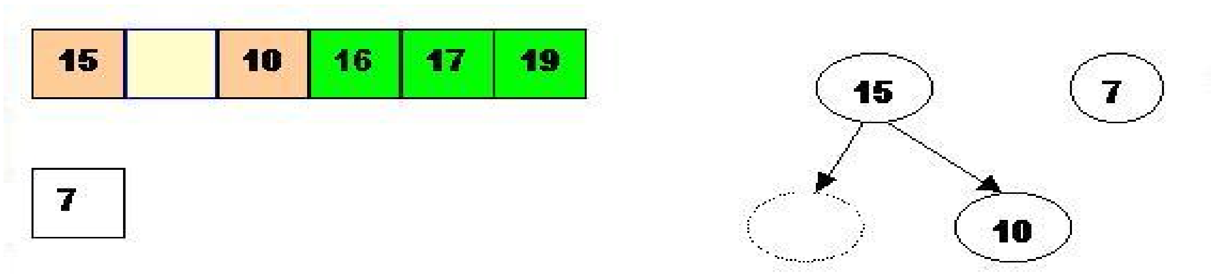
3.1. Store 16 in a temporary place. A hole is created at the top



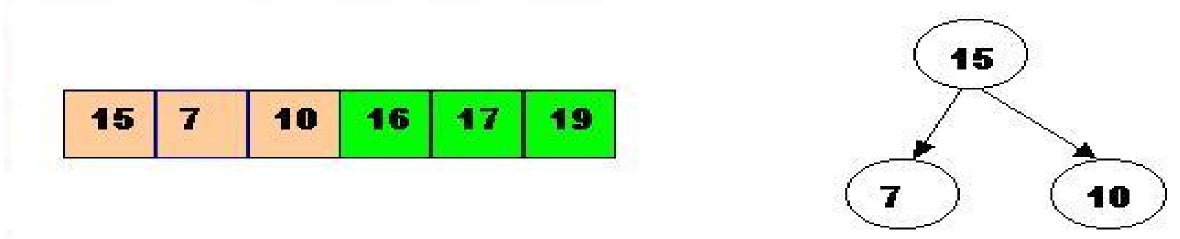
3.2. Swap 16 with the last element of the heap. As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



3.3. Percolate the hole down (7 cannot be inserted there - it is less than the children of the hole)

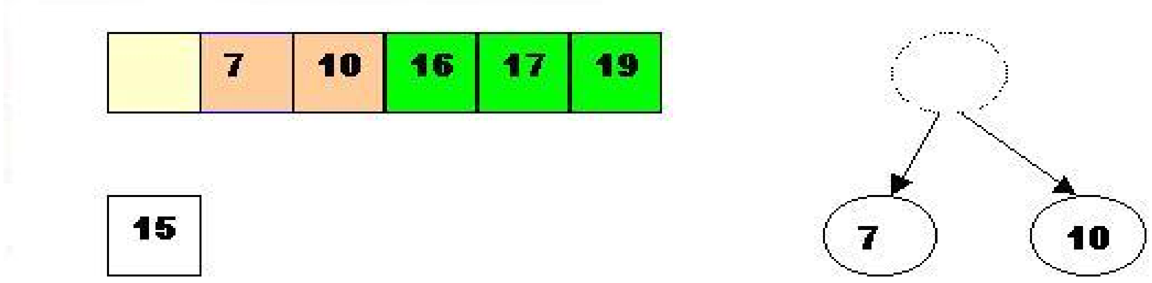


3.4. Insert 7 in the hole

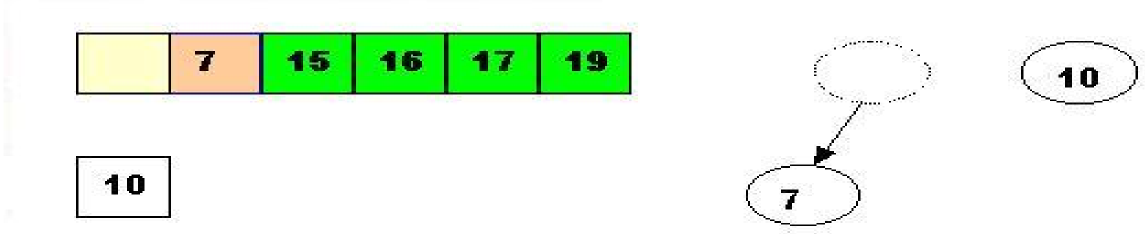


4. DeleteMax the top element 15

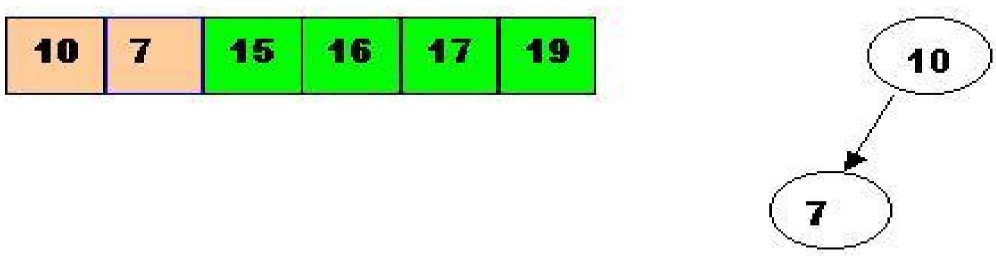
4.1. Store 15 in a temporary location. A hole is created.



4.2. Swap 15 with the last element of the heap. As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a position from the sorted arr

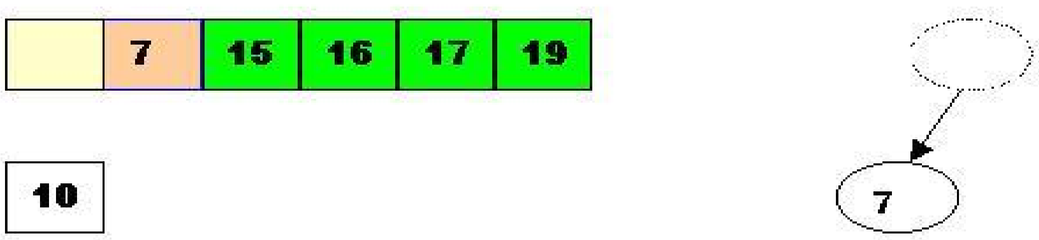


4.3. Store 10 in the hole (10 is greater than the children of the hole)

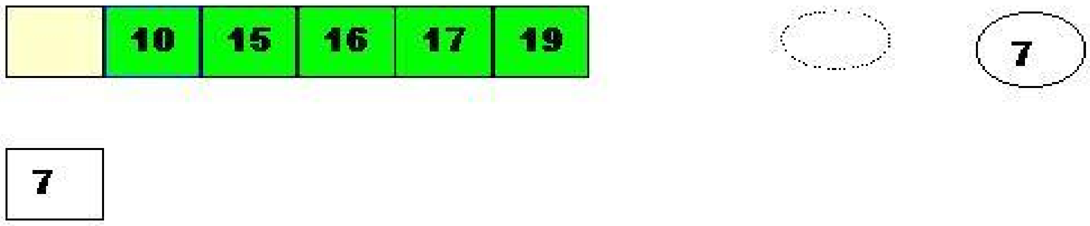


5. DeleteMax the top element 10.

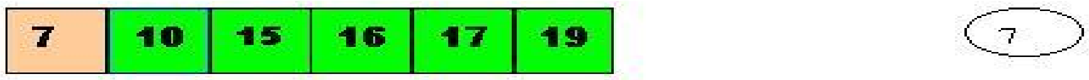
5.1. Remove 10 from the heap and store it into a temporary location.



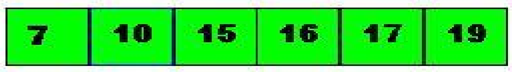
5.2. Swap 10 with the last element of the heap. As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



5.3. Store 7 in the hole (as the only remaining element in the heap



7 is the last element from the heap, so now the array is sorted



INSERTION SORT:

Tree Insertion Sort:

- This is inserting into a normal tree structure, i.e. data are put into the correct positionwhen they are inserted.
- Requires a find and an insert.
- The time complexity for one insert is O(*logN*) + O(1) = O(*logN*);
- Therefore to insert *N* items will have a complexity of O(*NlogN*).

Array Insertion Sort:

The array must be sorted; insertion requires a find + insert
To insert *N* items will have a complexity of O(*N2*)

The following table shows the steps for sorting the sequence 5 7 0 3 4 2 6 1. On the left side the sorted part of the sequence is shown in red. For each iteration, the number of positions the inserted element has moved is shown in brackets.

5	7	0	3	4	2	6	1	(0)
5	7	0	3	4	2	6	1	(0)
0	5	7	3	4	2	6	1	(2)
0	3	5	7	4	2	6	1	(2)
0	3	4	5	7	2	6	1	(2)
0	2	3	4	5	7	6	1	(4)
0	2	3	4	5	6	7	1	(1)
0	1	2	3	4	5	6	7	(6)

SHELL SORT:

Named after D.L. Shell! But it is also rather like shrinkingshells of sorting, for Insertion Sort.

Shell sort aims to reduce the work done by insertion sort (i.e. scanning a list and inserting into the right position).

Do the following:

- Begin by looking at the lists of elements *x*1 elements apart and sort those elements by insertion sort.
- Reduce the number *x*1 to *x*2

We now slice the array into a different number of slices. For this example we will use five slices, but other values are possible.

9	4	1	2	5	6	8	16	7	3	12	14	15	18	19	10	11	13	17	20

Slicing this into five we get:

9	4	1	2
5	6	8	16
7	3	12	14
15	18	19	10
11	13	17	20

Again we sort each column to give:

5	3	1	2
7	4	8	10
9	6	12	14
11	13	17	16
15	18	19	20

Logically reassembling, we now have the dataset:

5	3	1	2	7	4	8	10	9	6	12	14	11	13	17	16	15	18	19	20

SELECTION SORT:

The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

For example, consider the following array, shown with array elements in sequence separated by commas:

63, 75, 90, 12, 27

The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5). The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in bold. We then swap the element at index 2 with that at index 4. The result is:

63, 75, 27, 12, 90

We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (in bold):

63, 12, 27, 75, 90

The next two steps give us:

27, 12, 63, 75, 90

12, 27, 63, 75, 90

COMPARISONS

In this table, *n* is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. These are all comparison sorts. The run time and the memory of algorithms could be measured using various notations like theta, omega, Big-O, small-o, etc. The memory and the run times below are applicable for all the 5 notations.

Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes	Insertion	When using a self-balancing binary search tree
Bogosort	n	$n \cdot n!$	$n \cdot n! \to$	1	No	Luck	Randomly permute the array and check if sorted.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging	
Comb sort	n	$n \log n$	n^2	1	No	Exchanging	Small code size
Cycle sort	—	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Heapsort	$n \log$	$n \log n$	$n \log n$	1	No	Selection	
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes	Merging	Implemented in Standard Template Library (STL); can be

Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
							implemented as a stable sort based on stable in-place merging:
Insertion sort	n	n^2	n^2	1	Yes	Insertion	O($n + d$), where d is the number of inversions
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in SGI STL implementations
Library sort	—	$n \log n$	n^2	n	Yes	Insertion	
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends; worst case is n	Yes	Merging	Used to sort this table in Firefox [2].
Patience sorting	—	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences within O(n log n)
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends	Partitioning	Quicksort is usually done in place with O(log(n)) stack space.[*citation needed*] Most implementations are unstable, as stable in-place partitioning is more complex. Naïve variants use an O(n) space array to store the partition.[*citation needed*]

Comparison sorts							
Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, for example using lists [5]. Used to sort this table in Safari or other Webkit web browser.
Shell sort	n	$n(\log n)$ or $n^{3/2}$	Depends on gap sequence; best known is $n(\log n)^2$	1	No	Insertion	
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive sort - n comparisons when the data is already sorted, and 0 swaps.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	n comparisons when the data is already sorted or reverse sorted.
Tournament sort	—	$n \log n$	$n \log n$			Selection	

The following table describes integer sorting algorithms and other sorting algorithms that are not comparison sorts. As such, they are not limited by a $\Omega(n \log n)$lower bound. Complexities below are in terms of *n*, the number of items to be sorted, *k*, the size of each key, and *d*, the digit size used by the implementation. Many of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that $n << 2^k$, where << means "much less than."

Non-comparison sorts							
Name	**Best**	**Average**	**Worst**	**Memory**	**Stable**	**$n << 2^k$**	**Notes**
Pigeonhole sort	—	$n + 2^k$	$n + 2^k$	2^k	Yes	Yes	
Bucket sort (uniform keys)	—	$n + k$	$n^2 \cdot k$	$n \cdot k$	Yes	No	**Assumes uniform distribution of elements from the domain in the array.**
Bucket sort (integer keys)	—	$n + r$	$n + r$	$n + r$	Yes	Yes	**r is the range of numbers to be sorted. If r = $\mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$**
Counting sort	—	$n + r$	$n + r$	$n + r$	Yes	Yes	**r is the range of numbers to be sorted. If r = $\mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$**
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	n	Yes	No	[3][2]
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + \frac{k}{d} \cdot 2^d$	Yes	No	**Stable version uses an external array of size n to hold all of the bins**
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$\frac{k}{d} \cdot 2^d$	No	No	In-Place. k / d recursion levels, 2^d for count array
Spreadsort	—	$n \cdot \frac{k}{d}$	$n \cdot \left(\frac{k}{s} + d\right)$	$\frac{k}{d} \cdot 2^d$	No	No	Asymptotics are based on the assumption that $n << 2^k$, but the algorithm does not require this.

CONCLUSION

By analyzing an algorithm, we mean to study the performance of an algorithm including the assertion of its correctness and a determination of the cost of its execution. Although a given algorithm is often analyzed in a particular way that is most suitable for such an algorithm, we are more interested in general procedures and techniques that can be used to study the performance of classes of algorithms. To be able to talk about general analysis techniques will not only add to our understanding of the behavior of a class of algorithms but will also, in many cases, lead to useful synthesis procedures. A good example illustrating these points is the various techniques that can be used to analyze a class of sorting algorithms which can be modelled as networks made up of comparator modules. In this paper, we discuss several approaches to such an analysis problem. Moreover, synthesis procedures suggested by these analysis techniques will also be presented.