Sorting & Order Statistics

Shell Sort:

Shell Sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shell Sort is to allow exchange of far items. In shell Sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

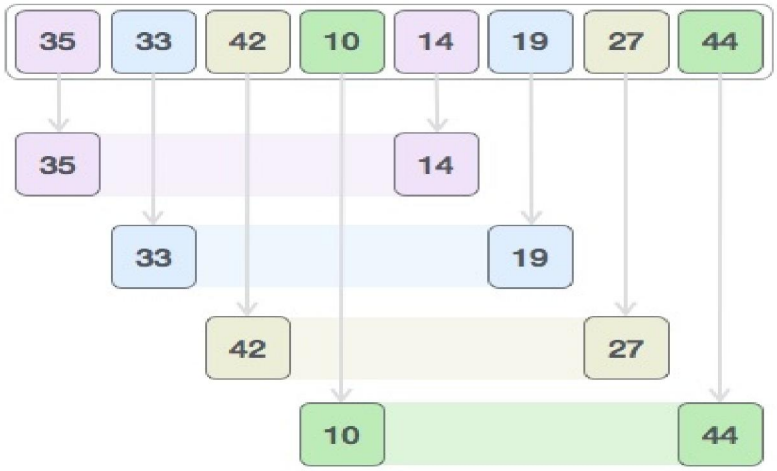
Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as –

Knuth's Formula
h = h * 3 + 1
where –h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

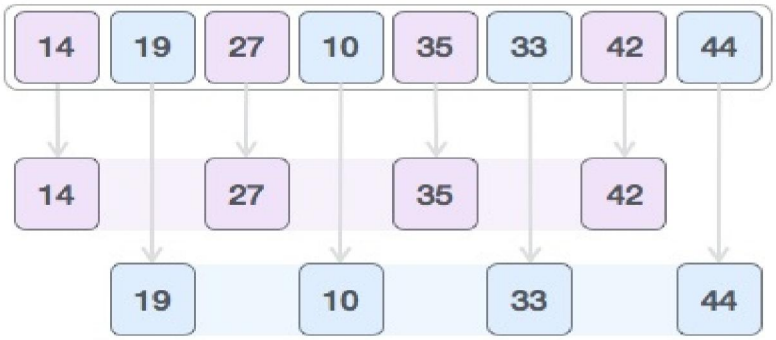
How Shell Sort Works?

Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}

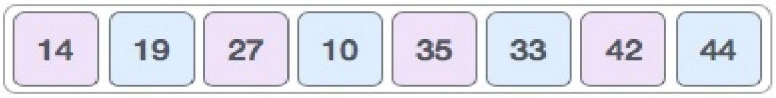


We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this –

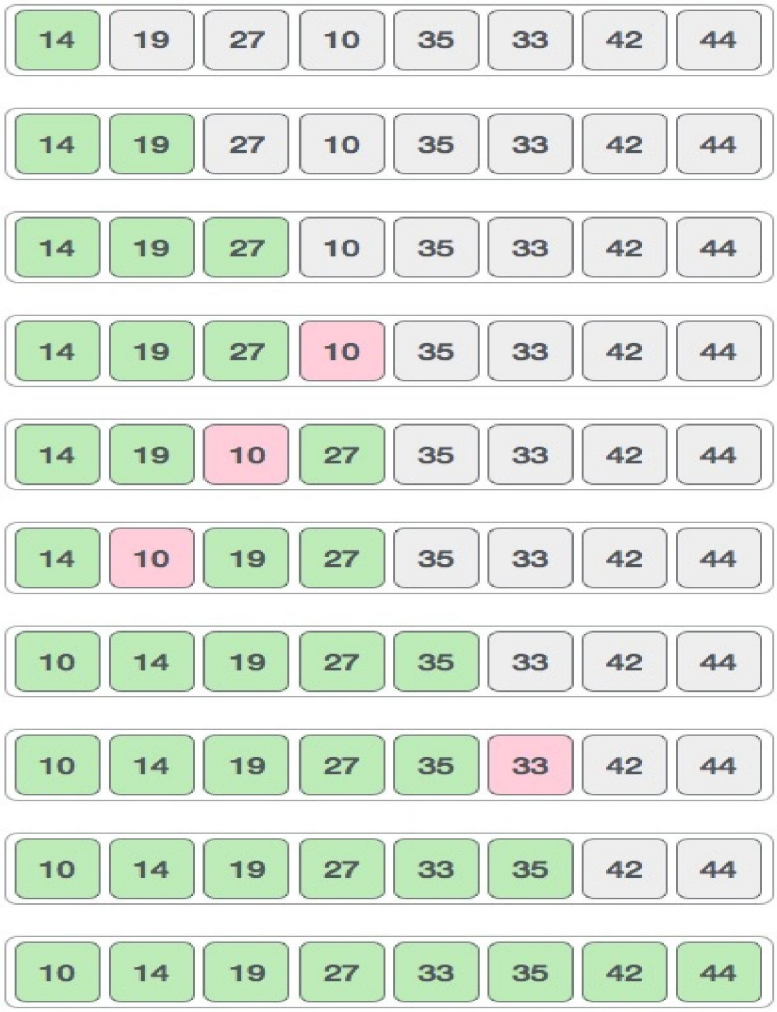
Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this –



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.
Following is the step-by-step depiction –



We see that it required only four swaps to sort the rest of the array.
Algorithm
Following is the algorithm for shell sort.
Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval *h*
Step 3 − Sort these sub-lists using insertion sort
Step 3 − Repeat until complete list is sorted.

Merge Sort:

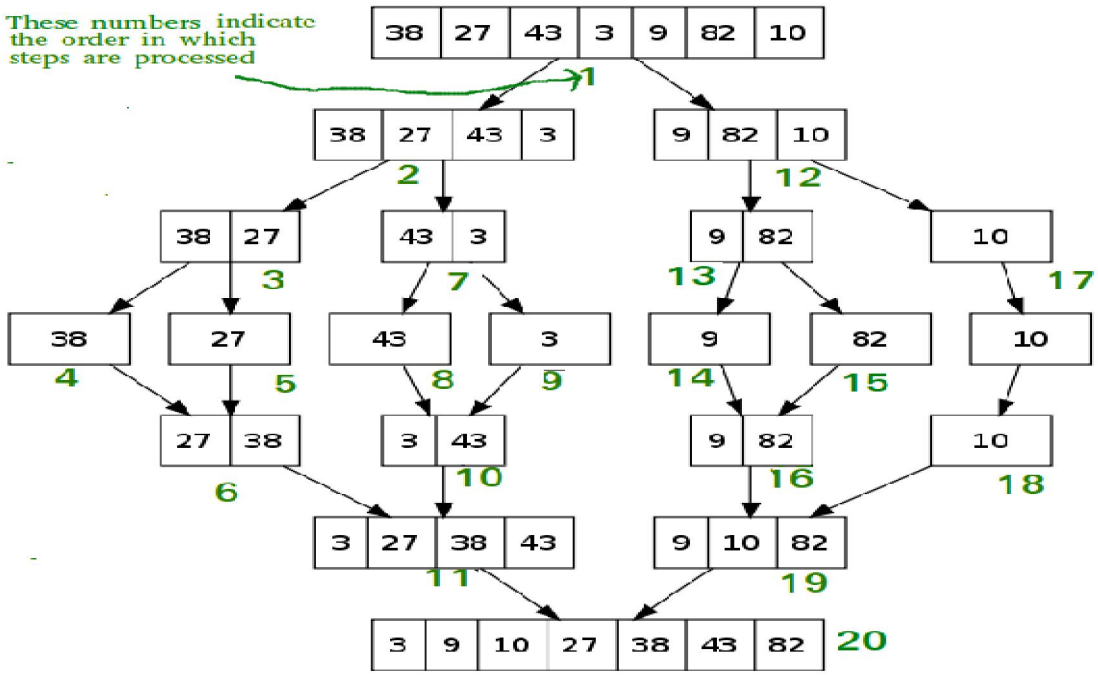
Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

MergeSort(arr[], l, r)

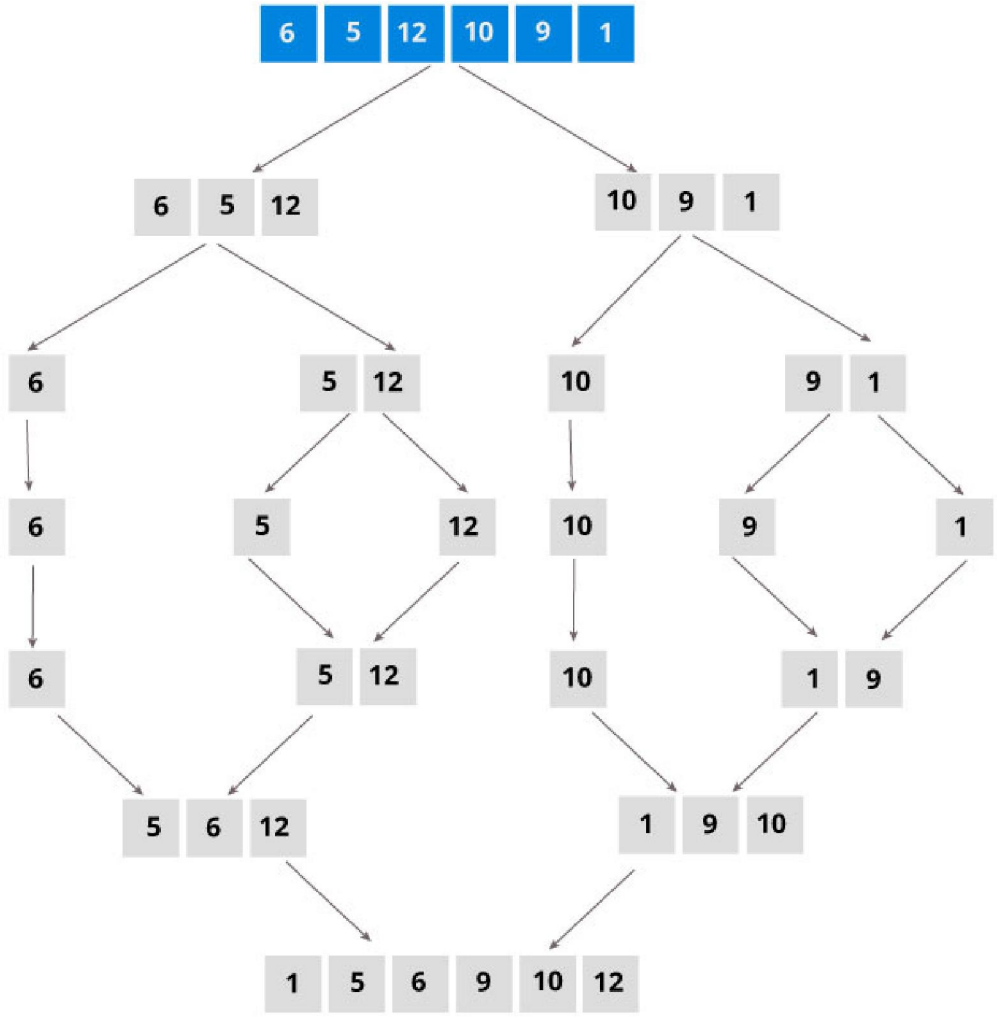
If r > l

1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



It is a great way to develop confidence in building recursive algorithms.



Divide and Conquer Strategy:

Using the Divide and Conquer technique, we divide a problem into sub problems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.
Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r]

The Merge Sort Algorithm:

The Merge Sort function repeatedly divides the array into two halves until we reach a stage where we try to perform Merge Sort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

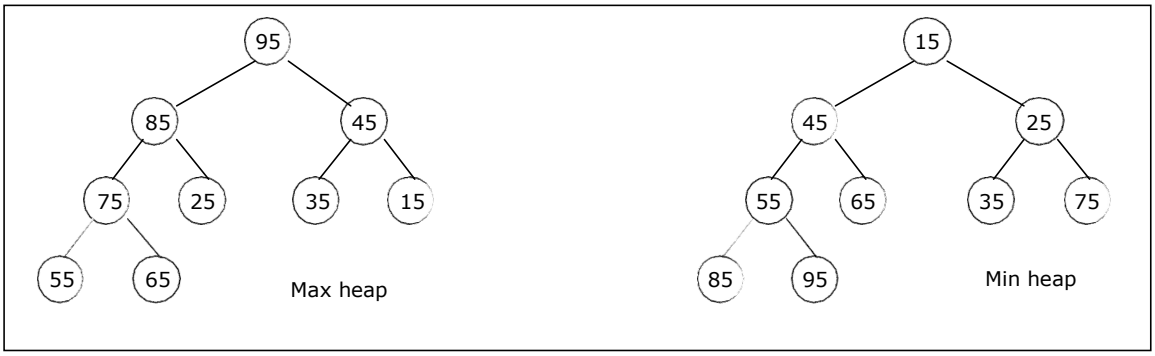
```
MergeSort(A, p, r)

If p > r
    return;
q = (p+r)/2;
mergeSort(A, p, q)
mergeSort(A, q+1, r)
merge(A, p, q, r)
```

Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children. Figure 2.1 shows the maximum and minimum heap tree.

Representation of Heap Tree:

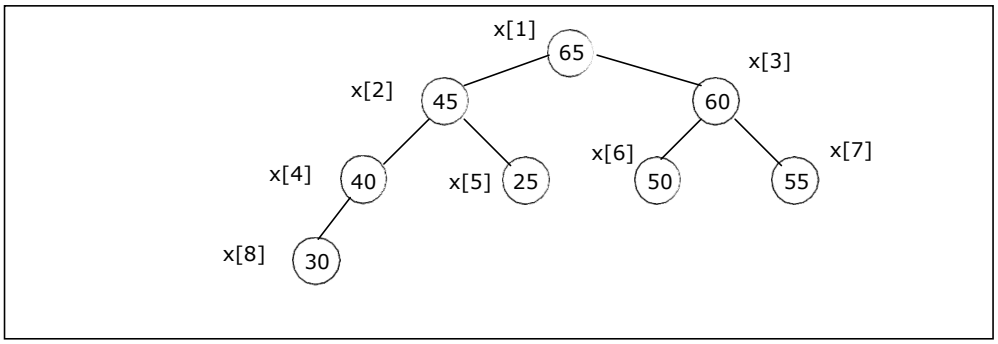
Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2*i + 1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

For example let us consider the following elements arranged in the form of array as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:



Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it

requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

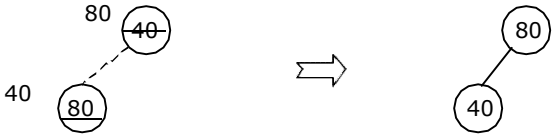
```
Max_heap_insert (a, n)
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
        integer i, n;
        i = n;
        item = a[n]  ;
        while ( (i > 1) and (a[⌊ i/2 ⌋] < item ) do
        {
                a[i] = a[⌊ i/2 ⌋ ] ;                    // move the parent down
                i =⌊ i/2 ⌋;
        }
        a[i] = item ;
        return true ;
}
```

Example:

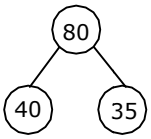
Form a heap by using the above algorithm for the given data 40, 80, 35, 90, 45, 50, 70.

1. Insert 40:

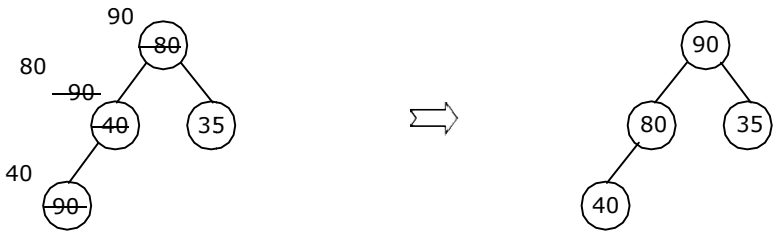
2. Insert 80:



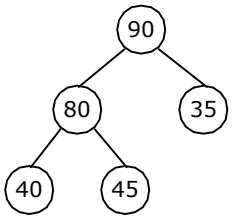
3. Insert 35:



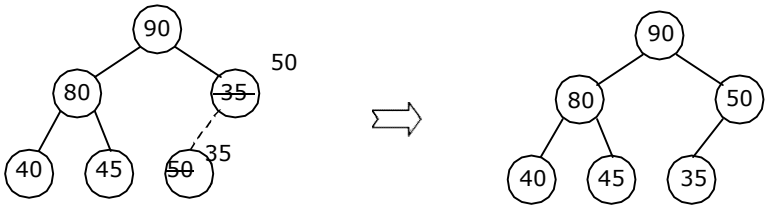
4. Insert 90:



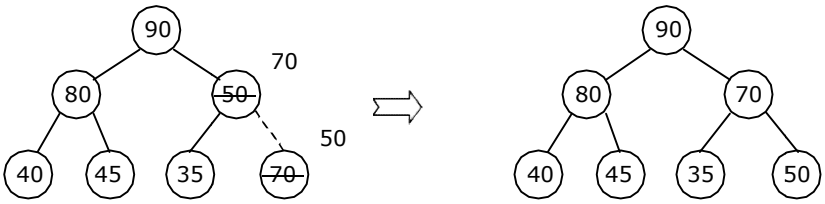
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
  - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

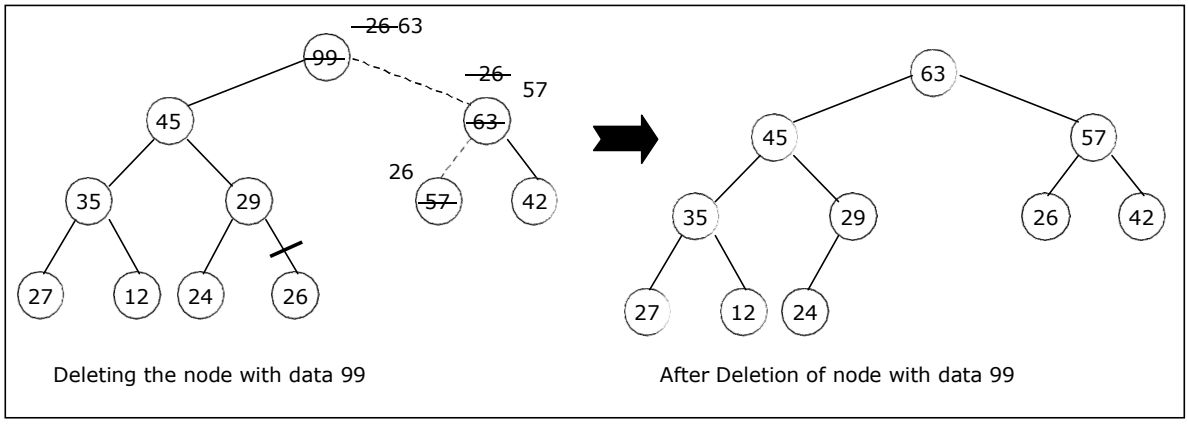
delmax (a, n, x)

```
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
        return true;
}
```

adjust (a, i, n)

```
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form a
single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                        // a position for item is found
                else a[⌊j / 2⌋] = a[j] // move the larger child up a level
                j = 2 * j;
        }
        a [⌊j / 2⌋] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appear as the leave node, hence re-heap is completed. This is shown in figure 2.3.



Deleting the node with data 99

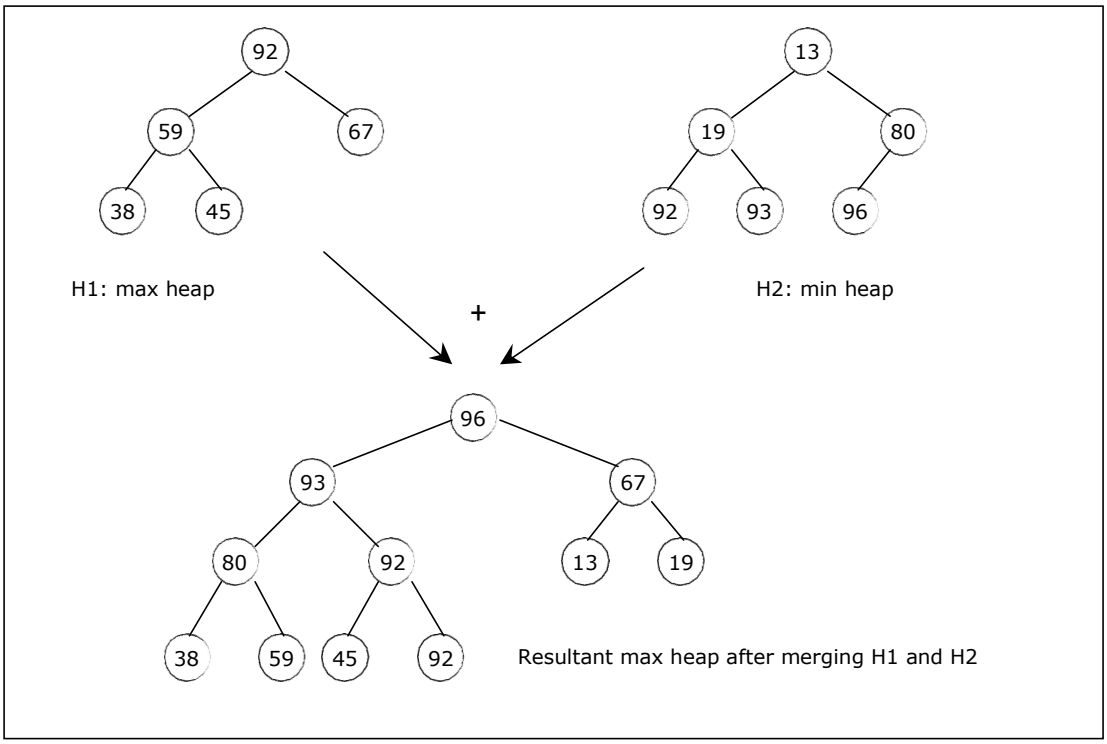
After Deletion of node with data 99

Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap.

Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



Applications of heap tree:

They are two main applications of heap trees known:

1. Sorting (Heap sort) and
2. Priority queue implementation.

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.
   b. Re-heapify the complete binary tree.
   c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp = a[I];
                a[i]  =  a[1];
                a[1] = t;
                adjust (a, 1, i – 1);
        }
}
```

heapify (a, n)

```
//Readjust the elements in a[n] to form a heap.
{
        for i ← ⌊ n/2 ⌋ to 1 by – 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

```
// The complete binary trees with roots a(2*i) and a(2*i+1) are combined
// with a(i) to form a single heap, 1≤i≤n. No node has an address greater
// than n or less than 1.
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                        // a position for item is found
                else a[ ⌊ j / 2 ⌋ ] = a[j] // move the larger child up a level
                j = 2 * j;
        }
        a [ ⌊ j / 2 ⌋ ] = item;
}
```

Time Complexity:

Each 'n' insertion operations takes O(log k), where 'k' is the number of elements in the heap at the time. Likewise, each of the 'n' remove operations also runs in time O(log k), where 'k' is the number of elements in the heap at the time. Since we always have $k \leq n$, each such operation runs in O(log n) time in the worst case.

Thus, for *n* elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on.

As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer 'up' until a[up] >= pivot.
2. Repeatedly decrease the pointer 'down' until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]

4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of ***O(n log n)***.

Algorithm

Sorts the elements a[p], ... ,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = +

```
quicksort (p, q)
{
if ( p < q ) then
{
call j = PARTITION(a, p, q+1);                    // j is the position of the partitioning element
call quicksort(p, j – 1);
call quicksort(j + 1 , q);
}
}

partition(a, m, p)
{
        v = a[m]; up = m; down = p;     // a[m] is the partition element
        Do
        {
        Repeat
        up = up + 1;
        until (a[up] > v);

repeat
down = down – 1;
until (a[down] ≤ v);
if (up < down) then call interchange(a, up, down); } while (up ≥
down);

a[m] = a[down];
a[down] = v;
return (down);
}
```

```
interchange(a, up, down)
{
p = a[up];
a[up] = a[down];
a[down] = p;
}
```

Example:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				Up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		Down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, Up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45**)**	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

7.5.1. Recursive program for Quick Sort:

```
 # include<stdio.h>
 # include<conio.h>

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];

int main()
{
int num, i = 0;
clrscr();
printf( "Enter the number of elements: " );
scanf( "%d", &num);
printf( "Enter the elements: " );
for(i=0; i < num; i++)
scanf( "%d", &array[i] );
quicksort(0, num -1);
printf( "\nThe elements after sorting are: " );
```

```
for(i=0; i < num; i++)
printf("%d ", array[i]);
return 0;
}

void quicksort(int low, int high)
{
int pivotpos;
if( low < high )
{
pivotpos = partition(low, high + 1);
quicksort(low, pivotpos - 1);
quicksort(pivotpos + 1, high);
}
}

int partition(int low, int high)
{
int pivot = array[low];
int up = low, down = high;

do
{
do
up = up + 1;
while(array[up] < pivot );

do
down = down - 1;
while(array[down] > pivot);

if(up < down)
interchange(up, down);

} while(up < down); array[low] =
array[down]; array[down] = pivot;
return down;
}

void interchange(int i, int j)
{
int temp;
temp = array[i];
array[i] = array[j];
array[j] = temp;
}
```

Comparing different Sorting Algorithms:

ShellSort:

The shell sort is by far the fastest of the class of sorting algorithms. It is more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

HeapSort:

It is the slowest of the sorting algorithms but unlike merge and quick sort it does not require massive recursion or multiple arrays to work

Merge Sort:

The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array.

Quick Sort:

The quick sort is an in-place, divide-and-conquer, massively recursive sot. It can be said as the faster version of the merge sort. The efficiency of the algorithm is majorly impacted by which element is chosen as the pivot point. The worst-case efficiency of the quick sort is when the list is sorted and left most element is chosen as the pivot. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of O(nlog n).

BASIS FOR COMPARISON	QUICK SORT	MERGE SORT
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	The splitting of a array of elements is in any ratio, not necessarily divided into half.
Worst case complexity	O(n2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor

Linear Time Sorting:

Merge Sort and Heap Sort achieve this upper bound in the worst case, and Quick Sort achieves this on Average Case.

Merge Sort, Quick Sort and Heap Sort algorithm share an interesting property: the sorted order they determined is based only on comparisons between the input elements. We call such a sorting algorithm **"Comparison Sort"**.

There is some algorithm that runs faster and takes linear time such as Counting Sort, Radix Sort, and Bucket Sort but they require the special assumption about the input sequence to sort.

Counting Sort and Radix Sort assumes that the input consists of an integer in a small range.

Bucket Sort assumes that a random process that distributes elements uniformly over the interval generates the input.