Introduction to Sorting

Sorting any sequence means to arrange the elements of that sequence according to some specific criterion.

For Example, the array arr[] = {5, 4, 2, 1, 3} after *sorting in increasing order* will be: arr[] = {1, 2, 3, 4, 5}. The same array after *sorting in descending order* will be: arr[] = {5, 4, 3, 2, 1}.

In-Place Sorting: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

In this tutorial, we will see three of such in-place sorting algorithms, namely:

- Insertion Sort
- Selection Sort
- Bubble Sort

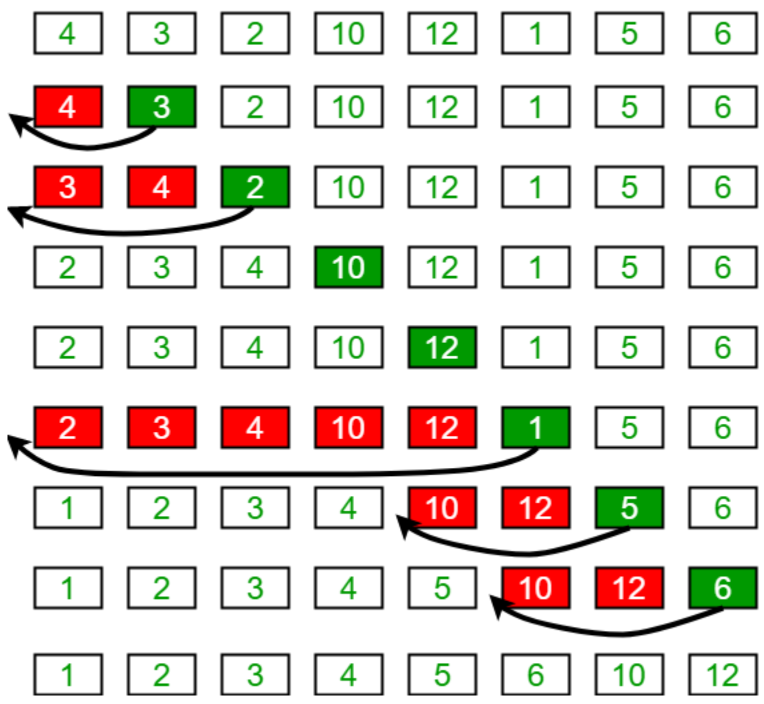
Insertion Sort

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.

The idea is to start iterating from the second element of array till last element and for every element insert at its correct position in the subarray before it.

In the below image you can see, how the array [4, 3, 2, 10, 12, 1, 5, 6] is being sorted in increasing order following the insertion sort algorithm.

Insertion Sort Execution Example



Algorithm:

```
Step 1: If the current element is 1st element of array,
        it is already sorted.
Step 2: Pick next element
Step 3: Compare the current element will all elements
        in the sorted sub-array before it.
Step 4: Shift all of the elements in the sub-array before
        the current element which are greater than the current
        element by one place and insert the current element
        at the new empty space.
Step 5: Repeat step 2-3 until the entire array is sorted.
```

Another Example:
arr[] = {**12**, 11, 13, 5, 6}

Let us loop for i = 1 (second element of the array) to 4 (Size of input array - 1).

- *i = 1*, Since 11 is smaller than 12, move 12 and insert 11 before 12.
  11, 12, 13, 5, 6
- *i = 2*, 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
  11, 12, 13, 5, 6
- *i = 3*, 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
  5, 11, 12, 13, 6
- *i = 4*, 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
  5, 6, 11, 12, 13

Function Implementation:

```

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;

       /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
```

```
            j = j-1;
        }
        arr[j+1] = key;
    }
}

```

Time Complexity: O(N^2), where N is the size of the array.

Bubble Sort

Bubble Sort is also an in-place sorting algorithm. This is the simplest sorting algorithm and it works on the principle that:

> In one iteration if we swap all adjacent elements of an array such that after swap the first element is less than the second element then at the end of the iteration, the first element of the array will be the minimum element.

Bubble-Sort algorithm simply repeats the above steps N-1 times, where N is the size of the array.

Example: Consider the array, arr[] = {5, 1, 4, 2, 8}.

- **First Pass:** (**5 1** 4 2 8) --> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  (1 **5 4** 2 8) --> (1 **4 5** 2 8), Swap since 5 > 4
  (1 4 **5 2** 8) --> (1 4 **2 5** 8), Swap since 5 > 2
  (1 4 2 **5 8**) --> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
- **Second Pass:** (**1 4** 2 5 8) --> (**1 4** 2 5 8)
  (1 **4 2** 5 8) --> (1 **2 4** 5 8), Swap since 4 > 2
  (1 2 **4 5** 8) --> (1 2 **4 5** 8)
  (1 2 4 **5 8**) --> (1 2 4 **5 8**)
  Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
- **Third Pass:** (**1 2** 4 5 8) --> (**1 2** 4 5 8)
  (1 **2 4** 5 8) --> (1 **2 4** 5 8)
  (1 2 **4 5** 8) --> (1 2 **4 5** 8)
  (1 2 4 **5 8**) --> (1 2 4 **5 8**)

Function Implementation:

```

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}

```

Note: The above solution can be further optimized by keeping a flag to check if the array is already sorted in the first pass itself and to stop any further iteration.

Time Complexity: O(N^2)

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

```
arr[] = 64 25 12 22 11.

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Function Implementation:

```

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

```

Time Complexity: O(N^2)

- Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

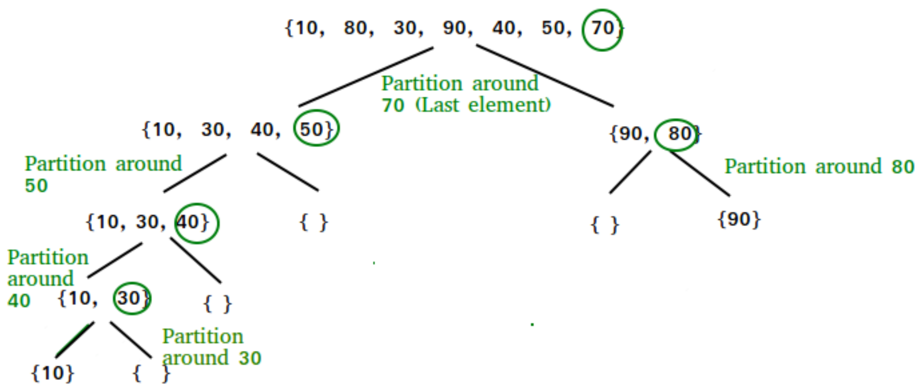
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2014/01/QuickSort2.png)

Partition Algorithm:

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }

    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes:  0   1   2   3   4   5   6



low = 0, high =  6, pivot = arr[h] = 70

Initialize index of smaller element, i = -1


Traverse elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j

                                     // are same



j = 1 : Since arr[j] > pivot, do nothing

// No change in i and arr[]



j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30



j = 3 : Since arr[j] > pivot, do nothing

// No change in i and arr[]



j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped



We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping

arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped



Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

it.
```

Implementation:

```

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
   arr[] --> Array to be sorted,
   low  --> Starting index,
   high  --> Ending index */
```

Run

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

```
T(n) = T(k) + T(n-k-1) + θ(n)
```

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

```
 T(n) = T(0) + T(n-1) + θ(n)
which is equivalent to
 T(n) = T(n-1) + θ(n)
```

The solution of above recurrence is $\Theta(n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

```
 T(n) = 2T(n/2) + θ(n)
```

The solution of above recurrence is **Θ(nLogn)**. It can be solved using case 2 of Master Theorem (http://en.wikipedia.org/wiki/Master_theorem).

Average Case: To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

```
 T(n) = T(n/9) + T(9n/10) + θ(n)
```

Solution of above recurrence is also O(nLogn)

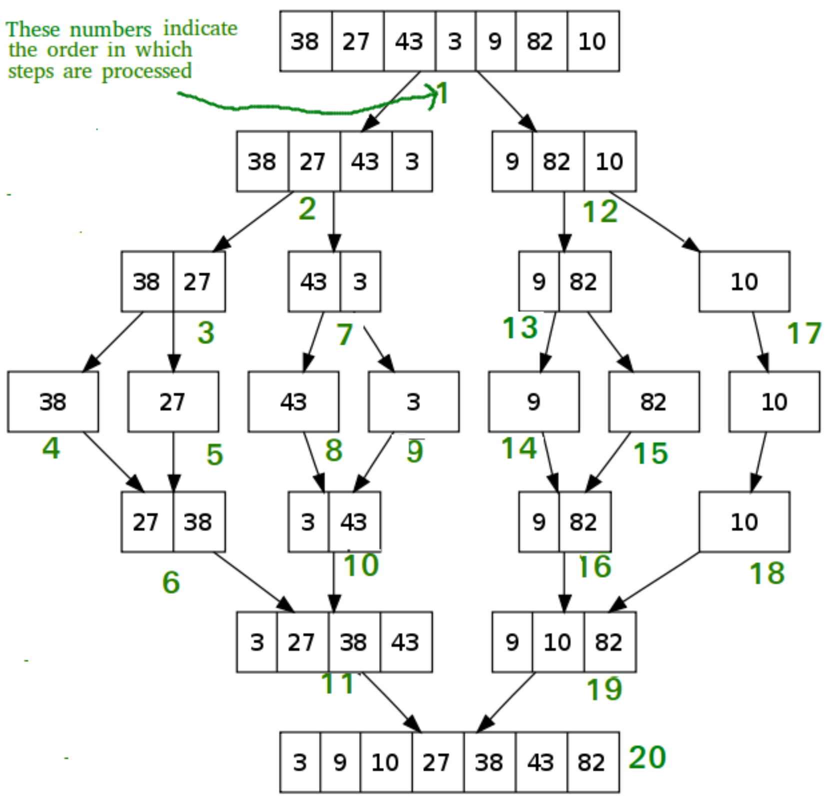
Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like **Merge Sort** and **Heap Sort**, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

- Merge Sort

Merge Sort is a Divide and Conquer (https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides the 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 in a sorted manner. See following 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 from wikipedia (http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg) 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.



Implementation:

```
        j = 0; // Initial index of second subarray
        k = l; // Initial index of merged subarray
        while (i < n1 && j < n2)
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }
            else
            {
                arr[k] = R[j];
                j++;
            }
            k++;
```

```
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
```

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \Theta(n)$
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(nLogn)$.

Time complexity of Merge Sort is **Θ(nLogn)** in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

- Counting Sort

Counting sort (http://en.wikipedia.org/wiki/Counting_sort) is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0   1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
the output sequence.

  3) Output each object from the input sequence followed by
  decreasing its count by 1.
  Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
  Put data 1 at index 2 in output. Decrease count by 1 to place
  next data 1 at an index 1 smaller than this index.
```

Implementation:

```

// The main function that sort the given string arr[] in
// alphabetical order
void countSort(char arr[])
{
    // The output character array that will have sorted arr
    char output[strlen(arr)];

    // Create a count array to store count of inidividul
    // characters and initialize count array as 0
    int count[RANGE + 1], i;
    memset(count, 0, sizeof(count));

    // Store count of each character
    for(i = 0; arr[i]; ++i)
        ++count[arr[i]];

    // Change count[i] so that count[i] now contains actual
    // position of this character in output array
    for (i = 1; i <= RANGE; ++i)
        count[i] += count[i-1];

    // Build the output character array
    for (i = 0; arr[i]; ++i)
    {
        output[count[arr[i]]-1] = arr[i];
        --count[arr[i]];
    }

    // Copy the output array to arr, so that arr now
```

Time Complexity: O(N + K) where N is the number of elements in input array and K is the range of input.
Auxiliary Space: O(N + K)

The problem with the previous counting sort was that it could not sort the elements if we have negative numbers in the array because there are no negative array indices. So what we can do is, we can find the minimum element and store count of that minimum element at zero index.

Implementation:

```

void countSort(vector <int>& arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int min = *min_element(arr.begin(), arr.end());
    int range = max - min + 1;

    vector<int> count(range), output(arr.size());
    for(int i = 0; i < arr.size(); i++)
        count[arr[i]-min]++;

    for(int i = 1; i < count.size(); i++)
           count[i] += count[i-1];

    for(int i = arr.size()-1; i >= 0; i--)
    {
         output[ count[arr[i]-min] -1 ] = arr[i];
              count[arr[i]-min]--;
    }

    for(int i=0; i < arr.size(); i++)
            arr[i] = output[i];
}

```

Important Points:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. It's running time complexity is O(n) with space proportional to the range of data.
3. It is often used as a sub-routine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).
5. Counting sort can be extended to work for negative inputs also.

- Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining elements.

What is Binary Heap (https://www.geeksforgeeks.org/binary-heap/)?
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia (http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees)).

A Binary Heap (https://www.geeksforgeeks.org/binary-heap/) is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Array based representation for Binary Heap: Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting an array in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: [4, 10, 3, 5, 1]
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)

The heapify procedure calls itself recursively to build heap
in top down manner.
```

Implementation:

```

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function for heap sort
void heapSort(int arr[], int n)
{
```

LIVE BATCHES

Important Notes:

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable (See this (https://www.geeksforgeeks.org/stability-in-sorting-algorithms/)).

Time Complexity: Time complexity of heapify is O(N*LogN). Time complexity of createAndBuildHeap() is O(N) and overall time complexity of Heap Sort is **O(N*LogN)** where N is the number of elements in the list or array.

Heap sort algorithm has limited use because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

+ sort() Function in C++ STL

+ Sorting using Built-in methods in Java

Report An Issue

If you are facing any issue on this page. Please let us know.

geeksforgeeks/)

LIVE BATCHES

▲