Heap Sort

What is Heap Sort

__Heap sort__ is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable (See this)
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Advantages of heapsort:

- **Efficiency** – The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
- **Memory Usage** – Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity** – It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

- **Costly**: Heap sort is costly.
- **Unstable**: Heap sort is unstable. It might rearrange the relative order.
- **Efficient:** Heap Sort are not very efficient when working with highly complex data.

Applications of HeapSort:

- Heapsort is mainly used in hybrid algorithms like the IntroSort.
- Sort a nearly sorted (or K sorted) array
- k largest(or smallest) elements in an array

The heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See Applications of Heap Data Structure

What is meant by Heapify?

Try It!

Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the last index of the non-leaf node whose index is given by n/2 – 1. Heapify uses recursion.

Algorithm for Heapify:

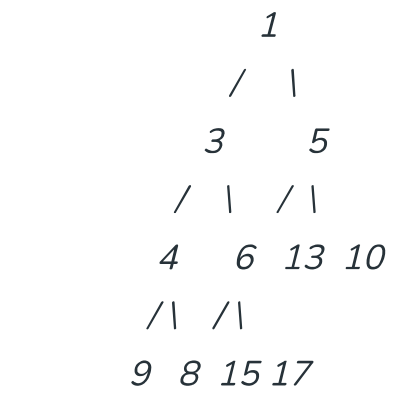
```
heapify(array)
 Root = array[0]

  Largest = largest( array[0] , array [2 * 0 + 1]/ array[2 * 0 + 2])
if(Root != Largest)
 Swap(Root, Largest)
```

How does Heapify work?

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

Corresponding Complete Binary Tree is:



***The task to build a Max-Heap from above array**.*

Total Nodes = 11.

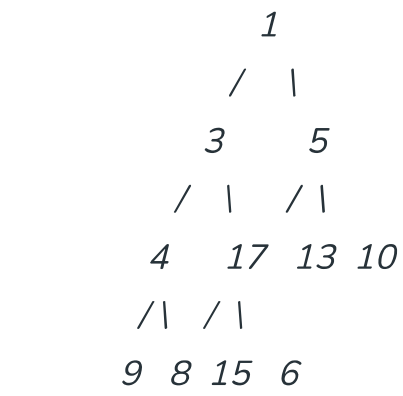
Total non-leaf nodes= (11/2)-1=5

last non-leaf node = 6.

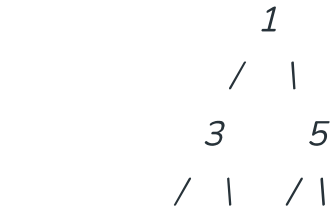
Therefore, Last Non-leaf node index = 4.

To build the heap, heapify only the nodes: [1, 3, 5, 4, 6] in reverse order.

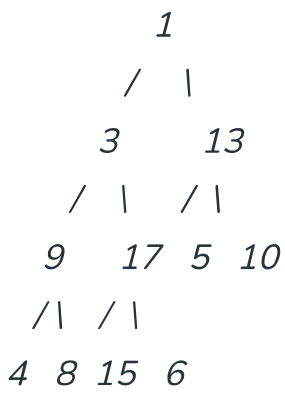
***Heapify 6**: Swap 6 and 17.*



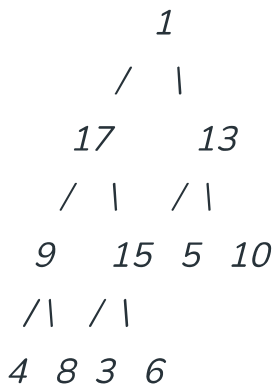
***Heapify 4**: Swap 4 and 9.*



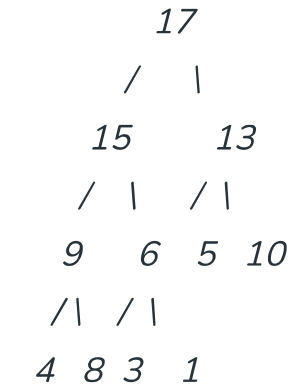
***Heapify 5**: Swap 13 and 5.*



***Heapify 3**: First Swap 3 and 17, again swap 3 and 15.*



***Heapify 1**: First Swap 1 and 17, again swap 1 and 15, finally swap 1 and 6.*



Heap Sort Algorithm

To solve the problem follow the below idea:

First convert the array into heap data structure using heapify, then one by

Build a heap from the given input array.

Repeat the following steps until the heap contains only one element:

a. Swap the root element of the heap (which is the largest element) with the last element of the heap.
b. Remove the last element of the heap (which is now in the correct position).
c. Heapify the remaining elements of the heap.

The sorted array is obtained by reversing the order of the elements in the input array.

Follow the given steps to solve the problem:

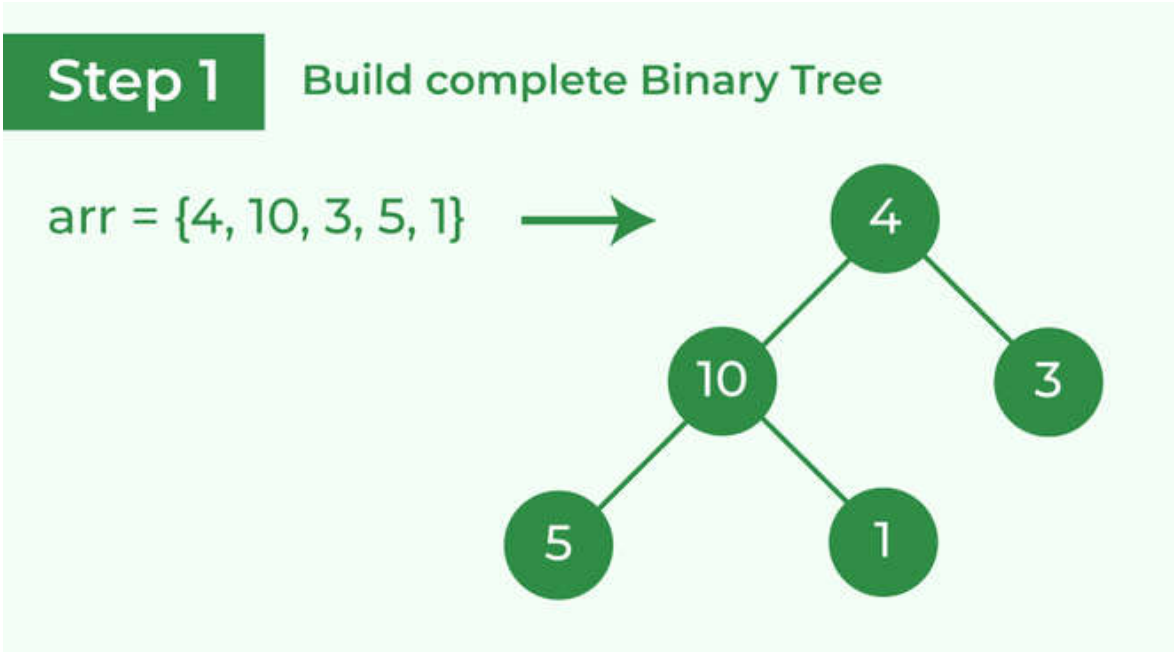
- Build a max heap from the input data.
- At this point, the maximum element is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of the heap by 1. Finally, heapify the root of the tree.
- Repeat step 2 while the size of the heap is greater than 1.

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

Detailed Working of Heap Sort

To understand heap sort more clearly, let's take an unsorted array and try to sort it using heap sort.
Consider the array: arr[] = {4, 10, 3, 5, 1}.

Build Complete Binary Tree: *Build a complete binary tree from the array.*



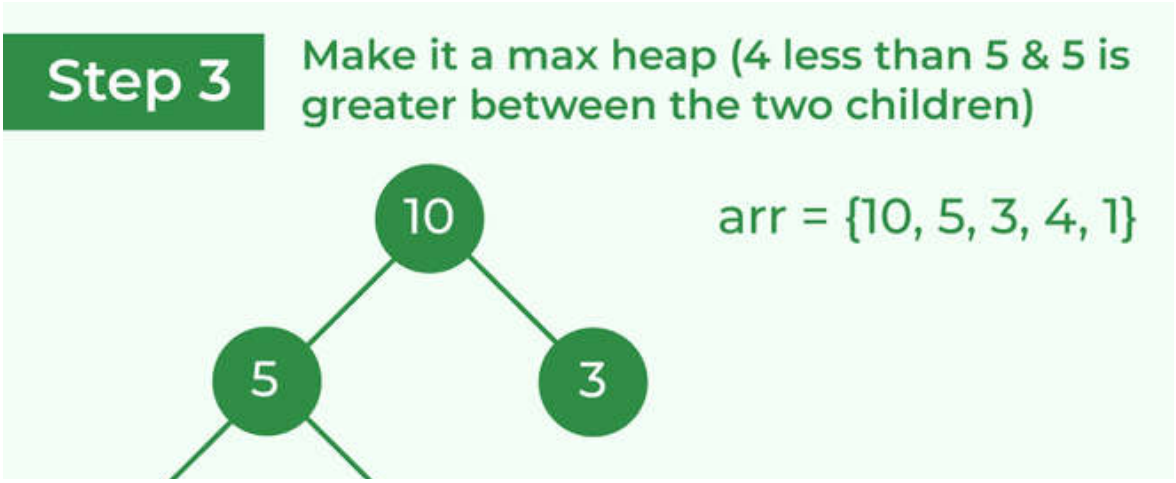
Build complete binary tree from the array

Transform into max heap: *After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.*

- *To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes*
  - *Here, in this example, as the parent node **4** is smaller than the child node **10,** thus, swap them to build a max-heap.*

Transform it into a max heap image widget

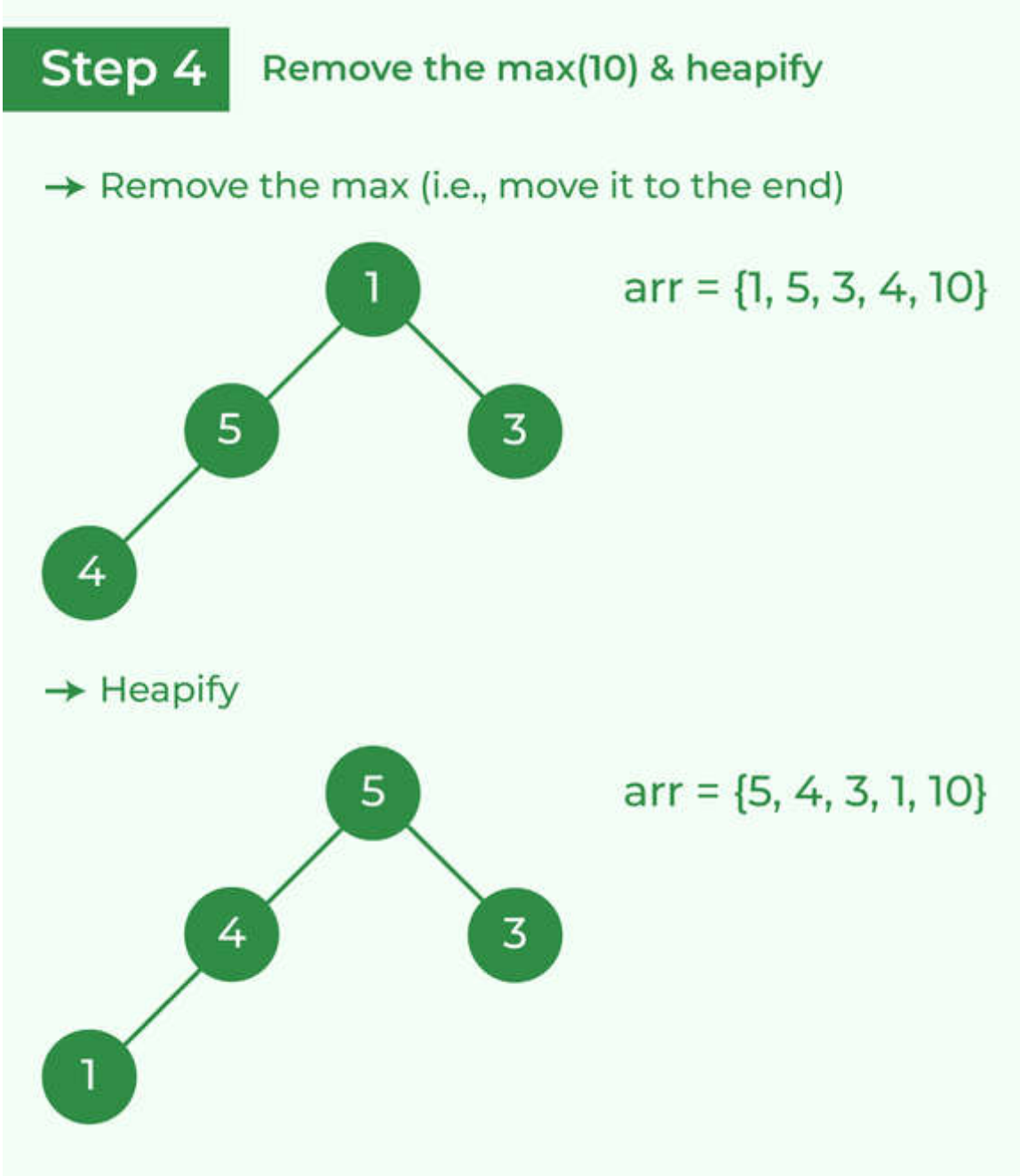
- *Now, as seen, **4** as a parent is smaller than the child **5**, thus swap both of these again and the resulted heap and array should be like this:*

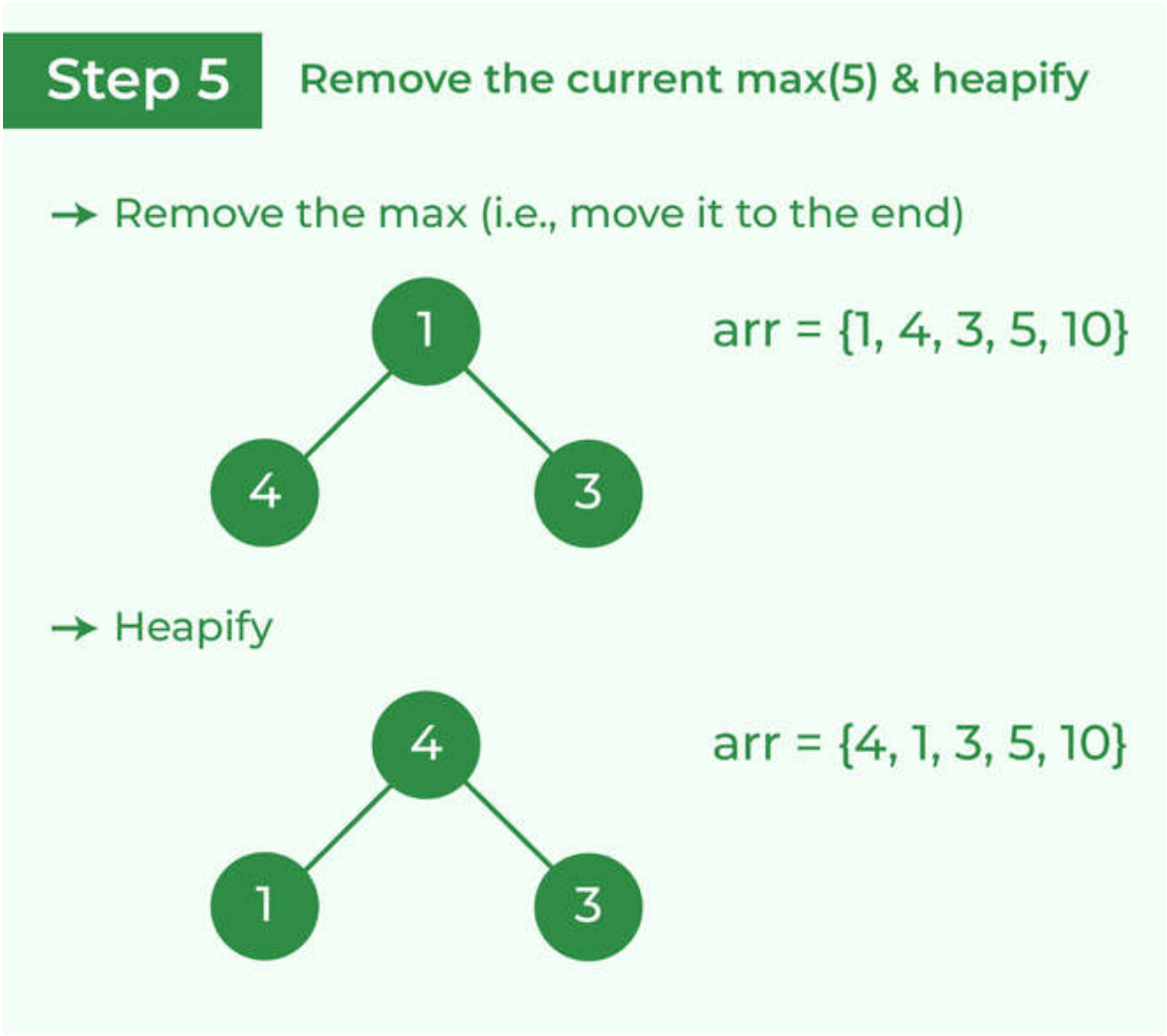


Make the tree a max heap

Perform heap sort: *Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.*

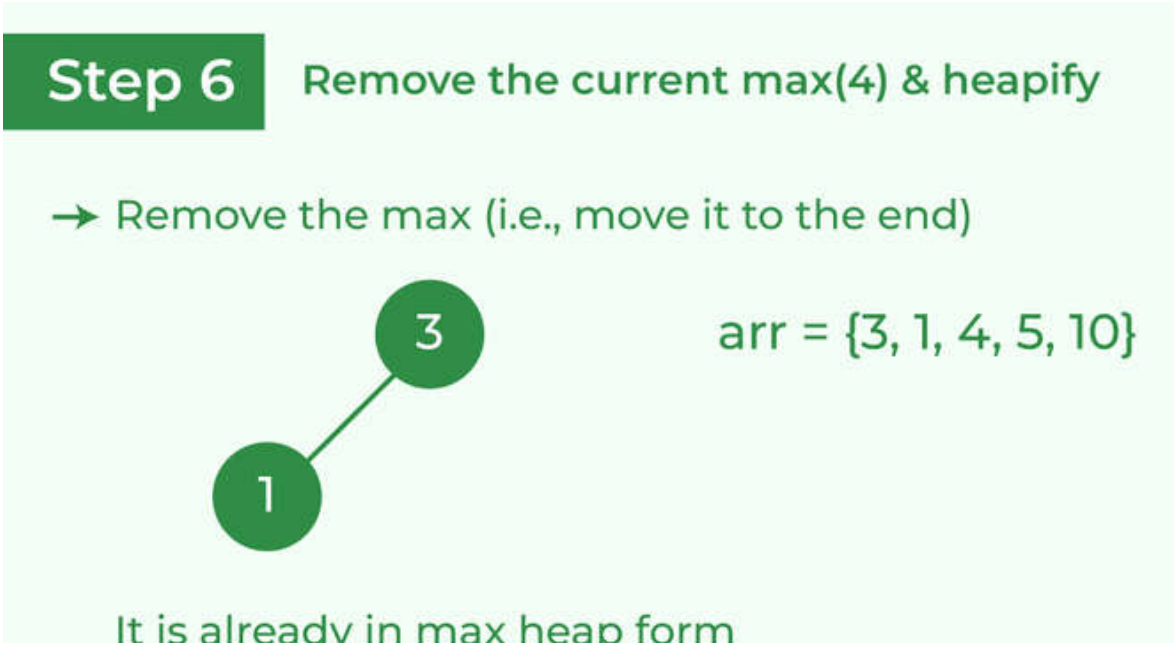
- *Delete the root element* ***(10)*** *from the max heap. In order to delete this node, try to swap it with the last node, i.e.* ***(1).*** *After removing the root element, again heapify it to convert it into max heap.*
  - *Resulted heap and array should look like this:*



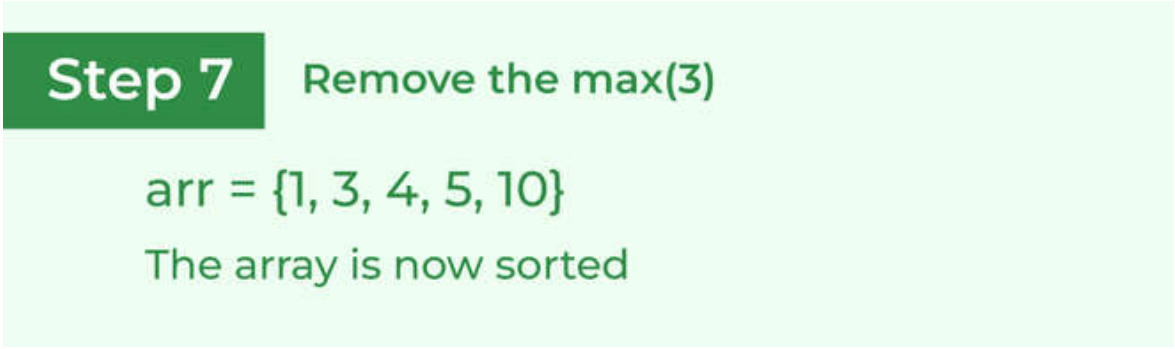


Remove 5 and perform heapify

- *Now remove the root (i.e. 3) again and perform heapify.*



- *Now when the root is removed once again it is sorted. and the sorted array will be like* ***arr[] = {1, 3, 4, 5, 10}****.*



The sorted array

Implementation of Heap Sort

Below is the implementation of the above approach:

C

```
// Heap Sort in C

#include <stdio.h>

// Function to swap the position of two elements

void swap(int* a, int* b)
{

    int temp = *a;

    *a = *b;

    *b = temp;
}

// 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)
{
    // Find largest among root, left child and right child
```

```
    int left = 2 * i + 1;

    // right = 2*i + 2
    int right = 2 * i + 2;

    // If left child is larger than root
    if (left < N && arr[left] > arr[largest])

        largest = left;

    // If right child is larger than largest
    // so far
    if (right < N && arr[right] > arr[largest])

        largest = right;

    // Swap and continue heapifying if root is not largest
    // If largest is not root
    if (largest != i) {

        swap(&arr[i], &arr[largest]);

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

// Main function to do heap sort
void heapSort(int arr[], int N)
{

    // Build max heap
    for (int i = N / 2 - 1; i >= 0; i--)

        heapify(arr, N, i);

    // Heap sort
    for (int i = N - 1; i >= 0; i--) {

        swap(&arr[0], &arr[i]);

        // Heapify root element to get highest element at
        // root again
        heapify(arr, i, 0);
    }
}
```

```
    printf("\n");
}

// Driver's code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    heapSort(arr, N);
    printf("Sorted array is\n");
    printArray(arr, N);
}

// This code is contributed by _i_plus_plus_.
```

C++

```
// C++ program for implementation of Heap Sort

#include <iostream>
using namespace std;

// 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)
{

    // Initialize largest as root
    int largest = i;

    // left = 2*i + 1
    int l = 2 * i + 1;

    // right = 2*i + 2
    int r = 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;
```

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

// Main function to do heap sort
void heapSort(int arr[], int N)
{

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element
    // from heap
    for (int i = N - 1; i > 0; i--) {

        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// A utility function to print array of size n
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver's code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    heapSort(arr, N);

    cout << "Sorted array is \n";
    printArray(arr, N);
}
```

```
public class HeapSort {
    public void sort(int arr[])
    {
        int N = arr.length;

        // Build heap (rearrange array)
        for (int i = N / 2 - 1; i >= 0; i--)
            heapify(arr, N, i);

        // One by one extract an element from heap
        for (int i = N - 1; i > 0; i--) {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // 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) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

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

```
        for (int i = 0; i < N; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver's code
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int N = arr.length;

        // Function call
        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

Python3

```
# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap


def heapify(arr, N, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < N and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < N and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap
```

```
# The main function to sort an array of given size


def heapSort(arr):
    N = len(arr)

    # Build a maxheap.
    for i in range(N//2 - 1, -1, -1):
        heapify(arr, N, i)

    # One by one extract elements
    for i in range(N-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)


# Driver's code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]

    # Function call
    heapSort(arr)
    N = len(arr)

    print("Sorted array is")
    for i in range(N):
        print("%d" % arr[i], end=" ")
# This code is contributed by Mohit Kumra
```

C#

```
// C# program for implementation of Heap Sort
using System;

public class HeapSort {
    public void sort(int[] arr)
    {
        int N = arr.Length;

        // Build heap (rearrange array)
        for (int i = N / 2 - 1; i >= 0; i--)
            heapify(arr, N, i);

        // One by one extract an element from heap
        for (int i = N - 1; i > 0; i--) {
            // Move current root to end
            int temp = arr[0];
```

```
            heapify(arr, i, 0);
        }
    }

    // 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) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

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

    /* A utility function to print array of size n */
    static void printArray(int[] arr)
    {
        int N = arr.Length;
        for (int i = 0; i < N; ++i)
            Console.Write(arr[i] + " ");
        Console.Read();
    }

    // Driver's code
    public static void Main()
    {
        int[] arr = { 12, 11, 13, 5, 6, 7 };
        int N = arr.Length;

        // Function call
        HeapSort ob = new HeapSort();
```

```
}

// This code is contributed
// by Akanksha Rai(Abby_akku)
```

PHP

```
<?php

// Php program for implementation of Heap Sort

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

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

// main function to do heap sort
function heapSort(&$arr, $N)
{
    // Build heap (rearrange array)
    for ($i = $N / 2 - 1; $i >= 0; $i--)
        heapify($arr, $N, $i);

    // One by one extract an element from heap
```

```
            $arr[0] = $arr[$i];
            $arr[$i] = $temp;

            // call max heapify on the reduced heap
            heapify($arr, $i, 0);
        }
    }

    /* A utility function to print array of size n */
    function printArray(&$arr, $N)
    {
        for ($i = 0; $i < $N; ++$i)
            echo ($arr[$i]." ") ;

    }

        // Driver's program
        $arr = array(12, 11, 13, 5, 6, 7);
        $N = sizeof($arr)/sizeof($arr[0]);

        // Function call
        heapSort($arr, $N);

        echo 'Sorted array is ' . "\n";

        printArray($arr , $N);

    // This code is contributed by Shivi_Aggarwal
    ?>
```

Javascript

```
    // JavaScript program for implementation
    // of Heap Sort

        function sort( arr)
        {
            var N = arr.length;

            // Build heap (rearrange array)
            for (var i = Math.floor(N / 2) - 1; i >= 0; i--)
                heapify(arr, N, i);

            // One by one extract an element from heap
            for (var i = N - 1; i > 0; i--) {
                // Move current root to end
                var temp = arr[0];
```

```
            heapify(arr, i, 0);
        }
    }

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

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

    /* A utility function to print array of size n */
    function printArray(arr)
    {
        var N = arr.length;
        for (var i = 0; i < N; ++i)
            document.write(arr[i] + " ");

    }


    var arr = [12, 11, 13, 5, 6, 7];
    var N = arr.length;

    sort(arr);

    document.write( "Sorted array is");
    printArray(arr, N);
```

Output

```
Sorted array is
5 6 7 11 12 13
```

Time Complexity: O(N log N)

Auxiliary Space: O(1)

Example: Another Approach

C++

```
#include <bits/stdc++.h>
using namespace std;

void heap_sort(vector<int>& arr);
void build_heap(vector<int>& arr);
void heapify(vector<int>& arr, int i, int heap_size);

void heap_sort(vector<int>& arr){
    // Build a heap from the input array
    build_heap(arr);

    // Repeat until the heap contains only one element
    for (int i = arr.size() - 1; i > 0; i--){
        // Swap the root element with the last element
        swap(arr[0], arr[i]);

        // Remove the last element (which is now in the correct position)
        int heap_size = i;
        heapify(arr, 0, heap_size);
    }
    // Reverse the sorted array and return it
    reverse(arr.begin(), arr.end());
}

// building heap
void build_heap(vector<int>& arr){
    // Build a max heap from the input array
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--){
        heapify(arr, i, n);
    }
}
```

```
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < heap_size && arr[left] > arr[largest])
        largest = left;
    if (right < heap_size && arr[right] > arr[largest])
        largest = right;

    if (largest != i){
        swap(arr[i], arr[largest]);
        heapify(arr, largest, heap_size);
    }
}

// Example usage
// Driver code to test above function
int main(){
    vector<int> arr = {5, 2, 9, 1, 5, 6};
    heap_sort(arr);
      cout<<"[";
    for(int i = 0; i<arr.size()-1; i++){
      cout<<arr[i]<<", ";
    }
      cout<<arr[arr.size()-1]<<"]";
    cout << endl; // Output: 1 2 5 5 6 9
    return 0;
}
```

Java

```
public class HeapSort
{
  public static void heapSort(int[] arr)
  {

    // Build a heap from the input array
    buildHeap(arr);

    // Repeat until the heap contains only one element
    for (int i = arr.length - 1; i >= 1; i--)
    {

      // Swap the root element with the last element
      int temp = arr[0];
      arr[0] = arr[i];
      arr[i] = temp;
```

```
    }

    // Reverse the order of the elements to obtain the sorted array
    reverseArray(arr);
  }

  private static void buildHeap(int[] arr)
  {

    // Build a max heap from the input array
    int n = arr.length;
    for (int i = n / 2 - 1; i >= 0; i--) {
      heapify(arr, i, n);
    }
  }

  private static void heapify(int[] arr, int i, int heapSize)
  {

    // Heapify the subtree rooted at i in the input array
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < heapSize && arr[left] > arr[largest]) {
      largest = left;
    }

    if (right < heapSize && arr[right] > arr[largest]) {
      largest = right;
    }

    if (largest != i) {
      int temp = arr[i];
      arr[i] = arr[largest];
      arr[largest] = temp;
      heapify(arr, largest, heapSize);
    }
  }

  private static void reverseArray(int[] arr) {
    // Reverse the order of the elements to obtain the sorted array
    int n = arr.length;
    for (int i = 0; i < n / 2; i++) {
      int temp = arr[i];
      arr[i] = arr[n - i - 1];
      arr[n - i - 1] = temp;
    }
```

```
    heapSort(arr);
    System.out.print("[");
    for (int i=0; i<arr.length-1; i++) {
      System.out.print(arr[i] + ", ");
    }
    System.out.print(arr[arr.length-1] + "]");
  }
}
```

Python3

```
def heap_sort(arr):
    # Build a heap from the input array
    build_heap(arr)

    # Repeat until the heap contains only one element
    for i in range(len(arr) - 1, 0, -1):
        # Swap the root element with the last element
        arr[0], arr[i] = arr[i], arr[0]

        # Remove the last element (which is now in the correct position)
        heap_size = i
        heapify(arr, 0, heap_size)

    # Reverse the order of the elements to obtain the sorted array
    arr.reverse()

def build_heap(arr):
    # Build a max heap from the input array
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, i, n)

def heapify(arr, i, heap_size):
    # Heapify the subtree rooted at i in the input array
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < heap_size and arr[left] > arr[largest]:
        largest = left

    if right < heap_size and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
```

```
heap_sort(arr)
print(arr)
```

C#

```
using System;

public class GFG{
    public static void heap_sort(int[] arr){
        // Build a heap from the input array
        build_heap(arr);

        // Repeat until the heap contains only one element
        for (int i = arr.Length - 1; i >= 1; i--){
            // Swap the root element with the last element
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Remove the last element (which is now in the correct position)
            int heapSize = i;
            heapify(arr, 0, heapSize);
        }

        // Reverse the order of the elements to obtain the sorted array
        ReverseArray(arr);
    }

    private static void build_heap(int[] arr){
        // Build a max heap from the input array
        int n = arr.Length;
        for (int i = n / 2 - 1; i >= 0; i--){
            heapify(arr, i, n);
        }
    }

    private static void heapify(int[] arr, int i, int heapSize){
        // Heapify the subtree rooted at i in the input array
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < heapSize && arr[left] > arr[largest]){
            largest = left;
        }

        if (right < heapSize && arr[right] > arr[largest]){
```

```
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            heapify(arr, largest, heapSize);
        }
    }

    private static void ReverseArray(int[] arr){
        // Reverse the order of the elements to obtain the sorted array
        int n = arr.Length;
        for (int i = 0; i < n / 2; i++){
            int temp = arr[i];
            arr[i] = arr[n - i - 1];
            arr[n - i - 1] = temp;
        }
    }

    // Example usage
    public static void Main(){
        int[] arr = { 5, 2, 9, 1, 5, 6 };
        heap_sort(arr);
        Console.Write("[");
        for (int i = 0; i < arr.Length - 1; i++){
            Console.Write(arr[i] + ", ");
        }
        Console.Write(arr[arr.Length - 1] + "]");
    }
}
```

Javascript

```
function heap_sort(arr)
{

    // Build a heap from the input array
    build_heap(arr);

    // Repeat until the heap contains only one element
    for (let i = arr.length - 1; i > 0; i--)
    {

        // Swap the root element with the last element
        [arr[0], arr[i]] = [arr[i], arr[0]];

        // Remove the last element (which is now in the correct position)
        let heap_size = i;
```

```
    return arr;
}

function build_heap(arr)
{

    // Build a max heap from the input array
    let n = arr.length;
    for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
        heapify(arr, i, n);
    }
}

function heapify(arr, i, heap_size)
{

    // Heapify the subtree rooted at i in the input array
    let largest = i;
    let left = 2 * i + 1;
    let right = 2 * i + 2;

    if (left < heap_size && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < heap_size && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        [arr[i], arr[largest]] = [arr[largest], arr[i]];
        heapify(arr, largest, heap_size);
    }
}

// Example usage
let arr = [5, 2, 9, 1, 5, 6];
arr = heap_sort(arr);
console.log(arr.reverse()); // output: [9, 6, 5, 5, 2, 1]
```

Output

```
[9, 6, 5, 5, 2, 1]
```

In this example, the input array [5, 2, 9, 1, 5, 6] is sorted in ascending order using the heap_sort function, which modifies the input array in-place. The sorted array

The **time complexity of Heap Sort is O(n log n)** in the worst and average cases, where n is the number of elements in the input array.

The build_heap() function takes O(n) time to build the heap from the input array. The heapify() function is called on each node of the heap once, so it takes O(log n) time. Therefore, the build_heap() function takes O(n log n) time in total.

The for loop in heap_sort() runs n-1 times, and each iteration involves swapping the root element with the last element and then performing heapify() on the remaining heap. Swapping the elements takes constant time, while heapify() takes O(log n) time. Therefore, the time complexity of heap_sort() is O(n log n) in the worst and average cases.

In the best case, where the input array is already sorted, heapify() is never called, and the time complexity of Heap Sort reduces to O(n). However, this is a very rare case.

Overall, Heap Sort has a worst-case time complexity of O(n log n), which is the same as the best-case time complexity of Merge Sort and Quick Sort. However, Heap Sort has a higher constant factor compared to Merge Sort and Quick Sort, making it slower in practice for small to medium-sized arrays. It is also not a stable sort, meaning that it may change the relative order of equal elements in the input array.

Some FAQs related to Heap Sort

What are the two phases of Heap Sort?

The heap sort algorithm consists of two phases. In the first phase the array is converted into a max heap. And in the second phase the highest element is removed (i.e., the one at the tree root) and the remaining elements are used to create a new max heap.

Why Heap Sort is not stable?

Heap sort algorithm is not a stable algorithm. This algorithm is not stable

Is Heap Sort an example of “Divide and Conquer” algorithm?

Heap sort is **NOT** at all a Divide and Conquer algorithm. It uses a heap data structure to efficiently sort its element and not a “divide and conquer approach” to sort the elements.

Which sorting algorithm is better – Heap sort or Merge Sort?

The answer lies in the comparison of their time complexity and space requirement. The Merge sort is slightly faster than the Heap sort. But on the other hand merge sort takes extra memory. Depending on the requirement, one should choose which one to use.

Why Heap sort better than Selection sort?

Heap sort is similar to selection sort, but with a better way to get the maximum element. It takes advantage of the heap data structure to get the maximum element in constant time.

Related articles:

- Quiz on Heap Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- QuickSort,
- Selection Sort,
- Bubble Sort,
- Insertion Sort,
- Merge Sort,
- Heap Sort,
- QuickSort,
- Radix Sort,
- Counting Sort,
- Bucket Sort

Coding practice for sorting

Last Updated : 05 Apr, 2023

475

Recommended Problem

Heap Sort

Sorting Heap +2 more Amazon Microsoft +7 more

Solve Problem

Submission count: 72K

Similar Reads

1. Difference between Binary Heap, Binomial Heap and Fibonacci Heap
2. Heap Sort for decreasing order using min heap
3. Difference between Min Heap and Max Heap
4. When building a Heap, is the structure of Heap unique?
5. Convert Min Heap to Max Heap
6. What's the relationship between "a" heap and "the" heap?
7. Python Code for time Complexity plot of Heap Sort
8. Sorting algorithm visualization : Heap Sort
9. Heap Sort Visualization using JavaScript
10. Lexicographical ordering using Heap Sort

Related Tutorials

1. Learn Data Structures with Javascript | DSA Tutorial

3. Introduction to Set – Data Structure and Algorithm Tutorials

4. Introduction to Map – Data Structure and Algorithm Tutorials

5. What is Dijkstra's Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy Normal Medium Hard Expert

Improved By : Shivi_Aggarwal, Akanksha_Rai, SoumikMondal, RishiAdvani, Vibhav Gupta, kushjaing, jainabhi279, _i_plus_plus_, Ameya Gharpure, rishiraj1996, itskawal2000, volumezero9786, triptisharma25220, sweetyty, nathanfriendgeeksforgeeks, sagartomar9927, amartyaghoshgfg, saicharan1153, amartajisce, yashmahajan3171, kashishkumar2, prabhat12pundeer, surwaseshrikant1729, janardansthox, anikettchpiow, ghoshtamojit531, chandanagarwv9m5

Article Tags : 24*7 Innovation Labs, Amazon, Belzabar, Heap Sort, Intuit, Oracle, Samsung, SAP Labs, Visa, DSA, Heap, Sorting

Practice Tags : 24*7 Innovation Labs, Amazon, Belzabar, Intuit, Oracle, Samsung, SAP Labs, Visa, Heap, Sorting

Improve Article Report Issue

feedback@geeksforgeeks.org

GATE CS Notes

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

Interview Corner

Company Preparation

Preparation for SDE

Company Interview Corner

Experienced Interview

Internship Interview

Competitive Programming

Aptitude

GfG School

CBSE Notes for Class 8

CBSE Notes for Class 9

CBSE Notes for Class 10

CBSE Notes for Class 11

CBSE Notes for Class 12

English Grammar

Data Science With Python

Data Science For Beginner

Machine Learning Tutorial

Maths For Machine Learning

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Python

Python Tutorial

Python Programming Examples

Django Tutorial

Python Projects

Python Tkinter

OpenCV Python Tutorial

UPSC/SSC/BANKING

SSC CGL Syllabus

SBI PO Syllabus

IBPS PO Syllabus

UPSC Ethics Notes

UPSC Economics Notes

UPSC History Notes