Heap Data Structure

[**Data Structure and Algorithms Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=gfg&utm_medium=header+link+click&utm_campaign=dsa+course+tracker&utm_term=dsa+course+promo&utm_content=heap-lp)

[**Practice Problems on Heap**](https://practice.geeksforgeeks.org/explore/?category%5B%5D=Heap&page=1&category%5B%5D=Heap&utm_source=gfg&utm_medium=header+link+click&utm_campaign=practice+tracker&utm_term=practice+promo&utm_content=heap-lp)

[**Recent articles on Heap !**](https://www.geeksforgeeks.org/category/data-structures/heap/?utm_source=gfg&utm_medium=header+link+click&utm_campaign=recent+article+tracker&utm_term=recent+article+tracker&utm_content=heap-lp)

[**What is Heap Data Structure?**](https://www.geeksforgeeks.org/introduction-to-heap-data-structure-and-algorithm-tutorials/)

*A Heap is a special Tree-based data structure in which the tree is a complete binary tree.*



*Heap Data Structure*

**Operations of Heap Data Structure:**

* **Heapify:** a process of creating a heap from an array.
* **Insertion:** process to insert an element in existing heap time complexity O(log N).
* **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
* **Peek:** to check or find the most prior element in the heap, (max or min element for max and min heap).

**Types of Heap Data Structure**

Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

**Topics :**

* [Introduction](https://www.geeksforgeeks.org/heap-data-structure/?ref=ghm#introduction)
* [Standard problems on heap](https://www.geeksforgeeks.org/heap-data-structure/?ref=ghm#standard)

**Introduction:**

1. [Introduction to Heap – Data Structure and Algorithm Tutorials](https://www.geeksforgeeks.org/introduction-to-heap-data-structure-and-algorithm-tutorials/)
2. [Binary Heap](https://www.geeksforgeeks.org/binary-heap/)
3. [Applications, Advantages and Disadvantages of Heap](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-heap/)
4. [Time Complexity of building a heap](https://www.geeksforgeeks.org/g-fact-85/)
5. [Applications of Heap Data Structure](https://www.geeksforgeeks.org/applications-of-heap-data-structure/)
6. [Comparison between Heap and Tree](https://www.geeksforgeeks.org/comparison-between-heap-and-tree/)
7. [When building a Heap, is the structure of Heap unique?](https://www.geeksforgeeks.org/when-building-a-heap-is-the-structure-of-heap-unique/)

**Standard problems on Heap:**

* **Easy:**
  1. [Binomial Heap](https://www.geeksforgeeks.org/binomial-heap-2/)
  2. [Fibonacci Heap](https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/)
  3. [Leftist Heap](https://www.geeksforgeeks.org/leftist-tree-leftist-heap/)
  4. [K-ary Heap](https://www.geeksforgeeks.org/k-ary-heap/)
  5. [Heap Sort](https://www.geeksforgeeks.org/heap-sort/)
  6. [Check if a given Binary Tree is Heap](https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-heap/)
  7. [How to check if a given array represents a Binary Heap?](https://www.geeksforgeeks.org/how-to-check-if-a-given-array-represents-a-binary-heap/)
  8. [Iterative Heap Sort](https://www.geeksforgeeks.org/iterative-heap-sort/)
  9. [K’th Largest Element in an array](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)
  10. [K’th Smallest/Largest Element in Unsorted Array | Set 1](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/)
  11. [Height of a complete binary tree (or Heap) with N nodes](https://www.geeksforgeeks.org/height-complete-binary-tree-heap-n-nodes/)
  12. [Heap Sort for decreasing order using min heap](https://www.geeksforgeeks.org/heap-sort-for-decreasing-order-using-min-heap/)
* **Medium:**
  1. [Sort an almost sorted array](https://www.geeksforgeeks.org/nearly-sorted-algorithm/)
  2. [Print all nodes less than a value x in a Min Heap.](https://www.geeksforgeeks.org/print-all-nodes-less-than-a-value-x-in-a-min-heap/)
  3. [Tournament Tree (Winner Tree) and Binary Heap](https://www.geeksforgeeks.org/tournament-tree-and-binary-heap/)
  4. [Connect n ropes with minimum cost](https://www.geeksforgeeks.org/connect-n-ropes-minimum-cost/)
  5. [Maximum distinct elements after removing k elements](https://www.geeksforgeeks.org/maximum-distinct-elements-removing-k-elements/)
  6. [K maximum sum combinations from two arrays](https://www.geeksforgeeks.org/k-maximum-sum-combinations-two-arrays/)
  7. [Median of Stream of Running Integers using STL](https://www.geeksforgeeks.org/median-of-stream-of-running-integers-using-stl/)
  8. [Median in a stream of integers (running integers)](https://www.geeksforgeeks.org/median-of-stream-of-integers-running-integers/)
  9. [K’th largest element in a stream](https://www.geeksforgeeks.org/kth-largest-element-in-a-stream/)
  10. [Largest triplet product in a stream](https://www.geeksforgeeks.org/largest-triplet-product-stream/)
  11. [Find k numbers with most occurrences in the given array](https://www.geeksforgeeks.org/find-k-numbers-occurrences-given-array/)
  12. [Convert min Heap to max Heap](https://www.geeksforgeeks.org/convert-min-heap-to-max-heap/)
  13. [Given level order traversal of a Binary Tree, check if the Tree is a Min-Heap](https://www.geeksforgeeks.org/given-level-order-traversal-binary-tree-check-tree-min-heap/)
* **Hard:**
  1. [Design an efficient data structure for given operations](https://www.geeksforgeeks.org/a-data-structure-question/)
  2. [Merge k sorted arrays | Set 1](https://www.geeksforgeeks.org/merge-k-sorted-arrays/)
  3. [Merge Sort Tree for Range Order Statistics](https://www.geeksforgeeks.org/merge-sort-tree-for-range-order-statistics/)
  4. [Sort numbers stored on different machines](https://www.geeksforgeeks.org/sort-numbers-stored-on-different-machines/)
  5. [Smallest Derangement of Sequence](https://www.geeksforgeeks.org/smallest-derangement-sequence/)
  6. [Largest Derangement of a Sequence](https://www.geeksforgeeks.org/largest-derangement-sequence/)
  7. [Maximum difference between two subsets of m elements](https://www.geeksforgeeks.org/difference-maximum-sum-minimum-sum-n-m-elementsin-review/)
  8. [Convert BST to Min Heap](https://www.geeksforgeeks.org/convert-bst-min-heap/)
  9. [Merge two binary Max Heaps](https://www.geeksforgeeks.org/merge-two-binary-max-heaps/)
  10. [K-th Largest Sum Contiguous Subarray](https://www.geeksforgeeks.org/k-th-largest-sum-contiguous-subarray/)
  11. [Minimum product of k integers in an array of positive Integers](https://www.geeksforgeeks.org/minimum-product-k-integers-array-positive-integers/)
  12. [Leaf starting point in a Binary Heap data structure](https://www.geeksforgeeks.org/leaf-starting-point-binary-heap-data-structure/)
  13. [Rearrange characters in a string such that no two adjacent are same](https://www.geeksforgeeks.org/rearrange-characters-string-no-two-adjacent/)
  14. [Sum of all elements between k1’th and k2’th smallest elements](https://www.geeksforgeeks.org/sum-elements-k1th-k2th-smallest-elements/)
  15. [Minimum sum of two numbers formed from digits of an array](https://www.geeksforgeeks.org/minimum-sum-two-numbers-formed-digits-array-2/)

**Easy Questions:**

**Binomial Heap**

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 14 Dec, 2022
* Read
* Discuss(36)
* Courses
* Practice
* Video

The main application of [Binary Heap](https://www.geeksforgeeks.org/binary-heap/)is as implement a priority queue. Binomial Heap is an extension of [Binary Heap](http://geeksquiz.com/binary-heap/)that provides faster union or merge operation with other operations provided by Binary Heap.

*A Binomial Heap is a collection of Binomial Trees*

**What is a Binomial Tree?**

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one the leftmost child of the other.

A Binomial Tree of order k the has following properties.

* It has exactly 2k nodes.
* It has depth as k.
* There are exactly kaiCi nodes at depth i for i = 0, 1, . . . , k.
* The root has degree k and children of the root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

k = 0 (Single Node)

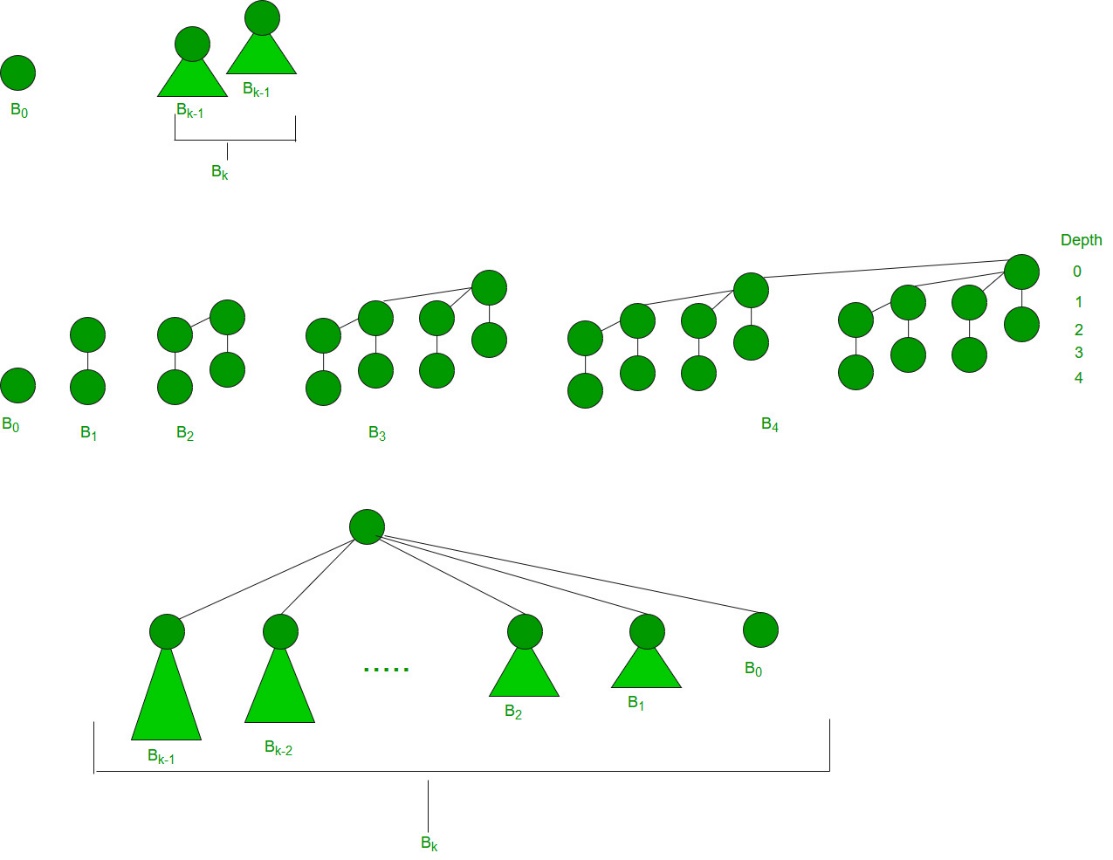
o

k = 1 (2 nodes)   
[We take two k = 0 order Binomial Trees, and  
make one as a child of other]  
 o  
 /   
o

k = 2 (4 nodes)  
[We take two k = 1 order Binomial Trees, and  
make one as a child of other]  
 o  
 / \  
 o o  
 /   
o

k = 3 (8 nodes)  
[We take two k = 2 order Binomial Trees, and  
make one as a child of other]  
 o   
 / | \   
 o o o  
 / \ |   
 o o o   
 /   
o

The following diagram is referred to from the 2nd Edition of the [CLRS book](http://www.flipkart.com/introduction-algorithms-english-3rd/p/itmdwxyrafdburzg?pid=9788120340077&affid=sandeepgfg).



**Binomial Heap:**

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows the Min Heap property. And there can be at most one Binomial Tree of any degree.

**Examples Binomial Heap:**

12------------10--------------------20  
 / \ / | \  
 15 50 70 50 40  
 | / | |   
 30 80 85 65   
 |  
 100  
A Binomial Heap with 13 nodes. It is a collection of 3   
Binomial Trees of orders 0, 2, and 3 from left to right.

10--------------------20  
 / \ / | \  
 15 50 70 50 40  
 | / | |   
 30 80 85 65   
 |  
 100

A Binomial Heap with 12 nodes. It is a collection of 2

Binomial Trees of orders 2 and 3 from left to right.

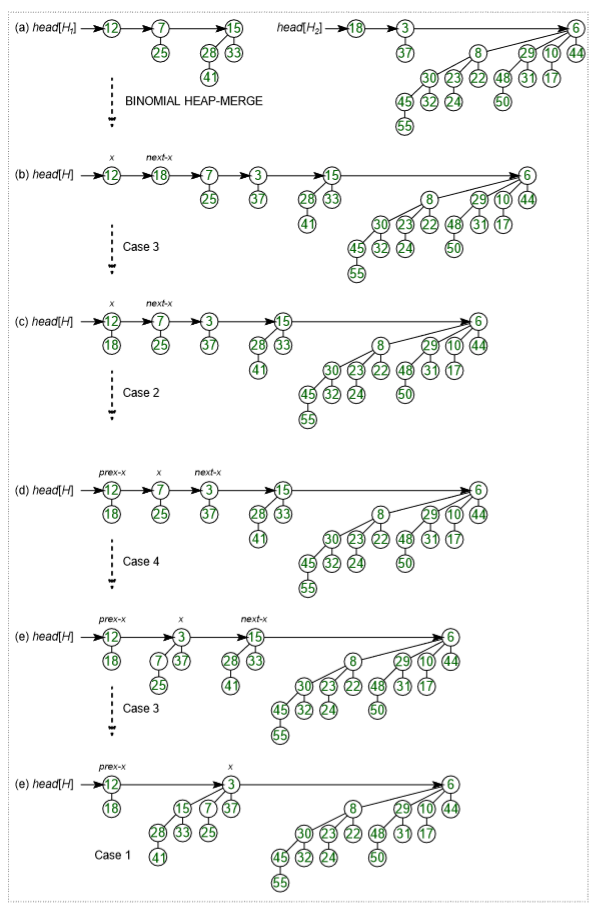
**Binary Representation of a number and Binomial Heaps**

A Binomial Heap with n nodes has the number of Binomial Trees equal to the number of set bits in the binary representation of n. For example, let n be 13, there are 3 set bits in the binary representation of n (00001101), hence 3 Binomial Trees. We can also relate the degree of these Binomial Trees with positions of set bits. With this relation, we can conclude that there are O(Logn) Binomial Trees in a Binomial Heap with ‘n’ nodes.

**Operations of Binomial Heap:**

The main operation in Binomial Heap is a union(), all other operations mainly use this operation. The union() operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss union later.

1. insert(H, k): Inserts a key ‘k’ to Binomial Heap ‘H’. This operation first creates a Binomial Heap with a single key ‘k’, then calls union on H and the new Binomial heap.
2. getting(H): A simple way to get in() is to traverse the list of the roots of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to the minimum key root.
3. extracting(H): This operation also uses a union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.
4. delete(H): Like Binary Heap, the delete operation first reduces the key to minus infinite, then calls extracting().
5. decrease key(H): decrease key() is also similar to Binary Heap. We compare the decreased key with its parent and if the parent’s key is more, we swap keys and recur for the parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node. The time complexity of the decrease key() is O(Logn).   
   **Union operation in Binomial Heap:**   
   Given two Binomial Heaps H1 and H2, union(H1, H2) creates a single Binomial Heap.
6. The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.
7. After the simple merge, we need to make sure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, prev, x, and next-x. There can be the following 4 cases when we traverse the list of roots.   
   —–Case 1: Orders of x and next-x are not the same, we simply move ahead.   
   In the following 3 cases, orders of x and next-x are the same.   
   —–Case 2: If the order of next-next-x is also the same, move ahead.   
   —–Case 3: If the key of x is smaller than or equal to the key of next-x, then make next-x a child of x by linking it with x.   
   —–Case 4: If the key of x is greater, then make x the child of next.   
   The following diagram is taken from the 2nd Edition of the [CLRS book](http://www.flipkart.com/introduction-algorithms-english-3rd/p/itmdwxyrafdburzg?pid=9788120340077&affid=sandeepgfg).



**Time Complexity Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operations** | **Binary Heap** | **Binomial Heap** | **Fibonacci Heap** |
| **Procedure** | **Worst-case** | **Worst-case** | **Amortized** |
| Making Heap | Θ(1) | Θ(1) | Θ(1) |
| Inserting a node | Θ(log(n)) | O(log(n)) | Θ(1) |
| Finding Minimum key | Θ(1) | O(log(n)) | O(1) |
| Extract-Minimum key | Θ(log(n)) | Θ(log(n)) | O(log(n)) |
| Union or merging | Θ(n) | O(log(n)) | Θ(1) |
| Decreasing a Key | Θ(log(n)) | Θ(log(n)) | Θ(1) |
| Deleting a node | Θ(log(n)) | Θ(log(n)) | O(log(n)) |

**How to represent Binomial Heap?**

A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this in and extracting() and delete()). The idea is to represent Binomial Trees as the leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and the other to the right sibling.

**Fibonacci Heap | Set 1 (Introduction)**

Heaps are mainly used for implementing priority queue. We have discussed the below heaps in previous posts.

1. [Binary Heap](http://geeksquiz.com/binary-heap/)
2. [Binomial Heap](https://www.geeksforgeeks.org/binomial-heap-2/)

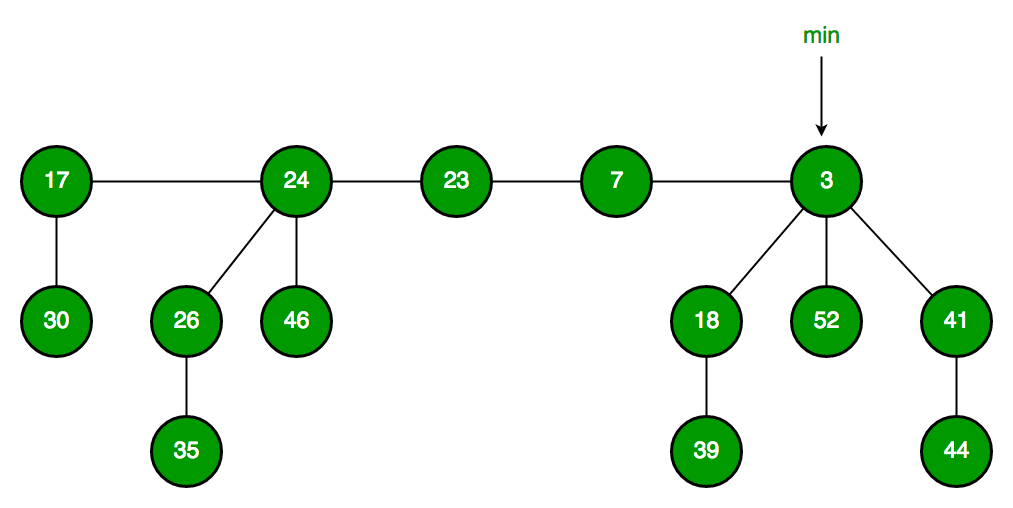
In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heap.

Below are [amortized time complexities](https://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/) of the **Fibonacci Heap**.

1) Find Min: **Θ(1)** [Same as Binary but not Binomial since binomial has o(log n)]  
2) Delete Min: **O(Log n)** [Θ(Log n) in both Binary and Binomial]  
3) Insert:  **Θ(1)** [Θ(Log n) in Binary and Θ(1) in Binomial]  
4) Decrease-Key: **Θ(1)** [Θ(Log n) in both Binary and Binomial]  
5) Merge:  **Θ(1)** [Θ(m Log n) or Θ(m+n) in Binary and  
 Θ(Log n) in Binomial]

Like [Binomial Heap](https://www.geeksforgeeks.org/binomial-heap-2/), Fibonacci Heap is a collection of trees with min-heap or max-heap properties. In Fibonacci Heap, trees can have any shape even if all trees can be single nodes (This is unlike Binomial Heap where every tree has to be a Binomial Tree).

Below is an example Fibonacci Heap taken from [here](https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf).



Fibonacci Heap maintains a pointer to the minimum value (which is the root of a tree). All tree roots are connected using a circular doubly linked list, so all of them can be accessed using a single ‘min’ pointer.

The main idea is to execute operations in a “lazy” way. For example merge operation simply links two heaps, insert operation simply adds a new tree with a single node. The operation extract minimum is the most complicated operation. It does delay the work of consolidating trees. This makes delete also complicated as delete first decreases the key to minus infinite, then calls extract minimum.

**Below are some interesting facts about Fibonacci Heap**

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, the time complexity of these algorithms is O(VLogV + ELogV). If Fibonacci Heap is used, then time complexity is improved to O(VLogV + E)
2. Although Fibonacci Heap looks promising time complexity-wise, it has been found slow in practice as hidden constants are high (Source [Wiki](https://en.wikipedia.org/wiki/Fibonacci_heap)).
3. Fibonacci heaps is mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has a degree at most O(log n) and the size of a subtree rooted in a node of degree k is at least Fk+2, where Fk is the kth Fibonacci number.

We will soon be discussing Fibonacci Heap operations in detail.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

*From <*[*https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/*](https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/)*>*

**Leftist Tree / Leftist Heap**

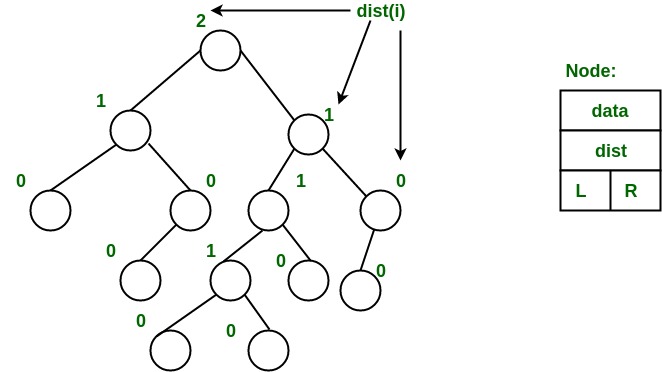
A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a [complete binary tree](https://www.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/)), a leftist tree may be very unbalanced. Below are [time complexities](https://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/) of **Leftist Tree / Heap**.

**Function Complexity Comparison**  
1) Get Min: **O(1)** [same as both Binary and Binomial]  
2) Delete Min: **O(Log n)** [same as both Binary and Binomial]  
3) Insert: **O(Log n)** [O(Log n) in Binary and O(1) in   
 Binomial and O(Log n) for worst case]   
4) Merge:  **O(Log n)** [O(Log n) in Binomial]

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property :**key(i) >= key(parent(i))
2. **Heavier on left side :**dist(right(i)) <= dist(left(i)). Here, dist(i) is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and dist( i ) = 1 + dist( right( i ) ).

**Example:** The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right subtree of this node is null and its parent has a distance of 1 by dist( i ) = 1 + dist( right( i )). The same is followed for each node and their s-value( or rank) is calculated.

[](https://media.geeksforgeeks.org/wp-content/uploads/leftist_tree.jpg)

 From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has x nodes, then leftist heap has atleast 2x – 1 nodes. This means the length of path to rightmost leaf is O(log n) for a leftist heap with n nodes.

**Operations :**

1. The main operation is merge().
2. deleteMin() (or extractMin() can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done be create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

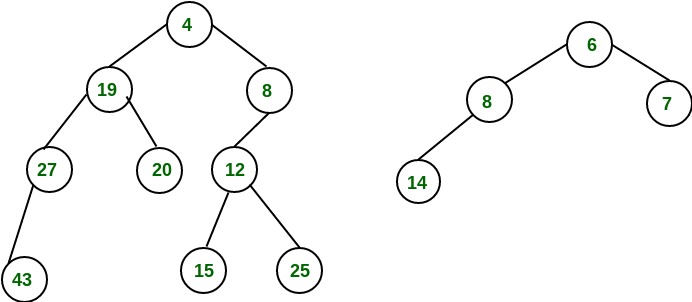
**Idea behind Merging :**Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion: – Update dist() of merged root. – Swap left and right subtrees just below root, if needed, to keep leftist property of merged result

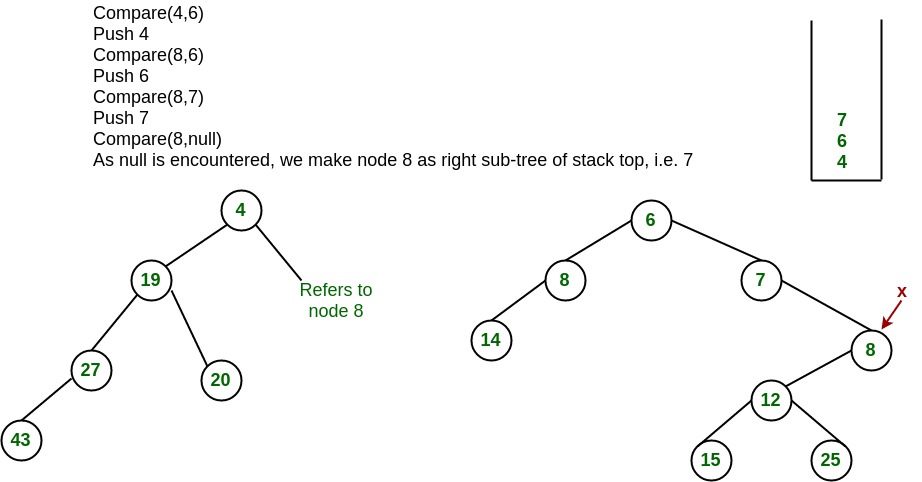
**Detailed Steps for Merge:**

1. Compare the roots of two heaps.
2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

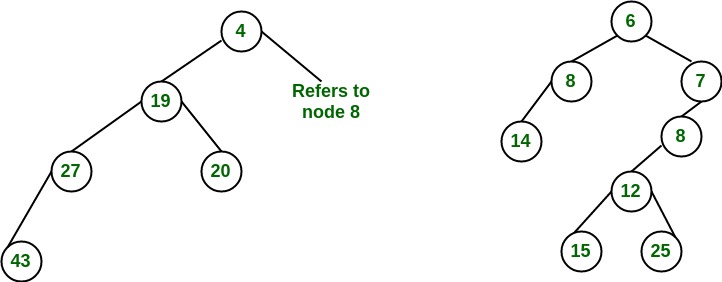
**Example:** Consider two leftist heaps given below:

[](https://media.geeksforgeeks.org/wp-content/uploads/leftist_heap1.jpg)

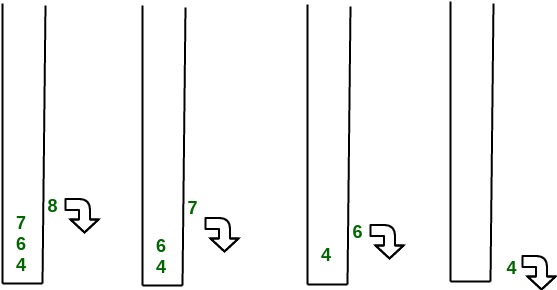
 Merge them into a single leftist heap

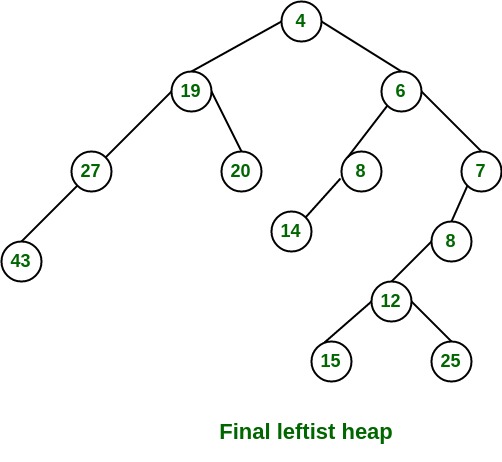
[](https://media.geeksforgeeks.org/wp-content/uploads/leftist_heap2.jpg)

 The subtree at node 7 violates the property of leftist heap so we swap it with the left child and retain the property of leftist heap.

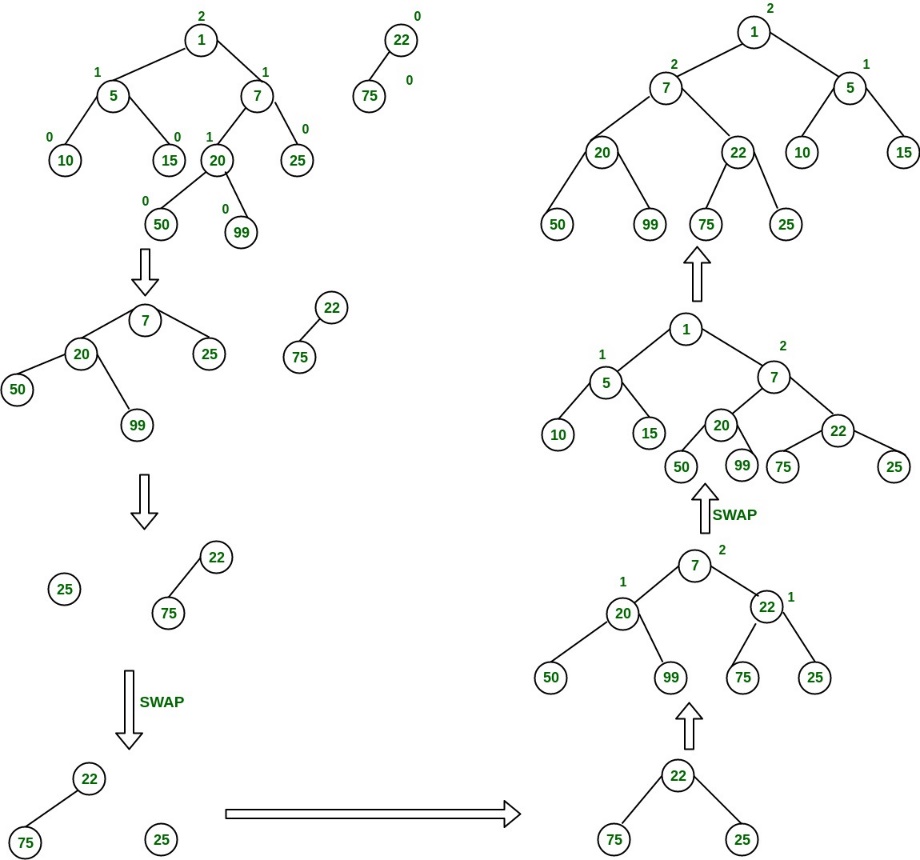
[](https://media.geeksforgeeks.org/wp-content/uploads/leftist_heap3.jpg)

 Convert to leftist heap. Repeat the process

[](https://media.geeksforgeeks.org/wp-content/uploads/leftist_heap4.jpg)

[](https://media.geeksforgeeks.org/wp-content/uploads/leftist_heap5.jpg)

 The worst case time complexity of this algorithm is O(log n) in the worst case, where n is the number of nodes in the leftist heap. **Another example of merging two leftist heap:**

[](https://media.geeksforgeeks.org/wp-content/uploads/mergingTwoLeftistTree-1.jpg)

**Implementation of leftist Tree / leftist Heap:**

**Heap Sort**

**What is Heap Sort**

***Heap sort****is a comparison-based sorting technique based on*[*Binary Heap*](http://www.geeksforgeeks.org/binary-heap/)*data structure. It is similar to the*[*selection sort*](http://www.geeksforgeeks.org/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.*

1. Heap sort is an in-place algorithm.
2. Its typical implementation is not stable, but can be made stable (See [this](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/))
3. Typically 2-3 times slower than well-implemented [QuickSort](http://www.geeksforgeeks.org/quick-sort/).  The reason for slowness is a lack of locality of reference.

**Advantages of heapsort:**

1. **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.
2. **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
3. **Simplicity –** It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion

**Applications of HeapSort:**

1. Heapsort is mainly used in hybrid algorithms like the [IntroSort](https://www.geeksforgeeks.org/introsort-or-introspective-sort/).
2. [Sort a nearly sorted (or K sorted) array](https://www.geeksforgeeks.org/nearly-sorted-algorithm/)
3. [k largest(or smallest) elements in an array](https://www.geeksforgeeks.org/k-largestor-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](https://www.geeksforgeeks.org/applications-of-heap-data-structure/)

Recommended Problem

Heap Sort

[Heap](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Heap&sortBy=submissions)

[Sorting](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Sorting&sortBy=submissions)

+2 more

[24\*7 Innovation Labs](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=24*7%20Innovation%20Labs&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

+7 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/heap-sort/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 67.7K

**What is meant by Heapify?**

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 first 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:*

*1*

*/     \*

*3         5*

*/    \     /  \*

*4      6   13  10*

*/ \    / \*

*9   8  15 17*

***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.*

*1*

*/     \*

*3         5*

*/    \      /  \*

*4      17   13  10*

*/ \    /  \*

*9   8  15   6*

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

*1*

*/     \*

*3         5*

*/    \      /  \*

*9      17   13  10*

*/ \    /  \*

*4   8  15   6*

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

*1*

*/     \*

*3         13*

*/    \      /  \*

*9      17   5   10*

*/ \    /  \*

*4   8  15   6*

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

*1*

*/     \*

*17         13*

*/    \      /  \*

*9      15   5   10*

*/ \    /  \*

*4   8  3   6*

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

*17*

*/      \*

*15         13*

*/    \      /  \*

*9      6    5   10*

*/ \    /  \*

*4   8  3    1*

**Heap Sort Algorithm**

To solve the problem follow the below idea:

*First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.*

Follow the given steps to solve the problem:

1. Build a max heap from the input data.
2. 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.
3. Repeat step 2 while the size of the heap is greater than 1.

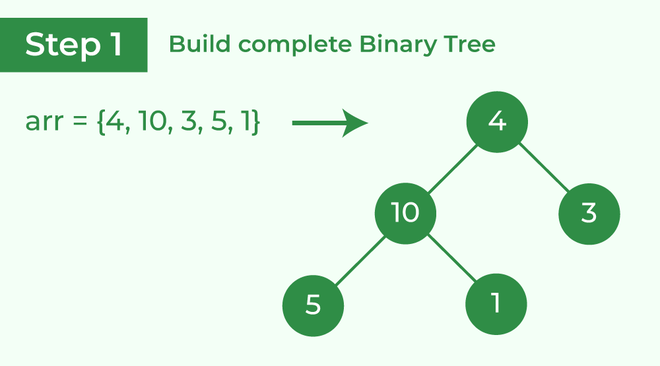
**Note:** The heapify procedure can only be applied to a node if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

**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.*



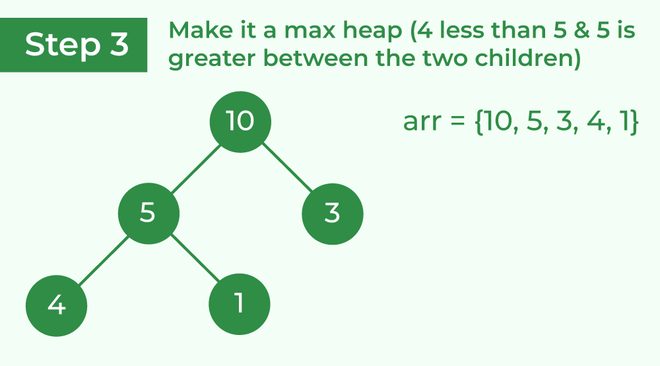
*Build complete binary tree from the array*

***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.*](https://www.geeksforgeeks.org/difference-between-min-heap-and-max-heap/)

1. *To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes*
2. *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*

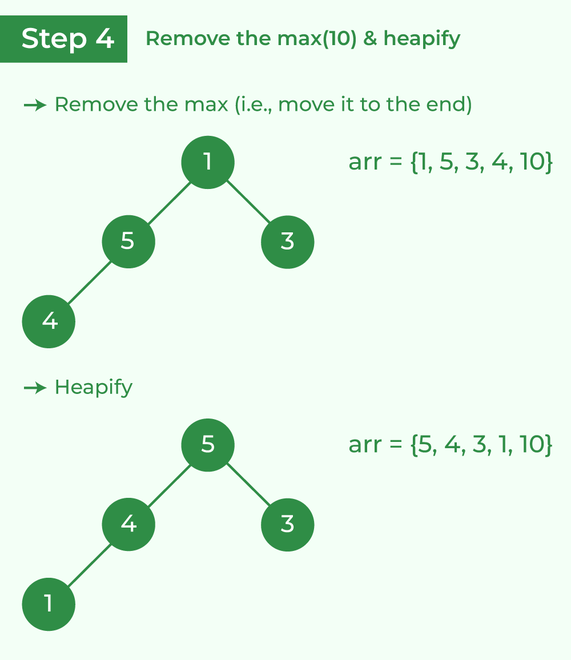
1. *Now, as seen,****4****as a parent is smaller than the child****5****, thus swap both of these again and the resulted heap and array should be like this:*



*Make the tree a max heap*

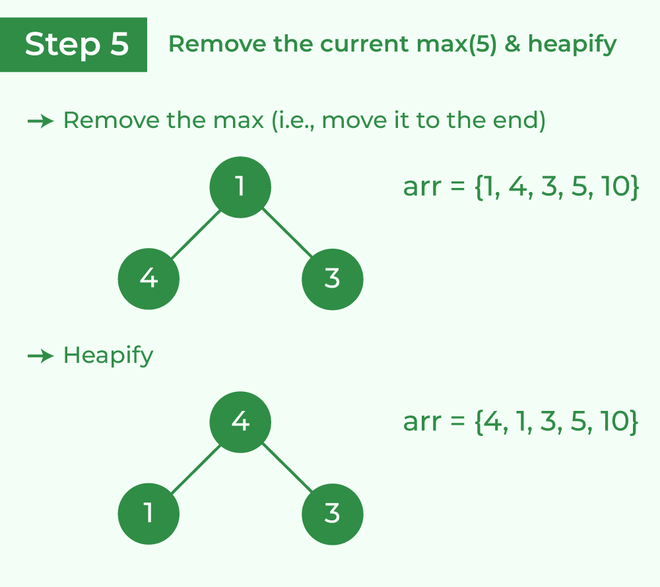
***Perform heap sort:****Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.*

1. *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.*
2. *Resulted heap and array should look like this:*



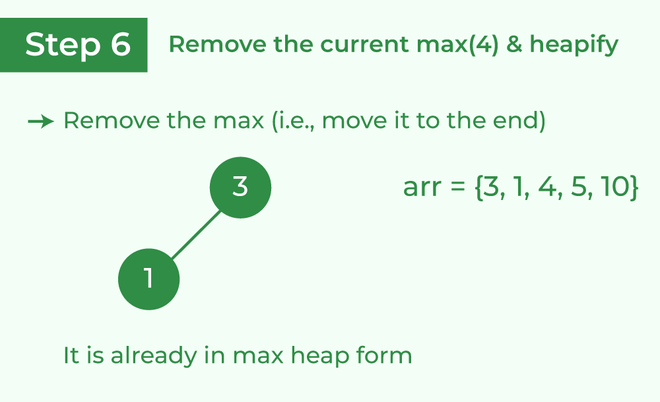
*Remove 10 and perform heapify*

1. *Repeat the above steps and it will look like the following:*



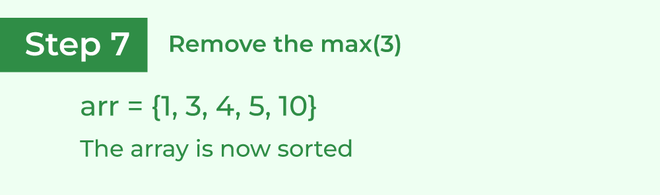
*Remove 5 and perform heapify*

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



*Remove 4 and perform heapify*

1. *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:

1. C
2. C++
3. Java
4. Python3
5. C#
6. PHP
7. Javascript

# 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

        # Heapify the root.

        heapify(arr, N, largest)

# 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

**Output**

Sorted array is   
5 6 7 11 12 13

**Time Complexity:**O(N log N)

**Auxiliary Space:**O(1)

**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 because the operations that are performed in a heap can change the relative ordering of the equivalent keys.

**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.

*From <*[*https://www.geeksforgeeks.org/heap-sort/*](https://www.geeksforgeeks.org/heap-sort/)*>*

**Check if a given Binary Tree is a Heap**

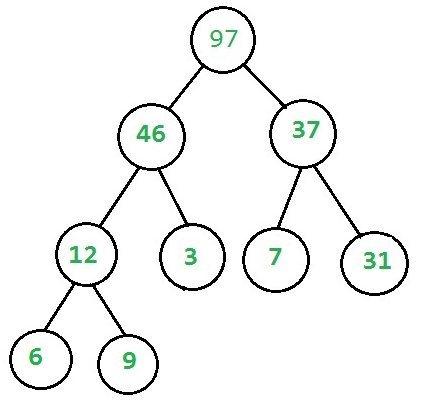
1. Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
2. Last Updated : 28 Oct, 2022
3. Read
4. Discuss(51)
5. Courses
6. Practice
7. Video

Given a binary tree, check if it has heap property or not, Binary tree needs to fulfill the following two conditions for being a heap –

1. It should be a complete tree (i.e. all levels except the last should be full).
2. Every node’s value should be greater than or equal to its child node (considering max-heap).

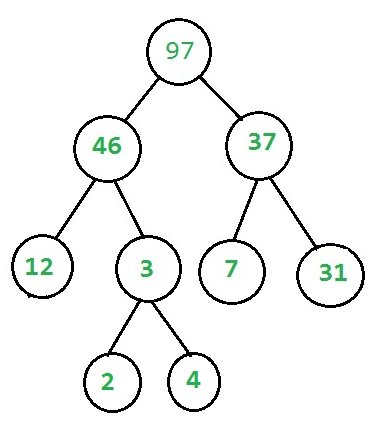
**Examples:**

***Input:***



***Output:****Given binary tree is a heap*

***Input:***



***Output:****Given binary tree is not a heap*

Recommended Problem

Is Binary Tree Heap

[Binary Search Tree](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Binary%20Search%20Tree&sortBy=submissions)

[Heap](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Heap&sortBy=submissions)

+2 more

[Hike](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Hike&sortBy=submissions)

[Solve Problem](https://practice.geeksforgeeks.org/problems/is-binary-tree-heap/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 55.6K

**Check if a given Binary Tree is Heap using Complete Binary Tree**

Follow the given steps to solve the problem:

1. Check each of the above conditions separately, for checking completeness **isComplete** and for checking heap **isHeapUtil** functions are written.
2. First, [check if the given binary tree is complete or not](https://www.geeksforgeeks.org/check-whether-binary-tree-complete-not-set-2-recursive-solution/).
3. Then to check if the binary tree is a heap or not, check the following points:
4. Every Node has 2 children, 0 children (last level nodes), or 1 child (there can be at most one such node).
5. If Node has No children then it’s a leaf node and returns true (Base case)
6. If Node has one child (it must be the left child because it is a complete tree) then compare this node with its single child only.
7. If the Node has both children then check heap property at this Node and recur for both subtrees.

Below is the implementation of the above approach:

1. C++
2. C
3. Java
4. Python3
5. C#
6. Javascript

# Python3 code To check if a binary

# tree is a MAX Heap or not

**class** GFG:

**def** \_\_init\_\_(self, value):

        self.key **=** value

        self.left **=** None

        self.right **=** None

**def** count\_nodes(self, root):

**if** root **is** None:

**return** 0

**else**:

**return** (1 **+** self.count\_nodes(root.left) **+**

                    self.count\_nodes(root.right))

**def** heap\_property\_util(self, root):

**if** (root.left **is** None **and**

                root.right **is** None):

**return** True

**if** root.right **is** None:

**return** root.key >**=** root.left.key

**else**:

**if** (root.key >**=** root.left.key **and**

                    root.key >**=** root.right.key):

**return** (self.heap\_property\_util(root.left) **and**

                        self.heap\_property\_util(root.right))

**else**:

**return** False

**def** complete\_tree\_util(self, root,

                           index, node\_count):

**if** root **is** None:

**return** True

**if** index >**=** node\_count:

**return** False

**return** (self.complete\_tree\_util(root.left, 2 **\***

                                        index **+** 1, node\_count) **and**

                self.complete\_tree\_util(root.right, 2 **\***

                                        index **+** 2, node\_count))

**def** check\_if\_heap(self):

        node\_count **=** self.count\_nodes(self)

**if** (self.complete\_tree\_util(self, 0, node\_count) **and**

                self.heap\_property\_util(self)):

**return** True

**else**:

**return** False

# Driver's Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    root **=** GFG(5)

    root.left **=** GFG(2)

    root.right **=** GFG(3)

    root.left.left **=** GFG(1)

    # Function call

**if** root.check\_if\_heap():

        print("Given binary tree is a heap")

**else**:

**print**("Given binary tree is not a Heap")

# This code has been

# contributed by Yash Agrawal

**Output**

Given binary tree is a Heap

**Time Complexity:**O(N), where N is the number of nodes

**Auxiliary Space:**O(logN), for recursive stack space.

**Check if a given Binary Tree is Heap using Level Order Traversal:**

*Level order traversal can be used to check heap properties at each level of the binary tree. Check whether value of each node is greater than the value of its children and keep track of when the last node is encountered and whether it is following the heap properties using a boolean flag*

Follow the given steps to solve the problem:

1. declare a queue for level order traversal and a flag variable nullish equal to false
2. Start level order traversal
3. Check for the left child of the node and if either the nullish is true or root’s value is less than its left child node, then return false, else push this node into the queue
4. If the node’s left child is null then set nullish equal to true, which means we have already encountered the last node, as the node with only zero or one children can occur only once in the complete tree
5. Now check the right child of the node and if either the nullish is true or root’s value is less than its right child node, then return false, else push this node into the queue.
6. If the node’s right child is null then set nullish equal to true, which means we have already encountered the last node, as the node with only zero or one children can occur only once in the complete tree
7. Return true after checking every node in the level order traversal
8. C++
9. Java
10. Python3
11. C#
12. Javascript

# Python3 program to check if a binary tree is max heap or not.

**from** collections **import** deque

**class** Node:

**def** \_\_init\_\_(self, value):

        self.key **=** value

        self.left **=** None

        self.right **=** None

**def** isHeap(root):

    queue **=** deque()

    queue.append(root)

    nullish **=** False

**while** len(queue) > 0:

        temp **=** queue[0]

        queue.popleft()

**if** temp.left:

**if** nullish **or** temp.left.key > temp.key:

**return** False

            queue.append(temp.left)

**else**:

            nullish **=** True

**if** temp.right:

**if** nullish **or** temp.right.key > temp.key:

**return** False

            queue.append(temp.right)

**else**:

            nullish **=** True

**return** True

# Driver's code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    root **=** Node(10)

    root.left **=** Node(9)

    root.right **=** Node(8)

    root.left.left **=** Node(7)

    root.left.right **=** Node(6)

    root.right.left **=** Node(5)

    root.right.right **=** Node(4)

    root.left.left.left **=** Node(3)

    root.left.left.right **=** Node(2)

    root.left.right.left **=** Node(1)

    # Function call

**if** isHeap(root):

**print**("Given binary tree is a Heap")

**else**:

**print**("Given binary tree is not a Heap")

# This code is contributed by lokeshmvs21.

**Output**

Given binary tree is a Heap

**Time Complexity**: O(N) where N is the total number of nodes in a given binary tree.

**Auxiliary Space:** O(N)

*From <*[*https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-heap/*](https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-heap/)*>*

**How to check if a given array represents a Binary Heap?**

Given an array, how to check if the given array represents a [Binary Max-Heap](https://www.geeksforgeeks.org/binary-heap/).

**Examples:**

Input: arr[] = {90, 15, 10, 7, 12, 2}   
Output: True  
The given array represents below tree  
 90  
 / \  
 15 10  
 / \ /  
 7 12 2   
The tree follows max-heap property as every  
node is greater than all of its descendants.

Input: arr[] = {9, 15, 10, 7, 12, 11}   
Output: False  
The given array represents below tree  
 9  
 / \  
 15 10  
 / \ /  
 7 12 11  
The tree doesn't follows max-heap property 9 is   
smaller than 15 and 10, and 10 is smaller than 11.

Recommended Problem

Does array represent Heap

[Arrays](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Arrays&sortBy=submissions)

[Binary Search Tree](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Binary%20Search%20Tree&sortBy=submissions)

+1 more

[Cisco](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Cisco&sortBy=submissions)

[Solve Problem](https://practice.geeksforgeeks.org/problems/does-array-represent-heap4345/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 22.9K

A **Simple Solution** is to first check root if it’s greater than all of its descendants. Then check for children of the root. Time complexity of this solution is O(n2)

An **Efficient Solution** is to compare root only with its children (not all descendants), if root is greater than its children and the same is true for all nodes, then tree is max-heap (This conclusion is based on transitive property of > operator, i.e., if x > y and y > z, then x > z).

The last internal node is present at index (n-2)/2 assuming that indexing begins with 0.

Below is the implementation of this solution.

# Python3 program to check whether a

# given array represents a max-heap or not

# Returns true if arr[i..n-1]

# represents a max-heap

**def** isHeap(arr, i, n):

    # If (2 \* i) + 1 >= n, then leaf node, so return true

**if** i >**=** int((n **-** 1) **/** 2):

**return** True

    # If an internal node and is greater

    # than its children, and same is

    # recursively true for the children

**if**(arr[i] >**=** arr[2 **\*** i **+** 1] **and**

       arr[i] >**=** arr[2 **\*** i **+** 2] **and**

       isHeap(arr, 2 **\*** i **+** 1, n) **and**

       isHeap(arr, 2 **\*** i **+** 2, n)):

**return** True

**return** False

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [90, 15, 10, 7, 12, 2, 7, 3]

    n **=** len(arr) **-** 1

**if** isHeap(arr, 0, n):

        print("Yes")

**else**:

**print**("No")

# This code is contributed by PranchalK

**Output**

Yes

**Time complexity:** O(n)

**Auxiliary Space:**O(h), Here h is the height of the given tree and the extra space is used due to the recursion call stack.

An **Iterative Solution** is to traverse all internal nodes and check id the node is greater than its children or not.

1. C++
2. Java
3. Python3
4. C#
5. PHP
6. Javascript

# Python3 program to check whether a

# given array represents a max-heap or not

# Returns true if arr[i..n-1]

# represents a max-heap

**def** isHeap(arr, n):

    # Start from root and go till

    # the last internal node

**for** i **in** range(int((n **-** 2) **/** 2) **+** 1):

        # If left child is greater,

        # return false

**if** arr[2 **\*** i **+** 1] > arr[i]:

**return** False

        # If right child is greater,

        # return false

**if** (2 **\*** i **+** 2 < n **and**

            arr[2 **\*** i **+** 2] > arr[i]):

**return** False

**return** True

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [90, 15, 10, 7, 12, 2, 7, 3]

    n **=** len(arr)

**if** isHeap(arr, n):

        print("Yes")

**else**:

**print**("No")

# This code is contributed by PranchalK

**Output**

Yes

**Time complexity**: O(n), Where n is the total number of elements in the given array.

**Auxiliary Space:**O(1), As constant extra space is used.

*From <*[*https://www.geeksforgeeks.org/how-to-check-if-a-given-array-represents-a-binary-heap/*](https://www.geeksforgeeks.org/how-to-check-if-a-given-array-represents-a-binary-heap/)*>*

**Iterative HeapSort**

1. Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
2. Read
3. Discuss(7)
4. Courses
5. Practice
6. Video

[**HeapSort**](https://www.geeksforgeeks.org/heap-sort/) is a comparison-based sorting technique where we first build Max Heap and then swap the root element with the last element (size times) and maintains the heap property each time to finally make it sorted.

**Examples:**

Input : 10 20 15 17 9 21  
Output : 9 10 15 17 20 21

Input: 12 11 13 5 6 7 15 5 19  
Output: 5 5 6 7 11 12 13 15 19

*In first Example, first we have to build Max Heap.*

*So, we will start from 20 as child and check for its parent. Here 10 is smaller, so we will swap these two.*

*Now, 20 10 15 17 9 21*

*Now, child 17 is greater than its parent 10. So, both will be swapped and order will be 20 17 15 10 9 21*

*Now, child 21 is greater than parent 15. So, both will be swapped.*

*20 17 21 10 9 15*

*Now, again 21 is bigger than parent 20. So,****21 17 20 10 9 15***

*This is Max Heap.*

*Now, we have to apply sorting. Here, we have to swap first element with last one and we have to maintain Max Heap property. So, after first swapping : 15 17 20 10 9 21 It clearly violates Max Heap property.*

*So, we have to maintain it. So, order will be*

*20 17 15 10 9 21*

*17 10 15 9 20 21*

*15 10 9 17 20 21*

*10 9 15 17 20 21*

***9 10 15 17 20 21***

*Here, underlined part is sorted part.*

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Implementation:**

1. C++
2. C
3. Java
4. Python3
5. C#
6. Javascript

# Python3 program for implementation

# of Iterative Heap Sort

# function build Max Heap where value

# of each child is always smaller

# than value of their parent

**def** buildMaxHeap(arr, n):

**for** i **in** range(n):

        # if child is bigger than parent

**if** arr[i] > arr[int((i **-** 1) **/** 2)]:

            j **=** i

            # swap child and parent until

            # parent is smaller

**while** arr[j] > arr[int((j **-** 1) **/** 2)]:

                (arr[j],

                 arr[int((j **-** 1) **/** 2)]) **=** (arr[int((j **-** 1) **/** 2)],

                                           arr[j])

                j **=** int((j **-** 1) **/** 2)

**def** heapSort(arr, n):

    buildMaxHeap(arr, n)

**for** i **in** range(n **-** 1, 0, **-**1):

        # swap value of first indexed

        # with last indexed

        arr[0], arr[i] **=** arr[i], arr[0]

        # maintaining heap property

        # after each swapping

        j, index **=** 0, 0

**while** True:

            index **=** 2 **\*** j **+** 1

            # if left child is smaller than

            # right child point index variable

            # to right child

**if** (index < (i **-** 1) **and**

                arr[index] < arr[index **+** 1]):

                index **+=** 1

            # if parent is smaller than child

            # then swapping parent with child

            # having higher value

**if** index < i **and** arr[j] < arr[index]:

                arr[j], arr[index] **=** arr[index], arr[j]

            j **=** index

**if** index >**=** i:

**break**

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [10, 20, 15, 17, 9, 21]

    n **=** len(arr)

**print**("Given array: ")

**for** i **in** range(n):

**print**(arr[i], end **=** " ")

**print**()

    heapSort(arr, n)

    # print array after sorting

    print("Sorted array: ")

**for** i **in** range(n):

        print(arr[i], end **=** " ")

# This code is contributed by PranchalK

**Output**

Given array: 10 20 15 17 9 21

Sorted array: 9 10 15 17 20 21

**Time Complexity:**O(n log n), Here, both function buildMaxHeap and heapSort runs in O(nlogn) time.

**Auxiliary Space:**O(1)

*From <*[*https://www.geeksforgeeks.org/iterative-heap-sort/*](https://www.geeksforgeeks.org/iterative-heap-sort/)*>*

**k largest(or smallest) elements in an array**

1. Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
2. Last Updated : 21 Dec, 2022
3. Read
4. Discuss(313)
5. Courses
6. Practice
7. Video

Write an efficient program for printing K largest elements in an array. Elements in an array can be in any order

**Examples:**

***Input:****[1, 23, 12, 9, 30, 2, 50], K = 3*

***Output:****50, 30, 23*

***Input:****[11, 5, 12, 9, 44, 17, 2], K = 2*

***Output:****44, 17*

Recommended Problem

k largest elements

[Arrays](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Arrays&sortBy=submissions)

[Heap](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Heap&sortBy=submissions)

+3 more

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

[Google](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Google&sortBy=submissions)

+3 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/k-largest-elements4206/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 1.1L

**Naive Approaches:** To solve the problem follow the below ideas:

**1. Using**[Bubble sort](https://www.geeksforgeeks.org/bubble-sort/)**:**

Follow the below steps to solve the problem:

1. Modify Bubble Sort to run the outer loop at most K times.
2. Print the last K elements of the array obtained in step 1

**Time Complexity:** O(N \* K)

Thanks to Shailendra for suggesting this approach.

**Note:**Like Bubble sort, other sorting algorithms like [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) can also be modified to get the K largest elements.

**2. Using temporary array:**

Follow the below steps to solve the problem:

1. Store the first K elements in a temporary array temp[0..K-1]
2. Find the smallest element in temp[], and let the smallest element be min
3. For each element x in arr[K] to arr[N-1]. If x is greater than the min, remove min from temp[] and insert x
4. Then, determine the new min from temp[]
5. Print final K elements of temp[]

**Time Complexity:** O((N – K) \* K). If we want the output sorted then O((N – K) \* K + K \* log(K))

Thanks to nesamani1822 for suggesting this method.

**K largest(or smallest) elements in an array using sorting:**

To solve the problem follow the below idea:

*We can sort the input array in descending order so that the first K elements in the array are the K largest elements*

Follow the below steps to solve the problem:

1. Sort the elements in descending order
2. Print the first K numbers of the sorted array

Below is the implementation of the above approach:

1. C++
2. C
3. Java
4. Python
5. C#
6. PHP
7. Javascript

''' Python3 code for k largest elements in an array'''

**def** kLargest(arr, k):

    # Sort the given array arr in reverse

    # order.

    arr.sort(reverse**=**True)

    # Print the first kth largest elements

**for** i **in** range(k):

**print**(arr[i], end**=**" ")

# Driver code

arr **=** [1, 23, 12, 9, 30, 2, 50]

# n = len(arr)

k **=** 3

kLargest(arr, k)

# This code is contributed by shreyanshi\_arun.

**Output**

50 30 23

**Time complexity:** O(N \* log(N))

**Auxiliary Space:**O(1)

[](https://practice.geeksforgeeks.org/courses/complete-interview-preparation?utm_source=article&utm_medium=article&utm_campaign=complete-interview-preparation)

**Efficient Approaches:** To solve the problem follow the below ideas:

**1. Using**[Max-Heap](https://www.geeksforgeeks.org/difference-between-min-heap-and-max-heap/)**:**

Follow the below steps to solve the problem:

1. Build a Max Heap
2. Use Extract Max K times to get K maximum elements from the Max Heap

**Time complexity:** O(N \* log(N) + K \* log(N))

**2. Using order Statistics:**

Follow the below steps to solve the problem:

1. Use an order statistic algorithm to find the Kth largest element. Please [see the topic selection in worst-case linear time](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/)
2. Use the [QuickSort](https://www.geeksforgeeks.org/quick-sort/)Partition algorithm to partition around the Kth largest number
3. Sort the K-1 elements (elements greater than the Kth largest element)  
   **Note:**This step is needed only if the sorted output is required

**Time complexity:** O(N) if we don’t need the sorted output, otherwise O(N + K \* log(K))

Thanks to Shilpi for suggesting the first two approaches.

**K largest(or smallest) elements in an array using Min-Heap:**

To solve the problem follow the below idea:

*We can create a Min-Heap of size K and then compare the root of the Min-Heap with other elements and if it is greater than the root, then swap the value of the root and heapify the heap. This will help us to get the K largest elements in the end*

Follow the below steps to solve the problem:

1. Build a Min Heap MH of the first K elements (arr[0] to arr[K-1]) of the given array
2. For each element, after the Kth element (arr[K] to arr[N-1]), compare it with the root of MH
3. If the element is greater than the root then make it root and call [heapify](https://www.geeksforgeeks.org/binary-heap/)for MH
4. Else ignore it
5. Finally, MH has the K largest elements, and the root of the MH is the Kth largest element

**Note:**All of the above methods can also be used to find the kth smallest elements

Below is the implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#
5. Javascript

# Python3 program for the above approach

# importing heapq module

# to implement heap

**import** heapq as hq

**def** FirstKelements(arr, size, k):

    # Creating Min Heap for given

    # array with only k elements

    # Create min heap using heapq module

    minHeap **=** []

**for** i **in** range(k):

        minHeap.append(arr[i])

    hq.heapify(minHeap)

    # Loop For each element in array

    # after the kth element

**for** i **in** range(k, size):

        # If current element is smaller

        # than minimum ((top element of

        # the minHeap) element, do nothing

        # and continue to next element

**if** minHeap[0] > arr[i]:

**continue**

        # Otherwise Change minimum element

        # (top element of the minHeap) to

        # current element by polling out

        # the top element of the minHeap

**else**:

              # deleting top element of the min heap

            minHeap[0] **=** minHeap[**-**1]

            minHeap.pop()

            minHeap.append(arr[i])

            # maintaining heap again using

            # O(n) time operation....

            hq.heapify(minHeap)

    # Now min heap contains k maximum

    # elements, Iterate and print

**for** i **in** minHeap:

        print(i, end**=**" ")

# Driver code

arr **=** [11, 3, 2, 1, 15, 5, 4, 45, 88, 96, 50, 45]

size **=** len(arr)

# Size of Min Heap

k **=** 3

FirstKelements(arr, size, k)

'''Code is written by Rajat Kumar.....'''

**Output**

50 88 96

**Time Complexity:**O(N \* log K)

**Auxiliary Space:**O(K)

**K largest(or smallest) elements in an array using**[**Quick Sort partitioning algorithm**](https://www.geeksforgeeks.org/quick-sort/)**:**

To solve the problem follow the below idea:

*We will find the pivot in the array until pivot element index is equal to K, because in the quick sort partioning algorithm all the elements less than pivot are on the left side of the pivot and greater than or equal to that are on the right side. So we can print the array (low to pivot to get K-smallest elements and (N-pivot\_Index) to N for K-largest elements)*

Follow the below steps to solve the problem:

1. Choose a pivot number
2. if K is lesser than the pivot\_Index then repeat the step
3. if K is equal to pivot\_Index: Print the array (low to pivot to get K-smallest elements and (n-pivot\_Index) to n for K-largest elements)
4. if  K is greater than pivot\_Index: Repeat the steps for the right part

**Note:**We can improve on the standard quicksort algorithm by using the random() function. Instead of using the pivot element as the last element, we can randomly choose the pivot element randomly.

Below is the implementation of the above approach:

1. C++
2. C
3. Java
4. Python3
5. C#
6. Javascript

# Python3 program for the above approach

**import** random

**def** kthSmallest(arr, l, r, K, n):

    # If k is smaller than number of

    # elements in array

**if** (K > 0 **and** K <**=** r **-** l **+** 1):

        # Partition the array around last

        # element and get position of pivot

        # element in sorted array

        pos **=** partition(arr, l, r)

        # If position is same as k

**if** (pos **-** l **==** K **-** 1):

**return**

**if** (pos **-** l > K **-** 1):  # If position is more,

                              # recur for left subarray

**return** kthSmallest(arr, l, pos **-** 1, K, n)

        # Else recur for right subarray

**return** kthSmallest(arr, pos **+** 1, r,

                           K **-** pos **+** l **-** 1, n)

    # If k is more than number of

    # elements in array

    print("Invalid value of K")

**def** KthLargest(arr, l, r, K, N):

    # This function arranges k Largest elements in last k positions

    #   It means it arranges N-K-1 smallest elements from starting

    kthSmallest(arr, l, r, N **-** K **-** 1, N)

# Standard partition process of QuickSort().

# It considers the last element as pivot and

# moves all smaller element to left of it

# and greater elements to right

**def** partition(arr, l, r):

    x **=** arr[r]

    i **=** l

**for** j **in** range(l, r):

**if** (arr[j] <**=** x):

            arr[i], arr[j] **=** arr[j], arr[i]

            i **+=** 1

    arr[i], arr[r] **=** arr[r], arr[i]

**return** i

# Driver code

a **=** [11, 3, 2, 1, 15, 5, 4, 45, 88, 96, 50, 45]

n **=** len(a)

low **=** 0

high **=** n **-** 1

# assume k is 3

k **=** 3

#  Function call

#  For Smallest

kthSmallest(a, 0, n **-** 1, k, n)

#  Print KSmallest no.

**if** (k >**=** 1 **and** k <**=** n):

    print(str(k) **+** " smallest elements are :", end**=**" ")

**for** i **in** range(k):

**print**(a[i], end**=**" ")

**print**()

#  For Largest

KthLargest(a, 0, n**-**1, k, n)

#  Print KLargest no.

**if** (k >**=** 1 **and** k <**=** n):

    print(str(k) **+** " largest elements are :", end**=**" ")

**for** i **in** range(n **-** 1, n**-**k**-**1, **-**1):

**print**(a[i], end**=**" ")

# This code is contributed by shubhamm050402

**Output**

3 smallest elements are : 3 2 1   
3 largest elements are : 96 50 88

**Time Complexity:**O(N log N)

**Auxiliary Space:**O(1)

**K largest(or smallest) elements in an array using priority queue library:**

To solve the problem follow the below idea:

*Priority queue can be used in the Min-Heap method above to get the K largest or smallest elements*

Follow the below steps to solve the problem:

1. Push the first Kelements into the priority queue from the array
2. After comparing the top of the priority queue with the current array element, we will pop the element at the top of priority\_queue and insert the element.
3. In the case of the K largest element, the priority\_queue will be in increasing order, and thus top most element will be the smallest so we are removing it
4. Similarly, in the case of the K smallest element, the priority\_queue is in decreasing order and hence the topmost element is the largest one so we will remove it
5. In this fashion whole array is traversed and the priority queue of size K is printed which contains the K largest/smallest elements

Below is the implementation of the above approach:

1. C++
2. Java
3. Python3

# Python code for k largest/ smallest elements in an array

**import** heapq

# Function to find k largest array element

**def** kLargest(v, N, K):

    # Implementation using

    # a Priority Queue

    pq **=** []

    heapq.heapify(pq)

**for** i **in** range(N):

        # Insert elements into

        # the priority queue

        heapq.heappush(pq, v[i])

        # If size of the priority

        # queue exceeds k

**if** (len(pq) > K):

            heapq.heappop(pq)

    # Print the k largest element

**while**(len(pq) !**=** 0):

        print(heapq.heappop(pq), end**=**' ')

    print()

# Function to find k smallest array element

**def** kSmalest(v,  N, K):

    # Implementation using

    # a Priority Queue

    pq **=** []

**for** i **in** range(N):

        # Insert elements into

        # the priority queue

        heapq.heappush(pq, **-**1**\***v[i])

        # If size of the priority

        # queue exceeds k

**if** (len(pq) > K):

            heapq.heappop(pq)

    # Print the k largest element

**while**(len(pq) !**=** 0):

**print**(heapq.heappop(pq)**\*-**1, end**=**' ')

    print()

# driver program

# Given array

arr **=** [11, 3, 2, 1, 15, 5, 4, 45, 88, 96, 50, 45]

# Size of array

n **=** len(arr)

k **=** 3

print(k, " largest elements are : ", end**=**'')

kLargest(arr, n, k)

print(k, " smallest elements are : ", end**=**'')

kSmalest(arr, n, k)

# This code is contributed by Abhijeet Kumar(abhijeet19403)

**Output**

3 largest elements are : 50 88 96   
3 smallest elements are : 3 2 1

**Time Complexity:** O(N \* log(K))

**Auxiliary Space:** O(K)

**K largest(or smallest) elements in an array by creating a**[**BST**](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)**and Getting K greatest Elements:**

To solve the problem follow the below idea:

*We can create a BST of the given array elements and then print the K greatest/smallest elements*

Follow the below steps to solve the problem:

1. We will create a Binary Search Tree
2. Then traverse the BST in reverse inorder fashion for K times
3. Print the K largest elements

Below is the implementation of the above approach:

1. C++
2. Java
3. Javascript

// C++ program for the above approach

#include <bits/stdc++.h>

**using namespace** std;

**struct** Node {

**int** data;

**struct** Node\* left;

**struct** Node\* right;

};

**class** Tree {

**public**:

    Node\* root = NULL;

**void** addNode(**int** data)

    {

        Node\* newNode = **new** Node();

        newNode->data = data;

**if** (!root) {

            root = newNode;

        }

**else** {

            Node\* cur = root;

**while** (cur) {

**if** (cur->data > data) {

**if** (cur->left) {

                        cur = cur->left;

                    }

**else** {

                        cur->left = newNode;

**return**;

                    }

                }

**else** {

**if** (cur->right) {

                        cur = cur->right;

                    }

**else** {

                        cur->right = newNode;

**return**;

                    }

                }

            }

        }

    }

**void** printGreatest(**int**& K, vector<**int**>& sol, Node\* node)

    {

**if** (!node || K == 0)

**return**;

        printGreatest(K, sol, node->right);

**if** (K <= 0)

**return**;

        sol.push\_back(node->data);

        K--;

        printGreatest(K, sol, node->left);

    }

};

**class** Solution {

**public**:

    vector<**int**> kLargest(**int** arr[], **int** n, **int** k)

    {

        vector<**int**> sol;

        Tree tree = Tree();

**for** (**int** i = 0; i < n; i++) {

            tree.addNode(arr[i]);

        }

        tree.printGreatest(k, sol, tree.root);

**return** sol;

    }

};

// Driver code

**int** main()

{

**int** n = 5, k = 2;

**int** arr[] = { 12, 5, 787, 1, 23 };

    Solution ob;

**auto** ans = ob.kLargest(arr, n, k);

    cout << "Top " << k << " Elements: ";

**for** (**auto** x : ans) {

        cout << x << " ";

    }

    cout << "\n";

**return** 0;

}

**Output**

Top 2 Elements: 787 23

**Time Complexity:** O(N \* log(N)) + O(K) ~= O(N \* log(N)) (here making of Binary Search Tree from array take n\*log(n) time + O(n) time for finding top k element)

**Auxiliary Space:** O(N) (to store the tree with N nodes we need O(N) space + O(K) space for storing the top K element to print)

*From <*[*https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/*](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)*>*

**K’th Smallest/Largest Element in Unsorted Array**

1. Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
2. Last Updated : 26 Dec, 2022
3. Read
4. Discuss(221)
5. Courses
6. Practice
7. Video

Given an array and a number **K** where **K** is smaller than the size of the array. Find the K’th smallest element in the given array. Given that all array elements are distinct.

**Examples:**

***Input****: arr[] = {7, 10, 4, 3, 20, 15}, K = 3*

***Output****: 7*

***Input****: arr[] = {7, 10, 4, 3, 20, 15}, K = 4*

***Output****: 10*

We have discussed a similar [problem to print k largest elements](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/).

Recommended Problem

Kth smallest element

[Arrays](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Arrays&sortBy=submissions)

[Searching](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Searching&sortBy=submissions)

+3 more

[ABCO](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=ABCO&sortBy=submissions)

[Accolite](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Accolite&sortBy=submissions)

+8 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/kth-smallest-element5635/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 4L

[](https://practice.geeksforgeeks.org/courses/dsa-self-paced)

**K’th smallest element in an unsorted array using sorting:**

*Sort the given array and return the element at index K-1 in the sorted array.*

Follow the given steps to solve the problem:

1. Sort the input array in the increasing order
2. Return the element at the K-1 index (0 – Based indexing) in the sorted array

Below is the Implementation of the above approach:

1. C
2. C++
3. Java
4. Python3
5. C#
6. PHP
7. Javascript

# Python3 program to find K'th smallest

# element

# Function to return K'th smallest

# element in a given array

**def** kthSmallest(arr, N, K):

    # Sort the given array

    arr.sort()

    # Return k'th element in the

    # sorted array

**return** arr[K**-**1]

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [12, 3, 5, 7, 19]

    N **=** len(arr)

    K **=** 2

    # Function call

    print("K'th smallest element is",

          kthSmallest(arr, N, K))

# This code is contributed by

# Shrikant13

**Output**

K'th smallest element is 5

**Time Complexity:**O(N log N)

**Auxiliary Space:**O(1)

**K’th smallest element in an unsorted array using set data structure:**

*Set data structure can be used to find the kth smallest element as it stores the distinct elements in sorted order. Set can be used because it is mentioned in the question that all the elements in the array are distinct.*

Follow the given steps to solve the problem:

1. Insert all array elements into the set
2. Advance the iterator to the Kth element in the set
3. Return the value of the element at which the iterator is pointing

Below is the Implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#

# Python3 code for the above approach

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [12, 3, 5, 7, 19]

    N **=** len(arr)

    K **=** 4

    s **=** set(arr)

**for** itr **in** s:

**if** K **==** 1:

            print(itr)  # itr is the Kth element in the set

**break**

        K **-=** 1

# This code is contributed by Abhijeet Kumar(abhijeet19403)

**Output**

12

**Time Complexity:**  O(N\*log N)

**Auxiliary Space:** O(N)

**K’th smallest element in an unsorted array using heap data structure:**

**K’th smallest element in an unsorted array using Min-Heap**

[*Min-Heap*](https://www.geeksforgeeks.org/heap-data-structure/)*can be used to find the kth smallest element, by inserting all the elements into Min-Heap and then and call extractMin() function K times.*

Follow the given steps to solve the problem:

1. Insert all the array elements into the Min-Heap
2. Call extractMin() function K times
3. Return the value obtained at the last call of extractMin() function

Below is the Implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#
5. Javascript

# Python3 program to find K'th smallest element

# using min heap

# Class for Min Heap

**class** MinHeap:

    # Constructor

**def** \_\_init\_\_(self, a, size):

        # list of elements in the heap

        self.harr **=** a

        # maximum possible size of min heap

        self.capacity **=** None

        # current number of elements in min heap

        self.heap\_size **=** size

        i **=** int((self.heap\_size **-** 1) **/** 2)

**while** i >**=** 0:

            self.minHeapify(i)

            i **-=** 1

**def** parent(self, i):

**return** (i **-** 1) **/** 2

**def** left(self, i):

**return** 2 **\*** i **+** 1

**def** right(self, i):

**return** 2 **\*** i **+** 2

    # Returns minimum

**def** getMin(self):

**return** self.harr[0]

    # Method to remove minimum element (or root)

    # from min heap

**def** extractMin(self):

**if** self.heap\_size **==** 0:

**return** float("inf")

        # Store the minimum value

        root **=** self.harr[0]

        # If there are more than 1 items, move the last item

        # to root and call heapify

**if** self.heap\_size > 1:

            self.harr[0] **=** self.harr[self.heap\_size **-** 1]

            self.minHeapify(0)

        self.heap\_size **-=** 1

**return** root

    # A recursive method to heapify a subtree with root at

    # given index. This method assumes that the subtrees

    # are already heapified

**def** minHeapify(self, i):

        l **=** self.left(i)

        r **=** self.right(i)

        smallest **=** i

**if** ((l < self.heap\_size) **and**

                (self.harr[l] < self.harr[i])):

            smallest **=** l

**if** ((r < self.heap\_size) **and**

                (self.harr[r] < self.harr[smallest])):

            smallest **=** r

**if** smallest !**=** i:

            self.harr[i], self.harr[smallest] **=** (

                self.harr[smallest], self.harr[i])

            self.minHeapify(smallest)

# Function to return k'th smallest element in a given array

**def** kthSmallest(arr, N, K):

    # Build a heap of n elements in O(n) time

    mh **=** MinHeap(arr, N)

    # Do extract min (k-1) times

**for** i **in** range(K **-** 1):

        mh.extractMin()

    # Return root

**return** mh.getMin()

# Driver's code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [12, 3, 5, 7, 19]

    N **=** len(arr)

    K **=** 2

    # Function call

    print("K'th smallest element is", kthSmallest(arr, N, K))

# This Code is contributed by Kevin Joshi

**Output**

K'th smallest element is 5

**Time complexity:** O(N + K Log N).

**Auxiliary Space**: O(N)

**K’th smallest element in an unsorted array using Max-Heap**

[*Max-Heap*](https://www.geeksforgeeks.org/heap-data-structure/)*can be used to find the kth smallest element, by inserting first K elements into Max-Heap and then compare remaining elements with the root of the Max-Heap and if the element is less than the root then remove the root and insert this element into the heap and finally return root of the Max-Heap*

Follow the given steps to solve the problem:

1. Build a Max-Heap MH of the first K elements (arr[0] to arr[K-1]) of the given array.
2. For each element, after the Kth element (arr[K] to arr[n-1]), compare it with the root of MH.
3. If the element is less than the root then make it the root and call heapify for Max-Heap MH
4. b) Else ignore it.
5. Finally, the root of the MH is the Kth smallest element.

Below is the Implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#

# Python3 program to find K'th smallest element

# using max heap

# Class for Max Heap

**class** MaxHeap:

    # Constructor

**def** \_\_init\_\_(self, a, size):

        # list of elements in the heap

        self.harr **=** a

        # maximum possible size of max heap

        self.capacity **=** None

        # current number of elements in max heap

        self.heap\_size **=** size

        i **=** int((self.heap\_size **-** 1) **/** 2)

**while** i >**=** 0:

            self.maxHeapify(i)

            i **-=** 1

**def** parent(self, i):

**return** (i **-** 1) **/** 2

**def** left(self, i):

**return** 2 **\*** i **+** 1

**def** right(self, i):

**return** 2 **\*** i **+** 2

    # Returns maximum

**def** getMax(self):

**return** self.harr[0]

    # to replace root with new node x and heapify() new root

**def** replaceMax(self, x):

        self.harr[0] **=** x

        self.maxHeapify(0)

    # Method to remove maximum element (or root)

    # from max heap

**def** extractMin(self):

**if** self.heap\_size **==** 0:

**return** float("inf")

        # Store the maximum value.

        root **=** self.harr[0]

        # If there are more than 1 items, move the

        # last item to root and call heapify

**if** self.heap\_size > 1:

            self.harr[0] **=** self.harr[self.heap\_size **-** 1]

            self.maxHeapify(0)

        self.heap\_size **-=** 1

**return** root

    # A recursive method to heapify a subtree with root at

    # given index. This method assumes that the subtrees

    # are already heapified

**def** maxHeapify(self, i):

        l **=** self.left(i)

        r **=** self.right(i)

        largest **=** i

**if** ((l < self.heap\_size) **and**

                (self.harr[l] > self.harr[i])):

            largest **=** l

**if** ((r < self.heap\_size) **and**

                (self.harr[r] > self.harr[largest])):

            largest **=** r

**if** largest !**=** i:

            self.harr[i], self.harr[largest] **=** (

                self.harr[largest], self.harr[i])

            self.maxHeapify(largest)

# Function to return k'th smallest element in a given array

**def** kthSmallest(arr, N, K):

    # Build a heap of first k elements in O(k) time

    mh **=** MaxHeap(arr, K)

    # Process remaining n-k elements. If current element is

    # smaller than root, replace root with current element

**for** i **in** range(K, N):

**if** arr[i] < mh.getMax():

            mh.replaceMax(arr[i])

    # Return root

**return** mh.getMax()

# Driver's code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [12, 3, 5, 7, 19]

    N **=** len(arr)

    K **=** 4

    # Function call

**print**("K'th smallest element is", kthSmallest(arr, N, K))

# Code contributed by Kevin Joshi

**Output**

K'th smallest element is 12

**Time Complexity:**O(K + (N-K) \* Log K)

**Auxiliary Space:**O(K)

**K’th smallest element in an unsorted array using QuickSelect:**

*This is an optimization over method 1, if*[*QuickSort*](http://geeksquiz.com/quick-sort/)*is used as a sorting algorithm in first step. In QuickSort, pick a pivot element, then move the pivot element to its correct position and partition the surrounding array. The idea is, not to do complete quicksort, but stop at the point where pivot itself is k’th smallest element. Also, not to recur for both left and right sides of pivot, but recur for one of them according to the position of pivot.*

Follow the given steps to solve the problem:

1. Run quick sort algorithm on the input array
2. In this algorithm pick a pivot element and move it to it’s correct position
3. Now, if index of pivot is equal to K then return the value, else if the index of pivot is greater than K, then recur for the left subarray, else recur for the right subarray
4. Repeat this process until the element at index K is not found

Below is the Implementation of the above approach:

1. C
2. C++
3. Java
4. Python3
5. C#
6. Javascript

# Python3 code for the above approach

# This function returns k'th smallest element

# in arr[l..r] using QuickSort based method.

# ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT

**import** sys

**def** kthSmallest(arr, l, r, K):

    # If k is smaller than number of

    # elements in array

**if** (K > 0 **and** K <**=** r **-** l **+** 1):

        # Partition the array around last

        # element and get position of pivot

        # element in sorted array

        pos **=** partition(arr, l, r)

        # If position is same as k

**if** (pos **-** l **==** K **-** 1):

**return** arr[pos]

**if** (pos **-** l > K **-** 1):  # If position is more,

                              # recur for left subarray

**return** kthSmallest(arr, l, pos **-** 1, K)

        # Else recur for right subarray

**return** kthSmallest(arr, pos **+** 1, r,

                           K **-** pos **+** l **-** 1)

    # If k is more than number of

    # elements in array

**return** sys.maxsize

# Standard partition process of QuickSort().

# It considers the last element as pivot and

# moves all smaller element to left of it

# and greater elements to right

**def** partition(arr, l, r):

    x **=** arr[r]

    i **=** l

**for** j **in** range(l, r):

**if** (arr[j] <**=** x):

            arr[i], arr[j] **=** arr[j], arr[i]

            i **+=** 1

    arr[i], arr[r] **=** arr[r], arr[i]

**return** i

# Driver's Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [12, 3, 5, 7, 4, 19, 26]

    N **=** len(arr)

    K **=** 3

    print("K'th smallest element is",

          kthSmallest(arr, 0, N **-** 1, K))

# This code is contributed by ita\_c

**Output**

K'th smallest element is 5

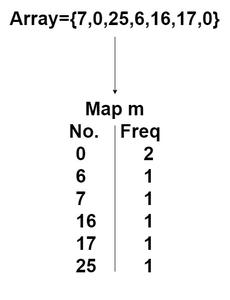
**Time Complexity:**O(N2) in worst case and O(N) on average

**Auxiliary Space:**O(1)

**K’th smallest element in an unsorted array using Map:**

*This approach is very much similar to the QuickSelect and counting sort algorithm but much easier to implement. Use a map and then map each element with its frequency. And as an ordered map would store the data in a sorted manner, so keep on adding the frequency of each element till it does not become greater than or equal to k so that the k’th element from the start can be reached i.e. the k’th smallest element.*

**Example:**A[] = {7, 0, 25, 6, 16, 17, 0}, K = 3



Follow the given steps to solve the problem:

1. Store frequency of every element in a Map mp
2. Now traverse over sorted elements in the Map mp and add their frequencies in a variable freq
3. If at any point the value of freq is greater than or equal to K, then return the value of iterator of Map mp

Below is the Implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#

# Python3 program for the above approach

**def** Kth\_smallest(mp, K):

    freq **=** 0

**for** it **in** sorted(mp.keys()):

        freq **+=** mp[it]  # adding the frequencies of

        # each element

**if** freq >**=** K:  # if at any point frequency becomes

**return** it   # greater than or equal to k then

            # return that element

**return -**1  # returning -1 if k>size of the array which

    # is an impossible scenario

# driver's code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    N **=** 5

    K **=** 2

    arr **=** [12, 3, 5, 7, 19]

    mp **=** {}

**for** i **in** range(N):

**if** arr[i] **in** mp:    # mapping every element with it's

            mp[arr[i]] **=** mp[arr[i]] **+** 1  # frequency

**else**:

            mp[arr[i]] **=** 1

    # Function call

    ans **=** Kth\_smallest(mp, K)

    print("The ", K, "th smallest element is ", ans)

# This code is contributed by Abhijeet Kumar(abhijeet19403)

**Output**

The 2th smallest element is 5

**Time Complexity:**O(N log N)

**Auxiliary Space:**O(N)

**K’th smallest element in an unsorted array using Priority Queue:**

*To find the Kth minimum element in an array, insert the elements into the priority queue until the size of it is less than K, and then compare remaining elements with the root of the priority queue and if the element is less than the root then remove the root and insert this element into the priority queue and finally return root of the priority queue*

Follow the given steps to solve the problem:

1. Build a priority queue of the first K elements (arr[0] to arr[K-1]) of the given array.
2. For each element, after the Kth element (arr[K] to arr[n-1]), compare it with the root of priority queue.
3. If the element is less than the root then remove the root and insert this element into the priority queue
4. b) Else ignore it.
5. Finally, the root of the priority queue is the Kth smallest element.

Below is the Implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#

# Python3 code to implement the approach

**import** heapq

# Function to find the kth smallest array element

**def** kthSmallest(arr, N, K):

    # For finding min element we need (Max heap)priority queue

    pq **=** []

**for** i **in** range(K):

        # First push first K elements into heap

        heapq.heappush(pq, arr[i])

        heapq.\_heapify\_max(pq)

    # Now check from k to last element

**for** i **in** range(K, N):

        # If current element is < first that means

        # there are  other k-1 lesser elements

        # are present at bottom thus, pop that element

        # and add kth largest element into the heap till curr

        # at last all the greater element than kth element will get pop off

        # and at the top of heap there will be kth smallest element

**if** arr[i] < pq[0]:

            heapq.heappop(pq)

            # Push curr element

            heapq.heappush(pq, arr[i])

            heapq.\_heapify\_max(pq)

    # Return first of element

**return** pq[0]

# Driver's code:

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    N **=** 10

    arr **=** [10, 5, 4, 3, 48, 6, 2, 33, 53, 10]

    K **=** 4

    # Function call

**print**("Kth Smallest Element is:", kthSmallest(arr, N, K))

# This code is contributed by Tapesh(tapeshdua420)

**Output**

Kth Smallest Element is: 5

**Time complexity:** O(K log K +  (N – K) log K)

**Auxiliary Space:** O(K)

**K’th smallest element in an unsorted array using Binary Search:**

*The idea to solve this problem is that the Kth smallest element would be the element at the kth position if the array was sorted in increasing order. Using this logic, binary search can be used to predict the index of an element as if the array was sorted but without actually sorting the array.*

Follow the given steps to solve the problem:

1. Find low and high that is the range where our answer can lie.
2. Apply Binary Search on this range.
3. If the selected element which would be mid has less than K elements lesser to it then increase the number that islow = mid + 1.
4. Otherwise, Decrement the high pointer, i.e high = mid
5. The Binary Search will end when only one element remains in the answer space which would be the answer.

Below is the implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#
5. Javascript

# Python3 code for kth smallest element in an array

**import** sys

# function to calculate number of elements

# less than equal to mid

**def** count(nums, mid):

    cnt **=** 0

**for** i **in** range(len(nums)):

**if** nums[i] <**=** mid:

            cnt **+=** 1

**return** cnt

**def** kthSmallest(nums, k):

    low **=** sys.maxsize

    high **= -**sys.maxsize **-** 1

    # calculate minimum and maximum the array.

**for** i **in** range(len(nums)):

        low **=** min(low, nums[i])

        high **=** max(high, nums[i])

        # Our answer range lies between minimum and maximum element

        # of the array on which Binary Search is Applied

**while** low < high:

        mid **=** low **+** (high **-** low) **//** 2

        # if the count of number of elements in the array less than equal

        # to mid is less than k then increase the number. Otherwise decrement

        # the number and try to find a better answer.

**if** count(nums, mid) < k:

            low **=** mid **+** 1

**else**:

            high **=** mid

**return** low

# Driver's code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    nums **=** [1, 4, 5, 3, 19, 3]

    k **=** 3

    # Function call

**print**("K'th smallest element is", kthSmallest(nums, k))

# This code is contributed by Tapesh(tapeshdua420)

**Output**

K'th smallest element is 3

**Time complexity:** O(n \* log (mx-mn)), where mn be minimum and mx be maximum.

**Auxiliary Space:**O(1)

*From <*[*https://www.geeksforgeeks.org/kth-smallest-largest-element-in-unsorted-array/*](https://www.geeksforgeeks.org/kth-smallest-largest-element-in-unsorted-array/)*>*

**Height of a complete binary tree (or Heap) with N nodes**

1. Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
2. Last Updated : 09 Aug, 2022
3. Read
4. Discuss
5. Practice
6. Video
7. Courses

Consider a [Binary Heap](https://www.geeksforgeeks.org/binary-heap/) of size N. We need to find the height of it.

**Examples:**

**Input :** N = 6  
**Output :** 2  
 ()  
 / \  
 () ()  
 / \ /  
 () () ()

**Input :** N = 9  
**Output :** 3  
 ()  
 / \  
 () ()  
 / \ / \  
 () () () ()  
 / \  
() ()

Recommended Problem

Height of Heap

[Heap](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Heap&sortBy=submissions)

[Tree](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Tree&sortBy=submissions)

+1 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/height-of-heap5025/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 17.2K

Let the size of the heap be **N** and the height be **h.**If we take a few examples, we can notice that the value of h in a [complete binary tree](https://www.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/) is floor(log2N).

**Examples:**

N h  
---------  
 1 0  
 2 1  
 3 1  
 4 2  
 5 2  
 .....  
 .....

**Implementation:**

1. C++
2. Java
3. Python 3
4. C#
5. PHP
6. Javascript

# Python 3 program to find

# height of complete binary

# tree from total nodes.

**import** math

**def** height(N):

**return** math.ceil(math.log2(N **+** 1)) **-** 1

# driver node

N **=** 6

print(height(N))

# This code is contributed by

# Smitha Dinesh Semwal

**Output**

1

**Time Complexity: O(1)**, Since performing constant operations.

**Auxiliary Space: O(1)**, Since constant extra space is used.

*From <*[*https://www.geeksforgeeks.org/height-complete-binary-tree-heap-n-nodes/*](https://www.geeksforgeeks.org/height-complete-binary-tree-heap-n-nodes/)*>*

**Heap Sort for decreasing order using min heap**

1. Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
2. Last Updated : 04 Aug, 2022
3. Read
4. Discuss(1)
5. Courses
6. Practice
7. Video

Given an array of elements, sort the array in decreasing order using min heap.

**Examples:**

**Input :** arr[] = {5, 3, 10, 1}  
**Output :** arr[] = {10, 5, 3, 1}

**Input :** arr[] = {1, 50, 100, 25}  
**Output :** arr[] = {100, 50, 25, 1}

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

Prerequisite : [Heap sort](https://www.geeksforgeeks.org/heap-sort/) using [min heap](https://www.geeksforgeeks.org/binary-heap/).

**Algorithm :**

1. Build a min heap from the input data.
2. At this point, the smallest 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.

**Note :**Heap Sort using min heap sorts in descending order where as max heap sorts in ascending order

**Implementation:**

* C++
* Java
* Python3
* C#
* Javascript

# Python3 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

**def** heapify(arr, n, i):

    smallest **=** i # Initialize smallest as root

    l **=** 2 **\*** i **+** 1 # left = 2\*i + 1

    r **=** 2 **\*** i **+** 2 # right = 2\*i + 2

    # If left child is smaller than root

**if** l < n **and** arr[l] < arr[smallest]:

        smallest **=** l

    # If right child is smaller than

    # smallest so far

**if** r < n **and** arr[r] < arr[smallest]:

        smallest **=** r

    # If smallest is not root

**if** smallest !**=** i:

        (arr[i],

         arr[smallest]) **=** (arr[smallest],

                           arr[i])

        # Recursively heapify the affected

        # sub-tree

        heapify(arr, n, smallest)

# main function to do heap sort

**def** heapSort(arr, n):

    # Build heap (rearrange array)

**for** i **in** range(int(n **/** 2) **-** 1, **-**1, **-**1):

        heapify(arr, n, i)

    # One by one extract an element

    # from heap

**for** i **in** range(n**-**1, **-**1, **-**1):

        # Move current root to end #

        arr[0], arr[i] **=** arr[i], arr[0]

        # call min heapify on the reduced heap

        heapify(arr, i, 0)

# A utility function to print

# array of size n

**def** printArray(arr, n):

**for** i **in** range(n):

**print**(arr[i], end **=** " ")

**print**()

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [4, 6, 3, 2, 9]

    n **=** len(arr)

    heapSort(arr, n)

**print**("Sorted array is ")

    printArray(arr, n)

# This code is contributed by PranchalK

**Output:**

Sorted array is   
9 6 4 3 2

**Time complexity:**It takes **O(logn)** for **heapify** and **O(n)** for **constructing a heap**. Hence, the overall time complexity of **heap sort**using**min heap** or**max heap** is **O(nlogn)**

**Space complexity: O(n)**for call stack