الجامعة العربية الأمريكية

فلسطين PALESTINE

2000

ARAB AMERICAN UNIVERSITY

Algorithms Analysis and Design

Heap Sorting Project

Under the supervision of Dr. Thaer Thaher

Yara Jehad Rabaya	202110451
Shereen Shehadeh	202110917
Shyma' khrewesh	202110883
Hala Jabareen	202111358
Malak Abualrob	202111533

Introduction:

Sorting Problem is a foundational challenge in computer science, involving the arrangement of a set of elements in a specific order. The objective is to organize the elements either in ascending or descending sequence based on a predefined rule. Various sorting algorithms address this challenge, each employing unique techniques.

- **Bubble Sort:** This algorithm iterates through the list, comparing adjacent elements and swapping them if they are in the wrong order until the entire list is sorted in ascending order.
- **Insertion Sort:** This algorithm divides the list into two sections, sorted and unsorted, and sequentially moves elements from the unsorted section to the sorted one.
- **Merge Sort:** Based on the divide-and-conquer concept, Merge Sort divides the list and independently sorts the parts before merging them back in a sequential order.
- **Quick Sort:** Quick Sort partitions the list using a pivot element and independently sorts the resulting partitions, achieving overall sorting.
- **Heap Sort:** This algorithm utilizes a binary heap data structure to achieve sorting. It involves building a max heap, repeatedly extracting the maximum element, and reconstructing the heap until the entire list is sorted.

These sorting algorithms play a pivotal role in enhancing program performance and expediting search and data retrieval operations. The choice of the appropriate algorithm depends on factors such as data size and application requirements.

In essence, tackling the Sorting Problem underscores the significance of data organization in computer science, contributing to improved program efficiency and faster data access. As we understand and utilize these algorithms, including Heap Sort, we can achieve effective and organized data sorting, thereby enhancing software quality and efficiency.

Complete Binary Tree:

Tree: is a connected acyclic graph that consists of n nodes and n-1 edges.

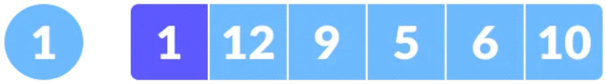
Binary: each node has at most 2 child (it could has 1 or no child).

Complete: all levels completely filled – except the last level.

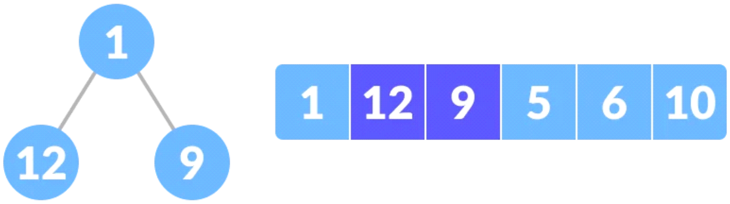
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

It can be stored compactly in an array, with the root at index 1, and the left and right children of node i stored at indices 2i and 2i+1 respectively.

Example:



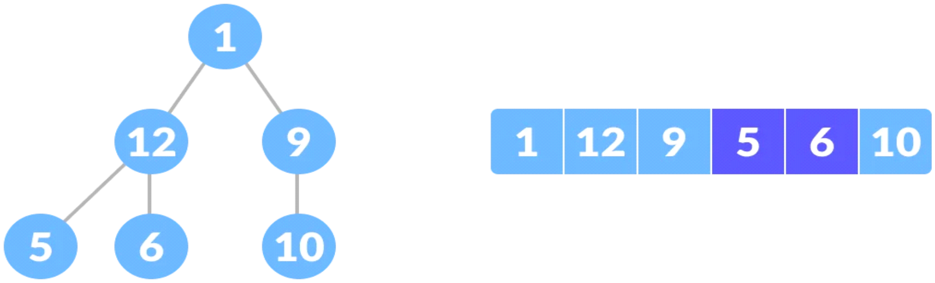
Root =array[1]=1



Index of left child=2 x i=2

Index of right child=(2 x i)+1=3

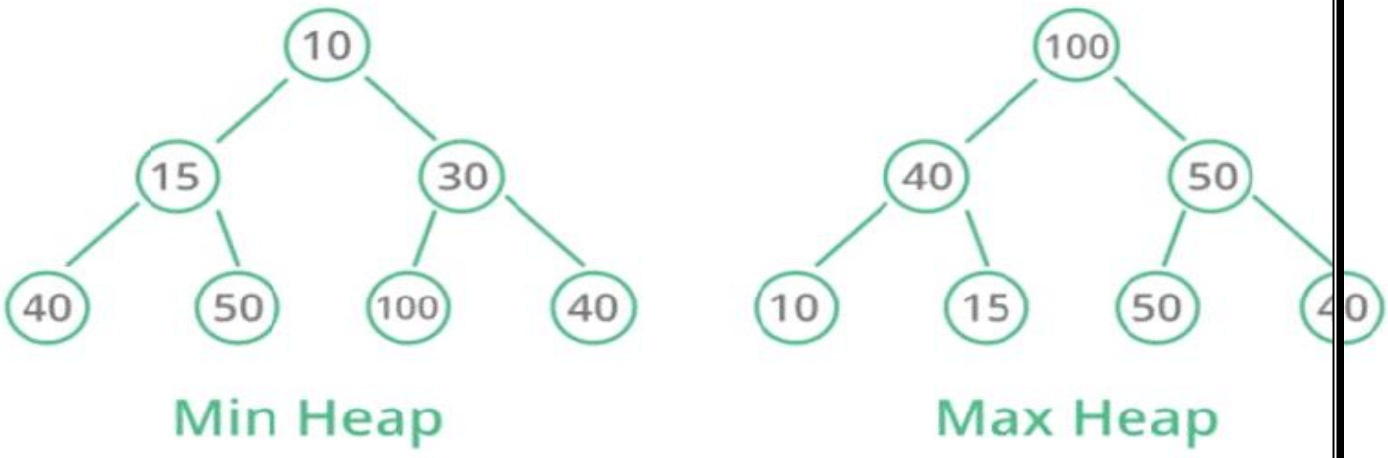
Index of parent=i/2=1



Heap:

A Heap is a fundamental type of data structure in computer science, representing a specialized form of binary trees. It comes in two main patterns: "Max Heap" and "Min Heap." In a Max Heap, each node's value is greater than or equal to the values of its children, with the highest value positioned at the root. Conversely, a Min Heap ensures that each node's value is less than or equal to the values of its children, and the smallest value resides at the root.

Heaps play a crucial role in various algorithms, proving to be efficient in tasks such as sorting, managing priority queues, and even certain graph algorithms. Thanks to their unique structure and properties, heaps are indispensable tools in optimizing data retrieval and storage efficiency.



Heap Sorting technic

The Heap sort algorithm to arrange a list of elements performed using following steps:

Step 1 - Construct a Binary Tree with given list of Elements.

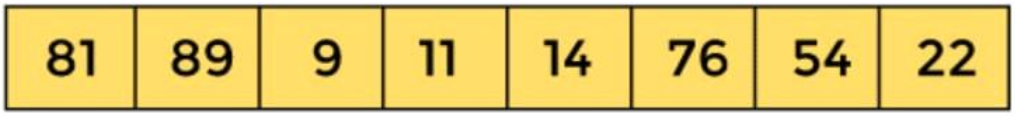
Step 2 - Transform the Binary Tree into Min/Max Heap.

Step 3 - Delete the root element from Min/Max Heap using Heapify method.

Step 4 - Put the deleted element into the Sorted list.

Step 5 - Repeat the same until Min /Max Heap becomes empty. Step 6 - Display the sorted list.

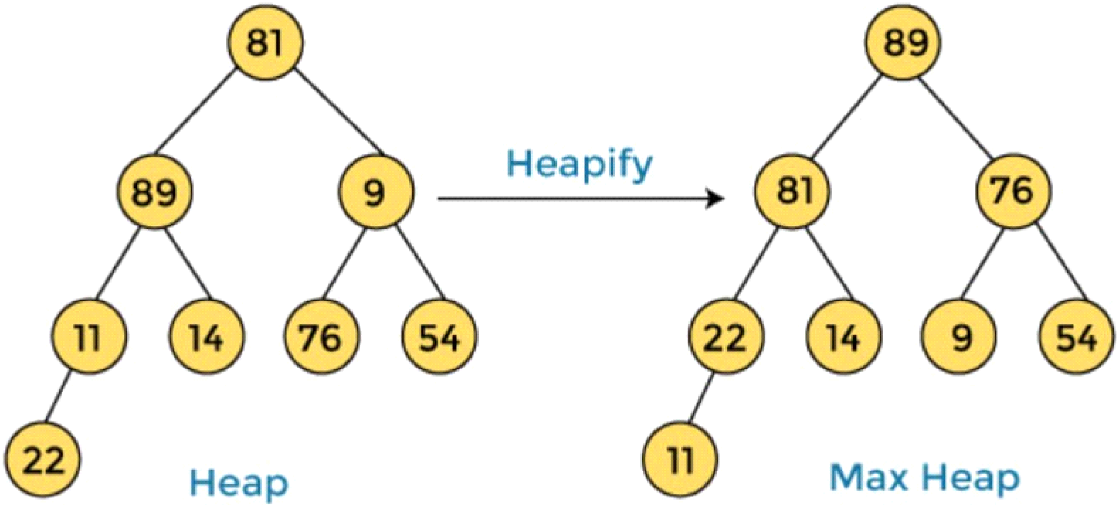
EX : max-heap

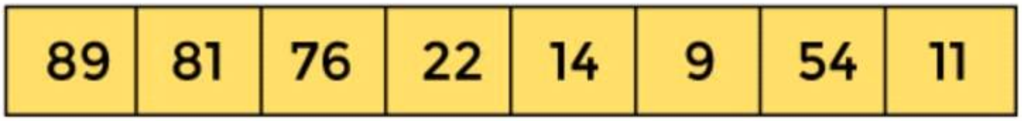


81	89	9	11	14	76	54	22

- First, we have to construct a heap from the given array and convert it into max heap.

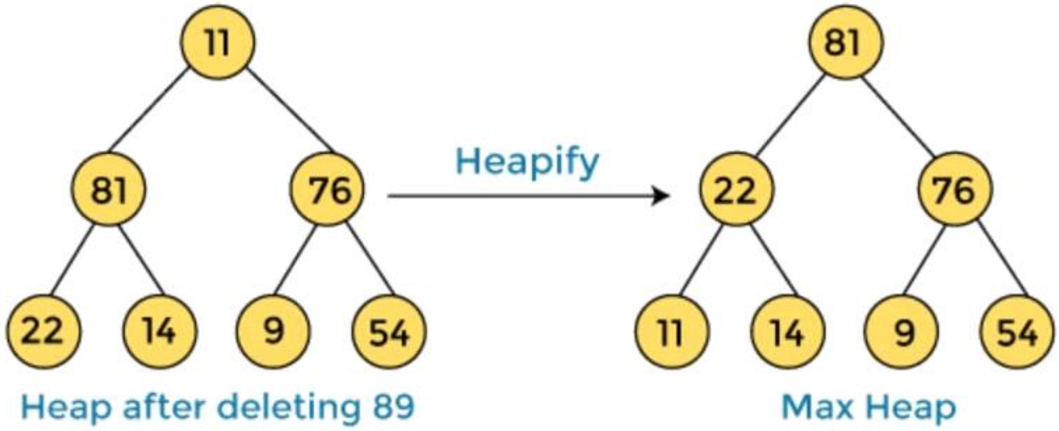
-After converting the given heap into max heap, the array elements are :



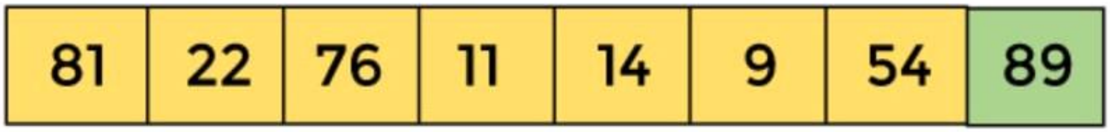


89	81	76	22	14	9	54	11

-Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.

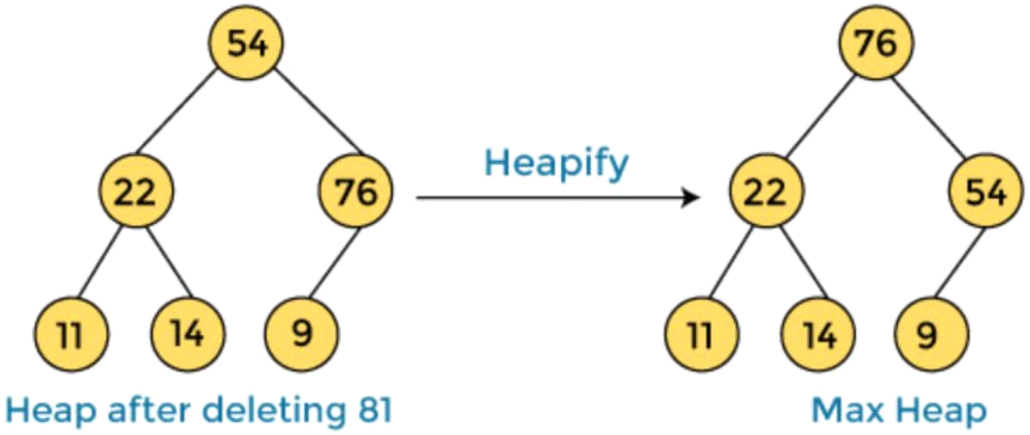


-After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are:

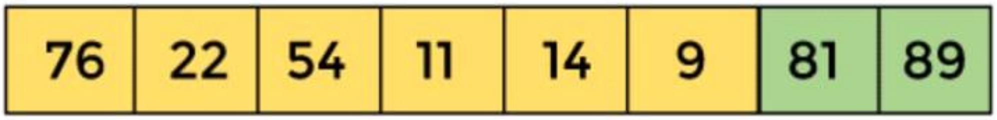


81	22	76	11	14	9	54	89

-In the next step, again, we have to delete the root element (81) from the max heap. To delete this node, we have to swap it with the last node, i.e. (54). After deleting the root element, we again have to heapify it to convert it into max heap.

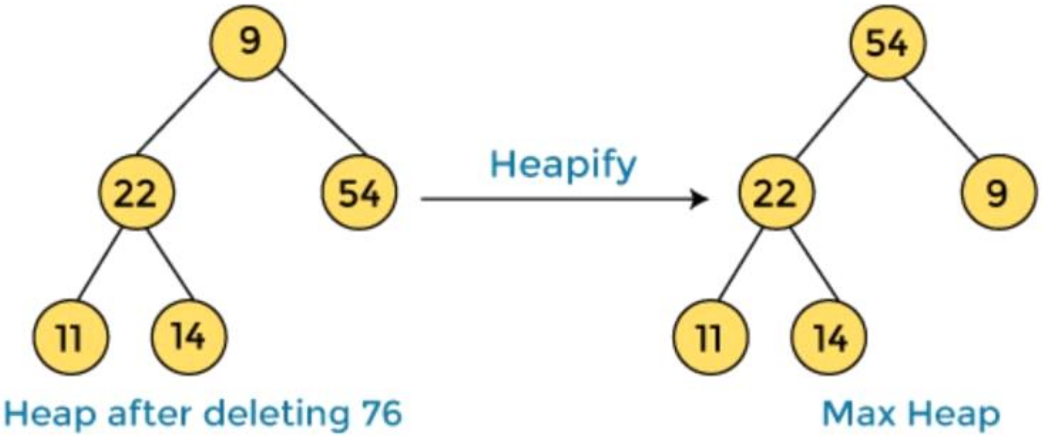


-After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are :

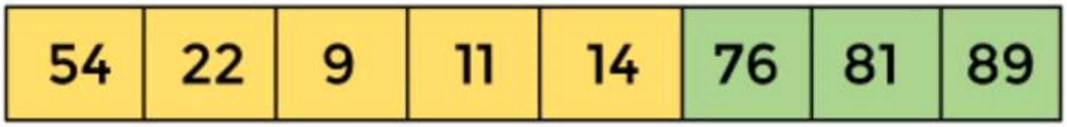


76	22	54	11	14	9	81	89

-In the next step, we have to delete the root element (76) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.

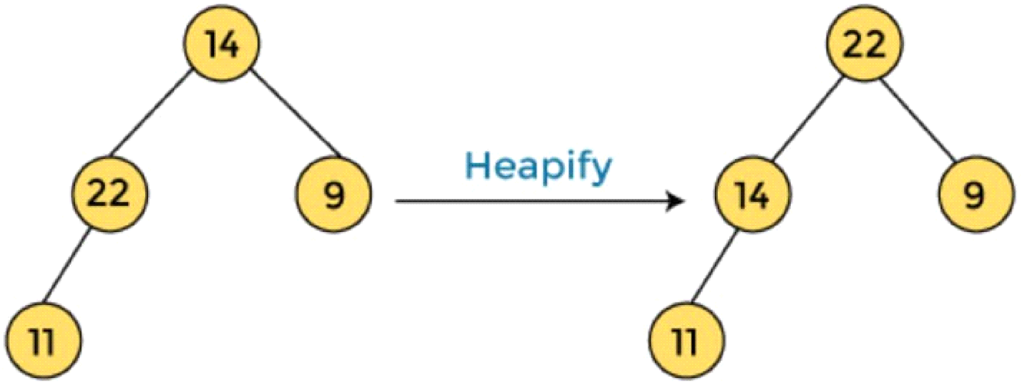


-After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are :



54	22	9	11	14	76	81	89

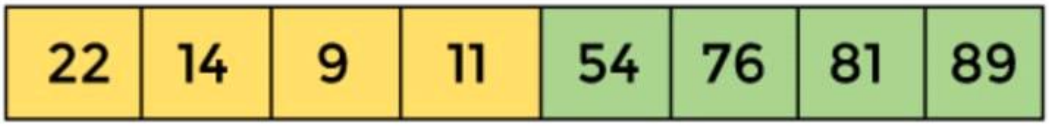
-In the next step, again we have to delete the root element (54) from the max heap. To delete this node, we have to swap it with the last node, i.e. (14). After deleting the root element, we again have to heapify it to convert it into max heap.



Heap after deleting 54

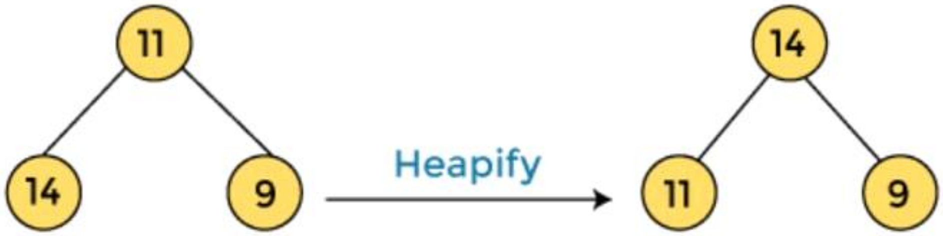
Max Heap

-After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are :



22	14	9	11	54	76	81	89

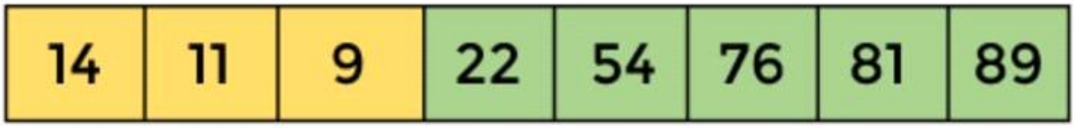
-In the next step, again we have to delete the root element (22) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



Heap after deleting 22

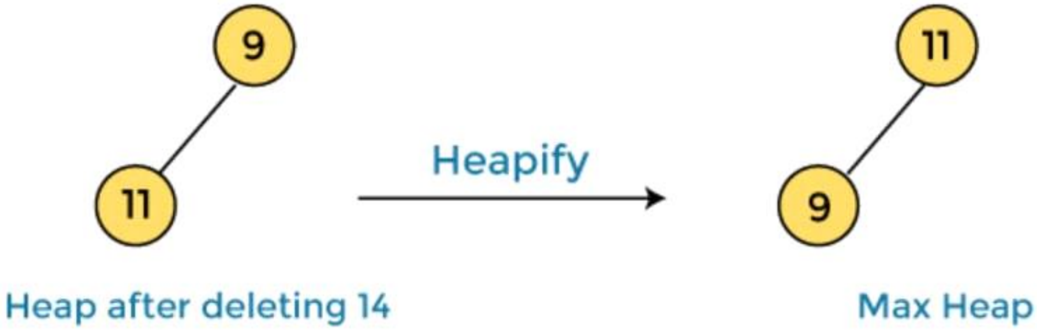
Max Heap

-After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are :

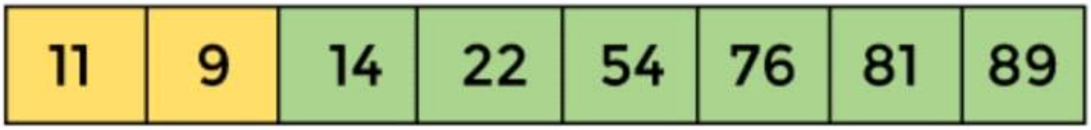


14	11	9	22	54	76	81	89

-In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap

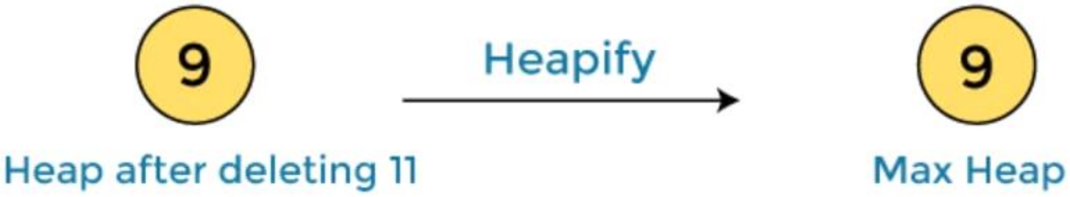


-After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are:

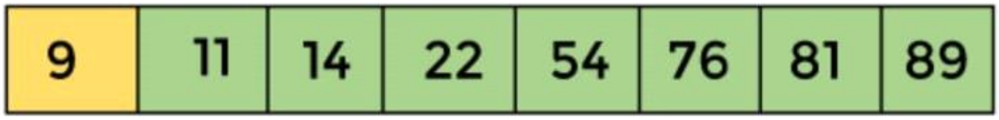


11	9	14	22	54	76	81	89

-In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



-After swapping the array element 11 with 9, the elements of array are:

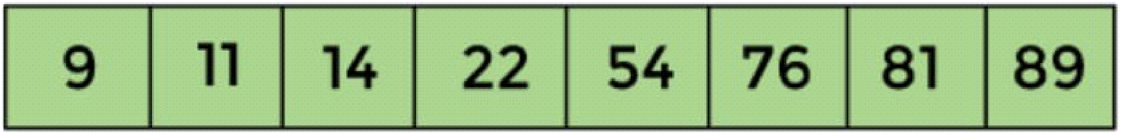


9	11	14	22	54	76	81	89

-Now, heap has only one element left. After deleting it, heap will be empty

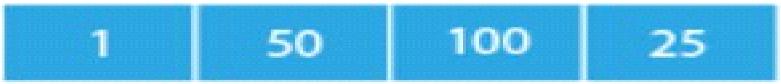
9
Remove 9
Empty

-After completion of sorting, the array elements are:



9	11	14	22	54	76	81	89

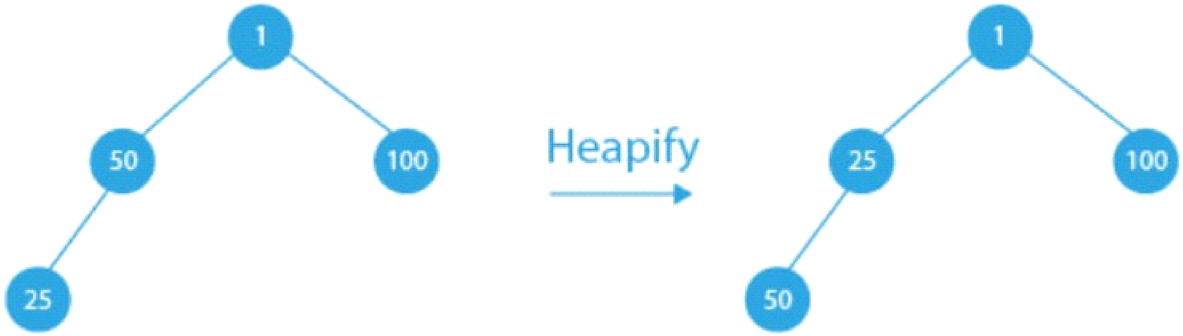
EX: MIN-heap



1	50	100	25

Step 1

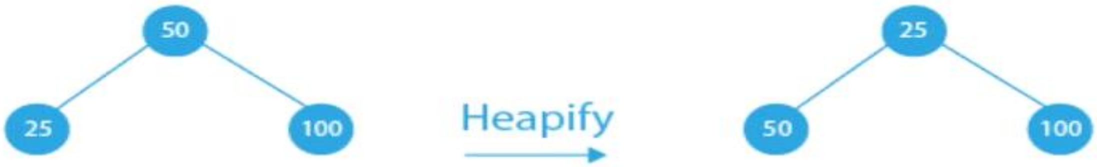
First, construct the heap from the given array & convert it into min heap.



1	25	100	50

Step 2

Delete the root (1) from the min heap. To delete this node, we have to swap it with the last node i.e. (50). After this, we have to heapify it again.



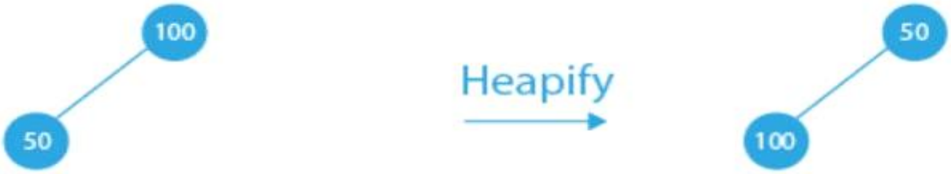
After deleting (1)

Min Heap

25	50	100	1

Step 3

In the next step, we have to delete the root element (25). To delete this, we have to swap it with the last node i.e. (100) & again perform heapify after swapping.



50	100	25	1

Step 4

In the next step, delete the root node (50). To delete this, swap it with the last node i.e.(100).

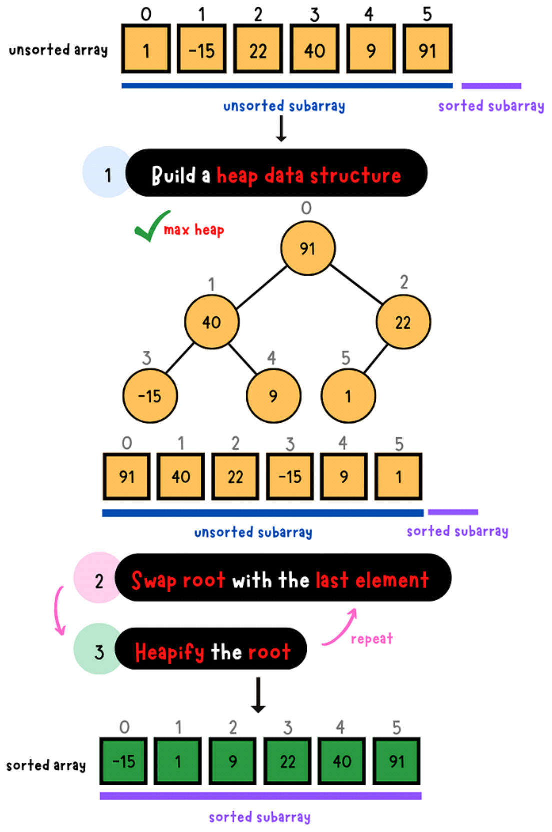
100
Heapify
100

100	50	25	1

Now, heap has left only 1 element, After deleting it heap will be empty and the array is completely sorted:

100	50	25	1

Heap Sort Algorithm



unsorted array: 1 | -15 | 22 | 40 | 9 | 91

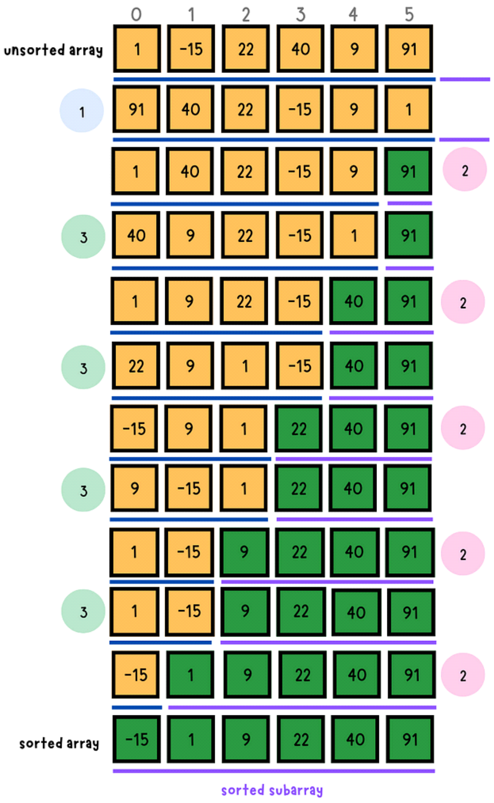
1 Build a heap data structure

91 | 40 | 22 | -15 | 9 | 1

2 Swap root with the last element

3 Heapify the root

sorted array: -15 | 1 | 9 | 22 | 40 | 91



unsorted array: 1 | -15 | 22 | 40 | 9 | 91

1: 91 | 40 | 22 | -15 | 9 | 1

2: 1 | 40 | 22 | -15 | 9 | 91

3: 40 | 9 | 22 | -15 | 1 | 91

2: 1 | 9 | 22 | -15 | 40 | 91

3: 22 | 9 | 1 | -15 | 40 | 91

2: -15 | 9 | 1 | 22 | 40 | 91

3: 9 | -15 | 1 | 22 | 40 | 91

2: 1 | -15 | 9 | 22 | 40 | 91

3: 1 | -15 | 9 | 22 | 40 | 91

2: -15 | 1 | 9 | 22 | 40 | 91

sorted array: -15 | 1 | 9 | 22 | 40 | 91

time complexity: O(nlog(n)) n: the total number of elements in the input array.

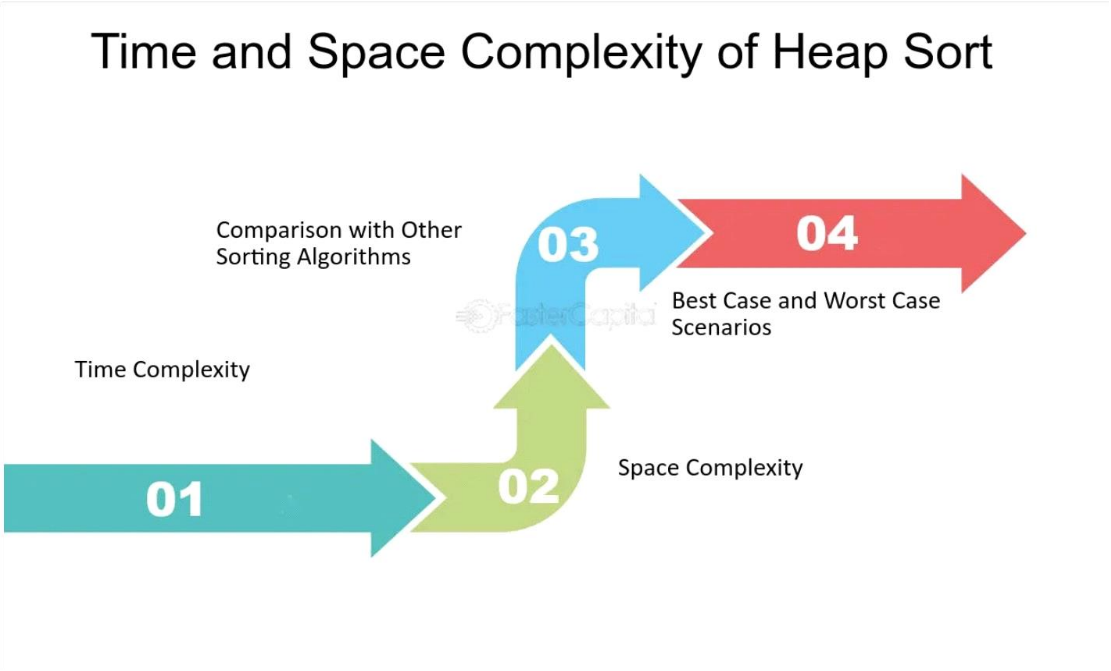
pseudo code:

```
Algorithm heapify(A, i)
1. l = 2*i
2. r = 2*i+1
3. if l <= heapsize[A] and A[l] > A[i]
 then max = l
 else max = i
4. if r <= heapsize[A] and A[r] > A[max]
 then max = r
5. if max != i
 then exchange A[i] with A[max]
 heapify(A, max)
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///////////
Algorithm Build_max_heap(A)
{
 1. heapsize[A] length[A]
 for i =length[A]/2 to 1
 do heapify(A, i)
}
Heapsort(A)
1. Build_max_heap(A)
10
2. for i = length[A] to 1
 do exchange A[1] with A[i]
 heapsize[A] = heapsize[A] - 1
 heapify(A, 1)
}
///////////////////////////////////////////////////////////////////////////
```

Implementation of Heapsort:

```
 heapify(int arr[], int n , int pos) {
 int max = pos;
 int L = 2 * pos + 1;
 int R = 2 * pos + 2;
 if (L < n && arr[L] > arr[max])
 max = L;
 if (R< n && arr[R] > arr[max])
 max = R ;
 if (max != pos) {
 swap( arr[pos] , arr[max]);
 heapify(arr, n, max); }}
//////////////////////////////////////////////////////////////////////
Buildheap (int arr[] , int n ) {
 for (int i = n / 2 - 1; i >= 0; i--)
 heapify(arr, n, i);
heapsort (int i = n - 1; i >= 0; i--) {
 swap(arr[0], arr[i]);
 heapify(arr, i, 0);
```

Time Complexity :


Time and Space Complexity of Heap Sort
Comparison with Other Sorting Algorithms
03
04
Best Case and Worst Case Scenarios
Time Complexity
01
02
Space Complexity

Time Complexity: Heapsort achieves a time complexity of O(n log n), making it an efficient sorting algorithm. The primary contributors to this time complexity are the build max-heap phase and heapify

- **Build Max-Heap:** The build max-heap phase takes O(n) time. As there are log(n) levels in the heap and at each level, a constant number of operations are performed, the total time complexity for building the max-heap is O(n).
- **Heapify:** This phase is repeated n times, and each iteration involves log(n) operations for heapification. Therefore, the time complexity for the extract-max and heapify phase is O(n log n).

- Build Max Heap: O(n)
- Heapify: O(log n)
- Heap Sort: O(n log n)

- **Worst-case:** O(n log n)
- **Average-case:** O(n log n)
- **Best-case:** O(n log n)

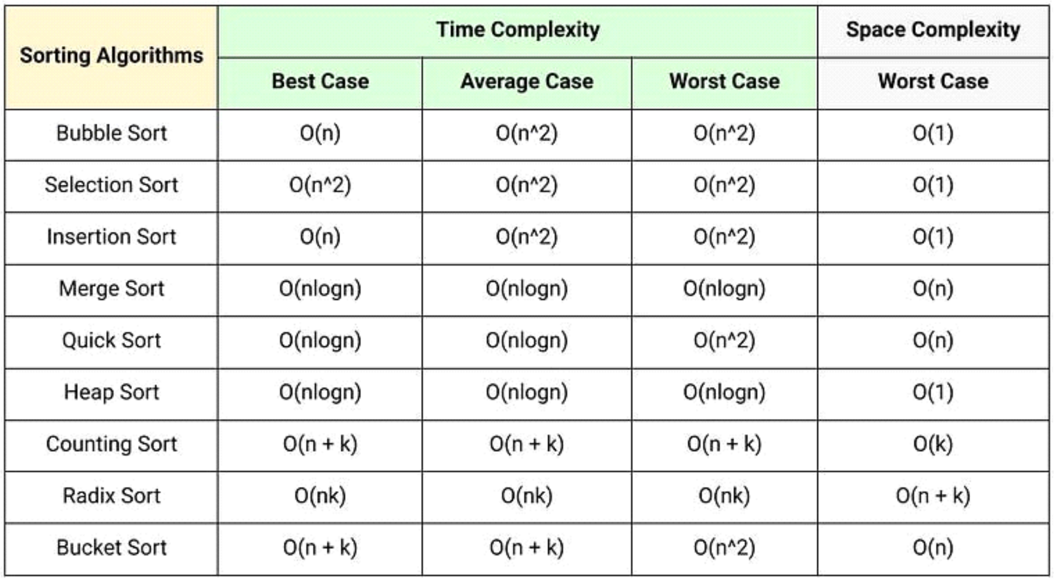
The overall time complexity is the sum of these two phases: O(n) + O(n log n) = O(n log n).

Space Complexity: Heapsort has a space complexity of O(1) as it operates in-place, requiring only a constant amount of additional memory regardless of the input size. The heap is built directly within the input array without the need for additional data structures.

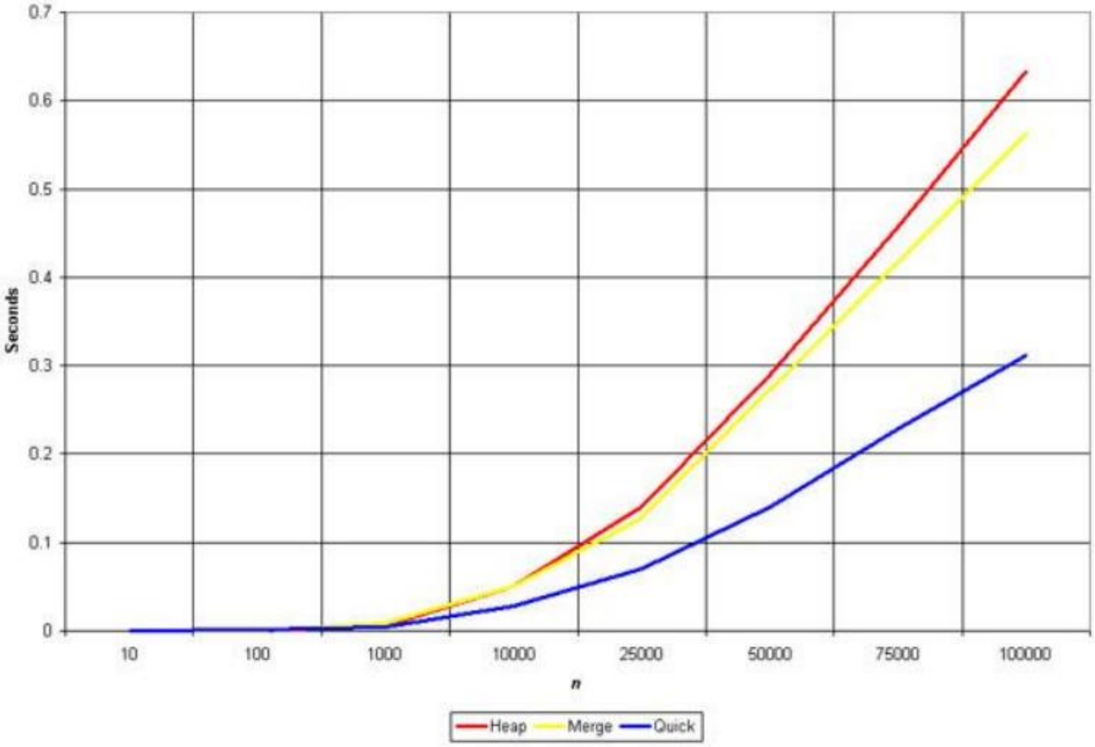
Comparison with Other Sorting Algorithms:

- **Quick Sort:**
    - Heap Sort and Quick Sort both have an average time complexity of O(n log n).
- **Merge Sort:**
    - Both Heap Sort and Merge Sort have a time complexity of O(n log n).
- **Bubble Sort and Insertion Sort:**
    - Heap Sort outperforms Bubble Sort and Insertion Sort in terms of time complexity.

Heap sort is often considered less memory-intensive compared to other O(n log n) algorithms like merge sort or quicksort, making it more suitable for situations where memory is a critical resource.



Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n)



Applications:

1. Sorting large data sets

Heap sort is ideal for sorting large data sets. It has a space complexity of O(1), which means it requires very little memory to execute. This makes it an ideal sorting algorithm for applications where memory is limited. Heap sort is also a stable sorting algorithm, which means it maintains the relative order of equal elements. This is important in applications where the order of equal elements needs to be preserved.

2. Implementing priority queues

Heap sort is also used to implement priority queues. Priority queues are data structures that allow efficient access to the element with the highest priority. In a priority queue, elements are assigned a priority value, and the element with the highest priority value is accessed first. Heap sort is ideal for implementing priority queues because it allows efficient access to the element with the highest priority. The heap data structure used by heap sort is also well-suited for implementing priority queues.

3. Building a Huffman tree

Heap sort can also be used to build a Huffman tree. A Huffman tree is a binary tree used in data compression. It is built by assigning a code to each character in a string based on its frequency of occurrence. The characters with the highest frequency are assigned the shortest codes, and the characters with the lowest frequency are assigned the longest codes. Heap sort is used to build the Huffman tree by creating a min heap of the character frequencies and repeatedly combining the two smallest frequencies until only one node remains.

Heap sort is a versatile sorting algorithm with a variety of applications. It is ideal for sorting large data sets, implementing priority queues and building a Huffman tree. Heap sort's efficiency, stability, and in-place sorting make it a popular choice for a wide range of applications.

Conclusion:

Heap Sort is an efficient comparison-based sorting algorithm that operates by creating a binary heap. This data structure ensures that the element at the root is the maximum (for max heap) or minimum (for min heap) of all elements in the heap. Heap Sort involves building a max heap, repeatedly extracting the maximum element, and reconstructing the heap until the entire list is sorted. Its effectiveness and time complexity make it a valuable addition to our discussion on sorting algorithms

The time complexity of heap sort is always O(n log n), regardless of the input distribution. This is because building the heap takes O(n) time, and the heapify operation (rearranging the heap after extracting the maximum/minimum element) is performed n times, where n is the number of elements in the array.