Heap Data Structure

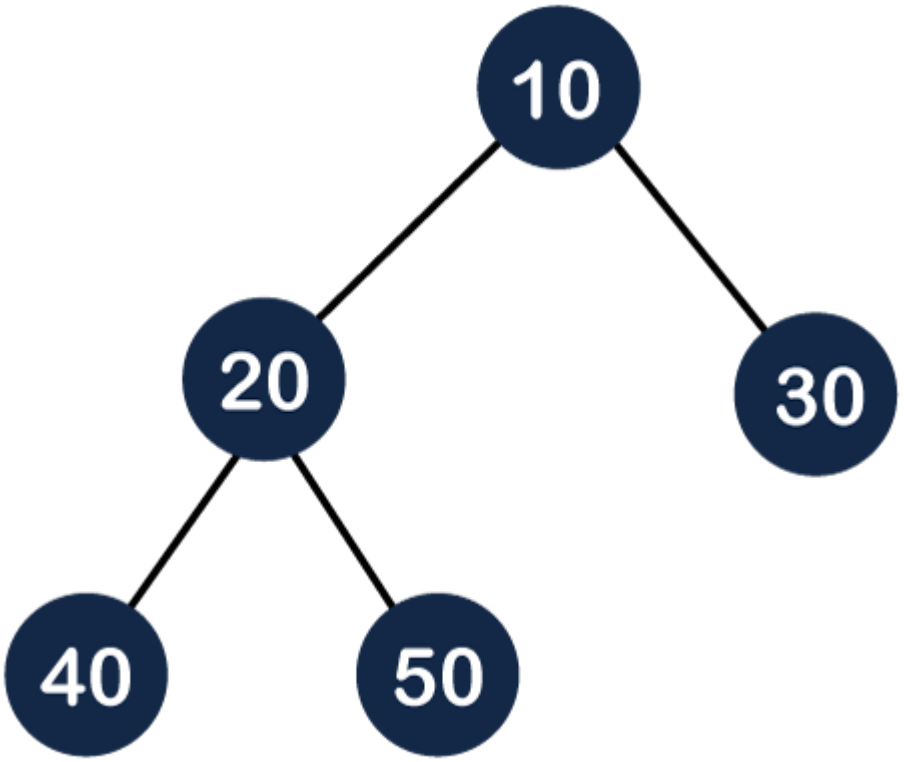
What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

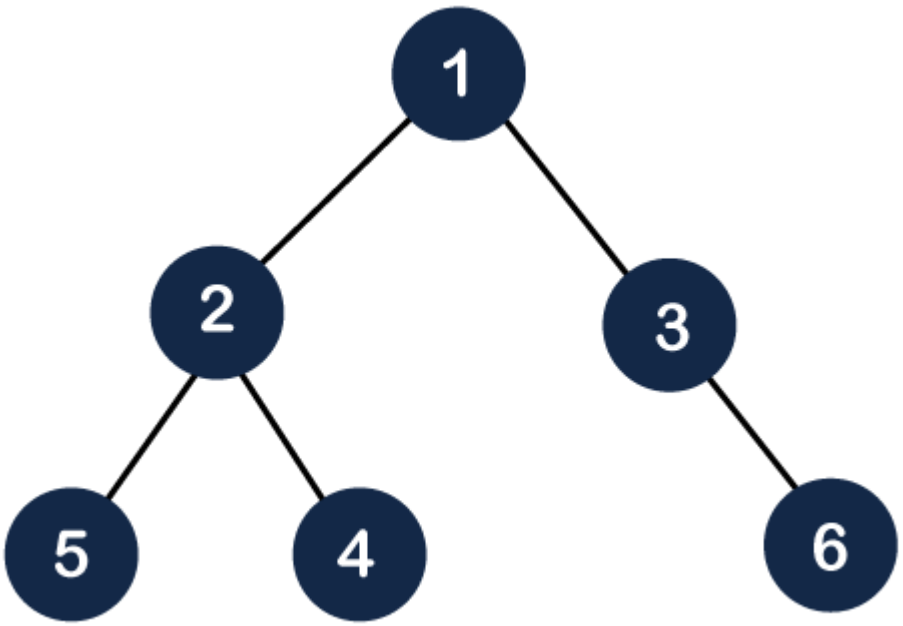
What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

> ***Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.***

How can we arrange the nodes in the Tree?

There are two types of the heap:

- Min Heap
- Max heap

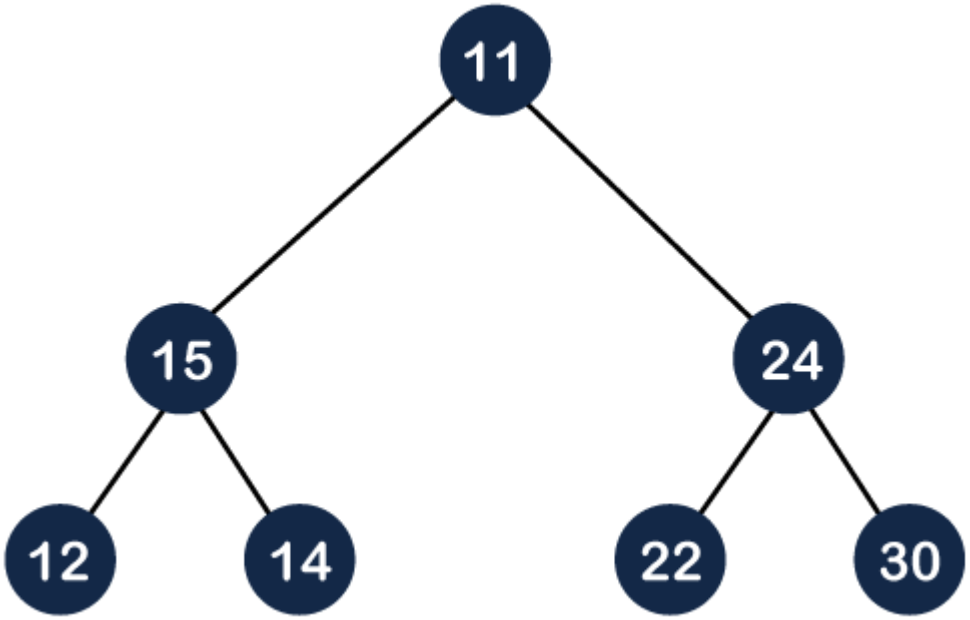
Min Heap: The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] <= A[i]

Let's understand the min-heap through an example.



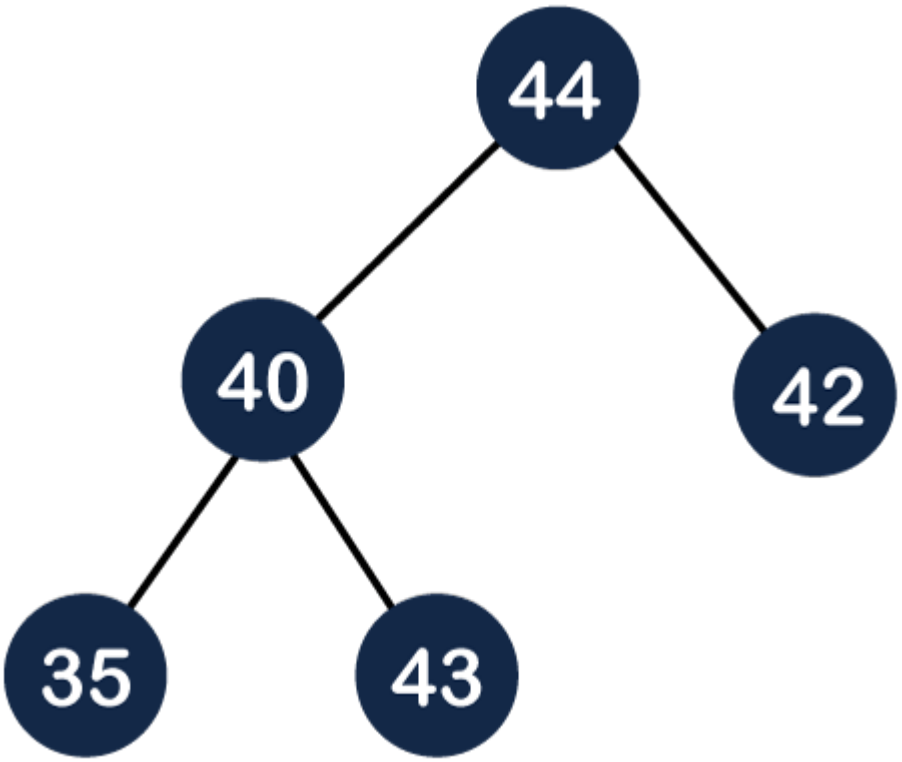
In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Algorithm of insert operation in the max heap.

```
1. // algorithm to insert an element in the max heap.
2. insertHeap(A, n, value)
3. {
4. n=n+1; // n is incremented to insert the new element
5. A[n]=value; // assign new value at the nth position
6. i = n; // assign the value of n to i
7. // loop will be executed until i becomes 1.
8. while(i>1)
9. {
10. parent= floor value of i/2; // Calculating the floor value of i/2
11. // Condition to check whether the value of parent is less than the given node or not
12. if(A[parent]<A[i])
13. {
```

```
14. swap(A[parent], A[i]);
15. i = parent;
16. }
17. else
18. {
19. return;
20. }
21. }
22. }
```

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

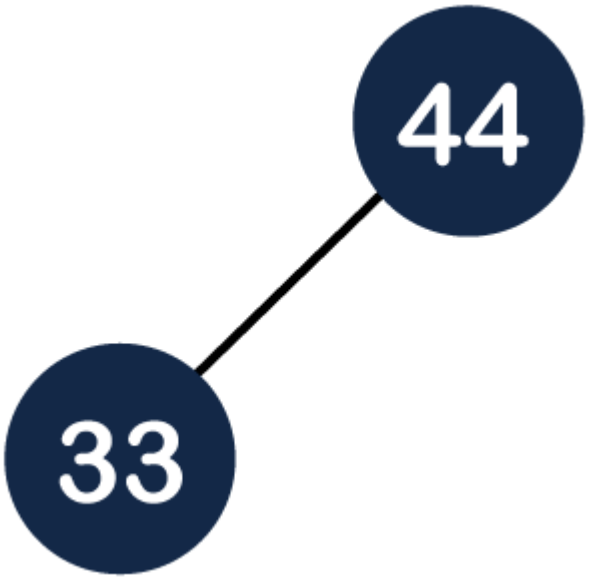
44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

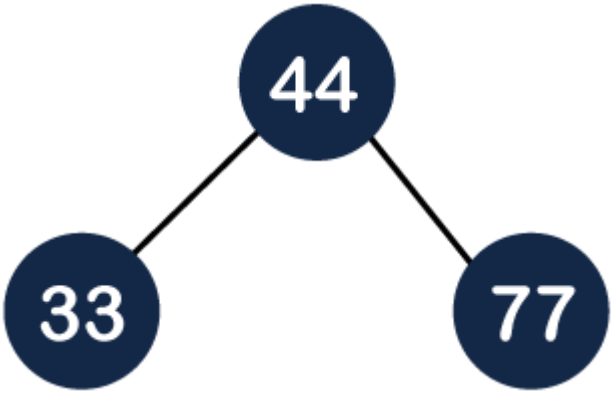
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

Step 1: First we add the 44 element in the tree as shown below:

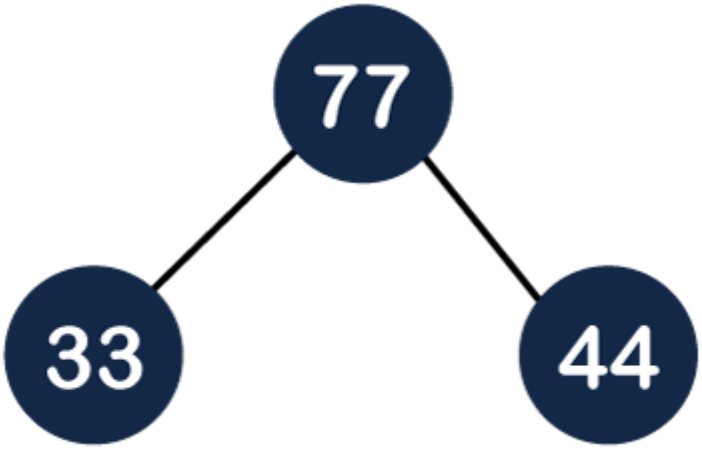
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



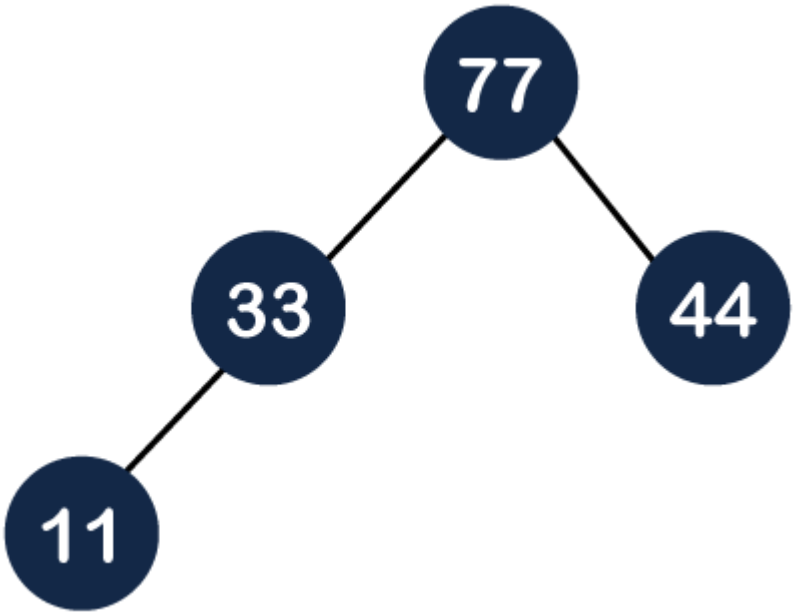
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



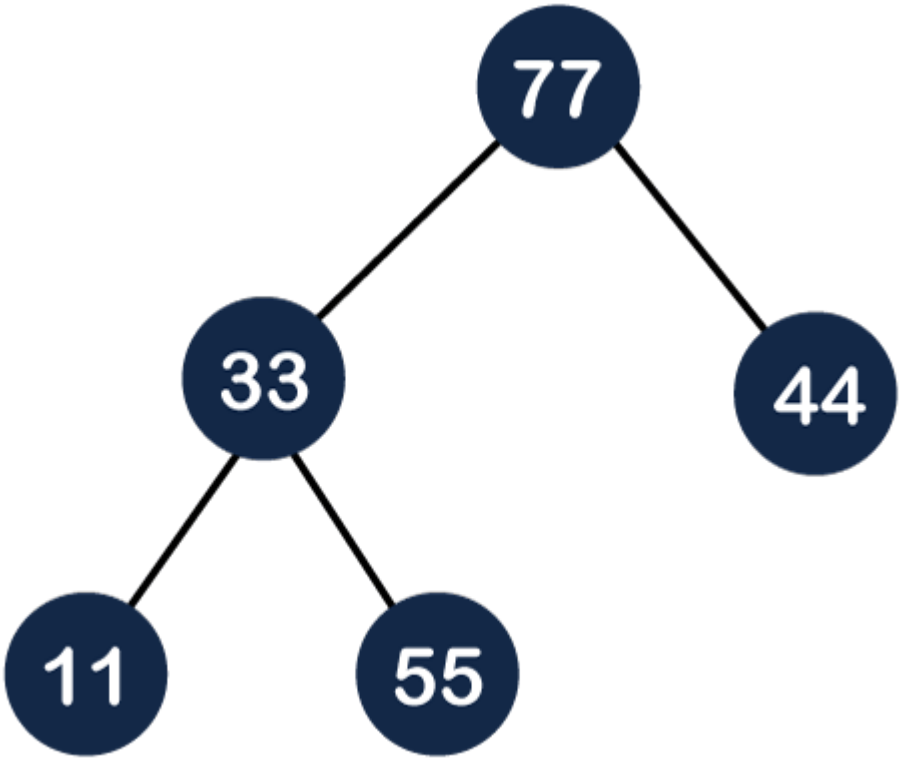
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



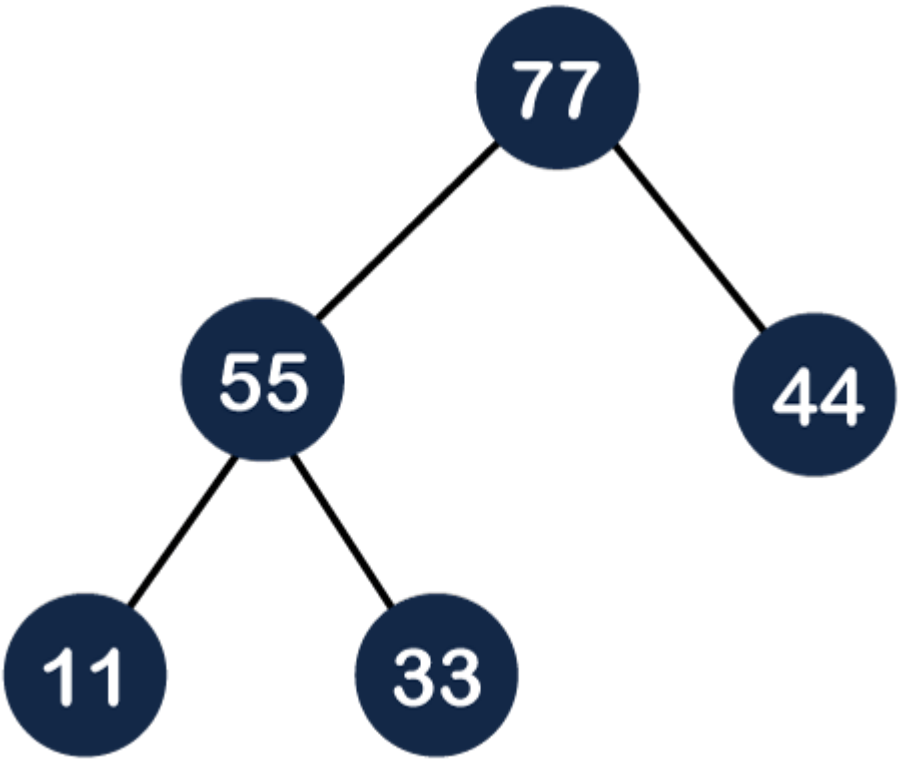
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



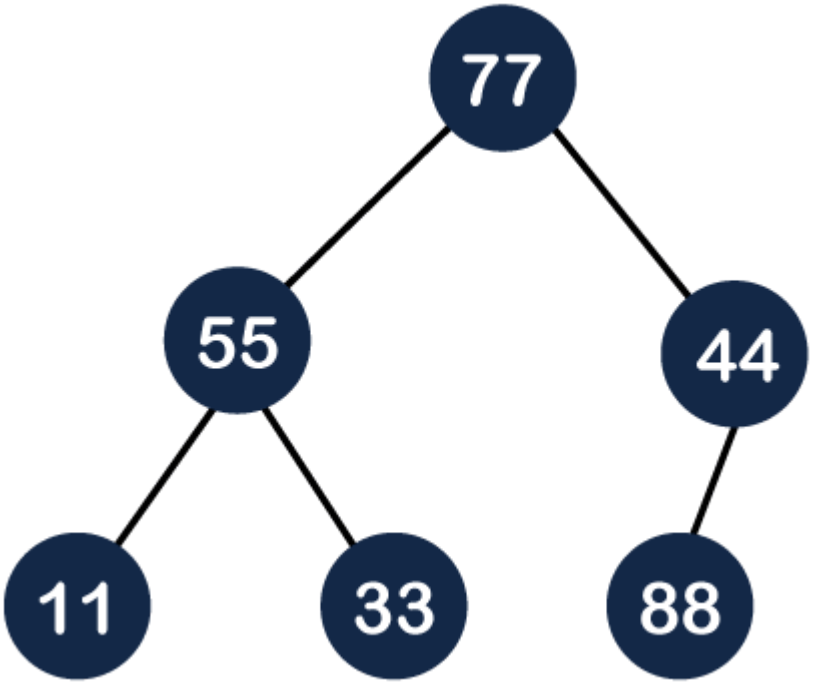
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:

Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

Let's understand the deletion through an example.

Step 1: In the above tree, the first 30 node is deleted from the tree and it is replaced with the 15 element as shown below:

Now we will heapify the tree. We will check whether the 15 is greater than either of its child or not. 15 is less than 20 so we will swap these two values as shown below:

Again, we will compare 15 with its child. Since 15 is greater than 10 so no swapping will occur.

Algorithm to heapify the tree

```
1. MaxHeapify(A, n, i)
2. {
3. int largest =i;
4. int l= 2i;
5. int r= 2i+1;
6. while(l<=n && A[l]>A[largest])
7. {
8. largest=l;
9. }
10. while(r<=n && A[r]>A[largest])
11.
12. {
13. largest=r;
14. }
15. if(largest!=i)
16. {
```

```
17. swap(A[largest], A[i]);
18. heapify(A, n, largest);
19. }}
```

The breadth-first search or BFS algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It begins at the root of the tree or graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.

What Is Graph Traversal Algorithms in Data Structure?

Graph traversal is a search technique to find a vertex in a graph. In the search process, graph traversal is also used to determine the order in which it visits vertices. Without producing loops, a graph traversal finds the edges to be employed in the search process. That is, utilizing graph traversal, you can visit all the graph's vertices without going through a looping path.

There are two methods for traversing graphs, which are as follows:

- Breadth-First Search or BFS Algorithm
- Depth- First Search or DFS Algorithm

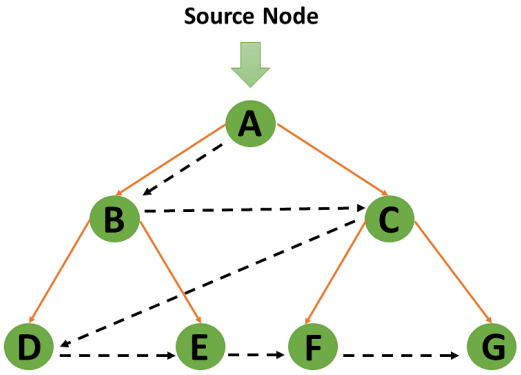
What Is the Breadth-First Search Algorithm?

Breadth-First Search Algorithm or BFS is the most widely utilized method.

BFS is a graph traversal approach in which you start at a source node and layer by layer through the graph, analyzing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the next-level neighbor nodes.

According to the BFS, you must traverse the graph in a breadthwise direction:

- To begin, move horizontally and visit all the current layer's nodes.
- Continue to the next layer.



Breadth-First Search uses a queue data structure to store the node and mark it as "visited" until it marks all the neighboring vertices directly related to it. The queue operates on the First In First Out (FIFO) principle, so the node's neighbors will be viewed in the order in which it inserts them in the node, starting with the node that was inserted first.

How Does the BFS Algorithm Work?

Breadth-First Search uses a queue data structure technique to store the vertices. And the queue follows the First In First Out (FIFO) principle, which means that the neighbors of the node will be displayed, beginning with the node that was put first.

The transverse of the BFS algorithm is approaching the nodes in two ways.

- Visited node
- Not visited node

How Does the Algorithm Operate?

- Start with the source node.
- Add that node at the front of the queue to the visited list.
- Make a list of the nodes as visited that are close to that vertex.
- And dequeue the nodes once they are visited.
- Repeat the actions until the queue is empty.

Why Do You Need Breadth-First Search Algorithm?

There are several reasons why you should use the BFS Algorithm to traverse graph data structure. The following are some of the essential features that make the BFS algorithm necessary:

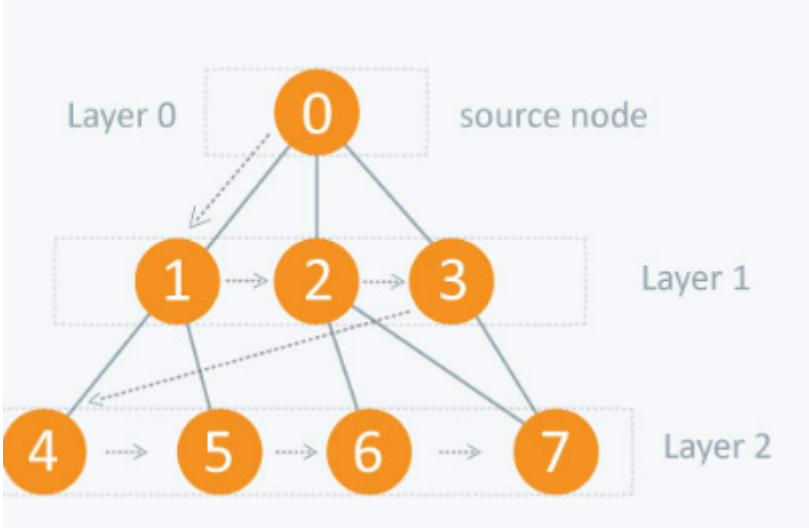
- The BFS algorithm has a simple and reliable architecture.
- The BFS algorithm helps evaluate nodes in a graph and determines the shortest path to traverse nodes.
- The BFS algorithm can traverse a graph in the fewest number of iterations possible.
- The iterations in the BFS algorithm are smooth, and there is no way for this method to get stuck in an infinite loop.
- In comparison to other algorithms, the BFS algorithm's result has a high level of accuracy.

Rules to Remember in the BFS Algorithm

- You can take any node as your source node or root node.
- You should explore all the nodes.
- And don't forget to explore on repeated nodes.
- You must transverse the graph in a breadthwise direction, not depthwise.

The Architecture of the BFS Algorithm

We understand the architecture of BFS and how the algorithm visits nodes in the tree diagram, which is added below.



The above tree diagram contains three layers, which are numbered from 0 to 2.

- We are allowed to use any node as our source node as per the law. However, in this case, we can use 0 as our source node.
- Then we explore breadthwise and find the nodes which are adjacently connected to our source node.
- Then we must come down to layer two and find the relative nodes that are adjacent to the layer 1 nodes.
- If you select the alternative source node, this order will change. And that's how the straightforward BFS algorithm operates.

Pseudocode Of Breadth-First Search Algorithm

The breadth-first search algorithm's pseudocode is:

```
Bredth_First_Serach( G, A ) // G ie the graph and A is the source node

    Let q be the queue

    q.enqueue( A ) // Inserting source node A to the queue
```

```
        Mark A node as visited.

        While ( q is not empty )

        B = q.dequeue( ) // Removing that vertex from the queue, which will be visited by its neighbour

Processing all the neighbors of B

For all neighbors of C of B

            If C is not visited, q. enqueue( C ) //Stores C in q to visit its neighbour

Mark C a visited
```

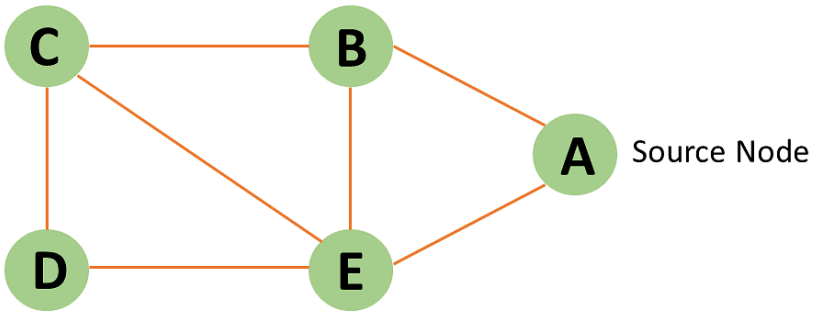
For a better understanding, you will look at an example of a breadth-first search algorithm later in this tutorial.

Example of Breadth-First Search Algorithm

In a tree-like structure, graph traversal requires the algorithm to visit, check, and update every single un-visited node. The sequence in which graph traversals visit the nodes on the graph categorizes them.

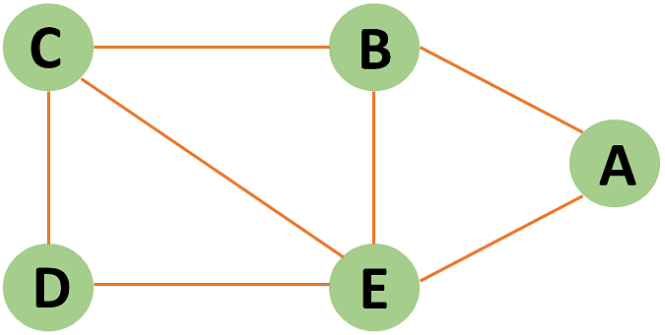
The BFS algorithm starts at the first starting node in a graph and travels it entirely. After traversing the first node successfully, it visits and marks the next non-traversed vertex in the graph.

Step 1: In the graph, every vertex or node is known. First, initialize a queue.



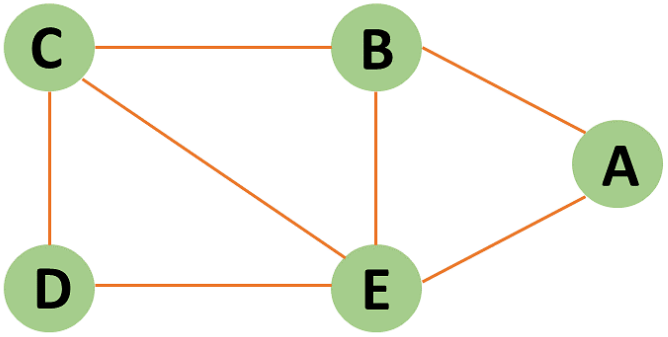
Queue

Step 2: In the graph, start from source node A and mark it as visited.

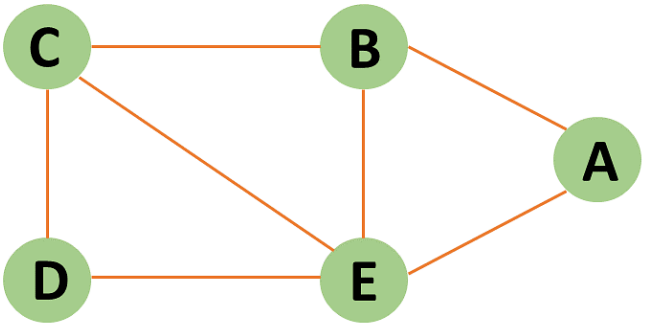


Queue A

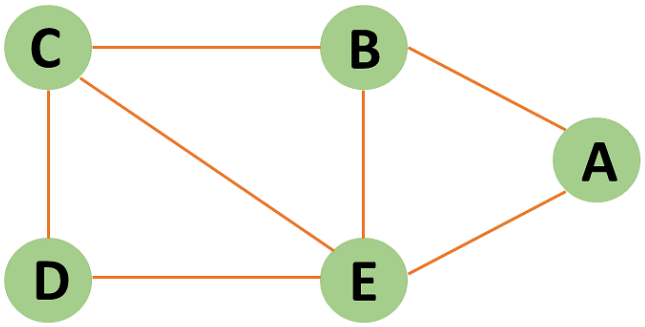
Step 3: Then you can observe B and E, which are unvisited nearby nodes from A. You have two nodes in this example, but here choose B, mark it as visited, and enqueue it alphabetically.



Step 4: Node E is the next unvisited neighboring node from A. You enqueue it after marking it as visited.



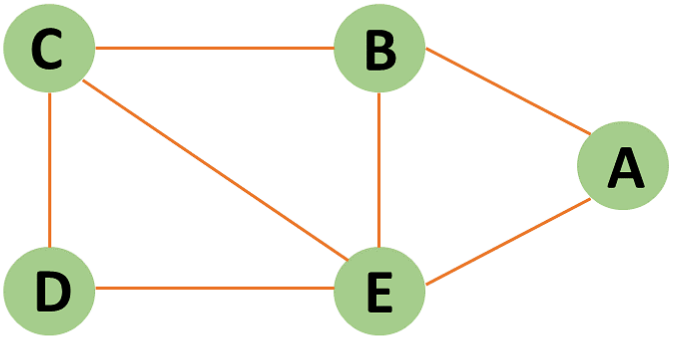
Step 5: A now has no unvisited nodes in its immediate vicinity. As a result, you dequeue and locate A.



Queue

E	B				

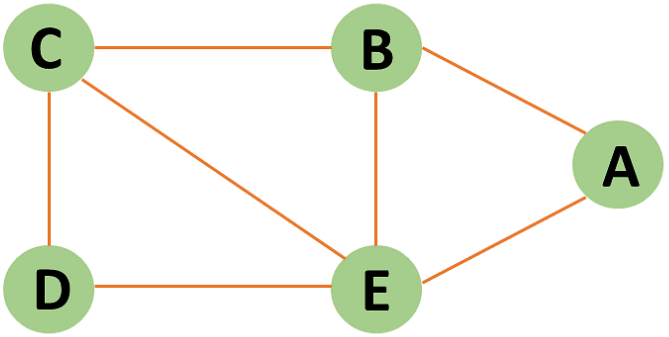
Step 6: Node C is an unvisited neighboring node from B. You enqueue it after marking it as visited.



Queue

E	B	C			

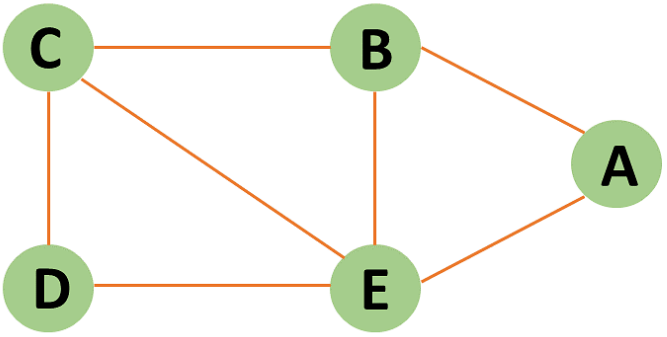
Step 7: Node D is an unvisited neighboring node from C. You enqueue it after marking it as visited.



Queue

E	B	C	D		

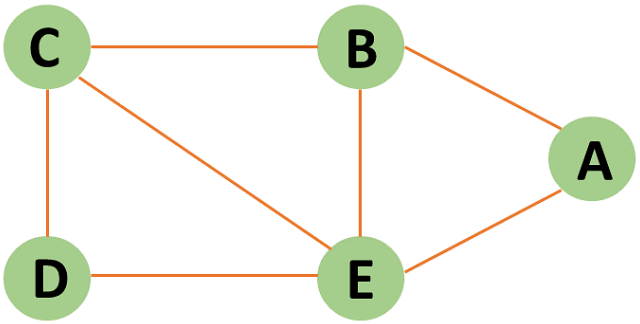
Step 8: If all of D's adjacent nodes have already been visited, remove D from the queue.



Queue

E	B	C			

Step 9: Similarly, all nodes near E, B, and C nodes have already been visited; therefore, you must remove them from the queue.



Queue

Step 10: Because the queue is now empty, the bfs traversal has ended.

Breadth-First Search Algorithm Code Implementation:

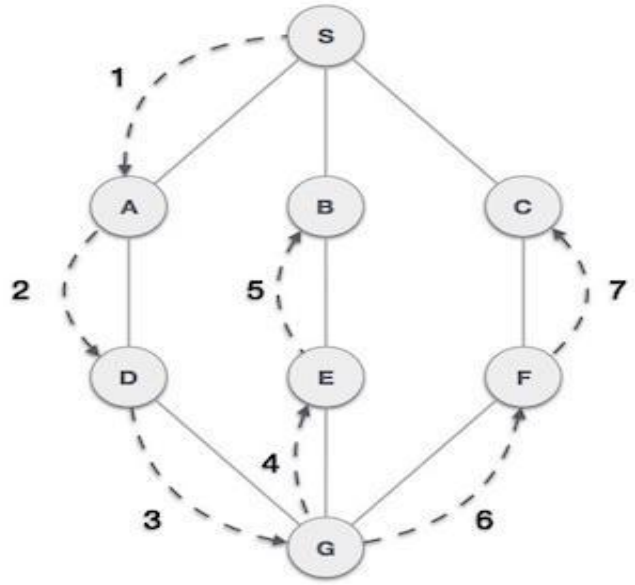
```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int twodimarray[10][10],queue[10],visited[10],n,i,j,front=0,rear=-1;

void breadthfirstsearch(int vertex) // breadth first search function

{

for (i=1;i<=n;i++)

if(twodimarray[vertex][i] && !visited[i])

queue[++rear]=i;

if(front<=rear)
```

```
{
visited[queue[front]]=1;
breadthfirstsearch(queue[front++]);
}
}
int main() {
int x;
printf("\n Enter the number of vertices:");
scanf("%d",&n);
for (i=1;i<=n;i++) {
queue[i]=0;
visited[i]=0;
}
printf("\n Enter graph value in form of matrix:\n");
for (i=1;i<=n;i++)
for (j=1;j<=n;j++)
scanf("%d",&twodimarray[i][j]);
printf("\n Enter the source node:");
scanf("%d",&x);
breadthfirstsearch(x);
```

```
printf("\n The nodes which are reachable are:\n");

for (i=1;i<=n;i++)

if(visited[i])

printf("%d\t",i);

    else

printf("\n Breadth first search is not possible");

getch();

}
```

Depth First Search (DFS) Algorithm

Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Refer https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

For implementation of dfs

Connected and Biconnected Components

Refer PPT

Quick sort:

Quick sort

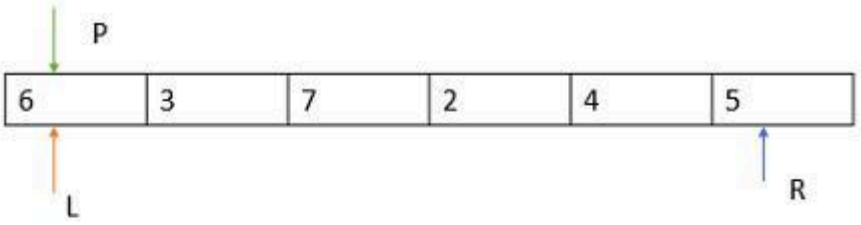
It is a **divide and conquer algorithm.**

- Step 1 − Pick an element from an array, call it as pivot element.
- Step 2 − Divide an unsorted array element into two arrays.
- Step 3 − If the value less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

Consider an example given below, wherein

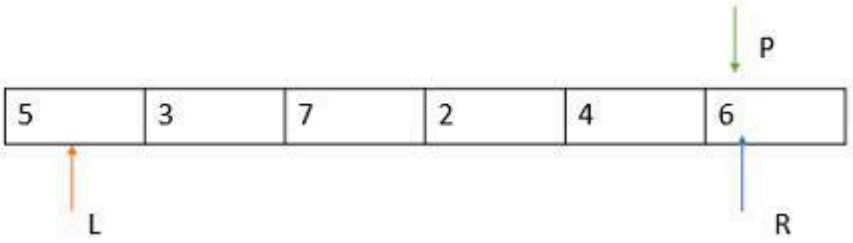
- P is the pivot element.
- L is the left pointer.
- R is the right pointer.

The elements are 6, 3, 7, 2, 4, 5.



Case 1: P=6 {Right side P is greater and Left side of is Leese}

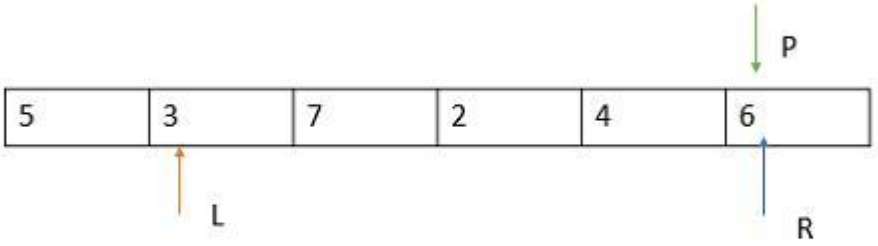
Is P<R => 6<5 {wrong}
So, swap P with R



Case 2: P=6 , L=5 {Right side P is greater and Left side of is Lesser}

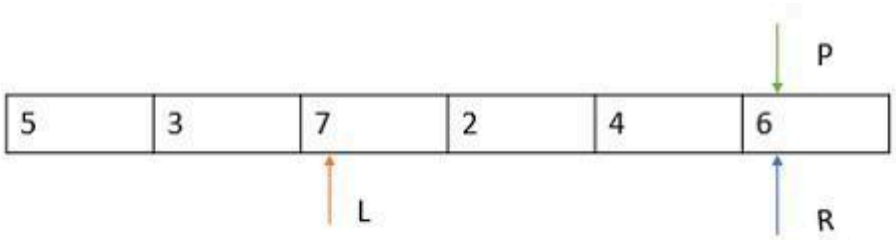
Is P>L => 6> 5 {right}
Move L towards right

Case 3:



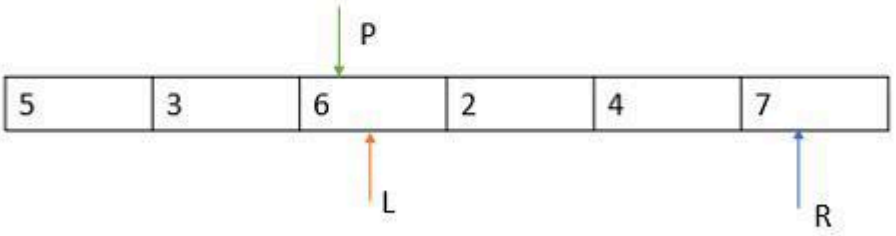
Is P>L, 6>3, Yes
So, move L towards right

Case 4:



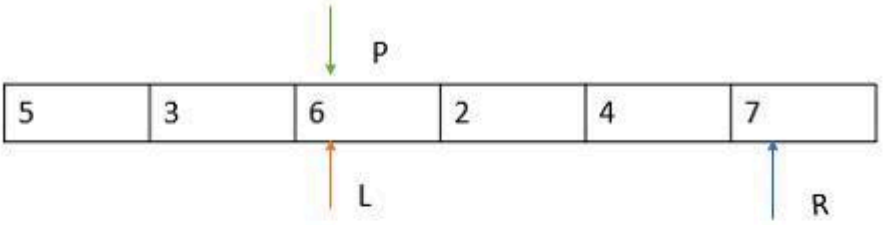
Is P>L => 6> 7 {wrong}
then swap P and L

Case5 :



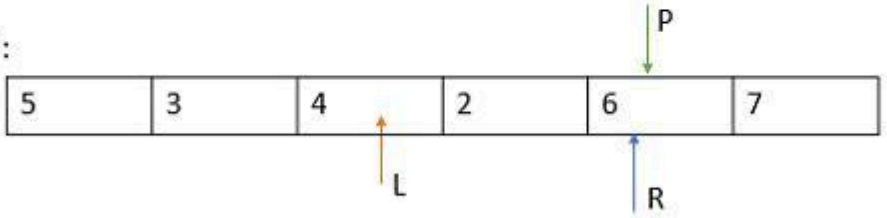
Is P<R => 6<7, => Yes
Decrement R

Case 6:



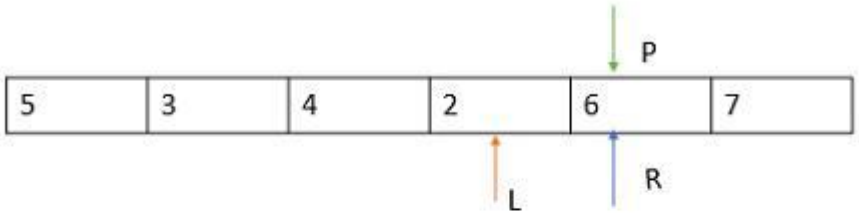
Is P<R => 6<4 {wrong}
then swap

Case 7 :



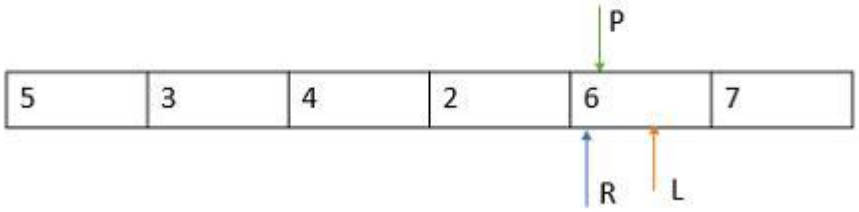
Is P>L => 6>4, => Yes
Move L to right

Case 8:



Is P>L => 6>2 {right}
move L to right

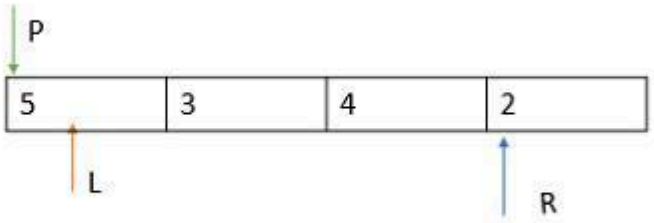
Case 9 :



Now,

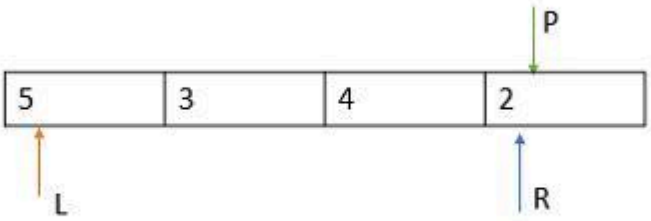
- The pivot is in fixed position.
- All the left elements are less.
- The right elements are greater than pivot.
- Now, divide the array into 2 sub arrays left part and right part.
- Take left partition apply quick sort.

Case 1:

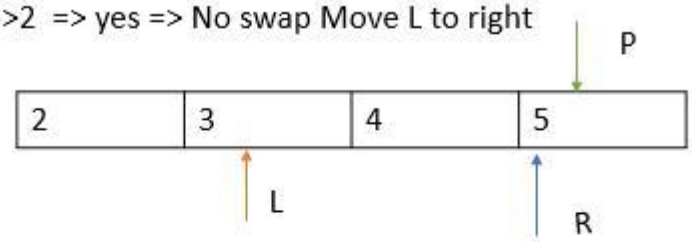


Is P<R => 5<2 {wrong} so, swap

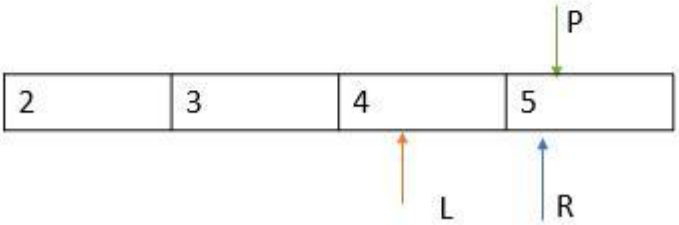
Case 2 :



Case 3: is P>L => 5>2 => yes => No swap Move L to right



Case 4: Is P> L => 5>3 => Yes => No swap => Move L to right



Now,

- The pivot is in fixed position.
- All the left elements are less and sorted
- The right elements are greater and are in sorted order.
- The final sorted list is combining two sub arrays is 2, 3, 4, 5, 6, 7

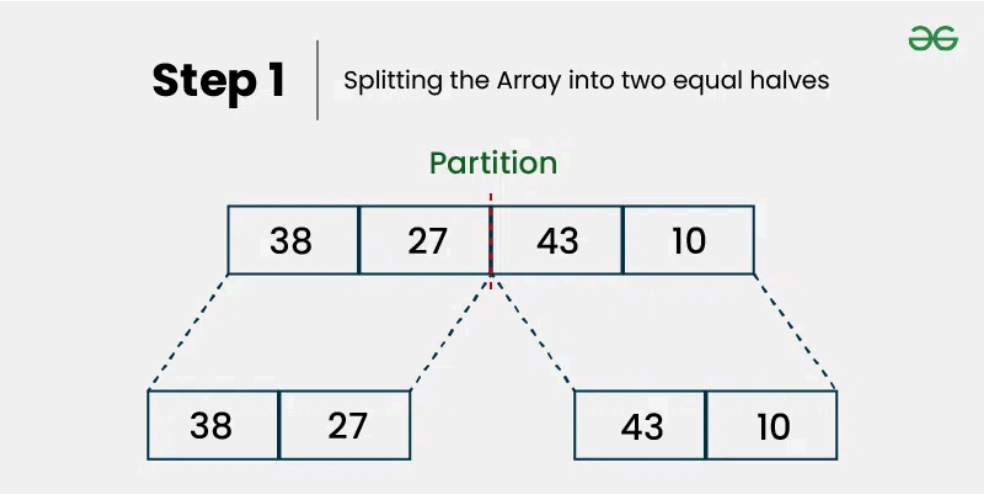
Merge Sort

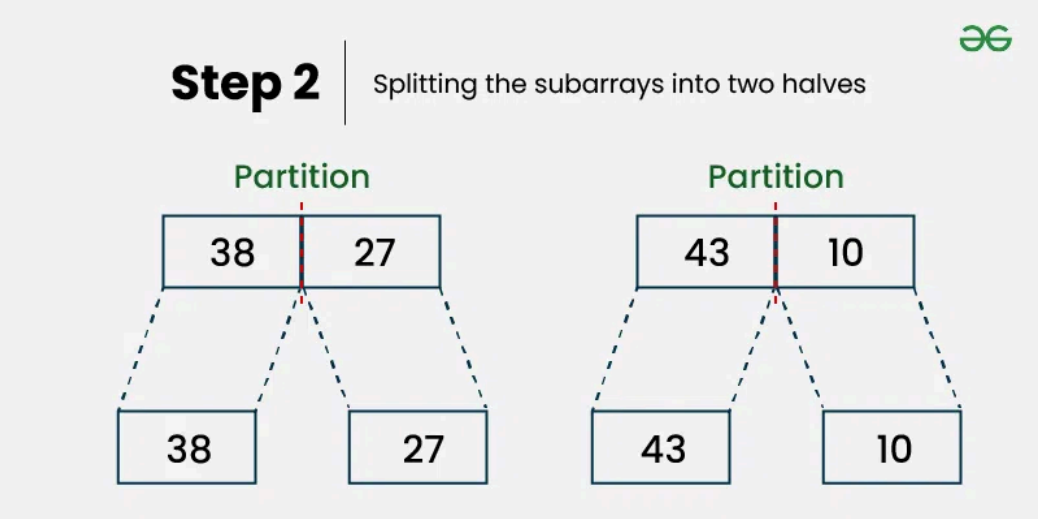
Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

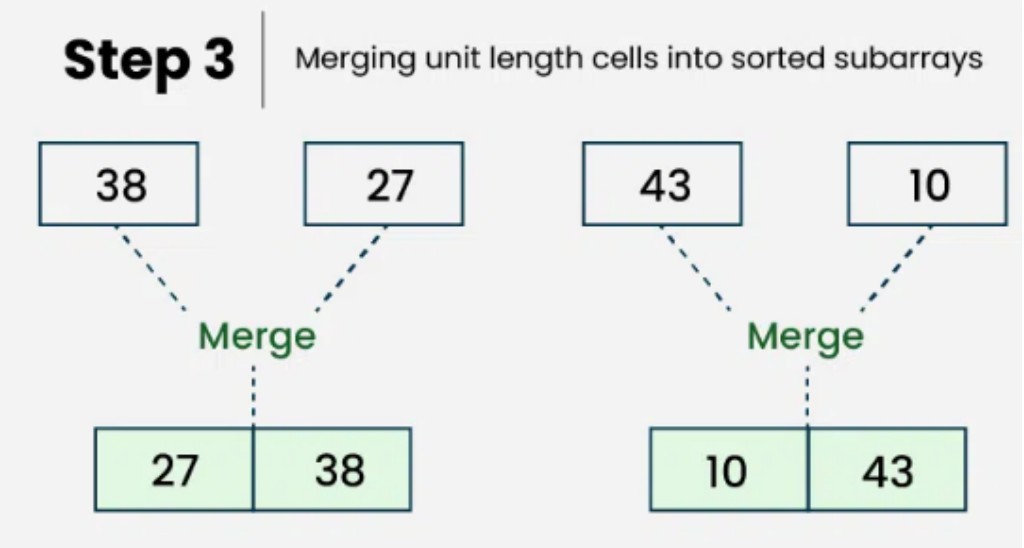
step-by-step explanation of how merge sort works:

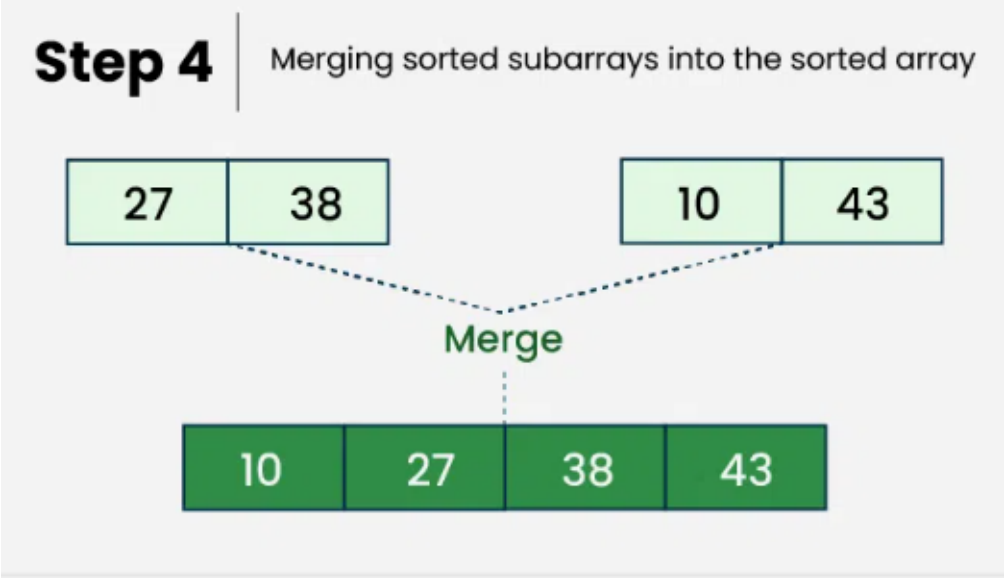
1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:
Let's sort the array or list **[38, 27, 43, 10]** using Merge Sort


Step 1
Splitting the Array into two equal halves
Partition
38
27
43
10
38
27
43
10


Step 2
Splitting the subarrays into two halves
Partition
Partition
38
27
43
10
38
27
43
10


Step 3
Merging unit length cells into sorted subarrays
38
27
43
10
Merge
Merge
27
38
10
43



Recurrence Relation of Merge Sort:

The recurrence relation of merge sort is:

$T(n)=\{\Theta(1)\text{if } n=1 2T(n2)+\Theta(n)\text{if } n>1\ T(n)=\{\Theta(1)2T(2n)+\Theta(n)\text{if } n=1\text{if } n>1$

- T(n) Represents the total time time taken by the algorithm to sort an array of size n.
- 2T(n/2) represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2).
- O(n) represents the time taken to merge the two sorted halves

Convex Hull

Refer video link :
https://www.youtube.com/watch?v=5D9F1HA6-f4 -----quick hull problem
https://www.youtube.com/watch?v=QYrpHE8iDGg ----------graham scan algorithm

Refer PPT as well