Advanced Data Structure

Syllabus

Unit -I: Linked List (10 Periods)

Drawbacks of Arrays, Introduction to Linked lists. Types of Linked Lists, Representation of Linked List in Memory, Operations on Singly Linked Lists (Traversing, Insertion, Deletion and modification), Doubly Linked List, Representation of Doubly Linked List in Memory, Operations on doubly Linked Lists (Traversing, Insertion, Deletion and modification).

Unit -II: Trees (10 Periods)

Introduction and key terminology, Binary Trees Binary Tree Creation and Traversal Using Arrays, Binary Tree Creation and Traversal Using Pointers, Expression Trees, traversing binary tree recursively and non-recursively (pre-order, in order, post order traversal). Application of trees (binary search tree).

Unit -III: Graphs(10 Periods)

Introduction and key terminology, graph representation in memory (static and dynamic), traversing a graph (breath first search, depth first search), spanning tree. Kruskal's Algorithm, Prim's Algorithm

Unit -IV: Advance Trees (10 Periods)

Heaps, Min/ Max Heap, Binomial Heap, Fibonacci Heap, Heap Sort, B Tree, B+ Tree.

Unit I

➢Introduction to Data Structure

- ✓ A data structure is a special way of organizing the data elements into a particular form. The arrangement of data in a particular order is very important to access the particular data element in less time easily without putting much effort.
- ✓ For example, in our daily life, when we used to put our clothes in a particular drawer properly, especially in a sequence so that whenever we want to wear a particular dress, we may not require to suffer in finding it out, and save our time from wasting.
- ✓ Similarly in this way, the computer system organizes the data in a particular specific manner, so that to access any particular data element, or to delete it or any other operation we require to perform on it, can be done easily without making many efforts and also it will be done in less time.
- ✓ We can even further do the arrangements of the data elements entered into the data structure like sorting the elements in ascending or descending order.

➢ Types of Data Structure

The Data structures are categorized on two bases:

1. Primitive data structures
2. Non-Primitive data structure

Let us look at what it is and how it derives non-linear data structures.

1. Primitive data structures

The primitive data structures are nothing but the predefined data structures, which are already defined; we do not require giving a particular definition. To derive the non-primitive data structures, we will use these primitive data structures to easily collect a large amount of data. These data structures involve int, float, char, etc. The ' int 'is a data type used to set the type of our data as an integer type; similarly, in this way, we have a float for assigning the type float to our data used to store the decimal values and so on.

2. Non-Primitive data structures

The non-primitive data structures are nothing but the defined data structures used to create particular data structures by using the primitive data structures. It is mainly used to store the collection of elements; it may be of the same data types and may differ depending on the program's need. In non - primitive data structures, we have a concept of Abstract data type. It is a derived data type that the user derives, and the user defines that data type. We must create an abstract data type for using it in many places. There are various non-primitive data structures like an array, linked list, queue, stack, etc.

Types of Non-primitive data structures

Linear Data structure:

- The linear data structure is nothing but arranging the data elements linearly one after the other. Here, we cannot arrange the data elements randomly as in the hierarchical order.

- This linear data structure will follow the sequential order of inserting the various data elements. Similarly, in this way, we perform the deletion operation onto the elements. Linear data structures are easy to implement because computer memory is arranged linearly. Its examples are **array, stack, queue, linked list, etc**.

Let us discuss some of its types:

Array:

- An array is a collection of homogeneous data elements consisting of mainly the same data types.
- An array consists of similar types of data elements present on the contiguous memory locations. Here the word contiguous means consecutive address locations. Suppose our particular array is starting from the address location 1000. Depending on its type, the next element is present at the consecutive memory location with the data type's difference in size.

Stack:

- Stack is also one of the important linear data structures based on the LIFO (Last In First Out) principle. Many computer applications and the various strategies used in the operating system and other places are based on the principle of LIFO itself. In this principle, the data element entered last must be popped out first from it, and the element pushed into the stack at the very first time is popped out last. In this approach, we will push the data elements into the stack until it reaches their end limit; after that, we will pop out the corresponding values.

Queue:

- A queue is one of the important linear data structures extensively used in various **computer applications**, and also it is based on the **FIFO** (First In First Out) principle. It follows a particular order of data execution for which operations are performed. In this data structure, data enters from one end, i.e., **the REAR** end, and the next data enters the queue after the previous one. Deletion operation is performed from another end, i.e**., FRONT**

Linked list:

- The linked list is another major data structure used in various programs, and even many non-linear data structures are implemented using this linked list data structure. As the name suggests, it consists of a link of the node connected by holding the address of the next node. It comes in the portion of the linear data structure because it forms the link-like structure the one data node is connected sequentially with the other node by carrying the address of that node.
- This data is not arranged in a sequential contiguous location as observed in the array. The homogeneous data elements are placed at the contiguous memory location to retrieve data elements is simpler.

Non-linear data structure

- A non-linear data structure is another important type in which data elements are not arranged sequentially; mainly, data elements are arranged in random order without forming a linear structure.
- Data elements are present at the multilevel, for example, tree.
- In trees, the data elements are arranged in the hierarchical form, whereas in graphs, the data elements are arranged in random order, using the edges and vertex.
- Multiple runs are required to traverse through all the elements completely. Traversing in a single run is impossible to traverse the whole data structure.
- Each element can have multiple paths to reach another element.
- The data structure where data items are not organized sequentially is called a non-linear data structure. In other words, data elements of the non-linear data structure could be connected to more than one element to reflect a special relationship among them.

Let us discuss some of its types:

Trees and **Graphs** are the types of non-linear data structures.

Tree:

- The tree is a non-linear data structure that is comprised of various nodes. The nodes in the tree data structure are arranged in hierarchical order.

- It consists of a root node corresponding to its various child nodes, present at the next level. The tree grows on a level basis, and root nodes have limited child nodes depending on the order of the tree.
- For example, in the binary tree, the order of the root node is 2, which means it can have at most 2 children per node, not more than it.
- The non-linear data structure cannot be implemented directly, and it is implemented using the linear data structure like an array and linked list.
- The tree itself is a very broad data structure and is divided into various categories like **Binary tree, Binary search tree, AVL trees, Heap, max Heap, min-heap**, etc.
- All the types of trees mentioned above differ based on their properties.

Graph

- A graph is a non-linear data structure with a finite number of vertices and edges, and these edges are used to connect the vertices.
- The graph itself is categorized based on some properties; if we talk about a complete graph, it consists of the vertex set, and each vertex is connected to the other vertexes having an edge between them.
- The vertices store the data elements, while the edges represent the relationship between the vertices.
- A graph is very important in various fields; the network system is represented using the graph theory and its principles in computer networks.

- Even in Maps, we consider every location a vertex, and the path derived between two locations is considered edges.
- The graph representation's main motive is to find the minimum distance between two vertexes via a minimum edge weight.

Properties of Non-linear data structures

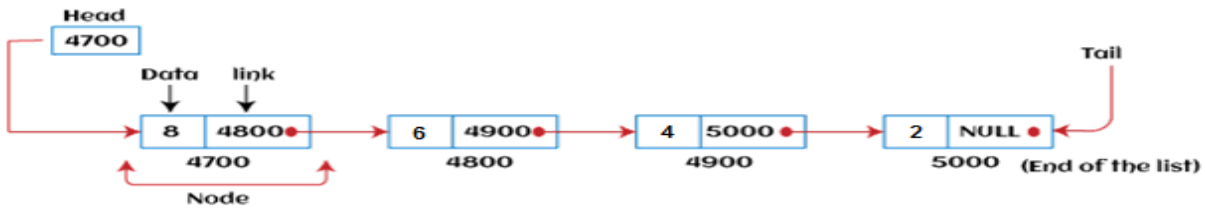
- It is used to store the data elements combined whenever they are not present in the contiguous memory locations.
- It is an efficient way of organizing and properly holding the data.
- It reduces the wastage of memory space by providing sufficient memory to every data element.
- Unlike in an array, we have to define the size of the array, and subsequent memory space is allocated to that array; if we don't want to store the elements till the range of the array, then the remaining memory gets wasted.
- So to overcome this factor, we will use the non-linear data structure and have multiple options to traverse from one node to another.
- Data is stored randomly in memory.
- It is comparatively difficult to implement.
- Multiple levels are involved.
- Memory utilization is effective.

Linked list

- ✓ Linked list is a linear data structure that includes a series of connected nodes.
- ✓ Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Till now, we have been using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages that must be known to decide the data structure that will be used throughout the program.

Now, the question arises why we should use linked list over array?

Why use linked list over array?

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

How to declare a linked list?

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type **structure.**

The declaration of linked list is given as follows -

```
struct node
{
int data;
struct node *next;
```

}

In the above declaration, we have defined a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of next node.

Advantages of Linked list

The advantages of using the Linked list are given as follows -

- **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
- **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation -** We can implement both stacks and queues using linked list.

Disadvantages of Linked list

The limitations of using the Linked list are given as follows -

- **Memory usage -** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked list

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.

- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list

The basic operations that are supported by a list are mentioned as follows -

- **Insertion -** This operation is performed to add an element into the list.
- **Deletion -** It is performed to delete an operation from the list.
- **Display -** It is performed to display the elements of the list.
- **Search -** It is performed to search an element from the list using the given key.

Types of Linked List

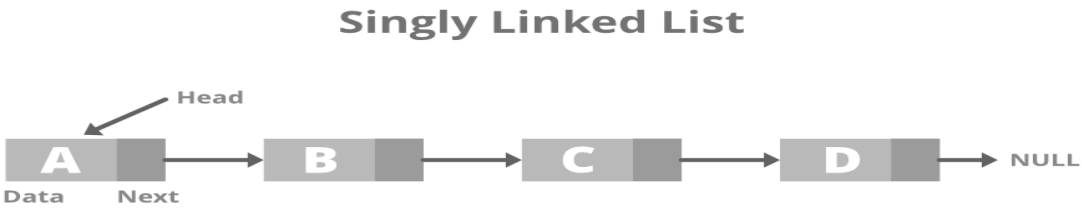
Before knowing about the types of a linked list, we should know what is ***linked list***. So, to know about the linked list, click on the link given below:

Types of Linked list

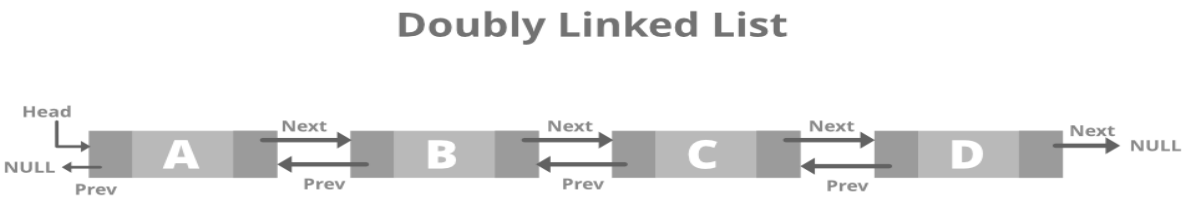
The following are the types of linked list:

- **Singly Linked list**
- **Doubly Linked list**

- **Singly Linked List:** It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows traversal of data only in one way. Below is the image for the same:\



- **Doubly Linked List:** A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in sequence, Therefore, it contains three parts are data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:



A **singly linked list** is the most simple type of **linked list**, with each node containing some data as well as a pointer to the next node. That is a singly linked list allows traversal of data only in one way. There are several linked list operations that allow us to perform different tasks.

The basic linked list operations are:

- **Traversal** – Access the nodes of the list.
- **Insertion** – Adds a new node to an existing linked list.
- **Deletion** – Removes a node from an existing linked list.
- **Search** – Finds a particular element in the linked list.

✓ Traverse a Linked List

Accessing the nodes of a linked list in order to process it is called **traversing** a linked list.

Normally we use the traverse operation to display the contents or to search for an element in the linked list. The algorithm for traversing a linked list is given below.

Algorithm: Traverse

```
Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL

Step 3: Apply process to PTR -> DATA

Step 4: SET PTR = PTR->NEXT

[END OF LOOP]

Step 5: EXIT
```

- We first initialize PTR with the address of HEAD. Now the PTR points to the first node of the linked list.
- A while loop is executed, and the operation is continued until PTR reaches the last node (PTR = NULL).
- Apply the process(display) to the current node.
- Move to the next node by making the value of PTR to the address of next node.

✓ **Inserting Elements to a Linked List**

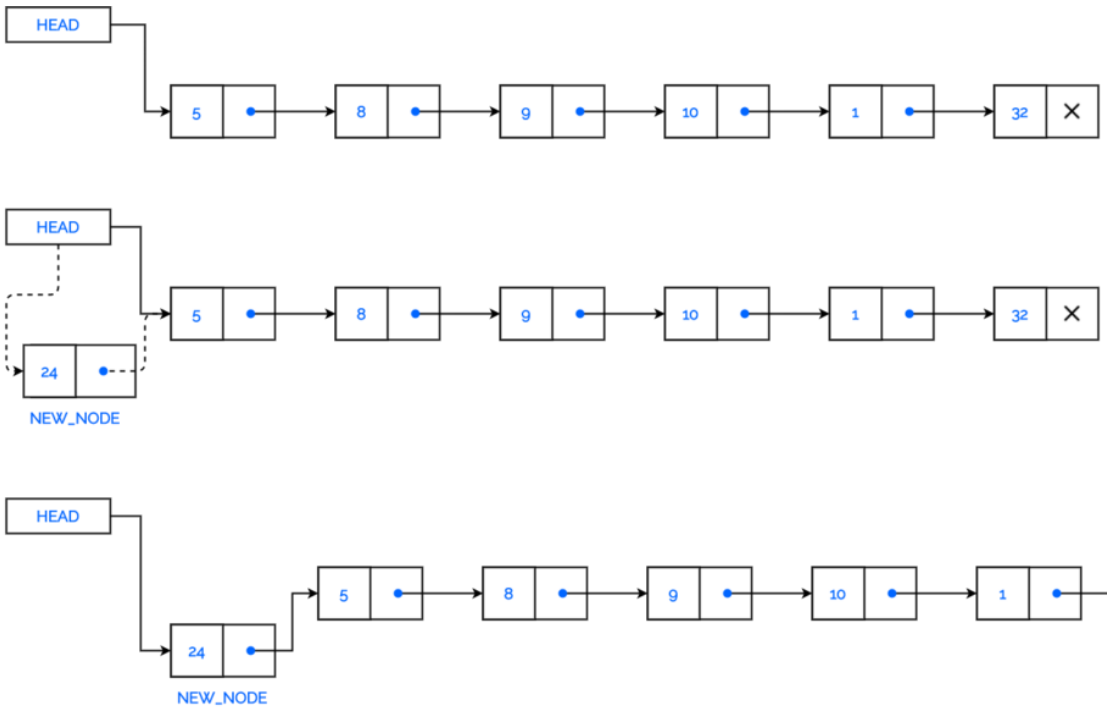
We will see how a new node can be added to an existing linked list in the following cases.

1. The new node is inserted at the beginning.
2. The new node is inserted at the end.
3. The new node is inserted after a given node.

Insert a Node at the beginning of a Linked list

Consider the linked list shown in the figure. Suppose we want to create a new node with data 24

and add it as the first node of the list. The linked list will be modified as follows.



- Allocate memory for new node and initialize its DATA part to 24.
- Add the new node as the first node of the list by pointing the NEXT part of the new node to HEAD.
- Make HEAD to point to the first node of the list.

Algorithm: InsertAtBeginning

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 7

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = HEAD

Step 6: SET HEAD = NEW_NODE

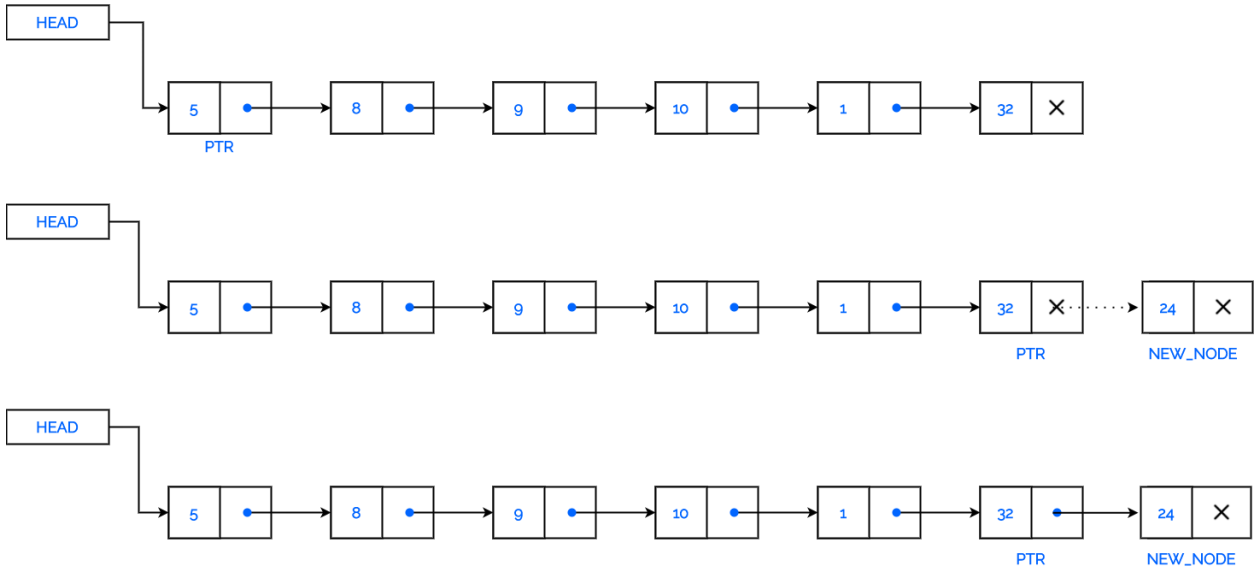
Step 7: EXIT

Note that the first step of the algorithm checks if there is enough memory available to create a new node. The second, and third steps allocate memory for the new node.

Insert a Node at the end of a Linked list

Take a look at the linked list in the figure. Suppose we want to add a new node with data 24 as

the last node of the list. Then the linked list will be modified as follows.



- Allocate memory for new node and initialize its DATA part to 24.
- Traverse to last node.
- Point the NEXT part of the last node to the newly created node.
- Make the value of next part of last node to NULL.

Algorithm: InsertAtEnd

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 10

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

```
Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET PTR = HEAD

Step 7: Repeat Step 8 while PTR -> NEXT != NULL

Step 8: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 9: SET PTR -> NEXT = NEW_NODE

Step 10: EXIT
```

- ✓ **Deleting Elements from a Linked List**

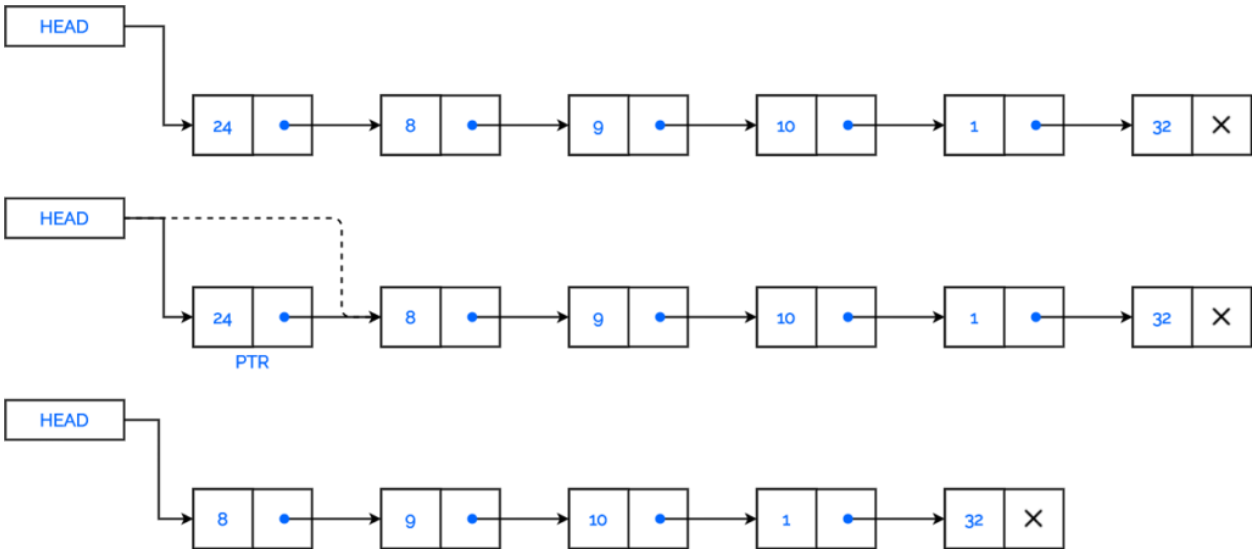
Let's discuss how a node can be deleted from a linked listed in the following cases.

1. The first node is deleted.
2. The last node is deleted.
3. The node after a given node is deleted.

Delete a Node from the beginning of a Linked list

Suppose we want to delete a node from the beginning of the linked list. The list has to be

modified as follows:



- Check if the linked list is empty or not. Exit if the list is empty.
- Make HEAD points to the second node.
- Free the first node from memory.

Algorithm: DeleteFromBeginning

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 5

[END OF IF]

Step 2: SET PTR = HEAD

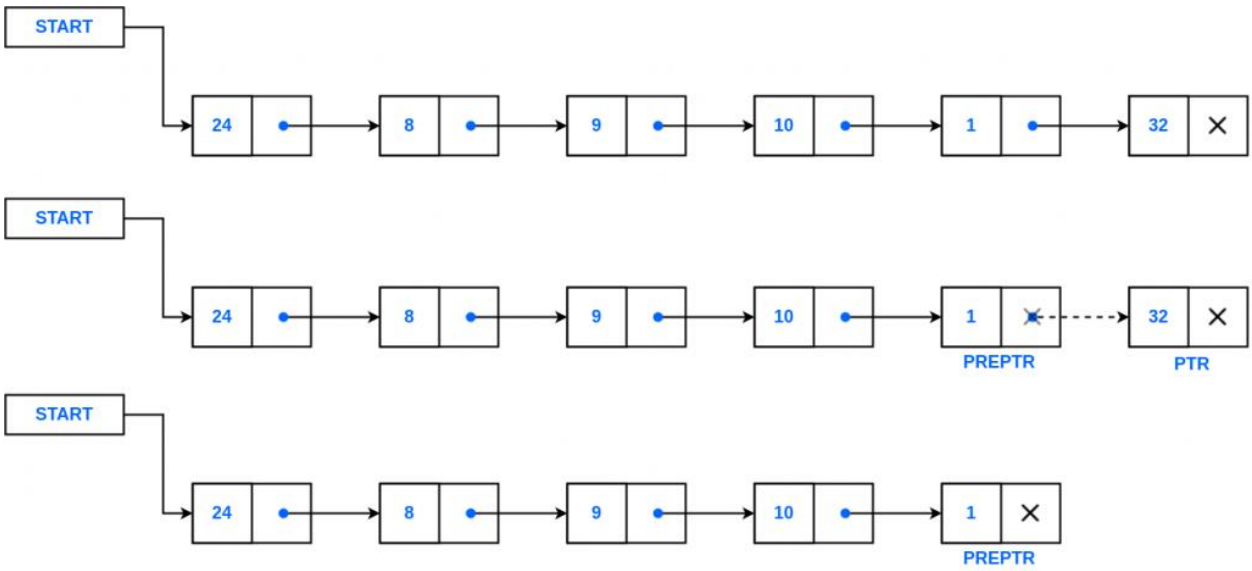
Step 3: SET HEAD = HEAD -> NEXT

Step 4: FREE PTR

Step 5: EXIT

Delete last Node from a Linked list

Suppose we want to delete the last node from the linked list. The linked list has to be modified as follows:



- Traverse to the end of the list.
- Change value of next pointer of second last node to NULL.
- Free last node from memory.
    - Step 1: **IF** HEAD = **NULL**
    - Write UNDERFLOW
    - Go to Step 8
    - [**END OF IF**]
    - Step 2: SET PTR = HEAD
    - Step 3: Repeat Steps 4 and 5 **while** PTR -> NEXT != **NULL**
    - Step 4: SET PREPTR = PTR
    - Step 5: SET PTR = PTR -> NEXT
    - [**END OF** LOOP]
    - Step 6: SET PREPTR -> NEXT = **NULL**
    - Step 7: FREE PTR
    - Step 8: EXIT

✓ **Search**

Finding an element is similar to a traversal operation. Instead of displaying data, we have to check whether the data matches with the **item** to find.

- Initialize PTR with the address of HEAD. Now the PTR points to the first node of the linked list.
- A while loop is executed which will compare data of every node with item.
- If item has been found then control goes to last step.

Algorithm: Search

Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL

Step 3: If ITEM = PTR -> DATA

SET POS = PTR

Go To Step 5

ELSE

SET PTR = PTR -> NEXT

[END OF IF]

[END OF LOOP]

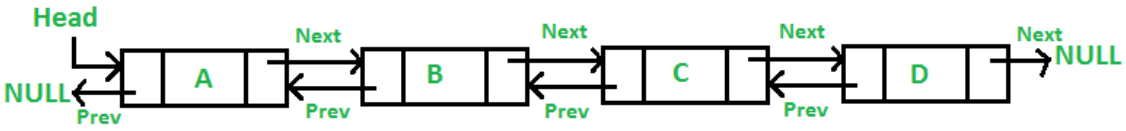
Step 4: SET POS = NULL

Step 5: EXIT

Operations of Doubly Linked List

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.

A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in a singly linked list.

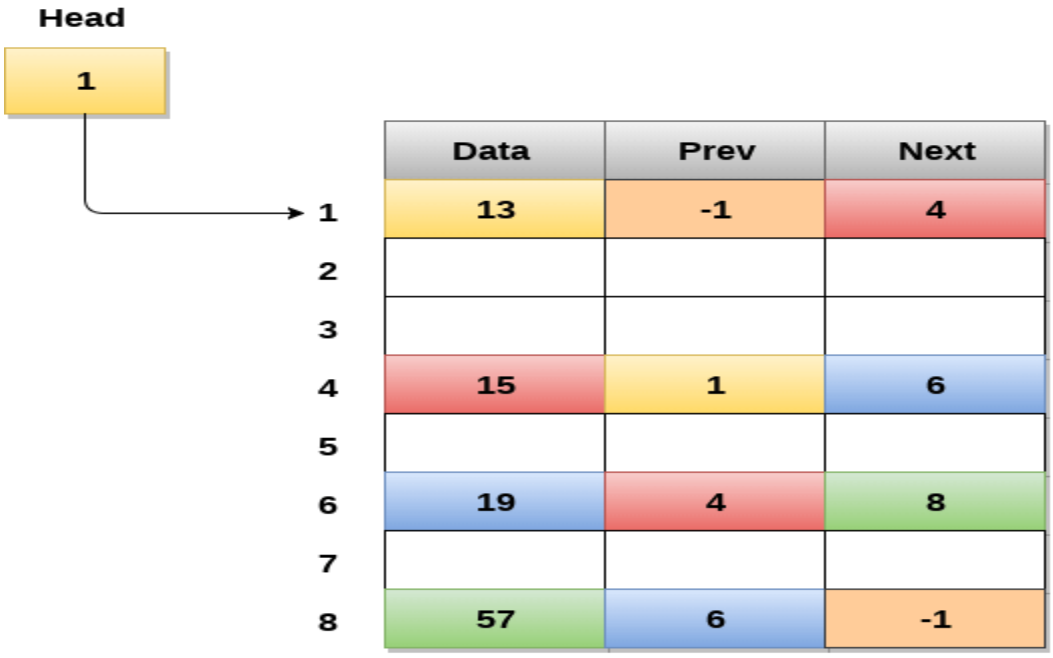


Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

1. **Add a node at the front of DLL:** The new node is always added before the head of the given Linked List. And the newly added node becomes the new head of DLL & maintaining a global variable for counting the total number of nodes at that time.

2. Traversal of a Doubly linked list
3. **Insertion of a node:** This can be done in three ways:
   - **At the beginning:** The new created node is insert in before the head node and head points to the new node.
   - **At the end:** The new created node is insert at the end of the list and tail points to the new node.
   - **At a given position:** Traverse the given DLL to that position(**let the node be X**) then do the following:
     1. Change the next pointer of new Node to the next pointer of Node X.
     2. Change the prev pointer of next Node of Node X to the new Node.
     3. Change the next pointer of node X to new Node.
     4. Change the prev pointer of new Node to the Node X.
4. **Deletion of a node:** This can be done in three ways:
   - **At the beginning:** Move head to the next node to delete the node at the beginning and make previous pointer of current head to NULL .
   - **At the last:** Move tail to the previous node to delete the node at the end and make next pointer of tail node to NULL.
   - **At a given position:** Let the prev node of Node at position pos be Node X and next node be Node Y, then do the following:
     1. Change the next pointer of Node X to Node Y.
     2. Change the previous pointer of Node Y to Node X.

UNIT III- Graph Data Structure

Introduction

- Graphs in data structures are non-linear data structures made up of a finite number of nodes or vertices and the edges that connect them.
- Graphs in data structures are used to address real-world problems in which it represents the problem area as a network like telephone networks, circuit networks, and social networks.
- For example, it can represent a single user as nodes or vertices in a telephone network, while the link between them via telephone represents edges.

❖ What Are Graphs in Data Structure?

- A graph is a non-linear kind of data structure made up of nodes or vertices and edges. The edges connect any two nodes in the graph, and the nodes are also known as vertices.

Graph Terminologies in Data Structure

The following are some of the commonly used terms in graph data structure:

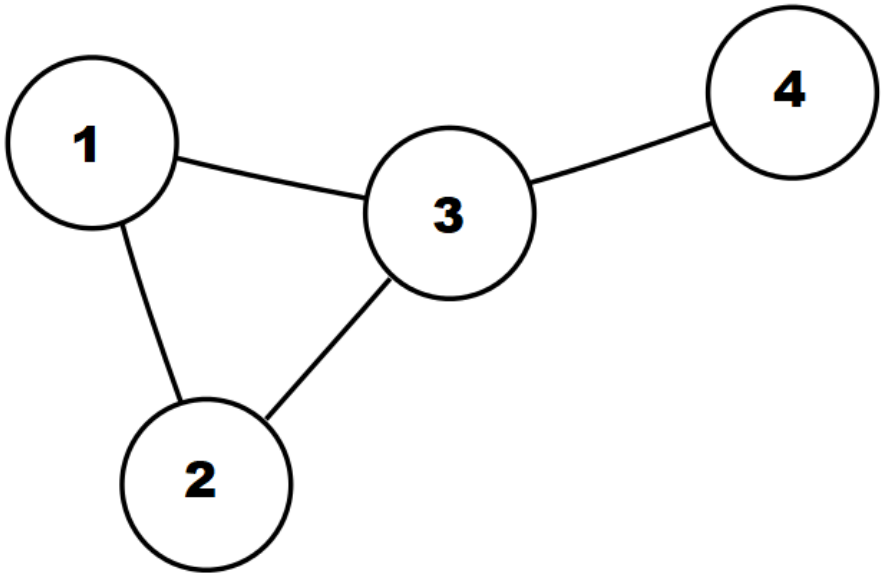
Term	Description
Vertex	Every individual data element is called a vertex or a node. In the above image, A, B, C, D & E are the vertices.
Edge (Arc)	It is a connecting link between two nodes or vertices. Each edge has two ends and is represented as (startingVertex, endingVertex).
Undirected Edge	It is a bidirectional edge.
Directed Edge	It is a unidirectional edge.
Weighted Edge	An edge with value (cost) on it.
Degree	The total number of edges connected to a vertex in a graph.
Indegree	The total number of incoming edges connected to a vertex.
Outdegree	The total number of outgoing edges connected to a vertex.

Self-loop	An edge is called a self-loop if its two endpoints coincide with each other.

Types of Graphs in Data Structure

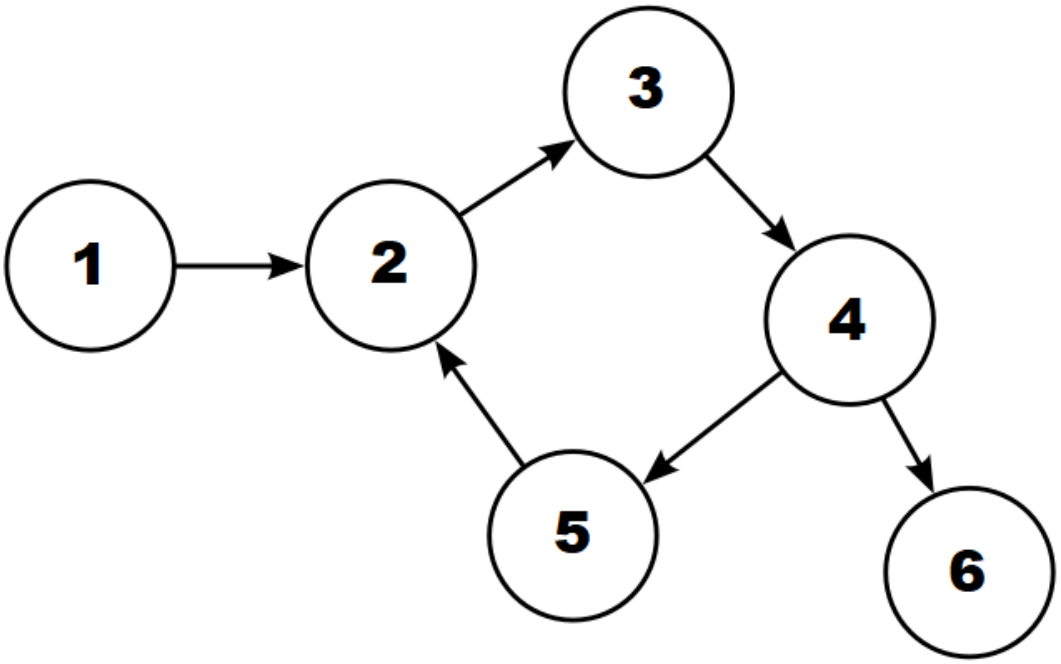
The most common types of graphs in the data structure are mentioned below:

1. Undirected: A graph in which all the edges are bi-directional. The edges do not point in a specific direction.



Undirected Graph

2. Directed: A graph in which all the edges are uni-directional. The edges point in a single direction.

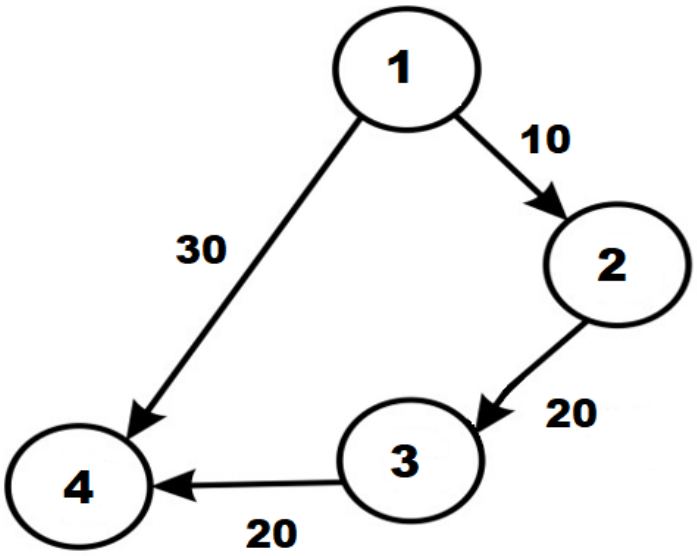


Directed Graph

3. Weighted Graph: A graph that has a value associated with every edge. The values corresponding to the edges are called weights. A value in a weighted graph can represent quantities such as cost, distance, and time, depending on the graph. Weighted graphs are typically used in modeling computer networks.

An edge in a weighted graph is represented as (u, v, w), where:

- u is the source vertex
- v is the destination vertex
- w represents the weight associated to go from u to v



Weighted Graph

4. Unweighted Graph: A graph in which there is no value or weight associated with the edge. All the graphs are unweighted by default unless there is a value associated.

An edge of an unweighted graph is represented as (u, v), where:

- u represents the source vertex
- v is the destination vertex

❖ Applications of Graphs in Data Structure

Graphs data structures have a variety of applications. Some of the most popular applications are:

- Helps to define the flow of computation of software programs.
- Used in Google maps for building transportation systems. In google maps, the intersection of two or more roads represents the node while the road connecting two nodes represents an edge. Google maps algorithm uses graphs to calculate the shortest distance between two vertices.
- Used in social networks such as Facebook and Linkedin.
- Operating Systems use Resource Allocation Graph where every process and resource acts as a node while edges are drawn from resources to the allocated process.
- Used in the world wide web where the web pages represent the nodes.

- Blockchains also use graphs. The nodes are blocks that store many transactions while the edges connect subsequent blocks.
- Used in modeling data.

❖ Graph Representations

In graph theory, a graph representation is a technique to store graph into the memory of computer.

To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

here are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

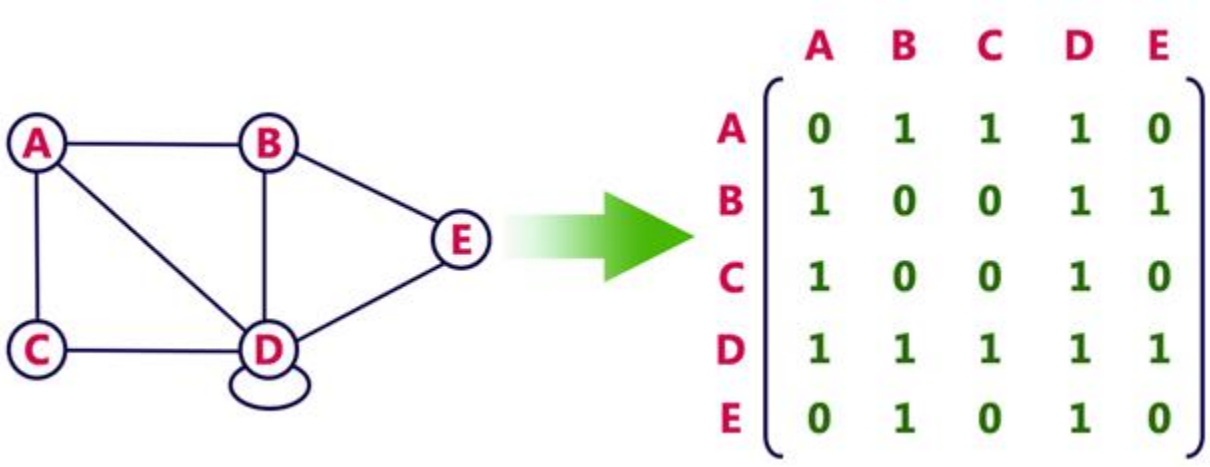
1. Adjacency Matrix

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$.
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

Example

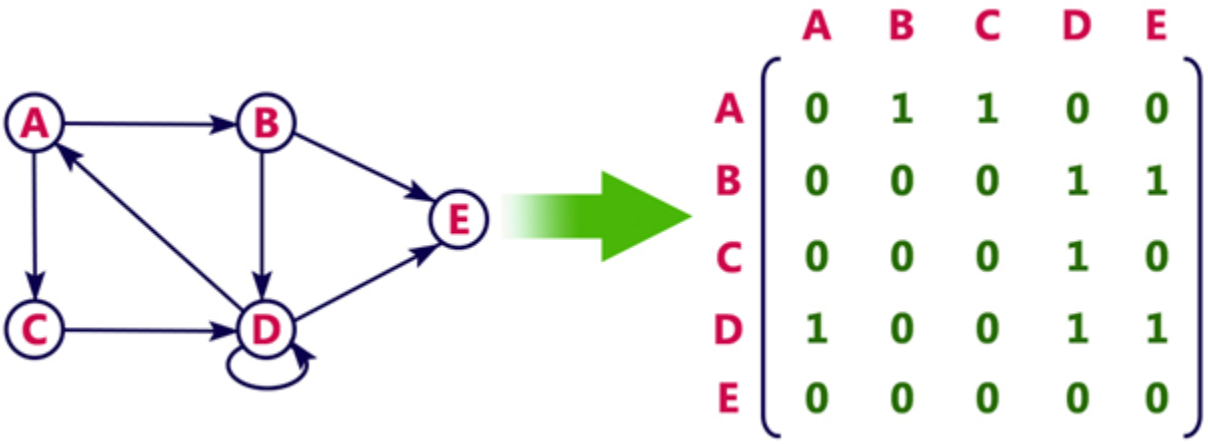
Consider the following **undirected graph representation**:

Undirected graph representation



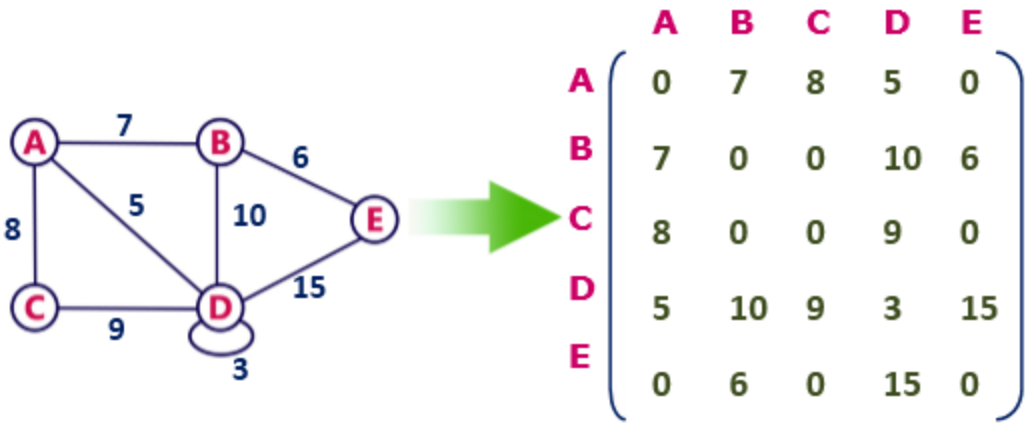
Directed graph represenation

See the directed graph representation:



In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.

Undirected weighted graph representation



Pros: Representation is easier to implement and follow.

Cons: It takes a lot of space and time to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph, which takes quite some time.

2. Incidence Matrix

In **Incidence matrix representation**, graph can be represented using a matrix of size:

Total number of vertices by total number of edges.

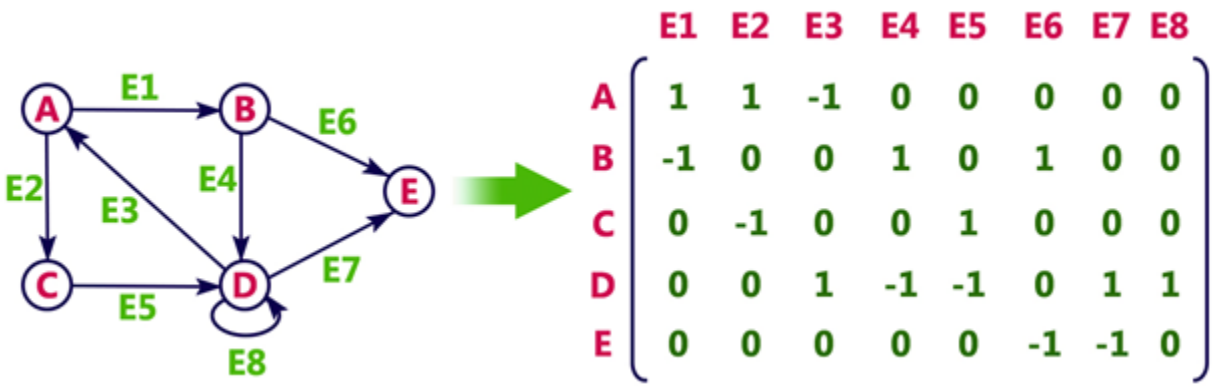
It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class. In this matrix, columns represent edges and rows represent vertices.

This matrix is filled with either **0 or 1** or -1. Where,

- 0 is used to represent row edge which is not connected to column vertex.
- 1 is used to represent row edge which is connected as outgoing edge to column vertex.
- -1 is used to represent row edge which is connected as incoming edge to column vertex.

Example

Consider the following directed graph representation.



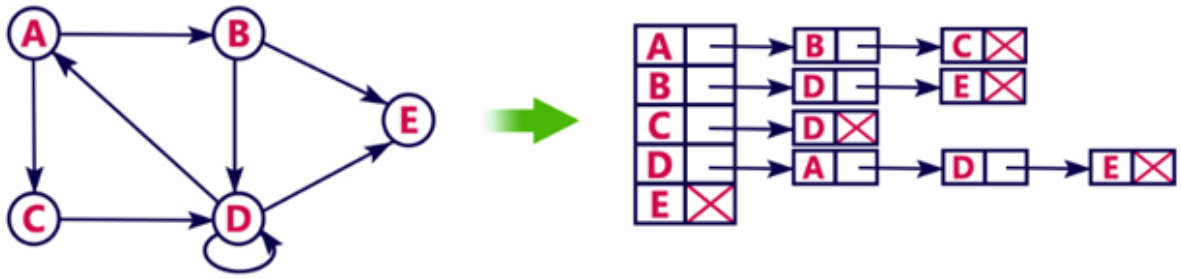
---

3. Adjacency List

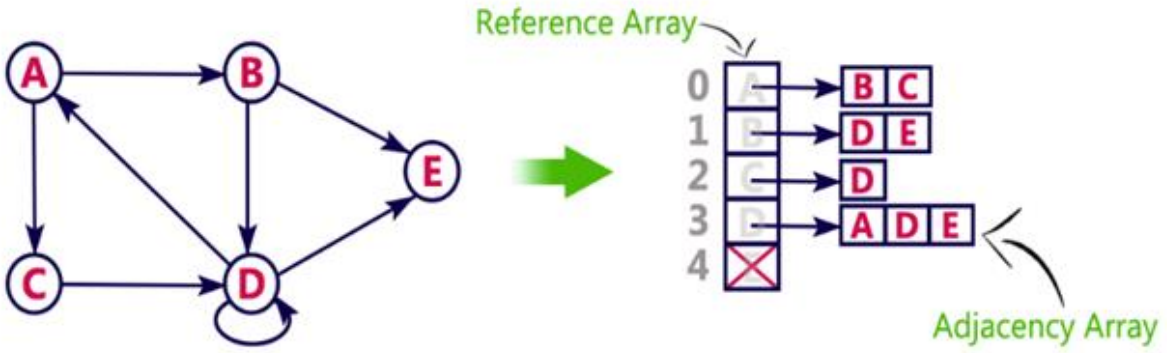
- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbors of v.

Example

Let's see the following directed graph representation implemented using linked list:



We can also implement this representation using array as follows:



Pros:

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

Cons:

- The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

❖ Operations on Graph in Data Structure

The operations you perform on the graphs in data structures are listed below:

- Creating graphs
- Insert vertex
- Delete vertex
- Insert edge
- Delete edge

1. Creating Graphs

- There are two techniques to make a graph:

1. Adjacency Matrix

- The adjacency matrix of a simple labeled graph, also known as the connection matrix, is a matrix

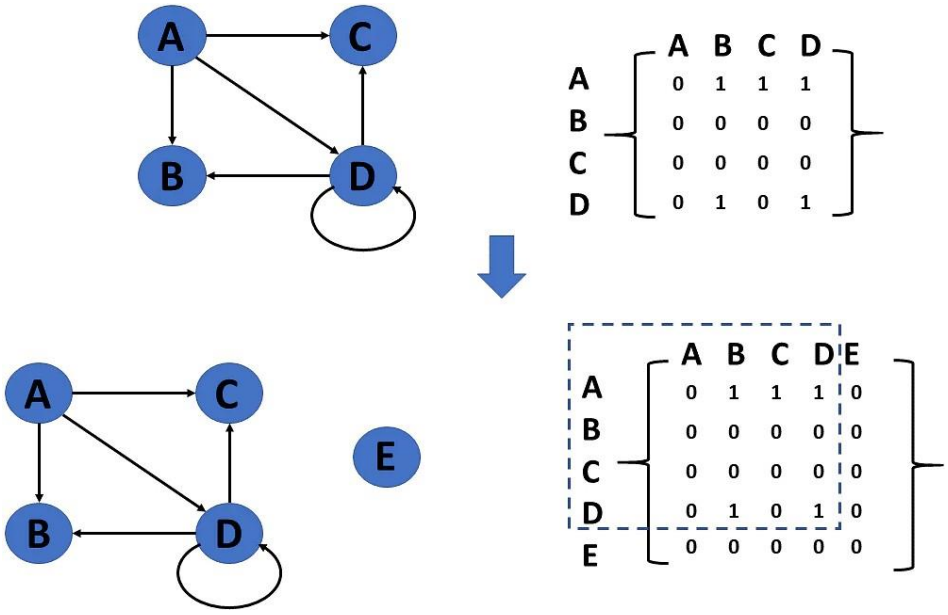
- with rows and columns labeled by graph vertices and a 1 or 0 in position depending on whether they
- are adjacent or not.

2. Adjacency List

- A finite graph is represented by an adjacency list, which is a collection of unordered lists. Each
- unordered list describes the set of neighbors of a particular vertex in the graph within an adjacency
- list.

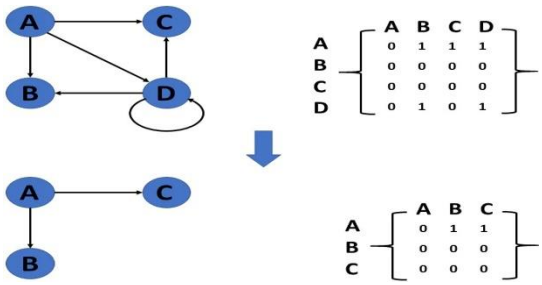
Insert Vertex

When you add a vertex that after introducing one or more vertices or nodes, the graph's size grows
by one, increasing the matrix's size by one at the row and column levels.



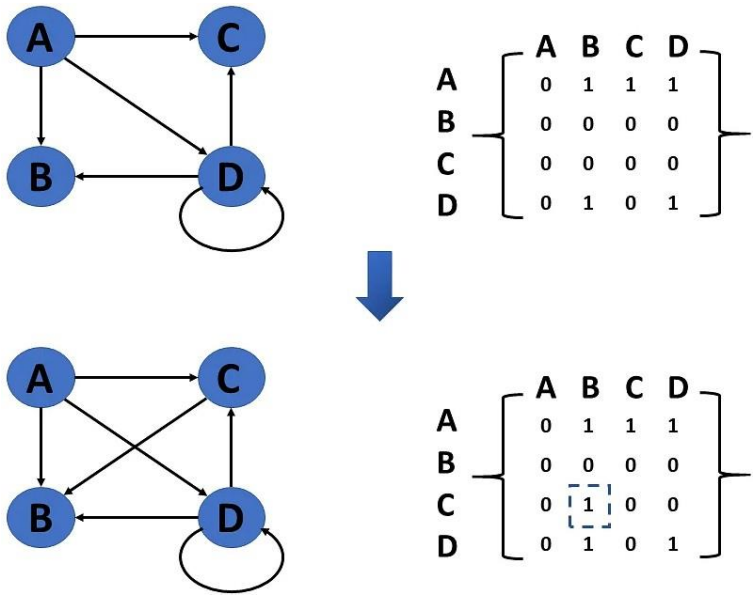
Delete Vertex

- Deleting a vertex refers to removing a specific node or vertex from a graph that has been saved.
- If a removed node appears in the graph, the matrix returns that node. If a deleted node does not appear in the graph, the matrix returns the node not available.



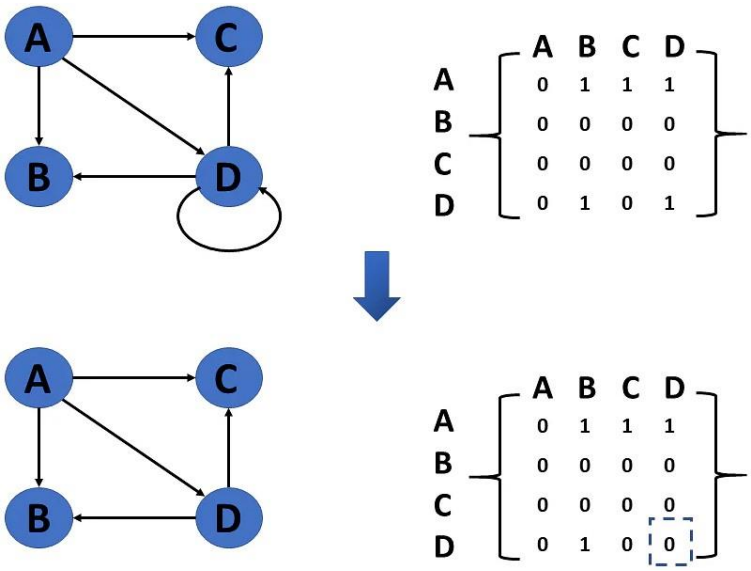
Insert Edge

Connecting two provided vertices can be used to add an edge to a graph.



Delete Edge

The connection between the vertices or nodes can be removed to delete an edge.



❖ **Graph Traversal Technique in Data Structure**

- Graph traversal is a technique to visit the each nodes of a graph G. It is also use to calculate the order of vertices in traverse process. We visit all the nodes starting from one node which is connected to each other without going into loop.
- Basically in graph it may happen some time visiters can visit one node more than once. So, this may cause the going the visiters into infinite loop. So, to protect from this infinite loop condition we keep record of each vertices. Like if visiters visited vertex then the value will be zero if not, then one.
  - ✓ **There are two graph traversal techniques**

1. Breadth-first search
2. Depth-first search

1. Breadth First Search (BFS) Traversal in Data Structure

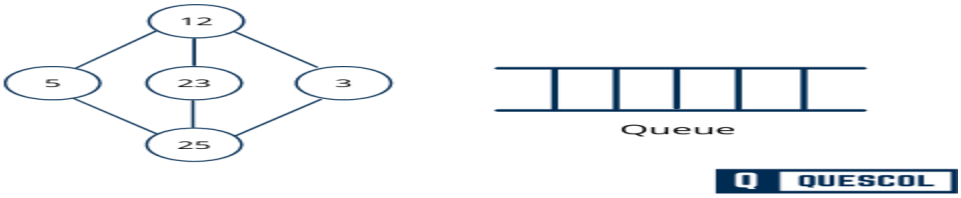
- Breadth-first search graph traversal techniques use a queue data structure as an auxiliary data structure to store nodes for further processing.
- The size of the queue will be the maximum total number of vertices in the graph.

❖ **Steps to implement BFS traversal**

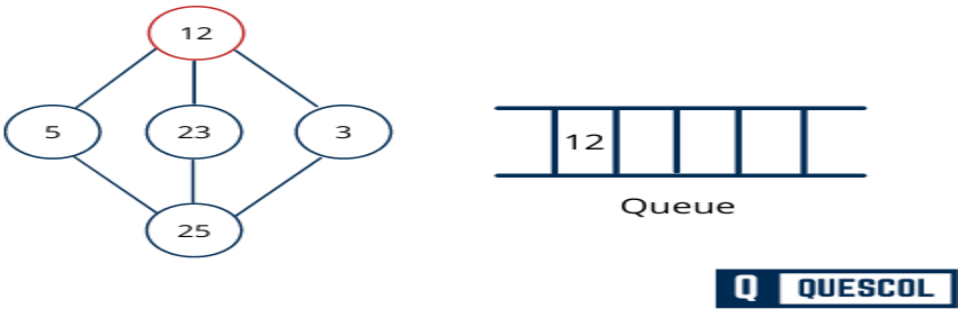
Step 1 – First define a Queue of size n. Where n is the total number of vertices in the graph.
Step 2 – Select any vertex which is a starting point from where traversal will start.
Step 3 – Visit starting vertex and insert it into the Queue.
Step 4 – Visit all the non-visited adjacent vertices which is connected to it and insert all non-visited vertices into the Queue.
Step 5 – When there is no new vertex to be visited from the element which is top in a queue, Remove the top element which is the vertex from the queue.
Step 6 – Repeat steps 4 and 6 until the queue becomes empty.
Step 7 – When all elements removed from the queue, then produce the final spanning tree by removing unused edges from the graph.

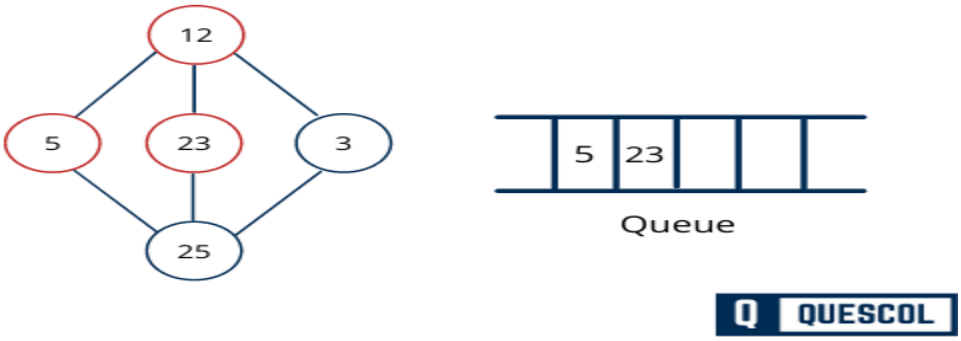
- **Example Of BFS Graph Traversal**

Initialize the queue

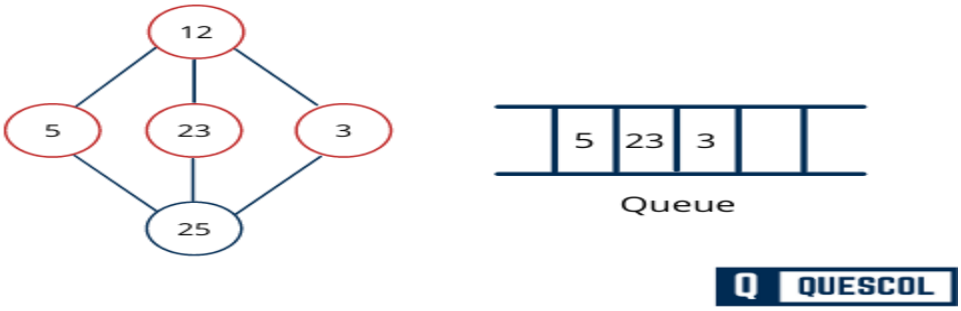


We start visiting from vertex **12** that can be considered as starting node, and mark it as visited.





Next, the unvisited adjacent node from **12** is **3**. We mark it as visited and enqueue it.



Now, all the connected adjacent node from **12** is traversed and no unvisited adjacent nodes left. So, we dequeue and find all connect vertex from **5**.

From **5** we have **25** as unvisited adjacent node. We mark it as visited and enqueue it.

Now if all nodes visited, we will dequeue all nodes from queue.

So BFS Traversal output is : 12, 5, 23, 3, 25

Complexity Analysis of BFS

Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Space Complexity: O(V).

Since, an extra visited array is needed of size V.

- ❖ **Applications of BFS**

- Finding the Shortest path in an unweighted graph
- Find a solution to a game with the least number of moves. In such a scenario each state of the game can be represented by a node and state transitions as edges
- Finding Connected Components in an unweighted graph
- Level Order Traversal in Tree
- Find the shortest paths in graphs with weights 0/1

2. Depth First Search(DFS) Traversal in Data Structure

- DFS stands for Depth First Search, is one of the graph traversal algorithms that uses Stack data structure.
- In DFS Traversal go as deep as possible of the graph and then backtrack once reached a vertex that has all its adjacent vertices already visited.
- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack data structure to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
- DFS traversal for tree and graph is similar but the only difference is that a graph can have a cycle but the tree does not have any cycle. So in the graph we have additional array which keeps the record of visited array to protect from infinite loop and not visited again the visited node.

❖ Steps to implement Depth First Traversal

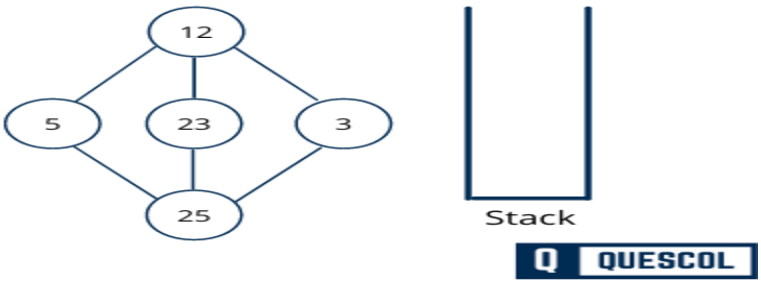
Step 1 – Visit all adjacent unvisited vertex. Mark it as visited. Print it and Push it in a stack.

Step 2 − If no adjacent vertex is found, pop up a vertex from the stack.

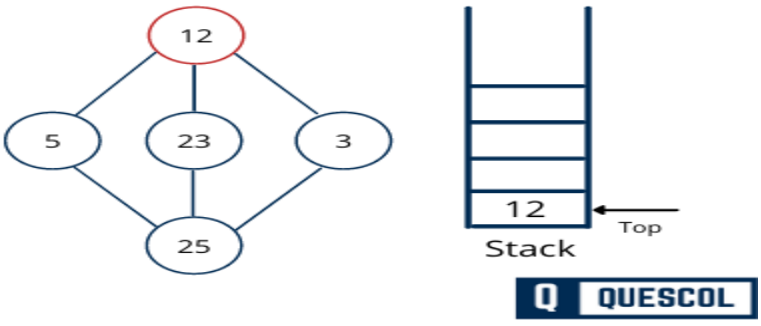
Step 3 − Repeat Step 1 and Step 2 until the stack is empty

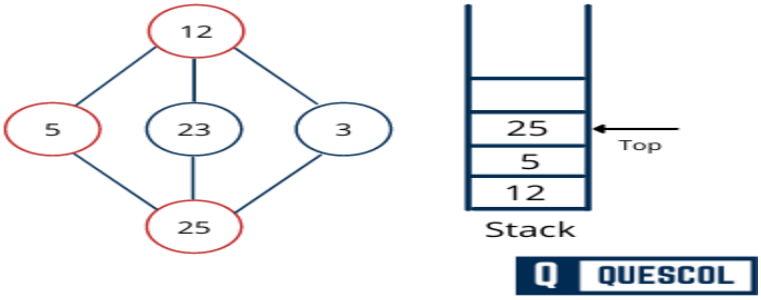
❖ Example:-

Initialize the stack

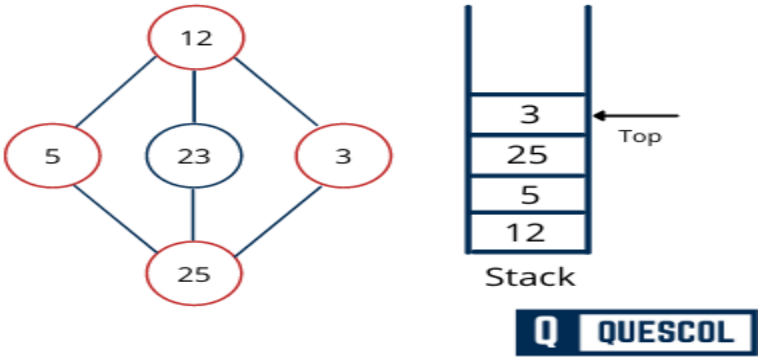


Mark **12** as visited and push into stack. Now Explore any unvisited adjacent node from **12**. In our example We have three nodes. We can pick any of them. Here we are going to pick **5**.

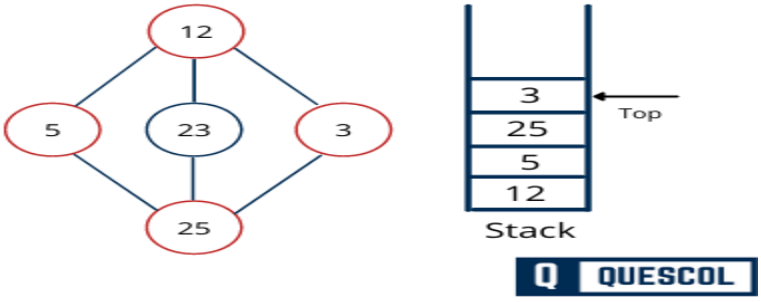




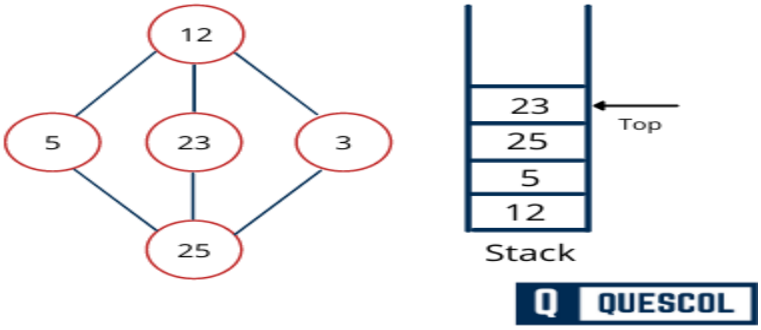
We choose **3**, mark it as visited and put onto the stack. Here **3** does not have any unvisited adjacent node. So, we pop **3** from the stack.



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **25** to be on the top of the stack.



Only unvisited adjacent node is from **D** is **23** now. So we visit **23**, mark it as visited and put it onto the stack.



Output of DFS Traversal will be : 3, 23, 25, 5, 12
Complexity Analysis

Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.
Space Complexity: O(V).
Since, an extra visited array is needed of size V.

Application of DFS

- Find a path from the source vertex to other vertices
- Find the connected components in a graph
- Topological Sorting
- Find bridges and articulation points in a graph
- Find LCA of two nodes in a graph
- Find cycles in a directed and undirected graph

❖ Spanning tree

What is a spanning tree?

- A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.
- A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

- A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125.}$

Applications of the spanning tree

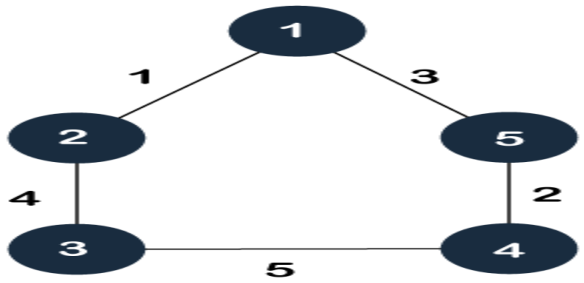
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

- Cluster Analysis
- Civil network planning
- Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

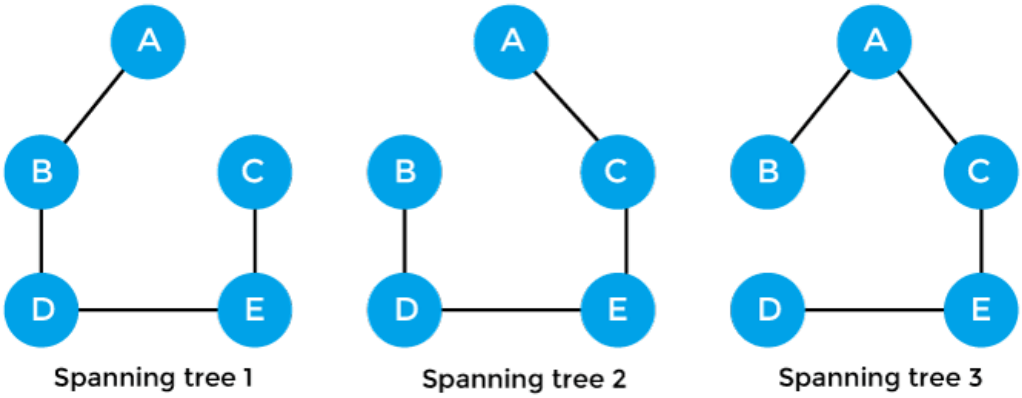
Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

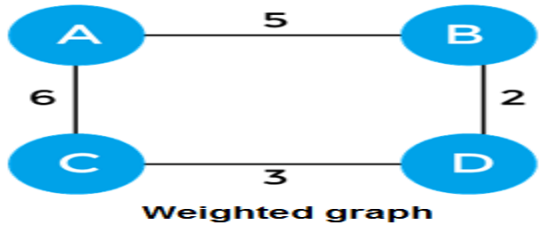
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Minimum Spanning tree

A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

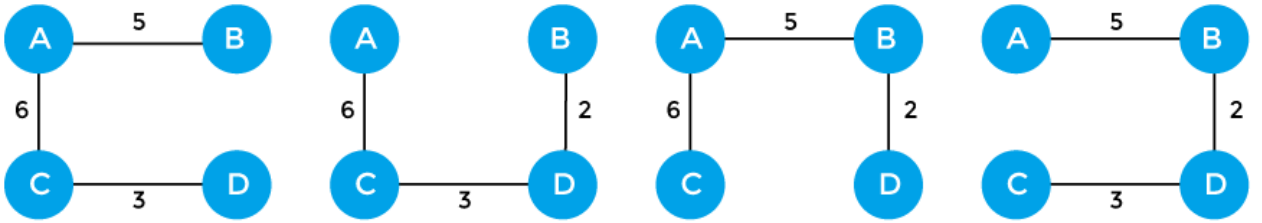
Example of minimum spanning tree

Let's understand the minimum spanning tree with the help of an example.



Weighted graph

The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



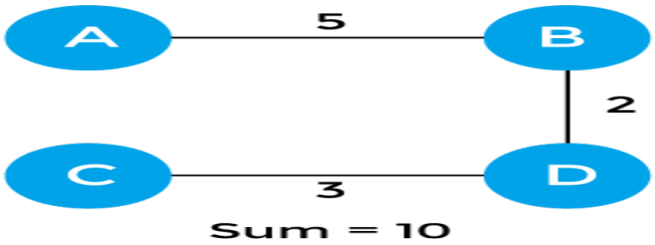
Sum = 14
Minimum spanning tree - 1

Sum = 11
Minimum spanning tree - 2

Sum = 13
Minimum spanning tree - 3

Sum = 10
Minimum spanning tree - 4

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Sum = 10

Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

- Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
- It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

- **Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

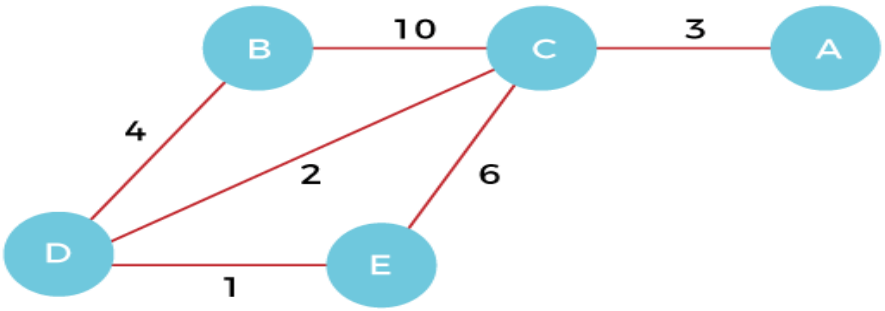
The applications of prim's algorithm are -

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

Example of prim's algorithm

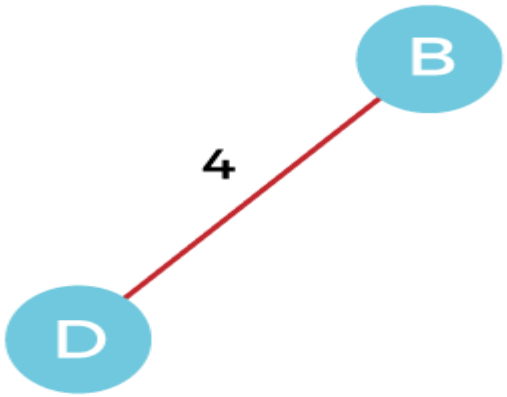
Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

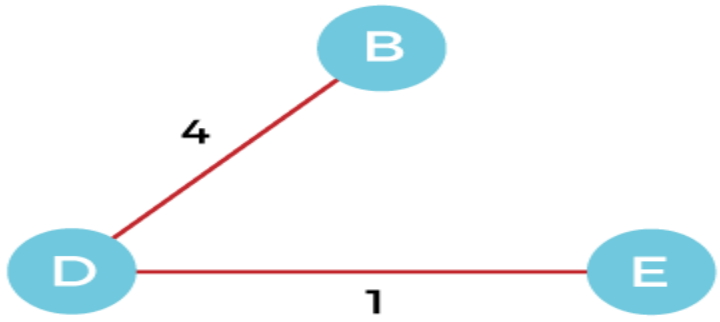


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

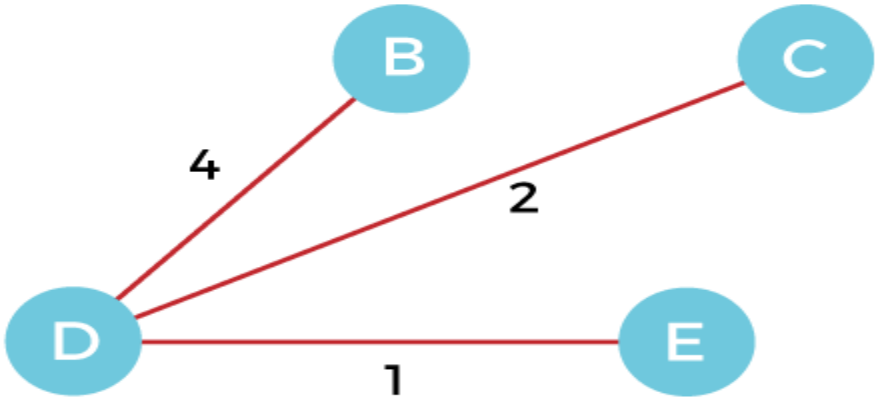
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



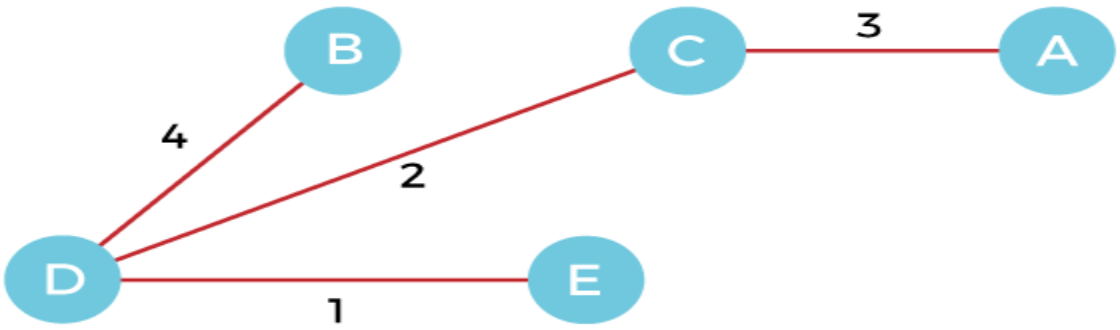
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

Kruskal's Algorithm

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

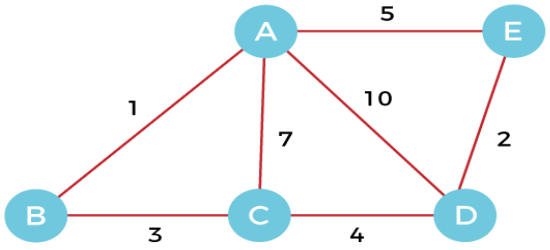
The applications of Kruskal's algorithm are -

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

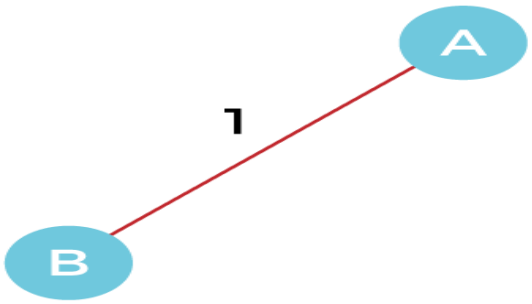
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

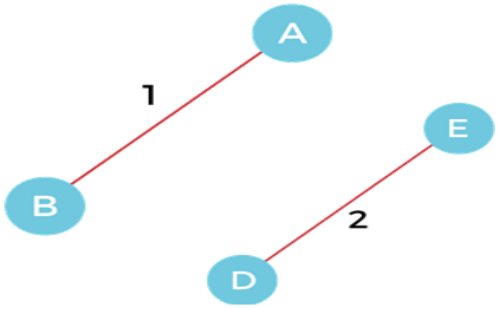
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

Step 1 - First, add the edge **AB** with weight **1** to the MST.

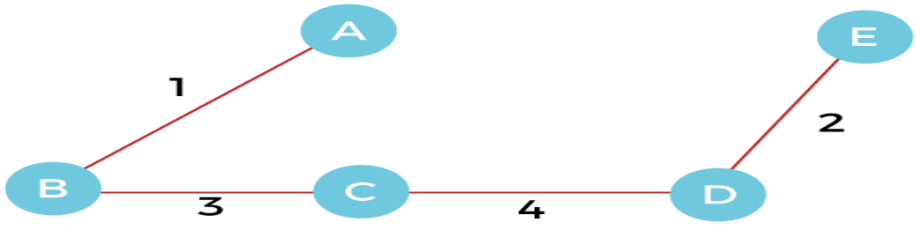


Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.

Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

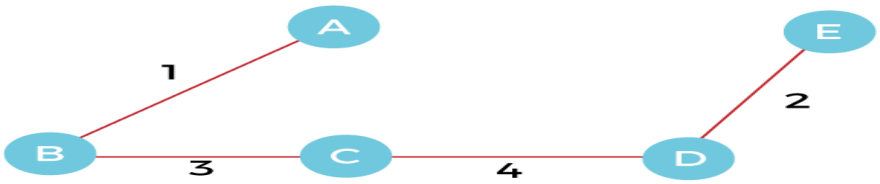


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Algorithm

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
6. (**for** combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

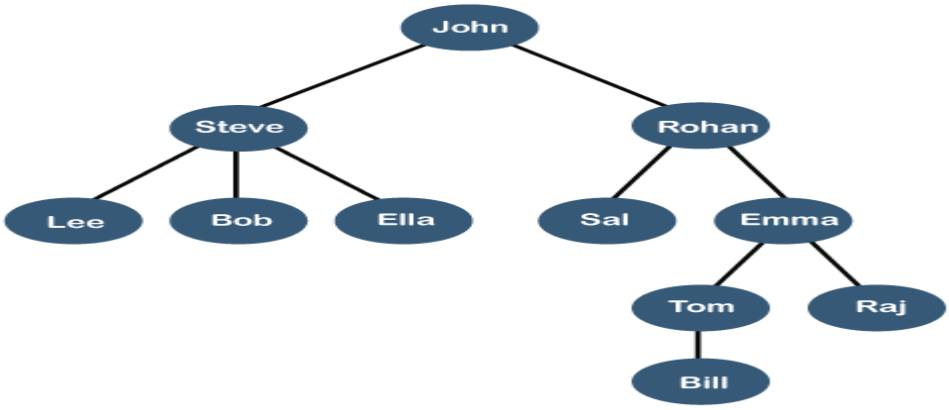
UNIT II-Tree Data Structure

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges", both directed and undirected.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



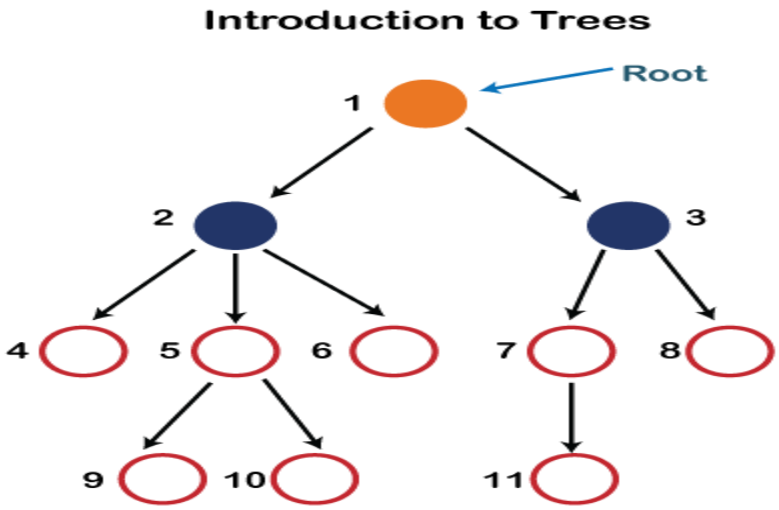
The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***. Tom has one direct report named ***Bill***. This particular logical structure is known as a ***Tree***. Its structure is similar to the real tree, so it is named a ***Tree***. In this structure, the ***root*** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.

Let's consider the tree structure, which is shown below:



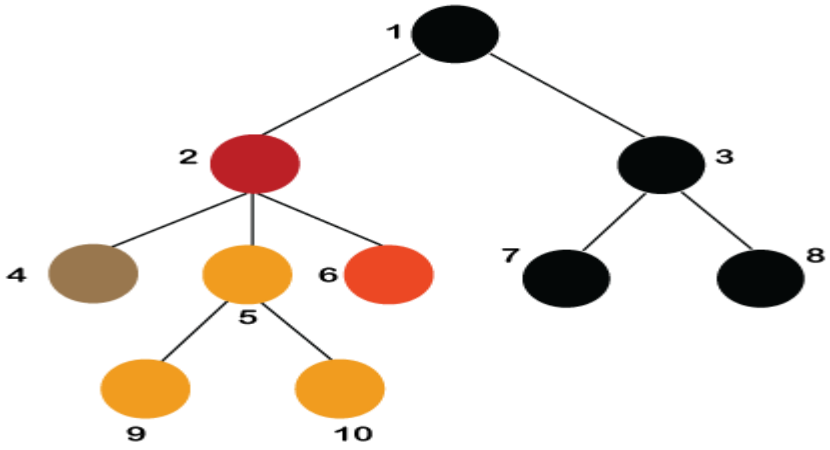
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.

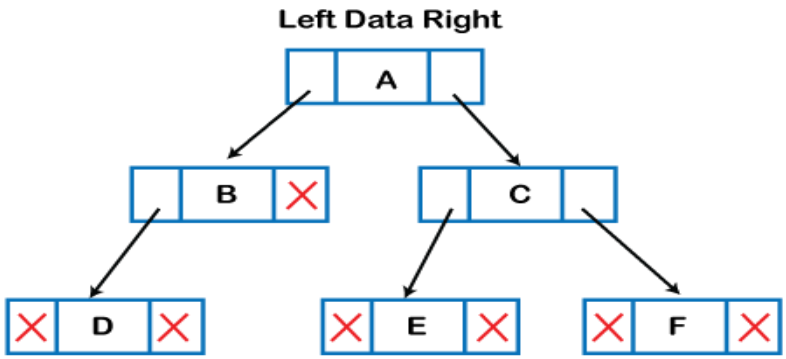


- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

```
1. struct node
2. {
3.   int data;
4. struct node *left;
5. struct node *right;
6. }
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

The following are the applications of trees:

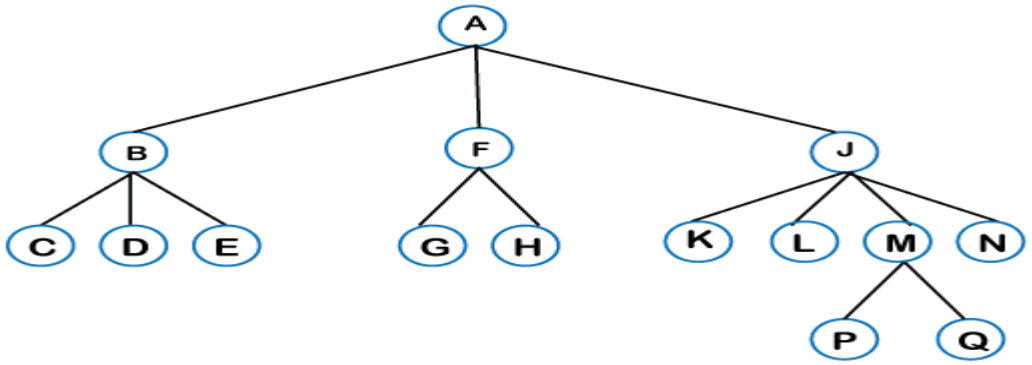
- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.

- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

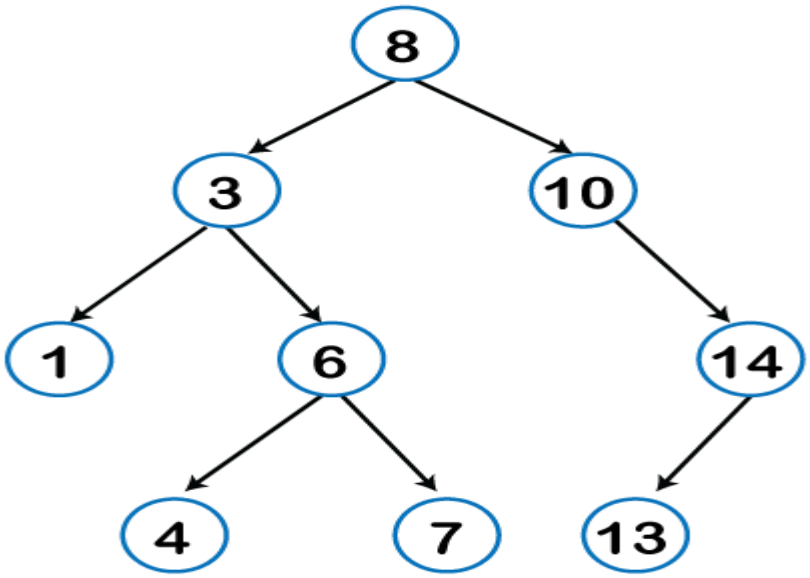
The following are the types of a tree data structure:

- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered. Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

- **Binary tree:** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



Binary Search tree: Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer).
Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

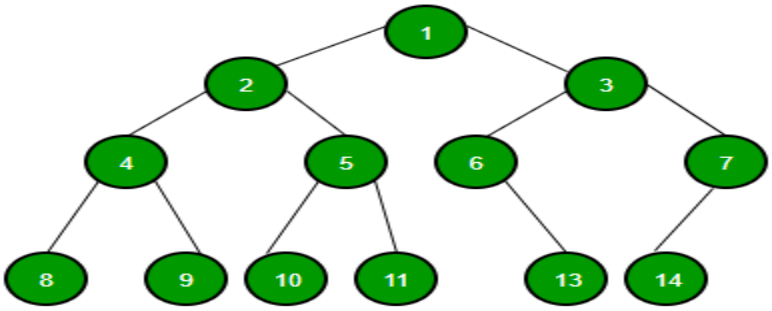
- **AVL tree**

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

Binary Tree Data Structure

Binary Tree is defined as a tree data structure where each node has at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



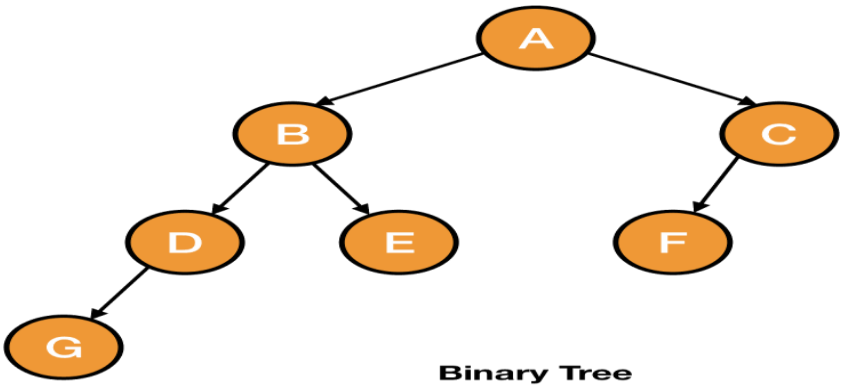
Binary Tree Representation

A Binary tree is represented by a pointer to the topmost node (commonly known as the “root”) of the tree. If the tree is empty, then the value of the root is NULL. Each node of a Binary Tree contains the following parts:

1. Data
2. Pointer to left child
3. Pointer to right child

Binary Trees

A binary tree is a tree in which every node has at most two children.

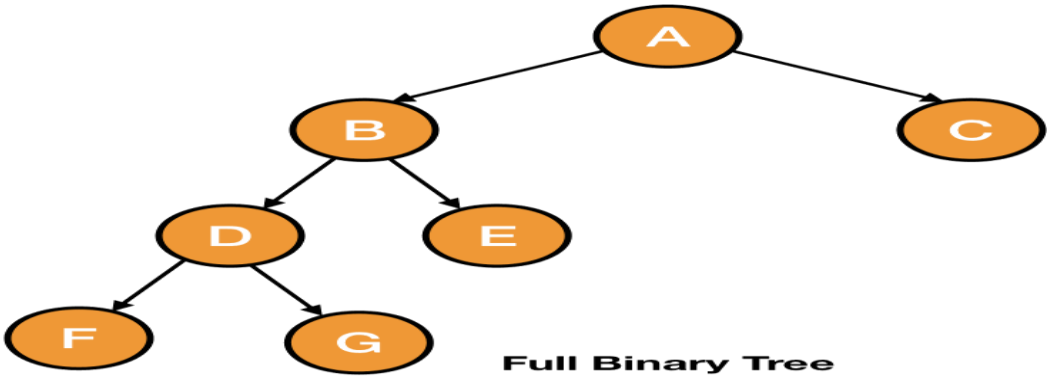


Binary Tree

As you can see in the picture given above, a node can have less than 2 children but not more than that.

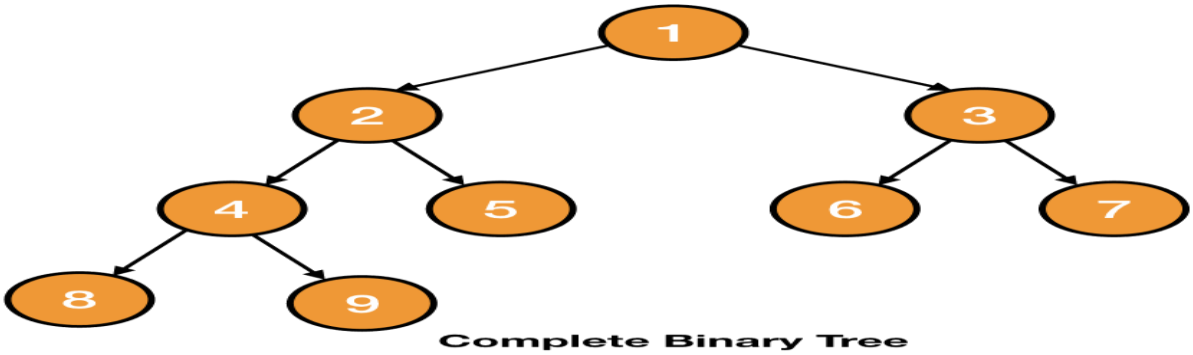
We can also classify a binary tree into different categories. Let's have a look at them:

Full Binary Tree → A binary tree in which every node has 2 children except the leaves is known as a full binary tree.



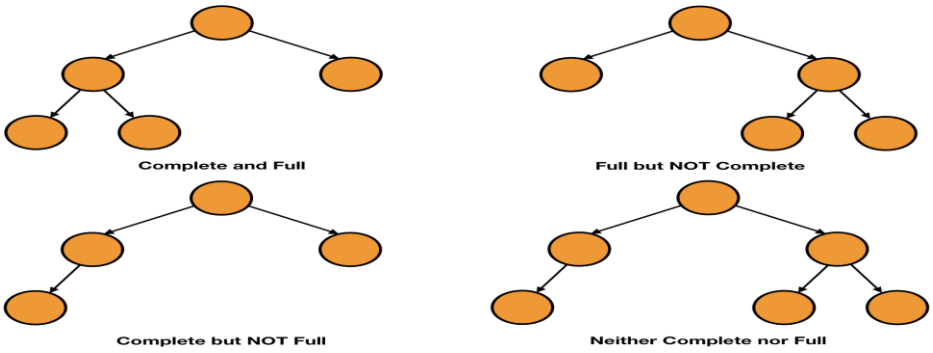
Full Binary Tree

Complete Binary Tree → A binary tree which is completely filled with a possible exception at the bottom level i.e., the last level may not be completely filled and the bottom level is filled from left to right.



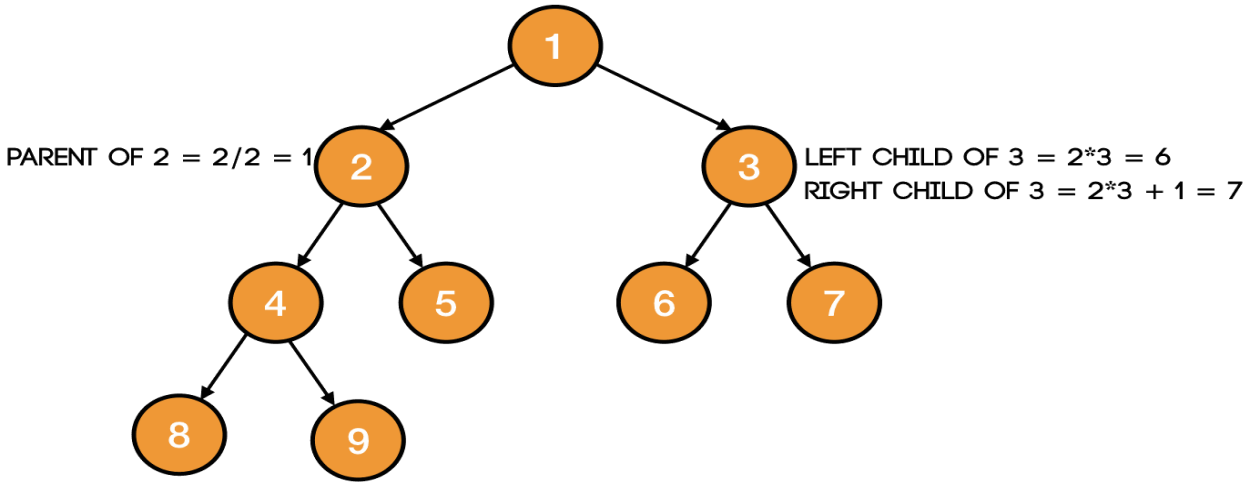
Complete Binary Tree

Let's look at this picture to understand the difference between a full and a complete binary tree.

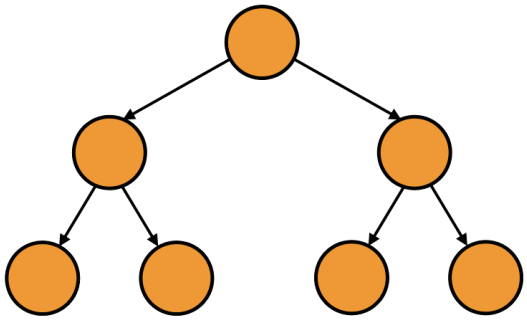


A complete binary tree also holds some important properties. So, let's look at them.

- The **parent of node i** is ⌊i2⌋⌊2⌋. For example, the parent of node 4 is 2 and the parent of node 5 is also 2.
- The **left child of node i** is 2i2.
- The **right child of node i** is 2i+12+1



Perfect Binary Tree → In a perfect binary tree, each leaf is at the same level and all the interior nodes have two children.



Perfect Binary Tree

Thus, a perfect binary tree will have the maximum number of nodes for all alternative binary trees of the same height and it will be $2^{h+1}-1$ $2h+1-1$ which we are going to prove next.

Applications of Binary Tree:

- In compilers, Expression Trees are used which is an application of binary trees.
- Huffman coding trees are used in data compression algorithms.
- Priority Queue is another application of binary tree that is used for searching maximum or minimum in O(1) time complexity.
- Represent hierarchical data.
- Used in editing software like Microsoft Excel and spreadsheets.
- Useful for indexing segmented at the database is useful in storing cache in the system,
- Syntax trees are used for most famous compilers for programming like GCC, and AOCL to perform arithmetic operations.
- For implementing priority queues.
- Used to find elements in less time (binary search tree)
- Used to enable fast memory allocation in computers.
- Used to perform encoding and decoding operations.

- Binary trees can be used to organize and retrieve information from large datasets, such as in inverted index and k-d trees.
- Binary trees can be used to represent the decision-making process of computer-controlled characters in games, such as in decision trees.
- Binary trees can be used to implement searching algorithms, such as in binary search trees which can be used to quickly find an element in a sorted list.
- Binary trees can be used to implement sorting algorithms, such as in heap sort which uses a binary heap to sort elements efficiently.

Binary Tree Traversal in Data Structure

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree.

There are three ways which we use to traverse a tree .

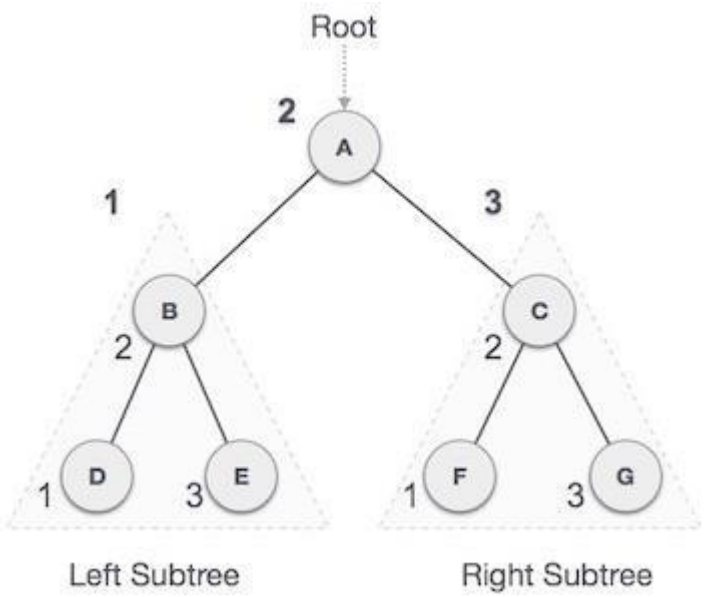
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**.**B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be –

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −

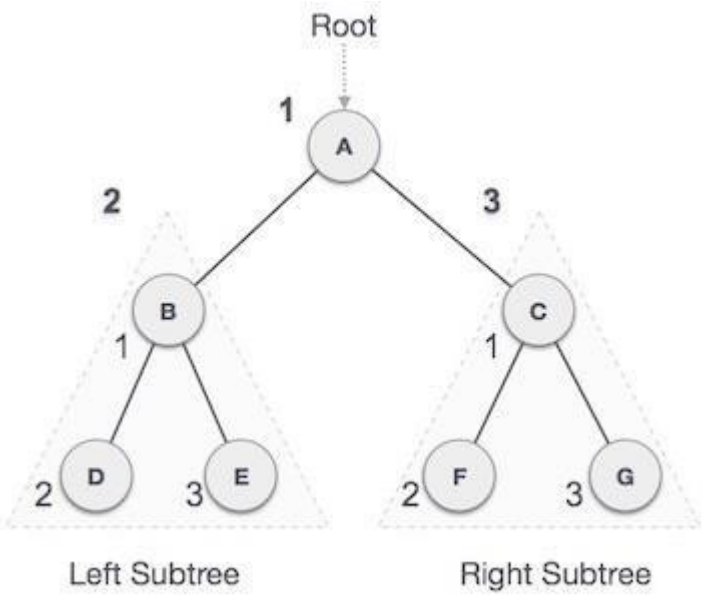
Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −

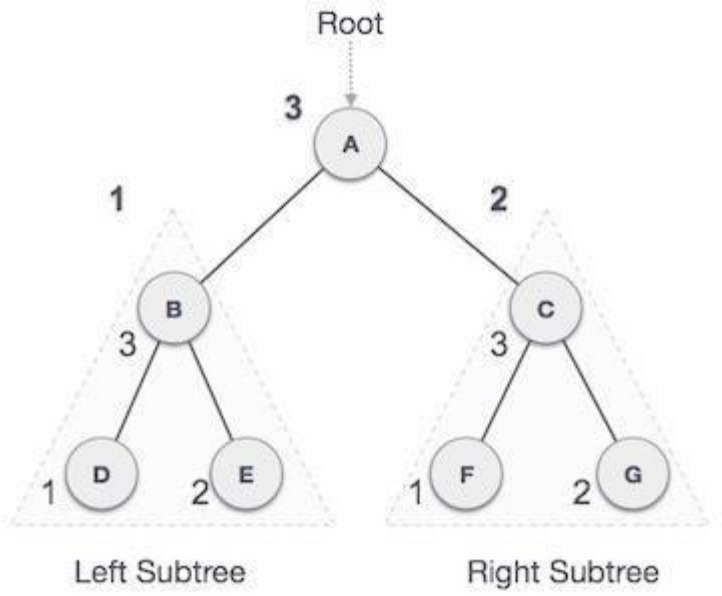
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

Expression Tree in Data Structure

An expression tree in data structure is used to represent an expression in the form of a tree.

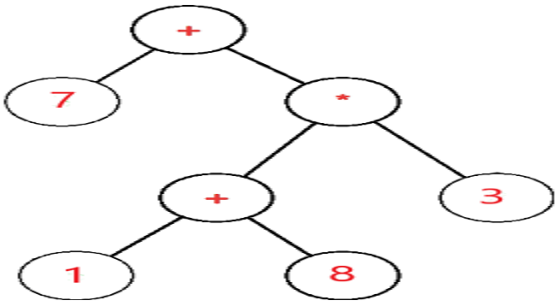
The expression tree is a tree used to represent the various expressions. The tree data structure is used to represent the expressional statements. In this tree, the internal node always denotes the operators.

- The leaf nodes always denote the operands.
- The operations are always performed on these operands.
- The operator present in the depth of the tree is always at the highest priority.
- The operator, which is not much at the depth in the tree, is always at the lowest priority compared to the operators lying at the depth.

- The operand will always present at a depth of the tree; hence it is considered the **highest priority** among all the operators.
- In short, we can summarize it as the value present at a depth of the tree is at the highest priority compared with the other operators present at the top of the tree.
- The main use of these expression trees is that it is used to **evaluate, analyze** and **modify** the various expressions.
- It is also used to find out the associativity of each operator in the expression.
- For example, the + operator is the left-associative and / is the right-associative.
- The dilemma of this associativity has been cleared by using the expression trees.
- These expression trees are formed by using a context-free grammar.
- We have associated a rule in context-free grammars in front of each grammar production.
- These rules are also known as semantic rules, and by using these semantic rules, we can be easily able to construct the expression trees.
- It is one of the major parts of compiler design and belongs to the semantic analysis phase.
- In this semantic analysis, we will use the syntax-directed translations, and in the form of output, we will produce the annotated parse tree.
- An annotated parse tree is nothing but the simple parse associated with the type attribute and each production rule.
- The main objective of using the expression trees is to make complex expressions and can be easily be evaluated using these expression trees.
- It is immutable, and once we have created an expression tree, we can not change it or modify it further.
- To make more modifications, it is required to construct the new expression tree wholly.
- It is also used to solve the postfix, prefix, and infix expression evaluation.

Expression trees play a very important role in representing the **language-level** code in the form of the data, which is mainly stored in the tree-like structure. It is also used in the memory representation of the **lambda** expression. Using the tree data structure, we can express the lambda expression more transparently and explicitly. It is first created to convert the code segment onto the data segment so that the expression can easily be evaluated.

The expression tree is a binary tree in which each external or leaf node corresponds to the operand and each internal or parent node corresponds to the operators so for example expression tree for 7 + ((1+8)*3) would be:



Use of Expression tree

1. The main objective of using the expression trees is to make complex expressions and can be easily be evaluated using these expression trees.
2. It is also used to find out the associativity of each operator in the expression.
3. It is also used to solve the postfix, prefix, and infix expression evaluation.

Generation of Expression Tree

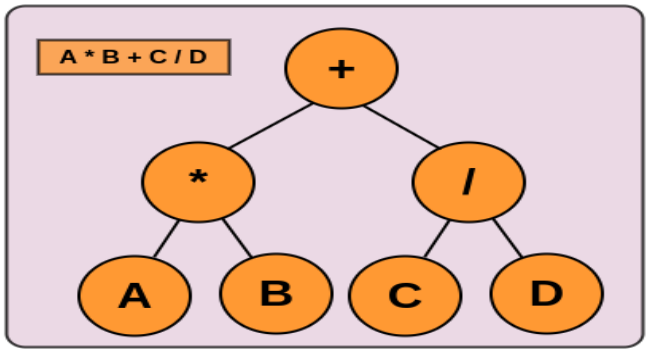
Let's understand the process of generation of an expression tree intuitively using the expression tree described in the previous section.
Expression: A * B + C / D

Scan the expression and according to the associativity and precedence find the operator which will be evaluated at last.

- In our example, the + operator will be evaluated in the last so keep it as the root node and divide the remaining expression into left and right subtrees.

- Now solve the left and right subtree similarly.

- After solving the right and left subtree our final expression tree would become like the image given below:



There are different types of expression formats:

- Prefix expression
- Infix expression and
- Postfix expression

Expression Tree is a special kind of binary tree with the following properties:

- Each leaf is an operand. Examples: a, b, c, 6, 100
- The root and internal nodes are operators. Examples: +, -, *, /, ^
- Subtrees are subexpressions with the root being an operator.

Traversal Techniques

There are 3 standard traversal techniques to represent the 3 different expression formats.

Inorder Traversal

We can produce an infix expression by recursively printing out

- the left expression,
- the root, and
- the right expression.

Postorder Traversal

The postfix expression can be evaluated by recursively printing out

- the left expression,
- the right expression and

- then the root

Preorder Traversal

We can also evaluate prefix expression by recursively printing out:

- the root,
- the left expressoion and
- the right expression.

Unit IV :- Heap Data Structure

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

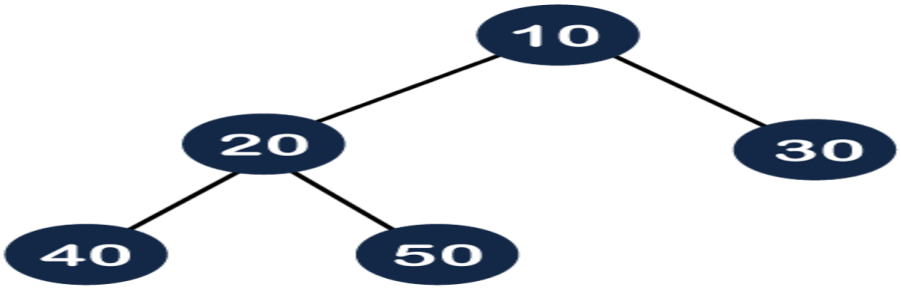
❖ What is Heap?

- 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.
- Heap is a specialized tree data structure. The heap comprises the topmost node called a root (parent). Its second child is the root's left child, while the third node is the root's right child. The successive nodes are filled from left to right. The parent-node key compares to that of its offspring, and a proper arrangement occurs. The tree is easy to visualize where each entity is called a node. The node has unique keys for identification.

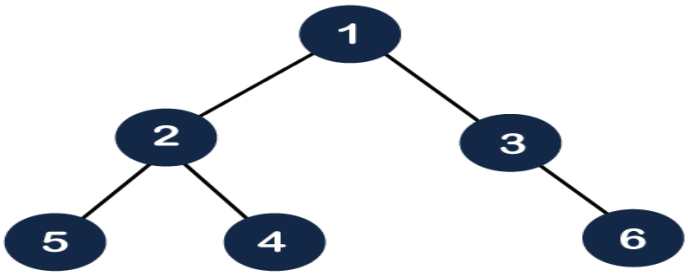
❖ What is a complete binary tree?

- 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.

❖ Why do you need Heap Data Structure?

Here are the main reasons for using Heap Data Structure:

- The heap data structure allows deletion and insertion in logarithmic time – $O(\log_2 n)$.
- The data in the tree is fashioned in a particular order. Besides updating or querying things such as a maximum or minimum, the programmer can find relationships between the parent and the offspring.
- You can apply the concept of the Document Object Model to assist you in understanding the heap data structure.

❖ Application of Heap Data Structure:

- **Priority queues:** The heap data structure is commonly used to implement priority queues, where elements are stored in a heap and ordered based on their priority. This allows constant-time access to the highest-priority element, making it an efficient data structure for managing tasks or events that require prioritization.
- **Heapsort algorithm:** The heap data structure is the basis for the heapsort algorithm, which is an efficient sorting algorithm with a worst-case time complexity of O(n log n). The heapsort algorithm is used in various applications, including database indexing and numerical analysis.
- **Memory management:** The heap data structure is used in memory management systems to allocate and deallocate memory dynamically. The heap is used to store the memory blocks, and the heap data structure is used to efficiently manage the memory blocks and allocate them to programs as needed.
- **Graph algorithms:** The heap data structure is used in various graph algorithms, including Dijkstra's algorithm, Prim's algorithm, and Kruskal's algorithm. These algorithms require efficient priority queue implementation, which can be achieved using the heap data structure.
- **Job scheduling**: The heap data structure is used in job scheduling algorithms, where tasks are scheduled based on their priority or deadline. The heap data structure allows efficient access to the highest-priority task, making it a useful data structure for job scheduling applications.

❖ Advantages of Heap Data Structure:

- **Efficient insertion and deletion:** The heap data structure allows efficient insertion and deletion of elements. When a new element is added to the heap, it is placed at the bottom of the heap and moved up to its correct position using the heapify operation. Similarly, when an element is removed from the heap, it is replaced by the bottom element, and the heap is restructured using the heapify operation.
- **Efficient priority queue:** The heap data structure is commonly used to implement a priority queue, where the highest priority element is always at the top of the heap. The heap

allows constant-time access to the highest priority element, making it an efficient data structure for implementing priority queues.

- **Guaranteed access to the maximum or minimum element**: In a max-heap, the top element is always the maximum element, and in a min-heap, the top element is always the minimum element. This provides guaranteed access to the maximum or minimum element in the heap, making it useful in algorithms that require access to the extreme values.
- **Space efficiency:** The heap data structure requires less memory compared to other data structures, such as linked lists or arrays, as it stores elements in a complete binary tree structure.
- **Heap-sort algorithm**: The heap data structure forms the basis for the heap-sort algorithm, which is an efficient sorting algorithm that has a worst-case time complexity of O(n log n).

❖ Disadvantages of Heap Data Structure:

- **Lack of flexibility:** The heap data structure is not very flexible, as it is designed to maintain a specific order of elements. This means that it may not be suitable for some applications that require more flexible data structures.
- **Not ideal for searching:** While the heap data structure allows efficient access to the top element, it is not ideal for searching for a specific element in the heap. Searching for an element in a heap requires traversing the entire tree, which has a time complexity of O(n).
- **Not a stable data structure:** The heap data structure is not a stable data structure, which means that the relative order of equal elements may not be preserved when the heap is constructed or modified.
- **Memory management:** The heap data structure requires dynamic memory allocation, which can be a challenge in some systems with limited memory. In addition, managing the memory allocated to the heap can be complex and error-prone.
- **Complexity:** While the heap data structure allows efficient insertion, deletion, and priority queue implementation, it has a worst-case time complexity of O(n log n), which may not be optimal for some applications that require faster algorithms.

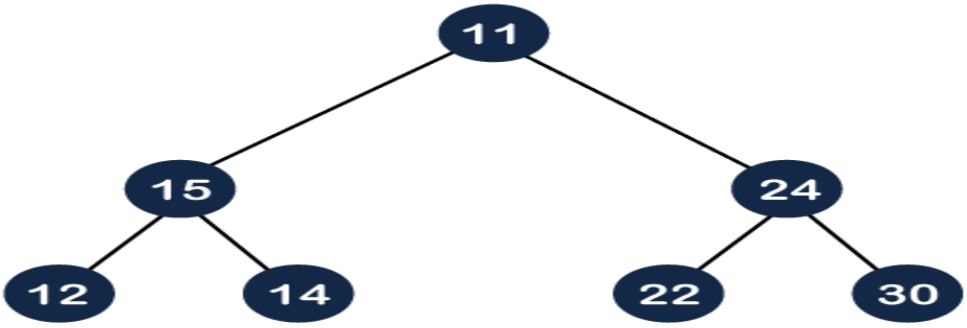
❖ Types of Heap Data Structure

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.
   - ✓ The value of the parent node should be less than or equal to either of its children.
   - ✓ 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]

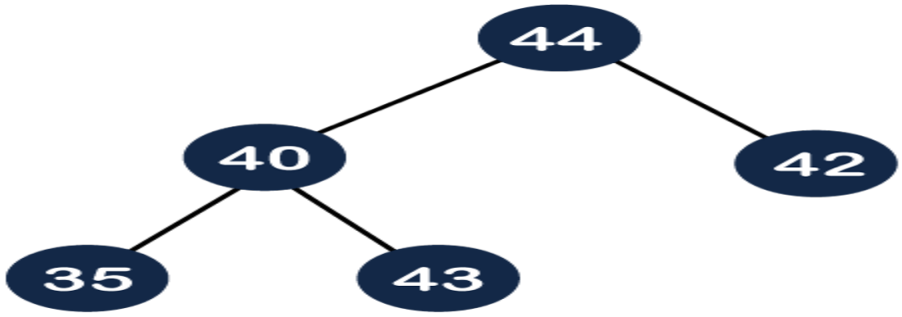
Let's understand the min-heap through an example.



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).

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.
   - ✓ The value of the parent node is greater than or equal to its children.
   - ✓ 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.

❖ ***Operations on Heaps***

The common operation involved using heaps are:

- **Heapify** → Process to rearrange the heap in order to maintain heap-property.

- **Find-max (or Find-min)** → find a maximum item of a max-heap, or a minimum item of a min-heap, respectively.
- **Insertion** → Add a new item in the heap.
- **Deletion** → Delete an item from the heap.
- **Extract Min-Max** → Returning and deleting the maximum or minimum element in max-heap and min-heap respectively.

1. Heapify

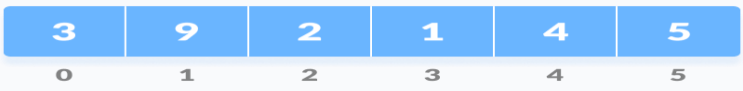
It is a process to rearrange the elements of the heap in order to maintain the heap property. It is done when a certain node causes an imbalance in the heap due to some operation on that node.

The heapify can be done in two methodologies:

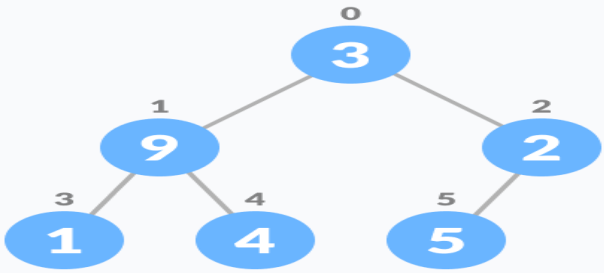
- **up_heapify()** → It follows the bottom-up approach. In this, we check if the nodes are following heap property by going in the direction of rootNode and if nodes are not following the heap property we do certain operations to let the tree follows the heap property.
- **down_heapify()** → It follows the top-down approach. In this, we check if the nodes are following heap property by going in the direction of the leaf nodes and if nodes are not following the heap property we do certain operations to let the tree follows the heap property.

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

1. Let the input array be Initial Array

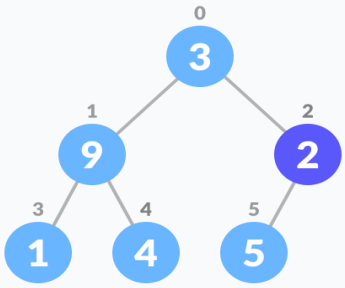


2. Create a complete binary tree from the array



Complete binary tree

3. Start from the first index of non-leaf node whose index is given by n/2 - 1.

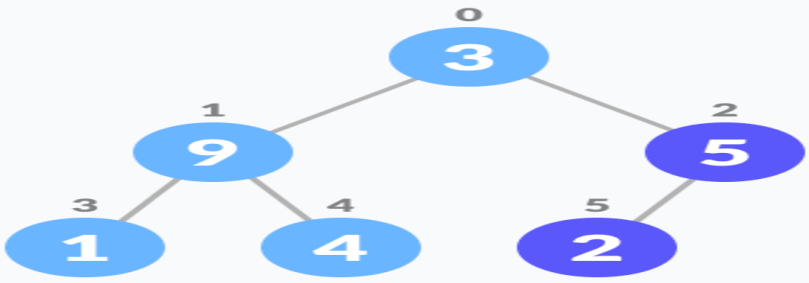


Start from the first on leaf node

4. Set current element i as largest.
5. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.

   If leftChild is greater than currentElement (i.e. element at ith index), set leftChildIndex as largest.

   If rightChild is greater than element in largest, set rightChildIndex as largest.
6. Swap largest with currentElement



7. Swap if necessary
8. Repeat steps 3-7 until the subtrees are also heapified.

2. *Insertion*

The insertion in the heap follows the following steps

- Insert the new element at the end of the heap.
- Since the newly inserted element can distort the properties of the Heap. So, we need to perform **up_heapify()** operation, in order to keep the properties of the heap in a bottom-up approach.

Initially the heap is **as** (It follows max-heap property)

New element to be inserted is 12
Step 1: Insert the new element at the end.

Step 2: Heapify the new element following bottom-up approach.
Final heap

3. *Deletion*

The deletion operations follow the following step:

- Replace the element to be deleted by the last element in the heap.
- Delete the last item from the heap.
- Now, the last element is placed at some position in heap, it may not follow the property of the heap, so we need to perform **down_heapify()** operation in order to maintain heap structure. The down_heapify() operation does the heapify in the top-bottom approach.

The standard deletion on Heap is to delete the element present at the root node of the heap.

Initially the heap **is**(It follows max-heap property)

Element to be deleted is 12

Step 1: Replace the last element with root, and delete it.

Step 2: Heapify root.
Final Heap:

4. Find-max (or Find-min)

The maximum element and the minimum element in the max-heap and min-heap is found at the root node of the heap.

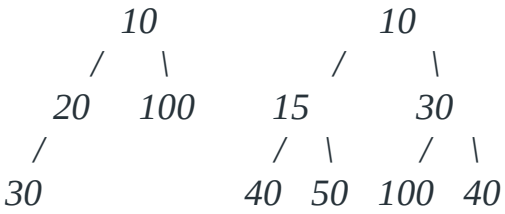
5. Extract Min-Max

This operation returns and deletes the maximum or minimum element in max-heap and min-heap respectively. The maximum element is found at the root node.

Binary Heap:-

- *A **Binary Heap** is a **complete Binary Tree** which is used to store data efficiently to get the max or min element based on its structure.*
- A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at the root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

Examples of Min Heap:

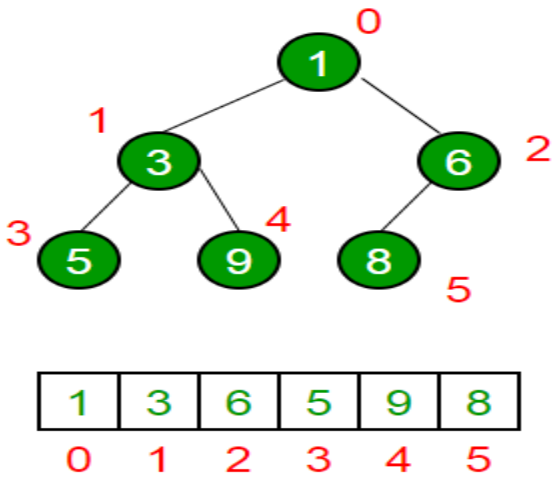


❖ How is Binary Heap represented?

A Binary Heap is a **Complete Binary Tree**. A binary heap is typically represented as an array.

- The root element will be at Arr[0].
- The below table shows indices of other nodes for the i^{th} node, i.e., Arr[i]:

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node



❖ Operations on Heap:

Below are some standard operations on min heap:

- **getMin():** It returns the root element of Min Heap. The time Complexity of this operation is **O(1)**. In case of a maxheap it would be **getMax()**.
- **extractMin():** Removes the minimum element from MinHeap. The time Complexity of this Operation is **O(log N)** as this operation needs to maintain the heap property (by calling **heapify()**) after removing the root.
- **decreaseKey():** Decreases the value of the key. The time complexity of this operation is **O(log N)**. If the decreased key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- **insert():** Inserting a new key takes **O(log N)** time. We add a new key at the end of the tree. If the new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- **delete():** Deleting a key also takes **O(log N)** time. We replace the key to be deleted with the minimum infinite by calling **decreaseKey()**. After decreaseKey(), the minus infinite value must reach root, so we call **extractMin()** to remove the key.

❖ Applications of Heaps:

- Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
- Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log N) time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
- Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

❖ Heap Sort :-

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

Heap Sort Algorithm

To solve the problem follow the below idea:

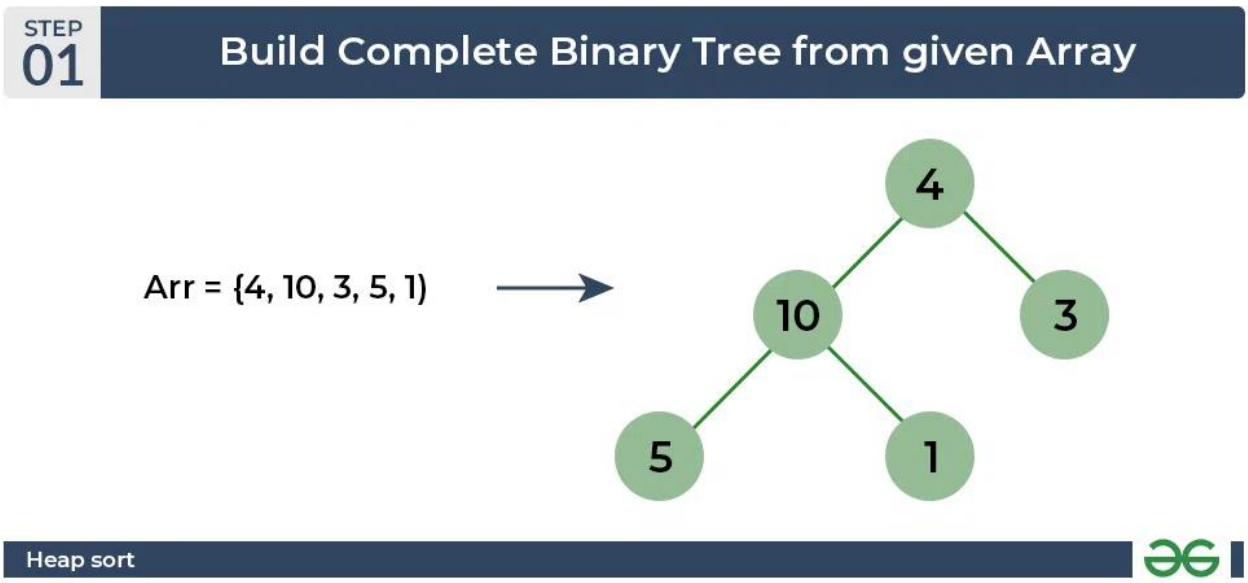
First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:
    - Swap the root element of the heap (which is the largest element) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position).
    - Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

Working of Heap Sort

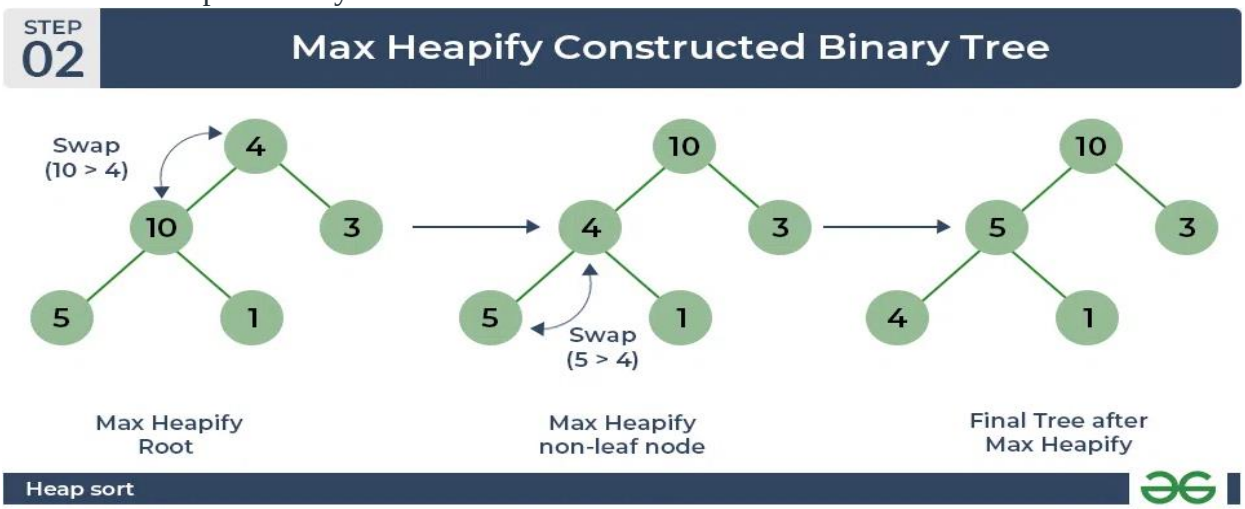
To understand heap sort more clearly, let's take an unsorted array and try to sort it using heap sort.
Consider the array: arr[] = {4, 10, 3, 5, 1}.

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



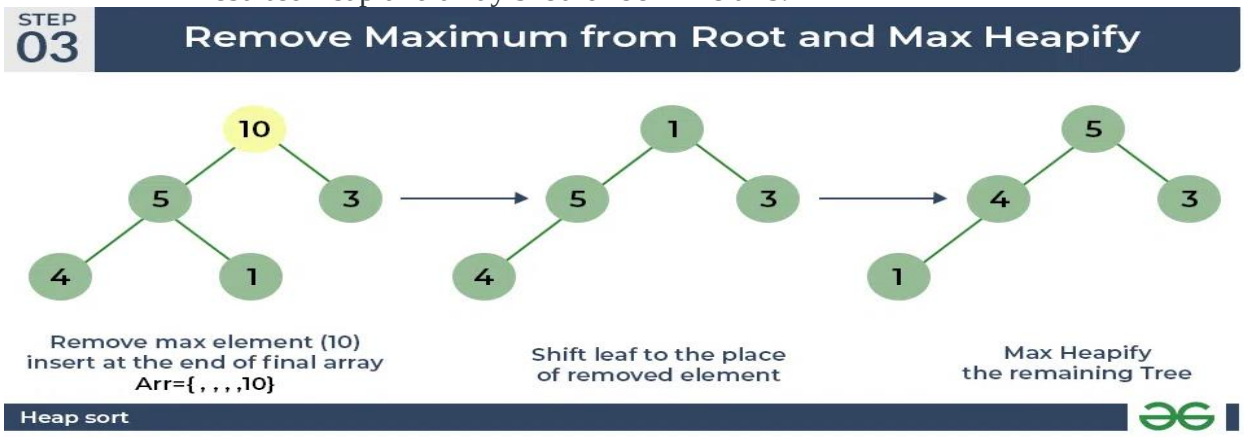
Transform into max heap: After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.

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

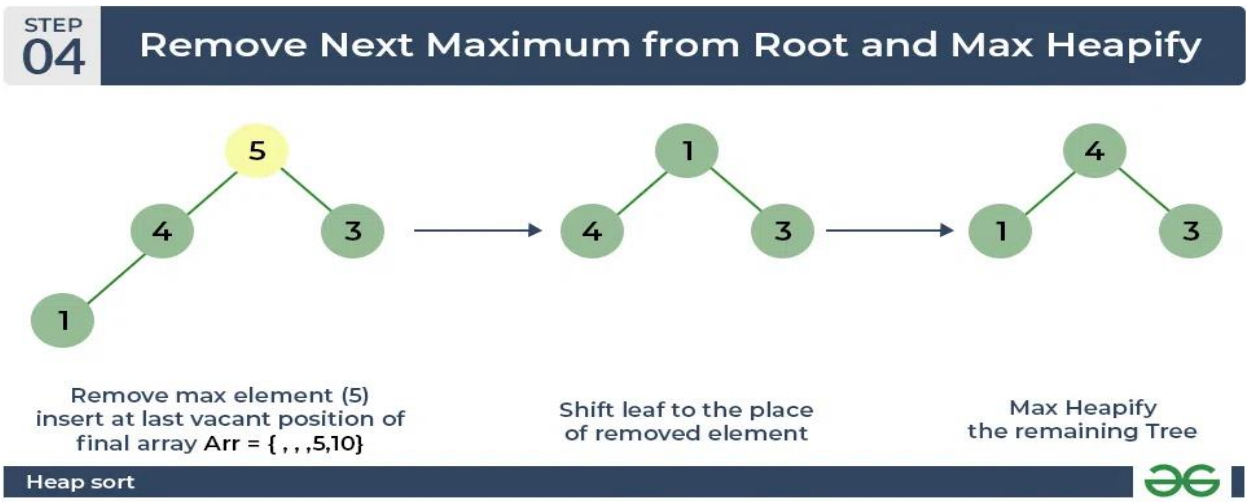


Perform heap sort: Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.

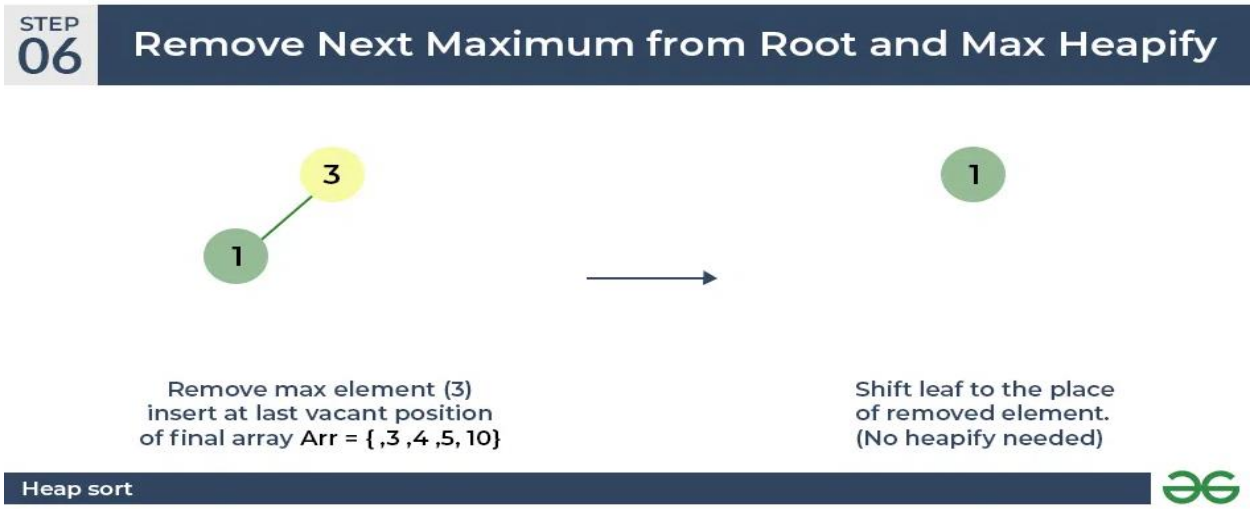
- Delete the root element **(10)** from the max heap. In order to delete this node, try to swap it with the last node, i.e. **(1).** After removing the root element, again heapify it to convert it into max heap.
    - Resulted heap and array should look like this:



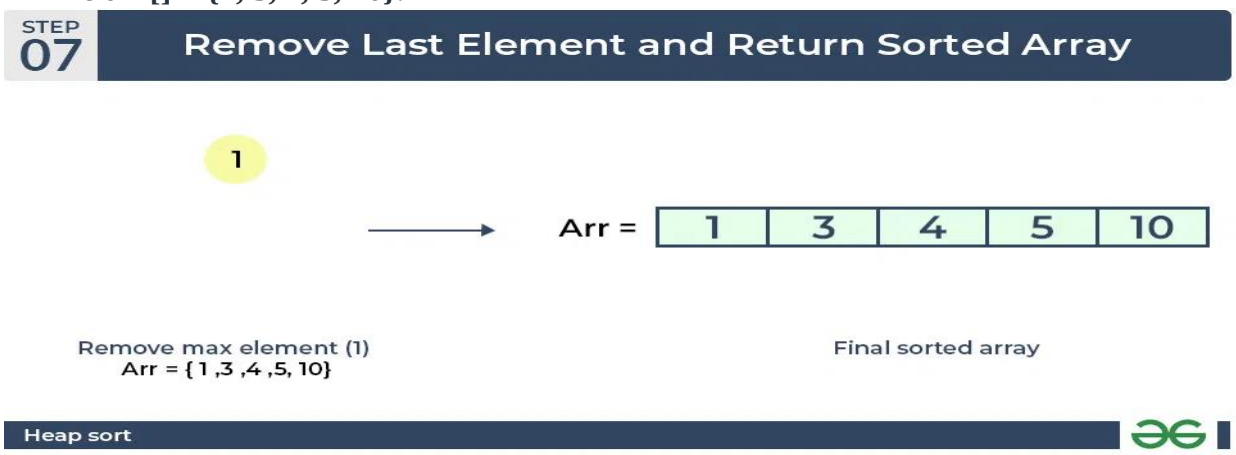
- Repeat the above steps and it will look like the following:



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



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



Heap sort algorithm | Final sorted array

Complexity Analysis of Heap Sort

Time Complexity: O(N log N)
Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.

❖ Important points about Heap Sort:

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

❖ Advantages of Heap Sort:

- **Efficient Time Complexity:** Heap Sort has a time complexity of O(n log n) in all cases. This makes it efficient for sorting large datasets. The **log n** factor comes from the height of the binary heap, and it ensures that the algorithm maintains good performance even with a large number of elements.
- **Memory Usage –** Memory usage can be minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity –** It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

❖ Disadvantages of Heap Sort:

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

Binomial Heaps in Data Structure:-

What is a Binomial Heap?

A binomial heap can be defined as the collection of binomial trees that satisfies the heap properties, i.e., min-heap. The min-heap is a heap in which each node has a value lesser than the value of its child nodes. Mainly, Binomial heap is used to implement a priority queue. It is an extension of binary heap that gives faster merge or union operations along with other operations provided by binary heap.

Properties of Binomial heap

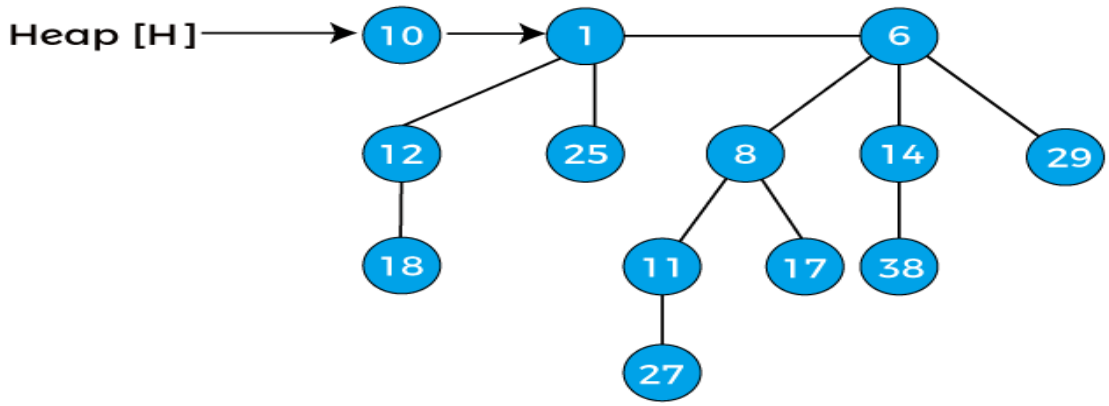
There are following properties for a binomial heap with **n** nodes -

- Every binomial tree in the heap must follow the **min-heap** property, i.e., the key of a node is greater than or equal to the key of its parent.

- For any non-negative integer k, there should be atleast one binomial tree in a heap where root has degree k.

The first property of the heap ensures that the min-heap property is hold throughout the heap. Whereas the second property listed above ensures that a binary tree with **n** nodes should have at most **1 + $\log_2$ n** binomial trees, here **$\log_2$** is the binary logarithm.

We can understand the properties listed above with the help of an example -



The above figure has three binomial trees, i.e., B_0, B_2, and B_3. The above all three binomial trees satisfy the min heap's property as all the nodes have a smaller value than the child nodes.

The above figure also satisfies the second property of the binomial heap. For example, if we consider the value of k as 3, we can observe in the above figure that the binomial tree of degree 3 exists in a heap.

Binomial Heap and the binary representation of a number

A binomial heap with **n** nodes consists the binomial trees equal to the number of set bits in the binary representation of n.

Suppose we want to create the binomial heap of 'n' nodes that can be simply defined by the binary number of 'n'. For example: if we want to create the binomial heap of 13 nodes; the binary form of 13 is 1101, so if we start the numbering from the rightmost digit, then we can observe that 1 is available at the 0, 2, and 3 positions; therefore, the binomial heap with 13 nodes will have B_0, B_2, and B_3 binomial trees.

We can use another example to understand it more clearly, suppose we have to create the binomial heap with 9 nodes. The binary representation of 9 is 1001. So, in the binary representation of 9, digit 1 is occurring at 0 and 3 positions, therefore, the binomial heap will contain the binomial trees of 0 and 3 degrees.

Now, let's move towards the operations performed on Binomial heap.

Operations on Binomial Heap

The operations that can be performed on binomial heap are listed as follows -

- Creating a binomial heap
- Finding the minimum key
- Union or merging of two binomial heaps
- Inserting a node
- Extracting minimum key
- Decreasing a key
- Deleting a node

Now, let's discuss the above-listed operations in detail.

➢ Creating a new binomial heap

✓ When we create a new binomial heap, it simply takes O(1) time because creating a heap will create the head of the heap in which no elements are attached.

➢ Finding the minimum key

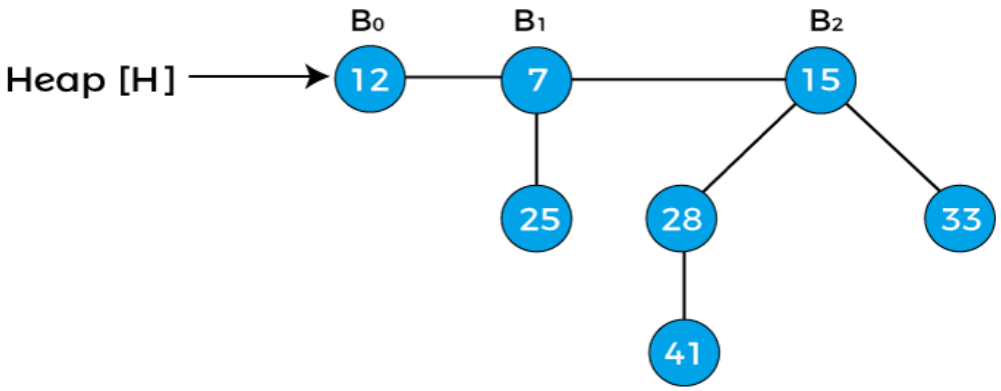
✓ As stated above, binomial heap is the collection of binomial trees, and every binomial tree satisfies the min-heap property. It means that the root node contains a minimum value. Therefore, we only have to compare the root node of all the binomial trees to find the minimum key. The time complexity of finding the minimum key in binomial heap is **O(logn).**

➢ Union or Merging of two binomial heap

✓ It is the most important operation performed on the binomial heap. Merging in a heap can be done by comparing the keys at the roots of two trees, and the root node with the larger key will become the child of the root with a smaller key than the other. The time complexity for finding a union is O(logn).

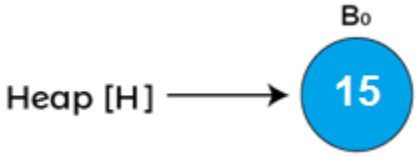
➢ Insert an element in the heap

✓ Inserting an element in the heap can be done by simply creating a new heap only with the element to be inserted, and then merging it with the original heap. Due to the merging, the single insertion in a heap takes O(logn) time. Now, let's understand the process of inserting a new node in a heap using an example.

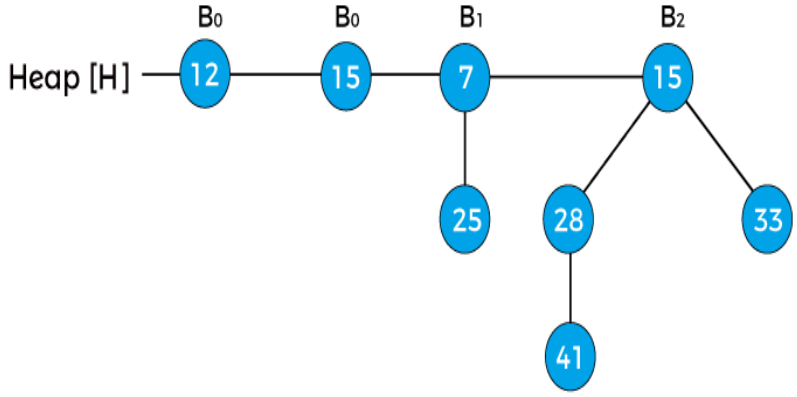


In the above heap, there are three binomial trees of degrees 0, 1, and 2 are given where B0 is attached to the head of the heap.

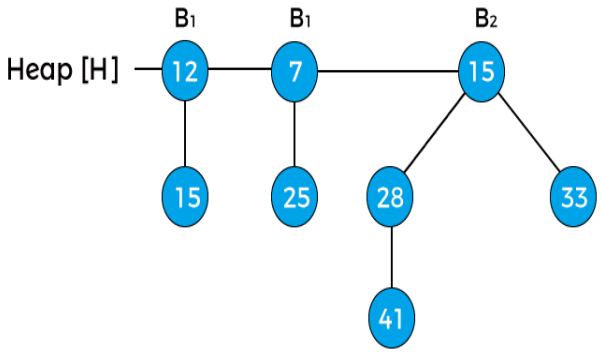
Suppose we have to insert node 15 in the above heap.



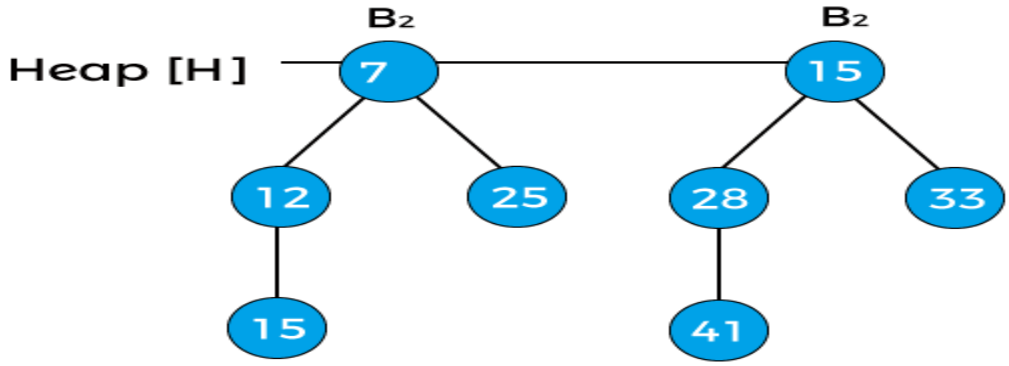
First, we have to combine both of the heaps. As both node 12 and node 15 are of degree 0, so node 15 is attached to node 12 as shown below -



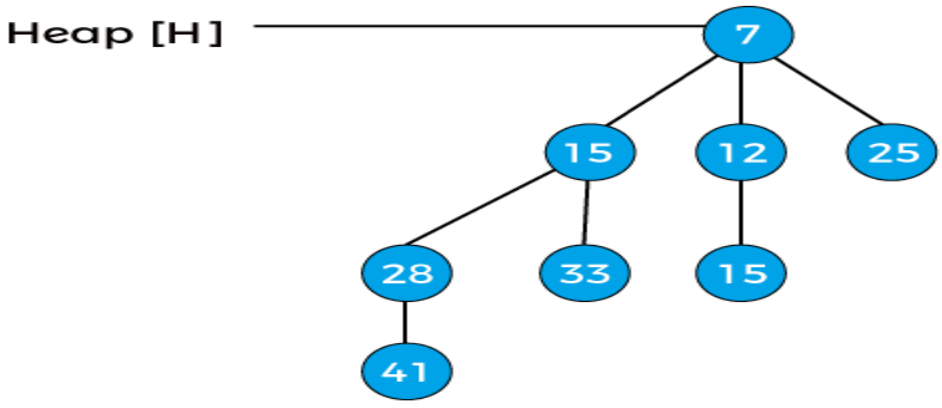
Now, assign x to B_0 with value 12, next(x) to B_0 with value 15, and assign sibling(next(x)) to B_1 with value 7. As the degree of x and next(x) is equal. The key value of x is smaller than the key value of next(x), so next(x) is removed and attached to the x. It is shown in the below image -



Now, x points to node 12 with degree B_1, next(x) to node 7 with degree B_1, and sibling(next(x)) points to node 15 with degree B_2. The degree of x is equal to the degree of next(x) but not equal to the degree of sibling(next(x)). The key value of x is greater than the key value of next(x); therefore, x is removed and attached to the next(x) as shown in the below image -



Now, x points to node 7, and next(x) points to node 15. The degree of both x and next(x) is B_2, and the key value of x is less than the key value of next(x), so next(x) will be removed and attached to x as shown in the below image -

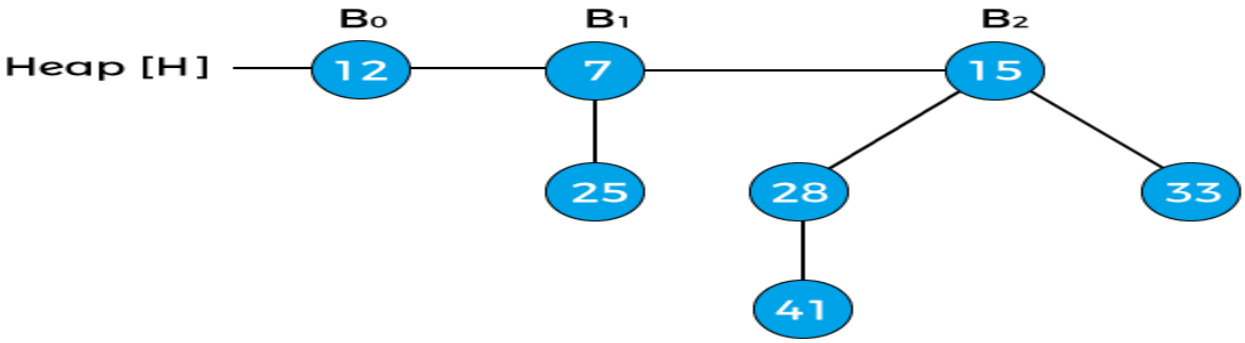


Now, the degree of the above heap is B_3, and it is the final binomial heap after inserting node 15.

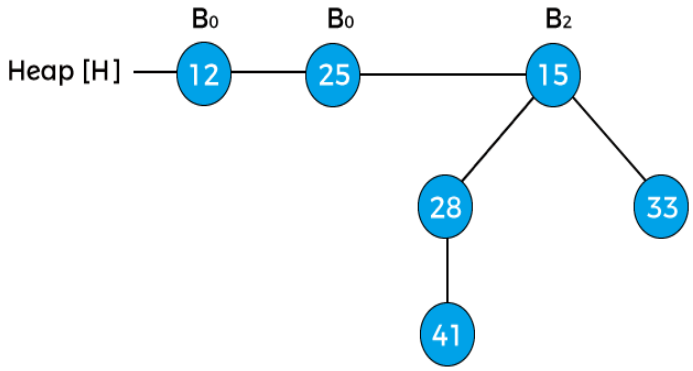
- Extracting the minimum key

- ✓ It means that we have to remove an element with the minimum key value. As we know, in min-heap, the root element contains the minimum key value. So, we have to compare the key value of the root node of all the binomial trees. Let's see an example of extracting the minimum key from the heap.

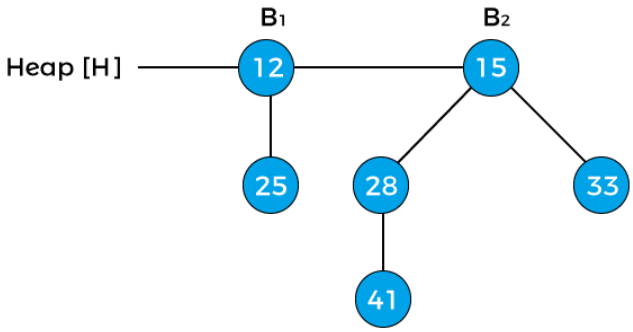
Suppose the heap is -



Now, compare the key values of the root node of the binomial trees in the above heap. So, 12, 7, and 15 are the key values of the root node in the above heap in which 7 is minimum; therefore, remove node 7 from the tree as shown in the below image -



Now, the degree of node 12 and node 25 is B_0, and the degree of node 15 is B_2. Pointer x points to the node 12, next(x) points to the node 25, and sibling(next(x)) points to the node 15. Since the degree of x is equal to the degree of next(x) but not equal to the degree of sibling(next(x)). Value of pointer x is less than the pointer next(x), so node 25 will be removed and attached to node 12 as shown in the below image -

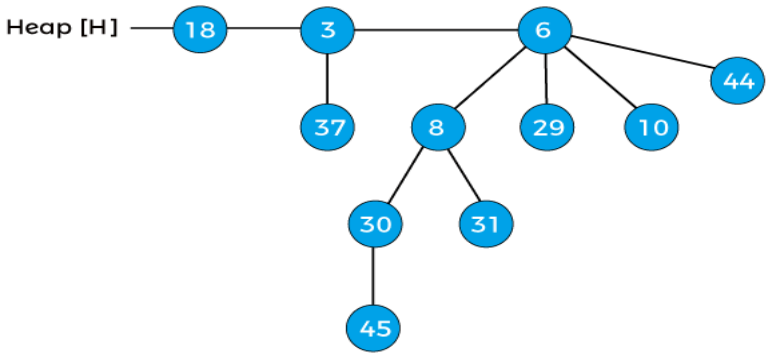


Now, the degree of node 12 is changed to B_1. The above heap is the final heap after extracting the minimum key.

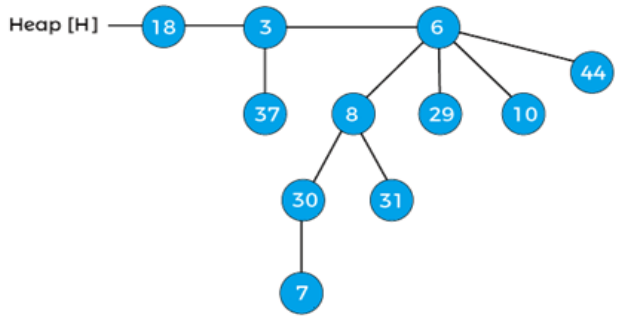
- Decreasing a key

Now, let's move forward to another operation to be performed on binomial heap. Once the value of the key is decreased, it might be smaller than its parent's key that results in the violation of min-heap property. If such case occurs after decreasing the key, then exchange the element with its parent, grandparent, and so on until the min-heap property is satisfied.

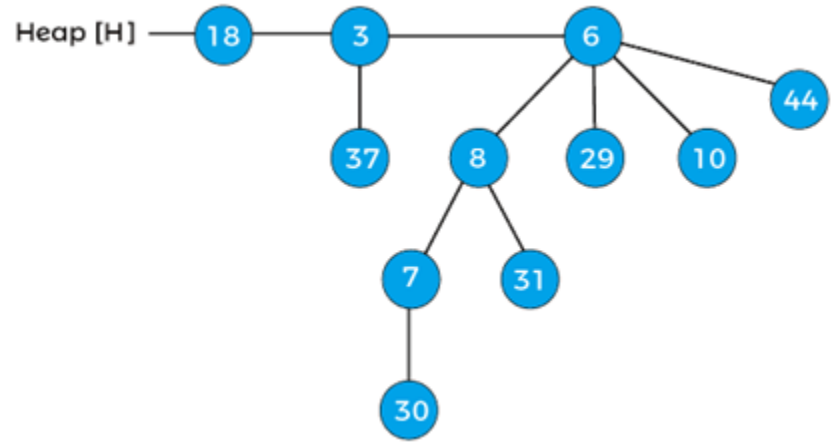
Let's understand the process of decreasing a key in a binomial heap using an example. Consider a heap given below -



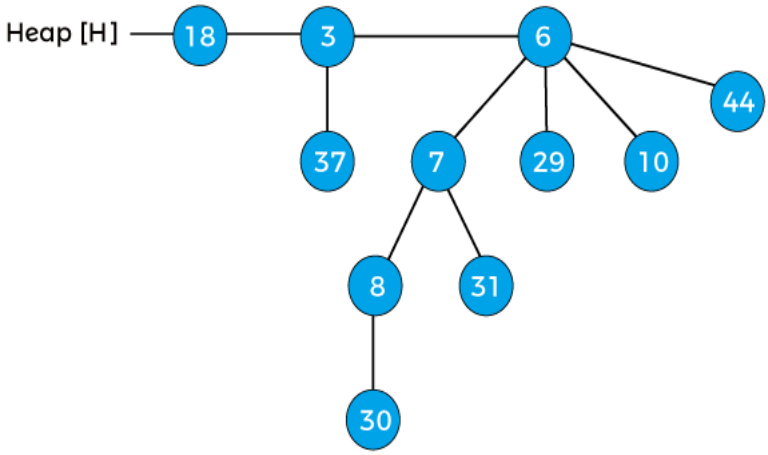
Decrease the key 45 by 7 of the above heap. After decreasing 45 by 7, the heap will be -



After decreasing the key, the min-heap property of the above heap is violated. Now, compare 7 wits its parent 30, as it is lesser than the parent, swap 7 with 30, and after swapping, the heap will be -



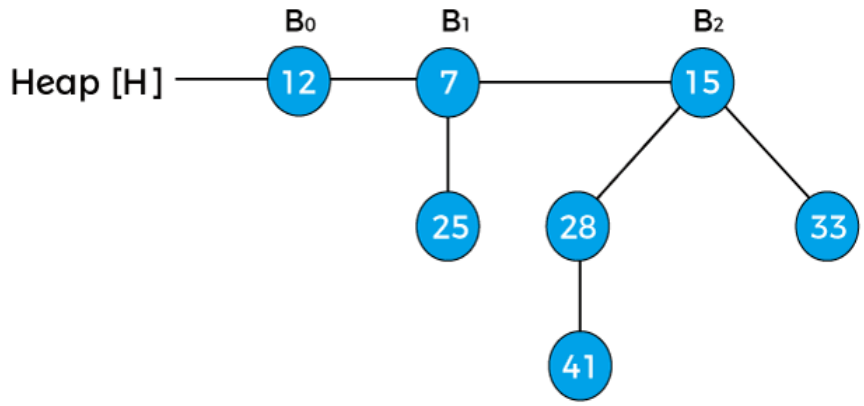
Again compare the element 7 with its parent 8, again it is lesser than the parent, so swap the element 7 with its parent 8, after swapping the heap will be -



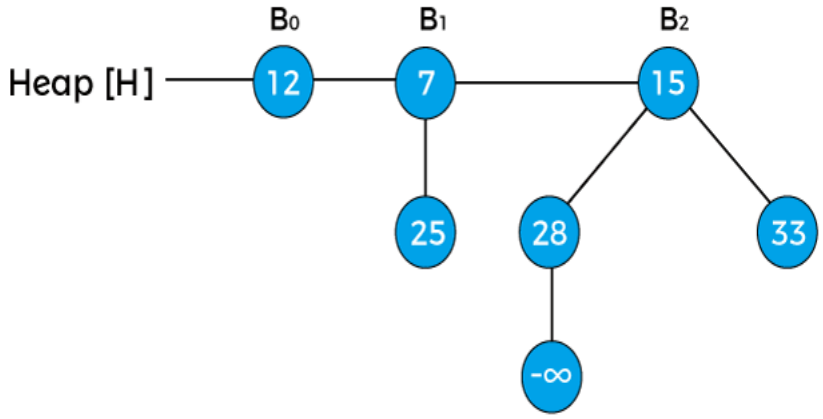
Now, the min-heap property of the above heap is satisfied. So, the above heap is the final heap after decreasing a key.

- Deleting a node from the heap

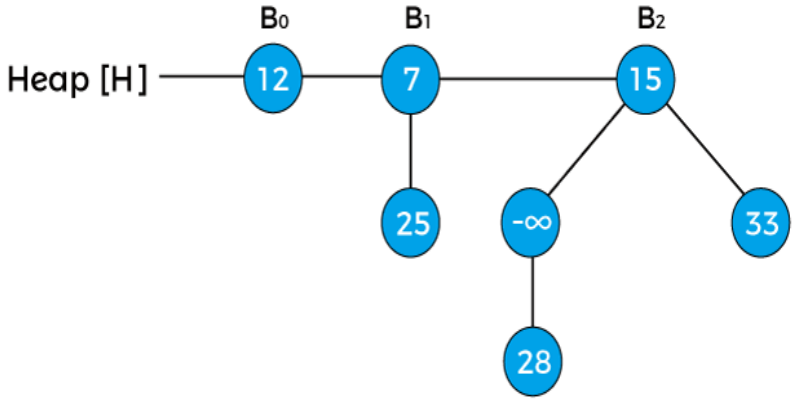
To delete a node from the heap, first, we have to decrease its key to negative infinity (or -∞) and then delete the minimum in the heap. Now we will see how to delete a node with the help of an example. Consider the below heap, and suppose we have to delete the node 41 from the heap -



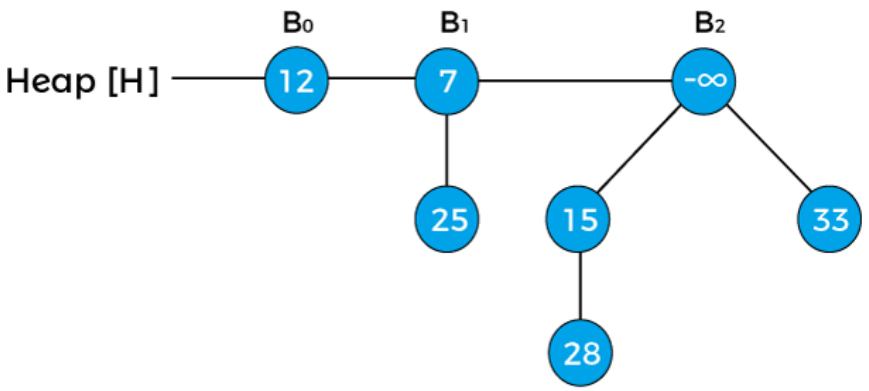
First, replace the node with negative infinity (or -∞) as shown below -



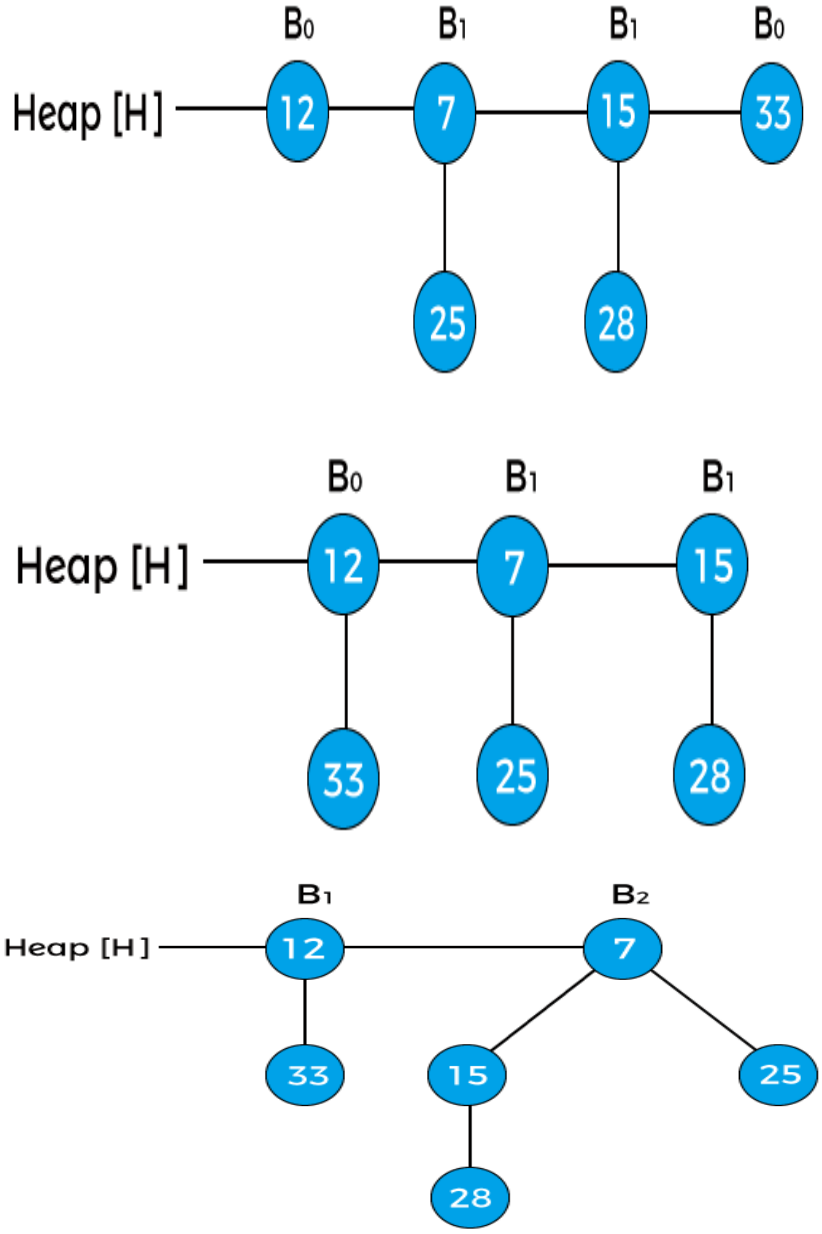
Now, swap the negative infinity with its root node in order to maintain the min-heap property.



Now, again swap the negative infinity with its root node.



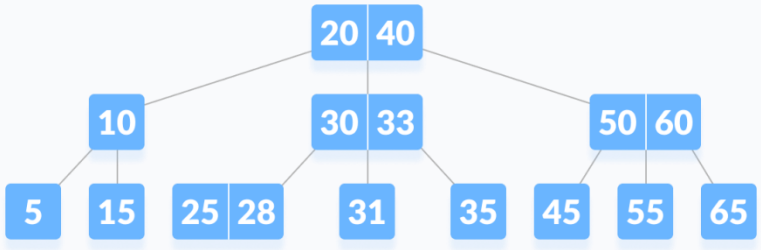
The next step is to extract the minimum key from the heap. Since the minimum key in the above heap is -infinity so we will extract this key, and the heap would be:



The above is the final heap after deleting the node 41.

B-tree

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.
It is also known as a height-balanced m-way tree.



B-tree

Why do you need a B-tree data structure?

The need for B-tree arose with the rise in the need for lesser time in accessing physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk access.

Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large, and the access time increases.

However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

B-tree Properties

1. For each node x, the keys are stored in increasing order.
2. In each node, there is a boolean value $x.leaf$ which is true if x is a leaf.
3. If n is the order of the tree, each internal node can contain at most $n - 1$ keys along with a pointer to each child.
4. Each node except root can have at most n children and at least $n/2$ children.
5. All leaves have the same depth (i.e. height-h of the tree).
6. The root has at least 2 children and contains a minimum of 1 key.

7. If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t (n+1)/2$.

Operations on a B-tree

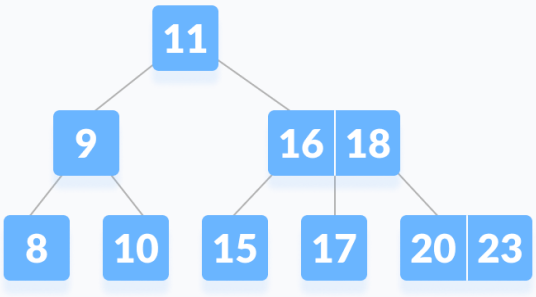
Searching an element in a B-tree

Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

1. Starting from the root node, compare k with the first key of the node.
   If k = the first key of the node, return the node and the index.
2. If k.leaf = true, return NULL (i.e. not found).
3. If k < the first key of the root node, search the left child of this key recursively.
4. If there is more than one key in the current node and k > the first key, compare k with the next key in the node.
   If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).
   Else, search the right child of the key.
5. Repeat steps 1 to 4 until the leaf is reached.

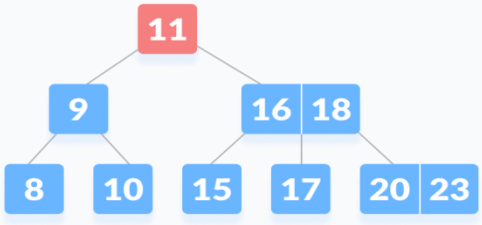
Searching Example

1. Let us search key k = 17 in the tree below of degree 3.



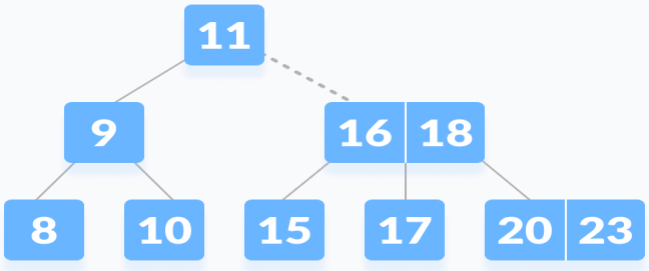
-tree

2. k is not found in the root so, compare it with the root key.



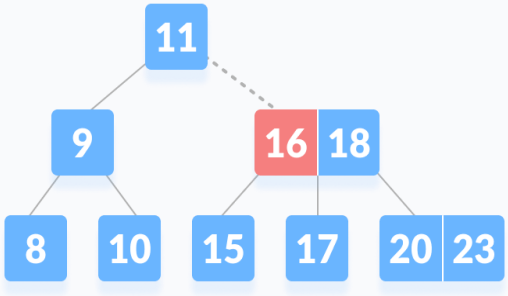
k is not found on the root node

3. Since k > 11, go to the right child of the root node.



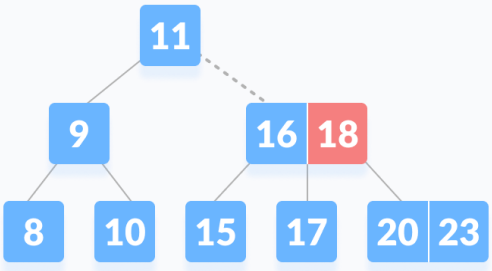
Go to the right subtree

4. Compare k with 16. Since k > 16, compare k with the next key 18.

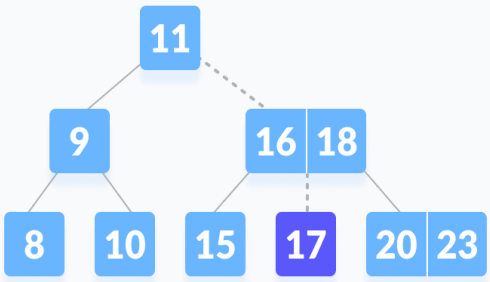


Compare with the keys from left to right

5. Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.



k lies in between 16 and 18



6. k is found.

k is found

Insertion into a B-tree

Inserting an element on a B-tree consists of two events: **searching the appropriate node** to insert the element and **splitting the node** if required.Insertion operation always takes place in the bottom-up approach.
Let us understand these events below.

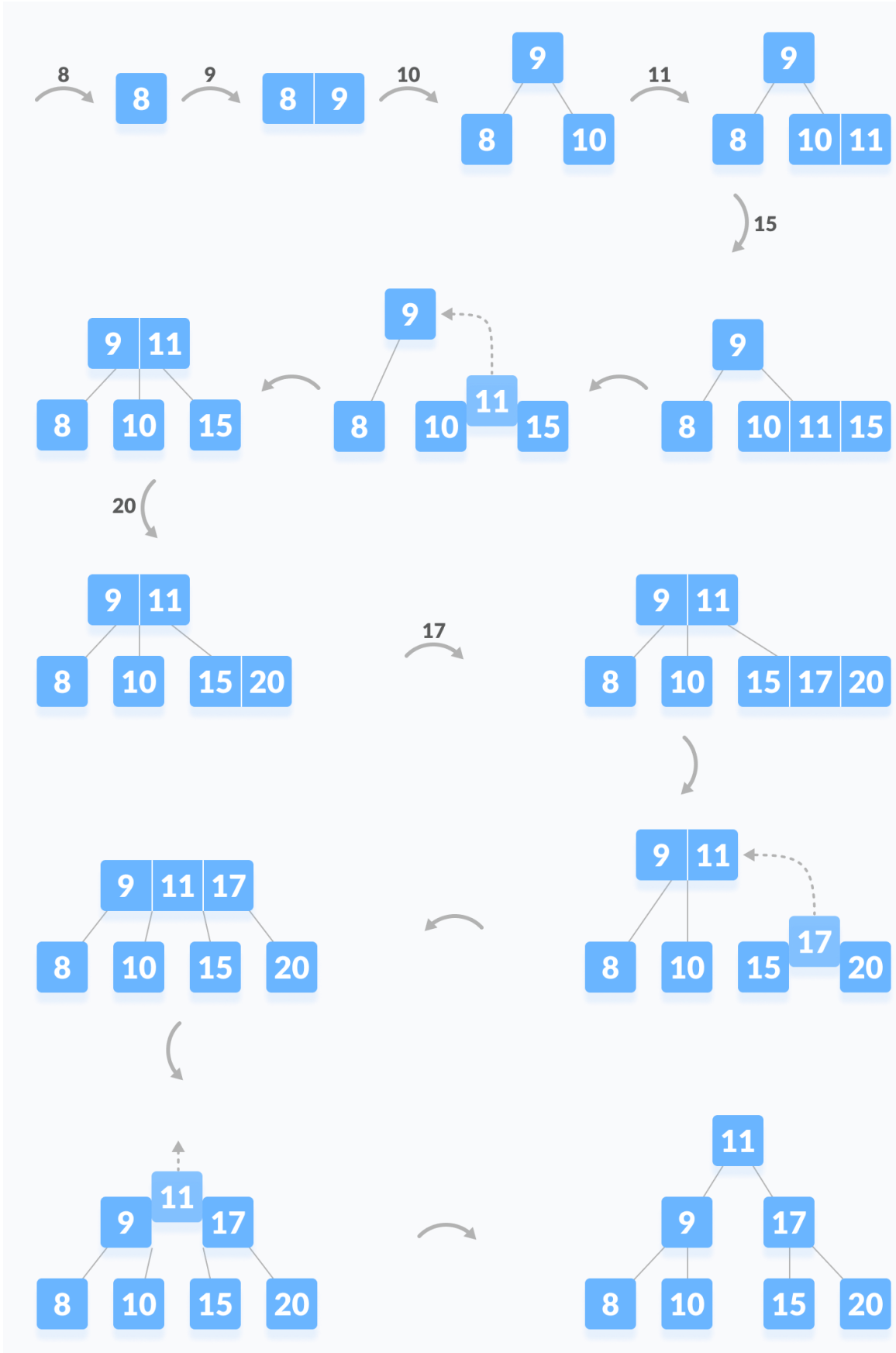
Insertion Operation

1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion.
4. If the node is full, follow the steps below.
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limit. So, split at the median.
7. Push the median key upwards and make the left keys as a left child and the right keys as a right child.
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.

Insertion Example

Let us understand the insertion operation with the illustrations below.

The elements to be inserted are 8, 9, 10, 11, 15, 20, 17.


8
8
9
8 9
10
9
8 10
11
9
8 10 11
15
9 11
8 10 15
9
8 10 11 15
9
8 10 11 15
20
9 11
8 10 15 20
17
9 11
8 10 15 17 20
9 11 17
8 10 15 20
9 11
8 10 15 17 20
11
9 17
8 10 15 20
11
9 17
8 10 15 20

Deletion from a B-tree

Deleting an element on a B-tree consists of three main events: **searching the node where the key to be deleted exists**, deleting the key and balancing the tree if required.
While deleting a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.
The terms to be understood before studying deletion operation are:

1. **Inorder Predecessor**
   The largest key on the left child of a node is called its inorder predecessor.
2. **Inorder Successor**
   The smallest key on the right child of a node is called its inorder successor.

Deletion Operation

Before going through the steps below, one must know these facts about a B tree of degree **m**.

1. A node can have a maximum of m children. (i.e. 3)
2. A node can contain a maximum of $m - 1$ keys. (i.e. 2)
3. A node should have a minimum of $\lceil m/2 \rceil$ children. (i.e. 2)
4. A node (except root node) should contain a minimum of $\lceil m/2 \rceil - 1$ keys. (i.e. 1)

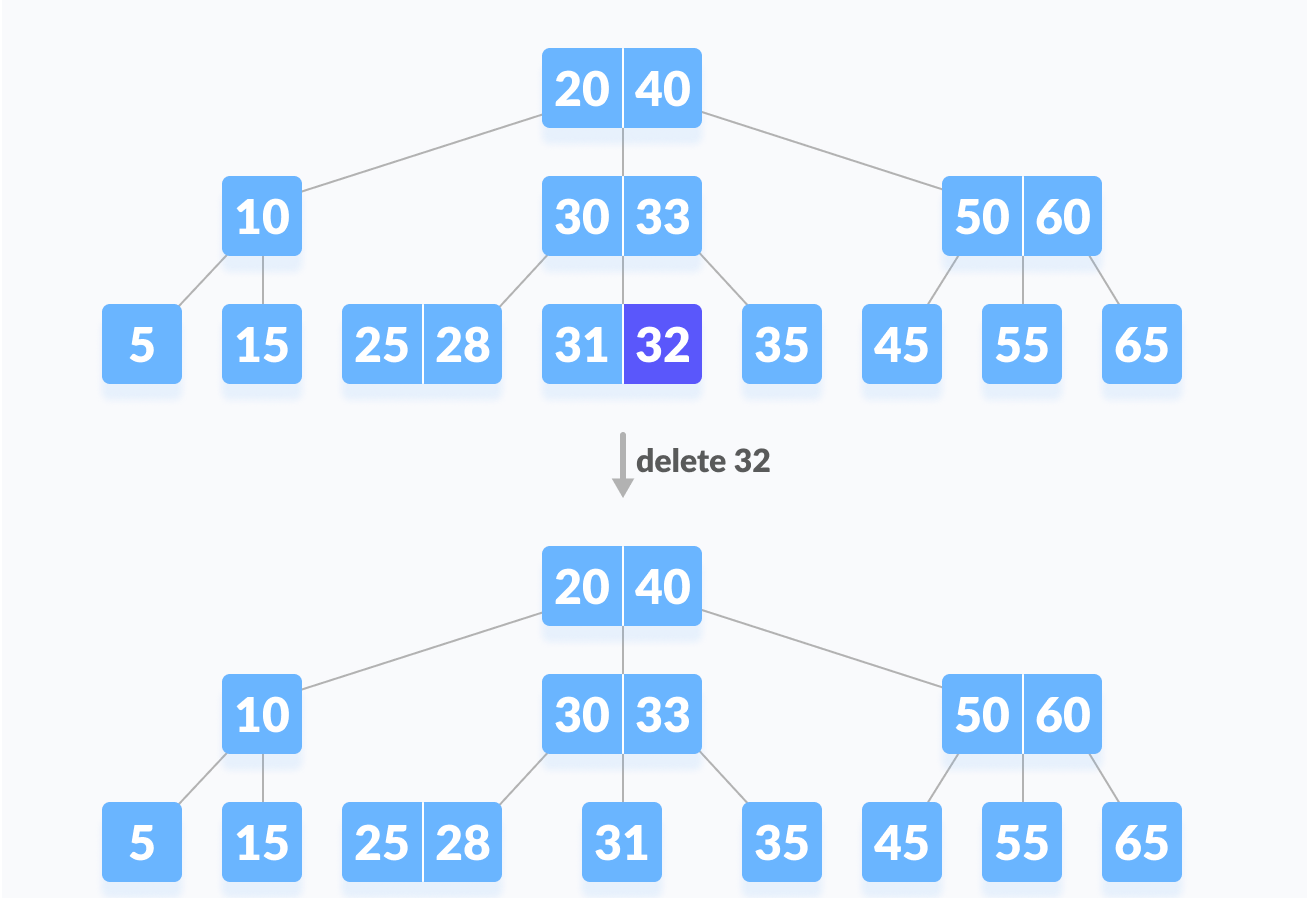
There are three main cases for deletion operation in a B tree.

Case I

The key to be deleted lies in the leaf. There are two cases for it.

1. The deletion of the key does not violate the property of the minimum number of keys a node should hold.

   In the tree below, deleting 32 does not violate the above properties.



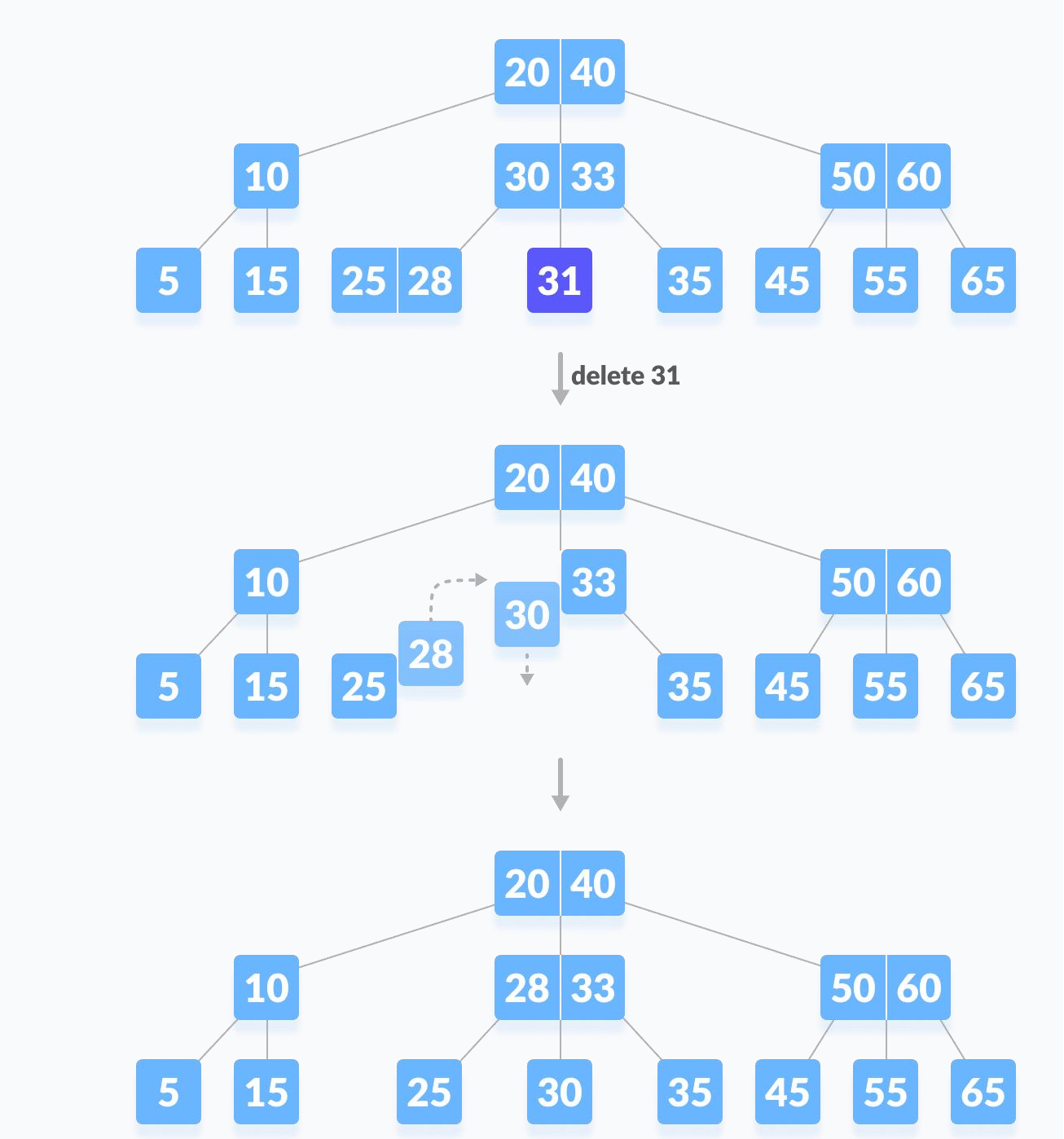
Deleting a leaf key (32) from B-tree

2. The deletion of the key violates the property of the minimum number of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.

   First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node.

   Else, check to borrow from the immediate right sibling node.

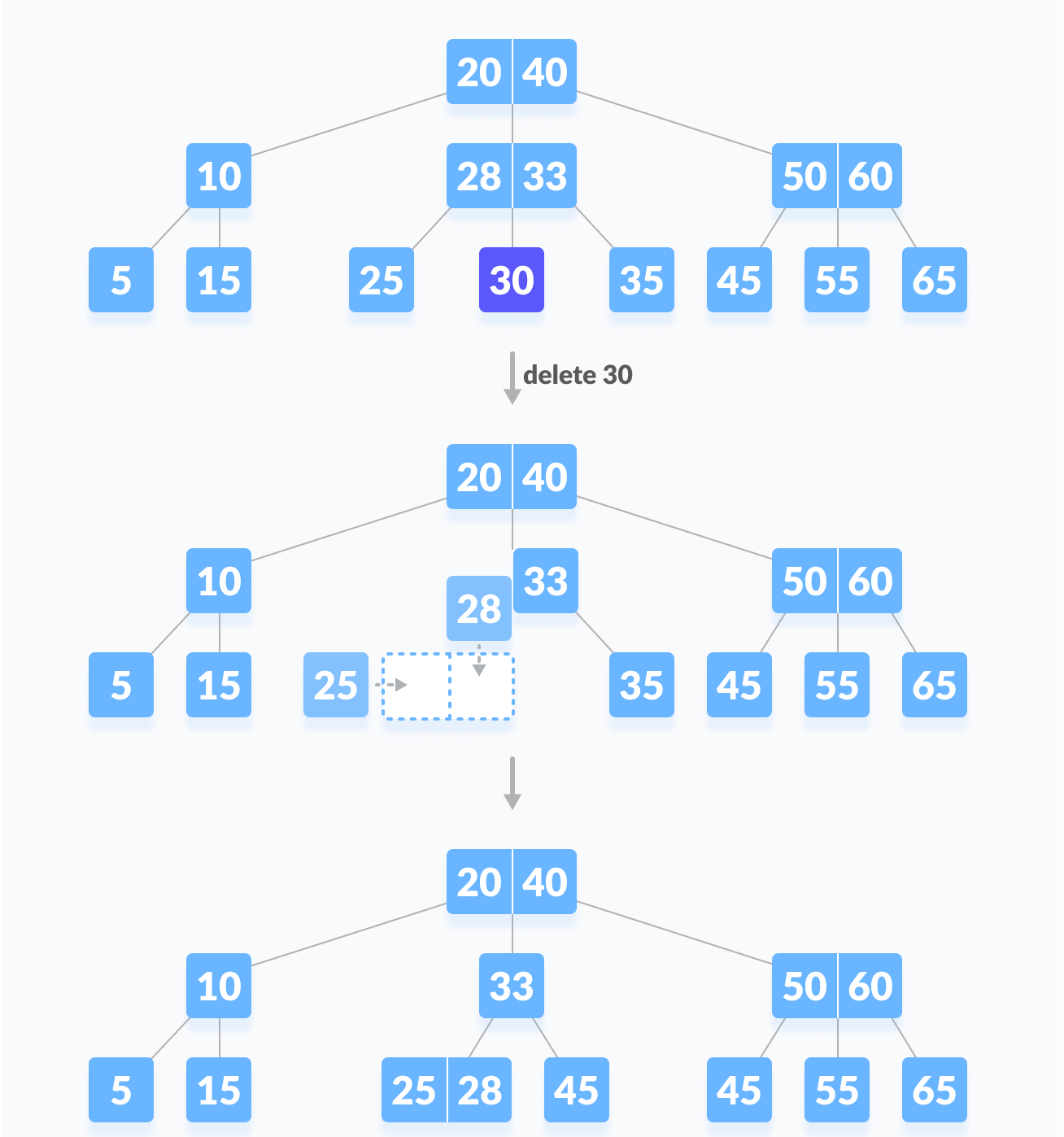
   In the tree below, deleting 31 results in the above condition. Let us borrow a key from the left sibling



node.

Deleting a leaf key (31)If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. **This merging is done through the parent node.**

Deleting 30 results in the above case.

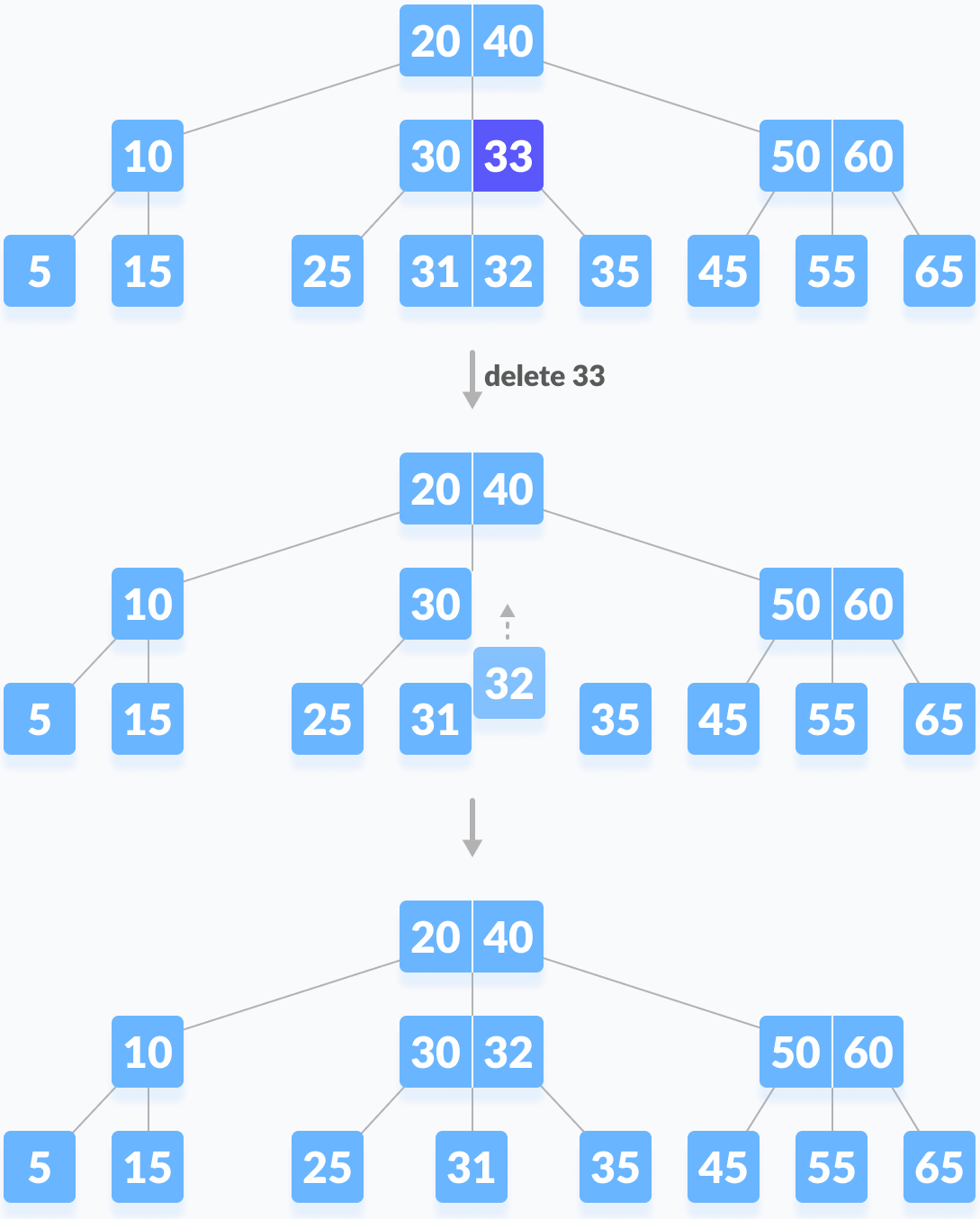


Delete a leaf key (30)

Case II

If the key to be deleted lies in the internal node, the following cases occur.

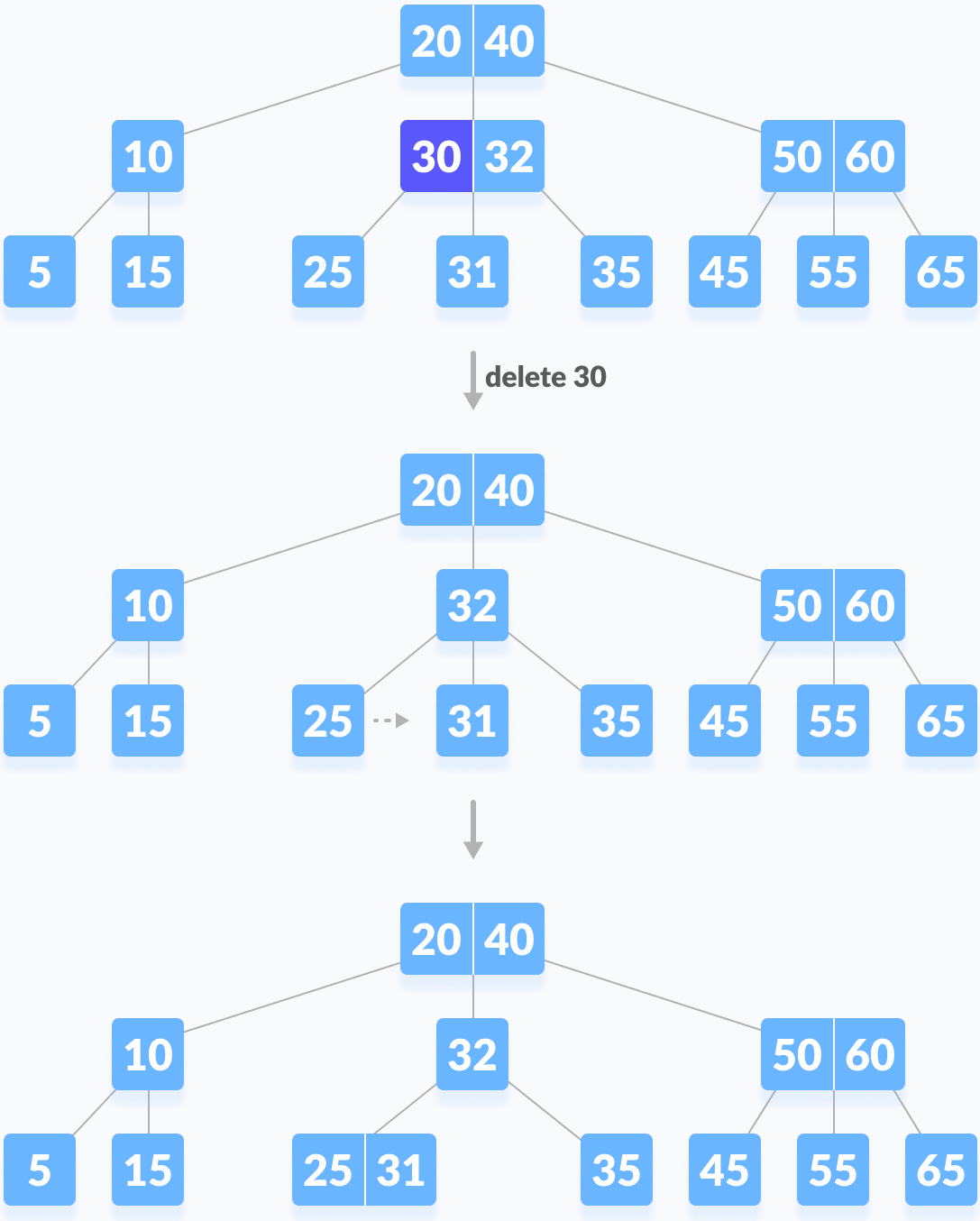
1. The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.



Deleting an internal node (33)

2. The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.

3. If either child has exactly a minimum number of keys then, merge the left and the right children.

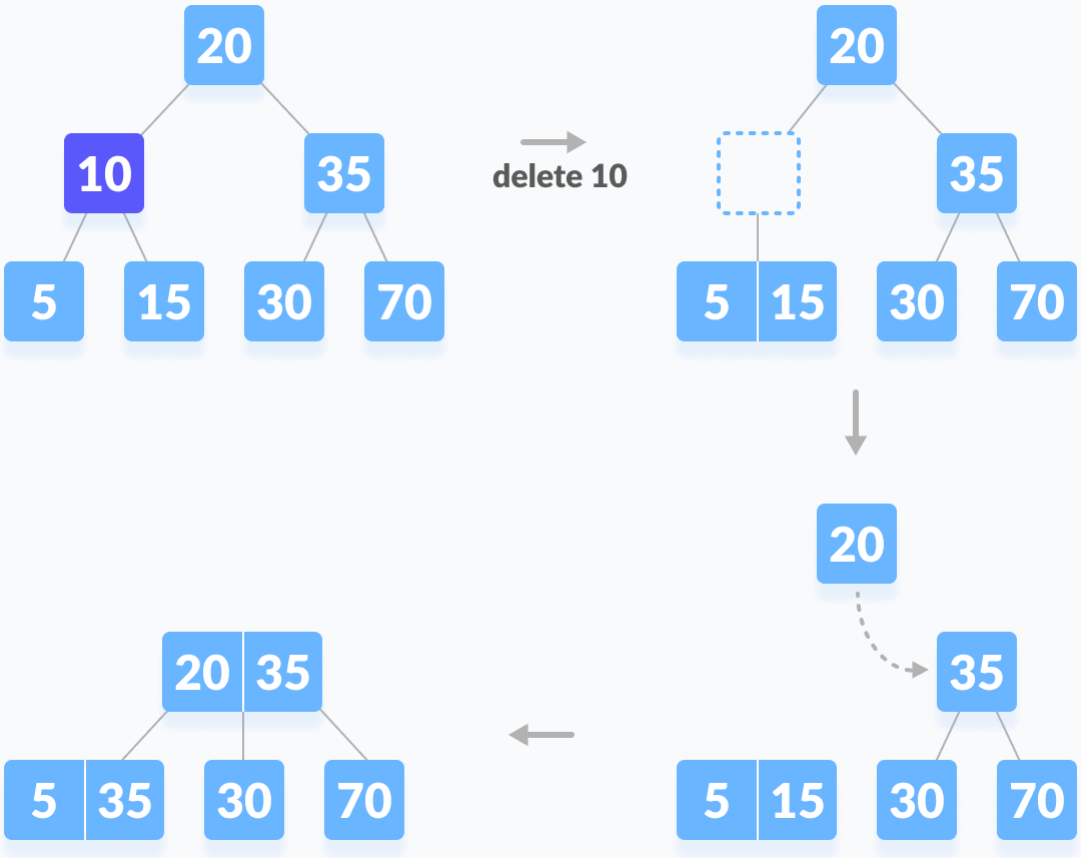


Deleting an internal node (30)After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.

Case III

In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.

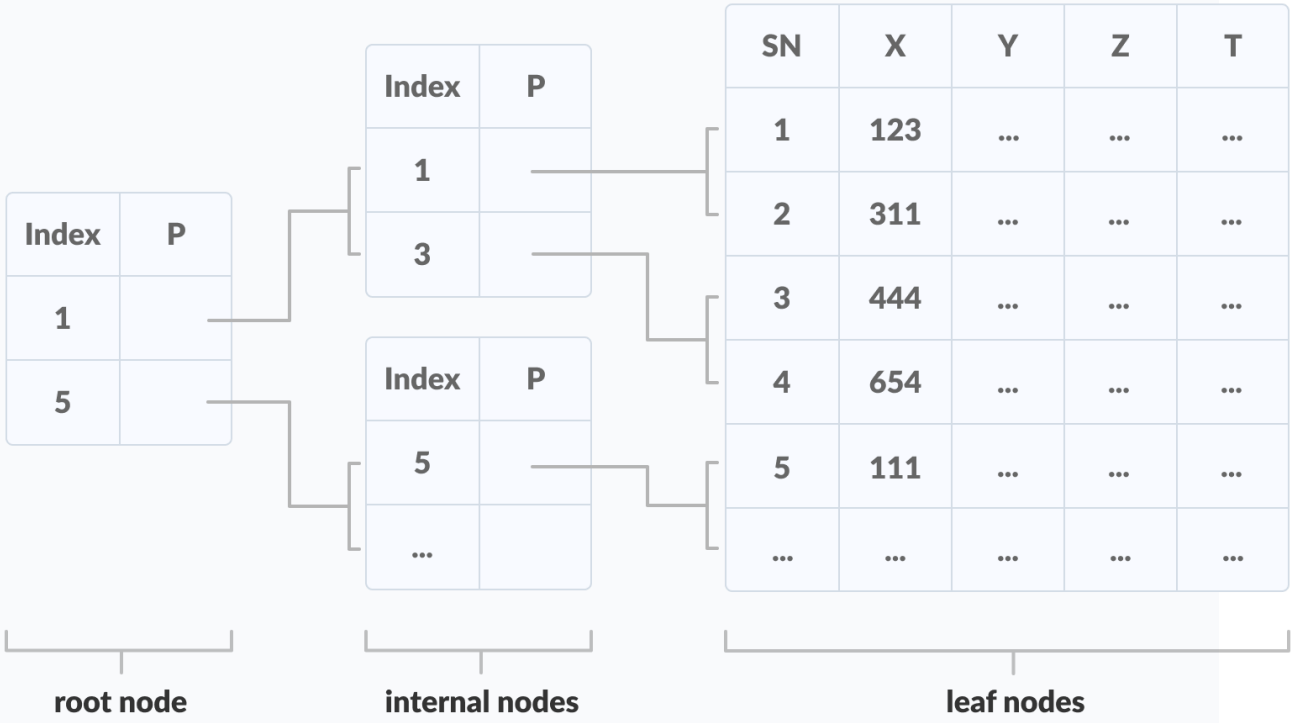
Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).



B+ Tree

A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.

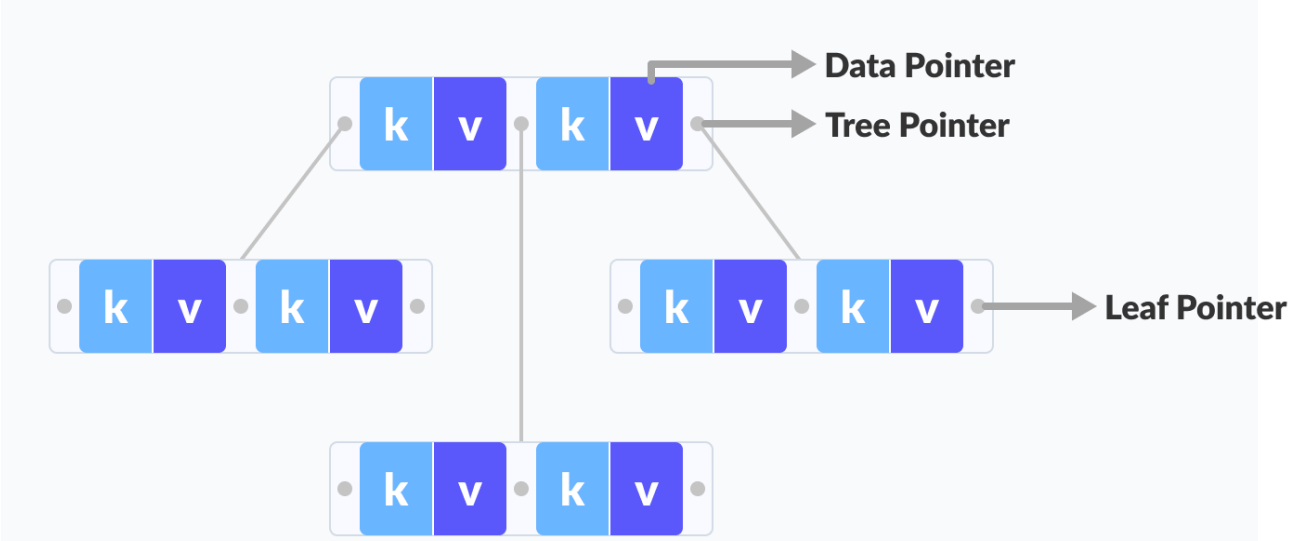
An important concept to be understood before learning B+ tree is multilevel indexing. In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.



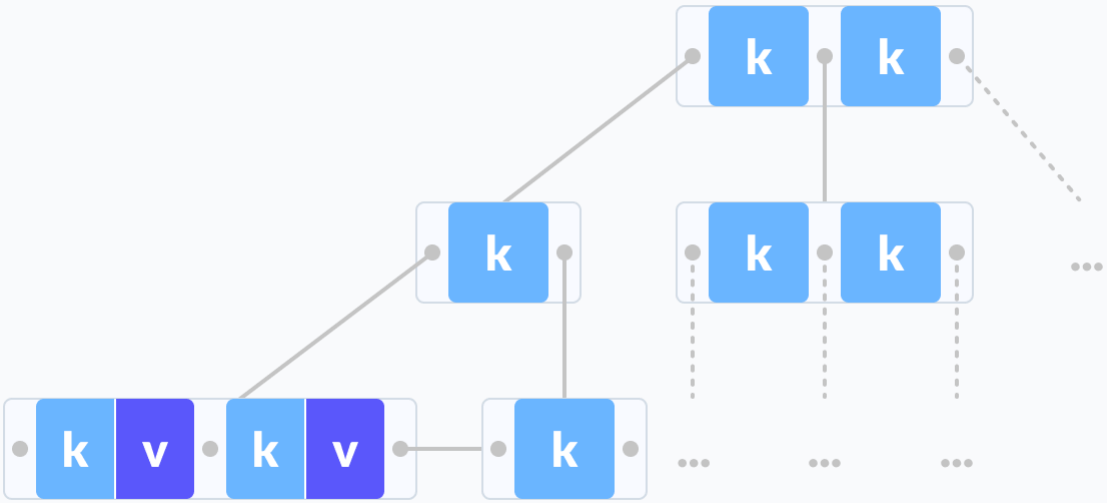
Properties of a B+ Tree

1. All leaves are at the same level.
2. The root has at least two children.
3. Each node except root can have a maximum of m children and at least $m/2$ children.
4. Each node can contain a maximum of $m - 1$ keys and a minimum of $\lceil m/2 \rceil - 1$ keys.

Comparison between a B-tree and a B+ Tree



B-tree



B+ tree

The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.

The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.

Operations on a B+ tree are faster than on a B-tree.

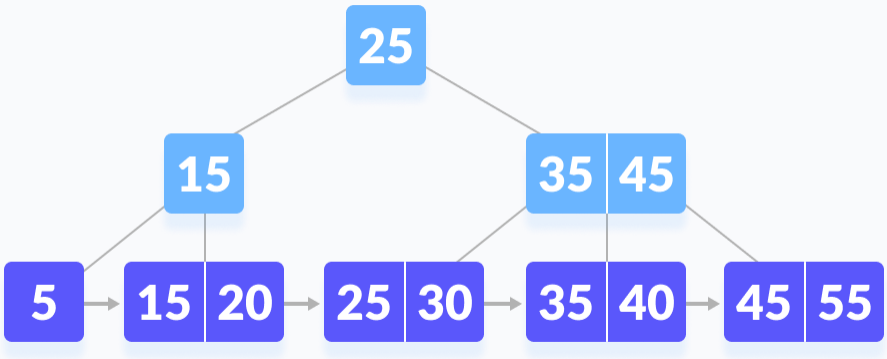
Searching on a B+ Tree

The following steps are followed to search for data in a B+ Tree of order m. Let the data to be searched be k.

1. Start from the root node. Compare k with the keys at the root node $[k1, k2, k3,.....km - 1]$.
2. If $k < k1$, go to the left child of the root node.
3. Else if $k == k1$, compare $k2$. If $k < k2$, k lies between $k1$ and $k2$. So, search in the left child of $k2$.
4. If $k > k2$, go for $k3, k4,...km-1$ as in steps 2 and 3.
5. Repeat the above steps until a leaf node is reached.
6. If k exists in the leaf node, return true else return false.

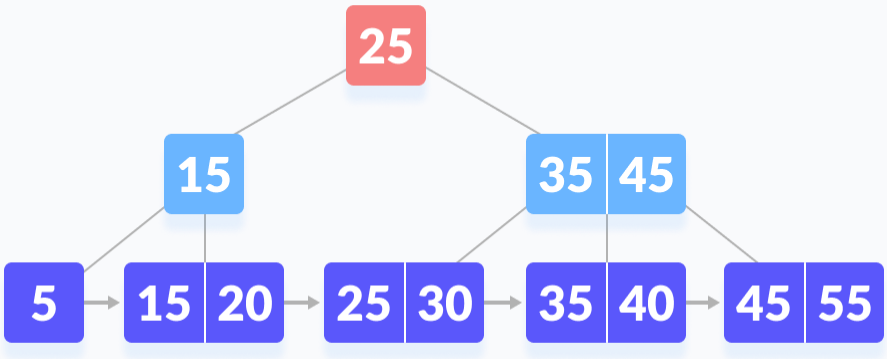
Searching Example on a B+ Tree

Let us search $k = 45$ on the following B+ tree.



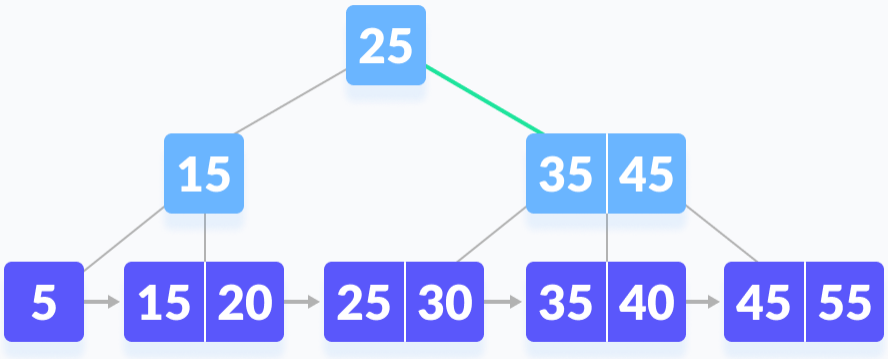
B+ tree

1. Compare k with the root node.



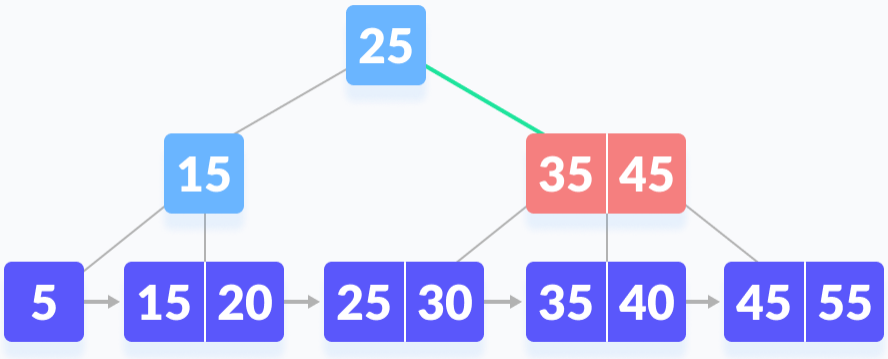
k is not found at the root

2. Since k > 25, go to the right child.



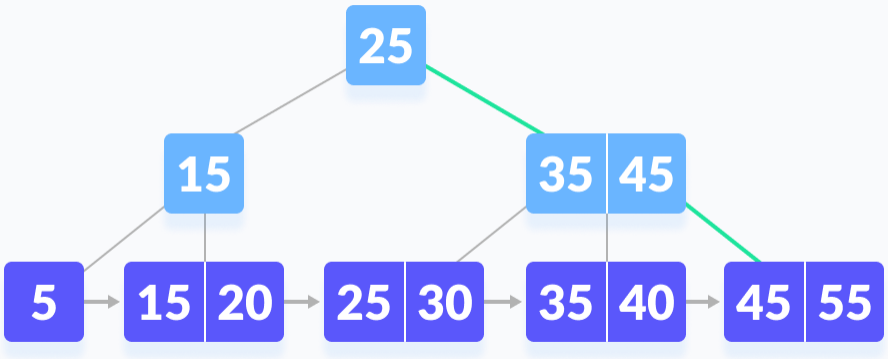
Go to right of the root

3. Compare k with 35. Since k > 30, compare k with 45.

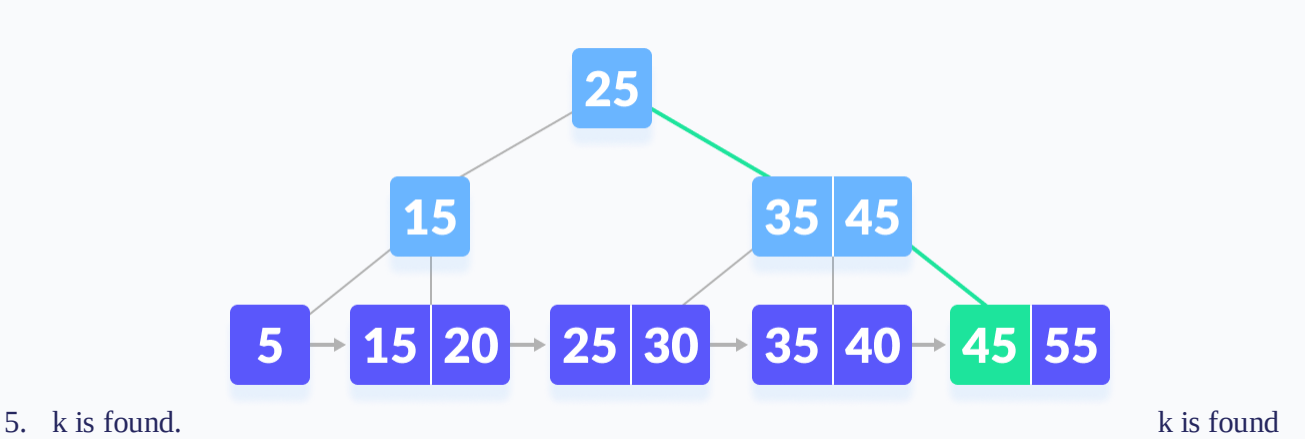


k not found

4. Since k ≥ 45, so go to the right child.



go to the right



5. k is found. k is found