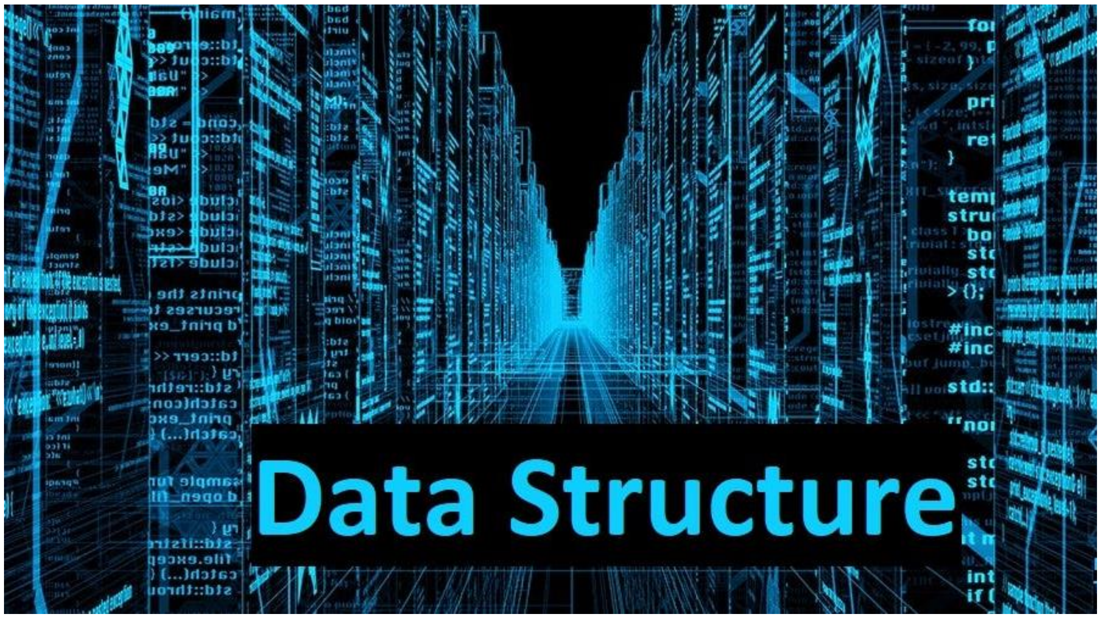
MATHEMATICAL AND COMPUTING SCIENCE DEPARTMENT

CSC 204



LECTURE NOTE

DR. R.O. FOLARANMI

AYEPEKU F.O

Course objectives:

At the end of this course, students will understand:

1. Basic Concepts of Data Structure
2. Implement Array and Linked Lists
3. implement Stacks and Queues
4. Trees and Binary Trees:
5. Graphs:

Table of contents:

CHAPTER ONE

Introduction to Abstract Data Types and Object-Oriented Programming

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data.

For instance, in an object-oriented programming language, the data structure and its associated methods are bound together as part of a class definition. In non-object-oriented languages, there may be functions defined to work with the data structure, but they are not technically part of the data structure.

Why are data structures important?

Typical base data types, such as integers or floating-point values, that are available in most computer programming languages are generally insufficient to capture the logical intent for data processing and use. Yet applications that ingest, manipulate and produce information must understand how data should be organized to simplify processing. Data structures bring together the data elements in a logical way and facilitate the effective use, persistence and sharing of data. They provide a formal model that describes the way the data elements are organized.

Data structures are the building blocks for more sophisticated applications. They are designed by composing data elements into a logical unit representing an abstract data type that has relevance to the algorithm or application. An example of an abstract data type is a "customer name" that is composed of the character strings for "first name," "middle name" and "last name."

It is not only important to use data structures, but it is also important to choose the proper data structure for each task. Choosing an ill-suited data structure could result in slow runtimes or unresponsive code. Five factors to consider when picking a data structure include the following:

1. What kind of information will be stored?
2. How will that information be used?
3. Where should data persist, or be kept, after it is created?
4. What is the best way to organize the data?
5. What aspects of memory and storage reservation management should be considered?

How are data structures used?

In general, data structures are used to implement the physical forms of abstract data types. Data structures are a crucial part of designing efficient software. They also play a critical role in algorithm design and how those algorithms are used within computer programs.

Early programming languages -- such as Fortran, C and C++ -- enabled programmers to define their own data structures. Today, many programming languages include an extensive collection of built-in data structures to organize code and information. For example, Python lists and dictionaries, and JavaScript arrays and objects are common coding structures used for storing and retrieving information.

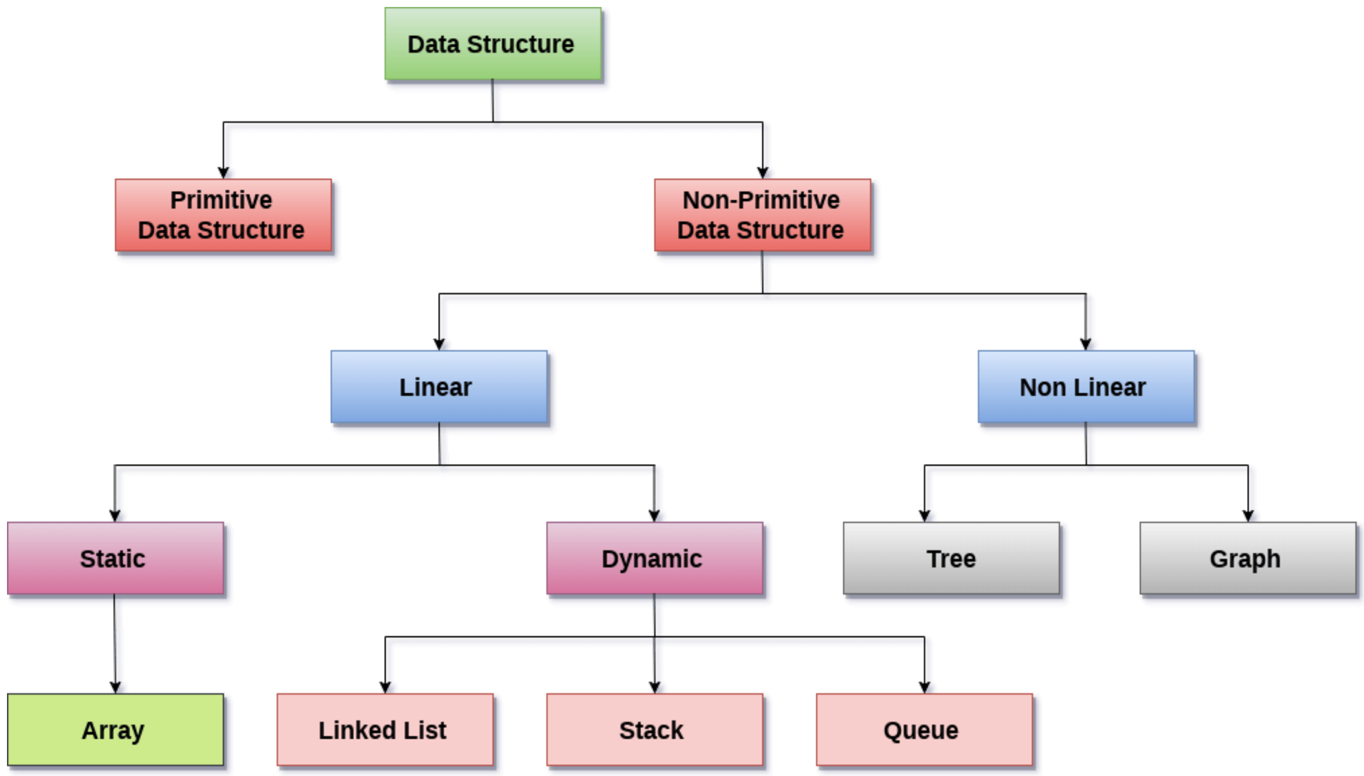
Software engineers use algorithms that are tightly coupled with the data structures -- such as lists, queues and mappings from one set of values to another. This approach can be fused in a variety of applications, including managing collections of records in a relational database and creating an index of those records using a data structure called a binary tree.

Some examples of how data structures are used include the following:

- **Storing data.** Data structures are used for efficient data persistence, such as specifying the collection of attributes and corresponding structures used to store records in a database management system.
- **Managing resources and services.** Core operating system (OS) resources and services are enabled through the use of data structures such as linked lists for memory allocation, file directory management and file structure trees, as well as process scheduling queues.

- **Data exchange.** Data structures define the organization of information shared between applications, such as TCP/IP packets.
- **Ordering and sorting.** Data structures such as binary search trees -- also known as an ordered or sorted binary tree -- provide efficient methods of sorting objects, such as character strings used as tags. With data structures such as priority queues, programmers can manage items organized according to a specific priority.
- **Indexing**. Even more sophisticated data structures such as B-trees are used to index objects, such as those stored in a database.
- **Searching.** Indexes created using binary search trees, B-trees or hash tables speed the ability to find a specific sought-after item.
- **Scalability.** Big data applications use data structures for allocating and managing data storage across distributed storage locations, ensuring scalability and performance. Certain big data programming environments -- such as Apache Spark -- provide data structures that mirror the underlying structure of database records to simplify querying.

Data Structure Classification



Characteristics of data structures

Data structures are often classified by their characteristics. The following three characteristics are examples:

1. **Linear or non-linear.** This characteristic describes whether the data items are arranged in sequential order, such as with an array, or in an unordered sequence, such as with a graph.
2. **Homogeneous or heterogeneous.** This characteristic describes whether all data items in a given repository are of the same type. One example is a collection of elements in an array, or of various types, such as an abstract data type defined as a structure in C or a class specification in Java.
3. **Static or dynamic.** This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time. Dynamic data structures have sizes, structures and memory locations that can shrink or expand, depending on the use.

Data types

If data structures are the building blocks of algorithms and computer programs, the primitive -- or base -- data types are the building blocks of data structures. The typical base data types include the following:

- **Boolean**, which stores logical values that are either true or false.
- **integer**, which stores a range on mathematical integers -- or counting numbers. Different sized integers hold a different range of values -- e.g., a signed 8-bit integer holds values from -128 to 127, and an unsigned long 32-bit integer holds values from 0 to 4,294,967,295.
- **Floating-point numbers**, which store a formulaic representation of real numbers.
- **Fixed-point numbers**, which are used in some programming languages and hold real values but are managed as digits to the left and the right of the decimal point.
- **Character**, which uses symbols from a defined mapping of integer values to symbols.
- **Pointers,** which are reference values that point to other values.

- **String**, which is an array of characters followed by a stop code -- usually a "0" value -- or is managed using a length field that is an integer value.

How to choose a data structure

When choosing a data structure for a program or application, developers should consider the answers to the following three questions:

1. **Supported operations.** What functions and operations does the program need?
2. **Computational complexity.** What level of computational performance is tolerable? For speed, a data structure whose operations execute in time linear to the number of items managed -- using Big O Notation: O(n) -- will be faster than a data structure whose operations execute in time proportional to the square of the number of items managed -- O(n^2).
3. **Programming elegance.** Are the organization of the data structure and its functional interface easy to use?

Some real-world examples include:

- **Linked lists** are best if a program is managing a collection of items that don't need to be ordered, constant time is required for adding or removing an item from the collection and increased search time is OK.
- **Stacks** are best if the program is managing a collection that needs to support a LIFO order.
- **Queues** should be used if the program is managing a collection that needs to support a FIFO order.
- **Binary trees** are good for managing a collection of items with a parent-child relationship, such as a family tree.
- **Binary search trees** are appropriate for managing a sorted collection where the goal is to optimize the time it takes to find specific items in the collection.
- **Graphs** work best if the application will analyze connectivity and relationships among a collection of individuals in a social media network.

Data Structures vs. Abstract Data Types

Data Structures

Data structures refer to the actual implementation of organizing, storing, and manipulating data in a computer's memory. They are concrete and specific, defining how data is arranged and accessed. Examples include arrays, linked lists, stacks, queues, and hash tables. Data structures are crucial for optimizing operations on data, as they determine how quickly and efficiently tasks can be performed.

Abstract Data Types (ADTs)

On the other hand, ADTs are more theoretical and high-level. They define a set of operations that can be performed on data without specifying how those operations are implemented. This allows programmers to think in terms of what a data type can do, rather than how it does it. For instance, a stack ADT involves operations like push, pop, and peek, without specifying whether it's implemented using an array, a linked list, or some other structure.

What is Abstract Data Type?

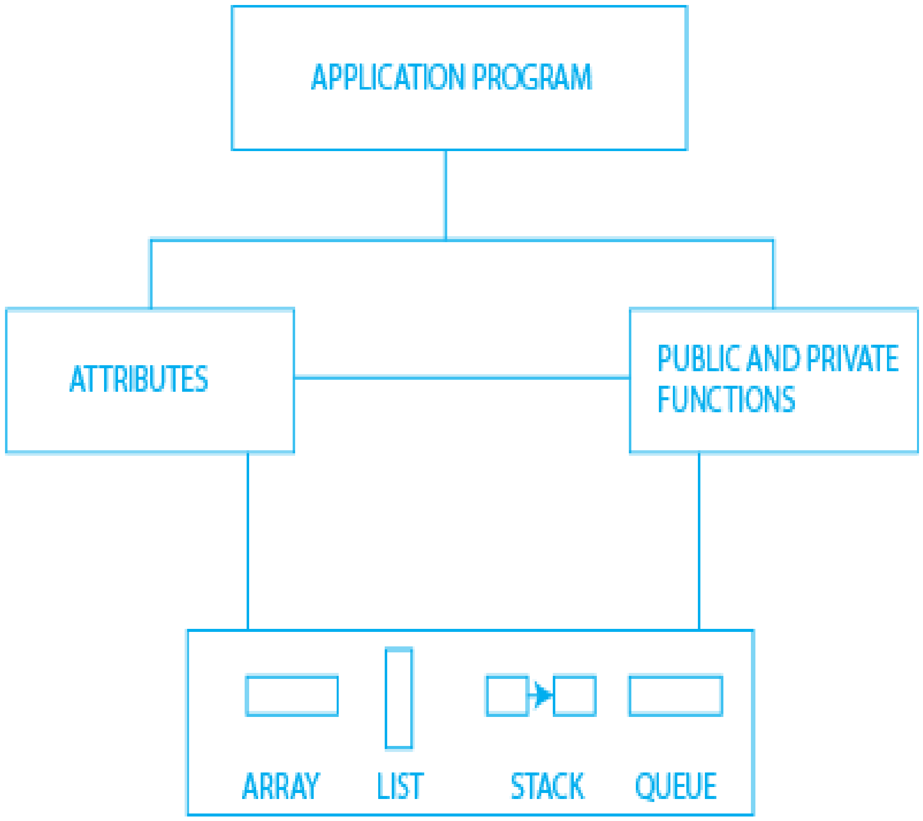
An Abstract Data Type (ADT) is a programming concept that defines a high-level view of a data structure, without specifying the implementation details. In other words, it is a blueprint for creating a data structure that defines the behavior and interface of the structure, without specifying how it is implemented.

An ADT in the data structure can be thought of as a set of operations that can be performed on a set of values. This set of operations actually defines the behavior of the data structure, and they are used to manipulate the data in a way that suits the needs of the program.

ADTs are often used to abstract away the complexity of a data structure and to provide a simple and intuitive interface for accessing and manipulating the data. This makes it easier for programmers to reason about the data structure, and to use it correctly in their programs.

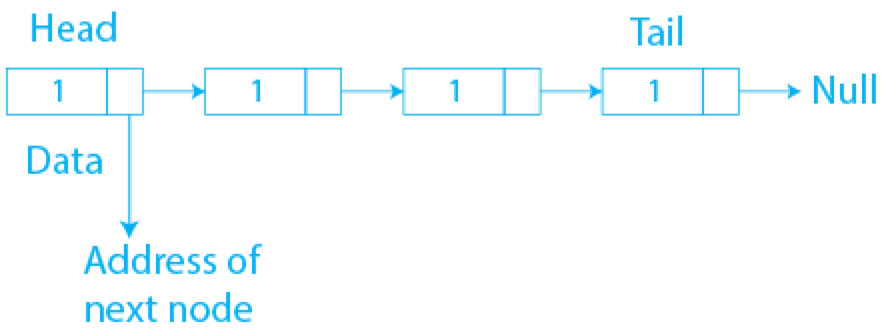
Examples of abstract data type in data structures are List, Stack, Queue, etc.

Abstract Data Type Model



List ADT

Lists are linear data structures that hold data in a non-continuous structure. The list is made up of data storage containers known as "nodes." These nodes are linked to one another, which means that each node contains the address of another block. All of the nodes are thus connected to one another via these links. You can discover more about lists in this article: Linked List Data Structure.

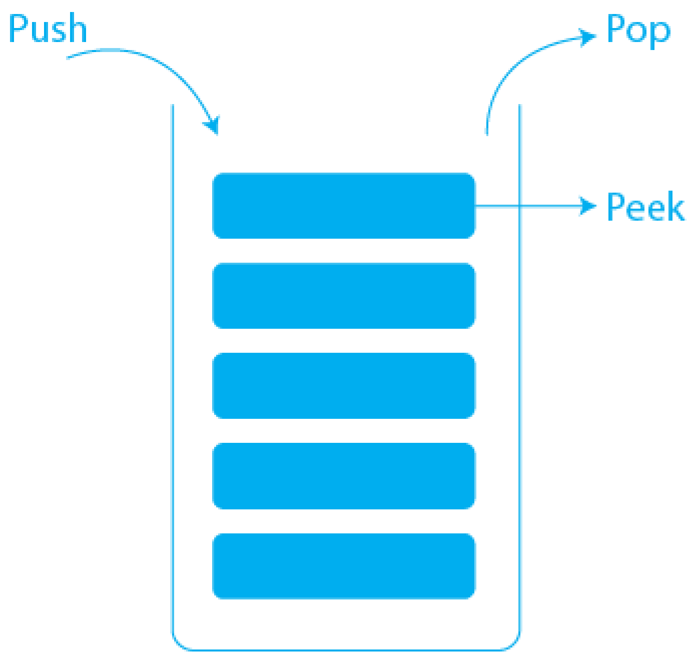


Some of the most essential operations defined in List ADT are listed below.

- **front():** returns the value of the node present at the front of the list.
- **back():** returns the value of the node present at the back of the list.
- **push_front(int val):** creates a pointer with value = val and keeps this pointer to the front of the linked list.
- **push_back(int val):** creates a pointer with value = val and keeps this pointer to the back of the linked list.
- **pop_front():** removes the front node from the list.
- **pop_back():** removes the last node from the list.
- **empty():** returns true if the list is empty, otherwise returns false.
- **size():** returns the number of nodes that are present in the list.

Stack ADT

A stack is a linear data structure that only allows data to be accessed from the top. It simply has two operations: push (to insert data to the top of the stack) and pop (to remove data from the stack). (used to remove data from the stack top).

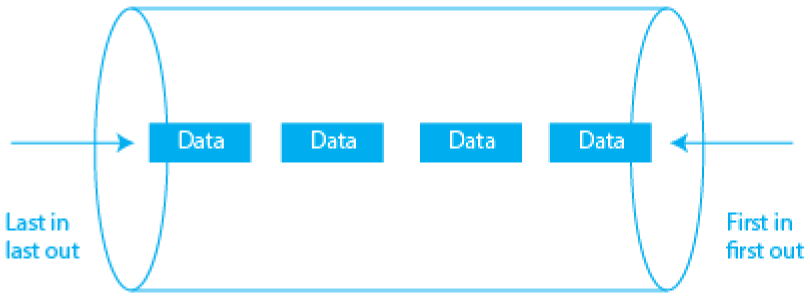


Some of the most essential operations defined in Stack ADT are listed below.

- **top():** returns the value of the node present at the top of the stack.
- **push(int val):** creates a node with value = val and puts it at the stack top.
- **pop():** removes the node from the top of the stack.
- **empty():** returns true if the stack is empty, otherwise returns false.
- **size():** returns the number of nodes that are present in the stack.

Queue ADT

A queue is a linear data structure that allows data to be accessed from both ends. There are two main operations in the queue: push (this operation inserts data to the back of the queue) and pop (this operation is used to remove data from the front of the queue).



Some of the most essential operations defined in Queue ADT are listed below.

- **front():** returns the value of the node present at the front of the queue.
- **back():** returns the value of the node present at the back of the queue.
- **push(int val):** creates a node with value = val and puts it at the front of the queue.
- **pop():** removes the node from the rear of the queue.
- **empty():** returns true if the queue is empty, otherwise returns false.
- **size():** returns the number of nodes that are present in the queue.

Advantages of ADT in Data Structures

The advantages of ADT in Data Structures are:

- Provides abstraction, which simplifies the complexity of the data structure and allows users to focus on the functionality.

- Enhances program modularity by allowing the data structure implementation to be separate from the rest of the program.
- Enables code reusability as the same data structure can be used in multiple programs with the same interface.
- Promotes the concept of data hiding by encapsulating data and operations into a single unit, which enhances security and control over the data.
- Supports polymorphism, which allows the same interface to be used with different underlying data structures, providing flexibility and adaptability to changing requirements.

Disadvantages of ADT in Data Structures

There are some potential disadvantages of ADT in Data Structures:

- **Overhead:** Using ADTs may result in additional overhead due to the need for abstraction and encapsulation.
- **Limited control:** ADTs can limit the level of control that a programmer has over the data structure, which can be a disadvantage in certain scenarios.
- **Performance impact:** Depending on the specific implementation, the performance of an ADT may be lower than that of a custom data structure designed for a specific application.

CHAPTER TWO
Primitive Data Structure

Primitive data types, also known as basic data types or fundamental data types, are the simplest data types provided by a programming language. They are the building blocks for constructing more complex data structures and represent basic values that can be manipulated directly by the computer's hardware. Primitive data types are typically supported directly by the language and are not composed of other data types.

Primitive Data Types: Primitive data types are fundamental data types provided by a programming language to represent basic values. They are directly supported by the language and typically correspond to data types natively supported by the computer's hardware.

Primitive data types are used to store simple values such as integers, floating-point numbers, characters, and boolean values. They are often classified based on the kind of data they represent and the amount of memory they occupy.

Key Features of Primitive Data Structures

Primitive data structures have several key features that make them useful in programming. Here are the details of each key feature:

1. **Size:** Primitive data structures have a fixed size, which means that they take up a predictable amount of memory. For example, an integer in most programming languages takes up four bytes of memory, regardless of the value it represents.
2. **Speed:** Primitive data structures are simple, which means that they can be processed quickly by the computer. Because they have a fixed size and format, the computer can easily perform calculations and operations on them without needing to spend extra time or resources on interpretation or conversion.
3. **Memory Efficiency:** Primitive data structures use a minimal amount of memory, which is important when working with large amounts of data. Because they have a fixed size, primitive data types are more efficient in terms of memory usage than complex data structures.
4. **Portability:** Primitive data structures are typically the same across different programming languages and platforms, which makes code more portable. This means

that code written in one programming language or for one platform can often be easily adapted to work on another platform or with another programming language.

Types of Primitive Data Structures

Let us now look at some of the most often-used primitive data structures.

1. Bool

The bool data type is a primitive data structure that represents a logical value, which can be either true or false. In most programming languages, the bool data type has a fixed size of one byte.

Syntax in C++:

```
bool a = true;
```

Syntax in Java:

```
boolean a = false;
```

Syntax in Python:

```
a = True
```

2. Byte

The byte data type is a primitive data structure that represents an 8-bit signed integer. In Java, the byte data type is represented using the keyword "byte" and has a size of 1 byte. It is useful for storing small integers in memory and is often used in situations where memory usage is critical.

Syntax in Java:

```
byte myByteVariable = 127;
```

In this example, "myByteVariable" is a byte variable that has been assigned the value 127. Note that the range of values that can be stored in a byte is -128 to 127, inclusive.

3. Char

The char data type is a primitive data structure that represents a single character, such as a letter, digit, or symbol. The char data type has a fixed size of two bytes.

Syntax in C++:

```
char myChar = 'A';
```

Syntax in Java:

```
char myChar = 'C';
```

4. Int

The int data type is a primitive data structure that is used to represent integer values. It can store both positive and negative whole numbers, such as 0, 1, -2, 100, etc. The int data type has a fixed size of 4 bytes in memory.

Syntax in C++:

```
int myInt = 15;
```

Syntax in Java:

```
int myInt = 25;
```

Syntax in Python:

```
myInt = 30
```

5. Float

The float data type is a primitive data structure that is used to represent floating-point numbers. It is used to store real numbers that require a decimal point, such as 3.14 or -2.5. The float data type has a fixed size of 4 bytes in memory.

Syntax in C++:

```
float myFloat = 3.14;
```

Syntax in Java:

```
float myFloat = 5.20f;
```

In this example, "myFloat" is a float variable that has been assigned the value 5.20. The "f" at the end of the value indicates that it should be treated as a float rather than a double.

Syntax in Python:

```
myFloat = 7.14
```

6. Double

The double data type is similar to the float data type but can store larger decimal values with more precision. It is a primitive data structure that is used to represent floating-point numbers with double precision. The double data type has a fixed size of 8 bytes in memory.

Syntax in C++:

```
double myDouble = 3.14159;
```

Syntax in Java:

```
double myDouble = 3.14159d;
```

7. Long

The long data type is a primitive data structure that is used to represent integer values that require more memory than an int. It can store larger whole numbers, such as 2147483647 or -2147483648. The long data type has a fixed size of 8 bytes in memory.

Syntax in C++:

```
long myLong = 2147483647;
```

Syntax in Java:

```
long myLong = 2147483647;
```

8. Short

The short data type is a primitive data structure that is used to represent integer values that require less memory than an int. It can store smaller whole numbers, such as 32767 or -32768. The short data type takes only 2 bytes in memory.

Syntax in C++:

```
Short int num = 32767;
```

Syntax in Java:

```
short num = 32767;
```

9. Pointer

A pointer is a primitive data structure that stores the memory address of another variable or data structure. It is a powerful tool for memory management and data manipulation in C and C++. Pointers are not available in Java or Python.

Here is how a pointer is defined and used in C and C++:

In C and C++, a pointer is defined using the "*" symbol before the variable name. Here is an example of how to declare and use a pointer in C++:

```
int myInt = 42;
int* myPointer = &myInt;
```

In this example, "myPointer" is a pointer variable that has been assigned the memory address of "myInt" using the "&" symbol.

Memory Representation of Primitive Data Structure

These primitive data types serve as the basic building blocks for storing and manipulating data in programming languages. They are used extensively in variable declarations, function parameters, and return types to represent different kinds of data. Understanding and using these data types correctly is essential for writing efficient and correct programs.

1. **Integer**:
   - Integers are typically stored using a fixed number of bits, which determines their range and precision.
   - Common integer data types include int, long, short, and byte.
   - The memory representation of integers uses binary encoding, where each bit represents a binary digit (0 or 1).

Example in C++:

```
cpp
int number = 42; // 32 bits (4 bytes)
```

Memory Representation:

```
00000000 00000000 00000000 00101010
```

In this example, the integer 42 is stored in memory using 32 bits, with each bit representing a binary digit.

2. **Floating-Point**:
   - Floating-point numbers are stored using a binary representation that includes a sign bit, a significand (or mantissa), and an exponent.
   - Common floating-point data types include float and double.
   - The memory representation of floating-point numbers follows the IEEE 754 standard, which defines how numbers are encoded in binary format.

Example in Java:

```
java
double value = 3.14159; // 64 bits (8 bytes)
```

Memory Representation:

01000000 00010010 10010000 11110101 11100001 01000100 01000010 00101100

In this example, the floating-point number 3.14159 is stored in memory using 64 bits, following the IEEE 754 standard.

3. **Character**:
   - Characters are typically stored using a fixed number of bits, where each bit represents a character from a character set (e.g., ASCII or Unicode).
   - Character data types include char in C/C++ and Java.
   - The memory representation of characters depends on the character encoding used by the system.

Example in C:

```
c
char letter = 'A'; // 8 bits (1 byte)
```

Memory Representation (ASCII):

```
01000001
```

In this example, the character 'A' is stored in memory using 8 bits, following the ASCII character encoding.

4. **Boolean**:
   - Booleans are typically stored using a single bit, where true is represented by 1 and false is represented by 0.
   - Boolean data types include bool in C++ and C#, and boolean in Java.

Example in C#:

```
csharp
bool isValid = true; // 1 bit
```

Memory Representation:

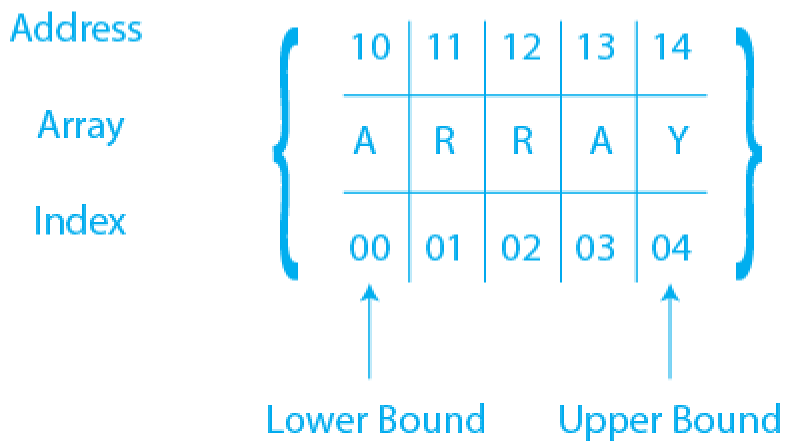
```
1
```

CHAPTER THREE

Arrays

Arrays are fundamental data structures used in computer science and programming. They provide a way to store a collection of elements of the same data type in contiguous memory locations. Arrays offer efficient access to elements using index-based retrieval, making them an essential component in various algorithms and applications. Understanding the different types of arrays and their characteristics is crucial for effective problem-solving and optimizing program performance.

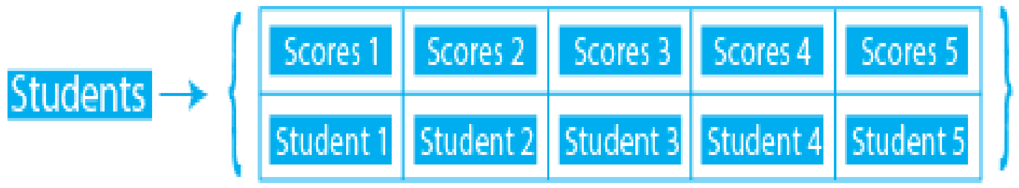
What is an Array Data Structure?



A linear data structure called an array contains elements of the same data type in contiguous and nearby memory regions. Arrays operate using an index system with values ranging from 0 to (n-1), where n is the array's size.

Although it is an array, arrays were introduced for a reason.

Why Do You Need Array Data Structure?



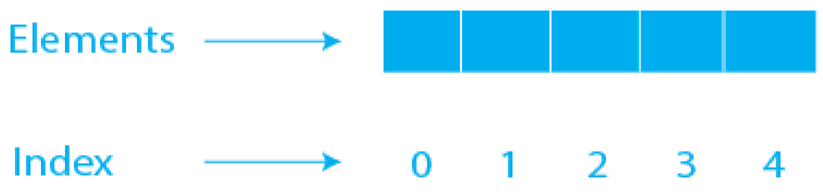
Consider a class of ten students that is required to report its results. It would be difficult to manipulate and preserve the data if you had specified each of the ten variables individually.

It would be more challenging to declare and maintain track of all the variables if more students joined. Arrays were introduced as a solution to this issue.

What Are the Types of Arrays?

There are primarily three types of arrays:

One-Dimensional Arrays:

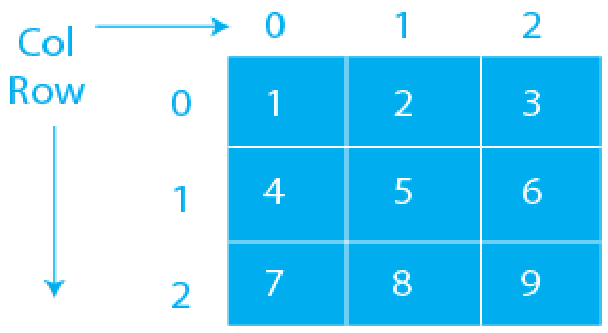


A one-dimensional array can be thought of as a row where elements are kept one after the other.

Multi-Dimensional Arrays:

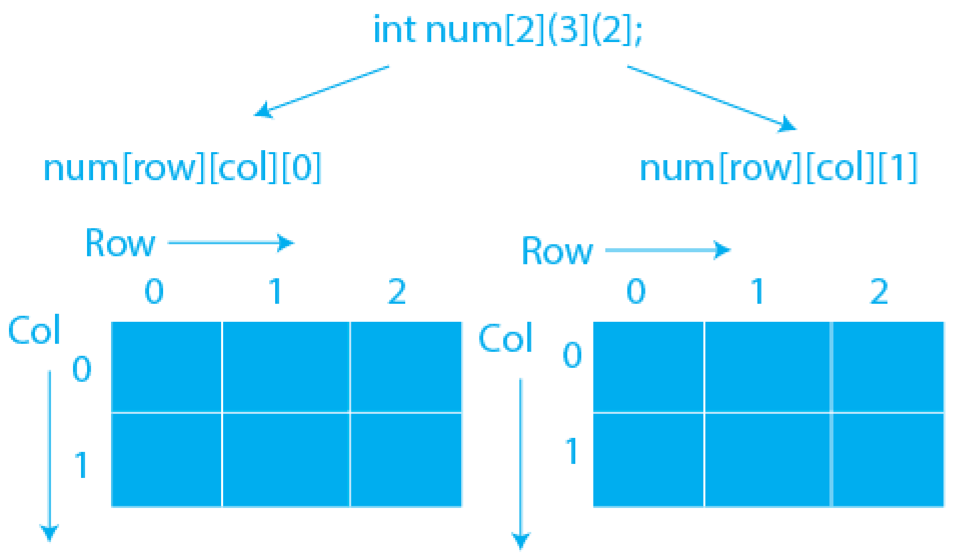
There are two different types of these multidimensional arrays. Those are:

Two-Dimensional Arrays:



It can be compared to a table with elements in each cell.

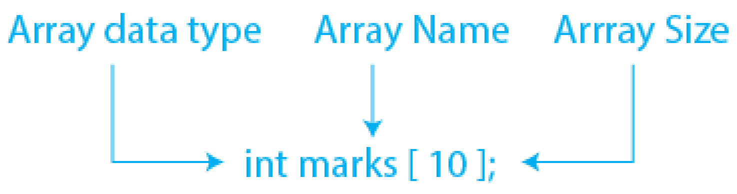
Three-Dimensional Arrays:



It is comparable to a larger cuboid composed of smaller cuboids, where each cuboid can hold a different element.

One-dimensional arrays will be used in this session on "arrays data structure."

How Do You Declare an Array?



The size of the arrays is often the argument in square bracket definitions for arrays.

The syntax for arrays is as follows:

- **1D Arrays: int arr[n];**
- **2D Arrays: int arr[m][n];**
- **3D Arrays: int arr[m][n][o];**

How Do You Initialize an Array?

An array can be initialized in four different ways.:

Method 1:

```
int a[6] = {2, 3, 5, 7, 11, 13};
```

Method 2:

```
int arr[]= {2, 3, 5, 7, 11};
```

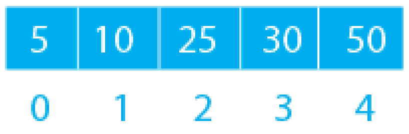
Method 3:

```
int n;

scanf("%d",&n);

int arr[n];

for(int i=0;i< 5;i++)

{

scanf("%d",&arr[i]);

}
```

Method 4:

```
int arr[5];
arr[0]=1;
arr[1]=2;
arr[2]=3;
arr[3]=4;
arr[4]=5;
```

How Can You Access Elements of Arrays in Data Structures?



The index where you stored the element can be used to access it. Let's talk about it using a code:

- C

```
#include<stdio.h>
int main()
{
int a[5] = {2, 3, 5, 7, 11};
printf("%d\n",a[0]); // we are accessing
printf("%d\n",a[1]);
printf("%d\n",a[2]);
printf("%d\n",a[3]);
printf("%d",a[4]);
return 0;
}
```

Output

```
2
3
5
7
11
```

Basic operations of array data structure

- **Traversal** – The array's elements are printed using this operation.
- **Insertion** – It's used to add an element to a specific index.
- **Deletion** – It is used to remove an element from a specific index.
- **Search** – It is used to search for an element using either the value or the specified index.
- **Update** – This operation updates an element at a specific index.

Traversal operation

The array elements are traversed using this operation. It sequentially prints each element of the array. The following program will help us to comprehend it.

```
#include <stdio.h>
void main() {
int Arr[5] = {18, 30, 15, 70, 12};
int i;
printf("Elements of the array are:\n");
for(i = 0; i<5; i++) {
printf("Arr[%d] = %d, ", i, Arr[i]);
}
}
```

Output

```
Elements of the array are:
Arr[0] = 18,  Arr[1] = 30,  Arr[2] = 15,  Arr[3] = 70,  Arr[4] = 12,
```

Insertion operation

One or more members are added to the array by using this method. An element can be added to the array at any index, at the beginning or end, or both, depending on the specifications. Let's see the implementation of adding an element to the array right away.

```
#include <stdio.h>

int main()
{
int arr[20] = { 18, 30, 15, 70, 12 };
int i, x, pos, n = 5;
printf("Array elements before insertion\n");
for (i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");
x = 50; // element to be inserted
pos = 4;
n++;
for (i = n-1; i >= pos; i--)
arr[i] = arr[i - 1];
arr[pos - 1] = x;
printf("Array elements after insertion\n");
for (i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");
return 0;
}
```

Output

```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Deletion operation

As the name suggests, this operation rearranges every element in the array after removing one element from it.

```
#include <stdio.h>
void main() {
int arr[] = {18, 30, 15, 70, 12};
int k = 30, n = 5;
int i, j;
printf("Given array elements are :\n");
for(i = 0; i<n; i++) {
printf("arr[%d] = %d, ", i, arr[i]);
}
j = k;
while( j < n) {
arr[j-1] = arr[j];
j = j + 1;
}
n = n -1;
printf("\nElements of array after deletion:\n");
for(i = 0; i<n; i++) {
printf("arr[%d] = %d, ", i, arr[i]);
}
}
```

Output

```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Elements of array after deletion:
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,
```

Search operation

Using the value or index, this operation is used to search for an element in the array.

```
#include <stdio.h>
void main() {
int arr[5] = {18, 30, 15, 70, 12};
int item = 70, i, j=0 ;
printf("Given array elements are :\n");
for(i = 0; i<5; i++) {
printf("arr[%d] = %d, ", i, arr[i]);
}
printf("\nElement to be searched = %d", item);
while( j < 5){
if( arr[j] == item ) {
break;
}
j = j + 1;
}
printf("\nElement %d is found at %d position", item, j+1);
}
```

Output

```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Element to be searched = 70
Element 70 is found at 4 position
```

Update operation

This action is used to update an array element that is already present and is located at the specified index.

```
#include <stdio.h>

void main() {
int arr[5] = {18, 30, 15, 70, 12};
int item = 50, i, pos = 3;
printf("Given array elements are :\n");
for(i = 0; i<5; i++) {
printf("arr[%d] = %d, ", i, arr[i]);
}
arr[pos-1] = item;
printf("\nArray elements after updation :\n");
for(i = 0; i<5; i++) {
printf("arr[%d] = %d, ", i, arr[i]);
}
}
```

Output

Given array elements are :

arr[0] = 18, arr[1] = 30, arr[2] = 15, arr[3] = 70, arr[4] = 12,

Array elements after updation :

arr[0] = 18, arr[1] = 30, arr[2] = 50, arr[3] = 70, arr[4] = 12,

Complexity of Array operations

The following table lists the time and space complexity of several array operations..

Advantages of Array

- The group of identically named variables is referred to as an array. As a result, it is simple to recall the names of all the array's elements.
- It is relatively easy to navigate an array; all we need to do is increase the array's base address to visit each element one at a time.
- The index can be used to directly access any element in the array.

Disadvantages of Array

- The array is uniform. This implies that it can store elements with similar data types.
- An array has static memory allocation, meaning that its size cannot be changed.
- If we store less elements than the given size, memory will be wasted.

Limitations of arrays

Fixed size in traditional arrays, making resizing difficult.

- Contiguous memory allocation can lead to memory fragmentation.
- Inefficient insertion or deletion operations in fixed-size arrays

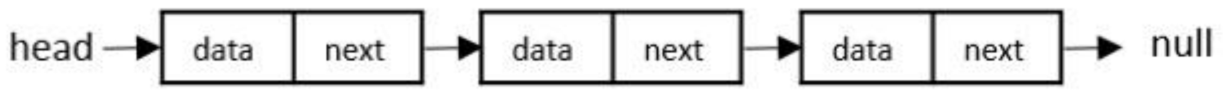
CHAPTER FOUR
Linked List

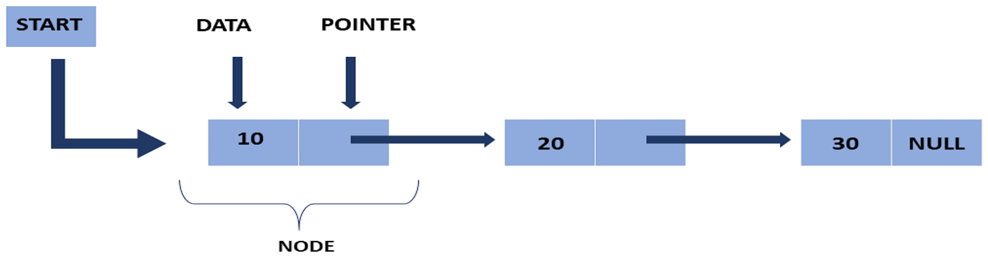
A linked list is a linear data structure which can store a collection of "nodes" connected together via links i.e. pointers. Linked lists nodes are not stored at a contiguous location, rather they are linked using pointers to the different memory locations. A node consists of the data value and a pointer to the address of the next node within the linked list.

A linked list is a dynamic linear data structure whose memory size can be allocated or de-allocated at run time based on the operation insertion or deletion, this helps in using system memory efficiently. Linked lists can be used to implement various data structures like a stack, queue, graph, hash maps, etc.

Representation of a Linked List

This representation of a linked list depicts that each node consists of two fields. The first field consists of data, and the second field consists of pointers that point to another node.





A linked list starts with a **head** node which points to the first node. Every node consists of data which holds the actual data (value) associated with the node and a next pointer which holds the memory address of the next node in the linked list. The last node is called the tail node in the list which points to **null** indicating the end of the list.

Creation of Node and Declaration of Linked Lists

```
struct node

{

 int data;

 struct node * next;

};

struct node * n;

n=(struct node*)malloc(sizeof(struct node*));
```

It is a declaration of a node that consists of the first variable as data and the next as a pointer, which will keep the address of the next node.

Here you need to use the malloc function to allocate memory for the nodes dynamically.

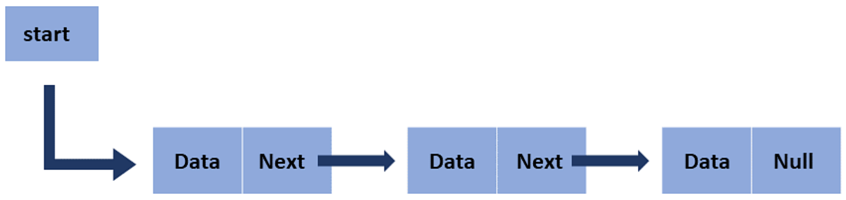
Types of Linked Lists

The linked list mainly has three types, they are:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Singly Linked List

A singly linked list is the most common type of linked list. Each node has data and an address field that contains a reference to the next node.



Advantages:

1. **Dynamic Size:** Singly linked lists can dynamically grow and shrink in size during execution, unlike arrays, whose size is fixed at compile time.
2. **Efficient Insertions and Deletions:** Insertions and deletions at the beginning of the list are very efficient with singly linked lists, as they require only updating the pointers and do not involve shifting elements like in arrays.
3. **Ease of Implementation:** Singly linked lists are relatively easy to implement and understand compared to other complex data structures like trees and graphs.
4. **Memory Efficiency:** Singly linked lists consume memory only for the data and the pointer to the next node, making them memory-efficient when the exact size of the data structure is unknown.

Disadvantages:

1. **No Random Access:** Unlike arrays, singly linked lists do not support random access to elements. Traversal must start from the head node, making operations like accessing the nth element less efficient.
2. **Extra Memory Overhead:** Each node in a singly linked list requires extra memory for storing the pointer to the next node, leading to higher memory overhead compared to arrays for storing the same amount of data.
3. **Traversal Overhead:** Traversing a singly linked list requires traversing each node sequentially from the head to the desired node, which can be slower compared to direct access in arrays.

Use Cases:

1. **Dynamic Data Structures:** Singly linked lists are suitable for implementing dynamic data structures where the size of the data structure changes frequently during runtime.
2. **Frequent Insertions and Deletions:** Applications that involve frequent insertions and deletions, especially at the beginning of the list, can benefit from using singly linked lists due to their efficient insertion and deletion operations.
3. **Implementation of Stacks and Queues:** Singly linked lists serve as the underlying data structure for implementing stack and queue data structures, where elements are inserted and removed from one end of the list.

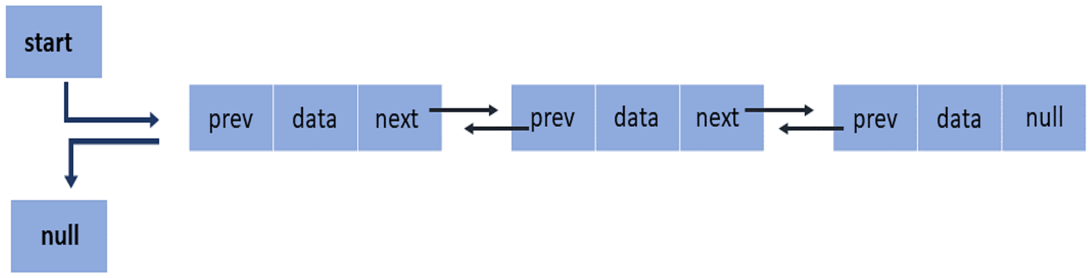
4. **Memory-Constrained Environments:** In memory-constrained environments where contiguous memory allocation is not possible, singly linked lists offer a flexible alternative for storing and managing data.

Algorithm: Singly Linked List Implementation

```
1. Define a structure for a node with two components: data and a pointer to the next node.
2. Define a structure for the linked list with a pointer to the head node.
3. Initialize the head pointer to NULL, indicating an empty list.
4. Define functions to perform various operations on the linked list:
   a. Insertion:
      i. Create a new node with the given data.
      ii. If the list is empty, set the new node as the head node.
      iii. Otherwise, traverse the list to the last node.
      iv. Set the next pointer of the last node to the new node.
   b. Deletion:
      i. If the list is empty, return an error.
      ii. If the node to be deleted is the head node, update the head pointer to point to the next node.
      iii. Otherwise, traverse the list to find the node before the node to be deleted.
      iv. Update the next pointer of the previous node to skip over the node to be deleted.
      v. Free the memory allocated to the deleted node.
   c. Traversal:
      i. Start from the head node.
      ii. Traverse each node in the list using the next pointer until NULL is reached.
      iii. Process each node as required.
5. Provide functions to create and destroy the linked list.
6. Test the implementation with various scenarios to ensure correctness.

End Algorithm
```

Doubly Linked List

There are two pointer storage blocks in the doubly linked list. The first pointer block in each node stores the address of the previous node. Hence, in the doubly linked inventory, there are three fields that are the previous pointers, that contain a reference to the previous node. Then there is the data, and last you have the next pointer, which points to the next node. Thus, you can go in both directions (backward and forward).



Advantages of Doubly Linked List:

1. **Bidirectional Traversal:** Unlike singly linked lists, doubly linked lists allow traversal in both directions, forward and backward.
2. **Insertion and Deletion Efficiency:** Insertions and deletions at the beginning, end, or middle of the list can be done efficiently with constant time complexity, O(1).
3. **Memory Overhead:** Each node contains two pointers, one for the next node and one for the previous node, allowing efficient memory utilization and easy navigation.

Disadvantages of Doubly Linked List:

1. **Increased Memory Usage:** The use of an additional pointer per node increases the memory overhead compared to singly linked lists.
2. **Complexity:** Implementing and maintaining doubly linked lists can be more complex than singly linked lists due to the bidirectional nature of the structure.
3. **Overhead for Traversal:** While doubly linked lists offer bidirectional traversal, this capability comes with the cost of additional memory overhead and potentially increased complexity.

Uses of Doubly Linked List:

1. **Implementation of Data Structures:** Doubly linked lists are used as a foundational data structure in various applications such as stacks, queues, and hash tables.
2. **Text Editors:** Doubly linked lists are commonly used in text editors to implement features like undo and redo functionality, where bidirectional traversal is essential.
3. **Memory Allocation:** Doubly linked lists can be used in memory allocation schemes to manage dynamic memory efficiently.

Algorithm: Doubly Linked List Implementation

```
1. Define a structure for a node with three components: data, a pointer to the next node, and a pointer to the previous node.
2. Define a structure for the doubly linked list with pointers to the head and tail nodes.
3. Initialize the head and tail pointers to NULL, indicating an empty list.
4. Define functions to perform various operations on the doubly linked list:
   a. Insertion:
      i. Create a new node with the given data.
      ii. If the list is empty, set the new node as both the head and tail node.
      iii. Otherwise, set the next pointer of the new node to the current head node.
      iv. Set the previous pointer of the current head node to the new node.
      v. Update the head pointer to point to the new node.
   b. Deletion:
      i. If the list is empty, return an error.
      ii. If the node to be deleted is the head node, update the head pointer to point to the next node.
      iii. If the node to be deleted is the tail node, update the tail pointer to point to the previous node.
      iv. Otherwise, traverse the list to find the node to be deleted.
      v. Update the next pointer of the previous node to skip over the node to be deleted.
      vi. Update the previous pointer of the next node to skip back to the previous node.
      vii. Free the memory allocated to the deleted node.
   c. Traversal:
      i. Start from the head node.
      ii. Traverse each node in the list using the next pointer until NULL is reached.
```

iii. Process each node as required.

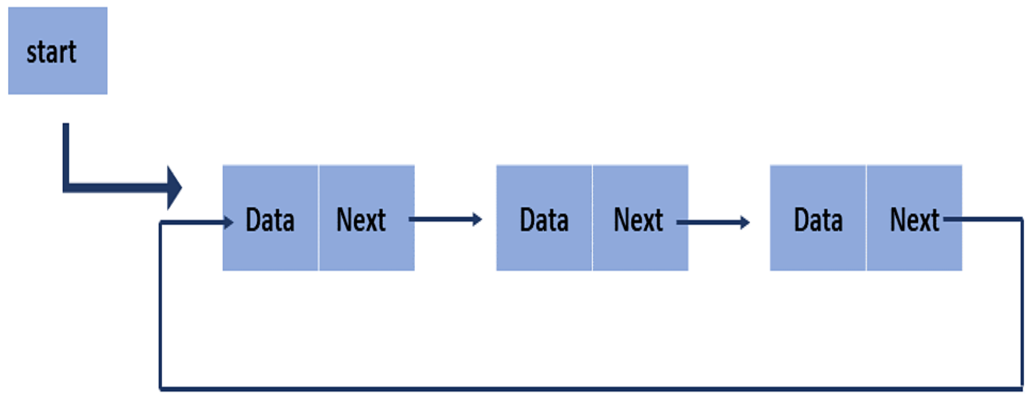
5. Provide functions to create and destroy the doubly linked list.

6. Test the implementation with various scenarios to ensure correctness.

End Algorithm

Circular Linked List

The circular linked list is extremely similar to the singly linked list. The only difference is that the last node is connected with the first node, forming a circular loop in the circular linked list.



Advantages of Circular Linked List:

1. **Efficient Memory Utilization:** Circular linked lists can efficiently utilize memory since each node only requires a pointer to the next node, and the last node points back to the first node, forming a loop.
2. **Dynamic Size:** Circular linked lists can dynamically grow and shrink during runtime by adding or removing nodes without the need to resize the structure.
3. **Circular Traversal:** Circular linked lists allow for circular traversal, meaning that after reaching the last node, the traversal continues from the first node, making it suitable for applications where continuous looping is required.

Disadvantages of Circular Linked List:

1. **Complexity:** Implementing and maintaining circular linked lists can be more complex compared to linear linked lists due to the circular nature of the structure, which requires careful handling to avoid infinite loops or memory leaks.
2. **Traversal Complexity:** Circular traversal can lead to infinite loops if not implemented properly, posing challenges for operations like searching or traversing the entire list.

Uses of Circular Linked List:

1. **Circular Buffer:** Circular linked lists are used to implement circular buffers or queues, where data items are stored in a fixed-size buffer and accessed in a circular manner.
2. **Round-Robin Scheduling:** Circular linked lists are employed in operating system scheduling algorithms like round-robin scheduling, where processes are scheduled in a circular order.
3. **Game Development:** Circular linked lists can be utilized in game development for managing game objects or entities that need to loop through a sequence of actions or events.

Algorithm for Circular Linked List Implementation:

```
1. Define a structure for a node with two components: data and a pointer to the next node.
2. Define a structure for the circular linked list with a pointer to the head node.
3. Initialize the head pointer to NULL, indicating an empty list.
4. Define functions to perform various operations on the circular linked list:
   a. Insertion:
      i. Create a new node with the given data.
      ii. If the list is empty, set the new node as both the head node and the last node.
      iii. Otherwise, traverse the list to reach the last node.
      iv. Set the next pointer of the last node to point to the new node.
      v. Set the next pointer of the new node to point back to the head node.
      vi. Update the head pointer to point to the new node if inserting at the beginning.
   b. Deletion:
      i. If the list is empty, return an error.
      ii. If there is only one node in the list, free the memory allocated to the node and set the
head pointer to NULL.
```

iii. Otherwise, traverse the list to find the node to be deleted.

iv. Update the next pointer of the previous node to skip over the node to be deleted.

v. Free the memory allocated to the deleted node.

c. Traversal:

i. If the list is empty, return.

ii. Start from the head node and traverse each node in the list using the next pointer until reaching the head node again.

iii. Process each node as required.

5. Provide functions to create and destroy the circular linked list.
6. Test the implementation with various scenarios to ensure correctness.

End Algorithm

Circular link lists can either be singly or doubly-linked lists.

- The next node's next pointer will point to the first node to form a singly linked list.
- The previous pointer of the first node keeps the address of the last node to form a doubly-linked list.

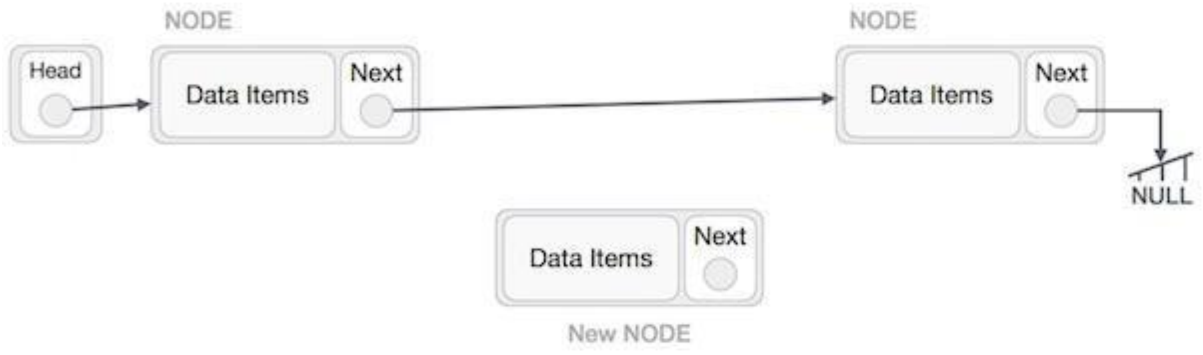
Basic Operations in Linked List

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below −

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Linked List - Insertion Operation

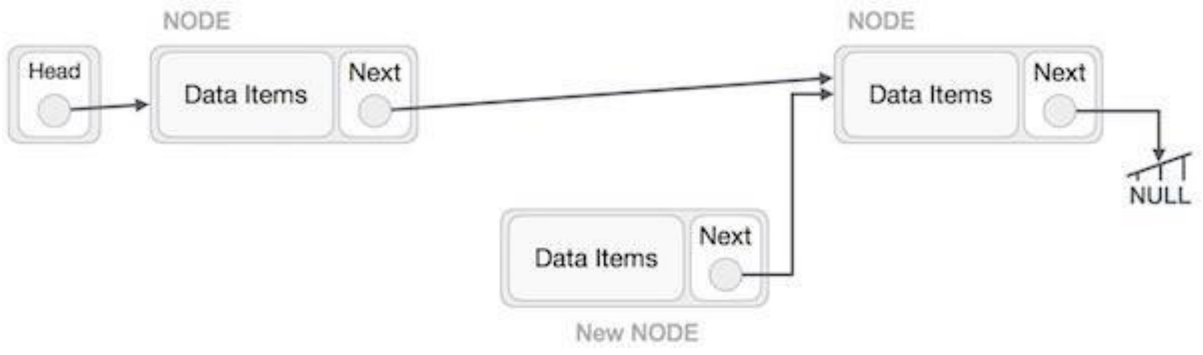
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −

NewNode.next -> RightNode;

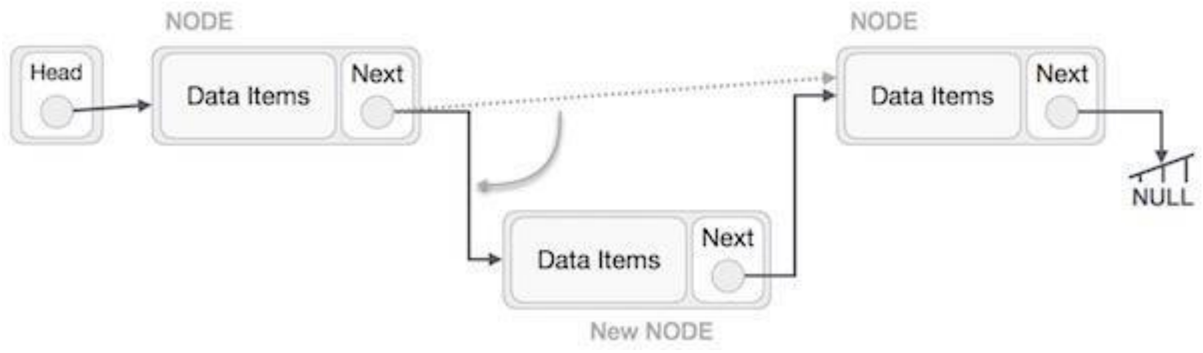
It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next -> NewNode;

This will put the new node in the middle of the two. The new list should look like this −



Insertion in linked list can be done in three different ways. They are explained as follows −

Insertion at Beginning

In this operation, we are adding an element at the beginning of the list.

Algorithm

```
1. START
2. Create a node to store the data
3. Check if the list is empty
4. If the list is empty, add the data to the node and
   assign the head pointer to it.
5. If the list is not empty, add the data to a node and link to the
   current head. Assign the head to the newly added node.
6. END
```

Example

Following are the implementations of this operation in various programming languages −

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");
   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(44);
   insertatbegin(50);
   printf("Linked List: ");

   // print list
   printList();
}
```

Output

```
Linked List:
[ 50  44  30  22  12 ]
```

Insertion at Ending

In this operation, we are adding an element at the ending of the list.

Algorithm

```
1. START
2. Create a new node and assign the data
3. Find the last node
4. Point the last node to new node
5. END
```

Insertion at a Given Position

In this operation, we are adding an element at any position within the list.

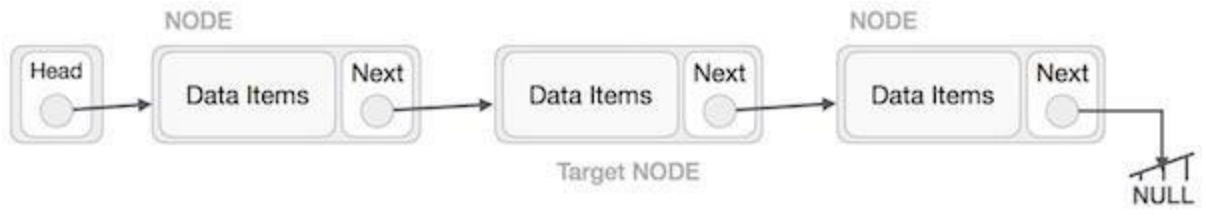
Algorithm

```
1. START
2. Create a new node and assign data to it
3. Iterate until the node at position is found
4. Point first to new first node
5. END
```

Example

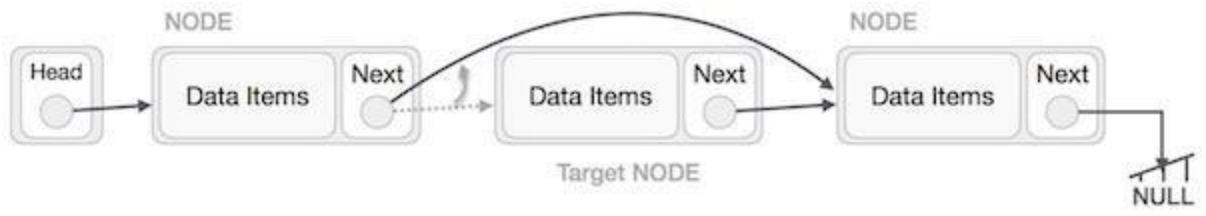
Linked List - Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



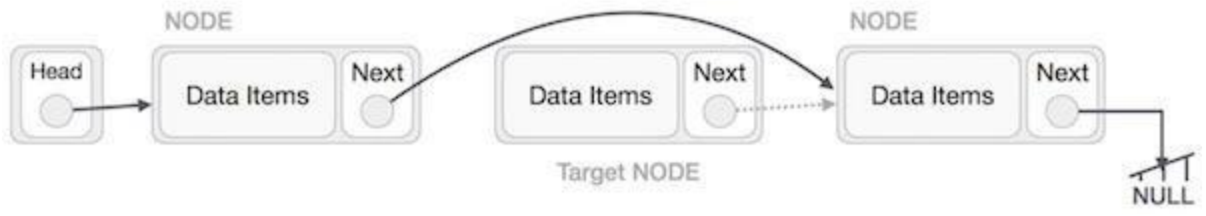
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next -> TargetNode.next;

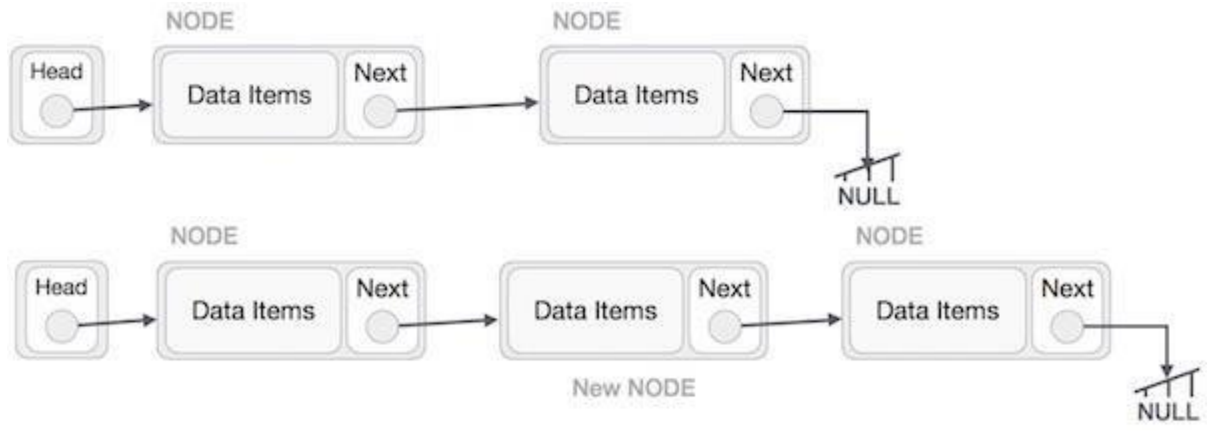


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next -> NULL;



We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion in linked lists is also performed in three different ways. They are as follows −

Deletion at Beginning

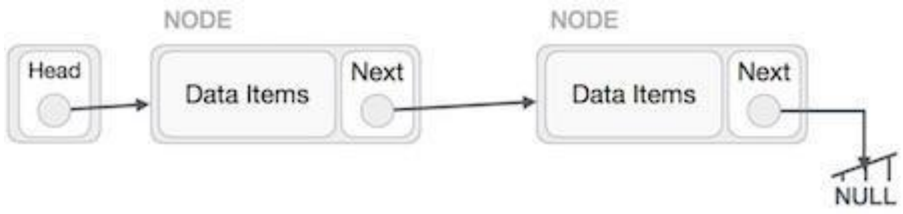
In this deletion operation of the linked, we are deleting an element from the beginning of the list. For this, we point the head to the second node.

Algorithm

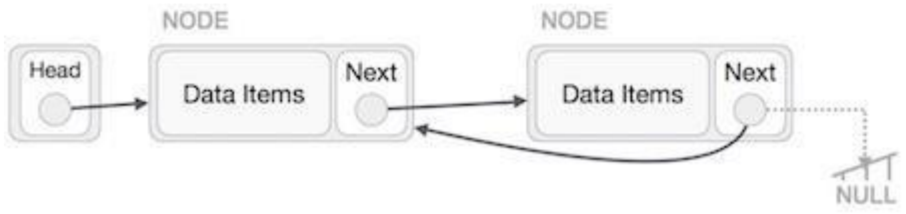
1. START
2. Assign the head pointer to the next node in the list
3. END

Linked List - Reversal Operation

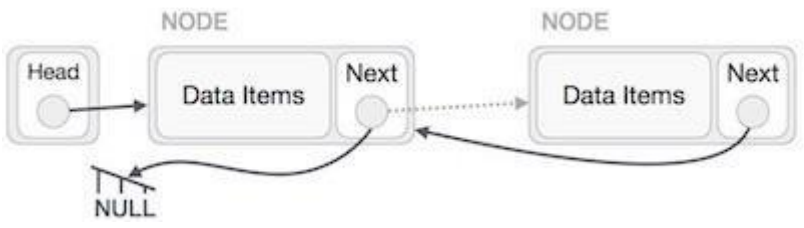
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



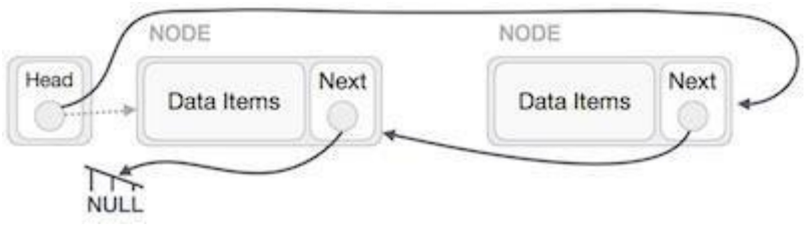
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



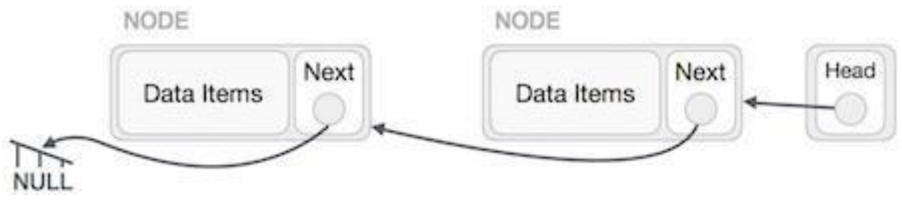
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



Algorithm

Step by step process to reverse a linked list is as follows −

1. START
2. We use three pointers to perform the reversing:
   prev, next, head.
3. Point the current node to head and assign its next value to
   the prev node.
4. Iteratively repeat the step 3 for all the nodes in the list.
5. Assign head to the prev node.

Linked List - Search Operation

Searching for an element in the list using a key element. This operation is done in the same way as array search; comparing every element in the list with the key element given.

Algorithm

```
1 START
2 If the list is not empty, iteratively check if the list
  contains the key
3 If the key element is not present in the list, unsuccessful
  search
4 END
```

Linked Lists vs Arrays

Linked lists and arrays are both data structures used to store collections of elements, but they have different characteristics and are suitable for different scenarios. Let's compare them across various aspects:

1. **Memory Allocation:**
    - **Arrays:** Contiguous block of memory is allocated to store elements. Memory is allocated at once, and the size typically cannot be changed dynamically.
    - **Linked Lists:** Elements are stored in nodes, where each node contains the data and a pointer/reference to the next node. Nodes can be scattered in memory, and memory is allocated dynamically as nodes are added.
2. **Insertion and Deletion:**
    - **Arrays:** Insertion and deletion operations can be inefficient, especially in the middle, as elements may need to be shifted.
    - **Linked Lists:** Insertion and deletion operations are generally efficient, as they involve changing pointers to rearrange the structure.
3. **Access Time:**
    - **Arrays:** Random access to elements is fast using indices. Access time is O(1).

   - **Linked Lists:** Access time is slower compared to arrays for random access, as elements must be traversed sequentially. Access time is O(n).
4. **Memory Efficiency:**
   - **Arrays:** May waste memory if the size is greater than needed, especially if the size needs to be dynamic.
   - **Linked Lists:** Typically more memory-efficient as memory is allocated dynamically for each element.
5. **Dynamic Size:**
   - **Arrays:** Size is fixed once allocated. Dynamic resizing may involve creating a new array and copying elements.
   - **Linked Lists:** Can easily grow or shrink in size by adding or removing nodes.
6. **Implementation Complexity:**
   - **Arrays:** Simpler to implement, with basic operations like indexing.
   - **Linked Lists:** More complex to implement due to pointer manipulation, but offer flexibility and efficiency in certain scenarios.
7. **Cache Performance:**
   - **Arrays:** Better cache performance due to contiguous memory allocation, which improves locality of reference.
   - **Linked Lists:** Poor cache performance as nodes may be scattered in memory, leading to more cache misses.
8. **Use Cases:**
   - **Arrays:** Suitable for scenarios where random access and fixed size are required, such as mathematical computations, lookup tables, and when memory usage is known in advance.
   - **Linked Lists:** Suitable for scenarios where dynamic size, efficient insertion/deletion, and flexibility are required, such as implementing stacks, queues, and when memory usage may vary dynamically.

Some of the applications for linked lists are as follows:

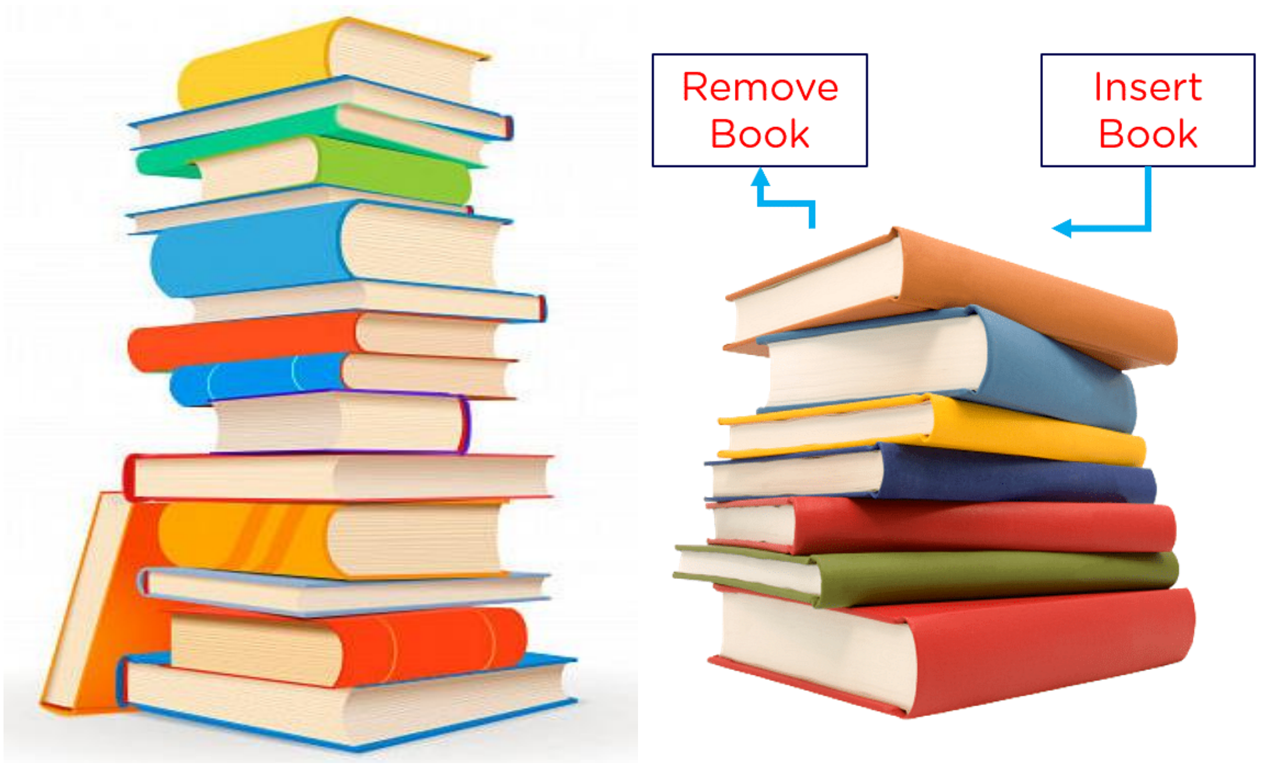
- A linked list can be used to implement stacks and queues.
- A linked list can also be used to implement graphs whenever we have to represent graphs as adjacency lists.

- A mathematical polynomial can be stored as a linked list.
- In the case of hashing technique, the buckets used in hashing are implemented using the linked lists.
- Whenever a program requires dynamic allocation of memory, we can use a linked list as linked lists work more efficiently in this case.

CHAPTER FIVE

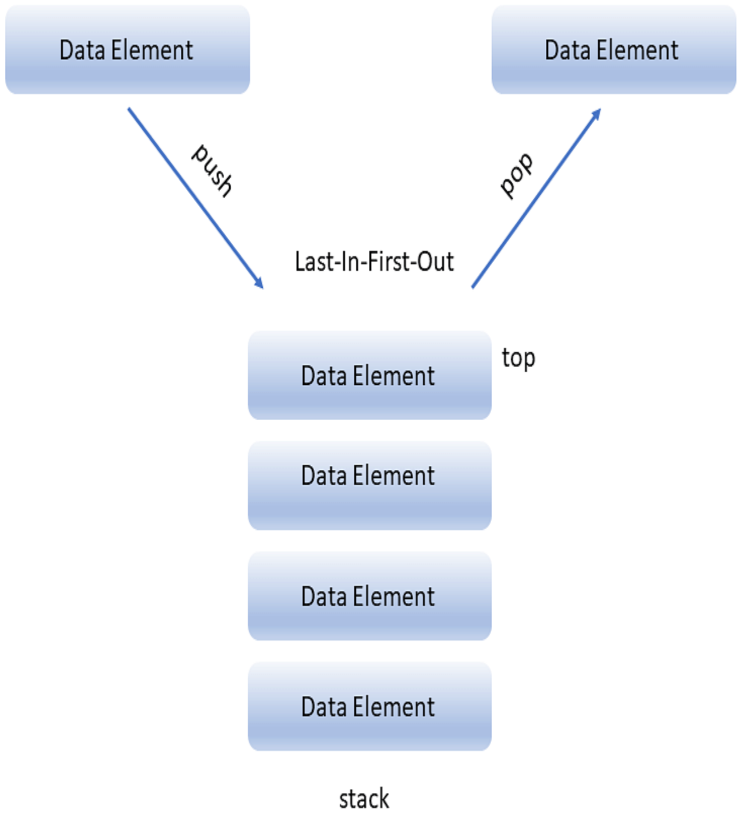
Stack in Data Structures

The stack data structure is a linear data structure that accompanies a principle known as LIFO (Last In First Out) or FILO (First In Last Out). Real-life examples of a stack are a deck of cards, piles of books, piles of money, and many more.



This example allows you to perform operations from one end only, like when you insert and remove new books from the top of the stack. It means insertion and deletion in the stack data structure can be done only from the top of the stack. You can access only the top of the stack at any given point in time.

- Inserting a new element in the stack is termed a push operation.
- Removing or deleting elements from the stack is termed pop operation.

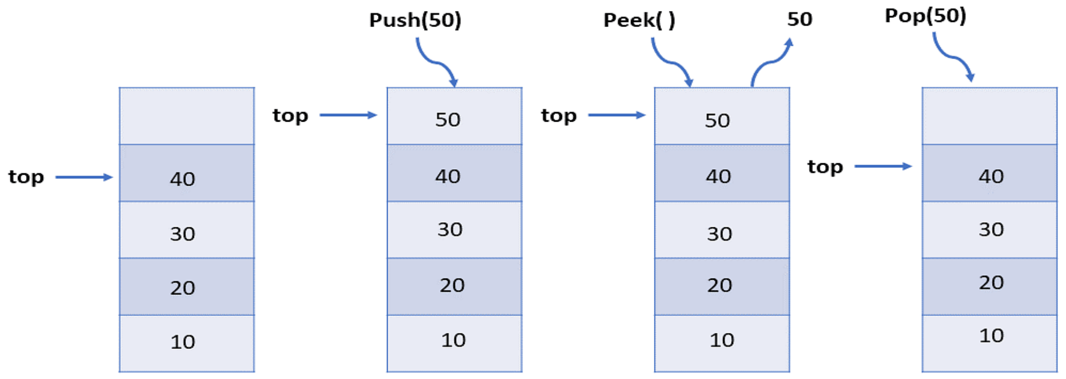


Stack Representation in Data Structures

Working of Stack in Data Structures

Now, assume that you have a stack of books. You can only see the top, i.e., the top-most book, namely 40, which is kept top of the stack.

If you want to insert a new book first, namely 50, you must update the top and then insert a new text. And if you want to access any other book other than the topmost book that is 40, you first remove the topmost book from the stack, and then the top will point to the next topmost book.



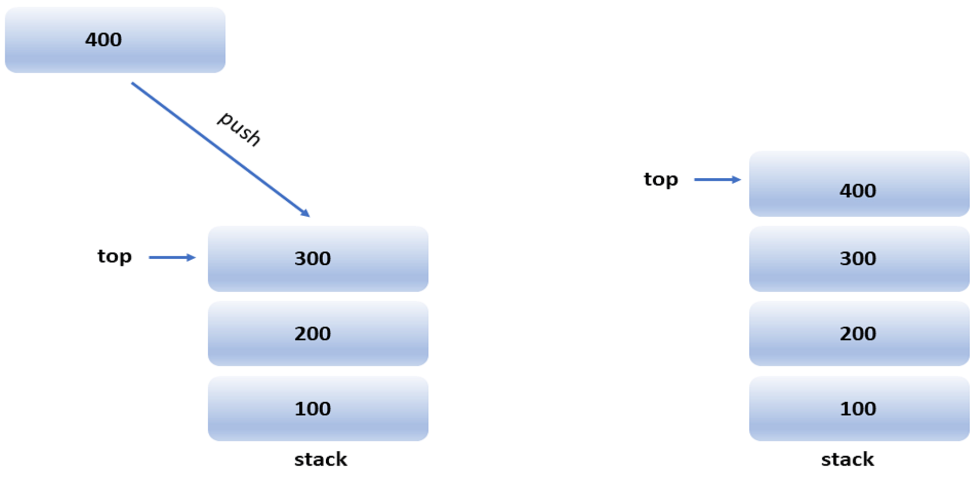
After working on the representation of stacks in data structures, you will see some basic operations performed on the stacks in data structures.

Basic Operations on Stack in Data Structures

There following are some operations that are implemented on the stack.

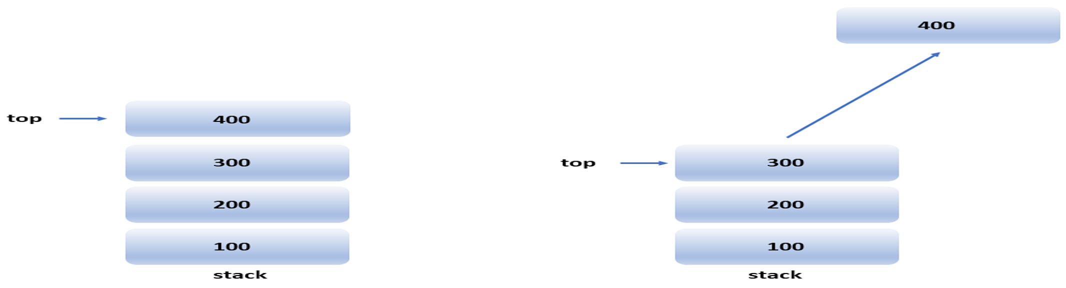
Push Operation

Push operation involves inserting new elements in the stack. Since you have only one end to insert a unique element on top of the stack, it inserts the new element at the top of the stack.



Pop Operation

Pop operation refers to removing the element from the stack again since you have only one end to do all top of the stack. So removing an element from the stack is termed pop operation.



Peek Operation: Peek operation refers to retrieving the topmost element in the stack without removing it from the collections of data elements.

isFull(): isFull function is used to check whether or not a stack is empty.

isEmpty(): isEmpty function is used to check whether or not a stack is empty.

First, you will learn about the functions:

isFull()

The following is the algorithm of the isFull() function:

```
begin
 If
    top equals to maxsize
        return true
else
       return false
else if
end
```

The implementation of the isFull() function is as follows:

```
Bool isFull()
{
 if(top == maxsize)
 return true;
else
 return false;
}
```

isEmpty()

The following is the algorithm of the isEmpty() function:

```
begin
 If
    topless than 1
        return true
else
       return false
else if
end
```

The implementation of the isEmpty() function is:

```
Bool isEmpty()
{
 if(top = = -1)
 return true;
else
 return false;
}
```

Push Operation

Push operation includes various steps, which are as follows :

Step 1: First, check whether or not the stack is full

Step 2: If the stack is complete, then exit

Step 3: If not, increment the top by one

Step 4: Insert a new element where the top is pointing

Step 5: Success

The algorithm of the push operation is:

```
Begin push: stack, item

If the stack is complete, return null

end if

top ->top+1;

stack[top] <- item

end
```

This is how you implement a push operation:

```
if(! isFull ())

{

top = top + 1;

stack[top] = item;

}

else {

 printf("stack is full");

}
```

Pop Operation

Step 1: First, check whether or not the stack is empty

Step 2: If the stack is empty, then exit

Step 3: If not, access the topmost data element

Step 4: Decrement the top by one

Step 5: Success

The following is the algorithm of the pop operation:

```
Begin pop: stack

If the stack is empty

 return null

end if

item -> stack[top] ;

Top -> top - 1;

Return item;

end
```

Implementing a pop operation is as follows:

```
int pop( int item){

If isEmpty()) {

item = stack[top];

top = top - 1;
return item;

}

else{

printf(“stack if empty”);

}

}
```

Peek Operation

The algorithm of a peek operation is:

```
begin to peek

return stack[top];

end
```

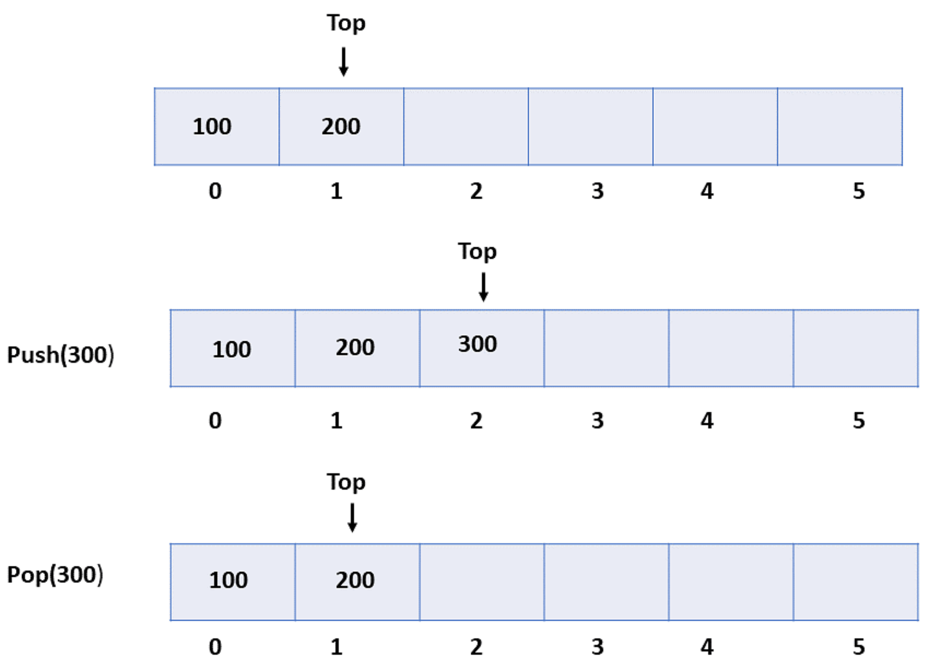
The implementation of the peek operation is:

```
int peek()

{

return stack[top];

}
```

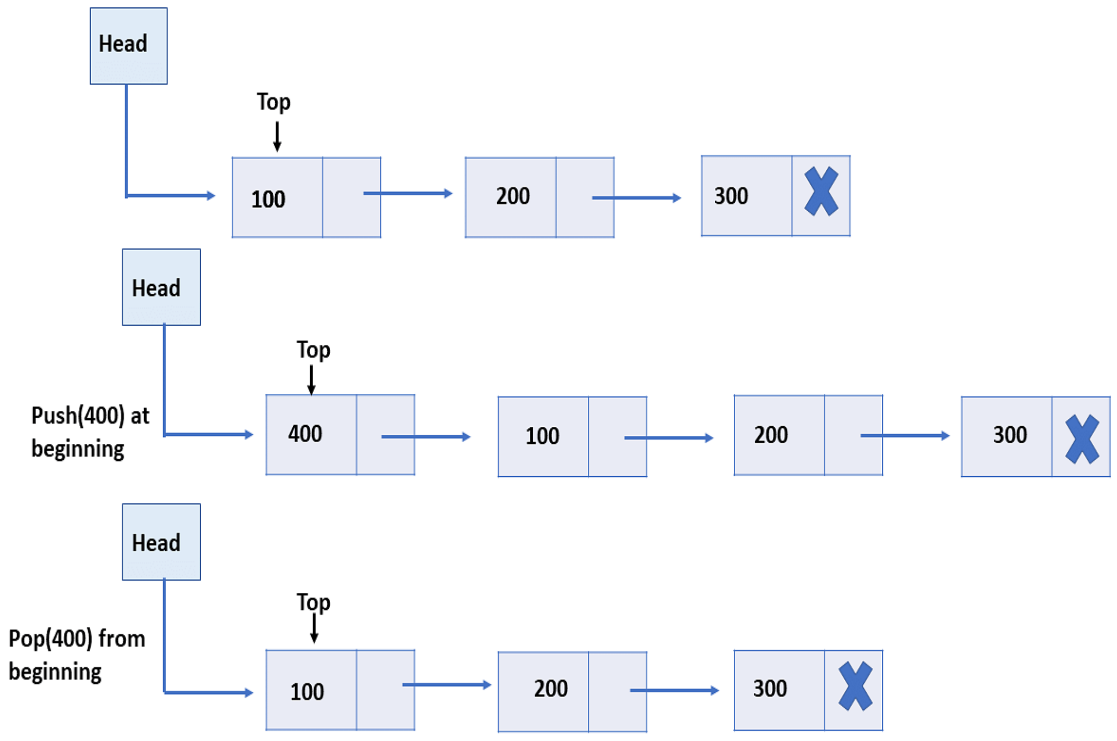
Implementation of Stack in Data Structures

You can perform the implementation of stacks in data structures using two data structures that are an array and a linked list.

- Array: In array implementation, the stack is formed using an array. All the operations are performed using arrays. You will see how all operations can be implemented on the stack in data structures using an array data structure.



- Linked-List: Every new element is inserted as a top element in the linked list implementation of stacks in data structures. That means every newly inserted element is pointed to the top. Whenever you want to remove an element from the stack, remove the node indicated by the top, by moving the top to its previous node in the list.



Application of Stack in Data Structures

- Expression Evaluation and Conversion
- Backtracking
- Function Call
- Parentheses Checking
- String Reversal
- Syntax Parsing
- Memory Management

1. Expression Evaluation and Conversion

There are three types of expression that you use in programming, they are:

Infix Expression: An infix expression is a single letter or an operator preceded by one single infix string followed by another single infix string.

- X

- X + Y
- (X + Y) + (A - B)

Prefix Expression: A prefix expression is a single letter or an operator followed by two prefix strings.

- X
- + X Y
- + + X Y - A B

Postfix Expression: A postfix expression (also called Reverse Polish Notation) is a single letter or an operator preceded by two postfix strings.

- X
- X Y +
- X Y + C D - +

Similarly, the stack is used to evaluate these expressions and convert these expressions like infix to prefix or infix to postfix.

2. Backtracking

Backtracking is a recursive algorithm mechanism that is used to solve optimization problems.

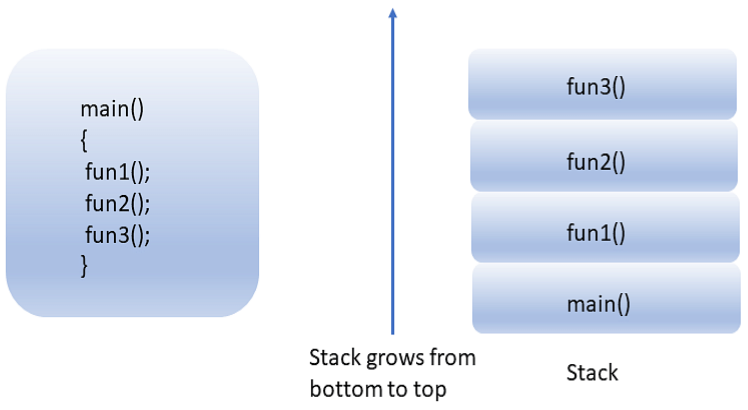
To solve the optimization problem with backtracking, you have multiple solutions; it does not matter if it is correct. While finding all the possible solutions in backtracking, you store the previously calculated problems in the stack and use that solution to resolve the following issues.

The N-queen problem is an example of backtracking, a recursive algorithm where the stack is used to solve this problem.

3. Function Call

Whenever you call one function from another function in programming, the reference of calling function stores in the stack. When the function call is terminated, the program control moves back to the function call with the help of references stored in the stack.

So stack plays an important role when you call a function from another function.



4. Parentheses Checking

Stack in data structures is used to check if the parentheses like ( ), { } are valid or not in programing while matching opening and closing brackets are balanced or not.

So it stores all these parentheses in the stack and controls the flow of the program.

For e.g ((a + b) * (c + d)) is valid but {{a+b})) *(b+d}] is not valid.

5. String Reversal

Another exciting application of stack is string reversal. Each character of a string gets stored in the stack.

The string's first character is held at the bottom of the stack, and the last character of the string is held at the top of the stack, resulting in a reversed string after performing the pop operation.

6. Syntax Parsing

Since many programming languages are context-free languages, the stack is used for syntax parsing by many compilers.

7. Memory Management

Memory management is an essential feature of the operating system, so the stack is heavily used to manage memory.

CHAPTER SIX

Queue in Data Structure

Queue in data structures is a linear collection of different data types which follow a specific order while performing various operations. It can only be modified by the addition of data entities at one end or the removal of data entities at another. By convention, the end where insertion is performed is called Rear, and the end at which deletion takes place is known as the Front.

These constraints of queue make it a First-In-First-Out (FIFO) data structure, i.e., the data element inserted first will be accessed first, and the data element inserted last will be accessed last. This is equivalent to the requirement that once an additional data element is added, all previously added elements must be removed before the new element can be removed. That's why more abstractly, a queue in a data structure is considered being a sequential collection.

Once you are clear with the key terms related to a queue data structure, you will now look at its representation details.

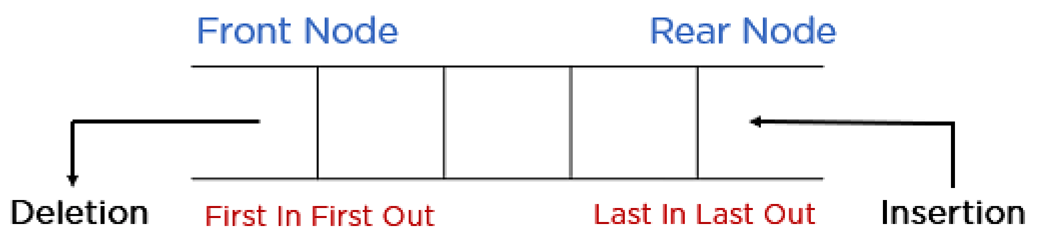
Queue Representation

Before dealing with the representation of a queue, examine the real-life example of the queue to understand it better. The movie ticket counter is an excellent example of a queue where the customer that came first will be served first. Also, the barricades of the movie ticket counter stop in-between disruption to attain different operations at different ends.

TICKET
Person who enters first receives the movie ticket first
Open at both ends!
Person who enters last will receive the ticket last

The queue in the data structure acts the same as the movie ticket counter. Both the ends of this abstract data structure remain open. Further, the insertion and deletion processes also operate analogously to the wait-up line for tickets.

The following diagram tries to explain queue representation as a data structure:



Queue Representation

A queue can be implemented using Arrays, Linked-lists, Pointers, and Structures. The implementation using one-dimensional arrays is the easiest method of all the mentioned methods.

With this understanding of queue representation, look at the different operations that can be performed on the queues in data structures.

Basic Operations for Queue in Data Structure

Unlike arrays and linked lists, elements in the queue cannot be operated from their respective locations. They can only be operated at two data pointers, front and rear. Also, these operations involve standard procedures like initializing or defining data structure, utilizing it, and then wholly erasing it from memory. Here, you must try to comprehend the operations associated with queues:

- Enqueue() - Insertion of elements to the queue.
- Dequeue() - Removal of elements from the queue.
- Peek() - Acquires the data element available at the front node of the queue without deleting it.
- isFull() - Validates if the queue is full.

- isNull() - Checks if the queue is empty.

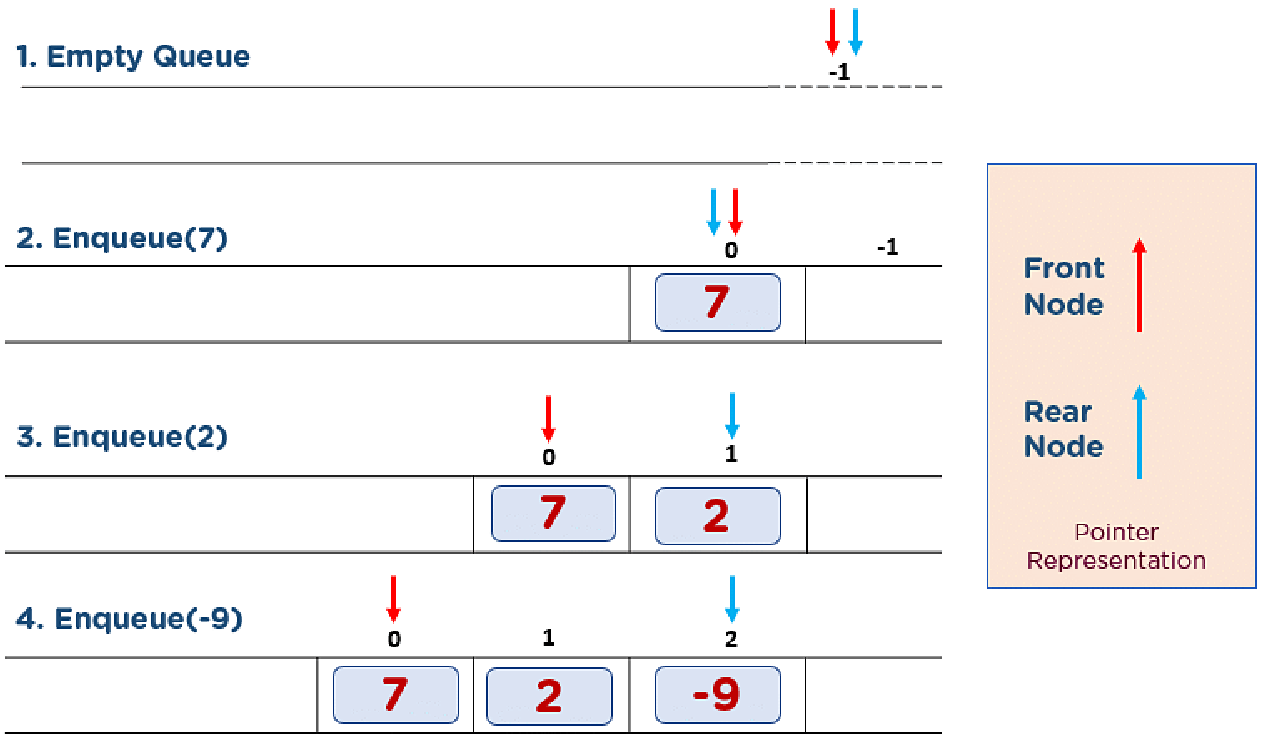
When you define the queue data structure, it remains empty as no element is inserted into it. So, both the front and rear pointer should be set to -1 (Null memory space). This phase is known as data structure declaration in the context of programming.

First, understand the operations that allow the queue to manipulate data elements in a hierarchy.

Enqueue() Operation

The following steps should be followed to insert (enqueue) data element into a queue -

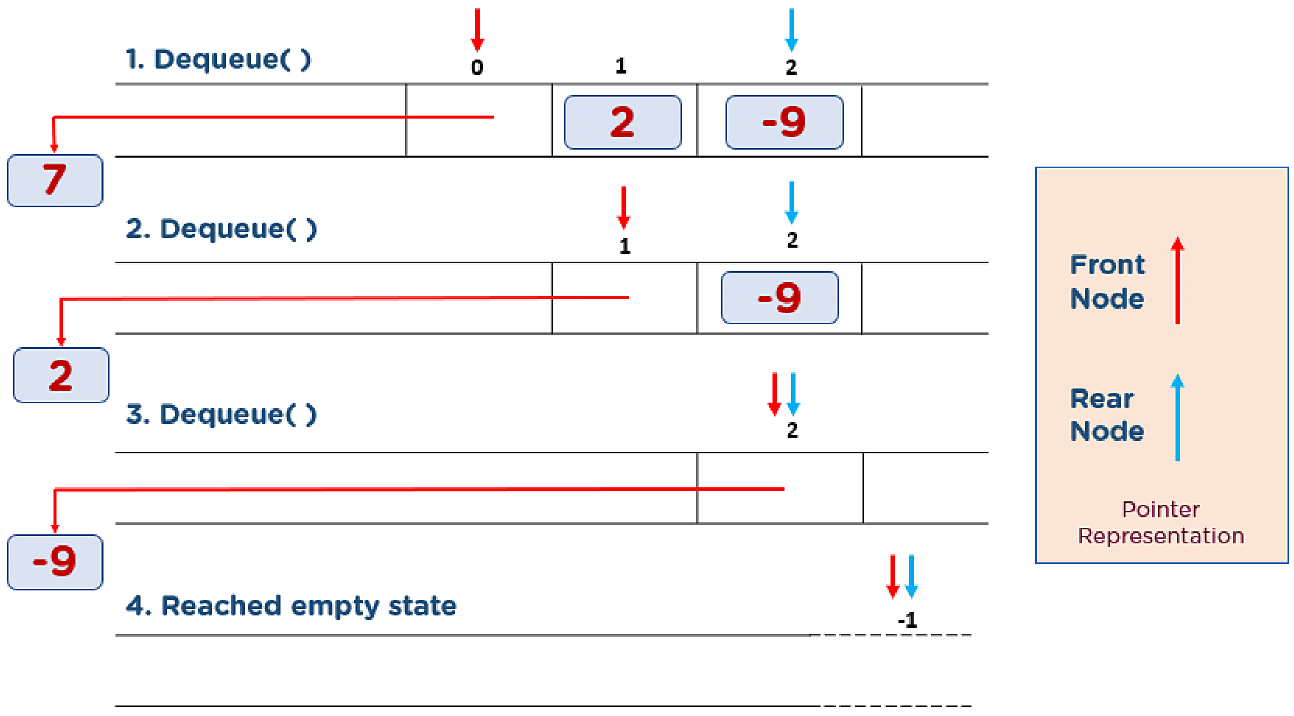
- Step 1: Check if the queue is full.
- Step 2: If the queue is full, Overflow error.
- Step 3: If the queue is not full, increment the rear pointer to point to the next available empty space.
- Step 4: Add the data element to the queue location where the rear is pointing.
- Step 5: Here, you have successfully added 7, 2, and -9.



Dequeue() Operation

Obtaining data from the queue comprises two subtasks: access the data where the front is pointing and remove the data after access. You should take the following steps to remove data from the queue -

- Step 1: Check if the queue is empty.
- Step 2: If the queue is empty, Underflow error.
- Step 3: If the queue is not empty, access the data where the front pointer is pointing.
- Step 4: Increment front pointer to point to the next available data element.
- Step 5: Here, you have removed 7, 2, and -9 from the queue data structure.



Now that you have dealt with the operations that allow manipulation of data entities, you will encounter supportive functions of the queues -

Peek() Operation

This function helps in extracting the data element where the front is pointing without removing it from the queue. The algorithm of Peek() function is as follows-

- Step 1: Check if the queue is empty.
- Step 2: If the queue is empty, return “Queue is Empty.”
- Step 3: If the queue is not empty, access the data where the front pointer is pointing.
- Step 4: Return data.

isFull() Operation

This function checks if the rear pointer is reached at MAXSIZE to determine that the queue is full. The following steps are performed in the isFull() operation -

- Step 1: Check if rear == MAXSIZE - 1.
- Step 2: If they are equal, return “Queue is Full.”
- Step 3: If they are not equal, return “Queue is not Full.”

isNull() Operation

The algorithm of the isNull() operation is as follows -

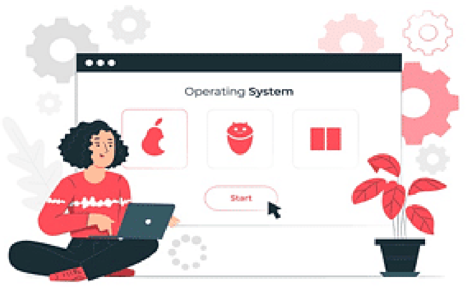
- Step 1: Check if the rear and front are pointing to null memory space, i.e., -1.
- Step 2: If they are pointing to -1, return “Queue is empty.”
- Step 3: If they are not equal, return “Queue is not empty.”

Now that you have covered all the queue operations, you will discover a few applications of the queue.

Applications of Queue

Queue, as the name suggests, is utilized when you need to regulate a group of objects in order. This data structure caters to the need for First Come First Serve problems in different software applications. The scenarios mentioned below are a few systems that use the queue data structure to serve their needs -

Printers and single shared resources



Operating Systems for task scheduling

Switches and Routers

Call Center phone systems

- Printers: Queue data structure is used in printers to maintain the order of pages while printing.
- Interrupt handling in computes: The interrupts are operated in the same order as they arrive, i.e., interrupt which comes first, will be dealt with first.
- Process scheduling in Operating systems: Queues are used to implement round-robin scheduling algorithms in computer systems.
- Switches and Routers: Both switch and router interfaces maintain ingress (inbound) and egress (outbound) queues to store packets.
- Customer service systems: It develops call center phone systems using the concepts of queues.
- CPU scheduling and disk scheduling, where one resource is shared among various consumers

- IO Buffers, files, and Pipes IO, where data is transferred asynchronously between two processes
- Semaphores in the operating system
- FCFS (First Come First Serve) scheduling
- Spooling in printers

- Queue in routers and switches in networking

Differences Between Stack and Queue

Parameter for Comparison	**Stack**	**Queue**
Operational principle	Follows Last In First Out or First In Last Out principle	Follows First In First Out or Last In Last Out principle
Structure	Insertion and deletion both take place from one end called as top	Insertion occurs at the rear end, whereas deletion happens at the front end
Number of pointers required	It contains only one pointer known as top, which stores the reference of the topmost element	It contains two pointers named front (holds the address of the first element in a list) and rear (holds the address of last queue element)
Primary operations	push(x) and pop() are two primary queue operations	Enqueue() and Dequeue() are two primary data manipulation operations
Condition to check empty state	If top == -1, the stack is considered as empty	If front == -1 && rear == -1, the queue is considered as empty
Condition to check full state	If top == MaxSize - 1, the stack is considered as full	If rear == Maxsize - 1, the queue is considered as full
Implementation	Easy implementation	Little complex implementation
Problem Solving	This data structure is used to solve recursive problems.	This data structure is used to solve problems that require sequential processing.

CHAPTER SEVEN

Non Linear Data Structure

Non-linear data structures are those where data items are not arranged in a sequential manner, unlike linear data structures. In these data structures, elements are stored in a hierarchical or a network-based structure that does not follow a sequential order. These data structures allow efficient searching, insertion, and deletion of elements from the structure.

Examples of Non Linear Data Structures:

- Trees
- Graphs, etc.

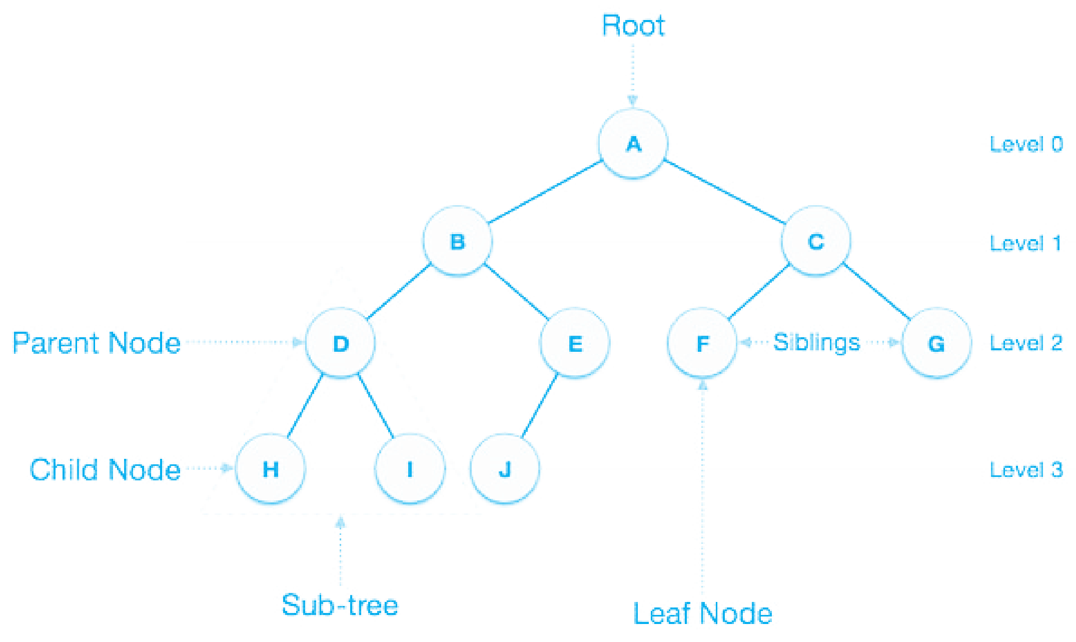
Properties of Non Linear Data Structures

The Non-linear data structures have the following properties.

- Non-linear data structures do not follow a sequential order.
- Elements are stored in a hierarchical or a network-based structure.
- These data structures allow efficient searching, insertion, and deletion of elements.
- Non-linear data structures are used to solve complex problems where data cannot be arranged in a linear manner.

Tree Data Structure

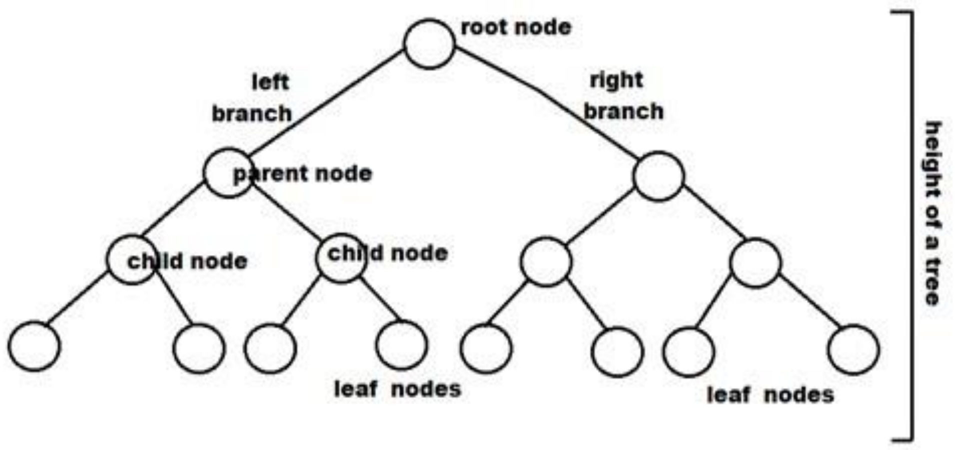
A tree is a non-linear data structure in which data is stored in a hierarchical structure. It is a collection of nodes connected by edges. Each node has a parent node and zero or more child nodes. The node that is present at the top of the hierarchy is called the root node. Trees are widely used in computer science and are an essential part of many algorithms and data structures.



Binary Tree Data Structure

A **binary tree** is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a **binary tree** has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.

A parent node has two child nodes: the left child and right child. Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.



Terminologies associated with Binary Trees and Types of Binary Trees

- **Node:** It represents a termination point in a tree.
- **Root:** A tree's topmost node.
- **Parent:** Each node (apart from the root) in a tree that has at least one sub-node of its own is called a parent node.
- **Child:** A node that straightway came from a parent node when moving away from the root is the child node.
- **Leaf Node:** These are external nodes. They are the nodes that have no child.
- **Internal Node:** As the name suggests, these are inner nodes with at least one child.
- **Depth of a Tree:** The number of edges from the tree's node to the root is.
- **Height of a Tree:** It is the number of edges from the node to the deepest leaf. The tree height is also considered the root height.

Understanding Properties of Binary Tree Or What Is Binary Tree?

At every level of it, the maximum number allowed for nodes stands at 2i.

The height of a binary tree stands defined as the longest path emanating from a root node to the tree's leaf node.

What Is Binary Tree– More Than The Binary Tree Definition

Say a binary tree placed at a height equal to 3. In that case, the highest number of nodes for this height 3 stands equal to 15, that is, (1+2+4+8) = 15. In basic terms, the maximum node number possible for this height h is $(2^0 + 2^1 + 2^2+\ldots.2^h) = 2^{h+1} -1$.

Now, for the minimum node number that is possible at this height h, it comes as equal to h+1.

If there are a minimum number of nodes, then the height of a binary tree would stand aa maximum. On the other hand, when there is a number of a node at its maximum, then the binary tree m height will be minimum. If there exists around 'n' number nodes in a binary tree, here is a calculation to clarify the binary tree definition.

The tree's minimum height is computed as:

n = 2h+1 -1

n+1 = 2h+1

Taking log for both sides now,

log2(n+1) = log2(2h+1)

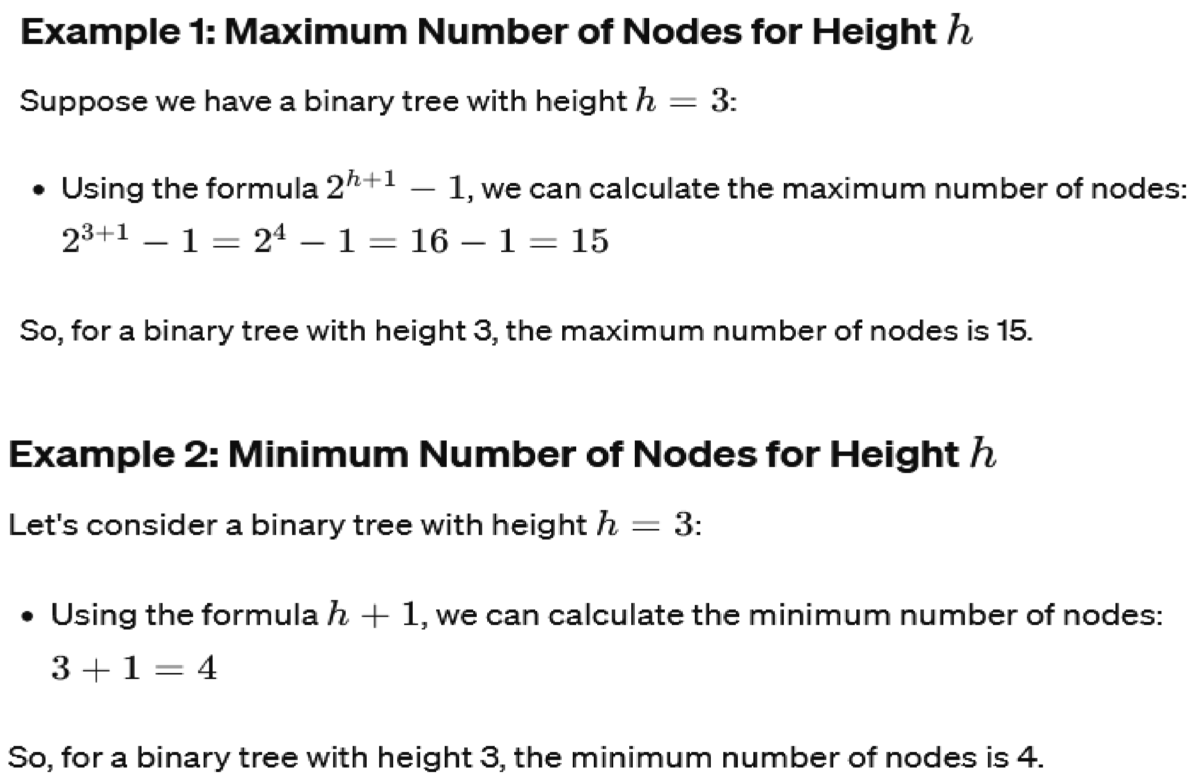
log2(n+1) = h+1

h = log2(n+1) – 1

The highest height will be computed as:

n = h+1

h= n-1



Example 1: Maximum Number of Nodes for Height h

Suppose we have a binary tree with height $h = 3$:

- Using the formula $2^{h+1} - 1$, we can calculate the maximum number of nodes: $2^{3+1} - 1 = 2^4 - 1 = 16 - 1 = 15$

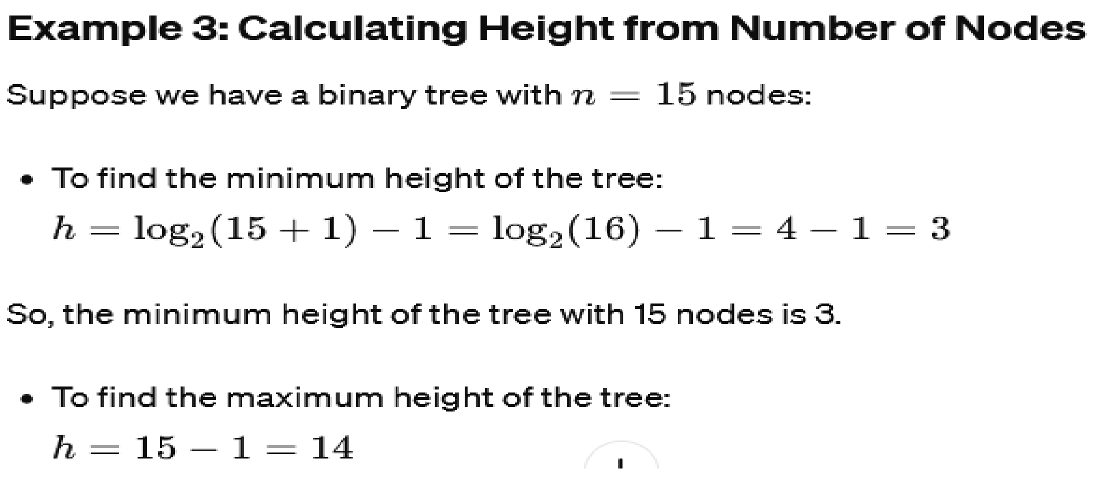
So, for a binary tree with height 3, the maximum number of nodes is 15.

Example 2: Minimum Number of Nodes for Height h

Let's consider a binary tree with height $h = 3$:

- Using the formula $h + 1$, we can calculate the minimum number of nodes: $3 + 1 = 4$

So, for a binary tree with height 3, the minimum number of nodes is 4.



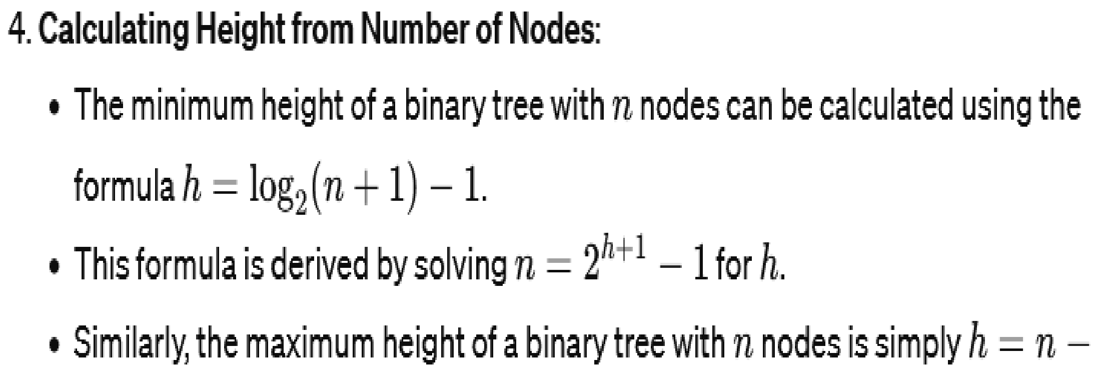
Example 3: Calculating Height from Number of Nodes

Suppose we have a binary tree with $n = 15$ nodes:

- To find the minimum height of the tree:
  $h = \log_2(15 + 1) - 1 = \log_2(16) - 1 = 4 - 1 = 3$

So, the minimum height of the tree with 15 nodes is 3.

- To find the maximum height of the tree:
  $h = 15 - 1 = 14$



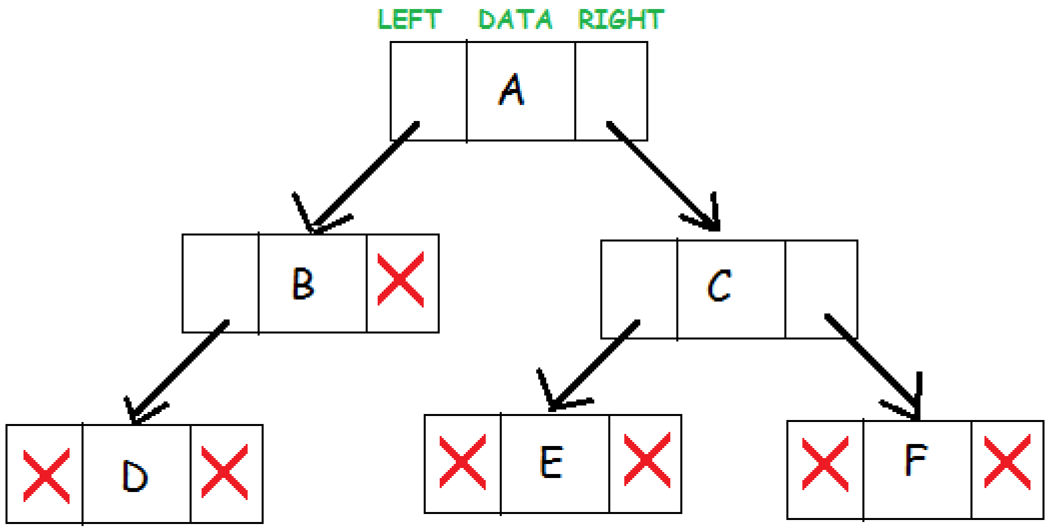
4. Calculating Height from Number of Nodes:

- The minimum height of a binary tree with n nodes can be calculated using the formula $h = \log_2(n + 1) - 1$.
- This formula is derived by solving $n = 2^{h+1} - 1$ for h.
- Similarly, the maximum height of a binary tree with n nodes is simply $h = n -$

Binary Tree Components

There are three **binary tree components**. Every **binary tree** node has these three components associated with it. It becomes an essential concept for programmers to understand these three **binary tree components:**

1. Data element
2. Pointer to left subtree
3. Pointer to right subtree



These three **binary tree components** represent a node. The data resides in the middle. The left pointer points to the child node, forming the left sub-tree. The right pointer points to the child node at its right, creating the right subtree. .

Types of Binary Trees

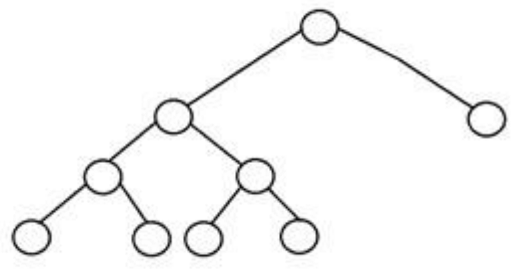
There are various **types of binary trees**, and each of these **binary tree types** has unique characteristics. Here are each of the **binary tree types** in detail:

1. Full Binary Tree

It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

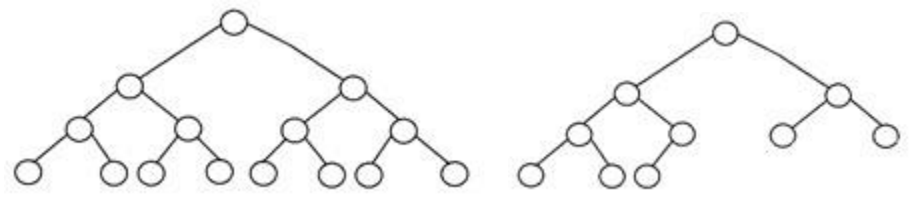
In other words, a full binary tree is a unique binary tree where every node except the external node has two children. When it holds a single child, such a binary tree will not be a full binary tree. *Here, the quantity of leaf nodes is equal to the number of internal nodes plus one. The equation is like* L=I+1, where L is the number of leaf nodes, and I is the number of internal nodes.

Here is the structure of a full binary tree:



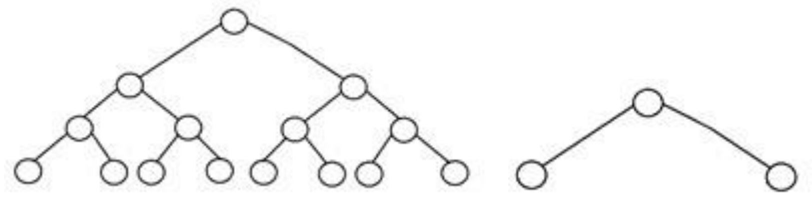
2. Complete Binary Tree

A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side. Here is the structure of a complete binary tree:



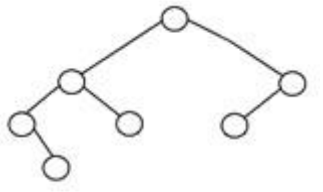
3. Perfect Binary Tree

A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height 'h' has 2h – 1 node. Here is the structure of a perfect binary tree:



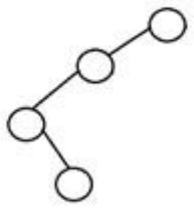
4. Balanced Binary Tree

A binary tree is said to be 'balanced' if the tree height is O(logN), where 'N' is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree. Here is an example of a balanced binary tree:



5. Degenerate Binary Tree

A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child. Such trees are similar to a linked list performance-wise. Here is an example of a degenerate binary tree:



Benefits of a Binary Tree

- The search operation in a binary tree is faster as compared to other trees
- Only two traversals are enough to provide the elements in sorted order
- It is easy to pick up the maximum and minimum elements
- Graph traversal also uses binary trees
- Converting different postfix and prefix expressions are possible using binary trees

Special Types of Binary Trees

Binary trees can also be grouped according to node values. The **types of binary tree** according to node structure include the following:

- **Binary Search Tree**

A binary search tree comes with the following properties:

- In the left subtree of any node, you will find nodes with keys smaller than the node’s key.
- The right subtree of any node will include nodes with keys larger than the node’s key.
- The left, as well as the right subtree, will be **types of binary search tree**.

Binary Search Tree: It is a special type of binary tree in which the value of the left node is always smaller than the root node which is always smaller than the right node.

Binary Search Tree (BST):

A Binary Search Tree is a hierarchical data structure that follows the principles of binary search. Here's a breakdown:

1. **Features**:
   - **Ordered Structure**: In a BST, each node has a key, and keys are arranged in a specific order (e.g., in an ascending order from left to right).
   - **Efficient Searching**: Due to its ordered structure, BSTs enable efficient searching, insertion, and deletion operations. The time complexity for these operations is O(log n) on average and O(n) in the worst case.
   - **Recursive Definition**: Each subtree of a BST is also a BST, which makes it convenient for recursive algorithms.
2. **Implementation**:
   - Each node in a BST contains a key (value) and references to its left and right children.
   - The key of each node must be greater than all keys in its left subtree and less than all keys in its right subtree.
   - Insertions and deletions in a BST require maintaining the BST properties by adjusting the structure as needed (e.g., through rotations).
3. **Example**: Let's say we have the following keys to be inserted into a BST: 50, 30, 70, 20, 40, 60, 80.

   Inserting them in the order mentioned will create a BST where each node's left child has a smaller value, and each node's right child has a larger value.

   The resulting BST might look like this:

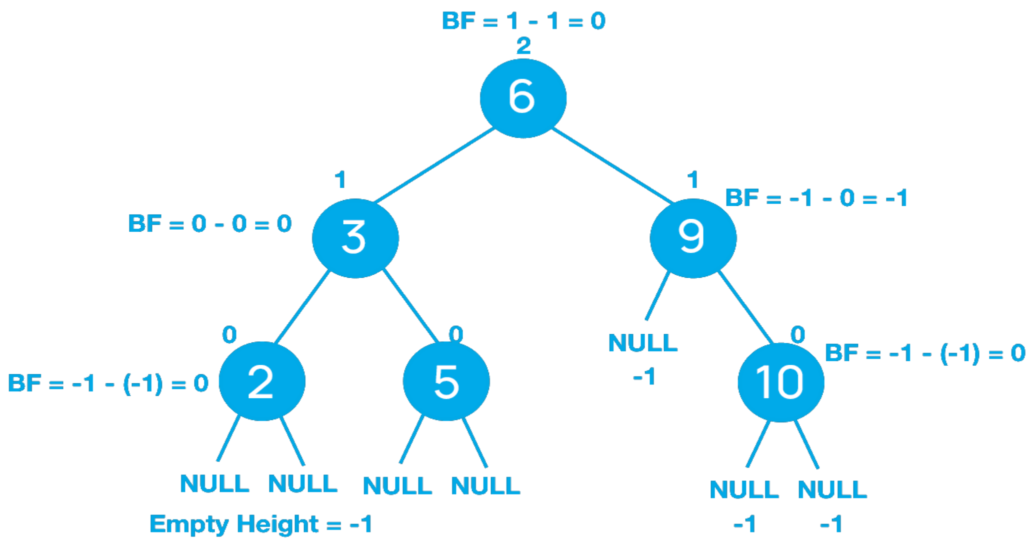
50

/ \

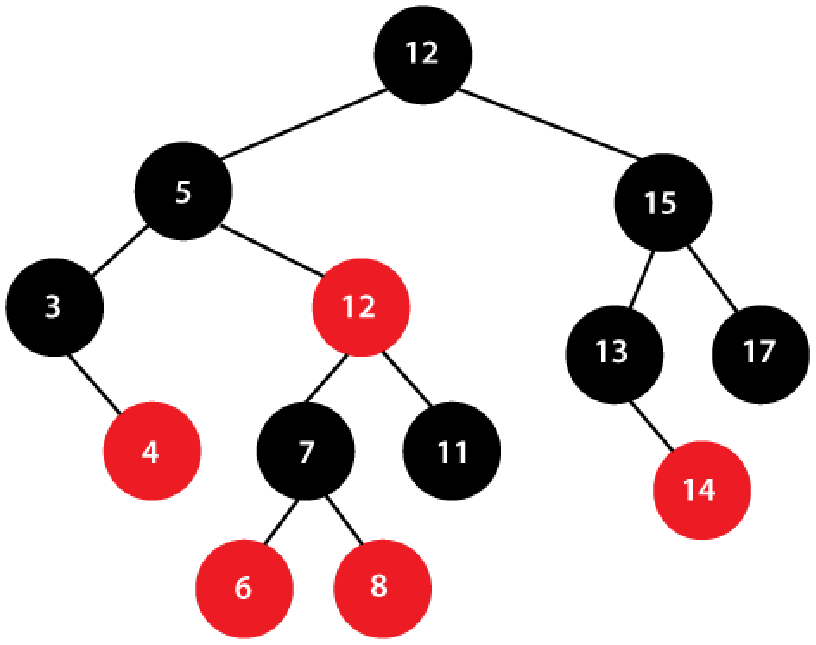
- 30 70
- / \ /

20 40 60 80

AVL Tree: An AVL binary tree in DSA is self-balanced. In such a tree, the difference between the heights of the left and right subtrees for all nodes cannot be greater than one. So, the nodes in the right as well as left subtrees of the AVL tree will be one or less than that. An AVL tree is a self-balancing binary search tree. The balancing factor is either 1, -1, or 0.



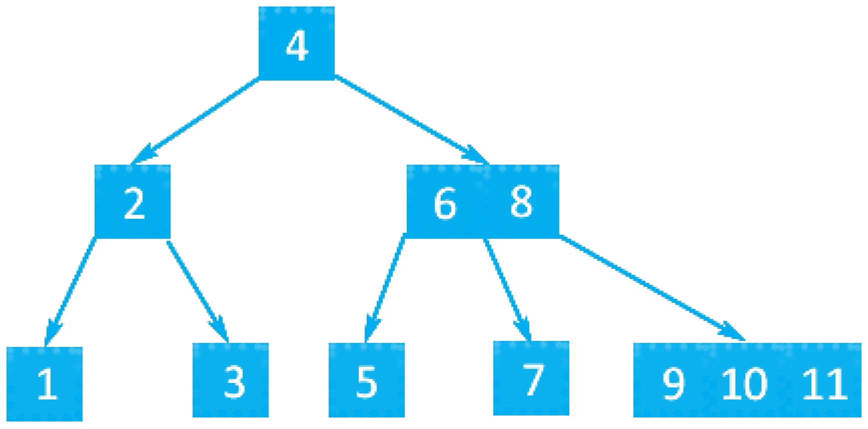
Red-black Tree: A red-black tree is a type of binary tree that is self-balanced and each node in it is either colored either red or black. The colors in a red black binary tree in data structure are useful for keeping the whole tree balanced during deletions and insertions. The balance of a red black tree won't be perfect. But these binary trees are perfect for bringing down the search time.



Red Black Tree

- **B-tree:** A B-tree is a self-balancing tree data structure that is commonly used in file systems and DBMS as they facilitate the fast feature of fast searching.

B-TREE



B- Tree

A B- Tree is a type of self-balanced search tree in data structures. These binary trees support smooth access, deletion, and insertion of data items. B- trees are particularly common in file systems and databases.

Among the different types of binary tree, a B- tree helps with efficient storage and retrieval of large volumes of data. A fixed maximum degree or order is a key characteristic of a B- tree. This fixed value helps determine the total number of child nodes in a parent node.

The nodes present in a B- binary tree can include several keys and child nodes. The keys of a B- **binary tree in algorithm design** can help in indexing and locating data items.

B+ Tree

A **binary tree in data structure** can also be classified as B+, which is one variant of the B- tree. Since a B+ tree comes with a fixed maximum degree, it enables efficient insertion, access, and deletion of data items. But a B+ binary tree includes all data items inside the leaf nodes.

The internal nodes of a B+ binary tree only include keys for locating and indexing data items. Due to this design, searches using a B+ tree will be a lot faster, and you will also be able to access data items sequentially. Moreover, the leaf nodes of a binary tree remain together in a linked list.

Segment Tree

If you look into a binary tree and its types, you will come across one category called the segment or statistic tree. This type of binary tree is usually responsible for storing information related to different segments or intervals. With a segment tree, you will be able to perform querying of the stored segments in a specific point.

Among the different types of binary tree in data structure, you will realize that a segment tree is static. Therefore, you won’t be able to modify the structure of a segment tree after it has been built.

Applications of Binary Tree in Data Structure

If you want to read more on binary tree in data structure**,** you should learn about their applications. Binary search trees are quite suitable for the following purposes:

- **Search Algorithms:** An algorithm for binary search tree can efficiently find a specific element. The search can be executed in O u(log n) time complexity, where n defines the number of nodes. A binary search tree is often useful for quickly finding particular elements in a sorted list.
- **Database Systems:** With each node of a binary tree representing a record, data can be stored in a database system. As a result, search operations may be completed quickly, and the database system can manage massive volumes of data.
- **Decision Trees:** Binary trees are a sort of machine learning technique that may be used to create decision trees. These decision trees are highly useful for regression analysis and classification.
- **File Systems:** File systems can be implemented using binary trees, in which every node corresponds to a directory or file. This enables quick and easy file system browsing and searching.
- **Compression Algorithms:** An algorithm for binary search tree in data structure can be useful for Huffman coding. A compression algorithm is responsible for assigning variable-length codes to characters according to their occurrence frequency in the input data.
- **Game AI:** Game AI can be implemented using binary trees, where every node indicates a potential move in the game. The optimal move can be found by the AI algorithm searching the tree.
- **Sorting Algorithms:** An algorithm of binary tree can also be used for efficient sorting. For instance, the search tree sort and heap sort are quite beneficial.

Why Should You Use a Binary Tree in Data Structure?

Once you learn about a binary tree and its types, you should try to figure out the benefits of these structures. Some key advantages of using a binary tree model include:

- **Ordered Traversal:** Binary trees are structured in such a way that you will succeed in traversing them in a particular order, such as post-order, in-order, and pre-order. As a result, you will succeed in performing operations on the nodes in a particular order. For instance, you will be able to easily print nodes in a sorted order.
- **Efficient Searching:** A binary tree in data structure can be efficiently used to find a particular element. Each node comes with a maximum of two child nodes. So, search operations can be easily performed with the O(log n) time complexity.
- **Fast Insertion and Deletion:** Insertions and deletions can be done with binary trees in O(log n) time complexity. They are also a wise option for applications like database systems that need dynamic data structures.
- **Memory Efficient:** Since binary trees only need two child pointers per node, they are comparatively memory-efficient when compared to other tree designs. This implies that they can be utilized to maintain effective search functions even when storing substantial volumes of data in memory.
- **Valuable for Sorting:** If you understand the binary tree terminology in data structure, you will realize that it is extremely efficient for sorting. Therefore, you will find binary trees to be highly beneficial for heap sort and similar operations.
- **Easy to Implement:** It is quite simple and easy to understand and implement binary tree structures. That’s why binary tree algorithms are highly suitable for a large number of real-life applications.

Disadvantages of Binary Tree Structures

While a **binary tree in data structure** is highly beneficial, it also has some shortcomings. A few reasons why binary trees might not be beneficial include:

- **Limited Structure:** Every binary tree comes with a maximum of two children in each node. While it is a boon in many ways and makes these structures memory efficient, it is also a disadvantage. Due to their limited structure, binary trees cannot be used in certain cases. For instance, some trees require each node to have more than two children. In that case, a different tree format needs to be used in a data structure.
- **Space Inefficiency:** Binary trees are not as space efficient as some other types of data structures. Every node needs two child pointers. So, if it’s a large binary tree, a significant amount of memory will be required.

- **Slow Performances:** Binary trees are responsible for extremely slow performances in the worst-case scenarios. The worst-case scenario might even degenerate a binary tree. If that happens, every node will end up with just one child instead of two. As a result, search operations will degrade.
- **Unbalanced Trees:** In an unbalanced binary tree, one subtree appears to be considerably larger than the other. This difference can easily render search operations inefficient. The difference is even more prominent when the tree isn't properly balanced, or data is inserted within it in a non-random manner.
- **Complex Balancing Algorithms:** Several balancing algorithms can be used to keep a binary tree balanced. But these algorithms are extremely difficult to implement. Some of these algorithms also demand extra overhead, which makes them incapable of certain applications.

Operations to Perform on a Binary Tree

Some basic operations that can be implemented on a binary tree include:

- Insertion of an element
- Removal of an element
- Looking for an element
- Deletion of an element
- Traversing an element (You can perform four types of traversals in a binary tree structure.)

A binary tree is also suitable for performing a host of auxiliary operations. Some auxiliary operations to implement on a binary tree include:

- Detecting the height of the tree
- Figuring out the level of the tree
- Determining the right size of the whole tree

Tree Traversal Techniques – Data Structure and Algorithm Tutorials

Tree Traversal techniques include various ways to visit all the nodes of the tree. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

Tree Traversal Meaning:

Tree Traversal refers to the process of visiting or accessing each node of the tree exactly once in a certain order. Tree traversal algorithms help us to visit and process all the nodes of the tree. Since tree is not a linear data structure, there are multiple nodes which we can visit after visiting a certain node. There are multiple tree traversal techniques which decide the order in which the nodes of the tree are to be visited.

Types of Tree Traversal in Data Structure

You can perform tree traversal in the data structure in numerous ways, unlike arrays, linked lists, and other linear data structures, which are canonically traversed in linear order. You can walk through them in either a depth-first search or a breadth-first search. In-order, pre-order, and post-order are the three most frequent ways to go over them in depth-first.

Beyond these fundamental traversals, more complex or hybrid techniques, such as depth-limited searches, such as iterative deepening depth-first search, are feasible.

Depth First Search

The depth-first search (DFS) is a method of traversing or searching data structures composed of trees or graphs.

To get to the recursion, go down one level. If the tree isn't empty, perform the three procedures below in a specific order:

Left: Recursively traversed left subtree

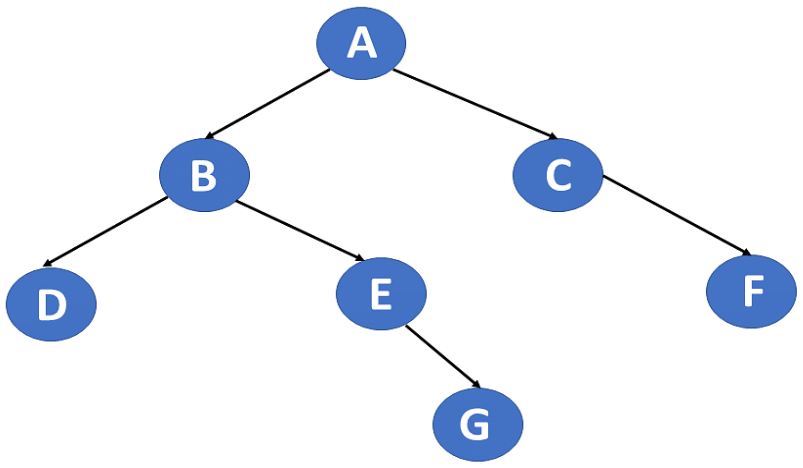
Right: Recursively traversed right subtree

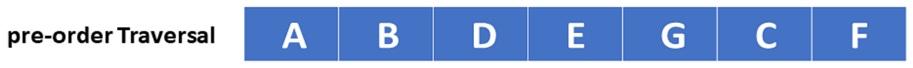
Node: Traverse the root node

Three orders execute tree traversal under this depth-first search algorithm.

Pre-Order Traversal (Root-Left-Right)

1. Traverse root node
2. Traverse left subtree
3. Traverse right subtree



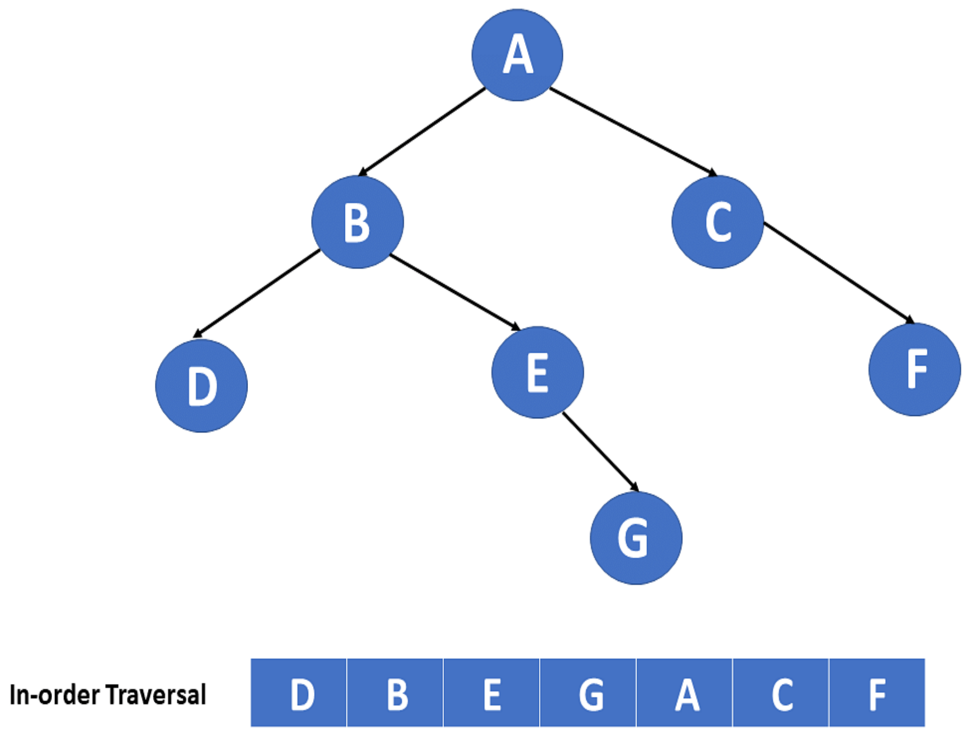


Uses of pre-order traversal

To duplicate the tree, you need to use pre-order traversal. Pre-order traversal is used to obtain a prefix expression from an expression tree.

In-Order Traversal (Left-Root-Right)

1. Traverse left subtree
2. Traverse root node
3. Traverse right subtree

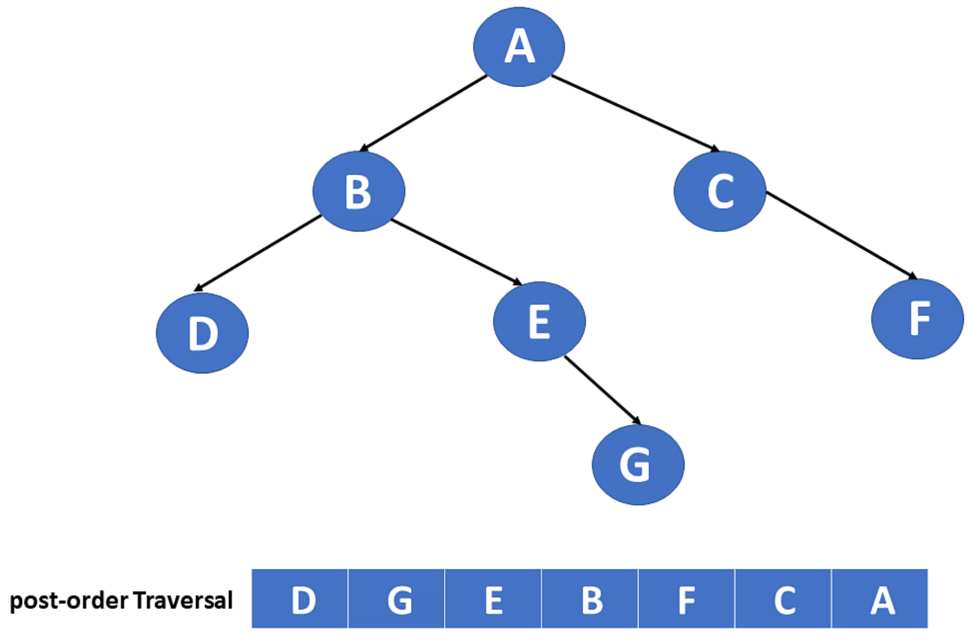


Uses of in-order traversal

Inorder traversal gives nodes in non-decreasing order in binary search trees or BST. A version of Inorder traversal with Inorder traversal reverse can access BST nodes in non-increasing order.

Post-Order Traversal (Left-Right-Root)

1. Traverse left subtree
2. Tereveser right subtree
3. Traverse root node



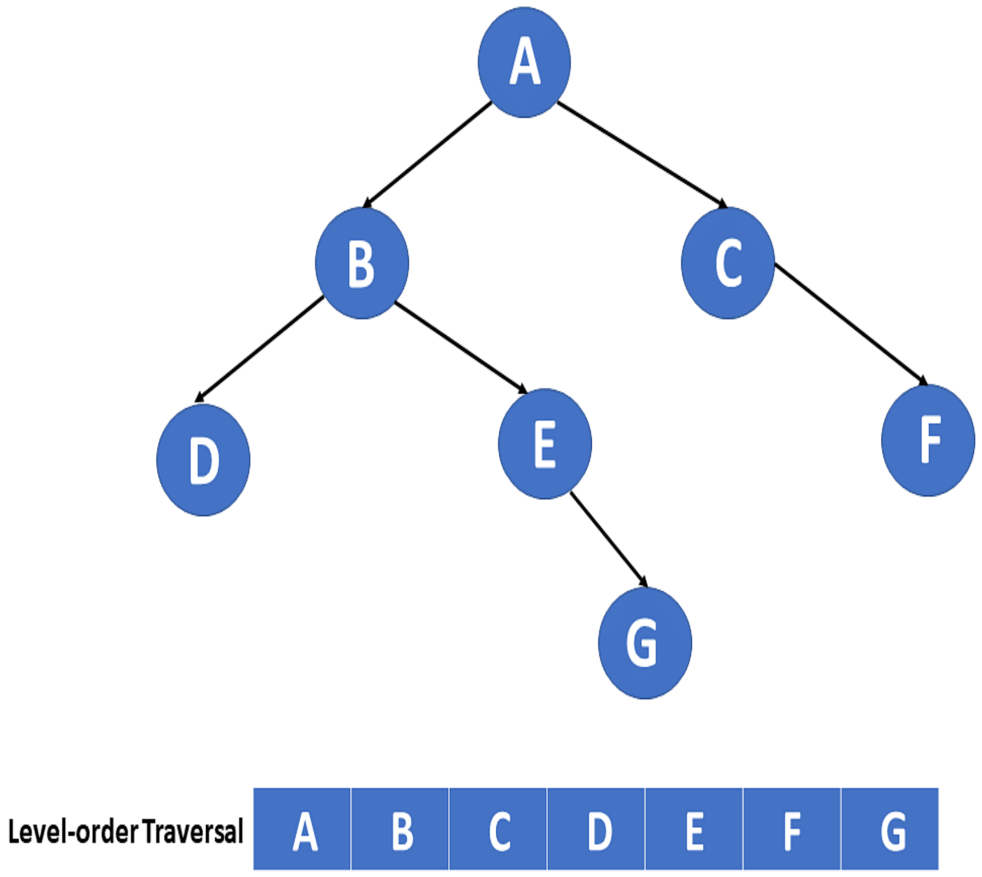
Uses of in-order traversal

You can delete the tree using post-order traversal. The postfix expression of tree data structure can also be obtained using post-order traversal.

Depth First Search or Level Order Traversal

BFS (Breadth-First Search) is a vertex-based method for determining the shortest path in a graph. It employs a queue data structure, where first in, it follows first out. In BFS, it visits one vertex and marks it at the time, after which it sees and queues its neighbors.

After understanding the types of tree traversal in data structures, you will now examine how to implement tree traversal in data structures.



Implementations of Tree Traversal in Data Structure

Pre-Order Traversal

```
preorder_traversal(node)

    if (node == null)

        return

    visit(node)

    preorder_traversal(node.left_subtree)
    preorder_traversal(node.right_subtree)
```

In-Order Traversal

```
inorder_traversal(node)

    if (node == null)

        return

    inorder_traversal(node.left_subtree)

    visit(node)

    inorder_traversal(node.right_subtree)
```

In-Order Traversal

```
postorder_traversal(node)

    if (node == null)

        return

    postorder_traversal(node.left_subtree)

    postorder_traversal(node.right_subtree)

    visit(node)
```

Code of implementing all three tree traversal in data structure

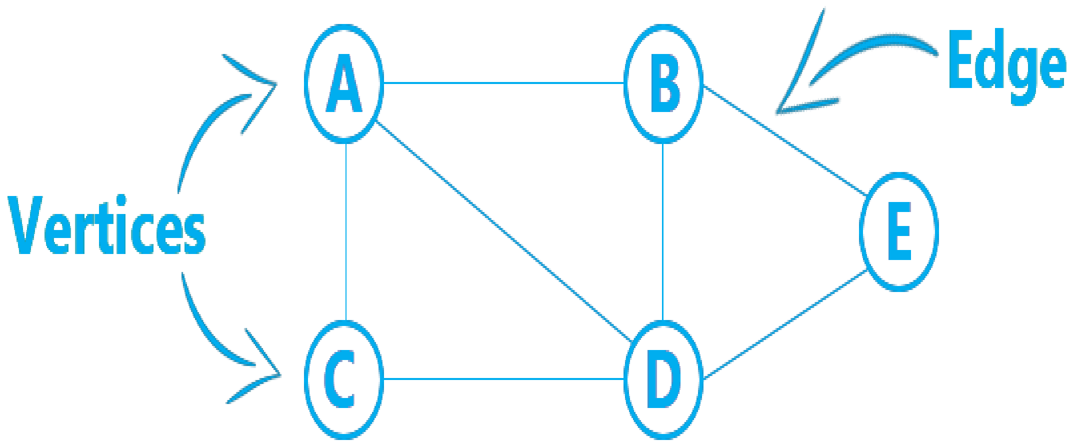
```
#include <stdio.h>

#include<conio.h>

#include <stdlib.h>

enum Traversal {Preorder_Traversal, Inorder_Traversal, Postorder_Traversal};
```

```
typedef enum Traversal trav;

typedef struct Node Node;

struct Node {

    int x;

    Node* left_node, *right_node;

};

Node* create_tree(int data) {                          //creating tree

    Node* root = (Node*) malloc (sizeof(Node));

    root->left_node= root->right_node = NULL;

    root->x = data;

    return root;

}

Node* create_node(int data) {                        //creating node

    Node* node = (Node*) malloc (sizeof(Node));

    node->x = data;

    node->left_node = node->right_node = NULL;

    return node;

}

void release_tree(Node* root) {                      //releasing tree
```

```
Node* t = root;

if (!t)

    return;

release_tree(t->left_node);

release_tree(t->right_node);

if (!t->left_node && !t->right_node) {
```

Graph Data Structure

A graph is a non linear data structure that consists of a set of vertices and a set of edges that connect them together. In a graph, vertices represent entities, while edges represent the relationships between them. Graphs are used to represent complex relationships between entities and are widely used in computer science.



Terminologies Related to Graphs

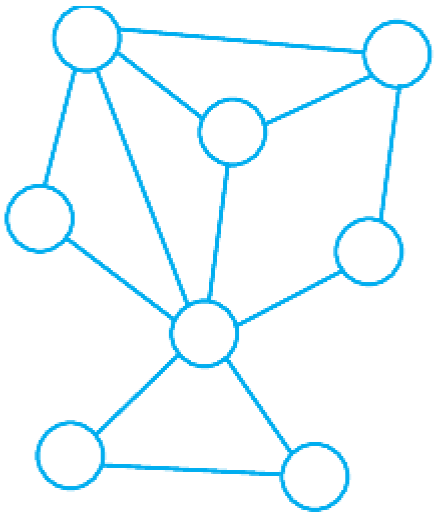
Here are some terminologies related to graphs.

- **Vertex:** A vertex is a data item that is stored in a graph data structure.
- **Edge:** An edge is a connection between two vertices in a graph.
- **Degree:** The degree of a vertex in a graph data structure is defined as the number of edges connected to it.
- **Weight:** The weight of an edge is a numerical value that is assigned to the edge to represent the cost or distance between the vertices.
- **Path:** A path is a sequence of edges that connects two vertices in a graph.
- **Cycle:** A cycle is a path in a graph that starts and ends at the same vertex.

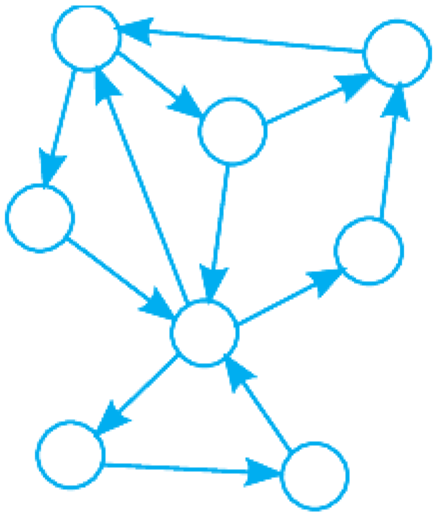
Types of Graph

Graphs are of the following different types.

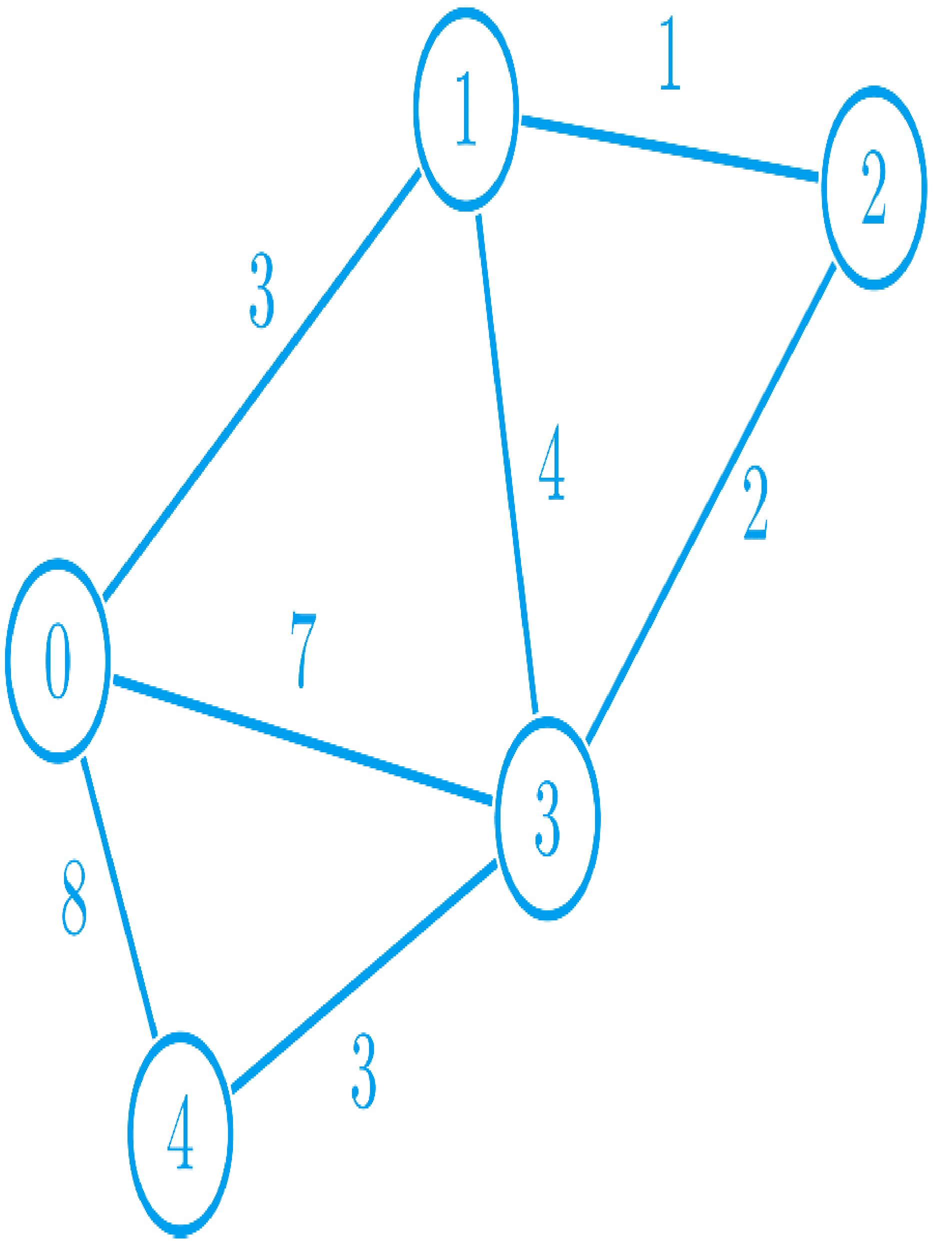
- **Undirected Graph:** In an undirected graph, edges do not have a direction. That is, the edge connecting vertex A to vertex B is the same as the edge connecting vertex B to vertex A.



- **Directed Graph:** In the case of a directed graph, the edges have a direction that means, the edge connecting vertex A to vertex B is different from the edge connecting vertex B to vertex A.



- **Weighted Graph:** In a weighted graph, edges have weights that represent the cost or distance between the vertices.


0
1
2
3
4
1
3
4
2
7
8
3

	Linear Data structure	**Non-Linear Data structure**
Basic	In this structure, the elements are arranged sequentially or linearly and attached to one another.	In this structure, the elements are arranged hierarchically or non-linear manner.
Types	Arrays, linked list, stack, queue are the types of a linear data structure.	Trees and graphs are the types of a non-linear data structure.
implementation	Due to the linear organization, they are easy to implement.	Due to the non-linear organization, they are difficult to implement.
Traversal	As linear data structure is a single level, so it requires a single run to traverse each data item.	The data items in a non-linear data structure cannot be accessed in a single run. It requires multiple runs to be traversed.
Arrangement	Each data item is attached to the previous and next items.	Each item is attached to many other items.
Levels	This data structure does not contain any hierarchy, and all the data elements are organized in a single level.	In this, the data elements are arranged in multiple levels.
Memory utilization	In this, the memory utilization is not efficient.	In this, memory is utilized in a very efficient manner.
Time complexity	The time complexity of linear data structure increases with the increase in the input size.	The time complexity of non-linear data structure often remains same with the increase in the input size.
Applications	Linear data structures are mainly used for developing the software.	Non-linear data structures are used in **image processing** and **Artificial Intelligence**.

<u>**Recommended Textbooks:**</u>

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. "Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss
3. "Data Structures and Algorithms in Java" by Robert Lafore
4. "Algorithms, Part I and Part II" by Robert Sedgewick and Kevin Wayne
5. "Data Structures and Algorithms in Python" by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
6. "Data Structures and Algorithms Made Easy" by Narasimha Karumanchi
7. "Data Structures and Algorithms Using Python and C++" by David M. Reed and John Zelle
8. "Data Structures and Problem Solving Using Java" by Mark Allen Weiss
9. "The Algorithm Design Manual" by Steven S. Skiena
10. "Data Structures and Algorithms in C++" by Michael T. Goodrich, Roberto Tamassia, and David M. Mount