Pogramme Name: BCA (NEP)

Course Code: NBCA-201

Faculty Name:

Mr. Rohit Kapoor

Assistant Professor, LPCPS

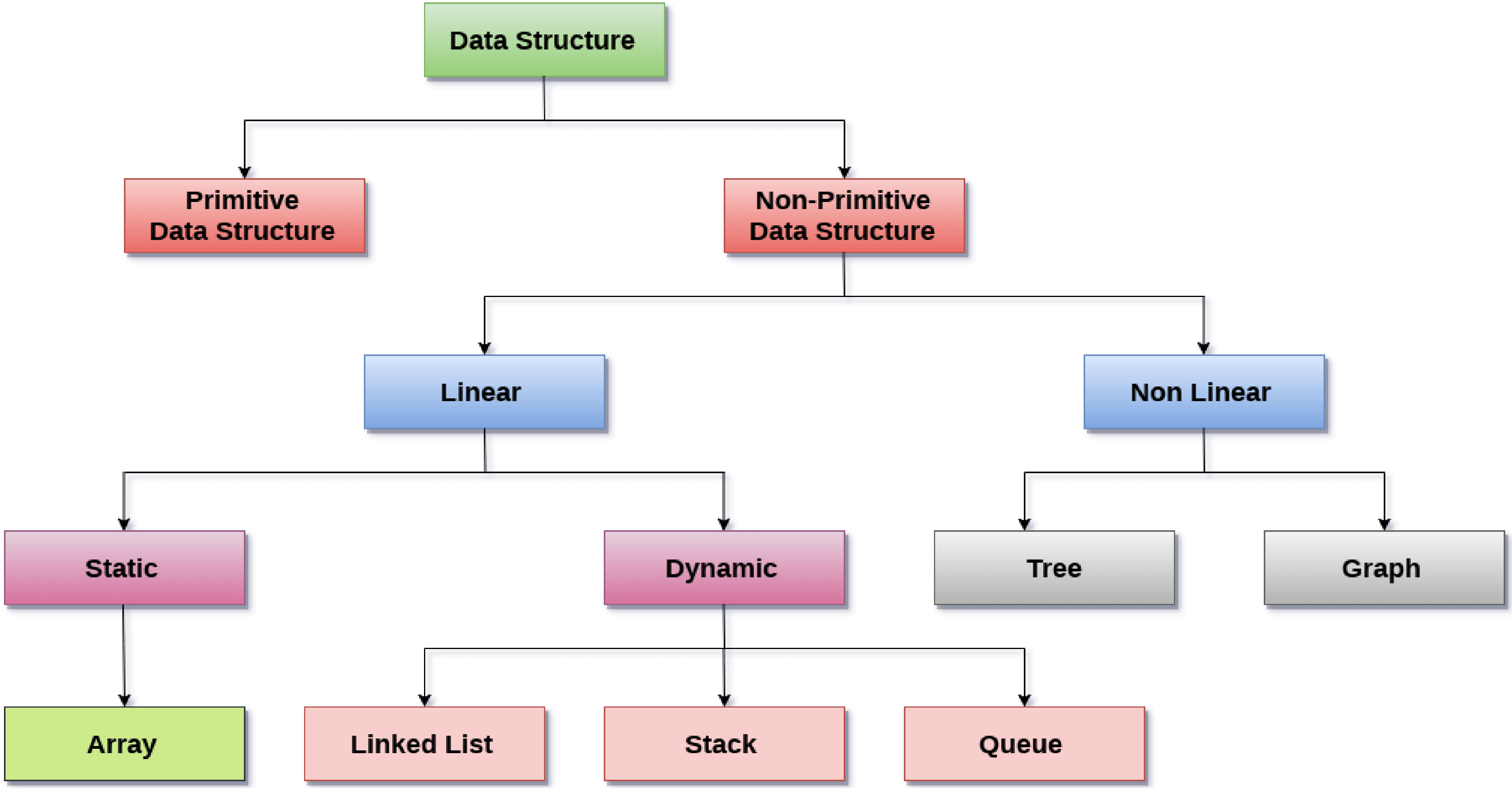
Objectives

- To provide the knowledge of fundamental concepts of data structures using the c programming language so that students should get to know that how we are managing various kinds of data in the computer system and how it is accessed in a proper way.
- Understand the use and working of the various data structures.
- Learn to be able to build own algorithms and pseudo codes for the various applications of the basic data structures.

Introduction

- "Data" indicates information saved or delivered by a computer.
- Data also occurs in order kinds.
- There is data in order types as well, though.
- Data may take many forms: handwritten notes including numbers and words, digital files held in memory of computers and other electronic devices, or even information held in a person's brain.
- This data became an integral part of everyone's daily lives as the globe began to modernise, and different implementations enabled people to store it differently.

F


Data Structure
Primitive Data Structure
Non-Primitive Data Structure
Linear
Non Linear
Static
Dynamic
Tree
Graph
Array
Linked List
Stack
Queue

- **Data** is a collection of facts and figures or a set of values or values of a specific format that refers to a single set of item values.

- The data items are then classified **into sub-items**, which is the group of items that are not known as the simple primary form of the item.

What is Data Structure?

- **Data Structure** is a branch of Computer Science.
- The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program.
- Data Structure is a particular way of **storing and organizing data in the memory of the computer** so that these data can easily be retrieved and efficiently utilized in the future when required.
- The data can be managed in various ways, like **the logical** or **mathematical model** for a specific organization of data is known as a data structure.

Scope

The scope of a particular data model depends on two factors:

1. First, it must be loaded enough into the structure to reflect the definite **correlation of the data with a real-world object**.
2. Second, the formation should be so straightforward that one can **adapt to process the data efficiently whenever necessary**.

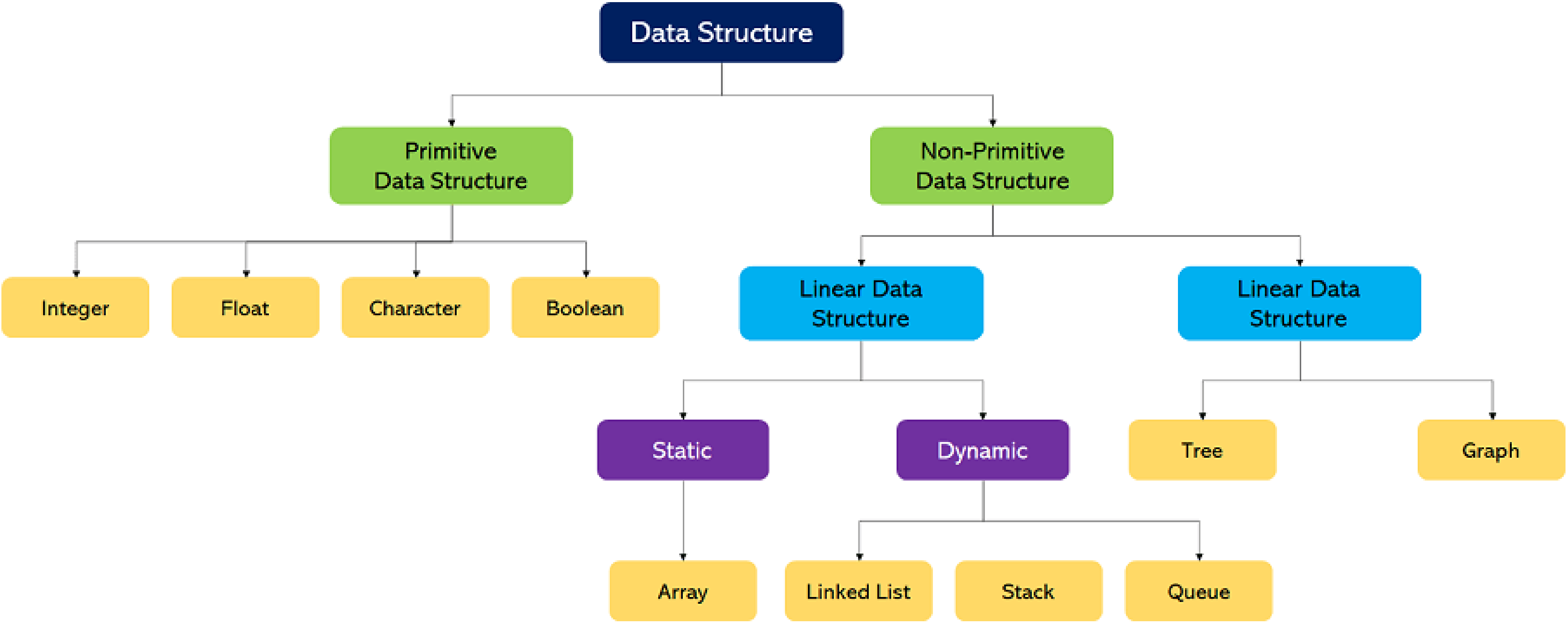
- Some examples of Data Structures are Arrays, Linked Lists, Stack, Queue, Trees, etc. Data Structures are widely used in almost every aspect of Computer Science, i.e., Compiler Design, Operating Systems, Graphics, Artificial Intelligence, and many more.

Basic Terminologies related to Data Structures

- Data Structures are the building blocks of any software or program.

- The following are some fundamental terminologies used whenever the data structures are involved:
  1. **Data:** We can define data as an elementary value or a collection of values. For example, the Employee's name and ID are the data related to the Employee.
  2. **Data Items:** A Single unit of value is known as Data Item.
  3. **Group Items:** Data Items that have subordinate data items are known as Group Items. For example, an employee's name can have a first, middle, and last name.
  4. **Elementary Items:** Data Items that are unable to divide into sub-items are known as Elementary Items. For example, the ID of an Employee.
  5. **Entity and Attribute:** A class of certain objects is represented by an Entity. It consists of different Attributes. Each Attribute symbolizes the specific property of that Entity. For example,

Classification of Data Structures



Primitive Data Structures

1. Primitive Data Structures are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

1. Non-Primitive Data Structures are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
    1. Linear Data Structures
    2. Non-Linear Data Structures

Linear Data Structures

- A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure.
- The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element.
- However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.
- Based on memory allocation, the Linear Data Structures are further classified into two types:

Linear Data Structures

- **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is **allocated at the compiler time**, and their size cannot be changed by the user after being compiled; however data in it can alerted.

  The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.

- **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is **allocated at the run time**, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code.

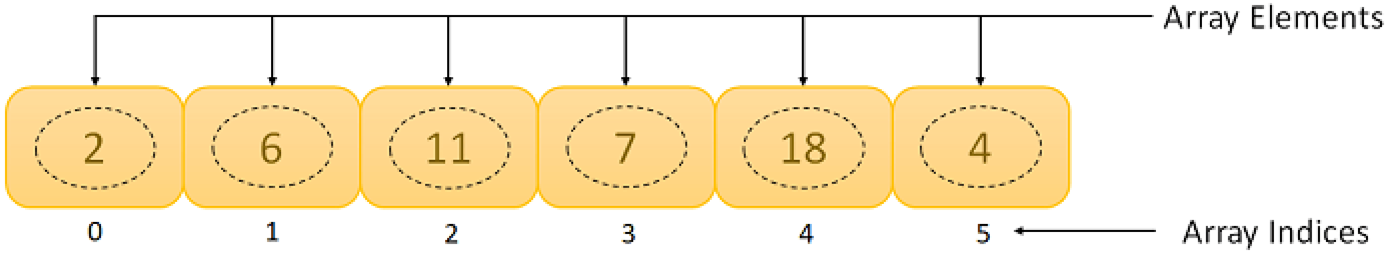
  Linked Lists, Stacks, and **Queues** are common examples of dynamic data structures

Types of Linear Data Structures

Arrays

- An **Array** is a data structure used to collect multiple data elements of the same data type into one variable.

- An Array is a list of elements where each element has a unique place in the list.
  - The data elements of the array share the same variable name; however, each carries a different index number called a subscript.
  - We can access any data element from the list with the help of its location in the list.
  - Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.



Arrays can be classified into different types:

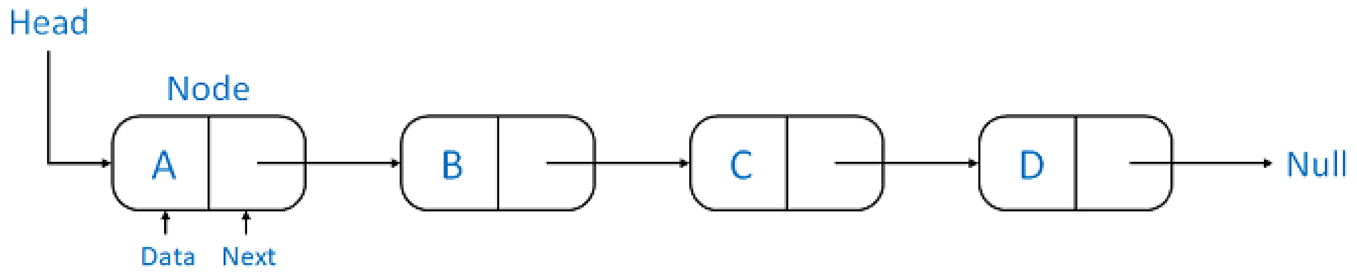
1. **One-Dimensional Array:** An Array with only one row of data elements is known as a One-Dimensional Array. It is stored in ascending storage location.

2. **Two-Dimensional Array:** An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.

3. **Multidimensional Array:** We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.

Linked Lists

- A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically.
- Data elements in this data structure are represented by the Nodes, connected using links or pointers.
- Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list.
- The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

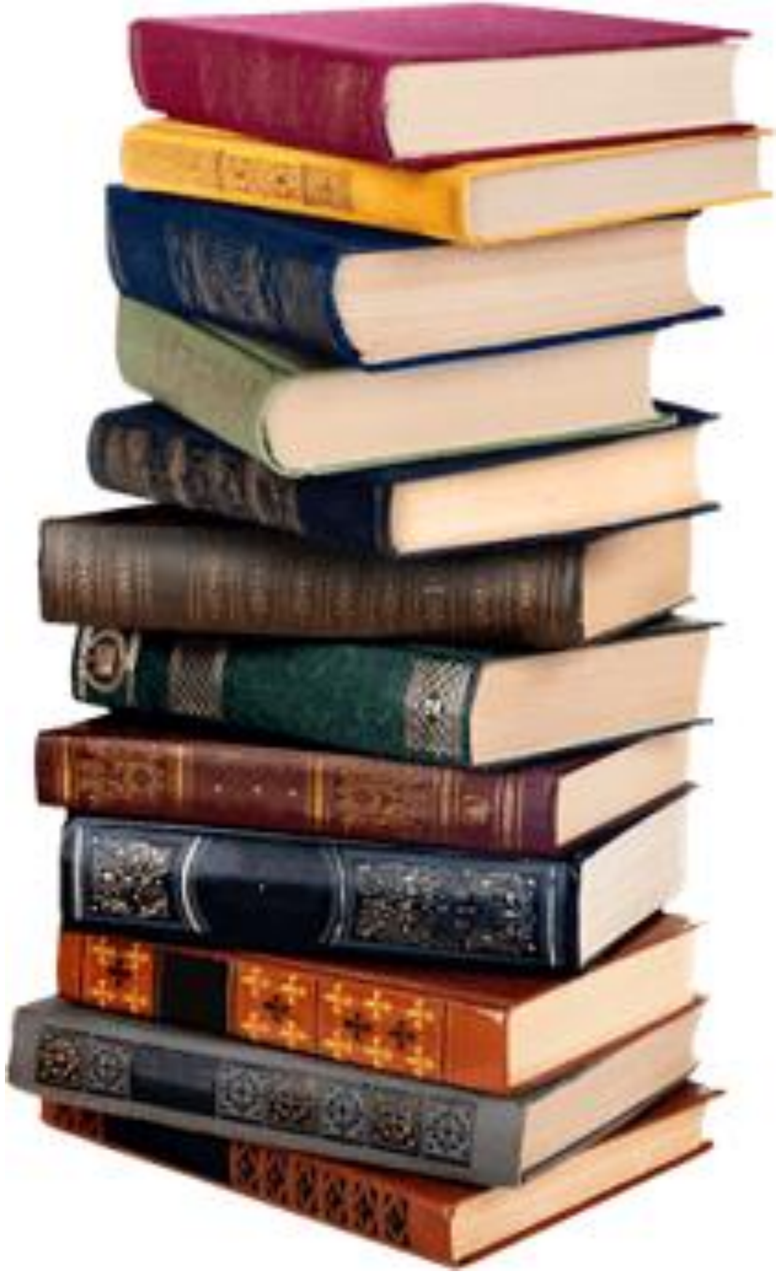


Linked Lists can be classified into different types:

1. **Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.
2. **Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).
3. **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

Some Applications of Linked Lists:

1. The Linked Lists help us implement stacks, queues, binary trees, and graphs of predefined size.
2. We can also implement Operating System's function for dynamic memory management.
3. Linked Lists also allow polynomial implementation for mathematical operations.
4. We can use Circular Linked List to implement Operating Systems or application functions that Round Robin execution of tasks.
5. Circular Linked List is also helpful in a Slide Show where a user requires to go back to the first slide after the last slide is presented.
6. Doubly Linked List is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

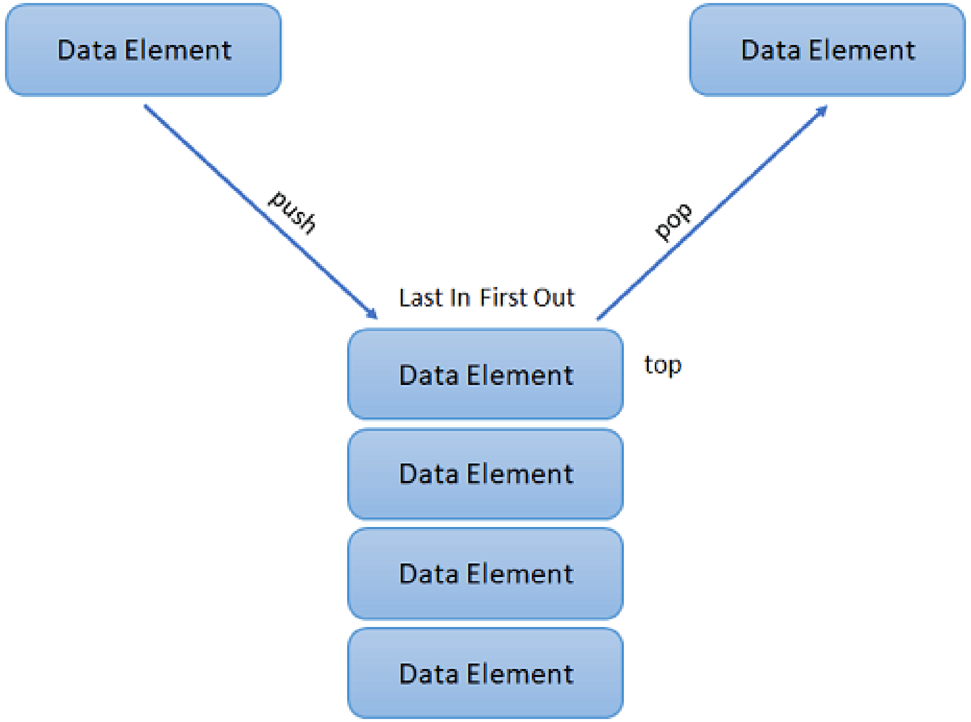


Stacks

- A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top.
- Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List.
- Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

  - **The insertion and removal of new books from the top of the Stack. It implies that the insertion and deletion in the Stack can be done only from the top of the Stack. We can access only the Stack's tops at any given time.**

The primary operations in the Stack are as follows:

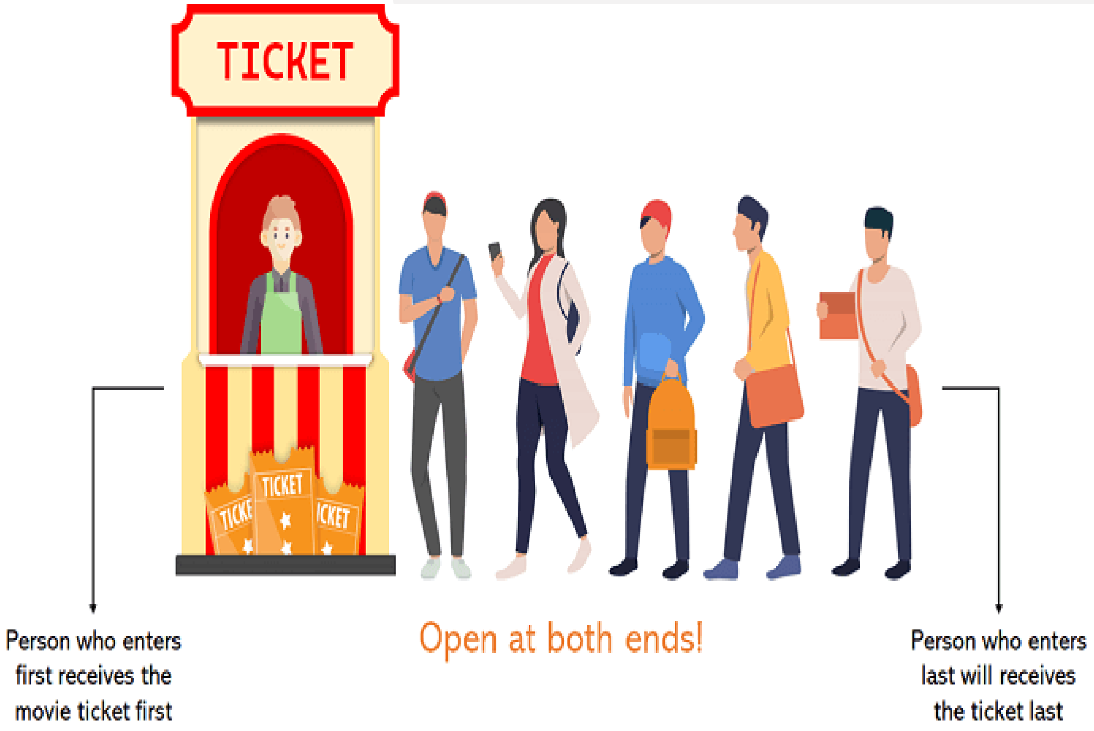


1. **Push:** Operation to insert a new element in the Stack is termed as Push Operation.
2. **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.

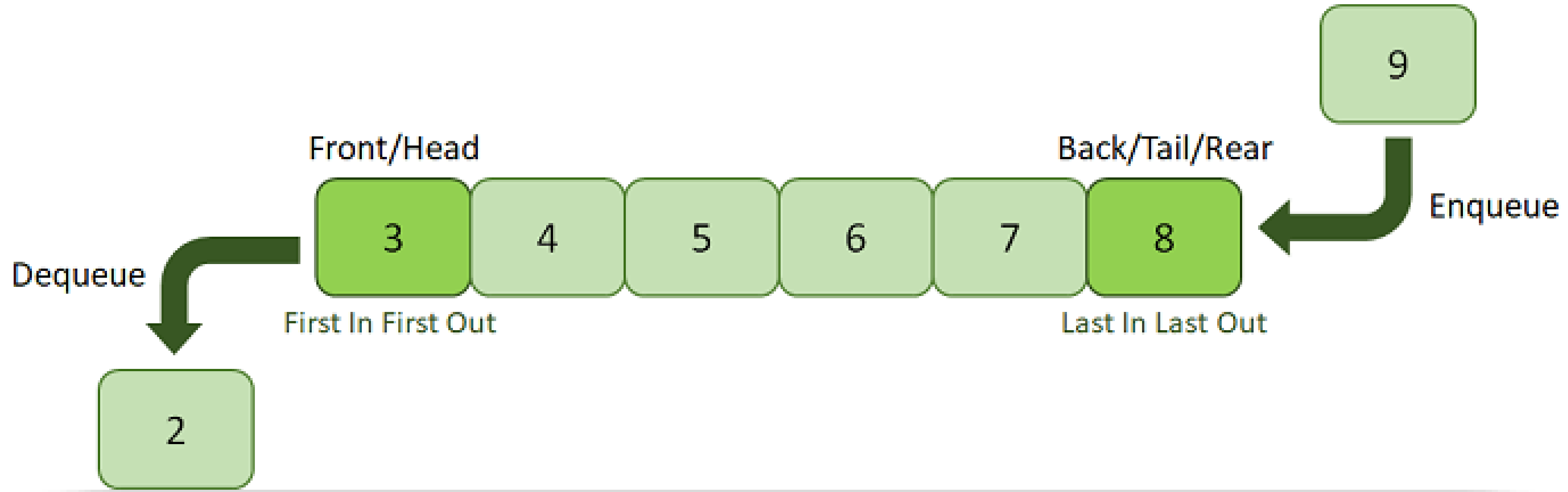
Some Applications of Stacks:

1. The Stack is used as a Temporary Storage Structure for recursive operations.
2. Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.
3. We can manage function calls using Stacks.
4. Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.
5. Stacks are also helpful in converting infix expressions to postfix expressions.
6. Stacks allow us to check the expression's syntax in the programming environment.
7. We can match parenthesis using Stacks.
8. Stacks can be used to reverse a String.
9. Stacks are helpful in solving problems based on backtracking.
10. We can use Stacks in depth-first search in graph and tree traversal.
11. Stacks are also used in Operating System functions.
12. Stacks are also used in UNDO and REDO functions in an edit.

Queues


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

- A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements.
- The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end.
- Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements.
- Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.



The following are the primary operations of the Queue:

1. **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.
2. **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.

Some Applications of Queues:

1. Queues are generally used in the breadth search operation in Graphs.
2. Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.
3. Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.
4. Priority Queues are utilized in file-downloading operations in a browser.
5. Queues are also used to transfer data between peripheral devices and the CPU.
6. Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

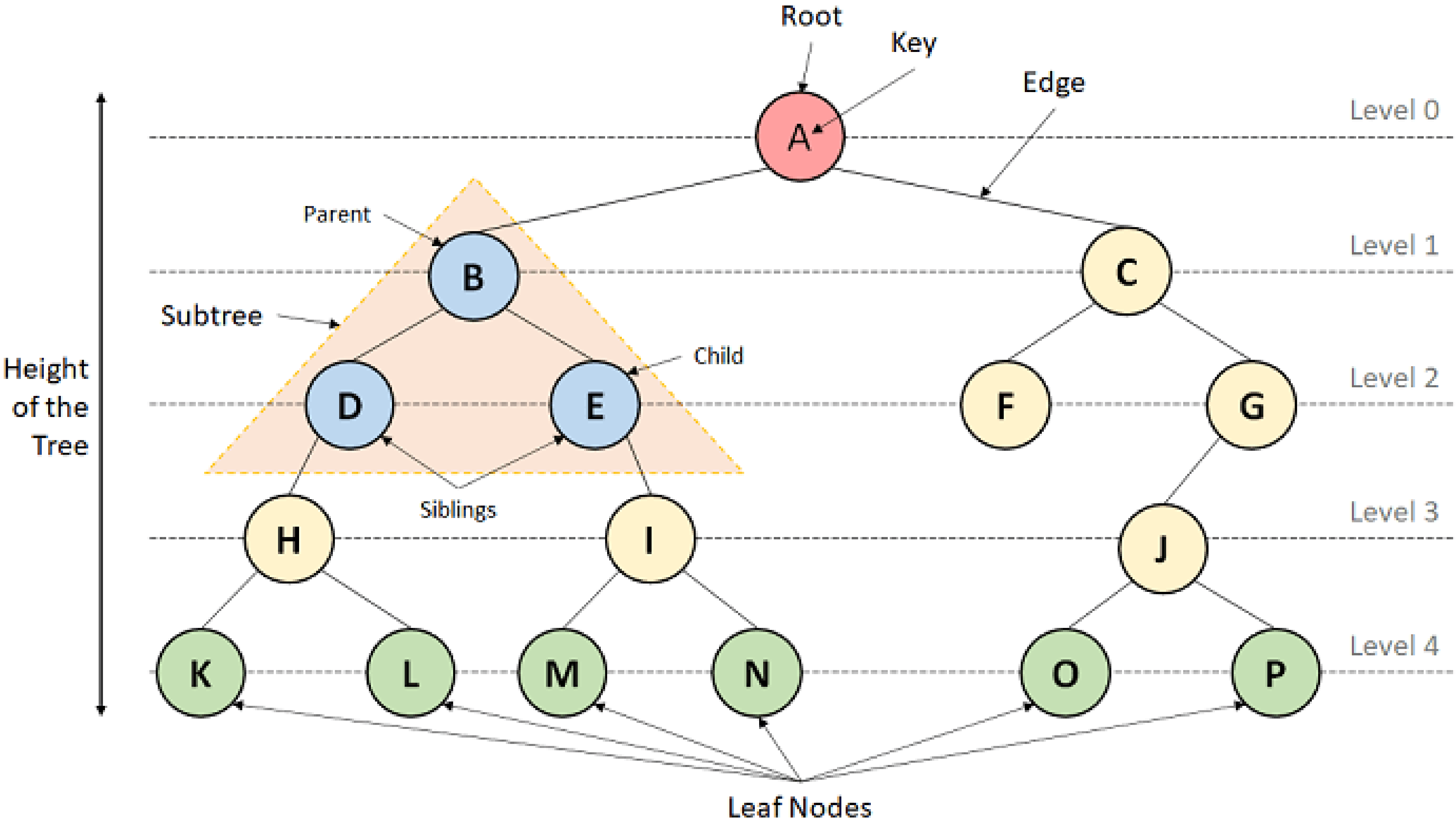
Non-Linear Data Structures

- Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order.
- Here, the insertion and removal of data are not feasible in a linear manner.
- There exists a hierarchical relationship between the individual data items.

Types of Non-Linear Data Structures

1. Trees

- A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").
- The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.


Root
Key
Edge
A
Level 0
Parent
B
C
Level 1
Subtree
Child
Height of the Tree
D
E
F
G
Level 2
Siblings
H
I
J
Level 3
K
L
M
N
O
P
Level 4
Leaf Nodes

Trees can be classified into different types:

1. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.
2. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.
3. **AVL Tree:** An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.
4. **B-Tree:** A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.

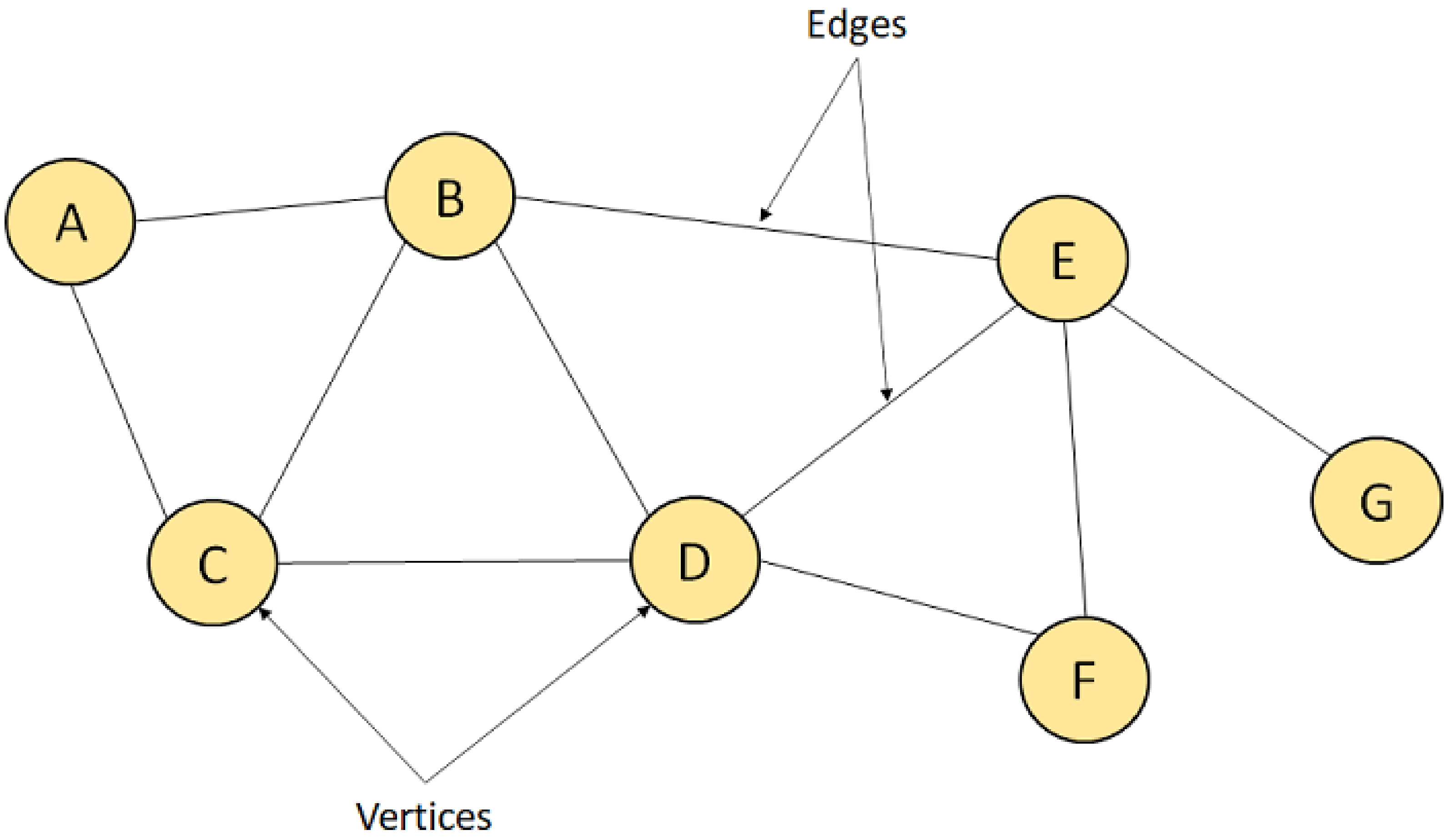
Some Applications of Trees:

1. Trees implement hierarchical structures in computer systems like directories and file systems.
2. Trees are also used to implement the navigation structure of a website.
3. We can generate code like Huffman's code using Trees.
4. Trees are also helpful in decision-making in Gaming applications.
5. Trees are responsible for implementing priority queues for priority-based OS scheduling functions.
6. Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.
7. We can use Trees to store data keys for indexing for Database Management System (DBMS).
8. Spanning Trees allows us to route decisions in Computer and Communications Networks.
9. Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications.

Graphs

- A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them.
- The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks.
- For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

- The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below:
- G = (V,E)



Some Applications of Graphs:

1. Graphs help us represent routes and networks in transportation, travel, and communication applications.
2. Graphs are used to display routes in GPS.
3. Graphs also help us represent the interconnections in social networks and other network-based applications.
4. Graphs are utilized in mapping applications.
5. Graphs are responsible for the representation of user preference in e-commerce applications.
6. Graphs are also used in Utility networks in order to identify the problems posed to local or municipal corporations.
7. Graphs also help to manage the utilization and availability of resources in an organization.
8. Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.
9. Graphs are also used in robotic motions and neural networks.

Basic Operations of Data Structures

1. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered.
2. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
3. **Insertion:** Insertion means inserting or adding new data elements to the collection.
4. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements.
5. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.

Basic Operations of Data Structures

1. **Merge:** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.

2. **Create:** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:

   1. Compile-time---------- --2.Run-time

   For example, the **malloc()** function is used in C Language to create data structure.

3. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.

4. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.

5. **Splitting:** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

Abstract Data Type

- As per the **National Institute of Standards and Technology (NIST)**, a data structure is an arrangement of information, generally in the memory, for better algorithm efficiency. Data Structures include linked lists, stacks, queues, trees, and dictionaries.

- From the definition mentioned above, we can conclude that the operations in data structure include:

1. A high level of abstractions like addition or deletion of an item from a list.
2. Searching and sorting an item in a list.
3. Accessing the highest priority item in a list.

Whenever the data structure does such operations, it is known as an **Abstract Data Type (ADT)**.

Abstract Data Type

- We can define it as a set of data elements along with the operations on the data.

- The term "abstract" refers to the fact that the data and the fundamental operations defined on it are being studied independently of their implementation.
- It includes what we can do with the data, not how we can do it.

- An ADI implementation contains a storage structure in order to store the data elements and algorithms for fundamental operation.
- All the data structures, like an array, linked list, queue, stack, etc., are examples of ADT.

Some Applications of Data Structures

1. Data Structures help in the organization of data in a computer's memory.
2. Data Structures also help in representing the information in databases.
3. Data Structures allows the implementation of algorithms to search through data (For example, search engine).
4. We can use the Data Structures to implement the algorithms to manipulate data (For example, word processors).
5. We can also implement the algorithms to analyse data using Data Structures (For example, data miners).
6. Data Structures support algorithms to generate the data (For example, a random number generator).
7. Data Structures also support algorithms to compress and decompress the data (For example, a zip utility).
8. We can also use Data Structures to implement algorithms to encrypt and decrypt the data (For example, a security system).
9. With the help of Data Structures, we can build software that can manage files and directories (For example, a file manager).
10. We can also develop software that can render graphics using Data Structures. (For example, a web browser or 3D rendering software).

The number of elements is called the length or size of the array. In the above situation, **n** is the length of the array.

Program

A program that shows how to perform the insertion, deletion, reversing, display, and searching operations on an array is given below.

1

```
#include<stdio.h>
#include<conio.h>
#define MAX 5
void insert(int*,int pos,int num);        (int*, int pos, int num)
void del (int*,int pos);                  (int*, int pos);
void reverse(int*);
void display(int*);
void search(int*,int num);
void main()
{
int a[5];
clrscr();
insert(a,1,11);
insert(a,2,22);
insert(a,3,33);
insert(a,4,44);
insert(a,5,55);
printf("\nElements of array: \n");
display(a);
del(a,5);
del(a,2);
printf("\n\nAfter deletion: \n");
display(a);
insert(a,2,200);
insert(a,5,500);
printf("\n After insertion: \n");
display(a);
reverse(a);
printf("\n\n After reversing: \n");
display(a);
search(a,200);
search(a,600);
getch();
}
void insert(int*a,int pos,int num) // inserts an element num at given position pos//
{
/* shift elements to right */
int i;
for(i=MAX–1;i>=pos;I– –)                 for( i= Max-1 ; i>=pos ; i--)
a[i]=a[i–1];                              a[i] = a[i-1];
a[i]=num;
 }
void del(int *a,int pos)          // deletes an element from the given position pos//
{
/* skip to the desired position*/
int i;
for(i=pos;i<MAX;i++)
a[i–1]=a[i];
a[i–1]=0;
 }
void reverse(int *a)          // reverses the entire array //
{
int i;
for(i=0;i<MAX/2;i++)          fa (i= ; i< max/2 ;i++)
 {                            {
int temp=a[i];                 int temp =a(i);
a[i]=a[MAX–1–i];               a(i) = a(max-1-i);
a[MAX–1–i]=temp;               a(max-1-i) = temp;
 }                             }
}
void search(int *a,int num)          // searches array for a given element num //
{
/* traverse the array */
int i;
for(i=0;i<MAX;i++)
{
if(a[i]==num)               a(i)= num
{
printf ("\n\nElement %d is present at %dth position",num,i+1);
return;               (   (i) = max )
 }
   }
if(i==MAX)
printf ("\nElement %d is not present in the array",num);
}
void display(int *a)          //displays the contents of a array //
{
/* traverse the entire array */
int i ;
printf ("\n");
for(i=0;i<MAX;i++)
printf("%d\t",a[i]);
}
```

Output:

```
Elements of array:
10     20     30     40     50
After deletion:
10     30     40     0      0
After insertion:
10     200    30     40     500
After reversing:
500    40     30     200    10
Element 200 is present at the 4th position
Element 600 is not present in the array
```

In this program, we created array a that contains 5 integers. Then, the base address of this array is passed to functions like *insert( ), del( ), display( ), reverse( ) and search( )* to perform different operations on the array.

DS Algorithm
What is an Algorithm?

- An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.
- The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task.
- It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Types of Algorithms

Search Algorithm

- On each day, we search for something in our day to day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user.

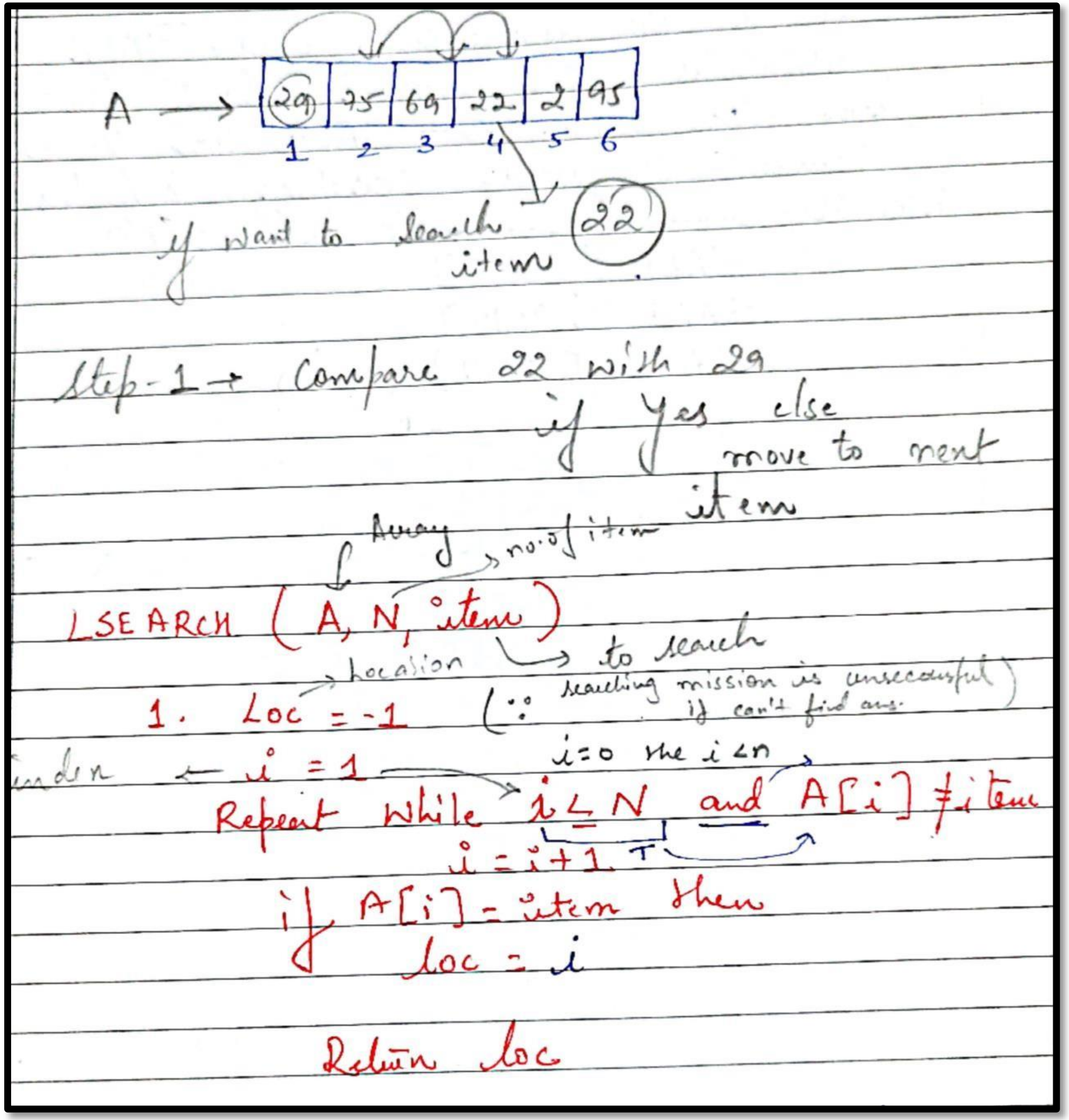
There are mainly two techniques available to search the data in an array:

- **Linear search**
- **Binary search**

Linear Search

- Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found.
- It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1.
- This algorithm can be implemented on the unsorted list.

Linear Search in Data Structure



A →

29	75	69	22	2	95
1	2	3	4	5	6

if want to search item 22.

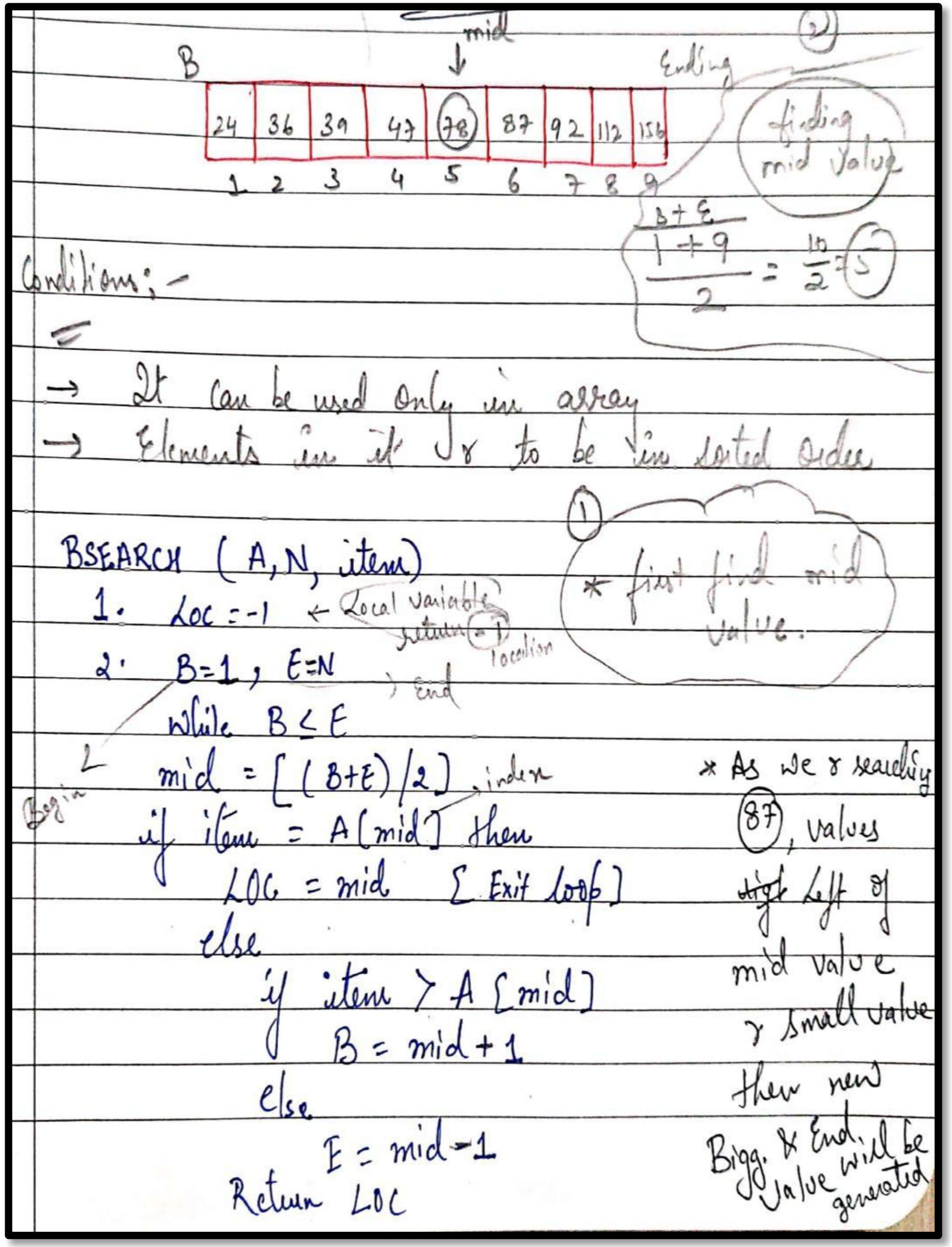
Step-1 → Compare 22 with 29
if yes else move to next item

```
        Array      no. of item
LSEARCH ( A, N, item )
                     to search
       location
1. Loc = -1   (∵ searching mission is unseccausful
                 if can't find ans.)
                          i=0 the i<n
index ← i = 1
   Repeat While i ≤ N and A[i] ≠ item
                   T
      i = i+1
   if A[i] = item then
      loc = i

   Return loc
```

Binary Search

- A Binary algorithm is the simplest algorithm that searches the element very quickly.
- It is used to search the element from the sorted list.
- The elements must be stored in sequential order or the sorted manner to implement the binary algorithm.
- Binary search cannot be implemented if the elements are stored in a random manner.
- It is used to find the middle element of the list.

Binary Search in Data Structure



mid

B ↓ Ending ②

24	36	39	47	78	87	92	112	156
1	2	3	4	5	6	7	8	9

finding mid Value

$\frac{B+E}{}$

$\frac{1+9}{2} = \frac{10}{2} = 5$

Conditions:-

→ It can be used Only in array

→ Elements in it Ur to be in sorted Order

①

* first find mid value.

```
BSEARCH ( A, N, item)
  1. Loc = -1   ← Local variable  return(-1) location
  2. B=1, E=N   ) End
     While B ≤ E
     mid = [ (B+E)/2 ] , index
     if item = A(mid) then
         LOC = mid    [ Exit loop ]
     else
         if item > A [mid]
             B = mid + 1
         else
             E = mid - 1
     Return LOC
```

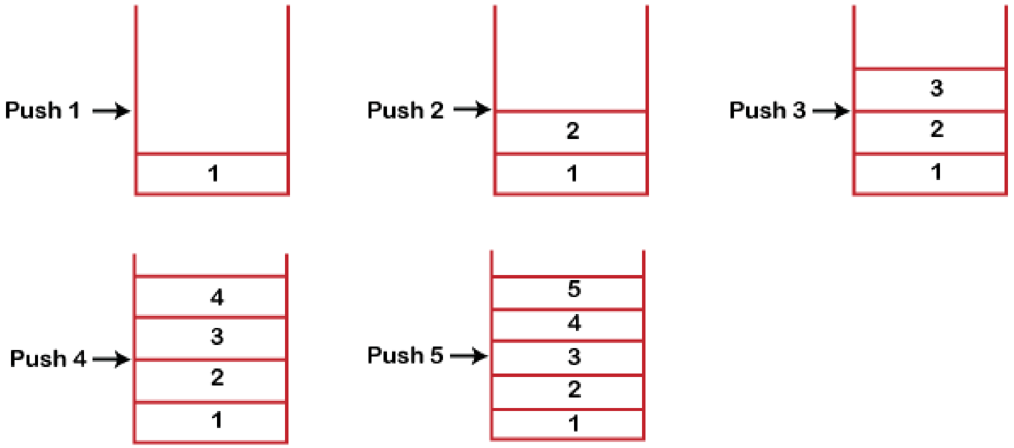
Begin

* As we r searching ⑧⑦, values ~~high~~ Left of mid value r small value then new Bigg. & End. Value will be generated

What is a Stack?

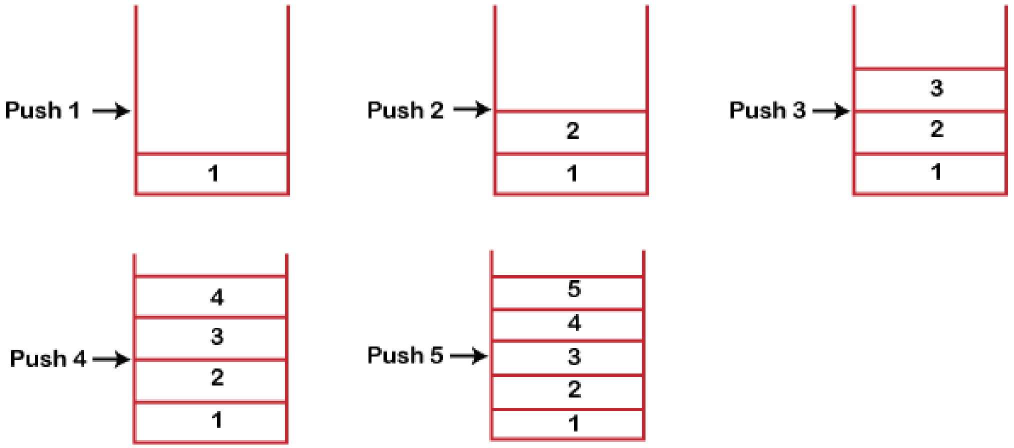
- A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
- Stack has one end, whereas the Queue has two ends (**front and rear**).
- It contains only one pointer **top pointer** pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Working of Stack



- Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.

Working of Stack



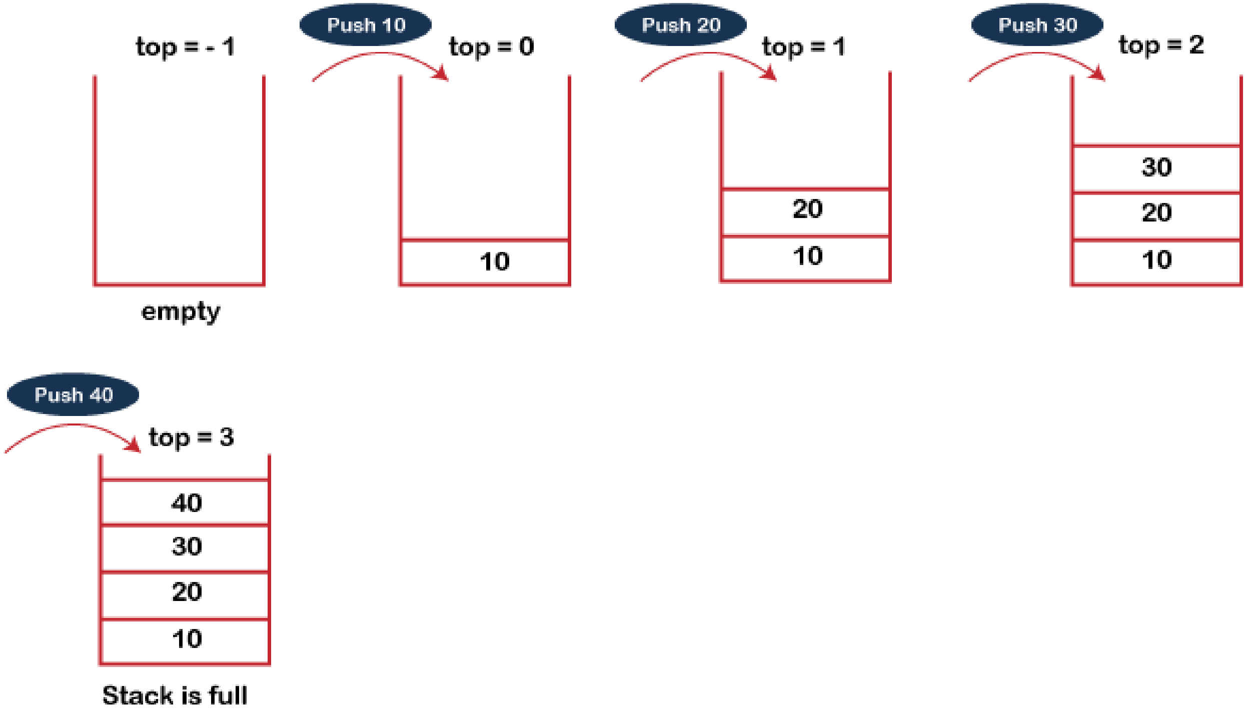
- When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

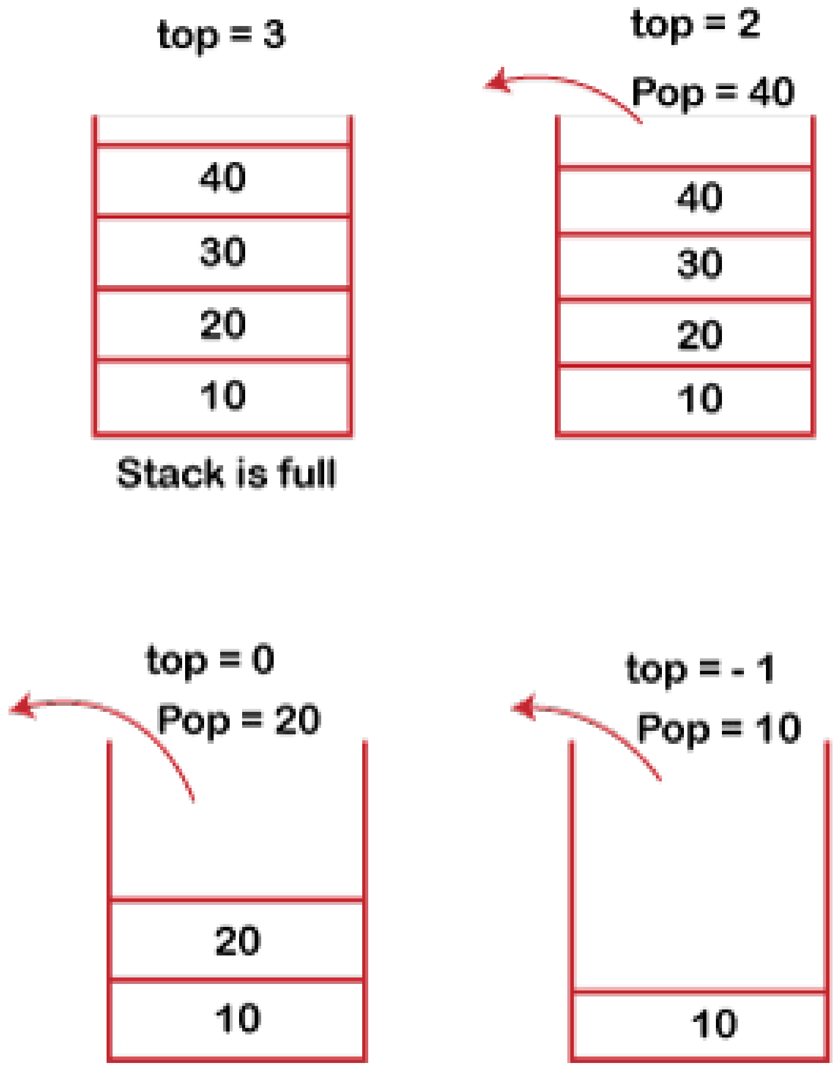
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Bubble sort Algorithm

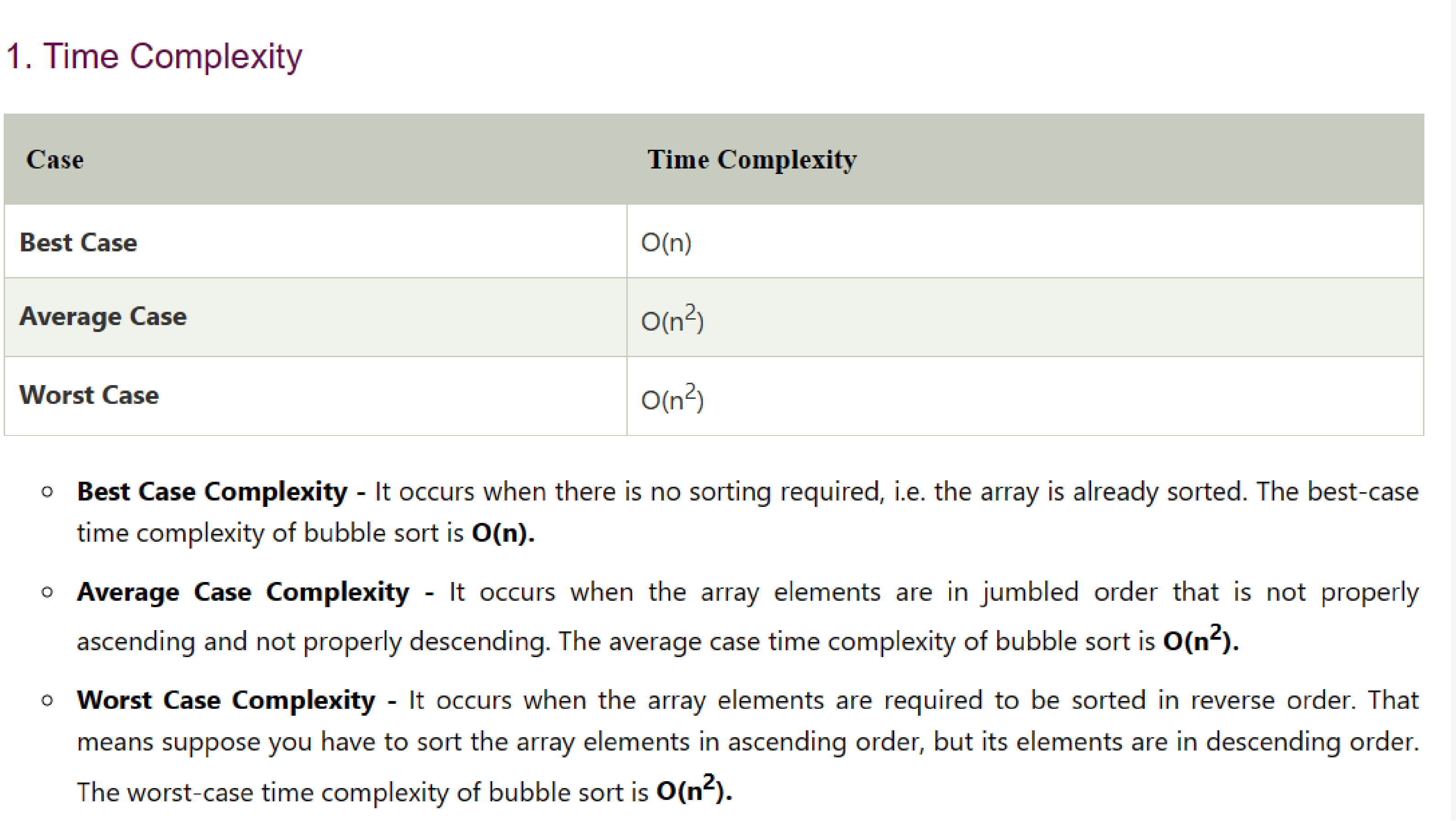
- Bubble sort works on the repeatedly swapping of adjacent elements until they are **not** in the intended order.
- It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water.
  - Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.
- Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets.
- The average and worst-case complexity of Bubble sort is **$O(n^2)$,** where **n** is a number of items.

Algorithm

In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```

Bubble sort complexity

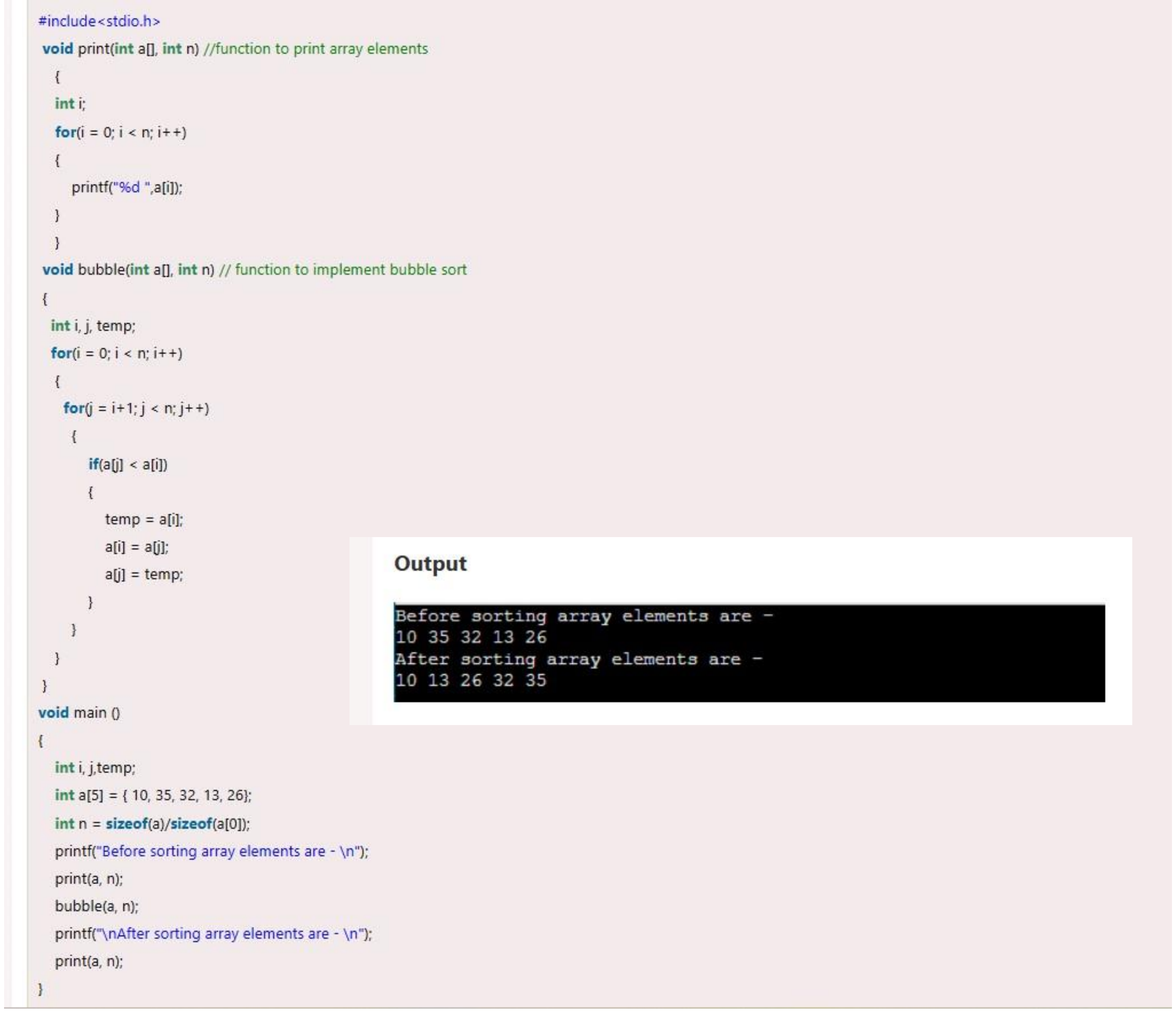


1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

Implementation of Bubble sort



```c
#include<stdio.h>
 void print(int a[], int n) //function to print array elements
   {
   int i;
   for(i = 0; i < n; i++)
   {
       printf("%d ",a[i]);
   }
   }
 void bubble(int a[], int n) // function to implement bubble sort
 {
  int i, j, temp;
  for(i = 0; i < n; i++)
   {
     for(j = i+1; j < n; j++)
       {
           if(a[j] < a[i])
           {
               temp = a[i];
               a[i] = a[j];
               a[j] = temp;
           }
       }
   }
 }
void main ()
{
   int i, j,temp;
   int a[5] = { 10, 35, 32, 13, 26};
   int n = sizeof(a)/sizeof(a[0]);
   printf("Before sorting array elements are - \n");
   print(a, n);
   bubble(a, n);
   printf("\nAfter sorting array elements are - \n");
   print(a, n);
}
```

Output

```
Before sorting array elements are -
10 35 32 13 26
After sorting array elements are -
10 13 26 32 35
```

Insertion Sort Algorithm

- Insertion sort works similar to the sorting of playing cards in hands.
- It is assumed that the first card is already sorted in the card game, and then we select an unsorted card.
- If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.
- The same approach is applied in insertion sort.
- The idea behind the insertion sort is that first take one element, iterate it through the sorted array. **Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$,** where n is the number of items.
- Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as –

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

- **Step 1 -** If the element is the first element, assume that it is already sorted. Return 1.
- **Step2 -** Pick the next element, and store it separately in a **key.**
- **Step3 -** Now, compare the **key** with all elements in the sorted array.
- **Step 4 -** If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
- **Step 5 -** Insert the value.
- **Step 6 -** Repeat until the array is sorted.

```
For j= 2 to A.length
      Key= A[j]
     //Insert A[j] into the sorted sequence A[1…… j-1]

     i=j-1
     While i>0 && a[i]>key
       A[i+1] = A[i]
            i=i-1
     A[i+1]=key
```

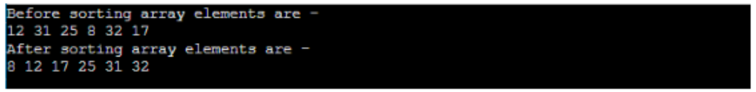
Time Complexity

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

```c
#include <stdio.h>

void insert(int a[], int n) /* function to sort an aay with insertion sort */
{
  int i, j, temp;
  for (i = 1; i < n; i++) {
    temp = a[i];
    j = i - 1;

    while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position ahead from their current position*/
    {
      a[j+1] = a[j];
      j = j-1;
    }
    a[j+1] = temp;
  }
}

void printArr(int a[], int n) /* function to print the array */
{
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", a[i]);
}

int main()
{
  int a[] = { 12, 31, 25, 8, 32, 17 };
  int n = sizeof(a) / sizeof(a[0]);
  printf("Before sorting array elements are - \n");
  printArr(a, n);
  insert(a, n);
  printf("\nAfter sorting array elements are - \n");
  printArr(a, n);

  return 0;
}
```

Output:



```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

Array implementation of Stack

- In array implementation, the stack is formed by using the array.

- All the operations regarding the stack are performed using arrays.

Adding an element onto the stack (push operation)

- Adding an element into the top of the stack is referred to as push operation.
- Push operation involves following two steps:-
  1. Increment the variable Top so that it can now refer to the next memory location.
  2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.
- Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
begin
    if top = n then stack full
    top = top + 1
    stack (top) : = item;
end
```

Implementation of push algorithm in C language

```
void push (int val,int n) //n is size of the stack
{
   if (top == n )
   printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
   }
}
```

Deletion of an element from a stack (Pop operation)

- Deletion of an element from the top of the stack is called pop operation.
- The value of the variable top will be incremented by 1 whenever an item is deleted from the stack.
- The top most element of the stack is stored in an another variable and then the top is decremented by 1.
- The operation returns the deleted value that was stored in another variable as the result.
- The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
    if top = 0 then stack empty;
    item := stack(top);
    top = top - 1;
end;
```

Implementation of POP algorithm using C language

```
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

C program

```c
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");

printf("\n --------------------------------------------\n");
```

```
while(choice != 4)
  {
    printf("Chose one from the below options...\n");
    printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
    printf("\n Enter your choice \n");
    scanf("%d",&choice);
    switch(choice)
    {
      case 1:
      {
        push();
        break;
      }
      case 2:
      {
        pop();
        break;
      }
```

```
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
default:
            {
                printf("Please Enter valid choice ");
            }
        };
    }
}
```

```
 void push ()
{
   int val;
   if (top == n )
   printf("\n Overflow");
   else
   {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = val;
   }
}
```

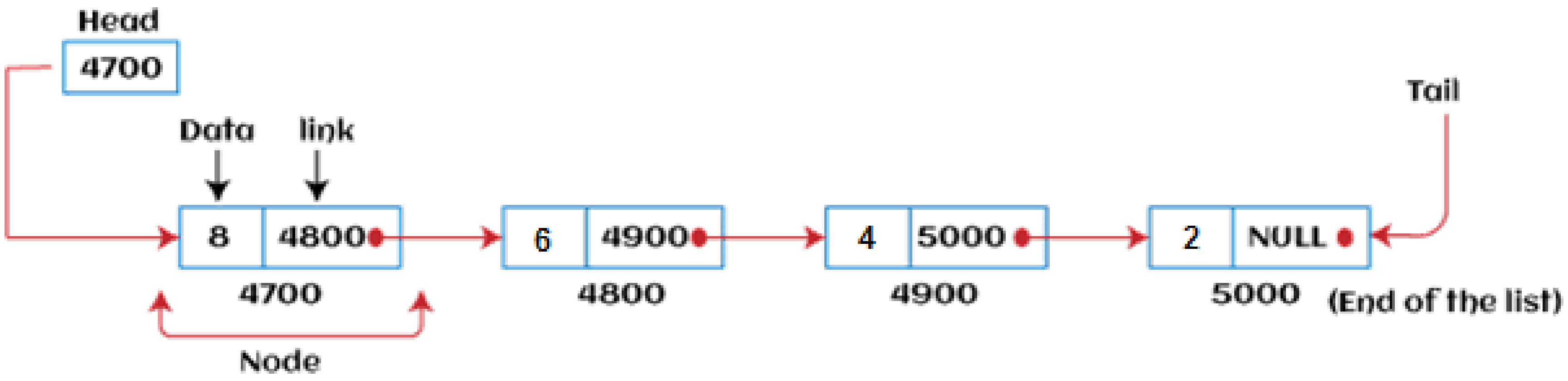
```
 void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Linked list

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

Representation of a Linked list

- Linked list can be represented as the connection of nodes in which each node points to the next node of the list.



Representation of a Linked list

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

Why use linked list over array?

- Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -
  - The size of the array must be known in advance before using it in the program.
  - Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
  - All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.
- Linked list is useful because -
  - It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
  - In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

How to declare a linked list?

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

The declaration of linked list is given as follows -

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

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

Types of Linked list

- **Singly-linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- **Doubly linked list -** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).

Types of Linked list

- **Circular singly linked list -** In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- **Circular doubly linked list -** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Comparison Between

Advantages of Linked list

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

Disadvantages of Linked list

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

Applications of Linked list

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

Operations performed on Linked list

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

Types of Linked List

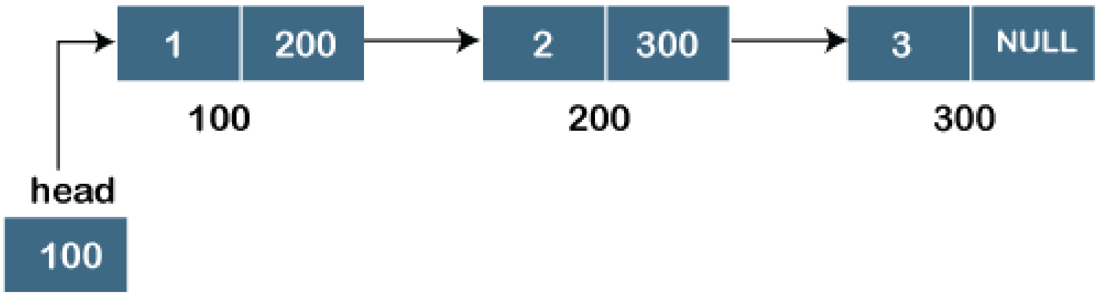
- Before knowing about the types of a linked list, we should know what is ***linked list***.

The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

Singly Linked list

- It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list.
- The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node.
- The address part in a node is also known as a **pointer**.
- Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively.



- Three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node.
- The pointer that holds the address of the initial node is known as a ***head pointer***.
- The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

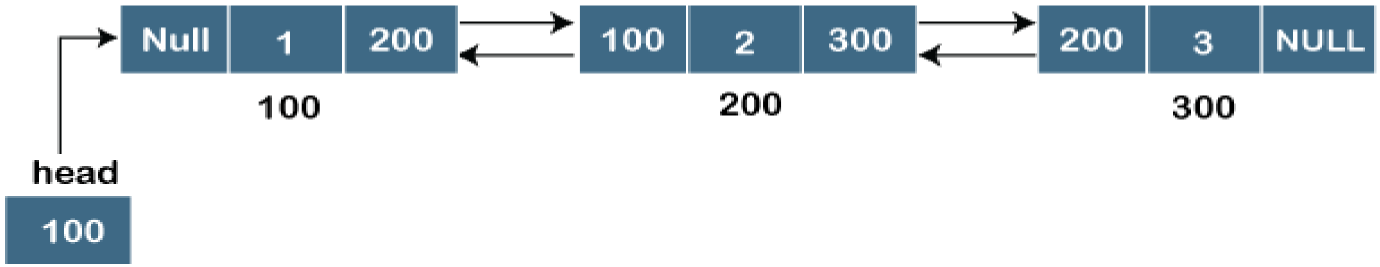
```
struct node
{
  int data;
 struct node *next;
}
```

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Doubly linked list

- The doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part.
- In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.
- Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively.

- The representation of these nodes in a doubly-linked list is shown below:



- As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

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

- In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type.
- The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node.
- The type of both the pointers, i.e., **next and prev** is **struct node** as both the pointers are storing the address of the node of the ***struct node*** type.

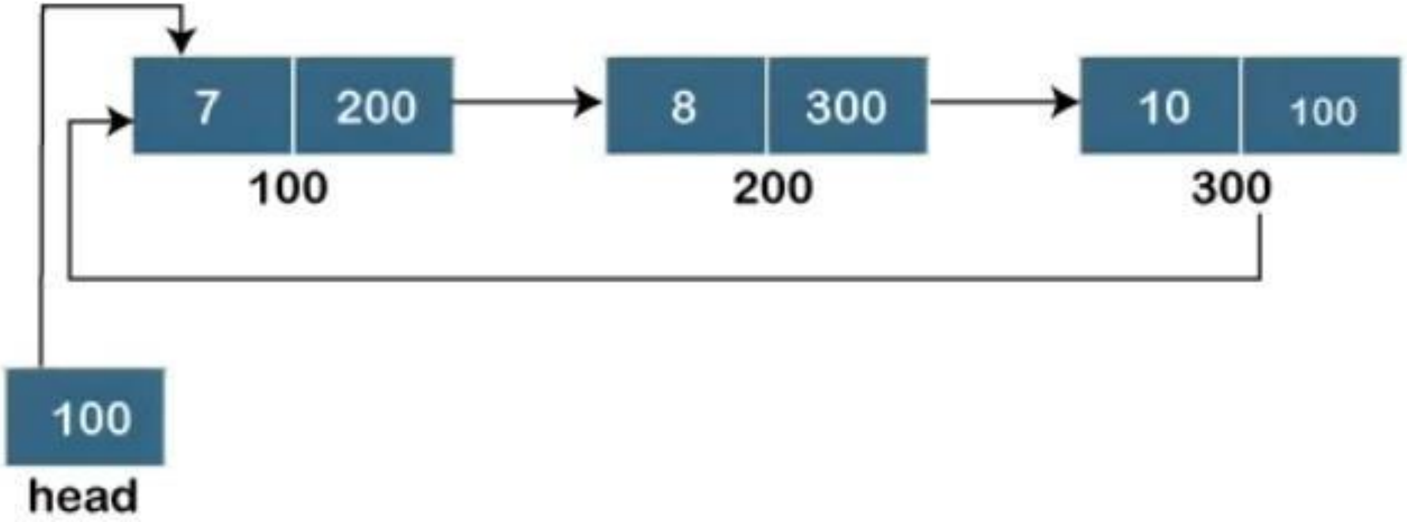
Circular linked list

- A circular linked list is a variation of a singly linked list.
- The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value.
- On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address.
- The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward.

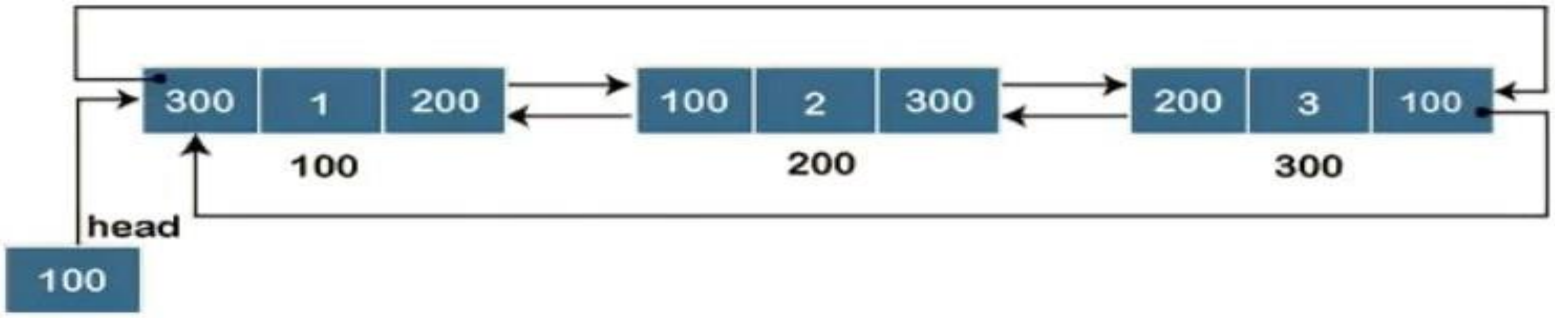
The diagrammatic representation of the circular linked list is shown below:

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

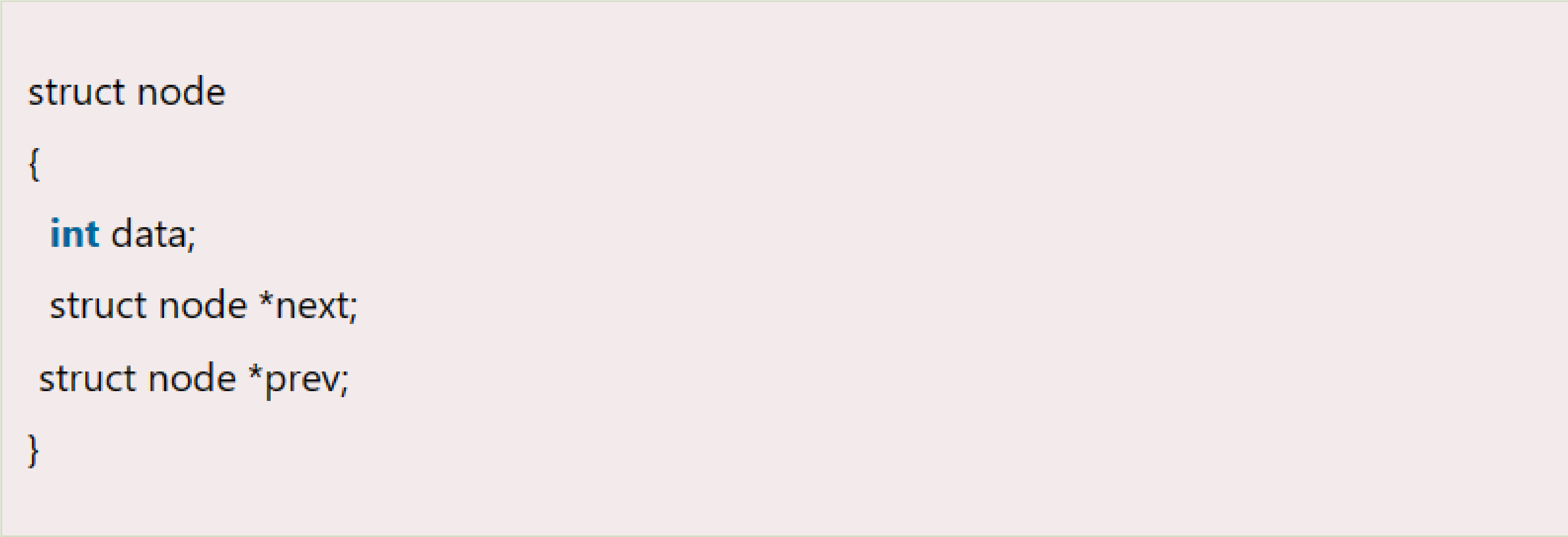
A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node.



Doubly Circular linked list



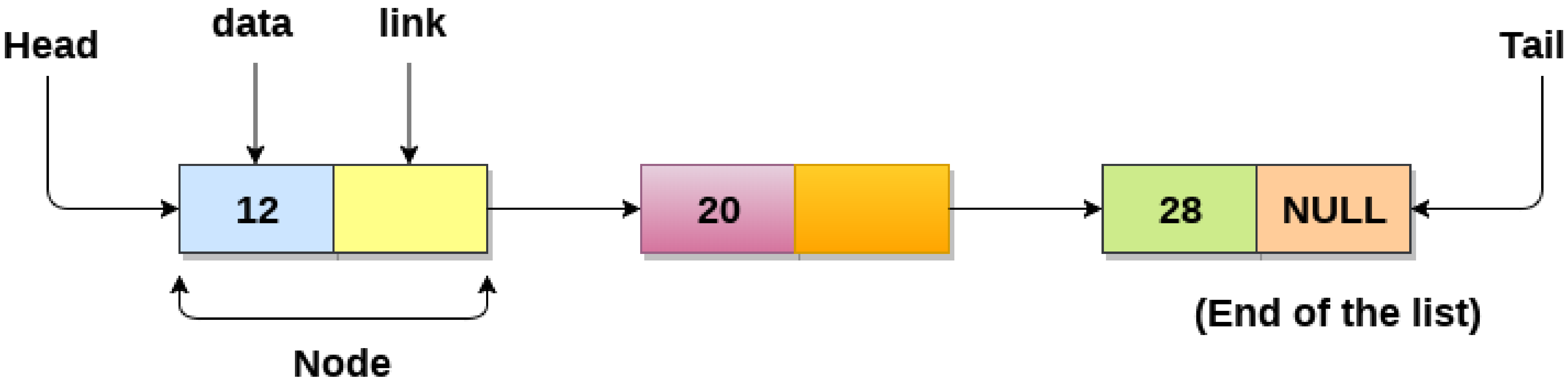
- The representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle.
- It is a doubly linked list also because each node holds the address of the previous node also.
- The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node.
- As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.



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

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list.
- This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

- Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory.
- However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.
- Array contains following limitations:

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

Why use linked list over array?

- Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

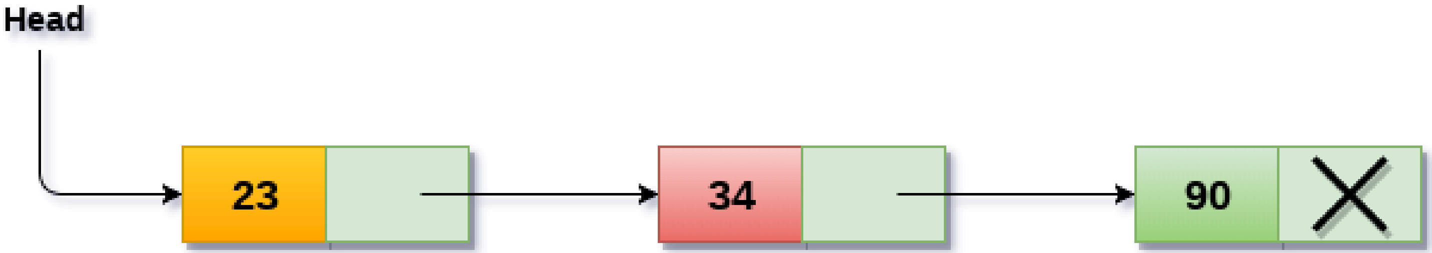
1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and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

Singly linked list or One way chain

- Singly linked list can be defined as the collection of ordered set of elements.
- The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part.
- Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

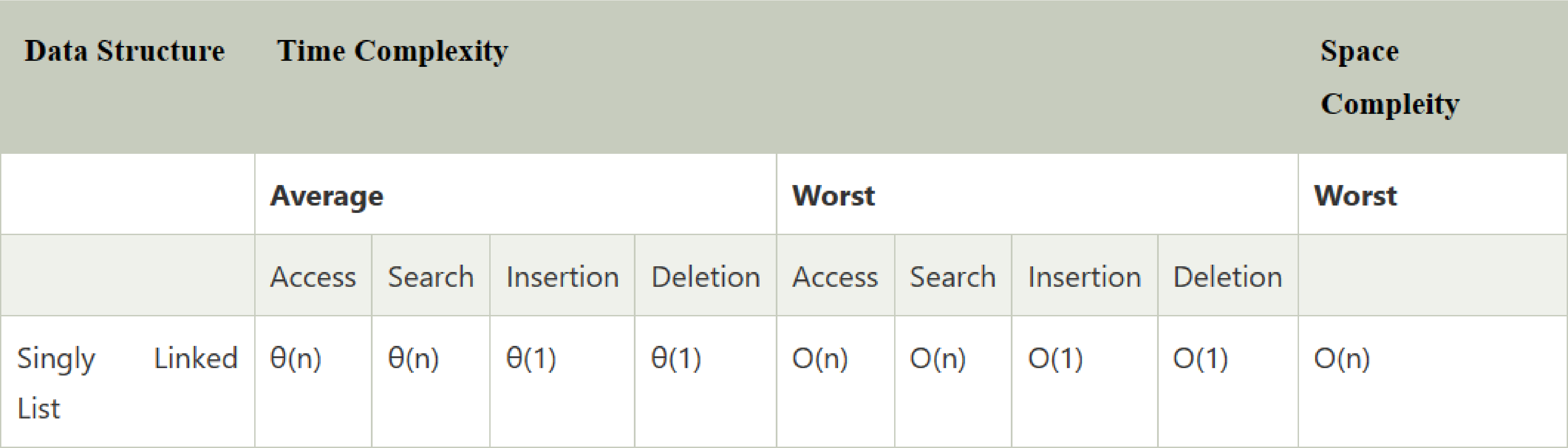
Singly linked list or One way chain

- One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.
- Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



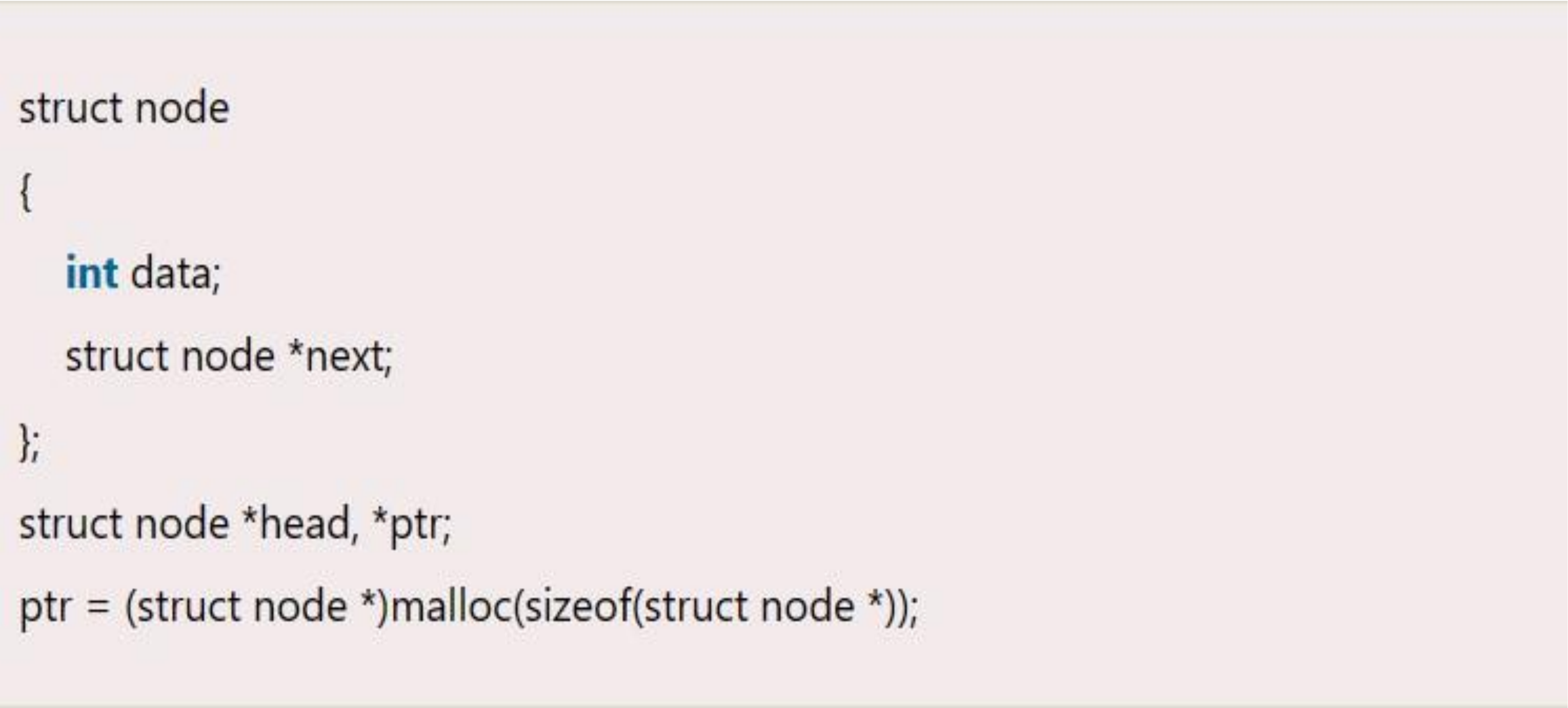
- The data part of every node contains the marks obtained by the student in the different subject.
- The last node in the list is identified by the null pointer which is present in the address part of the last node.
- We can have as many elements we require, in the data part of the list.

Complexity



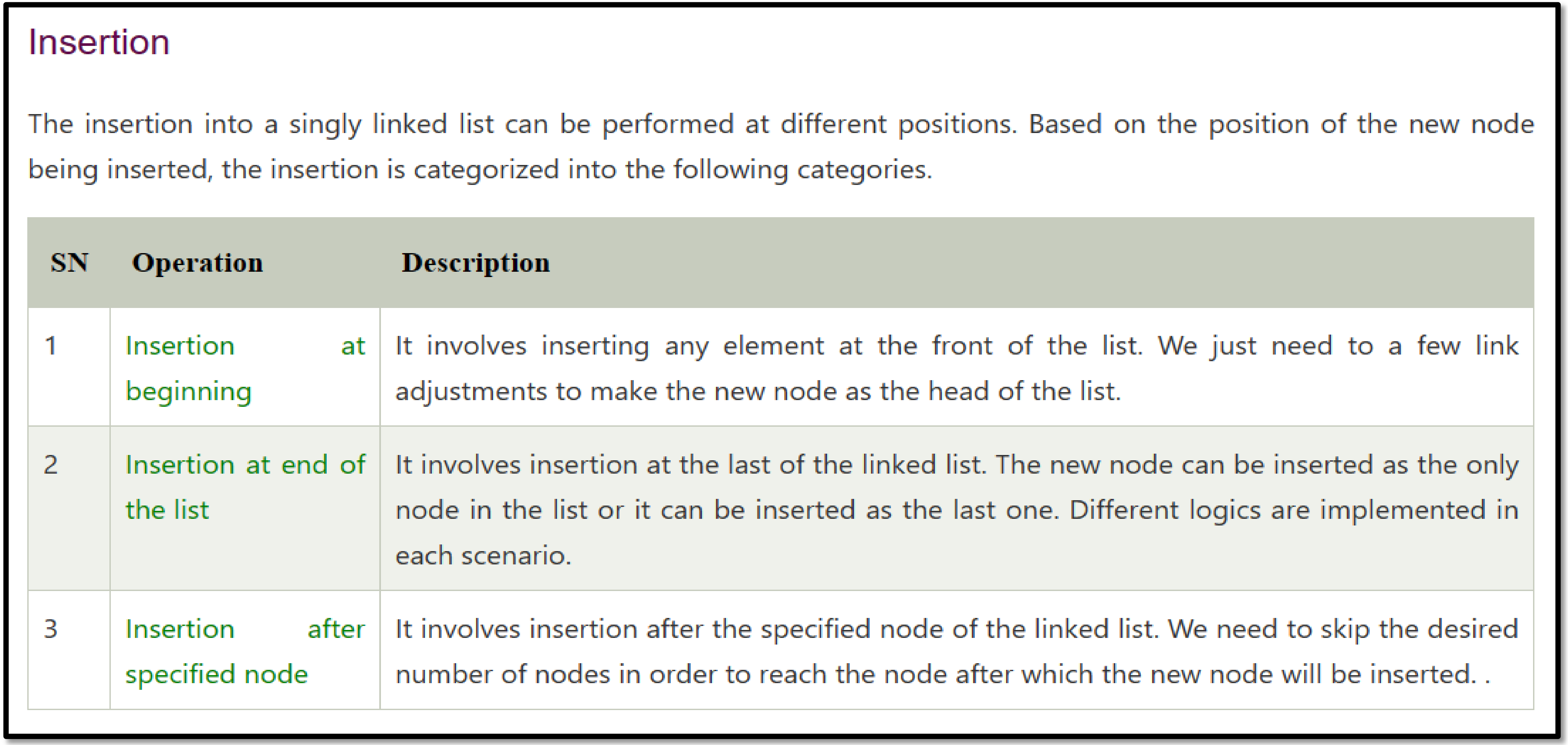
Data Structure	Time Complexity								Space Compleity
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Node Creation



```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion



Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .