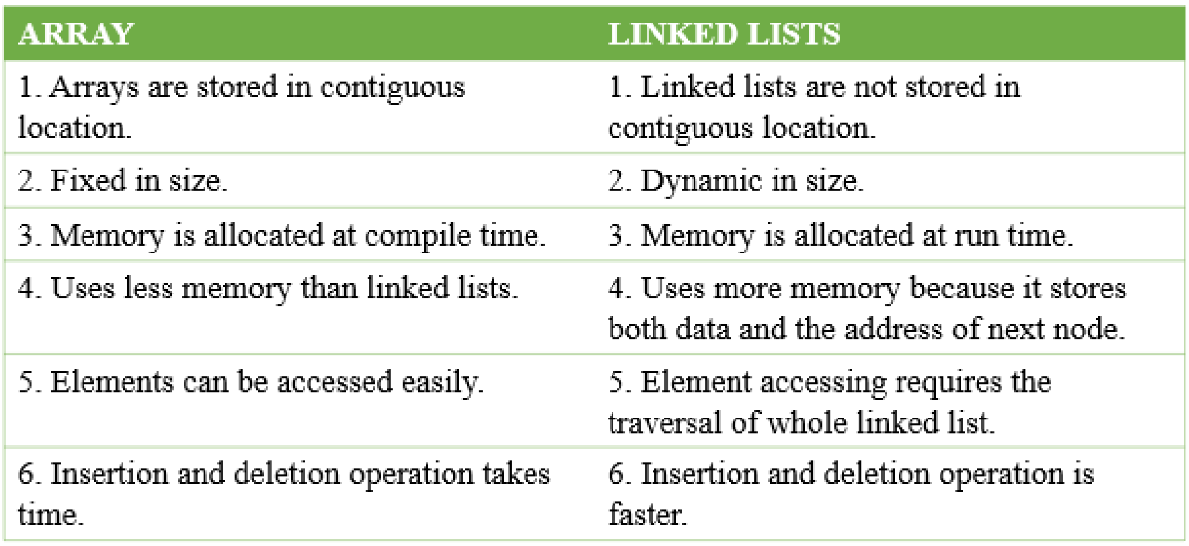
Linked List:

Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.

Arrays vs Linked List



ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Why Linked list over arrays?

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.

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

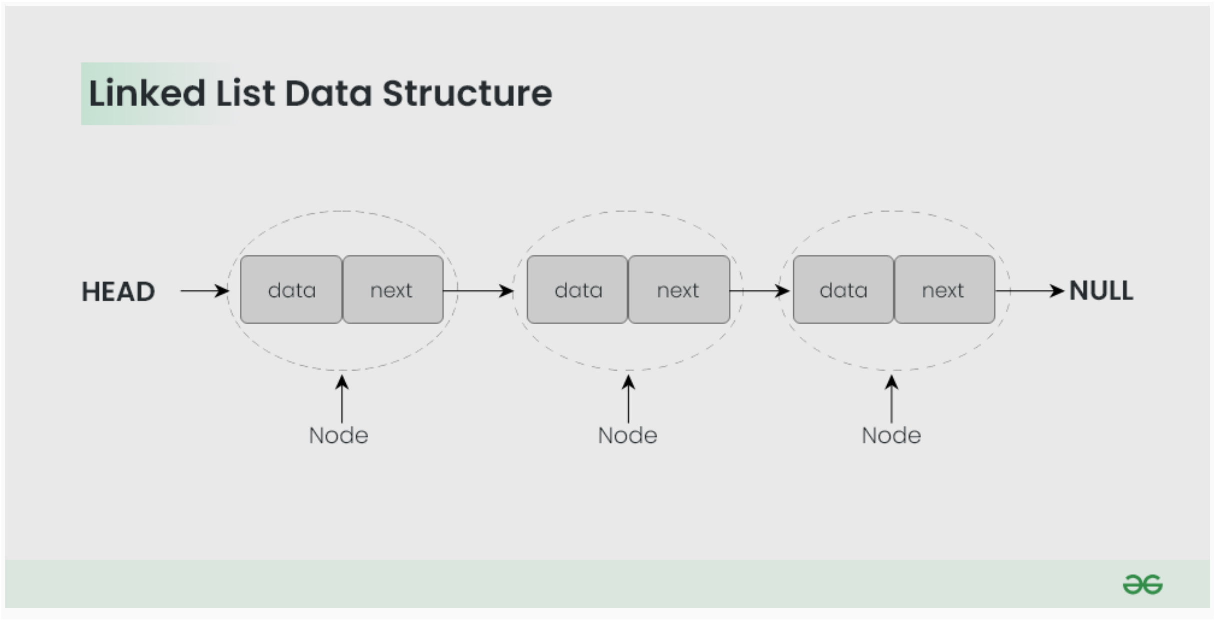
1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and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

Types of Linked List

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

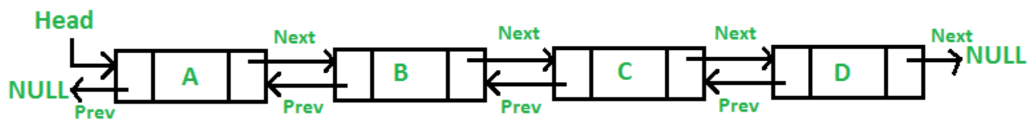
1.Singly Linked List

A singly linked list is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.



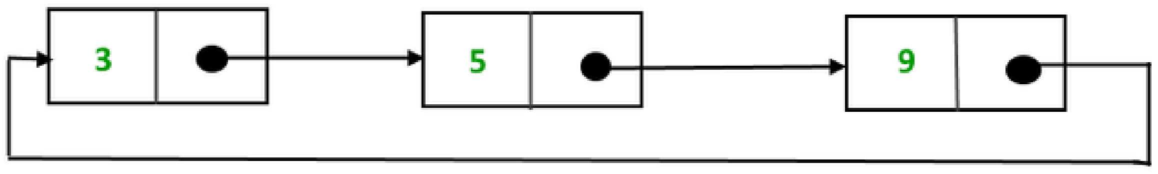
2.Doubly Linked List

A doubly linked list is a linear data structure where each element, or node, contains a value and references to both the next and previous nodes, allowing for efficient traversal in both directions.



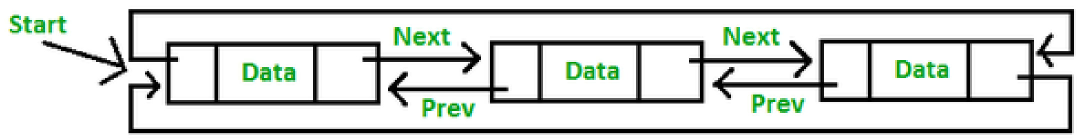
3.Circular singly linked list

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



4.Circular Doubly linked list

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



Skip List:

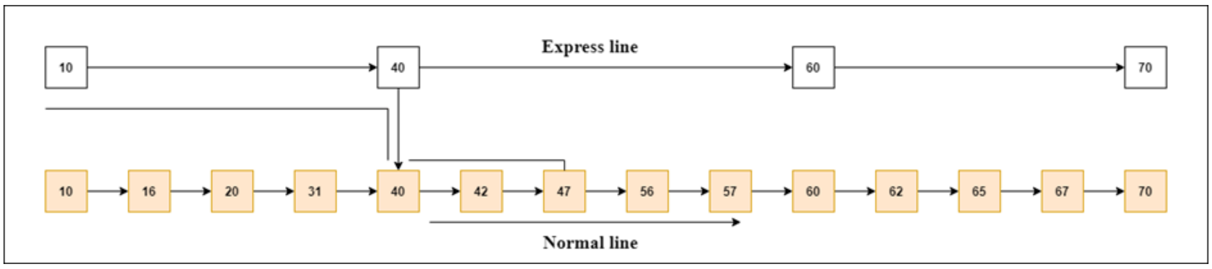
A Skip List is a data structure that is used for searching and inserting elements in a sorted list efficiently.It combines the characteristics of a linked list with a multilevel index structure, similar to a tree, to achieve fast search times with relatively simple and memory-efficient data storage.

Skip list structure

It is built in two layers: The lowest layer and Top layer.The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

Working of the Skip list

Let's take an example to understand the working of the skip list. In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.The lower layer is a common line that links all nodes, and the top layer is an express line that links only the main nodes, as you can see in the diagram.Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal a 47 or more than 47.You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.



Self-Organizing List:

A Self-Organizing List (SOL), also known as a Self-Adjusting List, is a data structure where the elements within the list are automatically reorganized or adjusted based on their access patterns. The goal of a self-organizing list is to make frequently accessed elements more quickly accessible by moving them closer to the front of the list, reducing the time it takes to retrieve them.

Move-To-Front (MTF):

In the MTF strategy, whenever an element is accessed (searched or retrieved), it is moved to the front of the list. This means that the most recently accessed element is always at the front, making it quicker to access again.

Drawbacks:

1)Overly aggressive: MTF moves an accessed element to the very front of the list, which can lead to thrashing (frequent items constantly moving) in scenarios with rapidly changing access patterns.

2)Poor performance for non-frequent items: Items that are accessed infrequently may remain near the front, causing inefficiencies.

3)Can be sensitive to initial order: The effectiveness of MTF can depend on the initial order of elements.

2)Transpose Method:The Transpose strategy aims to move the accessed element one step closer to the front whenever it is accessed. This strategy helps in gradually bringing frequently accessed items closer to the front but not as aggressively as the MTF method.

Drawbacks:

1)Less aggressive: Transpose moves an accessed element only one step closer to the front, which can be slow to adapt to rapidly changing access patterns.

2)May not provide significant improvements: For access patterns that involve multiple accesses to the same item in a short period, Transpose may not offer substantial benefits.

3. Count Method:

This method keeps track of how many times each element is accessed.When you access an element, the count for that element is incremented.The list can periodically or upon specific triggers, reorganize itself based on the access counts. Elements with higher access counts are moved closer to the front.The count method can be particularly effective when there are a few elements that are accessed very frequently and others that are rarely accessed.

Drawbacks:

1)Increased overhead: Maintaining access counts for each element can add overhead in terms of memory and processing, especially for large lists.

2)Delayed reorganization: The list may not reorganize immediately, and waiting for a trigger or threshold can delay the benefits of reordering.

3)May not handle all access patterns: The count method may not be effective for access patterns that are highly dynamic or unpredictable.

Algorithm:

1)Insertion at the beginning of the linked list

```
Step 1:Check if the node is empty .If it's empty terminate the process
 Step2:Create a new node
 Step 3:Update the pointer of new node to point to the current head
 Step 4:Update the head pointer to point at new node.
 Step 5: Stop
```

2)Insertion at the specified node of the linked list.

```
Step1:Check if the specified node exists.
 Step2:Create a new node
 Step3:Update the pointer of new node to point to the node that comes after
specified node.
 Step4:Update the pointer of the specified node to point at the new node.
 Step5:Stop.
```

3)Insertion at the end of the linked list.

Step1:Create a new node.

Step2:Update the pointer of last node to point to the new node.

Step3.Update the pointer of new node to point to null.

Step4:Stop.

4)Deletion at the specified node of the linked list

Step1:Find the node that you have to delete.

Step2:Update the pointer of previous node to point to the next node of specified node.

Step3:Delete the old node to free up space.

Step3:Stop.

Dynamic Programming

Dynamic Programming is a powerful technique used in computer science and mathematics to solve problems by breaking them down into smaller overlapping subproblems and efficiently solving each subproblem only once, storing the results to avoid redundant work. It is particularly useful for optimization and combinatorial problems where there are overlapping substructures.

Or

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimise it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

Here are some key points about Dynamic Programming:

Optimal Substructure: Dynamic Programming problems exhibit optimal substructure, which means that an optimal solution to the problem can be constructed from optimal solutions to its subproblems. In other words, problems can be broken down into smaller, more manageable parts.

Memoization: Dynamic Programming often involves memoization, which is a technique of storing the results of expensive function calls and reusing them when the same inputs occur again. This avoids recalculating the same subproblem multiple times, leading to significant performance improvements.

Top-Down vs. Bottom-Up: Dynamic Programming can be approached in two main ways: top-down (recursion with memoization) and bottom-up (iterative with tabulation). Top-down starts with the original problem and recursively solves smaller subproblems, while bottom-up starts with the smallest subproblems and builds up to the original problem.

Tabulation: In the bottom-up approach, you typically use an array or a table to store the results of subproblems. You iteratively fill in this table, ensuring that you solve each subproblem only once and use the results to solve larger problems.

Examples: Dynamic Programming can be applied to a wide range of problems, including the Fibonacci sequence, shortest path problems (like Dijkstra's algorithm), knapsack problems, and many more in fields like computer algorithms, economics, and bioinformatics.

Arrays: Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

Traversing (C++):

```
#include <iostream>

using namespace std;

int main()

{

    int arr[5]={1,2,3,4,5};

    cout<<"The integers in array are:\n";

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

    {

        cout<<arr[i]<<endl;
  }

    return 0;

}
```

Insertion at Specific Position (C++):

```
#include<iostream>

using namespace std;

int main(){

    int arr[10]={1,2,3,4,5};

    int x,p,n=5;

    cout<<"before insertion\n";

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

    {

        cout<<arr[i]<<"\n";
}

   cout<<"Enter the position\n";

   cin>>p;

   cout<<"Enter the number\n";

   cin>>x;

    n++;

    for (int i=n-1; i>=p; i--)

    {
        arr[i]=arr[i-1];
 }

    arr[p-1]=x;

    cout<<"after insertion\n";

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

    {

        cout<<arr[i]<<"\n";
 }

    return 0;
}
```

Array Deletion (At a specific Position)(C++):

```
#include<iostream>

using namespace std;

int main(){

    int arr[10]={10,20,30,40,50,60,70,80,90,100};

    int k,n=10;

    int i,j;

    cout<<"before deletion\n";

     for(int i = 0; i <n; i++)
    {

        cout<<arr[i]<<"\n";

    }

    cout<<"Enter postion you want to delete\n";

    cin>>k;

    j=k;

while (j<10)

    {

        arr[j-1]=arr[j];

        j=j+1;

    }

    n=n-1;

    cout<<"after deletion\n";

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

    {

        cout<<arr[i]<<"\n";

    }

    }
```

Array Search (C++):

```
#include<iostream>

using namespace std;

int main(){

    int arr[10]={1,2,3,4,5,45,12,43,44,67};

    int itr,i,j=0;

    cout<<"given array\n";

     for(int i = 0; i<10; i++)

    {
 cout<<arr[i]<<"\n";
 }

    cout<<"enter the element you want to find  position of\n";

    cin>>itr;

    while (j<10)

    {
if(arr[j]==itr){
Break;
 }
 j=j+1;

    }
    cout<<"found postion\n"<<j+1;

}
```

2-D ARRAYS

Two dimensional array is an array within an array. It is an array of arrays. In this type of array the position of an data element is referred by two indices instead of one. So it represents a table with rows an dcolumns of data.

Eg1.

```
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
```

 11 12 5 2

15 6 10

10 8 12 5

12 15 8 6

 Eg.2

```
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]

print(T[0])

print(T[1][2])
```

Output

```
[11, 12, 5, 2]

10
```

Inserting Values

We can insert new data elements at specific position by using the insert() method and specifying the index.

```
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]

T.insert(2, [0,5,11,13,6])

for r in T:

  for c in r:

        print(c,end = " ")

  print()
```

Output

```
11 12  5  2

15  6 10

 0  5 11 13 6

10  8 12  5

12 15  8  6
```

Updating Values

We can update the entire inner array or some specific data elements of the inner array by reassigning the values using the array index.

```
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
```

```
T[2] = [11,9]

T[0][3] = 7

for r in T:

   for c in r:

          print(c,end = " ")

   print()
```

Output

When the above code is executed, it produces the following result –

```
11 12 5  7

15  6 10

11  9

12 15 8  6
```

Deleting the Values

We can delete the entire inner array or some specific data elements of the inner array by reassigning the values using the del() method with index. But in case you need to remove specific data elements in one of the inner arrays, then use the update process described above.

```
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]

del T[3]

for r in T:

   for c in r:

         print(c,end = " ")

   print()
```

Output

When the above code is executed, it produces the following result –

```
11 12 5 2

15 6 10

10 8 12 5
```

What is a matrix?

A matrix can be defined as a two-dimensional array having 'm' rows and 'n' columns. A matrix with m rows and n columns is called m × n matrix. It is a set of numbers that are arranged in the horizontal or vertical lines of entries.

What is a sparse matrix?

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Why is a sparse matrix required if we can use the simple matrix to store elements?

There are the following benefits of using the sparse matrix -

Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Computing time: In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representation of sparse matrix

The non-zero elements in the sparse matrix can be stored using triplets that are rows, columns, and values. There are two ways to represent the sparse matrix that are listed as follows -

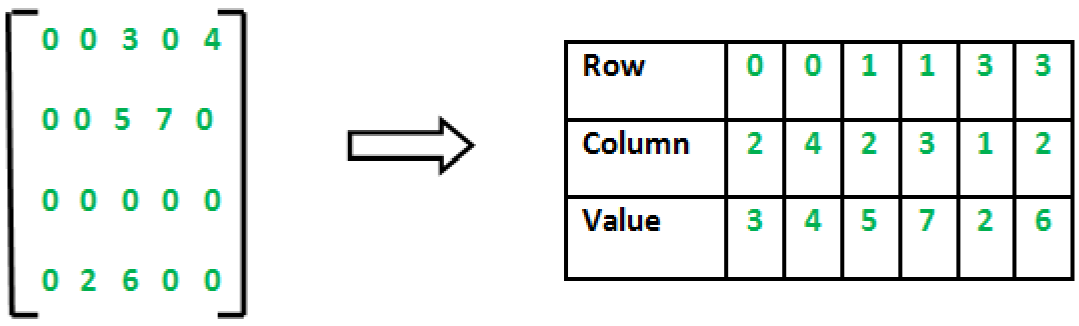
- Array representation
- Linked list representation

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

In 2D array representation of sparse matrix, there are three fields used that are named as -

- **Row -** It is the index of a row where a non-zero element is located in the matrix.
- **Column -** It is the index of the column where a non-zero element is located in the matrix.
- **Value -** It is the value of the non-zero element that is located at the index (row, column).



$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Code in C++:

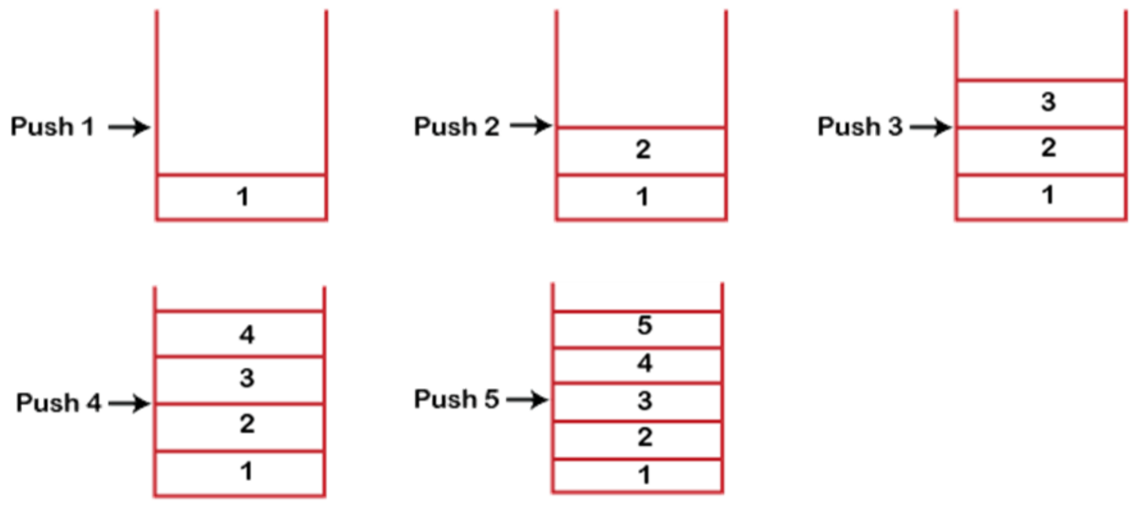
```
#include<iostream>

using namespace std;

int main()

{

     int non_zero_num=0;

    int zero_num=0;

int arr[2][5]={0,0,0,0,0,0,0,8,9,10};

        for(int i=0;i<2;i++)

    {
```

```
    for(int j=0;j<5;j++)

    {

        if(arr[i][j]==0)

        {

            zero_num++;

        }

        else

        {

            non_zero_num++;

        }

    }

    cout<<endl;

}

if ( non_zero_num<zero_num)

{

    cout<<"Given Matrix is Sparse\n";

}

else
```

```
    {

        cout<<"Given Matrix is Dense\n";

    }

    return 0;

}
```

Stacks:

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



1) 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.

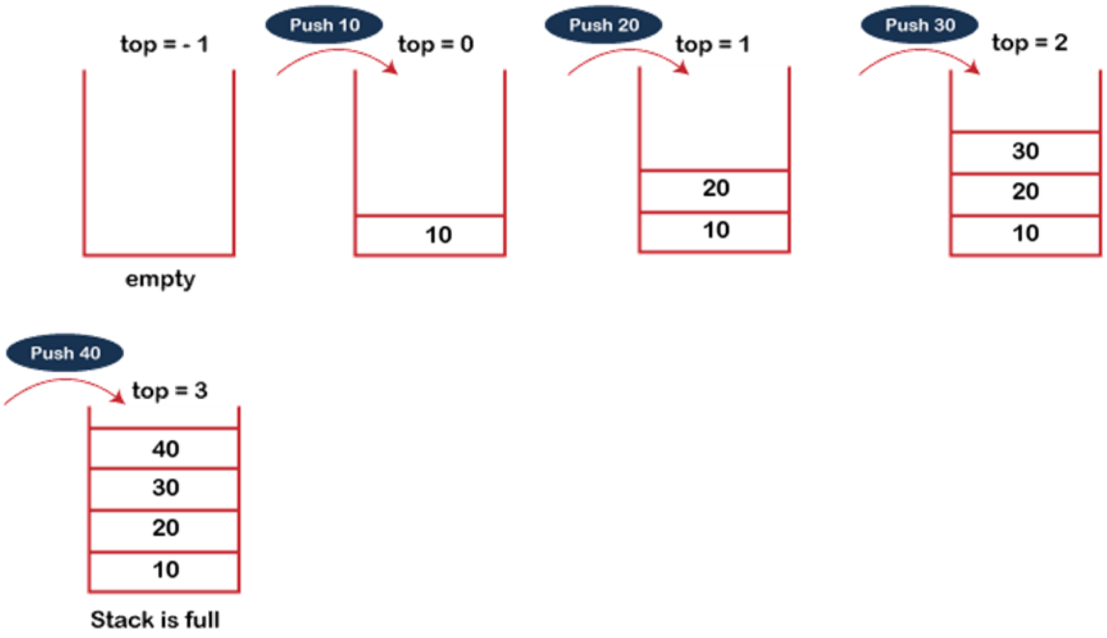
PUSH operation

The steps involved in the PUSH operation is given below:

- 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.

Code Snippet(C++):

```
if (top >= size)

    {

        cout << "Overflow condition\n";

    }

    else

    {

        top++;

        arr[top] = val;

    }
```



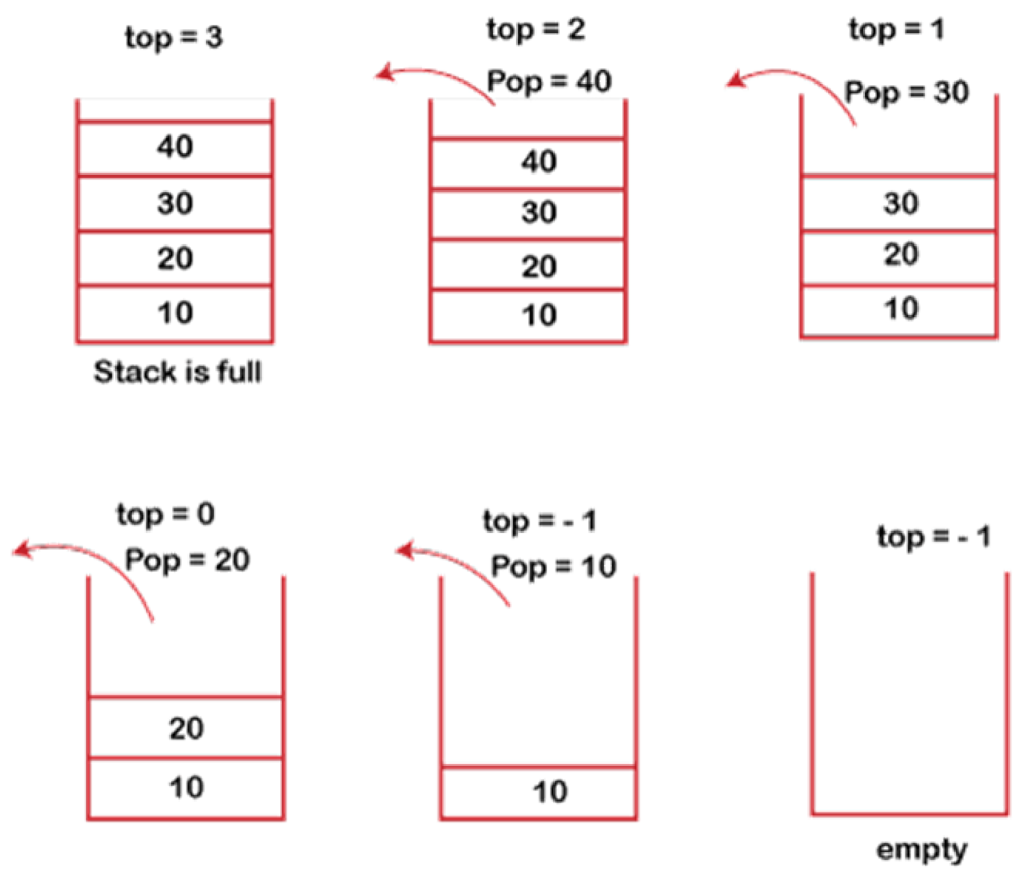
POP operation

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**.

Code Snippet(C++):

```
 if (top==-1)

        {

            cout<<"Underflow\n";

        }

          arr[top] = 0;

        top--;
```



Application of Stacks:

- Function Call Management: Stacks manage function calls in programming.
- Expression Evaluation: Stacks evaluate arithmetic expressions.
- Undo/Redo Functionality: Stacks handle undo and redo in software applications.
- Backtracking: Stacks support path exploration in algorithms.
- Parsing: Stacks help analyse programming languages or data formats.
- Memory Management: Stacks store function call information in systems programming.
- Expression Validation: Stacks check balanced brackets in expressions.
- History in Web Browsers: Stacks manage web page navigation history.
- Task Scheduling: Stacks track tasks in operating systems.
- Postfix Notation Calculators: Stacks evaluate mathematical expressions.
- Algorithm Implementations: Stacks are used in various algorithms.
- Call Stack in JavaScript: Stacks track function calls in JavaScript.
- Undo/Redo in Text Editors: Stacks manage text edit history.

Or

- Function Call Management: Stacks are essential for managing function calls in programming. When a function is called, its local variables, return address, and other relevant data are pushed onto the stack. This allows the program to keep track of function calls and their contexts, ensuring that control can return to the correct location after the function exits.
- Expression Evaluation: Stacks are used to evaluate arithmetic expressions, especially those in postfix (RPN) notation. Operators and operands are pushed onto the stack, and operations are performed when operators are encountered. This simplifies the process of evaluating complex expressions.
- Undo/Redo Functionality: Stacks are employed to implement undo and redo functionality in software applications. Each state change or action is recorded on a stack. When users want to undo an action, the most recent state is popped from the stack, effectively reversing the last action.
- Backtracking: In algorithms like depth-first search (DFS), backtracking is essential to explore different paths or solutions. Stacks are used to store the state of the search at each decision point, allowing the algorithm to backtrack and explore alternative paths when necessary.
- Parsing: Stacks play a crucial role in parsing algorithms used to analyze and interpret the structure of programming languages or data formats. They help keep track of nested structures and enforce proper syntax.
- Memory Management: Stacks are used in memory management, particularly in systems programming. The call stack is used to store information about function calls, including local variables and return addresses. This stack-based approach ensures efficient memory allocation and deallocation.
- Expression Validation: Stacks are employed to validate expressions with brackets, ensuring that they are correctly nested and balanced. An open bracket is pushed onto the stack when encountered, and a close bracket should match the one at the top of the stack.
- History in Web Browsers: Web browsers use stacks to manage navigation history. Each visited URL is pushed onto the history stack, allowing users to go back to previous web pages by popping URLs from the stack.

- Task Scheduling: In operating systems, stacks are used to keep track of tasks, processes, or interrupts that need to be handled. The stack helps manage the execution context for each task or process.
- Postfix Notation Calculators: Stacks are used to evaluate mathematical expressions in postfix notation (Reverse Polish Notation or RPN). Operators and operands are pushed onto the stack, and calculations are performed as operators are encountered.
- Algorithm Implementations: Stacks are an integral part of various algorithms, including depth-first search (DFS) in graph traversal, backtracking algorithms, and more. They provide a convenient way to maintain states and data during algorithm execution.
- Call Stack in JavaScript: In JavaScript, the call stack is implemented as a stack data structure. It keeps track of function calls and their execution contexts, allowing for a better understanding of asynchronous code execution and event handling.
- Undo/Redo in Text Editors: Text editors use stacks to implement undo and redo functionality for text changes. Each text change is pushed onto the stack, making it easy to revert changes and redo them.

Disadvantage of Arrays in Stacks:

- Fixed Size: Arrays have a fixed size.
- Inefficient Insertions/Deletions: Inserting or deleting elements can be inefficient.
- Wasted Memory: Memory may be wasted if not all elements are used.
- Inefficient Search: Searching can be slow in unsorted arrays.
- Lack of Dynamic Sizing: Arrays do not automatically resize.
- Memory Fragmentation: Manual memory management can lead to fragmentation.
- Homogeneous Elements: Arrays store elements of the same data type.
- Access Time Complexity: Searching by key can be slow.
- Not Suited for Sparse Data: Inefficient for sparse data.
- Lack of Built-in Functions: Arrays lack built-in functions for common operations.

REPRESENTATION OF A STACK

Stacks can be represented using various data structures. The two most common ways to represent stacks are using arrays and linked lists. Here's how stacks can be represented using each of these data structures:

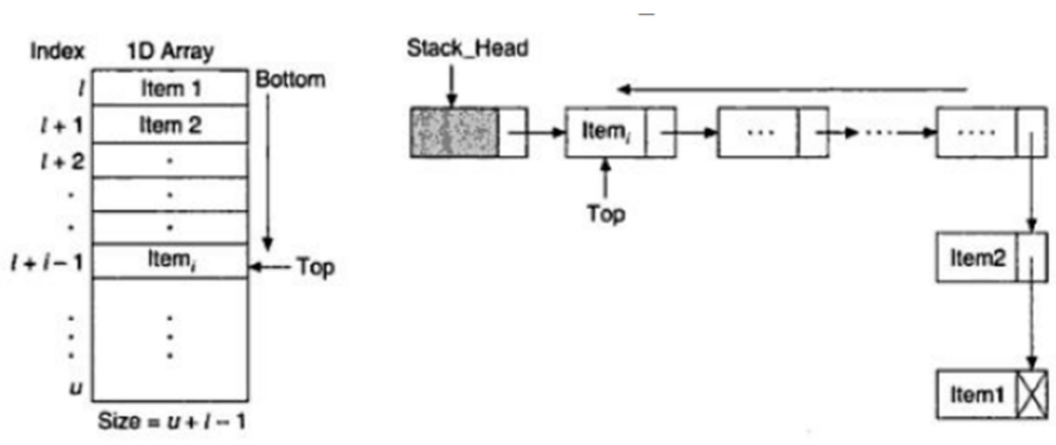
Array Representation of Stacks:

- Allocate a memory block with enough space for the stack.
- Store stack items sequentially in the memory block.
- Use the "TOP" pointer to keep track of the top item.
- Stack is empty if "TOP" is less than the lower index; stack is full if "TOP" is equal to or greater than the upper index.

Linked List Representation of Stacks:

- Represent the stack using a single linked list.
- Each node contains an item in the "DATA" field and points to the next item with the "LINK" field.
- The top of the stack is the first node in the list, and the bottom is the last node.
- Push operation adds a new node to the front of the list, and pop operation removes a node from the front.

In both representations, the essential operations of a stack (push, pop, peek, and size) are supported. The choice between these representations depends on the specific requirements of the application. Arrays are typically more memory-efficient and may be faster for operations like peek, while linked lists offer flexibility in terms of size and are suitable for dynamic data structures.



(a) Array representation of a stack (b) Linked list representation of a stack

Figure 4.3 Two ways of representing stacks.