MODULE – I

Introduction

1.1. Data Structures Introduction:

A data structures is a internal representation of the ordered state that exists between respective data components. In opposite words, data structures describes a method of handle all data items that takes into account not just the pieces contained but also their relationships to one another. The phrase data structures refers to the method by which data is accumulation.

To create an algorithm or program, we must first choose an acceptable data structure for that algorithm. As a result, the data structure is represented as:

Algorithms + Data Structures = Programmes

The elements of a data structures are said to be linear if they form a sequence or a linear list. Linear data structures, such as arrays, stack, queue, and linked list, manage data in a additive fashion. A data structures is said to be nonlinear if its elements create a hierarchic classification with data items appearing at different levels.

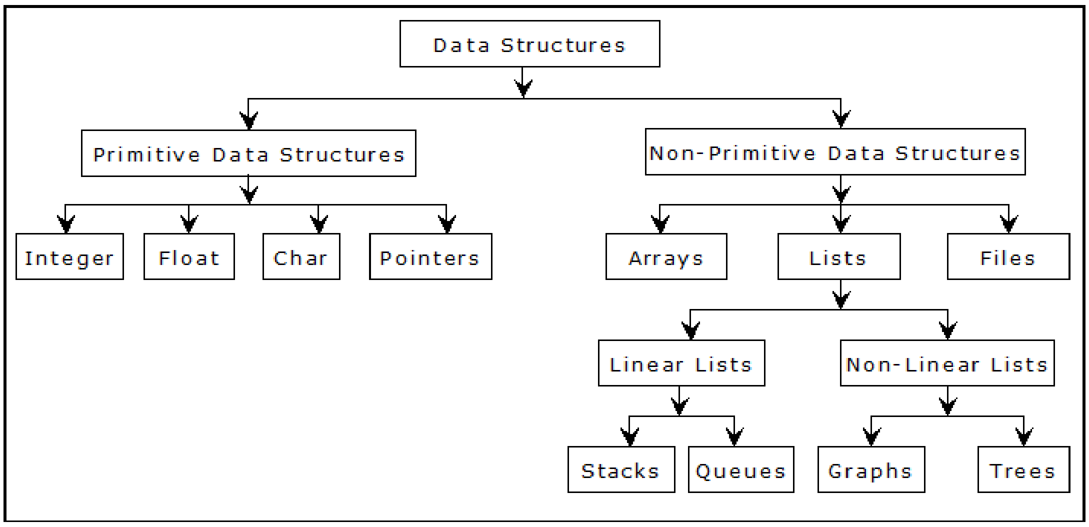
Non-linear data structure such as tree and graph are commonly employed. The hierarchical relationship between individual data pieces is represented by tree and graph structures. Graphs are simply trees with some constraints removed.

There are two kinds of data structures:

1. Data structure that are fundamental (Primitive).
2. Data structure that are not primitive (Non primitive).

Primitive Data Structures are the fundamental data structures that act directly on computer instructions. On different computers, they have different representations. This category includes integers, floating point numbers, character constants, string constants, and pointers.

Non-primitive data structures are more sophisticated than primitive data structures. They place emphasis on grouping similar or dissimilar data items with relationships between them. Arrays, lists, and files all fall within this category.

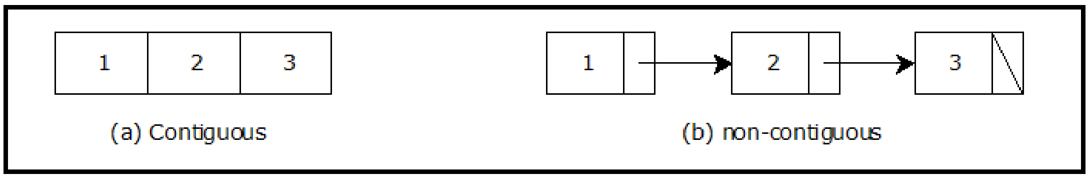


Data Structure Classification

Data structures: Organization of data

A program's data collecting has some form of structure or order to it. No matter how complicated your data structures are, they can be divided into two basic types: Non-contiguous vs. contiguous.

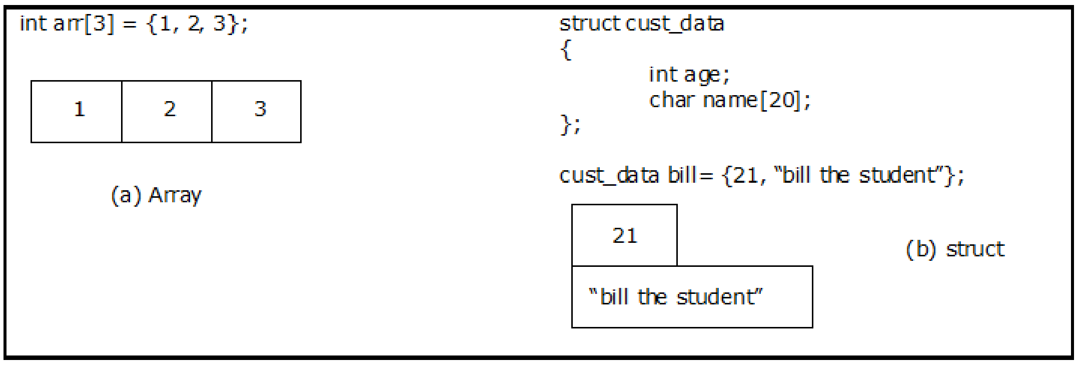
Terms of data are saved together in memory (either RAM or a file) in contiguous structures. A continuous structure is something like an array. Because each array element is adjacent to other elements. Items in a non immediate structure and disconnected throughout memory, on the other hand, were linked to each other in some way. Non-contiguous data structures include linked lists. The list's nodes are linked together via pointers contained in each node.



Contiguous structures:

Contiguous structures are further classified into two types: those containing data item of all the identical size and those containing data items of varying sizes. The first type is known as an arrays. For each one element in an array is of the corresponding type and so has the identical size.

Structure is the second form of contiguous structure. Elements in a struct might be of different data types and thus have varied sizes.



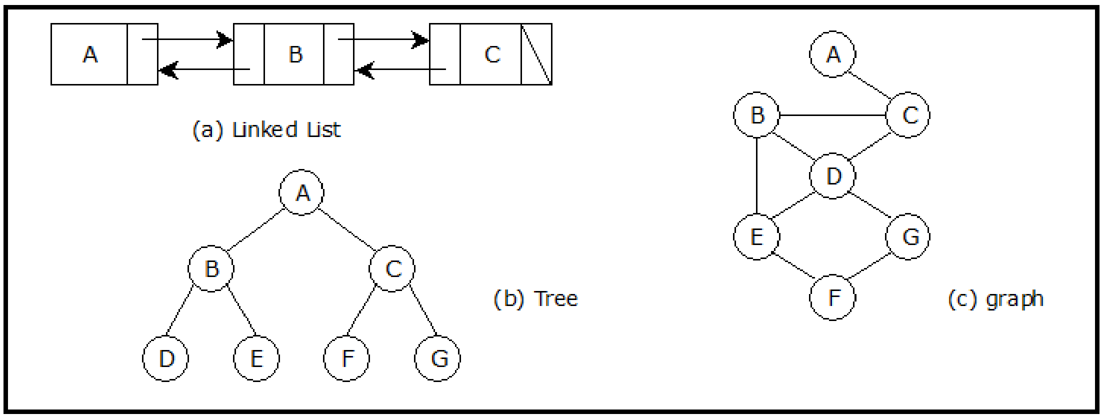
Examples of continuous structure

Non contiguous structures:

Non-contiguous structure is implemented as a set of data-items called nodes, with each node pointing to one or more other nodes in the set. The linked list is the most basic type of non-contiguous structure.

A linked list is a linear, one-dimensional type of non-contiguous structure using only backwards and forwards notation. There is the concept of up and down, as well as left and right.

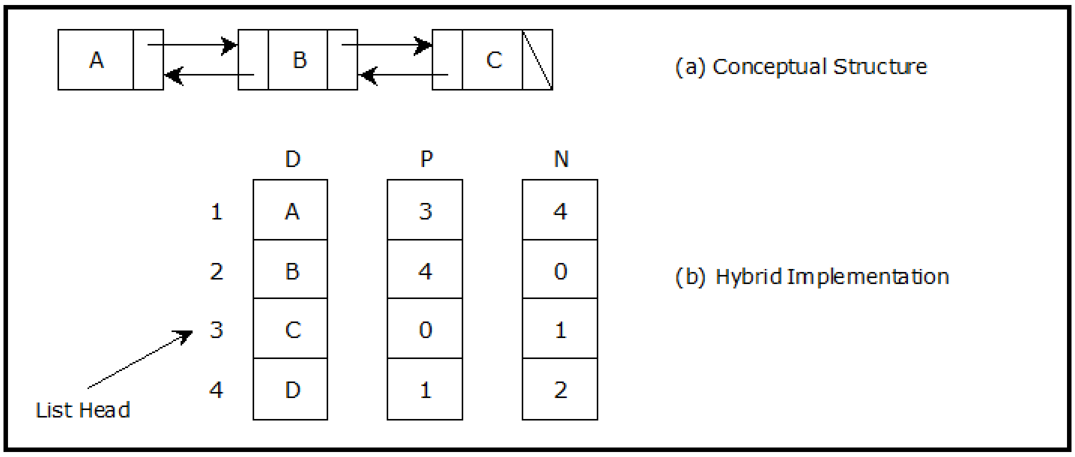
Each node in a trees has only one link that leads into it, and link can only go down the tree. A graph, the most unspecific sort of non contiguous structure, has no such constraints.



Examples of non continuous structure

Hybrid structures:

A hybrid form is formed when two basic types of structure are combined. Then there's a contiguous section and a non contiguous section.



Data Structure type of Hybrid

The array D holds the list's data, while the arrays P and N hold the previous and next "pointers." The pointers are basically just indexes into the D array. For example, D[i] carries the data for node i, but p[i] holds the index to the node before i, which may or may not be at position i-1. N[i] similarly holds the index to the next node in the list.

1.2 Basic Concepts and Notation:

There is a need for displaying higher level data from basic information and structures available at the machine level while constructing a solution in the problem solving process.

There is also a need for algorithm synthesis from fundamental machine-level operations to modify higher-level representations. These two are crucial in achieving the intended result. Data structures are required for data representation, whereas algorithms are required for data operations to yield correct results.

Data

These are truths that have meaning in a certain context and are represented by values, numbers, letters, and symbols.

Information

These are pieces of information that have value to the user or have been interpreted by the user.

When data is analyzed in a given circumstance or utilized to solve a problem, it becomes information.

- Data must be represented (named) or saved in order to be retrieved later.
- Data must be arranged so that it may be accessed selectively and efficiently.

-Data must be processed and displayed in such a way that it effectively supports the user's surroundings.

- Data must be safeguarded and handled in order to keep their worth.

1.3. Abstract Data Type (ADT):

A data structure design entails more than just its organization. You must also plan for how the data will be accessed and processed - that is, how the data will be interpreted. Non-contiguous structures, such as lists, trees, and graphs, can be implemented either contiguously or non-contiguously. Similarly, structures that are normally treated as contiguously, such as arrays and structures, can also be implemented non-contiguously.

The abstract concept of a data structure must be regarded differently than whatever is utilized to create the structure. The abstract concept of a data structure is determined by the operations we intend to execute on the data.

Abstract Data types = Data structures + Methods on the structures

Consideration of both data arrangement and expected operations on data leads to the concept of an abstract data type. An abstract data type is a theoretical construct that includes data as well as operations to be done on the data while concealing implementation.

A stack, for example, is an example of an abstract data type. Stack items can only be added and removed in a specific order - the last thing added is the first item removed. These operations are known as pushing and popping. We haven't indicated how objects are stored on the stack or how they are pushed and popped in this description. Only the valid operations that can be done have been mentioned.

To read a file, for example, we built the code to read the physical file device. That is, we may have to write the same code several times. As a result, we developed what is now known as an ADT. We built the code to read a file and stored it in a library for future use by programmers.

Another example is the code for reading from a keyboard, which is an ADT. It has a data structure, a character, and a set of operations for reading that data structure.

An abstract data type (such as stack) must be implemented before it can be used, which is where data structure comes into play. For example, we might use an array to represent the stack and then specify the relevant indexing methods to perform pushing and popping.

1.5. Algorithm

An algorithm is a finite sequence of instructions, each with a distinct meaning and capable of being completed with a finite amount of work in a finite period of time. An algorithm finishes after processing a finite amount of instructions, regardless of the input values. Furthermore, every algorithm must meet the following requirements:

Input: There are zero or more externally supplied quantities.

Output: At least one amount is produced as an output.

Definiteness: Each instruction must be precise and unambiguous.

Finiteness: If we trace out an algorithm's instructions, the method will always terminate after a finite number of steps.

Effectiveness: Every instruction must be simple enough that it can be carried out by a person utilizing only a pencil and paper. It is not enough for each operation to be specific; it must also be doable.

A distinction is made in formal computer science between an algorithm and a program. The fourth criteria is not always met by a program. One significant example of such a program for a computer is its operating system, which never ends (unless in the case of a system crash) but instead continues in a wait loop until new jobs are entered.

We represent an algorithm using pseudo language, which is a blend of computer language constructs and informal English remarks.

1.6. Analysis and Efficiency of Algorithms:

The selection of an efficient algorithm or data structure is only one aspect of the design process. Following that, we will look at some bigger design challenges. In a program, we should strive for three core design goals:

1. Time complexity: Try to save time.
2. Space complexity: Try to save space.
3. Try to have face.

A faster program is a better program, thus saving time is an apparent goal. A program that saves space over a rival software is also beneficial. We want to "save face" by preventing the application from freezing or producing massive amounts of distorted data.

If 'n' denotes the number of data items to be processed, the degree of polynomial, the size of the file to be sorted or searched, the number of nodes in a graph, and so on.

1: Most programs' next instructions are only executed once or a few times. If all of a program's instructions have this quality, the program's running time is said to be constant.

Log n: When a program's execution time is logarithmic, the program becomes slightly slower as n increases. This running time is frequent in programs that solve a large problem by changing it into a smaller problem, reducing the size by a constant fraction. When n is a million, log n is a constant that doubles every time n doubles, but log n does not double until n reaches n2.

n: When a program's execution time is linear, only a tiny amount of processing is performed on each input piece. This is the ideal circumstance for an algorithm with n inputs to process.

n.logn: This occurs for algorithms that solve a problem by dividing it down into smaller sub-problems, solving them separately, and then combining the solutions. The running time more than doubles when n doubles.

n^2: When an algorithm's execution time is quadratic, it can only be used on relatively modest problems. Quadratic running times are common in algorithms that process all pairs of data items (perhaps in a double nested loop); when n doubles, the running time quadruples.

n^3: Similarly, a method that processes triples of data items (perhaps in a triple-nested loop) has a cubic running time and is only applicable to modest tasks. When n doubles, the running time increases by an order of magnitude.

2^n: Few algorithms with exponential running times are likely to be useful in practice; such algorithms naturally emerge as "brute-force" solutions to issues.
The running time squares whenever n doubles.

1.7. Time and Space Complexity:

A program's performance is defined as the amount of computer memory and time required to run it. To assess a program's performance, we employ two methods. The first is analytical, whereas the second is experimental. We employ analytical methods in performance analysis and experiments in performance assessment.

Time Complexity:

The time required by an algorithm expressed as a function of issue size is referred to as the algorithm's TIME COMPLEXITY. A program's time complexity is the amount of computer time required to complete it.

Asymptotic temporal complexity refers to the limiting behavior of complexity as size grows. The asymptotic complexness of an algorithm eventually dictates the property of issues that the algorithm can solve.

Space Complexity:
A program's space complexity is the quantity of memory required to test it to ending.
A program's space requirements include the following elements:

Instruction space is the amount of space required to hold the compiled version of the program instructions.

Data space is the amount of space required to store all constant and variable values.

Data space is divided into two parts:

1. space required by constants and simple variables in programs.

2.Dynamically allocated objects, such as arrays and class instances, require space.

Environment stack space is used to save information required to resume execution of partially completed functions.

The quantity of instructions space required is determined by factors such as:

1. The compiler used to convert the program into machine code.
2. The compiler options that were in effect at the time of compilation.

Complexity of Algorithms

An algorithm's complexity M is the function f(n) that gives the algorithm's execution time and/or storage space need in terms of the size 'n' of the input data. Typically, an algorithm's storage space need is just a multiple of the data size 'n'. The running time of the algorithm is referred to as its complexity.

An algorithms asymptotic analytic thinking pertains to defining the numerical boundation / framing of its run time performance. We may very well deduce the best case, average case, and worst case scenarios of an algorithm using asymptotic analysis.

Asymptotic analysis is input bound, which means that if the algorithm receives no input, it is assumed to work in a constant time. Except for the "input," all other elements are assumed to be constant.

The computations of the run time of any activity in mathematical units of computation is referred to as asymptotic analysis. For representation, the run time of one activity may be computed as f(n), whereas the run time of another activity may be computed as g(n2).

This means that the first operation's running time will increase linearly as n increases, while the second operation's running time will increase exponentially as n increases. Similarly, if n is really small, the running time of both operations will be virtually the same.

There are three sorts of time required by an algorithm.

Minimum time necessary for program execution in the best-case scenario.

The average amount of time taken for program execution in the Average case scenario.

Maximum time necessary for program execution in the worst-case scenario.

Asymptotic Notations

The below are some common asymptotic-notations for calculating an algorithms run time complexity.

Big -O- Notation
Big- Ω -Notation
Big- θ -Notation

Big-Oh-Notation (O):

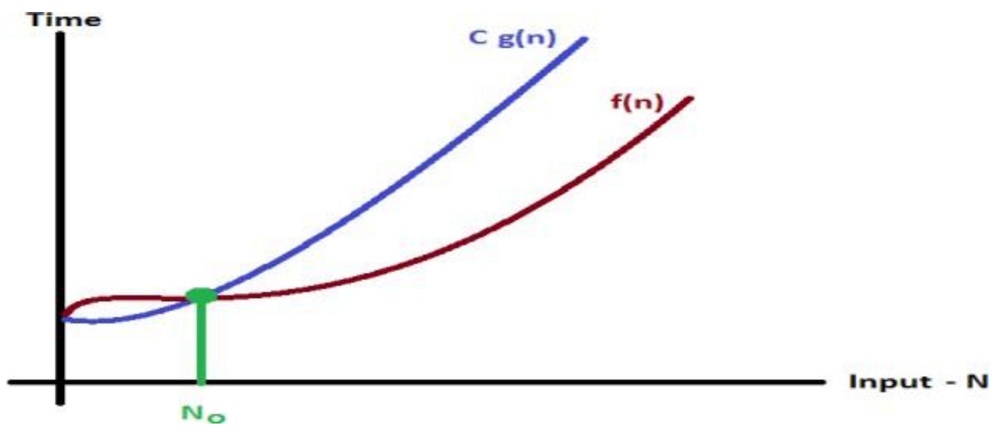
1.Big - Oh notation is used to define an algorithm's upper bound in terms of Time Complexity.

2.Big - Oh notation always represents the maximum time an algorithm requires for all input values.

3.Big - Oh notation describes the worst-case time complexity of an algorithm.

Big - Oh Notation is defined as follows... The graph below depicts the values of f(n) and C g(n) for input (n) value on the X-Axis and time required on the Y-Axis time complexity of an algorithm.

In the graph below, for a certain input value n0, C g(n) is always bigger than f(n), indicating the algorithm's upper bound.



Example

Consider the f(n) and g(n) expressions...

f(n) = 3n + 2

g(n) = n

If we want to describe f(n) as O(g(n)), it must satisfy f(n) = C g(n) for any C > 0 and n0 > 1 values.

f(n) <= C g(n)

⇒3n + 2 <= C n

For all values of C = 4 and n >= 2, the above condition is always TRUE.

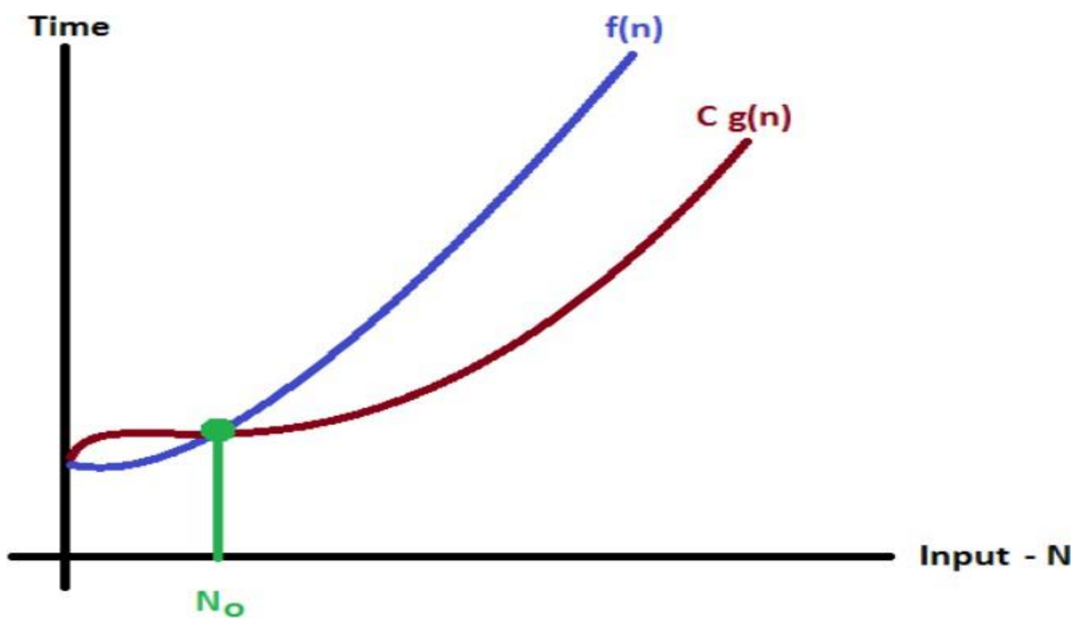
We can represent the time complexity as follows using Big - Oh notation-

3n + 2 = O(n).

Big - Omege-Notation (Ω):

1.Big - Omega notation is used to define an algorithm's lower bound in terms of Time Complexity.
2. Big-Omega notation always represents the shortest time an algorithm requires for all input values.
3. Big-Omega notation defines the best-case temporal complexity of an algorithm.

The graph below depicts the values of f(n) and C g(n) for input (n) values on the X-Axis and time required on the Y-Axis.



After a specific input value n0, C g(n) is always less than f(n), indicating the algorithm's lower bound.

Example
Consider the f(n) and g(n) expressions...
3n + 2 f(n) = n g(n) = n
If we want to describe f(n) as (g(n)), it must meet the following conditions: f(n) >= C g(n) for all values of C > 0 and n0 >= 1 f(n) >= C g(n) 3n + 2 >= C n

For all values of C = 1 and n >= 1, the above condition is always TRUE.
The temporal complexity can be represented using Big - Omega notation as
3n + 2 = (n).

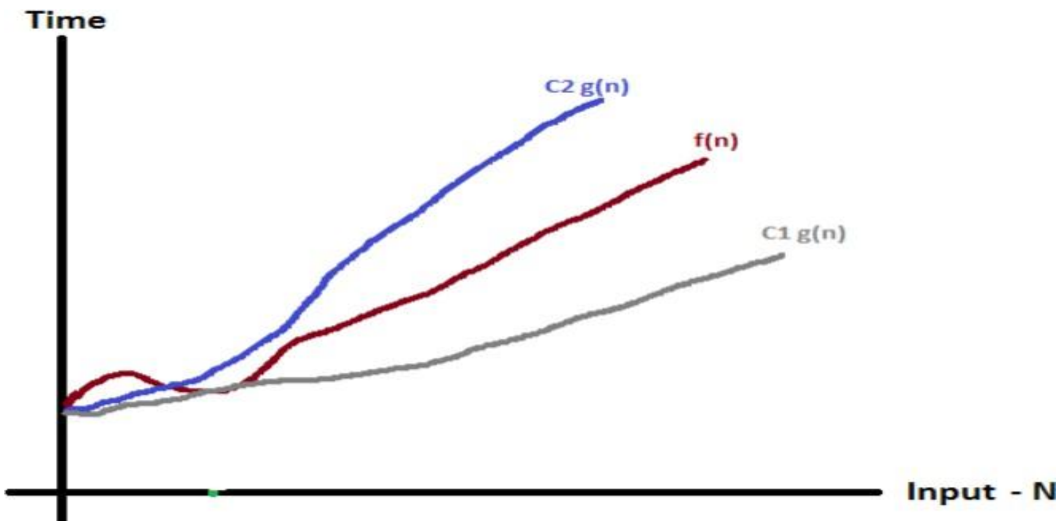
Big - Theta-Notation (Θ):

1.Big - Theta notation is used to define an algorithm's average bound in terms of Time Complexity.

2.Big - Theta notation always reflects an algorithm's average time for all input values.

3.Big - Theta notation describes an algorithm's average time complexity.

The graph below depicts the values of f(n) and C g(n) for input (n) values on the X-Axis and time required on the Y-Axis.



After a certain input value n0, C1 g(n) is always less than f(n), and C2 g(n) is always bigger than f(n), indicating the algorithm's average bound.

Example

Consider the f(n) and g(n) expressions...

3n + 2 f(n) = n g(n) = n

If we want to describe f(n) as (g(n)), it must meet C1 g(n) = f(n) = C2 g(n) for all values of C1 > 0, C2 > 0 and n0 >= 1 C1 g(n) = f(n) = C2 g(n)

For all values of C1 = 1, C2 = 4, and n >= 2, the above condition is always TRUE.

We can represent the time complexity as follows using Big - Theta notation…

3n + 2 = Θ(n).

Searching Techniques

1. Linear Search:

- **Definition:** Linear Search is a simple search algorithm that sequentially checks each element in a list or array until a match is found or the entire list has been traversed.
- **Explanation:** Linear Search is like searching for a specific item in an unordered list by checking each element one by one from the beginning until you either find the target element or reach the end of the list. It's a straightforward but not very efficient way to search for an item in a collection.
  - **Algorithm for Linear Search:**

```
#include <stdio.h>

int linearSearch(int arr[], int n, int target)
  { for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
        return i; // Return the index if the target element is found
      }
  }
  return -1; // Return -1 if the target element is not in the array
}
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  int n = sizeof(arr) / sizeof(arr[0]);
  int target = 3;
  int result = linearSearch(arr, n, target);
  if (result != -1) {
      printf("Element found at index: %d\n", result);
  } else {
      printf("Element not found in the array.\n");
  }
  return 0;
}
```

  - **Linear Search Time Complexity Analysis:**

    Time Complexity: O(n) - Linear Search checks each element in the array exactly once in the worst case, so its time complexness is linear with respect to the size of the array (n).

2. Binary-Search:

- **Definition:** Binary Search is a fast search algorithm that works on sorted lists or arrays. It repeatedly divides the search interval to half until the target element is found or the search interval is empty.
- **Explanation:** Binary Search is an efficient algorithm for finding a specific item in a sorted list or array. It starts by comparing the target value with the middle element and narrows down the search range by half in each iteration.

Algorithm for Binary Search:

```
#include <stdio.h>
int binarySearch(int arr[], int left, int right, int target)
  { while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) {
      return mid; // Return the index if the target element is found
    }
    if (arr[mid] < target)
      { left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return -1; // Return -1 if the target element is not in the array
}

int main() {
  int arr[] = {10, 21, 32, 43, 54};
  int n = sizeof(arr) / sizeof(arr[0]);
  int target = 32;
  int result = binarySearch(arr, 0, n - 1, target);
  if (result != -1) {
    printf("Element found at index: %d\n", result);
  } else {
    printf("Element not found in the array.\n");
  }
  return 0;
}
```

- **Binary-Search Time Complexity Analysis:**

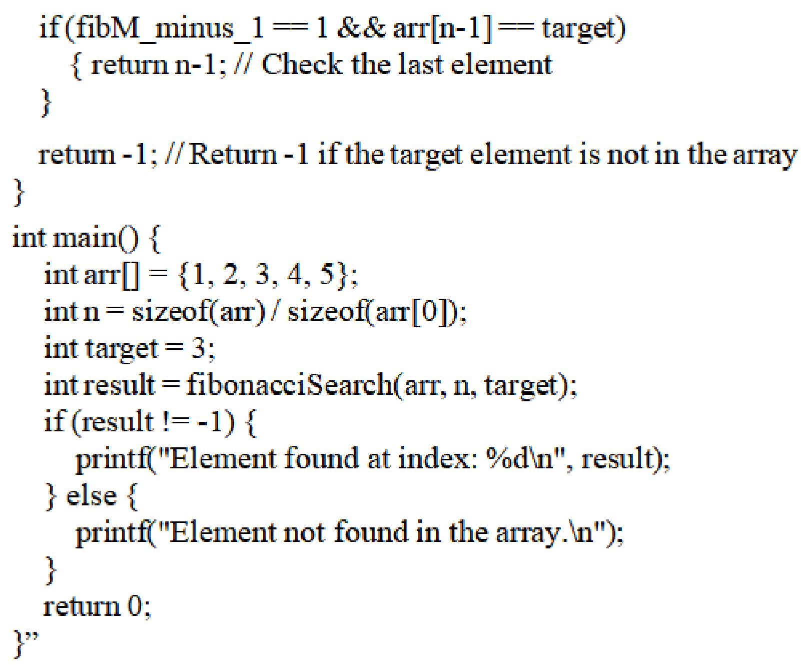
  Time Complexness: O(log n) - Binary-Search repeatedly divides the search interval in half, so its time complexity is logarithmic with respect to the size of the array (n).

3. Fibonacci Search:

- **Definition:** Fibonacci Search is an algorithm for searching a sorted array using a divide & conquer approach. It uses Fibonacci series numbers to determine the split points in the array.
- **Explanation:** Fibonacci Search divides the search interval into smaller sub intervals using Fibonacci numbers. It is similar to Binary Search but uses a more complex formula for selecting the split point.

Algorithm for Fibonacci Search:

```
#include   <stdio.h>

int min(int a, int b) {
   return (a < b) ? a : b;}
int fibonacciSearch(int arr[], int n, int target)
   { int fibM_minus_2 = 0;
   int fibM_minus_1 = 1;
   int fibCurrent = fibM_minus_1 + fibM_minus_2;
//Finding Fibonacci number that is greater than or equal to
   n while (fibCurrent < n) {
      fibM_minus_2 = fibM_minus_1;
      fibM_minus_1 = fibCurrent;
      fibCurrent = fibM_minus_1 + fibM_minus_2;
   }
   int offset = -1;
   while (fibCurrent > 1) {
      int i = min(offset + fibM_minus_2, n - 1);
     if (arr[i] < target) {
         fibCurrent = fibM_minus_1;
         fibM_minus_1 = fibM_minus_2;
         fibM_minus_2 = fibCurrent - fibM_minus_1;
         offset = i;
      } else if (arr[i] > target)
         { fibCurrent =
         fibM_minus_2;
         fibM_minus_1 = fibM_minus_1 - fibM_minus_2;
         fibM_minus_2 = fibCurrent - fibM_minus_1;
      } else {
         return i; // Return the index if the target element is found
      }}
```



```
    if (fibM_minus_1 == 1 && arr[n-1] == target)
      { return n-1; // Check the last element
    }
    return -1; // Return -1 if the target element is not in the array
  }
  int main() {
     int arr[] = {1, 2, 3, 4, 5};
     int n = sizeof(arr) / sizeof(arr[0]);
     int target = 3;
     int result = fibonacciSearch(arr, n, target);
     if (result != -1) {
        printf("Element found at index: %d\n", result);
     } else {
        printf("Element not found in the array.\n");
     }
     return 0;
  }"
```

Example: let n=11, x=85

i	1	2	3	4	5	6	7	8	9	10	11
arr[i]	10	22	35	40	45	50	80	82	85	90	100

Following is the Fibonacci numbers table for acknowledgment.

	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
Fib.no	0	1	1	2	3	5	8	**13**	21	34	55

Smallest Fibonacci number greater than or equal to 11 is 13.
Therefore fibM = 13, fibMm1 = 8, and fibMm2 = 5

offset=-1

fibM	fibMm1	fibMm2	offset	i	Arr[i]	Inference
13	8	5	-1	4	45	Move down by one, reset offset
8	5	3	4	7	82	Move down by one, reset offset
5	3	2	7	9	90	Move down by two
2	1	1	9	8	85	return i

- **Fibonacci Search Time Complexity Analysis:**

  Time Complexness: O(log n) - Fibonacci-Search also has a logarithmic time complexness similar to Binary-Search.

Each of the algorithms for Linear Search, Binary Search, and Fibonacci Search has its own characteristics and use cases, with Binary Search being the most efficient for sorted data, while Linear Search is simple but less efficient. Linear-Search has a linear time complexness, while Binary-Search and Fibonacci-Search both have logarithmic time complexity. Binary Search is generally preferred for searching in sorted arrays due to its simplicity and efficiency. Fibonacci Search is less commonly used.

Sorting Techniques:

1. Bubble Sort Algorithm

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list to be sorted, comparing adjacent elements and swapping them if they are in the wrong order. The process continues until no more swaps are needed, indicating that the list is sorted.

Steps to Implement Bubble Sort:

1. **Start from the first element** of the array.
2. **Compare each pair of adjacent elements**.
   - If the pair is in the wrong order (i.e., the first element is greater than the second), swap them.
3. **Move to the next pair** and repeat the comparison.
4. After each pass through the array, the largest unsorted element "bubbles" up to its correct position.
5. **Repeat the process** for the remaining unsorted elements.
6. Stop when no swaps are needed during a new pass through the list.

Pseudocode:

plaintext
CopyEdit

```
BubbleSort(arr)
    n = length of arr
    for i = 0 to n-1
        for j = 0 to n-i-2
            if arr[j] > arr[j+1]
                swap arr[j] and arr[j+1]
```

Example:

Let's sort the array: [5, 3, 8, 4, 2]

1. **First Pass:**
   - Compare 5 and 3: swap → [3, 5, 8, 4, 2]
   - Compare 5 and 8: no swap → [3, 5, 8, 4, 2]
   - Compare 8 and 4: swap → [3, 5, 4, 8, 2]
   - Compare 8 and 2: swap → [3, 5, 4, 2, 8]
   - After the first pass, the largest element 8 is at the end.
2. **Second Pass:**
   - Compare 3 and 5: no swap → [3, 5, 4, 2, 8]
   - Compare 5 and 4: swap → [3, 4, 5, 2, 8]
   - Compare 5 and 2: swap → [3, 4, 2, 5, 8]
   - After the second pass, the second-largest element 5 is now at its correct position.
3. **Third Pass:**
   - Compare 3 and 4: no swap → [3, 4, 2, 5, 8]
   - Compare 4 and 2: swap → [3, 2, 4, 5, 8]
   - After the third pass, 4 is in its correct position.
4. **Fourth Pass:**
   - Compare 3 and 2: swap → [2, 3, 4, 5, 8]

- After the fourth pass, the list is completely sorted.

Sorted Array: [2, 3, 4, 5, 8]

Time Complexity:

- **Best case:** O(n)O(n)O(n) when the list is already sorted.
- **Average case:** O(n2)O(n^2)O(n2), because we compare and swap each pair of adjacent elements for each element.
- **Worst case:** O(n2)O(n^2)O(n2), when the array is sorted in reverse order.

Space Complexity:

- **Space Complexity:** O(1)O(1)O(1), as Bubble Sort is an in-place sorting algorithm (it does not require additional space).

Advantages of Bubble Sort:

1. **Simple to understand and implement.**
2. **In-place sorting** (does not require extra space).
3. It is **adaptive** in the best case (if the array is already sorted).

Disadvantages of Bubble Sort:

1. **Inefficient for large datasets** because its average and worst-case time complexity is O(n2)O(n^2)O(n2).
2. It requires many comparisons and swaps, making it slower compared to more efficient algorithms like Quick Sort or Merge Sort.

2. Selection-Sort:

- **Definition:** Selection Sort is a simple comparison-based sorting algorithm. It repeatedly selects the minimum (or maximum) element from the unsorted portion of the array and places it at the beginning (or end) of the sorted portion.
- **Explanation:** Selection Sort divides the array into two parts: the sorted part on the left and the unsorted part on the right. It repeatedly finds the minimum (or maximum) element from the unsorted part and swaps it with the first element of the unsorted part. This process continues until the entire array is sorted.
- **Time Complexity:** O(n^2) in the worst and average cases, where n is the number of elements in the array.

- **Algorithm for Selection Sort:**

```
#include <stdio.h>
void selectionSort(int arr[], int n)
   { int i, j, minIndex, temp;
   for (i = 0; i < n - 1; i++) {
      minIndex = i; // Assume the current index contains the minimum element
      // Find the index of the minimum element in the unsorted part of the array
      for (j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIndex])
            { minIndex = j;
         }}
      // Swap the minimum element with the element at the current index
      temp = arr[i];   arr[i] = arr[minIndex]; arr[minIndex] = temp;
   }}
int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Unsorted array: ");
    for (int i = 0; i < n; i++)
      { printf("%d ", arr[i]);
   }
   selectionSort(arr, n);
   printf("\nSorted array: ");
    for (int i = 0; i < n; i++)
      { printf("%d ", arr[i]);
   }return 0;
}
```

In this code:

- Selection-Sort is a function that takes an integral array arr and its length n as parameters and sorts the array using the Selection-Sort Algorithm.
- The outer loop iterates through each element of the array from left to right, considering them as the minimum element initially.
- The inner loop finds the index of the minimum element in the unsorted part of the array.
- When the minimum element is found, it is swapped with the element at the current index of the outer loop.
- This process continues until the entire array is sorted.

The main utility shows the usage of the selection-Sort function on an example array.

After sorting, the array will be in ascending order.

3. Insertion Sort:

- **Definition:** Insertion Sort is a simple comparison-based sorting algorithm. It builds the sorted portion of the array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position within the sorted part.
- **Explanation:** Insertion Sort starts with a single element (the first element) considered as the sorted portion. It then iterates through the unsorted part, taking one element at a time and inserting it into the correct position within the sorted portion. This process continues until the entire array is sorted.
- **Time Complexity:** O(n^2) in the worst and average cases, making it suitable for small datasets or nearly sorted datasets.

Example:

Original	**34**	**8**	**64**	**51**	**32**	**21**	**Positions Moved**
After p=1	8	34	64	51	32	21	1
After p=2	8	34	64	51	32	21	0
After p=3	8	34	51	64	32	21	1
After p=4	8	32	34	51	64	21	3
After p=5	8	21	32	34	51	64	4

The Insertion Sort algorithm is a simple and intuitive sorting algorithm that is particularly efficient for small data sets. It works by building a sorted array one element at a time, by repeatedly taking the next element from the unsorted part of the array and inserting it into its correct position in the sorted part. This process continues until the entire array is sorted.

When you run this code, it will output the original array followed by the sorted array.

- **Algorithm for Insertion Sort:**

```
“#include <stdio.h>

void insertionSort(int arr[], int n)
   { int i, temp, j;
   for (i = 1; i < n; i++)
      { temp = arr[i];
      for(j=i; j > 0 && arr[j-1] > temp; j--)
         { arr[j] = arr[j-1];
       }
      arr[j] = temp;
   }
}

int main() {
   int arr[] = {34, 8, 64, 51, 32, 21};
   int n = sizeof(arr) / sizeof(arr[0]);

   insertionSort(arr, n);

   printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
     { printf("%d ", arr[i]);
   }
   printf("\n");
   return 0;
}”
```

4. Bubble-Sort:

- **Definition:** Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly compares adjacent elements in the array and swaps them if they are in the wrong order, gradually pushing the largest elements to the end of the array.
- **Explanation:** Bubble Sort repeatedly traverses the array, comparing adjacent elements, and swapping them if they are out of order. This process continues until no more swaps are needed, indicating that the array is sorted.
- **Time Complexity:** O(n^2) in the worst and average cases, making it suitable for small datasets.

- **Algorithm for Bubble Sort:**

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
  { int temp;
  int swapped;
  for (int i = 0; i < n - 1; i++) {
    swapped = 0; // Flag to optimize when the array is already sorted
    for (int j = 0; j < n - i - 1; j++) {
      // Compare adjacent elements and swap them if they are in the wrong order
      if (arr[j] > arr[j + 1]) {
        temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
        swapped = 1; // Set the flag to indicate a swap occurred
      }}
    // If no two elements were swapped in the inner loop, the array is already sorted
    if (swapped == 0) {
      break;
    }}}
    int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("Original Array: ");
   for (int i = 0; i < n; i++)
    { printf("%d ", arr[i]);
  }
  bubbleSort(arr, n);
  printf("\nSorted Array: ");
  for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }return 0;}
```

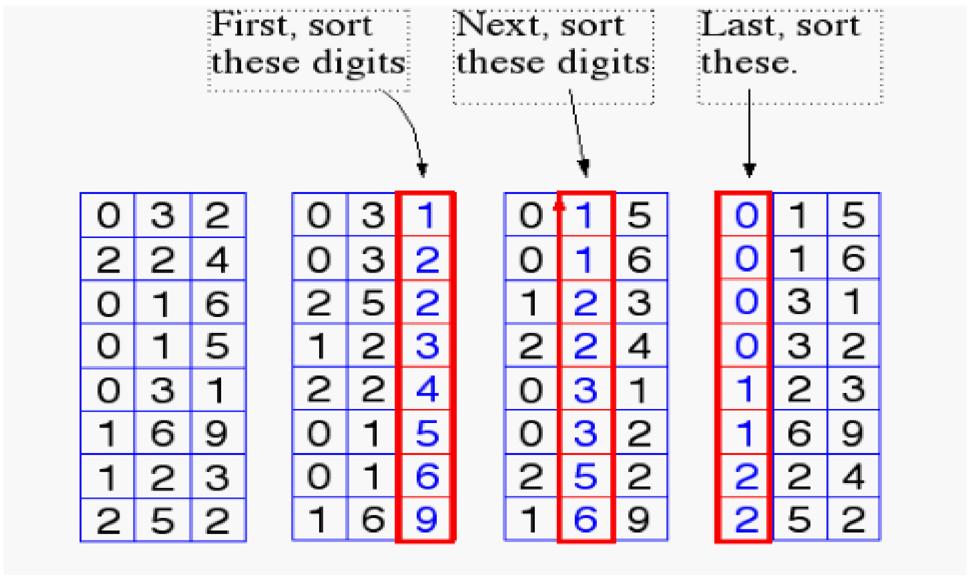
This code defines the bubble-Sort function, which takes an array of integers arr and its length , and sorts the array in ascending order using the Bubble Sort algorithm. The main function demonstrates how to use this sorting function on an example array.

When you run this code, it will output the original array followed by the sorted array.

5. Radix Sort or Bin Sort or Bucket Sort:

- **Definition:** Radix-Sort is a non comparable sorting algorithm that works by forwarding individual digits of the numbers in the array, from the (Minimum)Least significant digit (LSD) to the (Maximum)Most significant digit (MSD) or vice versa.
- **Explanation:** Radix Sort processes the array by considering the digits of the numbers. It starts by sorting based on the minimum significant digit and proceeds to the maximum significant digit.
- **Time Complexity:** O(n * k) in the worst, average, and best cases, where N is the number of components & K is the number of digits in the maximal number. It is efficient when k is small compared to n.

Radix Sort Example:



Radix-Sort Algorithm

The radix sort algorithm makes use of the counting sort algorithm while sorting in every phase. The detailed steps are as follows −

Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits (say k) in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

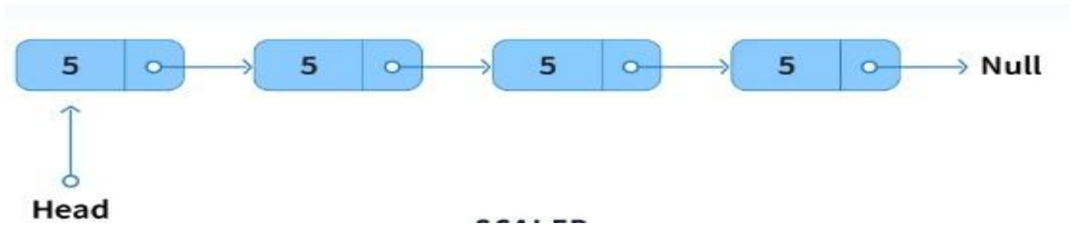
Step 5 − The final list of elements achieved after kth loop is the sorted output.

```
#include <stdio.h>
#include <stdlib.h>

// Function to get the maximum value in the array
int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

// Function to perform counting sort based on the digit represented by exp
void countingSort(int arr[], int n, int exp) {
    int output[n]; // output array to store sorted numbers
    int count[10] = {0}; // count array to store the frequency of digits (0-9)

    // Count the occurrences of each digit
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // Modify the count array such that count[i] contains the actual position of this digit in output[]
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Build the output array by placing the elements in the correct position
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the sorted output array to arr[], so that arr[] now contains the sorted numbers
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

// Main function to implement Radix Sort
void radixSort(int arr[], int n) {
    // Find the maximum number to determine the number of digits
    int max = getMax(arr, n);

    // Apply counting sort for every digit. The exp is 10^i where i is the current digit we are sorting by.
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
voids main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: \n");
    printArray(arr, n);

    radixSort(arr, n);

    printf("Sorted Array: \n");
    printArray(arr, n);

    }
```

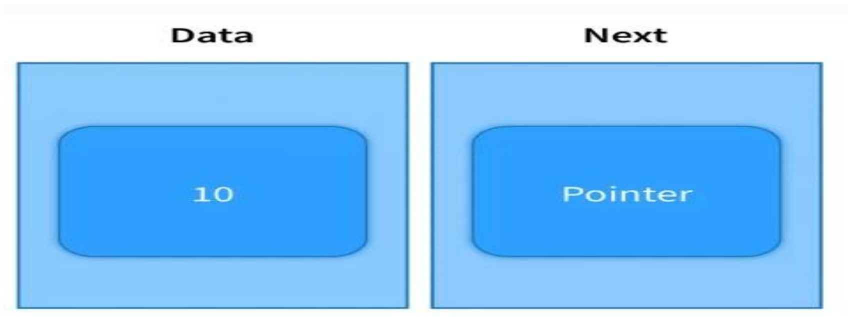
MODULE – 2

<u>LINKED-LIST</u>

A linked-list is a grouping of elements where each entry points to the one that comes after it in the sequence. A node is any one of the Linked List's elements. Multiple links are connected to form a Linked List.

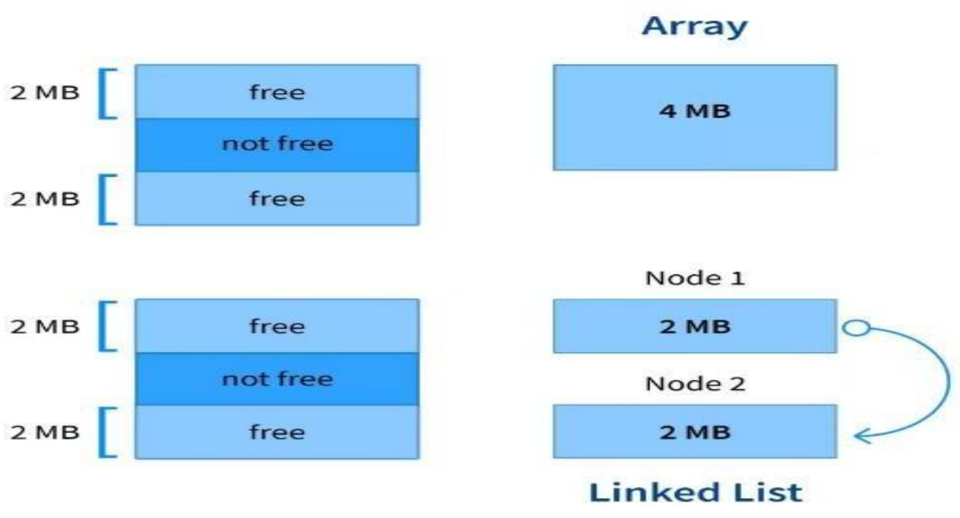


A node is composed of two components:
1. Data: This is the real, unprocessed information you wish to keep up to date. It could be a character, a number, a video item, etc.
2. Pointer to another node: It keeps track of the neighbouring node's link.



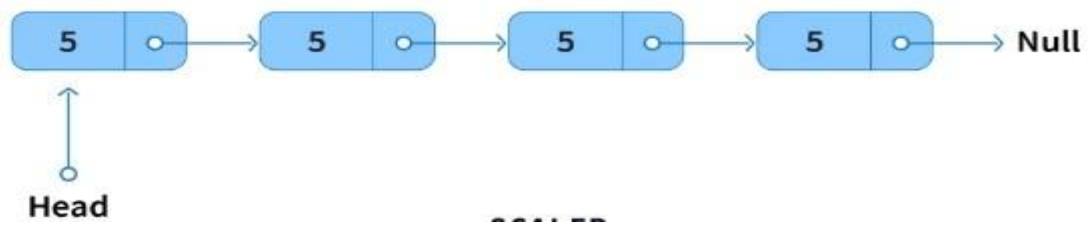
linked list over an array:

Dynamic Dimensions Arrays are limited in size. An array cannot be changed after its size is specified. On the other hand, the linked list's size is modifiable during runtime. Memory Consumption: Memory must be allocated to arrays in a continuous fashion. Memory for linked lists, however, can be dynamically allocated. A linked list's nodes are all connected by pointers and kept in non- contiguous memory regions.
For example: consider system has 4 MB of free space. Yet, the open area is not continuous. Currently, 4MB of linked lists can be stored here with ease, but 4MB of arrays cannot. Therefore, linked lists make the most of memory.
Quick Insertion and Removal of array, the period of time complexness of deletion and inserting is O(n). On the other hand, O(1) can be used for deletion and insertion in linked lists.



A single linked list: what is it?

A singly linked list is a particular example of a generic linked list. If we start at the first node in the list, we can only go in one way because every node in a singly linked list links to only the next node in the sequence..



The alone subsequent element's address is kept apart from the contents in the single linked list's node. This is how a node appears in its C representation

```
" Node{
 int data;
 Node next;
} "
```

.

A unique pointer called "head" for each linked list . This pointer contains the memory address of the initial node in the list. The following element can no thirster be present on the last node. Therefore, we designate NULL to the next element in the linked list to denote its termination.

1. Inserting in a single linked list

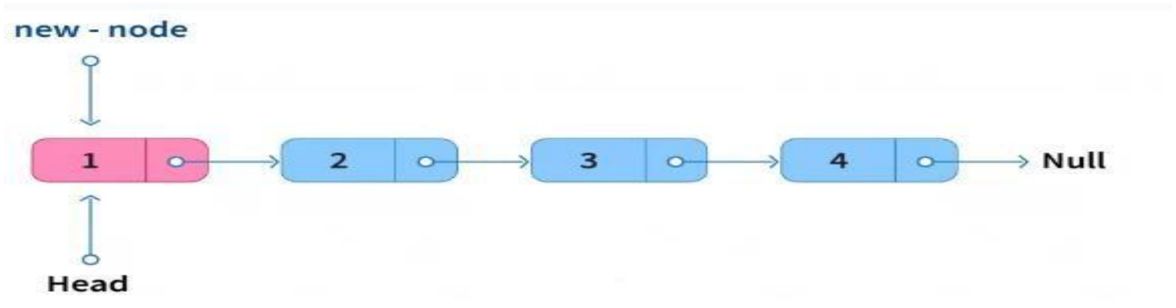
The following methods can be used to insert data into a single linked list:

Insertion at the outset To add a new node to the start of a singly linked list, perform these steps:.

Point the newly created node at HEAD.

Set the new node's HEAD to refer there.

O(1) to insert a new node at the beginning.



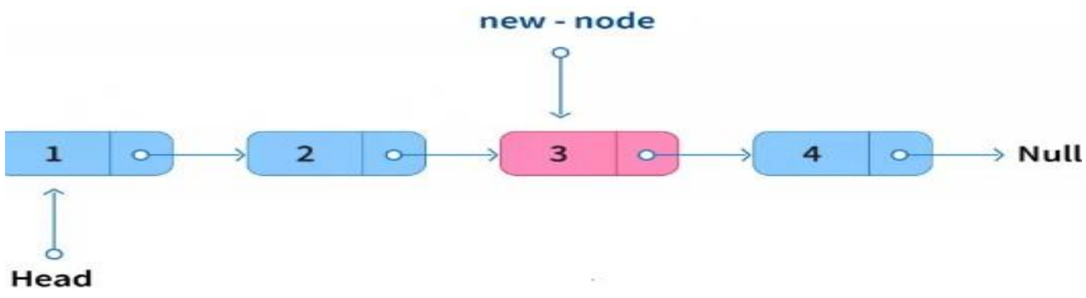
```
"void insertAtStart(Node newNode, Node head){
 newNode.data = 10;
 newNode.next = head;
 head.next = newNode;
}"
```

• Insertion following a few Nodes The process of adding a new node to a singly linked list after an existing node is as follows:

Insert the new node once you've reached the intended node.

o Configure the new node to point to the element that comes after the existing node.

o Aim the new node directly at the current node. Adding a new node after an existing one is an O(N) operation.

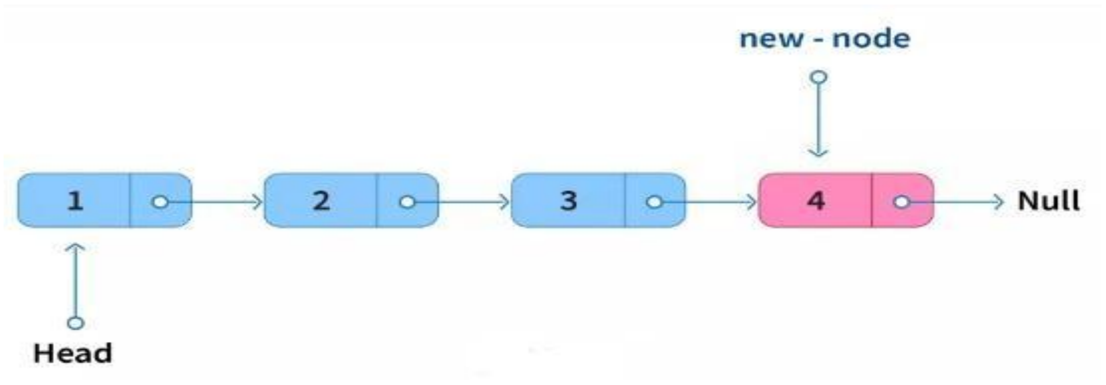


```
"void insertAfterTargetNode(Node newNode, Node head, int target){
 newNode.data = 10;
 Node temp = head;
 while(temp.data != target){
   temp = temp.next;
 }
 newNode.next = temp.next;
 temp.next = newNode;
}"
```

•Insertion at the end Insertion of a new node at the end of a singly linked list is performed in the following way,

o Go from beginning to end of the list and finish at the final node.

o Point the new node at the end of the node.In order to indicate the end of the list, set the new node to point to null. It takes O(N) operations to insert a new node at the end.



```
"void insertAtEnd(Node newNode, Node head){
  newNode.data = 10;
  Node temp = head;
  while(temp.next != null){
    temp = temp.next;
  }
  temp.next = newNode;
  newNode.next = null;
}"
```

2. Deletion

Deletion at the start

1. singly linked list's initial node in the following way:
2. Point the HEAD to the following element.
3. In a singly linked list, deleting the initial node is an O(1) operation.

```
"void deleteAtFirst(Node head){
  head = head.next;
}"
```

Deletion at the middle

The deletion activity after a particular node can be operated in the following way,

a. While at the particular node, remove the node from existence.
b. Set the current node to point to the element after this one.
c. An O(N) operation is when a node is deletion after a certain node.

```
"void deleteAfterTarget(Node head, int target){
  Node temp = head;
  while(temp.data != target){
    temp = temp.next;
 }
  temp.next= temp.next.next;
}"
```

Deletion at last

The process for deletion the last node is as follows:

a. Locate the second-to-last node in the singly linked list.

b. Set the second-to-last node point to zero.

Deleting the last node is an O(N) operation.

```
“void deleteLast(Node head){
 Node temp = head;
 while(temp.next.next != null){
  temp = temp.next;
 }
 temp.next = null;
}”
```

3. Display

traverse the single linked list from first to last in order to show it in its entirety.
Linked list nodes cannot be arbitrarily accessed, in contrast to arrays. Therefore, we must go through all (n-1) elements in order to get to the n-th element.
We acquire O(N) time complexity in this operation since the complete linked list is visited. The full single linked list can be shown using the sample that follows.

```
“void display(Node head){
 Node temp = head;
 while(temp != null){
  System.out.println(temp.data);
  temp = temp.next;
 }
 }”
```

4. Search

We must search the full linked list in order to find an entry in the singly linked list.
Every node undergoes a lookup to ascertain whether the target has been located; if so, the target node is returned; if not, we proceed to the subsequent element.
Searching an element in the singly linked list would cost us O(N) operating time if, in the worst scenario, we end up visiting every node in the list.

```
“Node search(Node head, int target){
 Node temp = head;
 while(temp != null && temp.data != target){
  temp = temp.next;
 }
 return temp;
}”
```

Linked List Applications

Some of the linked list applications are as follows:

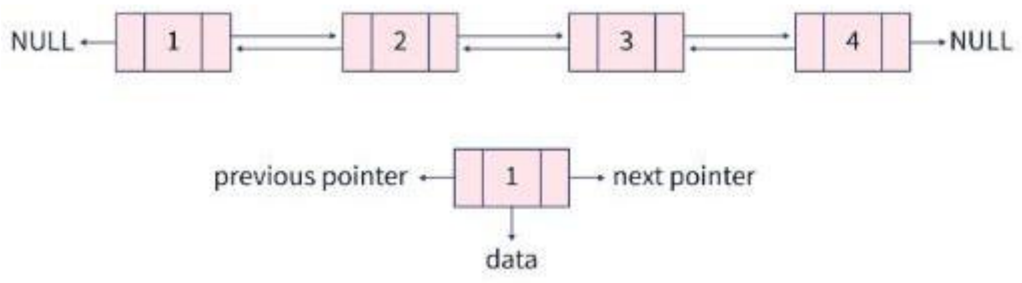
a. To hold polynomials that are single or bivariate.
b. The purpose serves as the base for certain data structures such as Graph, Stack, and Queue.
c. File allocation strategies per operating system strategy.
d. Monitor the secondary disk's spare space. It is possible to connect all of the empty areas.
e. To determine which player will be played next in turn-based games, a circular linked list may be used. We move on to the next player when that person has finished their turn.
f. To preserve data on things that may be accessed sequentially and linked to one another, such web pages, music, films, photos, and so forth.

Doubly Linked List

A doubly linked list, which permits traversal across the list in both forward and backward directions, is an alternative to a single linked list. Every node in the list has three pointers stored in it: one for the data itself, one for the node before it, and one for the node after it.

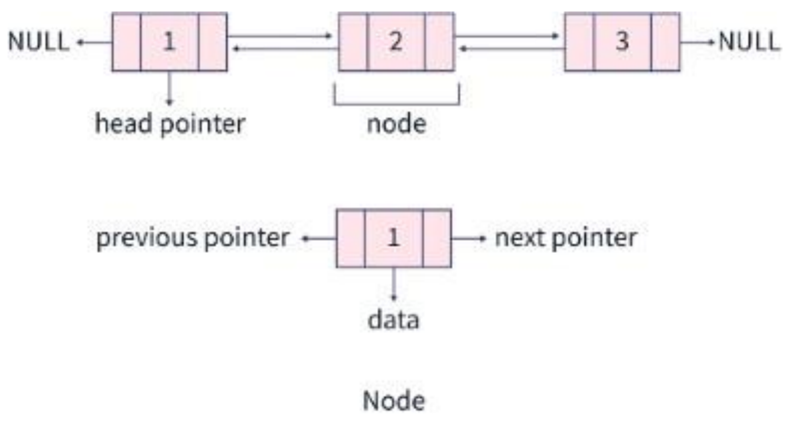
A Double Linked List in C:

A double linked list is an advanced data structures that is an improved version of a single linked list. An example of a linked data structures is a double linked list, which consists of links, or records, connected sequentially by pointers arranged in a chain. Each link in a doubly linked-list contains the data as well as pointers to the nodes that come before and after it in the sequence. The labels "previous" and "following" refer to these pointers, respectively. This pointer contains the memory locations of the previous and next points in the series. A unique node pointer head indicates the start of a double linked list. The start nodes previous pointer in a doubly linked list and the end node's next pointer points to either a sentinel value or a terminator. In C language, the sentinel value is the null pointer. Many activities can be accomplished with a doubly linked list, including showing the list from the head node down to the end and adding new nodes at specific positions or removing nodes from the list at specific positions.



The following elements must be defined in order to write the double linked list program in C:

Node: The fundamental component of a double linked-list is the node. A double linked-list is created by connecting nodes with pointers. The data item, the next pointer, and the prior pointer are all stored on each node.
Next Pointer: In a double linked-list, it holds the address of the node that comes after in the sequence.
Previous Pointer: In a doubly linked list, it holds the memory address of the node that came before it in the sequence.
Data: It keeps track of the node's data value.
Head: This denotes where the double linked-list begins.



double-linked-list operations:
A doubly linked list can be used for numerous activities. Later in the post, we will go over each operation's implementation strategy and approach.

Traversing:
Traversing the linked list, checking at each node along the way from the head pointer to the finish. This typically occurs to accomplish a specific objective, such finding a node or showing all of the node's data, among other things.
Insertion at begin:
in this case, a new node comes before the double linked-list. The modified Head pointer now points to the newly inserted node.
Insertion at last:
In this case, a new node is added at the end of the double linked-list. There is no need to change the head pointer.
Insertion after a given node:
When receive a pointer to a node in the doubly linked list, we must place the new node after the node whose pointer we have been given. We can add more nodes at any time to a doubly linked list. A node can be inserted into a doubly linked node in four main ways: In this case, the goal is to insert the new node before the node to which the pointer is supplied.

Deletion

The process of removing a node from a doubly linked list while keeping the list's structure intact is called deletion. There are three circumstances in which a node can be eliminated from a double linked-list:

Deletion at beginning:

The double linked-list's starting node is eliminated. By changing the head pointer, the next node of the deleted node is highlighted. The removed node disappears from the memory.

Deletion of the node at a given position:

Removal of the node at a specific position: In this instance, the node at the designated location needs to be removed from the double linked-list.

Last deletion:

The last node of the doubly linked-list is removed.

Elimination of the node at a certain location: In this case, the double linked-list must have the node at the specified place deleted from it.

.

Searching :

The procedure of searching for a node involves comparing each node's data in a double linked-list with the designated item to be sought. The address or position of the node inside the linked list is frequently returned as the output. This address can be used to visit the same node if needed. If such a node does not exist, NULL is returned in the output.

Insert node at the front:

The new node in front of the doubly linked list is updated. This new node is now the head pointer.

Step 1: To construct a new node, use the data item. This node will be referred to as new_node.

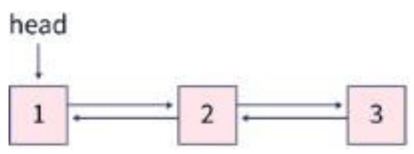
Step 2: Pass the next pointer of the new node to the head.

Step 3: If the head pointer is not NULL, assign the head's previous pointer to the new_node.

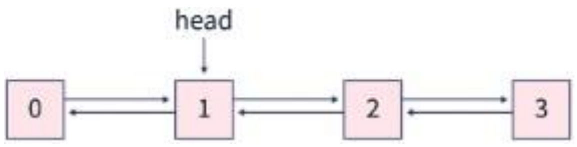
Step 4: Finally, update the head pointer.

Example:

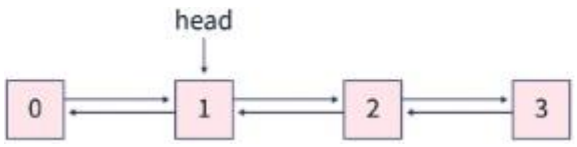
Think of a linked list with three nodes: 1 ⟷ 2 ⟷ 3. The node with data 0 at the front is what we now wish to add.



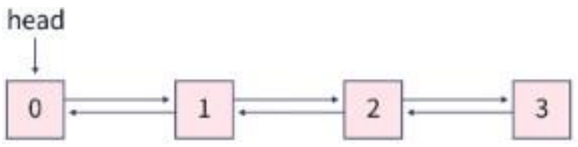
Initially, a new node is created with data 0, and its next pointer is assigned to the head pointer.



We now assign the head's previous pointer to the newly created node.



Lastly, we make updates to the head pointer.



note: In the function insertAtFront, we pass a Spanish pointer to the main head pointer. The head pointer will only be updated locally inside the function if we don't do this.

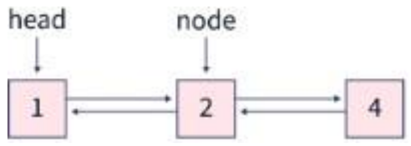
```
"void insertAtFront(struct Node** head, int data) {
   // Create a new struct node and allocate memory.
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   // Initialize data in the newly created node.
   newNode->data = data;
   // Make the next pointer of the new node point to the head.
   // In this way, this new node is added at front of the linked list.
   newNode->next = (*head);
   newNode->prev = NULL;
   // If the head is not NULL, then we update the prev pointer in the current head.
   // The prev pointer of the current head will point to the new node.
   if((*head) != NULL) {
      (*head)->prev = newNode;
   }
   // Update the head pointer to point to the new node.
   (*head) = newNode;
}"
```

Analysis of Complexity
Complexity of Time:

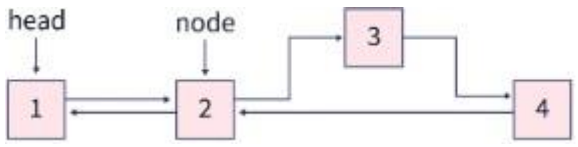
Since we are just changing a certain number of references when we add a new node at the front, the time complexity is O(1).
Complexness of Time: O(1)
Complexness of Space -All we need to do is create a pointer to the new node in order to add it to the front. Since no more memory is required, the space complexness is O(1).
Complexness of Space: O(1)

A C program that implements the double linked-list algorithm by inserting a node between two nodes
In this instance, we have to insert the new node after receiving a node.
Step 1: Using the provided data, create a fresh node called new_node.
Assign new_node_next to node_next in step two.
Step 3: Assign new_node to node→next.
Step 4: Assign node to new_node → prev.
Step 5: Assign new_node → next → prev to new_node if new_node →n ext is not NULL.

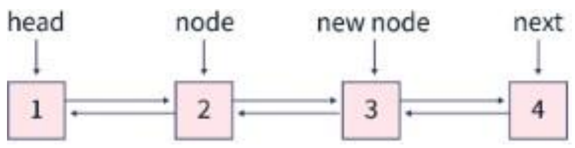
Suppose you have a linked list with three nodes: 1 ⟷2 ⟷ 4 . After the second node, we now wish to add the node with data 3. The reference to the second node is likewise provided to us.

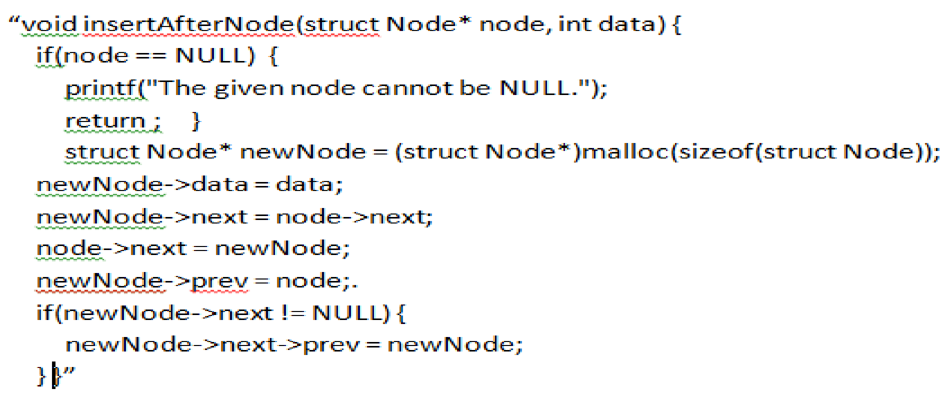


The next pointer of the new node will be updated to the next pointer of the existing node first. Next, we assign the new node's pointer to the next pointer of the provided node.



The next pointer of the new node will be updated to the next pointer of the existing node first. Next, we assign the new node's pointer to the next pointer of the provided node.





```
"void insertAfterNode(struct Node* node, int data) {
  if(node == NULL)  {
    printf("The given node cannot be NULL.");
    return ;    }
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->next = node->next;
  node->next = newNode;
  newNode->prev = node;.
  if(newNode->next != NULL) {
    newNode->next->prev = newNode;
  } }"
```

C Program for the Initialization of the Doubly Linked List Algorithm's Node Removal

Step 1: set up a pointer curver to point to the ℏ head pointer.

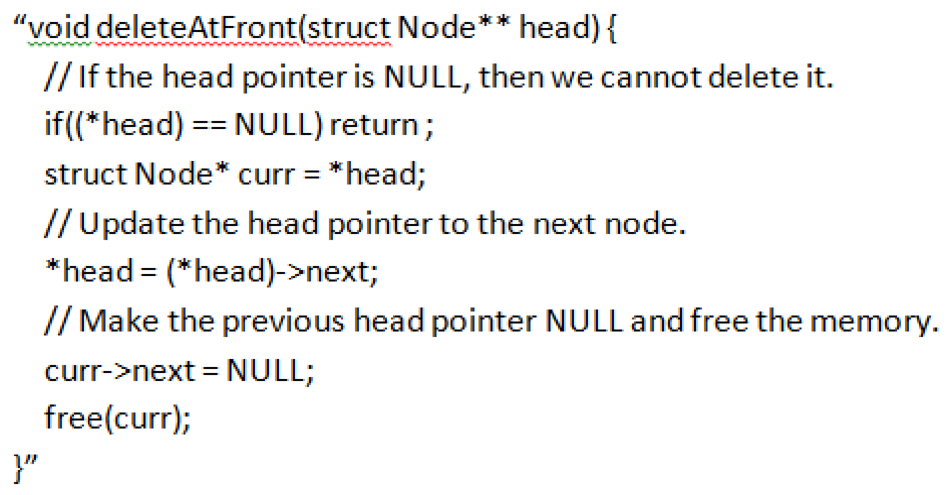
Step 2 is to increment the ℏ head pointer if it is not NULL.

Step 3: Release the memory and set the curr pointer to NULL.

As Example.

Assume linked list that has three nodes: 1⟷2⟷3. We now wish to remove the front node. In this instance, we just delete the first node in the series and update the head pointer to the next head pointer.

Finished Order: 2⟷3



```
"void deleteAtFront(struct Node** head) {
  // If the head pointer is NULL, then we cannot delete it.
  if((*head) == NULL) return ;
  struct Node* curr = *head;
  // Update the head pointer to the next node.
  *head = (*head)->next;
  // Make the previous head pointer NULL and free the memory.
  curr->next = NULL;
  free(curr);
}"
```

C Program to Remove a Code at the Double Linked List Algorithm's End

Step 1: set up a pointer curver to point to the ℏ head pointer.

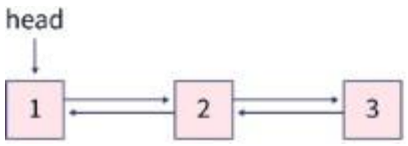
Step 2: Increase the curr pointer as long as curr→next is not NULL.

Step 3:Update curr→prev→next and curr→prev as NULL in

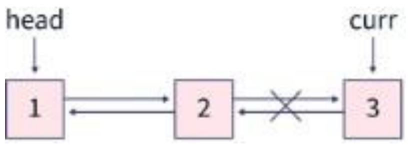
Step 4: Release the memory and set the curr pointer to NULL.

As example

assume a linked list that has three nodes: 1—2—3. At this point, we wish to remove the node.



First, we use a pointer curve to iterate to the final node. We now set curr→next to NULL.



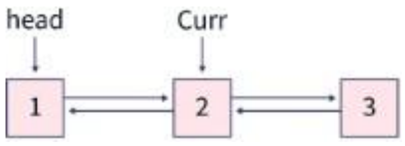
```
"void deleteAtLast(struct Node** head) {
   if((*head) == NULL) return ;
      struct Node* curr = *head;
   while(curr->next != NULL) {
      curr = curr->next;
   }
   struct Node* prev = curr->prev;
   prev->next = NULL;
   curr->prev = NULL;
   free(curr);
}"
```

C Program for the Doubly Linked List Algorithm's Node Removal at a Specified Position

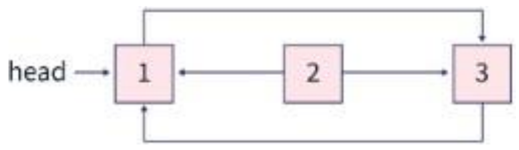
In this instance, the node at a given index position has to be deleted.

Step 1: The front node is deleted if position equals 1.
Step 2: Until position > 1, iterate a pointer cur
Step 3: The curr node must now be removed, and curr→prev and curr→next must be connected.
Assign curr→prev→next to curr→next & curr→next→prev to curr→prev in step four.
Step 5: Release the memory and set the cursor pointer to NULL.
As Example:
Think of a linked list that has three nodes: 1⟷2⟷3. The second node in the list is what we now wish to remove.

First, we use a pointer curve to navigate to the second node.



Next, we establish a connection between the earlierpresent node and the subsequent node of the curr node.



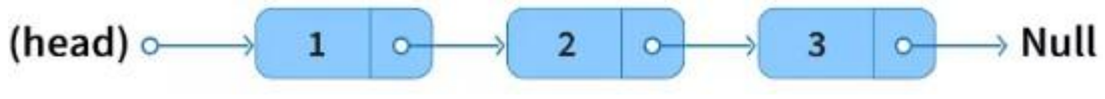
```
“void deleteAtPosition(struct Node** head, int position) {
   struct Node* curr = *head;
   if(position == 1) {
      deleteAtFront(head);
      return;
   }
   while(position > 1 && curr) {
      curr = curr->next;
      position--;
   }
   if(curr == NULL) {
      printf("Node at position is not present.\n");
      return;
   }
   struct Node* prevNode = curr->prev;
   struct Node* nextNode = curr->next;
   curr->prev->next = nextNode;
   if(nextNode != NULL) {
      nextNode->prev = prevNode;
   }
   curr->next = curr->prev = NULL;
   free(curr);
}
```

Circular-Linked-List

Introduction Circular-Linked-List

Circular-Linked-Lists are a kind of Linked List Data Structures, as the name implies. Before moving on to the Circular-Linked-List, over the Linked List ideas, if we choose to use arrays, our three items would be represented as {1, 2, 3}.

In contrast, every element in a linked list control a pointer to the element after it. would be represented as:



It has unique benefits and drawbacks in comparison to arrays. It is difficult to go around a linked list in a straightforward manner. "circle," mean that we may obtain the starting node from the last node after reached the last node in the list.

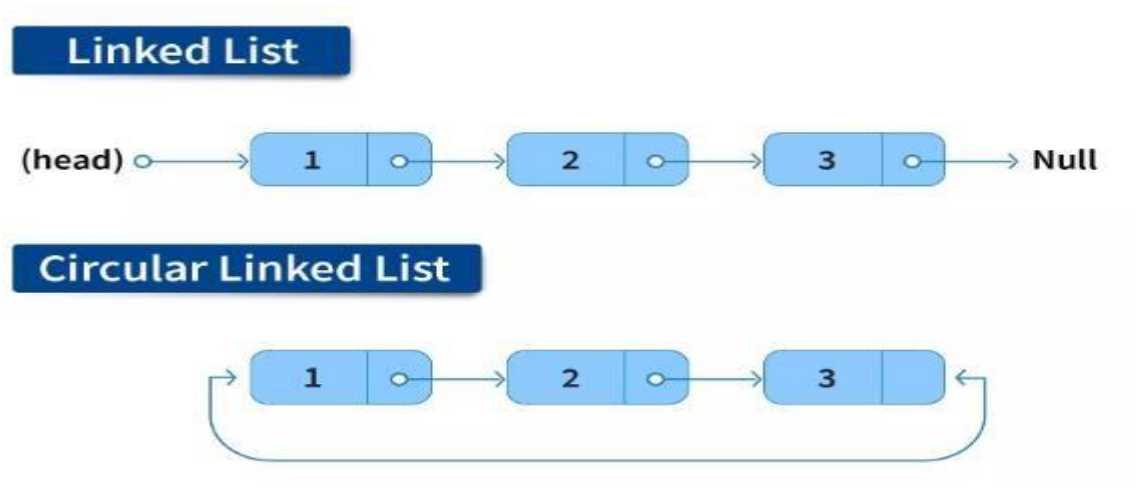
there are numerous real-world situations where it is necessary to repeatedly circle the elements on the list. Furthermore, as we've seen, the end node in a linked list leads to Null; as a result, rely on the Head pointer, which points to the first node, in order to determine the starting node.

Circular linked lists save the overhead of traversing until the Node refers to null and then using a head pointer to circle back from start; however, by default, an intrinsic feature of circular linked lists provides the ability to circle the linked list.

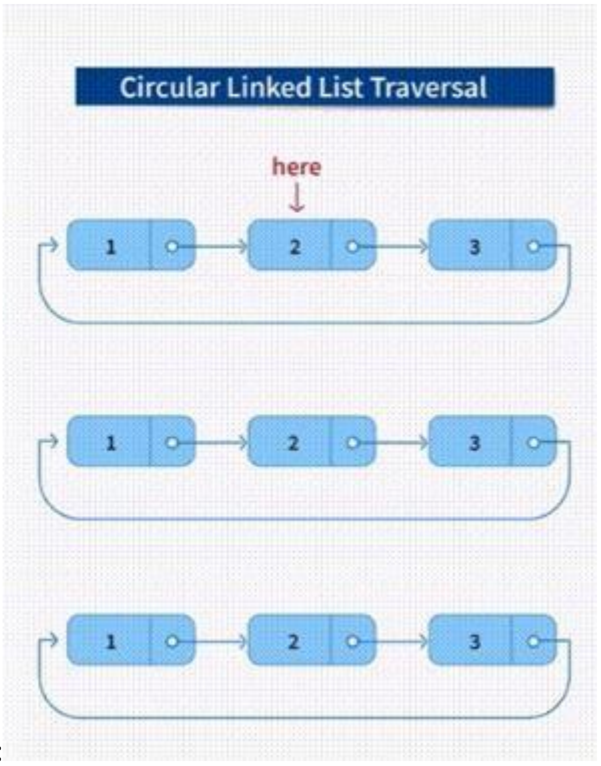
What is a Circular-Linked-List in Data Structures?

The circular linked lists are an alternative to linked lists, as we have seen. The last node in a linked-list leads to null; in circular linked-lists, link to points the last node to the fresh node. As a result, the components become connected in a circle, and the list is mention to as a circular linked-list.

A circular-linked-list can be traversed until the starting node is reached. There is no null value in the following section of any node in the circular linked list, and it has neither a beginning nor an end. It differs from the linked-list in the following ways.



Circular Singly Linked List



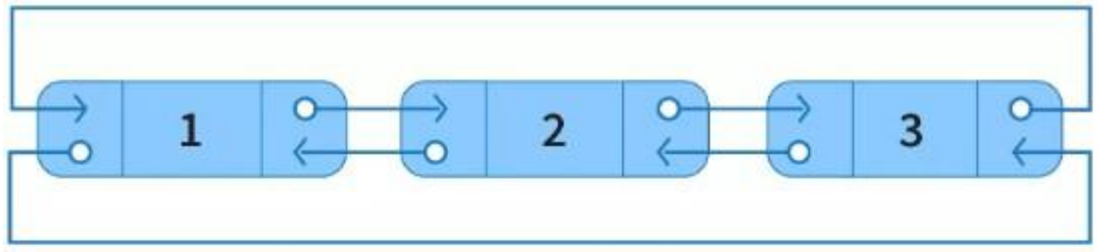
in a circular fashion:

Circular Doubly-Linked-List

As shown by Linked Lists, there are numerous uses for which it is necessary to visit the list in both the ways. We own the Doubly Linked List to facilitate the extra backward transversal.

Each node in a circular singly linked-list has 2 pointers, Next and Back, that points to the node after it and the node before it, respectively, plus the next of last, which points to the node before it and vice versa, forming a circle. These lists can also make use of this property. We call

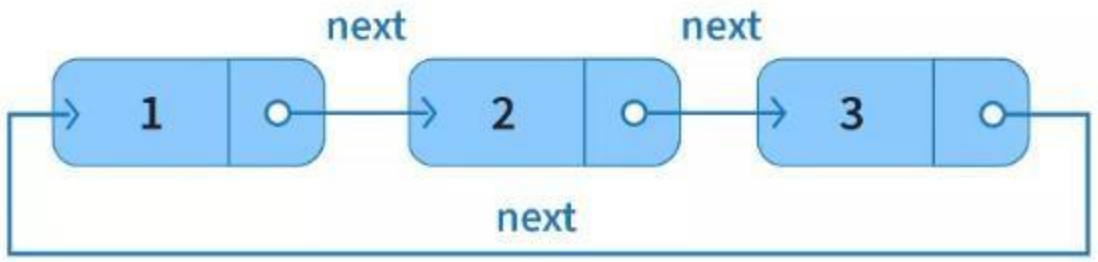
this kind of data structure a doubly-linked circular list.The appearance of the circular doubly linked list is depicted in the following image:
Data Structures' Representation of a Circular-Linked-List.



Node in the Circular-Linked-List can be represented as:

```
class Node{
 int value;
 Node next;
}"
```

Now we will create a simple circular linked list with three Nodes to understand how this works.

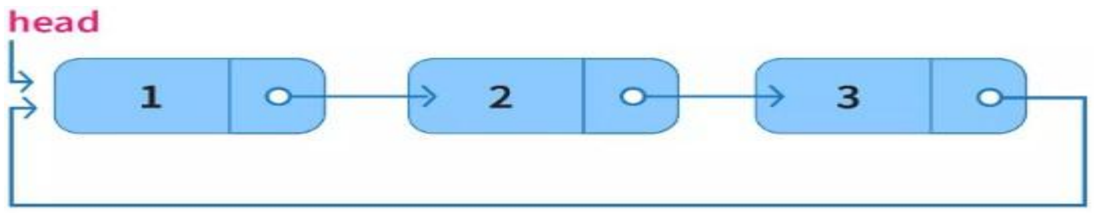


This can be represented as:
"// Initialize the Nodes.

```
 Node one = new Node(1);
 Node two = new Node(2);
 Node three = new Node(3);
```

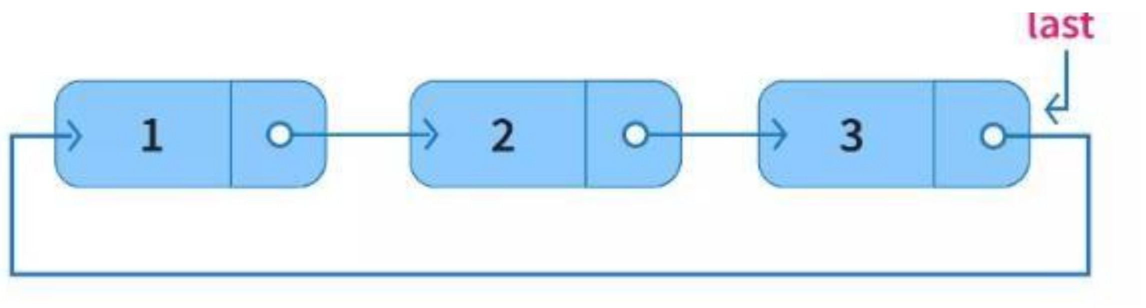
Data Structure Operations on a Circular-Linked-List
a representation of the circular-linked-list. We require a pointer to any Nodes in the Circular Linked List so that we can navigate the whole List in order to carry out additional actions on it.The head pointer, or pointer to the first node, is typically maintained in linked lists. Let's investigate whether the identical Node performs optimally in a circular-linked-list.

1. Insertion
Any new node can be added to the provided circular-linked-list.
If we maintain the Head Pointer and wish to add a new node at the start of the circular linked list, as demonstrated:



to add a new Node (let's say 4) in the start. How can the existing Nodes in the list be modified to make opportunity for this new Node?
The next of a new Node (i.e., 4) can point to Head since we maintain the head pointer (to 1), which takes care of adding 4 to the list.
But something is missing from this situation. The circular property hasn't been modified yet.
The last node, or 3, is still pointing to 1, even though 3 should now be pointing to 4 in light of the addition of 4. How is it possible to accomplish this?
We can visit the list, get to the circular-linked-list's last node (To 3), and then move on to the next node (i.e. 4). However, the procedure is slow, as you may imagine. Imagine a situation in which we have a circular linked list with 1000 nodes, and we need to visit all 1000 nodes in order to get to the final node in order to add one.

In a similar vein, we must go through If we need to add a fresh node at the end, we should reorder the list from head to last node (after 3)
It appears that the head pointer in a data structure is not the ideal pointer for a circular linked list. What would happen, if we kept the pointer to the Last Node instead?
In both scenarios, there will be no need to go through the entire list if we use a pointer to the last node in place of a start pointer.
Using last. next, we may hop directly to head from last, thus the head pointer is not really necessary. As a result, the pointer to the final node rather than the head can be preserved. So the circular linked list in Data Structure can be visualized as:



•

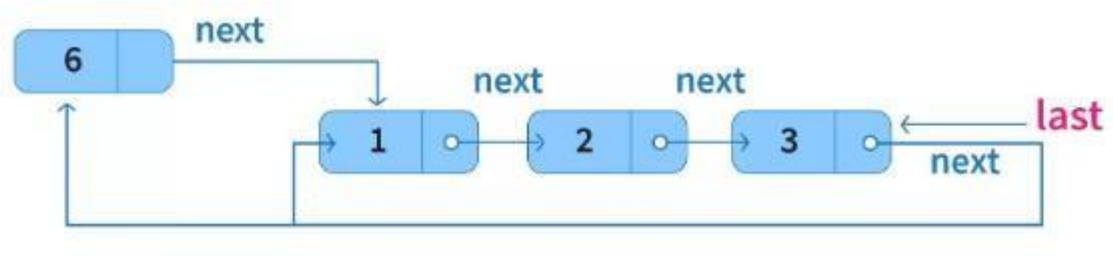
Insertion at the Beginning:

use Last.next to obtain the first Node directly because we have maintained the link to the Last Node.

The Node can be added at the start as:

• The New Node can point to this node as its next by obtaining the current First Node (via last.next). This fixes the New Node's pointer adjustments.

• Respect the circular characteristic by pointing the Last Node toward the New Node.This can be written programmatically as:



```
“void insertNodeAtBeginning(int value) {
Node newNode = new Node(value);
Node oldFirstNode = last.next;
 newNode.next = oldFirstNode;
 last.next = newNode;
}”
```

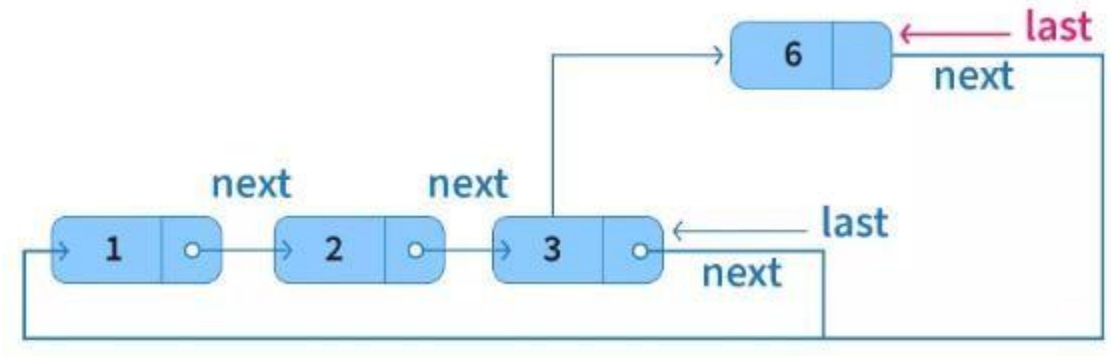
to modify the new Node's and the previous Node's references. Therefore, if we have a pointer to the last node, it requires constant time operation.

Additionally, since no additional space was utilized, the space complexness—that is, O(1) space complexity—to add a new node to the start of the circular linked list stays constant.

• **Insertion at the end:**

how the New Node can be inserted in-between 2 Nodes.

1. The next of a new node will point to the last.next.
2. This new node will be pointed to by Last.next.
3. Update the last node reference to point to the new node as we now have one at the end.



This can be written as:

```
"void addAtEnd(int value) {
 Node newNode = new Node(value);"

"// Adjusting the links.
 newNode.next = last.next;
 last.next = newNode;
 last = newNode;
}"
```

Because we just need to change the final node's pointers to add a new node to the circular linked-list, adding a fresh node at the end of the list requires constant processing time.

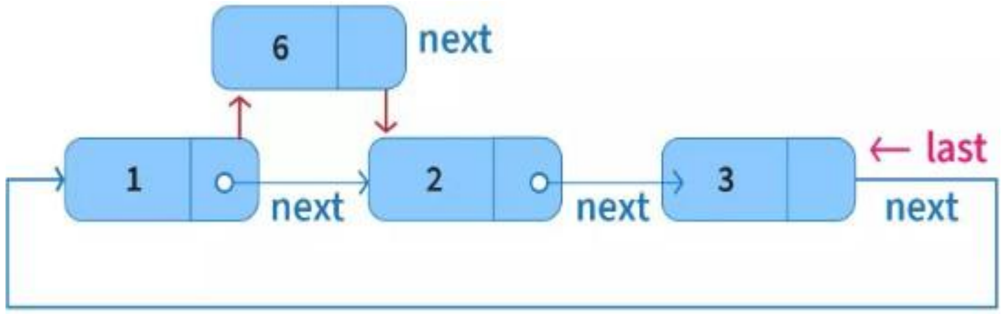
The space complexity to add a new node at the end of the circular-linked-list is O(1) since no more space has been used.

•**Insertion After Another Node:**

Let's see how the New Node can be inserted in between 2 Nodes.

1. Continue navigating until we arrive at the designated node (let's say X).
2. Set the NewNode's next to the X's next.

3. Point this new Node at the X.



This can be written programmatically as:

```
"void addAfter(int newValue, Node nodeBefore) {
 if (nodeBefore == null) {
  return;
 }
 Node newNode;
 Node transversalNode = last.next; // Transverse the List from last Node onwards.
 do {
  if (transversalNode.value == nodeBefore.value) { // We found the node.
   newNode = new Node(newValue);

   // Adjusting the links
   newNode.next = transversalNode.next;
   transversalNode.next = newNode;

   if (transversalNode == last) {
    // If the nodeBefore was LastNode itself, meaning we are inserting this node at the end,
    // adjust the last pointer to point to this node now.
    last = newNode;
   }
  }

  transversalNode = transversalNode.next;
  // Keep transversing until we reach the Node after which the new node has to be inserted.
 } while (transversalNode != last.next);
}"
```

2. Deletion

- There are three possible ways to removes a specific node from a circular-linked-list:
- The CLL becomes empty if it is the only node in it. In this instance, the last node, which was pointing to the sole node, might now point to null.

a. Should the node that needs to be removed by the final node within the CLL

b. To discover the node that needs to be deleted before another, iterate the circular linked list. Call it X if you will.

c. To remove the last from the CLL, point X.next to null.

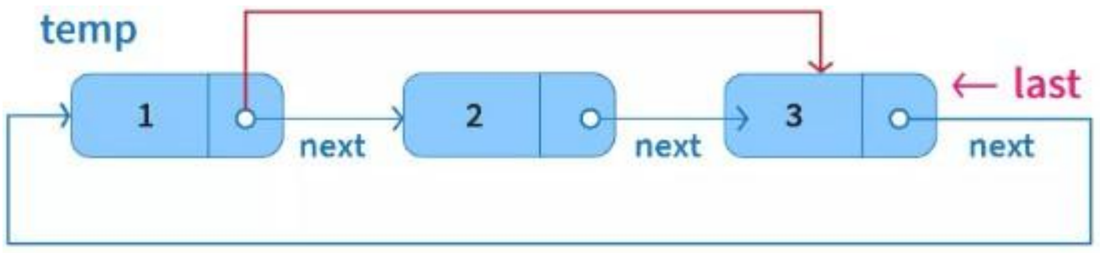
d. Set X as the new last node by setting the last pointer to X.

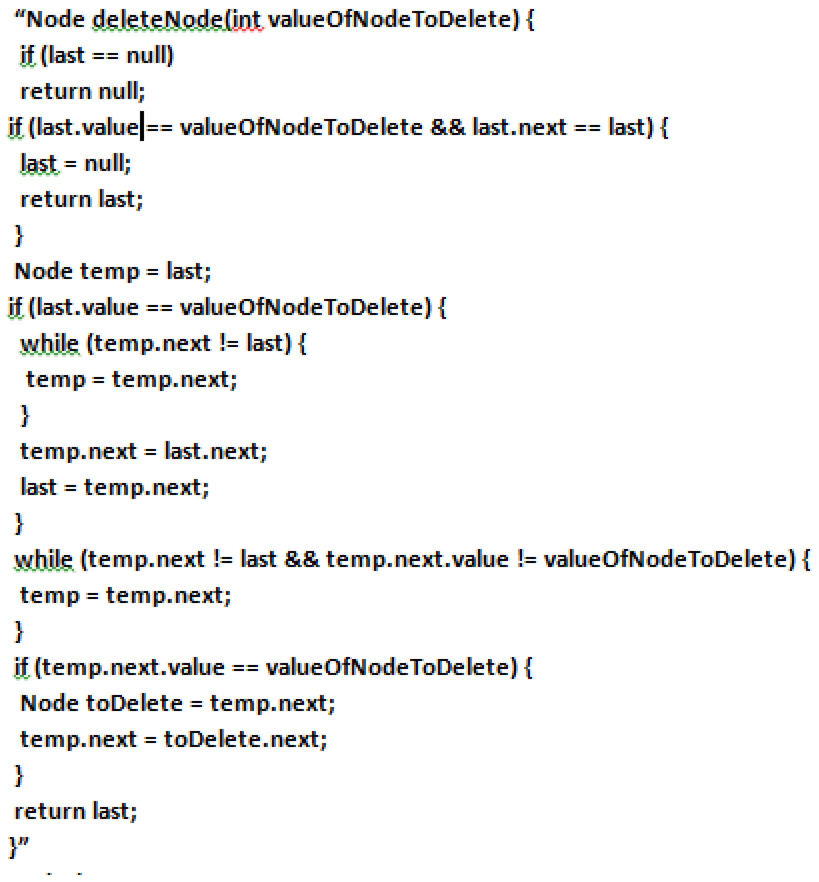
1. Deleting node in the CLL:

• To discover the node to delete before another, iterate the circular linked list. Please refer to it as X.

• Give the node that has to be erased the name Y.

• X.next must be changed to point to Y.next in order to delete Y.

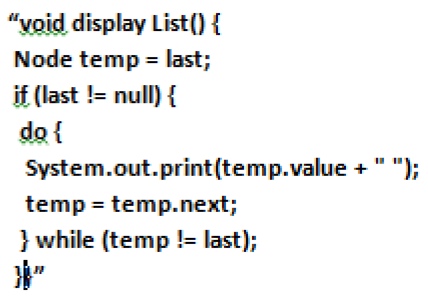




```
"Node deleteNode(int valueOfNodeToDelete) {
 if (last == null)
 return null;
if (last.value == valueOfNodeToDelete && last.next == last) {
 last = null;
 return last;
}
Node temp = last;
if (last.value == valueOfNodeToDelete) {
 while (temp.next != last) {
  temp = temp.next;
 }
 temp.next = last.next;
 last = temp.next;
}
while (temp.next != last && temp.next.value != valueOfNodeToDelete) {
 temp = temp.next;
}
if (temp.next.value == valueOfNodeToDelete) {
 Node toDelete = temp.next;
 temp.next = toDelete.next;
}
return last;
}"
```

3. Display

The circular linked list must be displayed by traversing from the last Pointer till we get back to it. We are able to print the Nodes' values as we traverse over them.



```
"void display List() {
 Node temp = last;
 if (last != null) {
  do {
   System.out.print(temp.value + " ");
   temp = temp.next;
  } while (temp != last);
 }}"
```

Applications of Circular Linked List

When we need to preserve the circular order between the various Nodes, circular linked lists come in helpful. as circular linked lists, no Node points to null, as contrast to linked lists.
Many real-world systems use circular linked lists because they make it simple to go back to the starting node from any given one. Among these are the following: • In games with multiple players, every player is represented by a Node in a circular-linked-list. By exploiting the circular aspect, we can quickly shuffle back to the first player from the final player, ensuring that every participant gets an opportunity to play. Operation systems apply this principle in a similar way to run many programs. These apps are all arranged in a circular linked list, so the system may quickly circle back to the beginning of the list without worrying about reaching the end of the running applications.

Summary

Two different kinds of circular linked lists exist:
1. Circular Singly Linked List: This type of list only allows one way of transverse traversal while preserving its circular nature.
2. Circular Doubly Linked Lists: these lists allow for both direction transverse traversal while preserving their circular nature.

The circular linked list's Nodes are simple to add and remove, and the process is remarkably similar to that used for linked lists. To make things easier, we use the circular-linked-lists "last" pointer rather than the linked list's "head" pointer.

1. **Dynamic Data Structures**

One of the primary applications of linked-lists is the creation of dynamic data structures. Unlike arrays, which have a immobile size, linked lists can grow or shrink dynamically as needed. This makes linked lists ideal for scenarios where the size of the data structure is unknown or may change over time. For example, linked lists are commonly used to implement stacks, queues, and hash tables.

2. **Memory Management**

Linked lists are also used for memory management in C programs. When dynamically allocating memory using functions like malloc or calloc, linked-lists can be used to keep track of allocated memory blocks. Each node in the linked list can represent a memory block, and the pointers between nodes can be used to navigate and manage the allocated memory.

3. **File Systems**

Linked lists are widely used in file systems to represent directories and files. Each node in the linked list can represent a file or a directory, and the pointers between nodes can be used to establish the hierarchical structure of the file system. Linked lists provide an efficient way to traverse and manipulate the file system structure.

4. **Graphs and Trees**

Linked-lists are often used to utilize graphs and trees, which are essential data structures in computer science. In a graph, each node can be represented by a linked list, where each component in the linked list represents an edge connecting the node to other nodes. Similarly, in a tree, each node can be represented by a linked list, where each component in the linked list represents a child node. Linked lists provide a flexible and efficient way to represent and traverse these complex data structures.

5. **Polynomial Implementation**

Linked-lists are commonly used to represent polynomials in mathematics. Each node in the linked list can represent a term in the polynomial, with the coefficient and exponent stored as data in the node. The pointers between nodes can be used to establish the order of the terms in the polynomial. Linked lists provide a convenient way to perform operations on polynomials, such as addition, subtraction, and multiplication.

6. **Undo/Redo Functionality**

Linked-lists can be used to implement undo/redo functionality in applications. Each node in the linked list can represent a state or action, and the pointers between nodes can be used to navigate between different states. This allows users to undo or redo previous actions in an application, providing a valuable feature for user interaction.

These are just a few examples of the many applications of linked lists in the C programming language. Linked lists are versatile and can be used in various scenarios where dynamic data structures, memory management, hierarchical structures, or ordered collections are required. Understanding and mastering linked lists is essential for any programmer working with the C language.

OPERATIONS ON LINKED-LIST :

A linked-list is a data structures made up of a series of nodes, each of which has a reference (or link) to the node after it in the sequence as well as a data element. Linked lists are dynamic data structures that have the ability to alteration size as a program runs

.

the main operations that can be performed on a linked-list:

Insertion: Inserting a fresh node into a linked-list can be done in several ways, depending on the position where the new node needs to be inserted. The common insertion operations are:

a) Inserting at the opening of the list: This involves creating a fresh node, setting its data, and updating the link to point to the current first node.

b) Inserting at the end of the list: This requires traversing the list until the last node is reached, creating a new node, and updating the link of the last node to point to the new node.

c) Inserting at a specific position: This involves traversing the list until the desired position is reached, creating a new node, and updating the links of the adjacent nodes to include the new node.

Deletion: Removes a node from a linked-list can also be done in various ways, depending on the position of the node to be deleted. The common deletion operations are:

b) Deleting the first node: This involves updating the link of the first node to point to the second node and freeing the memory occupied by the first node.

c) Deleting the last node: This requires traversing the linked-list further the second-to-last node is reached, updating its link to NULL, and freeing the memory inhabited by the last node.

d) Deleting a node at a specific position: This involves traversing the list until the desired position is reached, updating the links of the adjacent nodes to bypass the node to be deleted, and freeing its memory.

Traversal: Traversing a linked-list means visiting each node in the list and performing some operation on its data. This can be done using a loop that starts from the first node and continues until the last node is reached, updating the current node reference to the next node in each iteration.

Searching: Searching for a specific value in a linked-list involves traversing the list and comparing the data of each node with the target value. If a match is found, the search operation can be terminated, and the position or existence of the value can be determined.

Updating: Updating the data of a node in a linked list can be done by traversing the list until the desired node is reached and modifying its data field.

Counting: Counting the number of nodes in a linked-list requires traversing the list and incrementing a counter variable for each node visited.

Reversing: When a linked list is reversed, the links between its nodes are altered, reversing the order such that the last node becomes the first, the second-to-last node becomes the second, and so on. one for the current node, one for the node before it, and one for the node after it.

These are the main operations that can be performed on a Each operation has its own implementation logic and complexity considerations. It is important to handle edge cases, such as empty lists or operations on the first or last node, to ensure the correct functioning of the linked list.

MODULE-3

Stacks and Queues

In definite instances in computers sciences, it is preferable to limit insertion and deletion to the beginning or end of the list, rather than in the center. Stacks and queues are two illustration of useful data structures.

Linear lists and arrays allow you to insert and delete elements at any point in the list, whether at the beginning, end, or center.

STACK:

A stack is a list of elements in which a member can only be added or removed at one end, known as the topmost of the stack. Stack is also referred to as LIFO (last in, first out) list. Because items can only be add or remove from the top, the final item added to a stack is the first item remove.

Stacks are associated with two basic operations:

The phrase "push" refers to inserting an element into a stack.

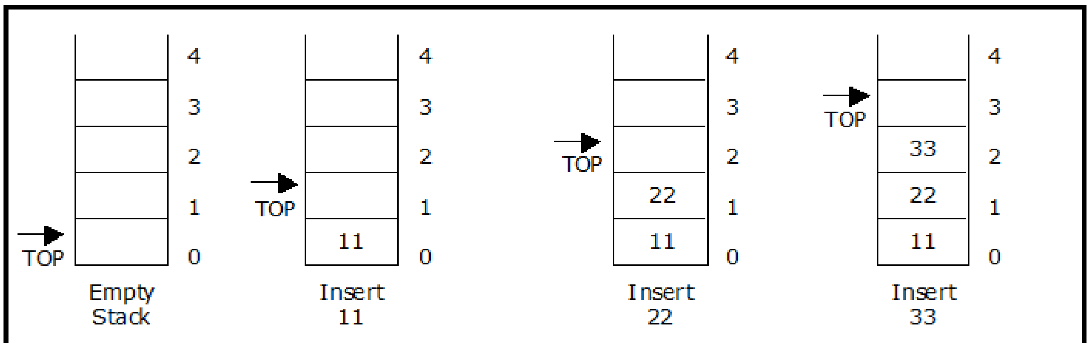
The term "pop" Mention to the act of removes an element from a stack.

Because all insertions and deletions occur at the same end of the stack, the final piece add to the stack is also the first element remove from the stacks. When you create a stack, the stacks base remains constant while the stack top changes as elements are add and remove. The top of the stack is the most accessible, and the bottom is the least accessible.

Stack Representation:

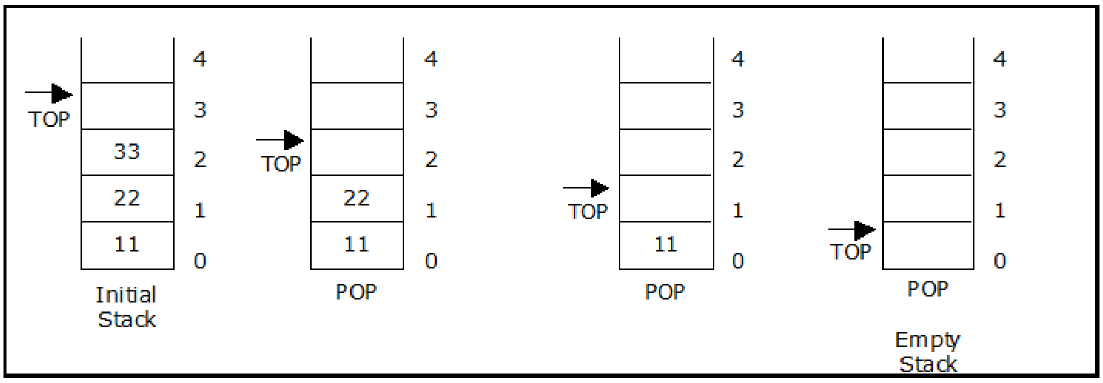
Consider a stacks with a content of six items. This is referred to as stack size. The number of components to be add shouldn't exceed the stack's maximal size. If we try to add a new element that is larger than the maximal sizing, we will hit a stack overrun problem. Similarly, you cannot delete pieces beyond the stack's base. If this occurs, we will encounter a stack underflow problem.

Push() performs the activity of adds an element to a stack.



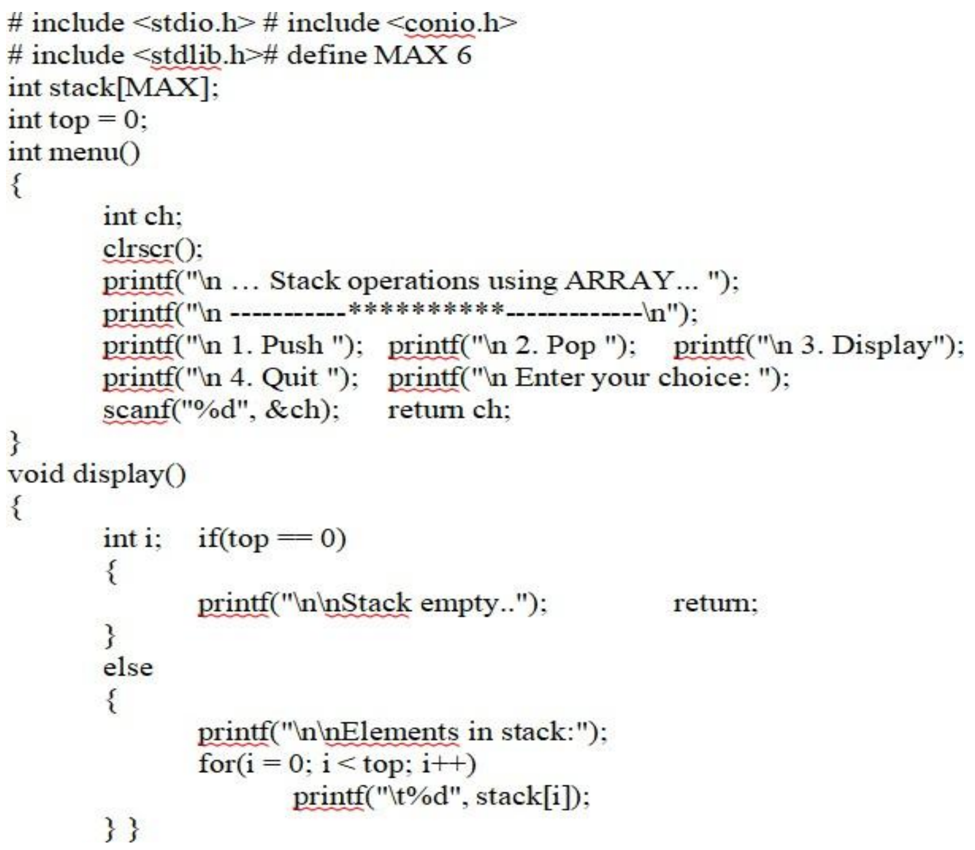
Pushing operation on Stack

Pop() performs the activity of removes an element from the stack.



Poping operation on Stack

Stacks operation using Array implementation:



```
# include <stdio.h> # include <conio.h>
# include <stdlib.h># define MAX 6
int stack[MAX];
int top = 0;
int menu()
{
        int ch;
        clrscr();
        printf("\n … Stack operations using ARRAY... ");
        printf("\n -----------**********-------------\n");
        printf("\n 1. Push ");  printf("\n 2. Pop ");   printf("\n 3. Display");
        printf("\n 4. Quit ");  printf("\n Enter your choice: ");
        scanf("%d", &ch);       return ch;
}
void display()
{
        int i;  if(top == 0)
        {
                printf("\n\nStack empty..");            return;
        }
        else
        {
                printf("\n\nElements in stack:");
                for(i = 0; i < top; i++)
                        printf("\t%d", stack[i]);
        } }
```

```
  void pop()
  {
          if(top == 0)
          {
                  printf("\n\nStack Underflow..");               return;
          }
          else
                  printf("\n\npopped element is: %d ", stack[--top]);
  }
  void push()
  {
          int data;        if(top == MAX)
          {
                  printf("\n\nStack Overflow..");                return;
          }
          else
          {
                  printf("\n\nEnter data: ");              scanf("%d", &data);
                  stack[top] = data;               top = top + 1;
                  printf("\n\nData Pushed into the stack");
          }
  }

void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                push();
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                display();
                                break;

                        case 4:
                                exit(0);
                }
                getch();
        } while(1);
}
```

This C program uses an array to implement basic stack operations. The stack is a data structures that adheres to the Last In First Out (LIFO) principle, which states that the last piece adding to the stack will be the first to be separate.

Here's a rundown of the operations:

1.push(): This function inserts a new element onto the stack. It checks for stack overflow (when the stack is full) and allows the user to enter data, which is subsequently placed to the top of the stack if the stack is not full.

2.pop(): This function is used to remove the stack's topmost member. It checks for stack underflow (when the stack is blank) and eliminates the top component if the stack is not empty.

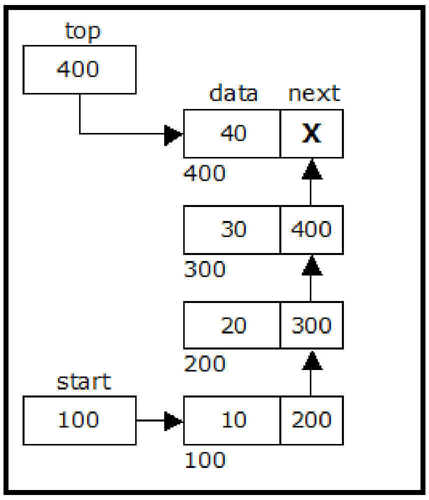
3.display(): This function shows all of the components in the stack, from bottom to top.

4.menu(): Displays a basic menu to the user, allowing them to select from several stack operations (push, pop, display, and quit).

The main() function executes an indefinite loop (do-while loop) to continue displaying the menu to the user until the user chooses to stop (case 4). It takes the user's selection and performs the corresponding operation within the loop.

Linked List Execution of Stack:

A stack can be described as a linked list. Push and pop activities are done at the topmost of a stack. Using the top pointer, we may execute comparable operations at one end of the list.



linked list Implementation

Stacks operations using linked lists Implementation:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct stack
{
        int data;
        struct stack *next;
};

typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
        node *temp;
        temp=(node *) malloc( sizeof(node)) ;
        printf("\n Enter the data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void push(node *newnode)
{
        node *temp;
        if(start == NULL)
        {
                start = newnode;

                top = newnode;
        }
        else
        {
                temp = start;
                while( temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
                top = newnode;
        }
        printf("\n\n\t Data pushed into stack");
}
void pop()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t Stack underflow");
                return;
        }
        temp = start;
        if( start -> next == NULL)
        {
                printf("\n\n\t Popped element is %d ", top -> data);
                start = NULL;
                free(top);
                top = NULL;
        }
        else
        {
                while(temp -> next != top)
                {
                        temp = temp -> next;
                }
                temp -> next = NULL;
                printf("\n\n\t Popped element is %d ", top -> data);
                free(top);
                top = temp;
        }
}
void display()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t\t Stack is empty ");
        }
        else
        {
                temp = start;

                 printf("\n\n\n\t\t Elements in the stack: \n");
                 printf("%5d  ", temp -> data);
                 while(temp != top)
                 {
                         temp = temp -> next;
                         printf("%5d   ", temp -> data);
                 }
         }
 }

 char menu()
 {
         char ch;
         clrscr();
         printf("\n \tStack operations using pointers.. ");
         printf("\n -----------**********-------------\n");
         printf("\n 1. Push ");
         printf("\n 2. Pop ");
         printf("\n 3. Display");
         printf("\n 4. Quit ");
         printf("\n Enter your choice: ");
         ch = getche();
         return ch;
 }
```

```
void main()
{
        char ch;
        node *newnode;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                newnode = getnode();
                                push(newnode);
                                break;
                        case '2' :
                                pop();
                                break;
                        case '3' :
                                display();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while( ch != '4' );
}
```

Algebraic Expressions:

A sanctioned collection of operators & operands is an algebraic expression. The measure on which a exact activity is performed is referred to as the operand. The operand can be a variable like a, b, or c or a constant like 2,3,4 and so on. The operator symbol represents a numerical or logical operation between the operands. +,
-, *, /, and similar operators are examples.

Three alternative notations can be used to represent an algebraic expression. There are three types of notations: infix, postfix, and prefix.

Infix: An arithmetical demonstration in which the arithmetical operator is fixed (placed) between the two operands.

For instance, (A + B) * (C - D)

Prefix: A kind of arithmetical notation in which the arithmetical operator is fixed (placed) before (pre) its two operands. Polish notation is named after the Polish mathematician Jan Lukasiewicz, who invented it in 1920.* + A B - C D as an example.

Postfix: The form of an arithmetic expression in which we fix (position) the arithmetic operator after (post) its two operands is known as a postfix. The postfix notation is also known as suffix notation and reverse polish notation.

For instance, A B + C D - *

The three most important characteristics of postfix reflection are:

1. The operands are ordered in the same way as in the corresponding infix expression.
2. Parenthesis are not required to unambiguously designate the expression.
3. The precedence of the operators is no longer significant when evaluating the postfix expression.

There are 5 binary transactions to consider: +, -, *, /, and $ or (exponentiation). The following multiple operations, in order of precedency (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Converting Expressions using Stack:

Let's change the expressions from one format to another. These can be accomplished as follows:

1. From In-fix to Post-fix
2. From Pre-fix to In-fix
3. From Post-fix to In-fix
4. From Post-fix to Pre-fix
5. From Pre-fix to In-fix
6. From Pre-fix to Post-fix

Converting from In-fix to Post-fix:

The following is the procedure for converting an In-fix expression to a Post-fix expression:

1. Begin From left to right, scan the infix phrase.

2. **a)** Place the scanned symbol on the stack if it is left parenthesis.

b) Place the scanned symbol immediately in the Post-fix expression (output) if it is an operand.

c)If the scanned symbol is a right parenthesis, continue popping items from the stack and inserting them into the postfix expression until we find the matching left parenthesis.

d)If the scanned symbol is an operator, continue removing all the operators from the stack and inserting them into the postfix expression if and only if the precedence of the operator at the top of the stack is greater than (or greater than or equal to) the precedence of the scanned operator and push the scanned operator onto the stack; otherwise, pop the scanned operator onto the stack.

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Converting from In-fix to Pre-fix:

For transforming an expression from infix to prefix, the precedence rules are the same. The sole difference between postfix and prefix conversion is that the expression is traversed from right to left and the operator is placed before the operands rather than after them. A complex expression's prefix form is not the inverse of its postfix form.

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	A B C D / / E F + G H	+ - * ↑	
End of string	+ - * ↑ A B C D / / E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

Converting from postfix to infix:

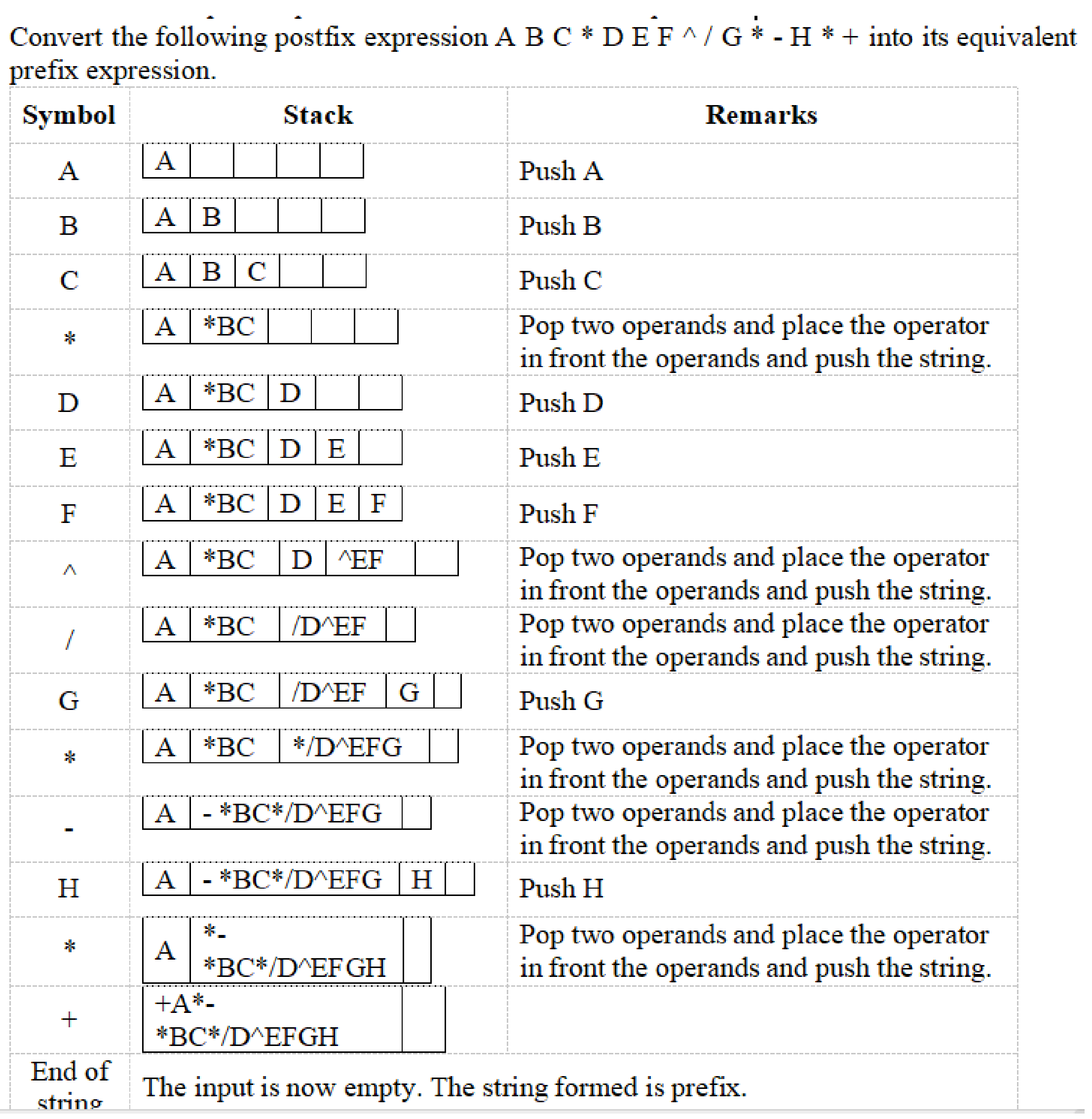
The following is the procedure for converting a postfix expression to an infix expression:

1. From left to right, scan the postfix phrase.
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by inserting the operator between the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.

Conversion from postfix to prefix:

The following is the procedure for converting a postfix expression to a prefix expression:

1. From left to right, scan the postfix phrase.
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by putting the operator in front of the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.



Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A *BC	Pop two operands and place the operator in front the operands and push the string.
D	A *BC D	Push D
E	A *BC D E	Push E
F	A *BC D E F	Push F
^	A *BC D ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A *BC /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A *BC /D^EF G	Push G
*	A *BC */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A - *BC*/D^EFG H	Push H
*	A *-*BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*-*BC*/D^EFGH	
End of string	The input is now empty. The string formed is prefix.	

Conversion from prefix to infix:

The following is the procedure for converting a prefix expression to an infix expression:

1. Scan the prefix expression from left to right (in reverse order).
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by inserting the operator between the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent infix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G (E^F)	Pop two operands and place the operator in between the operands and push the string.
D	H G (E^F) D	Push D
/	H G (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
*	H ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
C	H ((D/(E^F))*G) C	Push C
B	H ((D/(E^F))*G) C B	Push B
*	H ((D/(E^F))*G) (B*C)	Pop two operands and place the operator in front the operands and push the string.
-	H ((B*C)-((D/(E^F))*G))	Pop two operands and place the operator in front the operands and push the string.
*	(((B*C)-((D/(E^F))*G))*H)	Pop two operands and place the operator in front the operands and push the string.
A	(((B*C)-((D/(E^F))*G))*H) A	Push A
+	(A+(((B*C)-((D/(E^F))*G))*H))	Pop two operands and place the operator in front the operands and push the string.
End of string	The input is now empty. The string formed is infix.	

Converting from Pre-fix to Post-fix:

The following is the procedure for converting a prefix expression to a postfix expression:

1. Scan the prefix expression from left to right (in reverse order).
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by inserting the operator after the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent postfix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G EF^	Pop two operands and place the operator after the operands and push the string.
D	H G EF^ D	Push D
/	H G DEF^/	Pop two operands and place the operator after the operands and push the string.
*	H DEF^/G*	Pop two operands and place the operator after the operands and push the string.
C	H DEF^/G* C	Push C
B	H DEF^/G* C B	Push B
*	H DEF^/G* BC*	Pop two operands and place the operator after the operands and push the string.
-	H BC*DEF^/G*-	Pop two operands and place the operator after the operands and push the string.
*	BC*DEF^/G*-H*	Pop two operands and place the operator after the operands and push the string.
A	BC*DEF^/G*-H* A	Push A
+	ABC*DEF^/G*-H*+	Pop two operands and place the operator after the operands and push the string.
End of string	The input is now empty. The string formed is postfix.	

Evaluation of Post-fix expression:

A stack is used to easily evaluate the postfix expression. When a number is seen, it is added to the stack; when an operator is seen, it is applied to the two numbers that are popped from the stack, and the result is added to the stack. There is no need to know any precedence rules when an expression is given in postfix notation; this is our clear advantage.

Example-1: Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example -2: Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Applications of stacks:

1.Compilers employ stack to ensure that parentheses, brackets, and braces are balanced.

2.A postfix expression is evaluated using stack.

3.Stack is used to transform an infix expression into a postfix/prefix expression.

4.All intermediate arguments and return values are saved on the processor's stack during recursion.

5.During a function call, the return address and arguments are pushed into a stack and popped off upon return.

Queue:

A queue is a type of list in which things are added at one end called the back and deleted at the other end called the front. A queue is also known as a "FIFO" or "First-in-First-Out" list.

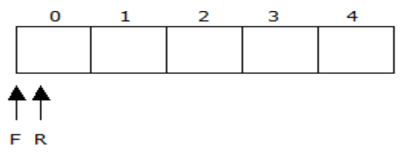
The operations of a queue are analogous to those of a stack, with the exception that insertions are made at the end of the list rather than the beginning. We will employ the following queue operations:

enqueue: inserts an element at the bottom of the queue.

dequeue: removes an element from the queue's beginning.

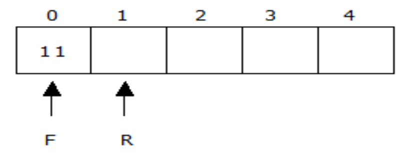
Representation of Queue:

Consider a queue with a maximal capacity of 5 elements. The queue is initially blank.



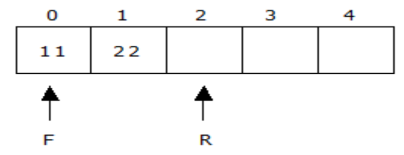
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



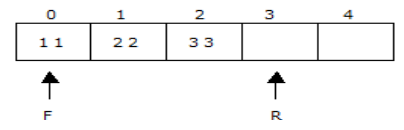
REAR = REAR + 1 = 1
FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



REAR = REAR + 1 = 2
FRONT = 0

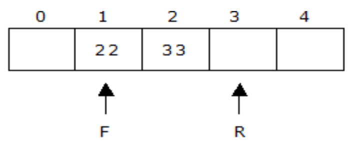
Again insert another element 33 to the queue. The status of the queue is:



REAR = REAR + 1 = 3
FRONT = 0

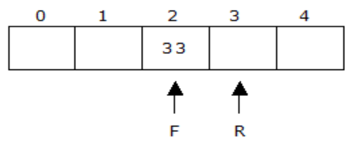
Now remove a piece. The element that was remove is the one at the top of the queue.

So the queues current state is:



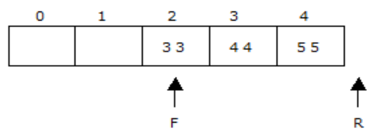
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



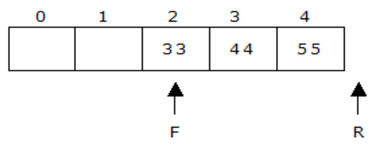
REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



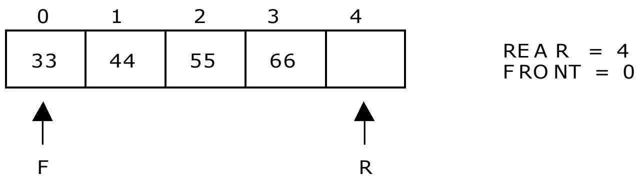
REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REAR = 5
FRONT = 2

Even if there are 2 open emplacement in the linear queue, inserting element 66 is no longer possible. To solve this problem, the queue items should be pushed to the front of the line, leaving a empty position at the back. The FRONT and REAR must then be appropriately well-adjusted. The component 66 might be placed at the back end. The queue status after this activity is as follows:



This challenge can be avoided by treating queue position 0 as a position that comes after position 4, i.e., by treating the queue as a circular queue.

Queue operations using array:

A one-dimensional array Q(1:n) and two variables front and back are required to form a queue. We will use the following conventions for these two variables: front is always one less than the real front of the queue, and rear always points to the last piece in the queue. Thus, front = rear if and only if the queue has no elements. Thus, the initial condition is front = rear = 0.

The following are the numerous queue operations for creating, deleting, and displaying queue elements:

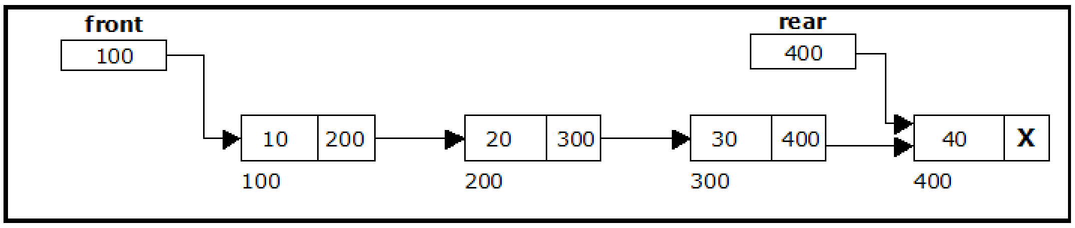
1.insertQ(): adds an element to the end of the queue. Q.

2.deleteQ(): This function deletes the first element of Q.

3.displayQ(): displays the queue's elements.

Linked List Execution of Queue:

A queue can be represented as a linked list. Data is erased from the front end and inserted at the back end of a queue. Similar procedures can be performed on the two ends of a list. For our linked queue implementation, we employ two pointers, one in front and one in back.



Queue Applications:

1.It is used to schedule jobs for the CPU to process.

2.When many users deliver print jobs to the same printer, each print job is stored in the printing queue. The printer then prints those jobs in the order of first in, first out (FIFO).

3.Breadth first search finds an element in a graph using a queue data structure.

Circular Queue:

By considering the array Q[MAX] as circular, a more efficient queue representation is created. The queue might hold an unlimited number of things.

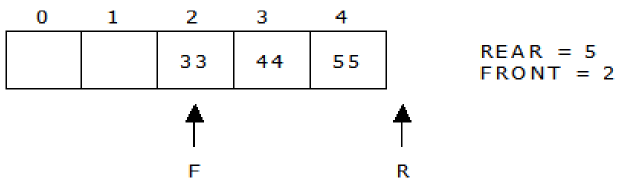
This queue implementation is known as a circular queue because it uses its storage array as if it were a circle rather than a linear list.

There are two issues with linear queueing. They are as follows:

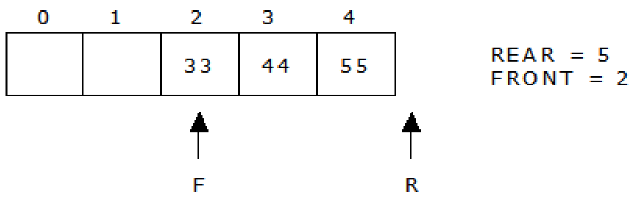
Time consuming: the linear time required to get the elements to the front of the queue.

Signaling queue full: even if the queue has an empty position.

For example, let us consider a linear queue status as follows:



Next, add another element to the queue, say 66. We are unable to add 66 to the queue because the back has exceeded the queue's maximum capacity (i.e., 5). There will be a signal indicating that the queue is filled. The current queue status is as follows:

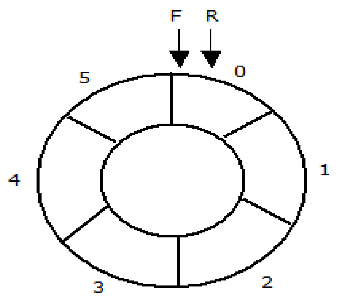


This problem can be solved by treating queue position zero as a position that comes after position four, and then treating the queue as a circular queue.

If we reach the end of the circular queue for inserting elements, we can insert additional components if the slots at the beginning of the circular queue are empty.

Circular Queue Representation:

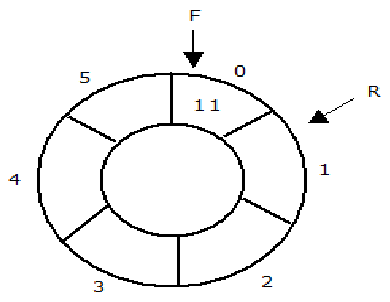
Consider a circular-queue with a maximal (MAX) capacity of 6 elements. The queue is initially blank.



Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

Circular Queue

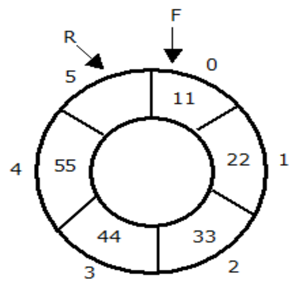
Now, insert 11 to the circular queue. Then circular queue status will be:



FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

Circular Queue

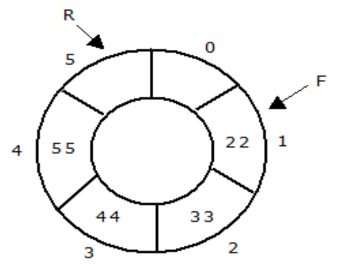
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

Circular Queue

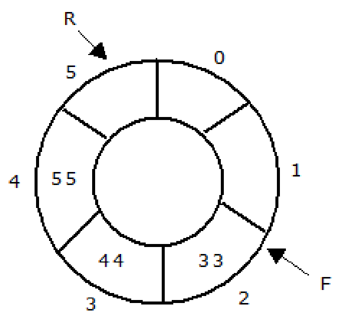
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4

Circular Queue

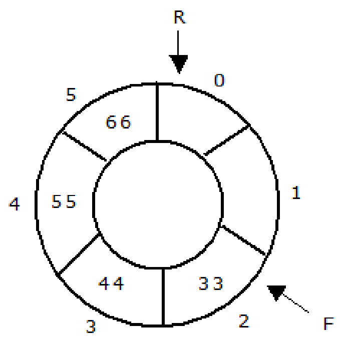
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

Circular Queue

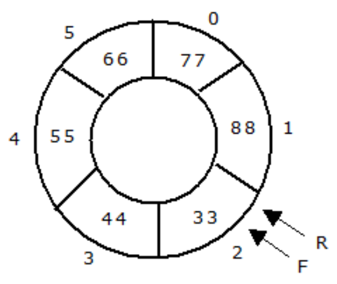
Again, insert another element 66 to the circular queue. The status of the circular queue is:



```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Circular Queue

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



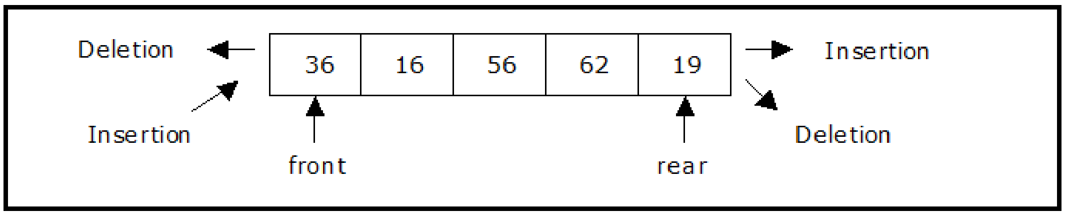
```
FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6
```

Circular Queue

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is full.

Deque:

In the preceding section, we saw a queue into which we enter items at one end and withdraw them at the other. In this section, we look at a queue extender that allows us to insert and remove items from both ends of the queue. This data structure is known as a deque. The term deque is an abbreviation for double-ended queue.



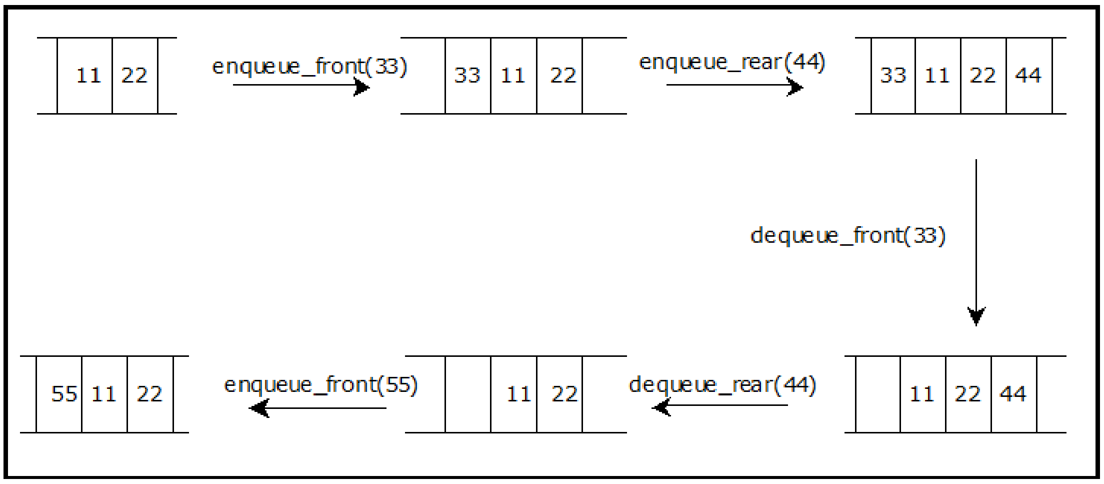
A deque has four operations.

enqueue_front: add an element to the front.

dequeue_front: remove an element from the front of the queue.

enqueue_rear: insert element in the back.

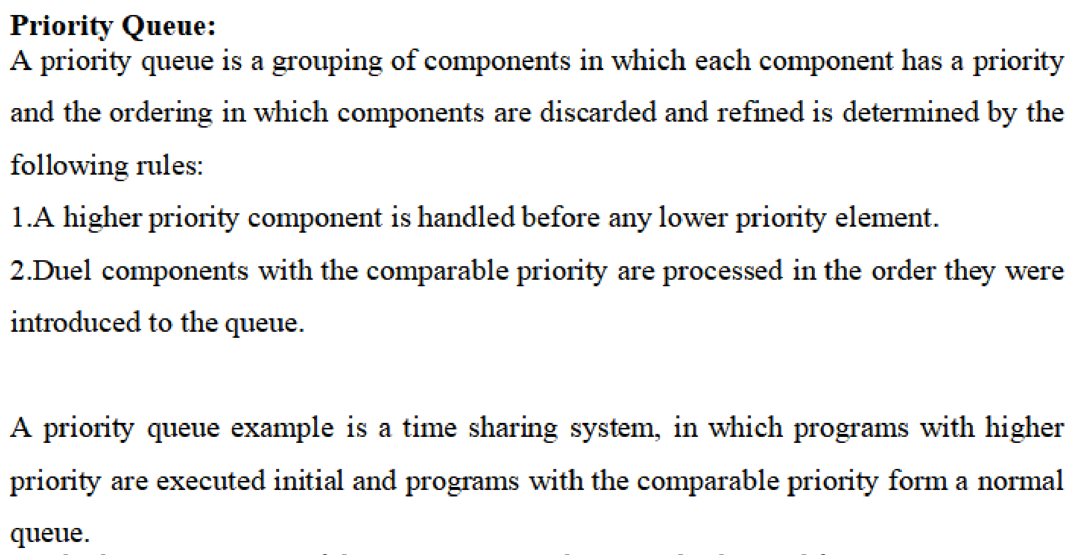
dequeue_rear: remove the element at the back of the queue.



Deque comes in two varieties. They are as follows: input restricted deque (IRD) and output restricted deque (ORD).

An Input restricted deque is a deque that enables insertions at one end but deletions at both ends.

A deque that allows deletions at one end but allows insertions at both ends is known as an output restricted deque.



Priority Queue:

A priority queue is a grouping of components in which each component has a priority and the ordering in which components are discarded and refined is determined by the following rules:

1.A higher priority component is handled before any lower priority element.

2.Duel components with the comparable priority are processed in the order they were introduced to the queue.

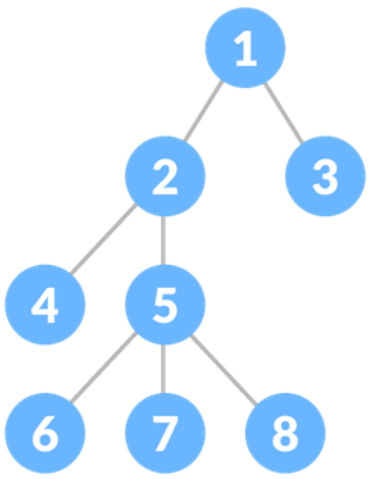
A priority queue example is a time sharing system, in which programs with higher priority are executed initial and programs with the comparable priority form a normal queue.

Heap is an high-octane version of the Priority Queue that may also be used for sorting purposes, which is known as heap sort.

MODULE-4

Trees

In computer science, trees are extensively employed and essential to many algorithms and data storage uses. The terms that are essential to understanding tree data structures are listed below.



Tree

Nodes are 1, 2, 3, 4, 5, 6, 7, and 8. A root node is 1. 2 and 3 are parents of 1.

2, 3, are brothers and sisters. The leaf nodes are 4, 6, 7, and 8. The internal, or non-leaf, nodes are 2, 5. The tree's height or depth is three.

A tree with two root nodes is called the left subtree. There is just one node in the Right-Sub-Tree, which is node 3.

1. **Tree:** An edge-connected hierarchical data structure with nodes that are each connected by at least one child node.

2. **Node:** A single link in a tree data structure that is composed of linkages to its child nodes as well as data.

3. **Root:** The highest point in a tree, which is where a tree journey begins. It is parentless.

4. **Parent Node:** In a tree data structure, a parent link is a node that is connected to one or more child nodes. In the hierarchy, it is positioned above its progeny.

5. **Child Node:** In a tree, nodes that are related to a parent node. In the hierarchy, they are beneath the parent node.

6. **Leaf Node (Terminal Node):** An unbroken link in a tree or a node with a zero out-degree.

7. **An internal or non leaf link** in a tree is one that has one or more child nodes.

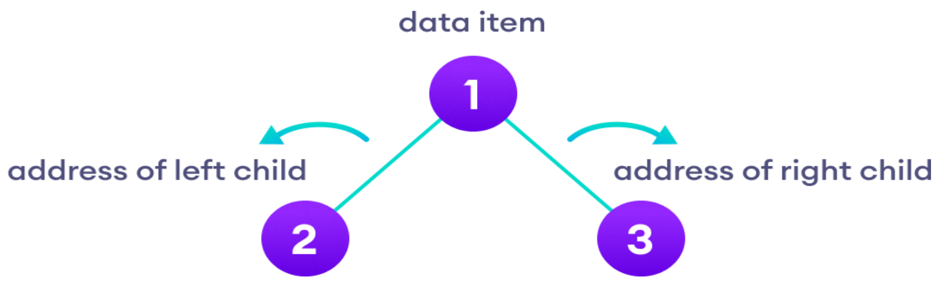
8. **Siblings:** In a tree, siblings are nodes that have the same parent node.

9. **Depth:** A node's elevation or separation from the root node. The root node has a height of zero.

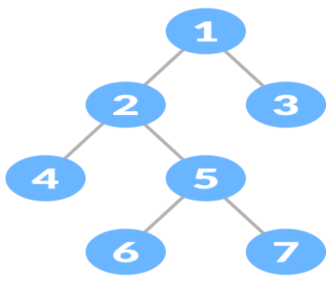
10. **Height or Depth of Tree:** The maximum distance that a tree's root can go to reach a leaf node. It stands for the tree's deepest point.

11. **Subtree:** A tiny tree contained within a bigger tree. It is made up of a node and every child that it has.

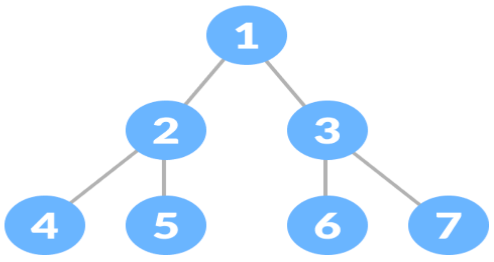
12. **Binary Tree:** A tree where each node has a maximum of two children, known as the left and right children.



Full-Binary Tree :



Perfect Binary-Tree:



13. **Balanced Tree:** Also known as a height-balanced binary tree, a balanced binary tree is one in which the difference in height between a node's left and right subtrees is limited to 1.

The prerequisites for a height-balanced binary tree are as follows:

1. For each node, there is only one difference between the left and right sub trees.

2. The balance of the left-sub-tree.

3. A balanced sub tree is selected.

Binary Tree in Balance:

Complete Binary Tree: In a complete binary tree, every level of the tree is filled to the brim, maybe with the exception of the final level, which is filled from left to right. Alternatively put, in an entire binary tree:

1. There are nodes on every level, possibly with the exception of the final one.

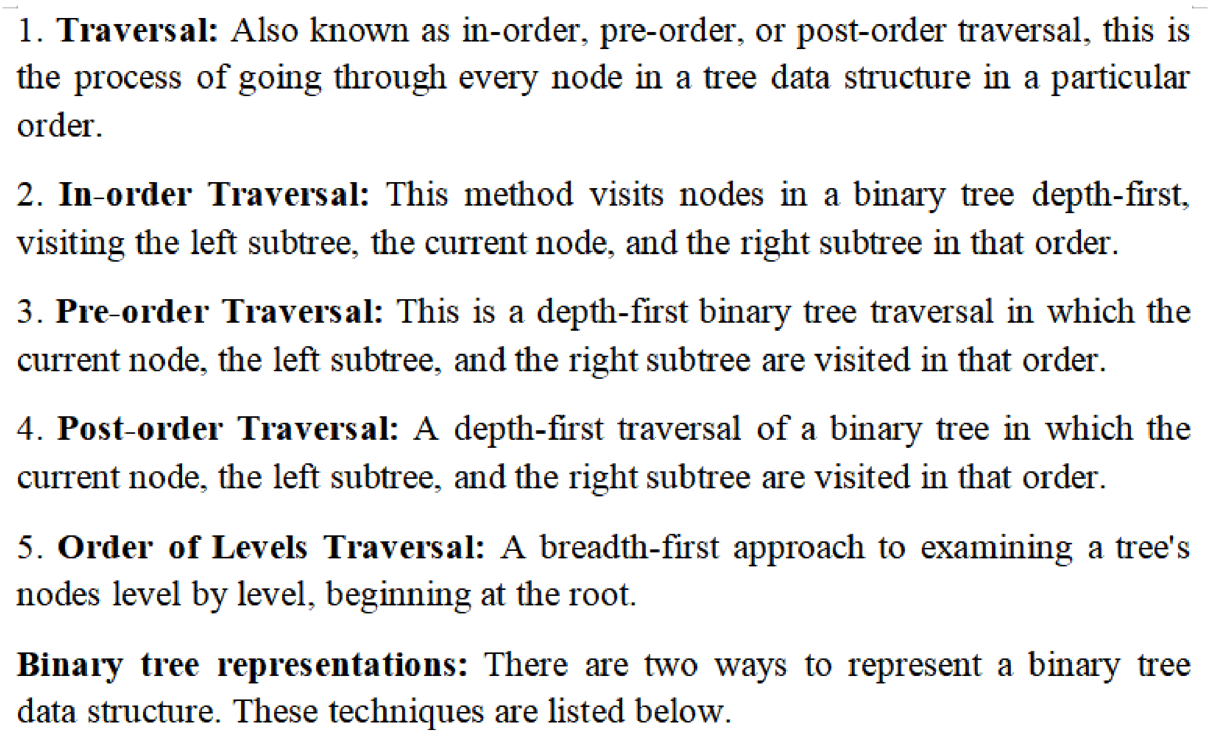
2. Nodes are added from left to right if the final level is not entirely occupied.

A complete binary tree has the following essential features:

1. It is a binary tree.

2. Because it is perfectly balanced, the tree's height is kept to a minimum relative to the number of nodes it has.

3. Because the structure is complete, adding and removing nodes from a binary tree is usually efficient.



1. **Traversal:** Also known as in-order, pre-order, or post-order traversal, this is the process of going through every node in a tree data structure in a particular order.

2. **In-order Traversal:** This method visits nodes in a binary tree depth-first, visiting the left subtree, the current node, and the right subtree in that order.

3. **Pre-order Traversal:** This is a depth-first binary tree traversal in which the current node, the left subtree, and the right subtree are visited in that order.

4. **Post-order Traversal:** A depth-first traversal of a binary tree in which the current node, the left subtree, and the right subtree are visited in that order.

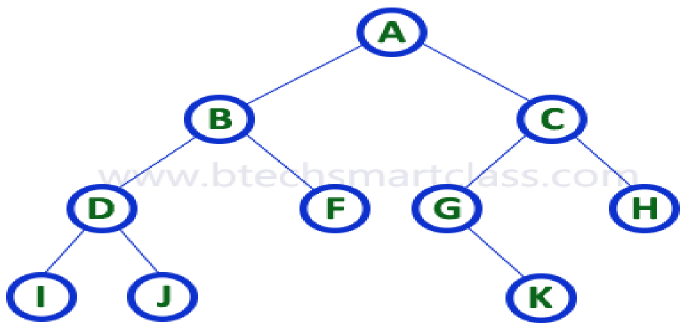
5. **Order of Levels Traversal:** A breadth-first approach to examining a tree's nodes level by level, beginning at the root.

Binary tree representations: There are two ways to represent a binary tree data structure. These techniques are listed below.

1. The Array Diagram

Linked List Illustration No. 2

Take a look at the binary tree below.

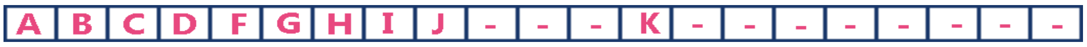


Binary Tree Array Representation

One-dimensional arrays, or 1-D arrays, are used in array representations of binary trees. Let's say that the array index is 1. Examine the binary tree example above, which is shown in the tailing manner.

An array is utilised to store a tree's root[1]. The left child of a node, let's say m, is stored in array[2*k], and the right child of a node, let's say m, is stored in array[2*k+1].

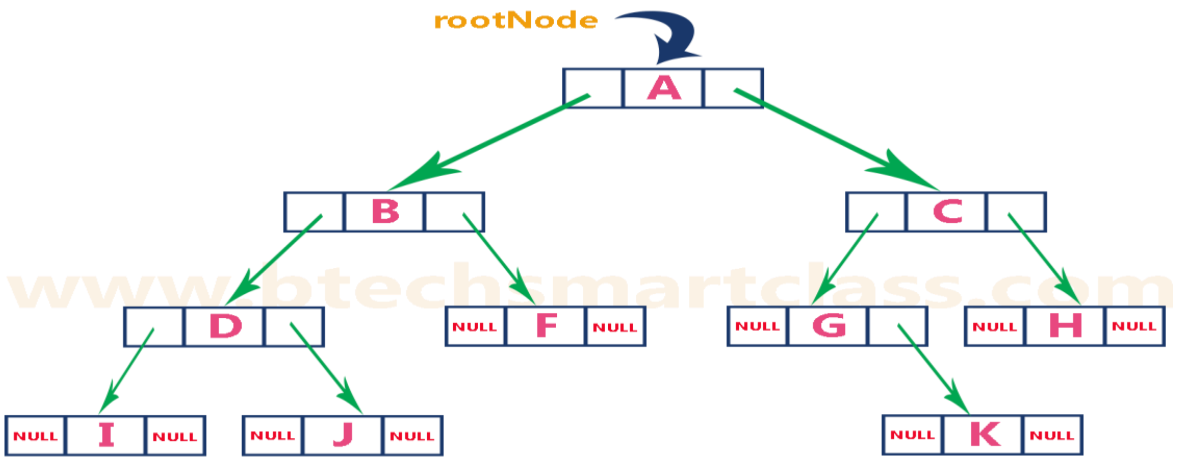
One-dimensional arrays with maximum sizes of 2n + 1 are required in order to use array representation to mean a binary tree of level 'n'.


A | B | C | D | F | G | H | I | J | - | - | - | K | - | - | - | - | - | - | - | -

Binary Tree Representation in Linked Lists

We represent a binary tree using a double-linked list. Each node in a double-linked list has three fields. The left child address is stored in the first field, the actual data is stored in the second, and the right child address is stored in the third.

A node in this representation of a linked list has the structure shown below.



The binary tree example above that uses a linked-list internal representation is displayed as follows.

C **Tree Traversal Algorithms:**

A binary-tree must be traversed by visiting each node in a certain order. In-order traversal, pre-order traversal, and post-order traversal are the three common techniques for navigating a binary tree. Every method specifies a distinct order for visiting nodes.

```
//#include <stdio.h>
//#include <stdlib.h>
// Define a basic binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;

// In-order traversal: left, root, right
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}
// Pre-order traversal: root, left, right
void preOrderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}
```

```
// Post-order traversal: left, right, root
void postOrderTraversal(struct Node* root) {
    if (root != NULL) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        printf("%d ", root->data);
    }}
int main() {
    // Creating a sample binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
     // Perform traversals
     printf("In-order traversal: ");
     inOrderTraversal(root);
     printf("\nPre-order traversal: ");
     preOrderTraversal(root);
     printf("\nPost-order traversal: ");
     postOrderTraversal(root);
     return 0;
}
```

1. **Recursion-Based In Order Traversal Implementation**

a. First visit the left node.

b. Then visit the root node.

c. Then visit the right node.

In the above order specified when performing an in order traversal.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.value, end=" ")
        inorder_traversal(node.right)
# Example usage:
# Construct a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
# Perform in-order traversal  print("In-order Traversal:")
inorder_traversal(root)
```

This will output:

```
In-order Traversal:4 2 5 1 3
```

2. Using Recursion to Implement Pre Order Traversal

a. First traverse the root node.

b. Then visit the left node.

c. Then visit the right node.

In the above order specified when performing a pre order traversal.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def preorder_traversal(root):
    if root is not None:
        # Visit the root node
        print(root.value, end=" ")
        # Recur on the left subtree
        preorder_traversal(root.left)
        # Recur on the right subtree
        preorder_traversal(root.right)
# Example usage:
# Create a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
# Perform pre-order traversal preorder_traversal(root)
```

3. Using Recursion to Implement Post-Order Traversal

a. First visit the left node.

b. Then visit the right node.

c. Then traverse the current node.

In the above order specified when performing a post-order traversal.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def postorder_traversal(node):
    if node:
        postorder_traversal(node.left)
        postorder_traversal(node.right)
        print(node.value, end=' ')

# Example usage:
# Construct a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
```

Binary Search Structure:

A particular kind of binary tree data structure called a Binary Search Tree (BST) is especially effective at finding, adding, and removing elements because it adheres to a predetermined set of rules. Each node in a BST can have a maximum of two offspring: a left and a right child. The values (or keys) of the nodes in the left subtree are less than the value of the current node, and the values in the right subtree are greater, which is the essential characteristic of a BST.

The following are the essential traits and guidelines of a Binary Search Tree:

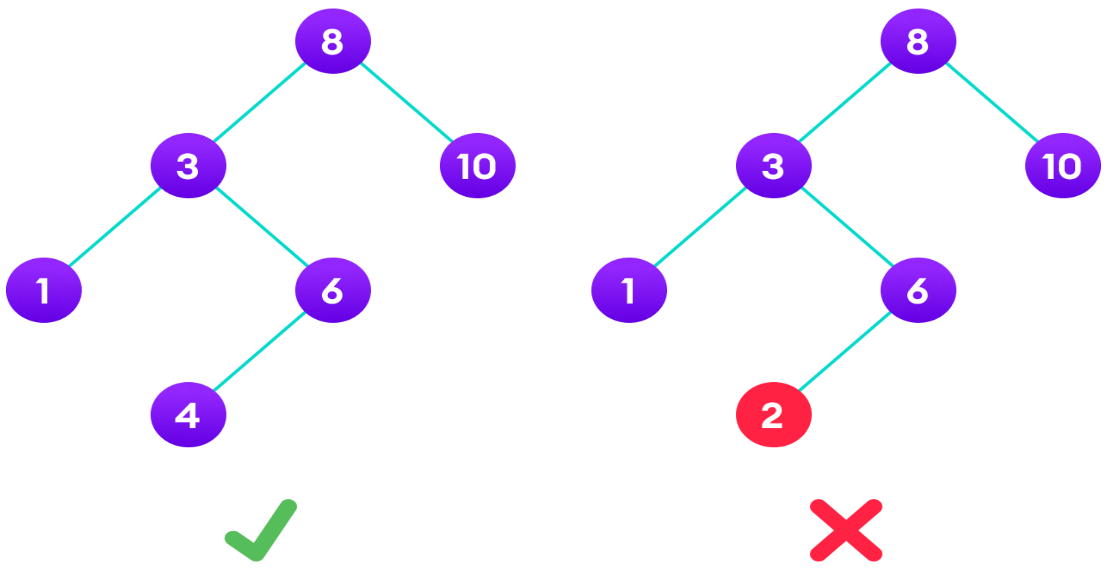
1. Structure of Binary Trees: A BST is a binary tree, as the name implies, meaning that each node can have a maximum of two children: a left child,
2. Performing a Binary Search Tree Search:

In order to find a particular value in a Binary Search Tree (BST), one must traverse the tree using its ordering property as a guide. Here is an explanation and a C algorithm for searching in a BST:

Performing a Binary-Search-Tree Search:

In order to find a particular value in a Binary Search Tree (BST), one must traverse the tree using its ordering property as a guide. This is an explanation and a C algorithm for searching in a BST.
1. The search function requires two inputs: the value we wish to search for (key) and a pointer to the BST root (root).
2. The function first determines whether the data of the root node matches the key or whether the tree is empty (root is NULL). The root node is returned, signifying that the key has been located, if either of these circumstances holds true.
3. Because the key needs to be on the left side according to the BST's ordering property, it recursively calls search on the left subtree if the key is smaller than the root's data.
4. Because the key must be on the right subtree, it recursively calls search on the right subtree if the key value is larger than the roots data.

In a Binary-Search-Tree (BST), adding a new node is as follows:



Finding the ideal location for a new node while preserving the ordering property of the Binary Search Tree (BST) is the process of inserting a new node. This is an example of a C algorithm for a BST insertion operation.

1. To create a new BST node with the given key, use the create Node function. The node's memory is allocated, its data is set to the specified key, and the left and right pointers are initialized to NULL.
2. The insert function requires two inputs: the value to be inserted (key) and a pointer to the BST root (root).
3. The insert function creates a new node with the key and returns it as the new root if the tree is initially empty (root is NULL).
4. The function compares the key with the data of the current node if the tree is not empty.
5. It recursively calls insert on the left subtree if the key is smaller than the data of the current node.
6. It recursively calls insert on the appropriate subtree if the key is greater than or equal to the data of the current node since the ordering property requires that the key be inserted in the appropriate subtree.
7. After inserting the key, the function returns the updated root.

Create an empty BST, use the insert function to add multiple values to it, and then run an in order traversal to copy the contents of the Tree in the main function.

In a Binary-Search-Tree (BST), deletion:
1. The delete function requires two inputs: the value to be deleted (key) and a pointer to the BST root (root).
2. Since there is nothing to delete, it returns NULL if the tree is initially empty (root is NULL).
3. It recursively calls delete on the left subtree if the key is less than the data of the current node.
4. It recursively calls delete on the appropriate subtree if the key exceeds the size of the data on the current node.
5. The node has been determined to be deleted if the key corresponds with the data currently on file.

Three cases are examined for deletion:

Node without children: To update the parent's pointer, we just release the node and return NULL.

When deleting a node with two children, it is necessary to locate the minimum value node in the appropriate subtree, which is done using the findMin function.

Create a sample BST in the main function, use the delete function to remove a particular value (key), and then print the tree before and after the deletion using an in-order traversal to show how the deletion operation was carried out.

The vertical or horizontal depth of a tree:
The longest downward path from a node to a leaf in a tree determines that node's height. The tree's height equals the height of its roots. Therefore, we must consider each link in the tree in-order to regulate its height.

AVL Tree:
The AVL tree is a type of binary search tree wherein the height difference between a node's left and right subtrees is either equal to or less than one. The term "AVL tree," or balanced binary tree, refers to the method of balancing the height of binary trees that was created by Adelson, Velsky, and Landis.

The factor of Balance in AVL Tree:

Height(left subtree) – height(right subtree) equals the balance factor.
Either -1, 0, or +1 should be the balanced factor. If not, the tree will be regarded as being out of balance.
The definition of an AVL tree is as follows: Let T be a non-empty binary tree, and its left and right subtrees are TL and TR.
If both TL and TR have the same height, then the tree is height-balanced.
• hL - hR<= 1, where TL and TR's heights are represented by hL - hR.
Depending on whether the height of a node's left subtree is greater than, less than, or equal to the height of its right subtree, the balance factor of that node in a binary tree can have a value of 1, -1, or 0.

AVL Tree Representation

AVL Rotations: In order to create a balanced tree, rotations are the mechanisms that move some of the unbalanced tree's subtrees. An AVL tree can rotate in any of the following four ways to balance itself:

1. Rotation to the left
2. Rotation to the right
3. Rotating left to right
4. Rotating right to left

The following two rotations are double rotations after the initial two single rotations.

Rotation to the left

We execute a single left rotation when a node is added to the right subtree of the right subtree if the tree becomes unbalanced.

In this case, a node has been inserted into the right subtree of node A, causing node A to become unbalanced. Making A the left subtree of B allows us to execute the left rotation.

Rotation to the right

AVL tree:The imbalanced node turns to the right and becomes the left child's right child, as seen.

Turning from the Left to the Right

A somewhat more complex variation of the rotations that have already been covered is called double rotations. To understand them better, we should watch each and every action taken during rotation. Let's practice turning from left to right first. A left-right rotation is produced by combining the rotations of the left and right.
The fundamental functions of AVL trees are as follows: 1. Locate a specific node; 2. Add a node; and 3. Remove a node.

Looking around

1. Begin at the root node in order to search for a node with a specific value x.
2. Examine this root node's value in relation to x.
3. Return a pointer to this node and declare the node found if the two are equal.
4. If the two do not equal, determine if x is greater or less than the root node's value.
5. Repeat the search for the correct subtree if x is greater than the value of the root node.
6. If the value of x is less than that of the root node, repeat the search for its Search for the left subtree of the root node again if x is less than the value of the root node.
7. Come to the conclusion that there are no nodes in the AVL tree with values equal to x if you come across a NULL at any stage of the search.
8. Return NULL

Insertion Algorithm:

To carry out the insertion operation in an AVL Tree, the following procedures must be followed.
Step 1: Establish a node
Step 2: Determine whether the tree is empty
Step 3: The newly created node will become the AVL Tree's root node if the tree is empty.
Step 4: In the event that the tree is not empty, we insert nodes using the Binary Search Tree and determine the node's balancing factor.
Step 5: If the balancing factor is greater than ±1, we rotate the aforementioned node appropriately and proceed with the insertion from Step 4.

Removal of a Node from an AVL Tree:

There are three distinct circumstances in which deletions occur in AVL Trees:
In case the node to be deleted is a leaf node, it can be deleted without any replacement since it doesn't affect the binary search tree property. This is scenario number one. Rotations are used to restore equilibrium because it can become upset.
• Scenario 2 (Deletion of a node with one child): If the node that is to be deleted has a single child, substitute the value from its child node for the node's value. Next, remove the child
In the third scenario, which involves deleting a node with two children, locate the node's in order successor and substitute its value with the in order successor value. Attempt to remove the in order successor node after that. Use balance algorithms if, after deletion, the balance factor is greater than 1.
An unbalanced AVL tree may result from a deletion operation. Rotations classified as L and R are required for rebalancing the tree following deletion. The R category is further divided into R0, R1, and R-1 rotations, whereas the L category is further divided into L0, L1, and L-1 rotations. When a node is deleted, the balance factors of both the parent and the parent's ancestor are altered. The parent's balancing factor could be zero.

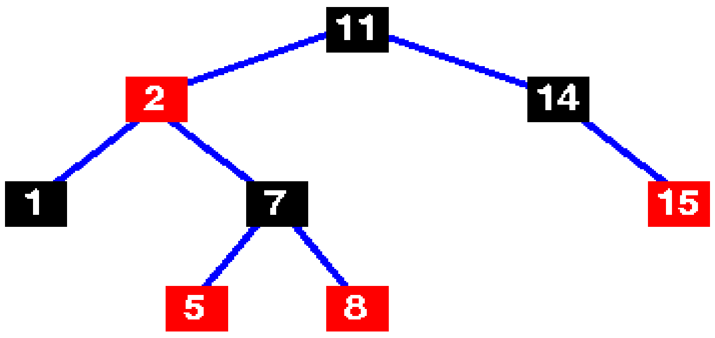
• Red -Black Trees

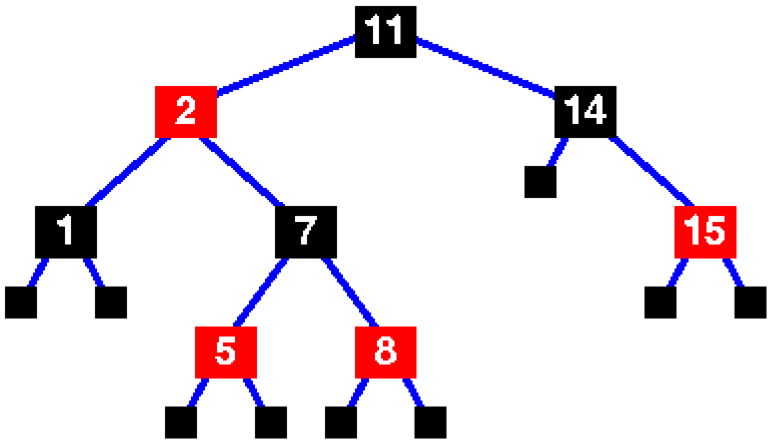
An additional attribute added to each node in a red-black tree is its color, which can be either red or black. This makes the tree a binary search tree. It is also necessary to monitor each node's parent.
Red-black tree definition / Red-Black Tree Properties:

A binary search tree with the following red-black properties is called a red-black tree:
1. All nodes are either black or red.

2. There is a black root node.
3. There are no black leaves (NULL).

1. A node's two offspring are black if it is red. It means that red nodes cannot be next to each other on any path that leads from the root to a leaf. Nonetheless, a sequence of black nodes could contain any number of them. We refer to this as red condition.





Advantages:

1. The time complexity of Red-Black Trees for simple operations such as searching, insertion, and deletion is guaranteed to be O(log n).

2. Red Black Trees can balance themselves.

3. Red Black Trees' adaptability and effective performance make them suitable for a variety of uses.

4. Red Black Trees have a reasonably straightforward and understandable balance system.

Disadvantages:

1. One drawback of Red-Black Trees is that each node needs an additional bit of storage to store its color—red or black.

2. Implementation Complexity.

3. While Red Black Trees offer effective performance for fundamental tasks, they might not be the ideal option for particular data kinds or use cases.

Working with Red-Black Trees:

1. Type in a crucial value (insert).

2. Use a lookup or search to see if a key value is present in the tree.

3. Delete the key value from the tree.

4. Print every key value in a sorted list (print)

5. Change the color

In Red-Black Trees, Rotation / Restructure Operations:

A Node is Added to a Red Black Tree:

The process for inserting a key into a red-black tree is exactly the same as that of a binary search tree. On the other hand, giving the inserted node a color could go against the red-black tree's characteristics and result in imbalances. Classifications for the imbalances include LLb, LLr, RLb, RLr, LRb, LRr, RRb, and RRr. The type LLr, RLr, LRr, and RRr imbalances with "r" as their suffix only call for.

Removing a Node from a Red Black Tree:

The process for removing a key from a red-black tree is precisely the same as that for a binary search tree. There is no way that the black condition will be broken if the deleted node is red. In the event that the deleted node, v, is a black node, there is a chance that the black condition will be broken. This throws the red-black tree out of balance.

Depending on whether the deleted node (v) occurs to the right or left of its parent, the imbalance is categorized as Left (L) or Right (R). If sibling_v, v's black sibling, is present, the imbalance is further categorized as either Lb or Rb. Should v's sibling, sibling_v, be red, then

Splay Trees :

Research on node access has demonstrated that once a node or piece of information is accessed, it is likely to be accessed again. Binary-search-trees with a self adjusting mechanism are called Splay Trees. The Splay Tree data structure drastically alters its shape when a node is accessed, pushing that link in the direction of the base node.

This modification improves the efficiency of subsequent accesses to the node. As a result, the most recent node that is accessed—whether for insertion or search—is moved closer to the root each time. Methodically, splay rotations—which are essentially AVL tree rotations—are followed in order to pushing the node upwards toward the base.

The other nodes would be forced to move away from the root as a result.

This modification improves the efficiency of subsequent accesses to the node. As a result, the most recent node that is accessed—whether for insertion or search—is moved closer to the root each time. Methodically, splay rotations—which are essentially AVL tree rotations—are followed in order to pushing the node upwards toward the base.

The other nodes would be forced to move away from the root as a result.

Below are the different rotations that need to be made in the splay tree.

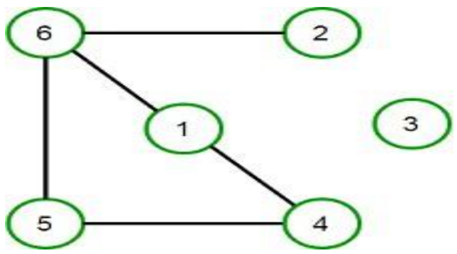
- Rotating rightward, or zigzag
- "Zig zig"—two rotations to the right
- Left rotation, or zigzag rotation
- [Two Left Rotations] Zagzag
- The zigzag [Zig came next to Zag]
- The zig zig [Zig came next to Zag]

MODULE-5

Graphs

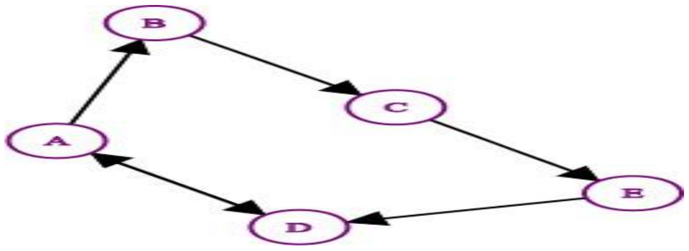
Graphs are primal data structures used in computers & scientific discipline to represent a wide range of relationships and connections between objects. Graph terminology belong of various terms and concepts used to describe and analyze graphs. Here are some key graph terminology and definitions:

1. A Graph is an Order pair Graph = (VS, ES) comprises a set VS of nodes or vertices and a set of edges ES. An edge has starting node and ending node from VS.

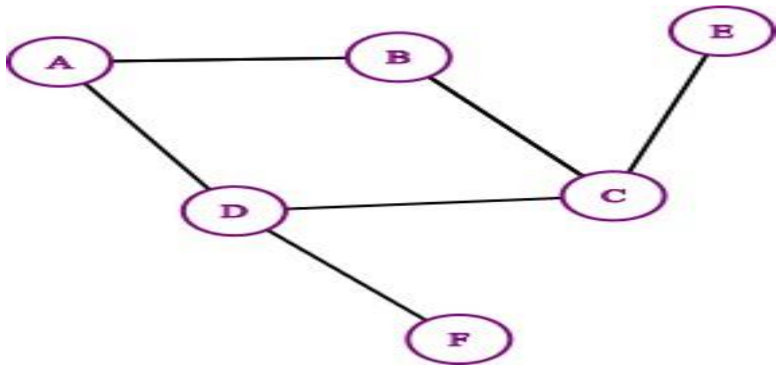


   V = { 1, 2, 3, 4, 5, 6 }
   E = { (1, 4), (1, 6), (2, 6), (4, 5), (5, 6) }

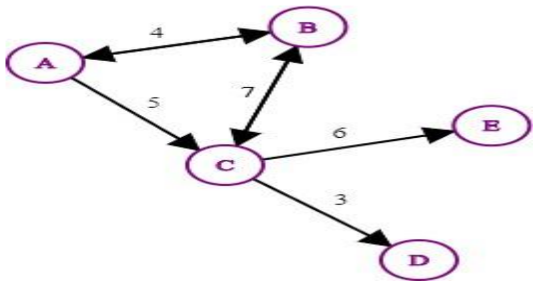
2. Vertex (Node): A vertex, often referred to as a node, is a fundamental unit of a graph. It represents an object or entity. In a social network, for example, each person can be represented as a vertex.
3. Edge (Arc): An edge is a connectivity between 2 vertices. It represents a relationship or connection between the corresponding objects. In a social network, an edge between two vertices (persons) may indicate a friendship.
4. Directed Graph (Digraph): In a directed graph, each edge has a direction, meaning it goes from one vertex (the source) to another vertex (the target). These are often used to model situations where the relationship between vertices is asymmetrical.

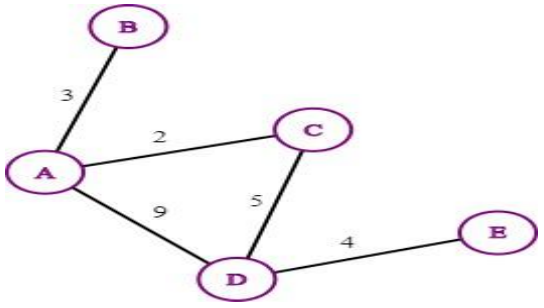


5. Undirected Graph: In an undirected graph, edges have no direction. They simply represent a mutual relationship between two vertices. If there's an edge between vertex A and vertex B, it implies that B is also connected to A.



6. Weighted Graph: If edges of a graph are associated with weights, then it is referred as a weighted. These weights can refers to various properties such as distance, cost, or capacity. Weighted graphs are often used in algorithms like Dijkstras shortest path algorithm. The weighted edge graph may be directed or undirected.

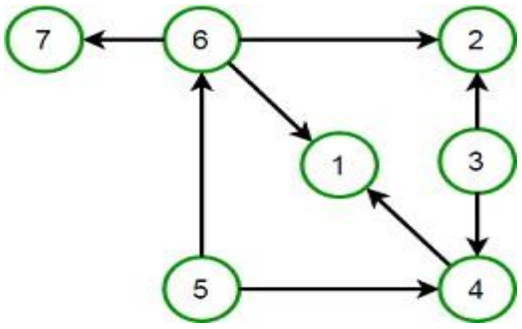




Some interpretations of a weight on an edge:

- Weight may refer to the distance between two nodes.
- Weight may refer to the time required to travel between two nodes.

7. Directed-Acyclic-Graph(DAG):
   A Directed-Acyclic-Graph (DAG) is a oriented graph that consists no cycles.

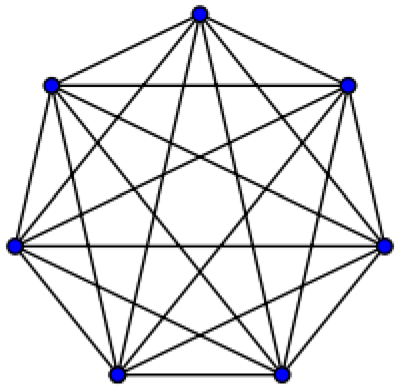


8. Multi graph

A Multi-graph is an undirected graph in which multiple edges (and sometimes loops) are allowed. Multiple edges join same two nodes. A loop is an edge line that joins a node to itself.

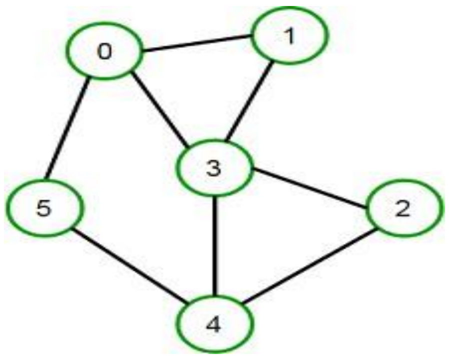
9. Complete graph

Complete graph is a one in which every pair of node links are adjacent

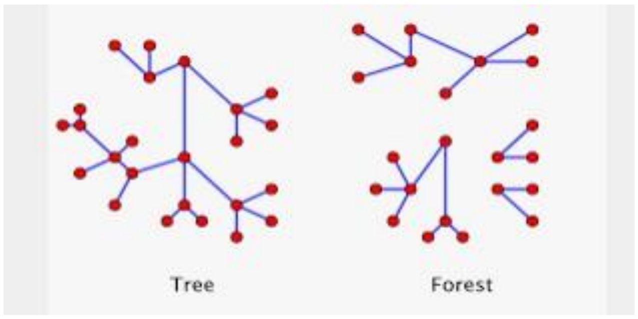


10. Connected graph

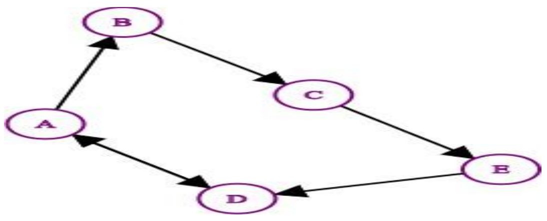
A Connected graph has a path between every couple of nodes. In other words, there are no unreachable nodes in it. A disconnected graph is a graph that is not connected.



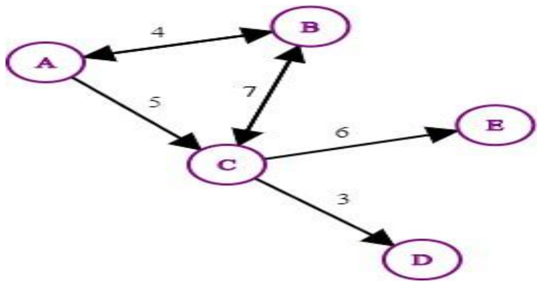
11. A graph is Strongly Connected if there exists a directed path from A to B for every pair of nodes A, B.
12. Degree of a Vertex: The amount of edges connecting to a node determines its degree. In an undirected graph, it is simply the count of adjacent edges. In a directed graph, there are 2 degrees: in-degree (measure of incoming edges) and out-degree (measure of outgoing edges) for each vertex.
13. Path: In a graph, a path line is a sequence of nodes where each adjacent pairs are connected by an edge. The measure of edges in a path is referred as path length.
14. Cycle: If first vertex and last vertex are same in a path, then it is called as a cycle. Cycles can exist in both directed and undirected graphs.
15. Disconnected Graph: A graph is disconnected if it is not connected. This means there are at least two separate groups of vertices that have no path between them.
16. A bridge is an edge whose removal would disconnect the graph.
17. A forest is a graph that lacks cycles. A connected graph without any cycles is referred to as a tree. In a forest, each connected component is a tree.



18. If we remove all the cycles from a Graph, it becomes a tree, and if we remove any edge in the tree, it will become a forest.
19. Subgraph: A subgraph is a graph formed by selecting a subset of vertices and a subset of edges from a larger graph. It preserves the relationships between the selected vertices.
20. Adjacency Matrix (AM): AM is a square matrix used for representing a graph. It is a n by n square matrix, where n represents the number of nodes. The cell value at ith row and jth columnj shows whether an edge between ith node and jth node exists or not. For every pair of nodes, AM keeps a value **1/0/edge-weight** to specify whether the edge exists or not. It requires n^2 space. They can be efficiently used only when the graph is dense.

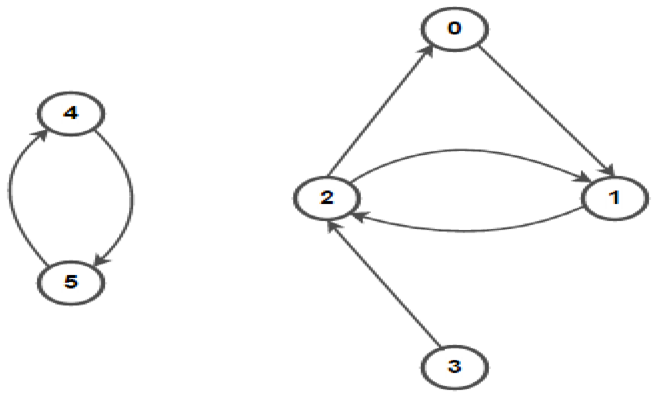


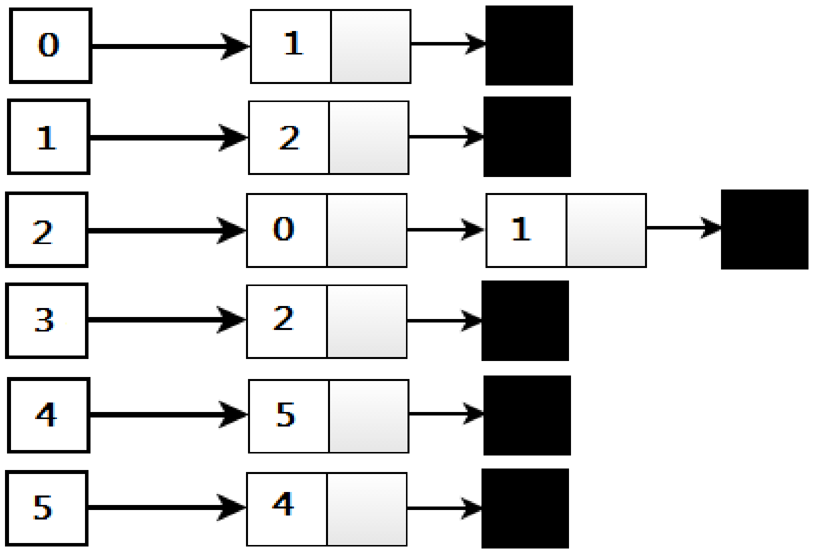
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	1	0	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0



	A	B	C	D	E
A	0	4	5	0	0
B	4	0	7	0	0
C	0	7	0	3	6
D	0	0	0	0	0
E	0	0	0	0	0

21. Adjacency List: An adjacency list is a data structure that represents a graph by storing a list of neighbors for each vertex. It's often more memory-efficient than an adjacency matrix, especially for sparse graphs. Each vertex in the graph is linked to a set of its neighboring vertices or edges, meaning that each vertex maintains a list of adjacent vertices. The specific representation of the adjacency list can vary depending on the implementation.





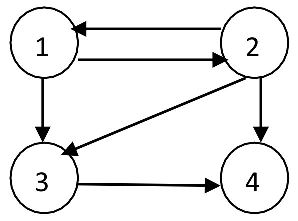
22. Operations on Graph Data Structure

Following are the basic graph operations in graph data structure:

- Add Node, Remove Node – Insert or Delete a node in a graph. (Insert/Delete Vertex)
- Add edge, Remove Edge – Insert or Delete an edge between two nodes. (Insert/Delete Edge)
- Check if the graph contains a given value (Search or Lookup)
- Find path – Discover a path to a destination node from source node. (Find Path)

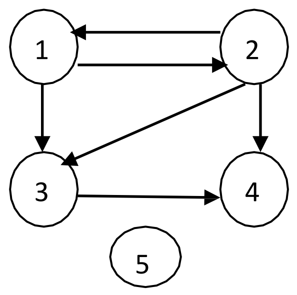
Inserting a Vertex into a Graph:

Example Graph (G) and its Adjacency Matrix: Call this graph as original graph.



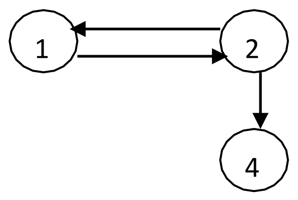
Adjacency Matrix				
	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	0	0	0

Insert Vertex 5 in to graph (G) and show the changes to Adjacency Matrix (addVertex(V_n)):



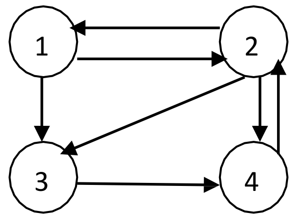
Adjacency Matrix					
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	0	0	0

Delete Vertex 3 in original graph (G) and show the changes to Adjacency Matrix (deleteVertex(V_n)):



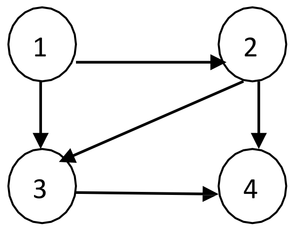
Adjacency Matrix			
	1	2	4
1	0	1	0
2	1	0	1
4	0	0	0

Insert Edge from Vertex 4 to Vertex 2 into original graph (G) and show the changes to Adjacency Matrix (addEdge(V_s, V_e)):



Adjacency Matrix				
	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	1	0	0

Delete Edge from Vertex 2 to Vertex 1 in original graph (G) and show the changes to Adjacency Matrix (deleteEdge(V_s, V_e)):



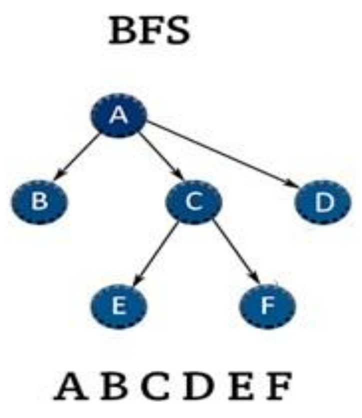
Adjacency Matrix				
	1	2	3	4
1	0	1	1	0
2	0	0	1	1
3	0	0	0	1
4	0	0	0	0

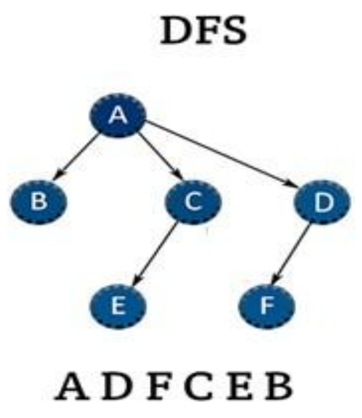
23. **Graph Traversal in Data Structure**

Graph traversal is visiting or updating each node in a graph. It is classified based on the order visiting the nodes. Below are the the 2 traversal techniques:

1. Breadth First Search (BFS) – It is a traversal operation that horizontally traverses the graph. It traverses all nodes at a single level before moving to next level. It begins at the graph's root and traverses all nodes at single depth level before moving on to next level.

2. Depth-First Search (DFS): This is another traversal operation that traverses the graph vertically. It starts with the root node of the graph and investigates each branch as far as feasible before backtracking.





// Algorithm for BFS():

Step1. Initialize all the nodes to ready state (set STATUS = 1)

Step2. Add the starting node into QUEUE. The change the status of that node to waiting (set STATUS = 2)

Step 3: Repeat Step 4 and Step 5 until QUEUE is EMPTY

Step 4: Remove the front node from QUEUE, Process the node, Change its status to processed state (set STATUS = 3)

Step 5: ADD all neighbors of the node which are in the ready state (STATUS = 1) to the REAR of the QUEUE and change status of those nodes to waiting state (set STATUS = 2)

Step 6: Quit.

Implementation of BFS using C:

```
//#include <stdio.h>//
//#include <stdlib.h>//
//#include <stdbool.h>//
//#define// MAXIMUM_NODES 100
// Queue data structure for BFS
struct Queue
{
   int items[MAXIMUM _NODES];
   int f, r;
};
struct Queue* createQueue() {
   struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
   q->f = -1;
   q->r = -1;
   return q;
}
bool isEmpty(struct Queue* q) {
   return q->f == -1;
}
void enqueue(struct Queue* q, int item) {
   if (q->r == MAXIMUM _NODES - 1) {
      printf("Queue is full.\n");
      return;
   }
   if (isEmpty(q))
      q->f = 0;
   q->r++;
   q->items[q->r] = item;
```

```
}
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return -1;
    }
    int item = q->items[queue->front];
    q->f++;
    if (q->f > q->r) {
        q->f = q->r = -1;
    }
    return item;
}
// Graph data structure
struct Graph {
    int V; // Number of vertices
    int** adjMatrix; // Adjacency matrix
};
struct Graph* createGraph(int V) {
    struct Graph* g = (struct Graph*)malloc(sizeof(struct Graph));
    g->V = V;
    g->adjMatrix = (int**)malloc(V * sizeof(int*));
    for (int m = 0; m < V; m++) {
        g->adjMatrix[m] = (int*)malloc(V * sizeof(int));
        for (int n = 0; n < V; n++) {
            g->adjMatrix[m][n] = 0;
        }
    }
    return g;
}
void edgeAdd(struct Graph* g, int s, int d) {
    g->adjMatrix[s][d] = 1;
    g->adjMatrix[s][d] = 1;
}
void BFS(struct Graph* g, int startVertex) {
    bool* visited = (bool*)malloc(g->V * sizeof(bool));
    for (int m = 0; m < g->V; m++) {
        visited[m] = false;
    }
    struct Queue* queue = createQueue();
    visited[startVertex] = true;
    printf("Breadth First Traversal starting from vertex %d:\n", startVertex);
    printf("%d ", startVertex);
    enqueue(queue, startVertex);
    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
        for (int m = 0; m < g->V; m++) {
            if (g->adjMatrix[currentVertex][m] == 1 && !visited[m]) {
                printf("%d ", i);
                enqueue(queue, i);
                visited[i] = true;
            }
        }
    }
    printf("\n");
}
int main() {
    int k = 7;
    struct Graph* g = createGraph(k);
    edgeAdd(g, 0, 1);
    edgeAdd(g, 0, 2);
    edgeAdd(g, 1, 3);
    edgeAdd(g, 1, 4);
    edgeAdd(g, 2, 5);
    edgeAdd(g, 2, 6);
    BFS(g, 0);
    return 0;
}
```

Explanation:

1. The program starts by defining a structure for a queue that will be used for the BFS traversal. The queue is implemented as an array-based data structure.
2. A structure for the graph is defined, including the number of vertices and an adjacency matrix to represent the graph.
3. Functions for creating a graph, adding edges, and performing BFS are defined.
4. In the BFS function, we use a visited array to keep track of visited vertices. A queue is used to keep track of the vertices to be explored. We start with the startVertex and enqueue it. Then, we enter a loop where we dequeue a vertex, mark it as visited, and enqueue its unvisited neighbors. This process continues until the queue is empty.
5. In the main function, a sample graph is created and edges are added to it. The BFS traversal is initiated with a starting vertex (in this case, vertex 0).
6. The BFS traversal is performed and the nodes are printed in the order they were visited, demonstrating the breadth-first exploration of the graph.

//Algorithm for DFS()

Step1. Initialise all the links to ready state (set STATUS = 1)

Step2. Put opening node onto STACK and modify its status as waiting (set STATUS = 2)

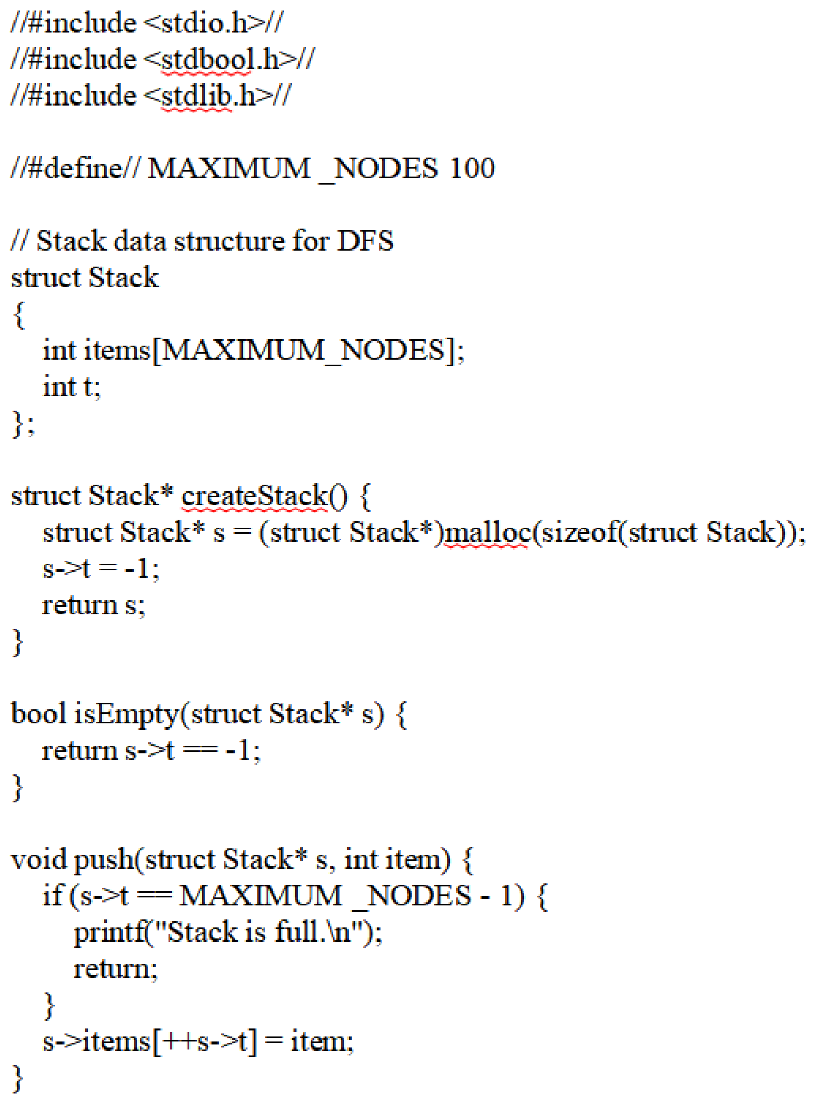
Step 3: Repeat Step 4 and Step 5 till STACK becomes BLANK

Step 4: Remove topmost node on STACK and Process the node. Then modify its state as processed (set STATUS = 3)

Step 5: Add all neighbors of the node which are in ready state (i.e., STATUS = 1) to the STACK and modify their state to waiting (set STATUS = 2)

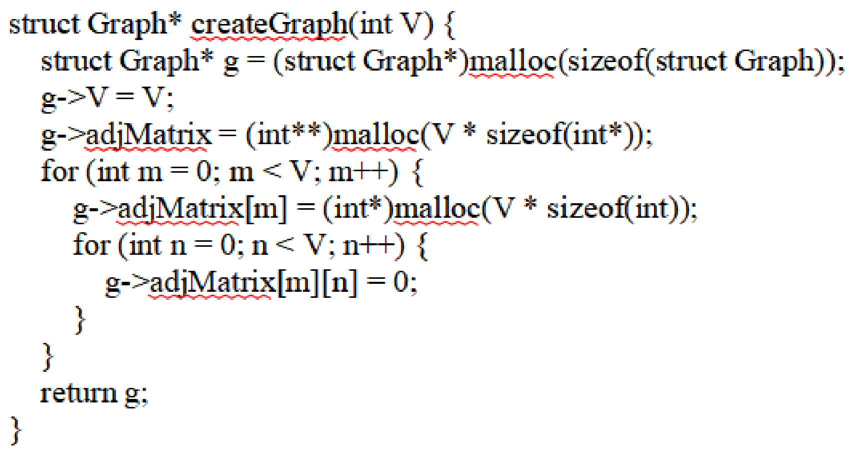
Step 6: Quit.

Implementation of DFS using C:



```
//#include <stdio.h>//
//#include <stdbool.h>//
//#include <stdlib.h>//

//#define// MAXIMUM _NODES 100

// Stack data structure for DFS
struct Stack
{
   int items[MAXIMUM_NODES];
   int t;
};

struct Stack* createStack() {
   struct Stack* s = (struct Stack*)malloc(sizeof(struct Stack));
   s->t = -1;
   return s;
}

bool isEmpty(struct Stack* s) {
   return s->t == -1;
}

void push(struct Stack* s, int item) {
   if (s->t == MAXIMUM _NODES - 1) {
      printf("Stack is full.\n");
      return;
   }
   s->items[++s->t] = item;
}
```

```
int pop(struct Stack* s) {
    if (isEmpty(s)) {
        printf("Stack is empty.\n");
        return -1;
    }
    return s->items[s->t--];
}

// Graph data structure
struct Graph {
    int V; // Number of vertices
    int** adjMatrix; // Adjacency matrix
};
```



```
    struct Graph* createGraph(int V) {
        struct Graph* g = (struct Graph*)malloc(sizeof(struct Graph));
        g->V = V;
        g->adjMatrix = (int**)malloc(V * sizeof(int*));
        for (int m = 0; m < V; m++) {
            g->adjMatrix[m] = (int*)malloc(V * sizeof(int));
            for (int n = 0; n < V; n++) {
                g->adjMatrix[m][n] = 0;
            }
        }
        return g;
    }
```

```
void edgeAdd(struct Graph* g, int s, int d) {
    g->adjMatrix[s][d] = 1;
    g->adjMatrix[d][s] = 1;
}

// Iterative DFS
void DepthFirstSearch(struct Graph* g, int startVertex) {
    bool visited[g->V];
    for (int m = 0; m < g->V; m++) {
        visited[m] = false;
    }

    struct Stack* s = createStack();

    visited[startVertex] = true;
    printf("Depth First Traversal starting from vertex %d:\n", startVertex);
    printf("%d ", startVertex);
    push(s, startVertex);

    while (!isEmpty(s)) {
        int currentVertex = s->items[s->top];

        int found = 0;
        for (int m = 0; m < g->V; m++) {
            if (g->adjMatrix[currentVertex][m] == 1 && !visited[m]) {
                printf("%d ", m);
                visited[m] = true;
                push(s, m);
                found = 1;
                break;
            }
        }

        if (!found) {
            pop(s);
        }
    }
}

int main() {
    int V = 7;
    struct Graph* g = createGraph(V);

    edgeAdd(g, 0, 1);
    edgeAdd(g, 0, 2);
    edgeAdd(g, 1, 3);
    edgeAdd(g, 1, 4);
    edgeAdd(g, 2, 5);
    edgeAdd(g, 2, 6);
    DepthFirstSearch(g, 0);
    return 0;}
```

Explanation:

1. The program starts by defining a structure for a stack, which is used for managing the depth-first traversal.
2. A structure for the graph is defined, including the number of vertices and an adjacency matrix to represent the graph.
3. Functions for creating a graph, adding edges, and performing DFS are defined.
4. The DFS function performs an iterative depth-first traversal. It uses a stack to manage the order of node exploration, starting from the specified source vertex (in this case, vertex 0).
5. The main function creates a sample graph, adds edges to it, and initiates the DFS traversal starting from vertex 0.
6. The DFS traversal is performed iteratively, and the nodes are printed in the order they are visited, demonstrating the depth-first exploration of the graph.

24. Shortest Paths and Minimum Spanning Trees

Spanning trees is a sub graph that connects all nodes of a given graph using minimum number of edges. It may or may not be weighted and does not have cycles.

Spanning trees of any connected undirected graph is a sub-graph, i.e., a tree that binds all nodes that contribute to minimization of amount of the weights of the edges.

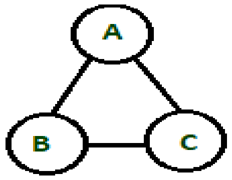
Let G be a graph with nodes set V and edges set E, i.e., G(V, E). Then G'(V', E') is a Spanning Tree of G if it obeys following constraints.

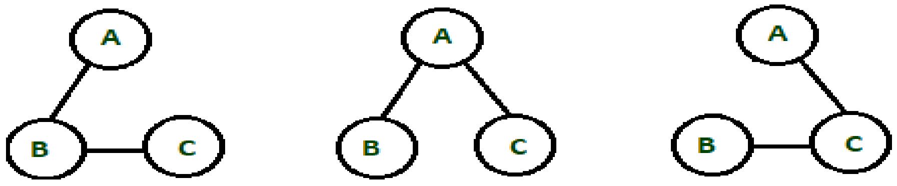
1. V' = V (number of nodes in G' = number of node in G)
2. E' = |V| - 1 (Number of edges G' = number of nodes in G minus 1)

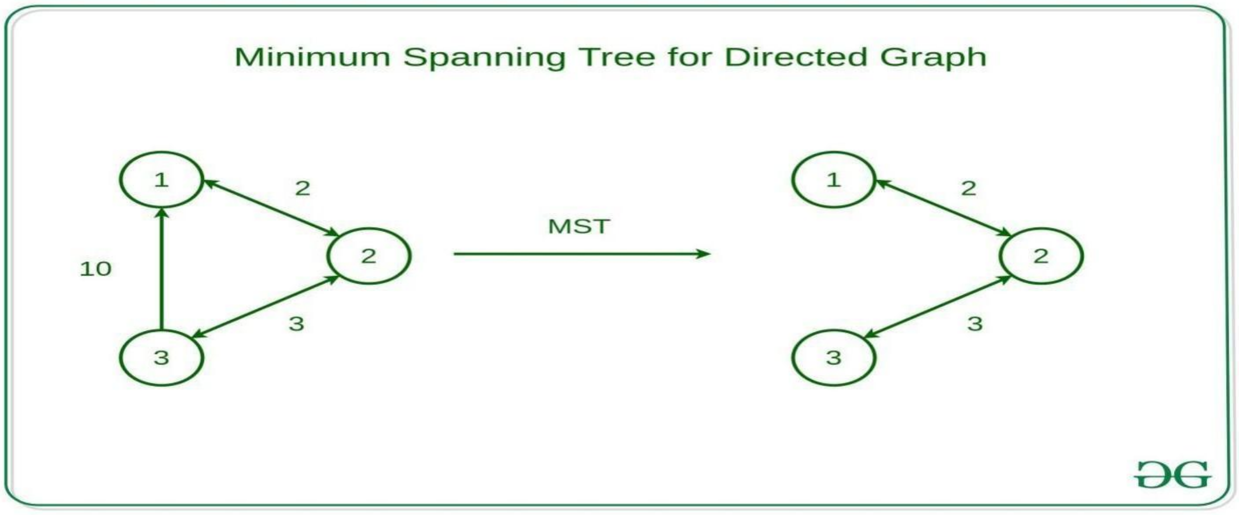
There might be many spanning trees possible for a given graph.

Properties:

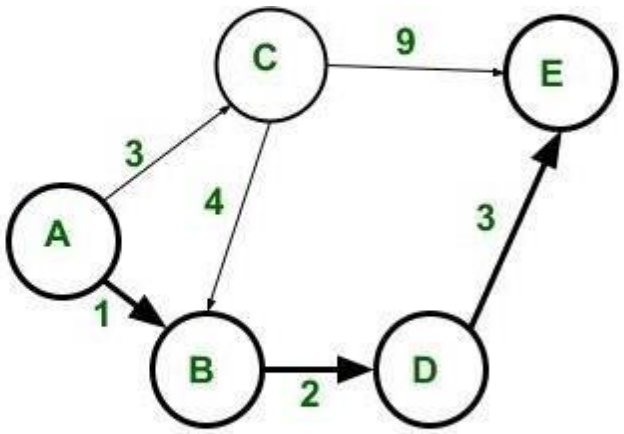
- A spanning trees can exist if a graph is connected. Otherwise many spanning trees called forest exists.
- The number of edges in a spanning tree = e-1, where e= number of nodes a given graph.
- **Cayleys formula**: measure of spanning trees in a complete undirected graph having n nodes K_n **= $\mathbf{n^{n-2}}$.**
- For the K_3 graph, total number of spanning trees = $3^{3-2} = 3$.







The Shortest path:



- Sum of the weights on edges in shortest path (i.e. ABDE) of any pair of vertices (say between A and E) is minimum among all possible paths between that pair of nodes. (i.e., A and E).
- Computing shortest path can be done for directed, undirected or mixed graphs.
- The problem for discovering a shortest path is classified as
    - Single-source the shortest path: Here the calculation of shortest path is done from given source node to every other node of the graph.

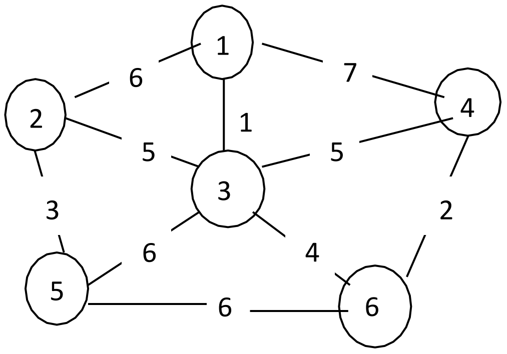
- Single-destination the shortest path: Here the computation of shortest path is done from all nodes of a graph to the given destination node.
- All pairs the shortest path: Here the calculation of shortest path is done for each pair of nodes.

Prim's Algorithm: This algorithm is to get least cost Spanning Tree for a undirected connected graph:

Let G1=(V1, E1) be an connected undirected graph and T1=(V1, E1') is sub-graph of G1 and is the spanning Tree for G1 if T1 is a Tree.

```
Prim(G)
    Begin
        E'= Φ;
        Choose a least cost edge (u1, v1) from E1;
        V1' = {u1};
        While V1' ≠ V1 do
            Let (u1, v1) be a least cost edge such that u1 is in V1' and v1 is in V1 –
            V1';
            Add edge (u1, v1) to set E1';
            Add v1 to set V1'
        End while
    End Prim
```

Example Graph:



Trace of Prims Algorithm:

V = {1, 2, 3, 4, 5, 6}

E = {(1,2), (1,3), (1,4), (2,3), (2,5), (3,4), (3,5), (3,6), (4,6), (5,6)}

E' = {}

Edge	Cost of the Edge	V’	E’
(1,3)	1	1	{}
(1,3)	1	{1, 3}	{(1,3)}
(3,6)	4	{1, 3, 6}	{(1,3), (3,6)}
(6,4)	2	{1, 3, 6, 4}	{(1,3), (3,6), (6,4)}
(3,2)	5	{1, 3, 6, 4, 2}	{(1,3), (3,6), (6,4),(3,2)}
(2,5)	3	{1, 3, 6, 4, 2, 5}	{(1,3), (3,6), (6,4),(3,2),2,5)}

Kruskal's Spanning Tree Algorithm

This algorithm is aimed to discover Minimum spanning Tree (MST) of a given weighted, connected, undirected graph.

In case, the graph is disconnected, on applying Kruskal's algorithm can find the MST of each connected component.

Spanning tree is a tree and also a sub-graph of connected, weighted and undirected graph that contains all the nodes. A minimum spanning tree corresponds to a spanning tree that has minimum total weight, where the overall weights of the spanning tree is equal to sum of the weights of the edges present in it.

Spanning Tree Basic properties:

- More than one spanning tree can exist to the connected graph.
- Spanning trees do not contain loops or cycles.
- As a spanning tree is loosely connected, if removing an edge from the tree will disconnect the graph.
- Addition of an edge to a spanning tree creates a loop because it is acyclic.
- Maximum number of spanning trees = $\mathbf{n^{n-2}}$. N= number of nodes in a complete graph
- In a spanning tree, total number of edges = n-1, where n = total number of nodes.

Steps in Kruskal's Algorithm:

Objective is to remove all parallel edges and loops in given weighted Graph G (V,E).

Step 1 - Initialize MST to null.

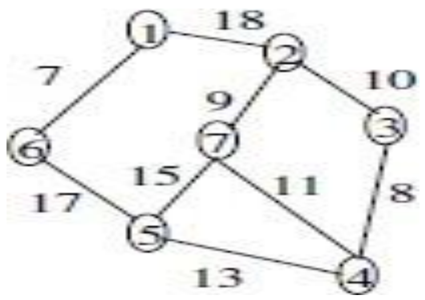
Step 2 − Order all edges of G in their ascending order of weights.

Step 3 – Select the edge that has the minimum or least weight. Add the selected edge to MST if its addition keeps the spanning tree properties remain intact. Otherwise ignore it.

Step 4 - Repeat step 3 till spanning tree is complete. i.e. all nodes are connected in MST.

.

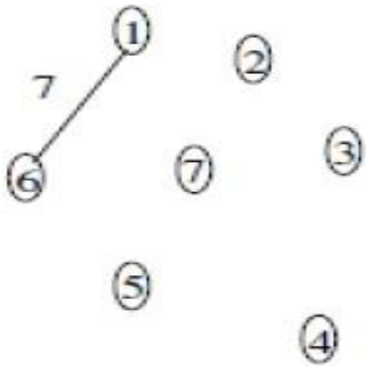
Example: Look at the following example to understand the algorithm of Krushkal.



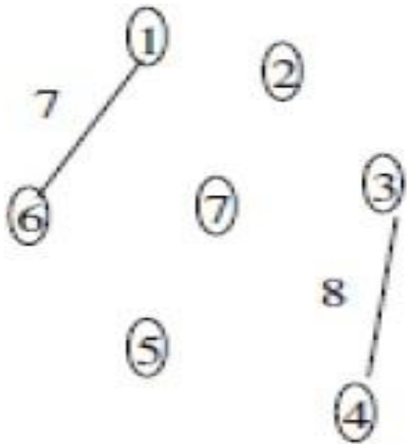
Edges of given graph with ascending order of weights:

Edge	(1, 6)	(4, 3)	(2, 7)	(2, 3)	(7, 4)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	7	8	9	10	11	13	15	17	18

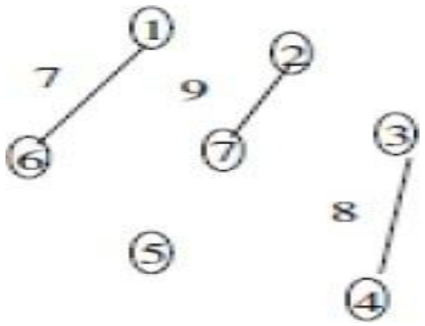
Take edge (1, 6): No loop is formed. Hence include it.



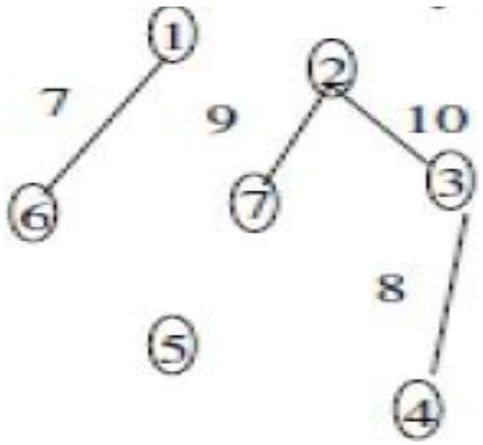
Take edge (4,3): No loop is formed. Hence include it.



Take edge (2,7): No loop is formed. Hence include it.

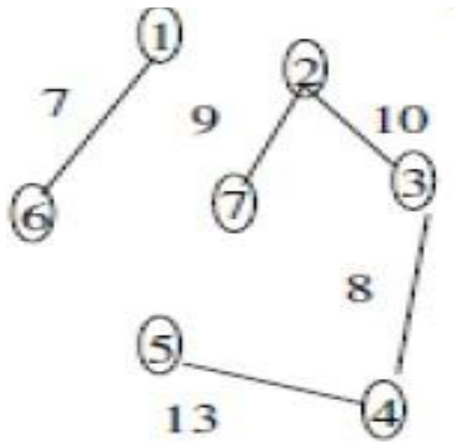


Take edge (2,3): No loop is formed. Hence include it.



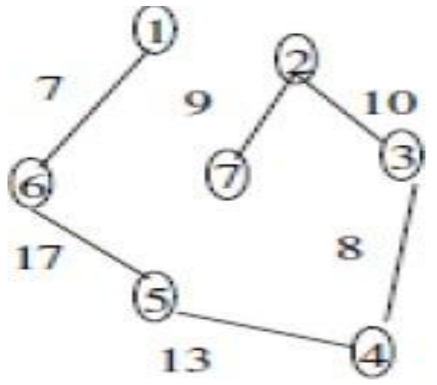
Take edge (7,4): Inclusion of this edge causes a loop. Hence discard it.

Take edge (4,5): No loop is formed. Hence include it.



Take edge (5,7): Inclusion of this edge causes a loop. Hence discard it.

Take edge (5,6): No loop is formed. Hence include it.

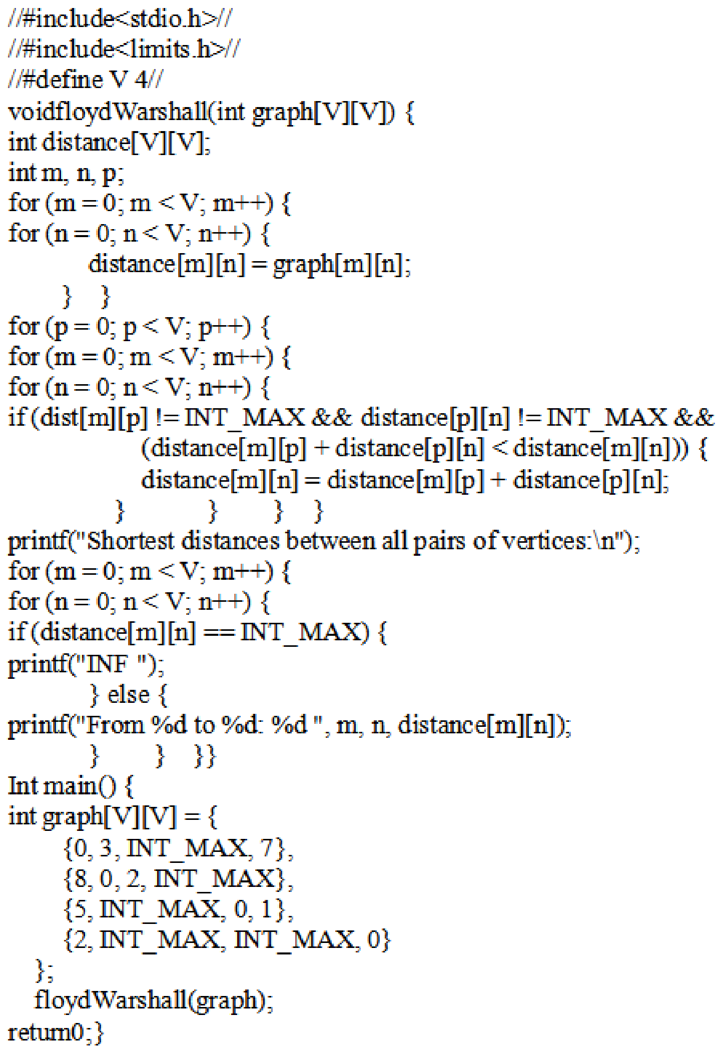


Total edges included = (n– 1). So the algorithm terminates here.

Floyd Warshall's algorithm:

- Floyd Warshall's algorithm is applied to discover shortest paths between every pairs of nodes in a given weighted graph.
- The complexity of this algorithm is O(n^3), where n = the number of nodes in a weighted graph.
- This algorithm is also known as Floyd's algorithm, Roy-Floyd algorithm, or Roy Warshall algorithm.
- This algorithm uses dynamic programming paradigm to discover shortest paths.

Algorithm in C:



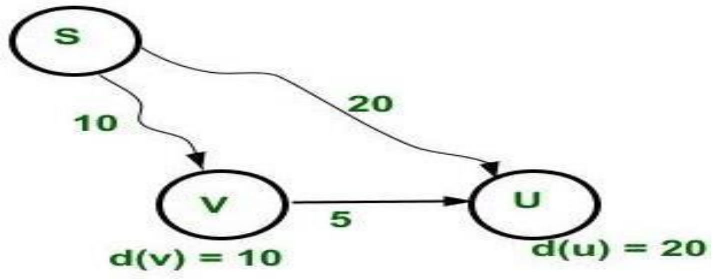
```
//#include<stdio.h>//
//#include<limits.h>//
//#define V 4//
voidfloydWarshall(int graph[V][V]) {
int distance[V][V];
int m, n, p;
for (m = 0; m < V; m++) {
for (n = 0; n < V; n++) {
            distance[m][n] = graph[m][n];
        }   }
for (p = 0; p < V; p++) {
for (m = 0; m < V; m++) {
for (n = 0; n < V; n++) {
if (dist[m][p] != INT_MAX && distance[p][n] != INT_MAX &&
                    (distance[m][p] + distance[p][n] < distance[m][n])) {
                    distance[m][n] = distance[m][p] + distance[p][n];
                }           }       }   }
printf("Shortest distances between all pairs of vertices:\n");
for (m = 0; m < V; m++) {
for (n = 0; n < V; n++) {
if (distance[m][n] == INT_MAX) {
printf("INF ");
            } else {
printf("From %d to %d: %d ", m, n, distance[m][n]);
            }       }   }}
Int main() {
int graph[V][V] = {
        {0, 3, INT_MAX, 7},
        {8, 0, 2, INT_MAX},
        {5, INT_MAX, 0, 1},
        {2, INT_MAX, INT_MAX, 0}
    };
    floydWarshall(graph);
return0;}
```

Dijkstra's Algorithm:

- This algorithm is developed to discover shortest path from one node to other nodes in a given graph. It can be used on directed graphs as well as undirected graphs.
- Because it calculates the shortest path from a source node to all other nodes in the given graph, this approach is also known as the single-source shortest path algorithm. This algorithm produces the shortest path spanning tree.
- The algorithm execution starts from source node. Graph G is the input to the algorithm. **Shortest path spanning tree** is the output of this algorithm.

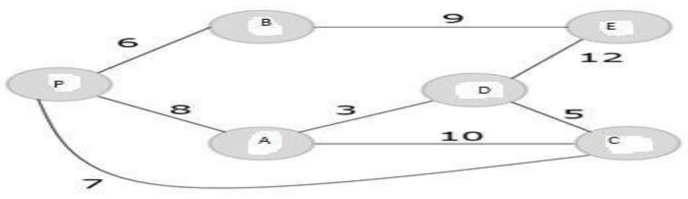
Dijkstra's Algorithm Steps:

1. Declare two vectors –
   - *Distance*[] for maintaining distances from the source node to all other nodes in the graph
   - *Visited*[] for maintaining the visited nodes.
2. Intialise distance[S] = ‘0’ and Distance[v] = ∞, here v refers all the other nodes in the given graph.
3. Add source node S to *Visited*[] Vector. Then find the adjacent nodes of S and update *Distance*[] vector for adjacent nodes of S.
4. Select a node from (*Vertex*[] – *Visited*[]) that has shortest distance. Add this node to Visited. Find all the adjacent nodes to this node and perform relaxation using each of the adjacent nodes. Update the *Distance*[] vector accordingly.
5. Repeat 4th step until all the nodes are in *Visited*[] vector and the shortest path spanning tree is formed.
6. **Relaxation Property:**
   Modify distances for all the adjacent nodes of node, say u, using Relaxation rule discussed below using diagram. For changing distance values, iteratively run through all adjacent nodes of u. For each adjacent nodes, say v, if sum of the weight of edge (u, v) and the distance of u from source is smaller than v's distance form the source, then change the length of v from the source.



- Let dist(U) = distance of node U from source node S
- Cost(V, U) = distance from node V to node U.
- If dist(U) > dist(V) + Cost(V, U) then dist(U) = dist(V) + Cost(V, U). This is called Relaxation.
- As an example, 20>10+5, dist(U) = 10+5 = 15

Example: Tracing the algorithm using the following graph



Step 1: Initialization

Initialize distances to all nodes from source node, P, to ∞. Make P to P as 0.

Node	P	B	E	D	A	C
Distance	0	∞	∞	∞	∞	∞

Let the source node P be visited. Add P to visited Vector. visited = {P}

Step 2:

The Node P has 3 adjacent nodes, B, A, C, with different distances. Hence The distance of S to A, S to D and S to E will be changed in the distance vector.

$P \rightarrow B = 6$
$P \rightarrow A = 8$
$P \rightarrow C = 7$

Node	P	B	E	D	A	C
Distance	0	6	∞	∞	8	7

Step 3: Repeat Step

- Select a node from (Vertex[] – Visited[]) that has shortest distance. Add this node to Visited. Find all the adjacent nodes to this node and perform relaxation for each of the adjacent nodes. Update the distance vector accordingly.

  The vertex having minimum distance among all the nodes in (Vertex[] – Visited[]) is B. Hence, B is added to Visited. The adjacent node(s) to B is E. Hence perform Relaxation and update the distance vector as follows.

Node	P	B	E	D	A	C
Distance	0	6	15	∞	8	7

  Now, Visited = {P, B}

  Exercise: Perform repeat step and discover shortest paths to all the nodes from the source node.

Requirements

1. Dijkstra's Algorithm is guaranteed to work properly if graphs have only **positive** weights. The reason for this is that the edge weights are to be added to discover shortest path.
2. The algorithm may not perform properly if any negative weight exists in graph. The movement a node is denoted as "visited", then the present path for that node can be marked as a shortest path. Existence of negative weights may alter this path if sum of weights gets reduced later.

25. **Applications of Graph Data Structure:**

Graph data structure has a variety of applications. Some of the most popular applications are:

- Graphs have been used for representing flow of computation in programs of software systems.
- Google Maps has been using for constructing digital transportation systems. The shortest path discovery between pair of vertices is being used by the Google.
- Linkedin and Facebook have been using for social networks analysis.
- Operating Systems use a Resource Allocation Graph where every process and resource acts as a node. While we draw edges from resources to the allocated process.
- Used in the world wide web where the web pages represent the nodes.
- Blockchains also use graphs. The nodes store many transactions while the edges connect subsequent blocks.
- Used in modelling data.
- Some other applications of graphs include:
  - Knowledge graphs
  - Biological networks
  - Neural networks
  - Product recommendation graphs

These are some of the fundamental terms and concepts used to describe and work with graphs in the field of data structures and algorithms. Graphs are incredibly versatile and find applications in various domains, including computer networks, social networks, recommendation systems, and route planning, among others.