Sagar Chhabriya

Sukkur iba university kandhkot campus

**Data Structure**

**Data Structure**

**Data Structure:** A data structure is a way to store and organize data in order to facilitate access and modifications. This is a way for programmers to develop their brains logically and apply the logic to solve problems related to data.

**Objective of Data Structuring:** Mathematical and Logical model for specific organizations.

**What is Data?**

Unprocessed information is called data.

**What is information?**

Unprocessed data is called information.

**What is Data Management?**

To organize and secure the data.

**Ex:** Data Science, AI, ML, DL, Data Mining, DBMS etc.

**What are the Shapes of Data?**

Image, audio, video, Text, bio-metric and so on.

When we talk about the Data Base Management the common terms we hear about DBM are entity, attribute, and records. Entity: An entity is a collection of attributes also known as fila e. Attributes or Fields: a unit information of entity, a single and very small piece of entity that can be accessed independently is called field. Record or Line: which contains the information.

**Entity VS Entity Set**

Entity Set: The File that doesn’t contain records. Like a new table in the Data Base Schema with not a single records (i.e., lines of information.)

**What is Structure?**

In data structures, a structure is a logical model that helps organize and manipulate data effectively.

**Why do we need data? What is the need to organize the data?**

In commonly we do not need the data to be organized. This need arises in large organizations, which have a huge amount of data, for storing, retrieving or manipulating.

**Algorithm:** An algorithm is the step-by-step solution of a problem in algorithm’s notations.

**Pseudo code:** Solution in steps using the language code.

**Data Items:**

The data items are containers which contains the data also known as “Variables”. There are two types of the Data Items.

1. **Elementary Data Items**
2. **Group Data Items**

**Elementary Data Items:** These are those items which can’t be sub-divided and they come with fixed length. They occupies a fixed amount of storage space, it means they occupy a constant number of bytes in memory, regardless of the actual value stored.

**Example:** CMS No, Roll No, Name.

These are the basic data types provided by the programming language, such as integers, floating-point numbers, characters, Booleans, etc. They represent individual values and cannot be further divided into smaller components.

**Group (or Aggregate) Data Items:** These are those items which can be sub-divided as per requirement. These refer to the data items those can occupy a variable amount of storage.

**Example:** Address → Street in a city, a city in a district, and a district in a province and so on.

These are data items formed by combining multiple elementary or aggregate data items. Examples include arrays, structures, classes, and objects. Aggregate data items can hold multiple values and have a more complex internal structure, often consisting of multiple elementary data items or a combination of elementary and aggregate data items. Aggregate data items like strings, arrays, and linked list often have dynamic lengths because they can grow or shrink during program execution as elements are added or removed.

**Program:** Collection of commands according to ideas.

**Algorithm:** Step-by-Step solution.

Algorithm

Programmer

Data Analyst

Program

Collection of Commands

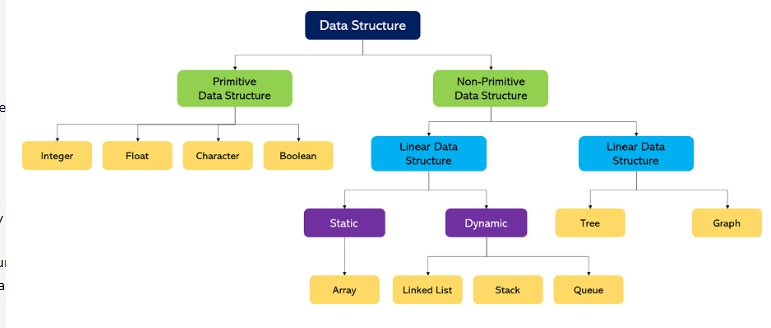
Solution

There are two terms that are mostly used while implementing the Logic (i.e., while writing the Program).

1. **PSN or USN: Programmer/User Supplied Names.**
2. **Reserved/Keywords: Predefined by Language.**

**Two Types of Data Items (i.e., variable)**

1. **Fixed Length Storage: Final, Const**
2. **Variable Length Storage: variable, dynamic initialization, reassign able**

**Data Structure Hierarchy**

* **Two Types of Data structure**

**Linear Non-Linear**

Sequential Access/Linear Models Non-Sequential Access/Non-Linear Models

Array, List, Linked List(s), Queue, etc. Tree, Graph

* **Linear Models**
* **Array:** Linear accessing
* **Linked List:** Linear data Structure in which elements are accessed according to pointer address, each element have address of other element. Pointer is nothing but memory address.
* **Stack:** It is a linear data structure in which elements are inserted and deleted from one end/one way. It is also known as First-in-First-out (FIFO/LILO). It is worth noting that the both deletion and insertion is performed at top.

**3**

**2**

**1**

**0**

**Towel 1**

**Towel 2**

**Towel 3**

**Towel 4**

**← Top**

**Push →**

**Pop →**

**Push:** Pushing something at the top of the stack.

**Pop:** removing something from the top of stack.

* **Queue:** It is a linear data structure in which elements are inserted from one side (Rear) and deleted from another side (LIFO/FILO). It is a two way structure.

Enqueue (IN Q | Offer) adding at the end of the line.

Dequeue (D Q | Poll) getting from the first of the line.

**Towel 4**

**Towel 3**

**Towel 2**

**Towel 1**

**0**

**1**

**2**

**3**

**Offer →**

**Poll →**

* **Non-Linear Models**
* **Tree:** It is a non-linear data structure in which elements are represented in hierarchical nature.

**Introduction to Binary Tree & its variations**

**Root**

**Internal**

**Leaves**

**Non-Leaves**

1. Non-cyclic
2. Every node has at most two children: 0,1, or 2(Bi-nary)
3. If a node has three children than it is known as trinary trees.
4. Binary tree: At most 2 children.

Full/Strict Binary tree: either 0 or 2 children

ACBT: Almost complete binary tree:

1. First child is left. 2. Every level is completely filled. Levels: Root, internal…

This is not ACBT because it has the right children first

This is ACBT because it has the left children first

**Complete/Perfect Binary tree:** No holes: Every internal node has 2 children.

**Heap Tree**

Insertion, Deletion, heap sort, Heapify, Array representation of Heap, Priority queue.

1. Structured property: It must be ACBT
2. Max Heap/Min Heap

Max Heap (parent>child) →root element is always maximum element

Min Heap (parent<child) →root element is always minimum element

**Heap Tree Construction**

Insert key one by one Heapify method: O(n)

In the given order.

O (nlogn)

Insertion: one by one key insertion method

1. Insert key one by one to insert a key into empty heap takes O (1) time.
2. To insert a key into the already constructed heap in worst case log(n) comparison

And log (n) swapping.

1. Total n element so (nlogn) time.
2. Deletion: we delete the Root element, than place the right extreme index at the place of root element (right extreme place: the element at the last on the right side, that is also known as right most extreme element, see video 3.10 max heap)

* **Graph:** It is a non-linear data structure in which elements are connected to each other.
* **Algorithm Complexity**

Performance or efficiency of algorithm is known as complexity of algorithms.

**Analysis of Algorithm**

Comparing Algorithms in terms of:

-Execution time

-Memory Usage

-Developer Efforts (number of statements)

**Types of Algorithm Analysis**

Worst Case: Algorithm takes a long time.

Best Case: Algorithm takes the least time.

Average Case: Run algorithms many times, using many different inputs.

**Asymptotic Notations**

Representation of the complexity of algorithms.

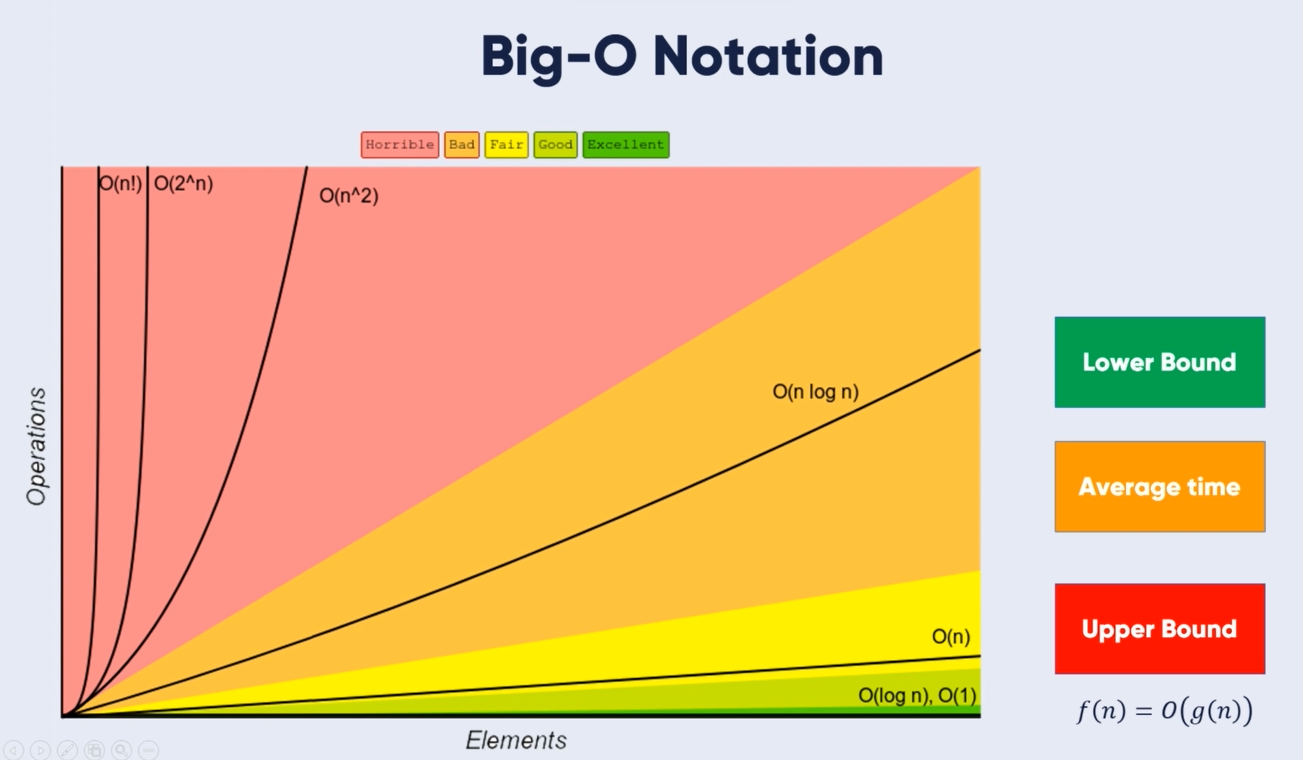
1. Big O Natation→ O( )
2. Big Omega Notation → Ω( )
3. Theta Notation → θ( )
4. Small O Notation → o( )
5. Small Omega Notation → ω( )

* **Big O Notation (Upper Bound)**

It is a mathematical notation that describes the limiting behavior of a function. In computer science the Big O notation is used to classify the algorithm according to how there runtime or space requirements grow as the input size grows. Big O notation characterize functions as there growth rate. Different function with the same growth rate may be representing using the same O notation. The letter ‘O’ used because the growth rate of a function is also referred to us as an order of a function. *So the ‘O’ is the order of a function*.

**Lower Bound Average Time Upper Bound**

**One second Ten Second**

We will not focus on the Lower Bound (best case), instead we’ll focus on the Upper bound to make it best case. And the upper bound is denoted by ‘O’. At larger values of n the upper bound of F (n) is g (n)

* **Omega – Ω Notation (Lower Bound)**

Ω (g (n)): gives the tighter lower bound of the given algorithm and we represent it as

f (n) = Ω (g (n)) The larger values of n, the tighter lower bound of f (n)

* **Theta Notation Θ (Between Lower and Upper)**

This notation decides whether the upper and lower bound of a given function (algorithm)



**Applications of Data Structure**

1: Data Structures help in the organization of data in a computer's memory.

2: Data Structures also help in representing the information in databases.

3: Data Structures allows the implementation of algorithms to search through data (For example, search engine).

4: We can use the Data Structures to implement the algorithms to manipulate data (For example, word processors).

5: We can also implement the algorithms to analyze data using Data Structures (For example, data miners).

6: Data Structures support algorithms to generate the data (For example, a random number generator).

7: Data Structures also support algorithms to compress and decompress the data (For example, a zip utility).

8: We can also use Data Structures to implement algorithms to encrypt and decrypt the data (For example, a security system).

9: With the help of Data Structures, we can build software that can manage files and directories (For example, a file manager).

10: We can also develop software that can render graphics using Data Structures. (For example, a web browser or 3D rendering software).

**Algorithm**

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

**Characteristics of Algorithm**

1: Input: An algorithm has some input values. We can pass 0 or some input value to an algorithm.

2: Output: We will get 1 or more output at the end of an algorithm.

3: Unambiguity: An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.

4: Finiteness: An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.

5: Effectiveness: An algorithm should be effective as each instruction in an algorithm affects the overall process.

6: Language independent: An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

**Data Flow**

1: Problem: A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.

2: Algorithm: An algorithm will be designed for a problem which is a step by step procedure.

3: Input: After designing an algorithm, the required and the desired inputs are provided to the algorithm.

4: Processing unit: The input will be given to the processing unit, and the processing unit will produce the desired output.

5: Output: The output is the outcome or the result of the program.

**Need of Algorithm**

1: Scalability: It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small steps to easily analyze the problem.

2: Performance: The real world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

**Algorithm Notations**

Those notations by which we are able to write algorithms in human-readable form. Also known as Algo-keywords.

Example: step1, stpe2, step3, step-n, write, display, read, get, exit, repeat for, repeat while, end-of-loop, end-of-if-structure and so on.

A real life algorithm to make lemon juice:

Step 1: First, we will cut the lemon into half.

Step 2: Squeeze the lemon as much you can and take out its juice in a container.

Step 3: Add two tablespoon sugar in it.

Step 4: Stir the container until the sugar gets dissolved.

Step 5: When sugar gets dissolved, add some water and ice in it.

Step 6: Store the juice in a fridge for 5 to minutes.

Step 7: Now, it's ready to drink.

Note: Here we assumed that we already have the pots and lemons, or any necessary stuff.

The above real world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Now we will look at an example of an algorithm in programming.

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: get the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.

Step 5: Print the sum

Step 6: Stop

**Factors of Algorithm**

**1: Modularity**: If any problem is given and we can break that problem into small modules or small steps, which is a basic definition of an algorithm, it means that this feature has been perfectly designed for the algorithm.

**2: Correctness**: The correctness of an algorithm is defined as when the given inputs produce the desired output, which means that the algorithm has been designed algorithm. The analysis of an algorithm has been done correctly.

**3: Maintainability**: Here, maintainability means that the algorithm should be designed in a very simple structured way so that when we redefine the algorithm, no major change will be made to the algorithm.

**4: Functionality**: It considers various logical steps to solve the real-world problem.

**5: Robustness**: Robustness means that how an algorithm can clearly define our problem.

**6: User-friendly**: If the algorithm is not user-friendly, then the designer will not be able to explain it to the programmer.

**7: Simplicity**: If the algorithm is simple then it is easy to understand.

**8: Extensibility**: If any other algorithm designer or programmer wants to use your algorithm then it should be extensible.

**Importance of Algorithm**

**1: Theoretical importance**: When any real-world problem is given to us and we break the problem into small modules. To break down the problem, we should know all the theoretical aspects.

**2: Practical importance**: As we know that theory cannot be completed without the practical implementation. So, the importance of algorithm can be considered as both theoretical and practical.

**Issues of Algorithms**

The following are the issues that come while designing an algorithm:

How to design algorithms: As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.

How to analyze algorithm efficiency

* **Algorithms in Algo-Notations**

1. **Traversing Algorithm**

**Traversing Array** Traversing the elements of array according to linear.

**N** is the number of elements.

**D** is the name of array.

**Step 1: SET a := 1**

**Step 2: Repeat While a <= N**

**Step 3: write D[a]**

**Step 4: SET a := a+1**

**[End of Step2 loop]**

**Step 5: Exit**

* **Traversing algorithm Implementation using java:**

**package Lab; //package name**

**import java.util.Arrays; //importing to execute the Arrays class method**

**public class TraversingAlgorithm\_1 {**

**public static void main(String[] args) {**

**char arr[] ={'a','b','c','d'}; //char type array**

**System.out.println("Printing via loop"); //printing array using loop**

**for(int i=0; i<arr.length; i++)**

**System.out.print(arr[i] + " ");**

**System.out.println("Printing via using the Arrays class");**

**System.out.println(Arrays.toString(arr)); //print array using Arrays class method**

**}**

**}**

1. **Find the Maximum Element**

**Max Element** Find the maximum element of array according to linear.

**N** is the number of elements.

**arr** is the name of array.

**Step 1: SET a:=1 AND k:=5 AND max=arr[1]**

**Step 2: Repeat While a<=k**

**Step 3: if(arr[a]>max)**

**Step 4: SET max=arr[a]**

**[End of IF Structure]**

**Step 5: a:=a+1**

**[End of Step2 loop]**

**Step 6: Exit**

* **Find The Maximum Element Implementation Using Java:**

**package Lab; //package name**

**public class MaxElementOfArray\_2 {**

**public static void main(String[] args) {**

**int arr[]={1,2,3,4,5,6,7} , max=arr[0]; //array of type int and max = arr[0]= 1→ max = 1**

**for(int i=0; i<arr.length; i++)**

**if(arr[i]>max) max=arr[i];**

**System.out.println(max); //print the maximum element**

**}**

**}**

1. **Find the Minimum Element**

**Min Element** Find the minimum element of array according to linear.

**N** is the number of elements.

**arr** is the name of array.

**Step 1: SET a:=1 AND k:=5 AND min=arr[1]**

**Step 2: Repeat While a<=k**

**Step 3: if(arr[a]<min)**

**Step 4: SET max=arr[a]**

**[End of IF Structure]**

**Step 5: SET a:=a+1**

**[End of Step2 loop]**

**Step 6: Exit**

* **Find The Minimum Element implementation using java:**

**package Lab; //package name**

**public class MinElementOfArray\_3 {**

**public static void main(String[] args) {**

**int arr[]={1,2,3,4,0,6,7} , min=arr[0];**

**for(int i=0; i<arr.length; i++)**

**if(arr[i]<min) min=arr[i];**

**System.out.println(min);**

**}**

**}**

1. **Insert The Element into Array.**

**Insertion** Insert the element into array at desired location according to linear.

**N** is the number of elements.

**arr** is the name of array.

**Step 1: SET Loc:=2 AND i:=N AND item=’b’**

**Step 2: Repeat While i>Loc**

**Step 3: SET arr[i]=arr[i-1]**

**Step 4: SET i:=i-1**

**Step 5: SET arr[Loc]=item**

**[End of Step2 loop]**

**Step 6: Exit**

* **Insert the Element in the array using java:**

**package Lab;**

**import java.util.Arrays; //import the Arrays class**

**public class Insertion\_3 {**

**public static void main(String[] args) {**

**char arr[]={'a','c','d','e',' '}; //array of character type**

**char item ='b'; //assign ‘b’ to the item variable**

**int loc=1; //assign 1 to the location**

**for(int i=arr.length-1; i>loc; i--)**

**arr[i]=arr[i-1];**

**arr[loc]=item; //insert the item on desired location**

**System.out.println(Arrays.toString(arr)); //print the array using Arrays class method**

**}**

**}**

1. **Delete The Element fromArray.**

**Deletion** Delete the element from array at desired location according to linear.

**N** is the number of elements.

**arr** is the name of array.****

**Step 1: SET i:=2 AND item=’b’**

**Step 2: Repeat While i<N**

**Step 3: SET arr[i]=arr[i+1]**

**Step 4: SET i:=i+1**

**Step 5: [End of Step2 Loop]**

**Step 6: SET arr[N]=null**

**[End of Step2 loop]**

**Step 6: Exit**

* **Delete The Element from the Array**

**package Lab;**

**import java.util.Arrays;**

**public class Deletion\_5 {**

**public static void main(String[] args) {**

**char arr[]={'a','b','c','d','e'};**

**char item ='b';**

**int j = 1;**

**System.out.println(Arrays.toString(arr));**

**while(j<arr.length-1)**

**{**

**arr[j]=arr[j+1];**

**j++;**

**}**

**arr[arr.length-1]=' ';**

**System.out.println(Arrays.toString(arr));**

**}**

**}**

1. **Search The Element from Array Using Linear Search.**

**Linear Search** Search the element from the array and get the location according to linear.

**N** is the number of elements. Item is the ‘item’ to find.

**arr** is the name of array.

****

**Step 1: SET i:=1 AND Loc:=0**

**Step 2: Repeat While i<N**

**Step 3: If item equals arr[i]**

**Step 4: SET Loc:=i AND WRITE Loc AND Goto Step5 [End of If Structure]**

**Step 5: SET i := i+1**

**[End of Step2 Loop]**

**Step 6: Exit**

* **Linear Search Implementation using java:**

**package Lab;**

**public class LinearSearch\_6 {**

**public static void main(String[] args) {**

**int arr[]={1,2,3,4,5,6,};**

**int item =3;**

**// using for loop**

**for(int i=0; i<arr.length; i++)**

**{**

**if(arr[i]==item)**

**System.out.println("location using for loop : "+i);**

**}**

**// using while loop**

**int j =0;**

**int search = 3;**

**while(j<arr.length)**

**{**

**if(arr[j]==search)**

**System.out.println("Location using while loop : "+j);**

**j++;**

**}**

**int k=0;**

**int find = 3;**

**//using do while loop**

**do{**

**if(arr[k]==find)**

**System.out.println("Location using do while loop : "+k);**

**k++;**

**}**

**while(k<arr.length);**

**}**

**}**

1. **Search The Element from Array Using Binary Search.**

**Binary Search** Search the element from the array and get the location according to non-linear.

**N** is the number of elements. Item is the ‘item’ to find.

**arr** is the name of array.

**Step 1: SET beg:= 0 AND end:=N AND mid = (beg + end)/2 AND item:=value**

**Step 2: Repeat While(beg <= end AND item # arr[mid])**

**Step 3: if(item>arr[mid]) beg:= mid + 1**

**Step 4: else if(item<arr[mid]) end:= mid - 1**

**Step 5: SET mid := (beg + end)/2**

**[End of if Structure]**

**[End of step2 loop]**

**Step 6: if(arr[mid] =item)**

**Step 7: Write arr[mid]**

**[End of if Structure]**

**Step 8: Else Write “Item Not Found”**

**Step 9: Exit**

* **Binary Search Implementation using java:**

**package Lab;**

**public class BinarySearch {**

**public static void main(String[] args) {**

**int arr[]={1,2,3,4,5,6,7,8,9};**

**int beg=0, end=arr.length-1, mid = (beg+end)/2,item =10;**

**while(beg<=end && arr[mid] !=item)**

**{**

**if(item>arr[mid]) beg=mid+1;**

**else if (item<arr[mid]) end=mid-1;**

**mid = (beg+end)/2;**

**}**

**if(item==arr[mid])**

**System.out.println("Item foudn at indes"+mid);**

**else**

**System.out.println("Item not found");**

**}**

**}**

* **Sorting**

***Process of arranging the data.***

1. **Sort The Elements of Array in non-decreasing order Using Bubble Sort.**

**Bubble Sort** Sort the elements of the array in non-decreasing order.

**N** is the number of elements.

**arr** is the name of array.

**Step 1: SET K:= 0 AND J:=0**

**Step 2: Repeat While( K<N )**

**Step 3: SET J :=0**

**Step 4: Repeat While( J < N – K )**

**Step 5: If (arr[j] > arr[j+1]) than [Interchange]**

**[End of if Structure]**

**Step 6: SET J:= J+1**

**[End of step4 loop]**

**Step 7: SET K:=K+1**

**[End of Step2 loop]**

**Step 8: Exit**

**Bubble Sort Implementation using java:**

**package Lab;**

**import java.util.Arrays;**

**public class BubbleSort {**

**static void sortB(int[] arr)**

**{**

**int k=0;**

**int temp=0;**

**int j=0;**

**while(k<arr.length-1)**

**{**

**j=0;**

**while(j<(arr.length-1)-k)**

**{**

**if(arr[j]>arr[j+1])**

**{**

**temp = arr[j];**

**arr[j]=arr[j+1];**

**arr[j+1]=temp;**

**}**

**j++;**

**}**

**k++;**

**}**

**System.out.println(Arrays.toString(arr));**

**}**

**public static void main(String[] args) {**

**int[] arr={2,5,6,4,3};**

**sortB(arr);**

**}**

**}**

**// output: [2, 3, 4, 5, 6]**

**Selection Sort Sort the elements according to linear,=**

**Let the name of array be arr,**

**Let the size of array be n.**

**Step1: Set beg:= 1 AND End:= n**

**Step2: Repeat While beg < n**

**Step3: Set j:= beg + 1**

**Step4: Repeat While j<n**

**Step5: if arr[beg] > arr[j] than**

**[Interchange]**

**[End of if structure]**

**Step6: Set j:= j + 1**

**[End of step4 loop]**

**Step7: Set beg:= beg+1**

**[End of step2 loop]**

**Step8: Exit**

**Insertion Sort Sort the elements according to linear,**

**Let the array name be arr,**

**Let the size of array be n.**

**Step1: Set k:=2**

**Step2: Repeat while k < n**

**Step3: Set item:= arr[k] AND j:=k-1**

**Step4: Repeat while j>0 and arr[j]> item**

**Set arr[j+1] := arr[j]**

**Set j:= j – 1**

**[End of step4 loop]**

**Step5: Set arr[j+1] := item**

**Step6: Set k:= k + 1**

**Step7: Exit**

**package Lab;**

**import java.util.Arrays;**

**public class InsertionSort {**

**static int[] sort(int[] arr)**

**{**

**for(int i=1; i<arr.length; i++)**

**{**

**int j=i-1;**

**int item = arr[i];**

**while(j>=0 && arr[j]>item)**

**{**

**arr[j+1]=arr[j];**

**j--;**

**}**

**arr[ j+1 ] = item;**

**}**

**return arr;**

**}**

**public static void main(String[] args) {**

**int[] arr={7,6,8,3,2,10,5,9,4,1};**

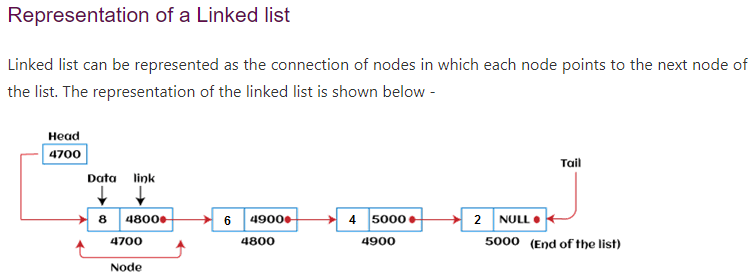
**System.out.println(Arrays.toString(sort(arr)));**

**}**

**}**

* **Linked List**

Linked list is a linear data structure that includes the series of connected nodes in which each node has address of another node and they are accessed by pointer addresses. Linked list can be defined as the nodes that are randomly stored in the memory. A node in linked list contains two parts i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link. The first node is also known as head and the last node tail. One major difference between the array and Linked list is that arrays are static, means they can’t grow at runtime but in contrast the linked lists are dynamic means they can grow at the run time.



Fundamental methods of Linked list for manipulation:

1. addFirst( )
2. addLast( )
3. getFirst( )
4. getLast( )
5. removeFirst( )
6. removeLast( )

* **Singly Linked List**

A linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence. In a singly linked list, each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.

**Inserting an element at the head of the Singly linked list**

addFirst( e )

Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after a new node is created and linked to the existing head; (c) after reassignment of the head reference to the newest node

**Algorithm addFirst(e):**

**newest = Node(e) {create new node instance storing reference to element e}**

**newest.next = head {set new node’s next to reference the old head node}**

**head = newest {set variable head to reference the new node}**

**size = size+1 {increment the node count}**

Inserting a new element at the beginning of a singly linked list. Note that we set the next pointer of the new node before we reassign variable head to it. If the list were initially empty (i.e., head is null), then a natural consequence is that the new node has its next reference set to null.

**Inserting an element at the tail of the Singly linked list**

addLast( e )

Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after a new node is created and linked to the existing head; (c) after reassignment of the head reference to the newest node.

**Algorithm addLast(e):**

**newest = Node(e) {create new node instance storing reference to element e}**

**newest.next = null {set new node’s next to reference the null object}**

**tail.next = newest {make old tail node point to new node}**

**tail = newest {set variable tail to reference the new node}**

**size = size+1 {increment the node count}**

Inserting a new node at the end of a singly linked list. Note that we set the next pointer for the old tail node before we make variable tail point to the new node. This code would need to be adjusted for inserting onto an empty list, since there would not be an existing tail node.

**Removing an element from a Singly linked list**

removeFirst( )

Removal of an element at the head of a singly linked list: (a) before the removal; (b) after “linking out” the old head; (c) final configuration.

**Algorithm removeFirst( ):**

**if head == null then**

**the list is empty.**

**head = head.next {make head point to next node (or null)}**

**size = size−1 {decrement the node count}**

Unfortunately, we cannot easily delete the last node of a singly linked list. Even if we maintain a tail reference directly to the last node of the list, we must be able to access the node before the last node in order to remove the last node. But we cannot reach the node before the tail by following nex t links from the tail. The only way to access this node is to start from the head of the list and search all the way through the list. But such a sequence of link-hopping operations could take a long time. If we want to support such an operation efficiently, we will need to make our list **doubly linked**.In this section, we present a complete implementation of a SinglyLinkedList class, supporting the following methods:

**size( ):**  Returns the number of elements in the list.

**isEmpty( ):** Returns true if the list is empty, and false otherwise.

**first( ):** Returns (but does not remove) the first element in the list.

**last( ):** Returns (but does not remove) the last element in the list.

**addFirst(e):** Adds a new element to the front of the list.

**addLast(e):** Adds a new element to the end of the list.

**removeFirst( ):** Removes and returns the first element of the list.

**Singly Linked List Class implementation in java using the generics.**

public class SinglyLinkedList<E> {

private static class Node<E> {

private E element; **//generic type variable; reference to the element stored at this node**

Node<E> next; **//reference to the subsequent node in the list**

public Node(E e, Node<E> n ){

element = e;

next = n;

}

public E getElement()

{return element;}

public Node<E> getNext()

{return next;}

public void setNext(Node<E> n)

{next = n; }

**//instance variables of the SinglyLinkedList**

private Node<E> head = null; **//head node of the list (or null if empty)**

private Node<E> tail = null; **//last node of the list (or null if empty)**

private int size = 0; **//number of nodes in the list**

// public SinglyLinkedList<E>( )

// {}

//access methods

public int size( )

{return size;}

public boolean isEmpty()

{return size == 0;}

public E first() { **//return (but does not remove) the element**

if(isEmpty()) return null;

return head.getElement();

}

public E last(){ **//return (but does not remove) the last element**

if(isEmpty()) return null;

return tail.getElement();

}

// update methods

public void addFirst(E e){ **//adds element to the front of the list**

head = new Node<>(e, head); **//create and link a new node**

if(size == 0)

tail = head; **//special case: new node becomes tail also**

size++;

}

public void addLast(E e){ **//add element to the end of the list**

Node<E> newest = new Node<>(e, null); **//node will eventually be the tail**

if (isEmpty())

head = newest; **//special case: previously empty list**

else

tail.setNext(newest); **//new node after existing tail**

tail = newest; **//new node becomes the tail**

size++;

}

public E removeFirst(){ **//removes and returns the first element**

if(isEmpty()) return null; **//nothing to remove**

E answer = head.getElement();

head = head.getNext(); **//will become null if list had only one node**

size--;

if(size == 0)

tail = null; **//special case as list is now empty**

return answer;

}

}

}

If first( ), last( ), or removeFirst( ) are called on a list that is empty, we will simply return a null reference and leave the list unchanged.

**Circularly Linked List**

Linked lists are traditionally viewed as storing a sequence of items in a linear order, from first to last. However, there are many applications in which data can be more naturally viewed as having a cyclic order, with well-defined neighboring relationships, but no fixed beginning or end. For example, many multiplayer games are turn-based, with player A taking a turn, then player B, then player C, and so on, but eventually back to player A again, and player B again, with the pattern repeating. As another example, city buses and subways often run on a continuous loop, making stops in a scheduled order, but with no designated first or last stop per se. We next consider another important example of a cyclic order in the context of computer operating systems.

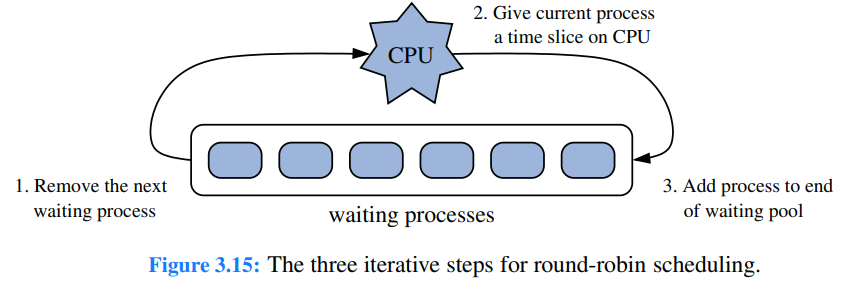
**Round-Robin Scheduling**

One of the most important roles of an operating system is in managing the many processes that are currently active on a computer, including the scheduling of those processes on one or more central processing units (CPUs). In order to support the responsiveness of an arbitrary number of concurrent processes, most operating systems allow processes to effectively share use of the CPUs, using some form of an algorithm known as round-robin scheduling. A process is given a short turn to execute, known as a time slice, but it is interrupted when the slice ends, even if its job is not yet complete. Each active process is given its own time slice, taking turns in a cyclic order. New processes can be added to the system, and processes that complete their work can be removed. A round-robin scheduler could be implemented with a traditional linked list, by repeatedly performing the following steps on linked list L.

1. process p = L.removeFirst( )

2. Give a time slice to process p

3. L.addLast(p)  
Unfortunately, there are drawbacks to the use of a traditional linked list for this purpose. It is unnecessarily inefficient to repeatedly throw away a node from one end of the list, only to create a new node for the same element when reinserting it, not to mention the various updates that are performed to decrement and increment the list’s size and to unlink and relink nodes.



package LinkedList;

public class CircularSinglyLinkedList<E> {

private static class Node<E> {

private E element; //generic type variable; reference to the element stored at this node

Node<E> next; //reference to the subsequent node in the list

public Node(E e, Node<E> n )

{

element = e;

next = n;

}

public E getElement()

{return element;}

public Node<E> getNext()

{return next;}

public void setNext(Node<E> n)

{next = n; }

//instance variables of the SinglyLinkedList

private Node<E> tail = null; //last node of the list (or null if empty)

private int size = 0; //number of nodes in the list

// public CircularSinglyLinkedList<E>( )

// {}

//access methods

public int size( )

{return size;}

public boolean isEmpty()

{return size == 0;}

public E first() { //return (but does not remove) the element

if(isEmpty()) return null;

return tail.getNext().getElement();

}

public E last(){ //return (but does not remove) the last element

if(isEmpty()) return null;

return tail.getElement();

}

// update methods

public void rotate()

{

if(tail !=null)

tail = tail.getNext();

}

public void addFirst(E e){

if(size == 0){

tail = new Node<>(e, null);

tail.setNext(tail);

}

else {

Node<E> newest = new Node<>(e,tail.getNext());

tail.setNext(newest);

}

size++;

}

public void addLast(E e){ //add element to the end of the list

addFirst(e);

tail = tail.getNext();

}

public E removeFirst(){ //removes and returns the first element

if(isEmpty()) return null; //nothing to remove

Node<E> head = tail.getNext();

if(head == tail ) tail = null;

else tail.setNext(head.getNext());

size--;

return head.getElement();

}

}

}

**Link List [**Traverse the Linked List according to Linear**]**

**Step1: Set ptr:= start**

**Step2: Repeat While ptr # null**

**Step3: write data[ptr]**

**Step4: Set ptr:= Link[ptr]**

**[End of Step2 loop]**

**Step5: Exit**

**Deletion**

**Step1: Set ptr:= Start AND item:=value**

**Step2: Repeat while ptr # null**

**Step3: if data[ptr] = item**

**Delete data[ptr] AND Set ptr:= link [ptr]**

**[End of Step 3 if structure]**

**[End of step loop]**

**Step4: Exit**

**Insertion**

**Step1: Set prt:=start AND item:= value**

**Step2: Repeat while ptr # null**

**Step3: if data[ptr] = empty**

**Set data[ptr]=item AND ptr:= link[ptr]**

**[End of step 3 if structure]**

**[End of Step 2 loop]**

**Step4: Exit**

**Recursion**

The process of solving a problem by reducing it to smaller versions of itself is called recursion. Recursion is a very powerful way to solve certain problems for which the solution would otherwise be very complicated. Recursion is the process of defining something in terms of itself. As it relates to programming, recursion is the attribute that allows a method/function to call itself. A method that calls itself is said to be recursive.

The most common example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 is 1 x 2 x 3, or 6. Here is how a factorial can be computed by use of recursive method:

public class FactorialRecursion {

static int factorial(int n)

{

if(n==0) return 1;

return factorial(n-1)\*n;

}

public static void main(String[] args) {

System.out.println(factorial(4));

}

}

**The output from this program is shown here:**

24

Recursive version of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls. Each new call creates a new copy of these variables, it is possible that the stack could be exhausted. The main advantage to recursive method is that they can be used to create clearer and simpler version of several algorithms than can their iterative relatives. For example, the Quick-Sort sorting algorithm is quite difficult to implement in an iterative way. Here is one more example of recursion. The recursive method printArray( ) prints the first i elements in the array values.

**//Another example that uses recursion**

class RecTest{

static int values[]={1,2,3,4,5,6,7,8,9,10};

static void printArray(int i)

{

if(i==0) return;

printArray(i-1);

System.out.print("arr[ "+(i-1)+" ] = "+values[i-1]+" ,");

}

public static void main(String[] args) {

printArray(10);

}

}

**The output from this program is shown here:**

arr[ 0 ] = 1 ,arr[ 1 ] = 2 ,arr[ 2 ] = 3 ,arr[ 3 ] = 4 ,arr[ 4 ] = 5 ,arr[ 5 ] = 6 ,arr[ 6 ] = 7 ,arr[ 7 ] = 8 ,arr[ 8 ] = 9 ,arr[ 9 ] = 10 ,

**Direct and Indirect Recursion**

A function is called directly recursive if it calls itself. A function that calls another function and eventually results in the original function call is said to be indirectly recursive.

**Infinite Recursion**

A recursive method that does not check for a base case, or that misses the base case, will execute “forever.” This situation is known as infinite recursion.

**Tail Recursion**

Tail recursion occurs when the last action performed by a recursive method is a recursive call.

**Stack overflow**

A recursive method that makes many recursive calls will place many activation records in the program stack. Too many recursive calls can use all the memory available for the program stack, making it full. As a result, the error message “stack overflow” occurs. Infinite recursion or large-size problems are the likely causes of this error.

**Debugging A recursive method**

Does the method have at least one input value?

Does the method contain a statement that tests an input value and leads to different cases?

Did you consider all possible cases?

Does at least one of these cases cause at least one recursive call?

Do these recursive calls involve smaller arguments, smaller tasks, or tasks that get closer to the solution?

If these recursive calls produce or return correct results, will the method produce or return a correct result?

Is at least one of the cases a base case that has no recursive call?

Are there enough base cases? Does each base case produce a result that is correct for that case?

If the method returns a value, does each of the cases return a value?

**Algorithms of Fibonacci Series and Factorial Using Recursion**

1. **Fibonacci Series Algorithm with Recursive approach.**

**Fibonacci Series** Declare a method named fibo with a single integer parameter named n &

define the body of method.

**Step 1: Declare fibo( n )**

**Step 2: if(n<0) return null [End of Step2 if Structure]**

**Step 3: else if (n==1) return 0 [End of Step3 else if Structure]**

**Step 4: else if (n==2) return 1 [End of Step4 else if Structure]**

**Step 5: else return fibo(n-1) + fibo(n-2) [make a recursive call]**

**Step 6: Exit**

class fibonacci{

static int fibo(int n)

{

if(n<0) return 0;

else if(n==1) return 0;

else if(n==2) return 1;

return fibo(n-1) + fibo(n-2);

}

public static void main(String[] args) {

System.out.println("Here is the Fibonacci series of number 8: "+fibo(8));

}

}

Here is the Fibonacci series of number 8: 13

1. **Factorial Algorithm with Recursive approach.**

Factorial Declare a method named fact with a single integer parameter named n &

define the body of method.

**Step 1: Declare fact ( n )**

**Step 2: if (n<0) return null [End of Step2 if Structure]**

**Step 3: else if (n==0) return 1 [End of Step2 if Structure]**

**Step 4: return fact (n-1) \* n [Make a recursive call]**

**Step 5: Exit**

class factorial{

static int fact(int n)

{

if(n<0) return 0;

else if (n==0) return 1;

return fact(n-1)\*n;

}

public static void main(String[] args) {

System.out.println("Here is the factorial of 5 : "+fact(5));

}

}

**The output of this program is shown here**

Here is the factorial of 5 : 120

**Divide and Conquer**

When you divide a problem into two or more smaller but distinct problems, solve each new problem, and then combine their solutions to solve the original problem, the strategy is said to be a divide and conquer algorithm. That is, you divide the problem into pieces and conquer each piece to reach a solution. Although divide and conquer algorithms often are expressed recursively, this is not a requirement. When expressed recursively, a divide and conquer algorithm contains two or more recursive calls. Most of the recursive solutions that you have seen so far do not use the divide and conquer strategy.

**Stack Operations**

**Push( )**

**Step1: if top > maxStack**

**[write overflow] And Exit**

**Step2: Set top:= top + 1**

**Step3: push[top, item]**

**Step4: Got Step1 [while stack has space]**

**Pop( )**

**Step1: if top < 0**

**[write underflow] And Exit**

**Step2: Set top := top -1**

**Step3: push[ top, item]**

**Step4: Goto Step1**

**Polish Notations**

Polish Notation in the data structure is a method of expressing mathematical, logical, and algebraic equations universally. This notation is used by the compiler to evaluate mathematical equations based on their order of operations.

1. Infix: An arithmetic expression in which the operator symbol appears between the operands.
2. Prefix: An arithmetic expression in which the operator symbol appears before the operands. Polish notation is also called prefix notation. Left to Right.
3. Postfix: An arithmetic expressions in which the operator symbol appears after the operands. Reverse polish notation is also known as postfix notation. Right to left.

For Example:

|  |  |  |
| --- | --- | --- |
| Infix | Prefix | Postfix |
| A + B | +AB | AB+ |
| A + B \* C | +A \* BC | ABC\*C |
| ( A + B ) \* C | \*+ABC | AB+C\* |
| ( AX + ( B \* C ) ) | AX BC \* + | +AX \* BC |

The time Complexity of converting a prefix notation to infix notation is O ( n ) where n is the length of the equation.

Given two processes ( Conversion of postfix equation to infix notation and conversion of prefix notation to infix notation), which is easier to implement?  
Answer: As the conversion of prefix notation to infix notation involves reversing the equation, the latter is harder to implement than the postfix to infix process.

Prefix → Polish Notation, or Warsaw notation.

Postfix→ Reverse Polish Notation.

The highest order of precedence ~ (tild i.e., not operator) and ^ (caret, i.e., power) followed by \*, /, +, - , ( ), |

**Infix to Postfix Conversion Algorithm**

Step 1: Add the “)” right bracket at the right side end of the expression and add the “(“ left bracket at the top of the stack.

Step2: Scan infix expression from left to right:

Case “(“ if left bracket is found in infix

Push it at the top of the stack and others remain the same.

Case “X” if any operand (i.e., variable) found in infix

Push it in the postfix and others remain the same.

Case “^, /, \*, +, -“ any operator found in infix

If the top element of the stack is not an operator, then Push it at the top of the stack.

If the infix operator’s precedence is lesser than (<) the operator at the top of the stack, then push the infix operator at the top of the stack.

If the infix operator’s precedence is greater than or equal to the operator which is the top of the stack (>=) then,

Shift the operator to the postfix which is at the top of the stack and push the infix operator at the top of the stack.

Case “)” right bracket found in the infix

Pop the RMSB (Right most significant bracket) and move the operator(s) to the postfix which is/are before the bracket (on the right side).

Step3: Exit

**Merge Sort Algorithm**

**Step1: Sort( arr[], low, high) //define the signature of function/method**

**Step2: If(low < high)**

**Step3: SET mid := (low + high) / 2**

**Sort( arr, low, mid) //make a recursive call for the left sub-array and pass arguments**

**Sort(arr, mid+1, high) //make a recursive call for right sub-array and pass arguments**

**Merge(arr, low, mid, high) //Merge the sorted array and pass the arguments to it.**

**Step4: Exit**

**[End of if structure]**

**[End of method’s body]**

**#Define the Merge method’s body**

**Step1: Merge(arr[], low, mid, high) //define the signature of function/method**

**Step2: SET I := low AND J := mid + 1 AND K := low AND size := arr.size AND newArray[size]**

**Step3: Repeat while I <= mid AND J <= high**

**If (arr[I] < arr[J])**

**SET newArray[K] := arr[I]**

**SET I := I + 1 [End of if structure]**

**Else**

**SET newArray[K] := arr[J]**

**SET J := J + 1 [End of Else structure]**

**SET K := K + 1**

**[End of Step3 Loop Control Structure]**

**Step4: Repeat while I <= mid**

**SET newArray[K] := arr[I]**

**SET I := I + 1**

**SET K := K + 1 [End of Step 4 Loop control structure]**

**Step5: Repeat while J<=high**

**SET newArray[K] := arr[J]**

**SET K := K + 1 [End of Step5 Loop control structure]**

**Step6: Copy all the elements from newArray to arr *OR* assign the reference of newArray to arr → arr := newArray in case of object-oriented programming.**

**Step7: Exit**

**[End of method’s body]**

***Note: Step4 and Step5 commands are written in case the elements are left by the Step3 whenever loop get terminated.***

**package Lab;**

**import java.util.Arrays;**

**public class MergeSort {**

**public void sort(int[] arr, int low, int high)**

**{**

**if(low < high)**

**{**

**int mid = (low + high)/2;**

**sort(arr, low, mid);**

**sort(arr, mid+1, high);**

**merge(arr, low, mid, high);**

**}**

**}**

**public void merge(int arr[], int low, int mid, int high)**

**{**

**int i=low,**

**j = mid + 1,**

**k = low;**

**int newArray[]= new int[arr.length];**

**while(i <= mid && j <=high)**

**{**

**if(arr[i] < arr[j])**

**{**

**newArray[k]=arr[i];**

**i++;**

**}**

**else**

**{**

**newArray[k]=arr[j];**

**j++;**

**}**

**k++;**

**}**

**while( j<=high)**

**newArray[k++]=arr[j++];**

**while( i <=mid)**

**newArray[k++]=arr[i++];**

**for(int a = low; a<=high; a++)**

**{**

**arr[a] = newArray[a];**

**}**

**//instead of for loop → arr = newArray**

**}**

**public static void main(String[] args) {**

**MergeSort obj = new MergeSort();**

**int[] arr={7,6,8,3,2,10,5,9,4,1};**

**obj.sort(arr, 0, arr.length-1);**

**System.out.println(Arrays.toString(arr));**

**}**

**}**

**Quick Sort Algorithm**

* **QuickSort( arr[], left, right)**
* **Step1: if arr.length equals to 0 or 1 return normally**
* **Step2: if left < right**
* **Step3: int mid = partition(arr, left , right)**
* **Step4: sort(arr, left, mid-1) //recursive call**
* **Step5: sort(arr, mid + 1, right) //recursive call**
* **Step6: return normally**

**The partition function**

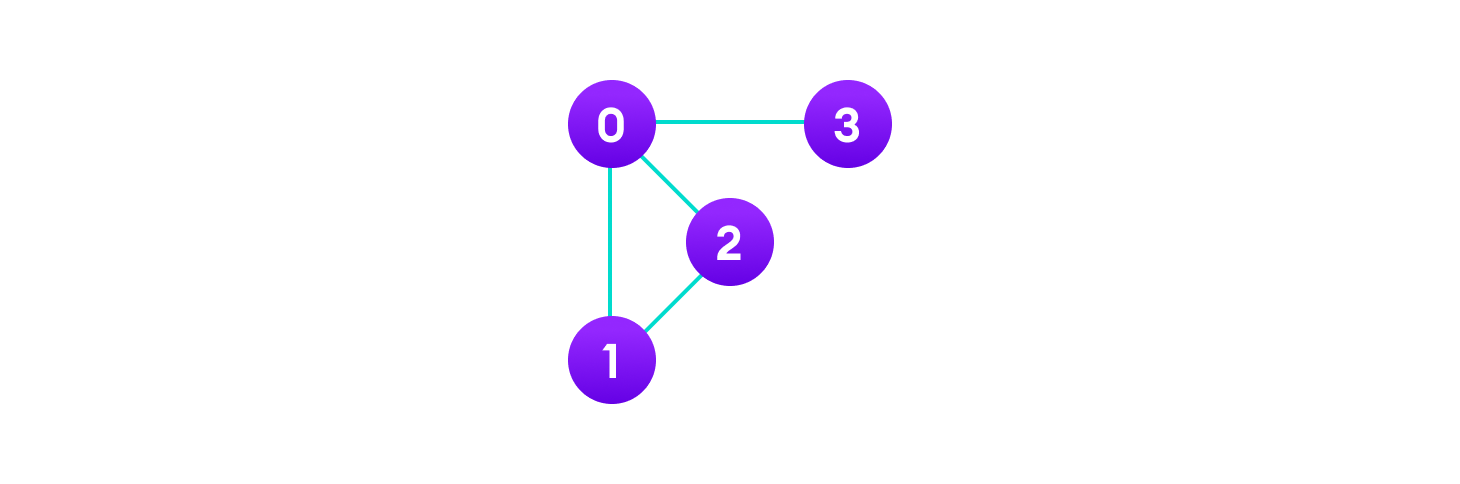
* **partition( arr [], left, right)**
* **step1: SET pivot = arr [left]**
* **step2: SET start = left AND end = right**
* **step3: Repeat while start <= end**
* **step4: Repeat while arr[start] <= pivot**
* **step5: SET start = start + 1 [End of step 4 loop]**
* **step6: Repeat while arr[end] > high**
* **SET high = high - 1 [End of step 6 loop]**
* **step7: If start < end**
* **[Interchange arr[start] with arr[end] ] [End of step 7 If Structure]**
* **[End of Step 3 loop]**
* **step8: [INTERCHANGE the arr[end] with the pivot]**
* **Step 9 : return with value end**

Graph Data Structure

* Collection of nodes that have data and are connected to other nodes by edges.
* Ex: On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, **Note...anything that has data is a node(data containing entity).**
* A purple circles with white icons

  Description automatically generatedEvery **relationship** is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.
* All of facebook is then a collection of these nodes and edges.

This is because facebook uses a graph data structure to store its data.



Graph Terminology

* **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them.
* **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path.
* **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Directed VS Undirected Graph

Directed Graph: The graph with direction.

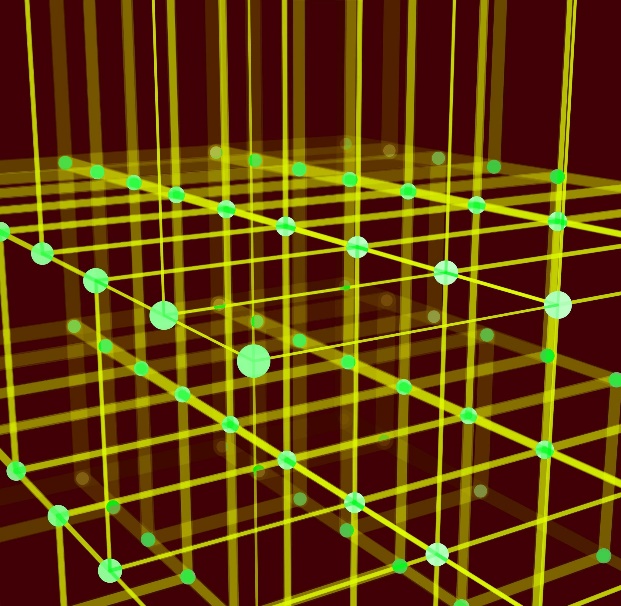
A graph of a graph

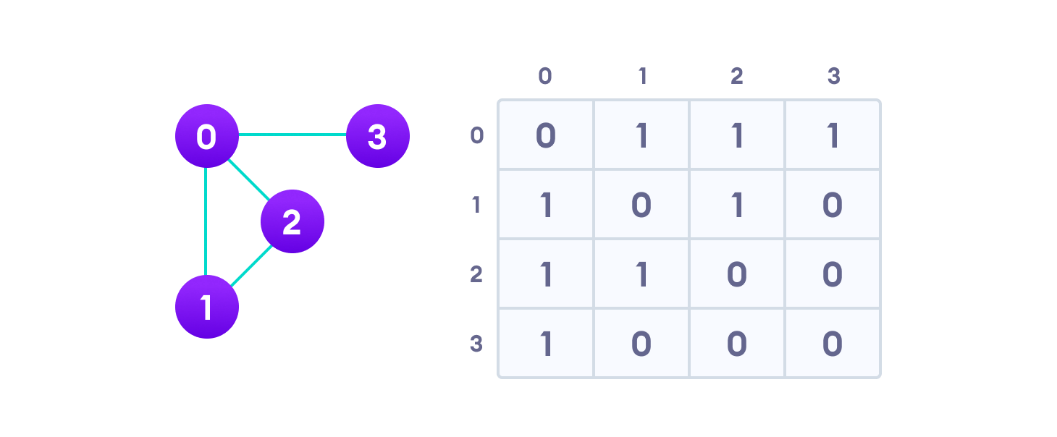
Description automatically generatedA diagram of a graph

Description automatically generatedUndirected: Bidirectional Graph with symmetric properties.Graph Representation

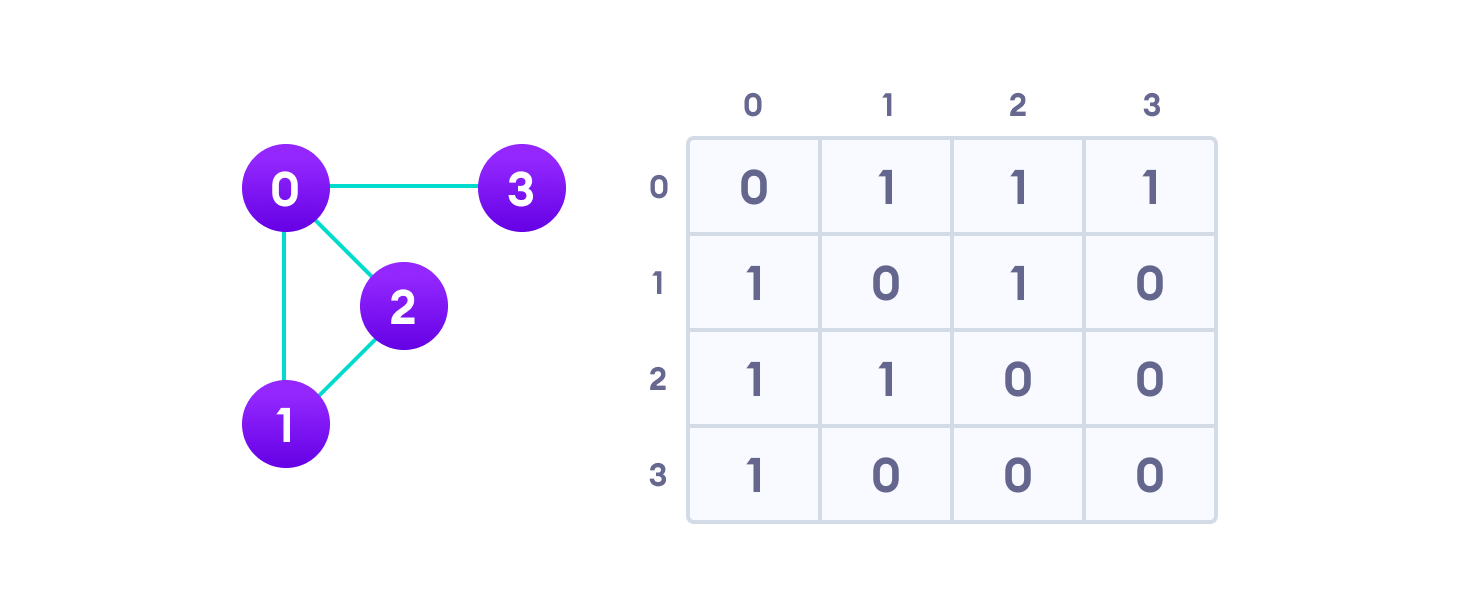
1. Adjacency Matrix:

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.



* Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.
* Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

****

1. **Adjacency List**

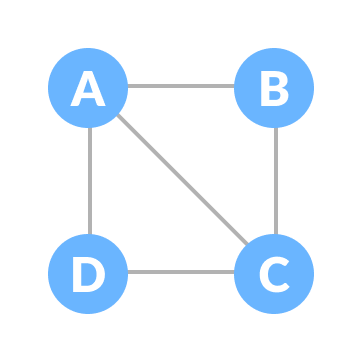
An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

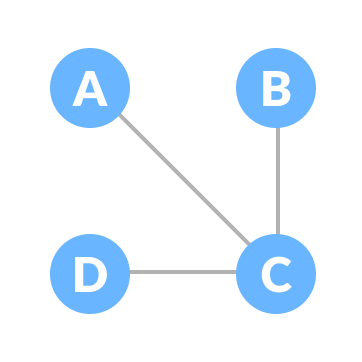
Graph Operations

* The most common graph operations are:
* Check if the element is present in the graph
* Graph Traversal
* Add elements(vertex, edges) to graph
* Finding the path from one vertex to another

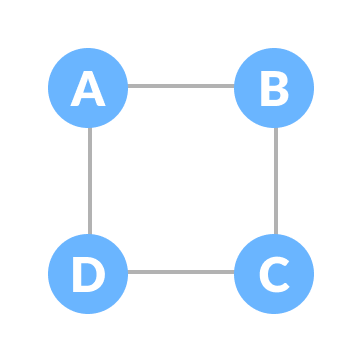
Spanning Tree and Minimum Spanning Tree

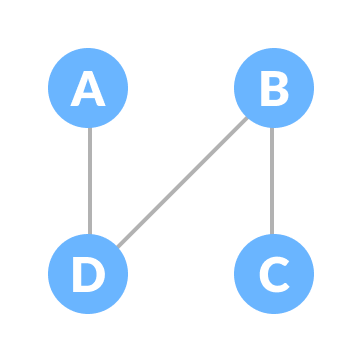
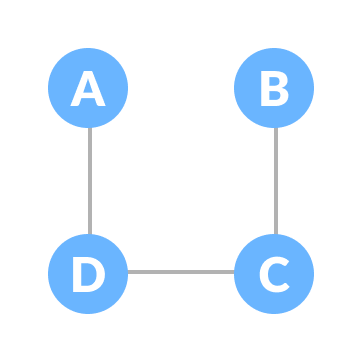
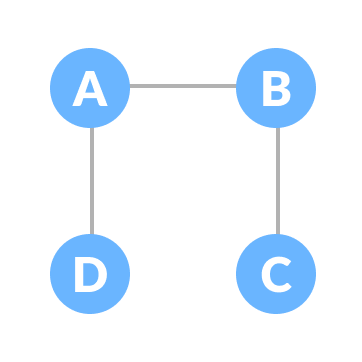
* Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.
* An **undirected graph** is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).

A connected graph is a graph in which there is always a path from a vertex to any other vertex.

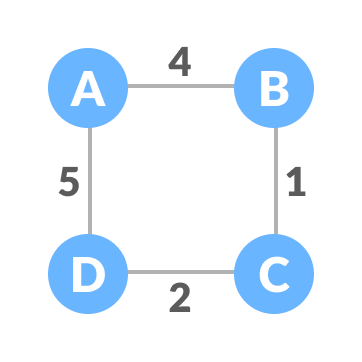
****

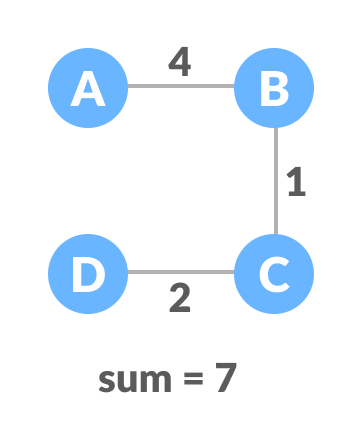
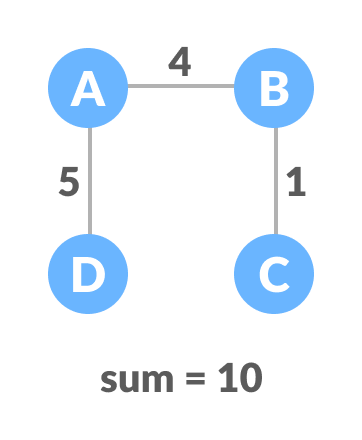
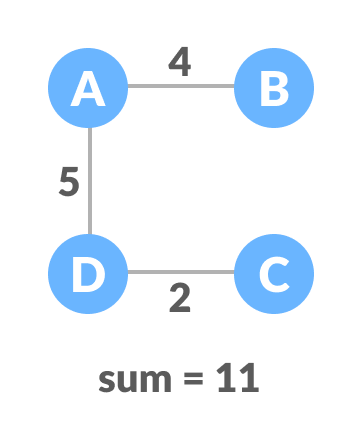
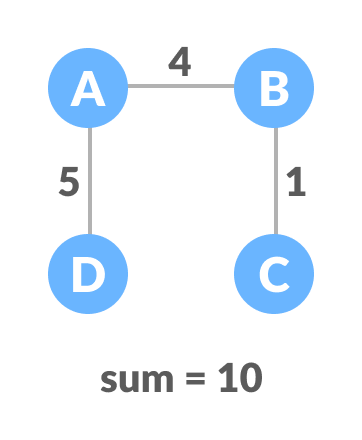
**Spanning tree**

* A spanning tree is a sub-graph of an *undirected connected* graph, which includes all the vertices of the graph with a minimum possible number of edges.
* If a vertex is missed, then it is not a spanning tree.
* *The edges may or may not have weights assigned to them.*
* Example of a Spanning Tree
* Let's understand the spanning tree with examples below:
* Let the original graph be:
* Some of the possible spanning trees that can be created from the above graph are:

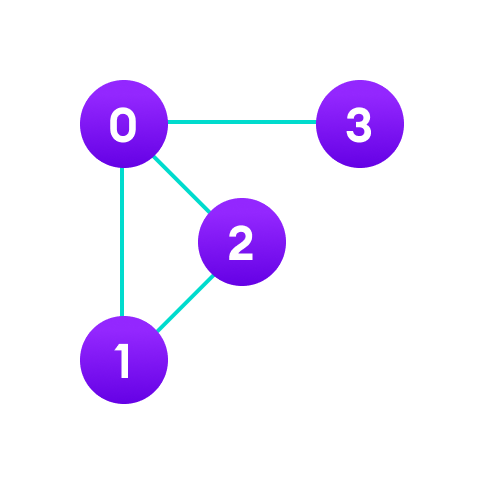
****

**Minimum Spanning Tree (weighted)**

* A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.
* The initial graph is: "Weighted graph"

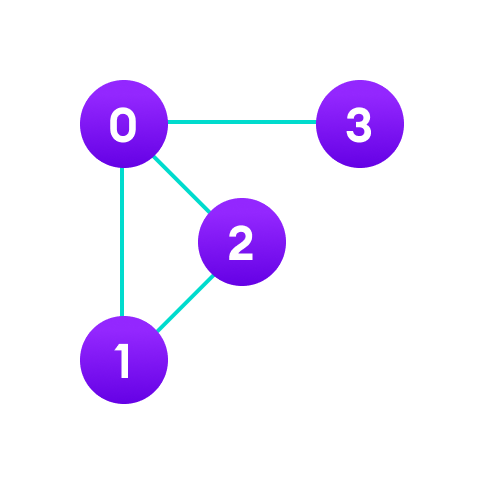
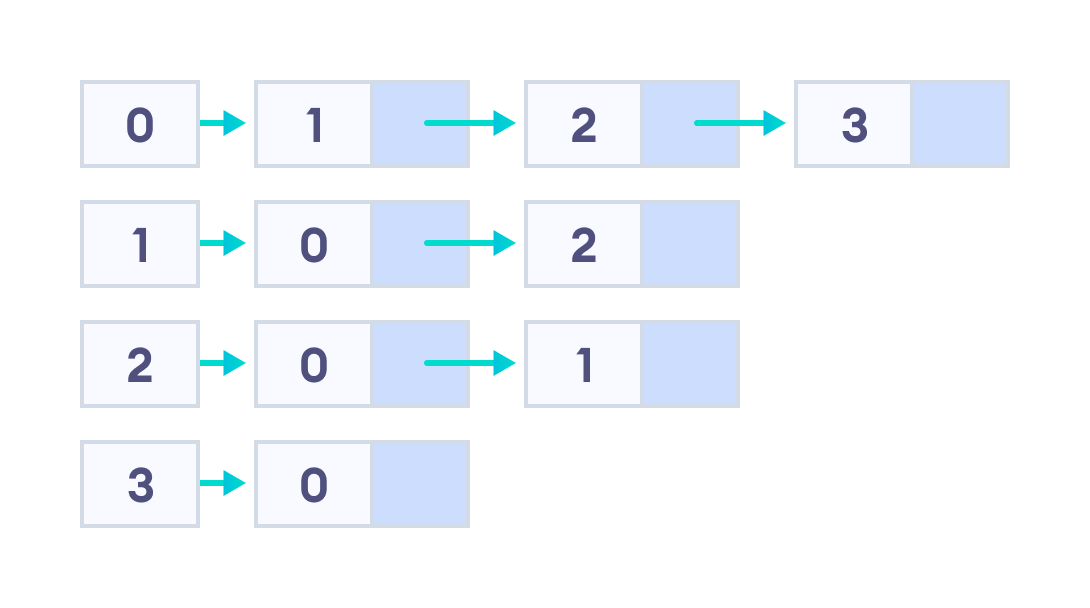
****

**Adjacency List**

* An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
* Array List: Contiguous
* Linked List: Non-Contiguous

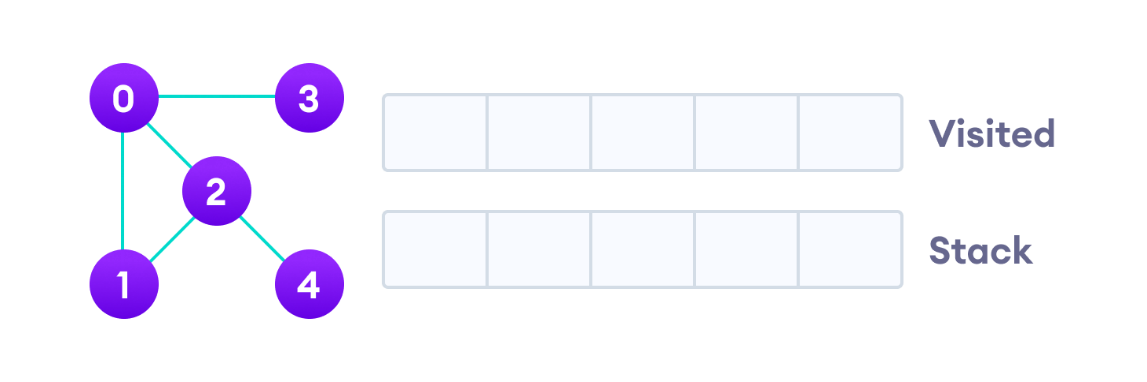
Vertices : **0**, **1**, **2**, **3**:

Each of them forms a linked list with all of its adjacent vertices.

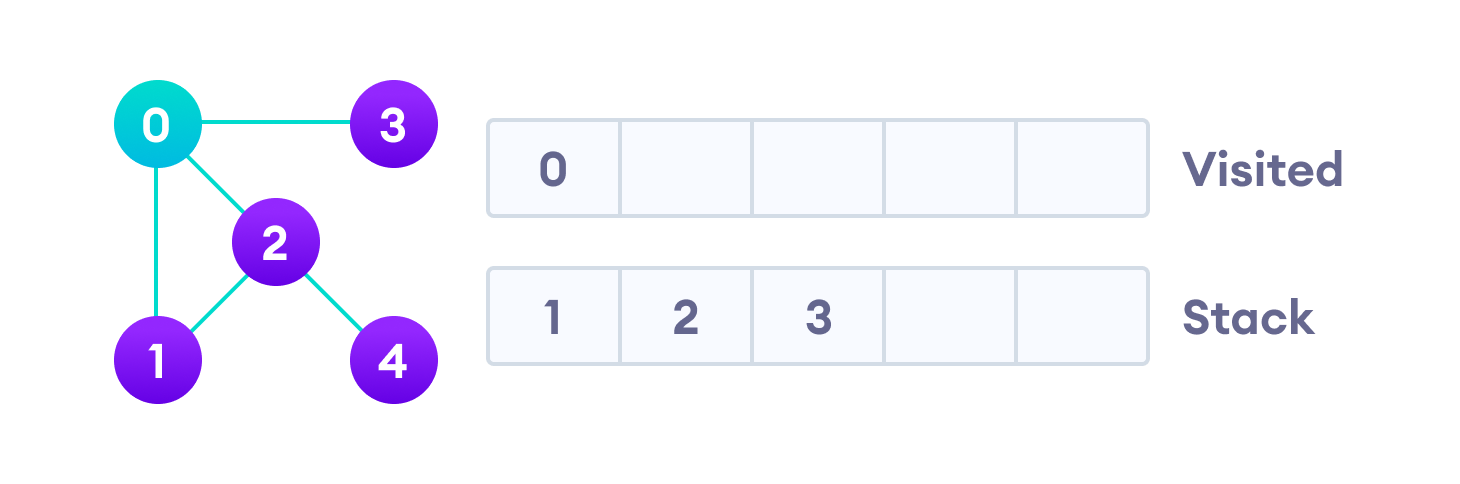


Depth First Search (DFS)

* Depth first Search or Depth first traversal is a algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.
* Depth First Search Example
* Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.
* Undirected graph with 5 vertices



We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

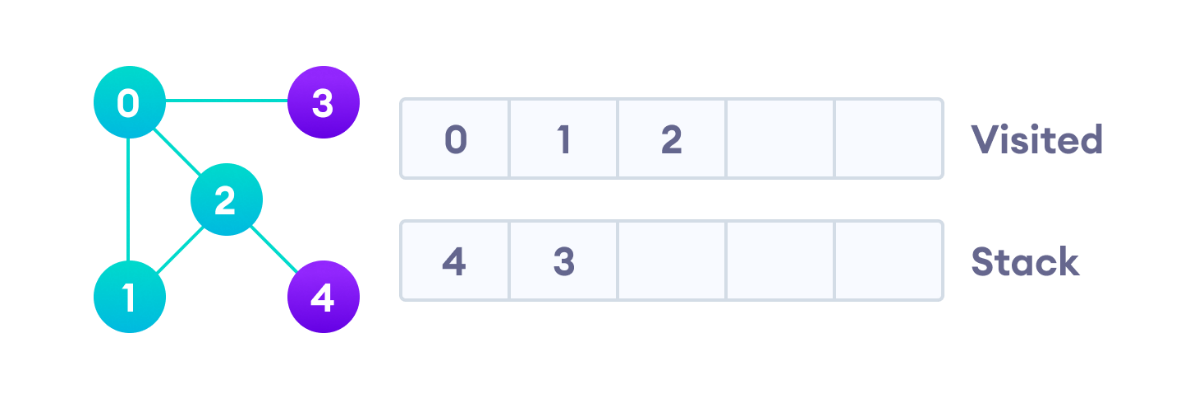
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

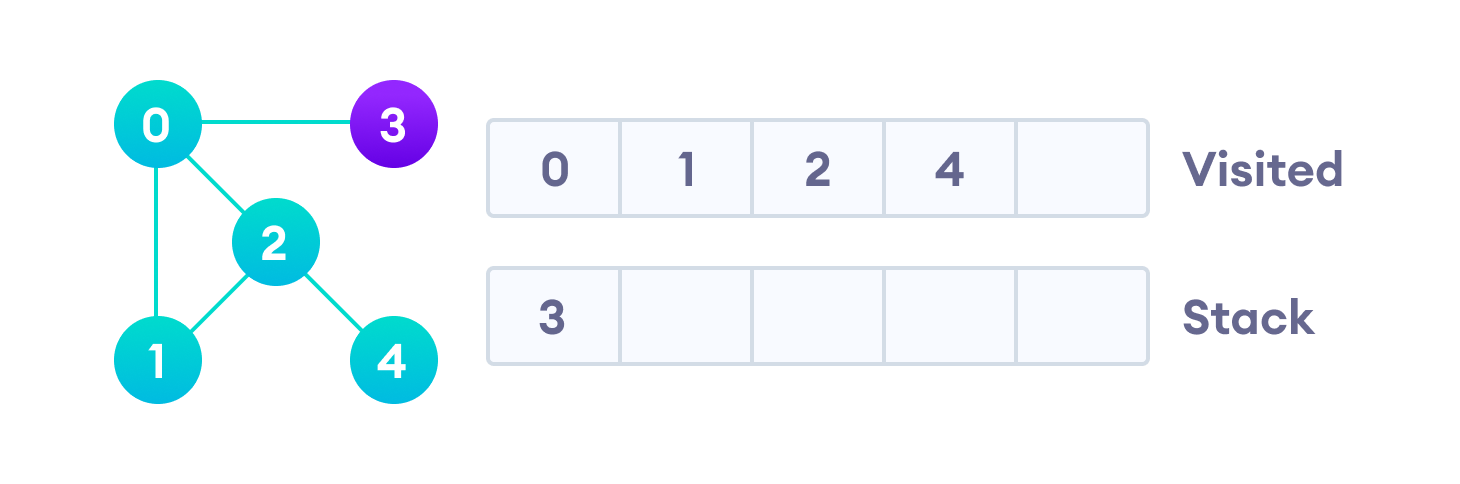
Visit the element at the top of stack

A screenshot of a computer

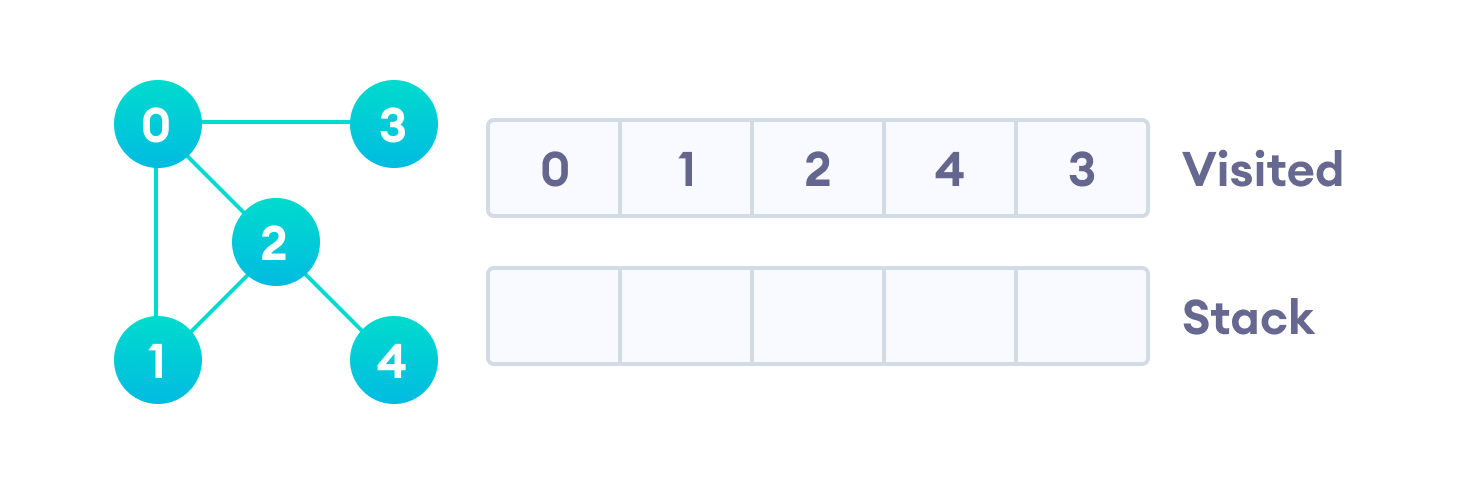
Description automatically generated

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

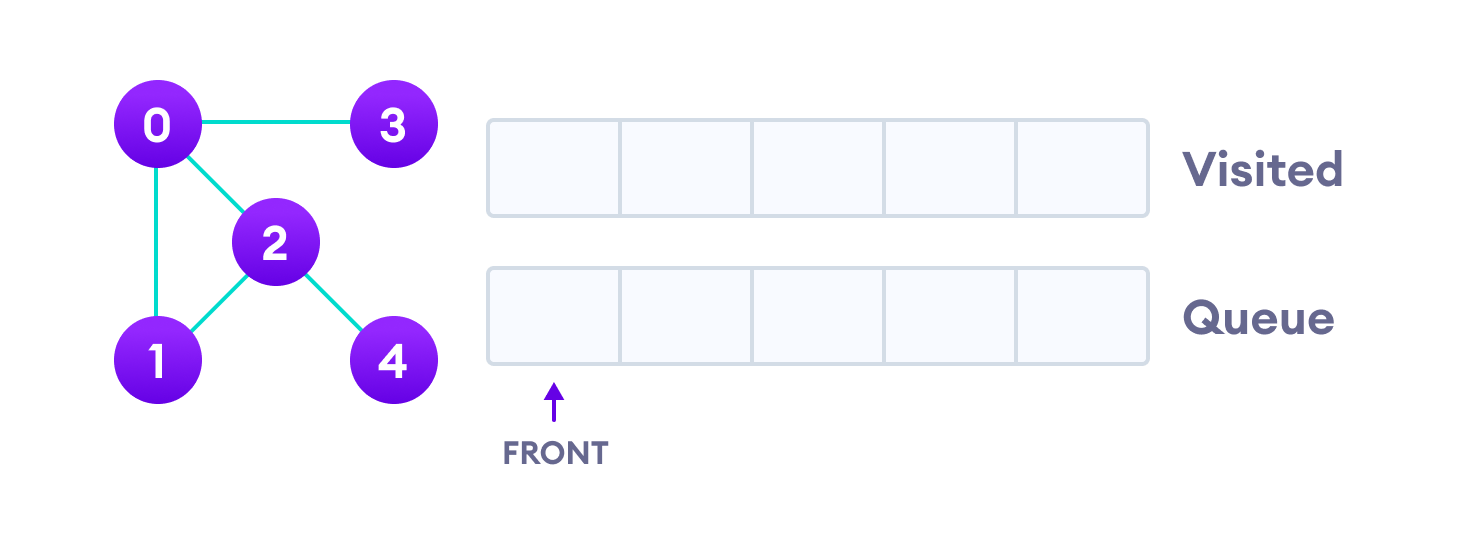
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

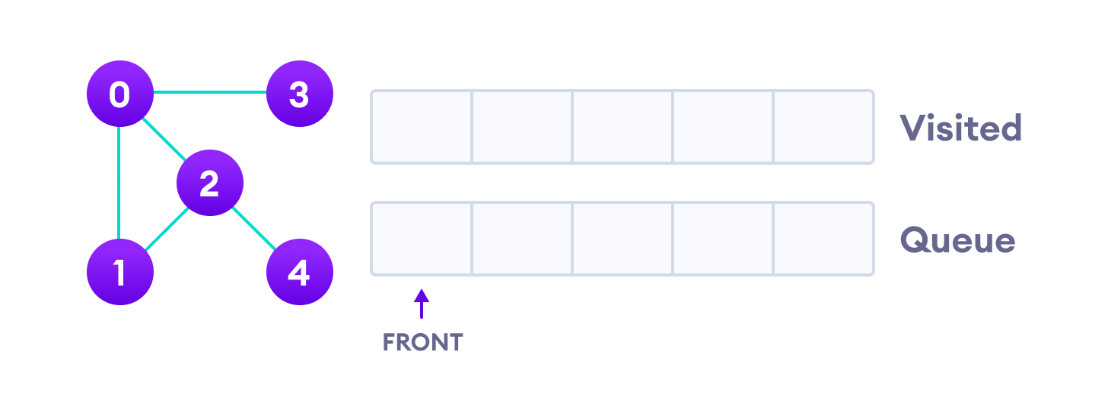


Breadth first search

* Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

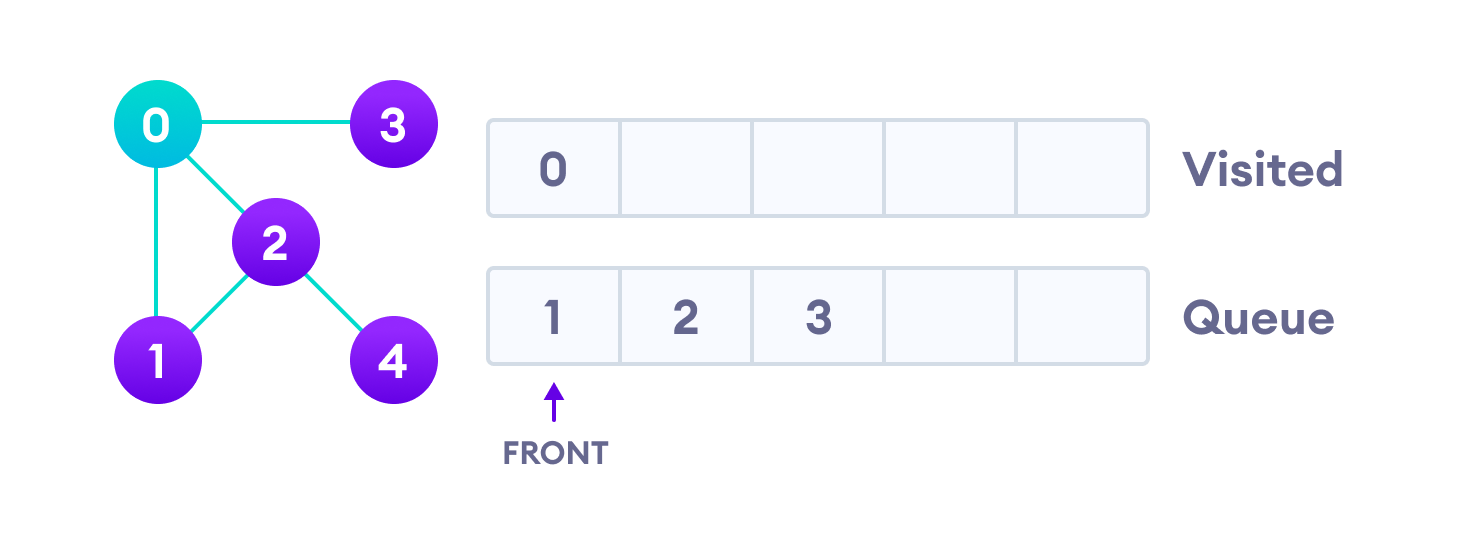
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.

Undirected graph with 5 vertices



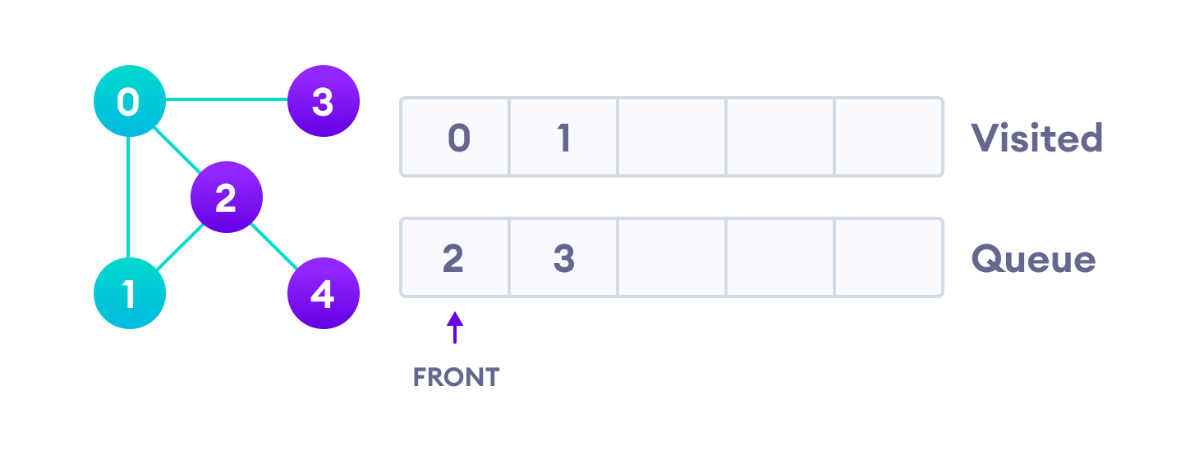
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the queue.

Visit start vertex and add its adjacent vertices to queue



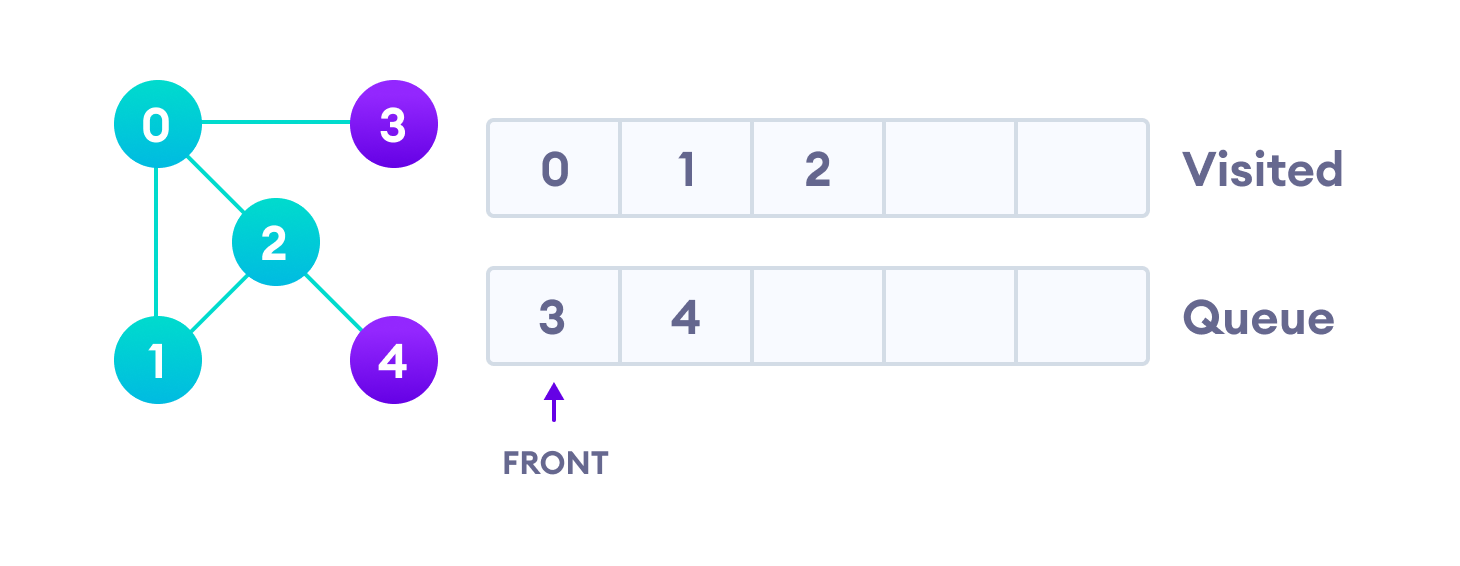
Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

Visit the first neighbour of start node 0, which is 1



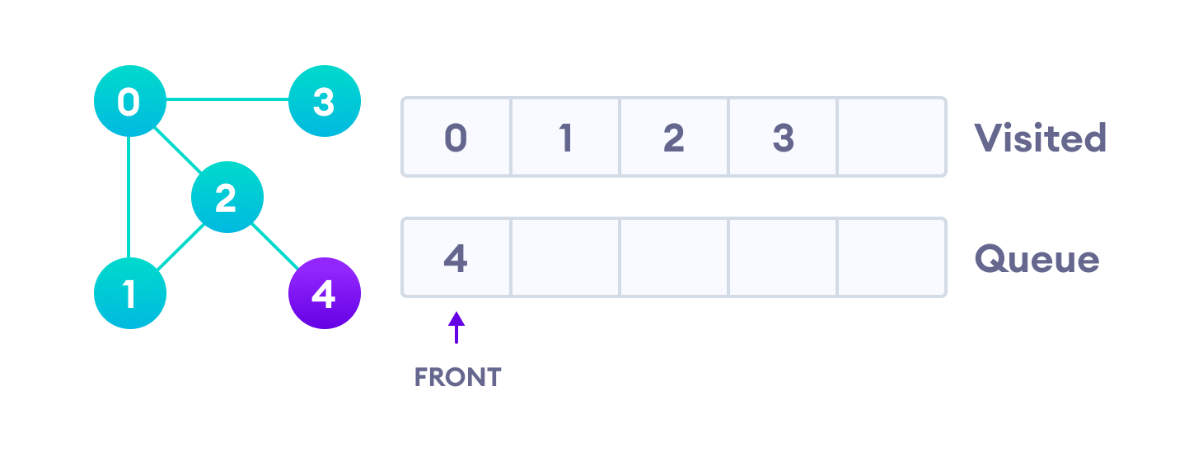
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

Visit 2 which was added to queue earlier to add its neighbours



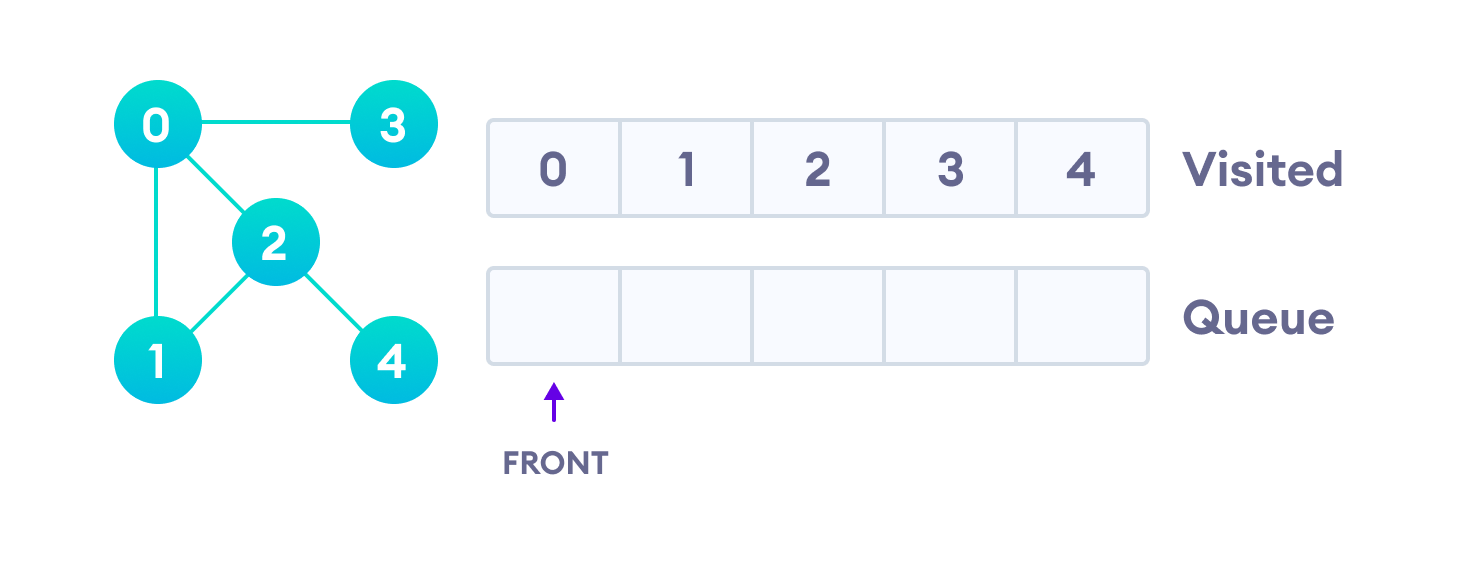
4 remains in the queue

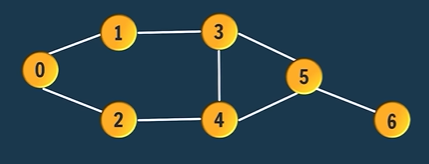
Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Visit last remaining item in the queue to check if it has unvisited neighbors

Since the queue is empty, we have completed the Breadth First Traversal of the graph.



Let’s create a graph using Adjacency list.

* **Graph Code**: Create *Adjacency List* for Graph memory representation
* Step1: Create a node class with Source and Destination Attributes.
* Step2: Create an Array of Array List, arraySize is equals to number of vertices, must initialize all the lists to **concrete list objects** to avoid NullPointerException.
* Array of Array Lists → ArrayList<Edge> graph[] = new ArrayList[vertices];
* Write down the code of step 2 inside the main method
* Step3: Create edges and connect them with vertices:

graph[vertex].add(new Edge(source, destination));

BFS(graphEdges[], vertices)

Step1: SET visited[vertices] //Boolean array

SET Queue[vertices] //i.e., int in this case

SET Q.add(initialSource)

Step4: Repeat While( Queue is not empty)

SET current = Q.remove

Step5: If(visited[current] # TRUE)

print(current)

SET visited[current] = TRUE

[End of If Structure]

SET LCV:= 0

Step6: Repeat while( LCV < graph[current].size)

print graphEdges[current]

Q.add(destination)

SET LCV := LCV +1 [End Step6Loop]

[End Step4 Loop]

Step7: Exit

DFS

* public static void dfs(ArrayList graph[], int curr, boolean visited[]) {

      if(visited[curr])

      { return; }

        System.out.print(curr+" ");

visited[curr] = true;

for(int i=0; i<graph[curr].size(); i++) {

Edge e = graph[curr].get(i);

dfs(graph, e.dest, visited);

}

}

* **Important Interview Questions**

**What is the difference between a stack and linked list?**

1. A stack strictly follows LIFO order. A linked list does not follow any order. It can store and retrieve data randomly.
2. A stack is generally used for the purpose of evaluation of expressions. A linked list is used to store and retrieve data.
3. Insertion and deletion of elements is possible only from the top of the stack. Insertion and deletion of elements from anywhere is possible in case of a linked list.

**When to use Linked list and Array list?**

Use Array list for Storing and accessing and use Linked list for manipulating.

**What is a collection framework?**

A collection framework is a class library to handle groups of objects. Collection framework is implemented in java.util.package.

**Does a collection object store copies of other objects or their references?**

A collection object stores references of other objects.

**Can you store a primitive data type into a collection?**

No, collections store only objects.

**What is the difference between iterator and ListIterator?**

Both are useful to retrieve elements from a collection. Iterator can retrieve the elements only in the forward direction. But ListIterator can retrieve the elements in forward and backward direction also. So ListIterator is preferred to Iterator.

What is the difference between the ArryaList and Vector?

|  |  |
| --- | --- |
| ArrayList | Vector |
| ArrayList object is not synchronized by default. | Vector object is synchronized by default. |
| In case of a single thread, using ArrayList is faster than the vector. | In case of multiple threads, using vector is advisable. With a single thread, vector becomes slow. |
| ArrayList increases its size every time by 50 percent (half) | Vector increases its size every time by doubling it. |