



Data Structures

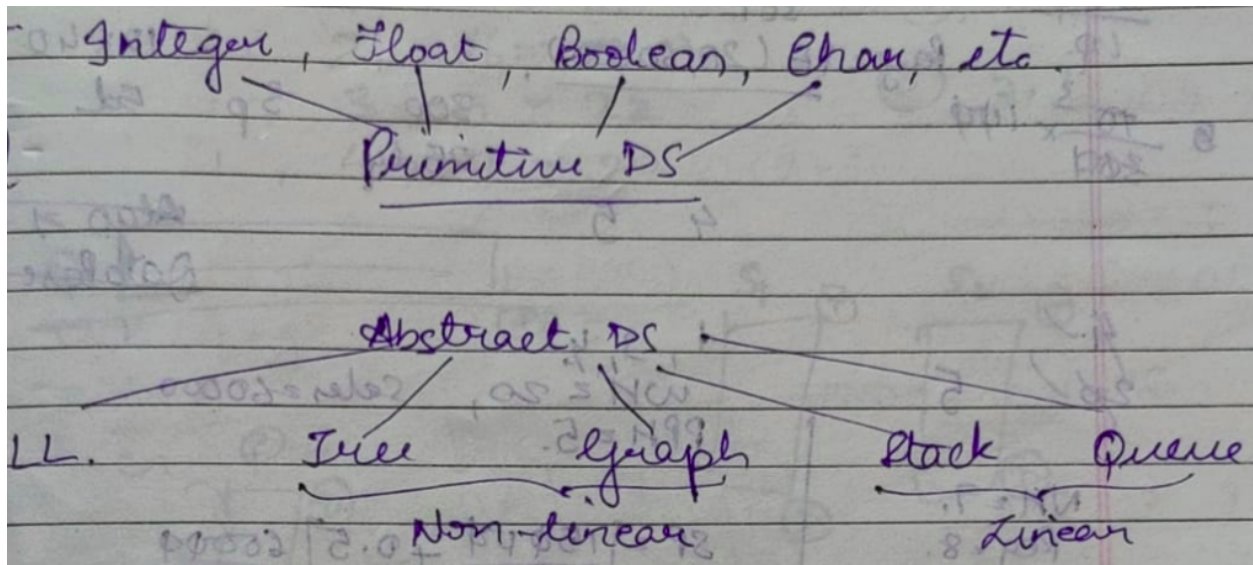
🕒 Created	@February 5, 2022 4:14 PM
☑ Reviewed	<input type="checkbox"/>

Table of contents

- [Table of contents](#)
- [Existence of different data structures](#)
- [Arrays](#)
- [Linked list](#)
- [Time and space complexity](#)
- [Stack](#)
- [Queue](#)
- [Tree concept](#)
- [Graphs](#)
- [Hash tables](#)

Data Structures

There is huge amount of data present today. Now the data need to be stored and processed. The base of that all is a data structure.



▼ Existence of different data structures

There are many basic data structures that can be used to solve application problems. Array is a good static data structure that can be accessed randomly and is fairly easy to implement.

Linked Lists on the other hand is dynamic and is ideal for application that requires frequent operations such as add, delete, and update. One drawback of linked list is that data access is sequential. Then there are other specialised data structures like, stacks and queues that allows us to solve complicated problems (eg: Maze traversal) using these restricted data structures.

One other data structure is the hash table that allows users to program applications that require frequent search and updates. They can be done in $O(1)$ in a hash table.

One of the disadvantages of using an array or linked list to store data is the time necessary to search for an item. Since both the arrays and Linked Lists are **linear structures** the time required to search a "linear" list is proportional to the size of the data set. For example, if the size of the data set is n , then the number of comparisons needed to find (or not find) an item may be as bad as some multiple of n . So imagine doing the search on a linked list (or array) with $n = 106$ nodes. Even on a machine that can do million comparisons per second, searching for m items will take roughly m seconds. This not acceptable in today's world where speed at which we complete operations is extremely important. Time is money. Therefore it seems that better (more efficient) data structures are needed to store and search data.

We can extend the concept of linked data structure (linked list, stack, queue) to a structure that may have multiple relations among its nodes. Such a structure is called a **tree**. A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a *nonlinear* data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more subtrees.

▼ Arrays

`int x[] = {5,8,2,7}`

int is a primitive data type which takes **32 bits (4 bytes)** to store.

Integer is an object which takes **128 bits (16 bytes)** to store int value.

5	8	2	7	← Value
101	103	105	107	← Contiguous location
0	1	2	3	← Index

Cons:

- Not expandable
- Size need to specify at the beginning
- Insertion and deletion are quite difficult in an array

Pros:

- Fast in terms of fetching values
- Arrays allocate memory in contiguous memory locations for all its elements. Hence there is no chance of extra memory being allocated in case of arrays. This avoids memory overflow or shortage of memory in arrays.

▼ Dynamic array:

Dynamic arrays are those arrays which are allocated memory at the run time with the help of heap. Thus Dynamic array can change its size during run time.

Dynamic array usually have a growth factor of $3/2$ to 2. But once the memory is allocated, **it is never shrink automatically**.

Output of LearnDynamicArray program:

```

Elements of array:
1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0
Size of array: 16
No of elements in array: 9
Elements of array after shrinkSize of array:
1 2 3 4 5 6 7 8 9
Size of array: 9
No of elements in array: 9
Elements of array after add an element at index 1:
1 22 2 3 4 5 6 7 8 9 0 0 0 0 0 0
Size of array: 18
No of elements in array: 10
Elements of array after delete last element:
1 22 2 3 4 5 6 7 8 0 0 0 0 0 0 0
Size of array: 18
No of elements in array: 9
Elements of array after delete element at index 1:
1 2 3 4 5 6 7 8 0 0 0 0 0 0 0 0
Size of array: 18
No of elements in array: 8

```

▼ Linked list

- Variable size
- Non-contiguous memory

All types are implemented:

Linked List - Singly + Doubly + Circular (Theory + Code + Implementation)

Learn complete Singly + Doubly + Circular #LinkedList in a single video! One of the most important data structures for coding interviews. Here we also build ...

📺 <https://www.youtube.com/watch?v=58YbpRDc4yw>



Basics - not all types are implemented and see subsequent videos:

#3 What is wrong with Array? | Why LinkedList?

Complete playlist of Data Structure Using Java : <https://goo.gl/3eQAYB> In this video we will see- What is Array- How to create Array- Arrays : pros and cons- ...

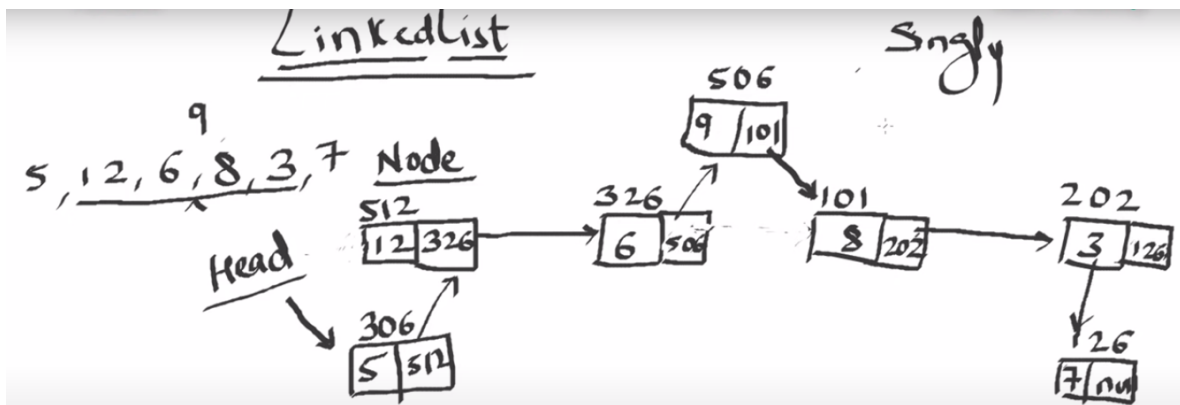
📺 <https://www.youtube.com/watch?v=48uzKhtyEvl&list=PLsyeobzWxl7oRKwDi7wjrANsbhTX0IK0J&index=3>



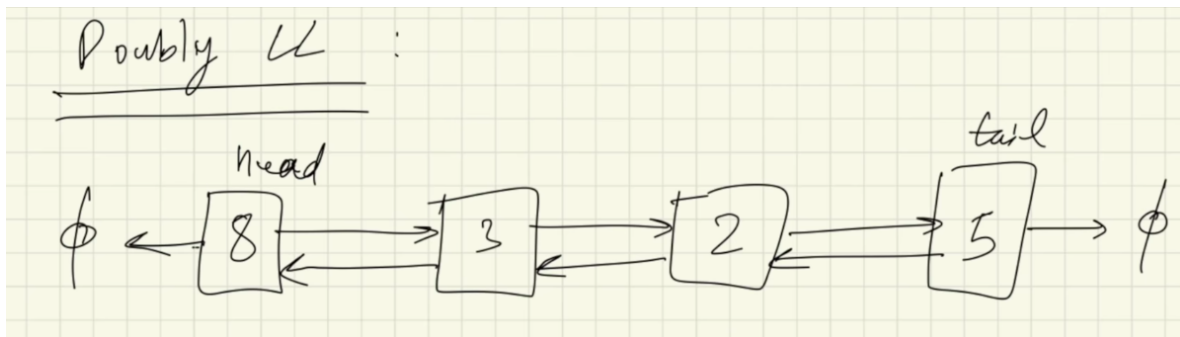
▼ Linked list vs Array

Array	Linked list
Can fetch value randomly $O(1)$	Need to start from head and then follow the nodes. Search time complexity $O(n)$
Slow Insert $O(n)$	Fast insert $O(1)$
Not expandable	Expandable

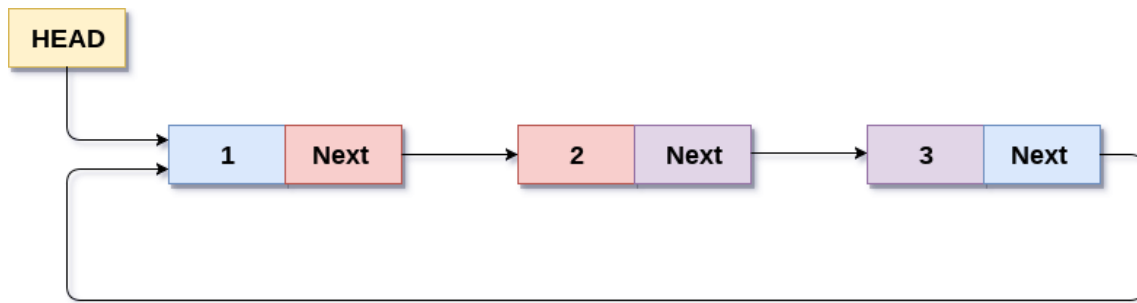
▼ Singly linked list



▼ Doubly linked list



▼ Circular linked list



Circular Singly Linked List

▼ Time and space complexity

▼ Time complexity

Relation between input size (let say input of value n which we take) & running time (operations)

```
Example: input -> n
psvm{
for(int i=0; i<n; i++){
sout("Hello")}
}
Running time -> n times it take to run 1 sout.
```

▼ Types

Understand using example of finding 1 in a sequence made using 1,2,3,4,5.

- **Best case:** 1 would be at first position.

1 operation i.e. 1 unit of time

Here, $\Omega(1)$

- **Average case:** it could be randomly at any position

so, $(1+2+3+4+5)/5 \rightarrow (1+2+3+...+n)/n = n(n+1)/2n = (n+1)/2$

It is linear relation only as it is proportional to n.

Here, $\theta(\frac{n+2}{2})$

- **Worst case:** when n=5 : 2,3,4,5,1

In case of n, we will have to check till the last element. So, proportional to n.

Here, $O(n)$

In general we refer to the **worst case** time complexity.

```

Example: input -> n
int[] arr = {1,2,3...n};
for(int a : arr){
    for(int b: arr){
        sout("Hey");
    }
}
Time complexity ->  $O(n*n) = O(n^2)$ 

```

Basics of Time Complexity and Space Complexity | Java | Complete Placement Course | Lecture 9

4 Time Complexity

```

public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int m = sc.nextInt();

    for(int i=0; i<n; i++) {
        for(int j=0; j<m; j++) {
            System.out.println("hello");
        }
    }
}

```

Handwritten notes on the video screen:

- For $i=0$, $j=0, 1, 2, \dots, m$
- For $i=1$, $j=0, 1, 2, \dots, m$
- For $i=2$, $j=0$ to m
- ...
- For $i=n$, $j=0$ to m
- Time complexity: $n \times m$
- Time complexity: $O(n \times m)$

Video player controls: 14:14 / 21:54

▼ Space complexity

Basics of Time Complexity and Space Complexity | Java | Complete Placement Course | Lecture 9

Solutions of the previous exercise:

<https://docs.google.com/document/d/1GH4dektLeTFyTEaowbPyv2ZvyCXnlp1Sbw34NzKBI4/edit?usp=sharing>

Link to Notes(Questions w...)

<https://www.youtube.com/watch?v=bQssdSrSGNE>



▼ Stack

- Stack using array
- Dynamic stack
- stack using linked list - collections

Implement a stack using singly linked list - GeeksforGeeks

To implement a stack using singly linked list concept, all the singly linked list operations are performed based on Stack operations LIFO(last in first out) and with the help of that knowledge we are going to implement a stack using single linked list.

<https://www.geeksforgeeks.org/implement-a-stack-using-singly-linked-list/>



- **Stack using double queue - collections**
- **Parenthesis program - collections**
- Trapping water program : Method 4 in below

Trapping Rain Water - GeeksforGeeks

A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and practice/competitive programming/company interview Questions.

 <https://www.geeksforgeeks.org/trapping-rain-water/>



▼ Queue

Implementation using array

Circular queue using array


Using linked list

Circular queue using linked list

Deque using doubly linked list:

Implementation of Deque using doubly linked list - GeeksforGeeks

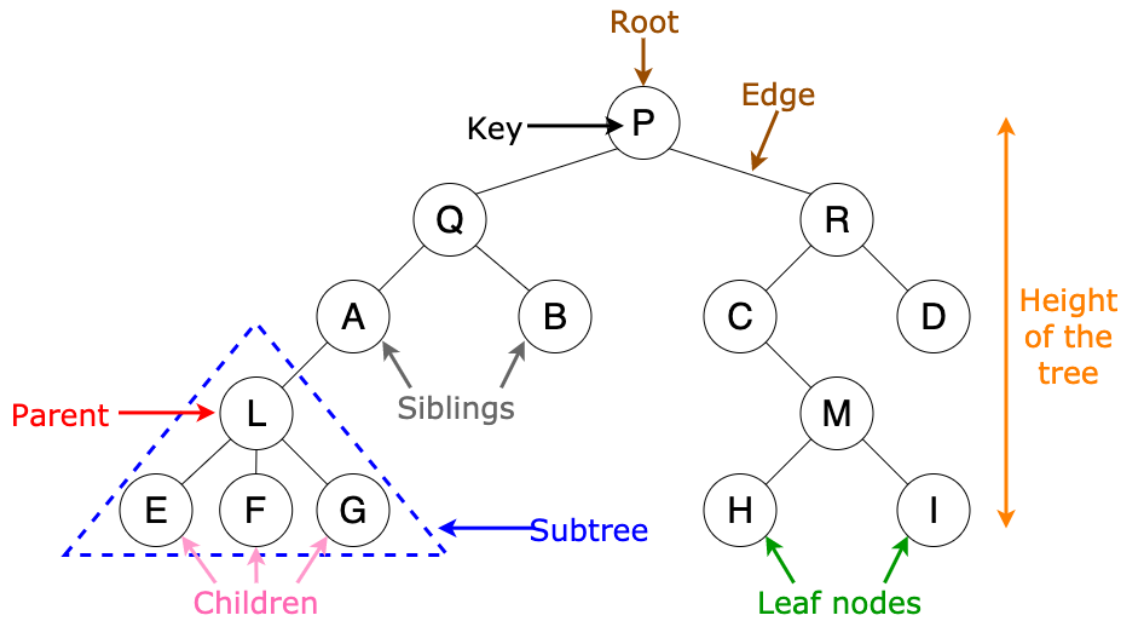
Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends. In previous post Implementation of Deque using circular array has been discussed. Now in this post we see how we implement Deque using Doubly Linked List.

 <https://www.geeksforgeeks.org/implementation-deque-using-doubly-linked-list/>



▼ Tree concept

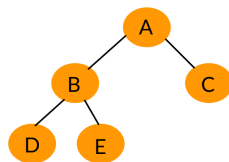
▼ Introduction



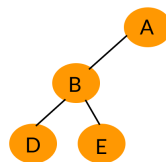
Height of L = 3 [consider it like index starting from root as 0]

Depth of L = 1

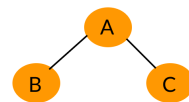
Height of tree = Height of root node = 4 (max edges)



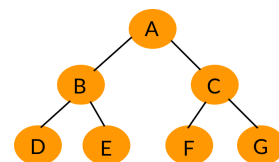
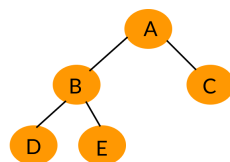
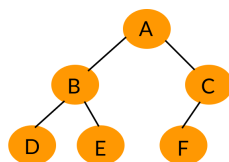
Complete Binary Tree



Full Binary Tree



Perfect Binary Tree



Complete binary tree: nodes should be on level I and $I-1$.

▼ Binary tree

Maximum 2 nodes are present.

Applications:

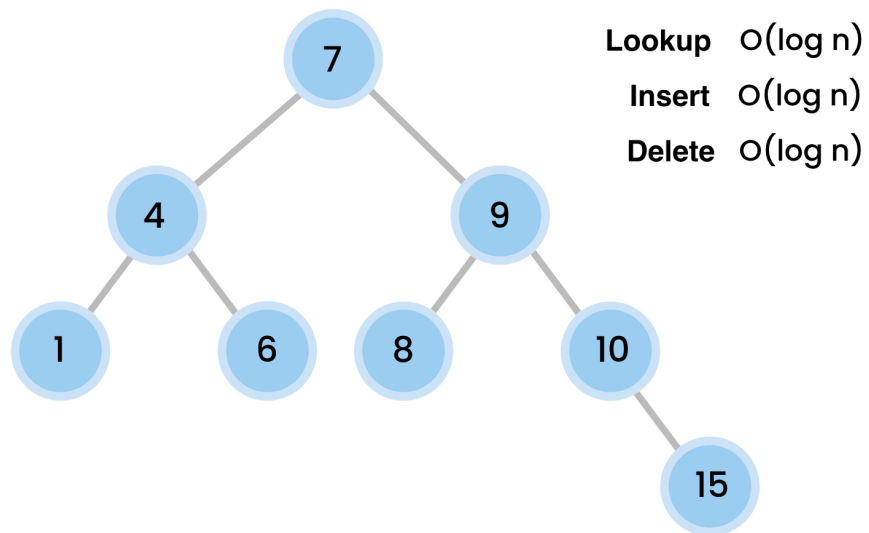
- Represent hierarchical data
- Autocompletion

- Compilers

▼ Binary search tree

left < node < right

In a Binary Search Tree (BST), all keys in left subtree of a key must be smaller and all keys in right subtree must be greater. So a Binary Search Tree **by definition has distinct keys and duplicates in binary search tree are not allowed.**



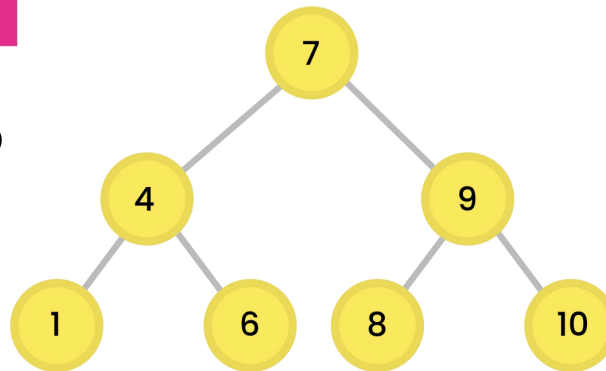
▼ Traversals

- **Breadth first:** level order traversal

BREADTH FIRST

Level Order

7, 4, 9, 1, 6, 8, 10



- **Depth first:**

- Pre-order: root, left, right

7 4 1 6 9 8 10

- In-order: left, root, right [ascending order]

1 4 6 7 8 9 10

- Post-order: left, right, root [leaves to nodes]

1 6 4 8 10 9 7

▼ Graphs

By node here means vertex present in the graph.

▼ Adjacency matrix

ADJACENCY MATRIX

Space $O(V^2)$

Add Edge $O(1)$

Remove Edge $O(1)$

Query Edge $O(1)$

Find Neighbors $O(V)$

Add Node $O(V^2)$

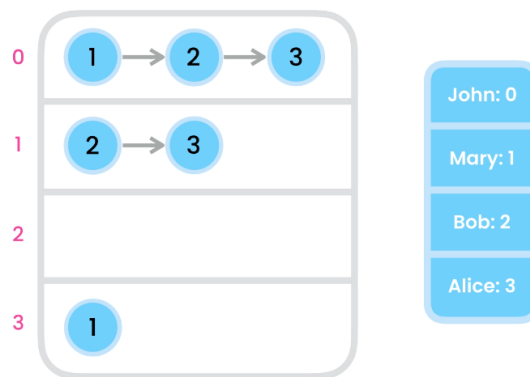
Remove Node $O(V^2)$

	John	Mary	Bob	Alice
John	0	1	1	1
Mary	0	0	1	1
Bob	0	0	0	0
Alice	0	0	1	0

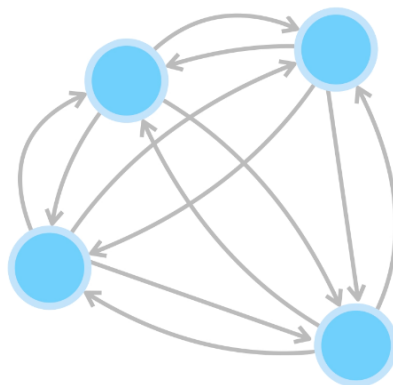
Blue line shows the connection of John with others. Hence, over here John is connected to Mary while Mary is not connected to John.

Space is wasted. Hence, solution adjacency list is used.

▼ Adjacency list



More space efficiency over here. $O(E + V)$



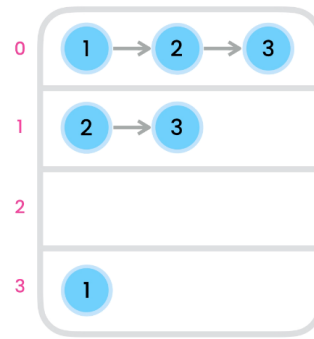
DENSE GRAPH

$$E = V \times (V - 1)$$

$$E = V^2 - V$$

$$O(V + E) = O(V^2)$$

Add node: $O(1)$



Remove node: $O(V^2)$ [remove nodes from each linked list as well]

Add edge: $O(K) = O(V)$ [just need to make sure that edge does not exist in the linked list of that node. V in case when every node is connected to every other]

Find neighbour: $O(K) = O(V)$

	MATRIX	LIST
Space	$O(V^2)$	$O(V + E)$
Add Edge	$O(1)$	$O(K)$
Remove Edge	$O(1)$	$O(K)$
Query Edge	$O(1)$	$O(K)$
Find Neighbors	$O(V)$	$O(K)$
Add Node	$O(V^2)$	$O(1)$

	MATRIX	LIST
Space	$O(V^2)$	$O(V^2)$
Add Edge	$O(1)$	$O(V)$
Remove Edge	$O(1)$	$O(V)$
Query Edge	$O(1)$	$O(V)$
Find Neighbors	$O(V)$	$O(V)$
Add Node	$O(V^2)$	$O(1)$
Remove Node	$O(V^2)$	$O(V^2)$

In case when we have **dense graph** then use **adjacency matrix**.

In most cases adjacency list is used.

▼ Graph implementation

We are passing a value which provides us the actual node which stores the data. We have kept it this way because when we use node object we can add the fields or methods which we need to store in it at any time.

Now this node points to a list of node to which it is connected.

```
private Map<Integer, Node> nodes = new HashMap<>();

private Map<Node, List<Node>> adjacencyList = new HashMap<>();

//Incase when we want to add a node and it is not present, we create an
//empty ArrayList and pass that.
```

▼ Complete Code

```
package com.java.data.structures;

import java.util.*;

public class Graph
{
    private class Node
    {
        private int label;

        public Node(int label)
        {
            this.label = label;
        }

        @Override
        public String toString()
        {
            return label + "";
        }
    }

    private Map<Integer, Node> nodes = new HashMap<>();

    private Map<Node, List<Node>> adjacencyList = new HashMap<>();
```

```

public void addNode(int label)
{
    Node node = new Node(label);

    nodes.putIfAbsent(label, node);

    adjacencyList.putIfAbsent(node, new ArrayList<>());
}

public void addEdge(int from, int to)
{
    try
    {
        Node nodeFrom = nodes.get(from);

        Node nodeTo = nodes.get(to);

        if (nodeFrom == null || nodeTo == null)
        {
            System.out.println("Node not present.");
        }

        adjacencyList.get(nodeFrom).add(nodeTo);

//         adjacencyList.get(nodeTo).add(nodeFrom);           //to implement non directed graph
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}

public void print()
{
    try
    {
        for (Node n : adjacencyList.keySet())
        {
            System.out.print(n.label + " is connected to ");

            System.out.println(adjacencyList.get(n));
        }
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}

public void removeNode(int label)
{
    try
    {
        Node node = nodes.get(label);

        if (node == null)
        {
            System.out.println("No node found");

            return;
        }

//         List<Node> list = adjacencyList.get(node);

        for (Node n : adjacencyList.keySet())
        {
            List<Node> list = adjacencyList.get(n);

            list.remove(node);
        }

        adjacencyList.remove(node);
    }
}

```

```

        nodes.remove(node);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}

public void removeEdge(int from, int to)
{
    try
    {
        Node toNode = nodes.get(to);

        Node fromNode = nodes.get(from);

        if (fromNode == null || toNode == null)
        {
            System.out.println("Node not present");

            return;
        }

        List<Node> list = adjacencyList.get(fromNode);

        if (!list.remove(toNode))
        {
            System.out.println("Edge not present");
        }

    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}

public static void main(String[] args)
{
    try
    {
        Graph graph = new Graph();

        graph.addNode(1);

        graph.addNode(2);

        graph.addNode(3);

        graph.addNode(4);

        graph.addEdge(1, 3);

        graph.addEdge(1, 4);

        graph.addEdge(2, 3);

        graph.addEdge(4, 3);

        graph.print();

        System.out.println("\nRemove node 2");

        graph.removeNode(2);

        graph.print();

        System.out.println("\nRemove edge 1,4");

        graph.removeEdge(1, 4);

        graph.print();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}

```

```

        System.out.println("\nRemove edge 3,4");

        graph.removeEdge(3, 4);

//        graph.print();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}
}

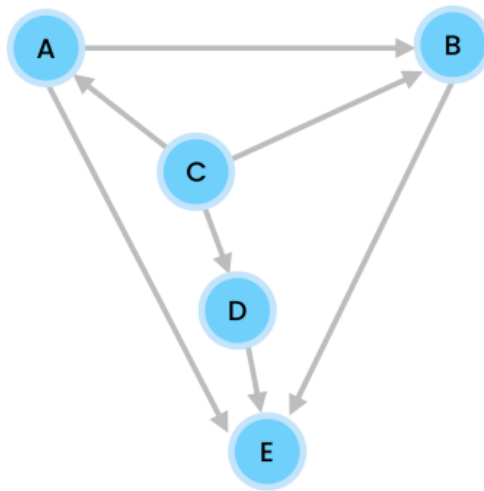
```

▼ Traversal algorithm example

Given the following graph, what would be the output of breadth-first and depth-first searches starting from C?

At each step, when deciding where to go next, pick the node based on alphabetical order. For example, if you can choose between A, B and D as the next step, go to A.

Solution: next page



Depth-first Traversal

Answer: [C, A, B, E, D]

We start at the C.

Output: [C]

C has three neighbours: A, B and D. Since we've assumed that these nodes are inserted in alphabetical order, we should go to A next.

Output: [C, A]

A has two neighbours: B and E. Based on our assumption, we should go to B next.

Output: [C, A, B]

B has only one neighbour, so we go to E next.

Output: [C, A, B, E]

We can't go any deeper since E has no neighbours, so we go back to B.

We've visited all the neighbours of B, so we go back to A.

A has one more neighbour (E) but we've visited that node, so we go back to C.

Once again, C has three neighbours: A, B and D. We've visited the first two, so now we should go to D.

Output: [C, A, B, E, D]

Breadth-first Traversal

Answer: [C, A, B, D, E]

We start by adding C in a queue.

Queue: [C]

We remove C from the queue and visit it.

Output: [C]

Queue: []

Now we should add its unvisited neighbours to the queue (in alphabetical order).

Output: [C]

Queue: [A, B, D]

We remove A from the queue and visit it.

Output: [C, A]

Queue: [B, D]

Then we add the unvisited neighbours of A to the queue.

Output: [C, A]

Queue: [B, D, B, E]

We remove B from the queue and visit it.

Output: [C, A, B]

Queue: [D, B, E]

Again, we add the unvisited neighbours of B to the queue.

Output: [C, A, B]

Queue: [D, B, E, E]

Now, we remove D from the queue and visit it.

Output: [C, A, B, D]

Queue: [B, E, E]

We add the unvisited neighbours of D to the queue.

Output: [C, A, B, D]

Queue: [B, E, E, E]

We remove B from the queue but we've already visited it.

Output: [C, A, B, D]

Queue: [E, E, E]

We remove E from the queue and visit it.

Output: [C, A, B, D, E]

Queue: [E, E]

E doesn't have any neighbours, so we continue polling the queue.

We remove E from the queue but we've already visited it.

Output: [C, A, B, D, E]

Queue: [E]

We repeat.

Output: [C, A, B, D, E]

Queue: []

The queue is empty and we're done.

- Depth first search - stack
- Breadth first search - queue

▼ Hash tables

Employee → employee number → hash function → store value at a given address based on the hash function's value

Hash function is deterministic means it will return the same value always when we input same employee number.

Hence, it can be used to **reliably fetch or insert a value**.

Uses array internally.

How does a Java HashMap handle different objects with the same hash code?

[https://stackoverflow.com/questions/6493605/how-does-a-java-hashmap-handle-different-objects-with-the-same-hash-code#:~:text=If two keys are the,look in the same bucket\).](https://stackoverflow.com/questions/6493605/how-does-a-java-hashmap-handle-different-objects-with-the-same-hash-code#:~:text=If two keys are the,look in the same bucket).)

Time complexity:

Insert $O(1)$

Lookup $O(1)$

Delete $O(1)$