# Lecture 2: Time Complexity and Space Complexity

## What is an algorithm?

Hensel and Gratel

- Two children + Father + Mother(died) ⇒ Step Mother
- Plan to abandon the kids.
- They take the kids to the forest for picnic.
- Hensel overheard. Collected pebbles.
- House → Picnic Spot. Dropping pebbles.
- At night follow the pebbles to home.

2 km → pebble every fifty meter → 40 pebbles.

Follow Pebbles To(Home): → Problem

1. Repeat until reach Home:
    a. Look for the next pebble → Problem
        i. Look around
        ii. Identify the pebble
    b. Walk upto the next pebble → Problem
    c. Collect the pebble

An algorithm is a **process** that a **computer** can follow to solve a **problem**.

- Problem
    - To reach home
    - Following a schedule
    - Tying your shoe

    - Cleaning the room
- Computer
    - Anything that can compute and compare
    - And follow clear instruction
- Process
    - English / Pseudocode
    - Clear and precise

Problems and processes → Functions and procedures →

# How should we compare algorithms?

## Empirical Measurement

Tale of two programmer

We have two programmers → Khyati and Radhika

And one problem → P

Two algorithms → Pk and Pr

Which one is the best solution?

- Time and Space
- Whichever takes less time and space to solve the problem.
- Whichever is easier to code.

How do you measure the time taken?

- Run the problem with the solution on your computer and report.
    - Computer speed differences.
- Run the problem with the solutions on the same computer and report.
    - CPU cycle differences.
    - Input(s) might be different for the two solution.
    - Khyati’s code might run faster on smaller inputs but Radhika’s code scales well.

- Who has the time for doing all these operations?

## Deductive Measurement

Pk

| Input Size | Number of Steps |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| 5 | 25 |
| | |
| | |

Example:

```
n = 10
sum = 0
for var in range(n): # [0, 1, 2, 3, ... , n-1]
#for(var = 0; var < n; var++)
  sum += 1

print(sum)
```

Assume each operation will take exactly the same time for one execution.

| Operations | Number of executions | Time per operations | Approximated time |
|---|---|---|---|
| n = 10 | 1 | 20ms | 60ms |
| sum = 0 | 1 | 30ms | 60ms |
| for var in range(n): | 2 * n | 50ms | 60ms |
| sum += 1 | 3 * n | 20ms | 60ms |
| print(sum) | 1 | 60ms | 60ms |

Time = 1 + 1 + 2n + 3n + 1 = 5n + 3 = O(n)

Approximation:

- Remove all the constants
- Remove all the coefficients
- Remove all the lower order terms

Time = 3n^2 + n^3 + 5n → O(n^3)

## Big O

Let’s say f(n) = O(g(n)),

For a sufficiently large input,

f(n) ≤ c * g(n), where c is a constant

Examples:

f(n) = 5n + 3

g(n) = n

f(n) ≤ 10 g(n) for n > 6

Thus f(n) = O(g(n)) = O(n)

# Lecture 3: Calculating Time Complexity

## Examples:

```
def printSum(n: int):
  sum = 0 # 1
  pie = 3.14 # 1
  for var in range(0, n, 3): # n/3 times
    print(pie) # 1 + 1 + 1 + 1 + ... (n/3 times) = (n/3) * 1
    for j in range(0, n, 2): #  (n/2)
      sum += 1 # (3)
      print(sum) # (1)

# T(printSum) = 1 + 1 + (n/3) + (n/3) * (loop part)
# = 2 + (n/3) + (n/3) * ( 1 + (n/2) + (n/2) * (inner loop part))
# = 2 + (n/3) + (n/3) * ( 1 + (n/2) + (n/2) * ( 3 + 1))
# * = O(n + n  * ( n + n ))
# = O(n + n * (2n))
# = O(n + n ^ 2)
# = O(n ^ 2)

def printLoop(n: int):
  # 1 + (n/2) + (n/2) * 3 + (n/2) * 1
  # 1 + (n/2) + (n/2) * (inner part of the loop)
  sum = 0 # 1
  for j in range(0, n, 2): # (n/2)
    sum += 1 # (n/2) * (3)
    print(sum) # (n/2) * (1)
```

```
void printSum(int n){
  int sum = 0;
  float pie = 3.14;
  for(int var = 0; var < n; var += 3){
    cout << pie << endl;
    for(int j = 0; j < n; j += 2){
      sum++;
      cout << sum << endl;
    }
  }
}
```

### Loop with Multiplication

```
def printLoopMulti(n : int):
  pie = 3.14
  var = 1
  total = 0
  while var < n: # log(n)
    # var will take the following values:
    # 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024,...
    print(pie) # log(n) * O(1)
    for j in range(var):
      # sum over i of (2 ^ i) = 2 ^ (Largest i) - 1 = 2 ^ (log n) = n
      total += 1
    var *= 2
    # let i be the number of iterations
    # var = 2 ^ i
    # the loop ends when var >= n
    # i.e. 2^i >= n
    # i.e. i >= log(n)
  print(total)


def printLoopMulti(n: int):
  pie = 3.14 # 1
  var = 1 # 1
  total = 0 # 1
  while var < n: # log(n)
    print(pie) # log(n) * 1
    for j in range(var): # n
      total += 1 # n * 3
    var *= 2 # log(n) * 3
  print(total) # 1

# T(printLoopMulti) = 1 + 1 + 1 + log(n) + log(n) * 1 + n + n * 3 + log(n) * 3 + 1
# = O(log(n) + log(n) + n + n  + log(n))
# = O(n + log(n))
# = O(n)
```

```
var, n = 1, 64
while var < n:
  for j in range(var):
    do something
  var *= 2

# If we fix n, then this can also be written as

for j in range(1):
  do something
for j in range(2):
  do something
for j in range(4):
  do something
```

```
for j in range(8):
  do something
for j in range(16):
  do something
for j in range(32):
  do something

# Terms = 1  2  4  8  16
# Sums =  1  3  7  15 31
# A given sum is 1 less than twice the last term
```

### Time Complexity

# For loop in Python vs C++

```
for i in range(n): # Will run for n times

for(int i = 0; i < n; i++) # Will run for n + 1 times
```

# Lecture 4: Space Complexity

- Time Complexity: Worse case, n = 100
- What is worse case?
  - You are searching for an element in an array.
  - The algorithm you follow is:
    - Iterate through each element
    - And match the element with target
  - How much time will this algorithm take for an input of size 100?
    - We need to find the time when the element is not present in the array.
    - That is our worse case.
  - T = 3 n + 4
- Three Notations: Big-O, Omega, Theta
  - Big-O: Upper Bound - T = O(n)
  - Omega: Lower Bound - T = Omega(n)
  - Theta: Combination of Upper Bound and Lower Bound - T = Theta(n)
- What are you measuring with Space Complexity?
  - You measure the space used in the memory.
  - It’s usually much easier. Because you can just add up the data structures you use.
  - list = []
  - len(list) = n

## Introduction to Lists

In Python, a list is an ordered sequence of *heterogeneous* elements. In other words, a list can hold elements with different data types. For example,

mylist = [‘a’, ‘apple’, 23, 3.14]

## Initializing a list

```
example_list = [3.14159, 'apple', 23]  # Create a list of elements
empty_list = []  # Create an empty list
sequence_list = list(range(10))  # Create a list of first 10 whole numbers
print(example_list)
print(empty_list)
print(sequence_list)
```

So lists can hold integers, strings, characters, functions, and pretty much any other data type including *other lists* simultaneously! Look at the following example. another_list contains two lists, a string, and a function! The elements can be accessed or ‘indexed’ using square brackets. The first element in a list is accessed using index 0 (as on **line 7**), the second element is accessed using 1, and so on. So list indices start at 0.

```
a_list = [2, 'Educative', 'A']


def foo():
    print('Hello from foo()!')


another_list = [a_list, 'Python', foo, ['yet another list']]

print(another_list[0])  # Elements of 'aList'
print(another_list[0][1])  # Second element of 'aList'
print(another_list[1])  # 'Python'
print(another_list[3])  # 'yet another list'

# You can also invoke the functions inside a list!
another_list[2]()  # 'Hello from foo()!'
```

## Important list functions

### The append() function

Use this function to add a single element to the end of a list. This function works in $O(1)$, constant time.

```
list = [1, 3, 5, 'seven']

print(list)

list.append(9)

print(list)
```

### The insert() function

Inserts elements to the list. Use it like list.insert(index, value). It works in O(n) time. The following use of list.insert(0,2) inserts the element 2 at index 0.

```
list = [1, 3, 5, 'seven']

list.insert(0, 2)

print(list)
```

### The remove() function

Removes the given element at a given index. Use it like list.remove(element). It works in $O(n)$ time. If the element does not exist, you will get a runtime error as in the following example.

```
list = [1, 3, 5, 'seven']

print(list)

list.remove('seven')

print(list)

list.remove(0)
```

```
print(list)
```

### The pop() function

Removes the element at given index. If no index is given, then it removes the last element. So list.pop() would remove the last element. This works in O(1). list.pop(2) would remove the element with index 2, i.e., 5 in this case. Also, popping the kth intermediate element takes $O(k)$ time where $k<n$.

```
list = [1, 3, 5, 'seven']

print(list)

list.pop(2)

print(list)
```

### The reverse() function

This function reverses the list. It can be used as list.reverse() and takes $O(n)$ time.

```
list = [1, 3, 5, 'seven']

print(list)

list.reverse()

print(list)
```

## What is Slicing?

Accessing and modifying several elements from objects such as lists/tuples/strings requires using a for loop in most languages. However, in Python, you can use square brackets and a colon to define a range of elements within a list that you want to access or ‘slice’.

list[start:end]

Here start and end indicate the starting and ending index of a list that is desired to be accessed. You can print these values, reinitialize them, and execute mathematical functions on them.

## Slice Notation Examples

The following examples use slicing to perform various operations on lists.

### Example 1: Indexing elements of a list

List elements can be indexed and printed as in the following code example:

```
list = list(range(10))

print(list)  # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

print(list[0:4])  # 0, 1, 2, 3
```

Also, note that it is not necessary to specify the last or the first index explicitly, you can simply leave the end or start index blank respectively.

- list[start:] means all numbers greater than start uptil the range
- list[:end] means all numbers less than end uptil the range
- list[:] means all numbers within the range

```
list = list(range(10))

print(list[3:])  # 3, 4, 5, 6, 7, 8, 9

print(list[:3])  # 0 1 2

print(list[:])  # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

### Example 2: Stepped Indexing

List[start:end:end]

```
list = list(range(10)) # define a range of values 0

print(list[0:9:2])  # 0, 2, 4, 6, 8

print(list[9:0:-2])  # 9, 7, 5, 3, 1
```

**line 2** prints every second value of the list starting from the beginning

**line 3** prints every second value of the list starting from the end

#### Example 3: Initializing list elements #

You can add/modify the contents of a list by specifying a range of elements that you want to update and setting it to the new value:

arr[start: end] = [list, of, New, values]

```
x = list(range(5))

print(x)  # 0, 1, 2, 3, 4

x[1:4] = [45, 21, 83]

print(x)  # 0, 45, 21, 83, 4
```

The 1:4 in the square brackets means that the elements at positions 1, 2 and 3, up to but not including position 4, would be set to new values, i.e., [45, 21, 83].

Note that in Python, range counts up to the second index given but never hits the index itself. So in this case, the 4th index, i.e., the element 4 is not replaced.

#### Example 4: Deleting elements from a list

The del keyword is used to delete elements from a list. In the following example, all the elements at even-numbered indices are deleted.

```
list = list(range(10))

print(list)  # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

del list[::2]

print(list)  # 1, 3, 5, 7, 9
```

**Line 3** uses the `del` function. Here, the empty start and end indices refer to 0 and length of the list by default, whereas 2 is the step size.

### Example 5: Negative Indexing

We can use negative numbers to begin indexing the list elements from the end. For example, to access the fifth-last element of a list, we use:

```
list[-5]
```

```
list = list(range(10))

print(list)

print(list[4:-1])  # 4, 5, 6, 7, 8
```

#### Example 6: Slicing in Strings#

We can also use slicing techniques on strings since strings *are* lists of characters! (well, technically, they're *arrays* of characters, but we'll get to that in a bit!) For example, the string "somehow" can be broken down into two strings like:

```
my_string = "somehow"

string1 = my_string[:4]

string2 = my_string[4:]

print(string1, string2)
```

## Introduction to Arrays

In Python, an array is just an ordered sequence of *homogeneous* elements. In other words, an array can only hold elements of one datatype. Python arrays are basically just wrappers for C arrays. The type is constrained and specified at the time of creation.

## Initializing Arrays

Python arrays are initialized using the array library:

import array

New_array = array.array(‘type’, [list])

## Array slicing

Array slicing is done in exactly the same way as list slicing is done. Look at the following example:

```
import array

initializer_list = [2, 5, 43, 5, 10, 52, 29, 5]

number_array = array.array('i', initializer_list)

print(number_array[1:5])  # 2nd to 5th

print(number_array[:-5])  # beginning to 3rd

print(number_array[5:])  # 6th to end

print(number_array[:])   # beginning to end
```

### Changing or adding array elements

Arrays are mutable; their elements can be changed in the same way as list elements. Have a look at the following coding widget.

```
import array

integers = array.array('i', [1, 2, 3, 5, 7, 10])

# changing first element

integers[0] = 0

print(integers)  # array('i', [0, 2, 3, 5, 7, 10])

# changing 3rd to 5th element

integers[2:5] = array.array('i', [4, 6, 8])

print(integers)     # Output: array('i', [0, 2, 4, 6, 8, 10])
```

Just as with lists, we can add one item to the end of an array using the append() method or add several items using the extend() method.

```
import array

numbers = array.array('i', [1, 2, 3])

numbers.append(4)

print(numbers)  # array('i', [1, 2, 3, 4])

# extend() appends iterable to the end of the array

numbers.extend([5, 6, 7])

print(numbers)     # array('i', [1, 2, 3, 4, 5, 6, 7])
```

You can concatenate two arrays using + operator.

```
import array

odd = array.array('i', [1, 3, 5])
```

```
even = array.array('i', [2, 4, 6])

integers = array.array('i')   # creating empty array of integer

integers = odd + even

print(integers)
```

### How do you remove/delete elements?

To delete one or more items from an array, use the del statement as with lists.

```
import array

integer_array = array.array('i', [1, 2, 3, 3, 4])

del integer_array[2]  # removing third element

print(integer_array)  # Output: array('i', [1, 2, 3, 4])

del integer_array  # deleting entire array

print(integer_array)  # Error: array is not defined
```

We can use the remove(val) method to remove the given item or pop(index) to remove an item at the given index. The remove(val) method removes the first element that is equal to val in the array.

```
import array

integer_array = array.array('i', [10, 11, 12, 12, 13])

integer_array.remove(12)

print(integer_array)   # array('i', [10, 11, 12, 13])

print(integer_array.pop(2))   # Output: 12

print(integer_array)   # array('i', [10, 11, 13])
```

## Lists vs Arrays

Python lists are very flexible and can hold completely heterogeneous arbitrary data, but they use a lot more space than Python arrays. Each list contains pointers to a block of pointers, each of which in turn points to a full Python

object. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Arrays lack this flexibility but are much more efficient for storing and manipulating data.

The differences between the two largely exist because of the aforementioned backend implementation. Arrays in Python are implemented just like C arrays, with a pointer pointing to the first element of the array with the rest existing contiguously in the memory.

# Lecture 5: Computational Thinking

This is a set of tools and a framework used for thinking through a bunch of problems.

Other ways of thinking: Critical thinking, Design thinking, Strategic thinking, Creative thinking.

## How do you solve problems with a computer?

$$People\ -Program-> Computers\ -ToSolve-> Problems$$

- But… Writing the computer program is not the first step.
- You have to engage in some thinking and problem solving of your own before you can write the computer program.

$$People\ -Develop-> Algorithm\ -Implemented-> Program -> Computer\ -solves-> Problem$$

## Computational Thinking Framework

We can think through the following questions to solve a problem.

1. Have we identified a problem that is “solvable” with a computer?
2. Can we break our problem down into smaller sub-problems?
3. Can we see familiar patterns or characteristics that can lead us to potential solutions?
4. Can we identify non-essential aspects of our problem that we can ignore?

## Problem Identification

- What is the input and output?
- Is it solvable by computers?
- Some knowledge about the context.
- Constraints : Scale, Time and budget
- How would we know that we have solved the problem?
- You need to think about the following:
  - Topic: What is the big problem you are trying to solve? Can you state this in a problem statement? Is the problem statement too broad or vague?
  - Data: What information or data do you have to contribute to a problem-solving approach? What other information or data might you need?
  - Feasibility: Can you solve the problem given what you know? Is this problem solvable with a computer?

## Decomposition

> Yard by yard life is hard. Inch by inch life is cinch.

- You have a problem statement… can we break our big problem down into smaller sub-problems that are more straightforward to solve?
- Once you decompose your big problem into smaller and smaller sub-problems, you can start to think about how you can solve those smaller, specific sub-problems.
- We have specific tools for specific problems.

```
   P
  / \
 s   s
```

```
 / \
s   s
```

## Pattern Recognition

- Can we see familiar patterns or characteristics that can lead us to potential solutions?
- Think about…
  - Have you seen solutions to similar problems previously that we can use here? Can you think of similar problems that have been solved before that could help you address these problems?
  - How is this problem that same or different from other problems you have identified or addressed?

## Abstraction

- Can we identify non-essential aspects of our problem that we can ignore?
- Think about…
  - Asking whether some parts of the problem are less relevant to a successful outcome? Are there any extraneous details?
  - Trying too much can be tiring.

# Iteration 1

Problem Identification

Help my grandmother with her mental illness.
We have identified that our grandmother is acting differently for sometime. She is forgetting things like names of people. She 70 year old and she has no history of mental illness. We don't know what problem she is having.
Her health should improve. We should know what problem(s) she is facing. We should know if we need to follow some treatment. We need to follow the treatment.

To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?

1. Identifying what has changed.
2. Finding the right kind of medical health practitioner for her case.
3. Find the most suitable Dr.
4. Book an appointment and consult
5. Follow the treatment (If needed) and monitor

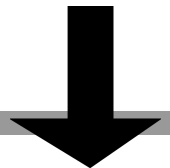
Pattern Recognition (Are there related solutions to draw on?)

1. For identifying what has changed, you can list down all the things that are different.
2. You can consult you family doctor / Practo
3. List all the potential options, have a list of criteria, find one that fits.
4. You can call the hospital
5. Create a process, get a nurse

Abstraction (How would you abstract this problem?)

1. We don't need to talk to friends about this

# Graphic Organizer



# Iteration 2

Problem Identification

How to make treatment for alzheimer's disease more accessible?
Who are you solving this problem for, i.e, who is you ideal patient?
What is the purchasing power of your patient?
What would it mean to help?
How will the system you are building to solve this problem look like?

Start a company that can make treatment for alzheimer's disease more accessible for old age middle class patients in rural areas.

To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?)

1. Identify the gaps in the current system
2. Develop a business plan
3. Iteratively execute on the plan

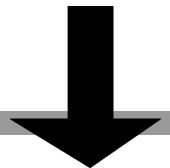
Pattern Recognition (Are there related solutions to draw on?)

1. To identify the gaps in the market we can do market research.

Abstraction (How would you abstract this problem?)

1. Solving it for rural population is too difficult due to logistic issues. Let's scope the problem down.

# Graphic Organizer



# Iteration 3

Problem Identification

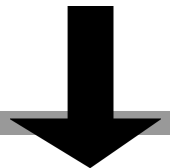
To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?)

Pattern Recognition (Are there related solutions to draw on?)

Abstraction (How would you abstract this problem?)

# Graphic Organizer



# Iteration 4

Problem Identification

To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?)

Pattern Recognition (Are there related solutions to draw on?)

Abstraction (How would you abstract this problem?)

# Graphic Organizer

# Iteration 5

Problem Identification

To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?)

Pattern Recognition (Are there related solutions to draw on?)

Abstraction (How would you abstract this problem?)

# Graphic Organizer

# Iteration 6

Problem Identification

To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?)

Pattern Recognition (Are there related solutions to draw on?)

Abstraction (How would you abstract this problem?)

# Graphic Organizer

# Iteration 7

Problem Identification

To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?)

Pattern Recognition (Are there related solutions to draw on?)

Abstraction (How would you abstract this problem?)

# Graphic Organizer

# Iteration 8

Problem Identification

To set up your identified problem

Decomposition (How would you break down your problem into sub-problems?)

Pattern Recognition (Are there related solutions to draw on?)

Abstraction (How would you abstract this problem?)

# Graphic Organizer

# Iteration 9

Problem Identification

To set up your identified problem

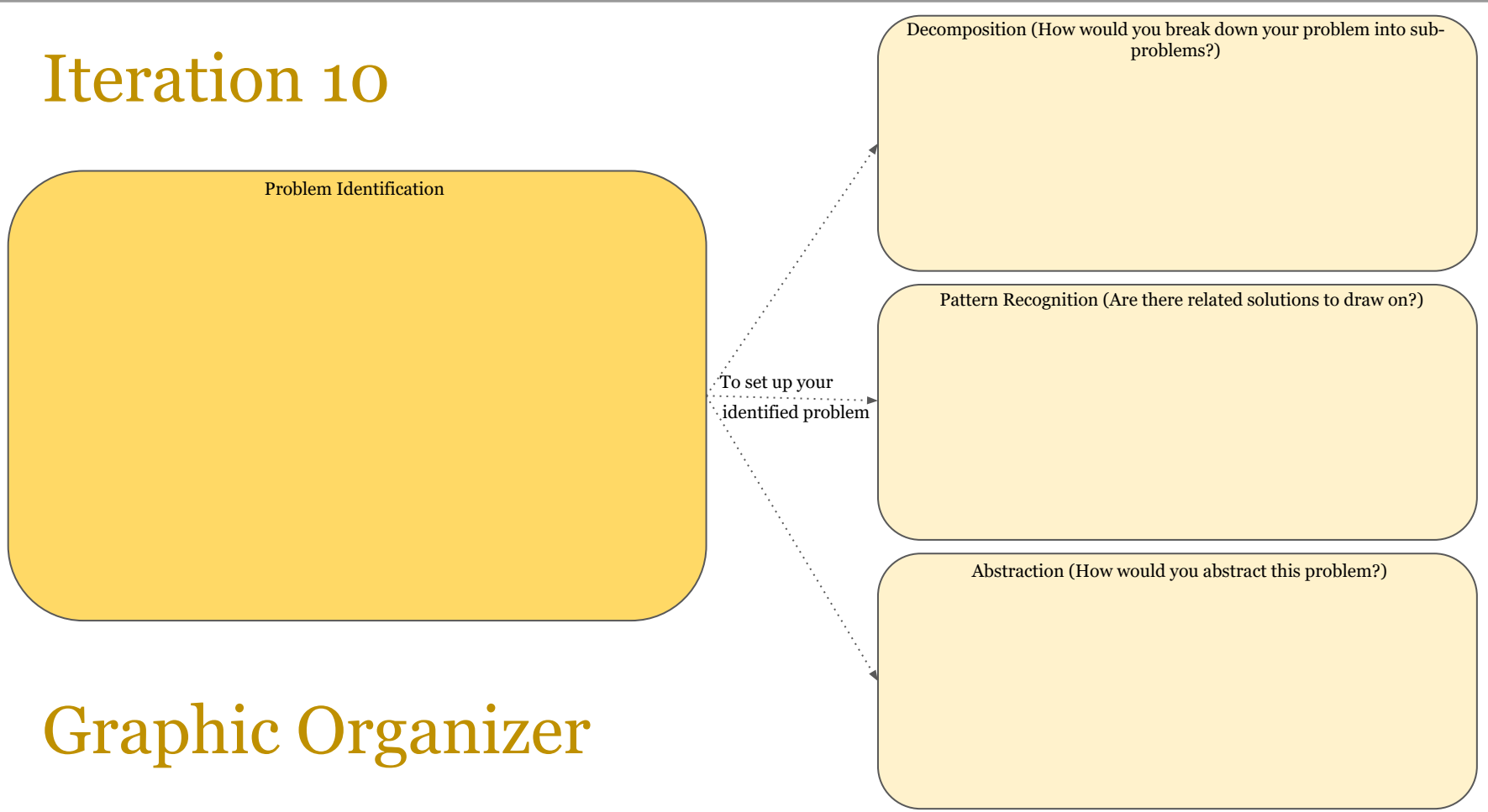
Decomposition (How would you break down your problem into sub-problems?)

Pattern Recognition (Are there related solutions to draw on?)

Abstraction (How would you abstract this problem?)

# Graphic Organizer

# Iteration 10




# Graphic Organizer

# Lecture 6: Arrays

## What is an array?

- Student Responses:
  - Contiguous memory block of integers/characters/strings.
  - It is a mutable data structure where all the elements are of same type.
  - Elements are stored at indexes and they can be accessed in $O(1)$ given the index.
  - For static C type arrays, insertions and deletions can be difficult. This is not the case with dynamic arrays like, Vectors in C++, Lists in Python, and ArrayList in Java.
  - The elements are stored in no specific relationship to one another. [1, 2, 3] , [3, 1, 19]. This is not the case with other data structures like heaps and BST.
  - Insertion/deletion at any index is allowed and take linear time.
  - Update command on any index is constant time.
- An array is ordered sequence of elements that allow for constant time access and update operations based on the indexes.

```
vector<int> nums;
nums.push_back(1); // O(1)'
nums.push_back(3);
nums.push_back(4); // nums = [1, 3, 4]
sort(nums.begin(), nums.end())
nums[0] = 5; // nums = [5, 3, 4]
```

```
nums = [1, 3, 4, "Hello"]
nums[0] = 5
nums.sort()
nums.append(7)
```

```
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add(1)
nums.add(3)
```

```
nums.add(4)
System.out.println("Nums : " + nums);
```

## Why is the time of index based access O(1)?

Imagine a chunk of memory. If I give you a memory address, the hardware promises to fetch the content at that memory address in constant time. Array stores the elements in a chunk of memory. For languages like C++, Java, all elements are of same type. Suppose you are storing 32 bit integers. Suppose first element is at memory location 1000 bit. Can you tell me where the 5th element will start? First element will be from 1000 to 1032, second element will be from 1032 to 1064, fifth element will start at 1128. The element at index i in this array will be at 1000 + 32 * i.

## What is linear time vs constant time?

Linear time: n = 1, t = 10; n = 2, t = 20; n = 3, t = 30

Constant time: n = 1, t = 5; n = 2, t = 5; n = 3, t = 5

## Why is the append operation O(1)?

……..[—100—]…….

……..[—101—]…….

```
nums = [1, 2, 3]
target = [2, 4, 6]
nums.append(5)
# .......[| 1 | 2 | 3 |][| 2 | 4 | 6 |]........
```

### Computational thinking:

1. Problem Identification
    a. We want to append an element at the end of an array.
2. Decomposition
    a. We need to increase the size of memory of our array.
3. Pattern Recognition

a. We can allocate more memory at a different address. And copy the contents of the pervious array to that location. This will take O(n).

b. int nums[5] = {1, 2, 3, 4, 5}
   int nums2[6]
   for(int i = 0; i < 5; i++) nums2[i] = nums[i]
   nums2[5] = 6

c. We can allocate a memory of size double the given memory.

| Size | Memory | Time to Append |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 4 | 3 |
| 4 | 4 | 1 |
| 5 | 8 | 5 |
| 6 | 8 | 1 |
| 7 | 8 | 1 |
| 8 | 8 | 1 |
| 9 | 16 | 9 |
| 10 | 16 | 1 |
| 11 | 16 | 1 |
| 12 | 16 | 1 |
| 13 | 16 | 1 |
| 14 | 16 | 1 |
| 15 | 16 | 1 |
| 16 | 16 | 1 |
| | | |
| Element | Sum | |
| 1 | 1 | |
| 2 | 3 | |
| 4 | 7 | |
| 8 | 15 | |
| 16 | 31 | n = 2^log(n) |
| 32 | 63 | |

| | | |
|---|---|---|
| 64 | 127 | |
| 128 | = 2 * Element - 1 | |
| | | |
| | | |

d. What is the time complexity of this approach?

i. For inserting n element we will take time = 1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 + 1 + 1 + …

$$Time = 1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 + 1 + 1 + 1...$$

1. For n element it is taking O(n), thus amortised it is O(1) per insertion.

$$Time = n + (1 + 2 + 4 + 8 + ... + 2^{logn-1})$$

$$Time = n + 2^{logn} - 1$$

$$Time = n + n = 2n = O(n)$$

4. Abstraction

a. Why are we increasing by only one element for memory?

i. We can also double the size of the array.

# Lecture 7: Arrays (Continued)

## Remove Even Integers

### Problem

Implement a function that removes all the even elements from a given list. Name it `remove_even(nums)`.

Input: A list with random integers

Output: A list with only odd integers

### Decomposition

- Iterate over each element
- Choose odd element / Remove even integers

### Pattern Recognition

- Time
  - Iteration over each element: `O(n)`
  - Remove even integers: `O(n)` * `O(n)`
    - [1, 2, 3, 4, 5] → remove(3) → [1, 2, 4, 5]
  - Choosing odd integers: `O(1)` * `O(n)`

```
newNums = [num for num in nums if num % 2 == 1]
```

## Merge Two Sorted Lists

### Problem

Implement a function `mergeArrays(int arr1[], int arr2[], int arr1Size, int arr2Size)` which merges two sorted arrays into another sorted array.

Input: Two sorted arrays with their sizes

Output: A merged sorted array consisting of all elements of both input arrays.

- We want all the elements of `arr1` and `arr2` in a new array and the elements should be in sorted order.

## Decomposition

- Combining two arrays into a third array.
- Sorting the third array.

## Pattern Recognition

- Time:
  - Combining two arrays : `O(n1 + n2)`
  - Sorting: `O((n1 + n2)log(n1 + n2))`
- Both arrays are sorted:
  - The first element of each array is the smallest in the array.
- The result has to be sorted:
  - The first element of the result array should be the smallest of all elements.

## Abstraction

- Where are we using the fact that the given arrays are sorted?

## Code

```
int * mergeArrays(int arr1[], int arr2[], int arr1Size, int arr2Size) {
  int * arr3 = new int[arr1Size + arr2Size];
  int index1 = 0, index2 = 0, index3 = 0;

  while (index1 < arr1Size && index2 < arr2Size) {
    if(arr1[index1] < arr2[index2]){
      arr3[index3++] = arr1[index1++];
```

```
    } else {
      arr3[index3++] = arr2[index2++];
    }
  }

  while (index1 < arr1Size)
    arr3[index3++] = arr1[index1++]

  while (index2 < arr2Size)
    arr3[index3++] = arr2[index2++]

  return arr3;
}
```

# Maximum Sum Sublist

Given an integer list, return the maximum sublist sum. The list may contain both positive and negative integers and is unsorted.

Input: a list

Output: a number that is the maximum subarray sum

### Subarray:

Subarray is a continuous chunk taken from an array.

For example: [1, 2, 3] is a subarray of [0, 1, 2, 3, 4], whereas [0, 2, 3] is not a subarray.

A subarray is uniquely identified by the start and end index.

## Decomposition

- We don’t know the length of the subarray.
- We don’t know the sum of each subarray.
- Iterate through each subarray → $O(n^2)$ time
- Calculate the sum of each subarray → $O(n)$ time
- Compare the sums to find the maximum

## Pattern Recognition

- Observation:
  - A subarray can further be divided into smaller subarrays.
  - And if a subarray has a negative sum prefix we can remove the prefix to get a larger sum.
    - Ex: [-1, 2, -3, 4, 5] = 7 → on removing [-1, 2, -3] we get [4, 5] → 9
- If I have to calculate the maximum subarray sum ending at index `i`, what do I need to do?
  - Look at the maximum subarray sum ending at index `i - 1` if it is negative ignore it, else add it `nums[i]` and the sum is the maximum subarray sum ending at index `i`.
- Babu Rao / Raju
  - Babu Rao
    - Debt, Clown, Drunked
    - Every day he wakes up and there is a queue of moneylenders trying to get their money back from Babu Rao.
    - Rao → [1, -2, 3, 4, -5, -6]
    - Babu Rao starts with an empty pocket.
    - If somebody gives him money he keeps it.
    - If somebody wants money and he has more than the want, he gives it.
    - If he has less than the want, he gives whatever he has. And start again with the empty pocket.
  - Raju
    - Mischievous Character
    - He keeps track of Babu Rao's pocket.
    - And reports the maximum money he had at any point.

```
def find_max_sum_sublist(moneyLenders):
  if len(moneyLenders) == 0:
    return 0

  babuRao = moneyLenders[0]
  maxSum = moneyLenders[0]

  for moneyLender in moneyLenders[1:]: # O(n)
    babuRao = max(moneyLender, babuRao + moneyLender)
    maxSum = max(maxSum, babuRao)

  return maxSum
```

# Singly Linked Lists (SLL)

Python does not have a built-in linked list structure, as it isn't required after the introduction of lists. However, knowledge about linked lists can be very useful in coding interviews! In this section, we will cover the basic behavior of linked lists and show how to implement them in Python syntax. This will also enhance your understanding on the back-end functionality of Python.

To implement a linked list, we need the following two classes:

- Class Node
- Class LinkedList

## Node Class

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next_element = None
```

**Explanation :** The code above provides a basic definition of a Node class structure with two data members: one to hold the data and the other one to store a reference to the next element. **Data** is the value you want to store in the node. Think of it as the value at a specific index in a list. The data type can range from *string* or *integer* to a user-defined class.
The **Pointer** refers us to the ext node in the list. It is essential for connectivity.

## LinkedList Class

```
class LinkedList:
    def __init__(self):
        self.head_node = None
```

**Explanation :** The linked list itself is a collection of Node objects which we defined above. To keep track of the list, we need a pointer to the first node in the list. This is where the principle of **head_node** comes in. The head doesn't contain any data and only points to the beginning of the list. This means that, for any operations on the list, we need to traverse it from the head to reach the desired node in the list.

As we can see in the implementation, when the list is first initialized it has no nodes, so the head is set to None.

# Basic Linked List Operations :

## is_empty()

The basic condition for our linked list to be considered empty is that there are no nodes in the list. i.e, **head** should point to None object.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next_element = None


class LinkedList:
    def __init__(self):
        self.head_node = None
    def is_empty(self):
        #Check whether the head is None
        if self.head_node is None:
            return True
        else:
            return False
    #Better way
```

```
    def is_empty(self):
        return self.head_node is None
```

## Insertion at the head in SLL

This type of insertion means that we want to insert a new element as the first element of the list. As a result, the newly added node will become the head, which in turn will point to the previous first node.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next_element = None

class LinkedList:
    def __init__(self):
        self.head_node = None
    def insert_at_head(self, data):
        #Create a new node containing your specified value
        temp_node = Node(data)
        #The new node points to the same node as the head
        temp_node.next_element = self.head_node
        self.head_node = temp_node
        #return the new list
        return self.head_node
    def print_list(self):
        if self.is_empty():
            print(“List is empty”)
```

```
            return False
        temp = self.head_node
        while temp.next_element is not None:
            print(temp.data, end = "->")
            temp = temp.next_element
        print(temp.data, "-> None")
        return True
linkedlist = LinkedList()
limit = 10
for i in range(1, limit):
    linkedlist.insert_at_head(i)
linkedlist.print_list()


#Output
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> None
```

**print_list()** simply starts at the head node, and iterates through the nodes using temp and displays their value. Our iteration ends when **temp.next_element** is **None**, which means that we've reached the last node in the list.

**insert_at_head()** takes an integer value as data and inserts it just after head to make it the first element of the list.

This function follows these steps to insert a new node:

- Create a new Node object with the given value, called temp_node.
- Make the next_element of temp_node equal tot he original head.
- temp_node will become the next_element of head

# SLL deletion

The deletion combines the principles of **insertion** and **search.** It uses the search functionality to find the value in the list.

There are three basic delete operations for linked lists

1. Deletion at the head
2. Deletion by value
3. Deletion at the tail

```
class Node:
        def __init__(self, data):
                self.data = data
                self.next_element = None

class LinkedList:
        def __init__(self):
                self.head_node = None
        def get_head(self):
                return self.head_node
        def is_empty(self):
                return self.head_node is None
        def insert_at_head(self, dt):
                temp_node = Node(dt)
                temp_node.next_element = self.head_node
                self.head_node = temp_node
                return self.head_node
        def print_list(self):
                if self.is_empty():
```

```
            print(“List is empty”)
            return False
        temp = self.head_node
        while temp.next_element is not None:
            print(temp.data, end = “->”)
            temp = temp.next_element
        print(temp.data, “-> None”)
        return True
def search(linkedlist, value):
        #Start from the first element
        current_node = linkedlist.get_head()
        #Traverse the list till you reach the end
        while current_node:
            if current_node.data == value:
                #Value found
                return True
            current_node = current_node_next_element
        #Value not found
        return False
def delete_at_head(linkedlist):
        #Get head as first_element of the list
        first_element = linkedlist.get_head()
        #If the list is not empty, then link head to the next_element of
#first_element
        if first_element is not None:
            linkedlist.head_node = first_element.next_element
```

```
                first_element.next_element = None
        return


linkedlist = LinkedList()
limit = 10
for i in range(limit + 1):
        linkedlist.insert_at_head(i)
linkedlist.print_list()
delete_at_head(linkedlist)
delete_at_head(linkedlist)
linkedlist.print_list()


#Output
10 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 -> None
8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 -> None
```

# Doubly Linked List (DLL)

The only difference between doubly and singly linked lists is that in DLLs each node contains pointers for both the previous and the next node. This makes the DLLs **bi-directional**.

To implement this in code, we simply need to add a new member to the Node class.

```
class Node:
        def __init__(self, data):
                self.data = data
                self.next_element = None
                self.previous_element = None
```

The **previous_element** pointer has been introduced to store information about the preceding node.

## Impact on deletion

The addition of a backwards pointer significantly improves the searching process during deletion as you don't need to keep track of the previous node. Let's rewrite the delete method for DLL

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next_element = None
        self.previous_element = None

class LinkedList:
    def __init__(self):
        self.head_node = None
    def get_head(self):
        return self.head_node
    def is_empty(self):
        return self.head_node is None
    def insert_at_head(self, dt):
        temp_node = Node(dt)
        if self.is_empty():
            self.head_node = temp_node
            return self.head_node
        temp_node.next_element = self.head_node
        self.head_node.previous_element = temp_node
        self.head_node = temp_node
```

```
            return self.head_node

        def print_list(self):
            if self.is_empty():
                print("List is empty")
                return False
            temp = self.head_node
            while temp.next_element is not None:
                print(temp.data, end = "->")
                temp = temp.next_element
            print(temp.data, "-> None")
            return True
        def delete(linkedlist, value):
            deleted = False
            if linkedlist.is_empty:
                print("List is empty")
                return deleted
            current_node = linkedlist.get_head()
            if current_node.data is value:
                #point head to the next element of the first element
                linkedlist.head_node = current_node.next_element
                if current_node.next_element !=  None and \
current_node.next_element.previous_element != None:
                    #point the next element of the first element to None
                    current_node.next_element.previous_element = None
                    #Both links have been changed
                    deleted = True
```

```
                    print(str(current_node.data) + “Deleted”)
                return deleted
        #Traversing/Searching for node to Delete
        while current_node:
            if value is current_node.data:
                if current_node.next_element:
                    #Link the next node and previous node to each other
                    prev_node = current_node.previous_element
                    next_node = current_node.next_element
                    prev_node.next_element = next_node
                    next_node.previous_element = prev_node
                    #Previous node pointer was maintained in SLL
                else:
                    current_node.previous_element.next_element = None
                deleted = True
                break
            current_node = current_node.next_element
        if deleted is False:
            print(str(value) + “is not in the List!”)
        else:
            print(str(value) + “is deleted”)
        return deleted
limit = 10
linkedlist = LinkedList()
for i in range(limit + 1):
        linkedlist.insert_at_head(i)
```

```
linkedlist.print_list()
delete(linkedlist, 5)
linkedlist.print_list()
delete(linkedlist, 0)
linkedlist.print_list()

#Output
10 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 -> None
5 Deleted!
10 -> 9 -> 8 -> 7 -> 6 -> 4 -> 3 -> 2 -> 1 -> 0 -> None
0 Deleted!
10 -> 9 -> 8 -> 7 -> 6 -> 4 -> 3 -> 2 -> 1 -> None
```

# Lecture 8: Linked Lists

Array = List of elements stored in a continuous chunk of memory. In an array, all elements are independent.

## Linked Data Structures

Graphs, Trees, and Heaps are linked data structures as there are links between the elements

## Linked Lists Implementation

For linked lists, you need two classes:

- Node class
  - Data
  - Pointer
- LinkedList class
  - Head

```
class Node:
  def __init__(self, data: int):
    self.data = data
    self.next_element = None

class LinkedList:
  def __init__(self):
    self.head_node = None
```

```
Head -> Node(Data1, Pointer1-)-> Node(Data2, Pointer2-)-> None
```

```
total = 0
for num in nums[1:]:
  total += num
  for element in elements[1:]:
    total += element

# Rewrite
total = 0
for num in nums[1:]:
  total += num
  total += sum(elements[1:])
```

```
class Node {
public:
  int data;
  Node* nextElement;

  Node(int data){
    this->data = data;
    this->nextElement = nullptr;
  }
};

class LinkedList {
  Node* head;

  LinkedList() {
    this->head = nullptr;
  }
}
```

# Linked List Operations

- `insertAtTail(data)`
- `insertAtHead(data)`
- `delete(data)`
- `deleteAtHead()`

- `search(data)`
- `getHead()`
- `isEmpty()`

```
#includ "LinkList.h"

bool LinkedList::isEmpty() {
  return head == nullptr;
}
```

```
class LinkedList:
  def __init__(self):
    self.head_node = None

  def isEmpty():
    return self.head_node == None
```

# Insertion in Linked List

## Problem Identification

Given a value `Data`, and a linked list `LinkedList`, we need to insert `Data` into `LinkedList`. Where in the `LinkedList` should we insert `Data`? We can insert `Data` at the following places:

1. Head
2. Tail
3. Insert at the N-th index

Let's focus on `insertionAtHead(data).`

## Decomposition

- Create a `Node` from data.
- Insert the new node before the head.
  - The `new_node.next_element` should point to the head
  - And make the `new_node` the head of the linked list.

```
#include  "LinkedList.h"

void LinkedList::insertAtHead(int value) {
  Node * newNode = new Node(value);
  newNode -> nextElement = head;
  head = newNode;
}
```

# Lecture 9: Linked List (Continues)

## How to take notes?

### Problem Definition

I want to take notes on different topics at different times.

1. I would like to write about four different topics in parallel.
2. In each of the topics, I want the ability to append new notes between previously written notes.
3. You need an easy reading experience.
    a. What does that mean?
        i. Ease of access and storage in one place.

Eg:

1. Maths in an art. And it should be taught as an art.
2. By extension data structures and algorithms is also an art.
    a. Because it is a field of maths.
3. Of course, data structures should also be taught as an art.

## Decomposition

- Find a notebook. Get a pencil and write.
- Solution for 1. Have four notebooks for four subjects.
- Solution for 1. Divide the notebook into four different parts.
- Solution for 2. Is to leave some space after each point or topic.
- Solution for 2. Use extra material like sticky notes.
- Solution for 2. Use page numbers.

> What is the difference between page number and the page itself? Same as the pointer to a variable and the variable itself.

- Solution for 3. Use Lose Pages notepads and save them topic-wise in a file.

# Linked List as a Spring Binder

Whenever faced with a linked list problem think of a spring binder. And see yourself operating on a spring binder.

# Insertion at Tail

## Problem

We need to insert a new object at the end of the linked list.

Input: A linked list and an integer value.

Output: The updated linked list with the value inserted at the end.

## Pattern

- Traverse to the tail of the linked list.
- Create a new node with the value.
- Make the tail point to the new node.

# Reverse a Linked List

You have to define the reverse function, which takes a singly linked list and produces the exact opposite list, i.e., the links of the output linked list should be reversed.

Input: A singly linked list.

Output: The reversed linked list.

Example: Input : 0 → 1 → 2 → 3

Output: 3 → 2 → 1 → 0

```
def reverse(head):
  current = head
  prev = None
  nextNode = head.next_element
  while current is not None:
    nextNode = current.next_element
    current.next_element = prev
    prev = current
    current = nextNode
  return prev
```

0 → 1 → 2 → 3 →4 | Current = 0, prev = None

←0 1 → 2 → 3 →4 | Current = 1, prev = 0

←0←1 2 → 3 →4 | Current = 2, prev = 1

←0←1←2 3 →4 | Current = 3, prev = 2

## Recursion

- Call reverse on `current.next` and add head at the tail of the returned list.

```
def reverse(head):
  if head.next_element is None:
    return head
  rest = reverse(head.next_element)
  head.next_element = None
  insertAtTail(rest, head) # O(n)
  return rest
```

- Time complexity: $O(n^2)$

## Smart Solution

```
def reverse(head):
  answer = None
  while head is not None:
    # Remove the head from the list
    temp = head
    head = head.next_element
    temp.next_element = None
    # Put the head into the new list
    temp.next_element = answer
```

```
      answer = temp
    return answer
```

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \text{None} \mid \text{None}$

$2 \rightarrow 3 \rightarrow 4 \rightarrow \text{None} \mid 1 \rightarrow \text{None}$

$3 \rightarrow 4 \rightarrow \text{None} \mid 2 \rightarrow 1 \rightarrow \text{None}$

$4 \rightarrow \text{None} \mid 3 \rightarrow 2 \rightarrow 1 \rightarrow \text{None}$

$\text{None} \mid 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow \text{None}$

# Lecture 10: Linked List (Continues) And Stacks

## Recap

1. How to prepare for STEP?
2. Complexity
3. Computational thinking - 4-step framework
4. Arrays
5. Linked List

## Detect Loop in a Linked List

### Problem Statement

By definition, a loop is formed when a node in your linked list points to a previously traversed node.

1 → 2 → 3 → 4 → 2 → 3 → 4 …

```
1 -> 2 -> 3 -> 4
     ^_________|
```

You must implement the `detect_loop()` function which will take a linked list as input and deduce whether or not a loop is present.

### Decomposition

- We need to traverse the linked list.
- We need to find when we are repeating nodes.

### Pattern Recognition

- To detect nodes flag them.
    - Store the pointer to nodes in a separate array and keep checking their frequency.
        - Iterate over the linked list: O(n)
            - Check the frequency of the pointer of the current node. O(n)
            - If the frequency is 1, return true.
            - Less make the frequency as 1.
        - O(n^2)
    - If we use a set, the checking becomes O(1).
        - Then time becomes O(n) for `detect_loop`.
        - Space complexity is O(n)
    - 0, 1, 2,… → “abc”, “bcd”, “xyz”, “fde”
    - [0, 5, 2, 1, 3]
- We can have two-pointers, viz., slow and fast iterating through our linked list.
    - If the pointers meet at some point, the linked list has a loop.
    - If there is no loop, the fast pointer will reach the end of the linked list.
    - slow = slow.next | fast = fast.next.next

## Abstraction

- We don’t need the frequency, we can just check if an element is present in the list.

---

# Stacks

Arrays are a continuous chunk of memory that stores elements of the same type. This provides us with random access to the elements, i.e., we can access any element in O(1) if we know its index.

**What is the price we pay for having random access?**

Due to the property of continuous storage, random insertion and deletion from an array is costly O(n).

Eg: Given an array [1, 2, 5, 9, 12, 3, 6], inserting 7 before 9 will require us to shift 9, 12, 3, and 6 to the right by one step.

**What workaround can we have for this problem?**

Solutions:

1. We can use a linked list. But we lose O(1) random access.
2. We can use a binary search tree. That allows you insertion, deletion, and access in O(log(n)).
3. We can restrict access to all but the last element of the array. We only allow the program to access the last element of the array. That means insertion, detection, and access all happen at the end of the array in O(1) time.

# Definition

> Stack is a linear data structure where all the insertions and deletions are restricted to one end. ~ Nihitha

> Stack is a Last in First out data structure. You fetch the element inserted at the last.

Suppose, there is a stack of books, i.e., books are kept on top of each other. Second, hand booksellers keep books in stacks. You can only remove the book from the top of the stack. And you can only put the book at the top of the stack without tearing all the other books.

Books in the library are stored in rows which is similar to an array. You can access any book easily.

# Operations on Stacks

| Function | What does it do? |
| --- | --- |
| push(element) | Inserts an element at the top |

| Function | What does it do? |
| --- | --- |
| pop() | Removes an element from the top and returns it |
| peek() or top() | Returns the top element of the stack |
| isEmpty() | Returns a boolean 1 if the stack is empty |
| size() | Returns the size of the stack |

# What is a Stack?

Stack is a container, in which we can add items and remove them. Only the top of this *container* is open, so the item we put in first will be taken out last, and the items we put in last will be taken out first. This is called the **last in first out (LIFO) ordering**.

A real-life example of a Stack can be a pile of books placed in a vertical order. So, to get the book that's somewhere in the middle, you need to remove all the books placed on top of it.

## What is Stack used for?

A stack is one of the most fundamental data structures. Its implementation is very simple, yet it can be used to solve complex problems!

There are many computer algorithms like **Depth First Search** and **Expression Evaluation Algorithm**, etc., which are dependent on stacks to run perfectly. Stacks are used for the below actions:

- To backtrack to the previous task/state, e.g., in a recursive code.
- To store a partially completed task, e.g., when you are exploring two different paths on a *Graph* from a point while calculating the smallest path to the target.

## Implementation of Stack

Typical stack contains following methods:

- push(element)
- pop()

- is_empty()
- peek()
- size()

Before we take a look at these methods one by one, let's construct a class of Stack, and create an instance.

```
class Stack:
        def __init__(self):
                self.stacklist = []

stack_obj = Stack()
```

Now let's  implement the methods :

```
class Stack:
        def __init__(self):
                self.stacklist = []
                self.stack_size = 0

        def is_empty(self):
                return self.stack_size == 0

        def peek(self):
                if self.is_empty():
                        return None
                return self.stack_list[-1]

        def size(self):
                return self.stack_size

if __name__ == "__main__":
        stack_obj = Stack()
        print("is_empty(): " + str(stack_obj.is_empty()))
        print("peek(): " + str(stack_obj.peek()))
```

```
    print("size(): " + str(stack_obj.size()))
```

OUTPUT :

```
is_empty(): True
peek(): None
size(): 0
```

Let's implement push and pop method

```
class Stack:
    def __init__(self):
        self.stacklist = []
        self.stack_size = 0

    def is_empty(self):
        return self.stack_size == 0

    def peek(self):
        if self.is_empty():
            return None
        return self.stack_list[-1]

    def size(self):
        return self.stack_size

    def push(self, value):
        self.stack_size += 1
        self.stack_list.append(value

    def pop(self):
        if self.is_empty():
            return None
```

```
            self.stack_size -= 1
            return self stack.list.pop()

if __name__ == “__main__”:
    stack_obj = MyStack()
    limit = 5
    print(“Pushing elements into the stack”)
    for i in range(limit):
        print(i)
        stack_obj.push(i)
    print(“is_empty(): “ + str(stack_obj.is_empty()))
    print(“peek(): “ + str(stack_obj.peek()))
    print(“size(): “ + str(stack_obj.size()))

    print(“Popping elements from the stack”)
    for x in range(limit):
        print(stack_obj.pop())
    print(“is_empty(): “ + str(stack_obj.is_empty()))
    print(“peek(): “ + str(stack_obj.peek()))
    print(“size(): “ + str(stack_obj.size()))
```

OUTPUT:

```
Pushing elements into the stack
0
1
2
3
4
is_empty(): False
peek(): 4
size(): 5
Popping elements from the stack
4
3
```

```
2
1
0
is_empty(): True
peek(): None
size(): 0
```

# What is a Queue?

Similar to Stack, Queue is another linear data structure that stores the elements in a sequential manner. The only significant difference between Stack and Queue is that instead of using the LIFO principle, Queue implements the FIFO method, which is short for **First in First Out**. Elements enter from one end (back) and leave from the other (front).

Queues are slightly trickier to implement compared to stacks, as we have to keep track of both ends of the array. The elements are inserted from the back and removed from the front.

A perfect real-life example of Queue is a line of people waiting to get a ticket from the booth.

## What are Queues used for?

Most operating systems also perform operations based on a Priority Queue—a kind of queue that allows operating systems to switch between appropriate processes. They are also used to store packets on routers in a certain order when a network is congested. Implementing a cache also heavily relies on queues. We generally use queues in the following situations:

- We want to prioritize something over another
- A resource is shared between multiple devices (e.g., Web Servers and Control Units)

## Types of Queues

There are three common types of queues which cover a wide range of problems:

- *Linear Queue*
- *Circular Queue*
- *Priority Queue*
- *Double-ended Queue*

Since we already discussed Linear Queue, let's discuss remaining types :

### Circular Queue :

Circular queues are almost similar to linear queues with only one exception. As the name itself suggests, circular queues are circular in structure which means that both ends are connected to form a circle. Initially, the front and rear part of the queue point to the same location. Eventually they move apart as more elements are inserted into the queue. Circular queues are generally used in:

- Simulation of objects
- Event handling (do something when a particular event occurs)

### Priority Queue :

In priority queues, all elements have a priority associated with them and are sorted such that the most prioritized object appears at the front and the least prioritized object appears at the end of the queue. These queues are widely used in most operating systems to determine which programs should be given more priority.

### Double-Ended Queue :

The double-ended queue acts as a queue from both ends(back and front). It is a flexible data structure that provides `enqueue` and `dequeue` functionality on both ends in *O(1)*. Hence, it can act like a normal linear queue if needed. Python has a built-in `deque` class that can be imported from the `collections` module. The class contains several useful methods such as `rotate`.

## Implementation of Queues :

A typical Queue must contain the following standard methods:

- enqueue(element)
- dequeue()
- is_empty()
- front()
- rear()

A typical Queue must contain the following standard methods: Let's construct a class Queue before implementing the methods and create an object.

```
class Node:
        def __init__(self, data):
                self.data = data
                self.next_element = None
                self.previous_element = None

class DoublyLinkedList:
        def __init__(self):
                self.head = None
                self.tail = None
                self.length = 0
```

```
def get_head(self):
        if self.head != None:
                return self.head.data
        else:
                return False

def is_empty(self):
        return self.head is None

def insert_tail(self, element):
        temp_node = Node(element)
        if self.is_empty():
                self.head = temp_node
                self.tail = temp_node
        else:
                self.tail.next_element = temp_node
                temp_node.previous_element = self.tail
                self.tail = temp_node
        self.length += 1
        return temp_node.data

def remove_head(self):
        if self.is_empty():
                return False
        node_to_remove = self.head
        if self.length == 1:
                self.head = None
                self.tail = None
        else:
                self.head = node_to_remove.next_element
                self.head.previous_element = None
                node_to_remove.next_element = None
        self.length -= 1
        return node_to_remove.data
```

```
        def tail_node(self):
                if self.head != None:
                        return self.tail.data
                else:
                        return False

        def __str__(self):
                val = ""
                if self.is_empty():
                        return val
                temp = self.head
                val = "[" + str(temp.data) + ","
                temp = temp.next_element

                while temp.next_element:
                        val = val + str(temp.data) + ","
                        temp = temp.next_element
                val = val + str(temp.data) + "]"
                return val

class Queue:
        def __init__(self):
                self.items = DoublyLinked_List()

queue = Queue()
```

Now let's implement working of methods

```
class Node:
        def __init__(self, data):
                self.data = data
                self.next_element = None
                self.previous_element = None

class DoublyLinkedList:
        def __init__(self):
```

```
        self.head = None
        self.tail = None
        self.length = 0



def get_head(self):
        if self.head != None:
                return self.head.data
        else:
                return False

def is_empty(self):
        return self.head is None

def insert_tail(self, element):
        temp_node = Node(element)
        if self.is_empty():
                self.head = temp_node
                self.tail = temp_node
        else:
                self.tail.next_element = temp_node
                temp_node.previous_element = self.tail
                self.tail = temp_node
        self.length += 1
        return temp_node.data

def remove_head(self):
        if self.is_empty():
                return False
        node_to_remove = self.head
        if self.length == 1:
                self.head = None
                self.tail = None
        else:
                self.head = node_to_remove.next_element
```

```
                self.head.previous_element = None
                node_to_remove.next_element = None
            self.length -= 1
            return node_to_remove.data


    def tail_node(self):
        if self.head != None:
            return self.tail.data
        else:
            return False

    def __str__(self):
        val = ""
        if self.is_empty():
            return val
        temp = self.head
        val = "[" + str(temp.data) + ","
        temp = temp.next_element

        while temp.next_element:
            val = val + str(temp.data) + ","
            temp = temp.next_element
        val = val + str(temp.data) + "]"
        return val

class Queue:
    def __init__(self):
        self.items = DoublyLinkedList()

    def is_empty(self):
        return self.items.length == 0

    def front(self):
        if self.is_empty():
            return None
```

```
            return self.items.get_head()

        def rear(self):
            if self.is_empty():
                return None
            return self.items.tail_node()

        def size(self):
            return self.items.length()

if __name__ == "__main__":
        queue_obj = Queue()

print("is_empty(): " + str(queue_obj.is_empty()))
print("rear(): " + str(queue_obj.rear()))
print("front(): " + str(queue_obj.front()))
print("size(): " + str(queue_obj.size()))
```

OUTPUT :

```
is_empty(): True
rear(): None
front(): None
size(): 0
```

Let us implement enqueue and dequeue methods

```
class Node:
        def __init__(self, data):
            self.data = data
            self.next_element = None
            self.previous_element = None

class DoublyLinkedList:
        def __init__(self):
            self.head = None
```

```
        self.tail = None
        self.length = 0

    def get_head(self):
        if self.head != None:
            return self.head.data
        else:
            return False

    def is_empty(self):
        return self.head is None

    def insert_tail(self, element):
        temp_node = Node(element)
        if self.is_empty():
            self.head = temp_node
            self.tail = temp_node
        else:
            self.tail.next_element = temp_node
            temp_node.previous_element = self.tail
            self.tail = temp_node
        self.length += 1
        return temp_node.data

    def remove_head(self):
        if self.is_empty():
            return False
        node_to_remove = self.head
        if self.length == 1:
            self.head = None
            self.tail = None
        else:
            self.head = node_to_remove.next_element
            self.head.previous_element = None
            node_to_remove.next_element = None
        self.length -= 1
```

```
            return node_to_remove.data


    def tail_node(self):
            if self.head != None:
                    return self.tail.data
            else:
                    return False

    def __str__(self):
            val = ""
            if self.is_empty():
                    return val
            temp = self.head
            val = "[" + str(temp.data) + ","
            temp = temp.next_element

            while temp.next_element:
                    val = val + str(temp.data) + ","
                    temp = temp.next_element
            val = val + str(temp.data) + "]"
            return val

class Queue:
    def __init__(self):
            self.items = DoublyLinkedList()

    def is_empty(self):
            return self.items.length == 0

    def front(self):
            if self.is_empty():
                    return None
            return self.items.get_head()

    def rear(self):
```

```
            if self.is_empty():
                return None
            return self.items.tail_node()

        def size(self):
            return self.items.length()

        def enqueue(self, value):
            return self.items.insert_tail(value)

        def dequeue(self):
            return self.items.remove_head()

        def print_list(self):
            return self.items.__str__()

if __name__ == “__main__”:
        queue_obj = Queue()
        print(“queue.enqueue(2)”)
        queue_obj.enqueue(2)
        print(“queue.enqueue(4)”)
        queue_obj.enqueue(4)
        print(“queue.enqueue(6)”)
        queue_obj.enqueue(6)
        print(“queue.enqueue(8)”)
        queue_obj.enqueue(8)
        print(“queue.enqueue(10)”)
        queue_obj.enqueue(10)
        queue_obj.print_list()
        print(“is_empty(): “ + str(queue_obj.is_empty()))
        print(“front(): “ + str(queue_obj.front()))
        print(“rear(): “ + str(queue_obj.rear()))
        print(“size(): “ + str(queue_obj.size()))
        print(“Dequeue(): “ + str(queue_obj.dequeue()))
        print(“Dequeue(): “ + str(queue_obj.dequeue()))
        print(“queue.enqueue(12)”)
```

```
        queue_obj.enqueue(12)
        print("queue.enqueue(14)")
        queue_obj.enqueue(14)

while queue_obj.is_empty() is False:
        print("Dequeue(): " + str(queue_obj.dequeue()))
print("is_empty(): " + str(queue_obj.is_empty()))
```

OUTPUT:

```
queue.enqueue(2)
queue.enqueue(4)
queue.enqueue(6)
queue.enqueue(8)
queue.enqueue(10)
is_empty(): False
front(): 2
rear(): 10
size(): 5
Dequeue(): 2
Dequeue(): 4
queue.enqueue(12)
queue.enqueue(14)
Dequeue(): 6
Dequeue(): 8
Dequeue(): 10
Dequeue(): 12
Dequeue(): 14
is_empty(): True
```

# Lecture 12: Graphs

## Fundamental Concepts of Graphs

A graph is a collection of elements that are related to each other. The elements are known as nodes or vertices. And the relationships are known as edges.

$$G = \{V, E\}$$

Where $G$ is the graph, $V, E$ are the sets of vertices and edges respectively.

### Real World Examples

| Graph | Vertices | Edges |
|---|---|---|
| Instagram | Users | Follows |
| Google Maps | Places | Paths |
| Food Chain | Species | Eats |
| Communication Network | Devices | Connections |
| Web Pages | HTML Page | Hyperlink, <a> |
| Array | Elements | {} |
| Classroom | Students | Blames |
| Variable | Variable | {} |
| Electric Circuits | Components | Wires |
| Family Tree | People | Children-Parent |
| Covid Crisis | People | Infector-Infectee |
| Molecule | Atoms | Chemical Bonds |
| Linked List | Nodes | Pointers |
| This Table | Cells | Right or Left Boarder |

## ▼ Covid Crisis

https://www.figma.com/file/LYorsBuz7yrr6eoCSE0JLg/Covid-Graph?type=design&node-id=0%3A1&mode=design&t=N9eiVyPLpTSdRnvY-1

- Nodes are the people and the edges are the relationships between infector and infectee.
- What are the types of nodes you have in this graph?
  - People are either infected or non-infected.
- What is common among all the infected people?
  - All infected people form a connected sub-graph.
- What will be your guess about the distance between two random infected persons in this graph?
  - What do you mean by distance?
    - Distance is the number of edges on the shortest path between two nodes.
  - Guesses → 1, 5000, total-geographic-area ÷ 7B, 152
  - The most accurate guess is 1 out of these guesses.
  - A slightly better guess would be 6 or 7.
- A fact:
  - Between you and any other person on the planet, say X32.
  - You know someone, who knows someone, who knows someone, who knows someone, who knows someone, who knows X32.
  - You are connected to everyone with a path of maximum length 6.

# Types of Graphs

- Why would you want to have different types of graphs?
  - To represent different types of data.

    - Consider a molecule and a Linkedlist. Edges in a Linkedlist are different from edges in a molecule.
        - Linkedlist can best be represented by one-directional edges whereas a molecule can best be represented by bi-directional edges.
    - Now consider Google Maps and Food Chain.
        - Google Maps' edges should be weighted whereas Food Chain edges need not be weighted.
- How can you differentiate between two graphs?
    - Graphs can be:
        - Weighted, Non-weighted
        - Directed, Undirected
        - Cyclic, Acyclic
        - Directed acyclic graphs, aka, DAGs
        - Connected, Not connected

# Graph Implementation

- Why do we need a standard graph implementation?
    - A standard makes the solutions to graph problems accessible to everyone.
    - A standard provides a common vocabulary to talk about the problems.
- What are the Graph Implementations we know?
    1. Adjacency list
    2. Adjacency matrix
- What is the Adjacency matrix?
    - The Adjacency matrix, say AdjMat is a 2D array of size $n \times n$, where $n$ is the number of nodes in the graph.
    - Here AdjMat[u][v] == 1 if there is an edge between u and v, it's zero otherwise.
- What would be a problem with Adjacency matrix?

- Imagine the face of Mark Zuckerberg, when he needs to create a user Graph of size 4B $\times$ 4B. That 16000 ZB.


- How can we have a better solution?
  -

## ▼ Example Question

- You are a traveling salesman in a country with many cities. A city may be connected to another city by a road or it may not be connected. The government is building roads between cities all the time. And you want to know whether you can move from one city to another at different times.

> https://www.figma.com/file/IvqbXmjK7EuPjZsZegKqdL/City-Roads?type=design&node-id=0%3A1&mode=design&t=EnG7aHdEIIE9Uy40-1

You will have to support two operations:

1. Connect(A, B): This creates a connection between A and B.

2. CanReach(A, B): This return true if you can reach B from A, and false other wise.

## Depth First Search (DFS) Algorithm

The idea of the depth-first search algorithm is to start from an arbitrary node of the graph and to explore as far as possible before coming back i.e. moving to an adjacent node until there is no unvisited adjacent node. Then backtrack and check for other unvisited nodes andrepeat the same process for them.

### Pseudo code of DFS Algorithm

```
DFS(visited, u):
  visited[u]=True
  Print u
  for each v in adj[u]:
    if(visited[v]=False)
      DFS(visited,v)
```

**DFS Algorithm Implementation**

```
  def DFS_helper(self, u, visited):

    visited.add(u)
    print(u)

    for v in self.adj[u]:
      if(v not in visited):
        self.DFS_helper(v, visited)

  def DFS(self, u):
    visited=set()
    self.DFS_helper(u,visited)
```

**Complexity Analysis of DFS Algorithm**

- **Time complexity:** Since we consider every vertex exactly once and every edge atmost twice, therefore time complexity is $O(V+2E) \simeq O(V+E)$, where $V$ is the number of vertices and $E$ is the numberof edges in the given graph.
- **Space Complexity:** $O(V)$, because we need an auxiliary visited array/set of size $V$.

## Breadth First Search | BFS

Breadth-First Search(also known as Breadth-First Traversal) is a traversal mechanism that start the search from a Node and then explore all of its neighbouring Vertices. Once these neighbouring vertices are explored, then we go ahead exploring all of their neighbouring vertices and so on.

### Pseudo code of Breadth-First Search

Create a Queue (say Q)
Mark Vertex V as Visited.
Put V into Queue

While Q is not empty
    Remove the head of Q (Let it be Vertex U)
    Iterate all Unvisited Neighbours of U
        Mark the neighbour as Visited
        Enqueue the Neighbour into Q.

### Implementation of Breadth-First Search Function

```
def bfs_traversal(self, vertex):

    visited = [False] * (max(self.graph) + 1)
    queue = []

    queue.append(vertex)
    visited[vertex] = True

    while queue:
      vertex = queue.pop(0)
      print (vertex, end = " ")

      for i in self.graph[vertex]:
        if visited[i] == False:
          queue.append(i)
          visited[i] = True
```

### Complexity Analysis of BFS

**Time Complexity Analysis:**

From each V, we iterate all of the other neighbour vertices i.e. at the other end of all of its edges, and the total edges that we can have in the graph is E. Then it means Breadth-First Search works in **O(E + V)** time.

**Space Complexity Analysis:**

Since both the Visited array and Queue can have a max size of V(equal to as many vertices),the overall space complexity will be **O(V)**.

# Lecture 14: Minimum Spanning Trees

| Done | Doing | To Do |
| --- | --- | --- |
| Introduction to Graphs | Minimum Spanning Tree | Shortest Path |
| Graph Representations - Adj List/Matrix | | Topological Sort |
| Graph Traversals | | |

## Minimum Spanning Tree

Suppose we are given a connected, undirected, weighted graph. The task is to find a minimum spanning tree T that minimises the sum of the weights of the tree edges.

```
graph LR
A((1)) ---7--- B((2))
A ---4--- C((3))
B ---1--- C
B ---5--- D((4))
C ---2--- D
A ---3--- D
```

The Minimum Spanning Tree would be:

```
graph LR
A((1,2))
B((2,2))
C((3,2))
D((4,2))
A ---1--- C
C ---2--- D
B ---5--- D
```

## Greedy Approach

Suppose you have the list of edges.

- Which edge will definitely be part of the MST?
    - The edge with the minimum weight.

- Why?
    - Imagine a spanning tree, T, without this minimum weight edge.
    - Let’s say the minimum weight edge connects vertex, u to v.
    - There will be a path from u to v in T. And this path will not contain the minimum weighted edge.
    - We can remove an edge from the this path and add the minimum weighted edge to get a smaller sum of weights.
- How can you extend this logic to create an algorithm?
    - Algorithm:
        - Sort the list of edges.
        - For each edge in the list
            - If the vertices of the edge are not connected in the Tree
                - Add the edge to the tree
- How are you going to remember if two vertices are connected?
    - Idea 1: We can have a Dictionary with key as a vertex and value as the list of all the vertices connected to it.
    - Idea 2: While running this algorithm, mark all the elements from a single component with a single number.
        - Vertices [1, 2, 3, 4] ⇒ Components [1, 1, 1, 2] ⇒ When you add the edge from 2 to 4, Components become [1, 1, 1, 1]
        - How will you check if two vertices are connected in this set up?
            - If two nodes are from the same component, they are connected.

> Read up about Union and Find.

# Problem

Link - Min Cost to Connect All Points - LeetCode

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        edges = self.getEdges(points)
```

```
        edges.sort(key = lambda x: x[2]) # O(n^2 log(n))
        return self.connectCost(len(points), edges)

    def connectCost(self, n, edges): # O(n^2)
        component = list(range(n)) # n == 4, [0, 1, 2, 3]
        cost = 0
        for edge in edges: # O(n^2)
            u, v = edge[0], edge[1]
            cu, cv = component[u], component[v]
            if cu != cv: # This will be true only n-1 times
                cost += edge[2]
                for i in range(n): # O(n)
                    if component[i] == cv:
                        component[i] = cu
        return cost

    def getEdges(self, points):
        answer = []
        for i in range(len(points)):
            for j in range(i + 1, len(points)):
                answer.append([i, j, self.getDistance(i, j, points)])
        return answer

    def getDistance(self, i, j, points):
        return abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j][1])
```

# Dijkstra's Algorithm | Shortest Path

```
from queue import PriorityQueue

class Graph:
    def __init__(self, num_of_vertices):
        self.v = num_of_vertices
        self.edges = [[-1 for i in range(num_of_vertices)] for j in range(num_of_vertices)]
        self.visited = []


    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
        self.edges[v][u] = weight
```

```
def dijkstra(graph, start_vertex):
    D = {v:float('inf') for v in range(graph.v)}
    D[start_vertex] = 0

    pq = PriorityQueue()
    pq.put((0, start_vertex))

    while not pq.empty():
        (dist, current_vertex) = pq.get()
        graph.visited.append(current_vertex)

        for neighbor in range(graph.v):
            if graph.edges[current_vertex][neighbor] != -1:
                distance = graph.edges[current_vertex][neighbor]
                if neighbor not in graph.visited:
                    old_cost = D[neighbor]
                    new_cost = D[current_vertex] + distance
                    if new_cost < old_cost:
                        pq.put((new_cost, neighbor))
                        D[neighbor] = new_cost
    return D
```

```
g = Graph(9)
g.add_edge(0, 1, 4)
g.add_edge(0, 6, 7)
g.add_edge(1, 6, 11)
g.add_edge(1, 7, 20)
g.add_edge(1, 2, 9)
g.add_edge(2, 3, 6)
g.add_edge(2, 4, 2)
g.add_edge(3, 4, 10)
g.add_edge(3, 5, 5)
g.add_edge(4, 5, 15)
g.add_edge(4, 7, 1)
g.add_edge(4, 8, 5)
g.add_edge(5, 8, 12)
g.add_edge(6, 7, 1)
g.add_edge(7, 8, 3)


#function call
D = dijkstra(g, 0)
print(D)
```

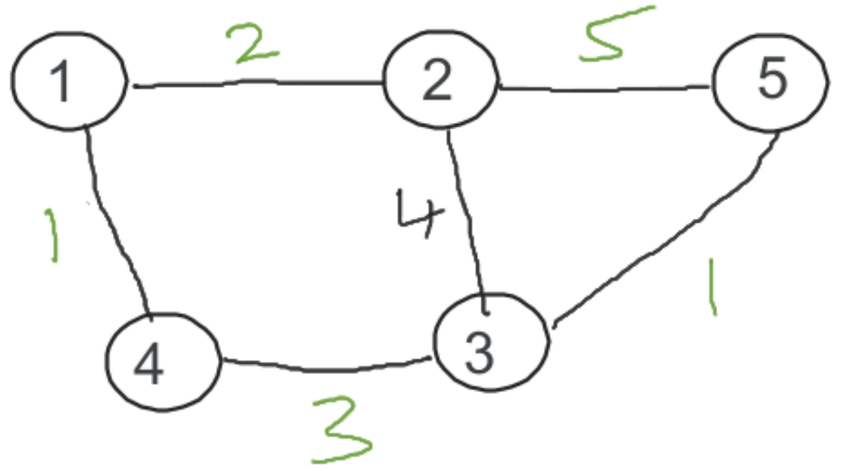
```
{0: 0, 1: 4, 2: 11, 3: 17, 4: 9, 5: 22, 6: 7, 7: 8, 8: 11}
```

```
1
```



✓ 0s completed at 5:56 PM

Undirected Weighted Graph:



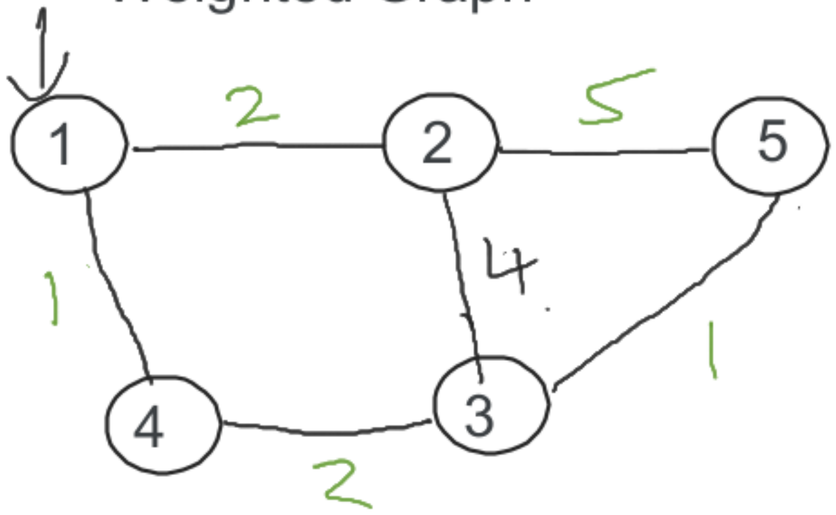

Source = 1

Adjacency list:

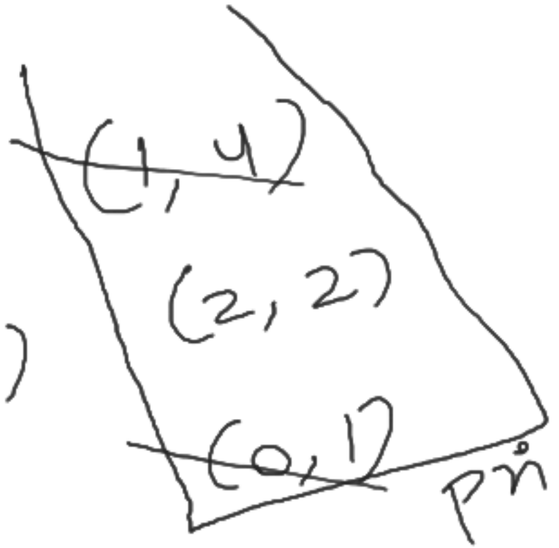
Node -> (Adj Node, Weight)

1 -> (2, 2), (4, 1)
2 -> (1, 2), (5, 5), (3, 4)
3 -> (4, 3), (5, 1), (2, 4)
4 -> (1, 1), (3, 3)
5 -> (2, 5), (3, 1)

Weighted Graph



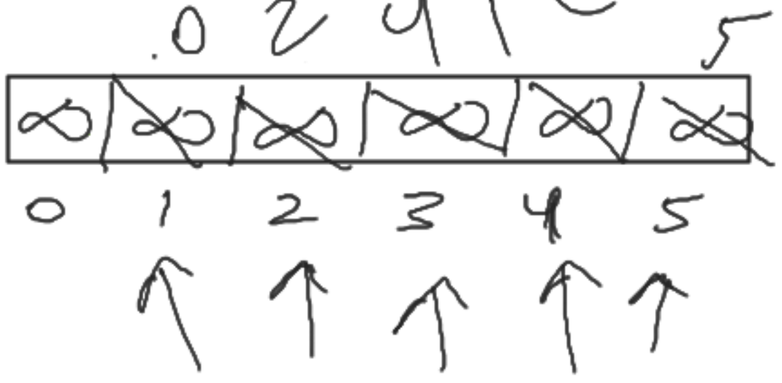

(dis, node)

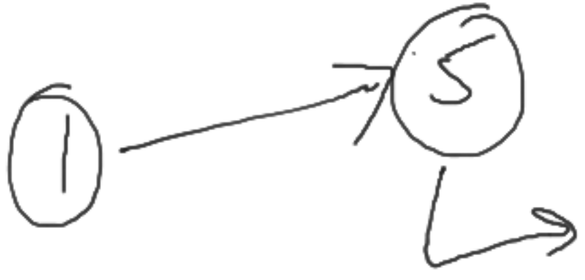



Priority Queue (minimum)

Source = 1

distances:




$2+5=7$

$2+4+1=7_{min}$

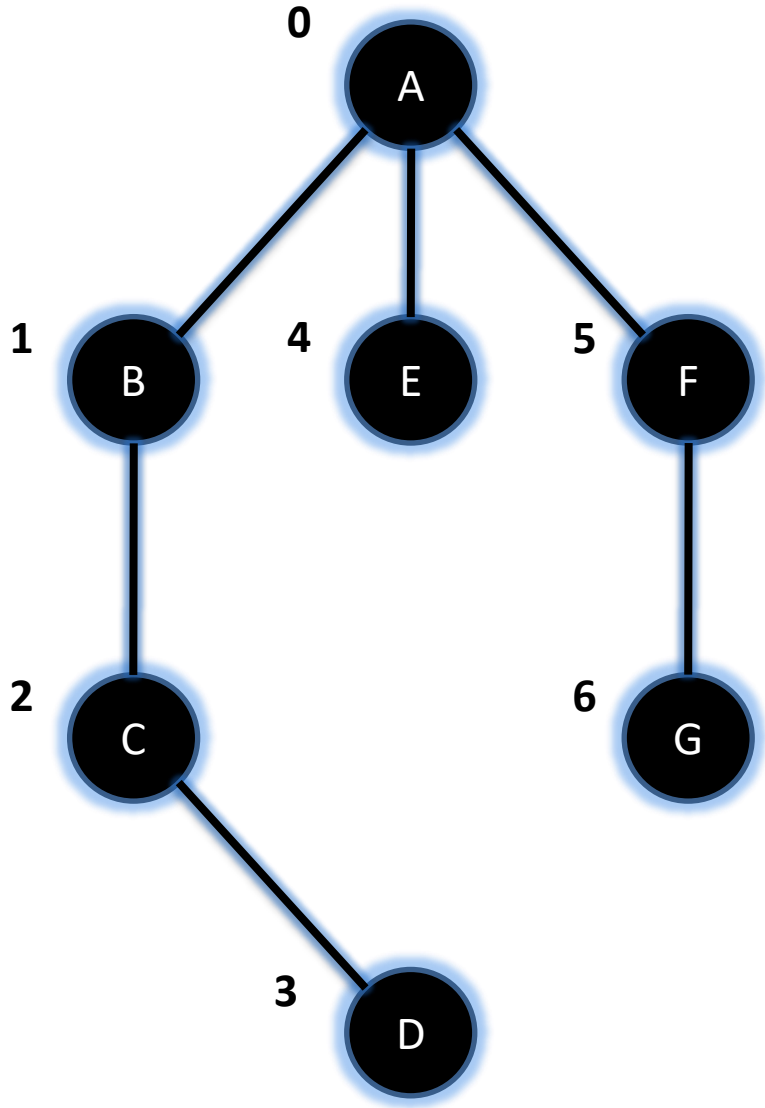
$1+3+1=5$

$1+3+4+5=13$

# GRAPHS

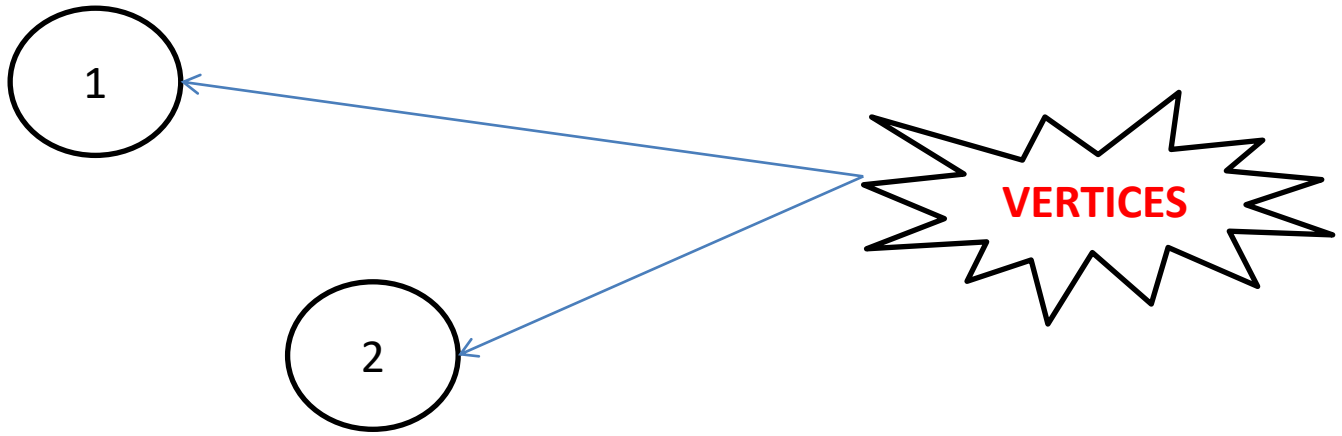
# Introduction

A graph is a **pictorial representation** of a set of objects connected by links.



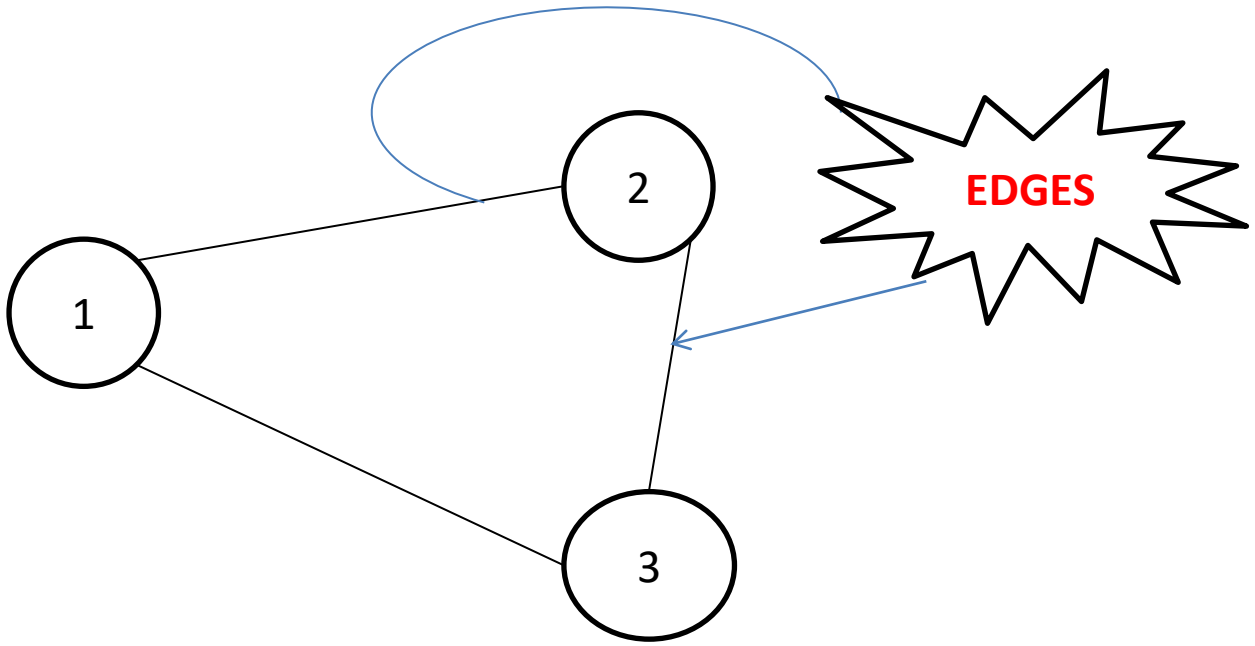
# Vertices

The  objects are represented by points termed as **vertices.**



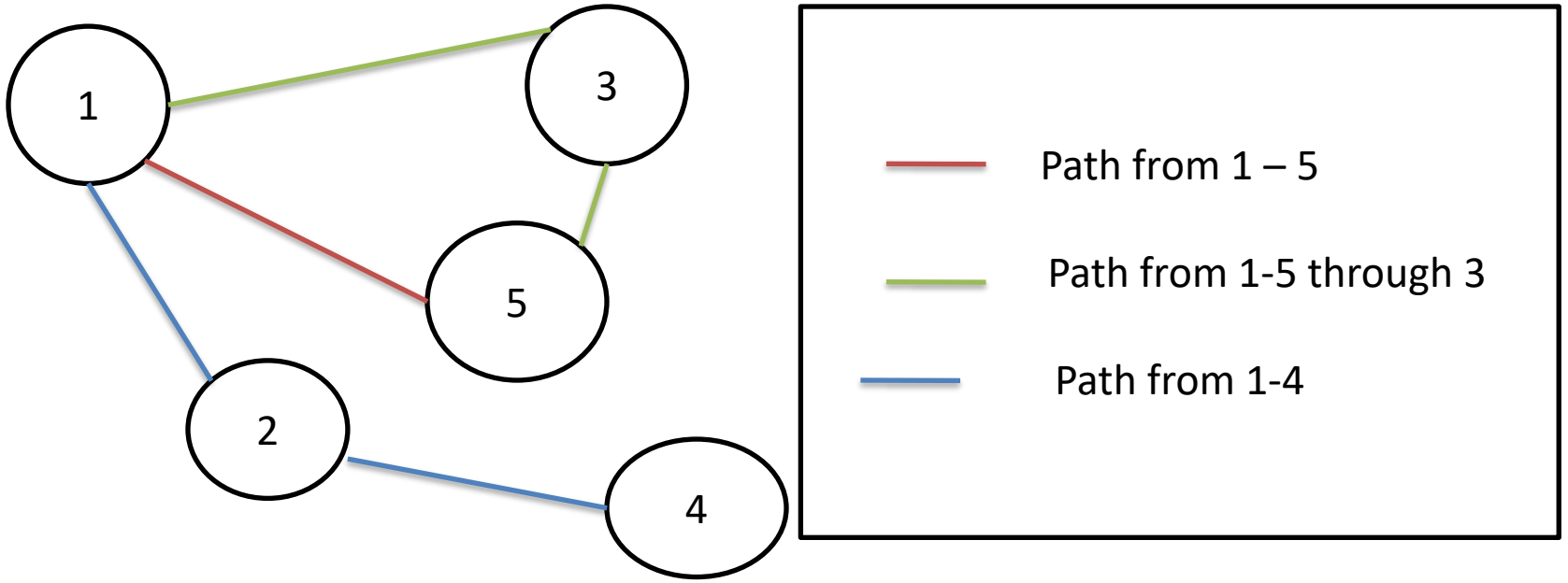
# Edges

The links that connect the vertices are called **edges**.



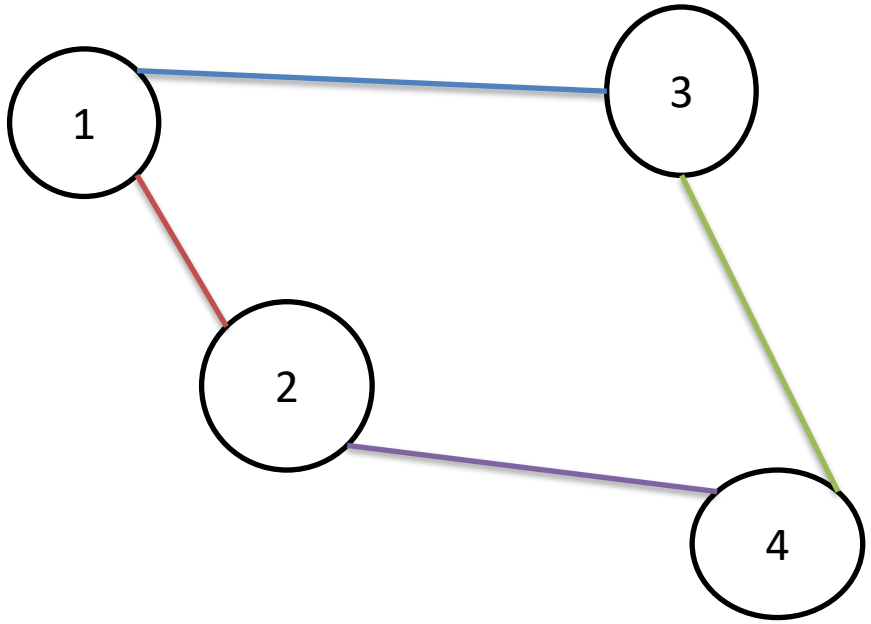
# Path

Path represents a **sequence of edges** between the two vertices.



# Adjacency

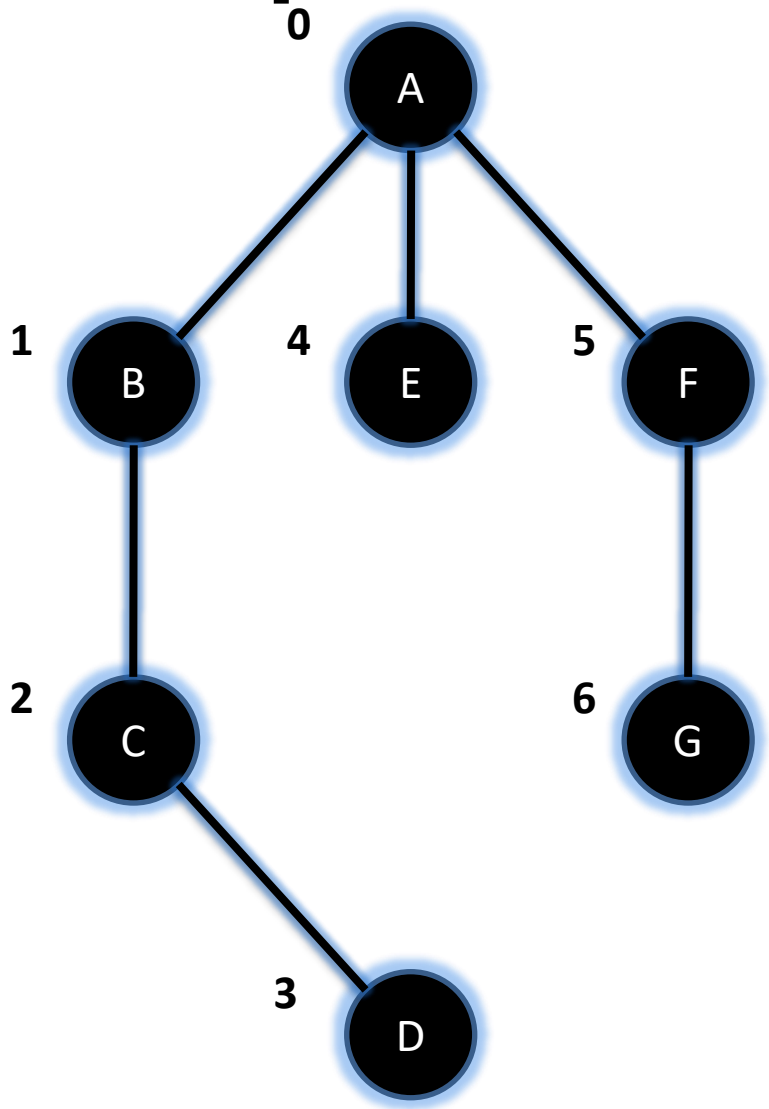
Two node or vertices are adjacent if they are **connected** to each other **through an edge**




(1,2) are adjacent.
(1,3) are adjacent.
(2,4) are adjacent.
(3,4) are adjacent

But, (2,3) are **not adjacent** as it is not connected by edges.

# Example For Graph




From the above example,

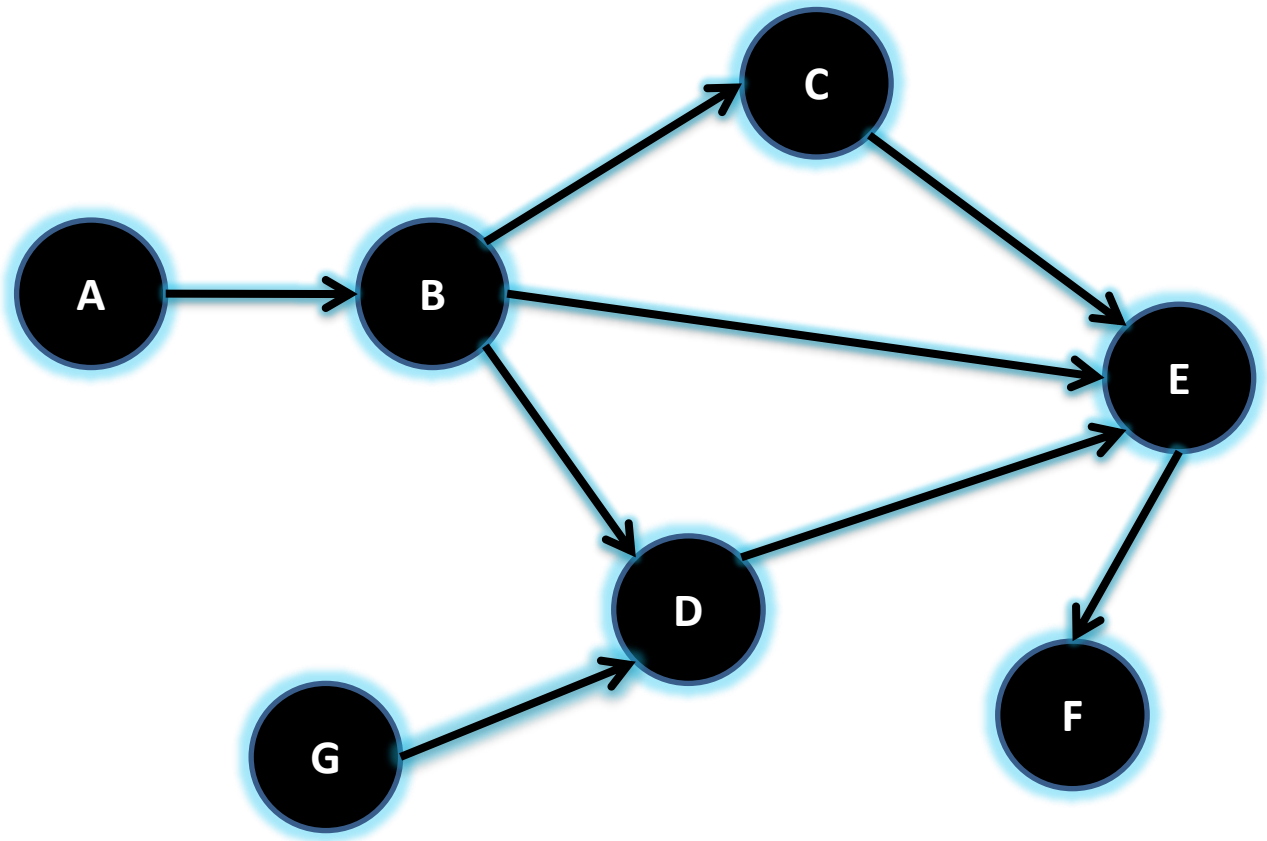
**VERTICES:** {A , B , C , D, E, F , G}.

**EDGES:** The lines from A to B, B to C, and so on represents edges.

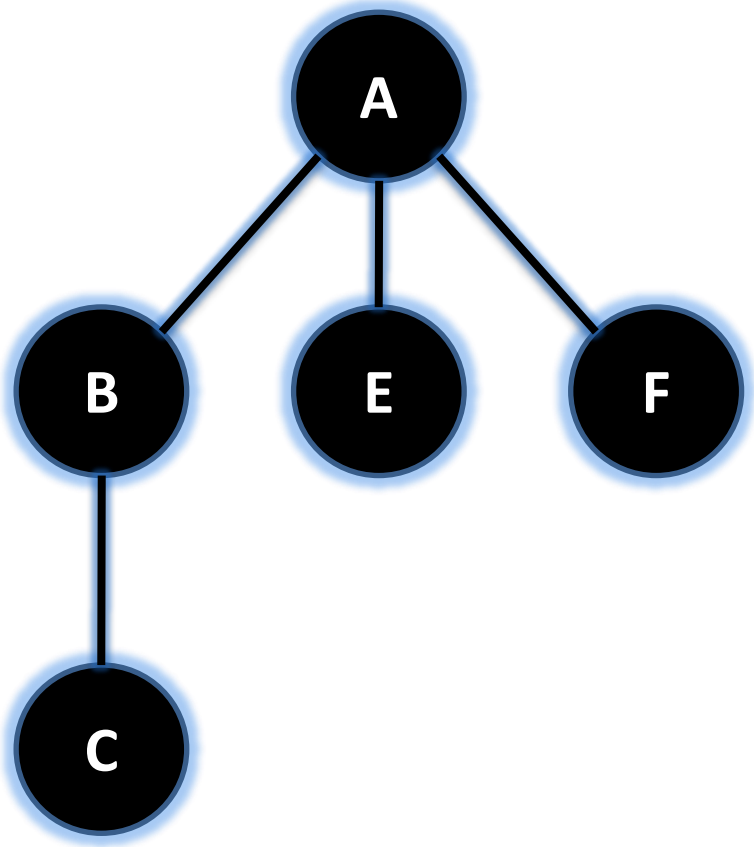
**PATH:** ABCD represents a path from A to D.

# Types of Graph

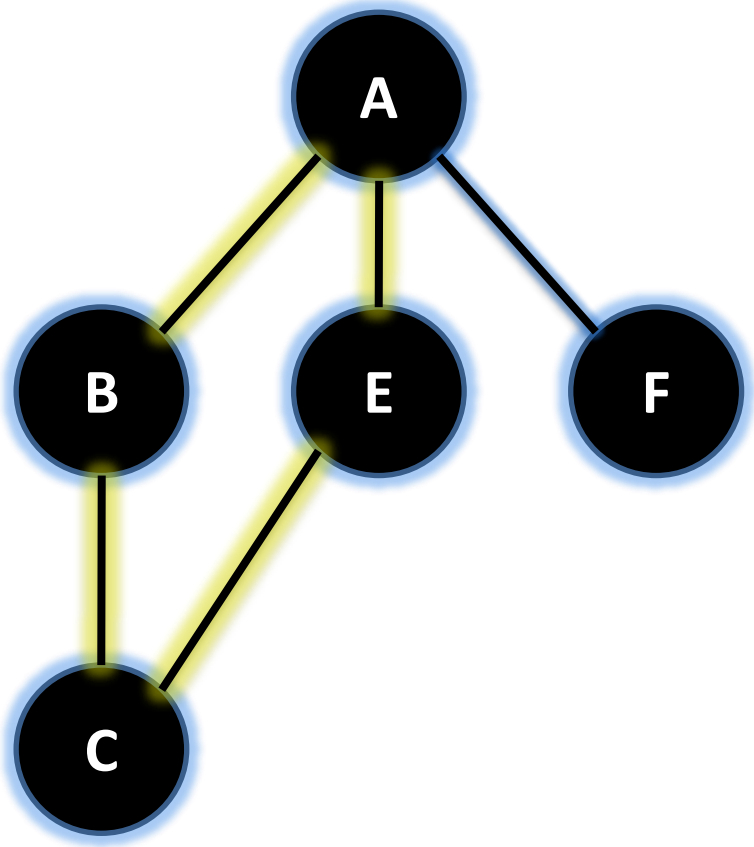
## A) DIRECTED GRAPH:



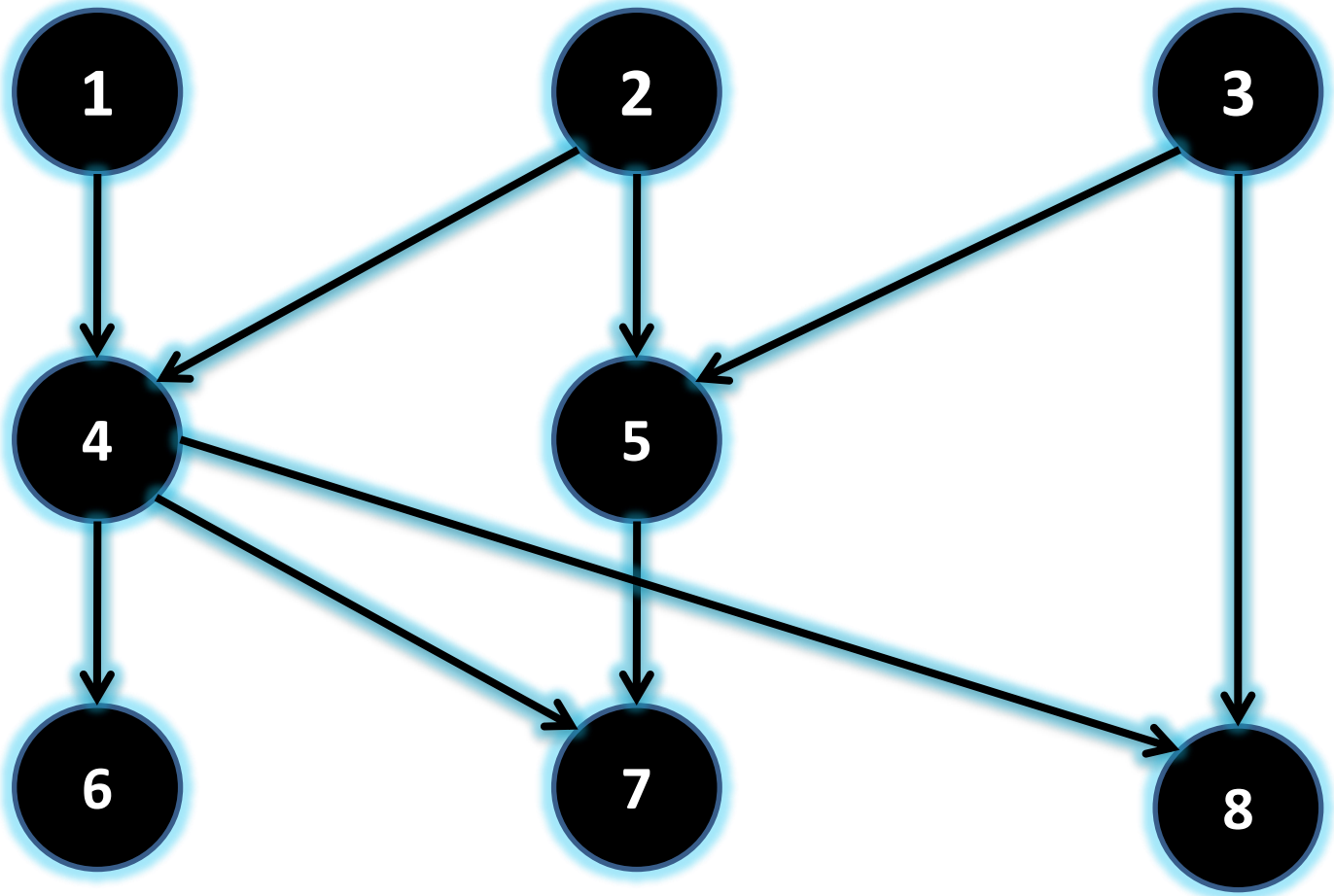
# B) UNDIRECTED GRAPH:



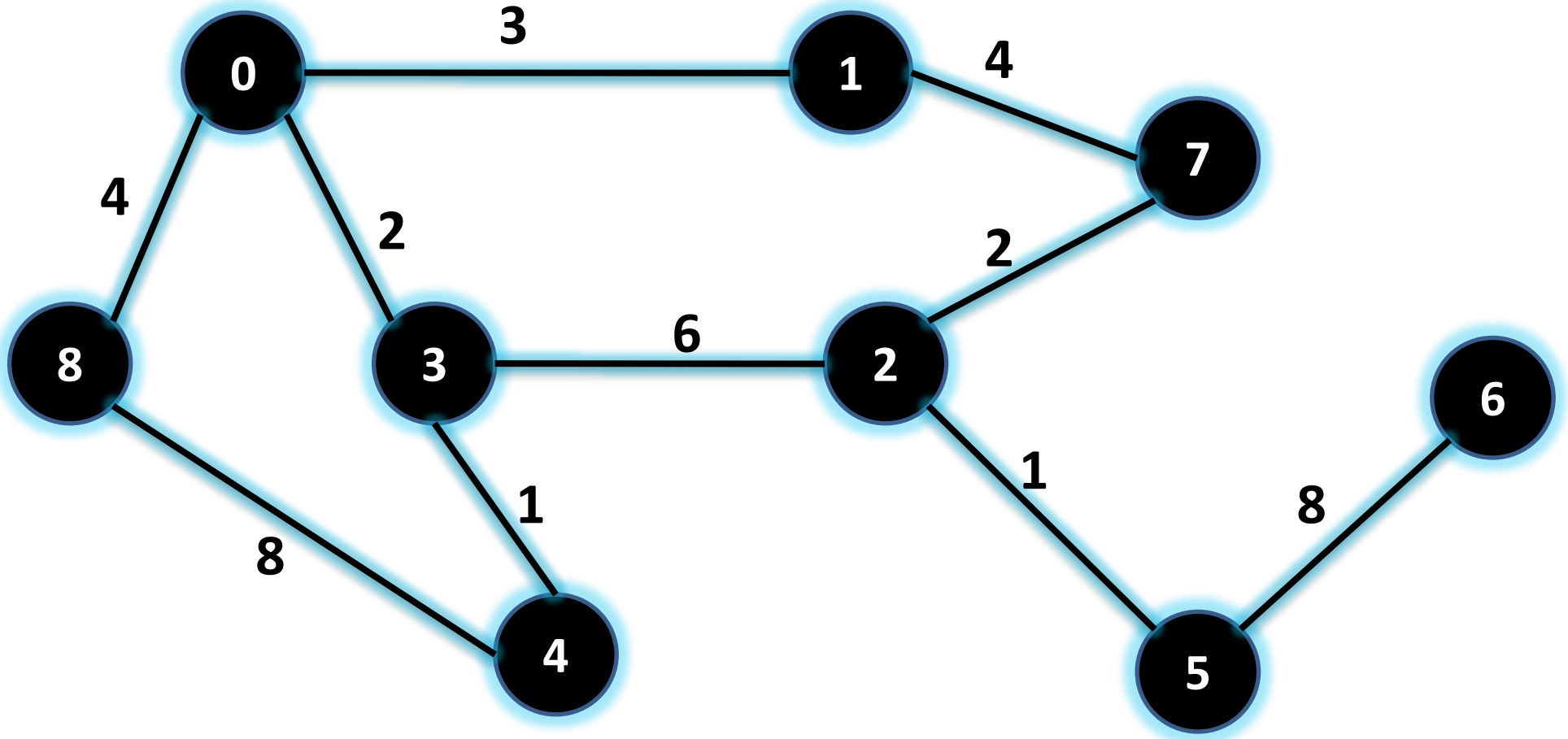
## c) CYCLIC GRAPH:



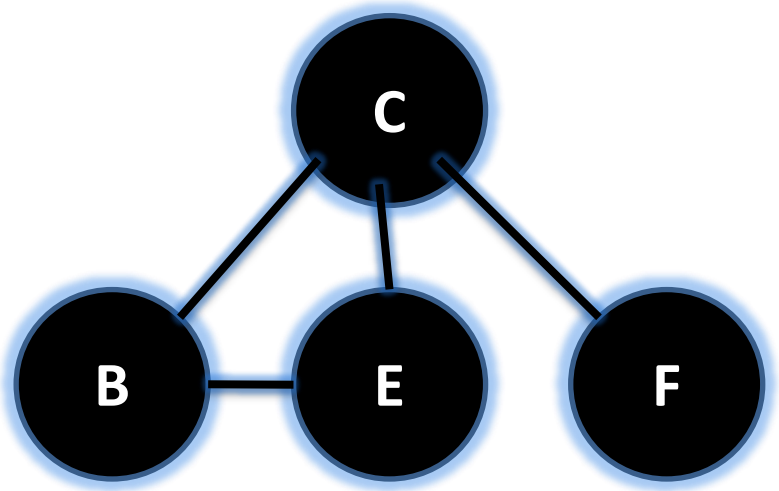
# D) ACYCLIC GRAPH :



# E) WEIGHTED GRAPH:



# F) UNWEIGHTED GRAPH:



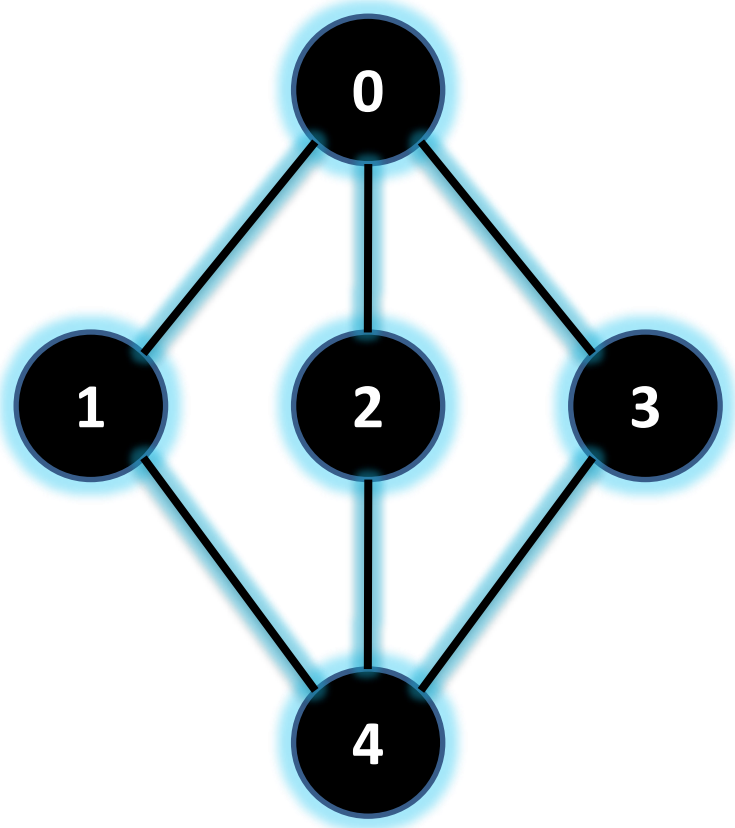
# Graph and its Representations

- Adjacency Matrix
- Adjacency List

# Adjacency Matrix

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

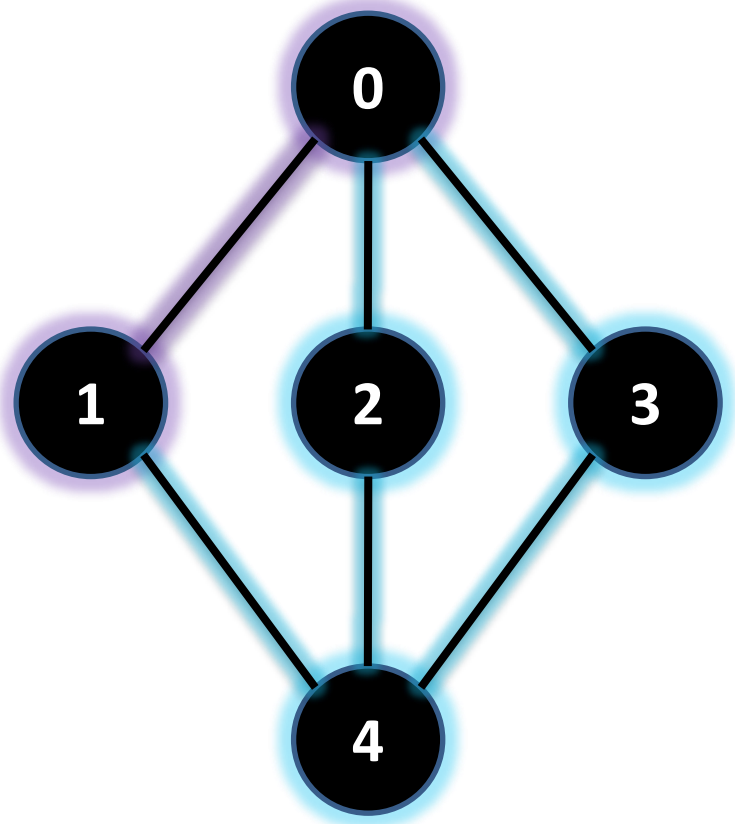
# Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |

**Initialize the matrix**

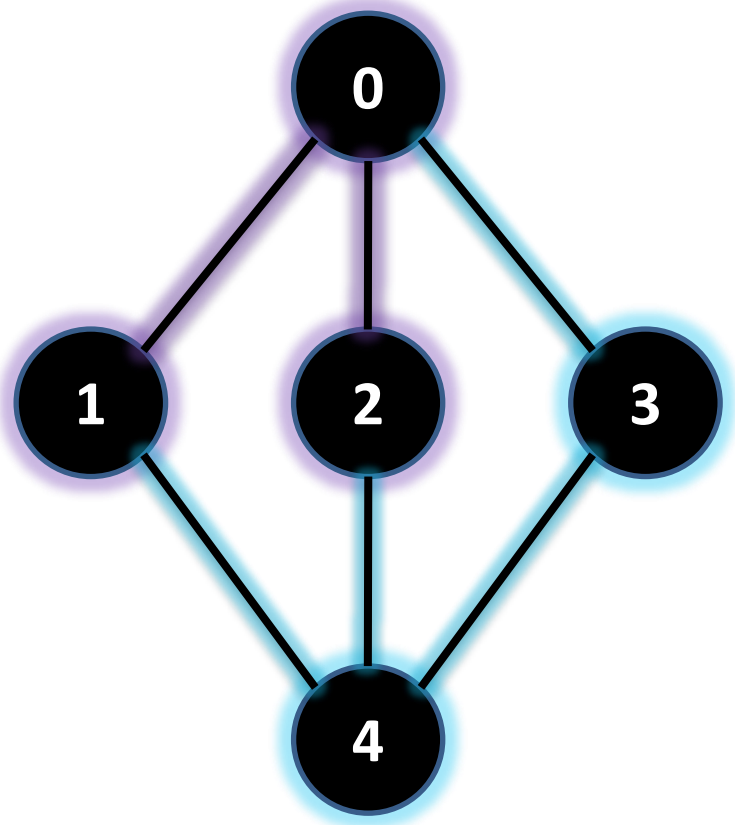
# Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |

**Edge 0->1. So, mark 1 A[0][1] and A[1][0] ( because this is an undirected graph.**

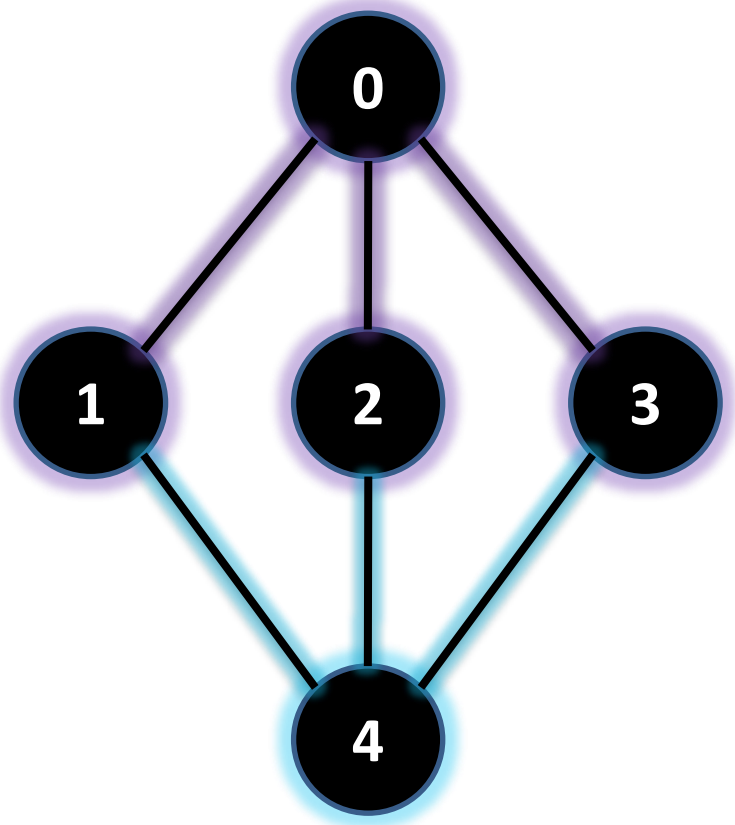
# Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |

**Edge 0->2. So, mark 1 A[0][2] and A[2][0].**

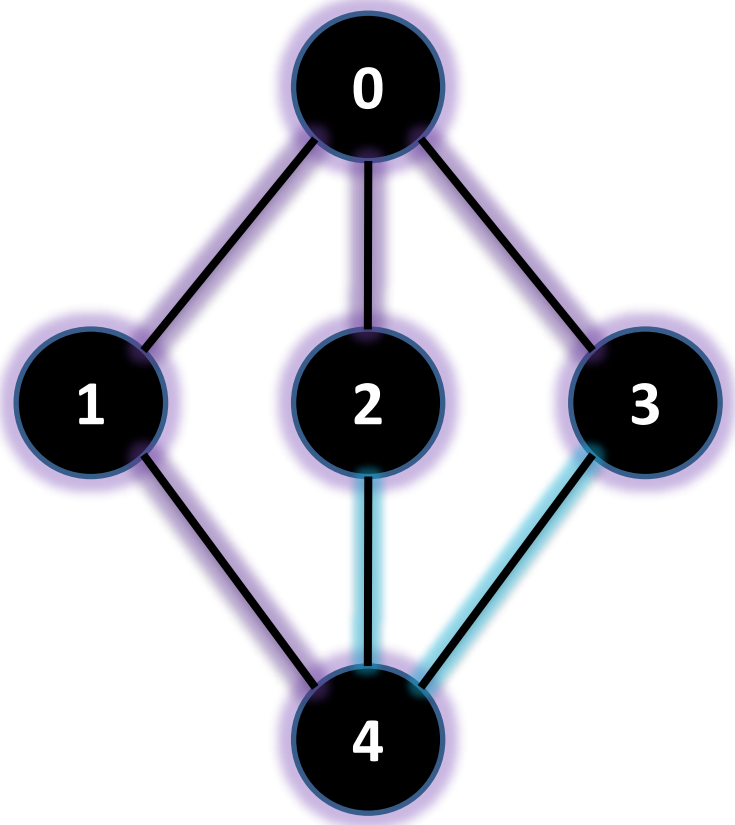
# Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |

**Edge 0->3. So, mark 1 A[0][3] and A[3][0].**

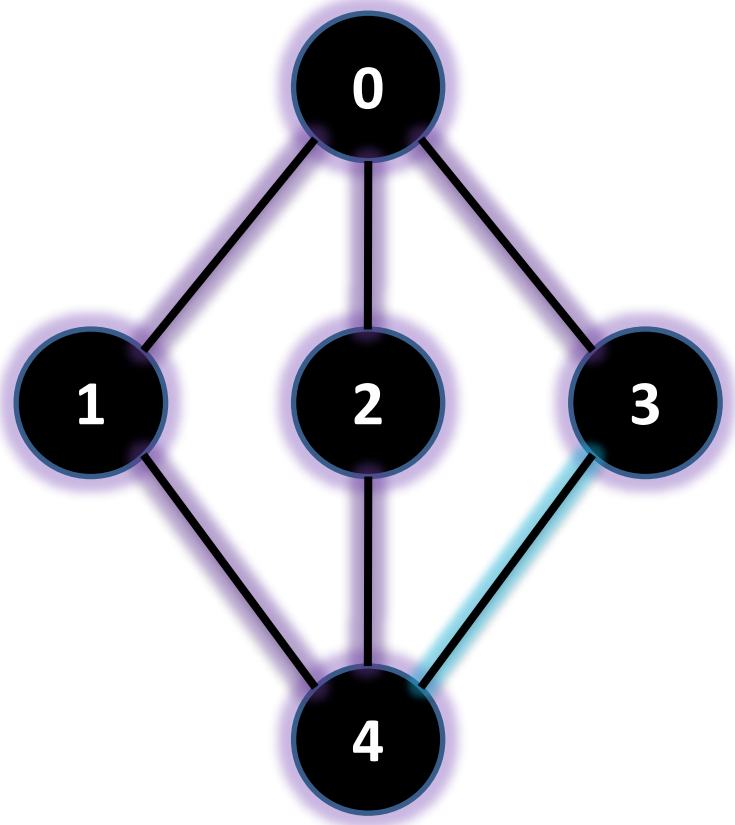
# Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 |

**Edge 1->4. So, mark 1 A[1][4] and A[4][1].**

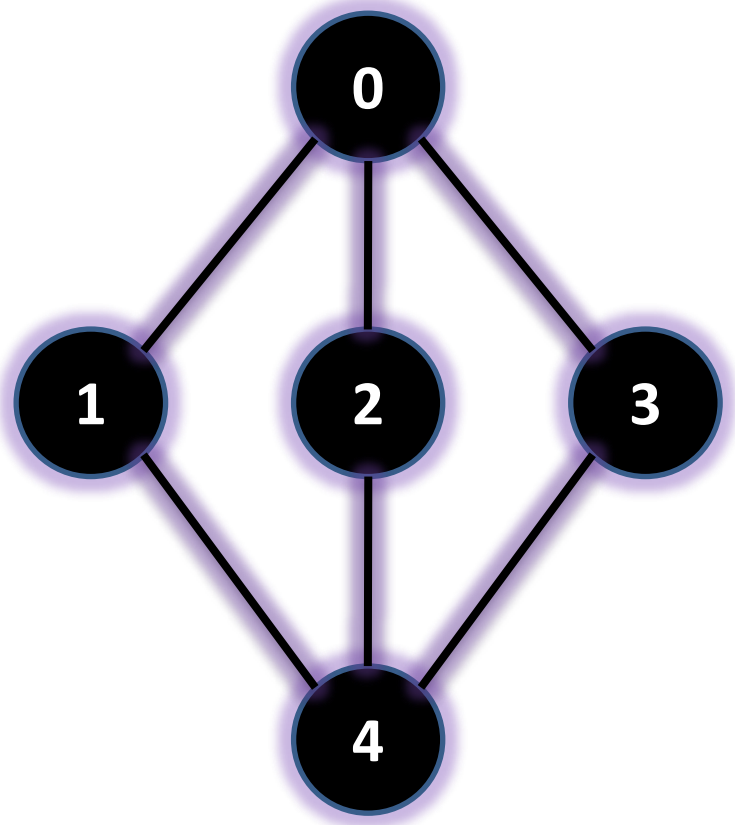
# Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 | 0 |

**Edge 2->4. So, mark 1 A[2][4] and A[4][2].**

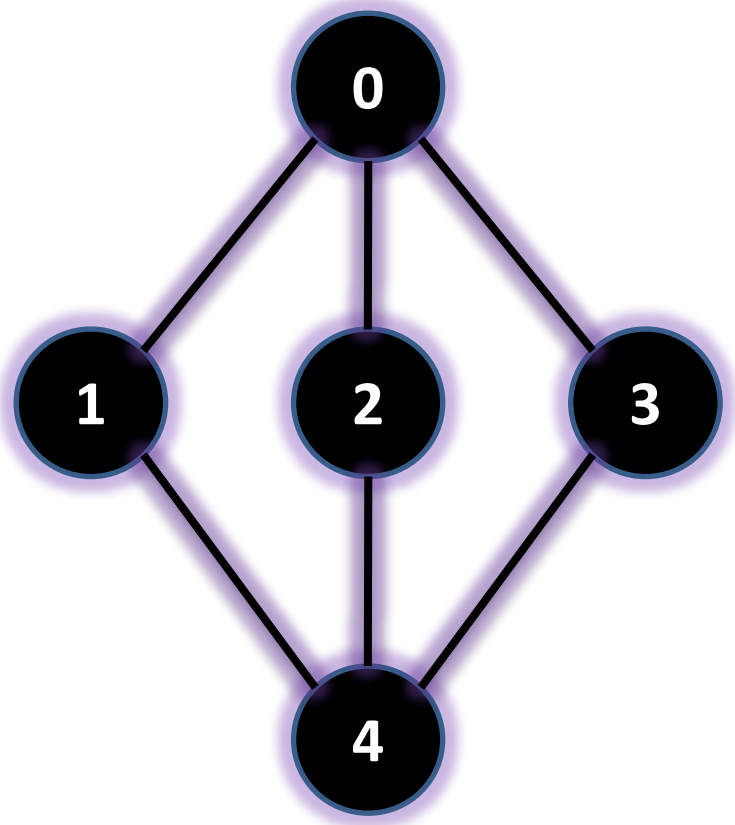
# Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 1 | 0 |

**Edge 3->4. So, mark 1 A[3][4] and A[4][3].**

# Adjacency Matrix



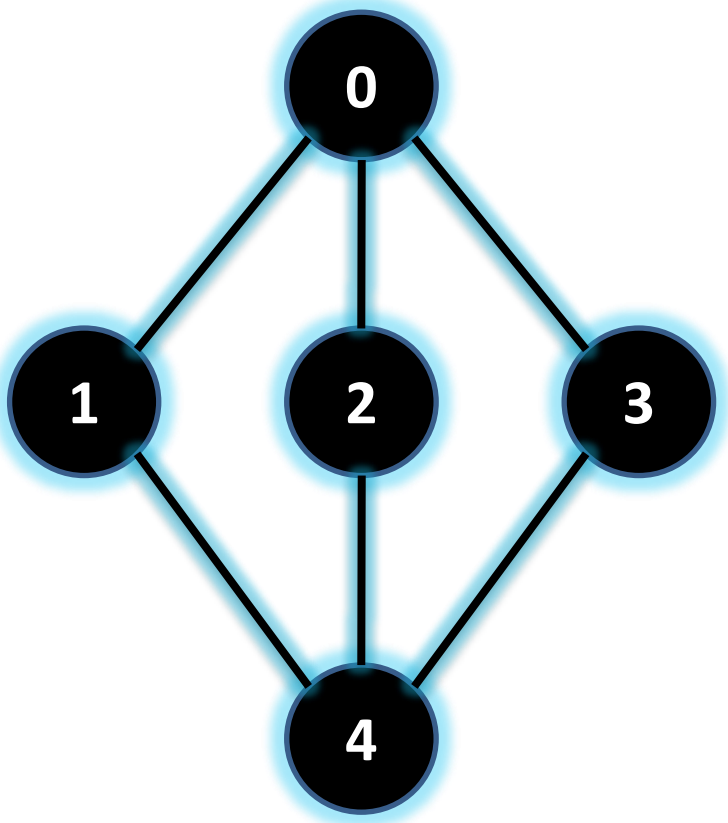

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 1 | 0 |

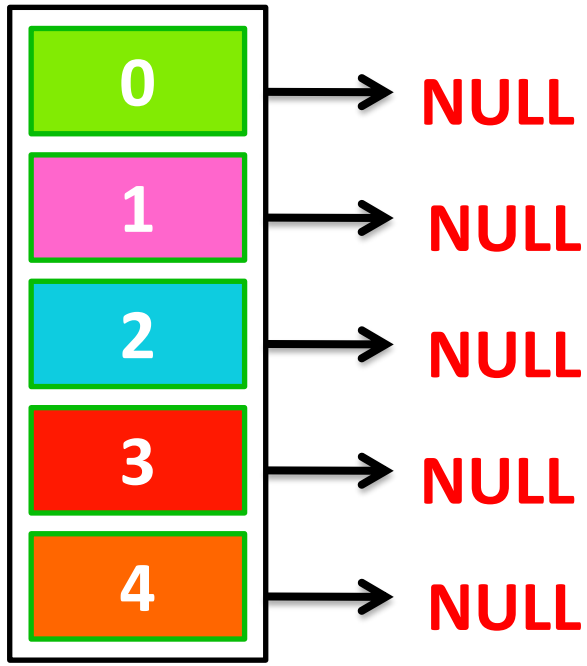
**All the edges have been marked. This is the adjacency matrix for the given graph.**

# Adjacency List

- An array of linked lists is used.
- Size of the array is equal to number of vertices. Let the array be array[].
- An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex.
- This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists.

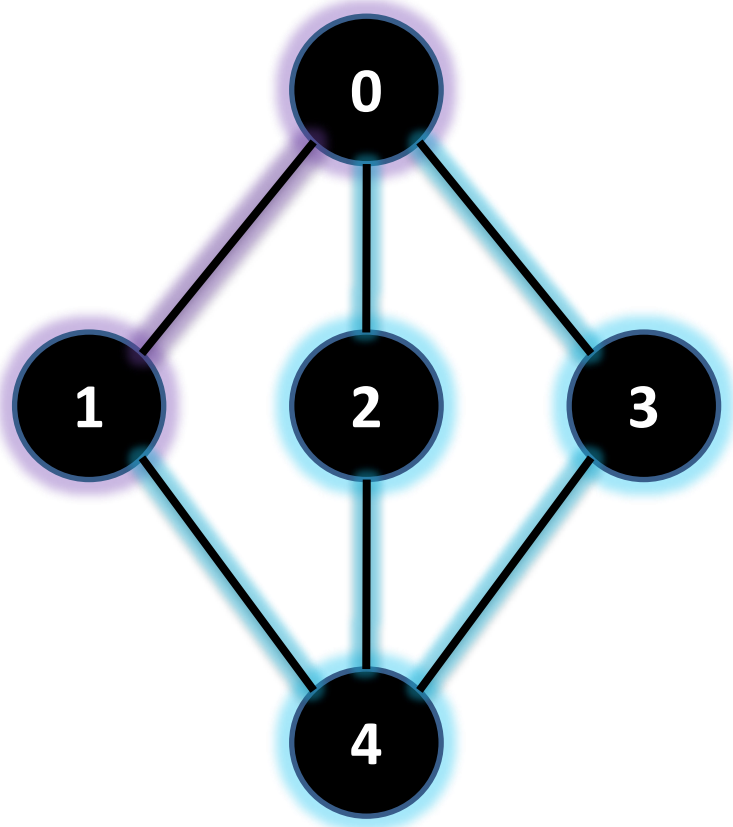
# Adjacency list

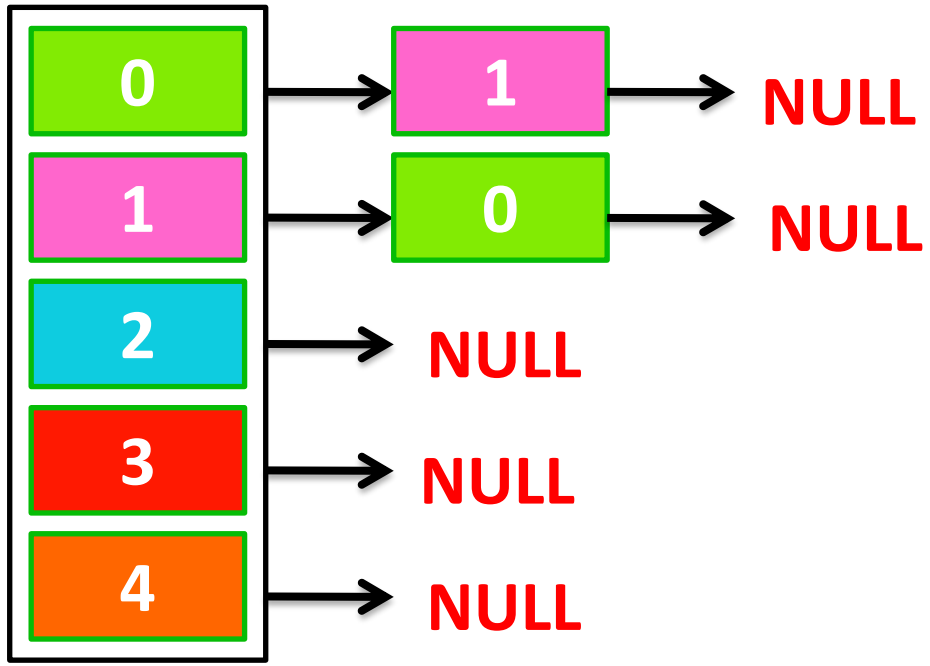



Initialize the list

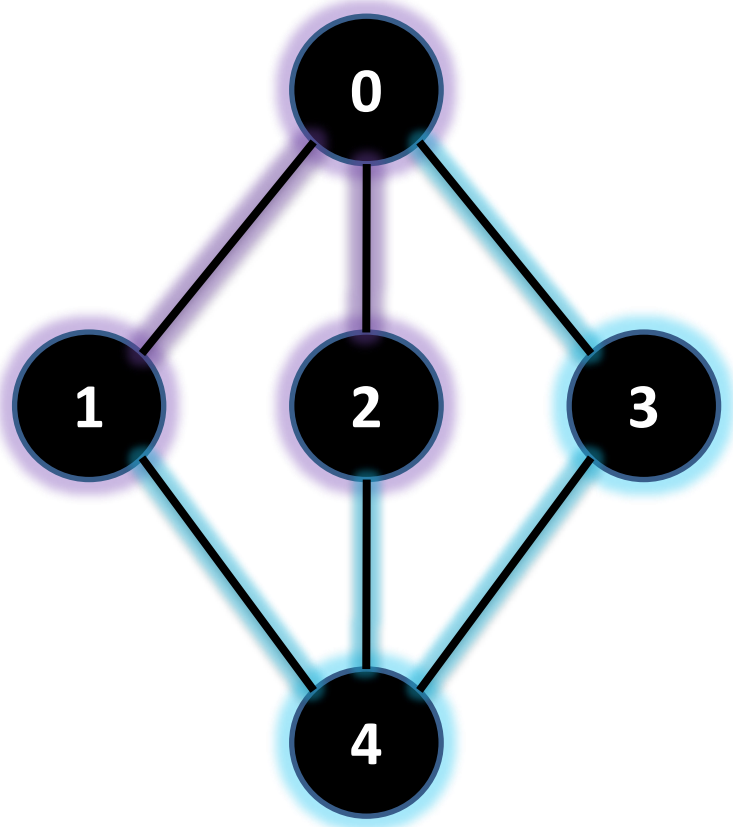
# Adjacency list

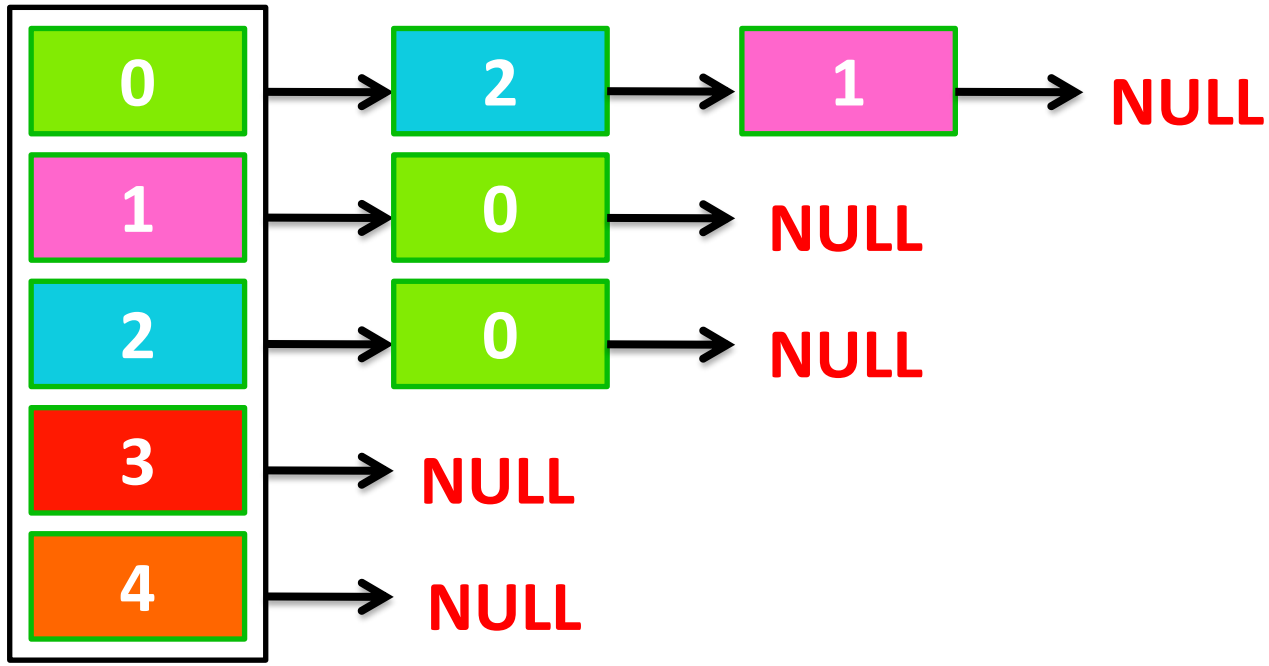



Edge 0->1.

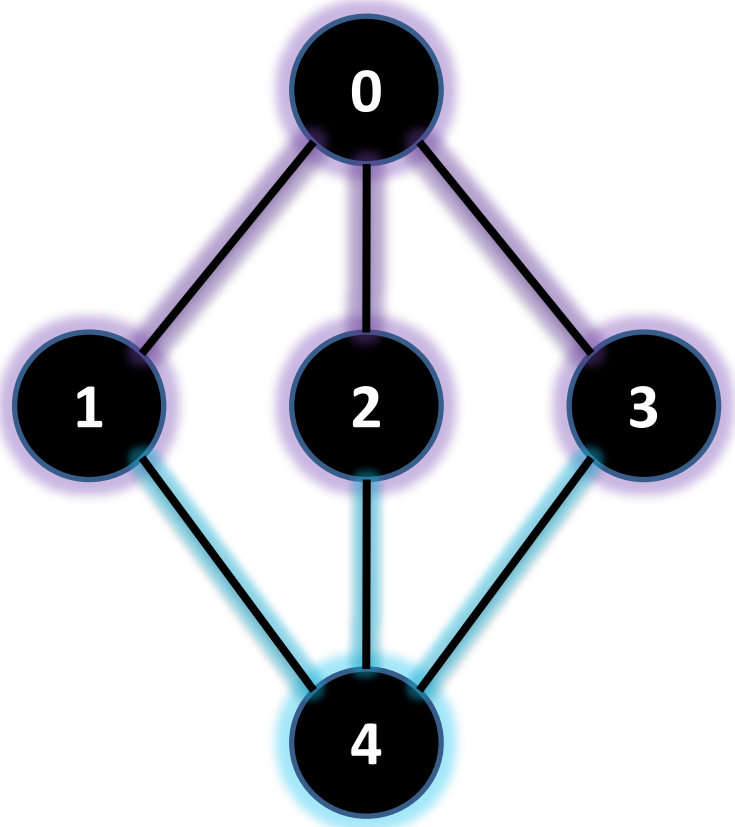
# Adjacency list

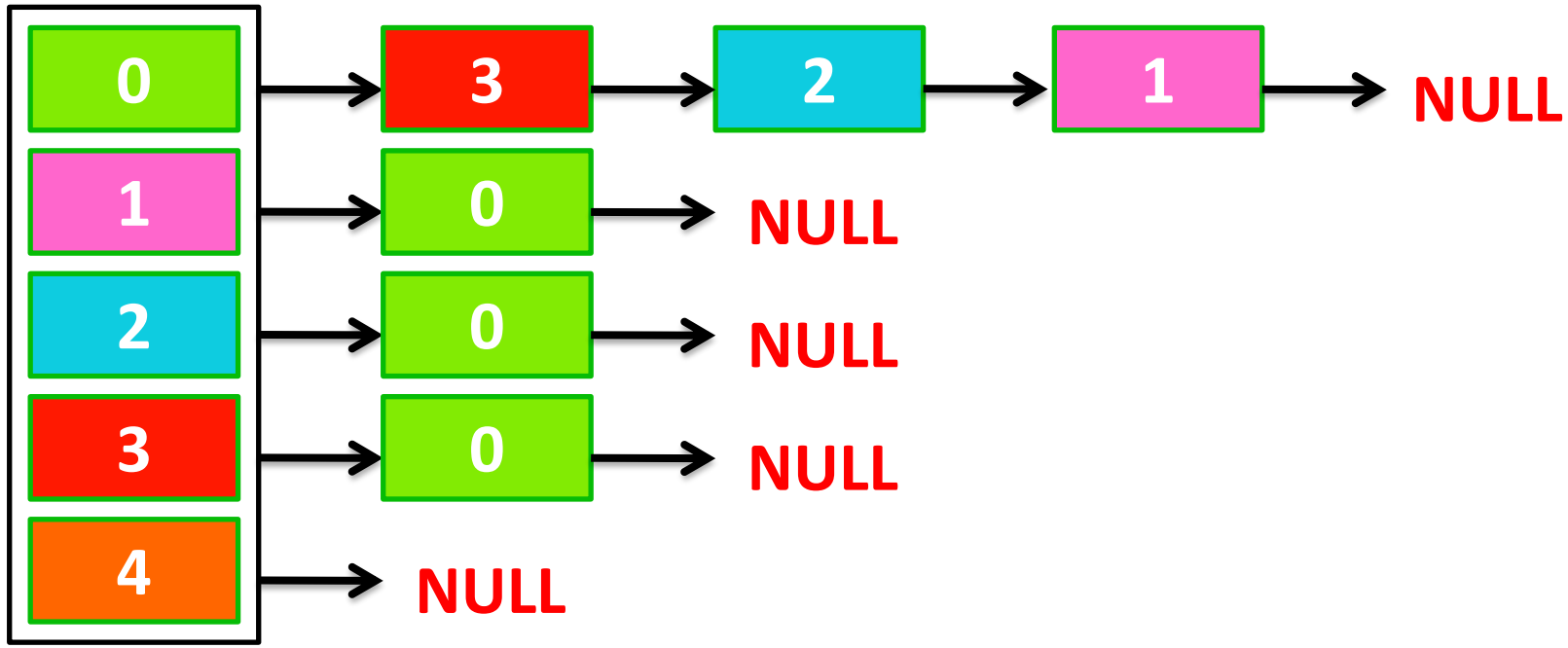



Edge 0->2.

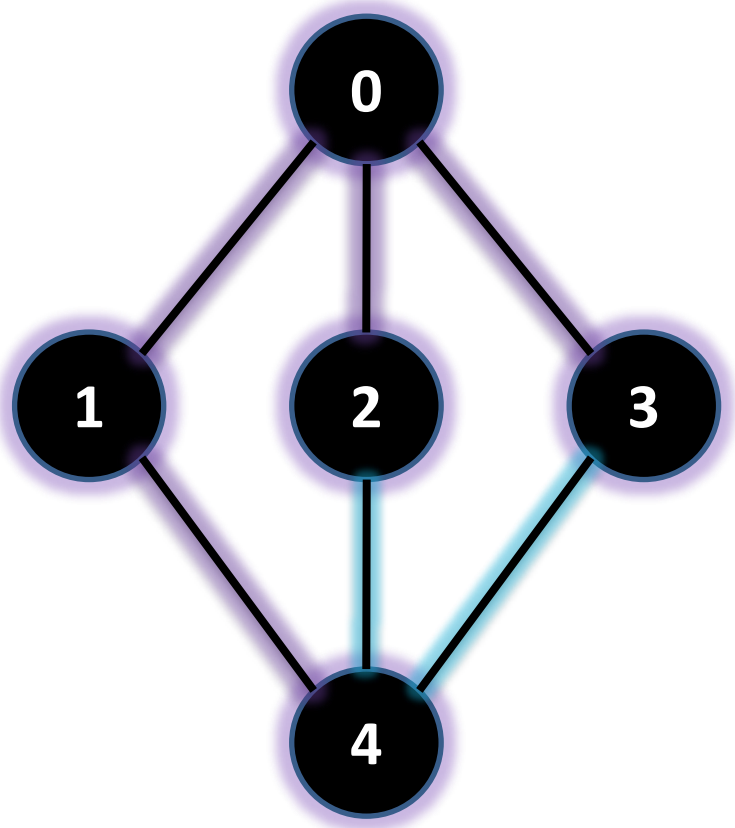
# Adjacency list

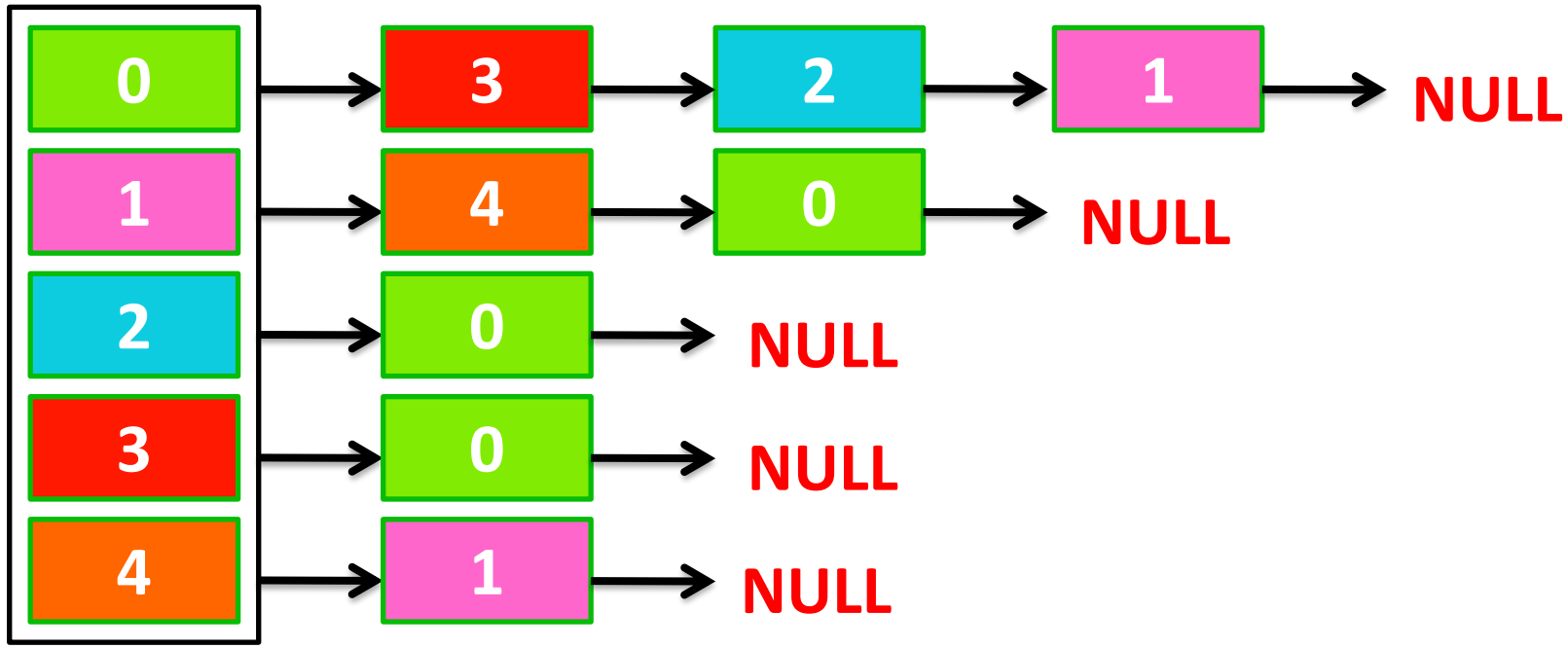



Edge 0->3.

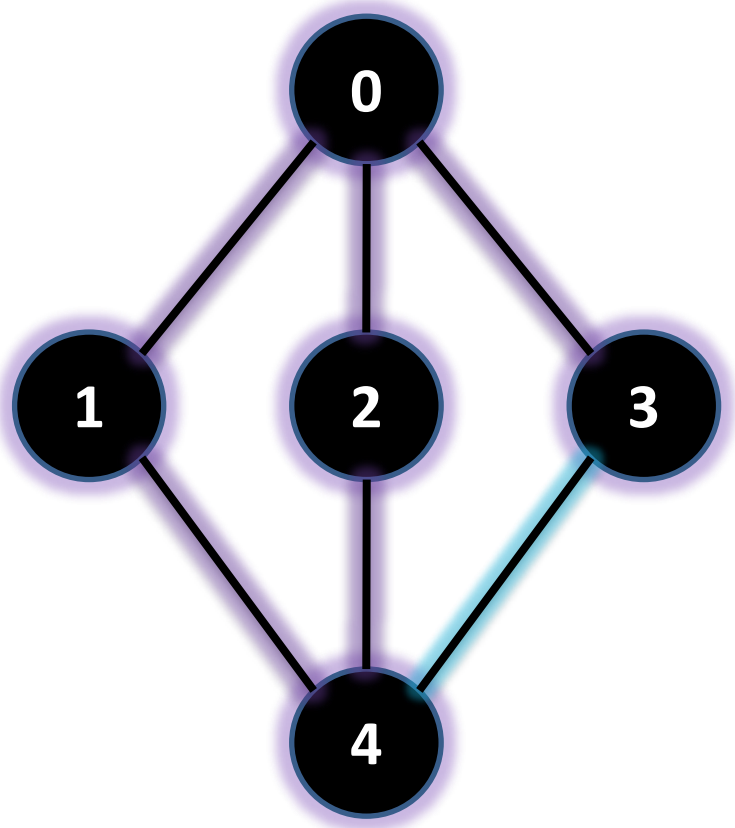
# Adjacency list

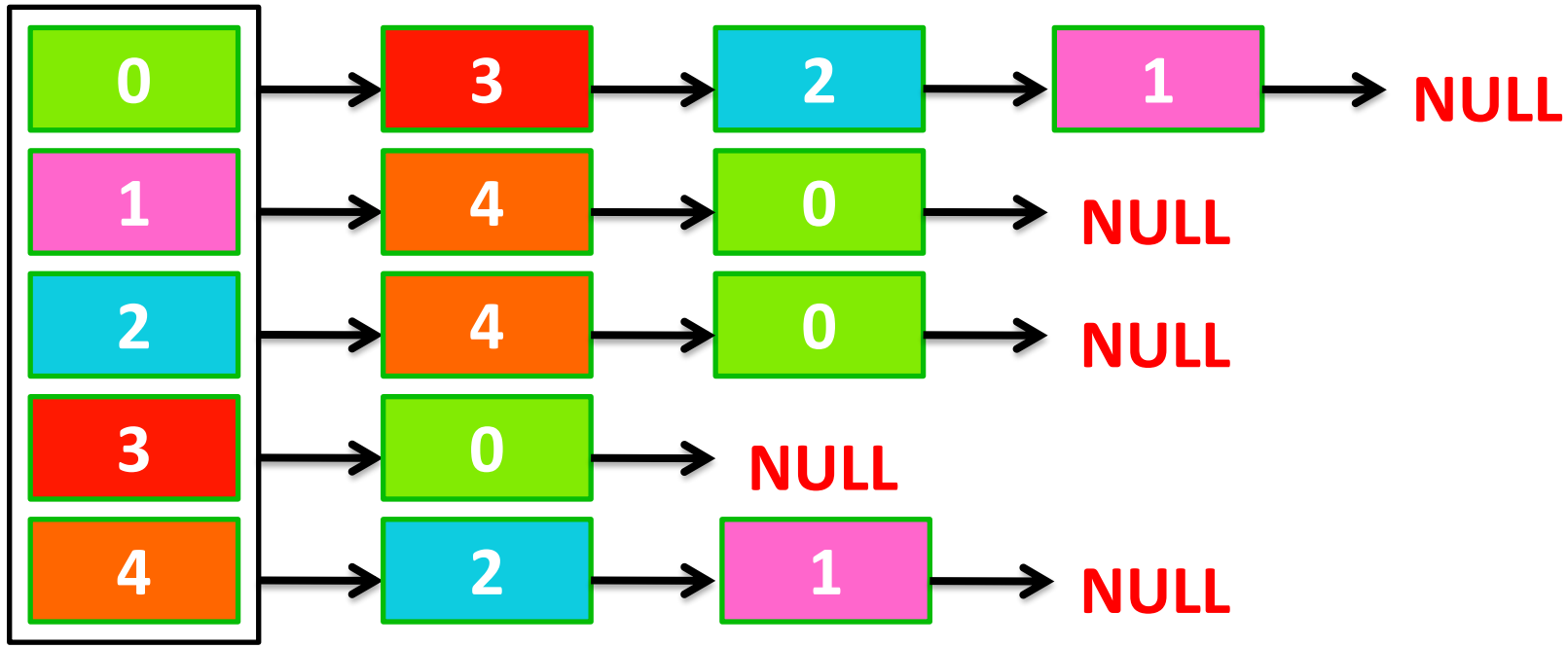



Edge 1->4.

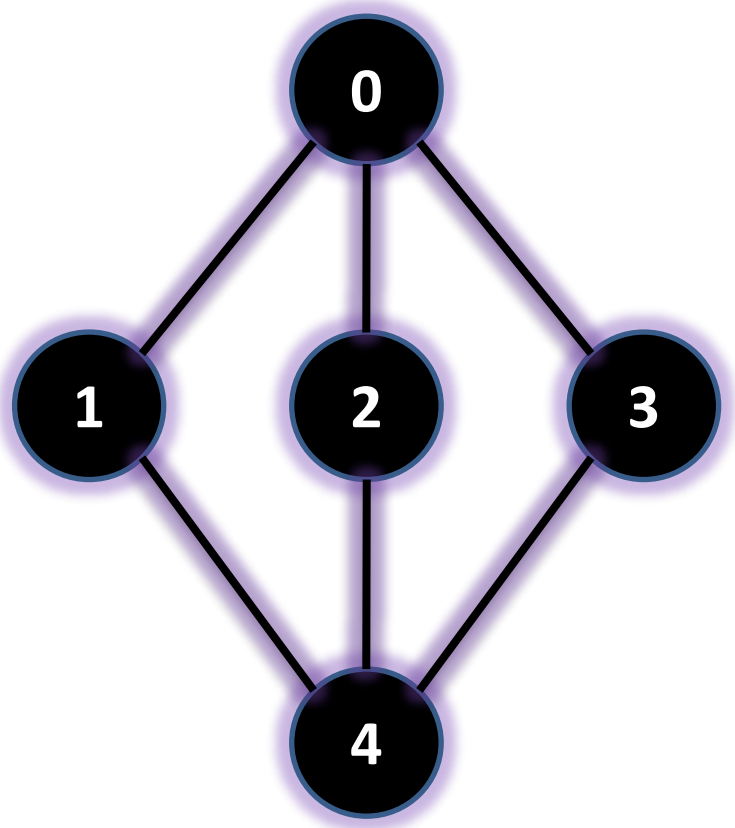
# Adjacency list

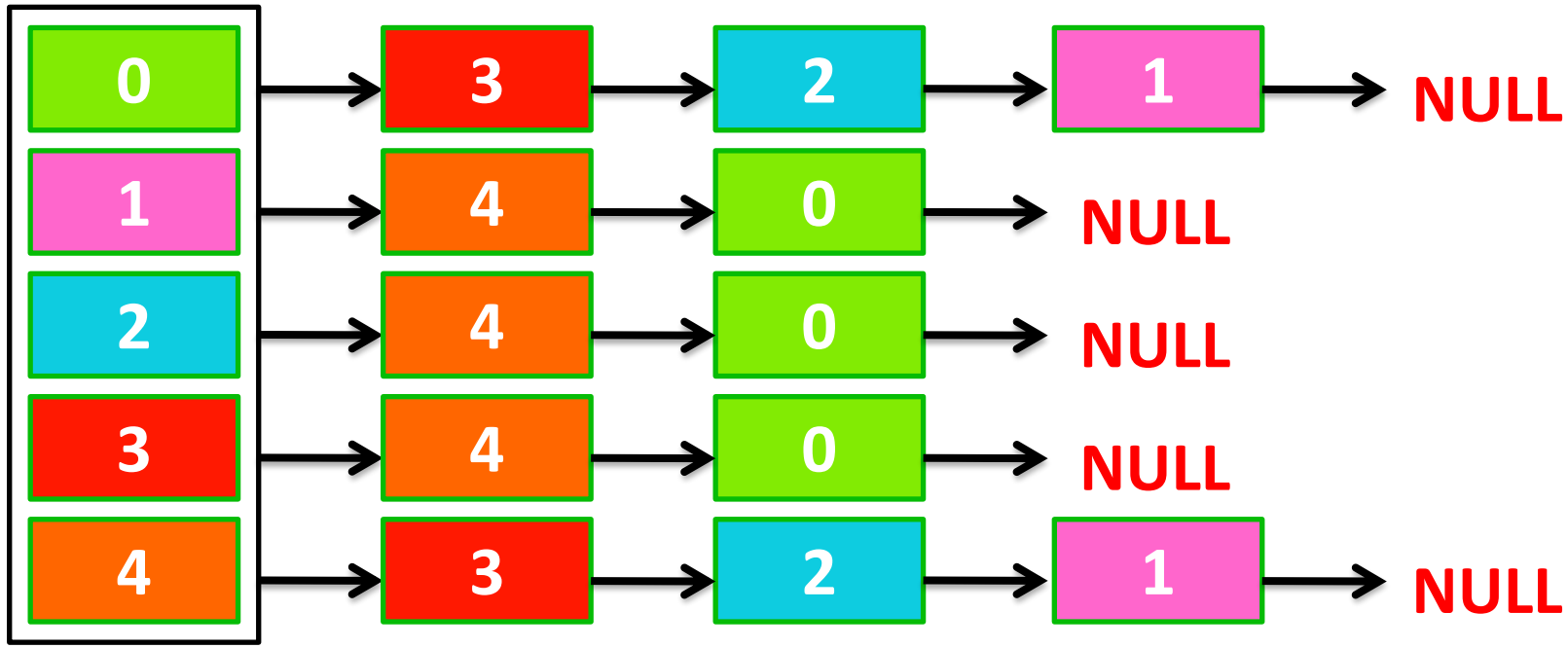



Edge 2->4.

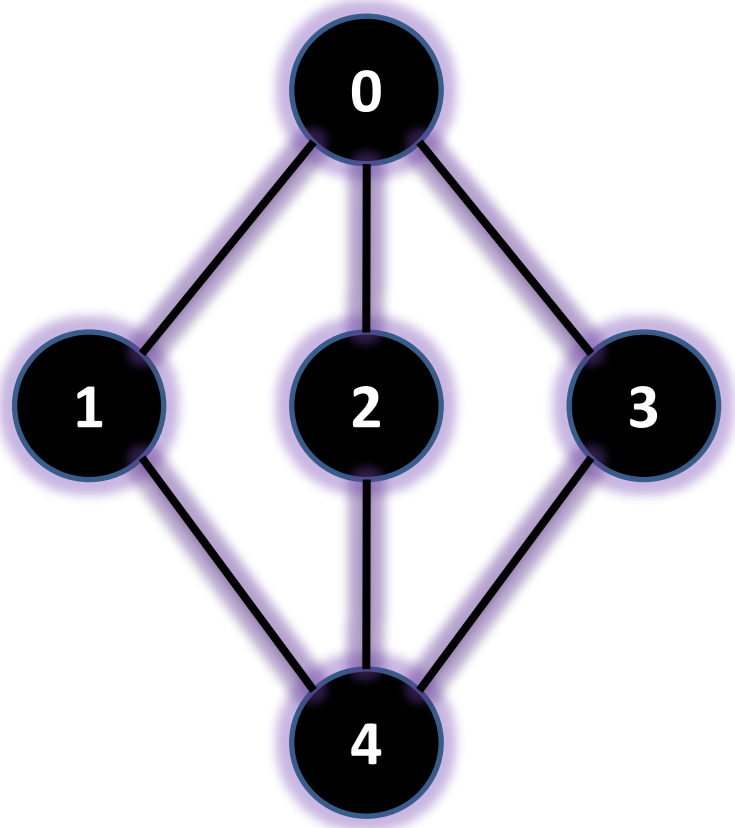
# Adjacency list

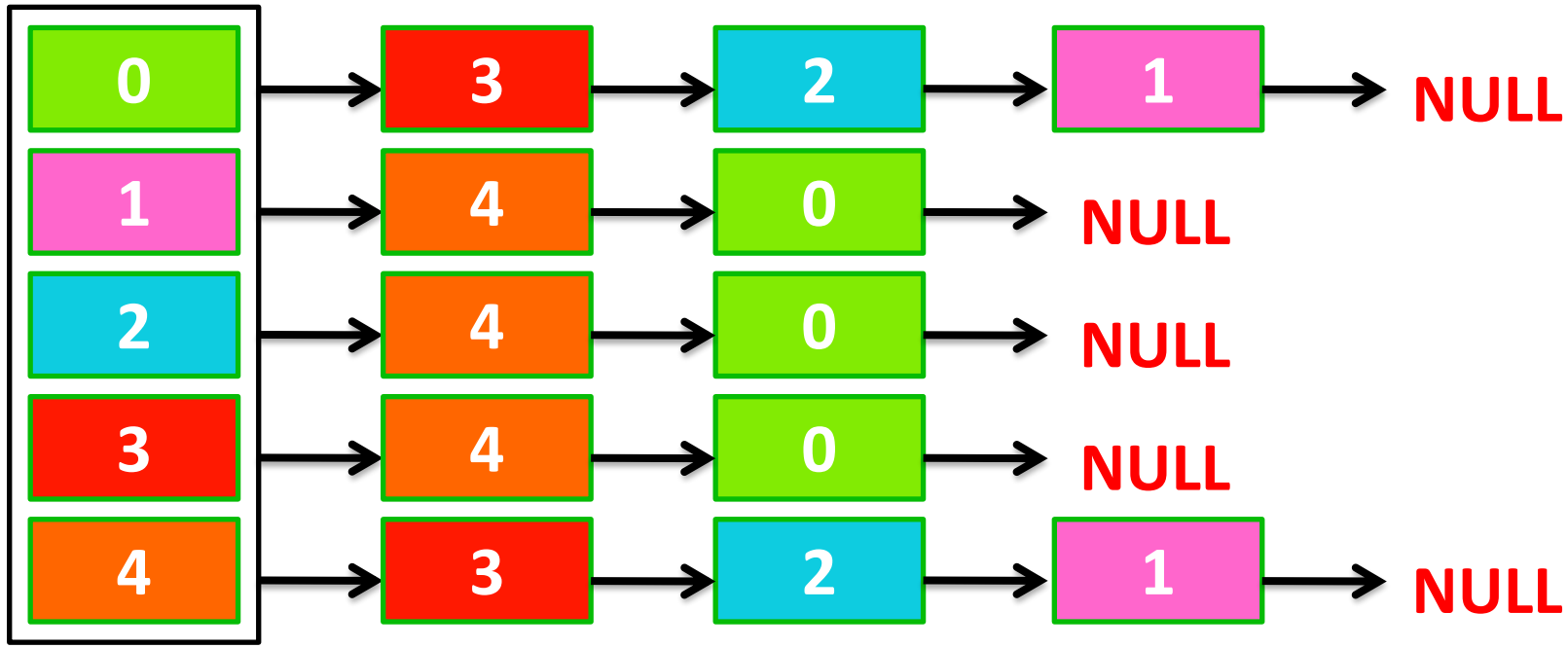



Edge 3->4.

# Adjacency list




This is the adjacency list for the given graph.

# Types of Traversals

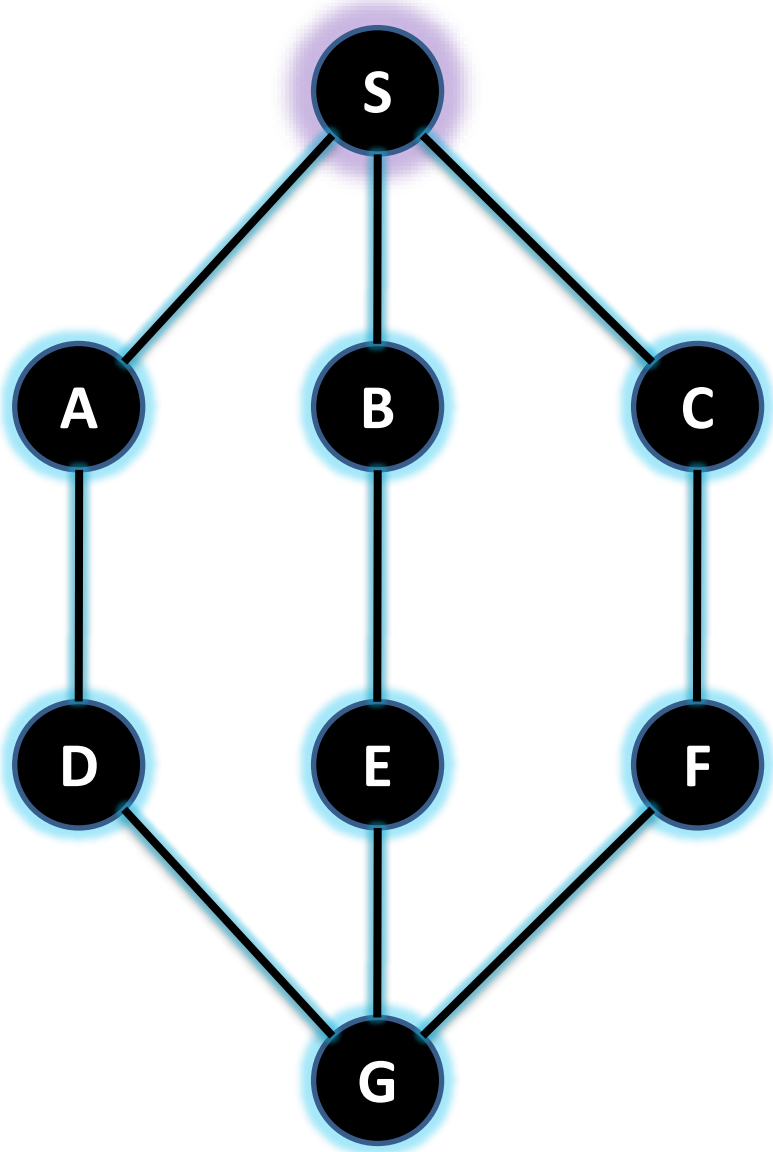
1) **Depth First Search (DFS) algorithm.**

2) **Breadth First Search (BFS) algorithm.**

# DFS

1)(DFS) algorithm traverses a graph in a depth ward motion

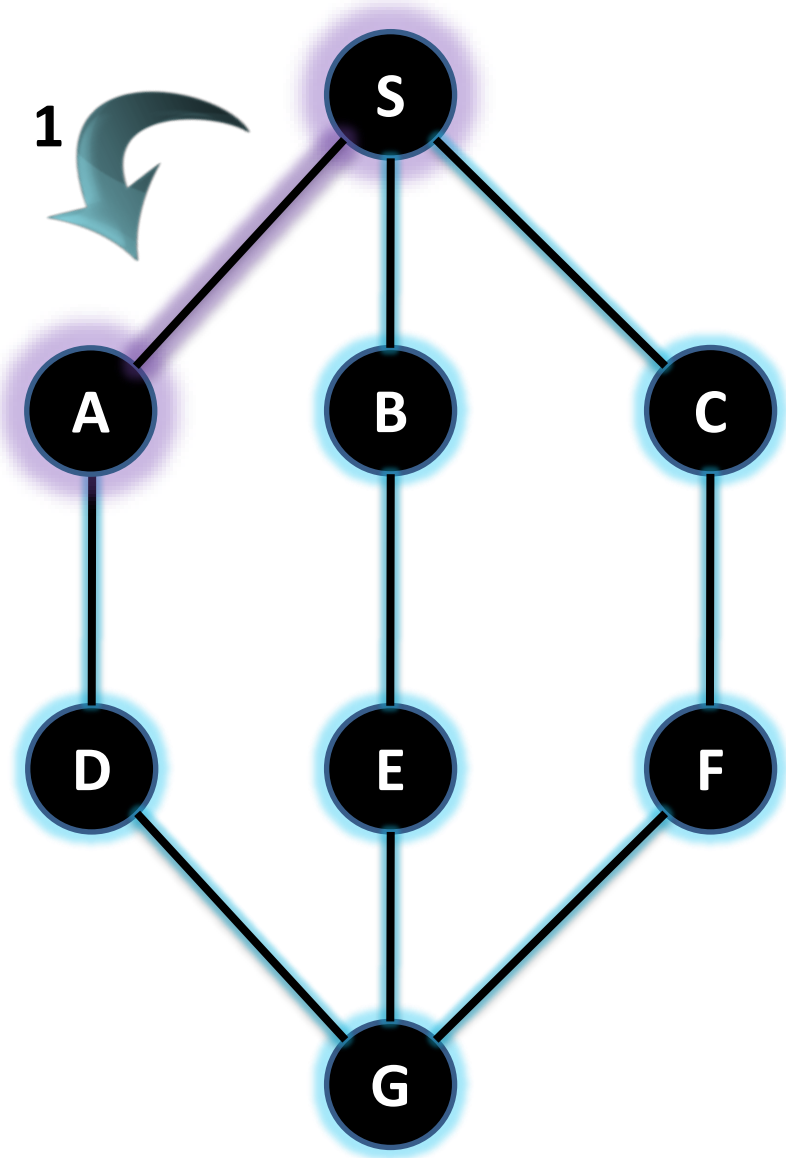
2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# DFS

1)(DFS) algorithm traverses a graph in a depth ward motion

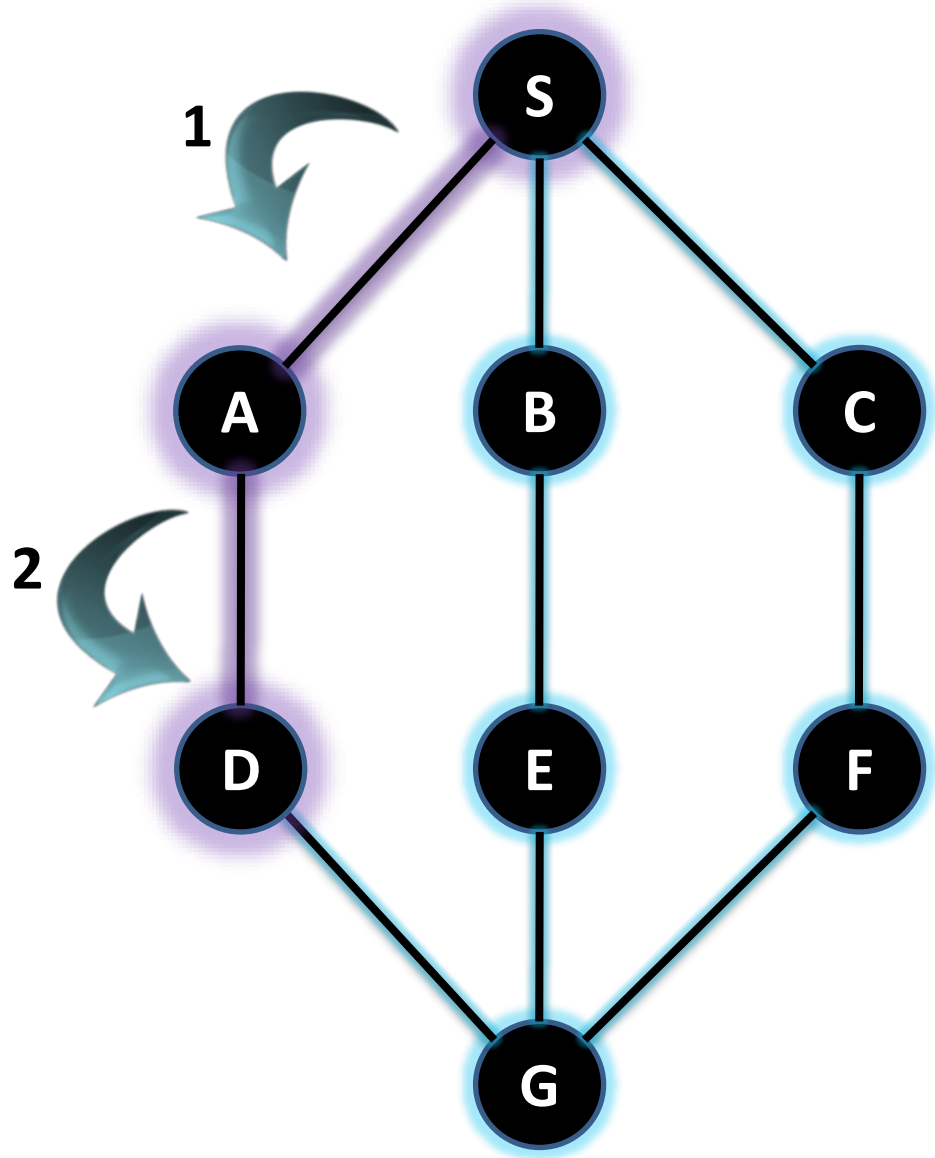
2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# DFS

1)(DFS) algorithm traverses a graph in a depth ward motion

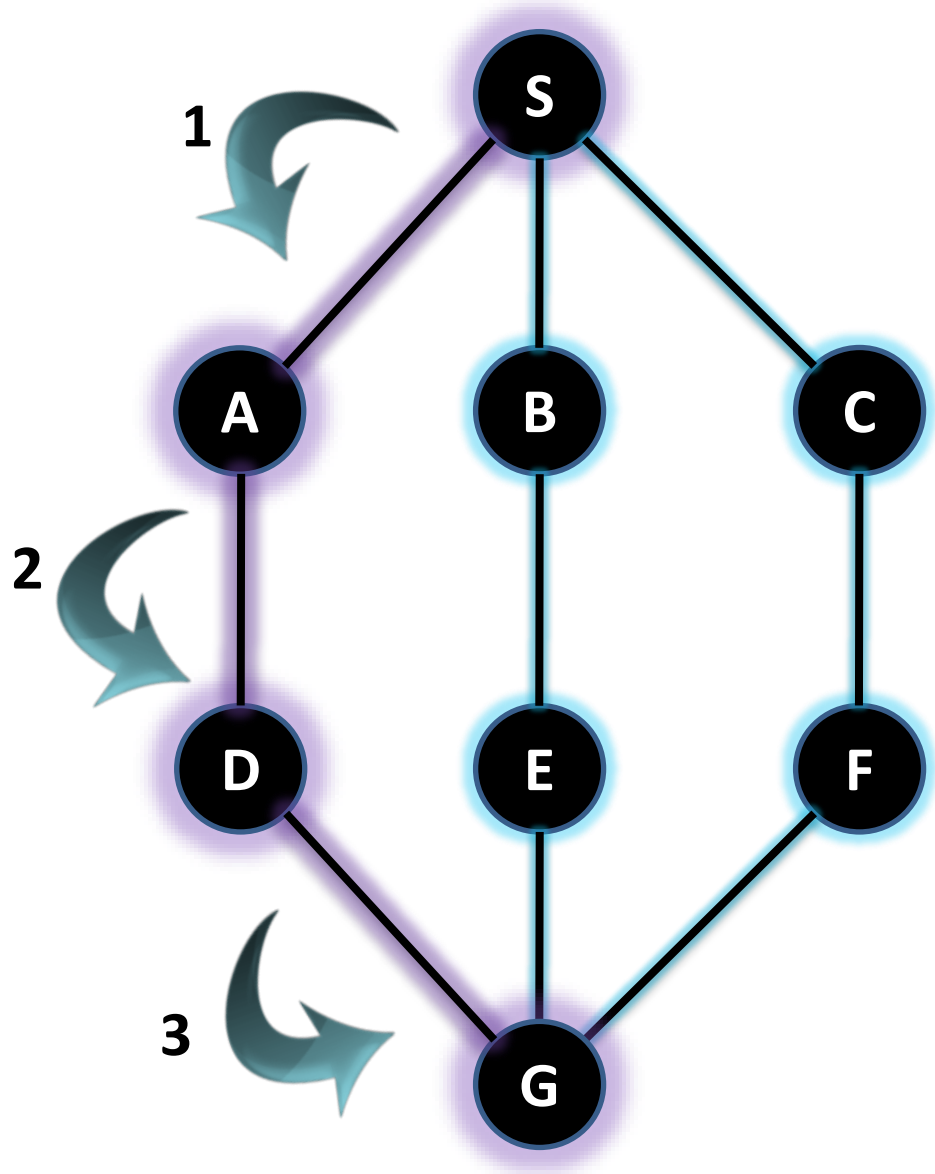
2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# DFS

1)(DFS) algorithm traverses a graph in a depth ward motion

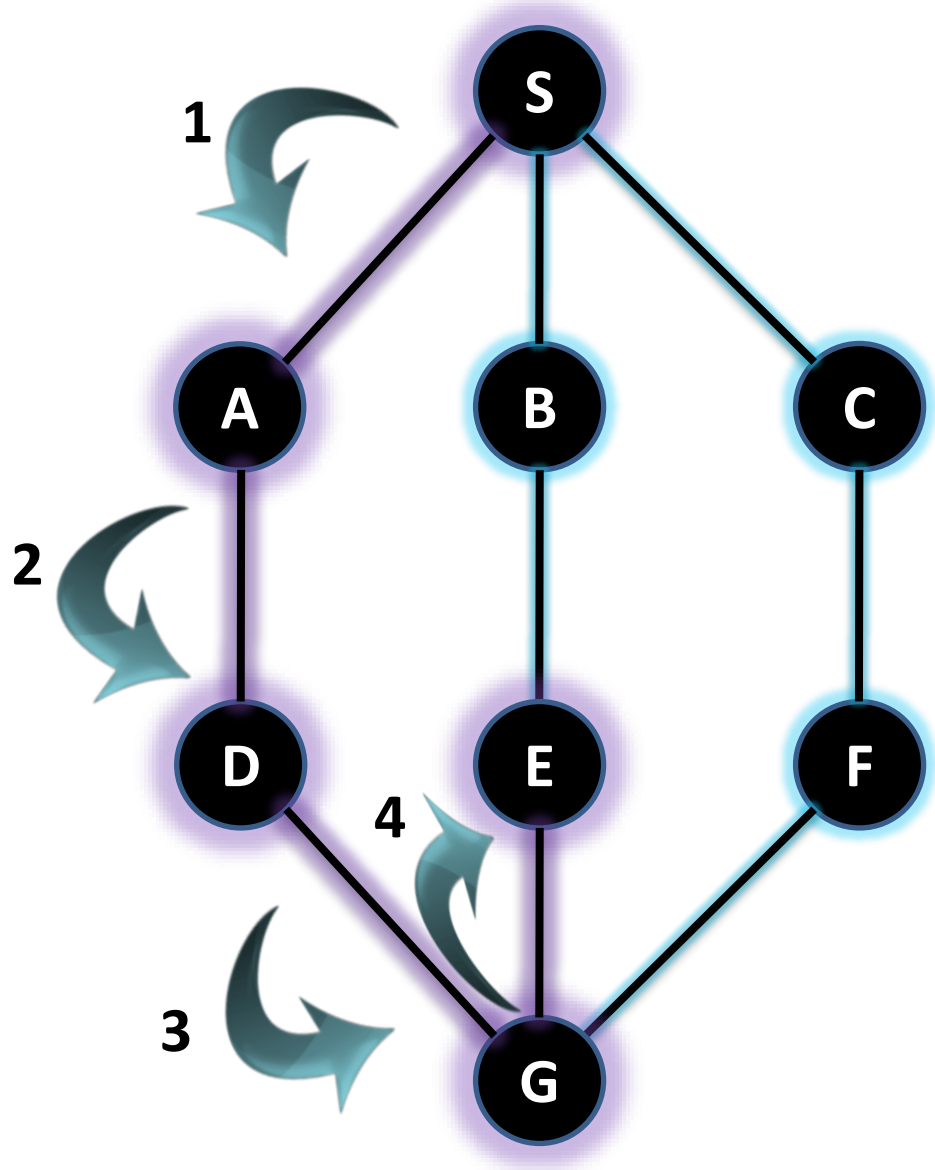
2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# DFS

1)(DFS) algorithm traverses a graph in a depth ward motion

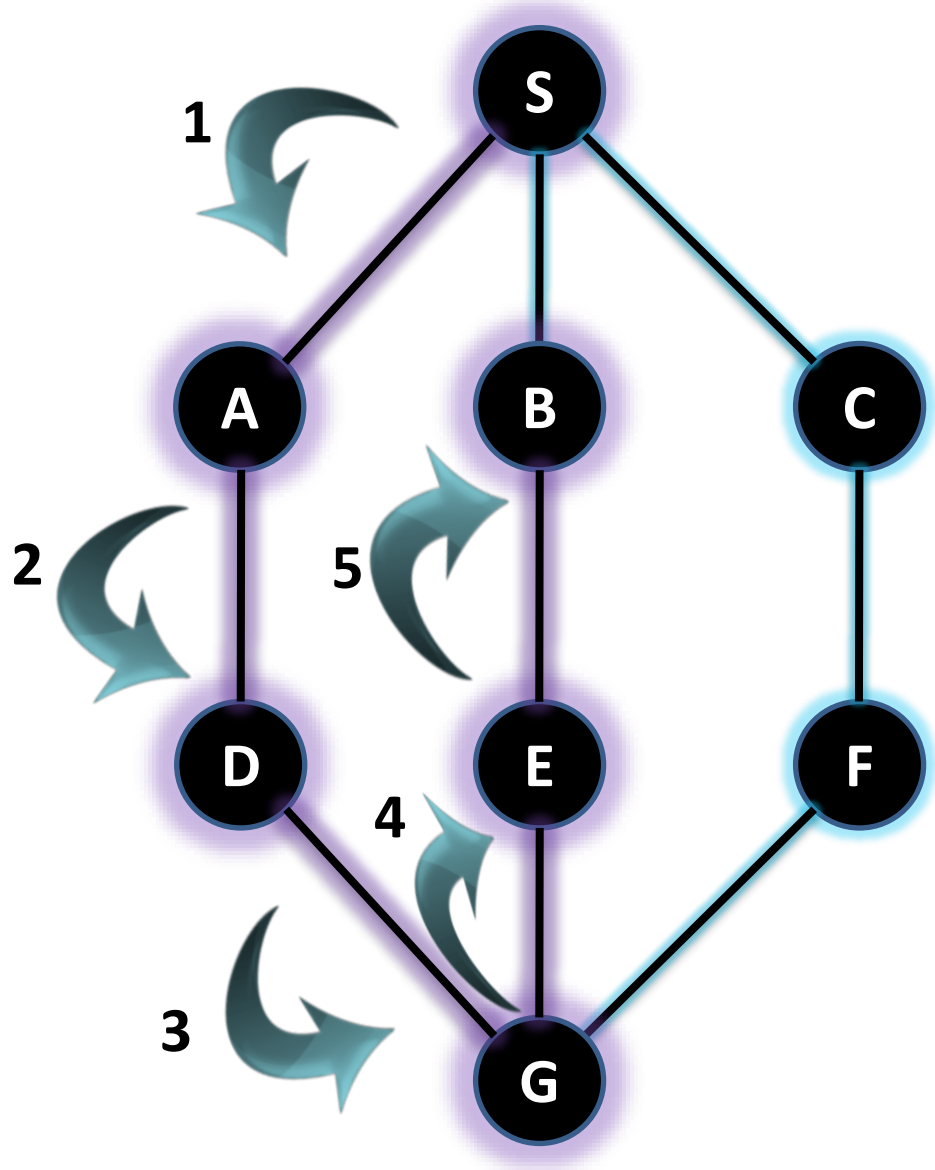
2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# DFS

1)(DFS) algorithm traverses a graph in a depth ward motion

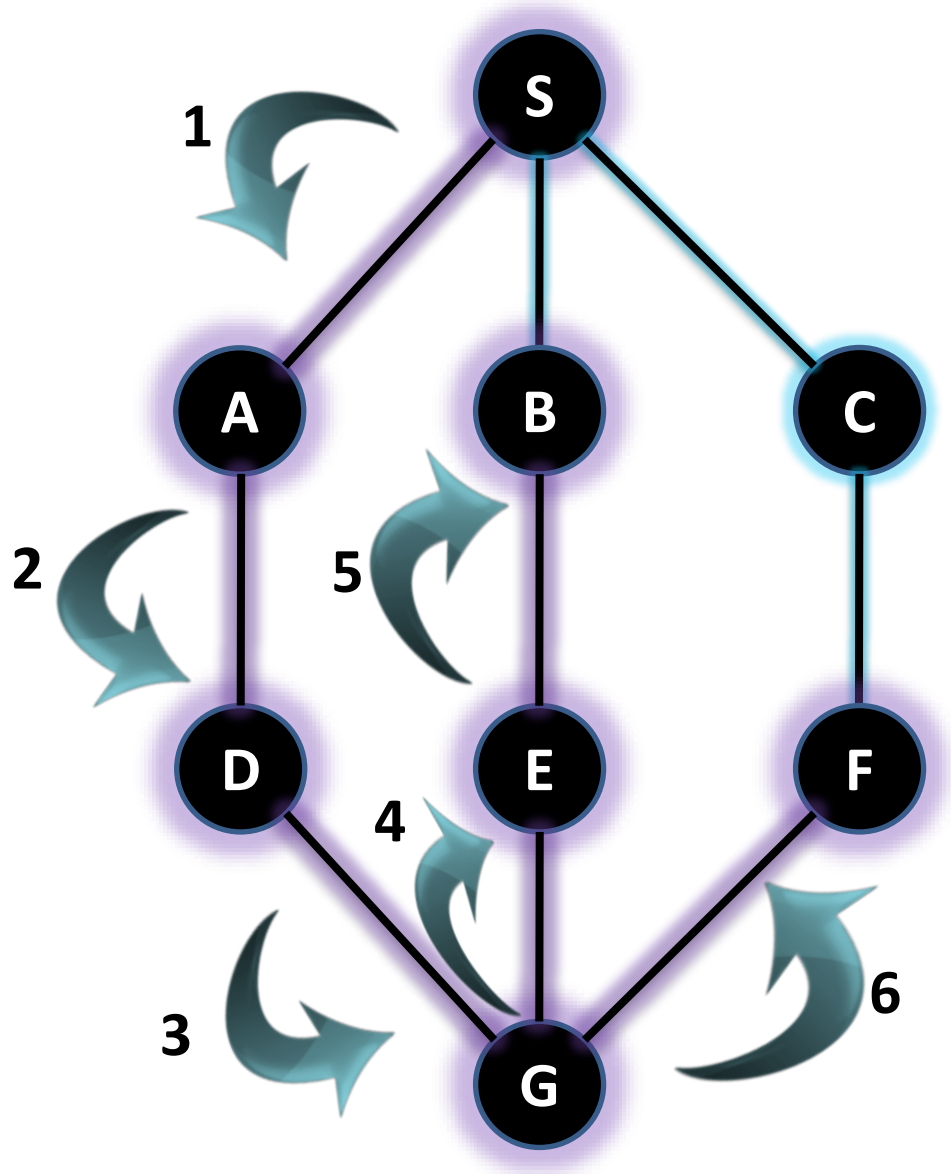
2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# DFS

1)(DFS) algorithm traverses a graph in a depth ward motion

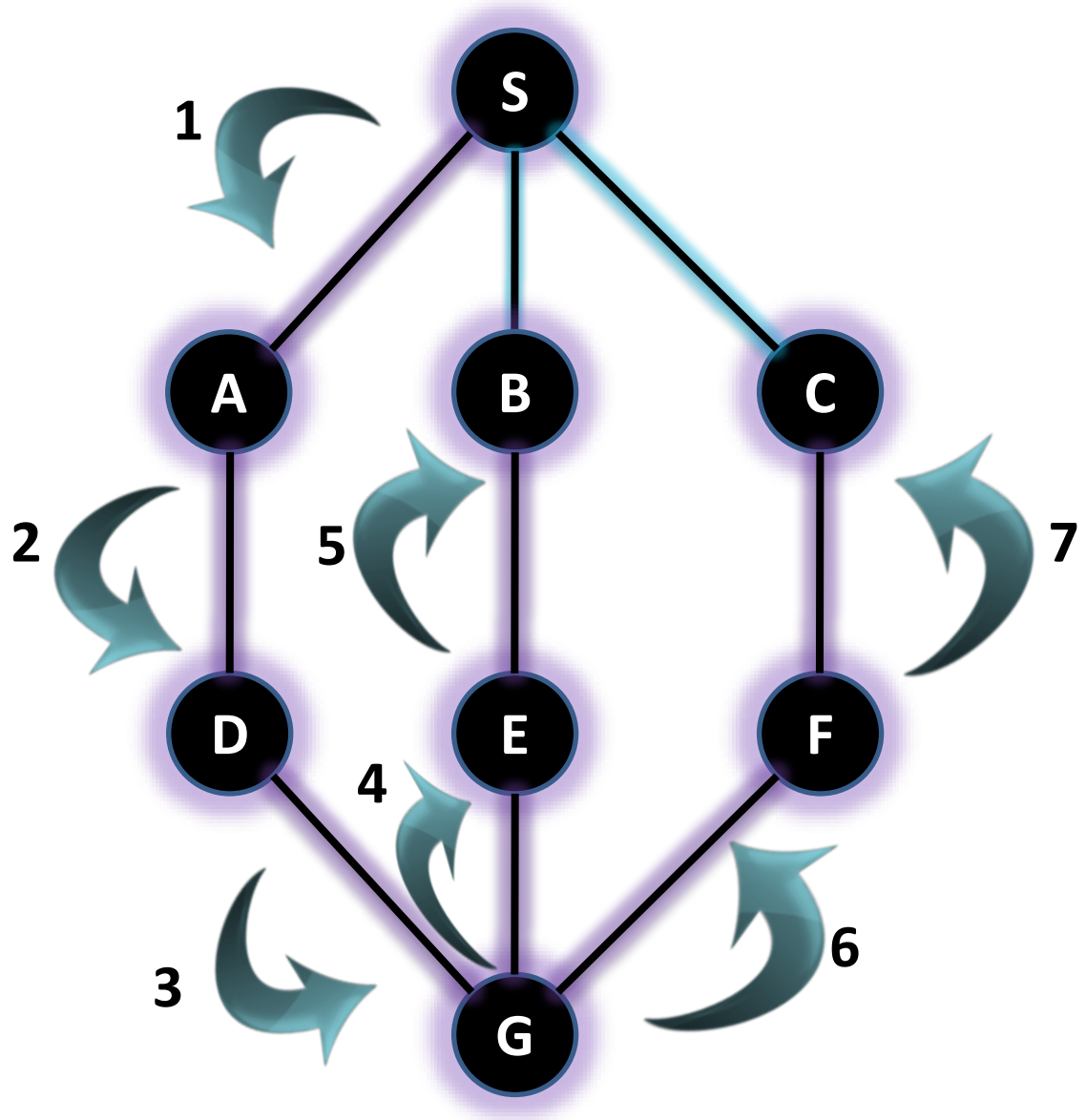
2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# DFS

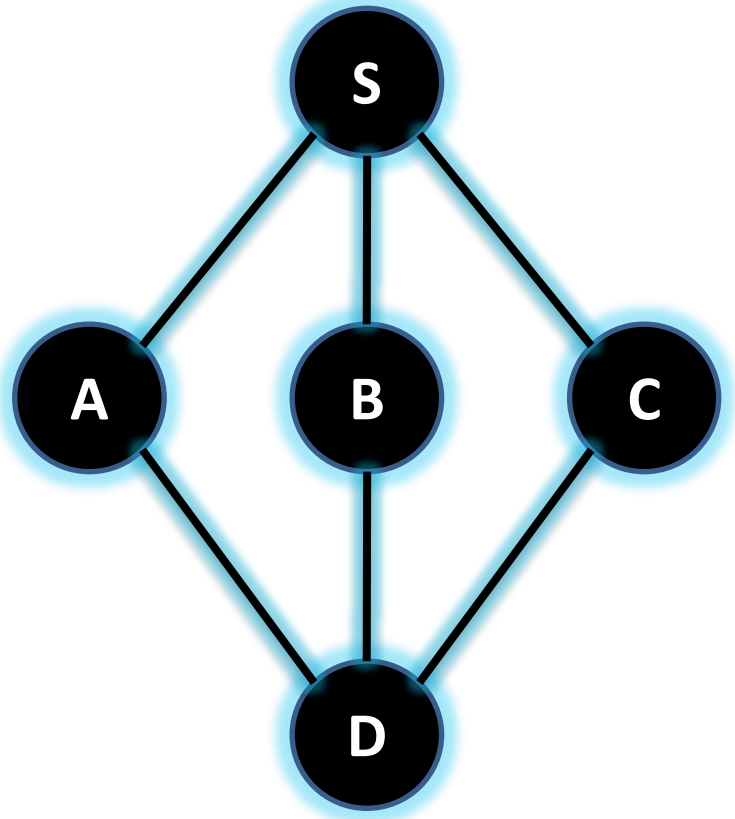
1)(DFS) algorithm traverses a graph in a depth ward motion

2)Uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Traversal and its Description

Determine the **Depth first traversal .**

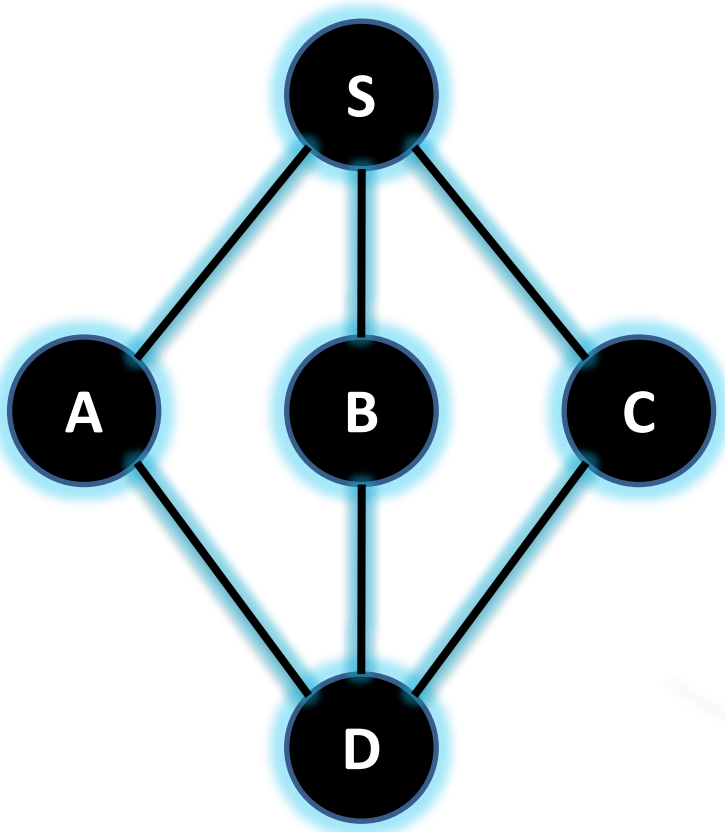


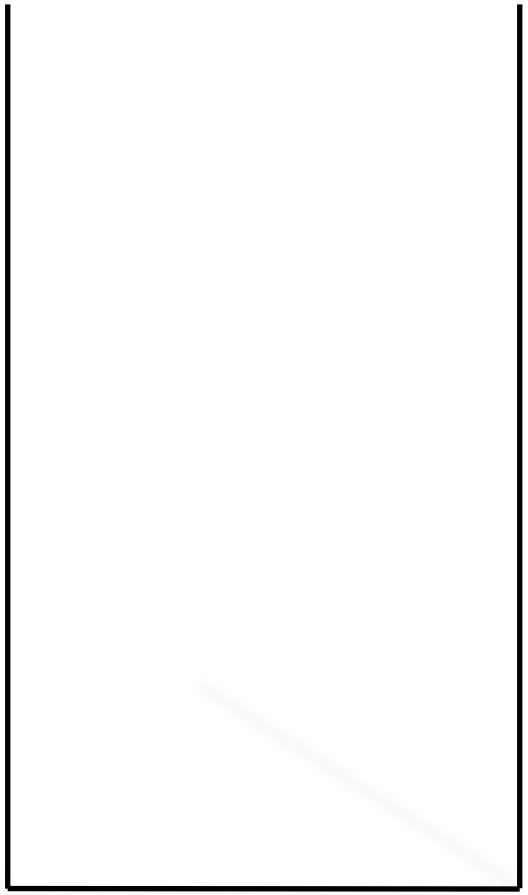
# DFS Traversal Rules

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

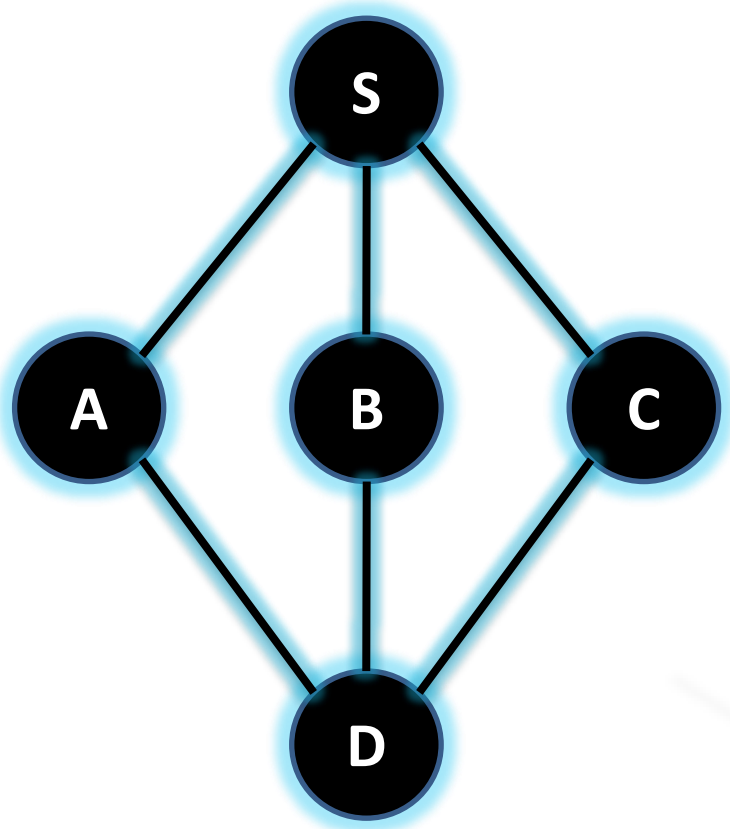
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

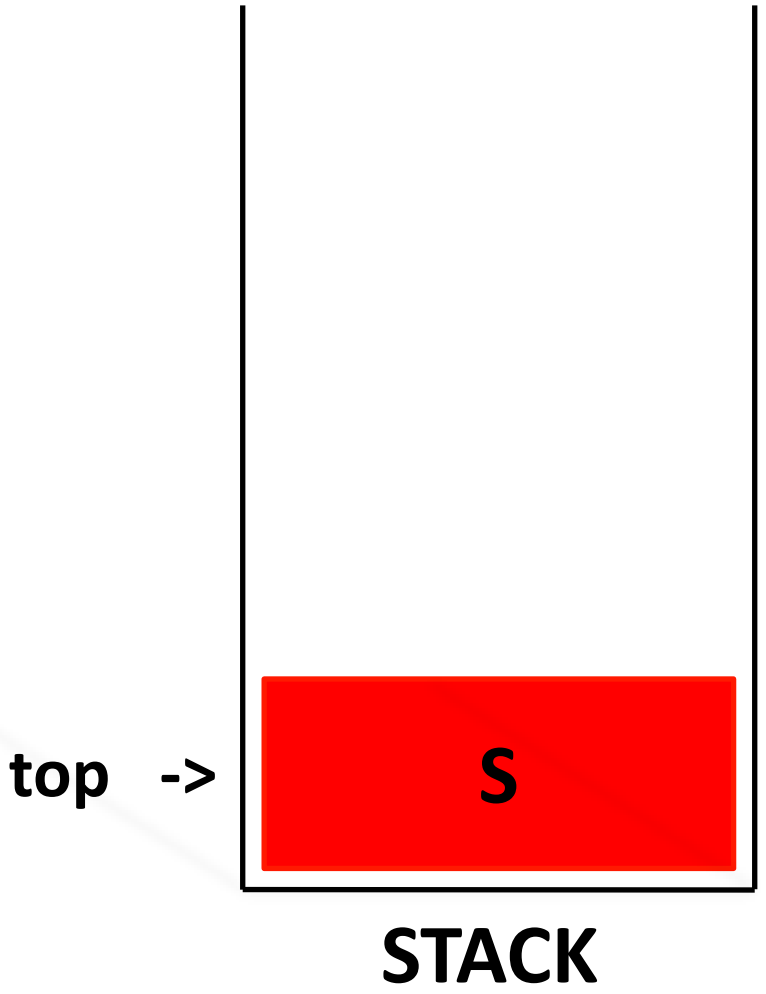


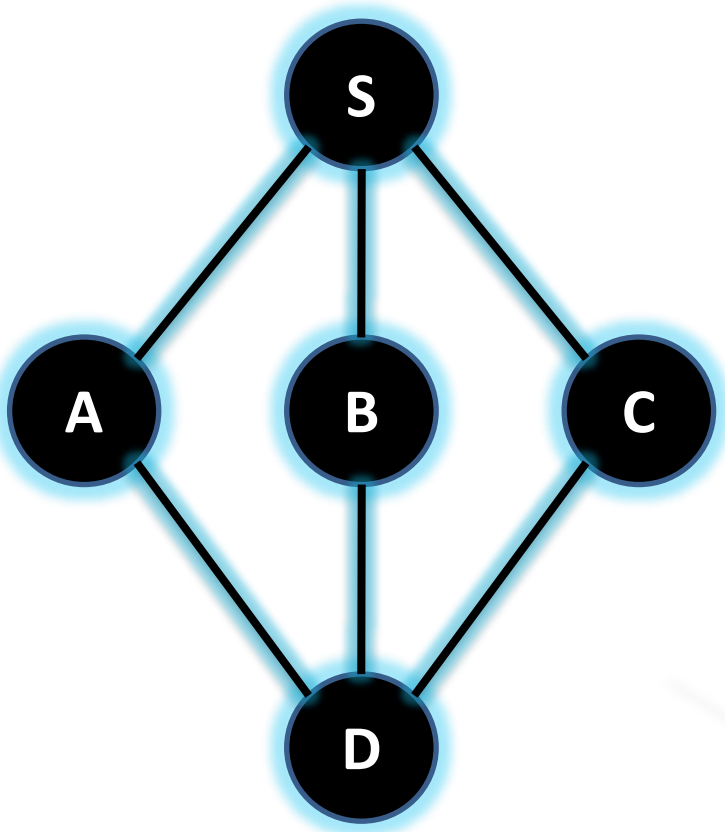
**STACK**

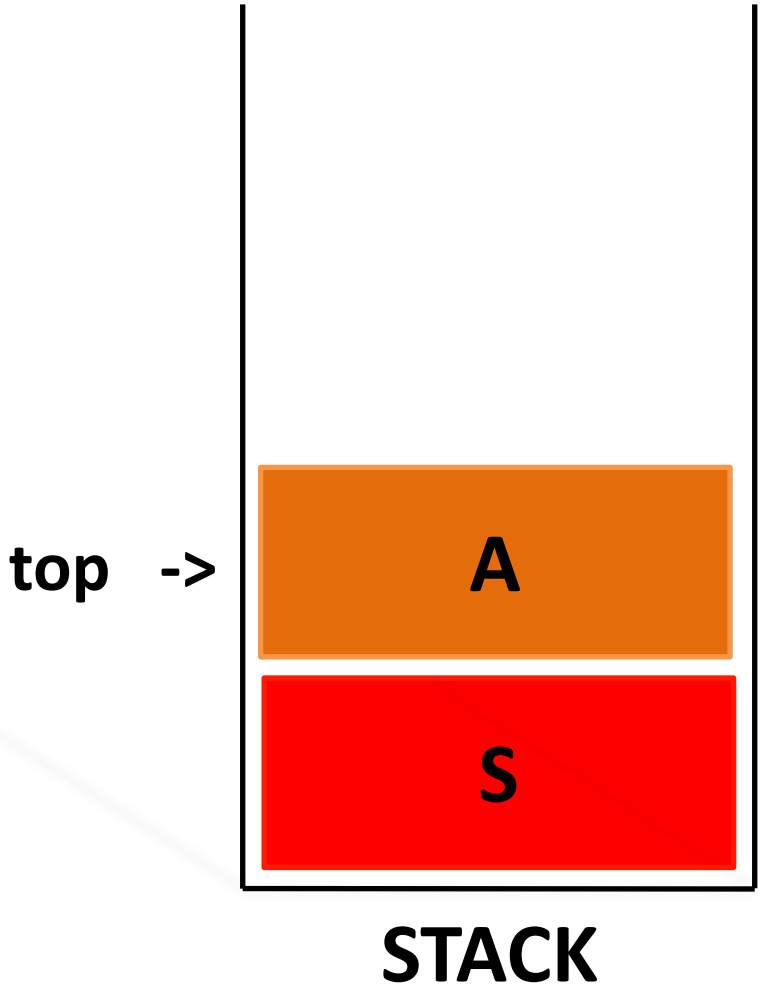
Initialize the stack



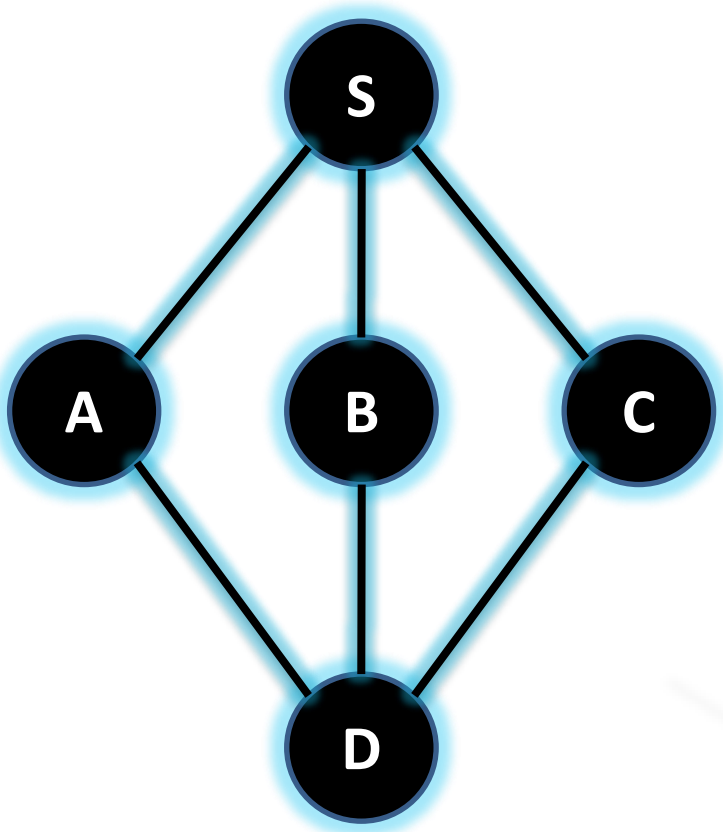



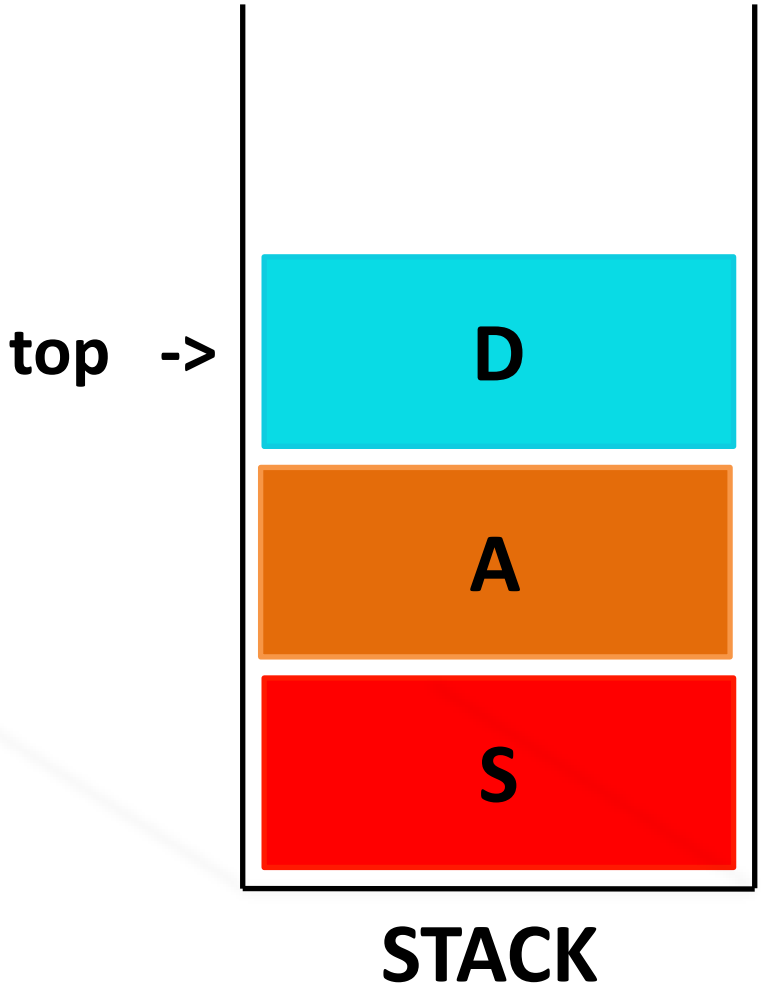

Mark S as visited and put onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.



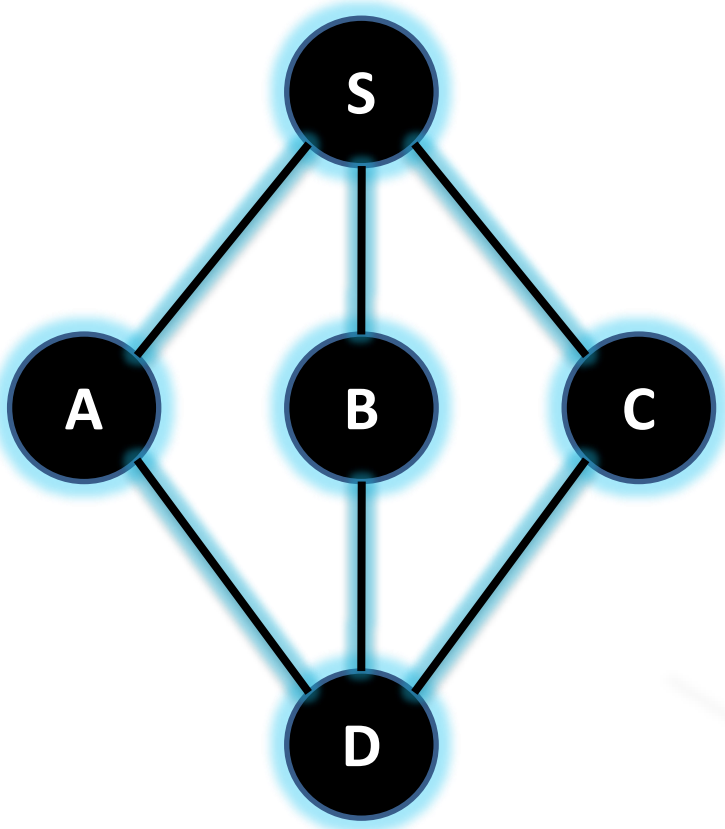



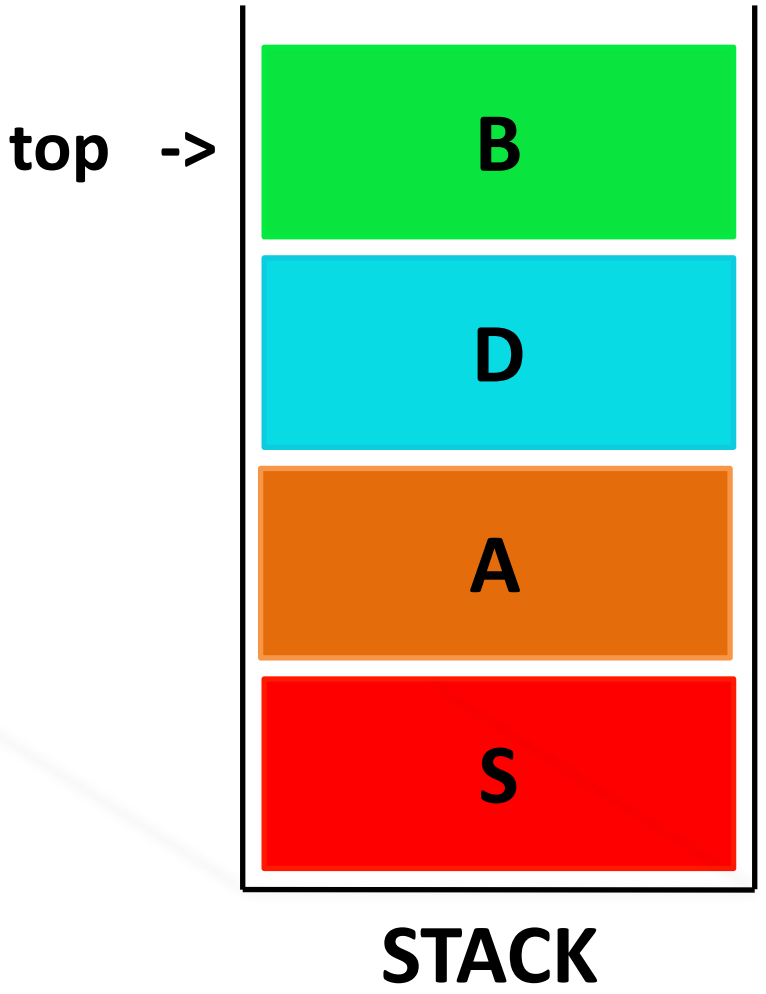

Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.



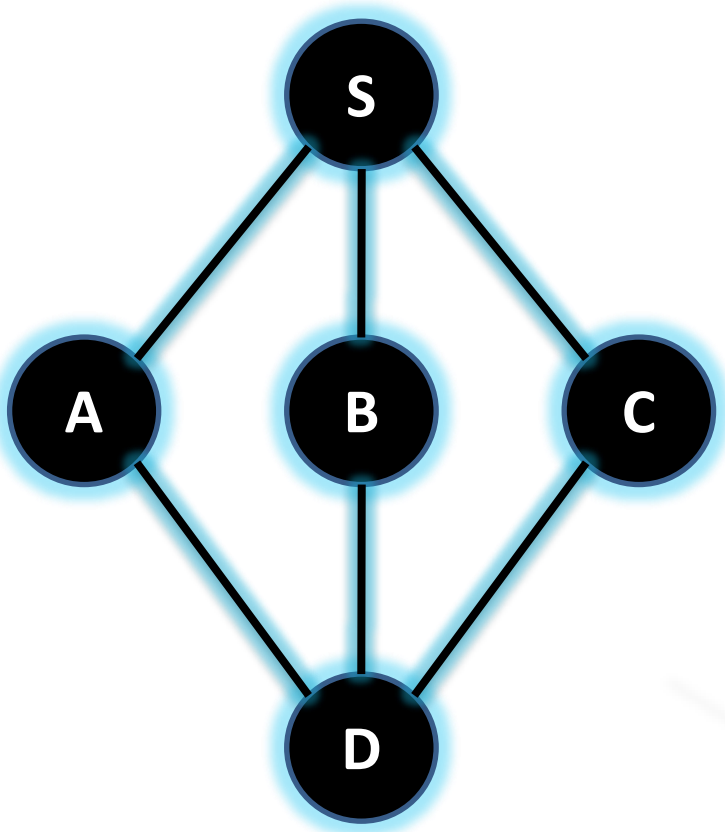



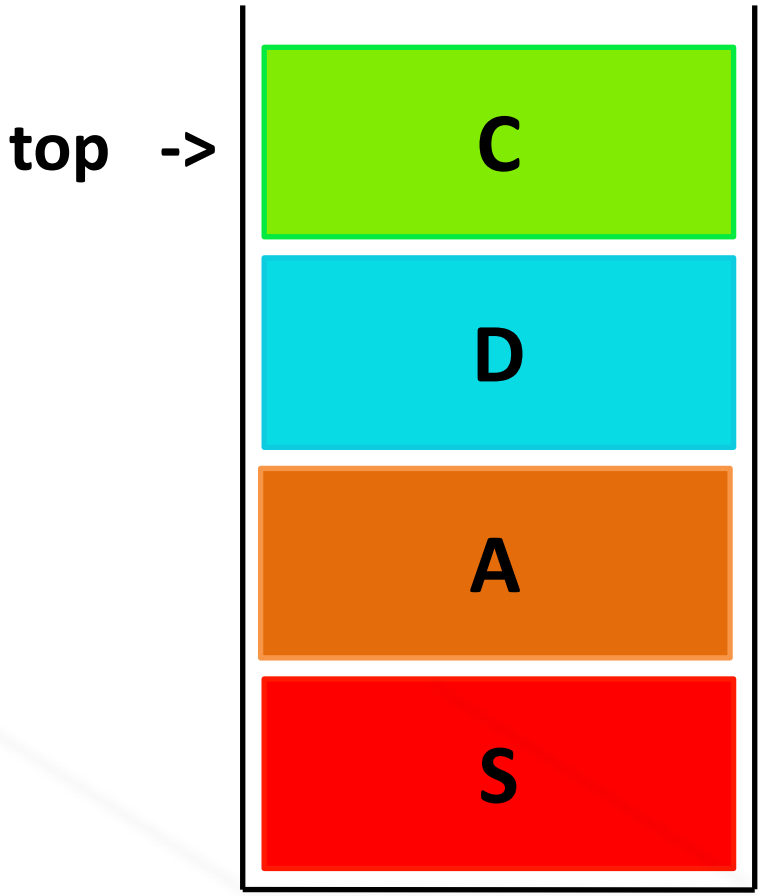

Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited.




We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.



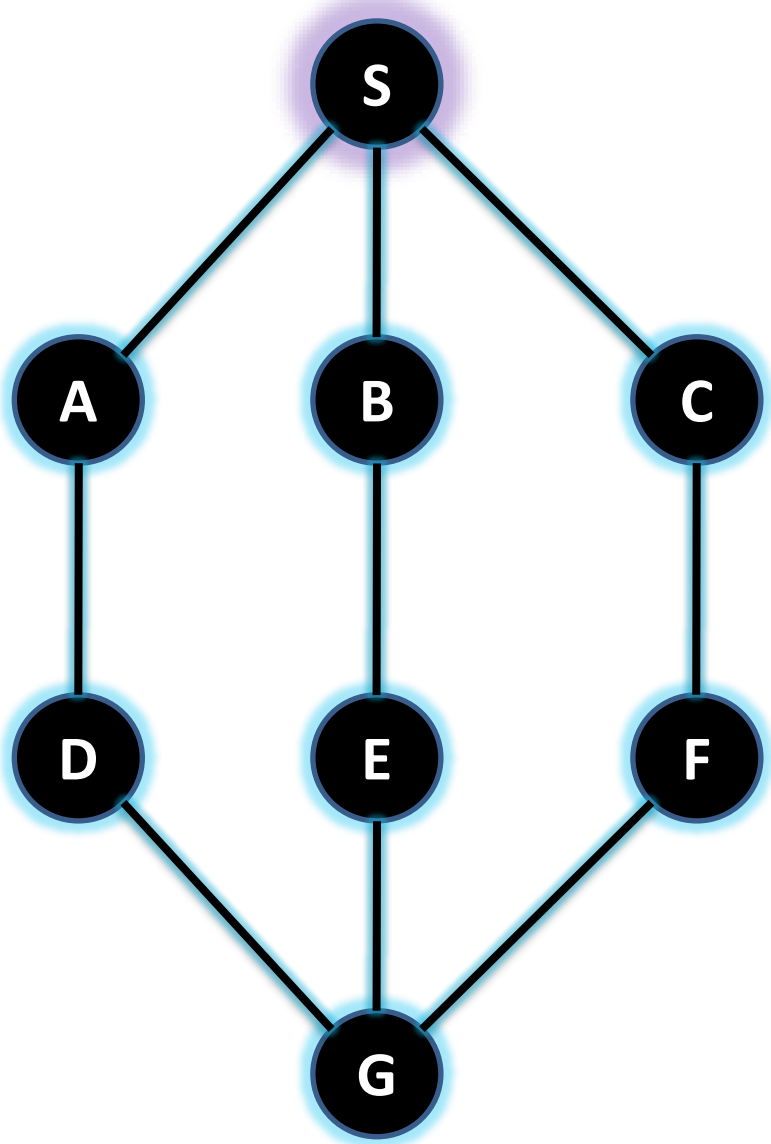




Now the only unvisited node from D is C.

# Breadth First Search

(BFS) algorithm traverses a graph in a breadth ward motion

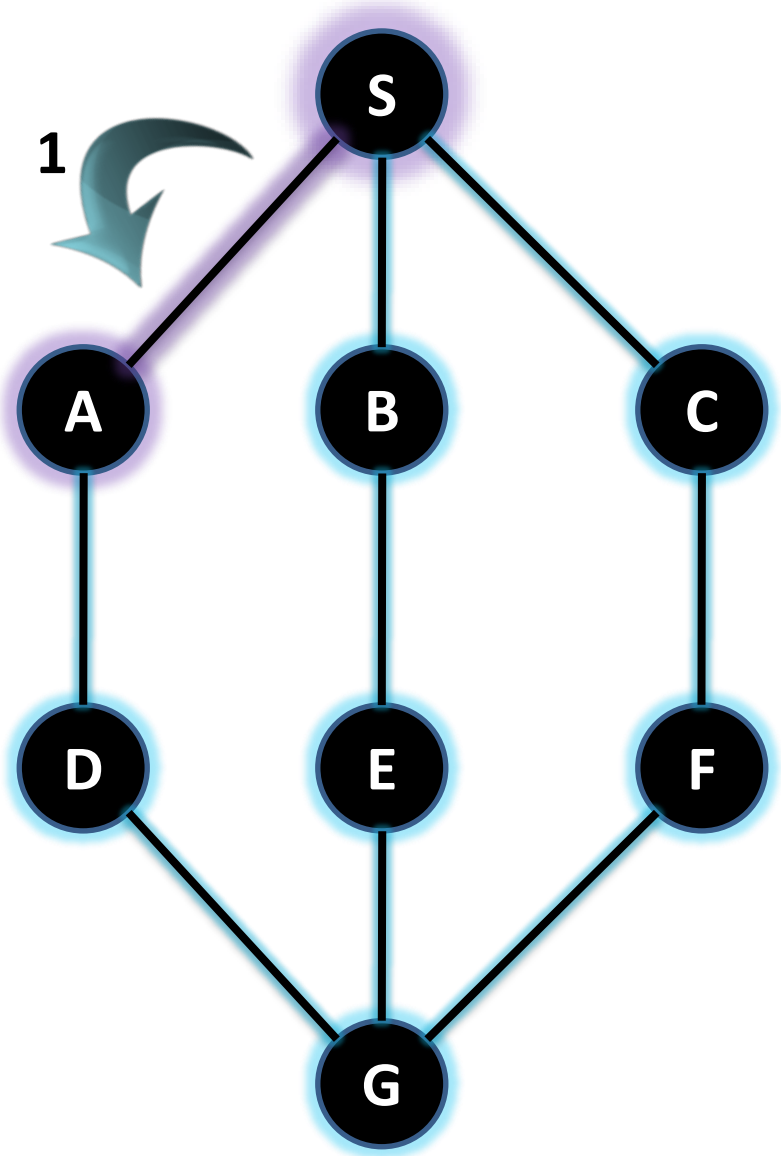
Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Breadth First Search

(BFS) algorithm traverses a graph in a breadth ward motion

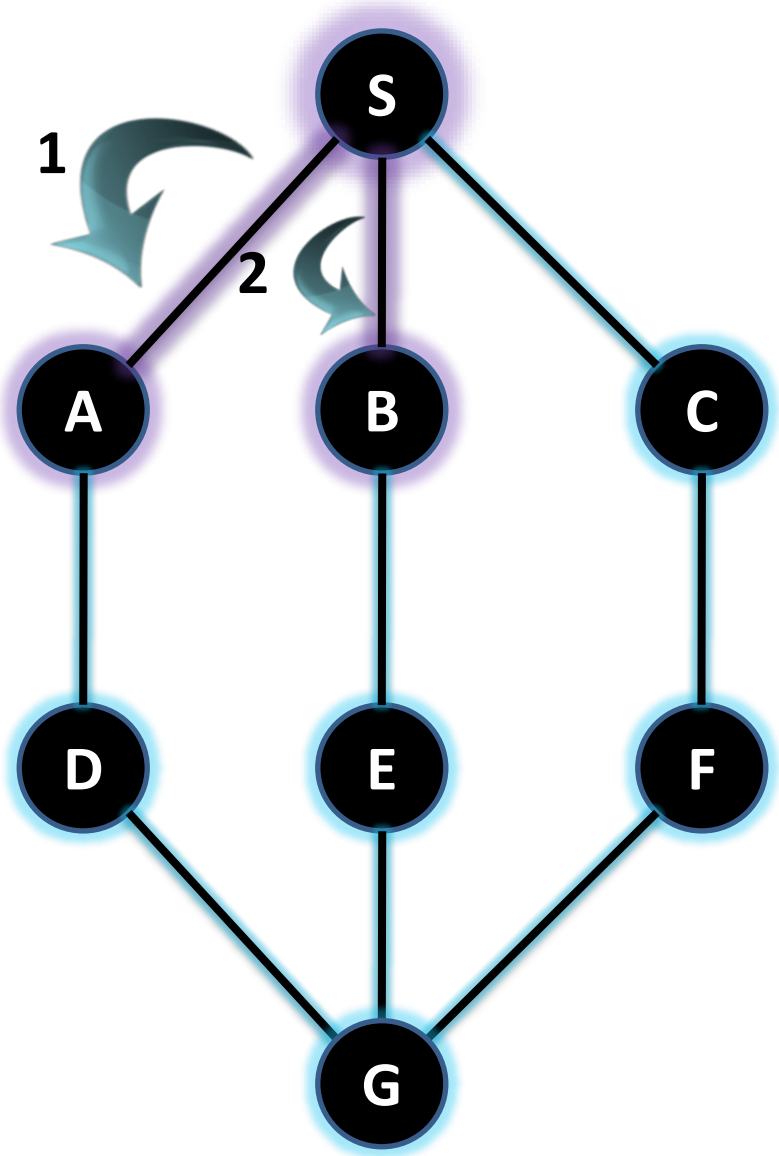
Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Breadth First Search

(BFS) algorithm traverses a graph in a breadth ward motion

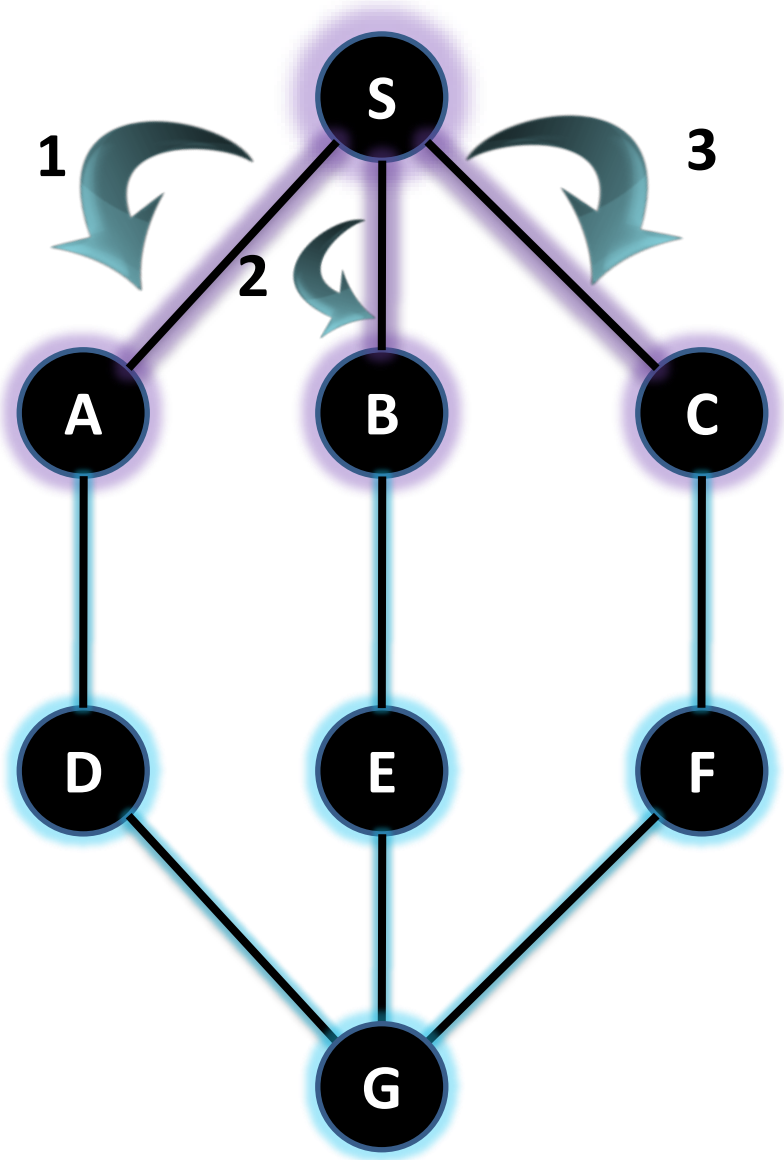
Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Breadth First Search

(BFS) algorithm traverses a graph in a breadth ward motion

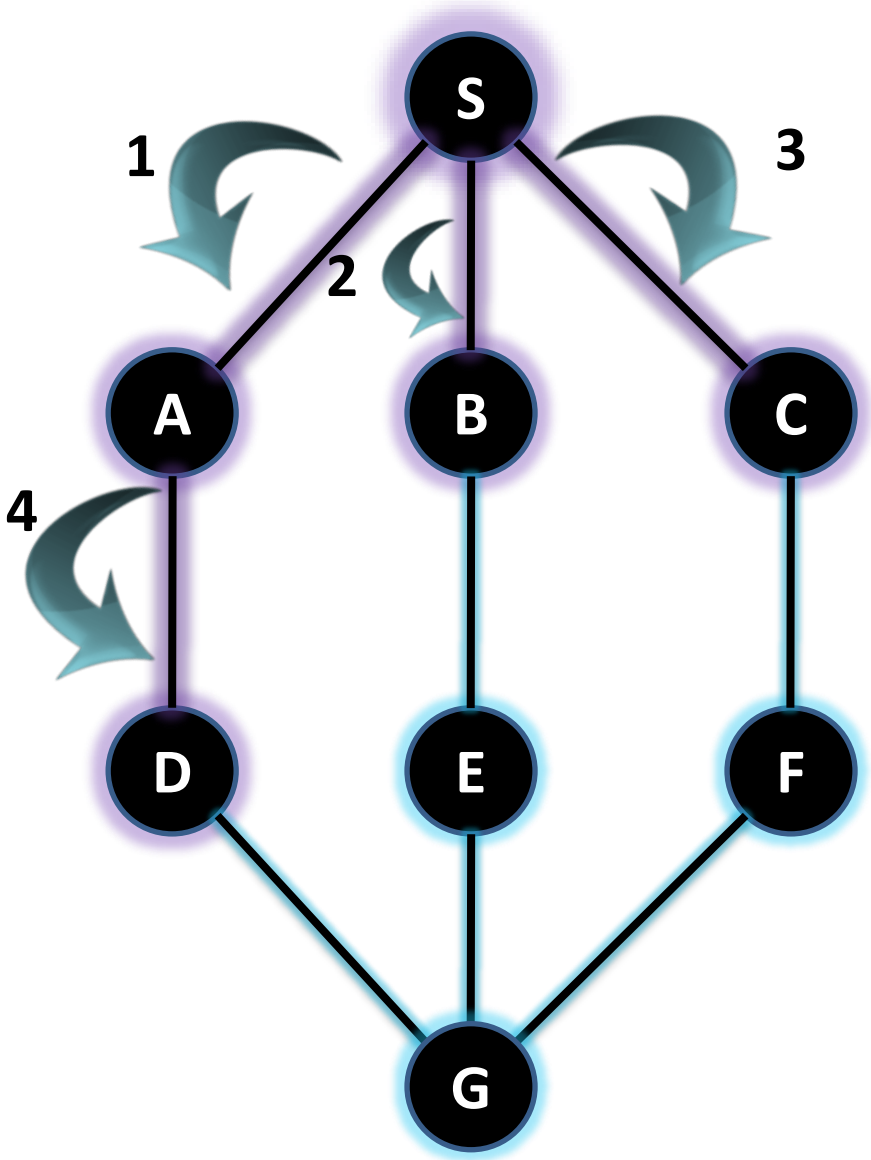
Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Breadth First Search

(BFS) algorithm traverses a graph in a breadth ward motion

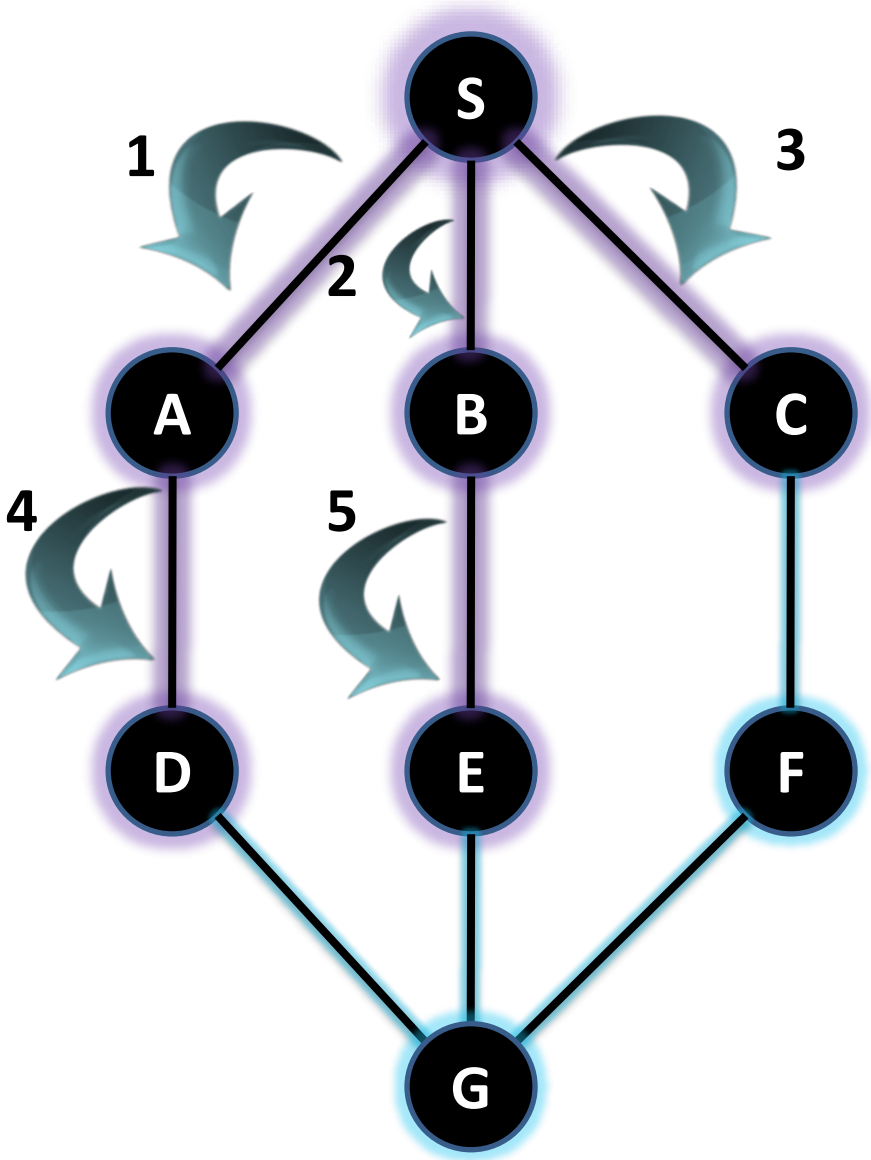
Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Breadth First Search

(BFS) algorithm traverses a graph in a breadth ward motion

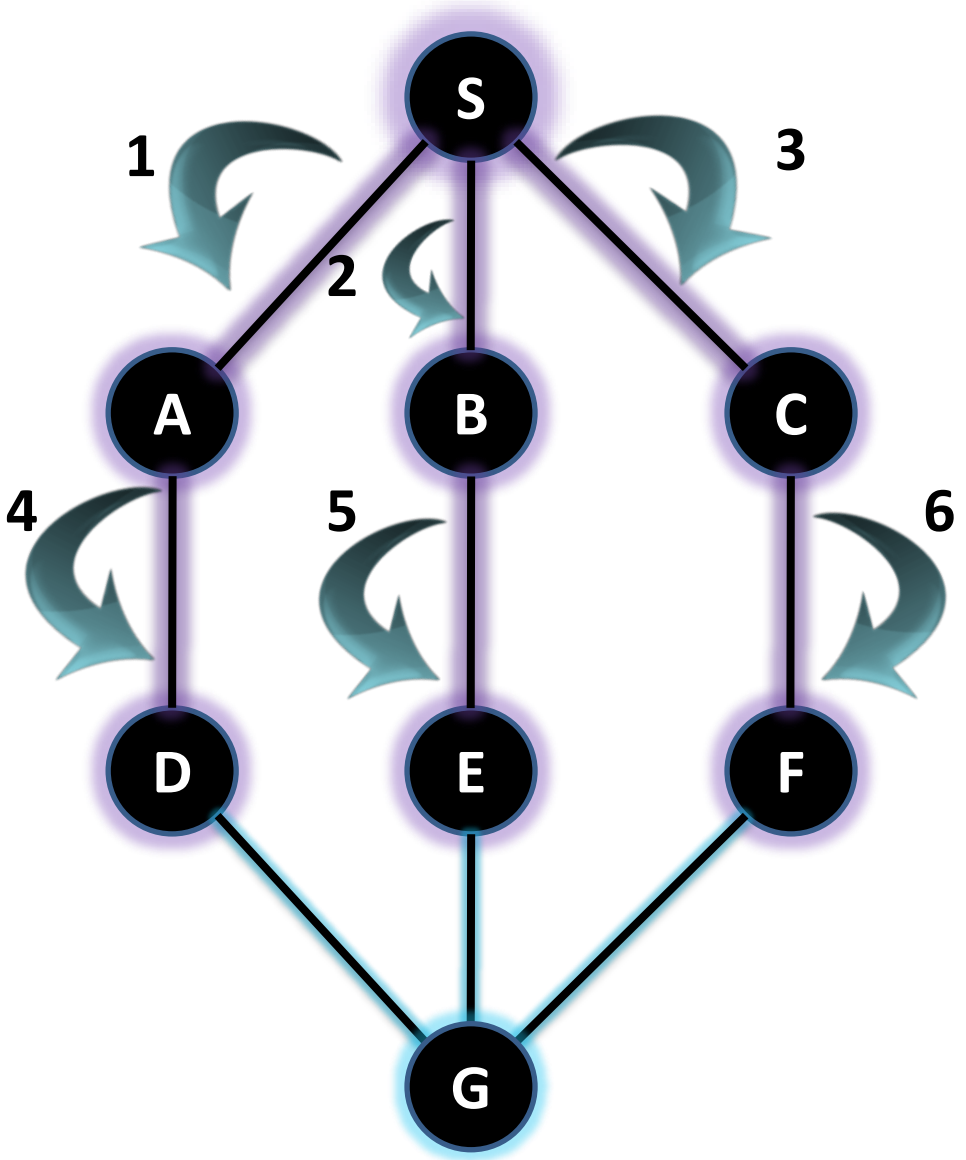
Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Breadth First Search

(BFS) algorithm traverses a graph in a breadth ward motion

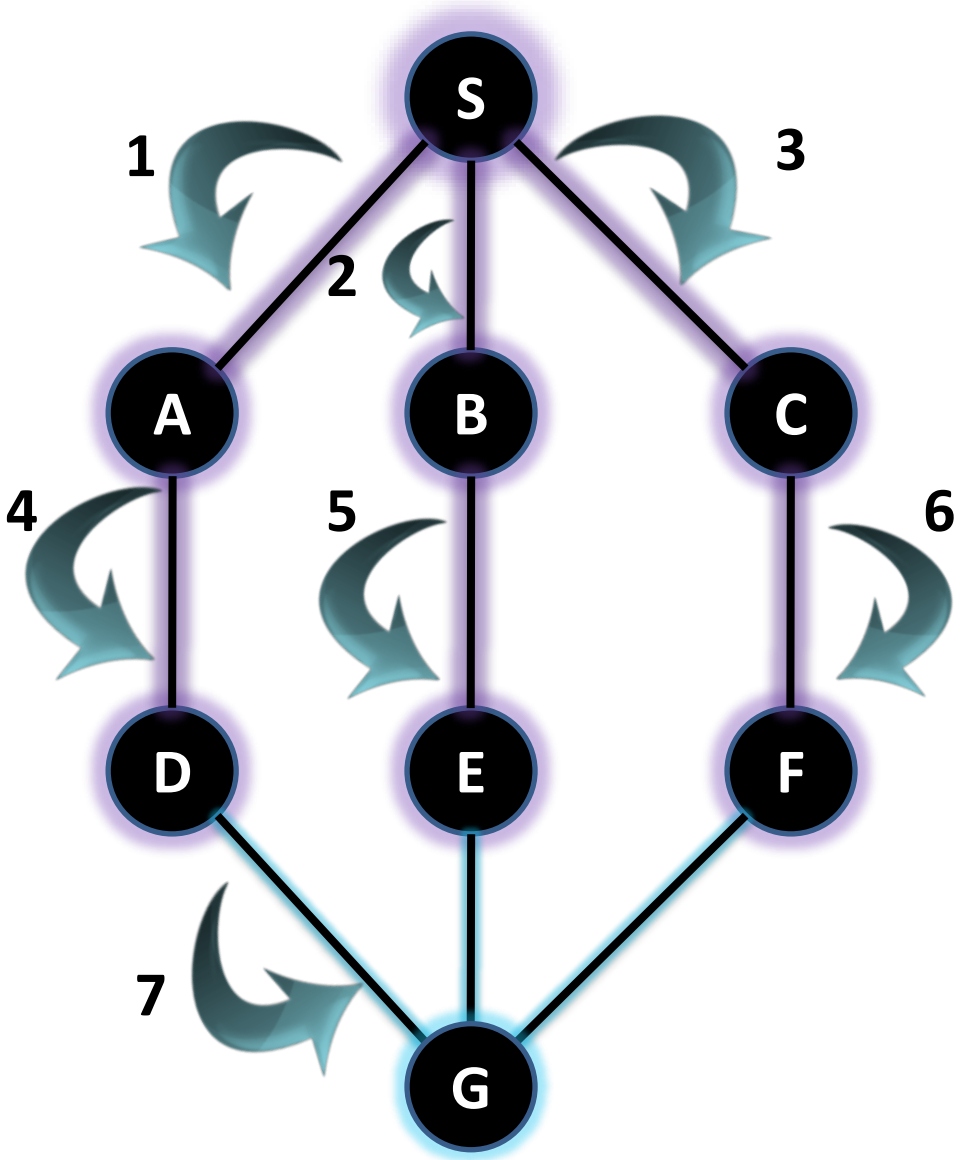
Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# Breadth First Search

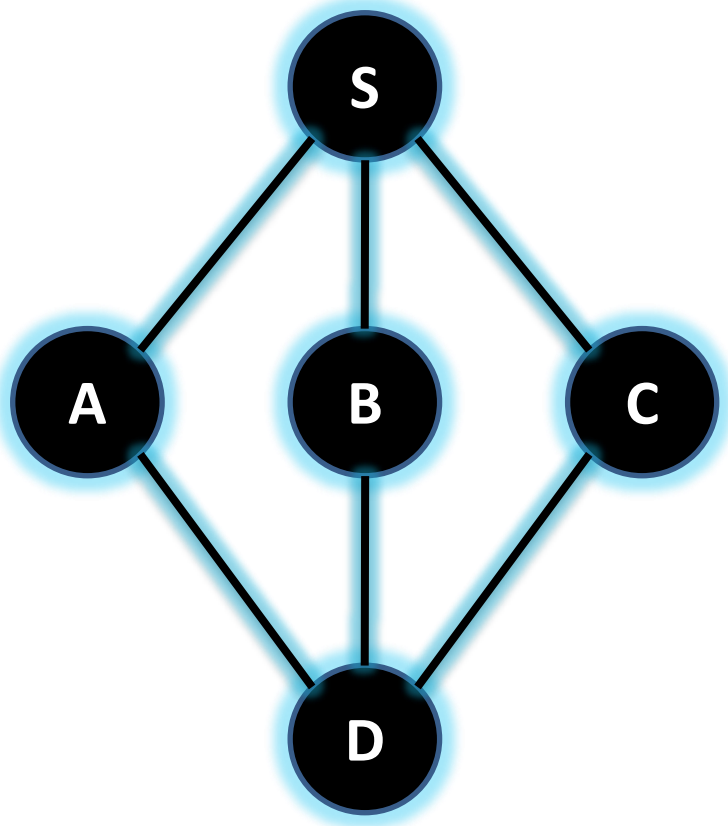
(BFS) algorithm traverses a graph in a breadth ward motion

Uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



# TRAVERSAL AND ITS DESCRIPTIONS

Determine the **Breadth first traversal .**

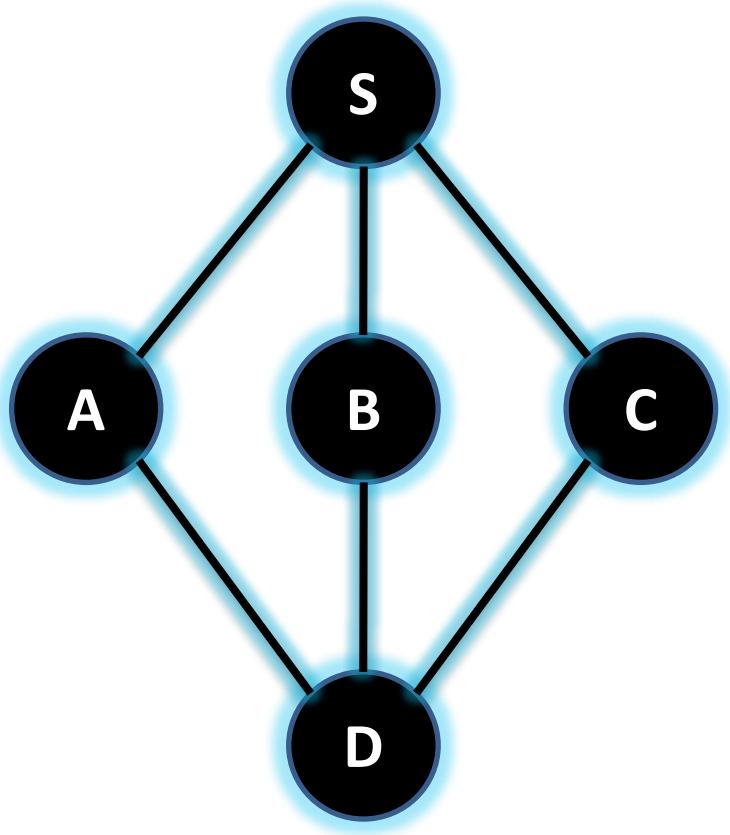


# BFS Traversal Rules

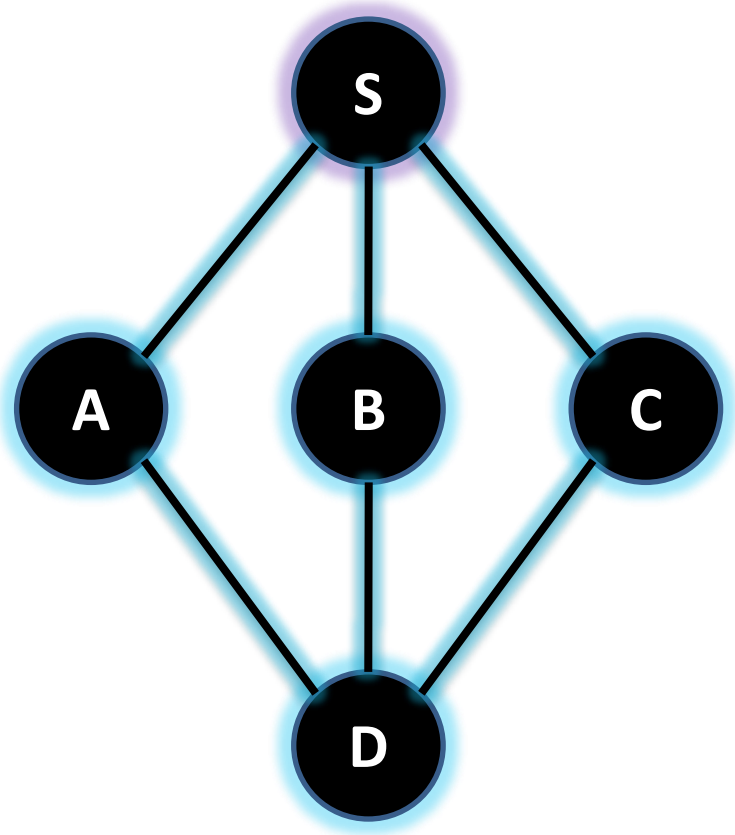
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.

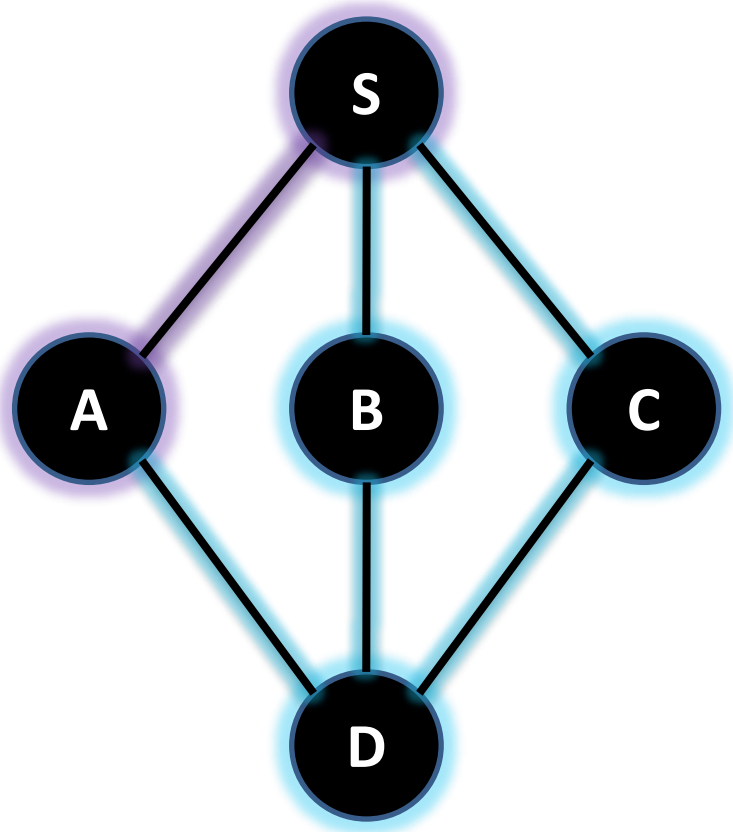
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.




Initialize the queue

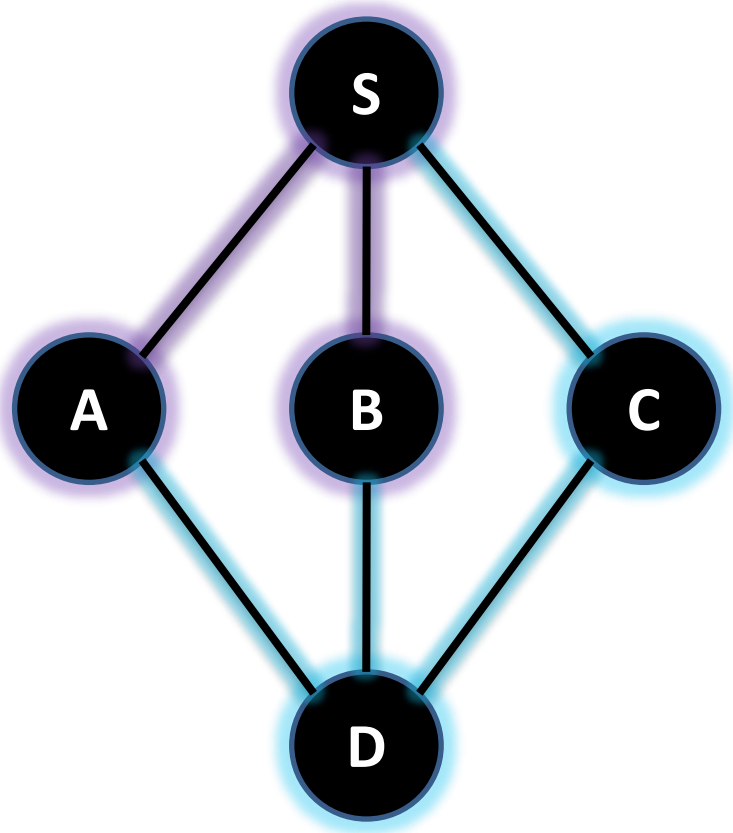



We start from visiting S(starting node), and mark it as visited.



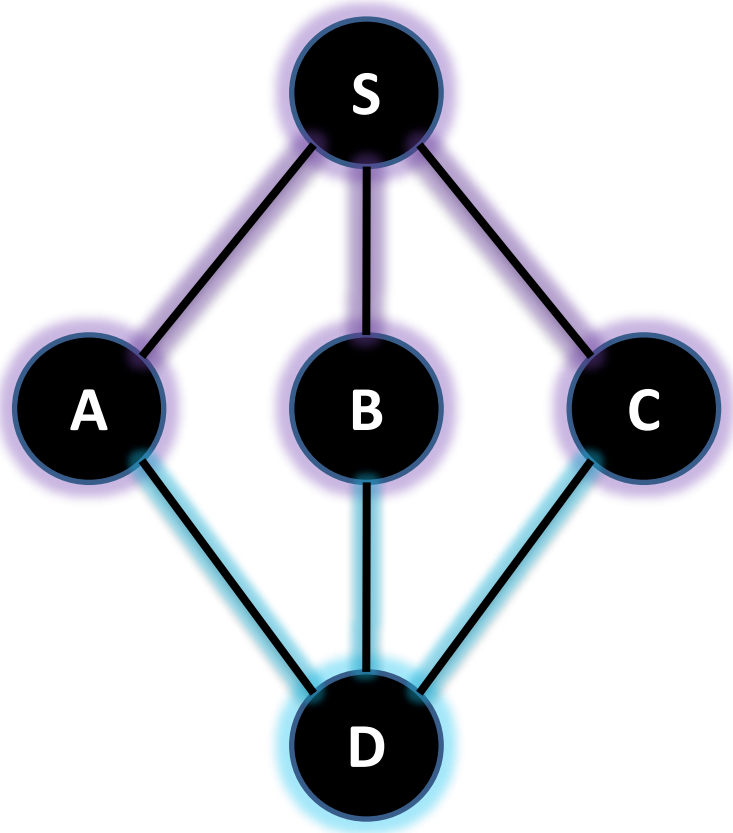


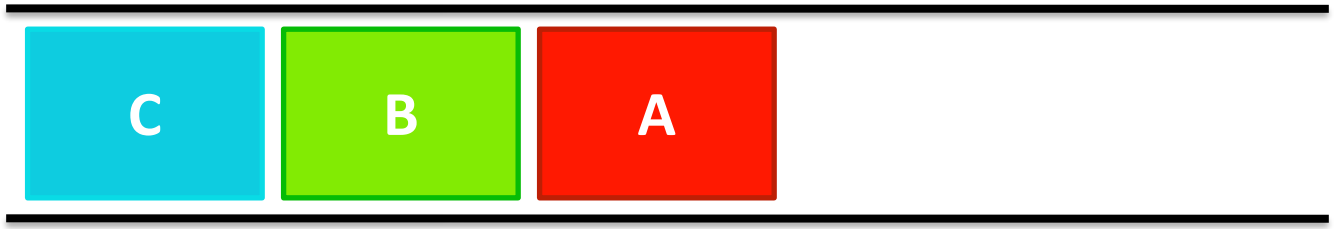
We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.



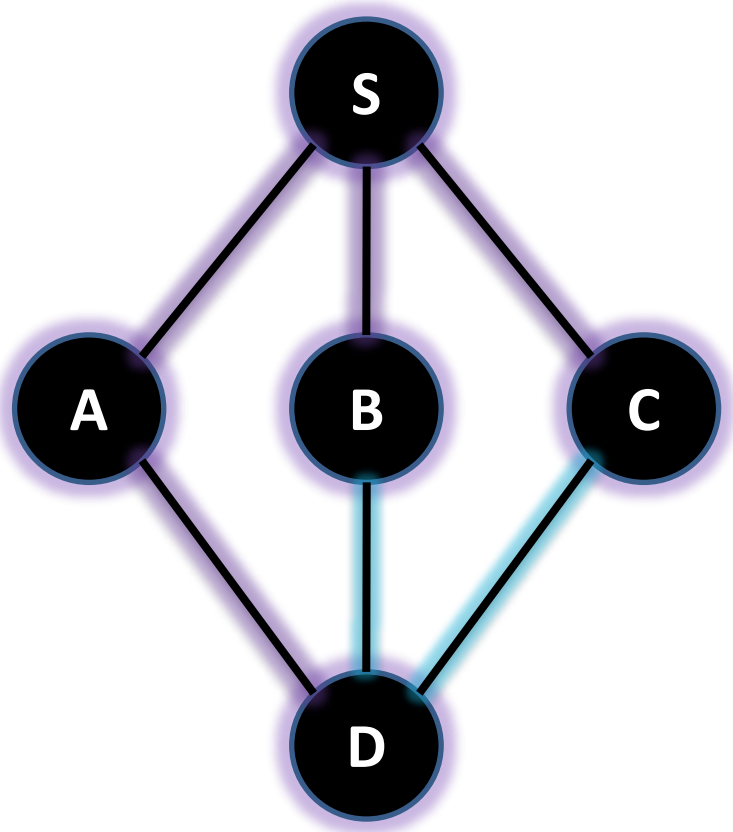


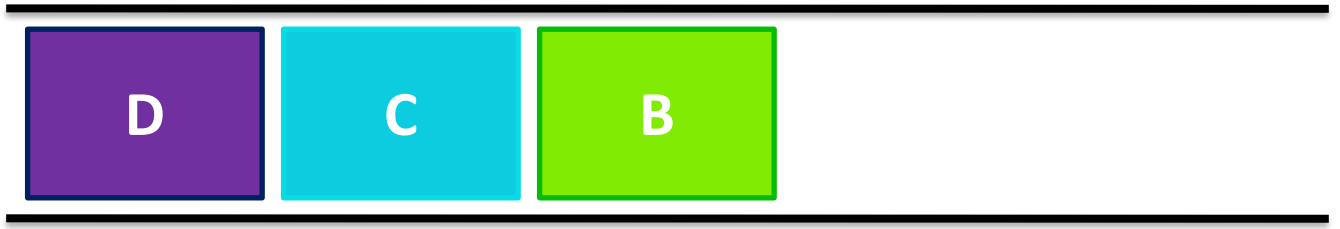
Next, the unvisited adjacent node from S is B.
We mark it as visited and enqueue it.




Next, the unvisited adjacent node from S is C.
We mark it as visited and enqueue it.




From A we have D as unvisited adjacent node.
We mark it as visited and enqueue it.

**Q1. What is a common way to represent a graph?**

**A)Adjacency Matrix**

**B)Adjacency List**

**C)Both A and B**

**D)None of the above**

**Q2. What is the maximum number of edges in a directed graph with 6 vertices that has no self loops?**

**A) 30**

**B) 12**

**C) 36**

**D) 11**

**Q3. What makes a graph different from a tree?**

**A) A graph can not have cycles**

**B) A graph can not be disconnected**

**C) Both A and B**

**D) None of the above**

**Q4. Which of the following best describes the tightest bound on the space complexity of a graph represented by an adjacency matrix?**

**A) O(V∗V)**

**B) O(V)**

**C) O(E)**

**D) O(V+E)**

**Q5. We want to check if there is an edge between vertices i and j in a graph. This operation can be done more efficiently in the worst case when using an adjacency matrix representation than an adjacency list.**

**A) True**

**B) False**

**Q6. Is an adjacency matrix symmetric for all types of graphs?**

**A) Yes**

**B) No**

**Q7. Which data structure is used in the breadth first search of a graph to store nodes ?**

**A) Array**

**B) Stack**

**C) Queue**

**D) Heap**

**Q8. Which data structure is used in the depth first search of a graph to store nodes ?**

**A) Array**

**B) Stack**

**C) Queue**

**D) Heap**

**9. Breadth First Search is equivalent to which of the traversal in the Binary Trees?**

**A) Pre-order Traversal**

**B) Post-order Traversal**

**C) Level-order Traversal**

**D) In-order Traversal**

**10. Time Complexity of Breadth First Search is? (V – number of vertices, E – number of edges)**

a) O(V + E)

b) O(V)

c) O(E)

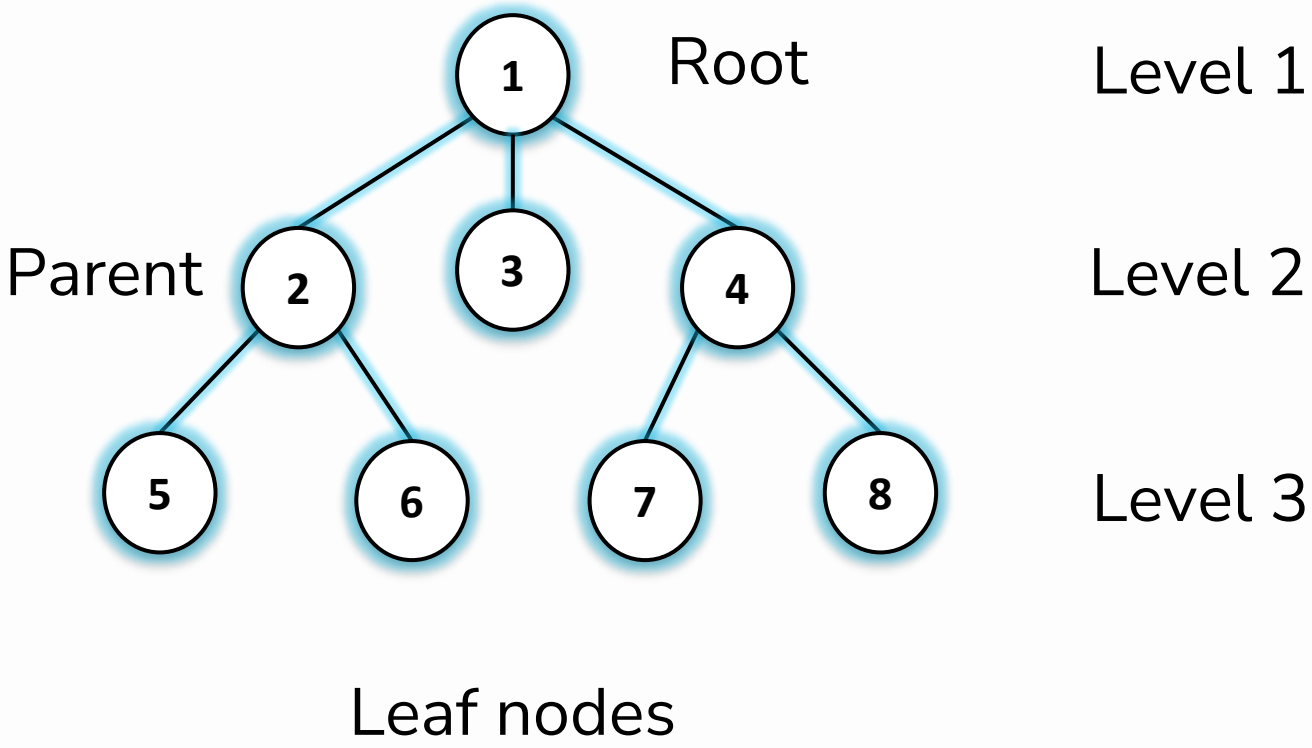
d) O(V*E)

# THANK YOU

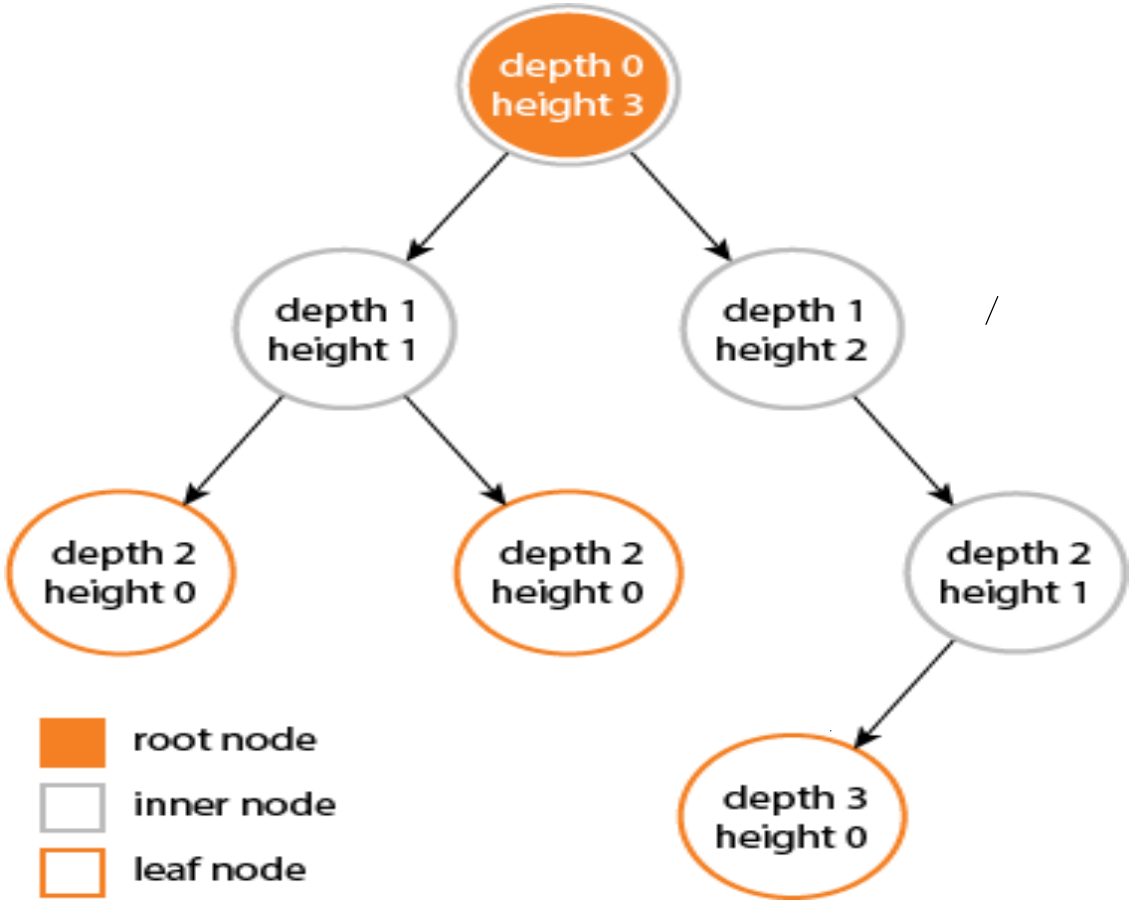
# Trees

- Non-Linear Hierarchical Data structure.
- Collection of nodes connected by edges.
- One of the nodes is designated as Root node and the remaining nodes are called child nodes or the leaf nodes of the root node.

# Trees

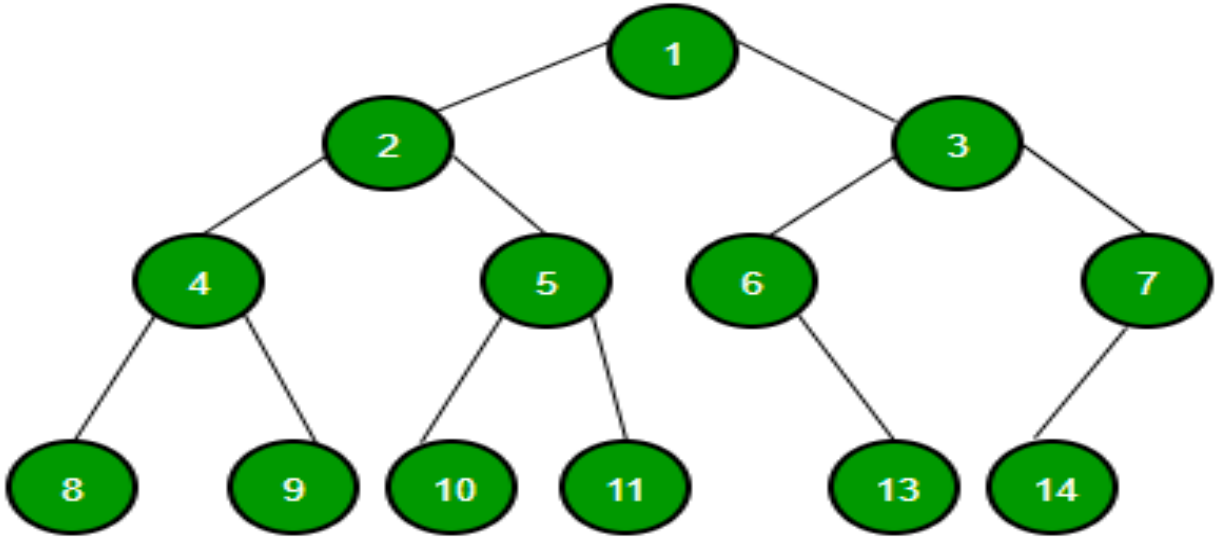


# Height and Depth of the Tree



# Binary Tree

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



# Binary Tree

**Binary Tree Representation in Python:**

The node class represents the structure of a particular node in the binary tree. The attributes of this class are values, left, right.

***Syntax:*** *binarytree.Node(value, left=None, right=None)*

***Parameters:***
***value:*** *Contains the data for a node. This value must be number.*
***left: Contains the details of left node child.***
***right: Contains details of the right node child.***

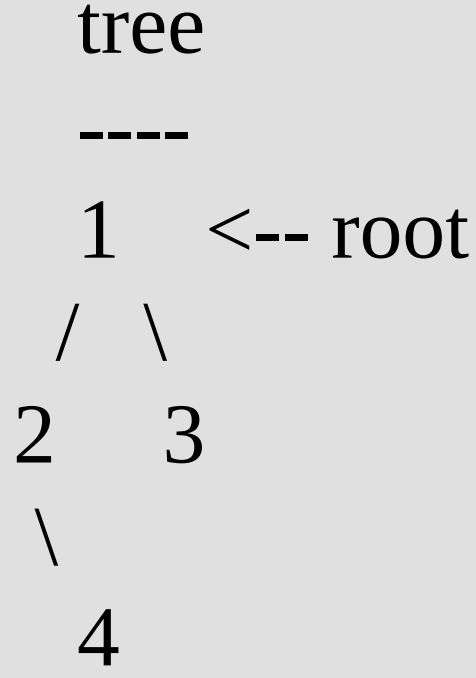
***Note:*** pip install binarytree

# Binary Tree

**First Simple Tree in Python**

Let us create a simple tree with 4 nodes in C.

The created tree would be as following.



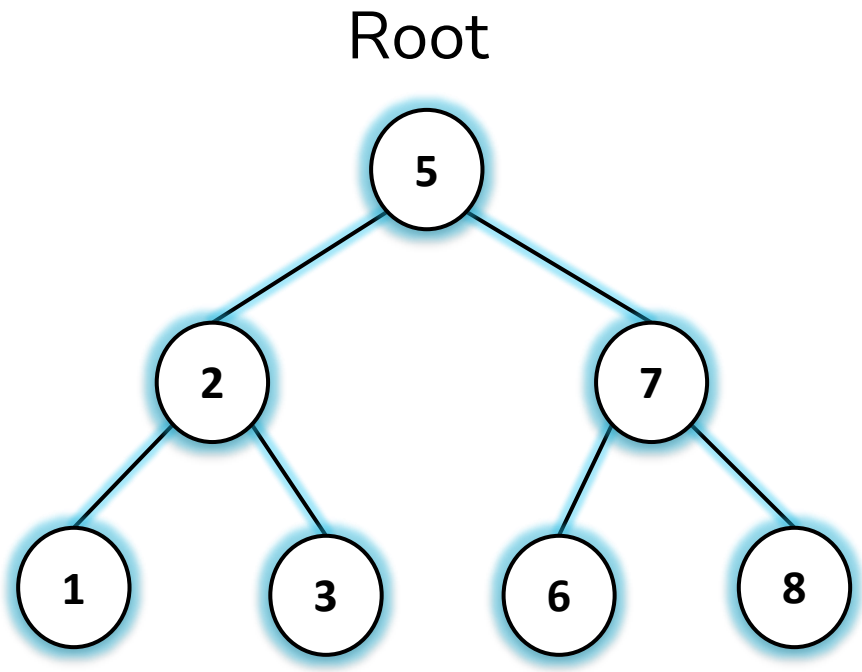
```
#pip install binarytree

from binarytree import Node
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.right = Node(4)

# Getting binary tree
print('Binary tree :', root)

# Getting list of nodes
print('List of nodes :', list(root))
```

# Level-order Traversal

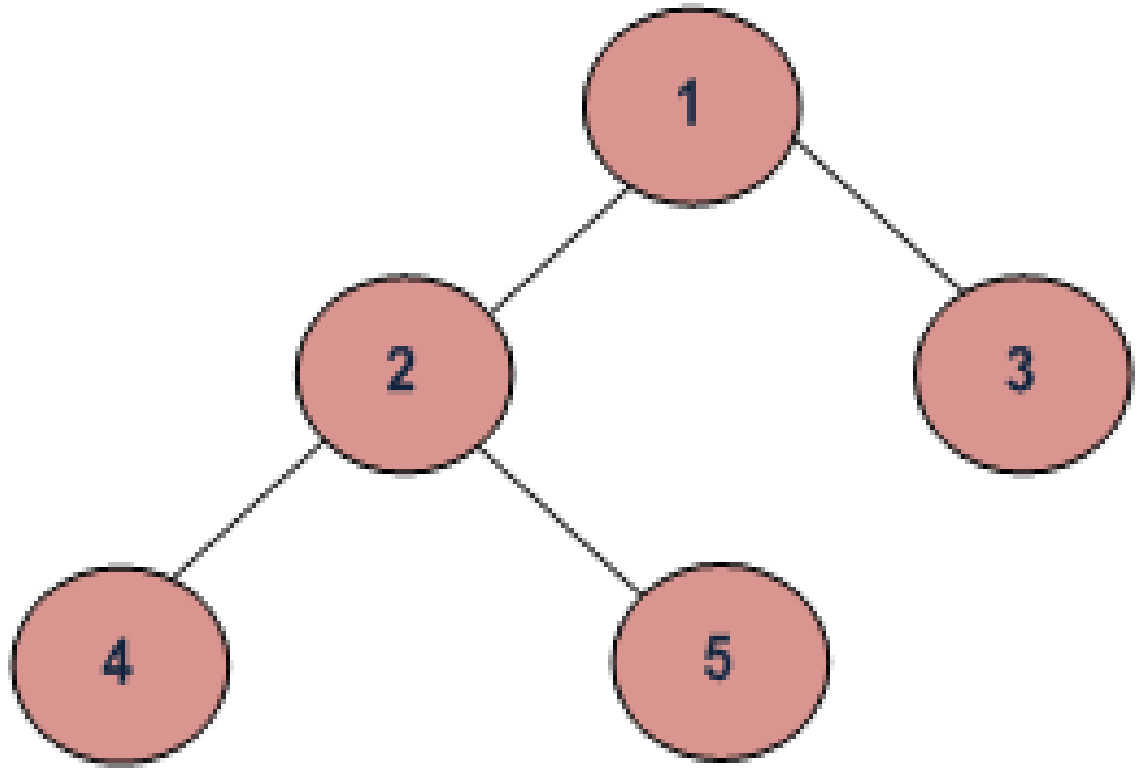



5 2 7 1 3 6 8

# Traversals

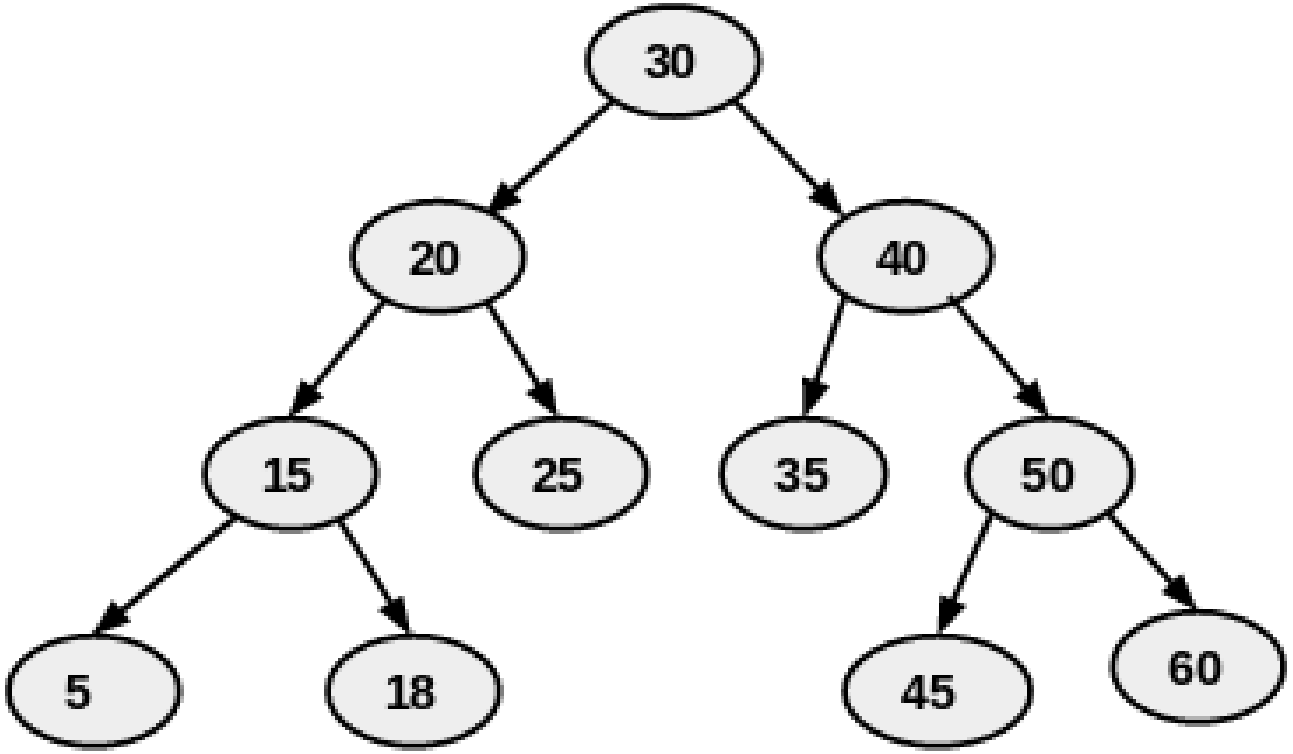
**Tree Traversals (Inorder, Preorder and Postorder)**

Following are the generally used ways for traversing trees.



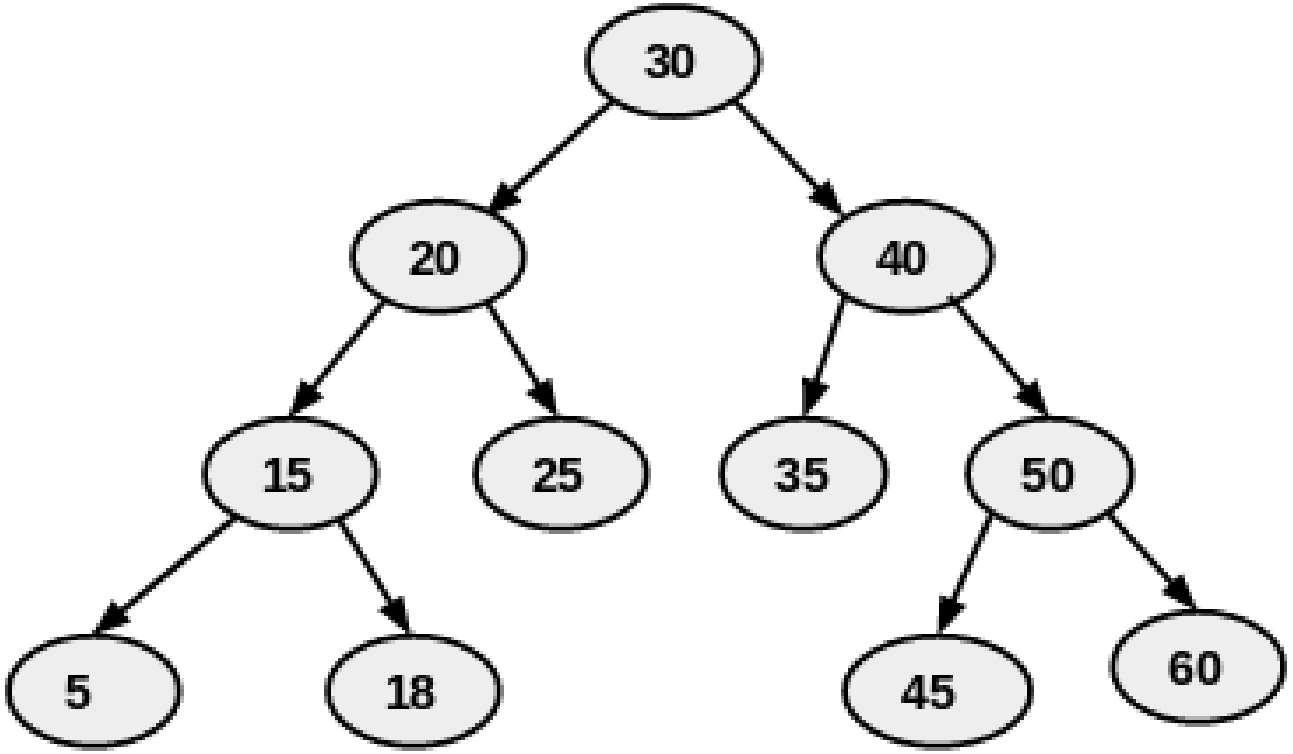
# Inorder Traversal

**Example:**



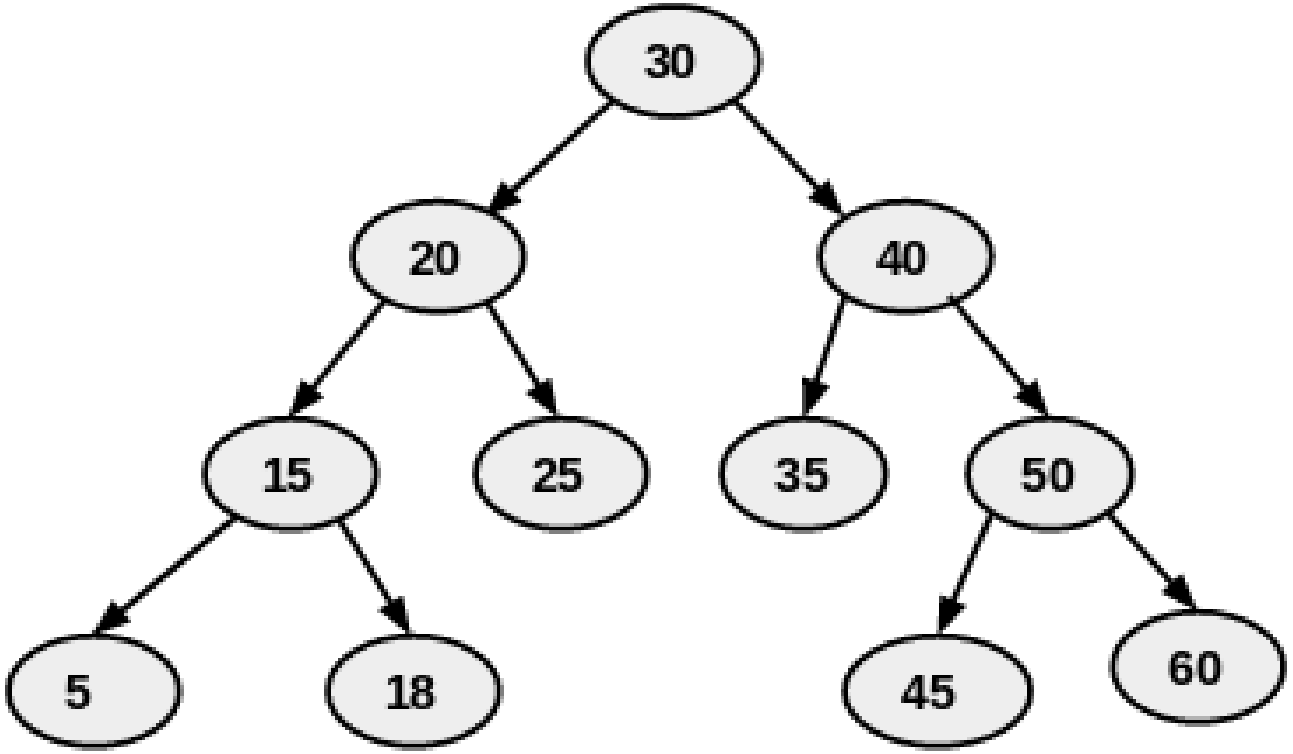
# Preorder Traversal

**Example:**



# Postorder Traversal

**Example:**

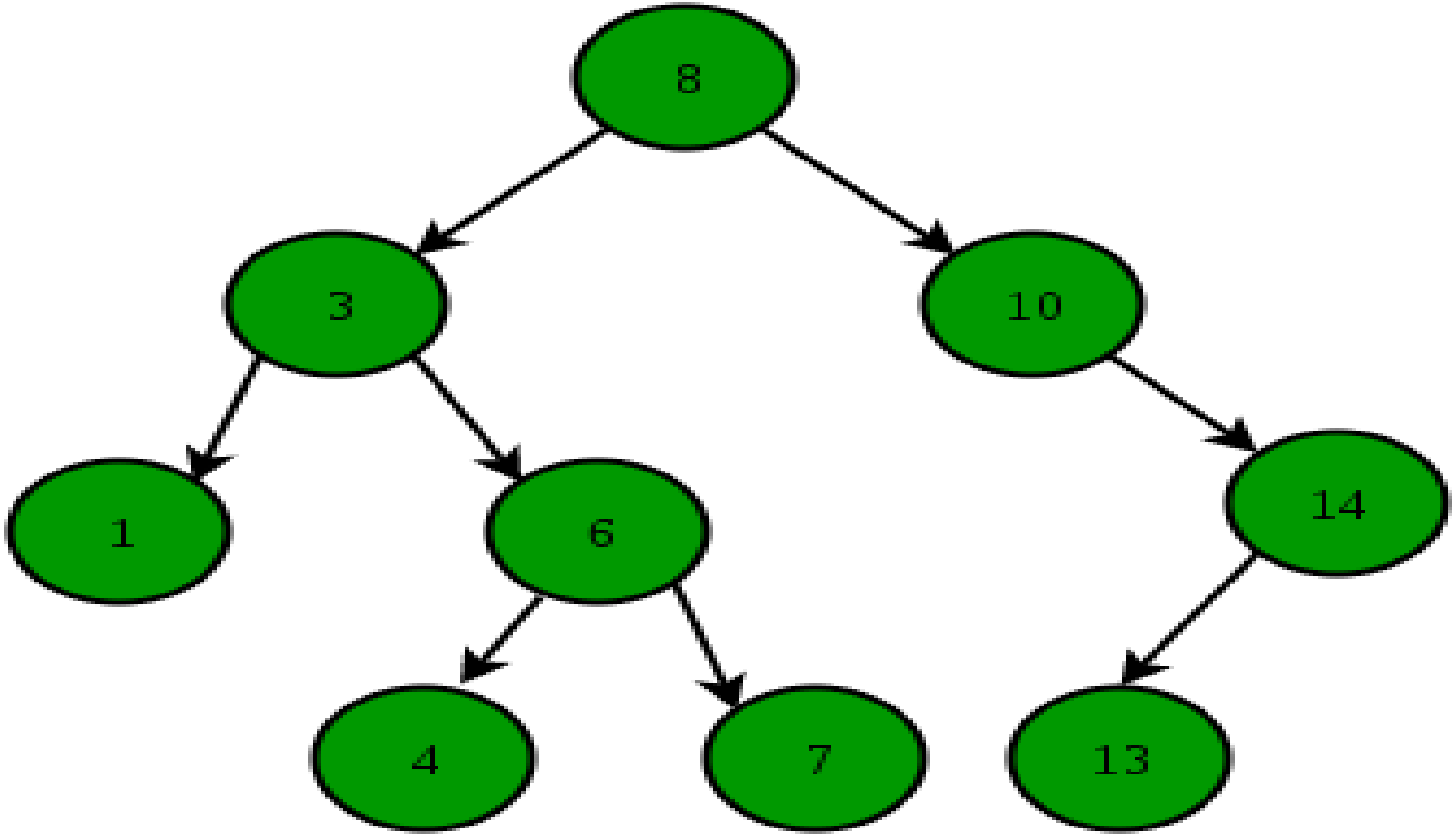


# Binary Search Tree(BST)

**Binary Search Tree** is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

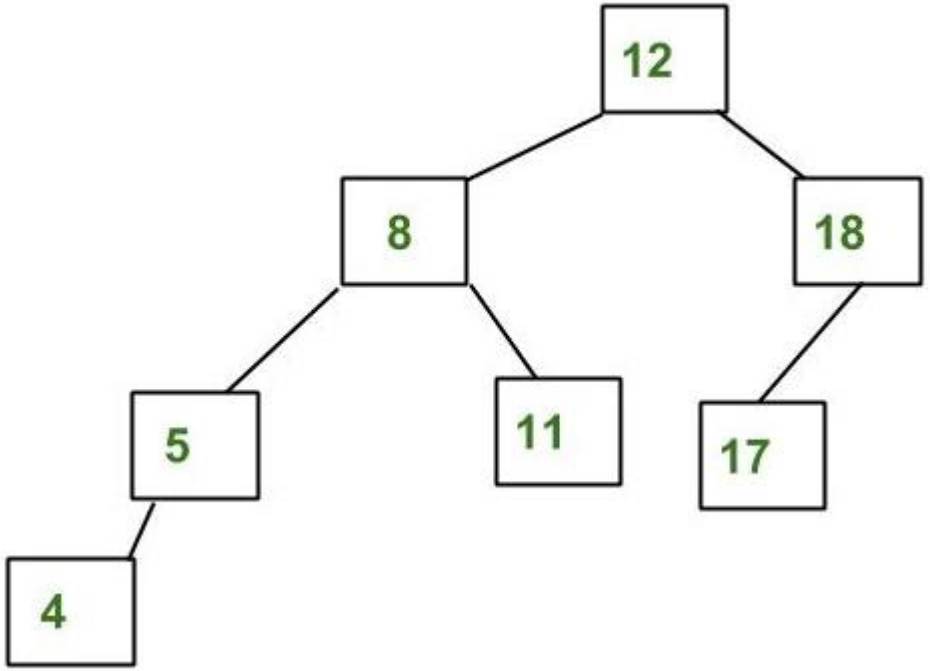
# Binary Search Tree(BST)



# AVL Tree

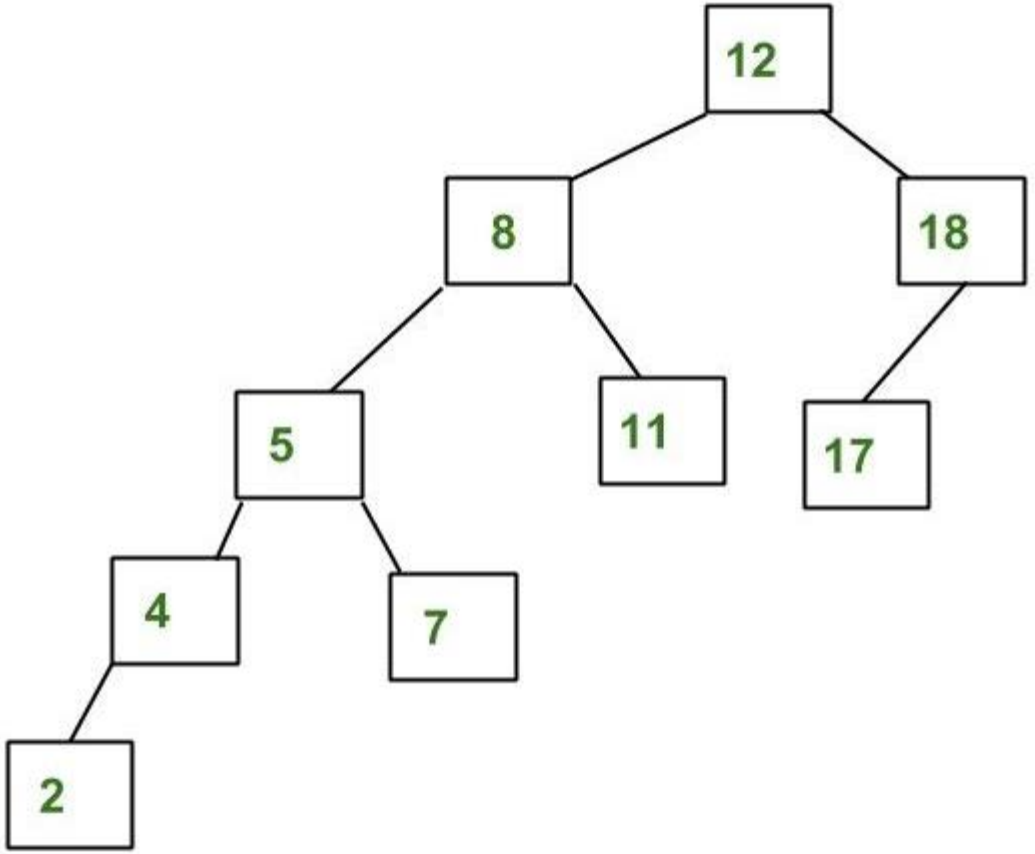
AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

**Example:**



# AVL Tree or Not ?

**Example:**



# Why AVL Tree ?

AVL tree **controls the height of the binary search tree.**

The time taken for all operations in a binary search tree of height h is O(h).

1. **Which of the following is false about a binary search tree?**

a) The left child is always lesser than its parent

b) The right child is always greater than its parent

c) The left and right sub-trees should also be binary search trees

d) In order sequence gives decreasing order of elements

## 2. What is the speciality about the inorder traversal of a binary search tree?

a) It traverses in a non increasing order

b) It traverses in an increasing order

c) It traverses in a random fashion

d) It traverses based on priority of the node

# 3. What does the following piece of code do?

```
public void func(Tree root)
{
   func(root.left());
   func(root.right());
   System.out.println(root.data());
}
```

a) Preorder traversal
b) Inorder traversal
c) Postorder traversal
d) Level order traversal

# 4. What does the following piece of code do?

```
public void func(Tree root)
{
 System.out.println(root.data());
   func(root.left());
   func(root.right());
}
```

a) Preorder traversal
b) Inorder traversal
c) Postorder traversal
d) Level order traversal

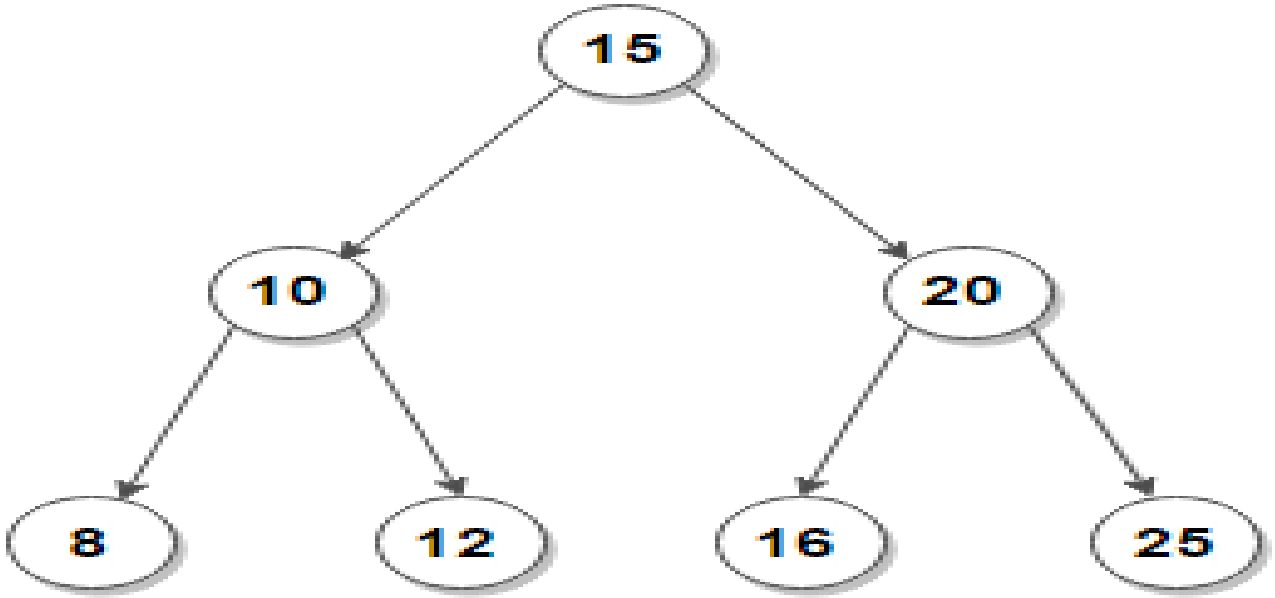
# 5. What does the following piece of code do?

```
public void func(Tree root)
{
   func(root.left());
   System.out.println(root.data());
   func(root.right());
}
```

a) Preorder traversal
b) Inorder traversal
c) Postorder traversal
d) Level order traversal

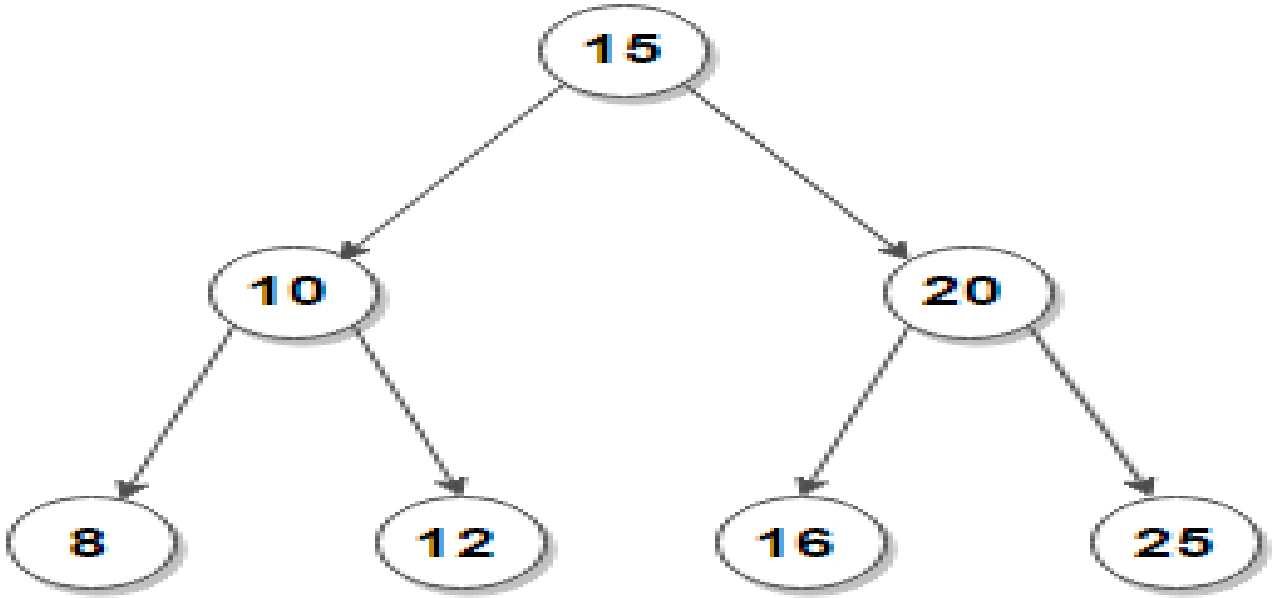
**6. Construct a binary search tree with the below information.**
**The preorder traversal of a binary search tree**
**10, 4, 3, 5, 11, 12.**

# 7. Pre-order of the below binary search tree ?



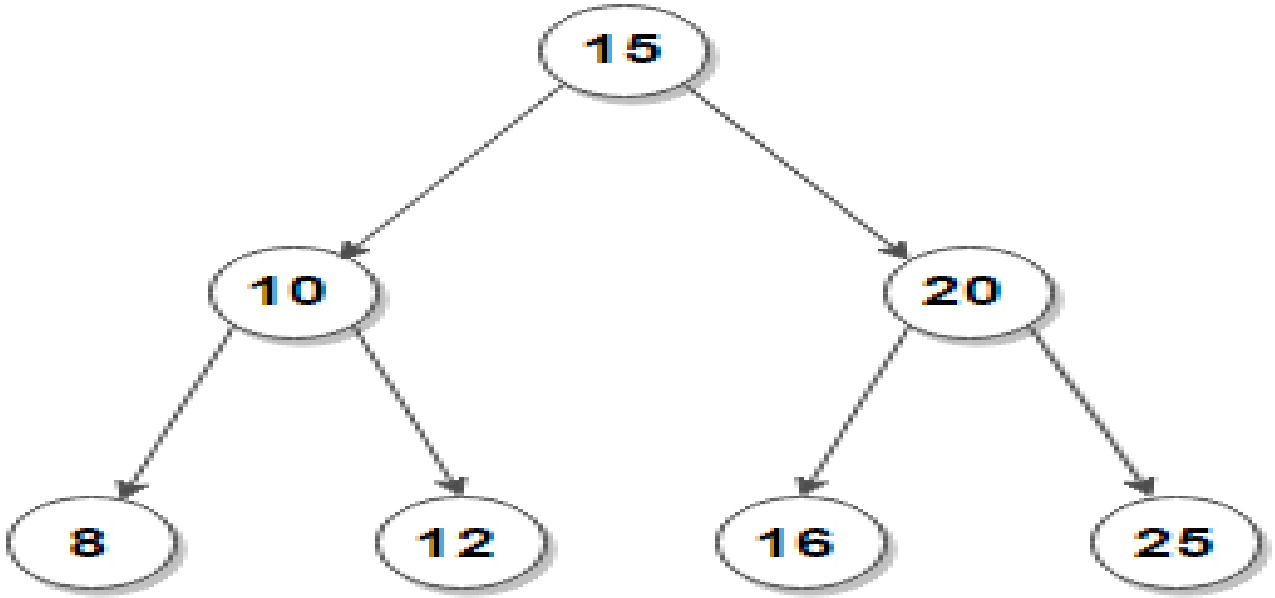

Binary Search Tree

# 8. In-order of the below binary search tree ?




Binary Search Tree

# 9. Post-order of the below binary search tree ?




Binary Search Tree

**10. Which of the following traversal outputs the data in sorted order in a BST**(Binary Search Tree)?

a) Postorder

b) Level order

c) Inorder

d) Preorder

# THANK YOU

```
class Node:
    def __init__(self, val):
        self.val = val
        self.leftChild = None
        self.rightChild = None

    def insert(self, val):
        if val < self.val:
            if self.leftChild:
                self.leftChild.insert(val)
            else:
                self.leftChild = Node(val)
                return
        else:
            if self.rightChild:
                self.rightChild.insert(val)
            else:
                self.rightChild = Node(val)
                return

    def search(self, val):
        if val < self.val:
            if self.leftChild:
                return self.leftChild.search(val)
            else:
                return False
        elif val > self.val:
            if self.rightChild:
                return self.rightChild.search(val)
            else:
                return False
        else:
            return True
        return False

    def delete(self, val):
        # if current node's val is less than that of root node,
        # then only search in left subtree otherwise right subtree
        if val < self.val:
            if(self.leftChild):
                self.leftChild = self.leftChild.delete(val)
            else:
                print(str(val) + " not found in the tree")
                return self
        elif val > self.val:
            if(self.rightChild):
                self.rightChild = self.rightChild.delete(val)
            else:
                print(str(val) + " not found in the tree")
                return self
        else:
            # deleting node with no children
            if self.leftChild is None and self.rightChild is None:
                self = None
                return None
            # deleting node with right child
            elif self.leftChild is None:
                tmp = self.rightChild
                self = None
                return tmp
            # deleting node with left child
            elif self.rightChild is None:
                tmp = self.leftChild
```

```
                self = None
                return tmp
            # deleting node with two children
            else:
                # first get the inorder successor
                current = self.rightChild
                # loop down to find the leftmost leaf
                while(current.leftChild is not None):
                    current = current.leftChild
                self.val = current.val
                self.rightChild = self.rightChild.delete(current.val)

        return self
```

```
class BinarySearchTree:
    def __init__(self, val):
        self.root = Node(val)

    def insert(self, val):
        if self.root:
            return self.root.insert(val)
        else:
            self.root = Node(val)
            return True

    def search(self, val):
        if self.root:
            return self.root.search(val)
        else:
            return False

    def delete(self, val):
        if self.root is not None:
            self.root = self.root.delete(val)
```

## Q1. Find minimum value in Binary Search Tree

Problem Statement#

Implement the findMin(root) function which will find the minimum value in a given Binary Search Tree. Remember, a Binary Search Tree is a Binary Tree which satisfies the following property. An illustration is also provided to jog your memory.

NodeValues(LeftSubtree)<=CurrentNodeValue<NodeValues(RightSubTree)

Output#

Returns minimum integer value from a given binary search tree

Sample Input#

The root of an object of the BST class which contains data such as.

bst = {

```
6 -> 4,9

4 -> 2,5

9 -> 8,12

12 -> 10,14
```

} where parent -> leftChild,rightChild

Sample Output#

2

```
def findMin(root):
  #write your code here
```

```
#Iterative findMin

def findMin(root):
    if root is None:  # check for None
        return None
    while root.leftChild:  # Traverse until the last child
        root = root.leftChild
    return root.val  # return the last child


BST = BinarySearchTree(6)
BST.insert(20)
BST.insert(-1)

print(findMin(BST.root))

```

```
-1
```

```
#Recursive findMin()

def findMin(root):
    if root is None:  # check if root exists
        return None
    elif root.leftChild is None:  # check if left child exists
        return root.val  # return if not left child
    else:
        return findMin(root.leftChild)  # recurse onto the left child


BST = BinarySearchTree(6)
BST.insert(20)
BST.insert(-1)

print(findMin(BST.root))

```

```
-1
```

## Q2. Find kth maximum value in Binary Search Tree

Problem Statement #

Implement a function findKthMax(root,k) which will take a BST and any number “k” as an input and return kth maximum number from that tree.

Output #

Returns kth maximum value from the given tree

Sample Input #

bst = {

```
6 -> 4,9

4 -> 2,5

9 -> 8,12

12 -> 10,14
```

}

where parent -> leftChild,rightChild

k = 3

Sample Output #

10

```
def findKthMax(root, k):
  #write your code here
```

```
#1: Sorting the tree in order #

def findKthMax(root, k):
    tree = []
    inOrderTraverse(root, tree)  # Get sorted tree list
    if ((len(tree)-k) >= 0) and (k > 0):  # check if k is valid
        return tree[-k]  # return the kth max value
    return None  # return none if no value found


def inOrderTraverse(node, tree):
    # Helper recursive function to traverse the tree inorder
    if node is not None:  # check if node exists
        inOrderTraverse(node.leftChild, tree)  # traverse left sub-tree
        if len(tree) is 0:
            # Append if empty tree
            tree.append(node.val)
        elif tree[-1] is not node.val:
            # Ensure not a duplicate
            tree.append(node.val)  # add current node value
        inOrderTraverse(node.rightChild, tree)  # traverse right sub-tree


BST = BinarySearchTree(6)
BST.insert(1)
BST.insert(133)
BST.insert(12)
print(findKthMax(BST.root, 3))
```

```
6
```

```
#2: Recursive Approach

def findKthMax(root, k):
    if k < 1:
        return None
    node = findKthMaxRecursive(root, k)  # get the node at kth position
    if(node is not None):  # check if node received
        return node.val  # return kth node value
    return None  # return None if no node found


counter = 0  # global count variable
current_max = None

def findKthMaxRecursive(root, k):
    global counter  # use global counter to track k
    global current_max # track current max
    if(root is None):  # check if root exists
        return None

```

```
    # recurse to right for max node
    node = findKthMaxRecursive(root.rightChild, k)
    if(counter is not k) and (root.val is not current_max):
        # Increment counter if kth element is not found
        counter += 1
        current_max = root.val
        node = root
    elif current_max is None:
        # Increment counter if kth element is not found
        # and there is no current_max set
        counter += 1
        current_max = root.val
        node = root
    # Base condition reached as kth largest is found
    if(counter == k):
        return node  # return kth node
    else:
        # Traverse left child if kth element is not reached
        # traverse left tree for kth node
        return findKthMaxRecursive(root.leftChild, k)


BST = BinarySearchTree(6)
BST.insert(4)
BST.insert(9)
BST.insert(5)
BST.insert(2)
BST.insert(8)

print(findKthMax(BST.root, 4))
```

```
5
```

## Q3. Find Ancestors of a given node in a BST

Problem Statement #

Implement the findAncestors(root, k) function which will find the ancestors of the node whose value is “k”. Here root is the root node of a binary search tree and k is an integer value of node whose ancestors you need to find. An illustration is also given. Your code is evaluated on the tree given in the example.

Output #

Returns all the ancestors of k in the binary tree in a Python list.

Sample Input #

bst = {

```
 6 -> 4,9

 4 -> 2,5

 9 -> 8,12

 12 -> 10,14
```

}

where parent -> leftChild,rightChild

k = 10

Sample Output #

[12,9,6]

```
def findAncestors(root, k):
  #write your code here
```

```
#1: Using a recursive helper function

def findAncestors(root, k):
    result = []
    recfindAncestors(root, k, result)  # recursively find ancestors
    return str(result)  # return a string of ancestors


def recfindAncestors(root, k, result):
    if root is None:  # check if root exists
        return False
    elif root.val is k:  # check if val is k
        return True
    recur_left = recfindAncestors(root.leftChild, k, result)
    recur_right = recfindAncestors(root.rightChild, k, result)
    if recur_left or recur_right:
        # if recursive find in either left or right sub tree
        # append root value and return true
        result.append(root.val)
        return True
    return False  # return false if all failed


BST = BinarySearchTree(6)
BST.insert(1)
BST.insert(133)
BST.insert(12)
print(findAncestors(BST.root, 12))
```

```
[133, 6]
```

```
#2: Iteration

def findAncestors(root, k):
    if not root:  # check if root exists
        return None
    ancestors = []  # empty list of ancestors
    current = root  # iterator current set to root

    while current is not None:  # iterate until we reach None
        if k > current.val:  # go right
            ancestors.append(current.val)
            current = current.rightChild
        elif k < current.val:  # go left
            ancestors.append(current.val)
            current = current.leftChild
        else:  # when k == current.val
            return ancestors[::-1]
    return []


BST = BinarySearchTree(6)
BST.insert(1)
BST.insert(133)
BST.insert(12)
print(findAncestors(BST.root, 12))
```

```
[133, 6]
```

## Q4: Find the Height of a BST

Problem Statement

Implement a function findHeight(root) which returns the height of a given binary search tree. An illustration is also provided for your understanding.

Height of a Node — the number of edges between a node and its deepest descendent

Height of a Tree — Height of its root node

Also, keep in mind that the height of an empty tree and leaf nodes is zero.

Output #

Returns the maximum depth or height of a binary tree

Sample Input #

bst = {

```
6 -> 4,9

4 -> 2,5

9 -> 8,12

12 -> 10,14
```

}

where parent -> leftChild,rightChild

Sample Output #

3

```
def findHeight(root):
  #write your code here
```

```
def findHeight(root):
    if root is None:  # check if root exists
        return -1  # no root means -1 height
    else:
        max_sub_tree_height = max(
            findHeight(root.leftChild),
            findHeight(root.rightChild)
        )  # find the max height of the two sub-tree
        # add 1 to max height and return
        return 1 + max_sub_tree_height


BST = BinarySearchTree(6)
BST.insert(4)
BST.insert(9)
BST.insert(5)
BST.insert(2)
BST.insert(8)
BST.insert(12)
BST.insert(10)
BST.insert(14)


print(findHeight(BST.root))
```

```
3
```

## Q5: Find Nodes at "k" distance from the Root

Problem Statement#

Implement a function findKNodes(root,k) which finds and returns nodes at k distance from the root in the given binary tree. An illustration is also provided for your understanding.

Output#

Returns all nodes in a list format which are at k distance from the root node

Sample Input#

bst = {

```
6 -> 4,9

4 -> 2,5

9 -> 8,12

12 -> 10,14
```

}

where parent -> leftChild,rightChild

k = 2

Sample Output#

[2,5,8,12]

```
1
2 def findKNodes(root, k):
3     res = []
4     findK(root, k, res)  # recurse the tree for node at k distance
5     return str(res)
6
7
8 def findK(root, k, res):
9     if root is None:  # return if root does not exist
10        return
11    if k == 0:
12        res.append(root.val)  # append as root is kth node
13    else:
14        # check recursively in both sub-tree for kth node
15        findK(root.leftChild, k - 1, res)
16        findK(root.rightChild, k - 1, res)
17
18
19 BST = BinarySearchTree(6)
20 BST.insert(4)
21 BST.insert(9)
22 BST.insert(5)
23 BST.insert(2)
24 BST.insert(8)
25 BST.insert(12)
26 print(findKNodes(BST.root, 2))
```

```
[2, 5, 8, 12]
```

```
1
```

```
pip install binarytree
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting binarytree
  Downloading binarytree-6.5.1-py3-none-any.whl (18 kB)
Collecting setuptools>=60.8.2
  Downloading setuptools-65.3.0-py3-none-any.whl (1.2 MB)
     |████████████████████████████████| 1.2 MB 33.6 MB/s
Requirement already satisfied: graphviz in /usr/local/lib/python3.7/dist-packages (from binarytree) (0.10.1)
Collecting setuptools-scm[toml]>=5.0.1
  Downloading setuptools_scm-7.0.5-py3-none-any.whl (42 kB)
     |████████████████████████████████| 42 kB 1.5 MB/s
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from setuptools-scm[toml]>=5.0.1->bi
Requirement already satisfied: tomli>=1.0.0 in /usr/local/lib/python3.7/dist-packages (from setuptools-scm[toml]>=5.0.1->binarytr
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/dist-packages (from setuptools-scm[toml]>=5.0.1->binar
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/dist-packages (from setuptools-scm[toml]>=5.0.1->bi
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.7/dist-packages (from packaging>=20.0->setuptoo
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata->setuptools-scm[toml
Installing collected packages: setuptools, setuptools-scm, binarytree
  Attempting uninstall: setuptools
    Found existing installation: setuptools 57.4.0
    Uninstalling setuptools-57.4.0:
      Successfully uninstalled setuptools-57.4.0
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the
ipython 7.9.0 requires jedi>=0.10, which is not installed.
Successfully installed binarytree-6.5.1 setuptools-65.3.0 setuptools-scm-7.0.5
```

```
from binarytree import Node
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.right = Node(4)

# Getting binary tree
print('Binary tree :', root)

# Getting list of nodes
print('List of nodes :', list(root))
```

```
Binary tree :
  __1
 /   \
2     3
 \
  4

List of nodes : [Node(1), Node(2), Node(3), Node(4)]
```

```
#BST traversals (In order, Pre-order, Post-order )

class Node:

    def __init__(self, data):

        self.left = None
        self.right = None
        self.data = data
# Insert Node
    def insert(self, data):

        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
```

```
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data


# Inorder traversal
# Left -> Root -> Right
    def inorderTraversal(self, root):
        res = []
        if root:
            res = self.inorderTraversal(root.left)
            res.append(root.data)
            res = res + self.inorderTraversal(root.right)
        return res


# Preorder traversal
# Root -> Left ->Right
    def PreorderTraversal(self, root):
        res = []
        if root:
            res.append(root.data)
            res = res + self.PreorderTraversal(root.left)
            res = res + self.PreorderTraversal(root.right)
        return res


# Postorder traversal
# Left ->Right -> Root
    def PostorderTraversal(self, root):
        res = []
        if root:
            res = self.PostorderTraversal(root.left)
            res = res + self.PostorderTraversal(root.right)
            res.append(root.data)
        return res
```

```
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
```

```
print(root.inorderTraversal(root))
print(root.PreorderTraversal(root))
print(root.PostorderTraversal(root))
```

```
[10, 14, 19, 27, 31, 35, 42]
[27, 14, 10, 19, 35, 31, 42]
[10, 19, 14, 31, 42, 35, 27]
```

+ Code + Text

```
1
```

## The Heap Order Property#

The nodes must be ordered according to the Heap Order Property. The heap order property is different for the two heap structures that we are going to study in this chapter:

- Min Heap
- Max Heap

Min Heaps are built based upon the *Min Heap property* and Max Heaps are built based upon *Max Heap Property*. Let's see how they are different.

### Max Heap Property:#

All the parent node keys must be greater than or equal to their child node keys in max-heaps. So the root node will always contain the largest element in the Heap. If Node A has a child node B, then:

$$key(A) >= key(B)$$

$$key(A)>=key(B)$$

### Min Heap Property:#

In Min-Heaps, all the parent node keys are less than or equal to their child node keys. So the root node, in this case, will always contain the smallest element present in the Heap. If Node A has a child node B, then:

$$key(A) <= key(B)$$

$$key(A) <= key(B)$$

## Building a Max-Heap#

As mentioned in the previous lesson, max heaps follow the max heap property which means that the key at the parent node is always greater than the keys at the child nodes. Heaps can be implemented using lists or using node and tree classes. Although they are generally implemented using lists or arrays as that is the more space-efficient approach! To build a heap, start with an empty one and successively `insert()` all the elements.

## Insertion in a Max-Heap#

Here is a high-level description of the algorithm to insert elements into a heap and maintain the heap property.

- Create a new child node at the end of the heap
- Place the new key at that node
- Then, restore the heap property by swapping parent and child values if the parent key is smaller than the child key. We call this 'percolating up'.
- Continue to percolate up until the heap property is restored.

## Remove Maximum in a Max Heap#

Here is the algorithm that you will follow to make sure the heap property still holds after deleting the root element

- Delete the root node
- Move the key of last child node at the last level to the root
- Now compare the key with its children and if the key is smaller than the key at any of the child nodes, swap values. We call this 'max heapifying.'
- Continue to max heapify until the heap property is restored.

## Max-heap Implementation #

Let's start with some function declarations for the heap class. The `__percolateUp()` function is meant to restore the heap property going up from a node to the root. The `__maxHeapify()` function restores the heap property starting from a given node down to the leaves. The two underscores before the `__percolateUp()` and `__maxHeapify()` functions imply that these functions should be treated as private functions although there is no actual way to *enforce* class function privacy in Python. You can still call these functions by prepending `_className` like so, `heap._maxHeap__percolateUp(index)`.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.__percolateUp(len(self.heap)-1)

    def getMax(self):
        if self.heap:
            return self.heap[0]
        return None

    def removeMax(self):
        if len(self.heap) > 1:
            max = self.heap[0]
            self.heap[0] = self.heap[-1]
            del self.heap[-1]
            self.__maxHeapify(0)
            return max
        elif len(self.heap) == 1:
            max = self.heap[0]
            del self.heap[0]
            return max
        else:
            return None

    def __percolateUp(self, index):
        parent = (index-1)//2
```

```
        if index <= 0:
            return
        elif self.heap[parent] < self.heap[index]:
            tmp = self.heap[parent]
            self.heap[parent] = self.heap[index]
            self.heap[index] = tmp
            self.__percolateUp(parent)

    def __maxHeapify(self, index):
        left = (index * 2) + 1
        right = (index * 2) + 2
        largest = index
        if len(self.heap) > left and self.heap[largest] < self.heap[left]:
            largest = left
        if len(self.heap) > right and self.heap[largest] < self.heap[right]:
            largest = right
        if largest != index:
            tmp = self.heap[largest]
            self.heap[largest] = self.heap[index]
            self.heap[index] = tmp
            self.__maxHeapify(largest)

    def buildHeap(self, arr):
        self.heap = arr
        for i in range(len(arr)-1, -1, -1):
            self.__maxHeapify(i)


heap = MaxHeap()
heap.insert(12)
heap.insert(10)
heap.insert(-10)
heap.insert(100)

print(heap.getMax())
```

## Building a Min-Heap

Min Heap property which means that the key at the parent node is always smaller than the keys at the child nodes. Heaps can be implemented using lists. Initially, elements are placed in nodes in the same order as they appear in the list. Then a function is called over the whole heap in a bottom-up manner that “Min Heapifies” or “percolates up” on this heap so that the heap property is restored. The “Min Heapify” function is bottom-up because it starts comparing and swapping parent-child key values from the last parent

## Insertion in Min Heap #

Here is a high-level description of the algorithm to insert elements into a heap and maintain the heap property:

- Create a new child node at the end of the heap
- Place the new key at that node (append it to the list or array)
- Percolate up until you reach the root node and the heap property is satisfied

## Remove Minimum in Min Heap #

Here is the algorithm that you will follow to make sure the heap property still holds after deleting the root element

- Delete the root node
- Move the key of the last child node to root
- Perculate down: if the key is larger than the key at any of the child nodes, swap values
- Repeat until you reach the last node

# Min Heap (Implementation)

```
class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.__percolateUp(len(self.heap)-1)

    def getMin(self):
        if self.heap:
            return self.heap[0]
        return None

    def removeMin(self):
        if len(self.heap) > 1:
            min = self.heap[0]
            self.heap[0] = self.heap[-1]
            del self.heap[-1]
            self.__minHeapify(0)
            return min
        elif len(self.heap) == 1:
            min = self.heap[0]
            del self.heap[0]
            return min
        else:
            return None

    def __percolateUp(self, index):
        parent = (index-1)//2
        if index <= 0:
            return
        elif self.heap[parent] > self.heap[index]:
            tmp = self.heap[parent]
            self.heap[parent] = self.heap[index]
            self.heap[index] = tmp
            self.__percolateUp(parent)

    def __minHeapify(self, index):
        left = (index * 2) + 1
        right = (index * 2) + 2
        smallest = index
        if len(self.heap) > left and self.heap[smallest] > self.heap[left]:
            smallest = left
        if len(self.heap) > right and self.heap[smallest] > self.heap[right]:
            smallest = right
```

```
        if smallest != index:
            tmp = self.heap[smallest]
            self.heap[smallest] = self.heap[index]
            self.heap[index] = tmp
            self.__minHeapify(smallest)

    def buildHeap(self, arr):
        self.heap = arr
        for i in range(len(arr)-1, -1, -1):
            self.__minHeapify(i)


heap = MinHeap()
heap.insert(12)
heap.insert(10)
heap.insert(-10)
heap.insert(100)

print(heap.getMin())
print(heap.removeMin())
print(heap.getMin())
heap.insert(-100)
print(heap.getMin())
```

# Lecture 19: Heaps Implementation

## MaxHeap Implementation

```
class MaxHeap:
    def __init__(self):
        self.items = []

    def insert(self, item):
        self.items.append(item)
        self.__percolate_up(len(self.items) - 1)

    def __swap(self, a, b):
        self.items[a], self.items[b] = self.items[b], self.items[a]

    def __percolate_up(self, index):
        if index == 0: return
        parent = (index - 1) // 2
        if self.items[index] > self.items[parent]:
            self.__swap(index, parent)
            self.__percolate_up(parent)

    def __max_heapify(self, index):
        left = (index * 2) + 1
        right = (index * 2) + 2
        largest = index
        if len(self.items) > left and self.items[left] > self.items[largest]:
            largest = left
        if len(self.items) > right and self.items[right] > self.items[largest]:
            largest = right

        if largest != index:
            self.__swap(index, largest)
            self.__max_heapify(largest)

    def get_max(self):
        if self.items:
            return self.items[0]
        return None

    def pop(self):
        if len(self.items) > 1:
            self.__swap(0, len(self.items) - 1)
            max = self.items.pop()
            self.__max_heapify(0)
        elif len(self.items) == 1:
            max = self.items.pop()
        else:
            max = None
        return max

    def isEmpty(self):
        return len(self.items) == 0
```

## Story of GodFather

```
corleoneFamily = MaxHeap()
corleoneFamily.insert([100, "Vito"])
corleoneFamily.insert([85, "Tom"])
corleoneFamily.insert([70, "Fredo"])
corleoneFamily.insert([80, "Sonny"])
corleoneFamily.insert([50, "Connie"])
corleoneFamily.insert([90, "Michael"])
print(corleoneFamily.items)
print("Popping the Boss")
corleoneFamily.pop()
print(corleoneFamily.items)
```

## K Smallest

Implement a function `findKSmallest(lst,k)` that takes an unsorted integer list as input and returns the “k” smallest elements in the list using a Heap.

```
# Every year many students wants to get into Harvard MBA program.
# But only a few get in.
# Harvard uses z-score to measure the fitness of a candidate.
# The z-score is calculated based GMAT score, GPA, work experience, etc.

# Given the list of z-scores of a set of candidates,
# write a program to output the k candidates with the highest z-scores.

import numpy as np
from maxHeap import MaxHeap
from minHeap import MinHeap

z_score = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.85, 0.97, 0.56, 0.43, 0.67, 0.78, 0.89, 0.98, 0.87, 0.76, 0.65, 0.54, 0.43, 0.32, 0.21, 0.1,
z_score = z_score * 20
k = 200

countBrute = 0

def kGreatestBruteForce(z_score, k):
    answer = []
    global countBrute
    for i in range(k):
        maxCandidate = 0
        for j in range(len(z_score)):
            countBrute += 1
            if z_score[j] > z_score[maxCandidate]:
                maxCandidate = j
        answer.append(z_score[maxCandidate])
        # z_score.pop(maxCandidate)
        # [0, 1, 2, 3, 4] -> pop(2) -> [0, 1, 3, 4]
        z_score[maxCandidate], z_score[-1] = z_score[-1], z_score[maxCandidate] # O(1)
        # [0, 1, 2, 3, 4] -> swap(2,4) -> [0, 1, 4, 3, 2] ->
        # pop() -> [0, 1, 4, 3]
        z_score.pop() # O(1)
    return answer

def kGreatestSort(z_score, k):
    z_score.sort()
    return z_score[-k:]

def kGreatestMaxHeap(z_score, k):
    heap = MaxHeap()
    for i in range(len(z_score)): # O(n log n)
        heap.insert(z_score[i])
    answer = []
    for i in range(k): # O(k log n)
        answer.append(heap.pop())
    return answer

n = len(z_score)
print(kGreatestBruteForce(z_score.copy(), k), n, k, countBrute)
print(kGreatestSort(z_score, k), n, k, n * np.log(n)//1)

# For Harvard, k << n, can we reduce the time to O(n * log(k)) ?
# If we can manage to keep the heap size to k,
# then we can reduce the time to O(n * log(k))

# Can you think of an approach that uses a heap and takes O(n * log(n))?

# Operations on a Heap of size n:
# 1. Insert - O(log(n))
# 2. Pop - O(log(n))
# 3. Get Max - O(1)
# 4. Is Empty - O(1)

# Insert n elements into a heap - O(n * log(n))
# Pop k elements from a heap - O(k * log(n))

# Insert first k elements in a Min Heap - O(k * log(k))
# For the remaining elements, if the element is greater than the
# root of the heap, then pop the root and insert
# the element - O((n-k) * log(k)), else ignore it.
```

```
# Total time complexity - O(k * log(k) + (n-k) * log(k)) = O(n * log(k))

def kGreatestMinHeap(z_score, k):
    heap = MinHeap()
    for i in range(k): # O(k * log(k))
        heap.insert(z_score[i])
    for i in range(k, len(z_score)): # O((n-k) * log(k))
        if z_score[i] > heap.get_min():
            heap.pop()
            heap.insert(z_score[i])
    answer = []
    for i in range(k): # O(k * log(k))
        answer.append(heap.pop())
    return answer

print(kGreatestMinHeap(z_score, k), n, k, n * np.log(k)//1)
```

## Min Heap Implementation

```
class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self.heapify_up(len(self.heap) - 1)

    def get_min(self):
        return self.heap[0]

    def pop(self):
        if len(self.heap) == 0:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()
        min_val = self.heap[0]
        self.heap[0] = self.heap.pop()
        self.heapify_down(0)
        return min_val

    def heapify_up(self, index):
        parent = (index - 1) // 2
        if parent < 0:
            return
        if self.heap[parent] > self.heap[index]:
            self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]
            self.heapify_up(parent)

    def heapify_down(self, index):
        left = 2 * index + 1
        right = 2 * index + 2
        smallest = index
        if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
            smallest = left
        if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
            smallest = right
        if smallest != index:
            self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
            self.heapify_down(smallest)

    def __str__(self):
        return str(self.heap)
```

# Lecture 20. Hash Tables

Store a set of values and retrieve them.

API:

- Push(x) - Insert the value x in the data structure
- Get(x) - Will get the value of x from the data structure if x is present
- Has(x) - Check if x is present in the data structure

## Existing Solutions

1. Arrays or Lists
    a. Static vs Dynamic - Dynamic
    b. Single Dimensional vs Multi Dimensional - Single Dimensional
    c. Sorted vs Unordered - Both
2. LinkedList
    a. Singly or Doubly
    b. Sorted vs Unordered - Here sorted is not better than unordered.
3. Binary Search Trees

## Time Complexity

Assume x is a positive integer with in the range [1, 100000].

| Operations | Array Sorted | Array Unordered | LinkedList | Balanced BST | A Boolean Array of Size 100001 |
|---|---|---|---|---|---|
| Push(x) | O(n) | O(1) | O(1) | O(log n) | O(1) |
| Has(x) | O(log n) | O(n) | O(n) | O(log n) | O(1) |

Visited Array like a Graph?

```
class Members:
    def __init__(self, maxValue):
        self.members =  [False] * (maxValue + 1)

    def add(self, value):
        self.members[value] = True

    def has(self, value):
        return self.members[value]
```

```
    def remove(self, value):
        self.members[value] = False
```

## What are the problems with this DataStructure?

1. The values should be non-negative integers. You can’t use Strings, or Tuples, or Floats, or Pointers.
2. Values should have an upper bound.
3. What if there are very few values in a very large range? Suppose you want to store the following array : [1000000000, 98957293587]
    a. You will be wasting a lot of space.

## How to solve these problems?

All these three problems are coming because we have coupled the index with the value.

How can we decouple?

Why do you want to decouple?

We want to fix the number of indices separately from the input.

Here is another implementation of the above function.

```
# I am sure there will be 10000 values distributed uniformly.

# Let's have values = [0, 1000007, 2000014, 3000021, 4000028,
# 5000035, 6000042, 7000049, 8000056, 9000063]

class Members:
    def __init__(self):
        self.members =  [[] for _ in range(1000007)]
        self.length = 1000007

    # f(value) = value % self.length is the hash function
    # What is the benefit of this hash function?
    # It decouples the values from the indexes of the array.
    def add(self, value):
        self.members[value % self.length].push(value)

    def has(self, value):
        return value in self.members[value % self.length]
```

```
    def remove(self, value):
        self.members[value % self.length].remove(value)
```

## Still we have the following problems

1. We can’t use String. Although we can now use integers without a limit.
2. In the worse case all the numbers can end up in a single slot, i.e., collisions.

## How can we solve these problems?

Hash Function

There is a great future in Hash Functions. All the problems can be solved using a better Hash Function.

Maanasa: “I can’t use Strings.”

Anshul: “Use a better Hash Function.”

Radhika: “I am having a lot of collisions.”

Anshul: “Use a better Hash Function.”

Upasana: “My Hash Function is talking too much time to compute.”

Anshul: “Use a better Hash Function.”

Joe: “I want the elements in sorted order.”

Anshul: “Hash Tables are not Boro Plus.”

Aashna: “Can I store duplicates?”

Anshul: “Yes”

Maanasa: “Can I store Key, Value pairs?”

Anshul: “Yes! You can.”

```
# I am sure there will be 10000 values distributed uniformly.

# Let's have values = [0, 1000007, 2000014, 3000021, 4000028,
# 5000035, 6000042, 7000049, 8000056, 9000063]

class Members:
    def __init__(self):
        self.members =  [[] for _ in range(1000007)]
        self.length = 1000007
        self.prime = 282589933
        self.hash = lambda value: (value * prime) % self.length
```

```
    # f(value) = value % self.length is the hash function
    # What is the benefit of this hash function?
    # It decouples the values from the indexes of the array.
    def add(self, key, value):
        self.members[self.hash(key)].push([key, value])

    def has(self, key):
        return key in self.members[self.hash(key)]

    def get(self, key):
        for pair in self.members[self.hash(key)]:
            if pair[0] == key:
                return pair[1]
        return None

    def remove(self, key):
        for pair in self.members:
            if pair[0] == key:
                self.members.remove(pair)
```

## Summary

- Hash Tables can store values and retrieve them both in O(1) time.

# Problems

## Find Two Pairs

In this problem, you have to implement the `find_pair()` function which will find two pairs, `[a, b]` and `[c, d]`, in a list such that :

*a* + *b* = *c* + *d*

You only have to find the first two pairs in the list which satisfies the above condition.

All the elements of the list are distinct integers.

```
nums = [23, 24, 1, 8, 6, 7, 9, 5, 3, 2, 73, 22]

def getMatchingPairs(nums):
    sumDict = {}
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            current = nums[i] + nums[j]
            if current not in sumDict:
```

```
                sumDict[current] = [nums[i], nums[j]]
            else:
                return [sumDict[current], [nums[i], nums[j]]]
    return -1

print(getMatchingPairs(nums))
```

# Lecture 21: Sorting Algorithms

## Introduction

Q. What is the difference between an algorithm and a data structure?

Answer:

- Algorithm is the process you follow to solve a problem. Data structure is a way to store data that supports efficient running of some algorithms.
    - Example: Heap Sort and Heap.
- Algorithm is the collection of actions you take to reach a goal. Data structure is the means through which you complete your journey.
    - Example: You want to generate a path from a source to a destination. You can use DFS algorithm to do that. But you would need a Graph data structure to store the information about the nodes including source and destination and their connections.

Q. What is the similarity between an algorithm and a data structure?

Answer:

- We make choice of algorithm as well as data structure.
- Both can impact Time and Space complexity.
- We study algorithms and data structures both because they are solution patterns for common computational problems.

Q. How will I know if my solution is suffering because of a bad data structure vs a bad algorithm?

Answer:

- I don’t know.

# Sorting Algorithms

| Algorithm | Coverage | Time Complexity |
|---|---|---|
| Insertion Sort, Bubble Sort and Selection Sort | Little | O(n^2) |
| Merge Sort and Quick Sort | Good | O(n log n) |
| Bucket Sort | Okay | O(n) |

## Sleep Sort

```
import time
from threading import Thread

def sleep_sort(nums):
    sorted_nums = []

    def sleep_and_append(num):
        time.sleep(num)
        sorted_nums.append(num)

    threads = []
    for num in nums:
        t = Thread(target=sleep_and_append, args=(num,))
        threads.append(t)
        t.start()

    for t in threads:
        t.join()
```

```
    return sorted_nums

nums = [5, 2, 8, 1, 9]
sorted_nums = sleep_sort(nums)
print(sorted_nums)
```

```
function sleepSort(nums) {
  let sortedNums = [];

  function sleepAndAppend(num) {
    setTimeout(() => {
      sortedNums.push(num);
    }, num);
  }

  nums.forEach((num) => {
    sleepAndAppend(num);
  });

  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(sortedNums);
    }, Math.max(...nums) + 100);
  });
}

const nums = [5, 2, 8, 1, 9];
sleepSort(nums)
  .then((sortedNums) => {
    console.log(sortedNums);
  })
  .catch((error) => {
    console.error(error);
  });
```

## Insertion Sort, Bubble Sort and Selection Sort

These algorithms are Brute force algorithms for solving the problem of sorting a random array of numbers.

Q. What is sorting?

Answer:

- Sorting is the process of putting a list into a sorted order.
- Sorted order is the order where elements are either increasing or decreasing in magnitude from left to right.
- To sort the array [5,1,2,4,3,6] to convert it into [1,2,3,4,5,6].

### Insertion Sort

InsertionSort(nums):

1. Create an empty array, SortedArray, which you intend to keep in sorted order.
2. For each element of nums:
    a. Take the element and insert it in its sorted position in the SortedArray.

```
nums = [5,3,4,7,6,3,2,1,8,9,2,1]

def insertInSorted(nums, element):
    nums.append(element)
    i = len(nums) - 1
    while i > 0 and nums[i] < nums[i-1]:
        nums[i], nums[i-1] = nums[i-1], nums[i]
        i -= 1

def insertionSort(nums):
```

```
    sorted_nums = []
    for num in nums:
        insertInSorted(sorted_nums, num)
    return sorted_nums

print("Sorting {}".format(nums))
sorted_nums = insertionSort(nums)
print("Sorted: {}".format(sorted_nums))
```

## Selection Sort

SelectionSort(nums):

1. Create an empty array, SortedArray, which you intend to keep in sorted order.
2. While nums is not empty:
    a. Find the minimum element in nums.
    b. Remove it from nums.
    c. Append it to the SortedArray.

## Question on Selection Sort

```
# Find the total cost of performing selection sort on a list of numbers.
# The algorithm is as follows:

# SelectionSort(nums):
# 1. Create an empty array, SortedArray, which you intend to keep in sorted order.
# 2. While nums is not empty:
#     1. Find the minimum element in nums.
#     2. Remove it from nums. Cost = min(nums) * len(nums)
#     3. Append it to the SortedArray.

# Every operation is free but the cost of removing an element from nums is equal to the size of nums at that time times the element its

# For example, if nums = [5, 3, 4, 7, 6, 3, 2, 1, 8, 9, 2, 1], then the cost of removing 1 is 12 * 1 = 12.

# You have to add all the costs across all the operations.
# Time complexity: O(n^2)
def selectionSortCost(nums):
    cost = 0
    # O(n) * O(n)
    while len(nums) > 0:
        min_num = min(nums) # O(n)
        cost += min_num * len(nums) # O(n)
        nums.remove(min_num) # O(n)
    return cost

# Time complexity: O(n log n)
def selectionSortCost2(nums):
    nums.sort()
    cost = 0
    for i in range(len(nums)):
        cost += nums[i] * (len(nums) - i)
    return cost

nums = [5, 3, 4, 7, 6, 3, 2, 1, 8, 9, 2, 1]
nums2 = [1, 2, 3, 4, 5]
print("Cost of sorting [5, 3, 4, 7, 6, 3, 2, 1, 8, 9, 2, 1] is {}".format(selectionSortCost(nums.copy())))
print("Cost of sorting [1, 2, 3, 4, 5] is {}".format(selectionSortCost(nums2.copy())))
print("Cost of sorting [5, 3, 4, 7, 6, 3, 2, 1, 8, 9, 2, 1] is {}".format(selectionSortCost2(nums)))
print("Cost of sorting [1, 2, 3, 4, 5] is {}".format(selectionSortCost2(nums2)))

# What will be the cost for [3,2,4,1,5]? 3*5 + 2*4 + 4*3 + 1*2 + 5*1 = 35
```

## Bubble Sort

BubbleSort(nums):

1. pairFound = True

2. while pairFound:
    a. pairFound = False
    b. For i : 0 → n - 1
        i. if nums[i] > nums[i+1]:
            1. swap nums[i] and nums[i+1]
            2. pairFound = True