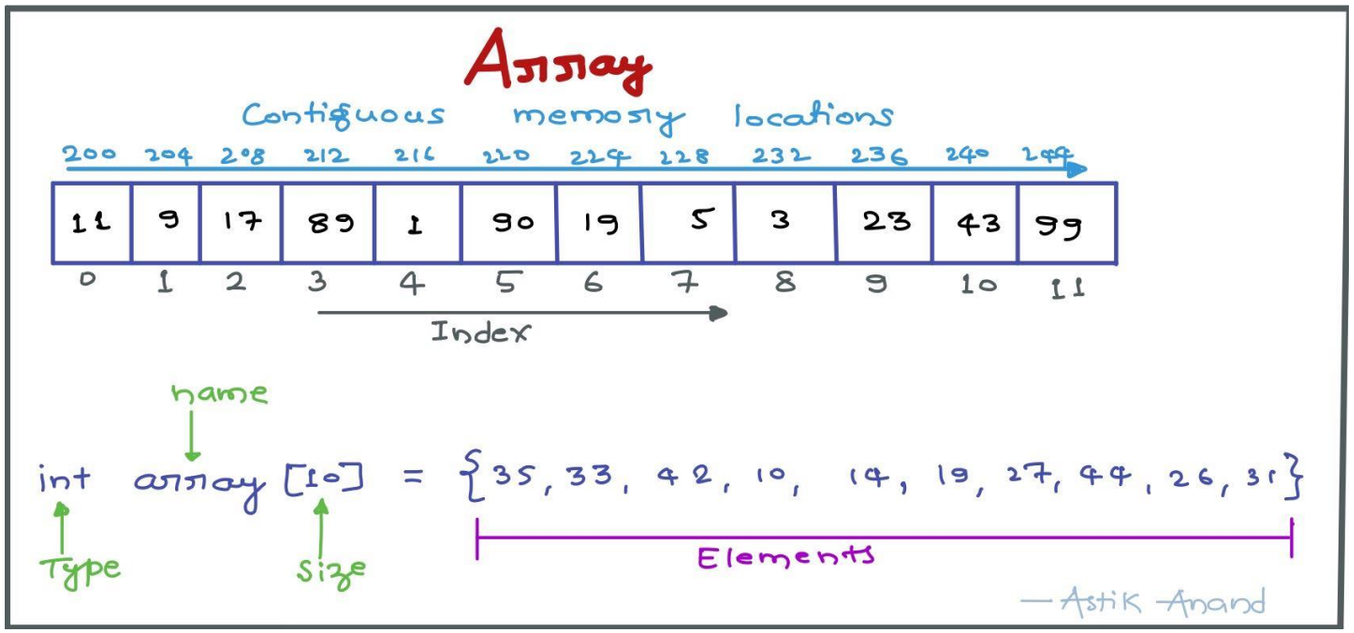
Data Structure By Astik Anand (Microsoft SDE2)

Contents

Array

What is an array ?

- An array is a sequential collection of elements of same data type and stores data elements in a continuous memory location.
- The elements of an array are accessed by using an index. The index of an array of size N can range from 0 to N−1.



Different Types of Array

- 1-D Array (Array)
- 2-D Array (Matrix)
- n-D Array

Applications of Array

- Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables.
- Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.
- Arrays are used to implement other data structures such as lists, heaps, hash tables, deques, queues, stacks, strings etc.

Standard Array Problems

1. Leaders in Array

Problem:

Print all the LEADERS in the array. An element is leader if it is greater than all the elements to its right side.

The rightmost element is always a leader.

Example:

Input: [13, 15, 6, 7, 8, 3]

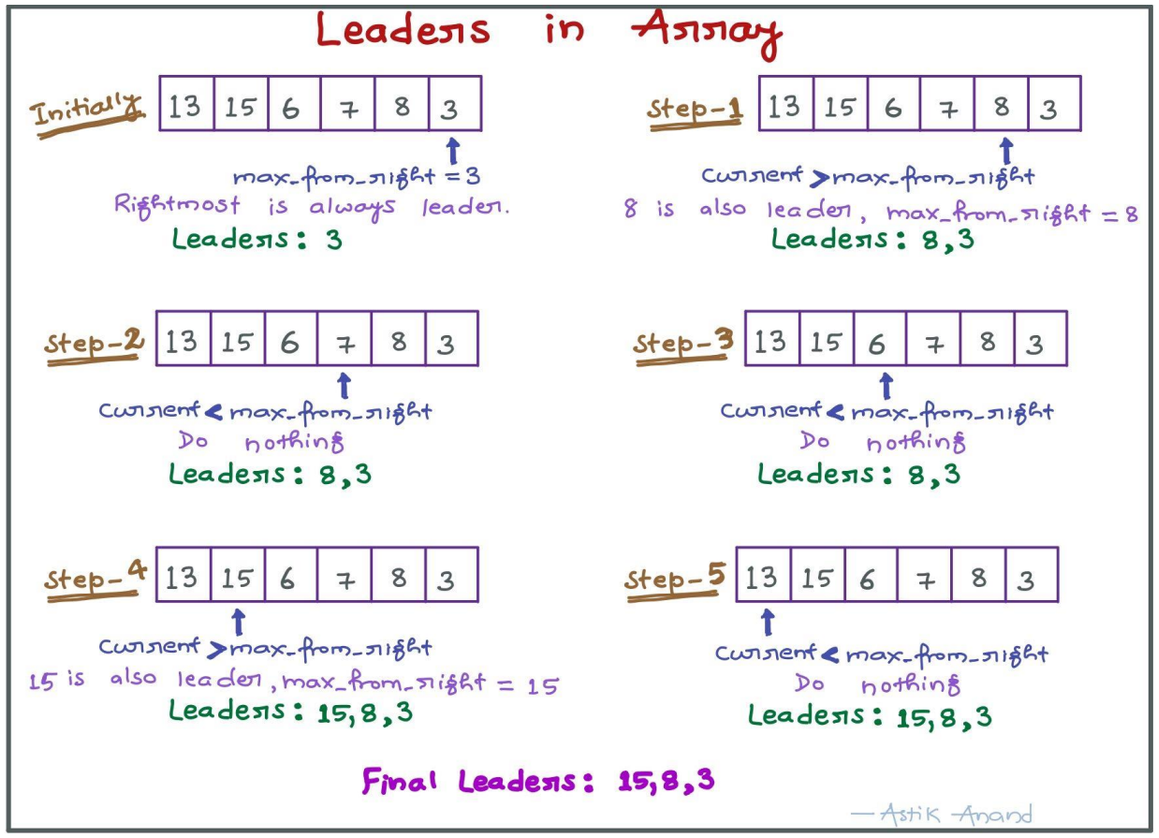
Output: 15, 8, 3

Approach-1: Brute Force

- Use two loops, the outer loop runs one by one picks all elements from left to right.
- The inner loop compares the picked element to all the elements to its right side.
- If the picked element is greater than all the elements to its right side, then the picked element is the leader.
- **Time Complexity: $O(n^2)$**

Approach-2: Scan from right

- Scan all the elements from right to left in an array and keep track of maximum till now.
- When maximum changes its value, print it.
- **Time Complexity: O(n)**



Implementation

```
def print_leaders(arr):
    n = len(arr)
    # Righmost element is always leader
    max_from_right = arr[-1]
    leaders = [arr[-1]]

    for i in range(n-2, -1, -1):
        if max_from_right < arr[i]:
            max_from_right = arr[i]
            leaders.append(arr[i])

    print(leaders[::-1])


print("Example-1: print_leaders([13, 15, 6, 7, 8, 3])")
print_leaders([13, 15, 6, 7, 8, 3])
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 17:51:34 ~/Personal/Notebooks/Data Structures/1. Array $ python3 1_leaders_in_array.py
Example-1: print_leaders([13, 15, 6, 7, 8, 3])
[15, 8, 3]
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)** as leaders array can be ignored and directly printed.

2. Maximum sum with no two adjacent elements***

Problem:

Given an array of positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7).

Example:

Input: arr[] = {5, 5, 10, 100, 10, 5} ***Output:*** 110

Input: arr[] = {1, 2, 3} ***Output:*** 4

Input: arr[] = {1, 20, 3} ***Output:*** 20

Algorithm:

- Start with two sums excluded and included.
- Loop for all the elements and:
  - Calculate **new_excluded** as the **max(included, excluded)** as current element is still not added to the included.
  - Now **change the included by adding current to excluded** as no two adjacents should be present.
  - Finally **update the exluded** with new_excluded.
- Return the **max(included, excluded)**.

Max Sum with no 2 adjacent elements

Initially

5	5	10	100	10	5

included = 0
excluded = 0

Step-1:

5	5	10	100	10	5

Current = 5
new_excluded = max(included, excluded)
= max(0,0) = 0
included = current + excluded
= 5 + 0 = 5
excluded = new_excluded = 0

Step-2:

5	5	10	100	10	5

Current = 5
new_excluded = max(included, excluded)
= max(5,0) = 5
included = current + excluded
= 5 + 0 = 5
excluded = new_excluded = 5

Step-3:

5	5	10	100	10	5

Current = 10
new_excluded = max(included, excluded)
= max(5, 5) = 5
included = current + excluded
= 10 + 5 = 15
excluded = new_excluded = 5

Step-4:

5	5	10	100	10	5

Current = 100
new_excluded = max(included, excluded)
= max(15,5) = 15
included = current + excluded
= 100 + 5 = 105
excluded = new_excluded = 15

Step-5:

5	5	10	100	10	5

Current = 10
new_excluded = max(included, excluded)
= max(105, 15) = 105
included = current + excluded
= 10 + 15 = 25
excluded = new_excluded = 105

Step-6:

5	5	10	100	10	5

Current = 5
new_excluded = max(included, excluded)
= max(25, 105) = 105
included = current + excluded
= 5 + 105 = 110
excluded = new_excluded = 105

Finally:

Output = max(included, excluded)
= max(110, 105)
= 110

— Astik Anand

Implementation

```
def max_sum_with_no_adjacents(arr):
    included = excluded = 0

    for current in arr:
        # Get the new excluded which is max(included, excluded) as current element is
        # still not added to the included
        new_excluded = max(included, excluded)

        # Now change the included by adding current to excluded as no two adjacents should be present.
        included = excluded + current

        # Finally update the exluded with new_excluded
        excluded = new_excluded

    print("Max sum: {}".format(max(included, excluded)))



print("Example-1: max_sum_with_no_adjacents([5, 5, 10, 100, 10, 5])")
max_sum_with_no_adjacents([5, 5, 10, 100, 10, 5])

print("\nExample-2: max_sum_with_no_adjacents([1, 2, 3])")
max_sum_with_no_adjacents([1, 2, 3])

print("\nExample-3: max_sum_with_no_adjacents([1, 20, 3])")
max_sum_with_no_adjacents([1, 20, 3])
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:34:31 ~/Personal/Notebooks/Data Structures/1. Array $ python3 3_max_sum_with_no_adjacents.py
Example-1: max_sum_with_no_adjacents([5, 5, 10, 100, 10, 5])
Max sum: 110

Example-2: max_sum_with_no_adjacents([1, 2, 3])
Max sum: 4

Example-3: max_sum_with_no_adjacents([1, 20, 3])
Max sum: 20
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

3. Smallest subarray with sum greater than a given value***

Problem:

Given an array of integers and a number x, find the smallest subarray with sum greater than the given value.

Example:

Input: arr = {1, 4, 45, 6, 0, 19} x = 51
Output: 3
Minimum length subarray is {4, 45, 6}

Input: arr = {1, 10, 5, 2, 7} x = 9
Output: 1
Minimum length subarray is {10}

Input: arr = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250} x = 280
Output: 4
Minimum length subarray is {100, 1, 0, 200}

Input: arr = {1, 2, 4} x = 8
Output: Not Possible
Whole array sum is smaller than 8.

Approach-1: Brute Force

- Use two nested loops, the outer loop picks a starting element, the inner loop considers all elements (on right side of current start) as ending element.
- Whenever sum of elements between current start and end becomes more than the given number, update the result if current length is smaller than the smallest length so far.
- **Time Complexity: $O(n^2)$**

Approach-2: Efficient

- Initialize **current_sum = 0** and **min_length = n+1** and also starting and ending indexes **start = 0**, **end =0**
- Take all the elements one by one while end is smaller than n.
    - Keep adding array elements while current sum is smaller than x and **increment end**.
    - Once current_sum becomes greater than x, **start removing the trailing statement**.
    - **Update the min_length** if needed and **increment start**.

- To also print the subarray, store the final_start and final_end while updating the min_length.
- **Time Complexity: O(n)**

Implementation

```
def smallest_subarray_with_atleast_given_sum(arr, x):
    n = len(arr)
    # Initialize current sum and minimum length
    current_sum = 0; min_length = n + 1

    # Initialize starting and ending indexes
    start = 0; end = 0
    final_start = 0; final_end = 0

    # Take all the elements one by one while end is smaller than n.
    while (end < n):
        # Keep adding array elements while current sum is smaller than x and increment end
        while (current_sum <= x and end < n):
            current_sum += arr[end]
            end += 1

        # Once current_sum becomes greater than x, start removing the trailing statement
        # Update the min_length if needed and increment start
        while (current_sum > x and start < n):
            if (end - start < min_length):
                min_length = end - start
                final_start = start
                final_end = end

            current_sum -= arr[start]
            start+= 1

    if(min_length == n+1):
        print("No Subarray Possible for given sum: {}".format(x))
    else:
        print("Min Length: {} and Subarray: {} for given sum: {}".format(
          min_length, arr[final_start:final_end], x))



print("Example-1: smallest_subarray_with_atleast_given_sum([1, 4, 45, 6, 0, 19], 51)")
smallest_subarray_with_atleast_given_sum([1, 4, 45, 6, 0, 19], 51)

print("\nExample-2: smallest_subarray_with_atleast_given_sum([1, 10, 5, 2, 7], 9)")
smallest_subarray_with_atleast_given_sum([1, 10, 5, 2, 7], 9)

print("\nExample-3: smallest_subarray_with_atleast_given_sum([1, 11, 100, 1, 0, 200, 3, 2, 1, 250], 280)")
smallest_subarray_with_atleast_given_sum([1, 11, 100, 1, 0, 200, 3, 2, 1, 250], 280)

print("\nExample-4: smallest_subarray_with_atleast_given_sum([1, 2, 4], 8)")
smallest_subarray_with_atleast_given_sum([1, 2, 4], 8)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 22:28:01 ~/Personal/Notebooks/Data Structures/1. Array $ python3 4_smallest_subarray_with_atleast_given_sum.py
Example-1: smallest_subarray_with_atleast_given_sum([1, 4, 45, 6, 0, 19], 51)
Min Length: 3 and Subarray: [4, 45, 6] for given sum: 51

Example-2: smallest_subarray_with_atleast_given_sum([1, 10, 5, 2, 7], 9)
Min Length: 1 and Subarray: [10] for given sum: 9

Example-3: smallest_subarray_with_atleast_given_sum([1, 11, 100, 1, 0, 200, 3, 2, 1, 250], 280)
Min Length: 4 and Subarray: [100, 1, 0, 200] for given sum: 280

Example-4: smallest_subarray_with_atleast_given_sum([1, 2, 4], 8)
No Subarray Possible for given sum: 8
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

4. Longest increasing subarray

Problem:

Given an array containing n numbers. Find the length of the longest contiguous subarray such that every element in the subarray is strictly greater than its previous element in the same subarray.

Examples:

Input: arr[] = {5, 6, 3, 5, 7, 8, 9, 1, 2} ***Output:*** 5 ***Subarray:*** {3, 5, 7, 8, 9}

Input: arr[] = {12, 13, 1, 5, 4, 7, 8, 10, 10, 11} ***Output:*** 4 ***Subarray:*** {4, 7, 8, 10}

Approach:

- Loop over the array and check.
- If current element is greater than previous element increment the curr_length.
- Else, as current element is smaller, check if curr_length is greater than max_length and set max_length as curr_length, end=i and curr_length=1.
- Finally outside the loop check if curr_length is greater than max_length and set max_length as curr_length, end=i.

- Finally return max_length.
- **Time Complexity: O(n)**

Implementation:

```
def longest_increasing_subarray(arr):
    n = len(arr)
    max_length = curr_length = 1
    end = 0

    for i in range(1, n):
        if (arr[i] > arr[i-1]) :
            curr_length += 1
        else:
            if(curr_length > max_length):
                max_length = curr_length
                end = i
                curr_length = 1

    if(curr_length > max_length):
        max_length = curr_length
        end = i

    print("Longest Increasing Subarray: {}".format(arr[end-max_length:end]))



print("Example-1: longest_increasing_subarray([5, 6, 3, 5, 7, 8, 9, 1, 2])")
longest_increasing_subarray([5, 6, 3, 5, 7, 8, 9, 1, 2])

print("\nExample-2: longest_increasing_subarray([12, 13, 1, 5, 4, 7, 8, 10, 10, 11])")
longest_increasing_subarray([12, 13, 1, 5, 4, 7, 8, 10, 10, 11])
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 16:56:35 ~/Personal/Notebooks/Data Structures/1. Array $ python3 5_longest_increasing_subarray.py
Example-1: longest_increasing_subarray([5, 6, 3, 5, 7, 8, 9, 1, 2])
Longest Increasing Subarray: [3, 5, 7, 8, 9]

Example-2: longest_increasing_subarray([12, 13, 1, 5, 4, 7, 8, 10, 10, 11])
Longest Increasing Subarray: [5, 4, 7, 8, 10]
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

String

What is a string ?

- A string is a data type used in programming, such as an integer and floating point unit.
- But it is used to represent text rather than numbers.
- It is comprised of a set of characters that can also contain spaces and numbers.

```
====== Examples ======
1) "hamburger"
2) "I ate 3 hamburgers"
```

Applications of String

- Computers are widely used for word processing applications such as creating, inserting, updating, and modifying textual data.
- Besides this, we need to search for a particular pattern within a text, delete it, or replace it with another pattern.
- So, there is a lot that we as users do to manipulate the textual data.

Standard String Problems

1. Find if first string is a subsequence of second***

Problem:
Given two strings str1 and str2, find if str1 is a subsequence of str2. A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

Example:
Input: str1 = “AXY”, str2 = “ADXCPY” ***Output:*** True
Input: str1 = “AXY”, str2 = “YADXCP” ***Output:*** False
Input: str1 = “mtsdet”, str2 = “meetsandmeets” ***Output:*** True

Approach

- Traverse both strings from one side to other side.
- If a matching character found, move ahead in both strings, otherwise move ahead only in str2.
- **Time Complexity: O(n)**

Implementation

```
def check_subsequence(str1, str2):
    n1 = len(str1)
    n2 = len(str2)
    i = j = 0

    while(i < n1 and j < n2):
        if(str1[i] == str2[j]):
            i += 1
        j += 1

    print(i==n1)


print("Example-1: check_subsequence('AXY', 'ADXCPY')")
check_subsequence('AXY', 'ADXCPY')

print("\nExample-2: check_subsequence('AXY', 'YADXCP')")
check_subsequence('AXY', 'YADXCP')

print("\nExample-3: check_subsequence('mtsdet', 'meetsandmeets')")
check_subsequence('mtsdet', 'meetsandmeets')
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 14:02:02 ~/Personal/Notebooks/Algorithms/13. Miscellaneous $ python3 4_check_subsequence.py
Example-1: check_subsequence('AXY', 'ADXCPY')
True

Example-2: check_subsequence('AXY', 'YADXCP')
False

Example-3: check_subsequence('mtsdet', 'meetsandmeets')
True
```

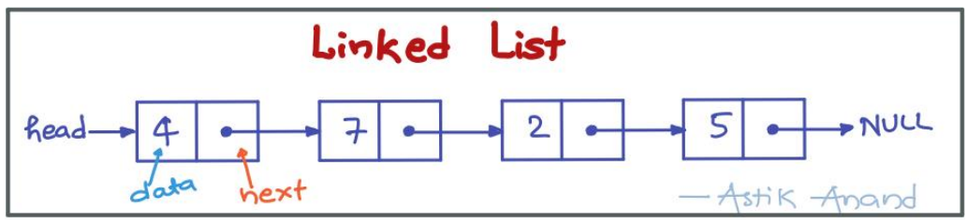
Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

Linked List

What is lInked list ?

- It is a linear data structure like array.
- But unlike arrays, elements are not stored at contiguous location, but are linked using pointers.



Operations:

- **insert(x):** Inserts an element into the list.
- **delete(x):** Deletes the element from the list.

Advantages over arrays:

1. Dynamic size
2. Ease of insertion/deletion

Drawbacks:

1. **Random access is not allowed**. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
2. **Extra memory space for a pointer** is required with each element of the list.
3. **Not cache friendly.** Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation of Linked List

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.

Each node in a list consists of at least two parts:

1. data
2. Pointer / Reference to the next node

Approach:

- Represent Linked list using Nodes which is a Data Type that consist **data** and **pointer to next node**.
- A linked list is initiated using a **head** which points to None initially.
- We can insert the node in linked list at head using **push()** method.
- The head starts pointing to most recent node inserted at head and successive nodes points to its next node and last node points to Null.
- We can print the nodes using **print_list()** function.
- Optionally we can also define **list_length()** which calculates length of list.

Implementation

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


class LinkedList:
    def __init__(self):
        self.head = None


    def list_length(self):
        temp = self.head
        length = 0
        while(temp):
            length += 1
            temp = temp.next
        return length


    def print_list(self):
        n = self.list_length()
        temp = self.head
        print('''+---+---+    '''*n)

        while(temp):
            print("| {} | •-|--->".format(temp.data), end="")
            temp = temp.next
        print("Null")

        print('''+---+---+    '''*n)


    def push(self, data):
        new_node = Node(data)
        new_node.next = self.head
```

```
        self.head = new_node


print("Linked List:")
linked_list = LinkedList()
linked_list.push(5)
linked_list.push(2)
linked_list.push(7)
linked_list.push(4)
linked_list.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:39:28 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 0_linked_list_representation.py
Linked List:
+---+---+    +---+---+    +---+---+    +---+---+
| 4 | •-|--->| 7 | •-|--->| 2 | •-|--->| 5 | •-|--->Null
+---+---+    +---+---+    +---+---+    +---+---+
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

Applications of Linked List

- Implementation of stacks and queues data structures.
- Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- **Image viewer –** Previous and next images are linked, hence can be accessed by next and previous button.
- **Previous and next page in web browser –** We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- **Music Player –** Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Standard Linked List Problems

1. Insert a node in linked list

Problem:

Given a linked list need to insert a node in it.

Example:

Linked List at Start: 5–>1–>6–>7—>Null

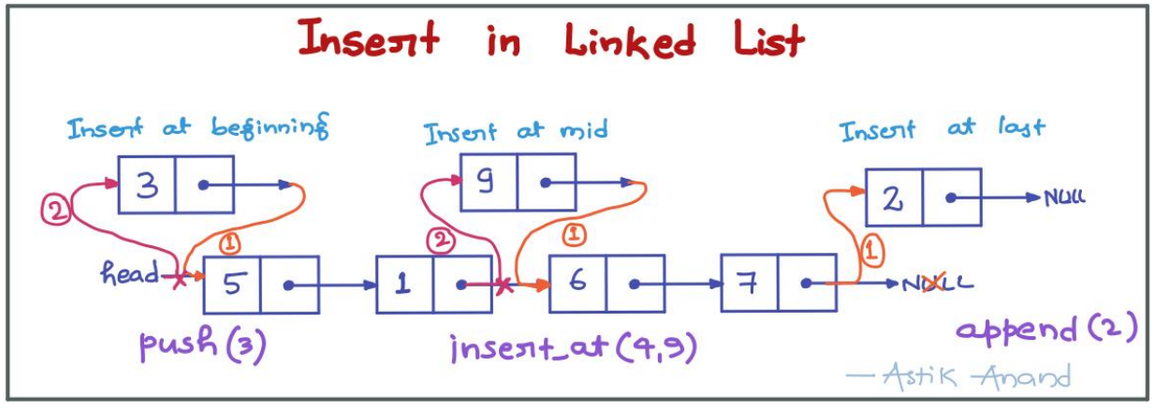
After push(3): 3–>5–>1–>6–>7–>Null

After insert_at(4, 9): 3–>5–>1–>9–>6–>7–>Null

After append(2): 3–>5–>1–>9–>6–>7–>2–>Null

A node can be inserted in 3 ways:

- At the front of the linked list: **push(data)**
- At a given position: **insert_at(position, data)**
- At the end of the linked list: **append(data)**



Approach:

- **push(data):**
  - Make a new_node.
  - (1) Next of new_node points to node where head was poiting earlier.
  - (2) Now, head points to new_node.
- **append(data):**
  - Make a new_node

    - (1) Find the last node using temp and make the next of last_node point to new_node.
- **insert_at(position, data):**
    - If position is 0 call push to insert at front.
    - Now find position to insert, need to jump only position-2 coz already at 1 and need to go 1 less
        - (Example-1: if position=5, already at 1 so ust 3 jumps to reach 4th posittion.)
    - Make new_node
    - (1) Next of new_node points to next of temp.
    - (2) Next of temp points to new_node.

Implementation

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


class LinkedList:
    def __init__(self):
        self.head = None


    def print_list(self):
        temp = self.head
        while(temp):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("Null")


    def push(self, data):
        # Make a new_node.
        new_node = Node(data)
        # (1) Next of new_node points to node where head was poiting earlier.
        new_node.next = self.head
        # (2) Now, head points to new_node.
        self.head = new_node


    def append(self, data):
        # Make a new_node.
        new_node = Node(data)

        # (1) Find the last node using temp and make the next of last_node point to new_node.
        temp = self.head
        while(temp.next):
            temp = temp.next
        temp.next = new_node


    def insert_at(self, position, data):
        # If position is 0 call push to insert at front.
```

```
        if(position == 0):
            self.push(data)
            return

        # Now find position to insert, need to jump only position-2 coz already at 1 and need to go 1 less.
        # (Example-1: if position=5, already at 1 so ust 3 jumps to reach 4th posittion.)
        temp = self.head
        while(temp.next and position-2 > 0):
            temp = temp.next
            position -= 1

        # Make new_node
        new_node = Node(data)
        # (1) Next of new_node points to next of temp.
        new_node.next = temp.next
        # (2) Next of temp points to new_node.
        temp.next = new_node


print("Linked List at Start:")
linked_list = LinkedList()
linked_list.push(7)
linked_list.push(6)
linked_list.push(1)
linked_list.push(5)
linked_list.print_list()
print("\nAfter push(3):")
linked_list.push(3)
linked_list.print_list()
print("\nAfter insert_at(4, 9):")
linked_list.insert_at(4, 9)
linked_list.print_list()
print("\nAfter append(2):")
linked_list.append(2)
linked_list.print_list()
```

Output

```
astik.anand@mac-C02XD95ZJG5H 01:18:36 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 1_insert_node_linked_list.py
Linked List at Start:
5-->1-->6-->7-->Null

After push(3):
3-->5-->1-->6-->7-->Null

After insert_at(4, 9):
3-->5-->1-->9-->6-->7-->Null

After append(2):
3-->5-->1-->9-->6-->7-->2-->Null
```

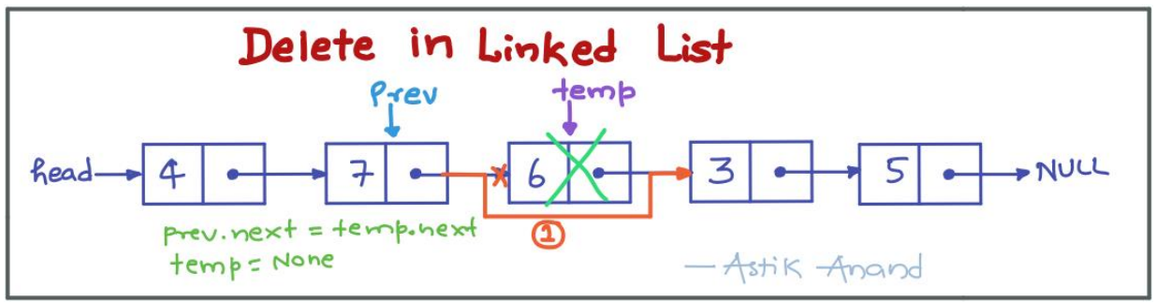
Complexity:

- **Time: O(1)**
- **Auxilliary Space: O(1)**

2. Delete a node in linked list

Problem:
Given a linked list need to delete a node from it. We will only delete the first occurrence of it.

Example:
Linked List at Start: 4–>7–>6–>3–>5–>Null
After delete(6): 4–>7–>3–>5–>Null
After delete(4): 7–>3–>5–>Null
After delete(2): 7–>3–>5–>Null
Element 2 not present in list.



Approach:

- Check if first node is the node to be deleted if it is point the head to next of temp and make temp None.
- Search for the node to be deleted using the **given data** in the nodes using **temp** and keep **prev_node** in store.
- Point the **next** of **prev_node** to the **next** of **temp** and make **temp None** if temp exists else element not present.

Implementation:

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


class LinkedList:
    def __init__(self):
        self.head = None


    def print_list(self):
        temp = self.head
        while(temp):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("Null")


    def push(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node


    def delete(self, data):
        temp = self.head

        # Check if first node is the node to be deleted if it is
        # point the head to next of temp and make temp None.
        if(temp and temp.data == data):
            self.head = temp.next
            temp = None
            return

        # Search for the node to be deleted using the given data in the nodes
        # using temp and keep prev in store.
        while(temp):
            if(temp.data == data):
                break
            prev = temp
            temp = temp.next

        # Point the next of prev to the next of temp and make temp None
        # if temp exists else element not present.
        if(temp):
            prev.next = temp.next
            temp = None
        else:
            print("Element {} not present in list.".format(data))


print("Linked List at Start:")
linked_list = LinkedList()
linked_list.push(5)
linked_list.push(3)
```

```
linked_list.push(6)
linked_list.push(7)
linked_list.push(4)
linked_list.print_list()
print("\nAfter delete(6):")
linked_list.delete(6)
linked_list.print_list()
print("\nAfter delete(4):")
linked_list.delete(4)
linked_list.print_list()
print("\nAfter delete(2):")
linked_list.delete(2)
linked_list.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 01:33:08 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 2_delete_node_linked_list.py
Linked List at Start:
4-->7-->6-->3-->5-->Null

After delete(6):
4-->7-->3-->5-->Null

After delete(4):
7-->3-->5-->Null

After delete(2):
Element 2 not present in list.
7-->3-->5-->Null
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

3. Reverse a linked list

Problem:

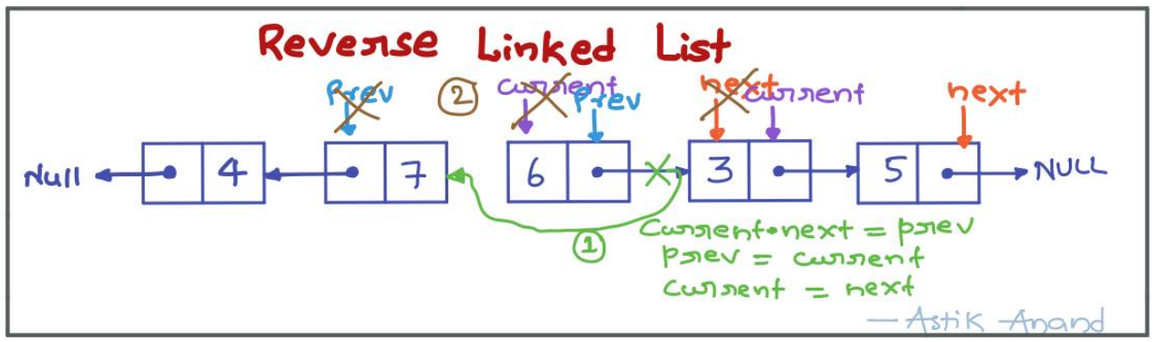
Given pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

Examples:

Input : 4->7->6->3->5->NULL Output : 5->3->6->7->4->NULL

Input : NULL Output : NULL

Input : 1->NULL Output : 1->NULL



Approach:

- Start with **prev_node as None** and **current_node as first node**.
- While **current_node is not None**, take the **next_node as next of current_node**, make the **next of current_node point to prev_node** and then update the **prev_node as current_node** and **current_node as next_node**.
- Make **head point to prev_node** which was the last node and now in reverse it becomes first.

Implementation:

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


class LinkedList:
    def __init__(self):
        self.head = None


    def print_list(self):
        temp = self.head
        while(temp):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("Null")


    def push(self, data):
        new_node = Node(data)
```

```
        new_node.next = self.head
        self.head = new_node


    def reverse(self):
        # Take prev_node as None and current_node as first node.
        prev_node = None
        current_node = self.head

        # While current_node doesn't become None keep going:
        while(current_node):
            # Take the next_node as next of current_node.
            next_node = current_node.next
            # Make the next of current_node point to prev_node
            current_node.next = prev_node
            # Update the prev_node as current_node and current_node as next_node
            prev_node = current_node
            current_node = next_node

        # Make head point to prev_node which was the last node and now in reverse it becomes first.
        self.head = prev_node



print("Linked List at Start:")
linked_list = LinkedList()
linked_list.push(5)
linked_list.push(3)
linked_list.push(6)
linked_list.push(7)
linked_list.push(4)
linked_list.print_list()
print("\nAfter reverse():")
linked_list.reverse()
linked_list.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:09:47 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 3_reverse_linked_list.py
Linked List at Start:
4-->7-->6-->3-->5-->Null

After reverse():
5-->3-->6-->7-->4-->Null
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

4. Merge 2 sorted linked list

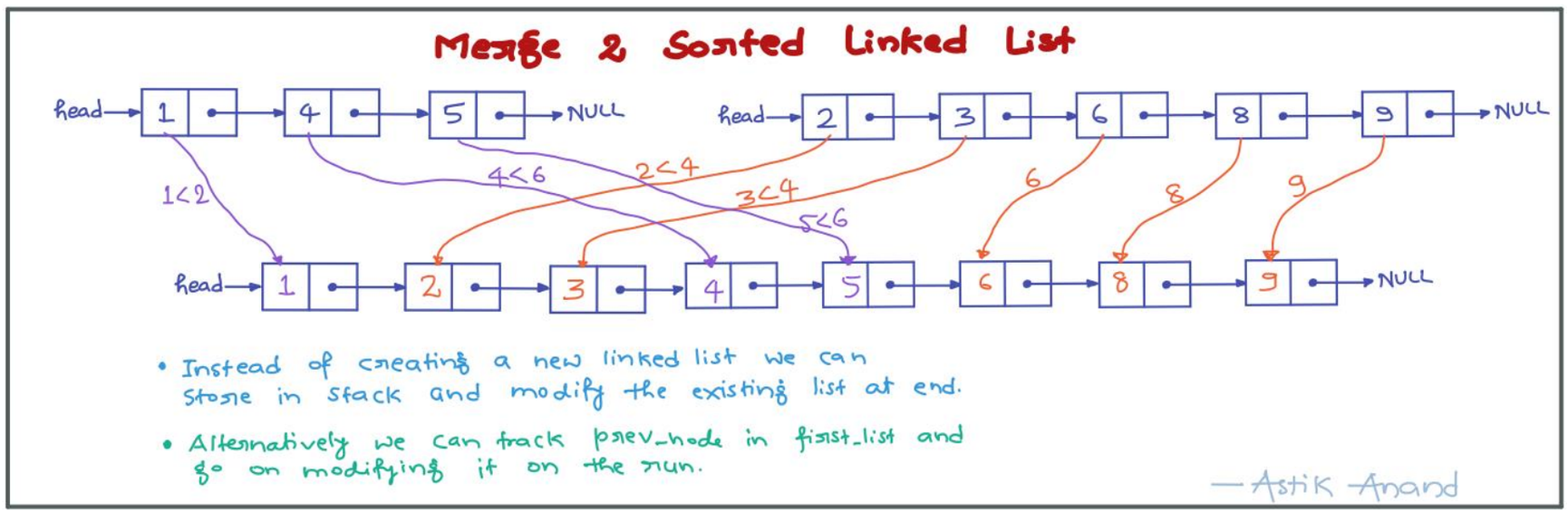
Problem:

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order.

SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists.

Example:

list_1: 1->4->5 **list_2:** is 2->3->6->8->9 **merged_list:** 1->2->3->4->5->6->8->9



Approach:

- Take 2 pointers temp_1 and temp_2 to track both lists and a stack to put the data.
- While both temp_1 and temp_2 are exists keep on going:
    - If data of temp_1 is lesser temp_1 is selected and incremented else temp_2 is selected and incrmented.
- If elements are left in temp_1, push all of them to stack.
- If elements are left in temp_2, push all of them to stack.
- Now, finally make head point to None and create the linked_list from popping from stack and pushing all nodes to it.

Implementation:

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


class LinkedList:
    def __init__(self):
        self.head = None


    def print_list(self):
        temp = self.head
        while(temp):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("Null")


    def push(self, data):
        new_node = Node(data)

        new_node.next = self.head
        self.head = new_node


    def merge_2_sorted_linked_lists(self, sorted_list_2):
        # Take 2 pointers temp_1 and temp_2 to track both lists and a stack to put the data.
        temp_1 = self.head
        temp_2 = sorted_list_2.head
        stack = []

        # While both temp_1 and temp_2 are exists keep on going:
        while(temp_1 and temp_2):
            # If data of temp_1 is lesser temp_1 is selected and incremented
            # else temp_2 is selected and incrmented.
            if(temp_1.data <= temp_2.data):
                stack.append(temp_1.data)
                temp_1 = temp_1.next
            else:
                stack.append(temp_2.data)
                temp_2 = temp_2.next

        # If elements are left in temp_1, push all of them to stack.
        while(temp_1):
            stack.append(temp_1.data)
            temp_1 = temp_1.next

        # If elements are left in temp_2, push all of them to stack.
        while(temp_2):
            stack.append(temp_2.data)
            temp_2 = temp_2.next

        # Now, finally make head point to None and create the linked_list from popping
        # from stack and pushing all nodes to it.
```

```
        self.head = None
        while(stack):
            self.push(stack.pop())



print("First Sorted Linked List:")
linked_list_1 = LinkedList()
linked_list_1.push(5)
linked_list_1.push(4)
linked_list_1.push(1)
linked_list_1.print_list()
print("\nSecond Sorted Linked List:")
linked_list_2 = LinkedList()
linked_list_2.push(9)
linked_list_2.push(8)
linked_list_2.push(6)
linked_list_2.push(3)
linked_list_2.push(2)
linked_list_2.print_list()
print("\nMerged List:")
linked_list_1.merge_2_sorted_linked_lists(linked_list_2)
linked_list_1.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:14:26 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 4_merge_2_sorted_linked_lists.py
First Sorted Linked List:
1-->4-->5-->Null

Second Sorted Linked List:
2-->3-->6-->8-->9-->Null

Merged List:
1-->2-->3-->4-->5-->6-->8-->9-->Null
```

Complexity:

- **Time: O(n) :** Whosever size is larger
- **Auxilliary Space: O(n1+n2):** Can be very easily done in O(1) with prev_pointer for linked_list_1.

5. Reverse linked list in a group of size***

Problem:

Given a linked list, write a function to reverse every k nodes.

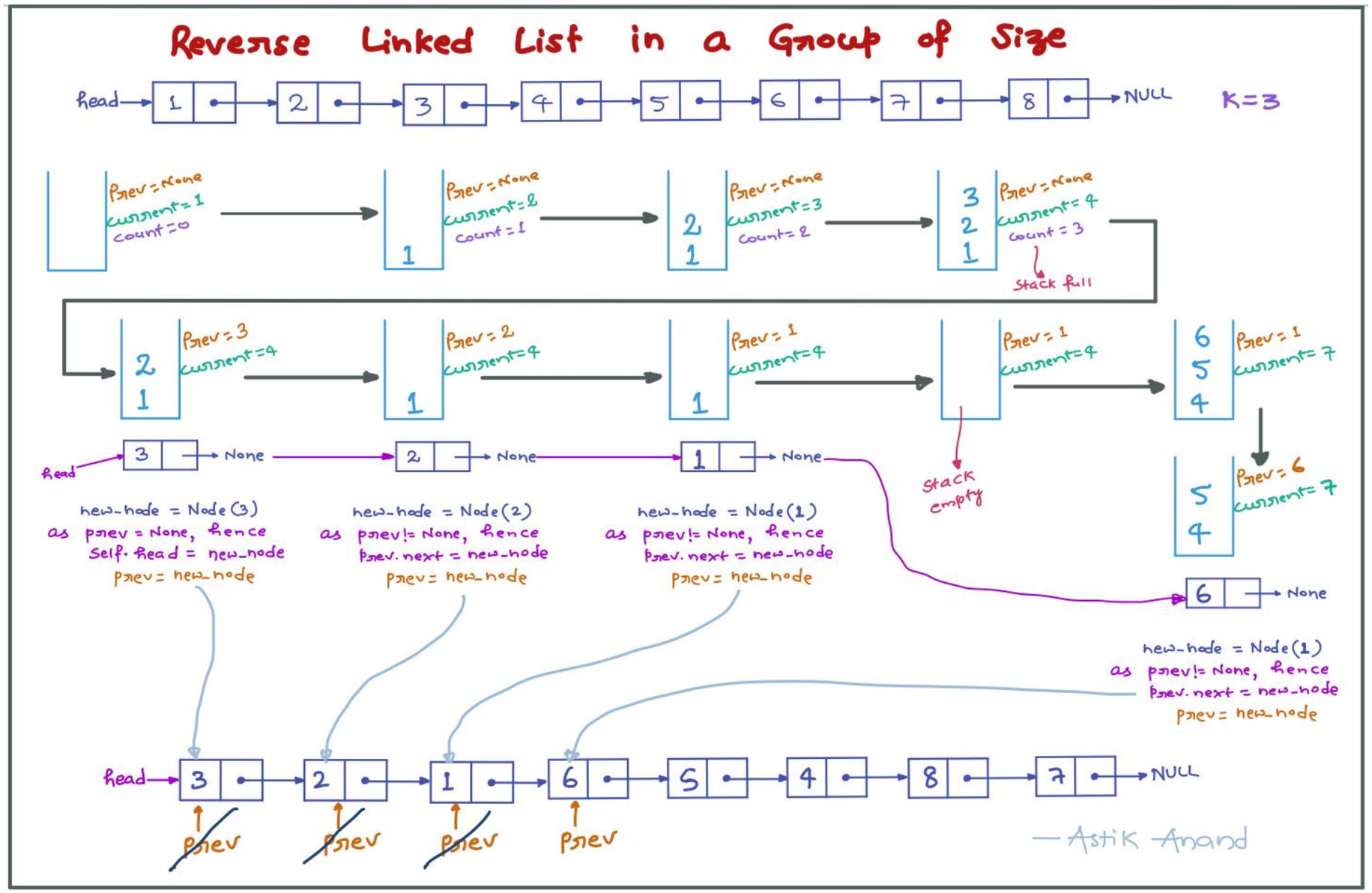
Examples:

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3

Output: 3->2->1->6->5->4->8->7->NULL

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5

Output: 5->4->3->2->1->8->7->6->NULL



Approach:

- Take **prev** as None, **current_nod**e as **start** node and an **empty stack** to store at most k items.
- Till the current_node is not None:
    - Take **at most k elements in stack** and keep **moving the current_node**.

- After k nodes are pushed to stack **current_node** will point to **k+1th** node but **prev** will still be **unchanged**.
- Now start **popping from stack** and **make new_node and check**:
  - If **prev is None**, then set the **head to point to prev** (it will only happen for the first element when prev is None).
  - Else set **next of prev as new_node**.
  - Update the **prev as new_node**.

Implementation:

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


class LinkedList:
    def __init__(self):
        self.head = None


    def print_list(self):
        temp = self.head
        while(temp):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("Null")

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node


    def reverse_in_group_of_size(self, k):
        # Take prev as None, start the current_node from first node and
        # take an empty stack to store atmost k items.
        prev = None
        current_node = self.head
        stack = []

        # Till the current_node is not None.
        while(current_node):
            count = 0
            # Take at most k elements in stack and keep moving the current_node.
            while(count<k and current_node):
                stack.append(current_node.data)
                current_node = current_node.next
                count += 1

            # After k nodes are pushed to stack current will point to k+1 the node
            # but prev will still be unchanged.
            # Now start popping from stack and make new_node and check:
            while(stack):
                new_node = Node(stack.pop())
```

```
                # If prev is None, then set the head to point to prev
                # (it will only happen for the first element when prev is None).
                if(prev is None):
                    self.head = new_node
                else:     # Else set next of prev as new_node
                    prev.next = new_node

                # Update the prev as new_node.
                prev = new_node



print("Linked List at Start:")
linked_list = LinkedList()
linked_list.push(8)
linked_list.push(7)
linked_list.push(6)
linked_list.push(5)
linked_list.push(4)
linked_list.push(3)
linked_list.push(2)
linked_list.push(1)
linked_list.print_list()
print("\nAfter reverse_in_group_of_size(3):")
linked_list.reverse_in_group_of_size(3)
linked_list.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:38:43 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 5_reverse_linked_list_in_group_k_size.py
Linked List at Start:
1-->2-->3-->4-->5-->6-->7-->8-->Null

After reverse_in_group_of_size(3):
3-->2-->1-->6-->5-->4-->8-->7-->Null
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(K)**

6. Add 2 numbers represented by linked list

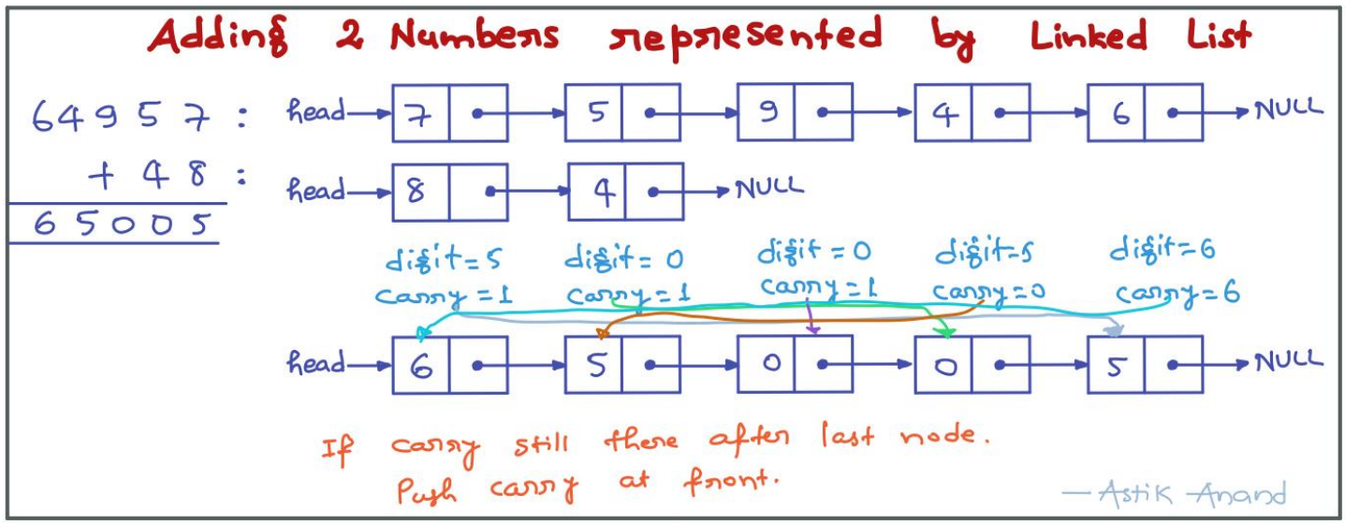
Problem:

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Examples:

Inputs: Num1 = 984 (4->8->9) Num2 = 978 (8->7->9) sum = 1962 (1->9->6->2)

Inputs: Num1 = 64957 7->5->9->4->6 Num2 = 48 8->4 sum = 65005 (6->5->0->0->5)



Approach:

- Take 2 pointers to track both the lists, take carry = 0 and also initialize result list.
- Now while any of temp_1 or temp_2 is there keep going.
  - Get the sum: carry + first_list digit sum if exists else 0 + second_list digit sum if exists else 0.
  - If sum geater than 10 then get the digit and carry separated.
  - Push the digit to result list and update the temp_1 and temp_2 pointers if exists else None.
- At the end if carry still is not zero, then push carry also to the result list.
- Lastly, set the head of the list as result list's head.

Implementation:

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


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

```
        self.head = None


    def print_list(self):
        temp = self.head
        while(temp):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("Null")


    def push(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node


    def add_2_numbers(self, number):
        # Take 2 pointers to track both the lists, take carry = 0 and also initialize result list.
        temp_1 = self.head
        temp_2 = number.head
        carry = 0
        result = LinkedList()

        # Now while any of temp_1 or temp_2 is there keep going.
        while(temp_1 or temp_2):
            # Get the sum: carry + first_list digit sum if exists
            # else 0 + second_list digit sum if exists else 0.
            digit_sum = carry + (0 if not temp_1 else temp_1.data) + (0 if not temp_2 else temp_2.data)
            # If sum geater than 10 then get the digit and carry separated.
            if(digit_sum >= 10):
                carry = 1
                digit_sum -= 10
            else:
                carry = 0

            # Push the digit to result list and update the temp_1 and temp_2 pointers if exists else None.
            result.push(digit_sum)
            temp_1 = None if not temp_1 else temp_1.next
            temp_2 = None if not temp_2 else temp_2.next

        # At the end if carry still is not zero, then push carry also to the result list.
        if(carry != 0):
            result.push(carry)

        # Lastly, set the head of the list as result list's head.
        self.head = result.head



print("Example-1:")
print("First Number: {}".format(984))
number_1 = LinkedList()
number_1.push(9)
number_1.push(8)
number_1.push(4)
print("Second Number: {}".format(978))
```

```
number_2 = LinkedList()
number_2.push(9)
number_2.push(7)
number_2.push(8)
print("Sum of both:")
number_1.add_2_numbers(number_2)
number_1.print_list()

print("\nExample-2:")
print("First Number: {}".format(64957))
number_1 = LinkedList()
number_1.push(6)
number_1.push(4)
number_1.push(9)
number_1.push(5)
number_1.push(7)
print("Second Number: {}".format(48))
number_2 = LinkedList()
number_2.push(4)
number_2.push(8)
print("Sum of both:")
number_1.add_2_numbers(number_2)
number_1.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:55:52 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 6_add_2_numbers_through_linked_list.py
Example-1:
First Number: 984
Second Number: 978
Sum of both:
1-->9-->6-->2-->Null

Example-2:
First Number: 64957
Second Number: 48
Sum of both:
6-->5-->0-->0-->5-->Null
```

Complexity:

- **Time: O(n) :** Whosever size is larger
- **Auxilliary Space: O(n):**

7. Detect and remove loop in linked list***

Problem:

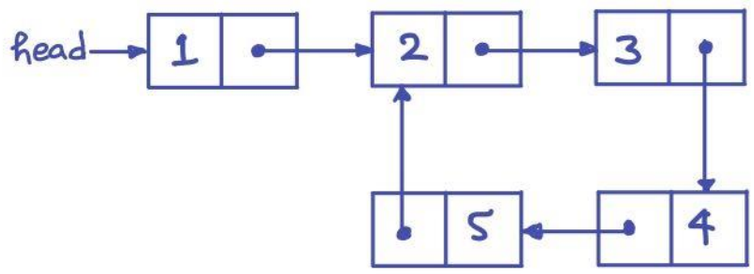
Check whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false.

Examples:

Input: 1->2->3->4->5->2 Output: 1->2->3->4->5->Null

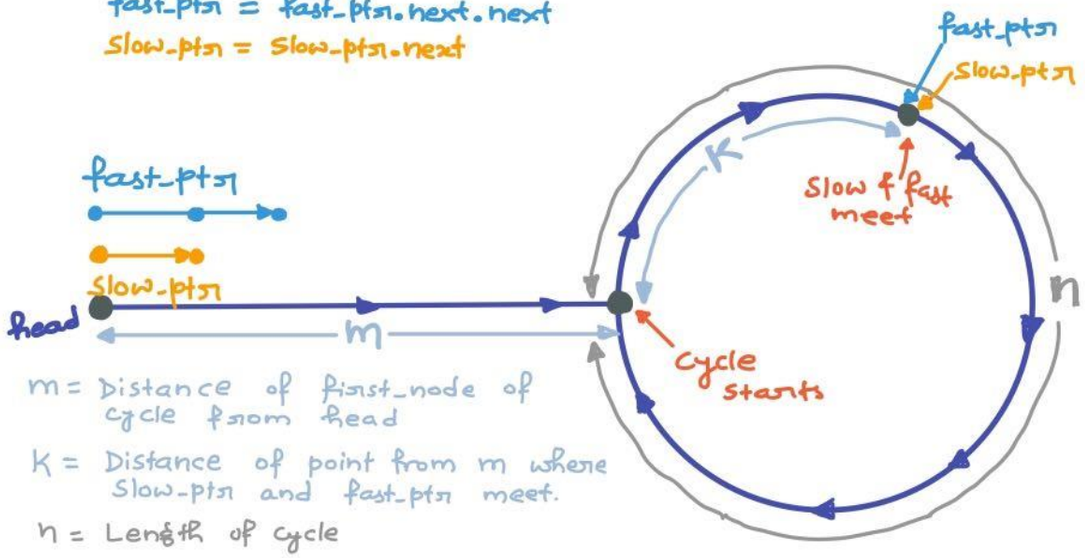
Input: 1->2->3->4. Output: False

Detect and Remove Loop in Linked List



Cycle-Finding : Hare-Tortoise
(Floyd's Cycle Finding Algorithm)

fast_ptr = fast_ptr.next.next
slow_ptr = slow_ptr.next



Distance travelled by fast_ptr = 2 × Distance travelled by slow_ptr

$m + n \times x + K = 2 \times (m + n \times y + K)$

x : No. of cycles travelled by fast_ptr before first meet.
y : No. of cycles travelled by slow_ptr before first meet.

$m + K = (x - 2y)n$

m+K is a multiple of n.

Now, if slow_ptr starts from head so after m steps it reaches beginning of Loop.

Also fast_ptr starts at K distance from start of loop and travels at same speed after m steps reaches beginning of loop.

Coz K+m = n.

Hence, they meet at start.

— Astik Anand

Approach:

- If only 0 node or 1 node and that too has no self loop the cycle doesn't exist.
- Initialize fast_ptr and slow_ptr to detect the loop.
- If both are equal then cycle exists and then just start the slow_ptr from head and fast_ptr from same place at same speed and check the next of both.

- The point at where next of both meet makes the loop just break using fast_ptr as we were checking next so we still have fast_ptr.
- To break loop just make next of fast_ptr as None.

Implementation:

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


class LinkedList:
    def __init__(self):
        self.head = None


    def print_list(self):
        temp = self.head
        while(temp):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("Null")


    def push(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node


    def detect_and_remove_loop(self):
        # If only 0 nodes or 1 node and that too  has no self.loop then cycle doesn't exist.
        if(self.head is None or self.head.next is None):
            print("Cycle doesn't exist!")
            return

        # Initialize fast and slow pointers
        slow_ptr = fast_ptr = self.head
        slow_ptr = slow_ptr.next
        fast_ptr = fast_ptr.next.next

        # Detecting the Loop
        while(slow_ptr and fast_ptr and fast_ptr.next):
            if(slow_ptr == fast_ptr):
                # Loop detected start removing the loop using fast_ptr and slow_ptr at same speed
                # and starting slow_ptr from head and fast_ptr from meet point
                slow_ptr =  self.head
                while(slow_ptr.next != fast_ptr.next):
                    slow_ptr = slow_ptr.next
                    fast_ptr = fast_ptr.next

                print("Cycle exists from Node-{} to Node-{}!".format(fast_ptr.data, fast_ptr.next.data))
                # Make next of fast_ptr None to break loop
                fast_ptr.next = None
```

```
                return

            slow_ptr = slow_ptr.next
            fast_ptr = fast_ptr.next.next

        # If reaches here mean cycle not exists
        print("Cycle doesn't exist!")



print("Detect and Remove Loop Example-1:")
linked_list = LinkedList()
linked_list.head = Node(1)
linked_list.head.next = Node(2)
linked_list.head.next.next = Node(3)
linked_list.head.next.next.next = Node(4)
linked_list.head.next.next.next.next = Node(5)
#Create a loop for testing
linked_list.head.next.next.next.next.next =  linked_list.head.next
linked_list.detect_and_remove_loop()
print ("After removing loop:")
linked_list.print_list()

print("\nDetect and Remove Loop Example-2:")
linked_list = LinkedList()
linked_list.push(5)
linked_list.push(3)
linked_list.push(6)
linked_list.push(7)
linked_list.push(4)
linked_list.detect_and_remove_loop()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:57:50 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 7_detect_and_remove_loop.py
Detect and Remove Loop Example-1:
Cycle exists from Node-5 to Node-2!
After removing loop:
1-->2-->3-->4-->5-->Null

Detect and Remove Loop Example-2:
Cycle doesn't exist!
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

Note: The problem can also be solved in O(n) using hashing the visited node address but it will take O(N) space.

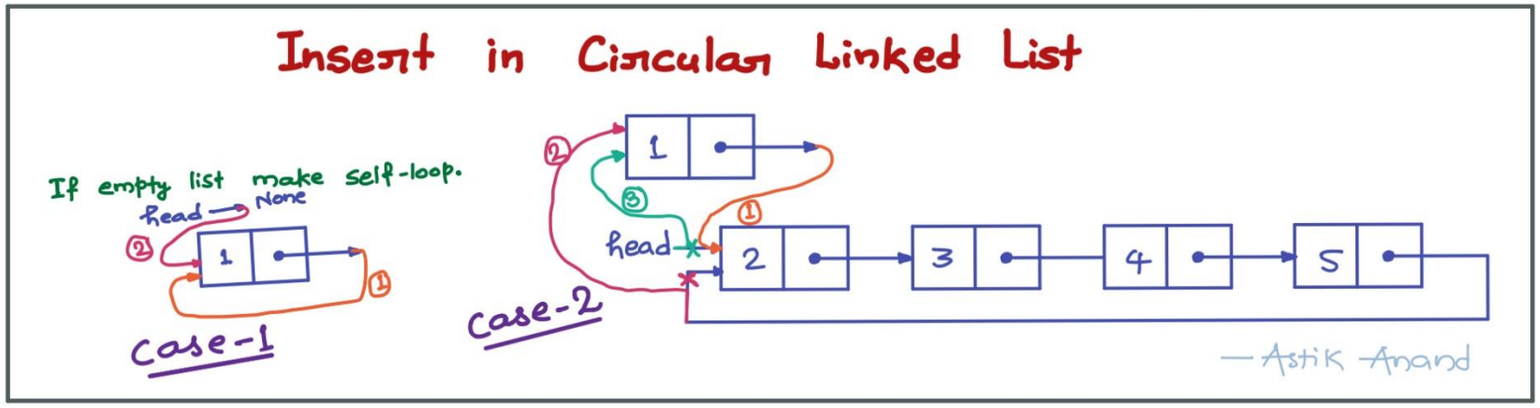
8. Circular linked list insertion

Problem:

Insert a node in circular linked list.

Example:

Input: 1->2->3->4->5->1 num=6 Output: 6->1->2->3->4->5->6



Approach:

- Make a new_node and take current_node as starting node.
- **Case-1: If this is first node being inserted make the self loop.**
  i. Make the next of new_node point to new_node itself (self-loop).
  ii. Point the head to new_node.
- **Case-2: Insert at starting before head.**
  i. Make the next of new_node point to where head was pointing earlier.
  ii. Find the last node and make last_node's next point to new_node.
  iii. Point the head to new_node.

Implementation:

```
class Node:
    def __init__(self, data):
```

```
        self.data = data
        self.next = None


class CircularLinkedList:
    def __init__(self):
        self.head = None


    def push(self, data):
        current_node = self.head
        new_node = Node(data)

        #### Case-1: If this is first node being inserted make the self loop.
        if current_node is None:
            # (1) Make the next of new_node point to new_node itself (self-loop).
            new_node.next = new_node
            # (2) Point the head to new_node.
            self.head = new_node
        #### Case-2: Insert at starting before head.
        else:
            # (1) Make the next of new_node point to where head was pointing earlier.
            new_node.next = self.head
            # (2) Find the last node and make last_node's next point to new_node.
            last_node = self.head
            while(last_node.next != self.head):
                last_node = last_node.next
            last_node.next = new_node
            # (3) Point the head to new node.
            self.head = new_node


    def print_list(self):
        temp = self.head
        while(temp.next != self.head):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("{}-->{}".format(temp.data, temp.next.data))



print("Original Circular Linked List:")
cll = CircularLinkedList()
cll.push(5)
cll.push(4)
cll.push(3)
cll.push(2)
cll.push(1)
cll.print_list()
print("\nCircular Linked List After Insertion of 6:")
cll.push(6)
cll.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 17:59:15 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 8_circular_linked_list_insertion.py
Original Circular Linked List:
1-->2-->3-->4-->5-->1

Circular Linked List After Insertion of 6:
6-->1-->2-->3-->4-->5-->6
```

Complexity:

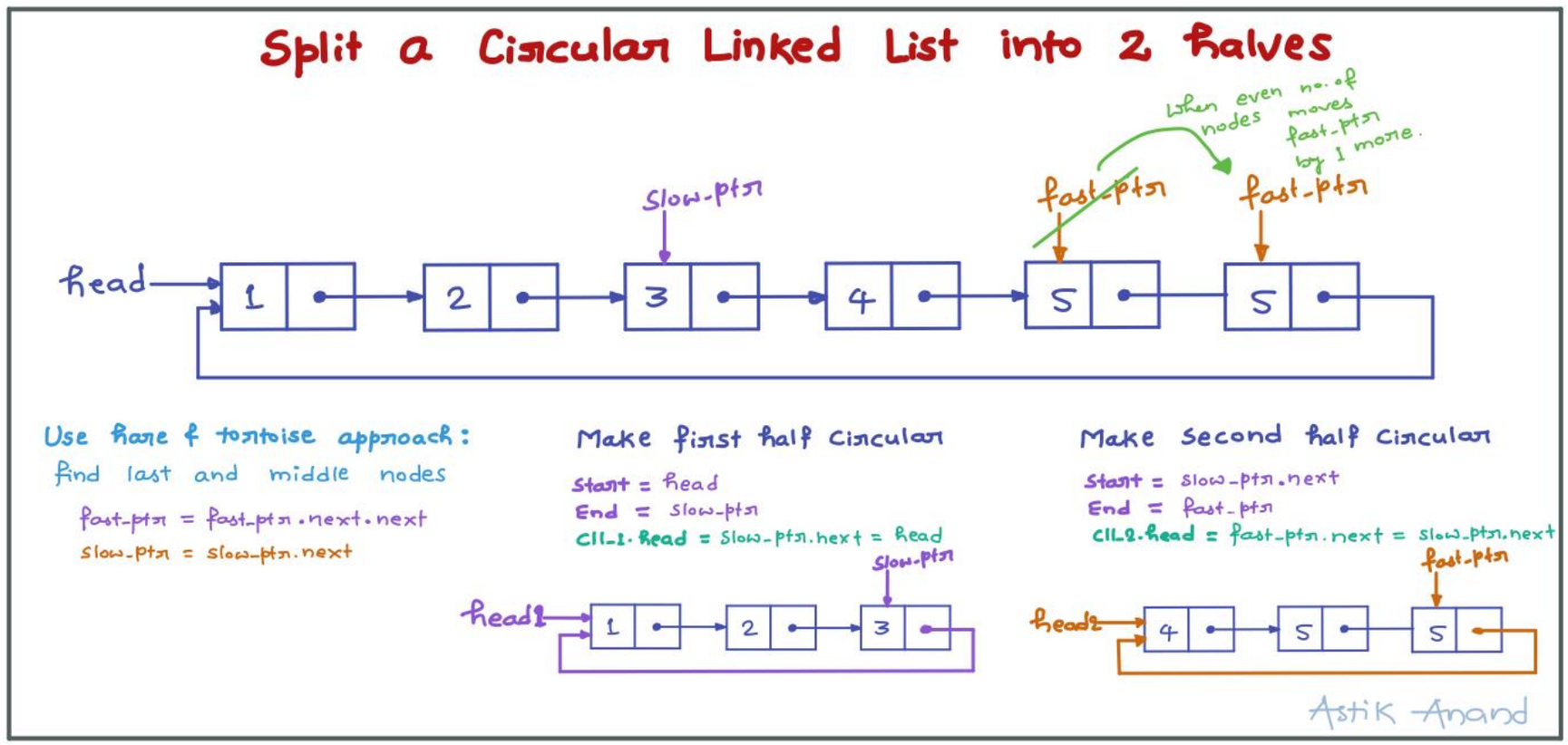
- **Time: O(1)**
- **Auxilliary Space: O(1)**

9. Split a circular linked list in 2 halves***

Problem:

Split a circular linked list into 2 circular list of equal sizes.

If there are odd number of nodes, then first list should contain one extra.



Approach:

- If **no nodes** then just **return**.
- Use **hare & tortoise approach** to find the **middle** and **last node** using fast_ptr = fast_ptr.next.next and slow_ptr = slow_ptr.next both starting from head.
- If Even no. of nodes shift the fast_ptr to point to last using fast_ptr = fast_ptr.next.
- **Make second half circular:**
  - start = slow_ptr.next and end = fast_ptr
  - cll_2.head = fast_ptr.next = slow_ptr.next
- **Make first half circular**
  - start = head and end = slow_ptr

Implementation:

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

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


    def push(self, data):
        new_node = Node(data)
        # new_node's next will point to the previous_first_node which head was pointing earlier
        new_node.next = self.head

        # Only 1 node : self loop
        if self.head is None:
            new_node.next = new_node
        else:
            # last_node's next will point to new_node
            last_node = self.head
            while(last_node.next != self.head):
                last_node = last_node.next
            last_node.next = new_node

        # Now the head will point to new_node
        self.head = new_node


    def print_list(self):
        temp = self.head
        while(temp.next != self.head):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("{}".format(temp.data))


def split_into_halves(self, second_cll):
    if(self.head is None):
        return

    fast_ptr = slow_ptr = self.head
    # If Odd no. of nodes in then fast_ptr->next becomes head and
    # if Even no. of nodes fast_ptr->next->next becomes head
    while(fast_ptr.next!=self.head and fast_ptr.next.next!=self.head):
        fast_ptr = fast_ptr.next.next
        slow_ptr = slow_ptr.next

    #======== Making 2nd half circular ========#
    # Start of 2nd half: slow_ptr.next
    # End of 2nd half : fast_ptr (Odd no. of nodes) || fast_ptr.next (Even no. of nodes)
    if fast_ptr.next.next == self.head:
        fast_ptr = fast_ptr.next
    second_cll.head = slow_ptr.next
    fast_ptr.next = slow_ptr.next

    #======== Making 1st half circular ========#
    # Start of 1st half: head (self.head = self.head)
    # End of 1st half : slow_ptr)
    slow_ptr.next = self.head
```

```
print("Example-1(Odd Nodes): Original Circular Linked List:")
cll = CircularLinkedList()
cll.push(5)
cll.push(4)
cll.push(3)
cll.push(2)
cll.push(1)
cll.print_list()
print("Circular Linked List after Splitting:")
second_cll = CircularLinkedList()
cll.split_into_halves(second_cll)
cll.print_list()
second_cll.print_list()

print("\nExample-2(Even Nodes): Original Circular Linked List:")
cll = CircularLinkedList()
cll.push(6)
cll.push(5)
cll.push(4)
cll.push(3)
cll.push(2)
cll.push(1)
cll.print_list()
print("Circular Linked List after Splitting")
second_cll = CircularLinkedList()
cll.split_into_halves(second_cll)
cll.print_list()
second_cll.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 18:49:48 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 9_split_circular_linked_list_into_halves.py
Example-1(Odd Nodes): Original Circular Linked List:
1-->2-->3-->4-->5
Circular Linked List after Splitting:
1-->2-->3
4-->5

Example-2(Even Nodes): Original Circular Linked List:
1-->2-->3-->4-->5-->6
Circular Linked List after Splitting
1-->2-->3
4-->5-->6
```

Complexity:

- **Time: O(n)**

- **Auxilliary Space: O(1)**

10. Sorted insert in circular linked list

Problem:

Write a function to insert a new value in a **sorted** Circular Linked List.

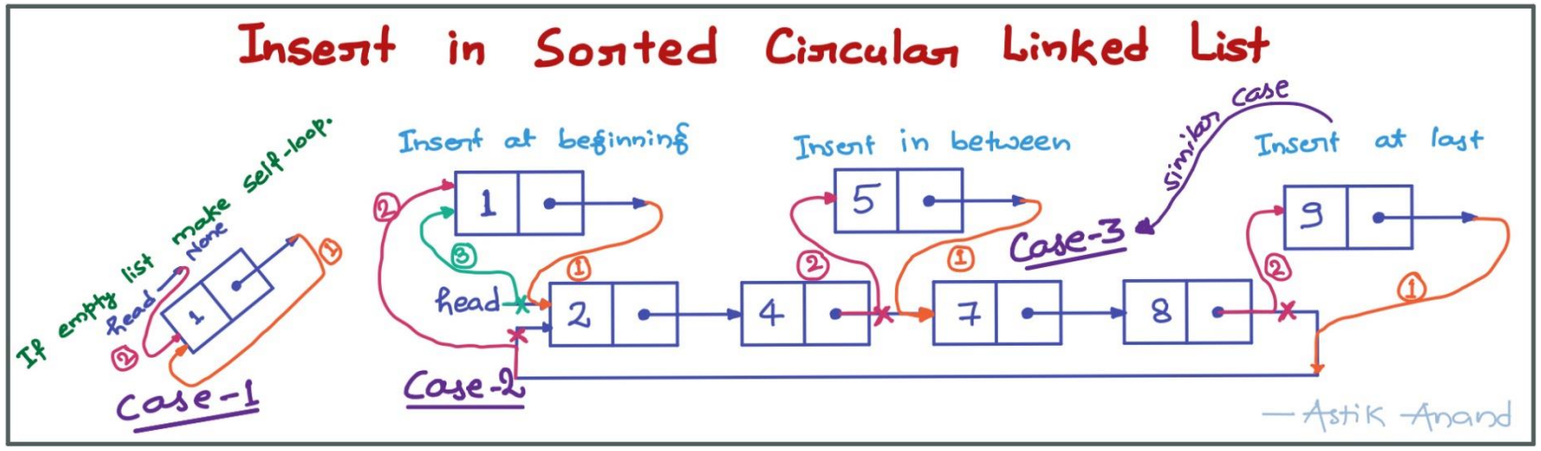
Examples:

Input: 2->4->7->8->2

Insert: 1 Output: 1-2->4->7->8->1 (At beginning)

Insert: 5 Output: 1->2->4->5->7->8->1 (In between)

Input: 9 Output: 1->2->4->5->7–>8->9->1 (At last)



Approach:

- Make a new_node and take current_node as starting node.
- **Case-1: If this is first node being inserted make the self loop.**
  i. Make the next of new_node point to new_node itself (self-loop).
  ii. Point the head to new_node.
- **Case-2: Insert at starting before head.**

i. Make the next of new_node point to where head was pointing earlier.
ii. Find the last node and make last_node's next point to new_node.
iii. Point the head to new_node.

- **Case-3: Insert by searching (linear search) the appropriate place to insert.**
  i. Make the next of new_node point to where current was pointing earlier (current.next).
  ii. Make the next of current_node point to new_node.

Implementation:

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


class CircularLinkedList:
    def __init__(self):
        self.head = None


    def sorted_insert(self, data):
        current_node = self.head
        new_node = Node(data)

        #### Case-1: If this is first node being inserted make the self loop.
        if current_node is None:
            # (1) Make the next of new_node point to new_node itself (self-loop).
            new_node.next = new_node
            # (2) Point the head to new_node.
            self.head = new_node

        #### Case-2: Insert at starting before head.
        elif(data <= current_node.data):
            # (1) Make the next of new_node point to where head was pointing earlier.
            new_node.next = self.head

            # (2) Find the last node and make last_node's next point to new_node.
            last_node = self.head
            while(last_node.next != self.head):
                last_node = last_node.next
            last_node.next = new_node

            # (3) Point the head to new_node.
            self.head = new_node

        #### Case-3: Insert by searching (linear search) the appropriate place to insert.
        else:
            while( current_node.next != self.head and data > current_node.next.data):
                current_node = current_node.next

            # (1) Make the next of new_node point to where current was pointing earlier (current.next).
            new_node.next = current_node.next
            # (2) Make the next of current_node point to new_node.
```

```
            current_node.next = new_node



    def print_list(self):
        temp = self.head
        while(temp.next != self.head):
            print("{}-->".format(temp.data), end="")
            temp = temp.next
        print("{}-->{}".format(temp.data, temp.next.data))



print("Original Circular Linked List:")
cll = CircularLinkedList()
cll.sorted_insert(4)
cll.sorted_insert(2)
cll.sorted_insert(8)
cll.sorted_insert(7)
cll.print_list()
print("\nInsertion at Beginning (1):")
cll.sorted_insert(1)
cll.print_list()
print("\nInsertion in Between (5):")
cll.sorted_insert(5)
cll.print_list()
print("\nInsertion at Last(9):")
cll.sorted_insert(9)
cll.print_list()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 04:14:24 ~/Personal/Notebooks/Data Structures/1. Linked List $ python3 10_sorted_insert_circular_linked_list.py
Original Circular Linked List:
2-->4-->7-->8-->2

Insertion at Beginning (1):
1-->2-->4-->7-->8-->1

Insertion in Between (5):
1-->2-->4-->5-->7-->8-->1

Insertion at Last(9):
1-->2-->4-->5-->7-->8-->9-->1
```

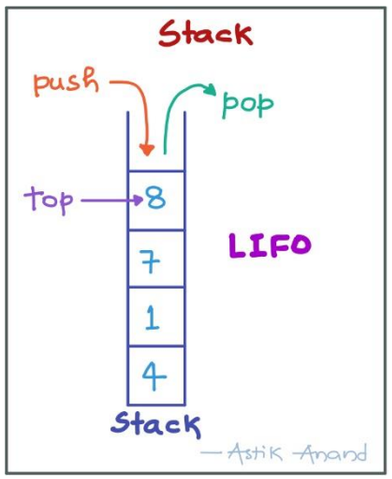
Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

Stack

What is Stack ?

- It is a linear data structure which follows a particular order in which the operations are performed.
- The order is of type last in and first out (LIFO) or (FILO).



Practical Understanding:

- There are many real life examples of stack:
- Consider the simple example of plates stacked over one another in canteen.
- The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time.
- So, it can be simply seen to follow LIFO/FILO order.

Operations:

- **push(key):** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **pop():** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek() or Top():** Returns top element of stack.
- **isEmpty():** Returns true if stack is empty, else false.

Time Complexities:

- push(), pop(), isEmpty() and peek() all take **O(1)** time.
- We do not run any loop in any of these operations.

Implementation Ways:

There are two ways to implement a stack:

1. **Using array**
2. **Using linked list**

Array Implementation of Stack

- To implement any data structure we need to define all its operations.
- So here we need to define all the operations of stack i.e. push(), pop(), top(), is_empty().
- **Pros:** Easy to implement. Memory is saved as pointers are not involved.
- **Cons:** It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

Implementation

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


    def is_empty(self):
        return len(self.stack) == 0


    def push(self, data):
        self.stack.append(data)


    def pop(self):
        if(self.is_empty()):
            return "Underflow"
        return self.stack.pop()


    def top(self):
        if(self.is_empty()):
            return "Underflow"
        return self.stack[-1]
```

```
print("Example:- Array Implementation of Stack.")
my_stack = Stack()
print("Push: 4, 1, 7, 8 to stack.")
my_stack.push(4)
my_stack.push(1)
my_stack.push(7)
my_stack.push(8)
print("Pop the first element: {}".format(my_stack.pop()))
print("Top element in stack: {}".format(my_stack.top()))
print("Is stack empty? {}".format(my_stack.is_empty()))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 14:02:01 ~/Personal/Notebooks/Data Structures/2. Stacks $ python3 0_stack_array_implementation.py
Example:- Array Implementation of Stack.
Push: 4, 1, 7, 8 to stack.
Pop the first element: 8
Top element in stack: 7
Is stack empty? False
```

Linked List Implementation of Stack

- Just define all the operations of stack i.e. push(), pop(), top(), is_empty().
- **Pros:** The linked list implementation of stack can grow and shrink according to the needs at runtime.
- **Cons:** Requires extra memory due to involvement of pointers.

Implementation

```
class StackNode:
    def __init__(self, data):
        self.data = data
        self.next = None


class Stack:
    def __init__(self):
        self.head = None


    def is_empty(self):
        return True if self.head is None else False


    def push(self, data):
        new_stack_node = StackNode(data)
        new_stack_node.next = self.head
        self.head = new_stack_node
```

```
    def pop(self):
        if(self.head is None):
            return "Underflow"

        temp = self.head
        self.head = self.head.next
        return temp.data


    def top(self):
        if(self.head is None):
            return "Underflow"

        return self.head.data



print("Example:- Linked List Implementation of Stack.")
my_stack = Stack()
print("Push: 4, 1, 7, 8 to stack.")
my_stack.push(4)
my_stack.push(1)
my_stack.push(7)
my_stack.push(8)
print("Pop the first element: {}".format(my_stack.pop()))
print("Top element in stack: {}".format(my_stack.top()))
print("Is stack empty? {}".format(my_stack.is_empty()))
```

Output:

```
[astikanand@Developer-MAC 02:19:12 ~/Interview-Preparation/Data Structures/2. Stacks $ python3 0_stack_linked_list_implementation.py
[Example:- Linked List Implementation of Stack.
Push: 4, 1, 7, 8 to stack.
Pop the first element: 8
Top element in stack: 7
Is stack empty? False
```

Applications of Stack

- Stack is used to evaluate prefix, postfix and infix expressions.
- An expression can be represented in prefix, postfix or infix notation and stack can be used to convert one form of expression to another.
- Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.
- Parnthesis Checking
- Backtracking
- Recursive Function Calls

Standard Stack Problems

1. Infix to Postfix Conversion***

Problem:

Given an infix expression conversation it to a postfix operation.

Example:

Input: `a+b*(c-d)` Output: `abcd-*+`

Input: `a+b*(c^d-e)^(f+g*h)-i` Output: `abcd^e-fgh*+^*+i-`

Infix expression: ** The expression of the form **a op b. When an operator is in-between every pair of operands.**

Postfix expression: The expression of the form **a b op**. When an operator is followed for every pair of operands.

Why postfix representation of the expression?

- The compiler scans the expression either from left to right or from right to left.
- *Consider the below expression: **a + b * c + d***
  - The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it.
  - The result is then added to d after another scan.
- The repeated scanning makes it very in-efficient.
- It is better to convert the expression to postfix(or prefix) form before evaluation.
- The corresponding expression in postfix form is: abc*+d+.
- The postfix expressions can be evaluated easily using a stack.

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
   - If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.
   - Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.
6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

Implementation

```
def infix_to_postfix(expr):
    operator_precedence = {'+':1, '-':1, '*':2, '/':2, '%':2, '^':3}
    # operator_stack
    stack = []

    # Start the scan from left to right
    for char in expr:
        # If scanned char is not an operand, then output it.
        if char.isalpha():
            print("{}".format(char), end="")

        # Else if the scanned character is an '(', push it to the stack.
        elif(char == "("):
            stack.append(char)

        # If the scanned character is an ')', pop and output from the stack until an '(' is encountered.
        elif(char == ")"):
            while(len(stack)>0 and stack[-1] != "("):
                print("{}".format(stack.pop()), end="")
            stack.pop()

        # Pop the operator from the stack until operator_precedence[char] <= operator_precedence[stack[-1]]
        # Push the scanned operator to the stack
        else:
            while(len(stack)>0 and stack[-1]!="("
                and operator_precedence[char] <= operator_precedence[stack[-1]]):
                print("{}".format(stack.pop()), end="")

            # If either stack is empty or precedence of scanned opeartor is greater
            # than the top of stack, Push it to stack
            stack.append(char)

    # Pop and output from the stack until it is not empty.
    while(len(stack)>0):
        print("{}".format(stack.pop()), end="")
    print()



print("Example-1: Infix-to-postfix")
infix_to_postfix("a+b*(c-d)")

print("\nExample-2: Infix-to-postfix")
infix_to_postfix("a+b*(c^d-e)^(f+g*h)-i")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:24:06 ~/Personal/Notebooks/Data Structures/2. Stacks $ python3 1_infix_to_postfix.py
Example-1: Infix-to-postfix
abcd-*+

Example-2: Infix-to-postfix
abcd^e-fgh*+^*+i-
```

Complexity:

- **Time: O(n)** : n is size of expression
- **Auxilliary Space: O(n)**

2. Postfix Evaluation

Problem:

Evaluate the value of the postfix expression.

Example:

Input: `12 3 *` ***Output:*** `36`

Input: `2 3 10 * + 9 -` ***Output:*** `23`

Why postfix evaluation?

- The Postfix notation is used to represent algebraic expressions.
- The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Algorithm

1. Create a stack to store operands.
2. Scan the given expression and do following for every scanned element.
    - If the element is a number, push it into the stack
    - If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3. When the expression is ended, the number in the stack is the final answer

Implementation:

```
def evaluate_postfix(expression):
    expr = expression.split()
    stack = []
    for i in expr:
        if i.isdigit():
            stack.append(i)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(str(eval(val2 + i + val1)))

    print(stack.pop())


print("Example-1: evaluate_postfix('12 3 *')")
evaluate_postfix('12 3 *')

print("Example-2: evaluate_postfix('2 3 10 * + 9 -')")
evaluate_postfix('2 3 10 * + 9 -')
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:07:25 ~/Personal/Notebooks/Data Structures/2. Stacks $ python3 2_postfix_evaluation.py
Example-1: evaluate_postfix('12 3 *')
36
Example-2: evaluate_postfix('2 3 10 * + 9 -')
23
```

Complexity:

- **Time: O(n)** : n is size of expression
- **Auxilliary Space: O(n)**

3. Check for balanced parentheses in an expression

Problem:

Given an expression string exp , write a program to examine whether the pairs and the orders of `"{", "}", "(", ")", "[", "]"` are correct in expression.

Example:

Input: `"[()]{}{[()()]()}"` ***Output:*** Balanced
Input: `"[(])"` ***Output:*** Not Balanced

Algorithm:

1. Declare a stack.
2. Now traverse the expression.
    - If the current character is a starting bracket (**‘(‘ or ‘{‘ or ‘[‘**) then push it to stack.
    - If the current character is a closing bracket (**‘)’ or ‘}’ or ‘]’**) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
3. After complete traversal, if there is some starting bracket left in stack then “not balanced”

Implementation:

```
def check_balanced_bracket(expression):
    stack = []
    balanced = True

    for bracket in expression:
        # If current is opening bracket push closing bracket to stack
        if bracket=='(':
            stack.append(')')
        elif bracket=='{':
            stack.append('}')
        elif bracket=='[':
            stack.append(']')
        # If current is not opening bracket or the bracket doesnt match the top of stack -> Unbalanced
        elif not stack or stack.pop() != bracket:
           balanced = False
           break

    # If Stack is empty it is balanced else not balanced
    if stack or not balanced:
        print("Not Balanced")
    else:
        print("Balanced")


print("Example-1: check_balanced_bracket('[()]{}{[()()]()}')")
check_balanced_bracket('[()]{}{[()()]()}')
print("\nExample-2: check_balanced_bracket('[(])')")
check_balanced_bracket('[(])')
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:52:05 ~/Personal/Notebooks/Data Structures/2. Stacks $ python3 3_balanced_parantheses_in_an_expression.py
Example-1: check_balanced_bracket('[()]{}{[()()]()}')
Balanced

Example-2: check_balanced_bracket('[(])')
Not Balanced
```

Complexity:

- **Time: O(n)** : n is parenthesis size
- **Auxilliary Space: O(n)**

4. Next greater for every element***

Problem:

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

Input: **[4, 5, 2, 25]**

Element	NGE
4	5
5	25
2	25
25	-1

Input: **[13, 7, 6, 12]**

Element	NGE
13	-1
7	12
6	12
12	-1

Input: **[7, 11, 15, 6]**

Element	NGE
7	11
11	15
15	-1
6	-1

Notes:

- For any array, rightmost element always has next greater element as -1.
- For an array which is sorted in decreasing order, all elements have next greater element as -1.

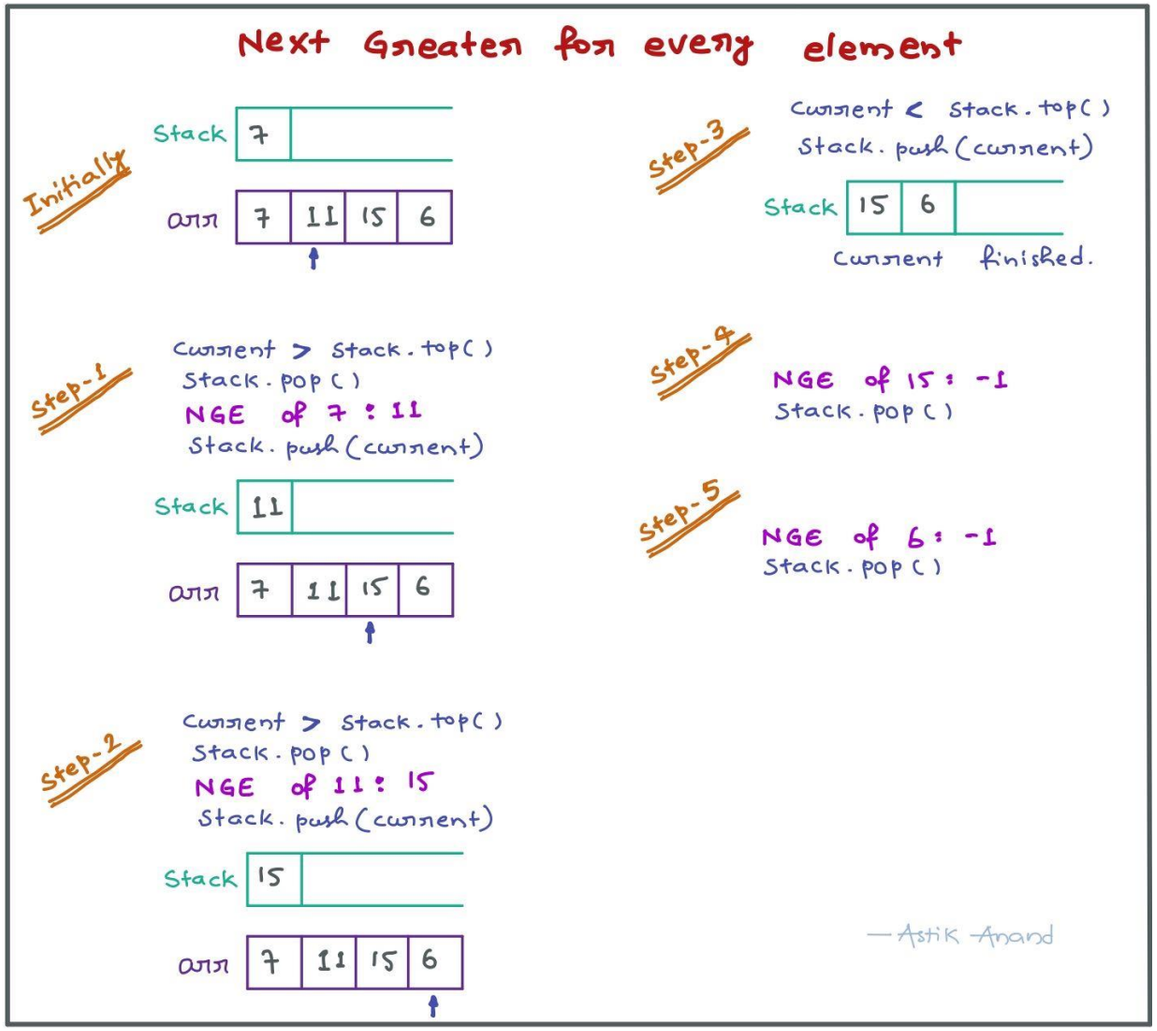
Approach-1: Brute-Force

- Use two loops: The outer loop picks all the elements one by one.
- The inner loop looks for the first greater element for the element picked by the outer loop.
- If a greater element is found then that element is printed as next, otherwise -1 is printed.
- **Time Complexity: $O(n^2)$**

Approach-2: Using Stacks

1. Push the first element to stack.
2. Pick rest of the elements one by one and follow the following steps in loop.
    - Mark the current element as current.
    - If stack is not empty, then pop an element from stack as popped_element.
        - If popped_element is smaller than current, then current is NGE for popped element.
        - keep popping while popped elements are smaller than current and stack is not empty.
        - If popped_element is greater than next, then push the popped_element back to stack.
    - Finally, push current to stack so that we can find next greater for it.
3. After iterating over the loop, the remaining elements in stack do not have the next greater so -1 for them.

Example-Run:



Implementation:

```
def next_greater_for_every_element(arr):
    n = len(arr)
    stack = []

    # Push the first element to stack
    stack.append(arr[0])

    print("Element -- NGE")

    # Iterate for rest of the elements
    for i in range(1, n):
        current = arr[i]
```

```
        # If stack is not empty, then pop an element from stack as popped_element
        if stack:
            popped_element = stack.pop()

            # If popped_element is smaller than current, then current is NGE for popped element
            # keep popping while popped elements are smaller than current and stack is not empty
            while popped_element < current:
                print("{} -------> {}".format(popped_element, current))
                if not stack:
                    break
                popped_element = stack.pop()

            # If popped_element is greater than next, then push the popped_element back to stack
            if popped_element > current:
                stack.append(popped_element)

        # Push current to stack so that we can find next greater for it.
        stack.append(current)

    # After iterating over the loop, the remaining elements in stack do not have the next greater
    # so -1 for them
    while stack:
        popped_element = stack.pop()
        current = -1
        print("{} -------> {}".format(popped_element, current))



print("Example-1: next_greater_for_every_element([4, 5, 2, 25])")
next_greater_for_every_element([4, 5, 2, 25])
print("\nExample-2: next_greater_for_every_element([13, 7, 6, 12])")
next_greater_for_every_element([13, 7, 6, 12])
print("\nExample-3: next_greater_for_every_element([11, 13, 21, 3])")
next_greater_for_every_element([11, 13, 21, 3])
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 21:10:42 ~/Personal/Notebooks/Data Structures/3. Stacks $ python3 5_next_greater_for_every_element.py
Example-1: next_greater_for_every_element([4, 5, 2, 25])
Element -- NGE
4 -------> 5
2 -------> 25
5 -------> 25
25 -------> -1

Example-2: next_greater_for_every_element([13, 7, 6, 12])
Element -- NGE
6 -------> 12
7 -------> 12
12 -------> -1
13 -------> -1

Example-3: next_greater_for_every_element([11, 13, 21, 3])
Element -- NGE
11 -------> 13
13 -------> 21
3 -------> -1
21 -------> -1
```

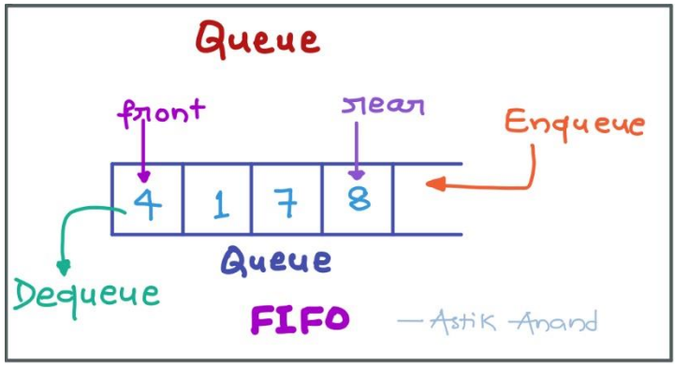
Complexity:

- **Time: O(n)** : n is array size
- **Auxilliary Space: O(n)**

Queue

What is Queue ?

- Like Stack, **Queue** is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (**FIFO**).



Practical Understanding:

- An example of queue is any queue of consumers for a resource where the consumer that came first is served first.

Difference b/w Stack & Queue:

- The difference between stacks and queues is in removing.
- In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations:

- **enqueue(key):** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **dequeue():** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **front():** Get the front item from queue.
- **rear():** Get the last item from queue.

Time Complexities:

- Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1).
- There is no loop in any of the operations.

Implementation Ways:

There are two ways to implement a stack:

1. **Using array**

2. **Using linked list**

Array implementation of Queue

- To implement any data structure we need to define all its operations.
- So here we need to define all the operations of queue i.e. enqueue(), dequeue(), front(), rear().
- For implementing queue, we need to keep track of two indices, front and rear.
- We enqueue an item at the rear and dequeue an item from front.
- If we simply increment front and rear indices, then there may be problems, front may reach end of the array.
- The solution to this problem is to increase front and rear in circular manner.

Pros: Easy to implement. Memory is saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

Implementation

```
class Queue:
    def __init__(self):
        self.queue = []

    def is_empty(self):
        return len(self.queue) == 0

    def enqueue(self, data):
        self.queue.append(data)

    def dequeue(self):
        if(self.is_empty()):
            return "Empty Queue"
        return self.queue.pop(0)

    def front(self):
        if(self.is_empty()):
            return "Empty Queue"
        return self.queue[0]

    def rear(self):
        if(self.is_empty()):
            return "Empty Queue"
        return self.queue[-1]
```

```
print("Example:- Array Implementation of Queue.")
my_queue = Queue()
print("Enqueue: 10, 20, 30, 40 to queue.")
my_queue.enqueue(10)
my_queue.enqueue(20)
my_queue.enqueue(30)
my_queue.enqueue(40)
print("Dequeue the first element: {}".format(my_queue.dequeue()))
print("Front element in queue: {}".format(my_queue.front()))
print("Rear element in queue: {}".format(my_queue.rear()))
print("Is queue empty? {}".format(my_queue.is_empty()))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 19:25:59 ~/Personal/Notebooks/Data Structures/3. Queues $ python3 0_queue_array_implementation.py
Example:- Array Implementation of Queue.
Enqueue: 10, 20, 30, 40 to queue.
Dequeue the first element: 10
Front element in queue: 20
Rear element in queue: 40
Is queue empty? False
```

Linked List Implementation of Queue

- Just define all the operations of queue i.e. enqueue(), dequeue(), front(), rear().

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.

Implementation

```
class QueueNode:
    def __init__(self, data):
        self.data = data
        self.next = None


class Queue:
    def __init__(self):
        self.head = None


    def is_empty(self):
        return True if self.head is None else False
```

```
def enqueue(self, data):
    new_queue_node = QueueNode(data)
    new_queue_node.next = self.head
    self.head = new_queue_node


def dequeue(self):
    if(self.is_empty()):
        return "Empty Queue"

    if(self.head.next is None):
        dequeued_data = self.head.data
        self.head = None
        return dequeued_data

    temp = self.head
    while(temp.next.next):
        temp = temp.next

    dequeued_data = temp.next.data
    temp.next = None
    return dequeued_data


def front(self):
    if(self.is_empty()):
        return "Empty Queue"
    temp = self.head
    while(temp.next):
        temp = temp.next

    return temp.data


def rear(self):
    if(self.is_empty()):
        return "Empty Queue"
    return self.head.data



print("Example:- LinkedList Implementation of Queue.")
my_queue = Queue()
print("Enqueue: 10, 20, 30, 40 to queue.")
my_queue.enqueue(10)
my_queue.enqueue(20)
my_queue.enqueue(30)
my_queue.enqueue(40)
print("Dequeue the first element: {}".format(my_queue.dequeue()))
print("Front element in queue: {}".format(my_queue.front()))
print("Rear element in queue: {}".format(my_queue.rear()))
print("Is queue empty? {}".format(my_queue.is_empty()))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 20:05:08 ~/Personal/Notebooks/Data Structures/3. Queues $ python3 0_queue_linked_list_implementation.py
Example:- LinkedList Implementation of Queue.
Enqueue: 10, 20, 30, 40 to queue.
Dequeue the first element: 10
Front element in queue: 20
Rear element in queue: 40
Is queue empty? False
```

Applications of Queue

- Queue is used when things don't have to be processed immediatly, but have to be processed in First InFirst Out order like Breadth First Search.
- This property of Queue makes it also useful in following kind of scenarios.
    i. When a resource is shared among multiple consumers.
        - Examples include CPU scheduling, Disk Scheduling.
    ii. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes.
        - Examples include IO Buffers, pipes, file IO, etc.

Standard Queue Problems

1. Queue using Stacks***

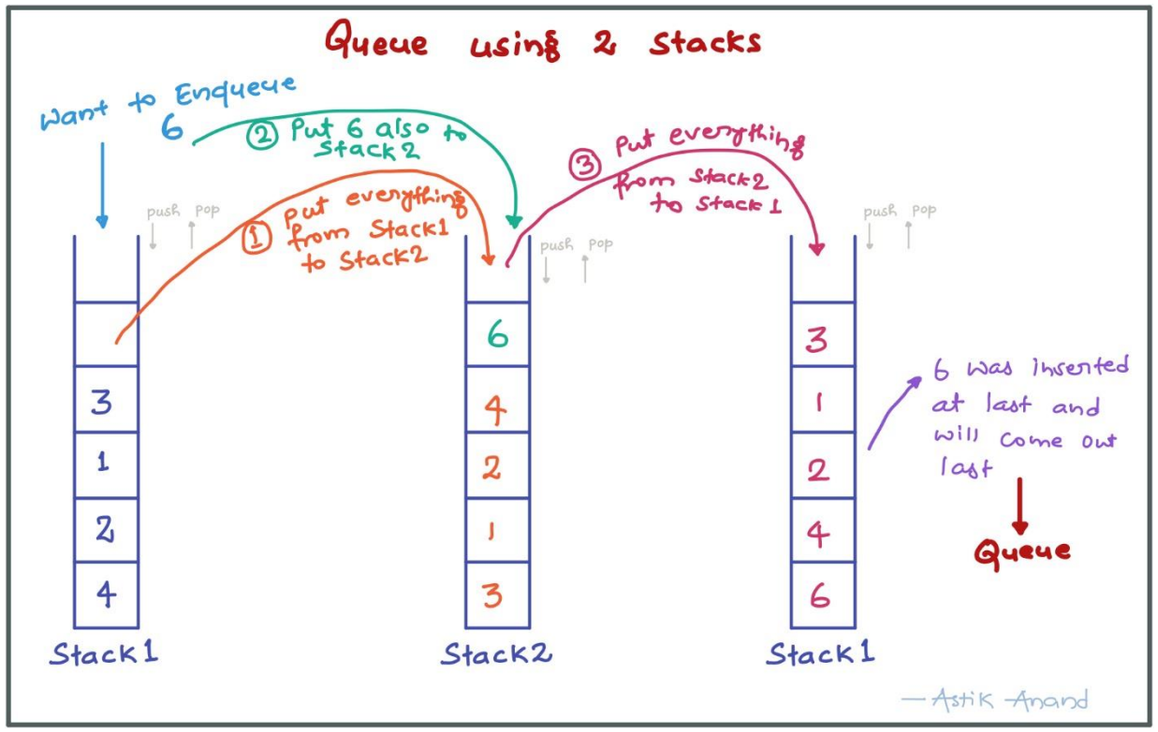
Problem:

Given a **stack** data structure with **push** and **pop** operations.

The task is to **implement a queue** using **instances of stack** data structure and **operations** on them.

Approach:

- A queue can be implemented using 2 stacks.
- Let queue to be implemented be q and stacks used to implement q be stack1 and stack2, then q can be implemented in two ways:
    i. By making **enqueue()** operation costly.
    ii. By making **dequeue()** operation costly.



Algorithm with costly Enqueue:

- **enqueue(q, x):** time complexity will be O(n)
  - While stack1 is not empty, push everything from stack1 to stack2.
  - Push x to stack1 (assuming size of stacks is unlimited).
  - Push everything back to stack1.
- **dequeue():** time will be O(1)
  - If stack1 is empty then error.
  - Pop an item from stack1 and return it.

Implementation:

```
from queue import LifoQueue

class Queue:
    def __init__(self):
        self.stack1 = LifoQueue()
        self.stack2 = LifoQueue()


    def enqueue(self, data):
```

```
        # (1) Put all the data first to stack2
        while(not self.stack1.empty()):
            self.stack2.put(self.stack1.get())

        # (2) Put the current data also to stack2
        self.stack2.put(data)

        # (3) Move evrything back to stack1 from stack2
        while(not self.stack2.empty()):
            self.stack1.put(self.stack2.get())


    def dequeue(self):
        if(self.stack1.empty()):
            print("Empty Queue")
            return
        else:
            return self.stack1.get()


    def front(self):
        if(self.stack1.empty()):
            print("Empty Queue")
            return
        else:
            return self.stack1.queue[-1]


    def rear(self):
        if(self.stack1.empty()):
            print("Empty Queue")
            return
        else:
            return self.stack1.queue[0]


    def is_empty(self):
        return self.stack1.empty()



print("Example:- Queue Using Stacks:")
my_queue = Queue()
print("Enqueue: 3, 1, 2, 4 to queue.")
my_queue.enqueue(3)
my_queue.enqueue(1)
my_queue.enqueue(2)
my_queue.enqueue(4)
my_queue.enqueue(6)

print("6 is enqueued to the queue.\n")
print("Dequeue the element: {}".format(my_queue.dequeue()))
print("Front element in queue: {}".format(my_queue.front()))
print("Rear element in queue: {}".format(my_queue.rear()))
print("Is queue empty? {}".format(my_queue.is_empty()))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 16:01:20 ~/Personal/Notebooks/Data Structures/3. Queues $ python3 3_queue_using_stacks.py
Example:- Queue Using Stacks:
Enqueue: 3, 1, 2, 4 to queue.
6 is enqueued to the queue.

Dequeue the element: 3
Front element in queue: 1
Rear element in queue: 6
Is queue empty? False
```

2. Stack using Queues***

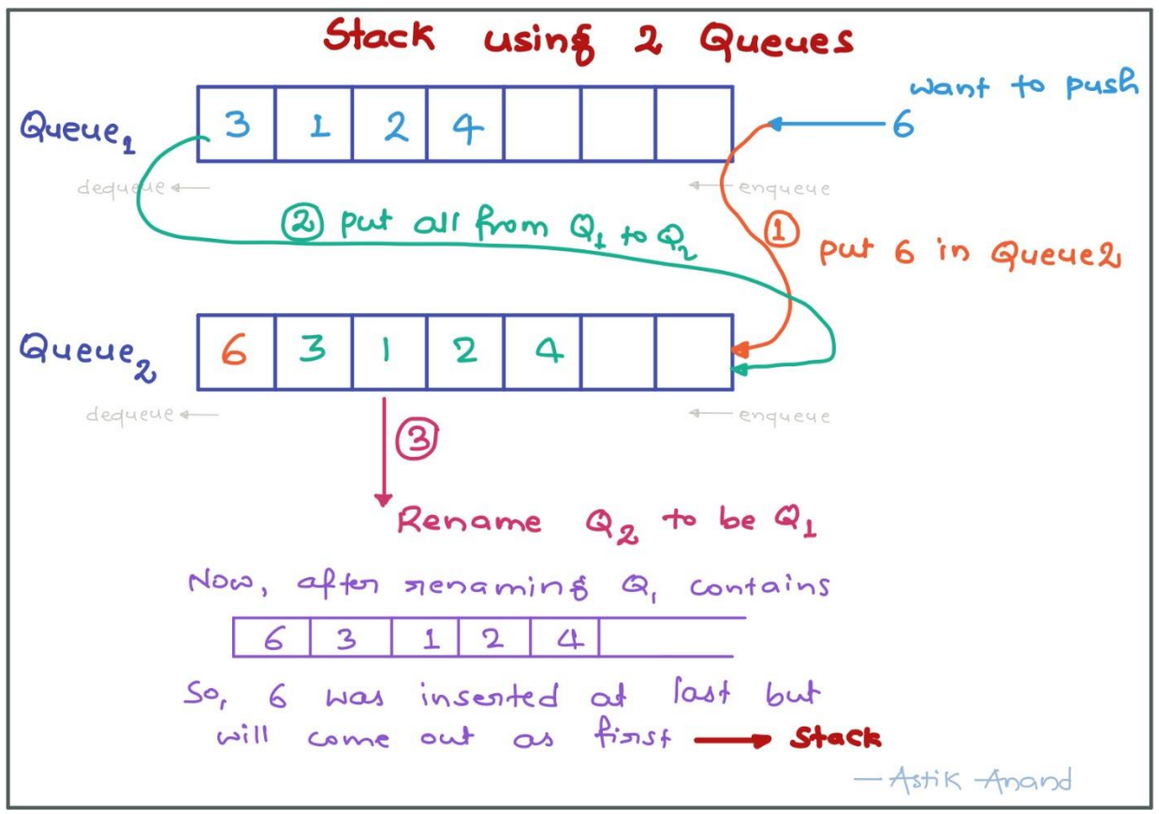
Problem:

Given a **queue** data structure that supports standard operations like **enqueue()** and **dequeue()**.

The task is to **implement a stack** data structure using only **instances of queue** and **queue operations** allowed on the instances.

Approach:

- A stack can be implemented using 2 queues.
- Let queue to be implemented be q and stacks used to implement q be stack1 and stack2, then q can be implemented in two ways:
    i. By making **push()** operation costly.
    ii. By making **pop()** operation costly.



Algorithm with costly Push:

- **push(s, x):** time complexity will be O(n)
  - Enqueue x to q2
  - One by one dequeue everything from q1 and enqueue to q2.
  - Swap the names of q1 and q2
  - // Swapping of names is done to avoid one more movement of all elements from q2 to q1.
- **pop():** time will be O(1)
  - Dequeue an item from q1 and return it.

Implementation:

```
from queue import Queue

class Stack:
    def __init__(self):
        self.queue1 = Queue()
        self.queue2 = Queue()
```

```
def push(self, data):
    # (1) Put the data first to queue2
    self.queue2.put(data)

    # (2) Put all the data from queue1 to queue2
    while(not self.queue1.empty()):
        self.queue2.put(self.queue1.get())

    # (3) Swap the names of two queues
    self.queue1, self.queue2 = self.queue2, self.queue1


def pop(self):
    if(self.queue1.empty()):
        print("Stack Underflow")
        return
    else:
        return self.queue1.get()


def top(self):
    if(self.queue1.empty()):
        print("Stack Underflow")
        return
    else:
        return self.queue1.queue[0]

    def is_empty(self):
        return self.queue1.empty()



print("Example:- Stack Using Queues:")
my_stack = Stack()
print("Pushed: 4, 2, 1, 3 to stack.")
my_stack.push(4)
my_stack.push(2)
my_stack.push(1)
my_stack.push(3)
my_stack.push(6)
print("6 is pushed to the stack.\n")

print("Pop the first element: {}".format(my_stack.pop()))
print("Top element in stack: {}".format(my_stack.top()))
print("Is stack empty? {}".format(my_stack.is_empty()))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 16:08:34 ~/Personal/Notebooks/Data Structures/3. Queues $ python3 4_stack_using_queues.py
Example:- Stack Using Queues:
Pushed: 4, 2, 1, 3 to stack.
6 is pushed to the stack.

Pop the first element: 6
Top element in stack: 3
Is stack empty? False
```

3. Find first circular tour that visits all petrol pumps***

Problem:

- Suppose there is a circle. There are n petrol pumps on that circle.
- We are given two sets of data.
    i. The amount of petrol that every petrol pump has.
    ii. Distance from that petrol pump to the next petrol pump.
- Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity).
- Assume for 1 litre petrol, the truck can go 1 unit of distance.
- Expected **time complexity** is **O(n)** and can use O(n) extra space.

Example:
Input: {4, 6}, {6, 5}, {7, 3} and {4, 5}. // 4 petrol pumps with amount of petrol and distance to next petrol pump
Output: 2 // The first point from where truck can make a circular tour is 2nd petrol pump.

Approach-1:

- A Simple Solution is to consider every petrol pumps as starting point and see if there is a possible tour.
- If we find a starting point with feasible solution, we return that starting point.
- **Time complexity: O(n^2)**.

Approach-2:

- We can use a Queue to store the current tour.
- We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative.
- If the amount becomes negative, then we keep dequeueing petrol pumps till the current amount becomes positive or queue becomes empty.
- **Time complexity: O(n)** and **Auxiliary Space: O(n)**
- To avoid O(n) space, instead of creating a separate queue, we use the given array itself as queue.
- We maintain two index variables start and end that represent rear and front of queue.
- **Time complexity: O(n)** and **Auxiliary Space: O(1)**

Implementation:

```
def first_circular_tour_visiting_all_pumps(pumps):
    n = len(pumps)
    # Consider first petrol pump as starting point
    start = 0
    end = 1

    curr_petrol = pumps[start][0] - pumps[start][1]

    # Run a loop while all petrol pumps are not visited
    while(end != start):
        # While curr_petrol is < 0, dequeue till it become positive.
        while(curr_petrol<0 and start!=end):
            # Remove starting petrol pump. Change start
            curr_petrol -= pumps[start][0] - pumps[start][1]
            start = (start +1)%n

            # If 0 is being considered as start again, then there is no possible solution
            if start == 0:
                return -1

        # Add a petrol pump to current tour
        curr_petrol += pumps[end][0] - pumps[end][1]
        end = (end +1) % n

    return start+1


print("Example-1: first_circular_tour_visiting_all_pumps()")
pumps = [(4, 6), (6, 5), (7, 3), (4, 5)]
print(first_circular_tour_visiting_all_pumps(pumps))

print("Example-2: first_circular_tour_visiting_all_pumps()")
pumps = [(4, 6), (6, 5), (3, 5), (10, 5)]
print(first_circular_tour_visiting_all_pumps(pumps))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 10:03:31 ~/Personal/Notebooks/Data Structures/3. Queues $ python3 5_first_circular_tour_visits_all_petrol_pumps.py
Example-1: first_circular_tour_visiting_all_pumps()
2
Example-2: first_circular_tour_visiting_all_pumps()
4
```

Complexity:

- **Time: O(n) :** Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the time complexity is O(n).
- **Auxilliary Space: O(1)**

4. Maximum of all subarrays of size k (Sliding Window Maximum)

Problem:

Given an array an integer k, find the maximum for each and every contiguous subarrays of size k.

Examples:

Input: 1, 2, 3, 1, 4, 5, 2, 3, 6 and k=3

Output: 3, 3, 4, 5, 5, 5, 6

Input: 8, 5, 10, 7, 9, 4, 15, 12, 90, 13 and k=4

Output: 10, 10, 10, 15, 15, 90, 90

Approach-1:

- Run two loops. In the outer loop take all subarrays of size k.
- In the inner loop, get the maximum of the current subarray.
- **Time Complexity: O(nk)**

Approach-2:

- First get the initial current_window of of first k elements and the first element of max_subarray is max of initial window.
- Now start from k+1th element, and in current_window:

- dequeue the first element from current_window and enqueue the k+1th element
- get max of current_window and put in max_subarray and continue till the last element.

Implementation

```
def max_of_all_subarrays_of_k_size(arr, k):
    n = len(arr)
    max_subarray = []

    # Initially current window
    current_window = arr[:k]

    # First get the initial current_window of of first k elements,
    # the first element of max_subarray is max of initial window.
    max_subarray.append(max(current_window))

    # Now start from k+1th element, and in current_window
    # dequeue the first element from current_window and enqueue the k+1th element
    # get max of current_window and put in max_subarray and continue till the last element.
    for i in range(k, n):
        # Dequeue the first element and enqueue the next element
        current_window.pop(0)
        current_window.append(arr[i])

        max_subarray.append(max(current_window))

    return max_subarray



print("Example-1: max_of_all_subarrays_of_k_size")
arr = [1, 2, 3, 1, 4, 5, 2, 3, 6]
print(max_of_all_subarrays_of_k_size(arr, 3))

print("\nExample-2: max_of_all_subarrays_of_k_size")
arr = [8,  5,  10, 7,  9,  4,  15, 12, 90, 13]
print(max_of_all_subarrays_of_k_size(arr, 4))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 23:11:53 ~/Personal/Notebooks/Data Structures/3. Queues $ python3 6_max_of_all_subarrays_size_k_sliding_window.py
Example-1: max_of_all_subarrays_of_k_size
[3, 3, 4, 5, 5, 5, 6]

Example-2: max_of_all_subarrays_of_k_size
[10, 10, 10, 15, 15, 90, 90]
```

Complexity:

- **Time: O(nlogk) :** Can do it in O(n) by avoiding finding max in current window by storing useful values in deque which stores max at top.
- **Auxilliary Space: O(1)**

5. Generate Binary Numbers from 1 to N (An Interesting Method)

Problem:

Given a number n, write a function that generates and prints all binary numbers with decimal values from 1 to n.

Examples:

Input: n = 2

Output: 1, 10

Input: n = 5

Output: 1, 10, 11, 100, 101

Approach:

- We can use queue data structure to achieve this.
- Start with 1, append 0 and enqueue and append 1 and enqueue.
- Dequeue one element and repeat the procedure.

Algorithm:

- Create an empty queue of strings
- Enqueue the first binary number “1” to queue.
- Now run a loop for generating and printing n binary numbers.
    - Dequeue and Print the front of queue.
    - Append “0” at the end of front item and enqueue it.
    - Append “1” at the end of front item and enqueue it.

Implementation:

```
def generate_1_to_n_binary_using_queue(n):
    binary = []
    queue = ["1"]

    while(n > 0):
        # Dequeue from the queue, add current to the list of binary numbers, then
        # append "0" to it and enqueue and again append "1" to it and enqueue to queue
```

```
        current = queue.pop(0)
        binary.append(current)

        queue.append(current+"0")
        queue.append(current+"1")

        n -= 1

    return binary



print("Example-1: generate_1_to_n_binary_using_queue(5)")
print(generate_1_to_n_binary_using_queue(5))

print("\nExample-2: generate_1_to_n_binary_using_queue(10)")
print(generate_1_to_n_binary_using_queue(10))
```

```
astik.anand@mac-C02XD95ZJG5H 00:23:30 ~/Personal/Notebooks/Data Structures/3. Queues $ python3 7_generate_1_to_n_binary_using_queue.py
Example-1: generate_1_to_n_binary_using_queue(5)
['1', '10', '11', '100', '101']

Example-2: generate_1_to_n_binary_using_queue(10)
['1', '10', '11', '100', '101', '110', '111', '1000', '1001', '1010']
```

Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

Matrix

What is Matrix ?

- It is a way to store data in an organized form in the form of rows and columns.
- A two-dimensional array can function exactly like a matrix and can be visualized as a table consisting of rows and columns.

Example: `matrix[3][4]`

	Column 0	Column 1	Column 2	Column 3
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Applications of Matrix

- Some AI programs use matrices and vectors.
- Neural Networks rely heavily on matrices and matrix operations.
- Some localization algorithms also use matrix operations.
- Matrices, and in general Linear Algebra, are the language and mathematical tools computer scientists constantly use.

Standard Matrix Problems

1. Search in a Row-wise & Column-wise sorted Matrix***

Problem:

Given an n x n matrix and a number x, find the position of x in the matrix if it is present in it. Otherwise, print “Not Found”.

In the given matrix, every row and column is sorted in increasing order.

The algorithm should have linear time complexity.

Examples:

```
Input: mat[4][4] = {{10, 20, 30, 40},
                    {15, 25, 35, 45},     x = 29
                    {27, 29, 37, 48},
                    {32, 33, 39, 50}}

Output: Found at (2, 1)


Input: mat[4][4] = {{10, 20, 30, 40},
                    {15, 25, 35, 45},     x = 100
                    {27, 29, 37, 48},
                    {32, 33, 39, 50}}

Output: Element not found
```

Approach-1: Brute-Force

- A simple solution is to search one by one.
- **Time complexity: $O(n^2)$**

Approach-2: Efficient

- **Start the search from the top-right corner of the array.**
- **There are 3 possible cases:**
  i. The number we are searching for is greater than the current number. This will ensure, that all the elements in the current row is smaller than the number we are searching for as we are already at the right-most element and the row is sorted. Thus, the entire row gets eliminated and we continue our search on the next row. Here elimination means we won't search on that row again.
  ii. The number we are searching for is smaller than the current number. This will ensure, that all the elements in the current column is greater than the number we are searching for. Thus, the entire column gets eliminated and we continue our search on the previous column i.e. the column at the immediate left.
  iii. The number we are searching for is equal to the current number. This will end our search.

Implementation

```python
def search_row_column_wise_sorted_matrix(matrix, key):
    n = len(matrix)

    # Start with top right corner
    i = 0; j = n - 1

    while i < n and j >= 0:
```

```
        # If righmost element is lesser than key, skip that row
        if matrix[i][j] < key:
            i += 1
        # Else if righmost element is greater than key, skip that column
        elif matrix[i][j] > key:
            j -= 1
        # Else if righmost element is equal to key, the key is found.
        else:
            print("Key {}: Found at ({}, {})".format(key, i, j))
            return

    print("Key {}: Not Found".format(key))



mat = [ [10, 20, 30, 40],
        [15, 25, 35, 45],
        [27, 29, 37, 48],
        [32, 33, 39, 50] ]

print("Example-1: search_row_column_wise_sorted_matrix(matrix, key)")
search_row_column_wise_sorted_matrix(mat, 29)

print("\nExample-2: search_row_column_wise_sorted_matrix(matrix, key)")
search_row_column_wise_sorted_matrix(mat, 38)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 22:57:39 ~/Personal/Notebooks/Data Structures/5. Matrix $ python3 1_search_row_column_wise_sorted_matrix.py
Example-1: search_row_column_wise_sorted_matrix(matrix, key)
Key 29: Found at (2, 1)

Example-2: search_row_column_wise_sorted_matrix(matrix, key)
Key 38: Not Found
```

Complexity:

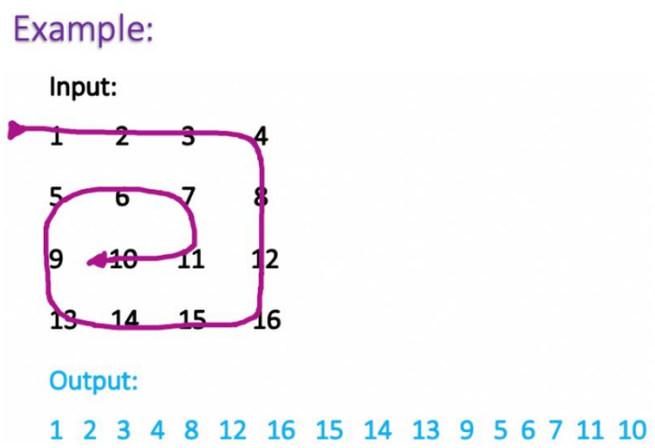
- **Time: O(N)**
- **Auxilliary Space: O(1)**

2. Print Matrix in spiral form***

Problem

Print a matrix in spiral form.

Example:



Approach: Use 4 for loops to print all 4 directions.

- Can be solved using four for loops which prints all the elements.
- Every for loop defines a single direction movement along with the matrix.
- **1st for loop** represents the movement from **left to right.**
- **2nd for loop** represents the movement from **top to bottom**.
- **3rd for loop** represents the movement from the **right to left**.
- **4th for loop** represents the movement from **bottom to up**.

Implementation

```
def matrix_spiral_print(matrix):
    ## Some Important Variables
    #   • row_start - starting row index
    #   • row_end - ending row index
    #   • col_start - starting column index
    #   • col_end - ending column index
    #   • i - iterator
    row_start = 0; row_end = len(matrix)
    col_start = 0; col_end = len(matrix[0])

    while (row_start<row_end and col_start<col_end) :
        # Print the first row, after printing done increment the row_start
        for i in range(col_start, col_end) :
            print(matrix[row_start][i], end = " ")

        row_start += 1
```

```
        # Print the last column, after printing done decrement the col_end
        for i in range(row_start, row_end) :
            print(matrix[i][col_end-1], end = " ")

        col_end -= 1

        # Print the last row, after printing done decrement the row_end
        if row_start < row_end:
            for i in range(col_end - 1, (col_start - 1), -1) :
                print(matrix[row_end - 1][i], end = " ")

            row_end -= 1

        # Print the first column, after printing done increment the col_start
        if (col_start < col_end) :
            for i in range(row_end - 1, row_start - 1, -1) :
                print(matrix[i][col_start], end = " ")

            col_start += 1

    print()



print("Example-1: matrix_spiral_print(matrix)")
mat = [ [ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12],
        [13, 14, 15, 16] ]
matrix_spiral_print(mat)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:47:16 ~/Personal/Notebooks/Data Structures/5. Matrix $ python3 2_matrix_spiral_print.py
Example-1: matrix_spiral_print(matrix)
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

Complexity:

- **Time: O(MN)**
- **Auxilliary Space: O(1)**

3. Flood Fill Algorithm - Implement fill() in paint***

Problem:

In MS-Paint, when we take the brush to a pixel and click, the color of the region of that pixel is replaced with a new selected color. Following is the problem statement to do this task. Given a 2D screen, location of a pixel in the screen and a color, replace color of the given pixel and all adjacent same colored pixels with the given color.

Example:

Input:

```
screen[M][N] = { {1, 1, 1, 1, 1, 1, 1, 1},
                 {1, 1, 1, 1, 1, 1, 0, 0},
                 {1, 0, 0, 1, 1, 0, 1, 1},
                 {1, 2, 2, 2, 2, 0, 1, 0},        x = 4, y = 4, new_color = 3
                 {1, 1, 1, 2, 2, 0, 1, 0},
                 {1, 1, 1, 2, 2, 2, 2, 0},
                 {1, 1, 1, 1, 1, 2, 1, 1},
                 {1, 1, 1, 1, 1, 2, 2, 1} }
```

The values in the given 2D screen indicate colors of the pixels. x and y are coordinates of the brush, new_color is the color that should replace the previous color on screen[x][y] and all surrounding pixels with same color.

Output:

Screen should be changed to following.

```
screen[M][N] = { {1, 1, 1, 1, 1, 1, 1, 1},
                 {1, 1, 1, 1, 1, 1, 0, 0},
                 {1, 0, 0, 1, 1, 0, 1, 1},
                 {1, 3, 3, 3, 3, 0, 1, 0},
                 {1, 1, 1, 3, 3, 0, 1, 0},
                 {1, 1, 1, 3, 3, 3, 3, 0},
                 {1, 1, 1, 1, 1, 3, 1, 1},
                 {1, 1, 1, 1, 1, 3, 3, 1} }
```

Approach - Simple:

- The idea is simple, we first replace the color of current pixel, then recur for 4 surrounding points.

Algorithm:

```
flood_fill(screen[M][N], x, y, prev_color, new_color)
```

1. If x or y is outside the screen, then return.
2. If color of `screen[x][y]` is not same as prev_color, then return
3. Replace the color at (x, y)
4. Recur for north, south, east and west
    - flood_fill(screen, x+1, y, prev_color, new_color)
    - flood_fill(screen, x-1, y, prev_color, new_color)
    - flood_fill(screen, x, y+1, prev_color, new_color)
    - flood_fill(screen, x, y-1, prev_color, new_color)

Implementation:

```
def flood_fill(screen, x, y, prev_color, new_color):
    m = len(screen)
    n = len(screen[0])

    # If x or y is outside the screen, then return.
    if(x < 0 or x >= m or y < 0 or y >= n):
        return

    # If color of screen[x][y] is not same as prev_color, then return
    if(screen[x][y] != prev_color):
        return

    # Replace the color at (x, y)
    screen[x][y] = new_color

    # Recur for north, south, east and west
    flood_fill(screen, x, y+1, prev_color, new_color)
    flood_fill(screen, x, y-1, prev_color, new_color)
    flood_fill(screen, x+1, y, prev_color, new_color)
    flood_fill(screen, x-1, y, prev_color, new_color)



print("Example-1: Flood Fill Algorithm")
screen = [ [1, 1, 1, 1, 1, 1, 1, 1],
           [1, 1, 1, 1, 1, 1, 0, 0],
           [1, 0, 0, 1, 1, 0, 1, 1],
           [1, 2, 2, 2, 2, 0, 1, 0],
           [1, 1, 1, 2, 2, 0, 1, 0],
           [1, 1, 1, 2, 2, 2, 2, 0],
           [1, 1, 1, 1, 1, 2, 1, 1],
           [1, 1, 1, 1, 1, 2, 2, 1] ]



flood_fill(screen, 4, 4, 2, 3);
m = len(screen)
n = len(screen[0])

for i in range(m):
    for j in range(n):
```

```
        print(screen[i][j], end=" ")
    print()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:15:04 ~/Personal/Notebooks/Algorithms/13. Miscellaneous $ python3 2_flood_fill.py
Example-1: Flood Fill Algorithm
1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0
1 0 0 1 1 0 1 1
1 3 3 3 3 0 1 0
1 1 1 3 3 0 1 0
1 1 1 3 3 3 3 0
1 1 1 1 1 3 1 1
1 1 1 1 1 3 3 1
```

Complexity:

- **Time: $O(n^2)$**
- **Auxilliary Space: O(1)**

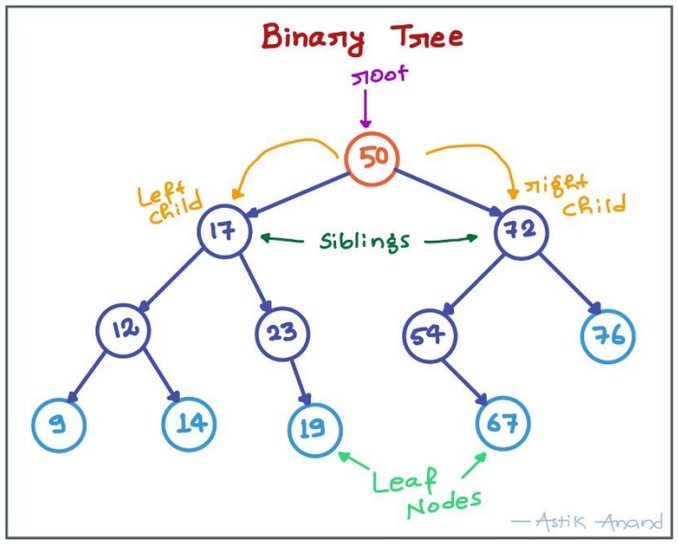
Problems To Do:

- Print elements in sorted order in row and column wise sorted matrix
- Maximum size of square sub-matrix with sum <= K

Binary Tree

What are Trees ?

- Array, Linked List, Stack & Queues are Linear Data Structure
- But Trees are Hierarchical data structure.

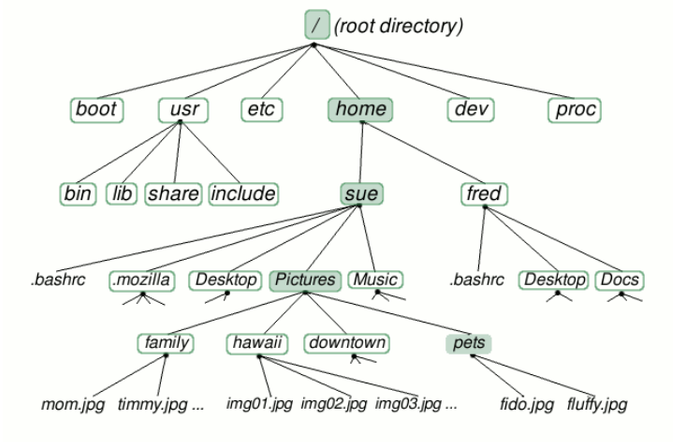


Tree Vocabulary:

- **node:** any element of a tree
- **root:** Topmost node
- **children:** nodes directly under a node
- **parent:** node directly above the node or nodes
- **leaves:** nodes with no children

Why trees?

- To store information that naturally forms a hierarchy. Ex: **Linux filesystem**



- Trees (BST) provide moderate access/search (quicker than Linked List and slower than arrays).
- Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
- Self-balancing search trees like **AVL** and **Red-Black** trees guarantee an upper bound of **O(Logn)** for insertion/deletion.
- Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Properties of Binary Tree

- Max no. of nodes at level L: $\mathbf{2^{L-1}}$
- Max no. of nodes in a binary tree of height "h": $\mathbf{2^{h} - 1}$

Types of Binary Trees

- **Full Binary Tree**:
  - Every node has 0 or 2 children
  - Leaf nodes = Internal nodes + 1
- **Complete Binary Tree:**
  - All levels are completely filled except possibly the last level and the last level has all keys as left as possible.
- **Perfect Binary Tree:**
  - All internal nodes have 2 children and all leaves are at same level.
- **Balanced Binary Tree:**
  - Height of the tree: **O(Log n)**
- **Degenerate Tree:**
  - Every internal node has 1 child.

Representation

- A tree whose elements have at most 2 children is called a binary tree.
- Typically named as left and right child.

Implementation:

```
class Node:
    def __init__(self, val):
        self.left = None
        self.right = None
        self.val = val


def print_inorder(root):
    if root:
        # First recur on left child
        print_inorder(root.left)
        # then print the data of node
        print(root.val, end=" ")
        # now recur on right child
        print_inorder(root.right)



print("Example:- Binary Tree")
# Root
root = Node(50)

# 1st Level
root.left      = Node(17)
root.right     = Node(72)

# 2nd Level
root.left.left   = Node(12)
root.left.right  = Node(23)
root.right.left  = Node(54)
root.right.right = Node(76)

# 3rd Level
root.left.left.left   = Node(9)
root.left.left.right  = Node(14)
root.left.right.right = Node(19)
root.right.left.right = Node(67)

# Print the tree in inorder
print_inorder(root)
print()
```

Output:

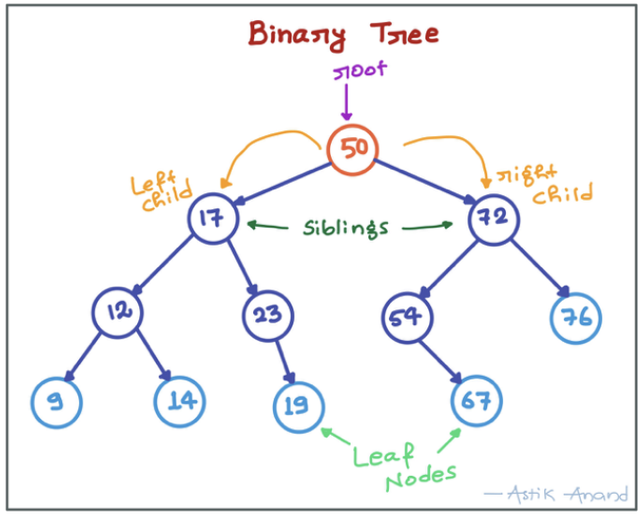
```
astik.anand@mac-C02XD95ZJG5H 01:04:57 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 0_binary_tree_representation.py
Example:- Binary Tree
9 12 14 17 23 19 50 54 67 72 76
```

Applications of Binary Trees

- Manipulate hierarchical data.
- Make information easy to search (Tree traversal).
- Manipulate sorted lists of data(BST).
- As a workflow for compositing digital images for visual effects.
- Router algorithms.
- Form of a multi-stage decision-making (see business chess).

Algo-1: Tree DFS Traversals - (Inorder, Preorder, Postorder)

- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.
- We have 3 options to traverse tree in DFS manner (Inorder, Preorder, Postorder).



Inorder Traversal

Algorithm:

1. Traverse the left subtree i.e., call inorder(left-subtree).
2. Visit the root.
3. Traverse the right subtree i.e. call inorder(right-subtree).

9 12 14 17 23 19 50 54 67 72 76

Uses:

- It **gives nodes in non-decreasing order**.
- To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal's reversed can be used.

Preorder Traversal

Algorithm:

- Visit the root.
- Traverse the left subtree i.e., call preorder(left-subtree).
- Traverse the right subtree i.e. call preorder(right-subtree).

50 17 12 9 14 23 19 72 54 67 76

Uses:

- Preorder traversal is used to **create a copy** of the tree.
- Preorder traversal is also used to get **prefix expression on of an expression tree**.

Postorder Traversal

Algorithm:

- Traverse the left subtree i.e., call postorder(left-subtree).
- Traverse the right subtree i.e. call postorder(right-subtree).
- Visit the root.

9 14 12 19 23 17 67 54 76 72 50

Uses:

- Postorder traversal is used to **delete the tree**.
- Postorder traversal is also useful to get the **postfix expression of an expression tree.**

Implementation

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


def inorder(root):
    if root:
```

```
        inorder(root.left)
        print(root.val, end=" ")
        inorder(root.right)


def preorder(root):
    if root:
        print(root.val, end=" ")
        preorder(root.left)
        preorder(root.right)


def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.val, end=" ")



# Root
root = Node(50)

# 1st Level
root.left      = Node(17)
root.right     = Node(72)

# 2nd Level
root.left.left   = Node(12)
root.left.right  = Node(23)
root.right.left  = Node(54)
root.right.right = Node(76)

# 3rd Level
root.left.left.left   = Node(9)
root.left.left.right  = Node(14)
root.left.right.right = Node(19)
root.right.left.right = Node(67)

# Print the tree traversals
print("Inorder Traversal:")
inorder(root)
print("\n\nPreorder Traversal:")
preorder(root)
print("\n\nPostorder Traversal:")
postorder(root)
print()
```

Output:

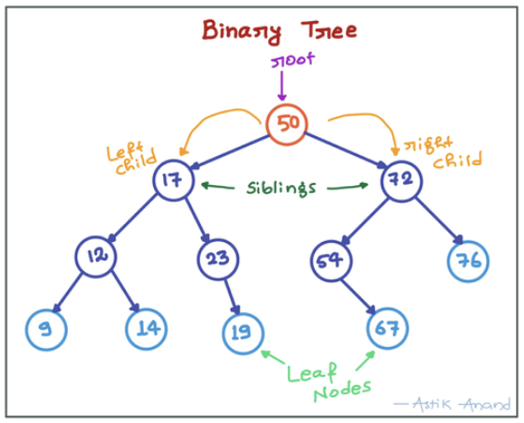
```
astik.anand@mac-C02XD95ZJG5H 23:18:45 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 1_tree_traversals.py
Inorder Traversal:
9 12 14 17 23 19 50 54 67 72 76

Preorder Traversal:
50 17 12 9 14 23 19 72 54 67 76

Postorder Traversal:
9 14 12 19 23 17 67 54 76 72 50
```

Complexity:

- **Time: O(n)** :- worst for skewed tree
- **Auxilliary Space:** If we don't consider size of stack for function calls then O(1) otherwise O(n)

Algo-2: Tree BFS Traversal - Level Order

Level order traversal of a tree is breadth first traversal for the tree.



50 17 72 12 23 54 76 9 14 19 67

Algorithm:

1. If root is None, then return.
2. Create an empty queue.
3. Enqueue the root to queue.
4. While queue is not empty:
    - Deque the temp_node from queue.
    - Print the temp_node.val.
    - If temp_node's left exists: enqueue left to queue
    - And if right exists: enqueue right also to queue

Implementation:

```
class Node:
```

```
    def __init__(self, val):
        self.left = None
        self.right = None
        self.val = val


def level_order_traversal(root):
    # (1) If root is None, then return.
    if root is None:
        return

    # (2) Create an empty queue.
    queue = []

    # (3) Enqueue root to queue.
    queue.append(root)

    # (4) while queue is not empty.
    while(queue):
        # Dequeue the temp_node from queue.
        temp_node = queue.pop(0)

        # Print temp_node.val
        print(temp_node.val, end=" ")

        # If temp_node's left exists: enqueue left to queue.
        if (temp_node.left):
            queue.append(temp_node.left)
        # And if right exists: enqueue right also to queue.
        if (temp_node.right):
            queue.append(temp_node.right)



# Root
root = Node(50)

# 1st Level
root.left       = Node(17)
root.right      = Node(72)

# 2nd Level
root.left.left   = Node(12)
root.left.right  = Node(23)
root.right.left  = Node(54)
root.right.right = Node(76)

# 3rd Level
root.left.left.left   = Node(9)
root.left.left.right  = Node(14)
root.left.right.right = Node(19)
root.right.left.right = Node(67)

print("Level Order Traversal:")
level_order_traversal(root)
print()
```

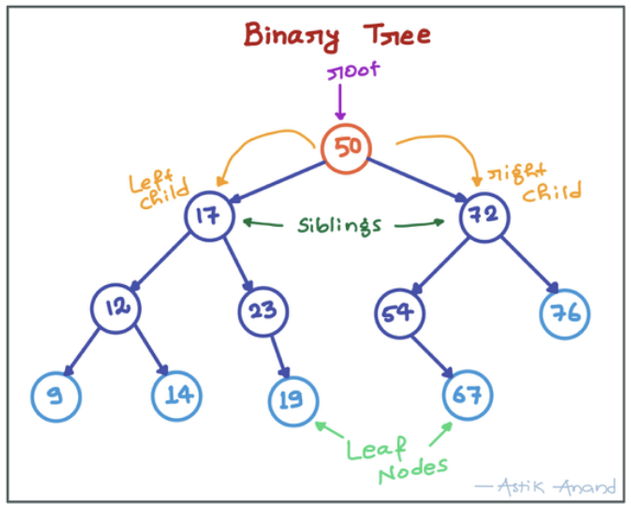
Output:

```
astik.anand@mac-C02XD95ZJG5H 02:07:25 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 2_level_order_traversal.py
Level Order Traversal:
50 17 72 12 23 54 76 9 14 19 67
```

Complexity:

- **Time : O(n)** :- where n is number of nodes in the binary tree
- **Auxilliary Space: O(w)** where w is maximum width of Binary Tree

BFS vs DFS in Binary Tree



Breadth First Traversals

- **Level Order Traversal:** 50 17 72 12 23 54 76 9 14 19 67

Depth First Traversals

- **Inorder Traversal:** 9 12 14 17 23 19 50 54 67 72 76
- **Preorder Traversal:** 50 17 12 9 14 23 19 72 54 67 76
- **Postorder Traversal:** 9 14 12 19 23 17 67 54 76 72 50

Why do we care?

There are many tree questions that can be solved using any of the above four traversals.

- **Using DFS Traversal:**
  - Finding Size of Tree.
  - Finding Height of Tree.
  - Finding Max or Min element in a Tree.

    - Diameter of Binary Tree.
    - Print nodes at K distance.
    - Checking if a binary tree is subtree of another binary tree.
    - Ancestors of a given node.
    -
- **Using BFS Traversal:**
    - Maximum Width of Binary Tree.
    - Left View of Tree.
    - Connect Nodes at same level.

Implementation:

```
import sys
INT_MIN = -sys.maxsize-1

class Node:
    def __init__(self, val):
        self.left = None
        self.right = None
        self.val = val


# Computes the number of nodes in tree
def size(root):
    if root is None:
        return 0
    else:
        return (1 + size(root.left) + size(root.right))


# Computes the height of tree
def height(root):
    if root is None:
        return 0
    else :
        return (1 + max(height(root.left), height(root.right)))


# Computes the number of nodes in tree
def maximum(root):
    if root is None:
        return INT_MIN
    else:
        return max(root.val, maximum(root.left), maximum(root.right))


# Root
root = Node(50)

# 1st Level
root.left      = Node(17)
```

```
root.right     = Node(72)

# 2nd Level
root.left.left   = Node(12)
root.left.right  = Node(23)
root.right.left  = Node(54)
root.right.right = Node(76)

# 3rd Level
root.left.left.left   = Node(9)
root.left.left.right  = Node(14)
root.left.right.right = Node(19)
root.right.left.right = Node(67)

print("Size of Tree: {}".format(size(root)))
print("Height of Tree: {}".format(height(root)))
print("Max of Tree: {}".format(maximum(root)))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:42:47 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 3_bfs_dfs_of_tree.py
Size of Tree: 11
Height of Tree: 4
Max of Tree: 76
```

Complexity:

- **Time:** All four traversals require **O(n) time** as they visit every node exactly once.
- **Space:**
  - Extra Space required for Level Order Traversal is O(w) where w is maximum width of Binary Tree.
  - In level order traversal, queue one by one stores nodes of different level.
  - Extra Space required for Depth First Traversals is O(h) where h is maximum height of Binary Tree.
  - In Depth First Traversals, stack (or function call stack) stores all ancestors of a node.

Important Points

- Maximum Width of a Binary Tree at depth (or height) h can be 2h where h starts from 0. So the maximum number of nodes can be at the last level.
- Worst case occurs when Binary Tree is a perfect Binary Tree with numbers of nodes like 1, 3, 7, 15, …etc. In worst case, value of 2h is Ceil(n/2).
- Height for a Balanced Binary Tree is O(Log n). Worst case occurs for skewed tree and worst case height becomes O(n).
- So in worst case extra space required is O(n) for both. But worst cases occur for different types of trees.
- *It is evident from above points that extra space required for Level order traversal is likely to be more when tree is more balanced and extra space for Depth First Traversal is likely to be more when tree is less balanced.*

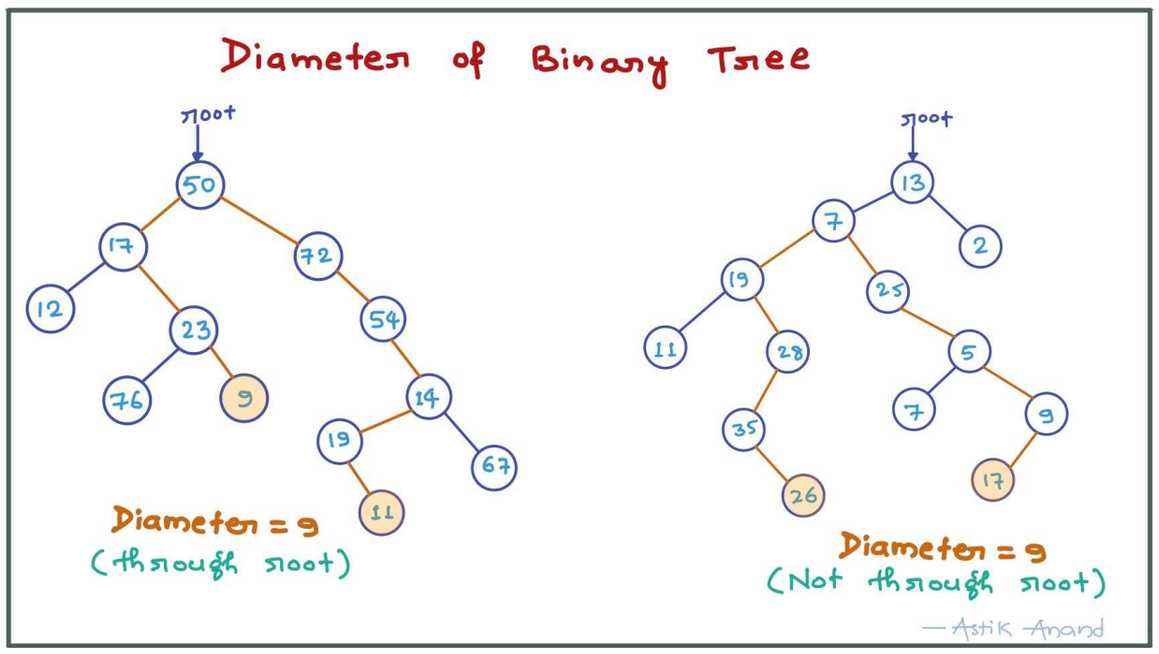
How to pick one?

- Extra Space can be one factor.
- Depth First Traversals are typically recursive and recursive code requires function call overheads.
- The most important points is, BFS starts visiting nodes from root while DFS starts visiting nodes from leaves.
- So if our problem is to search something that is more likely to closer to root, we would prefer BFS and if the target node is close to a leaf, we would prefer DFS.

Standard Binary Trees Problems

1. Diameter of Binary Tree***

- The diameter of a tree is the number of nodes on the longest path between two leaves in the tree.
- The diagram below shows two trees each with **diameter 9**.
- The leaves that form the ends of a longest path are colored (note that there may be more than one path in tree of same diameter).



Approach:

- Diameter can be calculated by using the height function.
- Because the diameter of a tree is nothing but maximum value of (left_height +right_height +1) **for each node**.
- So we need to calculate this value (left_height + right_height + 1) for each node as sometimes it may not contain root also (2nd case).
- Then we need to get the max of values present at each node.

Implementation:

```
import sys
INT_MIN = -sys.maxsize-1

class Node:
    def __init__(self, val):
        self.left = None
        self.right = None
        self.val = val


def height(root, max_diam):
    if root is None:
        return 0
    else :
        # Get the height of left and right nodes of the current_node
        left_height = height(root.left, max_diam)
        right_height = height(root.right, max_diam)

        # Get the diameter at current_node = 1 + left_height + right_height
        # Update the max_diam[0] if current_diam > max_diam[0]
        current_diam = 1 + left_height + right_height
        max_diam[0] = max(max_diam[0], current_diam)

        # return the height of the current_node
        return (1 + max(left_height, right_height))


def diameter(root):
    if root is None:
        return 0

    max_diam = [INT_MIN]    # This will store the final answer
    height(root, max_diam)
    return max_diam[0]


# Root
root = Node(13)

# 1st Level
root.left      = Node(7)
root.right     = Node(2)

# 2nd Level
```

```
root.left.left   = Node(19)
root.left.right  = Node(25)

# 3rd Level
root.left.left.left   = Node(11)
root.left.left.right  = Node(28)
root.left.right.right = Node(5)

# 4th Level
root.left.left.right.left = Node(35)
root.left.right.right.left = Node(7)
root.left.right.right.right = Node(9)

# 5th Level
root.left.left.right.left.right = Node(26)
root.left.right.right.right.left = Node(17)


print("Diameter of Tree: {}".format(diameter(root)))
```

Output:

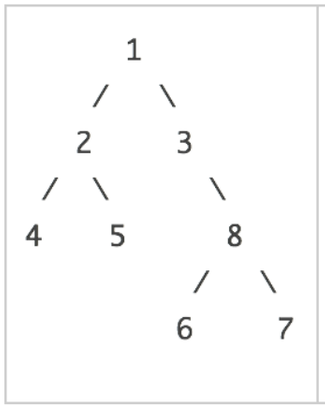
```
astik.anand@mac-C02XD95ZJG5H 01:42:30 ~/Personal/Notebooks/Data Structures/6. Binary Tree $ python3 4_diameter_binary_tree.py
Diameter of Tree: 9
```

Complexity:

- **Time: O(n)** DFS traversal
- **Auxilliary Space: O(h)** where h is maximum height of Binary Tree.

2. Maximum Width of Binary Tree***

Width of a tree is maximum of widths of all levels.



Max Width:

- width of level 0 is 1
- width of level 1 is 2
- width of level 2 is 3
- width of level 3 is 2

Maximum Width = 3

Approach:

- We will use level order traversal to solve this.
- In level order traversal, we use to dequeue one node and then enqueue it's left and right child.
- But here a slight modification is we will dequeue all the nodes present in queue instead of one and enqueue their left and right child.
- In fact this will also give the level order traversal only but with less recursion steps.

Algorithm:

- If root is None, then return 0.
- Create an empty queue and max_width=0.
- Enqueue root to queue.
- While queue is not empty.
  - Get the current_count and update max_width if it is greater.
  - Dequeue all the nodes instead of one.So, while current_count > 0:
    - Dequeue the temp_node from queue.
    - If temp_node's left exists: enqueue left to queue.
    - And if right exists: enqueue right also to queue.
    - Decrease the count by 1.

Implementation:

```
class Node:
    def __init__(self, val):
        self.left = None
        self.right = None
        self.val = val
```

```
def max_width_binary_tree(root):
    # (1) If root is None, then return 0.
    if root is None:
        return 0

    # (2) Create an empty queue and max_width=0.
    queue = []
    max_width = 0

    # (3) Enqueue root to queue.
    queue.append(root)

    # (4) While queue is not empty.
    while(queue):
        # Get the current_count and update max_width if it is greater.
        current_count = len(queue)
        max_width = max(max_width, current_count)

        # Dequeue all the nodes instead of one. So, while current_count > 0.
        while(current_count > 0):
            # Dequeue the temp_node from queue.
            temp_node = queue.pop(0)

            # If temp_node's left exists: enqueue left to queue.
            if (temp_node.left):
                queue.append(temp_node.left)

            # And if right exists: enqueue right also to queue.
            if (temp_node.right):
                queue.append(temp_node.right)

            current_count -= 1

    return max_width


# Root
root = Node(1)

# 1st Level
root.left      = Node(2)
root.right     = Node(3)

# 2nd Level
root.left.left   = Node(4)
root.left.right  = Node(5)
root.right.right = Node(8)

# 3rd Level
root.right.right.left  = Node(6)
root.right.right.right = Node(7)

print("Maximum Width = {}".format(max_width_binary_tree(root)))
```

Output:

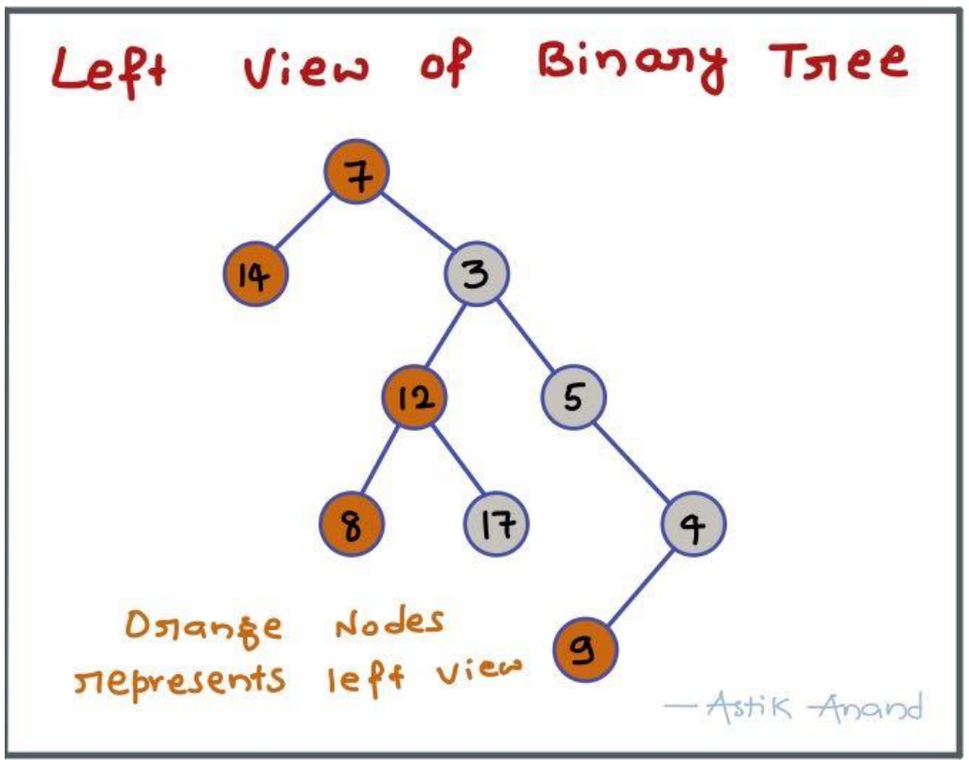
```
astik.anand@mac-C02XD95ZJG5H 12:12:17 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 5_max_width_binary_tree.py
Maximum Width = 3
```

Complexity:

- **Time : O(n)** :- BFS: Same as Level order traversal
- **Auxilliary Space: O(w)** where w is maximum width of Binary Tree

3. Left View of Tree***

Print all the tree elements when seen from left side.



Approach:

- We can solve using level order traversal.
- Just a **slight modification** of **maximum width problem.**
- In maximum width just put a printed flag to make sure we print only the first node of that level.

Implementation:

```
class Node:
    def __init__(self, val):
        self.left = None
        self.right = None
        self.val = val


def left_view_binary_tree(root):
    # (1) If root is None, then return.
    if root is None:
        return

    # (2) Create an empty queue.
    queue = []

    # (3) Enqueue root to queue.
    queue.append(root)

    # (4) While queue is not empty.
    while(queue):
        # Get the current_count.
        current_count = len(queue)

        # Flag to know if first element in this level is printed.
        printed = False

        # Dequeue all the nodes instead of one. So, while current_count > 0.
        while(current_count > 0):
            # Dequeue the temp_node from queue.
            temp_node = queue.pop(0)

            # Print the first element in this level if it is not printed.
            if(not printed):
                print(temp_node.val, end=" ")
                printed = True

            # If temp_node's left exists: enqueue left to queue.
            if (temp_node.left):
                queue.append(temp_node.left)

            # And if right exists: enqueue right also to queue.
            if (temp_node.right):
                queue.append(temp_node.right)

            current_count -= 1
```

```python
# Root
root = Node(4)

# 1st Level
root.left       = Node(5)
root.right      = Node(2)

# 2nd Level
root.right.left  = Node(3)
root.right.right = Node(1)

# 3rd Level
root.right.left.left  = Node(6)
root.right.left.right = Node(7)

print("Left View of Binary Tree: ")
left_view_binary_tree(root)
print()
```

Output:

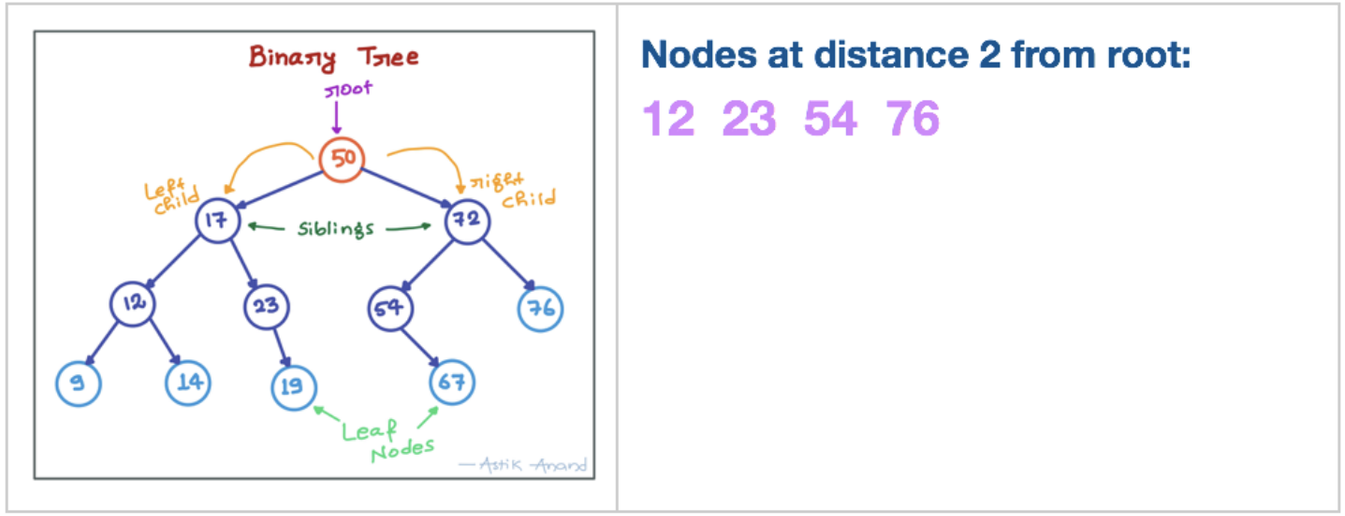
```
astik.anand@mac-C02XD95ZJG5H 01:42:37 ~/Personal/Notebooks/Data Structures/6. Binary Tree $ python3 6_left_view_of_tree.py
Left View of Binary Tree:
7 14 12 8 9
```

Complexity:

- **Time : O(n)** :- Same as maximum width problem
- **Auxilliary Space: O(w)** where w is maximum width of Binary Tree

4. Print Nodes at K distance from Root

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.



Nodes at distance 2 from root:

12 23 54 76

Algorithm:

- If distance is 0 : print the root.
- Else call the recursive function for root.left and root.left with distance-1.

Implementation:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


def print_nodes_at_k_distance(root, k):
    if root is None:
        return

    if k==0:
        print(root.val, end=" ")
    else:
        print_nodes_at_k_distance(root.left, k-1)
        print_nodes_at_k_distance(root.right, k-1)



# Root
root = Node(50)

# 1st Level
root.left      = Node(17)
root.right     = Node(72)
```

```python
# 2nd Level
root.left.left   = Node(12)
root.left.right  = Node(23)
root.right.left  = Node(54)
root.right.right = Node(76)

# 3rd Level
root.left.left.left   = Node(9)
root.left.left.right  = Node(14)
root.left.right.right = Node(19)
root.right.left.right = Node(67)

print("Nodes at distance of k=2:")
print_nodes_at_k_distance(root, 2)
print()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 21:54:13 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 7_print_nodes_at_k_distance.py
Nodes at distance of k=2:
12 23 54 76
```

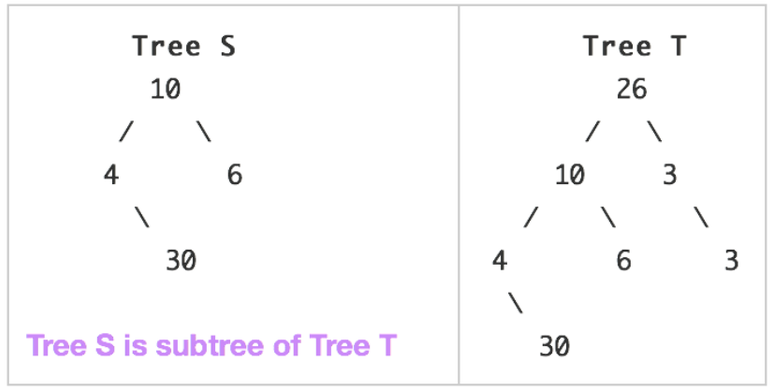
Complexity:

- **Time: O(n)** DFS traversal
- **Auxilliary Space: O(h)** where h is maximum height of Binary Tree.

5. Check if Binary Tree is Sub-tree of another Binary Tree***

Problem:

- Given two binary trees, check if the first tree is subtree of the second one.
- A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T.
- The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.



Approach:

- Write a **identical function** to check if 2 trees are identical.
- Traverse the tree T in preorder fashion.
- For every visited node in the traversal, check if the subtree rooted with this node is identical to S.

Algorithm:

- **are_identical(root1, root2):**
  - If both are NULL, then True.
  - If one is None and other not None, then False.
  - return True if the val of both roots is same and left subtree and right subtree are are_identical else False.
- **is_subtree(S, T):**
  - If S is None, always subtree.
  - If S is not None and T is none, not a subtree.
  - If both are identical return True.
  - Check if S is subtree of **T.left** or **T.right**.

Implementation:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


def are_identical(root1, root2):
    # If both are NULL, then True.
    if root1 is None and root2 is None:
```

```
        return True

    # If one is None and other not None, then False.
    if root1 is None or root2 is None:
        return False

    # Check if the val of both roots is same and left subtree and right subtree are are_identical.
    return (root1.val==root2.val and
            are_identical(root1.left, root2.left) and
            are_identical(root1.right, root2.right))


def is_subtree(S, T):
    # If S is None, always subtree.
    if S is None:
        return True

    # If S is not None and T is none, not a subtree.
    if T is None:
        return False

    # If both are identical return True.
    if are_identical(S, T):
        return True
    else:
        # Check if S is subtree of T.left or T.right
        return is_subtree(S, T.left) or is_subtree(S, T.right)


# Tree T
T = Node(26)
T.left   = Node(10)
T.right  = Node(3)
T.left.left   = Node(4)
T.left.right  = Node(6)
T.right.right = Node(3)
T.left.left.right = Node(30)

# Tree S
S = Node(10)
S.left   = Node(4)
S.right  = Node(6)
S.left.right = Node(30)
print("Example-1: Is Tree S subtree of Tree T ?: {}".format(is_subtree(S, T)))

# Tree S
S = Node(4)
S.right  = Node(30)
print("Example-2: Is Tree S subtree of Tree T ?: {}".format(is_subtree(S, T)))

# Tree S
S = Node(4)
S.right  = Node(15)
print("Example-3: Is Tree S subtree of Tree T ?: {}".format(is_subtree(S, T)))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:58:24 ~/Personal/Notebooks/Data Structures/6. Binary Tree $ python3 8_binary_tree_is_subtree_of_another.py
Example-1: Is Tree S subtree of Tree T ?: True
Example-2: Is Tree S subtree of Tree T ?: True
Example-3: Is Tree S subtree of Tree T ?: False
```

Complexity:

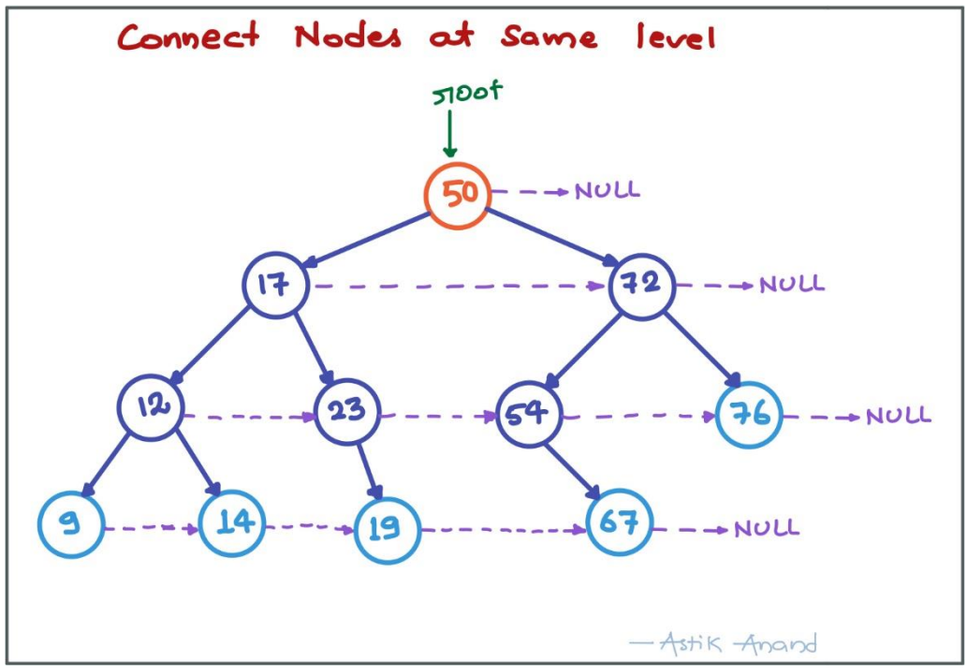
- **Time: O(n)** DFS traversal
- **Auxilliary Space: O(h)** where h is maximum height of Binary Tree.

6. Connect Nodes at Same Level***

Problem:

Write a function to connect all the adjacent nodes at the same level in a binary tree.

Initially, all the nextRight pointers point to garbage values, the function should set these pointers to point next right for each node.



Approach:

- Just do a level order traversal **similar** to **max_width** or **left_view** problem.
- While taking out node from queue just set the **nextright** as the node in front of queue.
- Set the next_right and consider that at every level the last element will not point to any other node as the front will be empty.

Implementation:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.nextright = None



def connect_nodes_at_same_level(root):
    # (1) If root is None, then return.
    if root is None:
        return

    # (2) Create an empty queue.
    queue = []

    # (3) Enqueue root to queue.
```

```
    queue.append(root)

    # (4) While queue is not empty.
    while(queue):
        # Get the current_count.
        current_count = len(queue)

        # Dequeue all the nodes instead of one. So, while current_count > 0.
        while(current_count > 0):
            # Dequeue the temp_node from queue.
            temp_node = queue.pop(0)

            print("{}--->".format(temp_node.val), end="")
            # Set the next_right and consider that at every level
            # the last element will not point to any other node as the front will be empty.
            if(current_count > 1):
                temp_node.nextright = queue[0]
            else:
                temp_node.nextright = None
                print("None")

            # If temp_node's left exists: enqueue left to queue.
            if (temp_node.left):
                queue.append(temp_node.left)

            # And if right exists: enqueue right also to queue.
            if (temp_node.right):
                queue.append(temp_node.right)
            current_count -= 1


# Root
root = Node(50)

# 1st Level
root.left      = Node(17)
root.right     = Node(72)

# 2nd Level
root.left.left   = Node(12)
root.left.right  = Node(23)
root.right.left  = Node(54)
root.right.right = Node(76)

# 3rd Level
root.left.left.left   = Node(9)
root.left.left.right  = Node(14)
root.left.right.right = Node(19)
root.right.left.right = Node(67)

# Call Function
connect_nodes_at_same_level(root)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 23:41:23 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 9_connect_nodes_at_same_level.py
50--->None
17--->72--->None
12--->23--->54--->76--->None
9--->14--->19--->67--->None
```

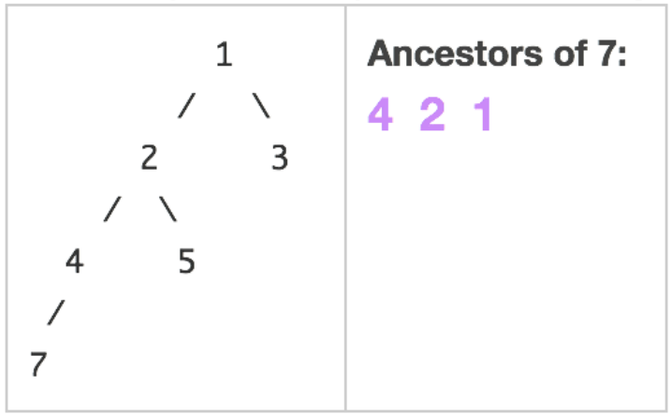
Complexity:

- **Time : O(n)** :- Same as **maximum_width** or **left_view** problem.
- **Auxilliary Space: O(w)** where w is maximum width of Binary Tree.

7. Ancestors of a Given Node in Binary Tree***

Problem:

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.



Algorithm:

- If root is None return False.
- If root.val is given_node then return True.
- Print the **root.data** if any of **root.left** or **root.right** contains the given_node and return True.

- If given_node not in tree, return False

Implementation:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


def print_ancestors(root, k):
    # If root is None return False.
    if root is None:
        return False

    # If root.val is given_node then return True.
    if root.val == k:
        return True

    # Print the root.data if any of root.left or root.right contains the given_node and return True.
    if(print_ancestors(root.left, k) or print_ancestors(root.right, k)):
        print(root.val, end=" ")
        return True

    # If given_node not in tree, return False
    return False


# Root
root = Node(1)
root.left      = Node(2)
root.right     = Node(3)
root.left.left   = Node(4)
root.left.right  = Node(5)
root.left.left.left    = Node(7)

print("Ancestors of given_node k=7:")
print_ancestors(root, 7)
print()

print("Ancestors of given_node k=3:")
print_ancestors(root, 3)
print()
```

```
astik.anand@mac-C02XD95ZJG5H 00:03:42 ~/Personal/Notebooks/Data Structures/4. Binary Tree $ python3 10_ancestor_of_a_node_binary_tree.py
Ancestors of given_node k=7:
4 2 1
Ancestors of given_node k=3:
1
```

Complexity:

- **Time: O(n)** DFS traversal
- **Auxilliary Space: O(h)** where h is maximum height of Binary Tree.

8. Check if Binary Tree is Balanced***

Problem:

Check if bianry tree is height balanced by considering a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

- An empty tree is always height-balanced.
- A non-empty binary tree T is balanced if:
    i. Left subtree of T is balanced
    ii. Right subtree of T is balanced
    iii. The difference between heights of left subtree and right subtree is not more than 1.
- This height-balancing scheme is used in AVL trees.

Approach-1: Brute-Force:

- Get the height of left subtree and right subtree and check if difference is 1.
- Also check if left subtree and right subtree is balanced.
- **Time Complexity: O(n^2):** Worst case occurs in case of skewed tree.

Implementation-Approach-1:

```
def height(root):
    if root is None:
        return 0
    return max(height(root.left), height(root.right)) + 1

# function to check if tree is height-balanced or not
def isBalanced(root):
    if root is None:
        return True

    # left and right subtree height
    lh = height(root.left)
```

```
    rh = height(root.right)

    # allowed values for (lh - rh) are 1, -1, 0
    if (abs(lh-rh) <= 1) and isBalanced(root.left) and isBalanced( root.right):
        return True

    # if we reach here means tree is not  height-balanced tree
    return False
```

Approach-2: Optimized:

- Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately.
- **Time complexity: O(n)**

Implementation-Approach-2:

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

class Height:
    def __init__(self):
        self.height = 0


def isBalanced(root, height):
    # lh and rh to store height of left and right subtree
    lh = Height()
    rh = Height()

    if root is None:
        return True

    # l and r are used to check if left and right subtree are balanced
    l = isBalanced(root.left, lh)
    r = isBalanced(root.right, rh)

    # Update height
    height.height = max(lh.height, rh.height) + 1

    if abs(lh.height-rh.height) <= 1 and l and r:
        return True

    # if we reach here then the tree is not balanced
    return False


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

```
root.left.right = Node(5)
root.right.left = Node(6)
root.left.left.left = Node(7)

if isBalanced(root, Height()):
    print('Tree is balanced')
else:
    print('Tree is not balanced')
```

11. Serialize and Deserialize Binary Tree

Serialization & Deserialization

It is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed (deserialized) later in the same or another computer environment.

Problem:

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Approach: Use DFS on Binary Tree

A simple solution is to store both Inorder and Preorder traversals. This solution requires requires space twice the size of Binary Tree. We can save space by storing Preorder traversal and a marker for NULL pointers.

Other Examples:

```
Let the marker for NULL pointers be '-1'
Input:
     12
    /
  13
Output: 12 13 -1 -1 -1

Input:
      20
    /   \
   8     22
Output: 20 8 -1 -1 22 -1 -1

Input:
         20
       /
      8
     / \
    4  12
      /  \
     10  14
Output: 20 8 4 -1 -1 12 10 -1 -1 14 -1 -1 -1
```

```
Input:
          20
         /
        8
       /
     10
    /
   5
Output: 20 8 10 5 -1 -1 -1 -1 -1

Input:
          20
            \
             8
              \
              10
                \
                 5
Output: 20 -1 8 -1 10 -1 5 -1 -1
```

Implementation:

```
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Codec:
    def serialize(self, root):
        serialized_btree = []
        self._modified_pre_order(root, serialized_btree)
        return ",".join(serialized_btree)

    def _modified_pre_order(self, root, serialized_btree):
        if root:
            serialized_btree.append(str(root.val))
```

```
            self._modified_pre_order(root.left, serialized_btree)
            self._modified_pre_order(root.right, serialized_btree)
        else:
            serialized_btree.append("#")

    def deserialize(self, data):
        nodes_list = data.split(",")
        nodes_list.reverse()
        return self._build_tree(nodes_list)

    def _build_tree(self, nodes_list):
        if not nodes_list:
            return

        key = nodes_list.pop()
        if key != "#":
            root = TreeNode(int(key))
            root.left = self._build_tree(nodes_list)
            root.right = self._build_tree(nodes_list)
            return root


def print_tree(root):
    if root:
        print(root.val, end=" ")
        print_tree(root.left)
        print_tree(root.right)


# Your Codec object will be instantiated and called as such:
#      1
#     / \
#    2   3
#       / \
#      4   5
#    /  \
#   7    6


root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.right.left = TreeNode(4)
root.right.right = TreeNode(5)
root.right.left.left = TreeNode(7)
root.right.left.right = TreeNode(6)

codec = Codec()
print_tree(codec.deserialize(codec.serialize(root)))
print()

#       20
#     /
#    8
#   / \
# 4   12
#    /  \
```

```
#  10  14

root = TreeNode(20)
root.left = TreeNode(8)
root.left.left = TreeNode(4)
root.left.right = TreeNode(12)
root.left.right.left = TreeNode(10)
root.left.right.right = TreeNode(14)

codec = Codec()
print_tree(codec.deserialize(codec.serialize(root)))
print()
```

Output:

```
Code/LeetCodePractice/3_Hard/Q3_#297_serialize_deserialize_tree.py
1 2 3 4 7 6 5
20 8 4 12 10 14
(.venv) astikanand@Astik-Macbook:~/LeetCode/LeetCodePractice$
```

Complexity Analysis:

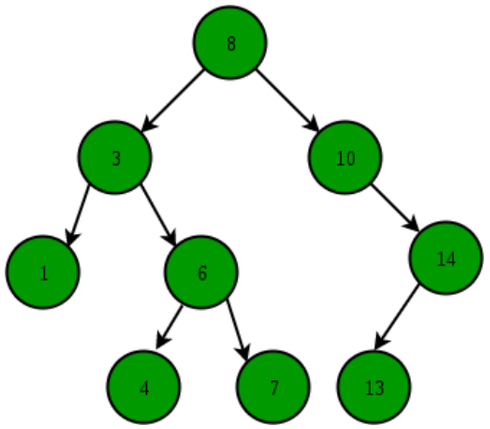
- **Time complexity: O(N)** In both serialization and deserialization functions, we visit each node exactly once, thus the time complexity is O(N), where N is the number of nodes, *i.e.* the size of tree.
- **Space complexity: O(N)** In both serialization and deserialization functions, we keep the entire tree, either at the beginning or at the end, therefore, the space complexity is O(N).

Binary Search Tree

What is Binary Search Tree ?

It 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.
- There must be **no duplicate** nodes.



Inserting a Key

```
        100                                  100
       /   \        Insert 40               /    \
     20     500    --------->             20     500
    /  \                                 /  \
   10   30                              10   30
                                              \
                                              40
```

Approach:

- A new key is always inserted at leaf.
- We start searching a key from root till we hit a leaf node.
- Once a leaf node is found, the new node is added as a child of the leaf node.

Searching a Key

- To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root.
- If key is greater than root's key, we recur for right subtree of root node.
- Otherwise we recur for left subtree.

Implementation:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None


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

```
        self.root = None

    def insert(self, key):
        self.root = self.insert_util(self.root, key)

    def insert_util(self, node, key):
        if node is None:
            return Node(key)
        elif key <= node.val:
            node.left = self.insert_util(node.left, key)
        else:
            node.right = self.insert_util(node.right, key)

        return node

    def print_inorder(self):
        self.print_inorder_util(self.root)
        print()

    def print_inorder_util(self, node):
        if node:
            self.print_inorder_util(node.left)
            print(node.val, end=" ")
            self.print_inorder_util(node.right)

    def search_key(self, key):
        return self.search_key_util(self.root, key)

    def search_key_util(self, node, key):
        if not node:
            return False
        elif node.val == key:
            return True
        elif key <= node.val:
            return self.search_key_util(node.left, key)
        else:
            return self.search_key_util(node.right, key)


print("Inserting 8, 3, 10, 1, 6, 14, 4, 7, 13 into BST")
nodes = [8, 3, 10, 1, 6, 14, 4, 7, 13]
b = BST()
for key in nodes:
    b.insert(key)

print("BST Now:")
b.print_inorder()

print("Searching if 5 is present ? : {}".format(b.search_key(5)))
print("Searching if 6 is present ? : {}".format(b.search_key(6)))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 10:14:06 ~/Personal/Notebooks/Data Structures/5. Binary Search Tree $ python3 0_insert_search_intro.py
Inserting 8, 3, 10, 1, 6, 14, 4, 7, 13 into BST.
BST Now:
1 3 4 6 7 8 10 13 14

Searching if 5 is present ? : False
Searching if 6 is present ? : True
```

Complexity:

- **Time: O(Logn)** for both insert and search. In skewed tree it maybe O(n)
- **Auxilliary Space: O(1)** if recursive call stack is not considered.

Some Interesting Facts:

- Inorder traversal of BST always produces sorted output.
- We can construct a BST with only Preorder or Postorder or Level Order traversal. Note that we can always get inorder traversal by sorting the only given traversal.
- Number of unique BSTs with n distinct keys is Catalan Number

Applications of Binary Search Tree

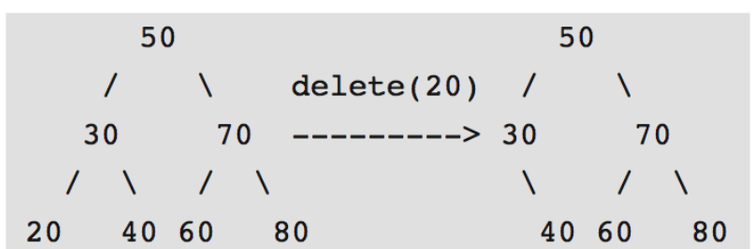
- Used to express arithmetic expressions.
- Used to evaluate expression trees.
- Used for managing virtual memory Areas (VMA's).
- Used for indexing IP addresses.
- Hashing would be faster, but want to avoid attacker sending IP packets with worst-case inputs.
- For dynamic sorting.

Standard BST Problems

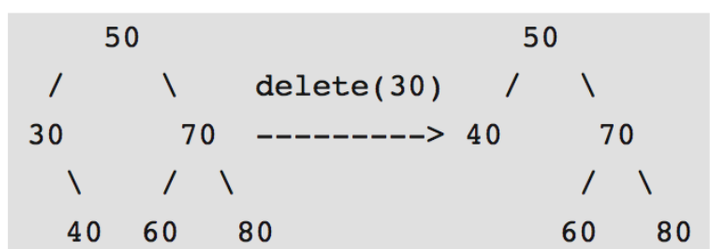
1. Delete Node in BST***

When we delete a node, three possibilities arise.

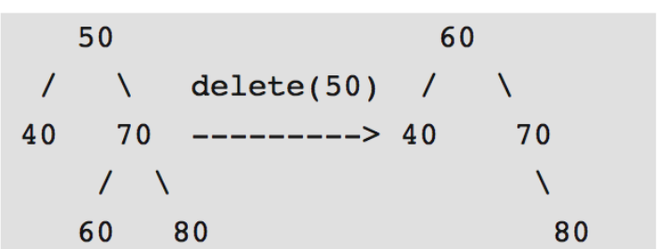
1) Node to be deleted is leaf:
Simply remove from the tree.



2) Node to be deleted has only one child:
Copy the child to the node and delete the child.



3) Node to be deleted has two children:
Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



Algorithm:

- Check if root None: key doesn't exist, not possible to delete.
- If key is lesser than root.val: **Delete** the **key** in **left subtree**.
- If key is greater than root.val: **Delete** the **key** in **right subtree**.
- If key is equal to root.val: Need to delete this root node.
  - If **no child exists:** make root None and return None.
  - If **left child exists**: make root None and return left child.
  - If **right child exists:** make root None and return the right child.
  - If **both child exists:**
    - Get the **min_node** from **right child subtree.**
    - Set the val of root as the val of min node.
    - Delete the min node from right subtree.
- Finally return the root.

Implementation:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

```
def delete(root, key):
    # Check if root None: key doesn't exist, not possible to delete.
    if root is None:
        return root

    # If key is lesser than root.val: Delete the key in left subtree.
    if(key < root.val):
        root.left = delete(root.left, key)
    # If key is greater than root.val: Delete the key in right subtree.
    elif(key > root.val):
        root.right = delete(root.right, key)
    # If key is equal to root.val: Need to delete this root node.
    else:
        # If no child exists: make root None and return None.
        if(root.left is None and root.right is None):
            root = None
            return None
        # If left child exists: make root None and return left child.
        elif(root.right is None):
            temp = root.left
            root = None
            return temp
        # If right child exists: make root None and return the right child.
        elif(root.left is None):
            temp = root.right
            root = None
            return temp
        # If both child exists:
        else:
            # Get the min_node from right child subtree.
            temp = min_node(root.right)
            # Set the val of root as the val of min node.
            root.val = temp.val
            # Delete the min node from right subtree.
            root.right = delete(root.right, temp.val)

    # Finally return the root.
    return root


def min_node(current_node):
    # If current_node is None, min_node not possible
    if(current_node is None):
        return None

    min_node = current_node
    while(min_node.left):
        min_node = min_node.left

    return min_node


def insert(root, key):
    if(root is None):
        root = Node(key)
```

```
    # If key is lesser than root.val insert key in left subtree
    if(key <= root.val):
        if(root.left is None):
            root.left = Node(key)
        else:
            insert(root.left, key)
    # If key is greater than root.val insert key in right subtree
    else:
        if(root.right is None):
            root.right = Node(key)
        else:
            insert(root.right, key)


def print_bst_inorder(root):
    if(root):
        print_bst_inorder(root.left)
        print(root.val, end=" ")
        print_bst_inorder(root.right)



print("Insert:- 50, 30, 70, 20, 40, 60, 80 into BST.")
root = Node(50)
insert(root, 30)
insert(root, 70)
insert(root, 20)
insert(root, 40)
insert(root, 60)
insert(root, 80)
print("BST at start:")
print_bst_inorder(root)
print("\n")

delete(root, 20)
print("BST after deleting 20:")
print_bst_inorder(root)
print()

delete(root, 30)
print("BST after deleting 30:")
print_bst_inorder(root)
print()

delete(root, 50)
print("BST after deleting 50:")
print_bst_inorder(root)
print()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 12:24:04 ~/Personal/Notebooks/Data Structures/5. Binary Search Tree $ python3 1_delete_node_bst.py
Insert:- 50, 30, 70, 20, 40, 60, 80 into BST.
BST at start:
20 30 40 50 60 70 80

BST after deleting 20:
30 40 50 60 70 80
BST after deleting 30:
40 50 60 70 80
BST after deleting 50:
40 60 70 80
```

Complexity:

- **Time: O(Logn)** for both insert and search. In skewed tree it maybe O(n)
- **Auxilliary Space: O(1)** if recursive call stack is not considered.

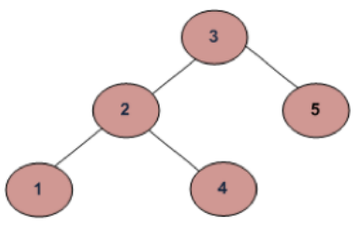
2. Check if Binary Tree is BST***

Problem:

Given a binary tree check if it is a binary search tree.

Approach-1: Simple but Wrong X

- For each node, check if left node is smaller and right node is greater.



This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3).

Approach-2: Correct but Inefficient

- For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.
- It runs slowly since it traverses over some parts of the tree many times and hence is inefficient.

Approach-3: Correct and Efficient

- Approach-2 runs slowly since it traverses over some parts of the tree many times.
- A better solution looks at each node only once.
- The trick is to write a utility helper function isBSTUtil(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once.
- The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

Approach-4: Inorder Traversal Method

- Do In-Order Traversal of the given tree and store the result in a temp array.
- Check if the temp array is sorted in ascending order, if it is, then the tree is BST.
- We can also avoid the use of Auxiliary Array.
    - While doing In-Order traversal, we can keep track of previously visited node.
    - If the value of the currently visited node is less than the previous value, then tree is not BST.

Implementation (Approach-3):

```
import sys
INT_MIN = -sys.maxsize-1
INT_MAX = sys.maxsize

class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None


def is_bst(root):
    return (is_bst_util(root, INT_MIN, INT_MAX))


def is_bst_util(root, min_val, max_val):
    # An empty tree is BST
    if root is None:
        return True

    # If current root's val is either less than min_val allowed or greater than max_val allowed return False.
    if root.val < min_val or root.val > max_val:
        return False
```

```
    # Check the subtrees recursively tightening the min or max constraint
    return (is_bst_util(root.left, min_val, root.val-1) and is_bst_util(root.right, root.val+1, max_val))


def print_tree(root):
    if(root):
        print_tree(root.left)
        print(root.val, end=" ")
        print_tree(root.right)



root = Node(4)
root.left = Node(2)
root.right = Node(5)
root.left.left = Node(1)
root.left.right = Node(3)
print("Binary Tree:")
print_tree(root)
print()
print("Is this binary Tree a BST ? : {}".format(is_bst(root)))

root = Node(3)
root.left = Node(2)
root.right = Node(5)
root.left.left = Node(1)
root.left.right = Node(4)
print("\nAnother Binary Tree:")
print_tree(root)
print()
print("Is this binary Tree a BST ? : {}".format(is_bst(root)))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 15:01:57 ~/Personal/Notebooks/Data Structures/5. Binary Search Tree $ python3 2_check_binary_tree_is_bst.py
Binary Tree:
1 2 3 4 5
Is this binary Tree a BST ? : True

Another Binary Tree:
1 2 4 3 5
Is this binary Tree a BST ? : False
```

Complexity:

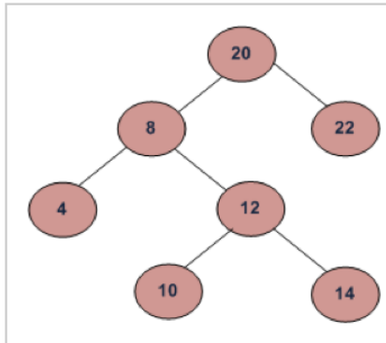
- **Time: O(n)**
- **Auxilliary Space: O(1)** if recursive call stack is not considered.

3. Lowest Common Ancestor in BST (LCA)***

Problem:

Given values of two values n1 and n2 in a Binary Search Tree, find the Lowest Common Ancestor (LCA).

We may assume that both the values exist in the tree.



LCA:

- LCA of 10 and 14 is 12
- LCA of 14 and 8 is 8
- LCA of 10 and 22 is 20

Approach:

- If both key1 and key2 is smaller than root's val, then lca exist in left subtree.
- If both key1 and key2 is greater than root's val, then lca exist in right subtree.
- Else this root is LCA.

Implementation:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None


def lowest_common_ancestor(root, key1, key2):
    while(root):
        # If both key1 and key2 is smaller than root's val, then lca exist in left subtree.
        if(key1 < root.val and key2 < root.val):
            root = root.left
        # If both key1 and key2 is greater than root's val, then lca exist in right subtree.
        elif(key1 > root.val and key2 > root.val):
            root = root.right
        # Else this root is LCA.
```

```
        else:
            break

    return root.val



root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)
print("Lowest Common Ancestor of 10 and 14 is : {}".format(lowest_common_ancestor(root, 10, 14)))
print("Lowest Common Ancestor of 14 and 8 is : {}".format(lowest_common_ancestor(root, 14, 8)))
print("Lowest Common Ancestor of 10 and 22 is : {}".format(lowest_common_ancestor(root, 10, 22)))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 19:25:02 ~/Personal/Notebooks/Data Structures/5. Binary Search Tree $ python3 3_lowest_common_ancestor.py
Lowest Common Ancestor of 10 and 14 is : 12
Lowest Common Ancestor of 14 and 8 is : 8
Lowest Common Ancestor of 10 and 22 is : 20
```

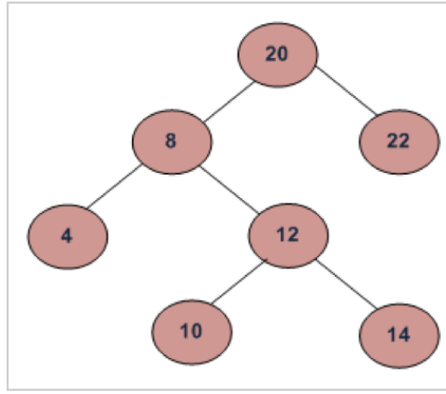
Complexity:

- **Time: O(n)**
- **Auxilliary Space: O(1)**

4. Inorder Successor in BST***

Problem:

- Inorder successor of a node is the next node in Inorder traversal of the Binary Tree.
- Inorder Successor is NULL for the last node in Inoorder traversal.
- Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of input node.
- So, it is sometimes important to find next node in sorted order.



Inorder Successor:

- Inorder successor of 8 is 10
- Inorder successor of 10 is 12
- Inorder successor of 14 is 20.

Approach:

The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

1. **If right subtree of given_node is NOT NULL:** then successor lies in right subtree.
    - Go to right subtree and return the node with minimum key value in right subtree.
2. **If right subtree of given_node is NULL:** then start from root and use search like technique.
    - Travel down the tree, if a node's data > root's data then go right side, otherwise go to left side.

Algorithm:

- Get the given node by search using key.
- If given_node's right exist, simply return the min_node from right.
- **Else:** Set successor as None and start from root and search for successor by travelling down the tree.

  given_node हमेशा successor के left subtree में होना चाहिए, इसलिए successor तभी update करेंगे जब left subtree में जाएंगे ।

    - If given_node's data < root's data then go left side and update the successor.
    - Else if a given_node's data < root's data then go to right side.
    - Else break when given_node's data and root's data are equal, given_node is found.
- Finally return the successor.

Implementation:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

```python
def inorder_successor(root, key):
    # Get the given node by search using key.
    given_node = search(root, key)

    # If given_node's right exist, simply return the min_node from right.
    if(given_node.right):
        return min_node(given_node.right)

    # Set successor as None and start from root and search for successor by travelling down the tree.
    successor = None
    while(root):
        # If given_node's data < root's data then go left side and update the successor.
        # given_node हमेशा successor के left subtree में होना चाहिए,
        # इसलिए successor तभी update करेंगे जब left subtree में जाएंगे ।
        if(given_node.val < root.val):
            successor = root
            root = root.left
        # Else if a given_node's data > root's data then go to right side.
        elif (given_node.val > root.val):
            root = root.right
        # Else break when given_node's data and root's data are equal, given_node is found.
        else:
            break

    # Finally return the successor.
    return successor


def search(root, key):
    # If root is None, then key doesn't exist.
    if root is None:
        return root

    # If root's val matches the key, then we have found the key in
    if(root.val == key):
        return root

    # If key is lesser than root's val search in left subtree else serach in right subtree.
    if(key < root.val):
        return search(root.left, key)
    else:
        return search(root.right, key)


def min_node(current_node):
    # If current_node is None, min_node not possible
    if(current_node is None):
        return None

    min_node = current_node
    while(min_node.left):
        min_node = min_node.left

    return min_node
```

```
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)
print("Inorder Successor of  8 is : {}".format(inorder_successor(root, 8).val))
print("Inorder Successor of 10 is : {}".format(inorder_successor(root, 10).val))
print("Inorder Successor of 14 is : {}".format(inorder_successor(root, 14).val))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:04:22 ~/Personal/Notebooks/Data Structures/5. Binary Search Tree $ python3 4_inorder_successor.py
Inorder Successor of  8 is : 10
Inorder Successor of 10 is : 12
Inorder Successor of 14 is : 20
```

Complexity:

- **Time: O(Logn)**
- **Auxilliary Space: O(1)**

Heap / Priority Queue

What is heap ?

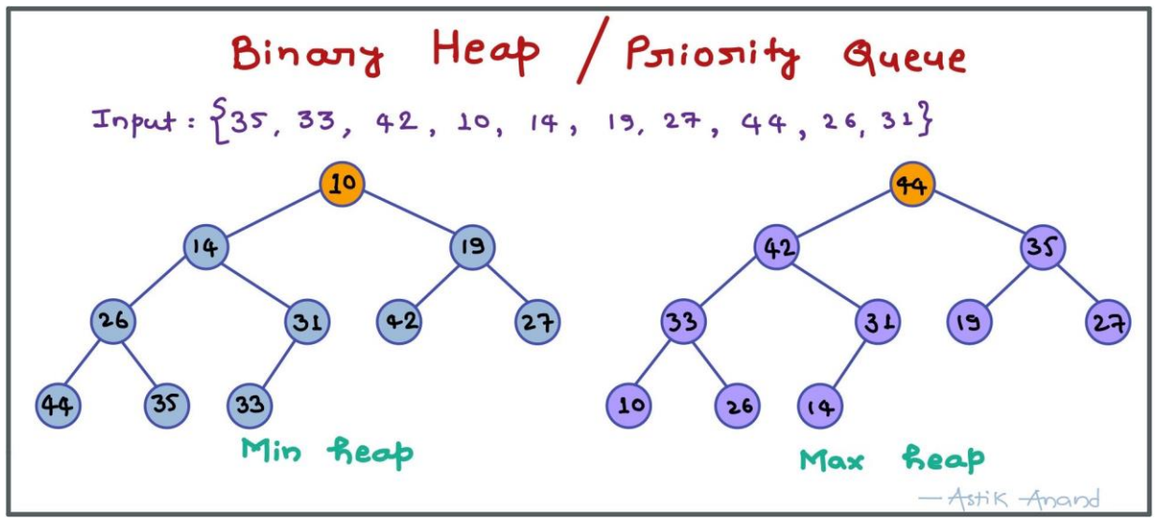
- It is a specialized tree-based data structure which is essentially an almost complete tree that satisfies the heap property.
- Satisfying heap property means in max-heap every parent will be greater than children and smaller in case of min-heap.
- The heap is one maximally efficient implementation of an abstract data type called a **priority queue** and often referred as “heaps”.

Different Variations of Heaps

- **Binary Heap**
- **Binomial Heap**
- **Fibonacci Heap**

Binary Heap

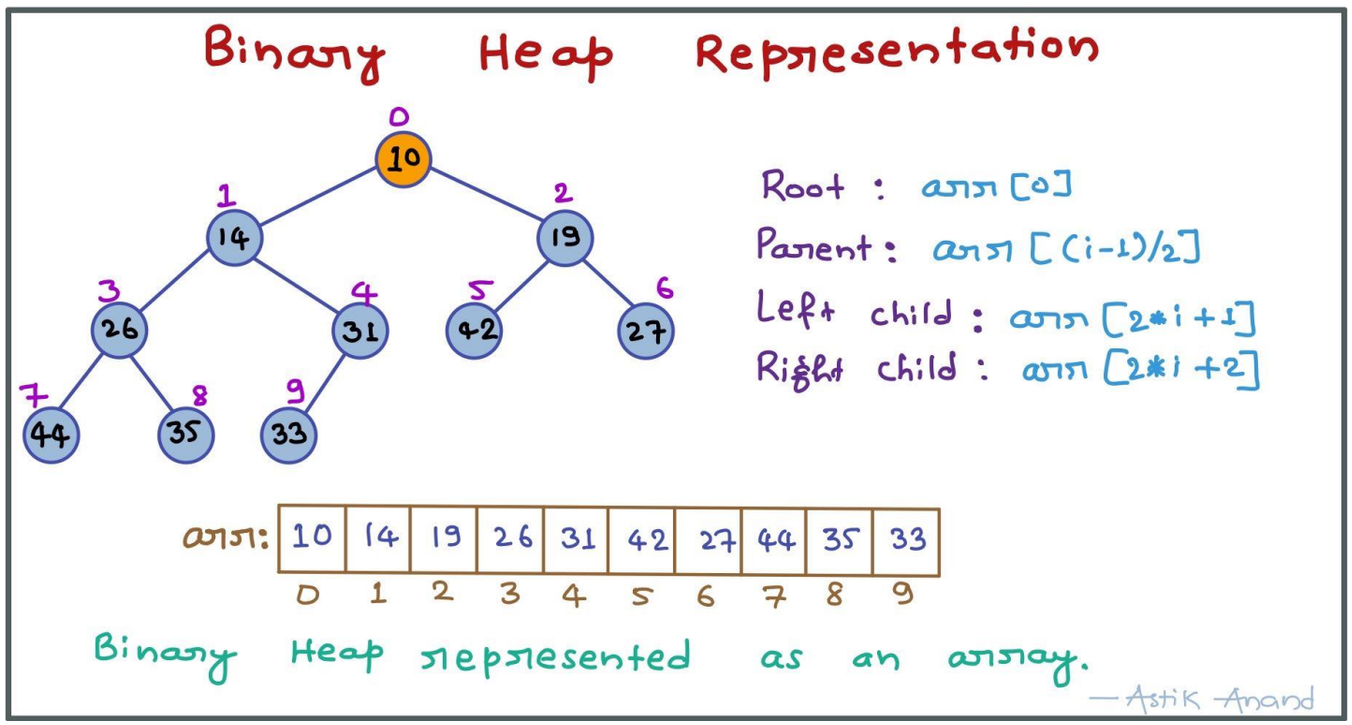
- Introduced by **J. W. J. William**s in **1964**.
- A common implementation of a heap is the binary heap, in which the tree is a binary tree
- **A Binary Heap is a Binary Tree with following properties:**
    i. It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
    ii. A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Heap is similar to MinHeap.



Binary Heap Representation

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- The root element will be at Arr[0].
- **Parent Node:** Arr[(i-1)/2]
- **Left Child Node:** Arr[(2*i)+1]
- **Right Child Node:** Arr[(2*i)+2]

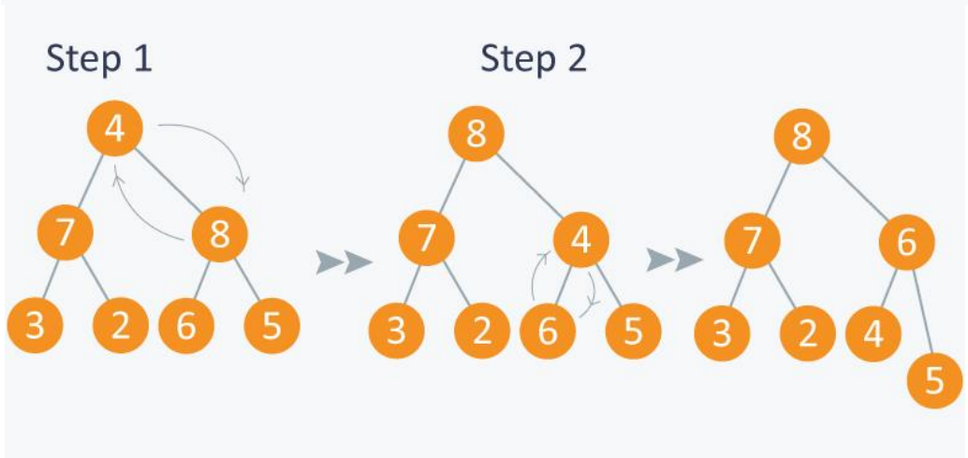


heapify():

- **To maintain the heap property.**
- Let's assume that we have a heap having some elements which are stored in array .
- The way to convert this array into a heap structure is the following.
    - Pick a node in the array, check if the left sub-tree and the right sub-tree are max heaps in themselves and the node itself is a max heap (it's value should be greater than all the child nodes).
- To do this we will implement a function that can maintain the property of max heap (i.e each element value should be greater than or equal to any of its child and smaller than or equal to its parent).
- **Time Complexity: O(logN)**

```
def max_heapify(arr, i, N):
    left_child = 2*i+1
    right_child = 2*i+2

    # Find the largest of left_child, right_child and parent which is i.
    largest = left_child if left_child < N and arr[left_child] > arr[i] else i
    largest = right_child  if right_child < N and arr[right_child] > arr[largest]

    # If Parent is not largest, swap and apply max_heapify on child to propagate it down
    if largest != i:
```

```
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, largest, N)
```



build_heap():

- **Builds the heap from the given array.**
- Now let's say we have **N** elements stored in the array **Arr** indexed from **0** to **N-1** .
- They are currently not following the property of max heap.
- So we can use max-heapify function to make a max heap out of the array.
- We know that elements from **Arr[n/2]** to **Arr[n-1]** are leaf nodes, and each node is a 1 element heap.
- Here, we can use **max_heapify** function in a bottom up manner on **remaining internal nodes**, so that we can cover each node of tree.
- **Time Complexity: O(N)**
- It seems complexity is O(NLogN) as we are calling heapify for N/2 times, but it has been proved that amortized time is O(N) only.

```
def build_max_heap(arr):
    n = len(arr)
    for i in range(n//2, -1, -1):
        max_heapify(arr, i, n)

    return arr
```

Operations on Heap

1. **getMin():** It returns the root element of Min Heap. Time Complexity: **O(1)**.

2. **extractMin():** Removes the minimum element from MinHeap. Time Complexity: **O(Logn)** as this operation needs to maintain the heap property (by calling heapify()) after removing root.
3. **decreaseKey():** Decreases value of key. Time Complexity: **O(Logn)**. If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
4. **insert():** Inserting a new key takes **O(Logn)** time. We add a new key at the end of the tree. If new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
5. **delete():** Deleting a key also takes **O(Logn)** time. We replace the key to be deleted with minum infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove the key.

Implementation

```
import sys
from heapq import heappush, heappop, heapify

# Heap functions from python library
#   • heappop - pop and return the smallest element from heap
#   • heappush - push the value item onto the heap, maintaining heap invarient
#   • heapify - transform list into heap, in place, in linear time

MIN = -sys.maxsize-1


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

    def parent(self, i):
        return int((i-1)/2)

    # Inserts a new key 'k'
    def insertKey(self, k):
        heappush(self.heap, k)

    # Decrease value of key at index 'i' to new_val
    # It is assumed that new_val is smaller than heap[i]
    def decreaseKey(self, i, new_val):
        self.heap[i]  = new_val
        while(i != 0 and self.heap[self.parent(i)] > self.heap[i]):
            # Swap heap[i] with heap[parent(i)]
            self.heap[i] , self.heap[self.parent(i)] = self.heap[self.parent(i)], self.heap[i]

    # Method to remove minium element from min heap
    def extractMin(self):
        return heappop(self.heap)

    # This functon deletes key at index i. It first reduces
    # value to minus infinite and then calls extractMin()
    def deleteKey(self, i):
        self.decreaseKey(i, MIN)
        self.extractMin()

    # Get the minimum element from the heap
```

```
    def getMin(self):
        return self.heap[0]



print("Example:- Array Implementation of Heap")
print("InsertKey: 3, 2 then DeleteKey: 1 and then InsertKey: 15, 5, 4, 45")
my_heap = MinHeap()
my_heap.insertKey(3)
my_heap.insertKey(2)
my_heap.deleteKey(1)
my_heap.insertKey(15)
my_heap.insertKey(5)
my_heap.insertKey(4)
my_heap.insertKey(45)

print("Extracted Min = {}".format(my_heap.extractMin()))
print("Get Min = {}".format(my_heap.getMin()))
my_heap.decreaseKey(2, 1)
print("After Decreasing Key of 2 to 1, Now Get Min = {}".format(my_heap.getMin()))
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 02:47:21 ~/Personal/Notebooks/Data Structures/6. Heap $ python3 0_heap_array_implementation.py
Example:- Array Implementation of Heap
InsertKey: 3, 2 then DeleteKey: 1 and then InsertKey: 15, 5, 4, 45
Extracted Min = 2
Get Min = 4
After Decreasing Key of 2 to 1, Now Get Min = 1
```

Applications of Heap

1. **Heap Sort:** It uses Binary Heap to sort an array in O(nLogn) time.
2. **Priority Queue:** Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
3. **Graph Algorithms:** The priority queues are especially used in Graph Algorithms like **Dijkstra's Shortest Path** and **Prim's Minimum Spanning Tree**.
4. Many standard interesting problems can be efficiently solved using Heaps.
    - ***K^{th} Largest Element in an array.***
    - ***Sort an almost Sorted Array.***
    - ***Merge K Sorted Arrays.***

Standard Heap Problems

1. Find k largest/smallest elements in an array***

Problem:

Given an array of elements find k largest or smallest elements of it.

Examples:

Input: [1, 23, 12, 9, 30, 2, 50] and k=3

Output: 50, 30, 23

Approach-1: Use Sorting

- Sort the elements and then pick its first k elements.
- **Time Complexity:** O(nlogn)

Approach-2 Use Heap

- Build a Max Heap tree in O(n).
- Use Extract Max k times to get k maximum elements from the Max Heap O(klogn).
- **Time complexity:** O(n + klogn)

Implementation

```
from heapq import heapify, nlargest, nsmallest

def find_k_largest(arr, k):
    heapify(arr)
    print("The {} largest numbers in list are : {}".format(k, nlargest(k, arr)))

def find_k_smallest(arr, k):
    heapify(arr)
    print("The {} smallest numbers in list are : {}".format(k, nsmallest(k, arr)))


print("Example-1: find_k_largest([1, 23, 12, 9, 30, 2, 50], 3)")
find_k_largest([1, 23, 12, 9, 30, 2, 50], 3)

print("\nExample-2: find_k_smallest([1, 23, 12, 9, 30, 2, 50], 2)")
find_k_smallest([1, 23, 12, 9, 30, 2, 50], 2)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 03:19:07 ~/Personal/Notebooks/Data Structures/6. Heap $ python3 2_find_k_largest.py
Example-1: find_k_largest([1, 23, 12, 9, 30, 2, 50], 3)
The 3 largest numbers in list are : [50, 30, 23]

Example-2: find_k_smallest([1, 23, 12, 9, 30, 2, 50], 2)
The 2 smallest numbers in list are : [1, 2]
```

Complexity:

- **Time: O(n+klogn)**

Hash

What is Hash Data Structure ?

- It is a dictionary type data structure which can give the value of key if present in O(1) time.
- It stores the key-value pairs.
- Actually, it is a technique that is used to uniquely identify a specific object from a group of similar objects.

Practical Life Examples:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

Representation of Hash

Data is stored using a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hash Function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. **Easy to compute:** It should be easy to compute and must not become an algorithm in itself.
2. **Uniform distribution:** It should provide a uniform distribution across the hash table and should not result in clustering.
3. **Less collisions:** Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Notes:

- Irrespective of how good a hash function is, collisions are bound to occur.
- Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Hashing Operations

- Hashing provides constant time search, insert and delete operations on average.
- This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc

Applications of Hashing

There are many other applications of hashing, including modern day cryptography hash functions. Some Examples are:

- Message Digest
- Password Verification
- Data Structures(Programming Languages)
- Compiler Operation
- Rabin-Karp Algortithm
- Linking File name and path together

Some Standard Hashing Problems

1. Find Pair With Given Sum

Problem:

Given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.

Examples:

Input: [1, 4, 45, 6, 10, -8] and sum = 16

Output: True (6, 10)

Approach-1: Sorting

- Sort the numbers and then start from leftmost and rightmost.
- If sum is larger shift rightmost to left and if smaller shift leftmost to right.
- Keep doing this until we find a sum.
- **Time Complexity: O(nlogn)**

Approach-2: Use Hashing

- Take an empty hash.
- For every element in array check if **x-current** is already present in hash and if not put current in hash.
- If x-current is already there then numbers are current and x-current.
- **Time Complexity: O(n)**

Implementation

```
def find_pair_with_given_sum(arr, x):
    my_hash = {}
    sum_found = False
    for i in arr:
        if my_hash.get(x-i, False):
            print("Pair with sum {} is: [{}, {}]".format(x, i, x-i))
            sum_found = True
        else:
            my_hash[i] = True

    if not sum_found:
        print("Pair with sum {} is NOT Present.".format(x))


print("Example-1: find_pair_with_given_sum([1, 4, 45, 6, 10, -8], 16)")
find_pair_with_given_sum([1, 4, 45, 6, 10, -8], 16)

print("\nExample-2: find_pair_with_given_sum([1, 4, 45, 6, 10, -8], 13)")
find_pair_with_given_sum([1, 4, 45, 6, 10, -8], 13)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 04:00:42 ~/Personal/Notebooks/Data Structures/7. Hash $ python3 1_pair_with_given_sum.py
Example-1: find_pair_with_given_sum([1, 4, 45, 6, 10, -8], 16)
Pair with sum 16 is: [10, 6]

Example-2: find_pair_with_given_sum([1, 4, 45, 6, 10, -8], 13)
Pair with sum 13 is NOT Present.
```

2. Find whether an array is subset of another array

Problem:

Given two arrays: arr1[0..m-1] and arr2[0..n-1].

Find whether arr2[] is a subset of arr1[] or not. Both the arrays are not in sorted order.

It may be assumed that elements in both array are distinct.

Examples:

Input: arr1[] = [11, 1, 13, 21, 3, 7] arr2[] = [11, 3, 7, 1] **Output**: True

Input: arr1[] = [1, 2, 3, 4, 5, 6] arr2[] = [1, 2, 4] **Output**: True

Input: arr1[] = [10, 5, 2, 23, 19] arr2[] = [19, 5, 3] **Output**: False

Approach-1: Brute-Force

- Use two loops: The outer loop picks all the elements of arr2[] one by one.
- The inner loop linearly searches for the element picked by the outer loop.
- If all elements are found then return True, else return False.
- **Time Complexity: $O(n^2)$**

Approach-2: Sorting and Binary Search

- Sort arr1[] which takes O(mLogm)
- For each element of arr2[], do binary search for it in sorted arr1[].
- If the element is not found then return False, If all elements are present then return True.
- **Time Complexity: O(mLogm + nLogm)**

Approach-3: Use Hashing

- Create a Hash for all the elements of arr1[].
- Traverse arr2[] and search for each element of arr2[] in the Hash Table.
- If the element is not found then return False, If all elements are present then return True.
- **Time Complexity: O(m+n)**

Implementation

```
def check_if_arr_is_subset(arr1, arr2):
    my_hash = {}
    for i in arr1:
        my_hash[i] = True

    status = True
    for i in arr2:
        if not my_hash.get(i, False):
            status = False
            break

    if status:
        print("True")
    else:
        print("False")


print("Example-1: check_if_arr_is_subset([11, 1, 13, 21, 3, 7], [11, 3, 7, 1])")
check_if_arr_is_subset([11, 1, 13, 21, 3, 7], [11, 3, 7, 1])
print("\nExample-2: check_if_arr_is_subset([1, 2, 3, 4, 5, 6], [1, 2, 4])")
check_if_arr_is_subset([1, 2, 3, 4, 5, 6], [1, 2, 4])
print("\nExample-3: check_if_arr_is_subset([10, 5, 2, 23, 19], [19, 5, 3])")
check_if_arr_is_subset([10, 5, 2, 23, 19], [19, 5, 3])
```

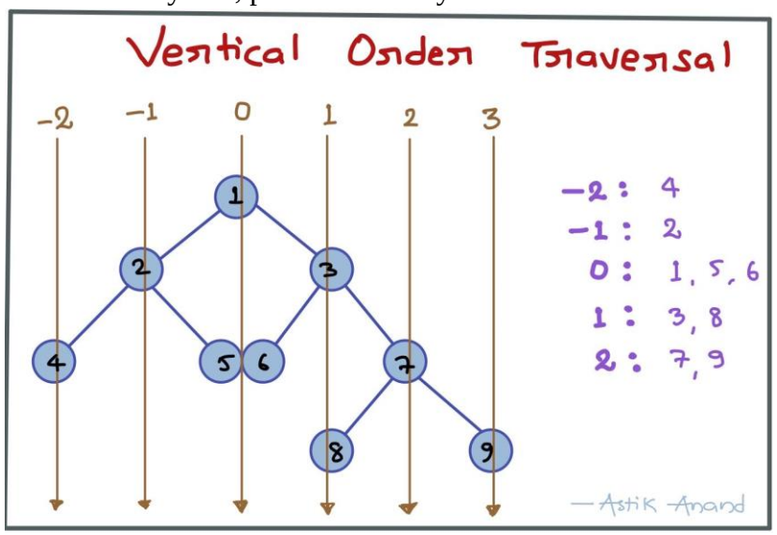
Output:

```
astik.anand@mac-C02XD95ZJG5H 04:18:18 ~/Personal/Notebooks/Data Structures/7. Hash $ python3 2_check_if_array_is_subset.py
Example-1: check_if_arr_is_subset([11, 1, 13, 21, 3, 7], [11, 3, 7, 1])
True

Example-2: check_if_arr_is_subset([1, 2, 3, 4, 5, 6], [1, 2, 4])
True

Example-3: check_if_arr_is_subset([10, 5, 2, 23, 19], [19, 5, 3])
False
```

3. Vertical Order Traversal***

Problem:

Given a binary tree, print it vertically.



Approach: Use Hashing

- Start from root as level=0, and when going left decrease the level by 1 and while going right increse the level by 1.
- Store the level as key and value as list of element at that particular level.

Implementation

```
from collections import defaultdict

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


def get_vertical_order(root, level, my_hash):
    if root is None:
        return

    # Put the root in hash with level
    my_hash[level].append(root.val)

    # Decrease the level while going in left subtree
```

```
    get_vertical_order(root.left, level-1, my_hash)

    # Increase the level while going in right subtree
    get_vertical_order(root.right, level+1, my_hash)


def print_vertical_order(root):
    # Hash to store vertical order
    my_hash = defaultdict(list)
    level = 0
    get_vertical_order(root, level, my_hash)

    for key, values in sorted(my_hash.items()):
        print("[{}] :-----> ".format(key), end = "")
        for v in values:
            print(v, end=" ")
        print()



root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)
root.right.right.right = Node(9)
print ("Vertical order traversal:")
print_vertical_order(root)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 04:41:13 ~/Personal/Notebooks/Data Structures/7. Hash $ python3 3_binary_tree_vertical_order_traversal.py
Vertical order traversal:
[-2] :-----> 4
[-1] :-----> 2
[0] :-----> 1 5 6
[1] :-----> 3 8
[2] :-----> 7
[3] :-----> 9
```

Graph

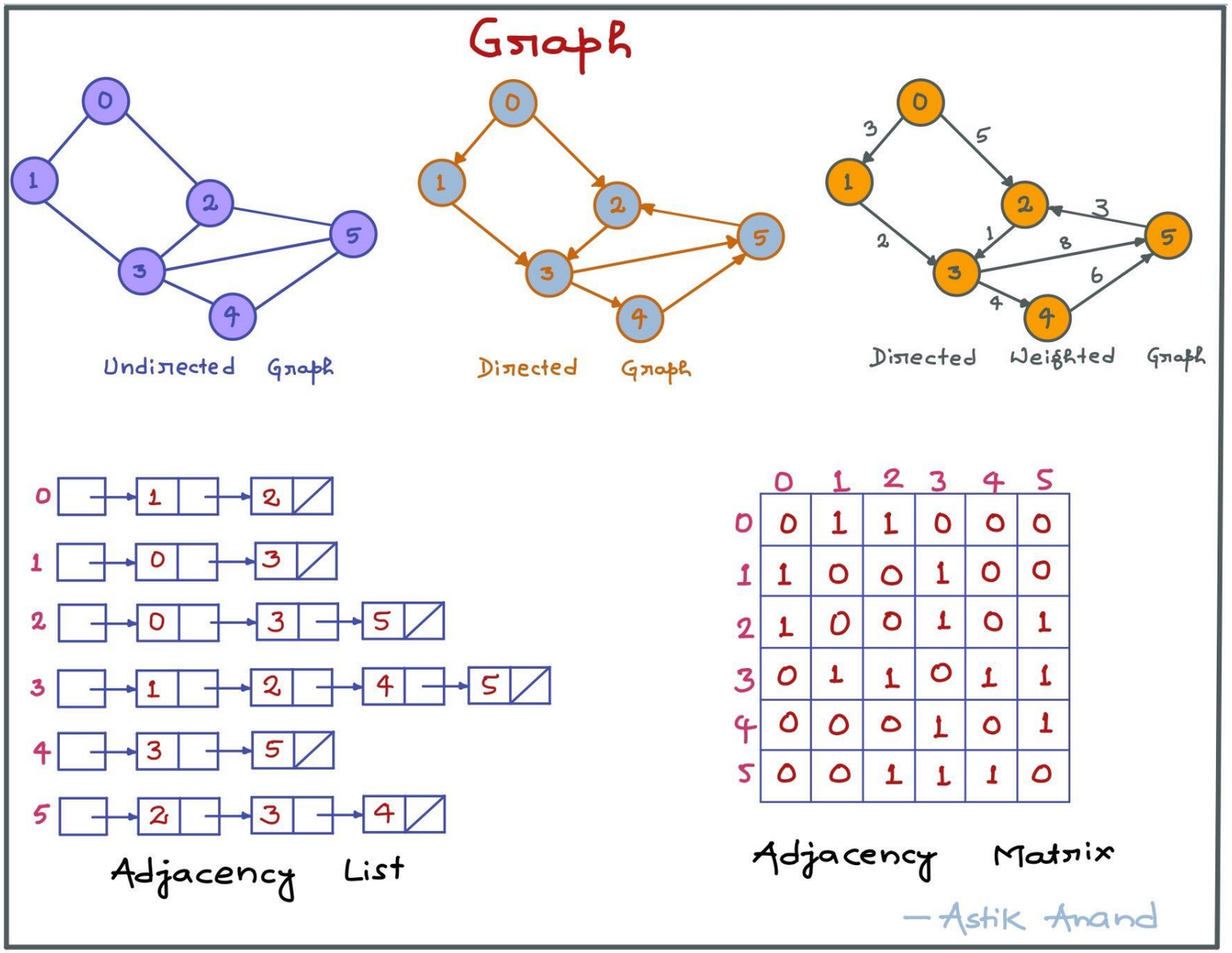
What is Graph ?

Graph is a data structure that consists of following two components:

- A finite set of vertices also called as **nodes**.
- A finite set of ordered pair of the form (u, v) called as **edge**.
- The pair is ordered because (u, v) is not same as (v, u) in case of a **directed graph(di-graph)**.
- The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v.
- The edges may contain weight/value/cost.

Representation

- Following are the 2 most commonly used representations of a graph.
    i. **Adjacency Matrix**
    ii. **Adjacency List**
- The choice of the graph representation is situation specific.
- It totally depends on the type of operations to be performed and ease of use.



1. 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.
- **Pros:**
    - Representation is easier to implement and follow.
    - Removing an edge takes O(1) time.
    - Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).
- **Cons:**
    - Consumes more space $O(V^2)$.
    - Even if the graph is sparse(contains less number of edges), it consumes the same space.
    - Adding a vertex is $O(V^2)$ time.

2. Adjacency List

- An array of linked lists is used.
- Size of the array is equal to the number of vertices.
- Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.
- **Pros:**
    - Saves space `O(|V|+|E|)`.
    - In the worst case, there can be C(V, 2) number of edges in a graph thus consuming $O(V^2)$ space.
    - Adding a vertex is easier.
- **Cons:**
- Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

Implementation

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)      # If undirected

    def print_graph(self):
        # print the vertex
        for vertex in self.graph:
            print("[{}]".format(vertex), end="")
```

```
            # print the connected_vertices to this vertex
            for connected_vertex in self.graph[vertex]:
                print("-->{}".format(connected_vertex), end="")
            print()



print("Example-1: Graph Representation")
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
g.add_edge(2, 5)
g.add_edge(3, 4)
g.add_edge(3, 5)
g.add_edge(4, 5)
g.print_graph()
```

Output:

```
astikanand@Developer-MAC 02:06:59 ~/Interview-Preparation/Data Structures/8. Graph $ python3 0_graph_representation.py
Example-1: Graph Representation
[0]-->1-->2
[1]-->0-->3
[2]-->0-->3-->5
[3]-->1-->2-->4-->5
[5]-->2-->3-->4
[4]-->3-->5
```

Applications of Graph

Graphs are used to represent many real-life applications:

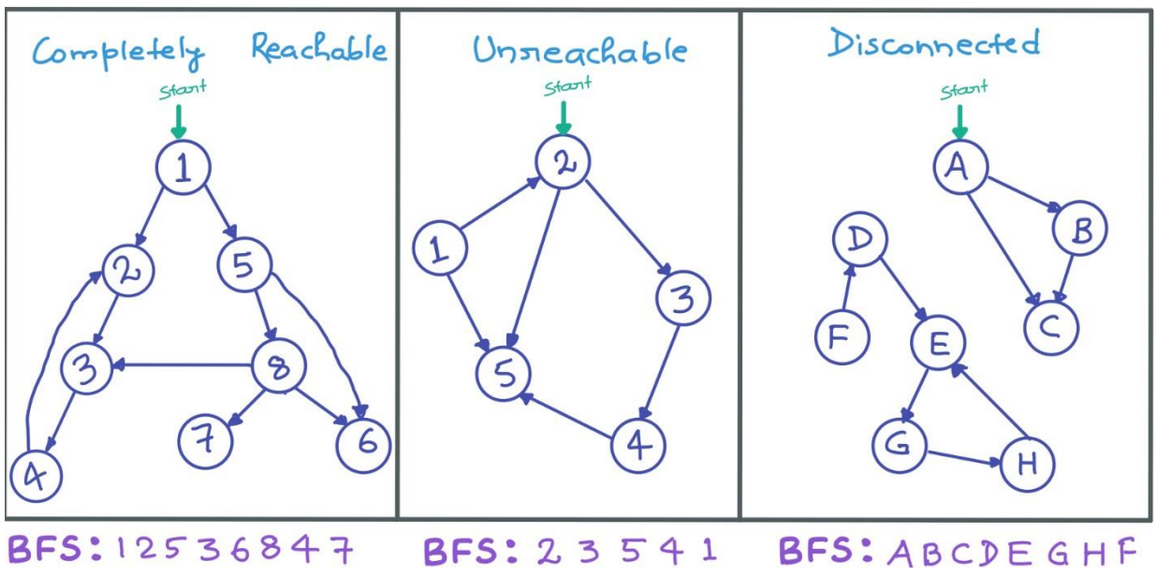
- **Computer Science**
  - Graphs are used to represent **networks**. The networks may include paths in a city, Telephone network or Circuit network
  - Graphs are also used in **social networks** like linkedIn, Facebook.
    - Example:- Facebook :: Each person is represented with a vertex/node.
    - Each node is a structure and contains information like person id, name, gender and locale.
  - The **links structure of a website** can be represented by a directed graph.
    - Vertices represent web pages and directed edges represent links from one page to another.
  - **Graph Databases** geared towards transaction-safe, persistent storing and querying of graph-structured data.
- **Linguistics**
  - Graph-theoretic methods, in various forms, have proven particularly useful in linguistics, since **natural language** often lends itself well to discrete structure.
  - Traditionally, **syntax and compositional semantics** follow tree-based structures, whose expressive power lies in the principle of compositionality modeled in a hierarchical graph.

    - More contemporary approaches such as **head-driven phrase structure grammar** model the syntax of natural language using typed features structures which are directed acyclic graphs.
- **Physics and Chemistry**
    - Graph theory is also used to study **molecules in chemistry and physics**.
    - In **condensed matter physics**, the 3-dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the **topology of the atoms**.
    - In chemistry a graph makes a **natural model for a molecule**, where vertices represent atoms and edges bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching.
    - In **statistical physics**, graphs can represent local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems.
    - Similarly, in **computational neuroscience**, graphs can be used to represent functional connections between brain areas that interact to give rise to various cognitive processes, where the vertices represent different areas of the brain and the edges represent the connections between those areas.
    - Graph theory plays an important role in **electrical modeling of electrical networks**, here, weights are associated with resistance of the wire segments to obtain electrical properties of network structures.
    - Graphs are also used to represent the **micro-scale channels of porous media**, in which the vertices represent the pores and the edges represent the smaller channels connecting the pores.
- **Social Sciences**
    - Graph theory is also widely used in **sociology** as a way, for example, to **measure actor's prestige** or to explore rumor spreading, notably through the use of social network analysis software.
    - Under the umbrella of social networks are many different types of graphs.
        - **Acquaintanceship and friendship graphs** describe whether people know each other.
        - **Influence graphs** model whether certain people can influence the behavior of others.
        - Finally, **collaboration graphs** model whether two people work together in a particular way, such as acting in a movie together.
- **Biology**
    - Useful in biology and conservation efforts where a vertex can represent regions where certain species exist (or inhabit) and the edges represent migration paths or movement between the regions.
    - This information is important when looking at breeding patterns or tracking the spread of disease, parasites or how changes to the movement can affect other species.
- **Mathematics**
    - Graphs are useful in **geometry** and certain parts of topology such as knot theory.
    - Algebraic graph theory has close links with group theory.
- **Others**
    - **Weighted graphs** are used to represent structures in which pairwise connections have some numerical values.
        - For example, if a graph represents a **road network**, the weights could represent the length of each road.
        - There may be several weights associated with each edge, including distance, travel time, or monetary cost.
        - Such weighted graphs are commonly used to **program GPS's**, and **travel-planning search engines** that compare flight times and costs.

Algo-1: Breadth First Seach (BFS)

What is BFS ?

- BFS for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.
- To avoid processing a node more than once, we use a boolean visited array.



Approach or Algorithm:

1. Start with the given start_vertex, take a queue and enqueue the start_vertex to it and mark it as visited.
2. While queue is not empty take out the current_vertex from queue and print it:
    - See all the connected vertices to current_vertex:
        - If any connected_vertex is not visited enqueue to the queue and mark it visited.
3. Check if any unvisited vertex (case of UNREACHABLE or DISCONNECTED graphs):
    - Call the function again from unvisited vertex

Implementation:

```
class Graph:
    def __init__(self):
        self.graph = {}
```

```
    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, u, v):
        self.add_vertex(u)
        self.add_vertex(v)
        self.graph[u].append(v)

    def unvisited(self, visited):
        for vertex in visited:
            if(visited[vertex] is False):
                return vertex
        return None

    def bfs_traversal_util(self, start_vertex, visited):
        # Take a queue and enqueue the start_vertex and mark it as visited
        queue = []
        queue.append(start_vertex)
        visited[start_vertex] = True

        # Dequeue out the current_vertex from queue and print it
        while(queue):
            current_vertex= queue.pop(0)
            print("{}".format(current_vertex), end=" ")
            # See all the connected vertices to current_vertex
            for connected_vertex in self.graph[current_vertex]:
                # If any connected_vertex is not visited enqueue to the queue and mark it visited
                if(visited[connected_vertex] is False):
                    queue.append(connected_vertex)
                    visited[connected_vertex] = True

    def bfs_traversal(self, start_vertex):
        # Mark every every vertex as unvisited
        visited = {}
        for vertex in self.graph:
            visited[vertex] = False

        # Call the bfs_traversal_util with start_vertex
        self.bfs_traversal_util(start_vertex, visited)

        # Check if there is still any unvisited vertex
        # Only when graph is UNREACHABLE or DISCONNECTED below lines will be executed
        while(self.unvisited(visited) is not None):
            # Call the bfs_traversal_util from the unvisited vertex
            self.bfs_traversal_util(self.unvisited(visited), visited)
        print()


print("Example-1: BFS Traversal of Graph from vertex-1:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(4, 2)
g.add_edge(1, 5)
```

```
g.add_edge(5, 6)
g.add_edge(5, 8)
g.add_edge(8, 3)
g.add_edge(8, 6)
g.add_edge(8, 7)
g.bfs_traversal(1)

print("Example-2: BFS Traversal of Graph from vertex-2:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 5)
g.add_edge(2, 3)
g.add_edge(2, 5)
g.add_edge(3, 4)
g.add_edge(4, 5)
g.bfs_traversal(2)

print("Example-3: BFS Traversal of Graph from vertex-A:")
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "C")
g.add_edge("D", "E")
g.add_edge("E", "G")
g.add_edge("F", "D")
g.add_edge("G", "H")
g.add_edge("H", "E")
g.bfs_traversal("A")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 00:27:09 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 1_bfs.py
Example-1: BFS Traversal of Graph from vertex-1:
1 2 5 3 6 8 4 7
Example-2: BFS Traversal of Graph from vertex-2:
2 3 5 4 1
Example-3: BFS Traversal of Graph from vertex-A:
A B C D E G H F
```

Complexity:

- **Time: O(V+E)** :- V is no. of vertices & E is no. of edges.
- **Auxilliary Space: O(V)**

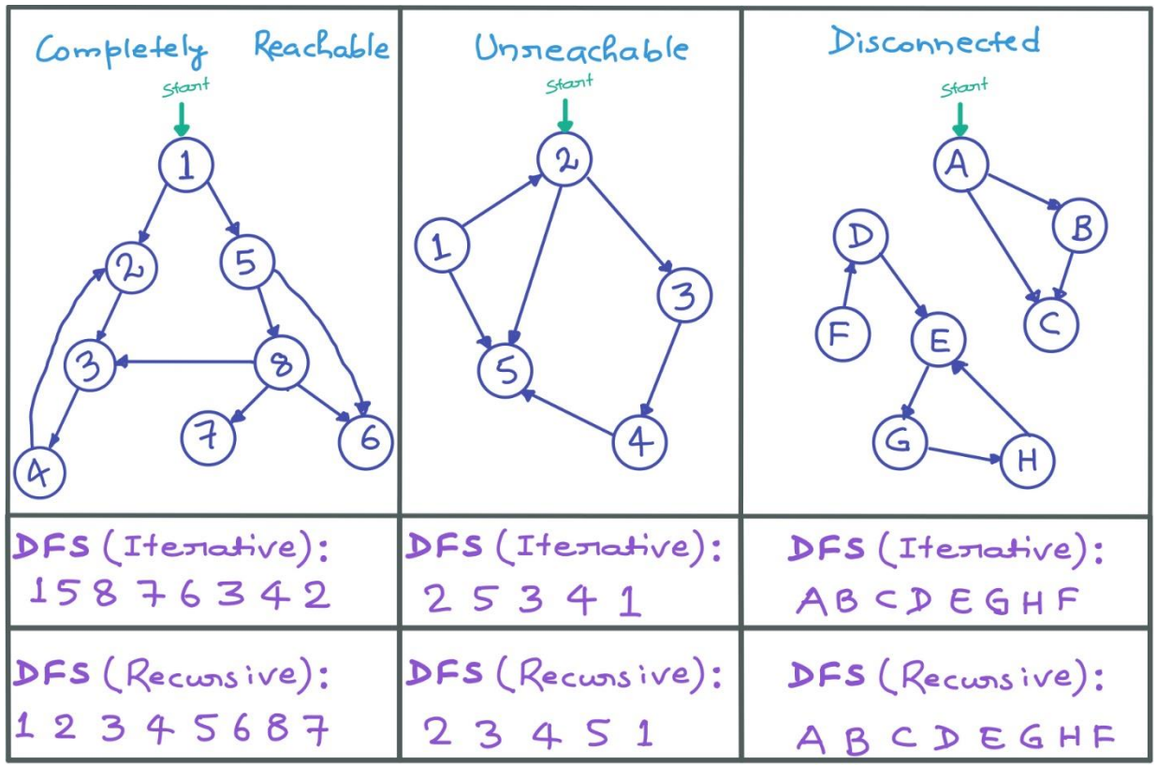
Applications of BFS

- **Shortest Path and Minimum Spanning Tree for unweighted graph**
  - In an unweighted graph, the shortest path is the path with least number of edges.
  - With Breadth First, we always reach a vertex from given source using the minimum number of edges.
  - Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
- **Peer to Peer Networks**
  - In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
- **Crawlers in Search Engines:**
  - Crawlers build index using Breadth First.
  - The idea is to start from source page and follow all links from source and keep doing same.
  - Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.
- **Social Networking Websites:**
  - In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
- **GPS Navigation systems:**
  - Breadth First Search is used to find all neighboring locations.
- **Broadcasting in Network:**
  - In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- **In Garbage Collection:**
  - Breadth First Search is used in copying garbage collection using **Cheney's algorithm**.
  - Breadth First Search is preferred over Depth First Search because of better locality of reference:
- **Cycle detection in undirected graph:**
  - In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle.
  - In directed graph, only depth first search can be used.
- **Ford-Fulkerson Max-Flow Algorithm:**
  - In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow.
  - Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.
- **To test if a graph is Bipartite:**
  - We can either use Breadth First or Depth First Traversal.
- **Path Finding:**
  - We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
- **Finding all nodes within one connected component:**
  - We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Algo-2: Depth First Search (DFS)

What is DFS ?

- DFS for a graph is similar to Depth First Traversal of tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.
- To avoid processing a node more than once, we use a boolean visited array.



Algorithm

1. Mark the given vertex as visited and print it.
2. See all the connected vertices to current_vertex:
    - If any connected_vertex is not visited make the recursive call from that vertex.
3. Check if any unvisited vertex (case of UNREACHABLE or DISCONNECTED graphs):
    - Call the function again from unvisited vertex.

Recursive Implementation

```
class Graph:
    def __init__(self):
        self.graph = {}
```

```
def add_vertex(self, vertex):
    if vertex not in self.graph:
        self.graph[vertex] = []


def add_edge(self, u, v):
    self.add_vertex(u)
    self.add_vertex(v)
    self.graph[u].append(v)


def unvisited(self, visited):
    for vertex in visited:
        if(visited[vertex] is False):
            return vertex
    return None


def dfs_traversal_recursive_util(self, vertex, visited):
    visited[vertex] = True
    print("{}".format(vertex), end=" ")

    # See all the connected vertices to vertex and make recursive call from
    # the vertex not already visited
    for connected_vertex in self.graph[vertex]:
        if(visited[connected_vertex] is False):
            self.dfs_traversal_recursive_util(connected_vertex, visited)

    def dfs_traversal_recursive(self, start_vertex):
         # Mark every every vertex as unvisited
        visited = {}
        for vertex in self.graph:
            visited[vertex] = False

        # Call the dfs_traversal_recursive_util with start_vertex
        self.dfs_traversal_recursive_util(start_vertex, visited)

        # Check if there is still any unvisited vertex
        # Only when graph is UNREACHABLE or DISCONNECTED below lines will be executed
        while(self.unvisited(visited) is not None):
            # Call the dfs_traversal_recursive_util from the unvisited vertex
            self.dfs_traversal_recursive_util(self.unvisited(visited), visited)
        print()


print("Example-1: DFS Traversal of Graph (Recursive) from vertex-1:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(4, 2)
g.add_edge(1, 5)
g.add_edge(5, 6)
```

```
g.add_edge(5, 8)
g.add_edge(8, 3)
g.add_edge(8, 6)
g.add_edge(8, 7)
g.dfs_traversal_recursive(1)

print("Example-2: DFS Traversal of Graph (Recursive) from vertex-2:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 5)
g.add_edge(2, 3)
g.add_edge(2, 5)
g.add_edge(3, 4)
g.add_edge(4, 5)
g.dfs_traversal_recursive(2)

print("Example-3: DFS Traversal of Graph (Recursive) from vertex-A:")
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "C")
g.add_edge("D", "E")
g.add_edge("E", "G")
g.add_edge("F", "D")
g.add_edge("G", "H")
g.add_edge("H", "E")
g.dfs_traversal_recursive("A")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 01:14:54 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 2_2_dfs_recursive.py
Example-1: DFS Traversal of Graph (Recursive) from vertex-1:
1 2 3 4 5 6 8 7
Example-2: DFS Traversal of Graph (Recursive) from vertex-2:
2 3 4 5 1
Example-3: DFS Traversal of Graph (Recursive) from vertex-A:
A B C D E G H F
```

Complexity:

- **Time: O(V+E)** :- V is no. of vertices & E is no. of edges.
- **Auxilliary Space: O(h)** :- h is height or level of graph

Applications of DFS

- **Shortest Path and Minimum Spanning Tree for unweighted graph**
  - For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

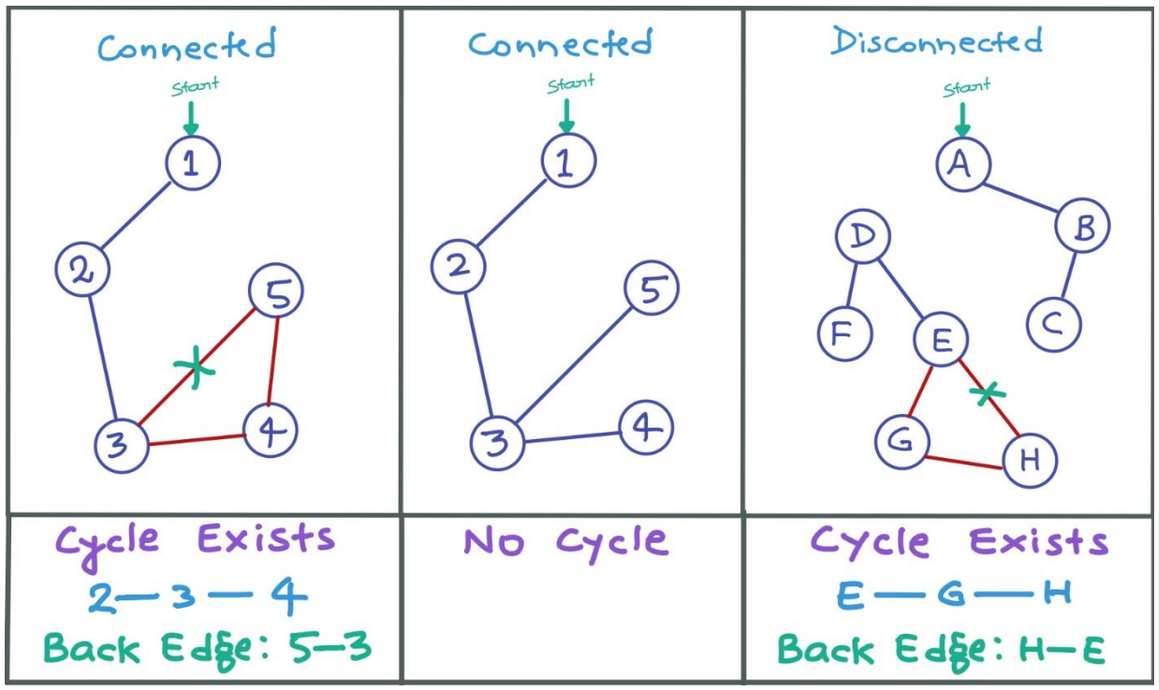
- **Detecting cycle in a graph**
  - A graph has cycle if and only if we see a back edge during DFS.
  - So we can run DFS for the graph and check for back edges.
- **Path Finding**
  - We can specialize the DFS algorithm to find a path between two given vertices u and z.
    - Call DFS(G, u) with u as the start vertex.
    - Use a stack S to keep track of the path between the start vertex and the current vertex.
    - As soon as destination vertex z is encountered, return the path as the contents of the stack
- **Topological Sorting**
  - Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs.
  - In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers..
- **To test if a graph is Bipartite:**
  - Augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link 2 vertices of the same color.
  - The first vertex in any connected component can be red or black.
- **Finding Strongly Connected Components of a graph:**
  - A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
- **Solving puzzles with only one solution** such as mazes.
  - DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.

Standard Graph Algorithms and Problems

1. Detect Cycle in Undirected Graph***

Problem:

Given a undirected graph, check whether the graph contains a cycle or not.



Approach:

- Mark the current_vertex as **visited**.
- See the connected vertices to current_vertex one by one:
    - If connected_vertex is **not visited** then make **recursive call** from the connected_vertex with current_vertex as parent and **return True** if it returns True.
    - Else if connected_vertex is already visited and it is not parent then **cycle found return True**.
- Once the current_vertex is processed **return False** to show that cycle was not found while processing this current_vertex**.**
- Check if any **unvisited** vertex still (case of DISCONNECTED graphs) and **cycle still not found**:
    - Call the function again from that **unvisited** vertex.

Implementation:

```
class Graph:
    def __init__(self):
        self.graph = {}


    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []
```

```
    def add_edge(self, u, v):
        self.add_vertex(u)
        self.add_vertex(v)
        self.graph[u].append(v)
        self.graph[v].append(u)


    def unvisited(self, visited):
        for vertex in visited:
            if(visited[vertex] == False):
                return vertex
        return None


    def detect_cycle_undirected_graph_util(self, current_vertex, parent, visited):
        # Mark the current_vertex as visited
        visited[current_vertex] = True

        # See the connected vertices to current_vertex one by one
        for connected_vertex in self.graph[current_vertex]:
            # If connected_vertex is not visited then make recursive call from the connected_vertex
            # with current_vertex as parent and return True if it returns True.
            if(visited[connected_vertex] == False):
                if(self.detect_cycle_undirected_graph_util(connected_vertex, current_vertex, visited) == True):
                    return True
            # Else if connected_vertex is already visited and it is not parent then cycle found return True.
            elif(connected_vertex != parent):
                print("Cycle Exists coz of {} ----- {}".format(current_vertex, connected_vertex))
                return True

        # Once the current_vertex is processed return False to show that
        # cycle was not found while processing this current_vertex.
        return False


    def detect_cycle_undirected_graph(self, start_vertex):
        # Mark every every vertex as unvisited initially
        visited = {}
        for vertex in self.graph:
            visited[vertex] = False

        # Call the detect_cycle_undirected_graph_util with start_vertex
        cycle_exists = self.detect_cycle_undirected_graph_util(start_vertex, None, visited)

        # Check if there is still any unvisited vertex and cycle still not found
        # Only when graph is DISCONNECTED below lines will be executed
        while(self.unvisited(visited) is not None and cycle_exists == False):
            # Call the detect_cycle_undirected_graph_util from the unvisited vertex
            cycle_exists = self.detect_cycle_undirected_graph_util(self.unvisited(visited), None, visited)

        if(not cycle_exists):
            print("Cycle doesn't exist!")


print("Example-1: Detect Cycle in Undirected Graph:")
g = Graph()
g.add_edge(1, 2)
```

```
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(3, 5)
g.add_edge(4, 5)
g.detect_cycle_undirected_graph(1)

print("\nExample-2: Detect Cycle in Undirected Graph:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(3, 5)
g.detect_cycle_undirected_graph(1)

print("\nExample-3: Detect Cycle in Undirected Graph:")
g = Graph()
g.add_edge("A", "B")
g.add_edge("B", "C")
g.add_edge("D", "E")
g.add_edge("D", "F")
g.add_edge("E", "G")
g.add_edge("E", "H")
g.add_edge("G", "H")
g.detect_cycle_undirected_graph("A")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 16:26:22 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 3_detect_cycle_undirected_graph.py
Example-1: Detect Cycle in Undirected Graph:
Cycle Exists coz of 5 ----- 3

Example-2: Detect Cycle in Undirected Graph:
Cycle doesn't exist!

Example-3: Detect Cycle in Undirected Graph:
Cycle Exists coz of H ----- E
```

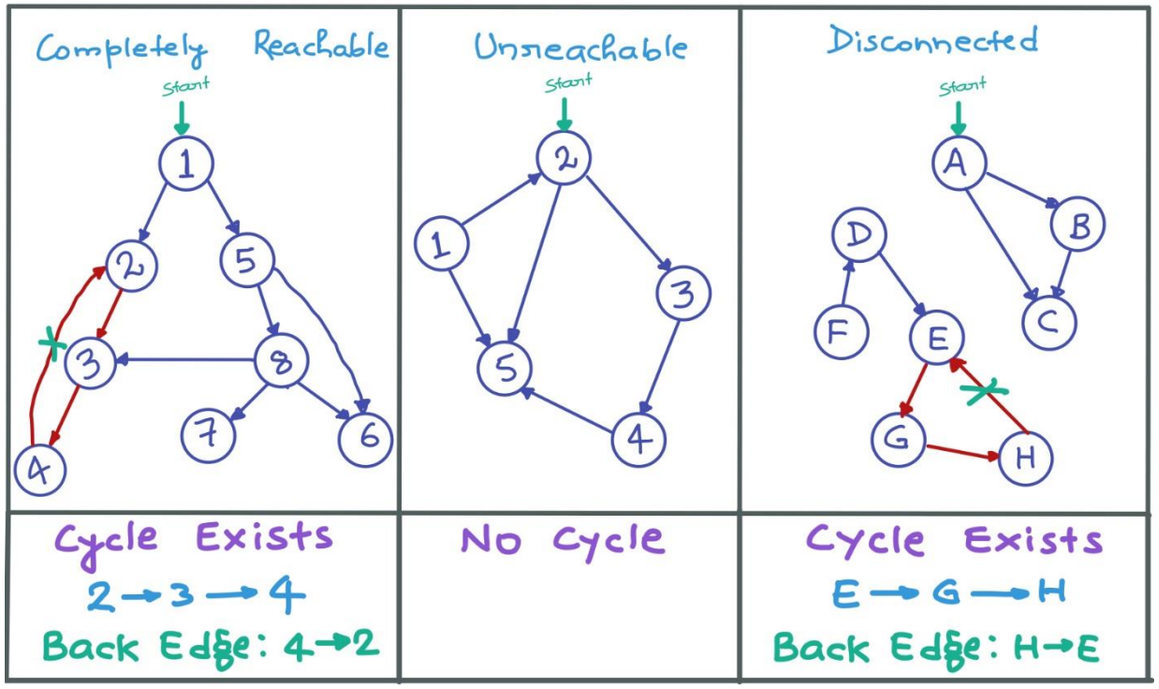
Complexity:

- **Time: O(V+E)** :- As Recursive DFS is used.
- **Auxilliary Space: O(h)**

2. Detect Cycle in Directed Graph***

Problem:

Given a directed graph, check whether the graph contains a cycle or not.



Important Facts:

- Depth First Traversal can be used to detect a cycle in a Graph.
- DFS for a connected graph produces a **tree**.
- There is a cycle in a graph only if there is a **back edge** present in the graph.
- A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS.
- DFS for a disconnected graph produces a **forest**, to detect cycle, we can check for a cycle in individual trees by checking back edges.
- To detect a back edge, we can keep track of vertices currently in DFS stack and if we reach a vertex that is already in the stack, then there is a cycle in the graph.
- The edge that connects current vertex to the vertex in the recursion stack is a back edge.

Approach:

- Mark the current_vertex as **“gray”** i.e. processing.
- See the connected vertices to current_vertex one by one:
    - If connected_vertex is **“white”** i.e. still to be processed then make recursive call from the connected_vertex and **return True** if it is True.

    - Else if connected_vertex is **“gray”** i.e. in processing then we have **found the cycle return True**.
- Once the current_vertex is processed mark it as **“black”** and **return False** to show that cycle was not found while processing this current_vertex.
- Check if any **“white”** vertex still (case of UNREACHABLE or DISCONNECTED graphs) and **cycle still not found**:
    - Call the function again from that **“white”** vertex.

Implementation

```
class Graph:
    def __init__(self):
        self.graph = {}


    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []


    def add_edge(self, u, v):
        self.add_vertex(u)
        self.add_vertex(v)
        self.graph[u].append(v)


    def unvisited(self, visited):
        for vertex in visited:
            if(visited[vertex] == "white"):
                return vertex
        return None


    def detect_cycle_directed_graph_util(self, current_vertex, color):
        # Mark the current_vertex as "gray" i.e. processing
        color[current_vertex] = "gray"

        # See the connected vertices to current_vertex one by one
        for connected_vertex in self.graph[current_vertex]:
            # If connected_vertex is "white" i.e. still to be processed then
            # make recursive call from the connected_vertex
            # and return True if it returns True
            if(color[connected_vertex] == "white"):
                if(self.detect_cycle_directed_graph_util(connected_vertex, color) == True):
                    return True
            # Else if connected_vertex is "gray" i.e. in processing then we have found the cycle return True
            elif(color[connected_vertex] == "gray"):
                print("Cycle Exists coz of {} -----> {}". format(current_vertex, connected_vertex))
                return True

        # If reaches here that means processing of current vertex is done, mark it black
        # return False to show that cycle was not found while processing this current_vertex
        color[current_vertex] = "black"
        return False


    def detect_cycle_directed_graph(self, start_vertex):
```

```
        # Mark every every vertex as "white" i.e. still to be processed
        color = {}
        for vertex in self.graph:
            color[vertex] = "white"

        # Call the detect_cycle_directed_graph_util with start_vertex
        cycle_exists = self.detect_cycle_directed_graph_util(start_vertex, color)

        # Check if there is still any "white" vertex and cycle still not found
        # Only when graph is UNREACHABLE or DISCONNECTED below lines will be executed
        while(self.unvisited(color) is not None and cycle_exists == False):
            # Call the detect_cycle_directed_graph_util from the "white" vertex
            cycle_exists = self.detect_cycle_directed_graph_util(self.unvisited(color), color)

        if(not cycle_exists):
            print("Cycle doesn't exist!")



print("Example-1: Detect Cycle in Directed Graph:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(4, 2)
g.add_edge(1, 5)
g.add_edge(5, 6)
g.add_edge(5, 8)
g.add_edge(8, 3)
g.add_edge(8, 6)
g.add_edge(8, 7)
g.detect_cycle_directed_graph(1)

print("\nExample-2: Detect Cycle in Directed Graph:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 5)
g.add_edge(2, 3)
g.add_edge(2, 5)
g.add_edge(3, 4)
g.add_edge(4, 5)
g.detect_cycle_directed_graph(1)

print("\nExample-3: Detect Cycle in Directed Graph:")
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "C")
g.add_edge("D", "E")
g.add_edge("E", "G")
g.add_edge("F", "D")
g.add_edge("G", "H")
g.add_edge("H", "E")
g.detect_cycle_directed_graph("A")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 14:02:00 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 4_detect_cycle_directed_graph.py
Example-1: Detect Cycle in Directed Graph:
Cycle Exists coz of 4 -----> 2

Example-2: Detect Cycle in Directed Graph:
Cycle doesn't exist!

Example-3: Detect Cycle in Directed Graph:
Cycle Exists coz of H -----> E
```

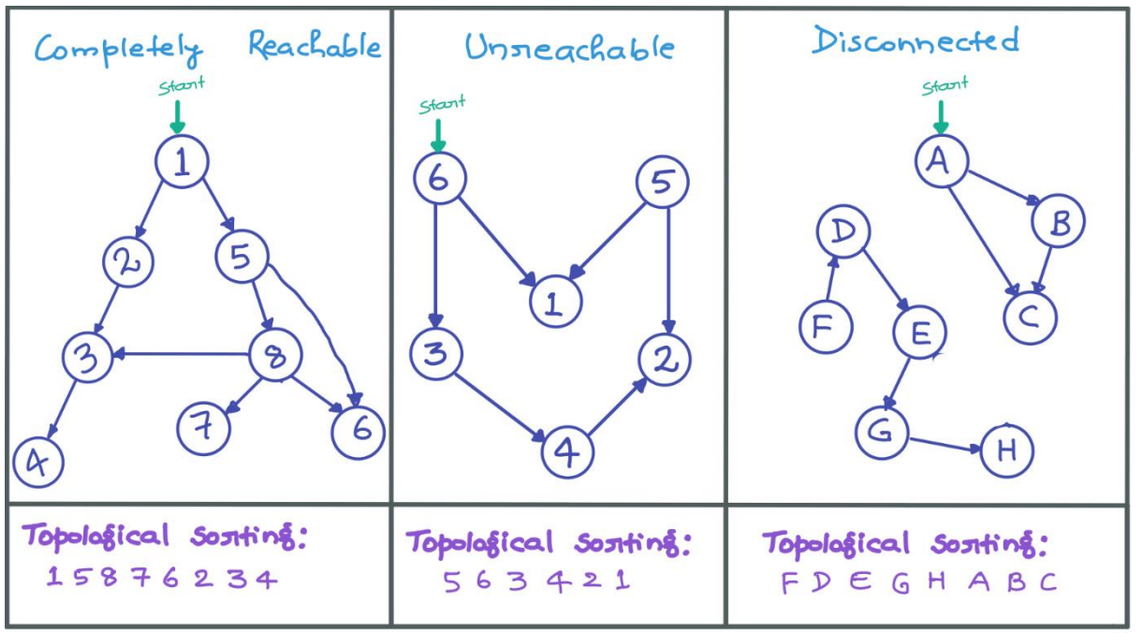
Complexity:

- **Time: O(V+E)** :- As Recursive DFS is used.
- **Auxilliary Space: O(h)**

3. Topological Sorting***

What is Topological Sorting:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering.



Important Facts:

- Topological Sorting for a graph is not possible if the graph is not a DAG.
- There can be more than one topological sorting for a graph. For example:- another topological sorting of the following graph-2: is: “6 5 3 4 2 1”.
- The first vertex in topological sorting is always a vertex with in-degree as 0.
- In DFS, we print a vertex and then recursively call DFS for its adjacent vertices but in topological sorting, we need to print a vertex before its adjacent vertices.

Approach:

1. Mark the given vertex as **visited**.
2. See all the connected vertices to current_vertex one by one:
    - If **connected_vertex** is not already visited then make **recursive call** from the connected_vertex.
3. Once processing of current_vertex is done **push it to result_stack**.
4. Check if any unvisited vertex still (case of UNREACHABLE or DISCONNECTED graphs):
    - Call the function again from unvisited vertex.
5. Print the result_stack in reverse order.

Implementation

```python
class Graph:
    def __init__(self):
        self.graph = {}


    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []


    def add_edge(self, u, v):
        self.add_vertex(u)
        self.add_vertex(v)
        self.graph[u].append(v)


    def unvisited(self, visited):
        for vertex in visited:
            if(visited[vertex] is False):
                return vertex
        return None


    def topological_sort_util(self, current_vertex, visited, result_stack):
        # Mark the current_vertex as visited
        visited[current_vertex] = True

        # See all the connected vertices to current_vertex one by one
        for connected_vertex in self.graph[current_vertex]:
            # If connected_vertex is not already visited then make recursive call from the connected_vertex
            if(visited[connected_vertex] is False):
                self.topological_sort_util(connected_vertex, visited, result_stack)

    # Once processing of current_vertex is done push it to result_stack
    result_stack.append(current_vertex)


def topological_sort(self, start_vertex):
     # Mark every every vertex as unvisited
    visited = {}
    for vertex in self.graph:
        visited[vertex] = False

    # Take a result_stack to store the result
    result_stack = []

    # Call the topological_sort_util with start_vertex
    self.topological_sort_util(start_vertex, visited, result_stack)

    # Check if there is still any unvisited vertex
    # Only when graph is UNREACHABLE or DISCONNECTED below lines will be executed
    while(self.unvisited(visited) is not None):
        # Call the topological_sort_util from the unvisited vertex
        self.topological_sort_util(self.unvisited(visited), visited, result_stack)

    # print the result_stack in reverse order
    result_stack.reverse()
```

```
        for vertex in result_stack:
            print("{} ".format(vertex), end="")
        print()



print("Example-1: Topological Sorting from vertex-1:")
g = Graph()
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(1, 5)
g.add_edge(5, 6)
g.add_edge(5, 8)
g.add_edge(8, 3)
g.add_edge(8, 6)
g.add_edge(8, 7)
g.topological_sort(1)

print("\nExample-2: Topological Sorting from vertex-6:")
g = Graph()
g.add_edge(3, 4)
g.add_edge(4, 2)
g.add_edge(5, 1)
g.add_edge(5, 2)
g.add_edge(6, 1)
g.add_edge(6, 3)
g.topological_sort(6)

print("\nExample-3: Topological Sorting from vertex-A:")
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "C")
g.add_edge("D", "E")
g.add_edge("E", "G")
g.add_edge("F", "D")
g.add_edge("G", "H")
g.topological_sort("A")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 19:22:02 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 5_topological_sorting.py
Example-1: Topological Sorting from vertex-1:
1 5 8 7 6 2 3 4

Example-2: Topological Sorting from vertex-6:
5 6 3 4 2 1

Example-3: Topological Sorting from vertex-A:
F D E G H A B C
```

Complexity:

- **Time: O(V+E)** :- As Recursive DFS is used.
- **Auxilliary Space: O(h)**

Applications of Topological Sorting

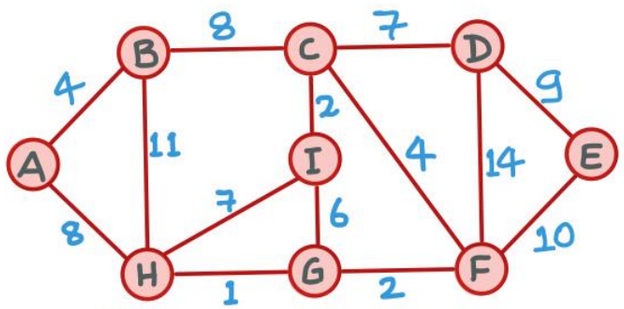
- Topological Sorting is mainly used for **scheduling jobs from the given dependencies among jobs**.
- In computer science, applications of this type arise in:
    - Instruction scheduling
    - Ordering of formula cell evaluation when recomputing formula values in spreadsheets
    - Logic Synthesis
    - Determining the order of compilation tasks to perform in makefiles
    - Data Serialization
    - Resolving symbol dependencies in linkers

4. Prim’s Minimum Spanning Tree (MST) - Greedy***

What is Minimum Spanning Tree?

- Given a **connected and undirected graph**, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A minimum spanning tree (MST) for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Prim's MST



mst_set = {"A": False, "B": False, "C": False, ... , "I": False}

Key = {"A": MAX , "B": MAX , "C": MAX , ... , "I": MAX}

Parent = {"A": -1 , "B": -1 , "C": -1 , ... , "I": -1 }

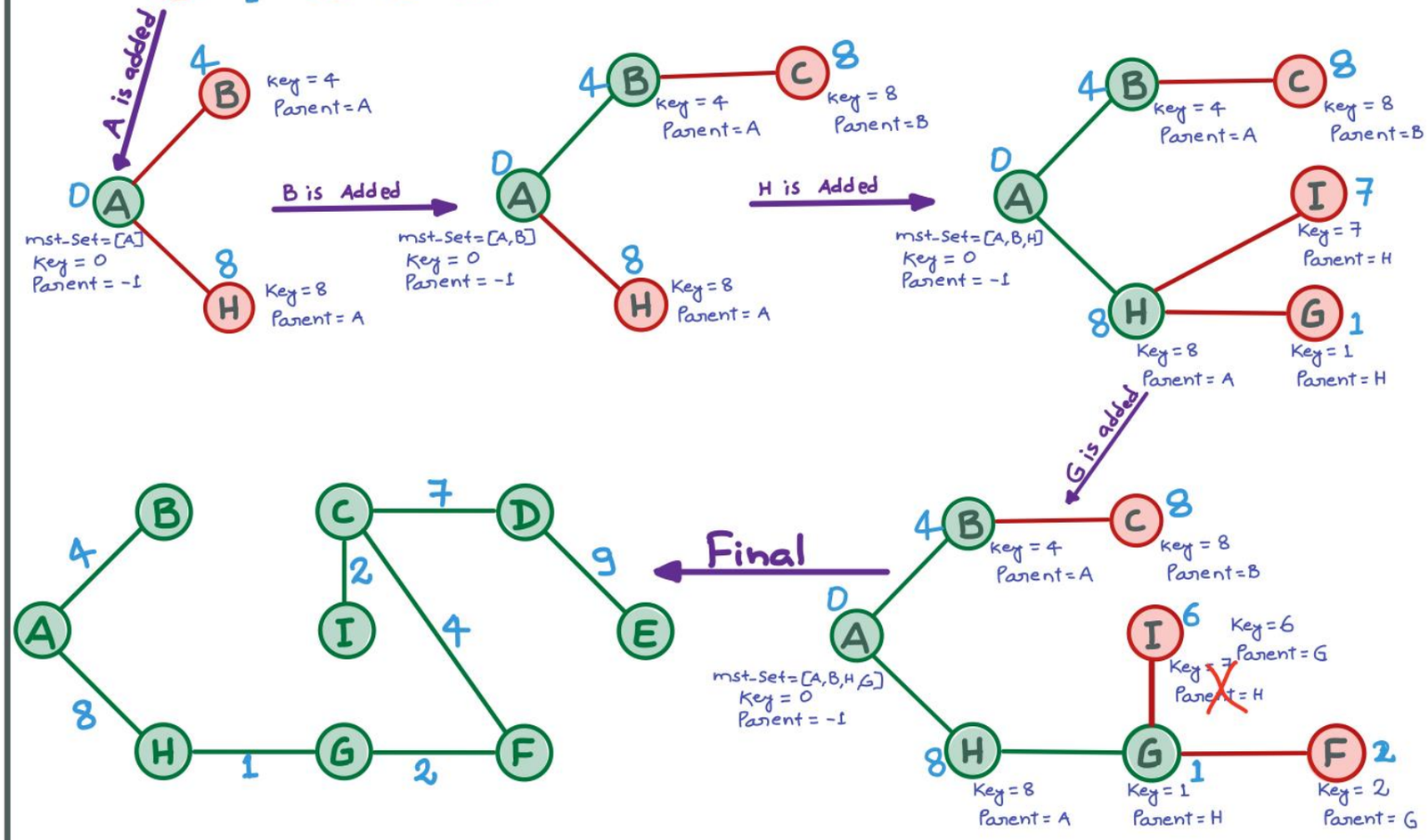
A is added

B is Added

H is Added

G is added

Final



— Astik Anand

Important Facts:

A minimum spanning tree has	V – 1	edges where V is the number of vertices in the given graph.
- **Total spanning trees possible in a graph = $^{|E|}C_{|V-1|}$- total cycles.**
- MST is not possible for disconnected graph.

Approach:

1. Initially, set **mst_set** of every vertex as **False**, **key** of every vertex as **"MAX"** and **parent** of every vertex as **"-"**.
2. Set the **key[start_vertex] = 0** and **min_vertex = start_vertex** :- from this vertex we will start the MST.
3. While min_vertex is not None:
   - Add the min_vertex to the mst_set.
   - See all the connected vertices to min_vertex one by one. If the **connected_vertex is not in mst_set** and also **weight of the connected_vertex < key[connected vertex]**,then update the key of connected vertex and also it's parent to be the min_vertex
   - Again call the min_vertex to **get new min_vertex** with updated mst_set and key.
4. **Print** the edges and their respective weights using **parent** and **key** dicts.

Implementation:

```
import sys

class Graph:
    def __init__(self):
        self.graph = {}


    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []


    def add_edge(self, u, v, w):
        self.add_vertex(u)
        self.add_vertex(v)
        self.graph[u].append((v, w))
        self.graph[v].append((u, w))


    def get_min_key_vertex_not_in_mst(self, mst_set, key):
        min_key = sys.maxsize
        min_vertex = None
```

```
        for k in key:
            if(key[k] < min_key and mst_set[k] == False):
                min_key = key[k]
                min_vertex = k

        return min_vertex



    def prims_mst(self, start_vertex):
        # Initially, set mst_set of every vertex as False,
        # key of every vertex as "MAX" and parent of every vertex as "-".
        mst_set = {}; key = {}; parent = {}
        for vertex in self.graph:
            mst_set[vertex] = False
            key[vertex] = sys.maxsize
            parent[vertex] = "-"

        # Set the key[start_vertex] = 0 and min_vertex = start_vertex :
        # From this vertex we will start the MST
        key[start_vertex] = 0
        min_vertex = start_vertex

        while(min_vertex is not None):
            # Add the min_vertex to the mst_set
            mst_set[min_vertex] = True

            # See all the connected vertices to min_vertex one by one
            for connected_vertex in self.graph[min_vertex]:
                con_vertex = connected_vertex[0]
                con_vertex_weight = connected_vertex[1]
                # If the connected_vertex is not in mst_set and also
                # weight of the connected_vertex < key[connected vertex],
                # then update the key of connected vertex and also it's parent to be the min_vertex
                if(mst_set[con_vertex] == False and con_vertex_weight < key[con_vertex]):
                    key[con_vertex] = con_vertex_weight
                    parent[con_vertex] = min_vertex

            # Again call the function to get new min_vertex with updated mst_set and key
            min_vertex = self.get_min_key_vertex_not_in_mst(mst_set, key)


        # Print the edges and their respective weights using parent and key dicts.
        ordered_parent = sorted(parent.items(), key=lambda k: (k[1]))
        print("="*23 + "\nEdges\t:\tWeights\n" + "="*23)
        for vertex, parent_vertex in ordered_parent:
            print("{} -- {} \t:\t {}".format(parent_vertex, vertex, key[vertex]))


print("Example: Prim's MST from vertex-A:")
g = Graph()
g.add_edge("A", "B", 4)
g.add_edge("A", "H", 8)
g.add_edge("B", "C", 8)
g.add_edge("B", "H", 11)
g.add_edge("C", "D", 7)
```

```
g.add_edge("C", "F", 4)
g.add_edge("C", "I", 2)
g.add_edge("D", "E", 9)
g.add_edge("D", "F", 14)
g.add_edge("E", "F", 10)
g.add_edge("F", "G", 2)
g.add_edge("G", "H", 1)
g.add_edge("G", "I", 6)
g.add_edge("H", "I", 7)
g.prims_mst("A")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 07:19:40 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 6_prims_mst.py
Example: Prim's MST from vertex-A:
========================
Edges   :       Weights
========================
- -- A  :         0
A -- B  :         4
A -- H  :         8
C -- D  :         7
C -- I  :         2
D -- E  :         9
F -- C  :         4
G -- F  :         2
H -- G  :         1
```

Complexity:

- **Time: O(ElogV)** :- Assuming minimum we are getting now can be done using heap in O(logV) for now it's O(V).
- **Auxilliary Space: O(V)**

Applications of MST

- **Network design:** Telephone, Electrical, Hydraulic, TV Cable, Computer, Road
    - The standard application is to a problem like phone network design.
    - We have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities and we want a set of lines that connects all your offices with a minimum total cost.
    - It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

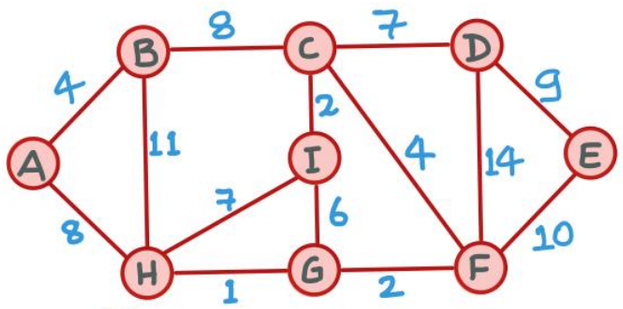
- **Approximation algorithms for NP-hard problems** – Traveling Salesperson Problem, Steiner tree
- **Indirect applications:**
    - Max Bottleneck Paths
    - LDPC Codes for Error Correction
    - Image Registration with Renyi Entropy
    - Learning Salient Features for Real-Time Face Verification
    - Reducing Data Storage in Sequencing Amino acids in a Protein
    - Model locality of particle interactions in turbulent fluid flows
    - Auto-config protocol for Ethernet bridging to avoid cycles in a network
- **Cluster analysis**
    - K Clustering Problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

5. Dijkstra's Shortest Path Algorithm - Greedy***

Problem:

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's Shortest Path Tree (SPT)



spt_set = {"A": False, "B": False, "C": False, ... , "I": False}

Vertex_dist = {"A": MAX , "B": MAX , "C": MAX , ... , "I": MAX}

Parent = {"A": -1 , "B": -1 , "C": -1 , ... , "I": -1 }

A is added

Ver_dist = 4
Parent = A

Ver_dist = 8
Parent = A

Spt_Set = [A]
Ver_dist = 0
Parent = -1

B is Added

Ver_dist = 12
Parent = B

Spt_Set = [A,B]
Ver_dist = 0
Parent = -1

H is Added

Ver_dist = 15
Parent = H

Ver_dist = 9
Parent = H

Spt_Set = [A,B,H]
Ver_dist = 0
Parent = -1

G is added

Ver_dist = 15
Parent = G

Final

Spt_Set = [A,B,H,G]
Ver_dist = 0
Parent = -1

Ver_dist = 11
Parent = G

— Astik Anand

Important Facts:

- Dijkstra's algorithm is **very similar to Prim's algorithm for minimum spanning tree**.
- Like Prim's MST, we generate aSPT (shortest path tree) with given source as root.
- At every step of the algorithm, we find a vertex which is not yet included in **set_set** and has a minimum distance from the source.

Approach:

1. Initially, set **spt_set** of every vertex as **False**, **vertex_distance** of every vertex as **"MAX"** and parent of every vertex as **"-"**.
2. Set the **key[start_vertex] = 0** and **min_vertex = start_vertex** :- from this vertex we will start the MST.
3. Set the **vertex_distance[start_vertex] = 0** and **min_distant_vertex = start_vertex** :- From this vertex we will start the SPT.
4. While min_distant_vertex is not None:
    - Add the min_distant_vertex to the **spt_set**.
    - See all the connected vertices to min_distant_vertex one by one. If the **connected_vertex is not in spt_set** and also **weight of connected_vertex + vertex_distance[min_distant_vertex] < vertex_distance[connected vertex]**, then update the vertex_distance of connected_vertex and also it's parent to be the min_distant_vertex.
    - Again call the function to get new min_distant_vertex with updated spt_set and vertex_distance.
5. **Print** the Vertices and their respective distances from source and Reach by using **parent** and **vertex_distance** dicts.

Implementation

```
import sys

class Graph:
    def __init__(self):
        self.graph = {}


    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []


    def add_edge(self, u, v, w):
        self.add_vertex(u)
        self.add_vertex(v)
        self.graph[u].append((v, w))
        self.graph[v].append((u, w))


    def get_min_distant_vertex_not_in_spt(self, spt_set, vertex_distance):
        min_vertex_distance = sys.maxsize
        min_distant_vertex = None
        for v in vertex_distance:
            if(vertex_distance[v] < min_vertex_distance and spt_set[v] == False):
```

```
                min_vertex_distance = vertex_distance[v]
                min_distant_vertex = v

        return min_distant_vertex



    def dijkstras_spt(self, start_vertex):
        # Initially, set spt_set of every vertex as False,
        # vertex_distance of every vertex as "MAX" and parent of every vertex as "-".
        spt_set = {}; vertex_distance = {}; parent = {}
        for vertex in self.graph:
            spt_set[vertex] = False
            vertex_distance[vertex] = sys.maxsize
            parent[vertex] = "-"

        # Set the vertex_distance[start_vertex] = 0 and min_distant_vertex = start_vertex :-
        # From this vertex we will start the SPT.
        vertex_distance[start_vertex] = 0
        min_distant_vertex = start_vertex

        while(min_distant_vertex is not None):
            # Add the min_distant_vertex to the spt_set
            spt_set[min_distant_vertex] = True

            # See all the connected vertices to min_distant_vertex one by one
            for connected_vertex in self.graph[min_distant_vertex]:
                con_vertex = connected_vertex[0]
                con_vertex_weight = connected_vertex[1]
                # If the connected_vertex is not in spt_set and also weight of connected_vertex +
                # vertex_distance[min_distant_vertex] < vertex_distance[connected vertex],
                # then update the vertex_distance of connected_vertex and also
                # it's parent to be the min_distant_vertex
                if(spt_set[con_vertex] == False and
                    con_vertex_weight + vertex_distance[min_distant_vertex] < vertex_distance[con_vertex]):
                    vertex_distance[con_vertex] = con_vertex_weight + vertex_distance[min_distant_vertex]
                    parent[con_vertex] = min_distant_vertex

            # Again call the function to get new min_distant_vertex with updated spt_set and vertex_distance
            min_distant_vertex = self.get_min_distant_vertex_not_in_spt(spt_set, vertex_distance)


        # Print the edges and their respective weights using parent and vertex_distance dicts.
        ordered_parent = sorted(parent.items(), key=lambda k: (k[0]))
        print("="*56 + "\nVertex\t:\tDistance from source\t:\tReach By\n" + "="*56)
        for vertex, parent_vertex in ordered_parent:
            print("{}\t:\t\t{}\t\t:\t{} --> {}".format(vertex, vertex_distance[vertex], parent_vertex, vertex))



print("Example: Dijkstra's Shortest Path from vertex-A:")
g = Graph()
g.add_edge("A", "B", 4)
g.add_edge("A", "H", 8)
g.add_edge("B", "C", 8)
g.add_edge("B", "H", 11)
g.add_edge("C", "D", 7)
```

```
g.add_edge("C", "F", 4)
g.add_edge("C", "I", 2)
g.add_edge("D", "E", 9)
g.add_edge("D", "F", 14)
g.add_edge("E", "F", 10)
g.add_edge("F", "G", 2)
g.add_edge("G", "H", 1)
g.add_edge("G", "I", 6)
g.add_edge("H", "I", 7)
g.dijkstras_spt("A")
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 07:34:32 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 7_dijkstras_shortest_path.py
Example: Dijkstra's Shortest Path from vertex-A:
============================================================
Vertex  :       Distance from source    :       Reach By
============================================================
A       :                0              :       - --> A
B       :                4              :       A --> B
C       :                12             :       B --> C
D       :                19             :       C --> D
E       :                21             :       F --> E
F       :                11             :       G --> F
G       :                9              :       H --> G
H       :                8              :       A --> H
I       :                14             :       C --> I
```

Complexity:

- **Time: O(ElogV)** :- Assuming minimum we are getting now can be done using heap in O(logV) for now it's O(V).
- **Auxilliary Space: O(V)**

Notes:

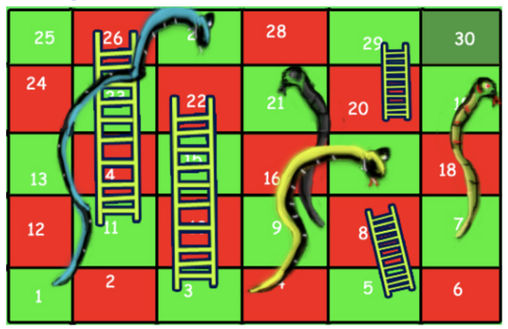
- The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- Dijkstra's algorithm doesn't work for graphs with negative weight edges.
- For graphs with negative weight edges, Bellman–Ford algorithm can be used, we will soon be discussing it as a separate post.

6. Snake and Ladder Problem***

Problem:

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell.Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.

Example:



Consider the board shown, the minimum number of dice throws required to reach cell 30 from cell 1 is 3.

Steps:

- a) First throw two on dice to reach cell number 3 and then ladder to reach 22
- b) Then throw 6 to reach 28.
- c) Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

Approach:

- Initially, mark every cell as **unvisited** and take an **empty queue**.
- Mark the **first_cell as visited** and enqueue the first cell to the queue with **dice_count=0** : We start from here.
- While the queue is not empty:
  - Get the **current** element from the queue and then fetch the **current_cell** and **current_dice_count** from it.
  - If **current_cell is last cell** we are done, **print** the dice_count and **break**.
  - Start seeing all the **6 reachable_cell** from current_cell one by one:
    - If **reachable_cell** is lesser than last cell and still **not visited**:
      - Mark the recahable_cell as **visited**.
      - If any of ladder or snake is available at reachable_cell **modify the reachable_cell** with the moves value.
      - Finally **update** the queue with reachable_cell and dice_count needed to reach there.

Implementation:

```
def get_min_dice_throws_snake_ladder(N, moves):
    # Mark every cell as unvisited and take an empty queue
    visited = [False]*(N)
    queue = []
```

```
    # Mark the first_cell as visited and enqueue the first cell to the queue
    # with dice_count=0 : We start from here.
    visited[0] = True
    queue.append((0, 0))

    while(queue):
        # Get the current element from the queue and then fetch the current_cell
        # and current_dice_count from it.
        current = queue.pop(0)
        current_cell = current[0]
        current_dice_count = current[1]

        # if current_cell is last cell we are done, print the dice_count and break
        if(current_cell == N-1):
            print("No. of min dice count: {}".format(current_dice_count))
            break

        # Start seeing all the 6 reachable_cell from current_cell one by one
        for reachable_cell in range(current_cell+1, current_cell+6):
            # If reachable_cell is lesser than last cell and still not visited
            # then mark the recahable_cell as visited
            if(reachable_cell < N and visited[reachable_cell] is False):
                visited[reachable_cell] = True
                # If any of ladder or snake is available at reachable_cell
                # modify the reachable_cell with the moves value
                if(moves[reachable_cell] != -1):
                    reachable_cell = moves[reachable_cell]
                # Update the queue with reachable_cell and dice_count needed to reach there.
                queue.append((reachable_cell, current_dice_count+1))


print("Example-1:- Snake and Ladder:")
N = 30
moves = [-1] * N

# Ladders
moves[2] = 21
moves[4] = 7
moves[10] = 25
moves[19] = 28

# Snakes
moves[26] = 0
moves[20] = 8
moves[16] = 3
moves[18] = 6
get_min_dice_throws_snake_ladder(N, moves)
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 15:34:59 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 8_snake_and_ladder.py
Example-1:- Snake and Ladder:
No. of min dice count: 3
```

Complexity:

- **Time: O(N)**
- **Auxilliary Space: O(N)**

7. Finding number of Islands

Problem:

Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island.

Example:

Input :

matrix[][] : {1, 1, 0, 0, 0},

{0, 1, 0, 0, 1},

{1, 0, 0, 1, 1},

{0, 0, 0, 0, 0},

{1, 0, 1, 0, 1}

Output : 5 // Total 5 islands.

Connected Component

- A connected component of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph.
- A graph where all vertices are connected with each other has exactly one connected component, consisting of the whole graph.
- Such graph with only one connected component is called as **Strongly Connected Graph**.

Important Facts:

- The problem can be easily **solved by applying DFS() on each component**. In each DFS() call, a component or a sub-graph is visited.
- We will call DFS on the next un-visited component.
- The number of calls to DFS() gives the number of connected components. **BFS can also be used.**
- A cell in 2D matrix can be connected to 8 neighbours.
- So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursively call for 8 neighbours only.
- We keep track of the visited 1s so that they are not visited again.

Approach:

- Initially, mark all the cells of graph or matrix as unvisited.
- Check in matrix if any cell which is 1 and not visited call DFS() on that cell and increase the count by 1.
- The number of times DFS() is called gives that much number of connected components or islands.
- In DFS(I, j, visited):
    - Mark the current_cell as visited.
    - Look for all the 8 possible neighbours.
    - If a neighbour is possible and the neighbour cell is safe then make call to recursive DFS() with the neighbour cell.
- is_safe_cell(x, y, visited):
    - A cell is safe if x is in range of row and y is also in range of col, the cell value is 1 and it is till not visited.

Implementation

```
POSSIBLE_MOVES = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1,1)]

class Graph:
    def __init__(self, matrix):
        self.graph = matrix
        self.row = len(matrix)
        self.col = len(matrix[0])


    def is_safe_cell(self, x, y, visited):
        # Cell is safe if x is in range(row), y is in range(column),
        # cell is still not visited and value of cell is 1.
        return (x>=0 and x<self.row and
                y>=0 and y<self.col and
                not visited[x][y] and self.graph[x][y] ==1)


    # Function to do DFS for a 2D boolean matrix, considers all the 8 neighbours as adjacent vertices.
    def DFS(self, i, j, visited):
        # Mark current_cell as visited
        visited[i][j] = True
```

```
        # For all neighbours check if the cell is safe then call the DFS again from that safe_cell.
        for x_move, y_move in POSSIBLE_MOVES:
            if(self.is_safe_cell(i+x_move, j+y_move, visited)):
                self.DFS(i+x_move, j+y_move, visited)


    def find_number_of_islands(self):
        # Mark all cells as unvisited initially
        visited = [[False]*self.col for i in range(self.row)]

        islands_count = 0
        for i in range(self.row):
            for j in range(self.col):
                # If a cell with value 1 is not visited yet, then new island is found
                # Visit all cells in this island and increment islands_count
                if(self.graph[i][j] == 1 and visited[i][j] is False):
                    self.DFS(i, j, visited)
                    islands_count += 1

        print("Total islands: {}".format(islands_count))



print("Example-1:- Find number of Islands:")
matrix=[[1, 1, 0, 0, 0],
        [0, 1, 0, 0, 1],
        [1, 0, 0, 1, 1],
        [0, 0, 0, 0, 0],
        [1, 0, 1, 0, 1]]
g = Graph(matrix)
g.find_number_of_islands()
```

Output:

```
astik.anand@mac-C02XD95ZJG5H 19:43:20 ~/Personal/Notebooks/Data Structures/8. Graph $ python3 9_find_number_of_islands.py
Example-1:- Find number of Islands:
Total islands: 5
```

Complexity:

- **Time: $O(V^2)$**
- **Auxilliary Space: $O(V^2)$**

Problems To Do:

- Order of Characters in Alien Language
- Hamiltonian Cycle
- Travelling Salesman Problem (TSP)