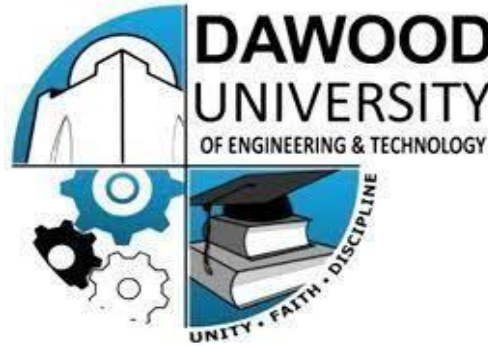


Data Structures & Algorithms

(Practical Manual)



5th Semester, 3rd Year
BATCH -2022

BS ARTIFICIAL INTELLIGENCE

DAWOOD UNIVERSITY OF ENGINEERING & TECHNOLOGY, KARACHI

Dawood University Of Engineering and Technology, Karachi.



CERTIFICATE

This is to certify that Mr./Ms. **MOIZ AHMED MANSOORI** with Roll # **22F-BSAI-32** of Batch 2022 has successfully completed all the labs prescribed for the course “**Data Structures & Algorithms**”.

Engr. Hamza Farooqui
Lecturer
Department of AI

S. No.	Date	Title of Experiment
1	04-Oct-24	Introduction to Programming in Python
2	11-Oct-24	Implementing Stack Data Structure in Python
3	18-Oct-24	Building and Utilizing Queues in Python
4	25-Oct-24	Working with Linked Lists and Node Insertion
5	01-Nov-24	Manipulating Linked Lists: Deletion and Merging
6	08-Nov-24	Exploring Recursion for Problem Solving
7	29-Nov-24	Understanding and Applying Basic Sorting Algorithms
8	06-Dec-24	Applying the Divide-and-Conquer Approach to Sorting
9	13-Dec-24	Utilizing HashMaps for Efficient Data Storage and Retrieval
10	10-Jan-25	Open Ended Lab - 1
11	17-Jan-25 24-Jan-25	Open Ended Lab – 2
12	27-Jan-25	Implementing Binary Search Trees for Efficient Searching

Lab No: 1

Objective:

To get introduced with fundamentals of programming with Python

1) Loops:

In *Python*, *for* and *while* loops follows the following syntax.

WHILE LOOP:-

```
In [10]: a,b=0,1
while b<1000:
|     print (b)
        a,b=b,a+b
```

```
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
```

while loop in Python

FOR LOOP:-

```
In [1]: for i in range(0,10):
        print(i)
        print('Marwa Ashfaq\n')
```

0
Marwa Ashfaq

1
Marwa Ashfaq

2
Marwa Ashfaq

3
Marwa Ashfaq

4
Marwa Ashfaq

5
Marwa Ashfaq

6
Marwa Ashfaq

7
Marwa Ashfaq

8
Marwa Ashfaq

9
Marwa Ashfaq

forloop inPython

2) Conditions:

```
In [2]: a=int(input('please enter a value:'))
        please enter a value:100
```

In [3]: if a<12:

```
        print('value is less than 12')
        else:
            print('value is greater than 12')
```

value is greater than 12

if-else condition in Python

3) Lists:

A list is created by placing all items in “square brackets []”. Elements can be added/appended in a list as well.

```
In [1]: #Defining a list
list=[0,1,2,3]
```

```
In [2]: list
```

```
Out[2]: [0, 1, 2, 3]
```

```
In [3]: #Adding elements in a list
list=list + [4]
```

```
In [4]: list
```

```
Out[4]: [0, 1, 2, 3, 4]
```

```
In [12]: #Appending a List
list.append(5)
```

```
In [11]: list
```

```
Out[11]: [0, 1, 2, 3, 4, 5]
```

listexample

4) User defined Functions:

Functions in *Python* can be created by using the syntax shown below. A function is a block of code which only runs when it is called. Defining and calling a function are explained as follows:

```
In [1]: #Defining Functions
def fib(n):
    a, b = 0,1
    while b<n:
        print(b)
        a,b = b, a+b
```

```
In [2]: def fib2(n):
result=[]
a,b = 0,1
while a<n:
    result.append(a)
    a,b= b, a+b
return result
```

```
In [3]: fib(100)
```

```
1
1
2
3
5
8
13
21
34
55
89
```


```
In [4]: fib2(100)
```

```
Out[4]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

WorkingwithfunctionsinPython

Saving and Importing user-defined function to a program:

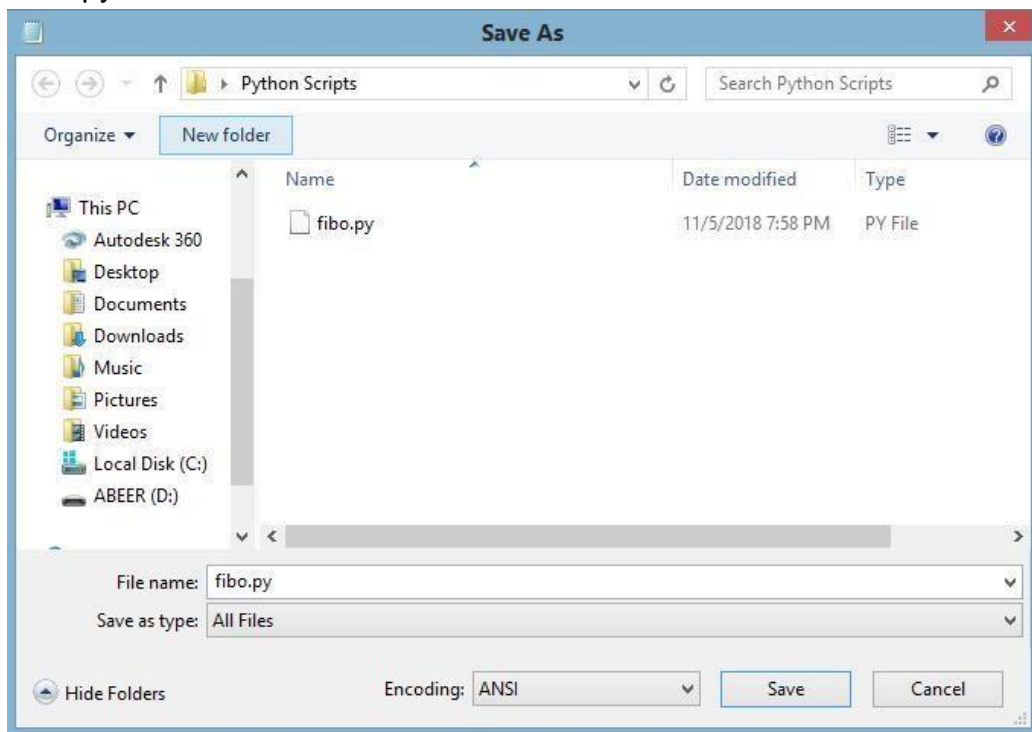
- Copy your desired code in notepad.



The screenshot shows a Notepad window titled "Untitled - Notepad". The menu bar includes File, Edit, Format, View, and Help. The code is as follows:

```
#Defining Functions
def fib(n):
    a, b = 0,1
    while b<n:
        print(b)
        a,b = b, a+b
def fib2(n):
    result=[]
    a,b = 0,1
    while a<n:
        result.append(a)
        a,b = b, a+b
    return result
```

- Save it as .py file.



- Change its extension from.txt to .py.
- Import as follows:

```
In [2]: import fibo
```

```
In [3]: fibo.fib(100)
```

```
1
1
2
3
5
8
13
21
34
55
89
```

```
In [4]: from fibo import fib2
```

```
In [5]: fib2(100)
```

```
Out[5]: [0, 1, 1, 2, 3, 5, 8, 13,
```

```
In [6]: from fibo import fib
```

```
In [7]: fib(100)
```

```
1
1
2
3
5
8
13
21
34
55
89
```

```
In [8]: f=fib2(1000)
```

```
In [9]: f
```

```
Out[9]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

Calling user-defined function in Python

5) Importing libraries to program:

Python library is a collection of functions and methods that allows you to perform lots of actions without writing your own code. For importing libraries, the “import” command is used.

Once the library is imported, its different functions can be called. Following is an example which makes use of a library

```
In [1]: import math
```

```
In [2]: math.sqrt(121)
```

```
Out[2]: 11.0
```

```
In [4]: math.factorial(6)
```

```
Out[4]: 720
```

```
In [5]: math.acos(1)
```

```
Out[5]: 0.0
```

```
In [6]: math.asin(1)
```

```
Out[6]: 1.5707963267948966
```

```
In [8]: math.pi
```

```
Out[8]: 3.141592653589793
```

Making use of libraries in Python

Tasks:

- 1) Write a program which can generate the following

Input a number: 10

10 x 1 = 10

10 x 2 = 20

10 x 3 = 30

10 x 4 = 40

10 x 5 = 50

10 x 6 = 60

10 x 7 = 70

10 x 8 = 80

10 x 9 = 90

10 x 10 = 100

main.py	   Share 	Output
<pre> 1 number = int(input("Enter a number: ")) 2 for multiple in range(1,11): 3 solution = number * multiple 4 print(f"{number} x {multiple} = {solution}") </pre>	<pre> Enter a number: 10 10 x 1 = 10 10 x 2 = 20 10 x 3 = 30 10 x 4 = 40 10 x 5 = 50 10 x 6 = 60 10 x 7 = 70 10 x 8 = 80 10 x 9 = 90 10 x 10 = 100 </pre>	

- 2) Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

>= 0.9 A
 >= 0.8 B
 >= 0.7 C
 >= 0.6 D
 < 0.6 F

Enter score: 0.95

A

Enter score: perfect

Bad score

Enter score: 10.0


Bad score

Enter score: 0.75

C

Enter score: 0.5

F

main.py	   Share	Run	Output
<pre>1 user_input_value = input("Enter a score: ") 2 3 try: 4 numeric_score = float(user_input_value) 5 6 if 0.0 <= numeric_score <= 1.0: 7 if numeric_score >= 0.9: 8 final_grade = "A" 9 elif numeric_score >= 0.8: 10 final_grade = "B" 11 elif numeric_score >= 0.7: 12 final_grade = "C" 13 elif numeric_score >= 0.6: 14 final_grade = "D" 15 else: 16 final_grade = "F" 17 18 print(f"Grade: {final_grade}") 19 else: 20 print("Bad Score") 21 22 except ValueError: 23 print("Invalid input, please enter a numeric value") 24</pre>	<div data-bbox="1079 220 1575 283">^ Enter a score: 1 Grade: A</div> <div data-bbox="1079 315 1575 346">=== Code Execution Successful ===</div>		

- 3) Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to target*. You may assume that each input would have **exactly one solution**, and you may not use the *same* element twice. You can return the answer in any order.

Example 1:

Input: `nums = [2,7,11,15]`, `target = 9`

Output: `[0,1]`

Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`

Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`

Output: `[0,1]`

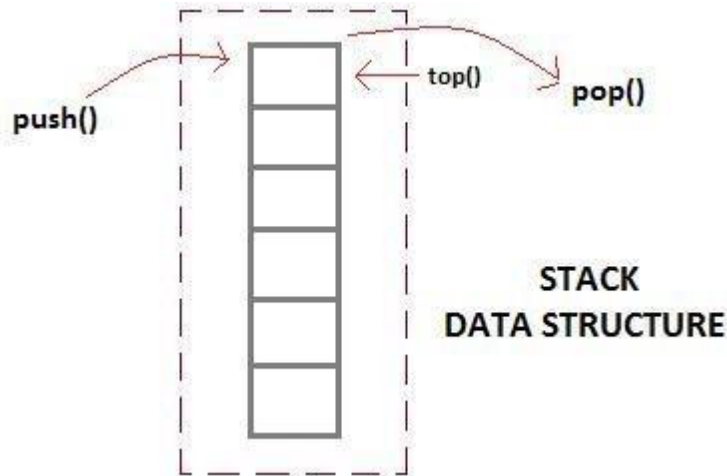
main.py	Run	Output
<pre>1 def two_sum(nums, target): 2 seen = {} 3 for i, num in enumerate(nums): 4 complement = target - num 5 if complement in seen: 6 return [seen[complement], i] 7 seen[num] = i 8 9 print(two_sum([2, 7, 11, 15], 9)) 10 print(two_sum([3, 2, 4], 6)) 11 print(two_sum([3, 3], 6)) 12</pre>		<pre>[0, 1] [1, 2] [0, 1] === Code Execution Successful ===</pre>

Lab No: 2

Objective: Implementing Stack Data Structure in Python

Theory:

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. `push()` function is used to insert new elements into the Stack and `pop()` function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

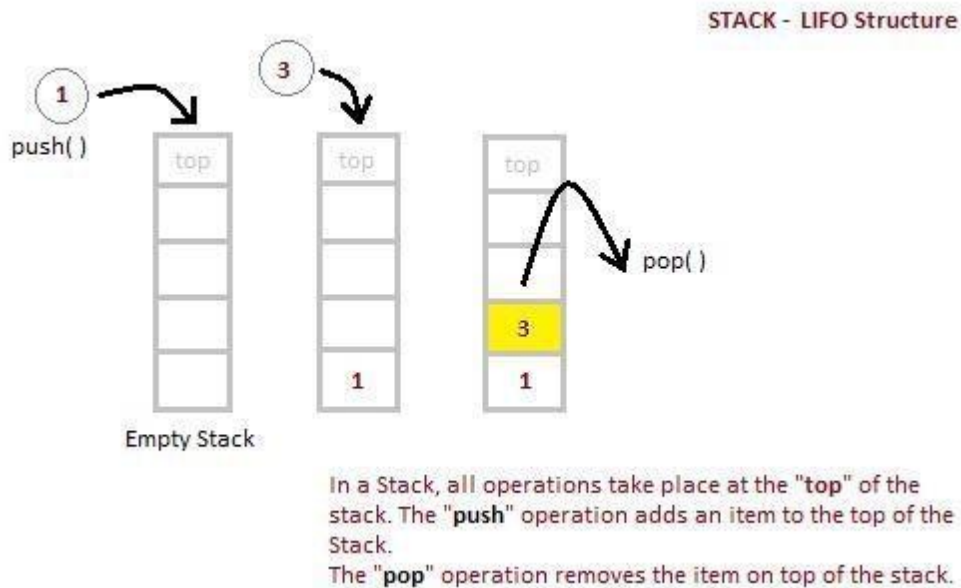
Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack. There are other uses also like:

1. Parsing
2. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



Algorithm for PUSH operation

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : $O(1)$
- **Pop Operation** : $O(1)$
- **Top Operation** : $O(1)$
- **Search Operation** : $O(n)$

The time complexities for `push()` and `pop()` functions are $O(1)$ because we always have to insert or remove the data from the **top** of the stack, which is a one step process.

CODE:

```

# Stack implementation in python

# Creating a stack
def create_stack():
    stack = []
    return stack

# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0

# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)

# Removing an element from the stack
def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"

    return stack.pop()

stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " +
      str(stack))

```

Tasks:

- 1) Execute the above code and observe its output
- 2) Given a string *s* containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:
 1. Open brackets must be closed by the same type of brackets.
 2. Open brackets must be closed in the correct order.
 3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"

Output: true

Example 2:

Input: s = "()[]{}"

Output: true

Example 3:

Input: s = "["

Output: false

Example 4:

Input: s = "([])"

Output: true

LAB 02



```
def isValid(s):
    stack = []
    bracket_map = {'(': ')', '[': ']', '{': '}'
    for char in s:
        if char in bracket_map:
            if stack and stack[-1] == bracket_map[char]:
                stack.pop()
            else:
                return False
        else:
            stack.append(char)
    return not stack

print(isValid("()"))
print(isValid "()[]{}"))
print(isValid "["))
print(isValid "([])"))
```

[2] ✓ 0.0s

... True
True
False
True

Lab No: 3

Objective: Building and Utilizing Queues in Python

Theory:

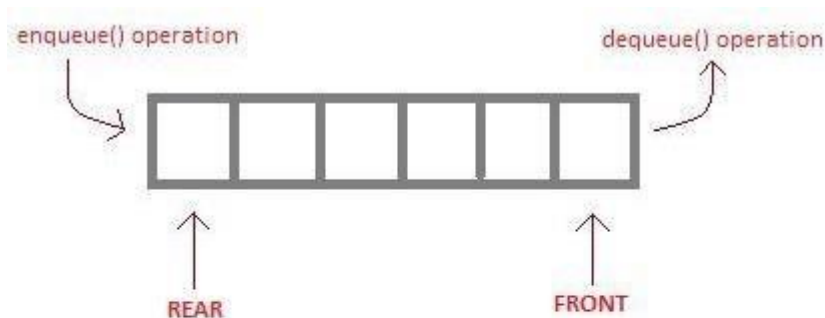
What is a Queue Data Structure?

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



`enqueue()` is the operation for adding an element into Queue.

`dequeue()` is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

`peek ()` 4.function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

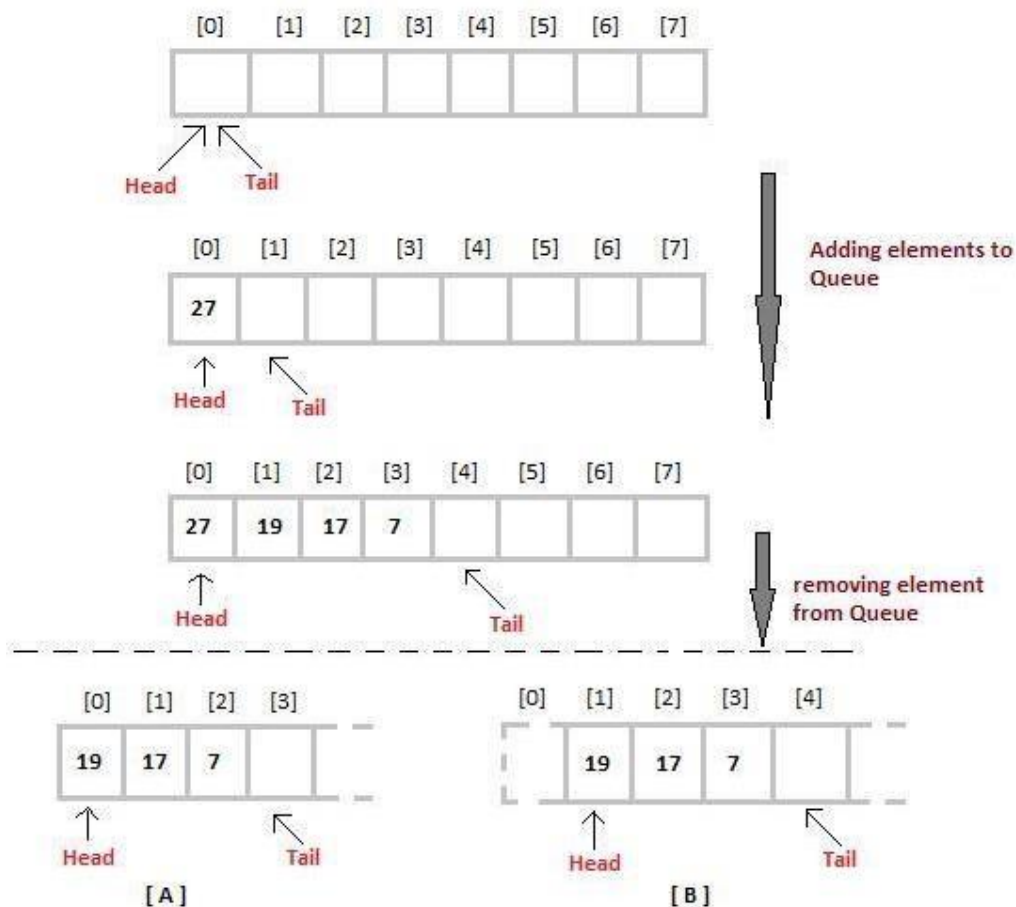
1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

CODE:

```
# Queue implementation in Python
```

```
class Queue:
```

```
    def __init__(self):
```

```
        self.queue = []
```

```
    # Add an element
```

```
    def enqueue(self, item):
```

```
        self.queue.append(item)
```

```
    # Remove an element
```

```
    def dequeue(self):
```

```
        if len(self.queue) < 1:
```

```
            return None
```

```
        return self.queue.pop(0)
```

```
    # Display the queue
```

```
    def display(self):
```

```
        print(self.queue)
```

```
    def size(self):
```

```
        return len(self.queue)
```

```
q = Queue()
```

```
q.enqueue(1)
```

```
q.enqueue(2)
```

```
q.enqueue(3)
```

```
q.enqueue(4)
```

```
q.enqueue(5)
```

```
q.display()
```

```
q.dequeue()
```

```
print("After removing an  
element") q.display()
```

Tasks:

- 1) Execute the above code and observe its output.
- 2) There are n people in a line queuing to buy tickets, where the 0th person is at the **front** of the line and the (n - 1)th person is at the **back** of the line.
You are given a **0-indexed** integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes **exactly 1 second** to buy a ticket. A person can only buy **1 ticket at a time** and has to go back to **the end** of the line (which happens **instantaneously**) in order to buy more tickets. If a person does not have any tickets left to buy, the person will **leave** the line.

Return the **time taken** for the person **initially** at position **k** (0-indexed) to finish buying tickets.

Example 1:

Input: tickets = [2,3,2], k = 2

Output: 6

Explanation:

- The queue starts as [2,3,2], where the kth person is underlined.
- After the person at the front has bought a ticket, the queue becomes [3,2,1] at 1 second.
- Continuing this process, the queue becomes [2,1,2] at 2 seconds.
- Continuing this process, the queue becomes [1,2,1] at 3 seconds.
- Continuing this process, the queue becomes [2,1] at 4 seconds. Note: the person at the front left the queue.
- Continuing this process, the queue becomes [1,1] at 5 seconds.
- Continuing this process, the queue becomes [1] at 6 seconds. The kth person has bought all their tickets, so return 6.

Example 2:

Input: tickets = [5,1,1,1], k = 0

Output: 8

Explanation:

- The queue starts as [5,1,1,1], where the kth person is underlined.
- After the person at the front has bought a ticket, the queue becomes [1,1,1,4] at 1 second.
- Continuing this process for 3 seconds, the queue becomes [4] at 4 seconds.
- Continuing this process for 4 seconds, the queue becomes [] at 8 seconds. The kth person has bought all their tickets, so return 8.

LAB 03



```
def timeRequiredToBuy(tickets, k):  
    time = 0  
    for i in range(len(tickets)):  
        if i <= k:  
            time += min(tickets[i], tickets[k])  
        else:  
            time += min(tickets[i], tickets[k] - 1)  
    return time  
  
print(timeRequiredToBuy([2, 3, 2], 2))  
print(timeRequiredToBuy([5, 1, 1, 1], 0))
```

[7] ✓ 0.0s

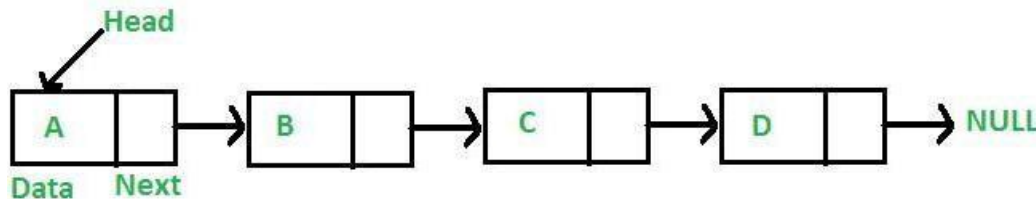
... 6
8

Lab No: 4

Objective: Working with Linked Lists and Node Insertion

Theory:

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

- 1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- 2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system if we maintain a sorted list of IDs in an array `id[]`. `id[]`

`= [1000, 1010, 1050, 2000, 2040]`.

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved.

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. Read about it here.
- 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:

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 (Or Reference) to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

Linked list implementation in Python

```
class Node: #
    Creating a node
    def __init__(self, item):
        self.item = item
        self.next = None

class LinkedList:

    def __init__(self):
        self.head = None

if __name__ == '__main__':

    linked_list = LinkedList()

    # Assign item values
    linked_list.head =
    Node(1) second =
    Node(2) third = Node(3)

    # Connect nodes
    linked_list.head.next = second
    second.next = third

    # Print the linked list item
    while linked_list.head !=
    None:
        print(linked_list.head.item, end=" ")
        linked_list.head =
        linked_list.head.next
```

Tasks:

- 1) Implement LinkedList Data Structure in Python.
- 2) Insert a node at Head, and End of the LinkedList
- 3) Insert a new node in between two nodes passing the index where the new node is to be inserted.


```

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

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

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    def insert_at_head(self, data):
        """ Insert a node at the beginning of the LinkedList """
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        """ Insert a node at the end of the LinkedList """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def insert_at_index(self, index, data):
        """ Insert a new node at a given index """
        if index < 0:
            print("Invalid index")
            return

        new_node = Node(data)

        if index == 0:
            new_node.next = self.head
            self.head = new_node
            return

        current = self.head
        position = 0

        while current and position < index - 1:
            current = current.next
            position += 1

        if current is None:
            print("Index out of range")
            return

        new_node.next = current.next
        current.next = new_node

ll = LinkedList()

ll.insert_at_head(3)
ll.insert_at_end(4)

ll.insert_at_index(2, 5)
ll.display()

```

Output:

```
3 -> 4 -> 5 -> None
```

Lab No: 5

Objective: Manipulating Linked Lists: Deletion and Merging

Deletion in Linked List

Deleting a node in a Linked List is an important operation and can be done in three main ways: removing the first node, removing a node in the middle, or removing the last node.

1. Deletion at the Beginning of Linked List

Deletion at the Beginning operation involves removing the first node of the linked list.

*To perform the deletion at the beginning of Linked List, we need to change the **head** pointer to point to the second node. If the list is empty, there's nothing to delete.*

Time Complexity: $O(1)$

Auxiliary Space: $O(1)$

2. Deletion at Specific Position of Linked List

Deletion at a specified position in a linked list involves removing a node from a specific index/position, which can be the first, middle, or last node.

*To perform the deletion, If the position is 1, we update the **head** to point to the **next node** and delete the current head. For other positions, we traverse the list to reach the node just before the specified **position**. If the target node exists, we adjust the next of this previous node to point to next of next nodes, which will result in skipping the target node.*

Time Complexity: $O(n)$, traversal of the linked list till its end, so the time complexity required is $O(n)$.

Auxiliary Space: $O(1)$

3. Deletion at the End of Linked List

Deletion at the end operation involves removing the last node of the linked list.

*To perform the deletion at the end of Linked List, we need to traverse the list to find the **second last node**, then set its next pointer to **null**. If the list is empty then there is no node to delete or has only one node then point **head** to **null**.*

Time Complexity: $O(n)$, traversal of the linked list till its end, so the time complexity required is $O(n)$.

Auxiliary Space: $O(1)$

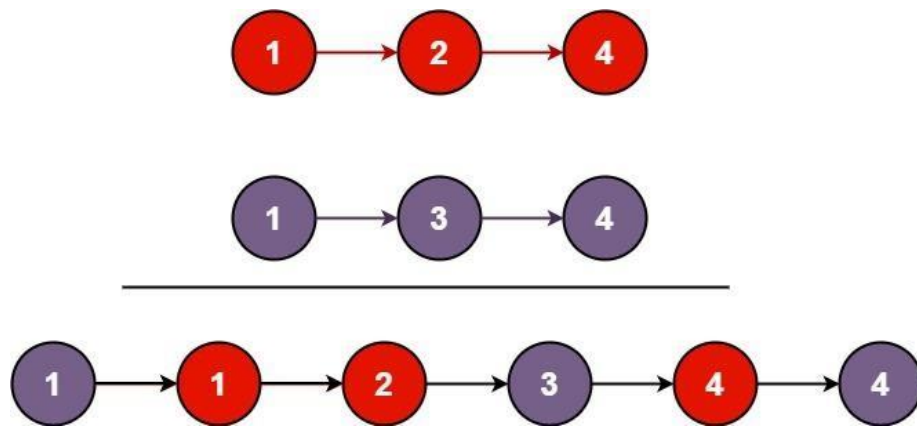
Tasks:

- 1) Implement Deletion of a node from LinkedList using the three ways explained in python.
- 2) You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

```

class Node:
    def __init__(self, value):
        self.value = value
        self.next_node = None

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

    def add_node(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
            return
        current_node = self.head
        while current_node.next_node:
            current_node = current_node.next_node
        current_node.next_node = new_node

    def display_list(self):
        current_node = self.head
        while current_node:
            print(current_node.value, end=" -> ")
            current_node = current_node.next_node
        print("None")

    def remove_node(self, target_value):
        if not self.head:
            return
        if self.head.value == target_value:
            self.head = self.head.next_node
            return
        current_node = self.head
        while current_node.next_node:
            if current_node.next_node.value == target_value:
                current_node.next_node = current_node.next_node.next_node
                return
            current_node = current_node.next_node

```

```

def merge_sorted(list1, list2):
    dummy_node = Node(0)
    merged_tail = dummy_node

    while list1 and list2:
        if list1.value < list2.value:
            merged_tail.next_node = list1
            list1 = list1.next_node
        else:
            merged_tail.next_node = list2
            list2 = list2.next_node
        merged_tail = merged_tail.next_node

    merged_tail.next_node = list1 if list1 else list2
    return dummy_node.next_node

if __name__ == "__main__":
    list1 = LinkedList()
    for num in [1, 2, 3, 4]:
        list1.add_node(num)

    print("Original List:")
    list1.display_list()

    list1.remove_node(3)
    print("After Removing 3:")
    list1.display_list()

    list2 = LinkedList()
    for num in [5, 6]:
        list2.add_node(num)

    merged_list = merge_sorted(list1.head, list2.head)
    print("Merged Sorted List:")
    current_node = merged_list
    while current_node:
        print(current_node.value, end=" -> ")
        current_node = current_node.next_node
    print("None")

```

Output:

```

[27] ✓ 0.0s

... Original List:
1 -> 2 -> 3 -> 4 -> None
After Removing 3:
1 -> 2 -> 4 -> None
Merged Sorted List:
1 -> 2 -> 4 -> 5 -> 6 -> None

```

Lab No: 6

Objective: Exploring Recursion for Problem Solving

Theory:

Recursion and iteration both repeatedly executes the set of instructions. Recursion is when a statement in a function calls itself repeatedly. The iteration is when a loop repeatedly executes until the controlling condition becomes false. The primary difference between recursion and iteration is that a recursion is a process, always applied to a function. The iteration is applied to the set of instructions which we want to get repeatedly executed.

Need of Recursion:

- Recursion helps in logic building. Recursive thinking helps in solving complex problems by breaking them into smaller subproblems.
- Recursive solutions work as a basis for Dynamic Programming and Divide and Conquer algorithms.
- Certain problems can be solved quite easily using recursion like Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Steps

Step1 – Define a base case: Identify the simplest (or base) case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 – Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 – Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

Step4 – Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

Example 1 : Sum of Natural Numbers

Let us consider a problem to find the sum of natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 0 to n. So the function simply looks like this,

approach(1) – Simply adding one by one

$$f(n) = 0 + 1 + 2 + 3 + \dots + n$$

but there is another mathematical approach of representing this,

approach(2) – Recursive adding

$$f(n) = 0 \quad n=0$$

$$f(n) = n + f(n-1) \quad n \geq 1$$

Recursive function to find the sum of

numbers from 0 to n

def findSum(n):

 # Base case

 if n == 0:

 return 0

```
# Recursive case
return n + findSum(n - 1)

n = 5
print(findSum(n))
```

Output

15

What is the base condition in recursion?

A recursive program stops at a base condition. There can be more than one base conditions in a recursion. In the above program, the base condition is when $n = 1$.

How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion.

Conclusion:

The recursive function is easy to write, but they do not perform well as compared to iteration whereas, the iteration is hard to write but their performance is good as compared to recursion.

Tasks:

- 1) Implement python code for factorial of a number.
- 2) Given an integer n , return *true if it is a power of two. Otherwise, return false*. An integer n is a power of two, if there exists an integer x such that $n == 2^x$.

Example 1:

Input: $n = 1$

Output: true

Explanation: $2^0 = 1$

Example 2:

Input: $n = 16$

Output: true

Explanation: $2^4 = 16$

Example 3:

Input: $n = 3$

Output: false

- 3) Given an integer n , return *true if it is a power of three. Otherwise, return false*. An integer n is a power of three, if there exists an integer x such that $n == 3^x$.

Example 1:

Input: $n = 27$

Output: true

Explanation: $27 = 3^3$

Example 2:

Input: $n = 0$

Output: false

Explanation: There is no x where $3^x = 0$.

Example 3:

Input: $n = -1$

Output: false

Explanation: There is no x where $3^x = (-1)$.

- 4) Given an integer n , return *true* if it is a power of four. Otherwise, return *false*. An integer n is a power of four, if there exists an integer x such that $n == 4^x$.

Example 1:

Input: $n = 16$

Output: true

Example 2:

Input: $n = 5$

Output: false

Example 3:

Input: $n = 1$

Output: true

```

# 1) Factorial of a Number
def factorial(n):
    if n < 0:
        return "Factorial not defined for negative numbers"
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

print("Factorial: ",factorial(5))
print("=====")

# 2) Check if a Number is a Power of Two
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0

print("Power of two: ",is_power_of_two(1))
print("Power of two: ",is_power_of_two(16))
print("Power of two: ",is_power_of_two(3))
print("=====")

# 3) Check if a Number is a Power of Three
def is_power_of_three(n):
    if n <= 0:
        return False
    while n % 3 == 0:
        n //= 3
    return n == 1

print("Power of three: ",is_power_of_three(27))
print("Power of three: ",is_power_of_three(8))
print("Power of three: ",is_power_of_three(-1))
print("=====")

# 4) Check if a Number is a Power of Four
def is_power_of_four(n):
    return n > 0 and (n & (n - 1)) == 0 and (n - 1) % 3 == 0

print("Power of Four: ",is_power_of_four(16))
print("Power of Four: ",is_power_of_four(5))
print("Power of Four: ",is_power_of_four(1))

```

3]

✓ 0.0s

Output

```

... 0.0s
Factorial: 120
=====
Power of two: True
Power of two: True
Power of two: False
=====
Power of three: True
Power of three: False
Power of three: False
=====
Power of Four: True
Power of Four: False
Power of Four: True

```


Lab No: 7

Objective: Understanding and Applying Basic Sorting Algorithms

Bubble Sort Algorithm

Theory:

Bubble sort is a simple and well-known sorting algorithm. It is used in practice once in a blue moon and its main application is to make an introduction to the sorting algorithms. Bubble sort belongs to $O(n^2)$ sorting algorithms, which makes it quite inefficient for sorting large data volumes. Bubble sort is **stable** and **adaptive**.

Algorithm:

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

You can imagine that on every step big bubbles float to the surface and stay there. At the step, when no bubble moves, sorting stops. Let us see an example of sorting an array to make the idea of bubble sort clearer.

Example. Sort {5, 1, 12, -5, 16} using bubble sort.

5	1	12	-5	16	unsorted
---	---	----	----	----	----------

Pass 1:

5	1	12	-5	16	5 > 1, swap
1	5	12	-5	16	5 < 12, ok
1	5	12	-5	16	12 > -5, swap
1	5	-5	12	16	12 < 16, ok

Pass 2:



1 < 5, ok

5 > -5, swap

5 < 12, ok

Pass 3:



1 > -5, swap

1 < 5, ok

Pass 4:



-5 < 1, ok

Pass 5:



sorted

Complexity analysis:

Average and worst case complexity of bubble sort is $O(n^2)$. Also, it makes $O(n^2)$ swaps in the worst case. Bubble sort is adaptive. It means that for almost sorted array it gives $O(n)$ estimation. Avoid implementations, which don't check if the array is already sorted on every step (any swaps made). This check is necessary, in order to preserve adaptive property.

Selection Sort Algorithm:

Theory:

Selection sort is the algorithms of the slower sort type, i.e they have a complexity of $O(n^2)$.

The selection sort algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

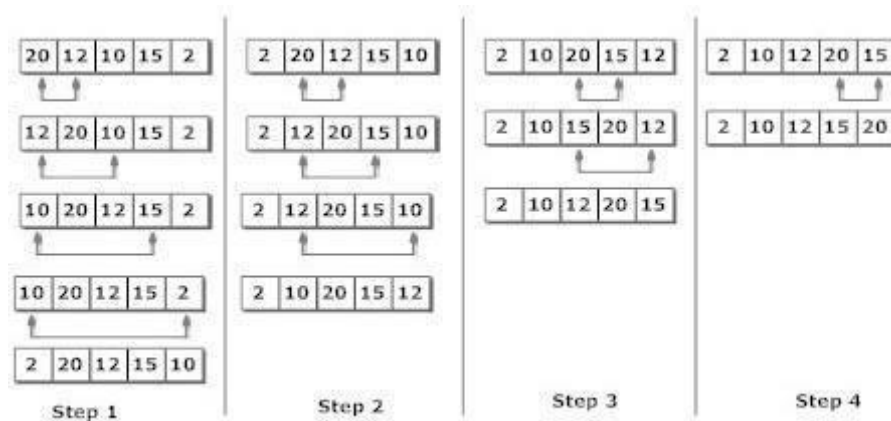


Figure: Selection Sort

Following are the steps involved in selection sort (for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

Complexity Analysis of Selection Sort

Selection Sort requires two nested for loops to complete itself, one for loop is in the function selection Sort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second(inner) for loop.

Hence for a given input size of n , following will be the time and space complexity for selection sort algorithm:

Worst Case Time Complexity [Big-O]: $O(n^2)$

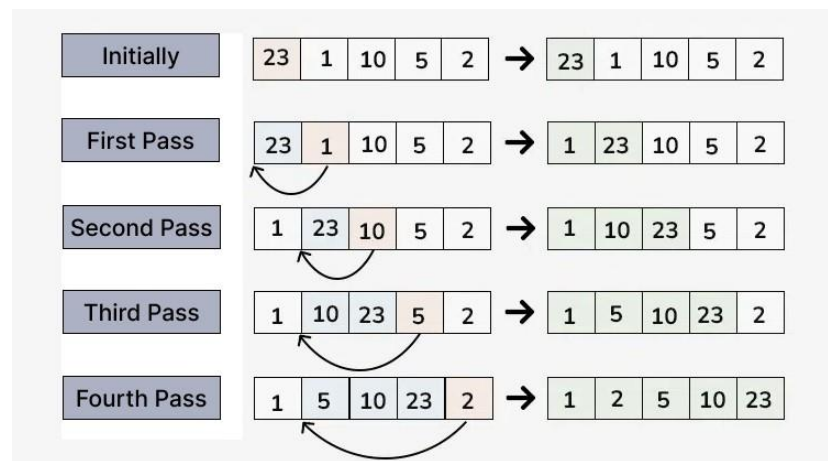
Best Case Time Complexity [Big-omega]: $O(n^2)$

Average Time Complexity [Big-theta]: $O(n^2)$

Insertion Sort Algorithm:

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.



Time Complexity of Insertion Sort:

Best case: $O(n)$, If the list is already sorted, where n is the number of elements in the list.

Average case: $O(n)$, If the list is randomly ordered

Worst case: $O(n^2)$, If the list is in reverse order

Tasks:

- 1) Develop Python programs for Bubble Sort, Selection Sort, and Insertion Sort.
- 2) Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1:

Input: nums = [1,2,3,1]

Output: true

Explanation: The element 1 occurs at the indices 0 and 3.

Example 2:

Input: nums = [1,2,3,4]

Output: false

Explanation: All elements are distinct.

Example 3:

Input: nums = [1,1,1,3,3,4,3,2,4,2]

Output: true

- 3) Given an array nums containing n distinct numbers in the range [0, n], return *the only number in the range that is missing from the array*.

Example 1:

Input: nums = [3,0,1]

Output: 2

Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

Example 2:

Input: nums = [0,1]

Output: 2

Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

Example 3:

Input: nums = [9,6,4,2,3,5,7,0,1]

Output: 8

Explanation: n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.

```

# 1) Sorting Algorithms
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

print("Bubble Sort: ",bubble_sort([64, 34, 25, 12, 22, 11, 90]))
print("=====")

# Selection Sort
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

# Example usage
print("Selection Sort: ",selection_sort([64, 25, 12, 22, 11]))
print("=====")

# Insertion Sort
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Example usage
print("Insertion Sort: ",insertion_sort([12, 11, 13, 5, 6]))
print("=====")

```

```

# 2) Check for Duplicates in an Array

def contains_duplicate(nums):
    return len(nums) != len(set(nums))

print("Duplicate:")
print(contains_duplicate([1, 2, 3, 1]))
print(contains_duplicate([1, 2, 3, 4]))
print(contains_duplicate([1, 1, 1, 3, 3, 4, 3, 2, 4, 2]))
print("=====")

# 3) Find the Missing Number in an Array
def missing_number(nums):
    n = len(nums)
    expected_sum = n * (n + 1) // 2
    actual_sum = sum(nums)
    return expected_sum - actual_sum

print("Missing numbers:")
print(missing_number([3, 0, 1]))
print(missing_number([0, 1]))
print(missing_number([9, 6, 4, 2, 3, 5, 7, 0, 1]))

```

```

... Bubble Sort: [11, 12, 22, 25, 34, 64, 90]
=====
Selection Sort: [11, 12, 22, 25, 64]
=====
Insertion Sort: [5, 6, 11, 12, 13]
=====
Duplicate:
True
False
True
=====
Missing numbers:
2
2
8

```

Lab No: 8

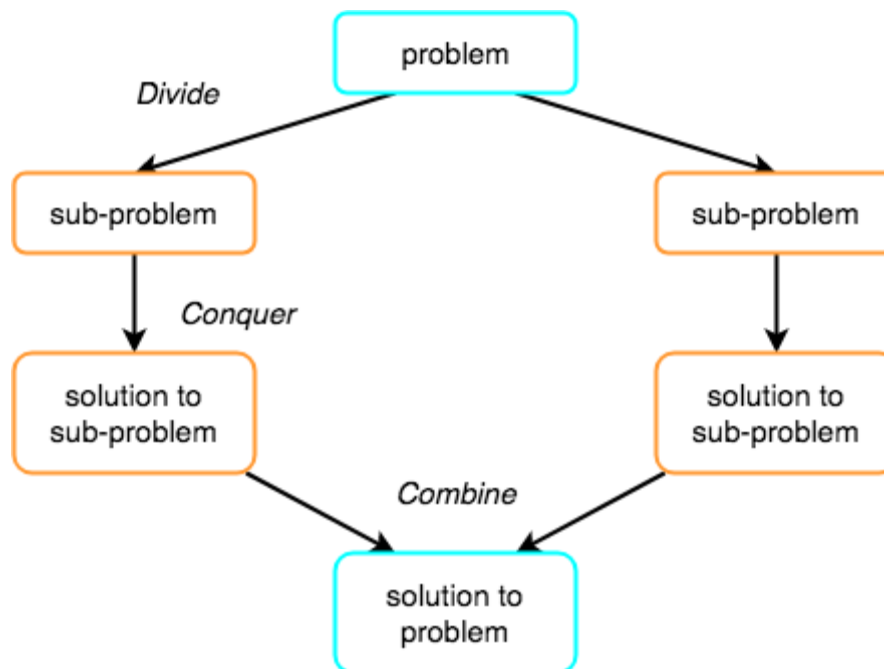
Objective: Applying the Divide-and-Conquer Approach to Sorting

Merge Sort Algorithm:

In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. **Combine** the solutions of the subproblems to find the solution of the actual problem.



How Merge Sort Works?

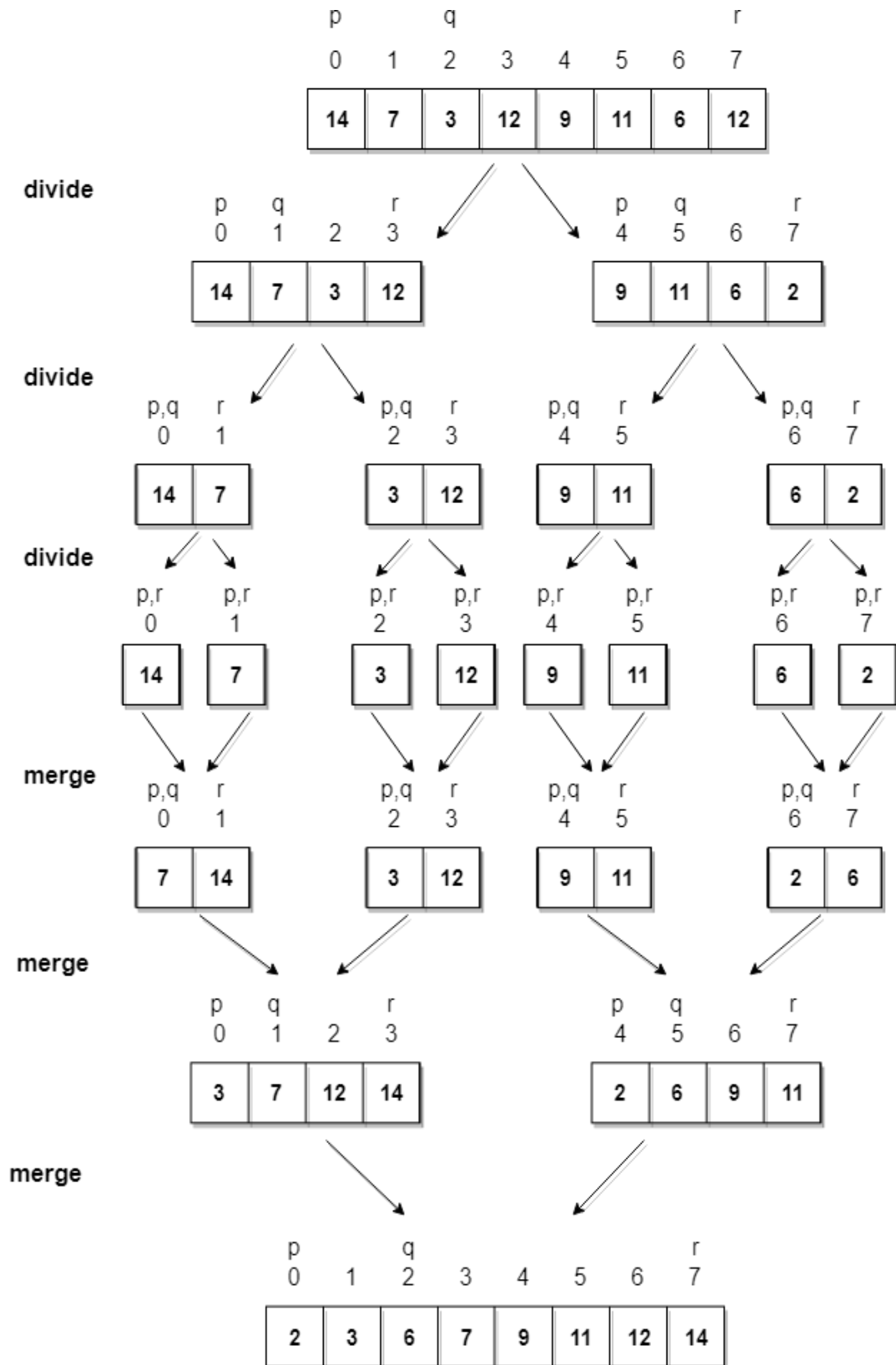
As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two subarrays with 3 elements each. But breaking the original array into 2 smaller subarrays is not helping us in sorting the array. So we will break these subarrays into even smaller subarrays, until we have multiple subarrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we

have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array. Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

Below, we have a pictorial representation of how merge sort will sort the given array.



In merge sort we follow the following steps:

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula $(p + r)/2$ and mark the middle index as q , and break the array into two subarrays, from p to q and from $q + 1$ to r index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

Complexity Analysis of Merge Sort

- Time complexity of Merge Sort is $O(n \cdot \log n)$ in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.
- It requires **equal amount of additional space** as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.
- It is the best Sorting technique used for sorting **Linked Lists**.

Quick Sort Algorithm:

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort Algorithm work?

QuickSort works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Complexity Analysis of Quick Sort

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Tasks:

- 1) Write python implementations on Merge Sort & Quick Sort
- 2) You are given an integer array score of size n, where score[i] is the score of the i^{th} athlete in a competition. All the scores are guaranteed to be **unique**.
The athletes are **placed** based on their scores, where the 1st place athlete has the highest score, the 2nd place athlete has the 2nd highest score, and so on. The placement of each athlete determines their rank:
 - The 1st place athlete's rank is "Gold Medal".
 - The 2nd place athlete's rank is "Silver Medal".
 - The 3rd place athlete's rank is "Bronze Medal".
 - For the 4th place to the nth place athlete, their rank is their placement number (i.e., the xth place athlete's rank is "x").
 Return an array answer of size n where answer[i] is the **rank** of the i^{th} athlete.

Example 1:

Input: score = [5,4,3,2,1]

Output: ["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]

Explanation: The placements are [1st, 2nd, 3rd, 4th, 5th].

Example 2:

Input: score = [10,3,8,9,4]

Output: ["Gold Medal", "5", "Bronze Medal", "Silver Medal", "4"]

Explanation: The placements are [1st, 5th, 3rd, 2nd, 4th].

- 3) Given two strings s and t, return true if t is an anagram of s, and false otherwise.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

- 4) Given an integer array nums, return *the maximum difference between two successive elements in its sorted form*. If the array contains less than two elements, return 0.
You must write an algorithm that runs in linear time and uses linear extra space.

Example 1:

Input: nums = [3,6,9,1]

Output: 3

Explanation: The sorted form of the array is [1,3,6,9], either (3,6) or (6,9) has the maximum difference 3.

Example 2:

Input: nums = [10]

Output: 0

Explanation: The array contains less than 2 elements, therefore return 0.

```
# 1) Merge Sort and Quick Sort

# Merge Sort
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

    return arr

# Example usage
print("Merge Sort: ", merge_sort([38, 27, 43, 3, 9, 82, 10]))
print("-----")

# Quick Sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Example usage
print("Quick Sort: ", quick_sort([38, 27, 43, 3, 9, 82, 10]))
print("-----")
```

```
# 2) Athlete Ranks Based on Scores
```

```
def find_relative_ranks(scores):  
    sorted_scores = sorted(enumerate(scores), key=lambda x: x[1], reverse=True)  
    ranks = [""] * len(scores)
```

```
    for rank, (index, score) in enumerate(sorted_scores):  
        if rank == 0:  
            ranks[index] = "Gold Medal"  
        elif rank == 1:  
            ranks[index] = "Silver Medal"  
        elif rank == 2:  
            ranks[index] = "Bronze Medal"  
        else:  
            ranks[index] = str(rank + 1)
```

```
    return ranks
```

```
# Example usage
```

```
print("Relative Rank: ", find_relative_ranks([5, 4, 3, 2, 1]))  
print("Relative Rank: ", find_relative_ranks([10, 3, 8, 9, 4]))  
print("-----")
```

```
# 3) Check if Two Strings are Anagrams
```

```
def is_anagram(s, t):  
    return sorted(s) == sorted(t)
```

```
# Example usage
```

```
print("Anagram: ", is_anagram("anagram", "nagaram")) # Output: True  
print("Anagram: ", is_anagram("rat", "car")) # Output: False  
print("-----")
```

```
# 4) Maximum Difference Between Successive Elements in Sorted Array
```

```
def maximum_gap(nums):
```

```
    if len(nums) < 2:  
        return 0
```

```
    min_num = min(nums)  
    max_num = max(nums)  
    bucket_size = max(1, (max_num - min_num) // (len(nums) - 1))  
    bucket_count = (max_num - min_num) // bucket_size + 1
```

```
    buckets = [[float('inf'), float('-inf')] for _ in range(bucket_count)]
```

```
    for num in nums:  
        bucket_idx = (num - min_num) // bucket_size  
        buckets[bucket_idx][0] = min(buckets[bucket_idx][0], num)  
        buckets[bucket_idx][1] = max(buckets[bucket_idx][1], num)
```

```
    max_gap = 0  
    previous_max = min_num
```

```
    for bucket in buckets:  
        if bucket[0] == float('inf'):  
            continue  
        max_gap = max(max_gap, bucket[0] - previous_max)  
        previous_max = bucket[1]
```

```
    return max_gap
```

```
# Example usage
```

```
print("Max Gap: ", maximum_gap([3, 6, 9, 1])) # Output: 3  
print("Max Gap: ", maximum_gap([10])) # Output: 0
```

```
✓ 0.0s  
Merge Sort: [3, 9, 10, 27, 38, 43, 82]
```

```
Quick Sort: [3, 9, 10, 27, 38, 43, 82]
```

```
Relative Rank: ['Gold Medal', 'Silver Medal', 'Bronze Medal', '4', '5']
```

```
Relative Rank: ['Gold Medal', '5', 'Bronze Medal', 'Silver Medal', '4']
```

```
-----  
Anagram: True
```

```
Anagram: False
```

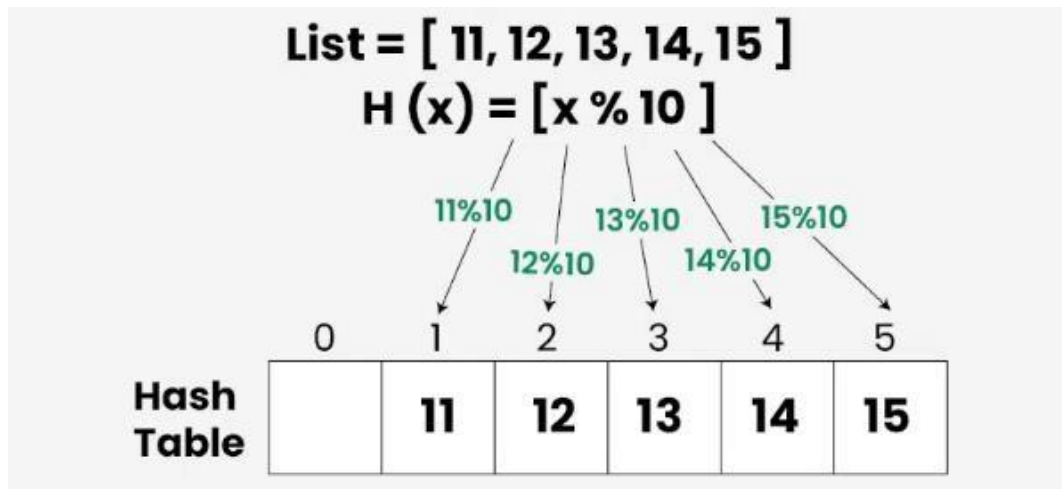
```
-----  
Max Gap: 3
```

```
Max Gap: 0
```

Lab No: 9

Objective: Utilizing HashMaps for Efficient Data Storage and Retrieval

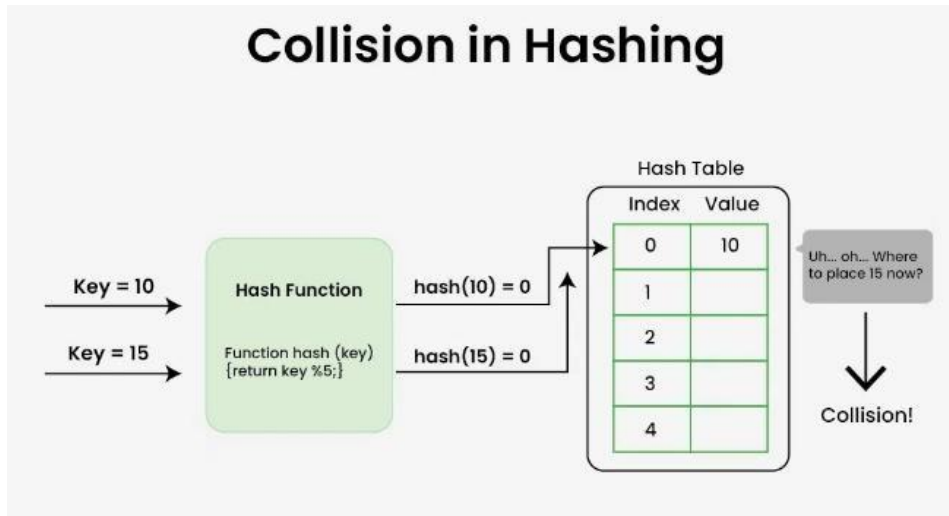
Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.



Collision Resolution Techniques

In **Hashing**, hash functions were used to generate hash values. The hash value is used to create an index for the keys in the hash table. The hash function may return the same hash value for two or more keys. When two or more keys have the same hash value, a **collision** happens. To handle this collision, we use **Collision Resolution Techniques**.

Collision in Hashing



Collision Resolution Techniques

There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing

1) Separate Chaining

The idea behind Separate Chaining is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

2) Open Addressing

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

Tasks:

- 1) Implement a Python function for hashing with collision handling using chaining.
- 2) Given two strings s and t , *determine if they are isomorphic*.

Two strings s and t are isomorphic if the characters in s can be replaced to get t .

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

Input: $s = \text{"egg"}, t = \text{"add"}$

Output: true

Explanation:

The strings s and t can be made identical by:

- Mapping 'e' to 'a'.
- Mapping 'g' to 'd'.

Example 2:

Input: s = "foo", t = "bar"

Output: false

Explanation:

The strings s and t can not be made identical as 'o' needs to be mapped to both 'a' and 'r'.

Example 3:

Input: s = "paper", t = "title"

Output: true

```
# 1) Hashing with Collision Handling Using Separate Chaining
class HashTable:
```

```
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
```

```
    def _hash(self, key):
        return hash(key) % self.size
```

```
    def insert(self, key, value):
        index = self._hash(key)
        for entry in self.table[index]:
            if entry[0] == key:
                entry[1] = value
                return
        self.table[index].append([key, value])
```

```
    def get(self, key):
        index = self._hash(key)
        for entry in self.table[index]:
            if entry[0] == key:
                return entry[1]
        return None
```

```
    def remove(self, key):
        index = self._hash(key)
        for i, entry in enumerate(self.table[index]):
            if entry[0] == key:
                del self.table[index][i]
                return
```

```
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 38)
print(hash_table.get("name"))
hash_table.remove("age")
print(hash_table.get("age"))
print("=====")
```

```
# 2) Check if Two Strings are Isomorphic
```

```
def are_isomorphic(s, t):
    if len(s) != len(t):
        return False

    mapping_s_to_t = {}
    mapping_t_to_s = {}

    for char_s, char_t in zip(s, t):
        if char_s in mapping_s_to_t:
            if mapping_s_to_t[char_s] != char_t:
                return False
        else:
            mapping_s_to_t[char_s] = char_t

        if char_t in mapping_t_to_s:
            if mapping_t_to_s[char_t] != char_s:
                return False
        else:
            mapping_t_to_s[char_t] = char_s

    return True
```

```
print(are_isomorphic("egg", "add"))
print(are_isomorphic("foo", "bar"))
print(are_isomorphic("paper", "title"))
```

[43] ✓ 0.0s

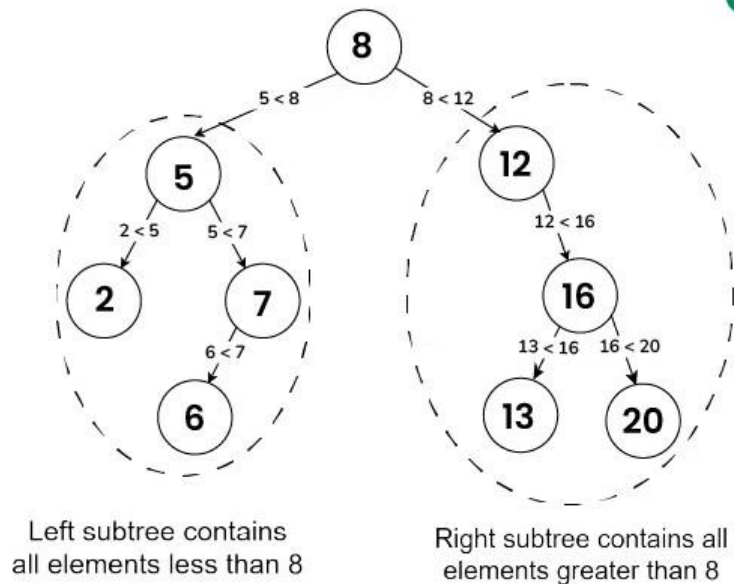
```
... Alice
None
=====
True
False
True
```


Lab No: 10

Objective: Implementing Binary Search Trees for Efficient Searching

Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner. Binary search tree follows all properties of binary tree and for every nodes, its **left** subtree contains values less than the node and the **right** subtree contains values greater than the node. This hierarchical structure allows for efficient **Searching**, **Insertion**, and **Deletion** operations on the data stored in the tree.

Binary Search Tree



Properties of Binary Search Tree:

- The left subtree of a node contains only nodes with keys less 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(BST may have duplicate values with different handling approaches).

Tasks:

Implement Binary Search Trees (BSTs) in Python with Inorder Traversal, a function to add nodes, and a searching function.

```

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def add(self, key):
        """Function to add a node with the given key to the BST."""
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._add_recursive(self.root, key)

    def _add_recursive(self, node, key):
        """Helper function to insert a new node recursively."""
        if key < node.val:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._add_recursive(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._add_recursive(node.right, key)

    def search(self, key):
        """Function to search for a node with the given key."""
        return self._search_recursive(self.root, key)

    def _search_recursive(self, node, key):
        if node is None or node.val == key:
            return node
        if key < node.val:
            return self._search_recursive(node.left, key)
        return self._search_recursive(node.right, key)

    def inorder_traversal(self):
        """Function to perform inorder traversal of the BST."""
        return self._inorder_recursive(self.root)

    def _inorder_recursive(self, node):
        if node is None:
            return []
        return self._inorder_recursive(node.left) + [node.val] + self._inorder_recursive(node.right)

```

```

bst = BinarySearchTree()
bst.add(10)
bst.add(5)
bst.add(15)
bst.add(3)
bst.add(7)
bst.add(12)
bst.add(18)

```

```

print("Inorder Traversal:", bst.inorder_traversal())

```

```

search_value = 7
search_result = bst.search(search_value)
if search_result:
    print("Found:", search_result.val)
else:
    print(f"Not Found: {search_value}")

```

```

search_value = 20
search_result = bst.search(search_value)
if search_result:
    print("Found:", search_result.val)
else:
    print(f"Not Found: {search_value}")

```

[1] ✓ 0.0s

```

... Inorder Traversal: [3, 5, 7, 10, 12, 15, 18]
Found: 7
Not Found: 20

```

