Activity 01

1. Write a python code to demonstrate working on list:

Coding

```
fruits = ['apple', 'banana', 'cherry', 'date']
# Accessing elements
print(fruits[0])
print(fruits[2])
# Modifying elements
fruits[1] = 'kiwi'
print(fruits)
# Appending elements
fruits.append('orange')
print(fruits)
# Inserting elements
fruits.insert(1, 'grape')
print(fruits)
# Removing elements
fruits.remove('cherry')
print(fruits)
# Slicing
print(fruits[1:4])
# Length of the list
print(len(fruits))
# Looping through the list
for fruit in fruits:
    print(fruit)
# Checking if an element exists in the list
if 'apple' in fruits:
    print('Apple is in the list.')
# Sorting the list
fruits.sort()
print(fruits)
```

Output

```
Run   python

C:\Users\Lenovo\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\Lenovo\PycharmProjects\pythonProject\python.py
apple
cherry
['apple', 'kiwi', 'cherry', 'date']
['apple', 'kiwi', 'cherry', 'date', 'orange']
['apple', 'grape', 'kiwi', 'cherry', 'date', 'orange']
['apple', 'grape', 'kiwi', 'date', 'orange']
['grape', 'kiwi', 'date']
5
apple
grape
kiwi
date
orange
Apple is in the list.
['apple', 'date', 'grape', 'kiwi', 'orange']

Process finished with exit code 0
```

Activity 02

2. Implement list, stack, queue adt:

List

```
# Creating a list
fruits = ['apple', 'banana', 'cherry', 'date']
# Accessing elements
print(fruits[0])
print(fruits[2])
# Modifying elements
fruits[1] = 'kiwi'
print(fruits)
# Appending elements
fruits.append('orange')
print(fruits)
# Inserting elements
fruits.insert(1, 'grape')
print(fruits)
# Removing elements
fruits.remove('cherry')
print(fruits)
# Slicing
print(fruits[1:4])
# Length of the list
print(len(fruits))
# Looping through the list
for fruit in fruits:
    print(fruit)
# Checking if an element exists in the list
if 'apple' in fruits:
    print('Apple is in the list.')
# Sorting the list
fruits.sort()
print(fruits)
```

Output

Stack

```
class Stack:
    def __init__(self):
        self.stack = []
    def is_empty(self):
        return len(self.stack) == 0
    def push(self, item):
        self.stack.append(item)
    def pop(self):
        if self.is_empty():
            return None
        return self.stack.pop()
    def peek(self):
        if self.is_empty():
            return None
        return self.stack[-1]
    def size(self):
        return len(self.stack)
stack = Stack()
stack.push(10)
stack.push(20)
```

```
stack.push(30)
print("Stack contents:", stack.stack)
print("Stack size:", stack.size())
print("Top of the stack:", stack.peek())
popped_item = stack.pop()
print("Popped item:", popped_item)
print("Stack contents after pop:", stack.stack)
```

Output

Queue

```
# Creating an empty queue
queue = []
# Enqueueing elements to the queue
queue.append("apple")
queue.append("banana")
queue.append("cherry")
# Printing the queue
print(queue)
# Dequeuing elements from the queue
item = queue.pop(0)
print("Dequeued item:", item)
# Printing the updated queue
print(queue)
# Checking if the queue is empty
if len(queue) == 0:
    print("Queue is empty.")
```

```
else:
    print("Queue is not empty.")
# Peeking at the front of the queue
front_item = queue[0]
print("Front item:", front_item)
```

Output

Activity 03

3. Implement Linear search and graph:

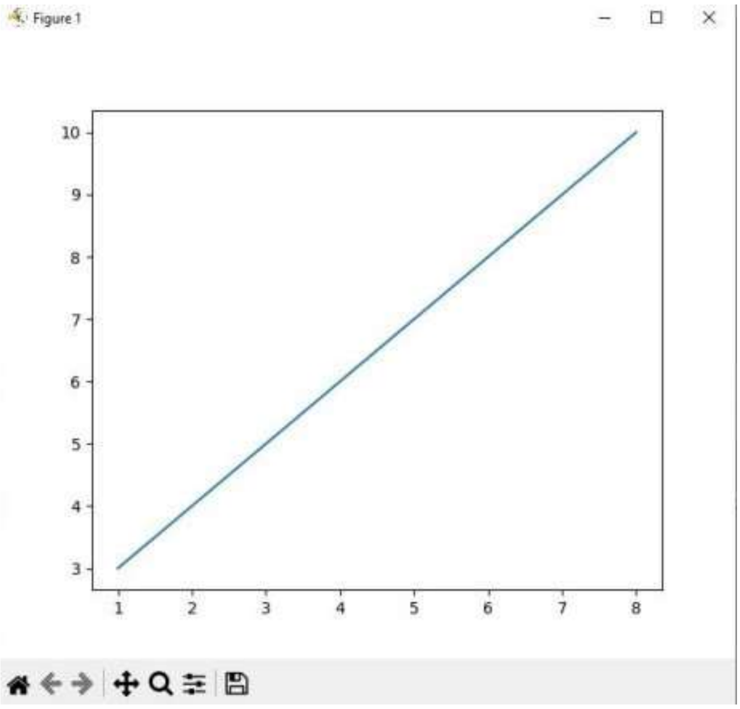
```
import time
start=time.time()
def linear_search (first, n, key):
    for i in range(n):
        if(first[i]==key):
            return 0
first=[]
n=int(input(("enter number of elements: ")))
for i in range(n):
    first.append(int(input()))
    print(first)
key=int(input("enter key:"))
res=linear_search(first, n, key)
if (res==0):
    print("element found")
else:
    print("element not found")
end=time.time()
print("running time of program is",{end-start})
```

Output

Graph

```
import matplotlib.pyplot as plt
import numpy as np
xpoints=np.array([1,8])
ypoints=np.array([3,10])
plt.plot(xpoints, ypoints)
plt.show()
```

Output



Activity 04

4. Implement quick sort:

Coding

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    lesser, equal, greater = [], [], []
    for num in arr:
        if num < pivot:
            lesser.append(num)
        elif num == pivot:
            equal.append(num)
        else:
            greater.append(num)
    return quick_sort(lesser) + equal + quick_sort(greater)
my_list = [7, 2, 1, 6, 8, 5, 3, 4]
sorted_list = quick_sort(my_list)
print(sorted_list)
```

output

Activity 05

5. Write down 10 differences between linear and nonlinear data structure:

Factor	Linear Data Structure	Non-Linear Data Structure
Data Element Arrangement	In a linear data structure, data elements are sequentially connected, allowing users to traverse all elements in one run.	In a non-linear data structure, data elements are hierarchically connected, appearing on multiple levels.
Implementation Complexity	Linear data structures are relatively easier to implement.	Non-linear data structures require a higher level of understanding and are more complex to implement.
Levels	All data elements in a linear data structure exist on a single level.	Data elements in a non-linear data structure span multiple levels.
Traversal	A linear data structure can be traversed in a single run.	Traversing a non-linear data structure is more complex, requiring multiple runs.
Memory Utilization	Linear data structures do not efficiently utilize memory.	Non-linear data structures are more memory-friendly.
Time Complexity	The time complexity of a linear data structure is directly proportional to its size, increasing as input size increases.	The time complexity of a non-linear data structure often remains constant, irrespective of its input size.
Applications	Linear data structures are ideal for application software development.	Non-linear data structures are commonly used in image processing and Artificial Intelligence.
Examples	Linked List, Queue, Stack, Array.	Tree, Graph, Hash Map.

Activity 6

6. Implement singly linked list:

Coding

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class singlyLinkedList:
    def __init__(self):
        self.head = None
    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
    def delete_node(self, key):
        current_node = self.head
        if current_node and current_node.data == key:
            self.head = current_node.next
            current_node = None
            return
        prev = None
        while current_node and current_node.data != key:
            prev = current_node
            current_node = current_node.next
```

```
        if current_node is None:
            return
        prev.next = current_node.next
        current_node = None
    def print_list(self):
        current_node = self.head
        while current_node:
            print(current_node.data, end="  ")
            current_node = current_node.next
        print("None")
if __name__ == "__main__":
    ll = singlyLinkedList()
    ll.append(1)
    ll.append(2)
    ll.append(3)
    ll.append(4)
    ll.prepend(0)
    ll.print_list()
    ll.delete_node(3)
    ll.print_list()
```

Output

Activity 7

7. Define doubly linked and give its representation:

A doubly linked list is a data structure where each element, besides storing its own data, also contains references or pointers to the previous and next elements in the list. This allows traversal in both forward and backward directions.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
class DoublyLinkedList:
    def __init__(self):
        self.head = None
    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
            new_node.prev = current
    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
    def delete(self, data):
        current = self.head
        while current:
```

```
            if current.data == data:
                if current.prev:
                    current.prev.next = current.next
                else:
                    self.head = current.next
                if current.next:
                    current.next.prev = current.prev
                return
            current = current.next
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print()
dll = DoublyLinkedList()
dll.append(1)
dll.append(2)
dll.append(3)
dll.prepend(0)
dll.display()
dll.delete(2)
dll.display()
```

Output

Representation:

Data: Holds the actual data that the node is intended to store.

Pointer to the next node: Contains the memory address of the next node in the sequence.

Pointer to the previous node: Contains the memory address of the previous node in the sequence.

Activity 8

8. Implement stack data structure:

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return len(self.items) == 0
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return None  # Stack is empty
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return None  # Stack is empty
    def size(self):
        return len(self.items)
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print("Stack:", stack.items)
print("Peek:", stack.peek())
print("Pop:", stack.pop())
print("Stack:", stack.items)
```

Output

Activity 9

9. List out advantage and disadvantage of recursion:

Advantages:

- Recursion is very simple and easy to understand.
- It requires a minimal number of programming statements.
- Recursion will break the problem into smaller pieces of sub problems for example Tower of Hanoi.
- It is used to solve mathematical, trigonometric, or any type of algebraic problems.
- It is more useful in multiprogramming and multitasking environments.
- It is useful in solving data structure problems like linked lists, queues and stacks.
- Recursive function is useful in tree traversal and stacks.
- Complex tasks can be solved easily.

Disadvantages:

- It consumes more storage space than other techniques such as Iteration, Dynamic programming etc.
- If base condition is not set properly then it may create a problem such as a system crash, freezing etc.,
- Compared to other techniques recursion is a time-consuming process and less efficient.
- It is difficult to trace the logic of the function.
- Computer memory is exhausted if recursion enters an infinite loop.
- Excessive function calls are being used.
- Each function called will occupy memory in stack. Which will lead to stack overflow.

Activity 10

10.Implement priority queue:

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.heap = []
        self.counter = 0  # Used for tie-breaking elements with the same priority
    def push(self, item, priority):
        heapq.heappush(self.heap, (priority, self.counter, item))
        self.counter += 1
    def pop(self):
        if self.heap:
            return heapq.heappop(self.heap)[2]
        else:
            raise IndexError("pop from an empty priority queue")
    def peek(self):
        if self.heap:
            return self.heap[0][2]
        else:
            return None
    def __len__(self):
        return len(self.heap)
    def is_empty(self):
        return len(self.heap) == 0
pq = PriorityQueue()
pq.push('Task 1', 5)
pq.push('Task 2', 3)
pq.push('Task 3', 7)
while not pq.is_empty():
    print(pq.pop())
```

Output

Activity 11

11. Implement binary search tree:

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
class BinarySearchTree:
    def __init__(self):
        self.root = None
    def insert(self, key):
        self.root = self._insert_recursively(self.root, key)
    def _insert_recursively(self, root, key):
        if root is None:
            return TreeNode(key)
        if key < root.key:
            root.left = self._insert_recursively(root.left, key)
        else:
            root.right = self._insert_recursively(root.right, key)
        return root
    def search(self, key):
        return self._search_recursively(self.root, key)
    def _search_recursively(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self._search_recursively(root.left, key)
        else:
            return self._search_recursively(root.right, key)
    def inorder_traversal(self):
        self._inorder_recursively(self.root)
    def _inorder_recursively(self, root):
        if root:
```

```
            self._inorder_recursively(root.left)
            print(root.key, end=" ")
            self._inorder_recursively(root.right)
bst = BinarySearchTree()
bst.insert(5)
bst.insert(3)
bst.insert(7)
bst.insert(2)
bst.insert(4)
bst.insert(6)
bst.insert(8)
print("Inorder Traversal:")
bst.inorder_traversal()  # Output: 2 3 4 5 6 7 8
print("\nSearch:")
print("Key 4 found:", bst.search(4) is not None)
print("Key 9 found:", bst.search(9) is not None)
```

Output

Activity 12

12.Write a python code for BFS:

```
from collections import defaultdict, deque
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
    def bfs(self, start):
        visited = set()
        queue = deque([start])
        visited.add(start)
        while queue:
            node = queue.popleft()
            print(node, end=" ")
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)
print("BFS Traversal starting from vertex 2:")
g.bfs(2)
```

Output

Activity 13

13.Implement hash function:

```
def custom_hash(key, table_size):
    """
    A simple hash function that converts a string key into a numerical hash value.
    Args:
    key (str): The input key to be hashed.
    table_size (int): The size of the hash table.
    Returns:
    int: The hashed value of the key.
    """
    hash_value = 0
    for char in key:
        hash_value = (hash_value * 31 + ord(char)) % table_size
    return hash_value
key = "example"
table_size = 10
hashed_value = custom_hash(key, table_size)
print("Hashed value of '{}': {}".format(key, hashed_value))
```

Output