DATA STRUCTURE LAB MANUAL

Dr. Muhammad Siddique

Data Structure Lab Manual
(Using Python)

Dr. Muhammad Siddique

February 2025

Contents

1 Week 1: Introduction to Python for Data Structures 2
2 Week 2: Understanding and Implementing Stacks in Python 8
3 Week 3: Queues 14
4 Week 4: Singly Linked Lists 20
5 Week 5: Doubly Linked Lists 23
6 Week 6: Trees 26
7 Week 7: Binary Search Trees (BST) 32
8 Week 8: Graphs 37
9 Week 9: Hashing Techniques 41
10 Week 10: Sorting and Searching Algorithms 46
11 Week 11: Advanced Sorting Techniques 51
12 Week 12: Priority Queues and Heaps 54
13 Week 13: Graph Algorithms 57
14 Week 14: Case Study — Real-World Applications 68
15 Week 15: Complex Problem-Solving Lab 72

1 Week 1: Introduction to Python for Data Structures

Objective: Understand the foundations of Python programming in the context of data structures. Students will learn how to use lists, functions, and input/output operations effectively to handle, process, and analyze small data sets.

Learning Outcomes:

1. Understand how Python lists are used to store and manipulate data.
2. Use built-in functions such as `sum()`, `max()`, and `min()` to compute useful statistics.
3. Develop modular code using user-defined functions.
4. Implement iterative and recursive algorithms for computational problems.
5. Analyze small data sets (e.g., marks, temperatures) and draw conclusions programmatically.

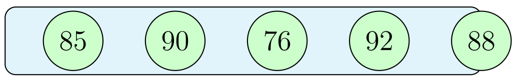
Tasks:

1. **Task 1:** Write a program to store marks of students in a list and calculate average, highest, and lowest score.
2. **Task 2:** Implement a small calculator using functions for addition, subtraction, multiplication, and division.
3. **Task 3:** Create a program that stores daily temperatures of a city for a week and finds the hottest and coldest days.
4. **Task 4:** Write a program to find the factorial of a number using iteration and recursion.
5. **Task 5:** Create a program to calculate the sum of even and odd numbers separately from a list.

Concept Overview: Lists and Functions

A `list` in Python is an ordered collection of items that can hold different types of data — numbers, strings, or even other lists. Lists are mutable, meaning their content can be changed. Functions, on the other hand, help divide a program into smaller, reusable pieces of logic.

Python List: `marks = [85, 90, 76, 92, 88]`



Details of Lab Experiment:

Task 1: Student Marks in List

We use a Python `list` to store marks of students. Then we apply built-in functions like `sum()`, `max()`, and `min()` to calculate useful statistics such as average, highest, and lowest score.

```
marks = [85, 90, 76, 92, 88]
average = sum(marks)/len(marks)
mx = max(marks)
mn = min(marks)

print(f"Average Marks = {average:.2f}")
print(f"Highest Marks = {mx}")
print(f"Lowest Marks  = {mn}")
```

Listing 1: Student Marks Example

- `sum(marks)` returns the total of all marks.
- `len(marks)` gives the number of elements in the list.
- `max()` and `min()` extract the highest and lowest marks.

You can extend this program by taking input dynamically using a loop and storing it using `append()`.

Task 2: Calculator using Functions

We define separate functions for each operation to promote modularity and code reuse.

```
def add(a,b):
    return a+b

def subtract(a,b):
    return a-b

def multiply(a,b):
    return a*b

def divide(a,b):
    return a/b if b != 0 else "Error: Division by zero"

x = float(input("Enter first number: "))
y = float(input("Enter second number: "))

print("Addition:", add(x,y))
print("Subtraction:", subtract(x,y))
print("Multiplication:", multiply(x,y))
print("Division:", divide(x,y))
```

Listing 2: Simple Calculator

Key Concepts

- Each operation is handled by a separate function.
- Functions make testing and debugging easier.
- Input from user makes the program interactive.

Task 3: Weekly Temperature Record

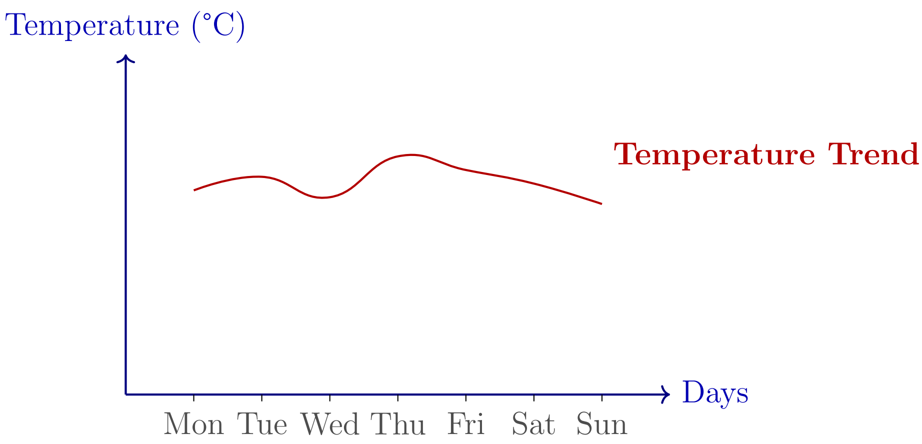
We represent a week's temperature readings using a list and analyze them using built-in functions.

```
temp = [30, 32, 29, 35, 33, 31, 28]

print(f"Weekly Temperatures: {temp}")
print(f"Hottest Day = {max(temp)}  C ")
print(f"Coldest Day = {min(temp)}  C ")

average_temp = sum(temp)/len(temp)
print(f"Average Temperature = {average_temp:.2f}  C ")
```

Listing 3: Temperature Tracker



Task 4: Factorial (Iterative and Recursive)

The factorial of a number n, represented as $n!$, is the product of all integers from 1 to n. Factorials are crucial in combinatorics, algorithm design, and complexity analysis.

Iterative Approach:

```python
def factorial_iter(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result

print("Factorial (Iterative):", factorial_iter(5))
```

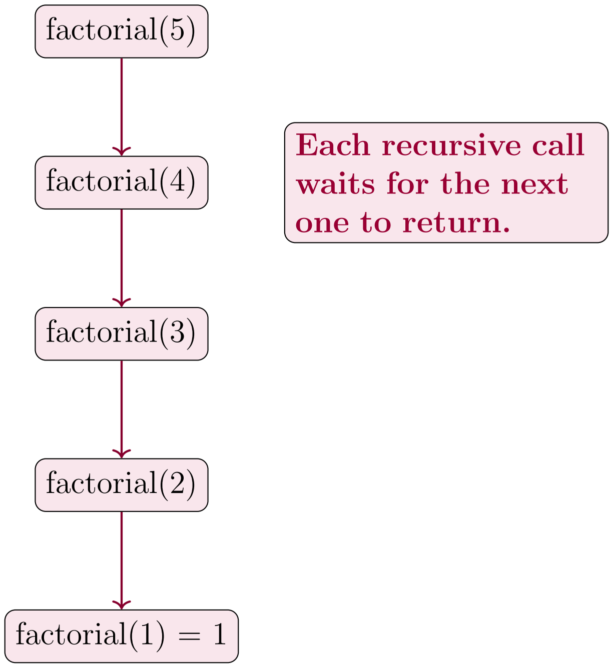
Listing 4: Factorial (Iterative)

Recursive Approach:

```python
def factorial_rec(n):
    if n <= 1:
        return 1
    else:
        return n * factorial_rec(n-1)

print("Factorial (Recursive):", factorial_rec(5))
```

Listing 5: Factorial (Recursive)



Task 5: Sum of Even and Odd Numbers

We traverse a list and use conditions to compute the sum of even and odd numbers separately.

```
arr = [1,2,3,4,5,6,7,8,9]
sum_even = sum(x for x in arr if x % 2 == 0)
sum_odd  = sum(x for x in arr if x % 2 != 0)

print("Sum of Even Numbers:", sum_even)
print("Sum of Odd Numbers:", sum_odd)
```

Listing 6: Sum of Even and Odd Numbers

List comprehensions combine looping and conditional filtering in one line, making code compact and expressive.

Additional Practice Tasks

1. Reverse a list without using `reverse()` function.
2. Find the second highest number in a list.

3. Count frequency of each element in a list.
4. Print only unique elements from a list.

Viva Questions

1. What is the difference between a list and a variable?
2. Why do we use functions in programming?
3. Give one real-world example of using lists.
4. What is recursion and how is it different from iteration?
5. Why is modular programming important?
6. What happens if a recursive function has no base condition?

Summary

This lab introduced the foundational concepts of Python programming that serve as the cornerstone for understanding advanced data structures like stacks, queues, and linked lists. The exercises emphasized clean, modular, and colorful coding practices for analyzing real-world data through simple yet powerful Python tools.

2 Week 2: Understanding and Implementing Stacks in Python

Objective: To understand the concept of stacks, their LIFO (Last-In First-Out) property, and implement real-world applications such as reversing strings, expression validation, and postfix evaluation.

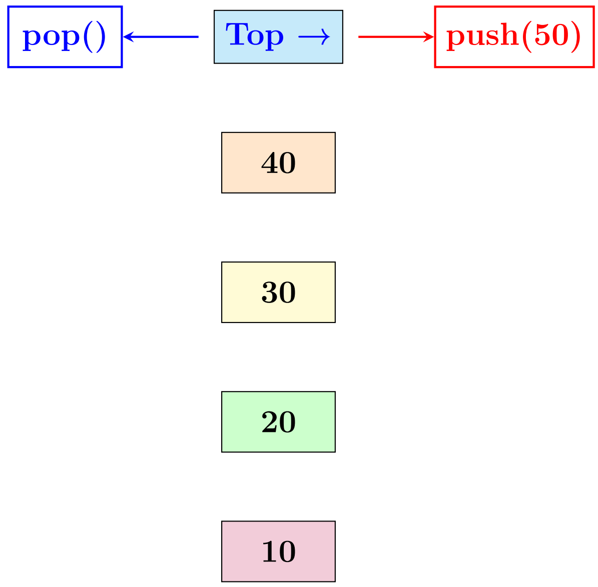
Tasks:

1. **Task 1:** Implement a stack to reverse a string.
2. **Task 2:** Use a stack to check balanced parentheses in an expression.
3. **Task 3:** Simulate the Undo feature of a text editor using a stack.
4. **Task 4:** Convert an infix expression to postfix using a stack.
5. **Task 5:** Evaluate a postfix expression using a stack.

Concept Overview:

A stack is a linear data structure that follows the **Last-In First-Out (LIFO)** principle — the last element added is the first one to be removed. Common stack operations include:

- **push()** — to insert (add) an element into the stack
- **pop()** — to remove (delete) the topmost element
- **peek() or top()** — to view the top element without removing it
- **isEmpty()** — to check if the stack is empty



Stack Visualization (LIFO: Last In, First Out)

Details of Lab Experiment

Task 1: Reverse a String using Stack

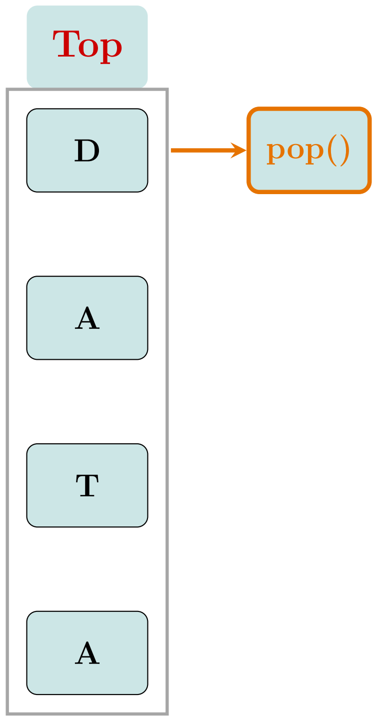
In this task, we reverse a string using the logic of a stack. Though Python allows easy reversal using slicing, we use a stack to better understand the push and pop mechanism.

```
def reverse_string(s):
    stack = []
    for char in s:
        stack.append(char)      # Push each character
    reversed_s = ''
    while stack:
        reversed_s += stack.pop()   # Pop characters in reverse
    return reversed_s

text = "DataStructures"
print("Original String:", text)
print("Reversed String:", reverse_string(text))
```

Listing 7: Reversing a String using Stack

Visualization of Stack (Characters)



Stack storing characters (LIFO Concept)

—

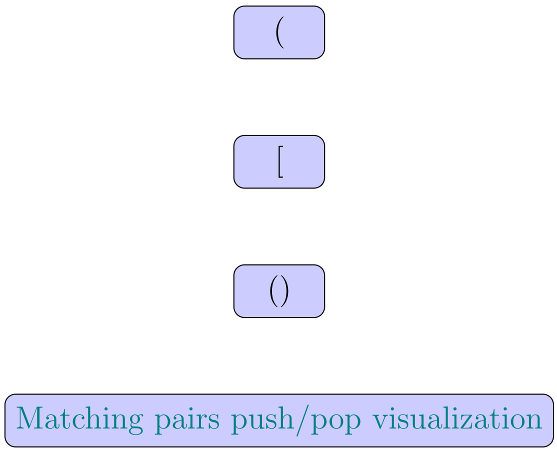
Task 2: Balanced Parentheses Checker

Stacks are extremely useful for validating balanced parentheses in mathematical or programming expressions.

```python
def is_balanced(expr):
    stack = []
    pairs = {')':'(', '}':'{', ']':'['}
    for ch in expr:
        if ch in '({[':
            stack.append(ch)
        elif ch in ')}]':
            if not stack or stack.pop() != pairs[ch]:
                return False
    return not stack

print(is_balanced("{[()]}"))  # True
print(is_balanced("{[(])}"))  # False
```

Listing 8: Balanced Parentheses Checker



—

Task 3: Simulate Undo Feature

A text editor’s Undo feature is stack-based. Every action (typing, deleting) is pushed into a stack. When the user undoes, the last action is popped.

```
actions = []
actions.append(’Type A’)
actions.append(’Type B’)
actions.append(’Delete C’)

print("Undo Action:", actions.pop())
print("Remaining Last Action:", actions[-1])
```

Listing 9: Undo Operation Simulation

—

Task 4: Infix to Postfix Conversion using Stack

The infix expression (e.g., A + B) requires operator precedence handling. Using a stack ensures correct order of operations during conversion to postfix (e.g., AB+).

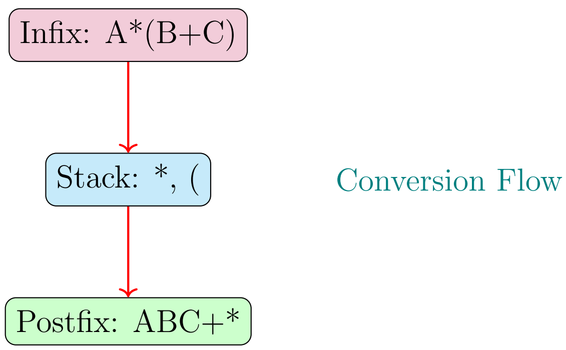
```
precedence = {’+’:1, ’-’:1, ’*’:2, ’/’:2}

def infix_to_postfix(exp):
    stack = []
    output = ’’
    for ch in exp:
        if ch.isalnum():
            output += ch
        elif ch == ’(’:
            stack.append(ch)
        elif ch == ’)’:
            while stack and stack[-1] != ’(’:
```

```
                output += stack.pop()
            stack.pop()
        else:
            while stack and stack[-1] != '(' and \
                  precedence.get(stack[-1],0) >= precedence.get(ch,0):
                output += stack.pop()
            stack.append(ch)
    while stack:
        output += stack.pop()
    return output

print("Postfix:", infix_to_postfix("A*(B+C)"))
```

Listing 10: Infix to Postfix Conversion



—

Task 5: Evaluate Postfix Expression

Now, we use a stack to evaluate postfix expressions directly. Each operand is pushed, and when an operator is encountered, we pop two operands and apply the operation.

```
def evaluate_postfix(exp):
    stack = []
    for ch in exp:
        if ch.isdigit():
            stack.append(int(ch))
        else:
            b = stack.pop()
            a = stack.pop()
            if ch == '+': stack.append(a + b)
            elif ch == '-': stack.append(a - b)
            elif ch == '*': stack.append(a * b)
            elif ch == '/': stack.append(a // b)
    return stack[0]

print("Result:", evaluate_postfix('23*54*+9-'))  # Output: 17
```

Listing 11: Evaluate Postfix Expression

—

Real-World Applications of Stacks:

- Expression evaluation and syntax parsing
- Undo/Redo functionality in editors
- Backtracking in games or recursion
- Function call management in recursion

—

Viva Questions

1. What does the term LIFO mean in stack operations?
2. Why are stacks essential in recursion?
3. How do stacks handle nested parentheses in expressions?
4. What are some differences between Stack and Queue?

3 Week 3: Queues

Objective: To understand the working of Queues and their various types (Linear, Circular, and Deque), and apply these structures to real-world problems like customer service systems, printer job handling, and supermarket billing counters.

Learning Outcomes:

- Understand the FIFO (First-In-First-Out) mechanism.
- Learn to implement Linear, Circular, and Double-Ended Queues.
- Apply queues to solve real-life scheduling and service problems.
- Develop logic for handling overflow and underflow conditions.

Tasks

1. Implement a queue to manage Customer Service Requests.
2. Simulate Printer Queue Operations.
3. Model a Supermarket Billing Counter.
4. Implement a Circular Queue with overflow/underflow handling.
5. Implement a Deque (Double-Ended Queue) for task scheduling.

—

Details of Lab Experiments

Task 1: Customer Service Queue

A Queue follows the First-In-First-Out (FIFO) principle — the first customer to enter is the first to be served. This is the basis of most real-world service systems such as call centers and helpdesks.

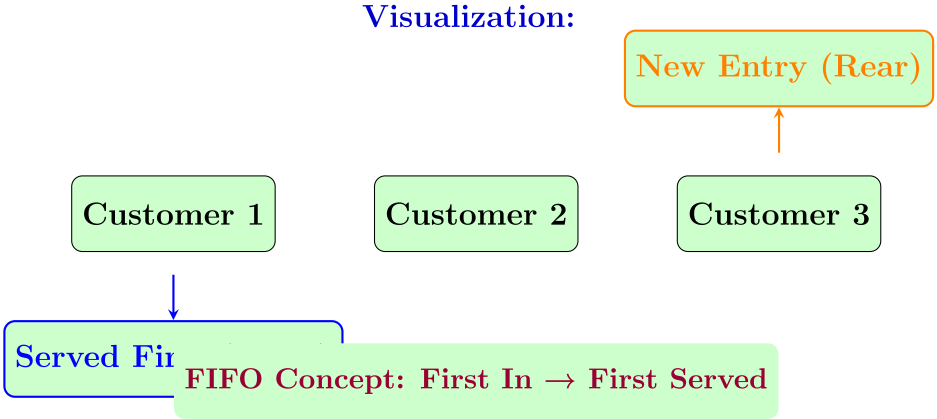
```
from collections import deque

service_queue = deque()
service_queue.append('Customer 1')
service_queue.append('Customer 2')
service_queue.append('Customer 3')

while service_queue:
    print('Serving:', service_queue.popleft())
```

Listing 12: Customer Service Queue

Visualization:



—

Task 2: Printer Queue Simulation

Printers maintain a queue of pending documents. The oldest document is printed first, maintaining the FIFO order.

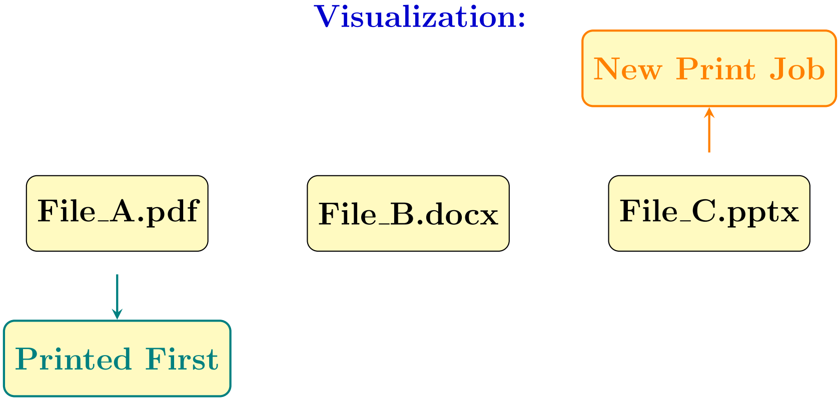
```
from collections import deque

printer_queue = deque(['File_A.pdf', 'File_B.docx', 'File_C.
    pptx'])

while printer_queue:
    print('Printing:', printer_queue.popleft())
```

Listing 13: Printer Queue Simulation

Visualization:



—

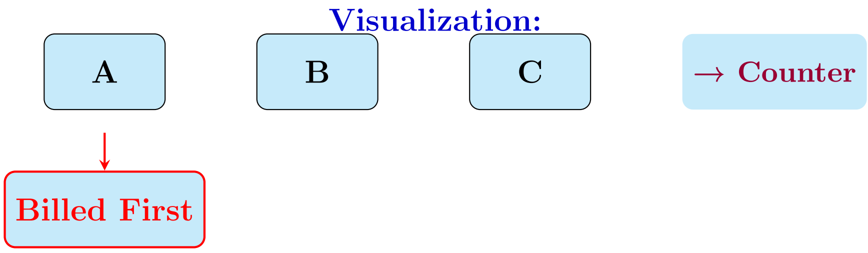
Task 3: Supermarket Billing Counter

This task models how customers are served at a supermarket checkout. Each customer waits in line and is billed in order.

```python
from collections import deque

billing_queue = deque(['Customer A', 'Customer B', 'Customer
    C'])
while billing_queue:
    print(billing_queue.popleft(), 'is being billed.')
```

Listing 14: Supermarket Billing Counter



—

Task 4: Circular Queue Implementation

Circular queues efficiently reuse space by connecting the end of the queue to the front. They are used in systems such as CPU scheduling and memory buffering.

```python
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None]*size
        self.front = 0
        self.rear = -1
        self.count = 0

    def enqueue(self, x):
        if self.count == self.size:
            print('Queue Overflow!')
            return
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = x
        self.count += 1

    def dequeue(self):
        if self.count == 0:
            print('Queue Underflow!')
            return
        val = self.queue[self.front]
        print('Removed:', val)
        self.front = (self.front + 1) % self.size
        self.count -= 1
```

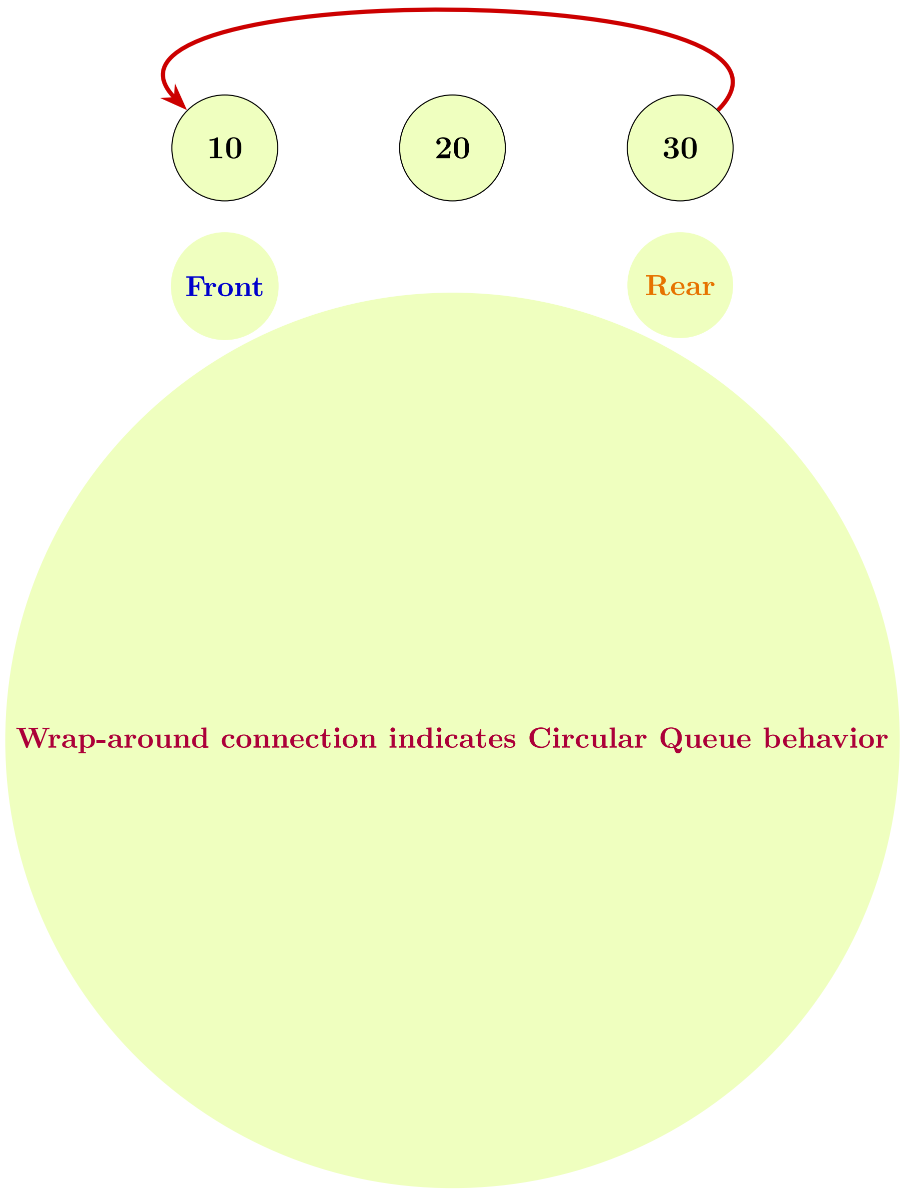
```python

cq = CircularQueue(3)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.dequeue()
cq.dequeue()
cq.dequeue()
cq.dequeue()
```

Listing 15: Circular Queue Implementation

tikz

Visualization of Circular Queue:



—

Task 5: Double-Ended Queue (Deque)

A Deque allows insertion and deletion from both ends. It is flexible and can be used for **task scheduling**, **palindrome checking**, and **sliding window algorithms**.

```
from collections import deque

tasks = deque()
tasks.append('Task 1')
tasks.appendleft('Urgent Task')
```

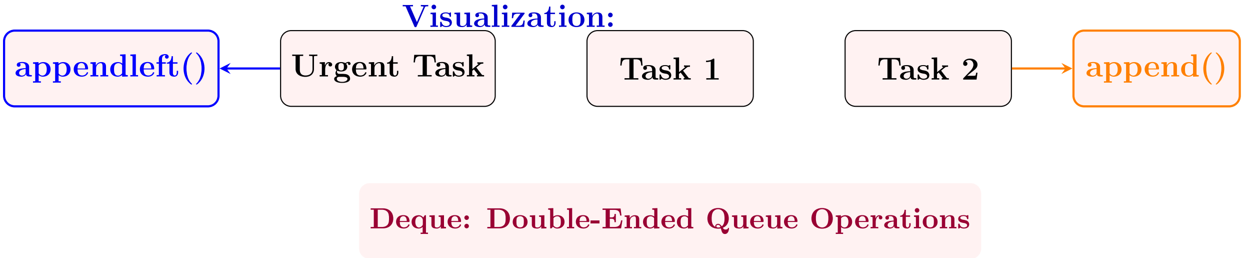
```python
tasks.append('Task 2')

print('Front:', tasks[0])
print('Back:', tasks[-1])

tasks.popleft()
print('After popping front, new front:', tasks[0])
```

Listing 16: Deque for Task Scheduling

Visualization:



—

Viva Questions

1. What is the FIFO property of a queue?
2. Differentiate between a Linear Queue and a Circular Queue.
3. How does a Deque differ from a standard queue?
4. What real-life systems use Circular Queues?
5. Can we implement a stack using two queues? Explain the logic.

4 Week 4: Singly Linked Lists

Objective: Understand how to implement singly linked lists in Python, perform basic operations (insertion, deletion, searching, reversing), and connect the concept to real-world examples like playlists and navigation systems.

Concept Overview:

A singly linked list is a dynamic linear data structure consisting of nodes, where each node holds:

- **Data:** The value stored in the node.
- **Pointer (next):** The reference (or address) of the next node in the sequence.

Unlike arrays, linked lists are not stored in contiguous memory. Each node dynamically connects to the next one using a pointer, forming a "chain" of data elements.

Visualization of a Singly Linked List:

Each node points to the next node in the list.

Tasks:

1. Create a singly linked list and display its elements.
2. Implement insertion at the beginning, middle, and end.
3. Implement deletion from the beginning, middle, and end.
4. Search for an element within the list.
5. Reverse a singly linked list.

Detailed Lab Implementation:

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None    # pointer to the next node
```

```python

class LinkedList:
    def __init__(self):
        self.head = None   # initially, list is empty

    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:  # if list is empty
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

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

    def delete_at_end(self):
        if not self.head:
            print("List is empty")
            return
        if not self.head.next:
            self.head = None
            return
        temp = self.head
        while temp.next.next:
            temp = temp.next
        temp.next = None

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

# Usage Example
ll = LinkedList()
ll.insert_at_end(10)
ll.insert_at_end(20)
ll.insert_at_end(30)
ll.insert_at_beginning(5)
ll.display()
ll.delete_at_end()
ll.display()
```

Listing 17: Basic Singly Linked List Implementation

Explanation: Each operation in a singly linked list manipulates the next pointer to establish or break connections between nodes. Insertions and deletions do not require shifting of elements (as in arrays), which improves efficiency.

Time Complexity Table:

Operation	Average Time Complexity	Best Use Case
Insertion (End)	$O(n)$	Append data dynamically
Insertion (Beginning)	$O(1)$	Stack or Queue models
Deletion	$O(n)$	Remove unwanted data
Search	$O(n)$	Find specific nodes
Reverse	$O(n)$	Reverse ordering logic

Real-World Applications:

- **Music Playlists:** Each song links to the next track.
- **Browser Tabs:** Forward/backward navigation.
- **Dynamic Memory Management:** Used in operating systems.

Viva Questions:

1. What is a linked list and how does it differ from arrays?
2. Why are linked lists preferred for dynamic data storage?
3. What happens if you forget to update the "next" pointer during insertion?
4. What is the time complexity of searching in a singly linked list?

5 Week 5: Doubly Linked Lists

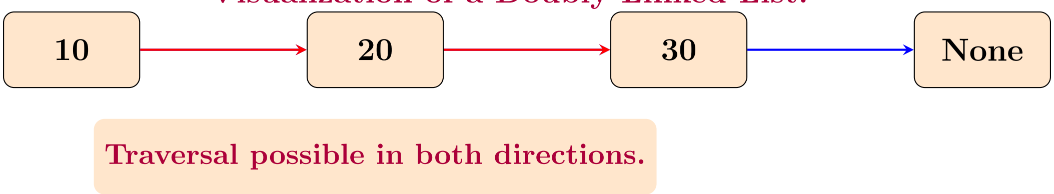
Objective: Extend understanding of linked lists by learning how to traverse in both directions using Doubly Linked Lists (DLL) and perform more flexible data operations.

Concept Overview:

A doubly linked list consists of nodes that store:

- **Data:** The value of the node.
- **Next Pointer:** Reference to the next node.
- **Previous Pointer:** Reference to the previous node.

Visualization of a Doubly Linked List:



Tasks:

1. Implement forward and backward traversal.
2. Simulate a music playlist with next/previous navigation.
3. Model browser history navigation.
4. Implement insertion and deletion at different positions.
5. Implement a program to remove duplicate elements.

Detailed Lab Implementation:

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

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

```

```
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
        new_node.prev = temp

    def display_forward(self):
        temp = self.head
        while temp:
            print(temp.data, end=" <-> ")
            last = temp
            temp = temp.next
        print("None")

    def display_backward(self):
        temp = self.head
        if not temp:
            return
        while temp.next:
            temp = temp.next
        while temp:
            print(temp.data, end=" <-> ")
            temp = temp.prev
        print("None")

# Usage Example
dll = DoublyLinkedList()
dll.insert_at_end(1)
dll.insert_at_end(2)
dll.insert_at_end(3)
dll.display_forward()
dll.display_backward()
```

Listing 18: Doubly Linked List Implementation

Explanation: A doubly linked list allows two-way navigation, improving flexibility for operations like undo/redo, backward browsing, and data iteration. However, it requires extra memory for the additional pointer.

Performance Comparison:

Feature	Singly Linked List	Doubly Linked List
Memory Usage	Low	Higher (extra pointer)
Traversal	One Direction	Two Directions
Insertion/Deletion	Moderate	Easier (bidirectional links)
Applications	Simple Queues, Stacks	Navigation, Undo/Redo

Real-World Applications:

- **Music Players:** Forward/backward song navigation.
- **Web Browsers:** Back and forward page history.
- **Text Editors:** Undo and redo functionality.

Viva Questions:

1. How is a doubly linked list different from a singly linked list?
2. Why does a doubly linked list require more memory?
3. Explain a real-world scenario where backward traversal is essential.
4. What would happen if we forget to update the “prev” pointer during insertion?

6 Week 6: Trees

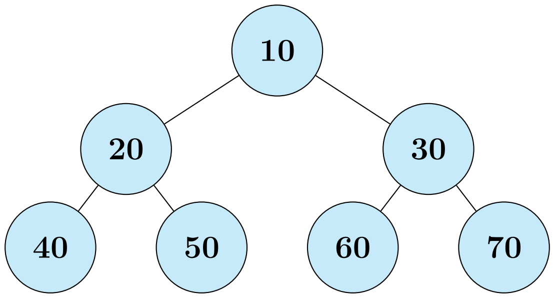
Objective: Understand hierarchical data organization using Tree Data Structures and implement key tree operations such as traversal, height calculation, and leaf node counting.

Concept Overview:

A Tree is a non-linear, hierarchical data structure composed of nodes connected by edges. It follows a **parent–child** relationship, making it suitable for representing hierarchical information such as organizational charts or file systems.

- **Root Node:** The topmost node of the tree.
- **Parent and Child Nodes:** Every node (except the root) has exactly one parent, and can have zero or more children.
- **Leaf Node:** A node with no children.
- **Height of Tree:** The number of edges on the longest path from the root to a leaf.

Visualization of a Binary Tree:



Example: A complete binary tree structure

Tasks:

1. Implement binary tree traversals (Inorder, Preorder, Postorder).
2. Represent an organizational hierarchy using a tree.
3. Store arithmetic expressions using a binary tree.
4. Implement a function to calculate the height of a binary tree.
5. Count the number of leaf nodes in a binary tree.

Details of Lab Experiment:

A binary tree is a special type of tree where each node can have at most two children—commonly referred to as the **left** and **right** child. Traversals are systematic ways to visit each node of a tree exactly once, allowing us to process data in different logical orders.

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

def inorder(root):
    """Left -> Root -> Right"""
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

def preorder(root):
    """Root -> Left -> Right"""
    if root:
        print(root.key, end=" ")
        preorder(root.left)
        preorder(root.right)

def postorder(root):
    """Left -> Right -> Root"""
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.key, end=" ")

def height(root):
    """Return the height of the binary tree"""
    if root is None:
        return 0
    return 1 + max(height(root.left), height(root.right))

def count_leaves(root):
    """Count the number of leaf nodes"""
    if root is None:
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaves(root.left) + count_leaves(root.right
)

# Tree Construction
root = Node(10)
root.left = Node(20)
root.right = Node(30)
root.left.left = Node(40)
root.left.right = Node(50)
root.right.left = Node(60)
```

```python
root.right.right = Node(70)

# Output Demonstration
print("Inorder Traversal:")
inorder(root)
print("\nPreorder Traversal:")
preorder(root)
print("\nPostorder Traversal:")
postorder(root)
print("\nHeight of Tree:", height(root))
print("Leaf Nodes Count:", count_leaves(root))
```

Listing 19: Binary Tree Example

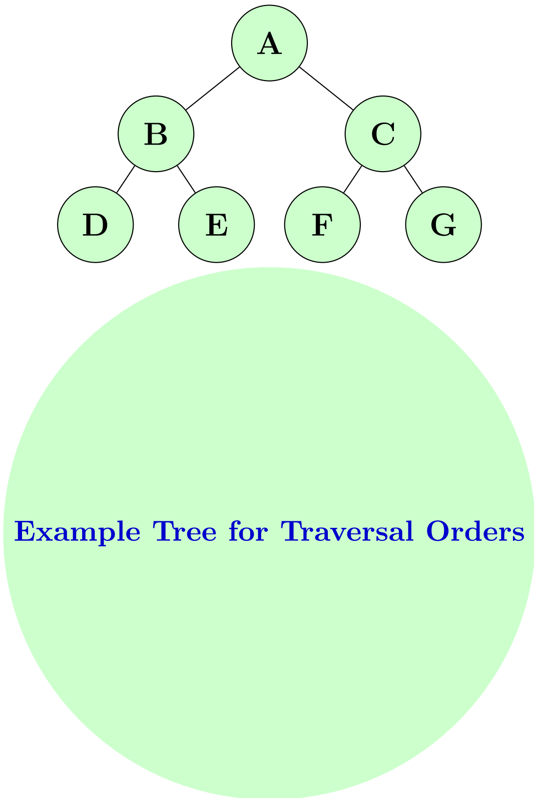
Explanation: Tree traversals are fundamental for performing operations like expression evaluation, file scanning, and hierarchical visualization.

- **Inorder Traversal** — Left → Root → Right (used in binary search trees to get sorted data).
- **Preorder Traversal** — Root → Left → Right (used for copying or serialization).
- **Postorder Traversal** — Left → Right → Root (used for deletion or evaluating expressions).

Traversal Example Output:

Traversal Type	Output Sequence
Inorder	40 20 50 10 60 30 70
Preorder	10 20 40 50 30 60 70
Postorder	40 50 20 60 70 30 10

Extended Visualization of Traversal Orders:



Inorder: D → B → E → A → F → C → G **Preorder:** A → B → D → E → C → F → G **Postorder:** D → E → B → F → G → C → A

Real-World Applications:

- **File Systems:** Hierarchical folder and file structures.
- **Organizational Hierarchy:** Representing reporting relationships.
- **Decision Trees:** Used in Artificial Intelligence and Machine Learning.
- **Binary Search Trees:** For fast data lookup and sorting.
- **Expression Trees:** Used in compilers and expression evaluation.

Performance Summary:

Operation	**Time Complexity (Average Case)**
Insertion	$O(\log n)$
Search	$O(\log n)$
Traversal	$O(n)$
Height Calculation	$O(n)$
Leaf Count	$O(n)$

Viva Questions:

1. What is the difference between a **Binary Tree** and a **Binary Search Tree**?
2. Explain **Inorder Traversal** in your own words.
3. Define **Leaf Node** and **Height** of a tree.
4. Mention two real-life applications of tree structures.
5. Why are trees considered non-linear data structures?

7 Week 7: Binary Search Trees (BST)

Objective:
To understand the concept, structure, and operations of Binary Search Trees (BST) in Python. Students will learn how to perform insertion, searching, deletion, and how to find the minimum, maximum, inorder successor, and predecessor. They will also apply BSTs in real-world applications such as phone directories and data organization systems.

Tasks:

1. Implement insertion and search operations in a BST.
2. Find the minimum and maximum values in a BST.
3. Use BST to manage a simple phone directory application.
4. Implement node deletion in a BST with all possible cases.
5. Find the inorder successor and predecessor in a BST.

Details of Lab Experiment:

Task 1: Insertion and Search in BST

Binary Search Trees maintain a unique property — for every node, the left subtree contains nodes with smaller values, while the right subtree contains nodes with larger values. This structure allows efficient searching and insertion in $O(\log n)$ time (for balanced trees).

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

def insert(root, val):
    if not root:
        return Node(val)
    if val < root.val:
        root.left = insert(root.left, val)
    elif val > root.val:
        root.right = insert(root.right, val)
    return root

def search(root, key):
    if not root:
```

```
        return False
    if root.val == key:
        return True
    return search(root.left, key) if key < root.val else
   search(root.right, key)

root = None
for v in [50, 30, 70, 20, 40]:
    root = insert(root, v)

print('Searching 40:', 'Found' if search(root, 40) else 'Not
     Found')
print('Searching 60:', 'Found' if search(root, 60) else 'Not
     Found')
```

Listing 20: BST Insert and Search Example

Explanation:
Insertion and searching in a BST depend on comparing values and following left or right subtrees recursively. This results in efficient lookups when the tree is balanced.

Task 2: Finding Minimum and Maximum in BST

```
def find_min(root):
    while root.left:
        root = root.left
    return root.val

def find_max(root):
    while root.right:
        root = root.right
    return root.val

print('Minimum Value:', find_min(root))
print('Maximum Value:', find_max(root))
```

Listing 21: Finding Min and Max in BST

Explanation:
The smallest value is located at the leftmost node, and the largest value is located at the rightmost node of the BST.

Task 3: Phone Directory using BST

Binary Search Trees can efficiently store key-value pairs such as names (keys) and phone numbers (values). This structure can be used to build simple databases or phone directories.

```python
class NodeP:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.left = None
        self.right = None

def insertP(root, name, phone):
    if not root:
        return NodeP(name, phone)
    if name < root.name:
        root.left = insertP(root.left, name, phone)
    elif name > root.name:
        root.right = insertP(root.right, name, phone)
    return root

def searchP(root, name):
    if not root:
        return "Not Found"
    if root.name == name:
        return root.phone
    return searchP(root.left, name) if name < root.name else
     searchP(root.right, name)

rootP = None
rootP = insertP(rootP, "Alice", "12345")
insertP(rootP, "Bob", "67890")
insertP(rootP, "Charlie", "54321")

print("Phone of Bob:", searchP(rootP, "Bob"))
print("Phone of David:", searchP(rootP, "David"))
```

Listing 22: Phone Directory using BST

Explanation:
The BST-based phone directory allows quick lookup and insertion of names in alphabetical order. This technique is foundational for dictionary data structures.

Task 4: Deletion in BST

Deletion in a BST requires careful handling of three scenarios:

- Deleting a leaf node (no children)
- Deleting a node with one child
- Deleting a node with two children — replace with inorder successor

```python
def find_min_node(root):
    while root.left:
        root = root.left
    return root

def delete_node(root, key):
    if not root:
        return None
    if key < root.val:
        root.left = delete_node(root.left, key)
    elif key > root.val:
        root.right = delete_node(root.right, key)
    else:
        if not root.left:
            return root.right
        if not root.right:
            return root.left
        temp = find_min_node(root.right)
        root.val = temp.val
        root.right = delete_node(root.right, temp.val)
    return root
```

Listing 23: Deleting Node in BST

Explanation:

When deleting a node with two children, the inorder successor (smallest node in the right subtree) is used to maintain BST ordering.

Task 5: Inorder Successor and Predecessor

```python
def inorder_successor(root, target):
    if target.right:
        cur = target.right
        while cur.left:
            cur = cur.left
        return cur
    succ = None
    cur = root
    while cur:
        if target.val < cur.val:
            succ = cur
            cur = cur.left
        elif target.val > cur.val:
            cur = cur.right
        else:
            break
    return succ
```

Listing 24: Inorder Successor in BST

Explanation:
The inorder successor is the next node in sorted order (smallest greater value). It is widely used in deletion operations and ordered traversals.

Real-World Applications:

- Phone and contact directory management
- Database indexing and search optimization
- Autocomplete systems and prefix searches
- File system hierarchies and compiler design

Viva Questions:

1. What is the main property that distinguishes a BST from a binary tree?
2. What are the time complexities of search, insertion, and deletion in a BST?
3. How do you find the inorder successor and predecessor in a BST?
4. What are some practical applications of BSTs in software systems?

8 Week 8: Graphs

Objective:
To understand how graphs represent relationships between entities. Students will learn graph representation (adjacency list and matrix), and perform traversal techniques such as Breadth-First Search (BFS) and Depth-First Search (DFS). They will also explore cycle detection and connected component identification, applying these concepts to model real-world problems like city maps, transportation networks, and social graphs.

Tasks:

1. Represent a city map using an adjacency list.
2. Implement BFS to find the shortest path in an unweighted graph.
3. Use DFS to detect cycles in an undirected graph.
4. Construct an adjacency matrix representation of the same graph.
5. Detect connected components using DFS.

Details of Lab Experiment:

Task 1: City Map using Adjacency List

Graphs are an excellent way to model cities, where intersections represent nodes and roads represent edges. An adjacency list is efficient in memory, especially when most pairs of nodes are not directly connected (sparse graphs).

```python
graph = {}

def add_edge(u, v):
    graph.setdefault(u, []).append(v)
    graph.setdefault(v, []).append(u)

add_edge('A', 'B')
add_edge('A', 'C')
add_edge('B', 'D')
add_edge('C', 'D')

for k, v in graph.items():
    print(k, '->', ' '.join(v))
```

Listing 25: City Map using Adjacency List

Explanation:
Each node maintains a list of its directly connected neighbors. This representation allows dynamic updates and is commonly used in network routing, city navigation, and friendship graphs.

Task 2: Breadth-First Search (BFS) for Shortest Path

BFS explores all neighboring nodes at the current depth before moving deeper. This property makes it ideal for finding the shortest path in an unweighted graph.

```
from collections import deque

def bfs(start):
    visited = set()
    q = deque([start])
    visited.add(start)
    order = []

    while q:
        u = q.popleft()
        order.append(u)
        for nbr in graph.get(u, []):
            if nbr not in visited:
                visited.add(nbr)
                q.append(nbr)
    return order

print('BFS from A:', bfs('A'))
```

Listing 26: BFS Traversal for Shortest Path

Explanation:
BFS uses a queue to visit nodes level by level. It guarantees the shortest path in an unweighted graph and is widely applied in GPS systems, social network suggestions, and recommendation engines.

Task 3: Cycle Detection using DFS

Depth-First Search (DFS) explores as far as possible along each branch before backtracking. It can be used to detect cycles by tracking visited nodes and their parents.

```
def dfs_cycle(node, parent, visited):
    visited.add(node)
    for nbr in graph.get(node, []):
        if nbr not in visited:
            if dfs_cycle(nbr, node, visited):
```

```python
                return True
        elif nbr != parent:
            return True
    return False

print('Graph has cycle?', dfs_cycle('A', None, set()))
```

Listing 27: Cycle Detection using DFS

Explanation:
If a back edge (an edge connecting to a visited node that is not the parent) is found, the graph contains a cycle. Cycle detection is fundamental in deadlock detection, network validation, and compiler dependency checks.

Task 4: Adjacency Matrix Representation

An adjacency matrix uses a 2D array where each cell (i, j) indicates whether an edge exists between nodes i and j. It's fast for edge lookup but consumes more memory for sparse graphs.

```python
nodes = ['A', 'B', 'C', 'D']
idx = {n: i for i, n in enumerate(nodes)}
mat = [[0] * len(nodes) for _ in nodes]

def add_edge_mat(u, v):
    i, j = idx[u], idx[v]
    mat[i][j] = 1
    mat[j][i] = 1

add_edge_mat('A', 'B')
add_edge_mat('A', 'C')
add_edge_mat('B', 'D')
add_edge_mat('C', 'D')

for row in mat:
    print(' '.join(map(str, row)))
```

Listing 28: Adjacency Matrix Representation

Explanation:
Adjacency matrices are preferred for dense graphs and are used in mathematical graph algorithms like Floyd-Warshall and Dijkstra's algorithm (weighted graphs).

Task 5: Connected Components

A connected component is a subset of nodes where every node is reachable from every other node within that subset. This concept helps identify isolated groups in a network.

```python
def dfs(node, visited):
    visited.add(node)
    print(node, end=' ')
    for nbr in graph.get(node, []):
        if nbr not in visited:
            dfs(nbr, visited)

visited = set()
for node in graph:
    if node not in visited:
        dfs(node, visited)
        print()
```

Listing 29: Finding Connected Components

Explanation:
By running DFS on each unvisited node, we can count and display all connected components. This method is essential for analyzing social clusters, subnetworks, and connectivity in communication systems.

Real-World Applications of Graphs:

- City navigation and route-planning (GPS).
- Internet and computer network modeling.
- Social network analysis and friend recommendations.
- Dependency graphs in software compilation.
- Electrical circuit and flow simulations.

Viva Questions:

1. Differentiate between adjacency list and adjacency matrix representations.
2. What are the time complexities of BFS and DFS traversals?
3. How can cycles be detected in undirected graphs?
4. Mention two real-world problems that can be modeled using graphs.

9 Week 9: Hashing Techniques

Objective: Learn how to use hashing to design efficient storage and retrieval systems. Understand collision handling methods such as linear probing, quadratic probing, and separate chaining, and explore real-world applications like employee databases and duplicate detection.

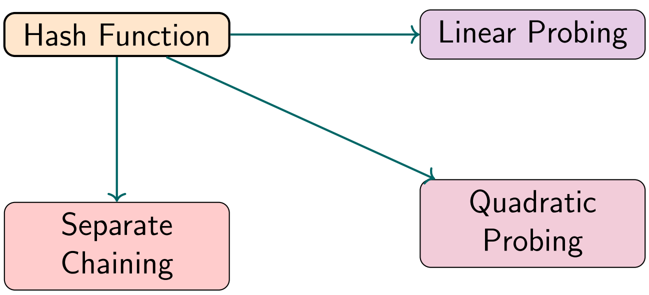
Tasks:

1. Implement a hash table with linear probing.
2. Apply hashing to store employee records.
3. Use hashing to detect duplicates.
4. Implement quadratic probing for collision resolution.
5. Implement separate chaining using lists.

Details of Lab Experiment:

Hashing is a powerful technique for achieving constant-time data access. It uses a **hash function** to map data (keys) to specific locations (indices) in a structure called a **hash table**. Collisions occur when multiple keys map to the same location, and are handled through techniques such as: linear probing, quadratic probing, or separate chaining.

Collision Handling Methods Overview



Task 1: Linear Probing Hash Table

If a collision occurs, the algorithm linearly scans for the next available slot.

```
SIZE = 10
table = [None] * SIZE

def hash_func(key):
    return key % SIZE

def insert(key):
    idx = hash_func(key)
    while table[idx] is not None:
        idx = (idx + 1) % SIZE
    table[idx] = key

for k in [23, 43, 13, 27]:
    insert(k)

for i, v in enumerate(table):
    print(i, '->', v)
```

Listing 30: Linear Probing Hash Table

Explanation: Linear probing stores colliding keys in subsequent free slots. While simple, it can lead to primary clustering where long chains of occupied cells form.

Task 2: Employee Records using Hashing

Store employee records using hash-based indexing for fast lookups.

```
SIZE = 10
emp_table = [None] * SIZE

def insert_emp(id, name):
    idx = id % SIZE
    while emp_table[idx] is not None:
        idx = (idx + 1) % SIZE
    emp_table[idx] = (id, name)

def search_emp(id):
    idx = id % SIZE
    start = idx
    while emp_table[idx] is not None:
        if emp_table[idx][0] == id:
            return emp_table[idx][1]
        idx = (idx + 1) % SIZE
        if idx == start: break
    return 'Not Found'

insert_emp(101, 'Ali')
```

```
insert_emp(202, 'Sara')
insert_emp(303, 'Ahmed')

print('Employee 202:', search_emp(202))
```

Listing 31: Employee Records with Hashing

Explanation: The hash index is computed from the employee ID. Collisions are resolved by probing sequential slots. This ensures fast, consistent retrieval with minimal memory overhead.

Task 3: Detect Duplicates using Hashing

Hashing can instantly detect duplicate data in logs or large datasets.

```
arr = [1, 2, 3, 2, 4, 5, 1]
seen = set()
dups = []

for x in arr:
    if x in seen:
        dups.append(x)
    else:
        seen.add(x)

print('Duplicates:', dups)
```

Listing 32: Detect Duplicates using Hashing

Explanation: Python's `set()` uses hashing internally. Each lookup and insertion is `O(1)` on average — ideal for large-scale duplicate detection.

Task 4: Quadratic Probing

Quadratic probing reduces clustering by increasing probe distance quadratically.

```
SIZE = 11
table = [None] * SIZE

def insert_quad(key):
    idx = key % SIZE
    i = 0
    while table[(idx + i*i) % SIZE] is not None:
        i += 1
    table[(idx + i*i) % SIZE] = key

for k in [23, 34, 45, 56]:
    insert_quad(k)

```

```
print(table)
```

Listing 33: Quadratic Probing Example

Explanation: Quadratic probing avoids consecutive collisions. However, hash tables should use prime sizes to minimize repetition and ensure full coverage.

Task 5: Separate Chaining

Each hash table slot holds a list of keys that share the same index.

```
SIZE = 7
table = [[] for _ in range(SIZE)]

def insert_chain(key):
    idx = key % SIZE
    table[idx].append(key)

for k in [10, 20, 15, 7, 17, 24]:
    insert_chain(k)

for i, chain in enumerate(table):
    print(i, ':', ' -> '.join(map(str, chain)) or 'NULL')
```

Listing 34: Separate Chaining Hash Table

Explanation: Separate chaining removes clustering completely. Python's built-in `dict` and `set` use a similar hybrid chaining model internally.

Real-World Applications

- **Databases:** Indexing and searching via primary keys.
- **Compilers:** Symbol tables for variable and address mapping.
- **Networking:** Routing tables, DNS caching.
- **Cybersecurity:** Password hashing for secure storage.
- **Data Science:** Duplicate detection in datasets.

Performance Summary

- **Average Time:** `O(1)` for insertion/search.
- **Worst Case:** `O(n)` (high collisions).

- **Space Complexity:** Depends on table size and strategy.

Common Mistakes to Avoid

1. Poor hash function causing clustering.
2. Ignoring table resizing when full.
3. Using non-prime table sizes in quadratic probing.
4. Forgetting wrap-around indexing.

Viva Questions

1. What is hashing and why is it used?
2. Difference between linear and quadratic probing?
3. Why is separate chaining preferred in high-collision cases?
4. Explain primary clustering and its prevention.
5. Why should hash table sizes be prime numbers?

10 Week 10: Sorting and Searching Algorithms

Objective: Implement and compare classical sorting and searching algorithms: Bubble Sort, Selection Sort, Insertion Sort, Binary Search, and Linear Search. Develop a deep understanding of their time complexities, behavior on various datasets, and real-world applications in data processing and retrieval systems.

Tasks

1. Implement Bubble Sort and Selection Sort.
2. Apply Binary Search to find items in an inventory.
3. Compare efficiency of sorting algorithms.
4. Implement Insertion Sort and compare with Bubble Sort.
5. Implement Linear Search and compare with Binary Search.

—

Overview of Sorting and Searching

Sorting and searching are two foundational pillars of computer science. Efficient sorting is essential for faster searching, data analysis, and algorithmic optimization. In this lab, you will see how each sorting and searching algorithm behaves differently under various data conditions.

Unsorted Data: 42 17 8 33 19

Sorted Data: 8 17 19 33 42

—

Task 1: Bubble Sort and Selection Sort

Bubble Sort: Repeatedly compares adjacent elements and swaps them if out of order. **Selection Sort:** Finds the smallest element and swaps it into position. Both have time complexity $\mathcal{O}(n^2)$, making them practical only for small datasets.

```python
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]

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]

a = [64, 25, 12, 22, 11]
b = a.copy()
bubble_sort(a)
print('Bubble Sorted:', a)
selection_sort(b)
print('Selection Sorted:', b)
```

Listing 35: Bubble and Selection Sort

—

Task 2: Binary Search in Inventory

Binary Search efficiently locates elements in sorted data, cutting the search space in half each step.

```python
def binary_search(arr, key):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1

inventory = [(101, 'Pen'), (102, 'Book'), (103, 'Pencil')]
inventory.sort()
ids = [x[0] for x in inventory]
idx = binary_search(ids, 103)
print('Item Found:', inventory[idx][1] if idx != -1 else '
    Not found')
```

Listing 36: Binary Search

Explanation: Binary Search runs in $\mathcal{O}(\log n)$ time and is fundamental to databases, search engines, and indexing.

—

Task 3: Compare Efficiency of Sorting Algorithms

```
import random, time

def time_sort(func, arr):
    a = arr.copy()
    start = time.time()
    func(a)
    return time.time() - start

n = 500
arr = [random.randint(0, 10000) for _ in range(n)]
print('Bubble:', time_sort(bubble_sort, arr))
print('Selection:', time_sort(selection_sort, arr))
print('Python built-in:', time_sort(sorted, arr))
```

Listing 37: Sorting Performance Comparison

Insight: For small datasets, all methods are similar. But as size increases, Python's built-in `sorted()` (Timsort) vastly outperforms $O(n^2)$ algorithms.

—

Task 4: Insertion Sort vs Bubble Sort

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

arr = [5, 2, 9, 1, 5, 6]
insertion_sort(arr)
print('Insertion Sorted:', arr)
```

Listing 38: Insertion Sort

Explanation: Insertion Sort is more efficient for nearly sorted lists and performs in $\mathcal{O}(n)$ for best cases.

—

Task 5: Linear Search vs Binary Search

```python
def linear_search(arr, key):
    for i, v in enumerate(arr):
        if v == key:
            return i
    return -1

arr = [10, 20, 30, 40, 50]
print('Linear Search index:', linear_search(arr, 40))
arr.sort()
print('Binary Search index:', binary_search(arr, 40))
```

Listing 39: Linear vs Binary Search

Explanation: Linear Search is ideal for unsorted data, while Binary Search achieves exponential improvement for sorted lists.

—

Real-World Applications

- **Sorting:** E-commerce product ranking, student grading, inventory management.
- **Searching:** Database indexing, search engines, authentication systems.
- **Insertion Sort:** Used in hybrid sorts like Timsort for small datasets.
- **Binary Search:** Applied in trading systems, autocomplete, and AI decision trees.

—

Complexity Summary

Algorithm	Best Case	Worst Case	Average Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

Table 1: Complexity Comparison of Sorting and Searching Algorithms

—

Performance Insights

- Binary Search requires sorted data, unlike Linear Search.
- Insertion Sort performs well for small or nearly ordered datasets.
- Quadratic sorts are educational but inefficient at scale.
- Modern systems rely on optimized hybrid algorithms.

—

Common Mistakes to Avoid

1. Using Binary Search on unsorted data.
2. Forgetting swap logic in Bubble or Selection Sort.
3. Ignoring loop boundaries.
4. Measuring performance without repeated trials.

—

Viva Questions

1. Explain the differences between Bubble, Selection, and Insertion Sort.
2. Why is Binary Search faster than Linear Search?
3. What happens if Binary Search is used on unsorted data?
4. Compare time complexities of all algorithms discussed.
5. When is Insertion Sort preferred?
6. Where can Linear Search still be useful?
7. Why is data ordering crucial in searching algorithms?

11 Week 11: Advanced Sorting Techniques

Objective: Learn advanced sorting algorithms like Merge Sort, Quick Sort, and Heap Sort. Compare their performance with basic algorithms.

Tasks:

1. Implement Merge Sort.
2. Implement Quick Sort.
3. Compare Merge and Quick Sort with Bubble Sort.
4. Implement Heap Sort.
5. Compare recursive vs iterative Quick Sort.

Details of Lab Experiment:

Task 1: Merge Sort

Merge Sort is a divide-and-conquer algorithm with guaranteed $O(n \log n)$ complexity.

```python
def merge_sort(arr):
    if len(arr)<=1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    i = j = 0
    res = []
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            res.append(left[i])
            i += 1
        else:
            res.append(right[j])
            j += 1
    res.extend(left[i:])
    res.extend(right[j:])
    return res

print(merge_sort([38,27,43,3,9,82,10]))
```

Listing 40: Merge Sort

Task 2: Quick Sort

Quick Sort uses the divide-and-conquer principle based on a pivot element. It has average complexity $O(n \log n)$ but worst-case $O(n^2)$.

```
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)

print(quick_sort([10,7,8,9,1,5]))
```

Listing 41: Quick Sort

Task 3: Comparing Merge, Quick, and Bubble Sorts

Performance testing on large random data highlights the efficiency of advanced algorithms compared to basic ones.

```
import random, time

arr = [random.randint(0,10000) for _ in range(1000)]

t0 = time.time(); sorted(arr)
print('Python Sort:', time.time()-t0)

t0 = time.time(); merge_sort(arr)
print('Merge Sort:', time.time()-t0)

t0 = time.time(); quick_sort(arr)
print('Quick Sort:', time.time()-t0)
```

Listing 42: Sorting Performance Comparison

Task 4: Heap Sort

Heap Sort uses the heap data structure to ensure $O(n \log n)$ performance and avoids recursion.

```
import heapq

def heap_sort(arr):
    h = arr[:]
    heapq.heapify(h)
    res = [heapq.heappop(h) for _ in range(len(h))]
    return res

```

```
print(heap_sort([12,11,13,5,6,7]))
```

Listing 43: Heap Sort

Task 5: Iterative Quick Sort (Concept)

Students can simulate recursion manually using a stack. This method prevents recursion depth errors on large datasets.

```
# Students should design Quick Sort using an explicit stack.
# This simulates recursion manually and improves control
# over stack usage in large data scenarios.
```

Listing 44: Iterative Quick Sort - Concept

Viva Questions

1. Why are Merge and Quick Sort generally faster than Bubble Sort?
2. What is the worst-case complexity of Quick Sort?
3. When would you prefer Heap Sort over Merge Sort?

12 Week 12: Priority Queues and Heaps

Objective: Understand priority queues and heap data structures, and apply them to real-world scheduling and management problems.

Tasks:

1. Implement a Max-Heap.
2. Implement a Min-Heap for scheduling tasks.
3. Simulate a hospital emergency room using a Priority Queue.
4. Implement Heap Sort using Priority Queues.
5. Simulate CPU process scheduling with Priority Queues.

Details of Lab Experiment:

Task 1: Max-Heap Implementation

A Max-Heap ensures the largest element is always on top by inverting values when using Python's built-in `heapq` module.

```
import heapq

arr = [10, 20, 15, 30, 40]
max_heap = [-x for x in arr]
heapq.heapify(max_heap)
print('Max Heap:', [-x for x in max_heap])
```

Listing 45: Max-Heap using heapq

Task 2: Min-Heap for Task Scheduling

Min-Heaps are ideal for scheduling systems where the smallest priority value is executed first.

```
import heapq

pq = []
heapq.heappush(pq, (3, 'Write Report'))
heapq.heappush(pq, (1, 'Fix Bug'))
heapq.heappush(pq, (2, 'Meeting'))

while pq:
    p, name = heapq.heappop(pq)
    print(name, '(Priority', p, ')')
```

Listing 46: Min-Heap for Tasks

Task 3: Hospital Emergency Room Simulation

A Priority Queue can simulate patient treatment order based on severity. Negative values convert the Min-Heap into a Max-Heap.

```
import heapq

patients = []
heapq.heappush(patients, (-5, 'Patient A'))
heapq.heappush(patients, (-9, 'Patient B'))
heapq.heappush(patients, (-2, 'Patient C'))

while patients:
    sev, name = heapq.heappop(patients)
    print(name, 'Severity', -sev)
```

Listing 47: Hospital Emergency Room Simulation

Task 4: Heap Sort using Priority Queue

Heap Sort can be efficiently implemented using the Priority Queue mechanism provided by Python's `heapq`.

```
import heapq

def heap_sort_desc(arr):
    h = []
    [heapq.heappush(h, x) for x in arr]
    res = [heapq.heappop(h) for _ in range(len(h))]
    return res[::-1]  # descending order

print(heap_sort_desc([12, 11, 13, 5, 6, 7]))
```

Listing 48: Heap Sort using heapq

Task 5: CPU Process Scheduling with Priority Queue

Operating systems often use Priority Queues to schedule processes — the higher the priority, the sooner it executes.

```
import heapq

processes = []
heapq.heappush(processes, (-3, 'P1'))
heapq.heappush(processes, (-1, 'P2'))
heapq.heappush(processes, (-5, 'P3'))

while processes:
    pr, pid = heapq.heappop(processes)
    print(pid, '(Priority', -pr, ')')
```

Listing 49: CPU Scheduling Simulation

Viva Questions

1. What is the difference between Min-Heap and Max-Heap?
2. Why are heaps suitable for implementing priority queues?
3. Give real-world scheduling examples that use priority queues.

13 Week 13: Graph Algorithms

Objective: To explore fundamental graph algorithms such as Dijkstra's, Bellman-Ford, Prim's, and Kruskal's algorithms. Students will understand how graphs can represent real-world networks and how these algorithms efficiently find shortest paths and minimum spanning trees (MSTs). The week also focuses on BFS/DFS traversals within social and computer networks.

Tasks:

1. Implement Dijkstra's Algorithm to find the shortest paths in a weighted graph.
2. Implement Kruskal's Algorithm to construct a Minimum Spanning Tree (MST).
3. Apply BFS/DFS to analyze a social network graph (friends-of-friends relationships).
4. Implement Prim's Algorithm for MST using a greedy approach.
5. Implement Bellman-Ford Algorithm for shortest paths (with negative weights).

Conceptual Overview:

Graphs are mathematical structures used to model pairwise relations between objects. Nodes (or vertices) represent entities, and edges denote relationships or connections. Real-world examples include:

- **City maps:** intersections as nodes, roads as edges.
- **Social networks:** users as nodes, friendships as edges.
- **Computer networks:** routers as nodes, links as edges.

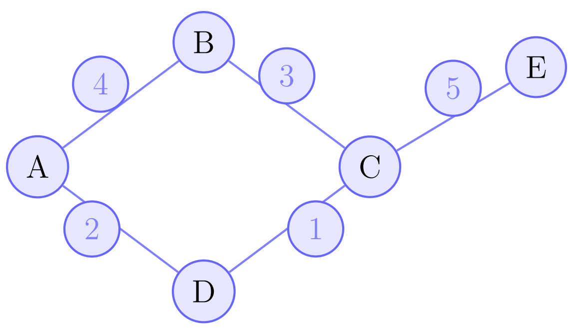


Figure 1: A sample weighted graph used for shortest path and MST experiments.

Details of Lab Experiment

Task 1: Dijkstra's Algorithm

Dijkstra's Algorithm is one of the most fundamental and powerful algorithms in computer science and networking. It is designed to find the **shortest path** from a single source node to all other nodes in a given graph, where each edge has a non-negative weight (or cost).

This algorithm has vast real-world applications such as:

- Network Routing — e.g., determining the fastest path between routers.
- GPS Navigation — e.g., computing the shortest driving route.
- Robotics — e.g., helping a robot plan the most efficient movement path.

Objective: To implement Dijkstra's Algorithm in Python and visualize its working through a graph, understanding how it explores the shortest routes step-by-step.

—

Step-by-Step Explanation

Step 1: Understanding the Graph A graph is composed of nodes (vertices) connected by edges. Each edge has a weight that represents cost or distance. In this lab, we use an **adjacency list** — a simple Python list of lists — to store connections and weights.

Step 2: Initialization We begin by setting the **distance of all nodes** to infinity (`INF`), except for the source node which starts at distance 0. A **priority queue (heapq)** is used to always select the node with the smallest tentative distance next.

Step 3: Relaxation Process This is the core part of Dijkstra's Algorithm. For each node removed from the queue:

1. Check all neighboring nodes connected to it.
2. If a shorter path is found through the current node, update that node's distance.
3. Push the updated distance and node back into the priority queue.

Step 4: Continue Until All Nodes are Finalized The algorithm keeps extracting and relaxing nodes until all reachable vertices have their shortest paths determined.

Step 5: Display Results At the end, the algorithm prints the minimum distance from the source node to every other node in the graph.

—

Python Implementation

```python
import heapq

def dijkstra(adj, src):
    n = len(adj)
    INF = 10**9
    dist = [INF] * n
    dist[src] = 0
    pq = [(0, src)]  # (distance, node)

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))

    return dist

# Adjacency list representation
adj = [
    [(1,10), (4,3)],   # A connects to B (10) and E (3)
    [(2,2), (4,4)],    # B connects to C (2) and E (4)
    [(3,9)],           # C connects to D (9)
    [],                # D has no outgoing edges
    [(1,1), (2,8), (3,2)]  # E connects to B, C, and D
]

print('Shortest distances from A:', dijkstra(adj, 0))
```

Listing 50: Dijkstra's Algorithm in Python

—

Illustrative Example

Let's go through the execution step by step to see how Dijkstra's Algorithm finds the shortest paths:

1. Start at node A. Set `dist[A] = 0`, and all others = .
2. From A, explore its neighbors:
   - A → B = 10
   - A → E = 3

   Now, E (with distance 3) is the closest, so we pick E next.

3. From E:
    - E → B = 3 + 1 = 4 (better than 10! Updated)
    - E → C = 3 + 8 = 11
    - E → D = 3 + 2 = 5
4. Next, pick B (distance = 4):
    - B → C = 4 + 2 = 6 (better than 11! Updated)
5. Continue until all nodes are finalized.

Final Shortest Distances from A:

$$A = 0, \quad B = 4, \quad C = 6, \quad D = 5, \quad E = 3$$

—

Graphical Visualization (Portrait Orientation)

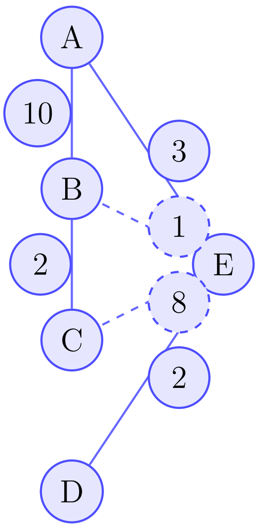


Figure 2: Portrait-Oriented Visualization of Dijkstra's Shortest Path Exploration.

—

Task 2: Kruskal's Algorithm

Kruskal's Algorithm is one of the most important algorithms for finding the **Minimum Spanning Tree (MST)** of a connected, weighted graph. The goal of MST is to connect all the vertices together with the minimum possible total edge weight, without forming any cycles.

This algorithm is especially useful in network design — for example, minimizing the total cost of laying cables or pipelines between cities.

—

Concept Overview

Kruskal's Algorithm works by repeatedly choosing the smallest available edge and adding it to the MST, as long as it doesn't form a cycle. To efficiently check for cycles, it uses the **Union-Find** (also known as **Disjoint Set Union — DSU**) data structure.

- **Edge Sorting:** Sort all edges by their weights in ascending order.
- **Union-Find:** Helps track which vertices are connected to avoid forming cycles.
- **Cycle Avoidance:** Only add an edge if it connects two different components.

—

Step-by-Step Explanation

Step 1: Sort the Edges First, sort all edges in non-decreasing order of their weights. This ensures that we always consider the smallest possible edge next.

Step 2: Initialize Disjoint Sets Each vertex is initially in its own set. We will merge (union) sets as we add edges to the MST.

Step 3: Add Edges One by One Take the smallest edge and check if it connects two different sets (no cycle). If yes — include it in the MST and merge the two sets.

Step 4: Continue Until All Vertices are Connected Stop when the MST has `(n { 1)` edges, where `n` is the number of vertices.

Step 5: Calculate the Total Weight Finally, sum the weights of all selected edges to get the total cost of the MST.

—

Python Implementation

```python
def kruskal(n, edges):
    parent = list(range(n))

    # Find with path compression
    def find(u):
        if parent[u] != u:
            parent[u] = find(parent[u])
        return parent[u]

    # Sort edges by weight
    edges.sort(key=lambda x: x[2])
    mst = []
```

```python
    cost = 0

    for u, v, w in edges:
        pu, pv = find(u), find(v)
        if pu != pv:  # No cycle
            parent[pu] = pv
            mst.append((u, v, w))
            cost += w
    return mst, cost

edges = [
    (0, 1, 10), (0, 2, 6),
    (0, 3, 5), (1, 3, 15),
    (2, 3, 4)
]

print("MST and Total Cost:", kruskal(4, edges))
```

Listing 51: Kruskal's Algorithm in Python

—

Illustrative Example

Let's understand Kruskal's Algorithm using the graph below. Each edge represents a connection between two vertices with a specific weight.

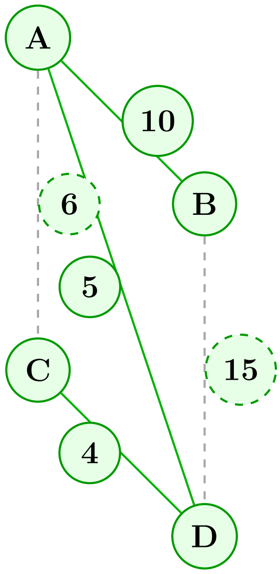
1. Sort all edges: (2–3:4), (0–3:5), (0–2:6), (0–1:10), (1–3:15)
2. Pick (2–3:4): No cycle → Add to MST
3. Pick (0–3:5): No cycle → Add to MST
4. Pick (0–2:6): Forms a cycle → Skip
5. Pick (0–1:10): No cycle → Add to MST

Final MST Edges: (2–3), (0–3), (0–1) **Total Cost:** $4 + 5 + 10 = 19$

—

Graphical Visualization (Portrait-Oriented)

Visualization of Kruskal's MST Construction



Solid Edges: Selected in MST
Dashed Edges: Skipped (cycle or heavier edge)

Figure 3: Kruskal's algorithm selecting minimum-weight edges while avoiding cycles.

Figure 3: Portrait-Oriented Visualization of Kruskal's MST Construction.

—

Task 3: BFS and DFS in a Social Network

Concept Overview: In a social network graph, each node represents a user, and edges represent connections between them.

Breadth-First Search (BFS) explores the graph level by level, visiting all nodes at a given distance from the source before moving to the next level.

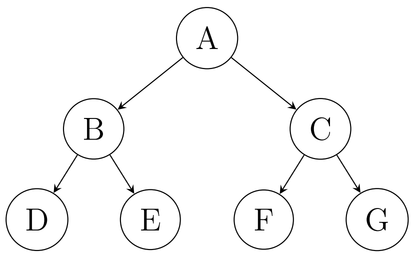
Depth-First Search (DFS) explores the graph by moving along one path as deep as possible before backtracking, which is useful to discover connected chains or clusters in the network.

Step-by-Step Explanation:

Consider the tree structure in the TikZ figure with nodes A, B, C, D, E, F, G:

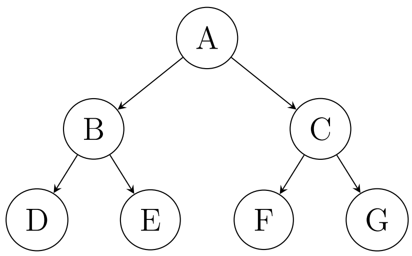
1. **Step 1:** Start at the root node **A**.
2. **BFS Traversal:**

- Visit all nodes at level 1: **B** and **C**.
- Then visit all nodes at level 2: **D**, **E**, **F**, **G**.
- The BFS order from the figure is: **A** → **B** → **C** → **D** → **E** → **F** → **G**.



BFS order: A → B → C → D → E → F → G

3. **DFS Traversal:**



DFS order: A → B → D → E → C → F → G

- Start at **A**, move to **B**, then to **D** (deepest unvisited node).
- Backtrack to **B** and visit **E**.
- Backtrack to **A** and move to **C**, then visit **F** and **G**.
- The DFS order from the figure is: **A** → **B** → **D** → **E** → **C** → **F** → **G**.

4. **Step 2:** Compare BFS and DFS traversal orders:

- **BFS** explores nodes by levels — ideal for shortest-path or "distance from source" analysis.
- **DFS** explores nodes deeply along one path — ideal for connectivity, detecting clusters, or chains in the network.

Python Implementation for Reference:

```python
def bfs(adj, src):
    from collections import deque
    visited=set([src])
    q=deque([src])
    order=[]
    while q:
        u=q.popleft()
        order.append(u)
        for v in adj[u]:
            if v not in visited:
                visited.add(v)
                q.append(v)
    return order

def dfs(adj, src):
    visited=set()
    def dfs_util(u):
        visited.add(u)
        print(u, end=' ')
        for v in adj[u]:
            if v not in visited:
                dfs_util(v)
    dfs_util(src)
    print()

# Adjacency list for the example tree
adj=[[1,2],[3,4],[5,6],[],[],[],[]]
print('BFS:', bfs(adj,0))
print('DFS:', end=' '); dfs(adj,0)
```

Listing 52: BFS and DFS Traversals

Figure: Portrait representation of DFS traversal in a social network.

Figure 4: BFS and DFS traversal in a social network graph (portrait view).

Observation: BFS reveals how influence spreads layer by layer across a social network, while DFS helps identify deeply connected communities. Together, they form the foundation of many network algorithms — such as friend suggestions, influencer detection, and cluster analysis.

Task 4: Prim's Algorithm

Prim's Algorithm starts from an arbitrary node and keeps expanding the MST by selecting the smallest edge connecting a visited and an unvisited node. It works best for dense graphs.

```
import heapq
def prim(adj):
    n=len(adj); INF=10**9
    key=[INF]*n; in_mst=[False]*n; key[0]=0
    pq=[(0,0)]; total=0
    while pq:
        w,u=heapq.heappop(pq)
        if in_mst[u]: continue
        in_mst[u]=True; total+=w
        for v,c in adj[u]:
            if not in_mst[v] and c<key[v]:
                key[v]=c; heapq.heappush(pq,(c,v))
    return total

adj=[[(1,10),(2,6),(3,5)],[(0,10),(3,15)],
     [(0,6),(3,4)],[(0,5),(1,15),(2,4)]]
print('MST Cost (Prim):', prim(adj))
```

Listing 53: Prim's Algorithm

Task 5: Bellman-Ford Algorithm

The Bellman-Ford Algorithm can handle negative edge weights and detect negative cycles. It performs $(n-1)$ relaxations over all edges.

```
def bellman_ford(n, edges, src):
    INF=10**9; dist=[INF]*n; dist[src]=0
    for _ in range(n-1):
        for u,v,w in edges:
            if dist[u]+w < dist[v]:
                dist[v]=dist[u]+w
    return dist

edges=[(0,1,-1),(0,2,4),(1,2,3),(1,3,2),
       (1,4,2),(3,2,5),(3,1,1),(4,3,-3)]
print('Distances:', bellman_ford(5, edges, 0))
```

Listing 54: Bellman-Ford Algorithm

Figure 5: Bellman-Ford Algorithm correctly computes shortest paths even with negative weights.

Viva Questions

1. Differentiate between Dijkstra's and Bellman-Ford algorithms.
2. Compare the approaches of Kruskal's and Prim's algorithms for MST.
3. Why does Dijkstra's Algorithm fail with negative weights?

4. Explain real-world applications of shortest path algorithms.

14 Week 14: Case Study — Real-World Applications

Objective: Integrate multiple data structures and algorithms into practical, real-world systems such as library management, routing, e-commerce, scheduling, and social networks.

Tasks

1. Use hashing and searching to implement a library management system.
2. Apply a graph algorithm to model and solve a transportation route problem.
3. Compare different data structures for handling an e-commerce order system.
4. Design a scheduling system for university courses using priority queues.
5. Build a mini social network model using graphs and hash tables.

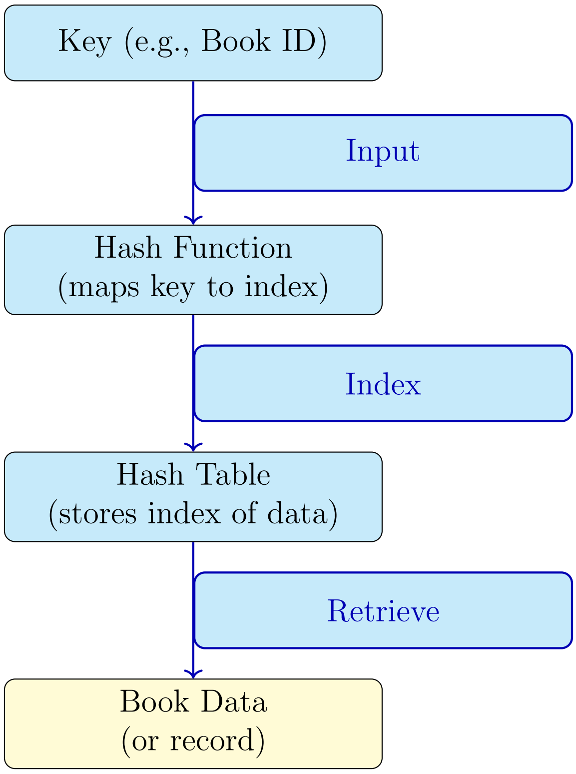
—

Task 1: Library Management System

Books are stored using hashing (Python dictionaries). Searching becomes $\mathcal{O}(1)$ on average.

```
library = {101: 'Data Structures', 102: 'Algorithms', 103: 'Database Systems'}
id = int(input('Enter book ID to search: '))
print('Book found:' , library[id] if id in library else 'Book not found!')
```

Listing 55: Library Management

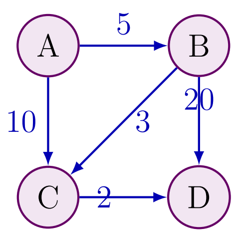


Task 2: Transportation Routing (Dijkstra)

Road networks are modeled as weighted graphs. Dijkstra's Algorithm finds the fastest path.

```python
adj = [[(1,5),(2,10)],[(2,3),(3,20)],[(3,2)],[]]
print(dijkstra(adj, 0))
```

Listing 56: Transportation Routing using Dijkstra



—

Task 3: E-commerce Order System

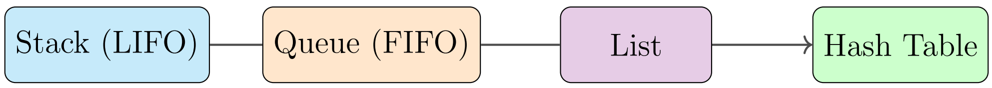
Different data structures serve different roles: Stacks (last-in orders), Queues (first-in orders), Lists (general storage), Hash Tables (quick lookup).

```python
from collections import deque
order_stack = []; order_queue = deque(); order_list = [];
    order_hash = {}

order_stack.append('Order#101'); order_stack.append('Order
    #102')
order_queue.append('Order#201'); order_queue.append('Order
    #202')
order_list.append('Order#301'); order_list.append('Order#302
    ')
order_hash[401] = 'Order#401'; order_hash[402] = 'Order#402'

print('Stack Top:', order_stack[-1])
print('Queue Front:', order_queue[0])
print('List Front:', order_list[0])
print('Hash Search (402):', order_hash[402])
```

Listing 57: E-commerce Order Handling



—

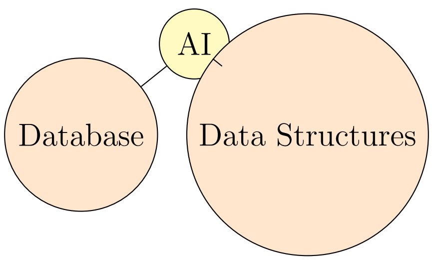
Task 4: Course Scheduling

Priority Queues ensure high-priority courses (core subjects) are scheduled first.

```python
import heapq
pq = []
heapq.heappush(pq, (1, 'AI'))
heapq.heappush(pq, (3, 'Data Structures'))
heapq.heappush(pq, (2, 'Database'))

while pq:
    p, c = heapq.heappop(pq)
    print(c)
```

Listing 58: Course Scheduling using Priority Queue



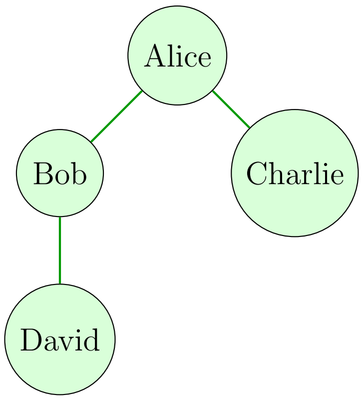
—

Task 5: Mini Social Network

Graphs and hash tables simulate friendships and quick lookup.

```
network = {'Alice':['Bob','Charlie'],
           'Bob':['Alice','David'],
           'Charlie':['Alice'],
           'David':['Bob']}
print("Alice's Friends:", ' '.join(network['Alice']))
```

Listing 59: Mini Social Network Simulation



—

Viva Questions

1. How does hashing improve library lookups?
2. Which data structure is most suitable for real-time e-commerce orders?
3. How would you extend routing to account for live traffic data?

15 Week 15: Complex Problem-Solving Lab

Objective: Students will develop a modular system to solve a real-world complex problem by integrating multiple data structures and algorithms. The case study focuses on a **Smart City Traffic Management System**.

Problem Statement and Tasks

Design a system capable of dynamically managing urban traffic flow, routing emergency vehicles efficiently, controlling parking, and recording violations. Students must combine several data structures for optimal performance.

- **Road networks:** Modeled as **Graphs** to represent intersections and connecting roads.
- **Vehicle priorities:** Managed using **Priority Queues** for ambulances, VIP convoys, and regular vehicles.
- **Shortest-path rerouting:** Computed with **Dijkstra's Algorithm** for congestion avoidance and emergency routing.
- **Parking management:** Implemented with **Hash Tables** to track slot availability in real time.
- **Violation records:** Stored using **Binary Search Trees** for fast insertion and lookup.

—

System Architecture Overview

The architecture integrates all modules to allow smooth traffic operation, real-time vehicle prioritization, and efficient data management.

Figure: System architecture showing interactions between road network, vehicle priority management, routing, parking, violation logging, and central traffic control.

—

Sample Implementation (Abridged)

```python
# --- Road network modeled as graph (adjacency list) ---
adj = [[(1,5),(2,2)], [(0,5),(2,3)], [(0,2),(1,3)]]

# --- Vehicle priority queue (higher priority = smaller number) ---
import heapq
vehicles = []
heapq.heappush(vehicles, (-3, 'Ambulance'))
heapq.heappush(vehicles, (-1, 'Car'))
heapq.heappush(vehicles, (-2, 'Bus'))

# --- Parking system using hashing ---
parking = {'A1':'Free', 'A2':'Occupied', 'B1':'Free'}

# --- Violation records using BST ---
# class Node, insert(), search() etc.
```

Listing 60: Smart City Traffic Management System

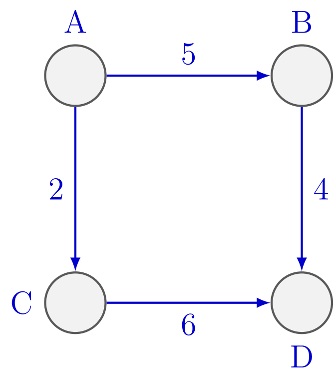


Figure: Simplified road network used in Dijkstra's routing algorithm.

—

Concept Highlights

Graphs: Represent road intersections and road lengths, providing a framework for shortest-path calculations.

Priority Queues: Ensure emergency vehicles are routed first by maintaining a dynamically sorted queue.

Dijkstra's Algorithm: Computes optimal routes for vehicles based on current traffic conditions, avoiding congested or blocked roads.

Hash Tables: Enable constant-time lookup for available parking slots, facilitating quick parking decisions.

Binary Search Trees: Efficiently insert, search, and update traffic violation records.

—

Viva Questions

1. Why are graphs ideal for modeling road networks in smart cities?
2. How do priority queues help ensure timely routing of emergency vehicles?
3. Why are hash tables efficient for managing parking slot availability?
4. What advantages do Binary Search Trees offer for storing violation records?
5. Explain how Dijkstra's algorithm updates routes dynamically when traffic conditions change.