UNIT 4

Computational Structures

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Define and explain the purpose of primitive computational structures, including lists, stacks, queues, trees, and graphs.
- Identify and describe the characteristics and properties of different computational structures.
- Perform basic operations such as insertion, deletion, traversal, and searching on various computational structures.
- Understand and implement the LIFO (Last-In, First-Out) and FIFO (First-In, First-Out) principles in stacks and queues, respectively.
- Compare and contrast different types of trees and graphs, and apply appropriate operations to them.
- Analyze and choose the most suitable computational structure based on problem requirements, data organization, and performance considerations.
- Apply computational structures in real-world scenarios, including data organization, task scheduling, and network modeling.
- Combine different computational structures to solve complex problems and enhance functionality.

Introduction

In this chapter, we will explore key computational structures, such as lists, stacks, queues, trees, and graphs, which are fundamental in programming. We will examine their properties, operations, and how to implement them efficiently. Additionally, we will discuss selecting the appropriate structure based on specific problem requirements and demonstrate their application in real-world scenarios.

4.1 Primitive Computational Structures

There are following commonly used computational software:

4.1.1 Lists

A list is a data structure used to store multiple pieces of data in a specific sequence. Each piece of data, known as an element, is positioned at a particular index within the list, facilitating easy access and management.

4.1.1.1 List Creation

In Python, Lists are created using square brackets '[]', with each item separated by a comma.

```
# Create a list of items
items= [ "Decorations" ,  "Snacks" ,  "cold drinks", "Plates", "Balloon"]
# Print the list
print(items)
```

In the above code:
"Items" is the name of the list. The items inside the list are "Decorations", "Snacks", "Cold drinks", "Plates", and "Balloons". Each item is enclosed in quotes (for text) and separated by a comma.

4.1.1.2 List Properties

List has following properties:

1. **Dynamic Size** A list in Python can change its size. You can add new items to the list or remove items without any problem. The list will automatically adjust to fit the changes.
2. **Index-Based Access** Every item in a list has a position, called an index. The first item has an index of 0, the second item has an index of 1, and so on. You can use these indexes to get specific items from the list.
3. **Ordered Collection** The order in which you add items to a list is preserved. This means that if you add an item first, it will stay in that position unless you change it.

4.1.1.3 List Operations:

Some common operations of a list are:

1. **Insertion:** Adding a new item to your list is like adding a new task to your to-do list. You can insert an item at different positions in the list. You can insert an item at any position in the list using the 'insert( )' function.

```
party_list = ["Buy drinks", "Buy decorations", "Buy snacks", "Buy cold drinks "]
party_list.insert (0 , "Invite friends") # add Invite friends at start
print(party_list)
# Output: ["Buy drinks, "Buy decorations", "Buy snacks", "Buy cold drinks "]
```

2. **Deletion:** Removing an item from your list is like crossing off a task you've completed. You can remove items in various ways
   a. **Removing by Value:** Use the 'remove()' function to delete the first occurrence of a specific item.

```
party_list = ["Invite friends ","Buy decorations", "Buy snacks", "Buy cold drinks"]
party_list.remove ("Buy snacks") # Removes 'Buy snacks ' from the list
print(party_list)
# Output: ['Invite friends', 'Buy decorations', 'Buy cold drinks ']
```

b. Removing by Index: Use the 'pop( )' function to remove an item at a specific index.

```
party_list = ["Invite friends ", "Buy decorations", "Buy cold drinks " ]
party_list.pop (0) # Removes the item at index 0
print(party_list)
# Output: ['Buy decorations', 'Buy cold drinks']
```

3. Searching: Finding an item in a list is similar to looking for a specific task in your to-do list. You can search for an item using different functions: Use the 'in' keyword to check if an item exists in the list.

```
party_list = ["Invite friends", "Buy decorations", "Buy cold drinks"]
if "Buy cold drinks" in party_list:
    print("Buy cold drinks is on the list.")  # Prints if 'Buy cold drinks' is found
else:
    print("Buy cold drinks is not on the list.")
# Output: Buy cold drinks is on the list.
```

4.1.1.4 Applications of Lists

- **Data Storage and Manipulation:** Lists are commonly used to store and manage collections of data, such as records, entries, or values. They allow for easy insertion, deletion, and access to elements.
- **Stack and Queue Implementations:** Lists can be used to implement stack (LIFO) and queue (FIFO) data structures, which are fundamental for various algorithms and tasks in computing.

4.1.2 Stacks

A stack is a simple data structure where you can only add or remove items from one end, known as the "top". Both insertion and deletion of elements occur at this top end. A stack operates on the Last-In, First-Out (LIFO) principle, meaning that the most recently added element is the first one to be removed.

Figure 4.1: Stack of Books

4.1.2.1 Stack Operations

There are two basic operations in a stack:

- **Push Operation:** Push means adding an item to the top of the stack.
- **Pop Operation:** Pop means removing the item from the top of the stack.

```
#    Create an empty stack of books
stack_of_books = []
print("Initial stack:", stack_of_books)          # Empty stack
#    Add books to the stack (push operation)
print("\n Adding books to the stack (push operation):")
stack_of_books.append('Book A')
print("Stack after pushing 'Book A':", stack_of_books)
stack_of_books.append('Book B')
print("Stack after pushing 'Book B':", stack_of_books)
# Remove the top book from the stack (pop operation)
print("\nDeletion of top book (pop operation):")
top_book = stack_of_books.pop()
print("Removed book:", top_book)
print("Stack after popping the top book:", stack_of_books)
```

The code creates an empty stack to hold books. It then adds books ("Book A" and "Book B") one by one to the top of the stack. Finally, it removes the top book "Book B" from the stack, showing how the last book added is the first one taken off.

4.1.3 Queues

A queue is like a line in front of a bank or a ticket counter. The first person to get in line is the first person to be served. In a computer, a queue works the same way. It keeps track of things so that the first item added is the first one to be taken out. Just like in a bank line, you add things to the back and remove them from the front, following the FIFO (First-In, First-Out) principle, as shown in Figure 4.2.

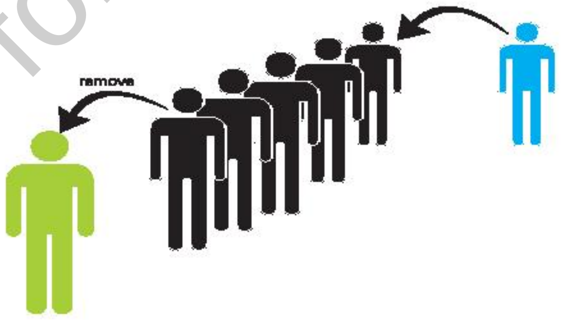

Figure 4.2: Queue of persons in front of the bank

4.1.3.1 Queue Operations

Queues support two primary operations:

- **Enqueue (Add an Item):** This is like adding a person to the end of the line. In a queue, you add items to the back.
- **Dequeue (Remove an Item):** This is like serving the person at the front of the line. In a queue, you take items out from the front. Additional operations might include checking if the queue is empty, retrieving the element at the front without removing it, and determining the size of the queue.

```
# Built-in module to implement queues in Python
from queue import Queue
# Create a new queue
q = Queue ()
# Add people to the queue (Enqueue)
q. put (" Ahmed") # Adds Ahmed to the end of the queue
q.put ("Fatima")   # Adds Fatima to the end of the queue
# View the person at the front of the queue (Peek)
front_person = q. queue [0] # Looks at the person at the front without
removing them
print(front_person) #Remove a person from the front of the queue(Dequeue)
removed_person = q.get ( ) # Removes and returns the person at the front
of the queue
print(removed_person)
# Add another person to the queue (Enqueue)
q.put("Sara") # Adds Sara to the end of the queue
# View the updated queue
updated_queue = list(q.queue)
print(updated_queue)
```

The code manages a line of people using a queue. It adds people to the end of the line, checks who is at the front without removing them, and then serves (removes) the person at the front. Finally, it adds another person to the end and shows the updated line.

4.1.4 Trees

A tree data structure organizes information in a way that spreads out from a main point called the root node. In a tree, each piece of information, called a node, can connect to other pieces, which are also nodes, forming a branching structure. This branching structure is different from a list, where items are organized one after the other in a straight line.

Example: In a family tree, the oldest ancestors represent the root node, serving as the starting point of the hierarchy. Each individual in the tree may have descendants, forming subsequent levels of the hierarchy, as illustrated in Figure 4.3. This hierarchical structure is not suitable for storage in a linear format, such as a list, due to the complex parent-child relationships. Therefore, a tree data structure is employed to efficiently store and access such hierarchical data, enabling clear representation and retrieval of information.

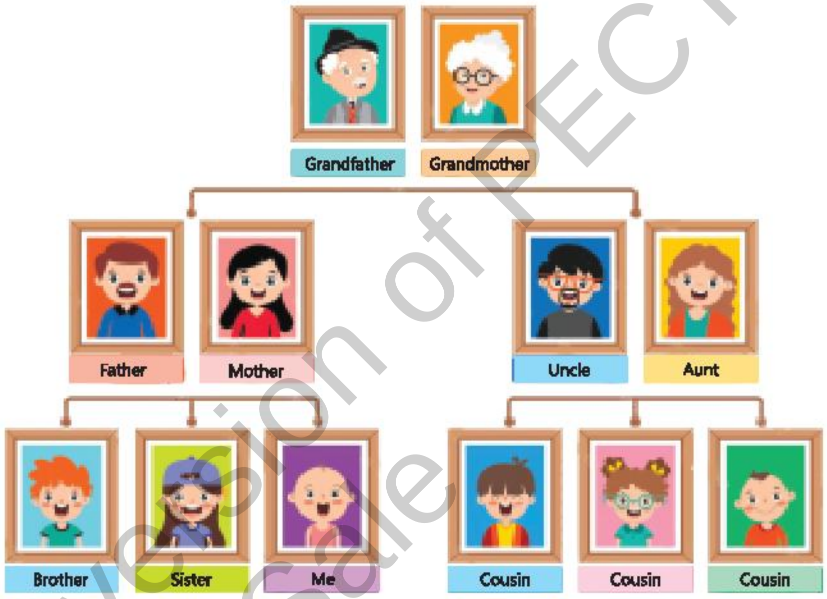

Figure 4.3: Family Tree

4.1.4.1 Properties of Trees

1. **Root Node:** The root is the very first or top node in a tree, like the main folder in a computer where all other folders and files are contained.
2. **Edges and Nodes:** Nodes are the individual elements in the tree, and they are connected by lines called edges. A node without any child nodes is called a leaf, similar to a file in a folder that doesn't contain any other files.
3. **Height:** The height of a tree is the longest path from the root node down to the farthest leaf. It tells us how deep or tall the tree is.
4. **Balanced Trees:** A tree is considered balanced if the branches on the left and right sides are nearly the same height.

4.1.4.2 Applications of Trees

1. **File Systems:** Pre-order tree traversal is useful for creating backups of file systems. By visiting the root first and then recursively backing up each directory, it ensures that directories are backed up before their contents.
2. **File System Deletion:** In file systems, Post-order traversal ensures that files and directories are deleted in the correct order; by first deleting all sub directories and files before deleting the parent directory.
3. **Hierarchical Data Representation:** Trees are used in representing data with a clear hierarchical relationship, such as organisational charts and family trees.
4. **Decision Making:** Trees, such as decision trees, are used in algorithms to make decisions based on various conditions and outcomes.

4.1.5 Introduction to Graphs

A Graph is a data structure that consists of a set of vertices (or nodes) connected by edges. Graphs are used to represent networks of connections, where each connection is a relationship between two vertices. These vertices can represent anything, like cities, people, or even abstract concepts, and the edges represent the relationships or pathways between them.

Imagine you are mapping out all the cities in Pakistan and the roads that connect them. Each city is a vertex, and each road between two cities is an edge. Unlike a tree, a graph does not have a single "root" and does not follow a hierarchical structure. In a graph, any two vertices can be connected, creating a complex web of relationships.

Example: In a social network, each person can be connected to many others, forming a graph. There is no single starting point, and people (vertices) can have multiple connections (edges) that do not follow a strict parent-child relationship like in a tree.

Difference from a Tree: While both graphs and trees are used to represent relationships between objects, a tree is a special kind of graph with some important differences:

- A Tree is hierarchical, meaning it has a single root node from which all other nodes branch out. In contrast, a Graph does not necessarily have a hierarchy or a root.
- In a Tree, there is exactly one path between any two nodes, ensuring no cycles (loops). However, in a Graph, there can be multiple paths between nodes, and cycles are allowed.
- Trees are often used to represent structured data like family trees or organizational charts. Graphs are more flexible and can represent a broader range of connections, such as networks, web links, or transport systems.

4.1.5.1 Characteristics of Graphs

Graphs have several defining features that help us understand and use them effectively:

4.1.5.2 Properties of Graphs

Graphs also have specific details that describe their structure:

- **Degree:** This is the number of edges connected to a vertex. For instance, if a city is connected to three other cities, the degree of that city's vertex is 3.
- **Weight:** In some graphs, edges have weights that represent values like distances or costs. For example, if a road between two cities is 50 kilometres long, its edge might have a weight of 50.
- **Direction:** Edges can be either directed or undirected. Directed edges have a one-way connection, meaning a road from city A to city B does not necessarily have a return road from B to A. Undirected edges represent a two-way connection.

4.1.5.3 Types of Graphs

Graphs can be classified into several types based on their structure and properties. The main types of graphs are directed, undirected, and weighted. Each type has its own characteristics, which can be better understood through simple examples.

- **Directed Graphs:** In a directed graph, edges have a direction, which means they go from one vertex to another in a specific way as shown in Figure 4.4.

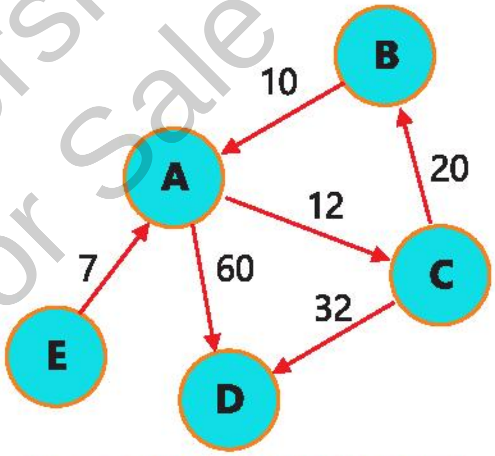

Figure 4.4: Directed Weighted Graph

Example: Consider a graph shown in Figure 4.4. If you want to travel from city A to city B, you can only go in the direction permitted by the city's sign. If there's no one-way street going from city A to city B, you cannot travel directly from city A to B.

- **Undirected Graphs:** In an undirected graph, edges do not have a direction. This means that if there is a connection between two vertices, you can travel in both directions.
  Example: Consider a graph shown in Figure 4.5, if Person A is friends with Person B, then Person B is also friends with Person A. There is no restriction on the direction of the friendship, so you can move freely between friends.

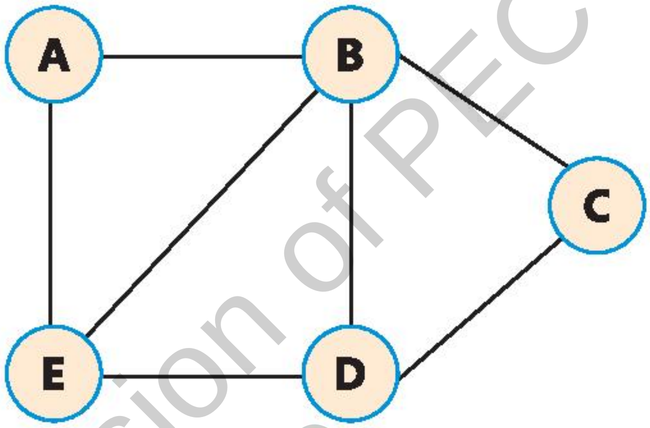

Figure 4.5: Undirected Graph

- **Weighted Graphs:** In a weighted graph, each edge has a weight or cost associated with it. This weight represents the distance, time, or cost required to travel from one vertex to another as shown in Figure 4.4.
  Example: Imagine a map of a city where each road has a different distance or travel time. If you want to travel from one landmark to another, the map provides the distance or travel time for each road. This information helps you determine the shortest or quickest route between landmarks.

EXERCISE

Multiple Choice Questions

1. The function used to add an item at the end of a list in Python:

   a) insert( ) b) append( ) c) remove( ) d) pop( )

2. The purpose of the in keyword used with a Python list:

   a) Adds an item to the list b) Removes an item from the list

   c) Checks if an item exists in the list d) Returns the length of the list

3. An operation that removes an item from the top of the stack:

   a) Push b) Pop c) Peek d) Add

4. The operation used to add an item to a queue:

   a) Dequeue b) Peek

   c) Enqueue d) Remove

5. True statement about the height of a tree:

   a) Number of edges from the root to the deepest node

   b) Number of nodes from the root to the deepest node

   c) Number of children of the root node

   d) Always equal to the number of nodes in the tree

6. A scenario where a graph data structure is most suitable:

   a) Managing a to-do list

   b) Modeling a line of customers in a store

   c) Representing connections in a social network

   d) All of the above

Short Questions

1. Explain how the ' insert( ) ' function works in python lists. Provide an example.

2. Explain the potential issues which could arise when two variables reference the same list in a program? Provide an example.
3. Define a stack and explain the Last-In, First-Out (LIFO) principle.
4. Differentiate between the Enqueue and Dequeue operations of queue.
5. Name two basic operations performed on stack
6. What is difference between enqueue ( ) and dequeue ( ).

Long Questions

1. Discuss the dynamic size property of lists in Python. How does this property make lists more flexible?
2. Explain the operations on stack with real-life example and Python code.
3. Write, a simple program to implement a queue (insertion and deletion).
4. Define Tree and explain its properties
5. What is a graph? Explain differences between directed and undirected graphs.