Introduction to Elementary Data Structures

Elementary data structures are foundational building blocks in programming, designed to store and organize data efficiently. They form the basis of more complex data structures and algorithms, enabling efficient data manipulation, access, and storage. Choosing the right data structure is crucial as it directly impacts an algorithm's efficiency, especially in terms of time and space complexity.

Elementary data structures are generally categorized into **linear** and **non-linear** structures, each suited for specific types of tasks and operations.

Types of Elementary Data Structures

1. **Linear Data Structures**

In linear data structures, elements are arranged sequentially, where each element is directly connected to its previous and next element. This allows for straightforward traversal and is ideal for scenarios where order is important. Common linear data structures include:

- **Arrays**: Fixed-size structures that store elements of the same type in contiguous memory locations. Access is fast (constant time), but **resizing is challenging.**
- **Linked Lists**: Dynamic structures where each element, called a node, contains data and a reference to the next node. They allow efficient insertion and deletion but require sequential access.
- **Stacks**: Follows the Last In, First Out (LIFO) principle, making it ideal for managing function calls, undo mechanisms, and more. Operations like push and pop are done only at the top of the stack.
- **Queues**: Follows the First In, First Out (FIFO) principle, making it suitable for scenarios like task scheduling and handling requests. Enqueue adds elements to the rear, while dequeue removes elements from the front.

2. **Non-Linear Data Structures**

Non-linear data structures allow hierarchical organization, enabling more complex relationships between elements, such as parent-child and sibling relationships. They’re suited for tasks like searching and hierarchical representation.

- **Trees**: Consist of nodes connected in a hierarchical manner, with a root node and sub-nodes, forming branches. Common types include binary trees, binary search trees, AVL trees, and heaps. Trees are widely used in databases, file systems, and for representing hierarchical data.
- **Graphs**: Consist of nodes (vertices) connected by edges, which may be directed or undirected. Graphs are ideal for representing networks such as social connections, web pages, or transport routes.

Summary

Elementary data structures, whether linear or non-linear, provide a framework for efficient data organization and manipulation, forming the backbone of effective algorithm design and data management.

Key Words:

Contiguous memory refers to a block of memory locations that are physically adjacent in the computer's memory. In this arrangement, data elements are stored sequentially, one after the other, in a continuous stretch of memory.

```
In [ ]:
```

What is a Stack?

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle, meaning the last element added to the stack is the first one to be removed. It resembles a stack of plates where the plate added last is taken out first.

Properties of a Stack

1. **LIFO Principle**: The last element pushed onto the stack is the first one popped out.
2. **Dynamic Nature**: The size of a stack can grow or shrink dynamically (in implementations using dynamic memory).
3. **Basic Operations**:
   - **Push**: Add an element to the top of the stack.
   - **Pop**: Remove and return the top element of the stack.
   - **Peek/Top**: View the top element without removing it.
   - **IsEmpty**: Check if the stack is empty.
4. **Restricted Access**: Elements are only added or removed from the top of the stack.
5. **Fixed Size (in some implementations)**: Some stacks (e.g., array-based) have a predefined maximum size.

Stack Implementation in Python

Python offers two primary ways to implement a stack:

1. Using **lists**.
2. Using the `collections.deque` for better performance in larger operations.

1. Stack Using Python Lists

Python lists can be used as stacks since they support append and pop operations.

```python
# Stack using Python list
stack = []

# Push operation
stack.append(10)
stack.append(20)
stack.append(30)

# Pop operation
print("Popped element:", stack.pop())  # Output: 30

# Peek operation
print("Top element:", stack[-1])  # Output: 20

# Check if stack is empty
print("Is stack empty?", len(stack) == 0)  # Output: False

# Final stack
print("Current Stack:", stack)  # Output: [10, 20]
```

Limitations:

- Not ideal for large stacks due to potential inefficiency with resizing.

---

2. Stack Using `collections.deque`

The `deque` class from the `collections` module is preferred for implementing stacks as it is optimized for fast appends and pops.

```python
from collections import deque

# Stack using deque
stack = deque()

# Push operation
stack.append(10)
stack.append(20)
stack.append(30)

# Pop operation
print("Popped element:", stack.pop())  # Output: 30

# Peek operation
print("Top element:", stack[-1])  # Output: 20

# Check if stack is empty
print("Is stack empty?", len(stack) == 0)  # Output: False

# Final stack
print("Current Stack:", list(stack))  # Output: [10, 20]
```

---

3. Stack Using Arrays (`numpy`)

For stacks where fixed-size arrays are preferred, you can use the `numpy` library. However, this is less common for dynamic stack implementations.

```python
import numpy as np

# Create a stack with a fixed size
stack = np.array([None] * 5)  # Fixed size of 5
top = -1  # Initialize the top pointer

# Push operation
def push(element):
    global top
    if top == len(stack) - 1:
        print("Stack Overflow!")
        return
    top += 1
    stack[top] = element

# Pop operation
def pop():
    global top
    if top == -1:
        print("Stack Underflow!")
        return None
    element = stack[top]
    stack[top] = None
    top -= 1
    return element

# Testing
push(10)
push(20)
push(30)
print("Popped element:", pop())  # Output: 30
print("Top element:", stack[top])  # Output: 20
print("Stack:", stack)  # Output: [10, 20, None, None, None]
```

Applications of Stacks

- **Expression Evaluation and Conversion**: Infix to postfix or prefix.
- **Undo Mechanisms**: In text editors.
- **Backtracking**: Solving mazes or recursion.
- **Function Call Management**: Used in function call stacks in programming.

Each approach provides flexibility based on requirements such as performance and fixed size. For general use, `deque` is highly recommended.

```
In [ ]:
```

Queues

Definition of Queue

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. This means that the first element inserted into the queue is the first one to be removed, similar to a real-world queue (e.g., a line at a ticket counter).

Properties of a Queue

1. **FIFO Principle**: The element added first will be removed first.
2. **Enqueue Operation**: Adds an element to the end of the queue.
3. **Dequeue Operation**: Removes an element from the front of the queue.
4. **Peek/Front**: Returns the front element without removing it.
5. **IsEmpty**: Checks if the queue is empty.
6. **IsFull** (in bounded implementations): Checks if the queue has reached its maximum capacity.

Types of Queues

1. **Simple Queue**:
    - Follows FIFO.
    - Operations occur at opposite ends (enqueue at the rear, dequeue at the front).
2. **Circular Queue**:
    - The rear connects back to the front to form a circle.
    - Efficient in utilizing memory.
3. **Priority Queue**:
    - Elements are dequeued based on priority, not just the FIFO principle.
4. **Double-Ended Queue (Deque)**:
    - Allows insertion and deletion from both ends.

Queue Implementation in Python

1. **Using Lists**:

```python
# Implementation of Queue using Python List
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)  # Add item at the rear

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)  # Remove item from the front
        return "Queue is empty!"

    def peek(self):
        if not self.is_empty():
            return self.queue[0]  # Return the front element
        return "Queue is empty!"

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

    def display(self):
        return self.queue

# Example Usage
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.display())  # Output: [1, 2, 3]
print(q.dequeue())  # Output: 1
print(q.display())  # Output: [2, 3]
```

2. **Using `collections.deque`**:

```python
from collections import deque

# Initialize a queue
queue = deque()

# Enqueue elements
queue.append(1)
queue.append(2)
queue.append(3)

# Display the queue
print(queue)  # Output: deque([1, 2, 3])

# Dequeue elements
print(queue.popleft())  # Output: 1
print(queue)            # Output: deque([2, 3])

# Peek
print(queue[0])  # Output: 2
```

3. **Using `queue.Queue`**:

```python
from queue import Queue

# Initialize a queue
q = Queue(maxsize=5)

# Enqueue elements
q.put(1)
q.put(2)
q.put(3)

# Display size
print(q.qsize())  # Output: 3

# Dequeue elements
print(q.get())  # Output: 1
print(q.qsize())  # Output: 2
```

Applications of Queues

1. **Resource Management**: Job scheduling, printer task queues, CPU scheduling.
2. **Data Transfer**: Buffers in IO operations and streaming.

3. **Breadth-First Search (BFS)** in graphs and trees.
4. **Customer Service Systems**: Handling requests or calls in a sequence.

Let me know if you want a detailed explanation of any specific implementation or concept! 😊

```
In [ ]:
```