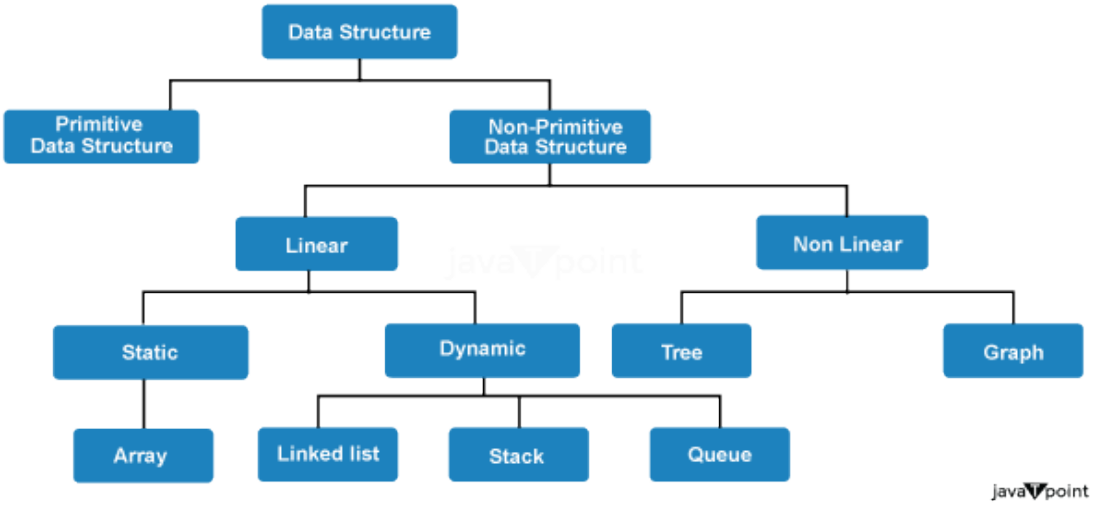
Primitive vs non primitive data structure

Introduction

In computer science and programming, data structures are important for effectively organizing and storing data. They are treated as the building blocks for developing efficient algorithms which optimize program performance. Data structures can be classified into two categories: primitive and non-primitive. This article will discuss the characteristics, applications, and differences between these two types of data structures.

There are two types of data structures.

- Primitive Data Structures
- Non-primitive Data Structure



Differences between Primitive and Non-Primitive Data Structures:

Primitive	Non- Primitive
Primitive data structures have a fixed size and consume constant memory, regardless of the amount of data stored.	Non-primitive data structures require additional memory to store metadata and provide a dynamic allocation of memory based on the data size.
Primitive data structures have limited flexibility in terms of operations and data organization.	Non-primitive data structures offer more flexibility, allowing complex operations, dynamic resizing, and hierarchical relationships between data elements.
Primitive data structures are simple and straightforward, with basic operations and constant time complexity.	Non-primitive data structures offer more flexibility, allowing complex operations, dynamic resizing, and hierarchical relationships between data elements.
Primitive data structures are straightforward, with basic operations and constant time complexity.	Non-primitive data structures can be more complex, with varying time complexity for different operations, depending on the structure's size and organization.
Primitive data structures are usually built into programming languages, with direct support from compilers and interpreters.	Non-primitive data structures require custom implementation using primitive data types and programming constructs.

Primitive Data Structures:

Primitive data structures are fundamental and built-in data types provided by programming languages. They are typically simple and have a fixed size. Examples of primitive data structures include integers, floating-point numbers, characters, booleans, and pointers. Here are some key aspects of primitive data structures:

- Primitive data structures are atomic, meaning they cannot be broken down into smaller components.
- They have a predefined range of values and operations associated with them.

- The hardware directly supports them and is often represented using a fixed number of bits.
- They are lightweight in terms of memory consumption and are optimized for performance.

Primitive data structures are commonly used for simple and basic operations in programming, such as arithmetic calculations, logical comparisons, and storing individual data elements. They serve as the building blocks for more complex data structures and algorithms.

Example

Integer

```
# Declare and assign an integer
num = 10
```

Floating-point Number

```
# Declare and assign a floating-point number
pi = 3.14159
```

Character

```
# Declare and assign a character
char = 'A'
```

Boolean

```
# Declare and assign a boolean value
is_true = True
```

Array

```
# Declare and assign an array
numbers = [1, 2, 3, 4, 5]
```

Applications:

- Primitive data structures are widely used in various programming scenarios, such as arithmetic operations, logical comparisons, and basic data manipulation.
- They serve as the foundation for more complex data structures and algorithms.
- Primitive data structures are essential for low-level programming, requiring direct control over memory and hardware.

Non-Primitive Data Structure

Non-primitive data structures are derived from primitive data types and offer more flexibility in storing and organizing data. They are also known as composite or abstract data types. Non-primitive data structures can be categorized into two main types: linear and nonlinear. Let's explore their characteristics and applications:

Further, Non-primitive Data Structure is classified into two types

- Linear
- Non-Linear

Linear Data Structures:

- Linear data structures organize data elements sequentially, one after another. Examples include arrays, linked lists, stacks, and queues.
- They allow easy access to data elements using indexes or pointers.
- Linear data structures are useful for scenarios involving iterative processing, searching, sorting, and managing data in a specific order.

Furthermore, Linear is Divided into two parts

- Statics
- Dynamic

Static: Fixed-size collections of elements stored in contiguous memory locations. The size is predetermined at compile time and cannot be changed during runtime.

- **Array -** An array is a collection of elements of the same type arranged in contiguous memory locations. Once the size of an array is defined, it cannot be changed during program execution.

Dynamic: Resizable arrays that can grow or shrink dynamically as needed. They allocate a new block of memory when the capacity is exceeded.

Now Dynamic is Classified into three types

- Linked List
- Stack
- Queue

- **Linked List -** A linked list is a linear data structure in computer science; linked lists are widely used in various applications and are fundamental to understanding data structures and algorithms. The linked list is called "linked" because these references connect or link the nodes.

Implementation of a Linked List data structure in Python

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

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

    def append(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.value, end=" ")
            current = current.next
        print()

# Example usage
linked_list = LinkedList()
linked_list.append(10)
linked_list.append(20)
linked_list.append(30)
linked_list.display()
```

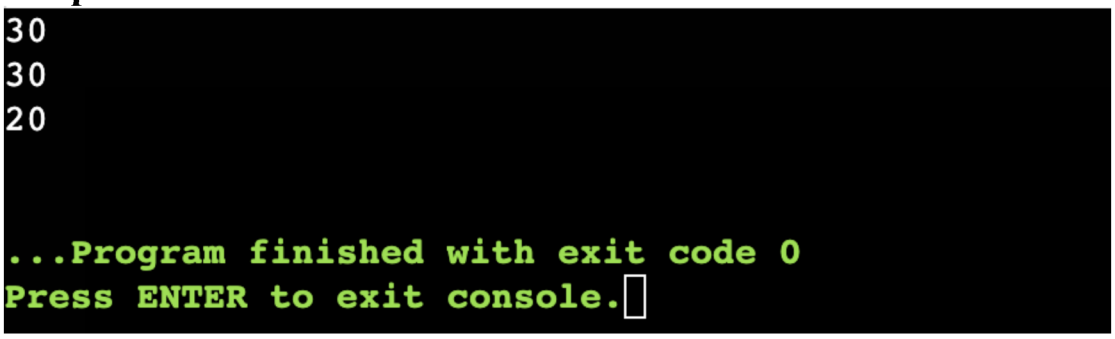
Output

- **Stack:** A Last-In-First-Out (LIFO) data structure where elements are added and removed from the top of the stack. It follows a "push" operation to add elements and a "pop" operation to remove elements.

Implementation of a stack data structure in Python

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if self.is_empty():
            return None
        return self.stack.pop()

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

    def peek(self):
        if self.is_empty():
            return None
        return self.stack[-1]

# Example usage
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
print(stack.peek())  # Output: 30
print(stack.pop())  # Output: 30
print(stack.pop())  # Output: 20
```

Output



- **Queue -** A queue is an abstract data type that works on First-In-First-Out (FIFO) principle. The first element inserted is the first to be removed, similar to people waiting in a queue. Queues are used in scenarios where the order of insertion and removal is significant, such as scheduling, resource allocation, and breadth-first search algorithms. It follows an enqueue to add elements and a dequeue operation to remove them.

Implementation of a queue data structure in Python

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, value):
        self.queue.append(value)

    def dequeue(self):
        if self.is_empty():
            return None
        return self.queue.pop(0)

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

    def peek(self):
        if self.is_empty():
            return None
        return self.queue[0]

# Example usage
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)
print(queue.peek())  # Output: 10
print(queue.dequeue())  # Output: 10
print(queue.dequeue())  # Output: 20
```

Output:

```
10
10
20
```

Nonlinear Data Structures:

Nonlinear data structures are widely used for hierarchical data representation, network analysis, indexing, and efficient data retrieval.

- Nonlinear data structures organize data elements in a hierarchical or non-sequential manner. Examples include trees, graphs, and hash tables.
- They provide more complex relationships between data elements, allowing efficient searching, insertion, and deletion operations.
- Nonlinear data structures are suitable for modeling real-world scenarios, such as hierarchical relationships, network connections, and complex data dependencies.

Nonlinear Data Structures are classified into two type

- Tree
- Graph

Tree

Trees are nonlinear data structures mainly composed of nodes connected by edges; there are many types of trees, such as binary trees, binary search trees, balanced trees, heaps, and many more. Below we will see the implementation of the binary tree.

Implementation of Binary Tree in Python

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Create the binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
```

```
# Traverse the binary tree (in-order traversal)
def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.data)
        inorder_traversal(node.right)

print("In-order Traversal:")
inorder_traversal(root)
```

Output

```
2
5
1
3


...Program finished with exit code 0
Press ENTER to exit console.
```

The code is the basic structure of a binary tree and how to perform an in-order traversal recursively. Binary trees are widely used data structures and traversals like in-order, pre-order, and post-order help visit and process nodes in different orders, depending on the desired operation.

Graph

Graphs are nonlinear data structures composed of nodes (vertices) connected by edges.

```
import networkx as nx
import matplotlib.pyplot as plt

# Create the graph
G = nx.Graph()

# Add nodes
G.add_nodes_from([1, 2, 3, 4, 5])

# Add edges
G.add_edges_from([(1, 2), (1, 3), (2, 4), (2, 5)])

# Visualize the graph
nx.draw(G, with_labels=True)
plt.show()
```

Note:- Considering these differences is important when selecting the appropriate data structure for a specific programming task.

Conclusion

Both primitive and non-primitive data structures are essential in programming and data management. Primitive data structures provide the basic building blocks for representing simple data types efficiently. On the other hand, non-primitive data structures offer more flexibility, allowing complex data organization, efficient searching, and dynamic resizing. Understanding the characteristics and differences between these two types of data structures is crucial for selecting the appropriate structure to optimize program performance and achieve efficient data management in various programming scenarios.