Trees, Dictionaries, and Priority Queues

---

1. Trees

Definition

A tree is a hierarchical data structure consisting of nodes connected by edges. It has the following properties:

1. **Root Node**: The topmost node of the tree.
2. **Child Node**: Nodes connected to a parent node.
3. **Parent Node**: The immediate predecessor of a node.
4. **Leaf Node**: A node with no children.
5. **Edge**: A connection between two nodes.
6. **Height of Tree**: The longest path from the root to a leaf.

Types of Trees

1. **Binary Tree**:
   - Each node has at most two children: left and right.
2. **Binary Search Tree (BST)**:
   - A binary tree where the left child is smaller than the parent, and the right child is greater.
3. **AVL Tree**:
   - A self-balancing binary search tree where the height difference of subtrees is at most one.
4. **Heap Tree**:
   - A binary tree where the parent node is either greater (max-heap) or smaller (min-heap) than its children.
5. **Trie**:
   - A tree used for storing strings, primarily for searching and autocomplete.
6. **General Tree**:
   - Each node can have any number of children.

Tree Implementation in Python

Using Classes

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self._insert(self.root, value)

    def _insert(self, current, value):
        if value < current.value:
            if current.left:
                self._insert(current.left, value)
            else:
                current.left = Node(value)
        else:
            if current.right:
                self._insert(current.right, value)
            else:
                current.right = Node(value)

    def in_order_traversal(self, node):
        if node:
            self.in_order_traversal(node.left)
            print(node.value, end=" ")
            self.in_order_traversal(node.right)

# Example Usage
tree = BinaryTree()
```

```
tree.insert(10)
tree.insert(5)
tree.insert(15)
tree.insert(3)
tree.insert(7)
print("In-order Traversal:")
tree.in_order_traversal(tree.root)
```

Applications of Trees

1. **Hierarchical Data**: File systems, organization charts.
2. **Searching**: Binary Search Trees, Heaps.
3. **Trie**: String matching, autocomplete features.
4. **Graph Representations**: Spanning trees in networks.
5. **Expression Parsing**: Representing mathematical expressions.

---

2. Dictionaries

Definition

A dictionary is a collection of key-value pairs. Each key is unique, and values can be of any data type. It is implemented as a hash table in Python, offering average **O(1)** time complexity for lookups, insertions, and deletions.

Dictionary Operations in Python

```
# Creating a dictionary
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing elements
print(my_dict["name"])  # Output: Alice

# Adding a key-value pair
my_dict["job"] = "Engineer"

# Updating a value
my_dict["age"] = 26

# Deleting a key-value pair
del my_dict["city"]

# Iterating through the dictionary
for key, value in my_dict.items():
    print(f"{key}: {value}")

# Check if a key exists
print("name" in my_dict)  # Output: True

# Get all keys and values
keys = my_dict.keys()
values = my_dict.values()
```

Applications of Dictionaries

1. **Data Lookup**: Mapping keys to values for fast access.
2. **Caching**: Implementing hash maps for temporary storage.
3. **Configurations**: Storing application settings.
4. **Data Aggregation**: Counting occurrences or categorizing data.

---

3. Priority Queues

Definition

A priority queue is an abstract data type where each element is associated with a priority. The element with the highest (or lowest) priority is dequeued first.

Types of Priority Queues

1. **Min-Priority Queue**:
   - The element with the smallest priority is dequeued first.
2. **Max-Priority Queue**:
   - The element with the largest priority is dequeued first.

Implementation in Python

Using `heapq` (Binary Heap)

```
import heapq

# Create a min-heap
priority_queue = []

# Adding elements
heapq.heappush(priority_queue, (2, "task2"))
heapq.heappush(priority_queue, (1, "task1"))
heapq.heappush(priority_queue, (3, "task3"))

# Removing elements
print(heapq.heappop(priority_queue))  # Output: (1, 'task1')
print(heapq.heappop(priority_queue))  # Output: (2, 'task2')

# Peek at the smallest element
print(priority_queue[0])  # Output: (3, 'task3')
```

Using `queue.PriorityQueue`

```
from queue import PriorityQueue

pq = PriorityQueue()

# Adding elements
pq.put((2, "task2"))
pq.put((1, "task1"))
pq.put((3, "task3"))

# Removing elements
print(pq.get())  # Output: (1, 'task1')
print(pq.get())  # Output: (2, 'task2')
```

Applications of Priority Queues

1. **Task Scheduling**: CPU scheduling in operating systems.
2. **Dijkstra's Algorithm**: Finding the shortest path in a graph.
3. **Event-Driven Simulations**: Managing events by priority.
4. **Data Compression**: Huffman encoding for efficient storage.

Summary

- **Trees**: Hierarchical data structures for efficient searching and representation.
- **Dictionaries**: Key-value stores with efficient lookups.
- **Priority Queues**: Abstract data structures managing elements by priority.

These structures are essential for solving diverse computational problems efficiently. Let me know if you'd like to dive deeper into any of these!

In [ ]: