Data Structures and Algorithms

Linked Lists

Introduction to Linked Lists

Linked Lists are a fundamental data structure used in computer science to organize items sequentially. Unlike arrays, linked lists are not contiguous in memory, which allows for efficient insertions and deletions. Each element in a linked list contains a data value and a reference (or link) to the next element in the sequence.

Types of Linked Lists

1. **Singly Linked Lists** - Each node contains a single link field pointing to the next node in the sequence.
2. **Doubly Linked Lists** - Each node contains two links: one pointing to the next node and another pointing to the previous node.
3. **Circular Linked Lists** - Similar to singly or doubly linked lists, but the last node points back to the first node, forming a circle.

Advantages of Linked Lists

- **Dynamic Data Structure**: Linked lists are dynamic and can expand and contract as needed.
- **Efficient Operations**: Insertions and deletions are more efficient at any point in the list compared to arrays.
- **No Memory Waste**: Only the needed memory is used, as nodes are created only when required.

Disadvantages of Linked Lists

- **Memory Usage**: Each node in a linked list typically requires more memory than an array because of the storage used by their pointers.
- **Traversal**: Elements can only be accessed sequentially, starting from the first node.
- **No Random Access**: Direct access to elements is not possible; access time is linear.

Implementing a Singly Linked List

Here's a simple implementation of a singly linked list in Python:

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

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

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def display(self):
        elements = []
        current_node = self.head
        while current_node:
            elements.append(current_node.data)
            current_node = current_node.next
```

```
        return elements
```

Basic Operations on Linked Lists

- **Append**: Add a node to the end of the list.
- **Prepend**: Add a node to the beginning of the list.
- **Insert After Node**: Insert a node after a given node.
- **Delete By Value**: Remove a node by its value.
- **Search**: Find a node containing a given value.

Example Usage

Here's how you can use the LinkedList class to create a list and add some elements:

```
# Creating a linked list and adding elements
llist = LinkedList()
llist.append(3)
llist.append(5)
llist.append(7)

# Displaying all elements in the list
print(llist.display())  # Output: [3, 5, 7]
```

CRUD Functions in LinkedList

Append (Add to the end)

Here's how to append a node to the end of the LinkedList:

```
    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
```

Prepend (Add to the start)

This code block shows how to prepend a node to the beginning of the LinkedList:

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

Adding in an arbitrary position

Insert a node after a given node in the LinkedList with this method:

```
    def insert_after_node(self, prev_node, data):
        if not prev_node:
            print("Previous node is not in the list")
            return
        new_node = Node(data)
        new_node.next = prev_node.next
        prev_node.next = new_node
```

Search the LinkedList and delete value

Here's how you can delete a node by its value from the LinkedList:

```
    def delete_by_value(self, data):
        if self.head is None:
            return

        # If the head needs to be removed
        if self.head.data == data:
            self.head = self.head.next
            return

        # Locate the node to be removed
        current = self.head
        while current.next:
            if current.next.data == data:
                current.next = current.next.next
                return
            current = current.next
        print("The data is not in the list")
```

Search

```
    def search(self, data):
        current = self.head
        while current:
            if current.data == data:
                return True
            current = current.next
        return False
```

Introduction to Binary Search Trees (BST)

A Binary Search Tree (BST) is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.
- There must be no duplicate nodes.

Advantages of Binary Search Trees

- **Efficient in terms of access, search, insertion, and deletion operations** compared to other trees.
- **Ordered Structure**: Maintaining an order makes it easy to retrieve elements in sorted order.
- **Flexibility**: The tree structure allows for dynamic resizing, with the ability to extend and prune the structure as required.

Basic Operations on a BST

- **Search**: Check whether a given key is present in the tree or not.
- **Insert**: Add a new node to the BST while maintaining the BST properties.
- **Delete**: Remove a node from the BST while ensuring the tree remains a BST (Note: not included in this guide).
- **Traversal**: There are several ways to traverse a BST such as In-order, Pre-order, Post-order, and Level-order that help in reading the nodes of the tree in various orders.

Implementing a Binary Search Tree

Here's a simple implementation of operations in a binary search tree using Python:

Node Structure

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

Insertion

```
    def insert_bst(self, root, key):
        if root is None:
            return TreeNode(key)
        if key < root.val:
            root.left = self.insert_bst(root.left, key)
        else:
            root.right = self.insert_bst(root.right, key)
        return root
```

Searching

```
    def search_bst(self, root, key):
        if root is None or root.val == key:
            return root
        if key > root.val:
            return self.search_bst(root.right, key)
        return self.search_bst(root.left, key)
```

Example Usage

```
# Creating a BST and inserting elements
bst_root = None
bst = BinaryTree()  # Assuming BinaryTree class has the methods
described
elements = [20, 10, 30, 5, 15, 25, 35]
for elem in elements:
    bst_root = bst.insert_bst(bst_root, elem)

# Searching for an element in the BST
found_node = bst.search_bst(bst_root, 15)
print("Found:" if found_node else "Not Found")
```

Tree Traversal Techniques

Tree traversal is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

Pre-order Traversal

In pre-order traversal, the nodes are recursively visited in this order: root, left subtree, right subtree. This method is used to create a copy of the tree.

```
def preorder_traversal(root):
    if root:
        print(root.val)  # Visit the root node
        preorder_traversal(root.left)  # Recursively visit left subtree
        preorder_traversal(root.right)  # Recursively visit right
subtree
```

In-order Traversal

In in-order traversal, the nodes are recursively visited in this order: left subtree, root, right subtree. This method is commonly used for binary search trees because it visits nodes in ascending order.

```
    def inorder_traversal(root):
        if root:
            inorder_traversal(root.left)  # Recursively visit left subtree
            print(root.val)  # Visit the root node
            inorder_traversal(root.right)  # Recursively visit right subtree
```

Post-order Traversal

In post-order traversal, the nodes are recursively visited in this order: left subtree, right subtree, root. This method is useful for deleting or freeing nodes and space of the tree.

```
    def postorder_traversal(root):
        if root:
            postorder_traversal(root.left)  # Recursively visit left subtree
            postorder_traversal(root.right)  # Recursively visit right
    subtree
            print(root.val)  # Visit the root node
```

Example Usage of Traversals

```
    # Assume 'root' is the root of a binary tree
    print("Pre-order Traversal:")
    preorder_traversal(root)

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

    print("Post-order Traversal:")
    postorder_traversal(root)
```

Introduction to Stacks

A stack is an abstract data type that serves as a collection of elements, with two principal operations:

- **Push**, which adds an element to the collection, and
- **Pop**, which removes the most recently added element that was not yet removed. The stack is known as a Last In First Out (LIFO) data structure, because the last element added to the stack is the first one to be removed.

Properties of Stacks

- **LIFO Principle**: The last item to be inserted into a stack is the first one to be deleted from it.
- **Simplicity**: Stacks are very simple to implement and use.
- **Functionality**: Provides essential operations like push, pop, peek, and isEmpty.

Implementing a Stack

Here's a simple implementation of a stack using Python:

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

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None
```

```
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None

    def size(self):
        return len(self.items)
```

Basic Operations on Stacks

- **Push**: Add an item to the top of the stack.
- **Pop**: Remove the top item from the stack.
- **Peek**: Get the top item of the stack without removing it.
- **IsEmpty**: Check whether the stack is empty.
- **Size**: Return the number of items in the stack.

Example Usage

```
# Create a new stack
stack = Stack()

# Push items
stack.push(1)
stack.push(2)
stack.push(3)

# Peek the top item
print(stack.peek())  # Output: 3

# Pop an item
print(stack.pop())  # Output: 3

# Check size
print(stack.size())  # Output: 2
```

Introduction to Queues

A queue is an abstract data type that serves as a collection of elements, with two principal operations:

- **Enqueue**, which adds an element to the end of the queue, and
- **Dequeue**, which removes the element at the front of the queue.
  This data structure follows a First In First Out (FIFO) model, where the first element added to the queue is the first one to be removed.

Properties of Queues

- **FIFO Principle**: The first item added to the queue will be the first one to be removed.
- **Versatility**: Queues are used in a wide range of situations where it is necessary to maintain order for items.
- **Functionality**: Provides essential operations like enqueue, dequeue, front, rear, and isEmpty.

Implementing a Queue

Here's a simple implementation of a queue using Python:

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

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

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
```

```
        if not self.is_empty():
            return self.items.pop(0)
        return None

    def front(self):
        if not self.is_empty():
            return self.items[0]
        return None

    def rear(self):
        if not self.is_empty():
            return self.items[-1]
        return None

    def size(self):
        return len(self.items)
```

Basic Operations on Queues

- **Enqueue**: Add an item to the rear of the queue.
- **Dequeue**: Remove the item from the front of the queue.
- **Front**: Returns the first item of the queue without removing it.
- **Rear**: Returns the last item of the queue without removing it.
- **IsEmpty**: Check whether the queue is empty.
- **Size**: Return the number of items in the queue.

Example Usage

```
# Create a new queue
queue = Queue()

# Enqueue items
queue.enqueue(1)
queue.enqueue(2)
```

```
queue.enqueue(3)

# Get front and rear item
print(queue.front())  # Output: 1
print(queue.rear())   # Output: 3

# Dequeue an item
print(queue.dequeue())  # Output: 1

# Check size
print(queue.size())  # Output: 2
```

Implementing Sort Algorithms

Sorting algorithms are a basic aspect of studying data structures and algorithms. They allow us to reorder the elements in an array or list according to a comparison criterion, typically in ascending or descending order. Here we'll look at four common sorting algorithms:

1. Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it has several advantages:

- Simple implementation
- Efficient for (quite) small data sets
- More efficient in practice than most other simple quadratic algorithms such as selection sort or bubble sort
- Adaptive: Efficient for data sets that are already substantially sorted

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

2. Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is known for:

- Its simplicity and ease of visualization
- Performance: $O(n^2)$ in the average and worst cases

```
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]
    return arr
```

3. Quicksort

Quicksort is a divide-and-conquer algorithm which relies on a partition operation: to partition an array into two sub-arrays, elements less than the pivot to the left of the pivot and those greater on the right. Benefits include:

- Often faster in practice than other O(n log n) algorithms
- Efficient for large data sets
- Not stable but can be made stable with some tweaks

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr.pop()
        left = [x for x in arr if x <= pivot]
        right = [x for x in arr if x > pivot]
        return quicksort(left) + [pivot] + quicksort(right)
```

4. Merge Sort

Merge Sort is an efficient, stable, divide-and-conquer sorting algorithm. This algorithm is known for its:

- Best and average time complexity of O(n log n)
- Stability, which makes it useful for sorting linked lists

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
```

```
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
    return arr
```

Implementing Search Algorithms

Search algorithms are essential for locating specific elements within datasets. We'll explore three common search methods: Linear Search, Binary Search, and Hash Table Search.

1. Linear Search

Linear Search is a simple search method that checks every element in a list until the desired element is found or the list ends. It is useful when dealing with small or unsorted data sets.

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1
```

Example Usage of Linear Search

```
arr = [5, 3, 8, 6, 2]
target = 8
result = linear_search(arr, target)
print(f'Target found at index: {result}' if result != -1 else 'Target
not found')
```

2. Binary Search

Binary Search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed the possible locations to just one.

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

Example Usage of Binary Search

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 7
result = binary_search(arr, target)
print(f'Target found at index: {result}' if result != -1 else 'Target
not found')
```

3. Hash Table (Dictionary) Search

A Hash Table Search involves using a hash table data structure to store key-value pairs. It provides very efficient average-case time complexity for search, insert, and delete operations. This method is particularly effective for datasets where quick lookups, inserts, and deletes are necessary.

```
def hash_table_search(hash_table, target):
    # Assuming hash_table is a dictionary
    return hash_table.get(target, 'Target not found')
```

Example Usage of Hash Table (Dictionary) Search

```
# Creating a hash table
hash_table = {i: f'item{i}' for i in range(10)}
target = 5
result = hash_table_search(hash_table, target)
print(f'Target found: {result}' if result != 'Target not found' else
'Target not found')
```