sleeps-ans-1

October 19, 2024

[16]:
```
print("Slip 1")
```

```
Slip 1
```

[22]:
```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break

if __name__ == "__main__":
    array = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", array)
    bubble_sort(array)
    print("Sorted array:", array)
```

```
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

[5]:
```
class DynamicStack:
    def __init__(self):
        self.stack = []
        self.size = 0

    def is_empty(self):
        return self.size == 0

    def push(self, item):
        self.stack.append(item)
        self.size += 1

    def pop(self):
        if self.is_empty():
```

```
            print("Stack is empty")
        item = self.stack.pop()
        self.size -= 1
        return item

    def peek(self):
        if self.is_empty():
            return self.stack[-1]

    def get_size(self):
        return self.size

# Example usage
if __name__ == "__main__":
    stack = DynamicStack()
    print("Is stack empty?", stack.is_empty())
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print("Top item:", stack.peek())
    print("Stack size:", stack.get_size())
    print("Popped item:", stack.pop())
    print("Stack size after pop:", stack.get_size())
    print("Is stack empty?", stack.is_empty())
```

```
Is stack empty? True
Top item: None
Stack size: 3
Popped item: 3
Stack size after pop: 2
Is stack empty? False
```

[6]:
```
print("Slip 2")
```

```
Slip 2
```

[7]:
```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Example usage
if __name__ == "__main__":
```

```
    array = [64, 25, 12, 22, 11]
    print("Original array:", array)
    selection_sort(array)
    print("Sorted array:", array)
```

```
Original array: [64, 25, 12, 22, 11]
Sorted array: [11, 12, 22, 25, 64]
```

[10]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

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

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
        new_node.prev = last

    def delete_last(self):
        if not self.head:
            print("The list is empty.")
            return
        if not self.head.next:
            self.head = None
            return
        last = self.head
        while last.next:
            last = last.next
        last.prev.next = None

    def insert_at_position(self, data, position):
        new_node = Node(data)
        if position == 0:
            new_node.next = self.head
            if self.head:
                self.head.prev = new_node
```

```
            self.head = new_node
            return
        current = self.head
        for _ in range(position - 1):
            if current is None:
                print("Position out of bounds.")
                return
            current = current.next
        if current is None:
            print("Position out of bounds.")
            return
        new_node.next = current.next
        if current.next:
            current.next.prev = new_node
        current.next = new_node
        new_node.prev = current

    def display(self):
        current = self.head
        if not current:
            print("The list is empty.")
            return
        while current:
            print(current.data, end=" <-> " if current.next else "")
            current = current.next
        print()

def main():
    dll = DoublyLinkedList()
    while True:
        print("\nMenu:")
        print("1. Create Doubly Linked List")
        print("2. Delete Last Node")
        print("3. Insert Node by Position")
        print("4. Display List")
        print("5. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = input("Enter data to append: ")
            dll.append(data)
        elif choice == 2:
            dll.delete_last()
        elif choice == 3:
            data = input("Enter data to insert: ")
            position = int(input("Enter position: "))
            dll.insert_at_position(data, position)
```

```
        elif choice == 4:
            dll.display()
        elif choice == 5:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

```
Menu:
1. Create Doubly Linked List
2. Delete Last Node
3. Insert Node by Position
4. Display List
5. Exit
Enter your choice: 1
Enter data to append: 12

Menu:
1. Create Doubly Linked List
2. Delete Last Node
3. Insert Node by Position
4. Display List
5. Exit
Enter your choice: 4
12

Menu:
1. Create Doubly Linked List
2. Delete Last Node
3. Insert Node by Position
4. Display List
5. Exit
Enter your choice: 5
Exiting…
```

[11]:
```
class Stack:
    def __init__(self):
        self.items = []

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

    def push(self, item):
```

```
        self.items.append(item)

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

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

def precedence(op):
    if op == '+' or op == '-':
        return 1
    if op == '*' or op == '/':
        return 2
    return 0

def infix_to_postfix(expression):
    output = []
    stack = Stack()

    for char in expression:
        if char.isalnum():  # If the character is an operand
            output.append(char)
        elif char == '(':  # If the character is '(', push to stack
            stack.push(char)
        elif char == ')':  # If the character is ')'
            while not stack.is_empty() and stack.peek() != '(':
                output.append(stack.pop())
            stack.pop()  # Remove '(' from stack
        else:  # An operator
            while (not stack.is_empty() and precedence(stack.peek()) >=␣
 ↪precedence(char)):
                output.append(stack.pop())
            stack.push(char)

    while not stack.is_empty():  # Pop all the operators from the stack
        output.append(stack.pop())

    return ''.join(output)


expression = input("Enter an infix expression: ")
postfix = infix_to_postfix(expression)
print("Postfix expression:", postfix)
```

```
Enter an infix expression: 1+2*3
Postfix expression: 123*+
```

[13]:
```
print("slip 3")
```

```
slip 3
```

[12]:
```
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Example usage
if __name__ == "__main__":
    array = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", array)
    insertion_sort(array)
    print("Sorted array:", array)
```

```
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

[14]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def create_list(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def insert_at_position(self, data, position):
        new_node = Node(data)
        if position == 0:
            new_node.next = self.head
```

```
            self.head = new_node
            return
        current = self.head
        for _ in range(position - 1):
            if current is None:
                print("Position out of bounds.")
                return
            current = current.next
        if current is None:
            print("Position out of bounds.")
            return
        new_node.next = current.next
        current.next = new_node

    def delete_first(self):
        if not self.head:
            print("The list is empty.")
            return
        self.head = self.head.next

    def display(self):
        current = self.head
        if not current:
            print("The list is empty.")
            return
        while current:
            print(current.data, end=" -> " if current.next else "")
            current = current.next
        print()

def main():
    sll = SinglyLinkedList()
    while True:
        print("\nMenu:")
        print("1. Create List")
        print("2. Insert Element by Position")
        print("3. Delete First Element")
        print("4. Display List")
        print("5. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = input("Enter data to append: ")
            sll.create_list(data)
        elif choice == 2:
            data = input("Enter data to insert: ")
            position = int(input("Enter position: "))
```

```
            sll.insert_at_position(data, position)
        elif choice == 3:
            sll.delete_first()
        elif choice == 4:
            sll.display()
        elif choice == 5:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

```
Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 1
Enter data to append: 12

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 1
Enter data to append: 5435

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 1
Enter data to append: 234

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
```

```
5. Exit
Enter your choice: 23234
Invalid choice. Please try again.

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 1
Enter data to append: 232

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 4
12 -> 5435 -> 234 -> 232

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 2
Enter data to insert: 231313445
Enter position: 0

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 2
Enter data to insert: 4
Enter position: 4

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
```

```
Enter your choice: 4
231313445 -> 12 -> 5435 -> 234 -> 4 -> 232

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 3

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 4
12 -> 5435 -> 234 -> 4 -> 232

Menu:
1. Create List
2. Insert Element by Position
3. Delete First Element
4. Display List
5. Exit
Enter your choice: 5
Exiting…
```

[1]:
```python
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)
        print(f"Enqueued: {item}")

    def dequeue(self):
        if not self.is_empty():
            item = self.queue.pop(0)
            print(f"Dequeued: {item}")
            return item
        else:
            print("Queue is empty. Cannot dequeue.")
            return None

    def peek(self):
```

```
        if not self.is_empty():
            print(f"Front element: {self.queue[0]}")
            return self.queue[0]
        else:
            print("Queue is empty.")
            return None

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

    def size(self):
        print(f"Queue size: {len(self.queue)}")
        return len(self.queue)

    def display(self):
        if not self.is_empty():
            print("Queue:", self.queue)
        else:
            print("Queue is empty.")


if __name__ == "__main__":
    q = Queue()

    q.enqueue(10)
    q.enqueue(20)
    q.enqueue(30)

    q.display()

    q.dequeue()
    q.peek()
    q.size()

    q.display()

    q.dequeue()
    q.dequeue()
    q.dequeue()
```

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue: [10, 20, 30]
Dequeued: 10
Front element: 20
Queue size: 2
```

```
Queue: [20, 30]
Dequeued: 20
Dequeued: 30
Queue is empty. Cannot dequeue.
```

[2]:
```python
print("slip 4")
```

```
slip 4
```

[3]:
```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

arr = [10, 20, 30, 40, 50]
target = 30

result = linear_search(arr, target)

if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
```

```
Element found at index 2
```

[4]:
```python
class Stack:
    def __init__(self, max_size):
        self.stack = []
        self.max_size = max_size

    def push(self, item):
        if self.is_full():
            print("Stack is full. Cannot push.")
        else:
            self.stack.append(item)
            print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
            print("Stack is empty. Cannot pop.")
        else:
            item = self.stack.pop()
            print(f"Popped: {item}")
            return item

    def is_empty(self):
```

```
        return len(self.stack) == 0

    def is_full(self):
        return len(self.stack) == self.max_size

    def display(self):
        print("Stack:", self.stack)

if __name__ == "__main__":
    s = Stack(3)

    s.push(10)
    s.push(20)
    s.push(30)
    s.push(40)

    s.display()

    s.pop()
    s.display()

    print("Is Stack empty?", s.is_empty())
    print("Is Stack full?", s.is_full())
```

```
Pushed: 10
Pushed: 20
Pushed: 30
Stack is full. Cannot push.
Stack: [10, 20, 30]
Popped: 30
Stack: [10, 20]
Is Stack empty? False
Is Stack full? False
```

[5]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def create(self, data_list):
        for data in data_list:
            self.insert(len(data_list), data)
```

```
    def insert(self, position, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
        else:
            if position == 1:
                temp = self.head
                while temp.next != self.head:
                    temp = temp.next
                new_node.next = self.head
                temp.next = new_node
                self.head = new_node
            else:
                temp = self.head
                for _ in range(position - 2):
                    temp = temp.next
                    if temp.next == self.head:
                        break
                new_node.next = temp.next
                temp.next = new_node

    def delete(self, position):
        if self.head is None:
            print("List is empty.")
            return
        temp = self.head
        if position == 1:
            while temp.next != self.head:
                temp = temp.next
            if temp == self.head:
                self.head = None
            else:
                temp.next = self.head.next
                self.head = self.head.next
        else:
            for _ in range(position - 2):
                temp = temp.next
                if temp.next == self.head:
                    break
            if temp.next == self.head:
                print("Invalid position.")
            else:
                temp.next = temp.next.next

    def display(self):
        if self.head is None:
```

```
            print("List is empty.")
            return
        temp = self.head
        while True:
            print(temp.data, end=" -> ")
            temp = temp.next
            if temp == self.head:
                break
        print("(head)")

def menu():
    csll = CircularSinglyLinkedList()
    while True:
        print("\nCircular Singly Linked List Operations")
        print("1. Create List")
        print("2. Insert Node at Position")
        print("3. Delete Node at Position")
        print("4. Display List")
        print("5. Exit")
        choice = int(input("Enter your choice: "))
        if choice == 1:
            data_list = list(map(int, input("Enter elements to create list␣
 ↪(space-separated): ").split()))
            csll.create(data_list)
        elif choice == 2:
            position = int(input("Enter position to insert node: "))
            data = int(input("Enter data: "))
            csll.insert(position, data)
        elif choice == 3:
            position = int(input("Enter position to delete node: "))
            csll.delete(position)
        elif choice == 4:
            csll.display()
        elif choice == 5:
            break
        else:
            print("Invalid choice, please try again.")

if __name__ == "__main__":
    menu()
```

```
Circular Singly Linked List Operations
1. Create List
2. Insert Node at Position
3. Delete Node at Position
4. Display List
```

```
5. Exit
Enter your choice: 1
Enter elements to create list (space-separated): 12

Circular Singly Linked List Operations
1. Create List
2. Insert Node at Position
3. Delete Node at Position
4. Display List
5. Exit
Enter your choice: 5
```

[6]:
```python
print("slip 5")
```

```
slip 5
```

[7]:
```python
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def push(self, item, priority):
        self.queue.append((priority, item))
        self.queue.sort(key=lambda x: x[0])
        print(f"Pushed: {item} with priority {priority}")

    def pop(self):
        if self.is_empty():
            print("Priority Queue is empty. Cannot pop.")
        else:
            priority, item = self.queue.pop(0)
            print(f"Popped: {item} with priority {priority}")
            return item

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

    def display(self):
        print("Priority Queue:", [item for priority, item in self.queue])

if __name__ == "__main__":
    pq = PriorityQueue()

    pq.push("Task 1", 2)
    pq.push("Task 2", 1)
    pq.push("Task 3", 3)

    pq.display()
```

```
    pq.pop()
    pq.display()

    pq.pop()
    pq.pop()
    pq.pop()
```

```
Pushed: Task 1 with priority 2
Pushed: Task 2 with priority 1
Pushed: Task 3 with priority 3
Priority Queue: ['Task 2', 'Task 1', 'Task 3']
Popped: Task 2 with priority 1
Priority Queue: ['Task 1', 'Task 3']
Popped: Task 1 with priority 2
Popped: Task 3 with priority 3
Priority Queue is empty. Cannot pop.
```

[8]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

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

    def create(self, data_list):
        for data in data_list:
            self.append(data)

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = new_node
            new_node.prev = temp

    def display_reverse(self):
        if self.head is None:
            print("List is empty.")
            return
```

```
        temp = self.head
        while temp.next:
            temp = temp.next
        while temp:
            print(temp.data, end=" <- ")
            temp = temp.prev
        print("(head)")

if __name__ == "__main__":
    dll = DoublyLinkedList()

    data_list = list(map(int, input("Enter elements to create the list␣
 ↪(space-separated): ").split()))
    dll.create(data_list)

    print("Doubly Linked List in Reverse Order:")
    dll.display_reverse()
```

```
Enter elements to create the list (space-separated): 243 35
Doubly Linked List in Reverse Order:
35 <- 243 <- (head)
```

[12]:
```
class Stack:
    def __init__(self):
        self.stack = []

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

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

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

def evaluate_postfix(expression):
    s = Stack()
    for token in expression.split():
        if token.isdigit():
            s.push(int(token))
        else:
            operand2 = s.pop()
            operand1 = s.pop()
            if token == '+':
                s.push(operand1 + operand2)
```

```
            elif token == '-':
                s.push(operand1 - operand2)
            elif token == '*':
                s.push(operand1 * operand2)
            elif token == '/':
                s.push(operand1 / operand2)
    return s.pop()

if __name__ == "__main__":
    expression = input("Enter a postfix expression: ")
    result = evaluate_postfix(expression)
    print("Result:", result)
```

```
Enter a postfix expression: 234 435 +
Result: 669
```

[13]:
```
print("slip 6")
```

```
slip 6
```

[15]:
```
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    target = 5
    result = binary_search(arr, target)

    if result != -1:
        print(f"Element found at index {result}")
    else:
        print("Element not found")
```

```
Element found at index 4
```

[16]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

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

    def reverse(self):
        prev = None
        current = self.head
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        self.head = prev

    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

if __name__ == "__main__":
    sll = SinglyLinkedList()

    data_list = list(map(int, input("Enter elements to create the list␣
 ↪(space-separated): ").split()))
    for data in data_list:
        sll.append(data)

    print("Original Singly Linked List:")
    sll.display()
```

```
    sll.reverse()
    print("Reversed Singly Linked List:")
    sll.display()
```

```
Enter elements to create the list (space-separated): 23 465 67
Original Singly Linked List:
23 -> 465 -> 67 -> None
Reversed Singly Linked List:
67 -> 465 -> 23 -> None
```

[18]:
```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

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

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursively(self.root, key)

    def _insert_recursively(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_recursively(node.left, key)
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_recursively(node.right, key)

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.val, end=" ")
            self.inorder(node.right)

    def preorder(self, node):
        if node:
            print(node.val, end=" ")
```

```
            self.preorder(node.left)
            self.preorder(node.right)

    def postorder(self, node):
        if node:
            self.postorder(node.left)
            self.postorder(node.right)
            print(node.val, end=" ")

if __name__ == "__main__":
    bst = BinarySearchTree()

    data_list = list(map(int, input("Enter integer numbers to create the BST␣
 ↪(space-separated): ").split()))
    for data in data_list:
        bst.insert(data)

    print("In-order Traversal:")
    bst.inorder(bst.root)
    print("\nPre-order Traversal:")
    bst.preorder(bst.root)
    print("\nPost-order Traversal:")
    bst.postorder(bst.root)
```

```
Enter integer numbers to create the BST (space-separated): 45 6587 34 5768
In-order Traversal:
34 45 5768 6587
Pre-order Traversal:
45 34 6587 5768
Post-order Traversal:
34 5768 6587 45
```

[19]:
```
print("slip 7")
```

```
slip 7
```

[21]:
```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

if __name__ == "__main__":
```

```
    arr = list(map(int, input("Enter numbers to sort (space-separated): ").
↪split()))
    sorted_arr = quick_sort(arr)
    print("Sorted array:", sorted_arr)
```

```
Enter numbers to sort (space-separated): 34 56 687 324
Sorted array: [34, 56, 324, 687]
```

[22]:
```
class LinearQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def is_empty(self):
        return self.front == -1

    def is_full(self):
        return self.rear == self.size - 1

    def enqueue(self, item):
        if self.is_full():
            print("Queue is full. Cannot enqueue.")
        else:
            if self.is_empty():
                self.front = 0
            self.rear += 1
            self.queue[self.rear] = item
            print(f"Enqueued: {item}")

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
        else:
            item = self.queue[self.front]
            if self.front == self.rear:
                self.front = -1
                self.rear = -1
            else:
                self.front += 1
            print(f"Dequeued: {item}")
            return item

    def display(self):
        if self.is_empty():
            print("Queue is empty.")
```

```
        else:
            print("Queue:", end=" ")
            for i in range(self.front, self.rear + 1):
                print(self.queue[i], end=" ")
            print()

if __name__ == "__main__":
    size = int(input("Enter the size of the queue: "))
    queue = LinearQueue(size)

    while True:
        print("\nMenu:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Display Queue")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            item = int(input("Enter the item to enqueue: "))
            queue.enqueue(item)
        elif choice == 2:
            queue.dequeue()
        elif choice == 3:
            queue.display()
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Enter the size of the queue: 2

Menu:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 1
Enter the item to enqueue: 324
Enqueued: 324

Menu:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 1
```

```
Enter the item to enqueue: 24
Enqueued: 24

Menu:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 1
Enter the item to enqueue: 34
Queue is full. Cannot enqueue.

Menu:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 3
Queue: 324 24

Menu:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 4
```

[23]:
```python
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def is_empty(self):
        return self.front == -1

    def is_full(self):
        return (self.rear + 1) % self.size == self.front

    def enqueue(self, item):
        if self.is_full():
            print("Queue is full. Cannot enqueue.")
        else:
            if self.is_empty():
                self.front = 0
            self.rear = (self.rear + 1) % self.size
```

```
            self.queue[self.rear] = item
            print(f"Enqueued: {item}")

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
        else:
            item = self.queue[self.front]
            if self.front == self.rear:
                self.front = -1
                self.rear = -1
            else:
                self.front = (self.front + 1) % self.size
            print(f"Dequeued: {item}")
            return item

    def display(self):
        if self.is_empty():
            print("Queue is empty.")
        else:
            print("Queue:", end=" ")
            i = self.front
            while True:
                print(self.queue[i], end=" ")
                if i == self.rear:
                    break
                i = (i + 1) % self.size
            print()

if __name__ == "__main__":
    size = int(input("Enter the size of the circular queue: "))
    queue = CircularQueue(size)

    while True:
        print("\nMenu:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Display Queue")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            item = int(input("Enter the item to enqueue: "))
            queue.enqueue(item)
        elif choice == 2:
            queue.dequeue()
        elif choice == 3:
```

```
            queue.display()
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Enter the size of the circular queue: 3

Menu:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 4
```

[25]:
```
print("slip 8")
```

```
slip 8
```

[28]:
```
def linear_search(arr, target):
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1

if __name__ == "__main__":
    arr = list(map(int, input("Enter integer elements of the array␣
 ↪(space-separated): ").split()))
    target = int(input("Enter the element to search for: "))

    result = linear_search(arr, target)

    if result != -1:
        print(f"Element found at index {result}")
    else:
        print("Element not found")
```

```
Enter integer elements of the array (space-separated): 34 456 13 64
Enter the element to search for: 34
Element found at index 0
```

[29]:
```
class Stack:
    def __init__(self):
        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 is_empty(self):
        return len(self.items) == 0

def evaluate_postfix(expression):
    stack = Stack()
    for token in expression.split():
        if token.isdigit():
            stack.push(int(token))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            if token == '+':
                stack.push(operand1 + operand2)
            elif token == '-':
                stack.push(operand1 - operand2)
            elif token == '*':
                stack.push(operand1 * operand2)
            elif token == '/':
                stack.push(operand1 / operand2)
    return stack.pop()

if __name__ == "__main__":
    expression = input("Enter a postfix expression: ")
    result = evaluate_postfix(expression)
    print("Result:", result)
```

```
Enter a postfix expression: 34 45 +
Result: 79
```

[30]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

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

    def create(self, data):
        new_node = Node(data)
```

```
        self.head = new_node

    def insert_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
        new_node.prev = last

    def delete_node(self, key):
        if self.head is None:
            return
        temp = self.head
        while temp:
            if temp.data == key:
                if temp.prev:
                    temp.prev.next = temp.next
                if temp.next:
                    temp.next.prev = temp.prev
                if temp == self.head:  # Move head if needed
                    self.head = temp.next
                del temp
                return
            temp = temp.next

    def display(self):
        temp = self.head
        if temp is None:
            print("List is empty.")
            return
        print("Doubly Linked List:", end=" ")
        while temp:
            print(temp.data, end=" <=> " if temp.next else " ")
            temp = temp.next
        print()

if __name__ == "__main__":
    dll = DoublyLinkedList()

    while True:
        print("\nMenu:")
        print("1. Create Doubly Linked List")
        print("2. Insert Node at the End")
```

```
        print("3. Delete Specific Element")
        print("4. Display List")
        print("5. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to create the list: "))
            dll.create(data)
            print(f"Doubly Linked List created with element: {data}")
        elif choice == 2:
            data = int(input("Enter integer to insert at the end: "))
            dll.insert_end(data)
            print(f"Inserted {data} at the end of the list.")
        elif choice == 3:
            key = int(input("Enter integer to delete from the list: "))
            dll.delete_node(key)
            print(f"Deleted {key} from the list.")
        elif choice == 4:
            dll.display()
        elif choice == 5:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Create Doubly Linked List
2. Insert Node at the End
3. Delete Specific Element
4. Display List
5. Exit
Enter your choice: 4
List is empty.

Menu:
1. Create Doubly Linked List
2. Insert Node at the End
3. Delete Specific Element
4. Display List
5. Exit
Enter your choice: 5
```

[31]:
```
print("slip 9")
```

```
slip 9
```

[32]:
```
class Stack:
    def __init__(self):
        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 is_empty(self):
        return len(self.items) == 0

def reverse_string(input_string):
    stack = Stack()

    for char in input_string:
        stack.push(char)

    reversed_string = ''
    while not stack.is_empty():
        reversed_string += stack.pop()

    return reversed_string

if __name__ == "__main__":
    input_string = input("Enter a string to reverse: ")
    result = reverse_string(input_string)
    print("Reversed string:", result)
```

```
Enter a string to reverse: apple
Reversed string: elppa
```

[33]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def create(self, data):
        new_node = Node(data)
        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):
        if self.head is None:
            print("List is empty.")
            return
        current = self.head
        print("Singly Linked List:", end=" ")
        while current:
            print(current.data, end=" -> " if current.next else " ")
            current = current.next
        print()

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

    def delete(self, key):
        if self.head is None:
            print("List is empty. Cannot delete.")
            return
        if self.head.data == key:
            self.head = self.head.next
            return
        current = self.head
        while current.next:
            if current.next.data == key:
                current.next = current.next.next
                return
            current = current.next
        print("Element not found.")

if __name__ == "__main__":
    sll = SinglyLinkedList()

    while True:
        print("\nMenu:")
        print("1. Create")
```

```
        print("2. Display")
        print("3. Search Specific Element")
        print("4. Delete Specific Element")
        print("5. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to add to the list: "))
            sll.create(data)
            print(f"Added {data} to the list.")
        elif choice == 2:
            sll.display()
        elif choice == 3:
            key = int(input("Enter integer to search for: "))
            if sll.search(key):
                print(f"Element {key} found in the list.")
            else:
                print(f"Element {key} not found in the list.")
        elif choice == 4:
            key = int(input("Enter integer to delete from the list: "))
            sll.delete(key)
            print(f"Attempting to delete {key}.")
        elif choice == 5:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Create
2. Display
3. Search Specific Element
4. Delete Specific Element
5. Exit
Enter your choice: 5
```

[34]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None

    def insert(self, data):
```

```
        new_node = Node(data)
        if self.rear is None:
            self.front = self.rear = new_node
            return
        self.rear.next = new_node
        self.rear = new_node

    def delete(self):
        if self.front is None:
            print("Queue is empty. Cannot delete.")
            return
        temp = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        print(f"Deleted: {temp.data}")

    def display(self):
        if self.front is None:
            print("Queue is empty.")
            return
        current = self.front
        print("Queue:", end=" ")
        while current:
            print(current.data, end=" -> " if current.next else " ")
            current = current.next
        print()

if __name__ == "__main__":
    queue = Queue()

    while True:
        print("\nMenu:")
        print("1. Insert")
        print("2. Delete")
        print("3. Display")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to insert: "))
            queue.insert(data)
            print(f"Inserted: {data}")
        elif choice == 2:
            queue.delete()
        elif choice == 3:
            queue.display()
```

```
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4
```

[35]:
```
print("slip 10")
```

```
slip 10
```

[36]:
```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

if __name__ == "__main__":
    arr = list(map(int, input("Enter sorted integer elements of the array␣
 ↪(space-separated): ").split()))
    target = int(input("Enter the element to search for: "))

    result = binary_search(arr, target)

    if result != -1:
        print(f"Element found at index {result}")
    else:
        print("Element not found")
```

```
Enter sorted integer elements of the array (space-separated): 324 45 465
Enter the element to search for: 45
Element found at index 1
```

[37]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

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

    def create(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
        new_node.prev = last

    def display_reverse(self):
        if self.head is None:
            print("List is empty.")
            return
        current = self.head
        while current.next:
            current = current.next
        print("Doubly Linked List in Reverse Order:", end=" ")
        while current:
            print(current.data, end=" <=> " if current.prev else " ")
            current = current.prev
        print()

    def delete_at_position(self, position):
        if self.head is None:
            print("List is empty. Cannot delete.")
            return
        temp = self.head
        if position == 0:
            self.head = temp.next
            if self.head:
                self.head.prev = None
            del temp
            return
        for _ in range(position):
            temp = temp.next
```

```
            if temp is None:
                print("Position out of bounds.")
                return
        if temp.next:
            temp.next.prev = temp.prev
        if temp.prev:
            temp.prev.next = temp.next
        del temp

if __name__ == "__main__":
    dll = DoublyLinkedList()

    while True:
        print("\nMenu:")
        print("1. Create")
        print("2. Display in Reverse Order")
        print("3. Delete a Node at Given Position")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to add to the list: "))
            dll.create(data)
            print(f"Added {data} to the list.")
        elif choice == 2:
            dll.display_reverse()
        elif choice == 3:
            position = int(input("Enter position to delete from the list␣
 ↪(0-indexed): "))
            dll.delete_at_position(position)
            print(f"Deleted node at position {position}.")
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Create
2. Display in Reverse Order
3. Delete a Node at Given Position
4. Exit
Enter your choice: 4
```

[38]:
```
class CircularQueue:
    def __init__(self, size):
        self.size = size
```

```
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def is_empty(self):
        return self.front == -1

    def is_full(self):
        return (self.rear + 1) % self.size == self.front

    def enqueue(self, data):
        if self.is_full():
            print("Queue is full. Cannot enqueue.")
            return
        if self.is_empty():
            self.front = self.rear = 0
        else:
            self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = data
        print(f"Enqueued: {data}")

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
            return
        removed = self.queue[self.front]
        if self.front == self.rear:
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.size
        print(f"Dequeued: {removed}")

    def display(self):
        if self.is_empty():
            print("Queue is empty.")
            return
        i = self.front
        print("Circular Queue:", end=" ")
        while True:
            print(self.queue[i], end=" -> " if i != self.rear else " ")
            if i == self.rear:
                break
            i = (i + 1) % self.size
        print()

if __name__ == "__main__":
    size = int(input("Enter size of circular queue: "))
```

```
    circular_queue = CircularQueue(size)

    while True:
        print("\nMenu:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Display")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to enqueue: "))
            circular_queue.enqueue(data)
        elif choice == 2:
            circular_queue.dequeue()
        elif choice == 3:
            circular_queue.display()
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Enter size of circular queue: 3

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
```

[39]:
```
print("slip 11")
```

```
slip 11
```

[41]:
```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

if __name__ == "__main__":
    arr = list(map(int, input("Enter integers to sort (space-separated): ").
 ↪split()))
```

```
    selection_sort(arr)
    print("Sorted array:", arr)
```

```
Enter integers to sort (space-separated): 243 45 46
Sorted array: [45, 46, 243]
```

[42]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
        print(f"Pushed: {data}")

    def pop(self):
        if self.is_empty():
            print("Stack is empty. Cannot pop.")
            return None
        popped_node = self.top
        self.top = self.top.next
        print(f"Popped: {popped_node.data}")
        return popped_node.data

    def is_empty(self):
        return self.top is None

if __name__ == "__main__":
    stack = Stack()

    while True:
        print("\nMenu:")
        print("1. Push")
        print("2. Pop")
        print("3. Check if Empty")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to push onto stack: "))
            stack.push(data)
```

```
        elif choice == 2:
            stack.pop()
        elif choice == 3:
            if stack.is_empty():
                print("Stack is empty.")
            else:
                print("Stack is not empty.")
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Push
2. Pop
3. Check if Empty
4. Exit
Enter your choice: 4
```

[45]:
```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

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

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert_rec(self.root, key)

    def _insert_rec(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert_rec(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert_rec(node.right, key)
```

```
    def inorder_traversal(self, node):
        if node:
            self.inorder_traversal(node.left)
            print(node.val, end=" ")
            self.inorder_traversal(node.right)

    def postorder_traversal(self, node):
        if node:
            self.postorder_traversal(node.left)
            self.postorder_traversal(node.right)
            print(node.val, end=" ")

if __name__ == "__main__":
    bst = BinarySearchTree()

    n = int(input("Enter number of integers to insert into the BST: "))
    for _ in range(n):
        key = int(input("Enter integer: "))
        bst.insert(key)

    print("In-order traversal:")
    bst.inorder_traversal(bst.root)
    print("\nPost-order traversal:")
    bst.postorder_traversal(bst.root)
```

```
Enter number of integers to insert into the BST: 3
Enter integer: 23
Enter integer: 345
Enter integer: 645
In-order traversal:
23 345 645
Post-order traversal:
645 345 23
```

[47]:
```
print("slip 12")
```

```
slip 12
```

[49]:
```
def linear_search(arr, target):
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1

if __name__ == "__main__":
    arr = list(map(int, input("Enter integers (space-separated): ").split()))
```

```
    target = int(input("Enter the element to search for: "))

    result = linear_search(arr, target)
    if result != -1:
        print(f"Element {target} found at index {result}.")
    else:
        print(f"Element {target} not found in the array.")
```

```
Enter integers (space-separated): 23 34 13 453
Enter the element to search for: 34
Element 34 found at index 1.
```

[51]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None

    def insert(self, data):
        new_node = Node(data)
        if self.rear is None:
            self.front = self.rear = new_node
            print(f"Inserted: {data}")
            return
        self.rear.next = new_node
        self.rear = new_node
        print(f"Inserted: {data}")

    def delete(self):
        if self.front is None:
            print("Queue is empty. Cannot delete.")
            return
        removed_node = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        print(f"Deleted: {removed_node.data}")

    def display(self):
        if self.front is None:
            print("Queue is empty.")
            return
        current = self.front
```

```
        print("Queue:", end=" ")
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

if __name__ == "__main__":
    queue = Queue()

    while True:
        print("\nMenu:")
        print("1. Insert")
        print("2. Delete")
        print("3. Display")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to insert into queue: "))
            queue.insert(data)
        elif choice == 2:
            queue.delete()
        elif choice == 3:
            queue.display()
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4
```

[52]:
```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

if __name__ == "__main__":
```

```
    arr = list(map(int, input("Enter integers to sort (space-separated): ").
↪split()))
    sorted_arr = quick_sort(arr)
    print("Sorted array:", sorted_arr)
```

```
Enter integers to sort (space-separated): 34 45 23 54 123
Sorted array: [23, 34, 45, 54, 123]
```

[53]:
```
print("slip 13")
```

```
slip 13
```

[54]:
```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

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

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

if __name__ == "__main__":
    arr = list(map(int, input("Enter integers to sort (space-separated): ").
↪split()))
    merge_sort(arr)
    print("Sorted array:", arr)
```

```
Enter integers to sort (space-separated): 34 456 24
Sorted array: [24, 34, 456]
```

[55]:
```python
class LinearQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def insert(self, data):
        if self.rear == self.size - 1:
            print("Queue is full. Cannot insert.")
            return
        if self.front == -1:
            self.front = 0
        self.rear += 1
        self.queue[self.rear] = data
        print(f"Inserted: {data}")

    def delete(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue is empty. Cannot delete.")
            return
        deleted_item = self.queue[self.front]
        self.front += 1
        if self.front > self.rear:
            self.front = self.rear = -1
        print(f"Deleted: {deleted_item}")

    def empty(self):
        return self.front == -1 or self.front > self.rear

if __name__ == "__main__":
    size = int(input("Enter the size of the queue: "))
    queue = LinearQueue(size)

    while True:
        print("\nMenu:")
        print("1. Insert")
        print("2. Delete")
        print("3. Check if Empty")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to insert into queue: "))
```

```
            queue.insert(data)
        elif choice == 2:
            queue.delete()
        elif choice == 3:
            if queue.empty():
                print("Queue is empty.")
            else:
                print("Queue is not empty.")
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Enter the size of the queue: 3

Menu:
1. Insert
2. Delete
3. Check if Empty
4. Exit
Enter your choice: 4
```

[56]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def create(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def display(self):
        if not self.head:
            print("List is empty.")
            return
        current = self.head
        print("Singly Linked List:", end=" ")
```

```
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

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

    def delete(self, target):
        if not self.head:
            print("List is empty.")
            return
        if self.head.data == target:
            self.head = self.head.next
            print(f"Deleted: {target}")
            return
        current = self.head
        while current.next:
            if current.next.data == target:
                current.next = current.next.next
                print(f"Deleted: {target}")
                return
            current = current.next
        print(f"Element {target} not found in the list.")

if __name__ == "__main__":
    sll = SinglyLinkedList()

    while True:
        print("\nMenu:")
        print("1. Create")
        print("2. Display")
        print("3. Search specific element")
        print("4. Delete specific element")
        print("5. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to insert into the list: "))
            sll.create(data)
        elif choice == 2:
            sll.display()
```

```
        elif choice == 3:
            target = int(input("Enter element to search for: "))
            if sll.search(target):
                print(f"Element {target} found in the list.")
            else:
                print(f"Element {target} not found in the list.")
        elif choice == 4:
            target = int(input("Enter element to delete: "))
            sll.delete(target)
        elif choice == 5:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Create
2. Display
3. Search specific element
4. Delete specific element
5. Exit
Enter your choice: 5
```

[57]:
```
print("slip 14")
```

```
slip 14
```

[58]:
```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break

if __name__ == "__main__":
    arr = list(map(int, input("Enter integers to sort (space-separated): ").
 ↪split()))
    bubble_sort(arr)
    print("Sorted array:", arr)
```

```
Enter integers to sort (space-separated): 243 45 324
Sorted array: [45, 243, 324]
```

[59]:
```python
class Stack:
    def __init__(self):
        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 is_empty(self):
        return len(self.items) == 0

def is_palindrome(string):
    stack = Stack()

    for char in string:
        stack.push(char)

    reversed_string = ''
    while not stack.is_empty():
        reversed_string += stack.pop()

    return string == reversed_string

if __name__ == "__main__":
    input_string = input("Enter a string: ")
    if is_palindrome(input_string):
        print(f"{input_string} is a palindrome.")
    else:
        print(f"{input_string} is not a palindrome.")
```

```
Enter a string: apples
apples is not a palindrome.
```

[60]:
```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
```

```
            print("Stack is empty. Cannot pop.")
            return None
        popped_item = self.items.pop()
        print(f"Popped: {popped_item}")
        return popped_item

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

if __name__ == "__main__":
    stack = Stack()

    while True:
        print("\nMenu:")
        print("1. Push")
        print("2. Pop")
        print("3. Check if Empty")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to push onto stack: "))
            stack.push(data)
        elif choice == 2:
            stack.pop()
        elif choice == 3:
            if stack.is_empty():
                print("Stack is empty.")
            else:
                print("Stack is not empty.")
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Push
2. Pop
3. Check if Empty
4. Exit
Enter your choice: 1
Enter integer to push onto stack: 23
Pushed: 23

Menu:
1. Push
```

```
2. Pop
3. Check if Empty
4. Exit
Enter your choice: 3
Stack is not empty.

Menu:
1. Push
2. Pop
3. Check if Empty
4. Exit
Enter your choice: 4
```

[61]:
```python
print("slip 15")
```

```
slip 15
```

[62]:
```python
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

if __name__ == "__main__":
    arr = list(map(int, input("Enter integers to sort (space-separated): ").
 ↪split()))
    insertion_sort(arr)
    print("Sorted array:", arr)
```

```
Enter integers to sort (space-separated): 34 45 65
Sorted array: [34, 45, 65]
```

[63]:
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None

    def insert(self, data):
        new_node = Node(data)
```

```
        if self.rear is None:
            self.front = self.rear = new_node
            print(f"Inserted: {data}")
            return
        self.rear.next = new_node
        self.rear = new_node
        print(f"Inserted: {data}")

    def delete(self):
        if self.front is None:
            print("Queue is empty. Cannot delete.")
            return
        deleted_item = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        print(f"Deleted: {deleted_item}")

    def is_empty(self):
        return self.front is None

if __name__ == "__main__":
    queue = Queue()

    while True:
        print("\nMenu:")
        print("1. Insert")
        print("2. Delete")
        print("3. Check if Empty")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter integer to insert into queue: "))
            queue.insert(data)
        elif choice == 2:
            queue.delete()
        elif choice == 3:
            if queue.is_empty():
                print("Queue is empty.")
            else:
                print("Queue is not empty.")
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please try again.")
```

```
Menu:
1. Insert
2. Delete
3. Check if Empty
4. Exit
Enter your choice: 1
Enter integer to insert into queue: 23
Inserted: 23

Menu:
1. Insert
2. Delete
3. Check if Empty
4. Exit
Enter your choice: 3
Queue is not empty.

Menu:
1. Insert
2. Delete
3. Check if Empty
4. Exit
Enter your choice: 4
```

[64]:
```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.value = key

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

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_rec(self.root, key)

    def _insert_rec(self, node, key):
        if key < node.value:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_rec(node.left, key)
```

```
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_rec(node.right, key)

    def pre_order(self, node):
        if node:
            print(node.value, end=' ')
            self.pre_order(node.left)
            self.pre_order(node.right)

    def post_order(self, node):
        if node:
            self.post_order(node.left)
            self.post_order(node.right)
            print(node.value, end=' ')

if __name__ == "__main__":
    bst = BST()
    elements = list(map(int, input("Enter integers to insert into BST␣
 ↪(space-separated): ").split()))

    for element in elements:
        bst.insert(element)

    print("Pre-order traversal:")
    bst.pre_order(bst.root)

    print("\nPost-order traversal:")
    bst.post_order(bst.root)
```

```
Enter integers to insert into BST (space-separated): 343 654 65
Pre-order traversal:
343 65 654
Post-order traversal:
65 654 343
```

[65]:
```
print("slip 16")
```

```
slip 16
```

[66]:
```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
```

```
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

if __name__ == "__main__":
    arr = list(map(int, input("Enter integers to sort (space-separated): ").
 ↪split()))
    selection_sort(arr)
    print("Sorted array:", arr)
```

```
Enter integers to sort (space-separated): 343 45 23
Sorted array: [23, 45, 343]
```

[67]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def reverse(self):
        prev = None
        current = self.head
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        self.head = prev

    def display(self):
        current = self.head
        while current:
            print(current.data, end=' ')
            current = current.next
```

```
        print()

if __name__ == "__main__":
    linked_list = SinglyLinkedList()

    elements = list(map(int, input("Enter integers to insert into linked list␣
 ↪(space-separated): ").split()))
    for element in elements:
        linked_list.append(element)

    print("Original linked list:")
    linked_list.display()

    linked_list.reverse()

    print("Reversed linked list:")
    linked_list.display()
```

```
Enter integers to insert into linked list (space-separated): 23 45 67 23
Original linked list:
23 45 67 23
Reversed linked list:
23 67 45 23
```

[70]:
```
class Stack:
    def __init__(self):
        self.items = []

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

    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 precedence(self, op):
        if op == '+' or op == '-':
            return 1
```

```
        if op == '*' or op == '/':
            return 2
        return 0

def infix_to_postfix(expression):
    stack = Stack()
    postfix = []
    for char in expression:
        if char.isalnum():
            postfix.append(char)
        elif char == '(':
            stack.push(char)
        elif char == ')':
            while not stack.is_empty() and stack.peek() != '(':
                postfix.append(stack.pop())
            stack.pop()
        else:
            while (not stack.is_empty() and
                   stack.precedence(char) <= stack.precedence(stack.peek())):
                postfix.append(stack.pop())
            stack.push(char)

    while not stack.is_empty():
        postfix.append(stack.pop())

    return ''.join(postfix)

if __name__ == "__main__":
    expression = input("Enter an infix expression: ")
    result = infix_to_postfix(expression)
    print("Postfix expression:", result)
```

```
Enter an infix expression: 2+3
Postfix expression: 23+
```

[71]:
```
print("slip 17")
```

```
slip 17
```

[74]:
```
print("wait")
```

```
wait
```

[78]:
```
class Stack:
    def __init__(self):
        self.items = []
```

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

    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 precedence(self, op):
        if op == '+' or op == '-':
            return 1
        if op == '*' or op == '/':
            return 2
        return 0

def infix_to_prefix(expression):
    expression = expression.replace('(', ' ) ').replace(')', ' ( ')
    tokens = expression.split()[::-1]
    stack = Stack()
    prefix = []

    for token in tokens:
        if token.isalnum():
            prefix.append(token)
        elif token == '(':
            stack.push(token)
        elif token == ')':
            while not stack.is_empty() and stack.peek() != '(':
                prefix.append(stack.pop())
            stack.pop()
        else:
            while (not stack.is_empty() and
                   stack.precedence(token) < stack.precedence(stack.peek())):
                prefix.append(stack.pop())
            stack.push(token)

    while not stack.is_empty():
        prefix.append(stack.pop())
```

```
    return ' '.join(prefix[::-1])

if __name__ == "__main__":
    expression = input("Enter an infix expression: ")
    result = infix_to_prefix(expression)
    print("Prefix expression:", result)
```

```
Enter an infix expression: v + f * c
Prefix expression: + v * f c
```

[79]:
```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.adjacency_matrix = [[0 for _ in range(vertices)] for _ in␣
 ↪range(vertices)]

    def add_edge(self, u, v):
        self.adjacency_matrix[u][v] = 1
        self.adjacency_matrix[v][u] = 1  # For undirected graph

    def display(self):
        print("Adjacency Matrix:")
        for row in self.adjacency_matrix:
            print(" ".join(map(str, row)))

if __name__ == "__main__":
    n = int(input("Enter the number of vertices: "))
    graph = Graph(n)

    while True:
        print("\nEnter edges (u, v) to add an edge between vertices u and v:")
        print("Type 'done' to finish adding edges.")
        user_input = input("Edge (u, v): ")

        if user_input.lower() == 'done':
            break

        try:
            u, v = map(int, user_input.split(','))
            if 0 <= u < n and 0 <= v < n:
                graph.add_edge(u, v)
            else:
                print(f"Vertices must be between 0 and {n - 1}.")
        except ValueError:
            print("Invalid input. Please enter in the format 'u, v'.")

    graph.display()
```

```
Enter the number of vertices: 2

Enter edges (u, v) to add an edge between vertices u and v:
Type 'done' to finish adding edges.
Edge (u, v): 2 3
Invalid input. Please enter in the format 'u, v'.

Enter edges (u, v) to add an edge between vertices u and v:
Type 'done' to finish adding edges.
Edge (u, v): 2 3
Invalid input. Please enter in the format 'u, v'.

Enter edges (u, v) to add an edge between vertices u and v:
Type 'done' to finish adding edges.
Edge (u, v): 2 3
Invalid input. Please enter in the format 'u, v'.

Enter edges (u, v) to add an edge between vertices u and v:
Type 'done' to finish adding edges.
Edge (u, v): 2 3
Invalid input. Please enter in the format 'u, v'.

Enter edges (u, v) to add an edge between vertices u and v:
Type 'done' to finish adding edges.
Edge (u, v): 4 5
Invalid input. Please enter in the format 'u, v'.

Enter edges (u, v) to add an edge between vertices u and v:
Type 'done' to finish adding edges.
Edge (u, v): 3 5
Invalid input. Please enter in the format 'u, v'.

Enter edges (u, v) to add an edge between vertices u and v:
Type 'done' to finish adding edges.
Edge (u, v): done
Adjacency Matrix:
0 0
0 0
```