```
1.def factorial(n):
 if n == 0 or n == 1:
   return 1
    else:
    return n * factorial(n - 1)
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
  else:
        return fibonacci(n - 1) + fibonacci(n - 2)
num = int(input("Enter a number: "))
print(f"Factorial of {num} is: {factorial(num)}")
print(f"Fibonacci series up to {num} terms:")
for i in range(num):
    print(fibonacci(i), end=" ")
2. import time
def factorial_recursive(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial_recursive(n - 1)
def factorial_iterative(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
def fibonacci_recursive(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
def fibonacci_iterative(n):
    a, b = 0, 1
```

```
    for _ in range(n):
        a, b = b, a + b
    return a
n = int(input("Enter a number: "))
start = time.time()
fact_r = factorial_recursive(n)
end = time.time()
print(f"Recursive Factorial of {n} = {fact_r}")
print(f"Time (recursive): {end - start:.8f} seconds")
start = time.time()
fact_i = factorial_iterative(n)
end = time.time()
print(f"Iterative Factorial of {n} = {fact_i}")
print(f"Time (iterative): {end - start:.8f} seconds")
print("-" * 50)
start = time.time()
fib_r = fibonacci_recursive(n)
end = time.time()
print(f"Recursive Fibonacci({n}) = {fib_r}")
print(f"Time (recursive): {end - start:.8f} seconds")
start = time.time()
fib_i = fibonacci_iterative(n)
end = time.time()
print(f"Iterative Fibonacci({n}) = {fib_i}")
print(f"Time (iterative): {end - start:.8f} seconds")
```

3.
```
arr = [10, 20, 30, 40, 50]
print("Initial Array:", arr)
print("\nTraversal of array:")
for i in range(len(arr)):
    print(arr[i], end=" ")
print()
index = 2
element = 25
arr.insert(index, element)
print(f"\nAfter inserting {element} at index {index}: {arr}")
middle_index = len(arr) // 2
```

```
element = 35
arr.insert(middle_index, element)
print(f"After inserting {element} at middle index {middle_index}: {arr}")
index = 3
deleted_element = arr.pop(index)
print(f"\nAfter deleting element at index {index} ({deleted_element}): {arr}")
middle_index = len(arr) // 2
deleted_element = arr.pop(middle_index)
print(f"After deleting middle element ({deleted_element}): {arr}")
element_to_search = 40
found = False
for i in range(len(arr)):
    if arr[i] == element_to_search:
        print(f"\nElement {element_to_search} found at index {i}")
        found = True
        break
if not found:
    print(f"\nElement {element_to_search} not found in the array")
print("\nFinal Array Traversal:")
for i in range(len(arr)):
    print(arr[i], end=" ")
print()
```

4. # Node class to represent each element in the linked list

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
    def insert_at_end(self, data):
        new_node = Node(data)
```

```
    if not self.head:
        self.head = new_node
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.next = new_node
def insert_at_index(self, index, data):
    if index == 0:
        self.insert_at_beginning(data)
        return
    new_node = Node(data)
    temp = self.head
    for _ in range(index - 1):
        if not temp:
            print("Index out of range")
            return
        temp = temp.next
    new_node.next = temp.next
    temp.next = new_node
def insert_after(self, target, data):
    temp = self.head
    while temp and temp.data != target:
        temp = temp.next
    if not temp:
        print(f"Element {target} not found.")
        return
    new_node = Node(data)
    new_node.next = temp.next
    temp.next = new_node
def insert_before(self, target, data):
    if not self.head:
        print("List is empty.")
        return
    if self.head.data == target:
        self.insert_at_beginning(data)
```

```
            return
        temp = self.head
        while temp.next and temp.next.data != target:
            temp = temp.next
        if not temp.next:
            print(f"Element {target} not found.")
            return
        new_node = Node(data)
        new_node.next = temp.next
        temp.next = new_node
    def delete_at_beginning(self):
        if self.head:
            self.head = self.head.next
    def delete_at_end(self):
        if not self.head:
            return
        if not self.head.next:
            self.head = None
            return
        temp = self.head
        while temp.next.next:
            temp = temp.next
        temp.next = None
    def delete_at_index(self, index):
        if not self.head:
            return
        if index == 0:
            self.delete_at_beginning()
            return
        temp = self.head
        for _ in range(index - 1):
            if not temp.next:
                print("Index out of range")
                return
            temp = temp.next
        if temp.next:
```

```
        temp.next = temp.next.next
    def delete_after(self, target):
        temp = self.head
        while temp and temp.data != target:
            temp = temp.next
        if temp and temp.next:
            temp.next = temp.next.next
    def delete_before(self, target):
        if not self.head or not self.head.next:
            return
        if self.head.next.data == target:
            self.delete_at_beginning()
            return
        prev = None
        temp = self.head
        while temp.next and temp.next.data != target:
            prev = temp
            temp = temp.next
        if temp.next:
            prev.next = temp.next
    def traverse(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")
    def size(self):
        count = 0
        temp = self.head
        while temp:
            count += 1
            temp = temp.next
        return count
    def sort_list(self):
        if not self.head:
            return
```

```
        temp1 = self.head
        while temp1:
            temp2 = temp1.next
            while temp2:
                if temp1.data > temp2.data:
                    temp1.data, temp2.data = temp2.data, temp1.data
                temp2 = temp2.next
            temp1 = temp1.next
ll = SinglyLinkedList()
ll.insert_at_end(50)
ll.insert_at_beginning(20)
ll.insert_at_end(70)
ll.insert_at_index(1, 30)
ll.insert_after(30, 40)
ll.insert_before(50, 45)
print("After Insertions:")
ll.traverse()
ll.delete_at_beginning()
ll.delete_at_end()
ll.delete_at_index(2)
ll.delete_after(30)
ll.delete_before(70)
print("\nAfter Deletions:")
ll.traverse()
ll.sort_list()
print("\nAfter Sorting:")
ll.traverse()
print(f"\nSize of the list: {ll.size()}")
print("\nFinal Linked List Elements after all operations:")
ll.traverse()
```

5. class Node:

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

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        if self.head:
            self.head.prev = new_node
        self.head = new_node
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
        new_node.prev = temp
    def insert_at_index(self, index, data):
        if index == 0:
            self.insert_at_beginning(data)
            return
        new_node = Node(data)
        temp = self.head
        for _ in range(index - 1):
            if not temp:
                print("Index out of range.")
                return
            temp = temp.next
        if not temp:
            print("Index out of range.")
            return
        new_node.next = temp.next
        new_node.prev = temp
        if temp.next:
```

```
        temp.next.prev = new_node
    temp.next = new_node
def insert_after(self, target, data):
    temp = self.head
    while temp and temp.data != target:
        temp = temp.next
    if not temp:
        print(f"Element {target} not found.")
        return
    new_node = Node(data)
    new_node.next = temp.next
    new_node.prev = temp
    if temp.next:
        temp.next.prev = new_node
    temp.next = new_node
def insert_before(self, target, data):
    temp = self.head
    while temp and temp.data != target:
        temp = temp.next
    if not temp:
        print(f"Element {target} not found.")
        return
    new_node = Node(data)
    new_node.next = temp
    new_node.prev = temp.prev
    if temp.prev:
        temp.prev.next = new_node
    else:
        self.head = new_node
    temp.prev = new_node
def delete_at_beginning(self):
    if not self.head:
        return
    if not self.head.next:
        self.head = None
        return
```

```
    self.head = self.head.next
    self.head.prev = None
def delete_at_end(self):
    if not self.head:
        return
    if not self.head.next:
        self.head = None
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.prev.next = None
def delete_at_index(self, index):
    if not self.head:
        return
    if index == 0:
        self.delete_at_beginning()
        return
    temp = self.head
    for _ in range(index):
        if not temp:
            print("Index out of range.")
            return
        temp = temp.next
    if not temp:
        print("Index out of range.")
        return
    if temp.next:
        temp.next.prev = temp.prev
    if temp.prev:
        temp.prev.next = temp.next
def delete_after(self, target):
    temp = self.head
    while temp and temp.data != target:
        temp = temp.next
    if temp and temp.next:
```

```
            to_delete = temp.next
            temp.next = to_delete.next
            if to_delete.next:
                to_delete.next.prev = temp
    def delete_before(self, target):
        temp = self.head
        while temp and temp.data != target:
            temp = temp.next
        if temp and temp.prev:
            to_delete = temp.prev
            if to_delete.prev:
                to_delete.prev.next = temp
                temp.prev = to_delete.prev
            else:
                self.head = temp
                temp.prev = None
    def traverse(self):
        temp = self.head
        while temp:
            print(temp.data, end=" <-> ")
            temp = temp.next
        print("None")
    def size(self):
        count = 0
        temp = self.head
        while temp:
            count += 1
            temp = temp.next
        return count
    def sort_list(self):
        if not self.head:
            return
        swapped = True
        while swapped:
            swapped = False
            temp = self.head
```

```
        while temp.next:
            if temp.data > temp.next.data:
                temp.data, temp.next.data = temp.next.data, temp.data
                swapped = True
            temp = temp.next
dll = DoublyLinkedList()
dll.insert_at_end(50)
dll.insert_at_beginning(20)
dll.insert_at_end(80)
dll.insert_at_index(1, 30)
dll.insert_after(30, 40)
dll.insert_before(50, 45)
print("After Insertions:")
dll.traverse()
dll.delete_at_beginning()
dll.delete_at_end()
dll.delete_at_index(2)
dll.delete_after(30)
dll.delete_before(80)
print("\nAfter Deletions:")
dll.traverse()
dll.sort_list()
print("\nAfter Sorting:")
dll.traverse()
print(f"\nSize of the list: {dll.size()}")
print("\nFinal Doubly Linked List Elements after all operations:")
dll.traverse()
```

6. class Node:

```
    def __init__(self, data):
        self.data = data
        self.next = None
class CircularLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_beginning(self, data):
        new_node = Node(data)
```

```
        if not self.head:
            new_node.next = new_node
            self.head = new_node
            return
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        new_node.next = self.head
        temp.next = new_node
        self.head = new_node
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            new_node.next = new_node
            self.head = new_node
            return
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = new_node
        new_node.next = self.head
    def insert_at_index(self, index, data):
        if index == 0:
            self.insert_at_beginning(data)
            return
        new_node = Node(data)
        temp = self.head
        count = 0
        while count < index - 1:
            temp = temp.next
            count += 1
            if temp == self.head:
                print("Index out of range.")
                return
        new_node.next = temp.next
        temp.next = new_node
```

```
def insert_after(self, target, data):
    temp = self.head
    while True:
        if temp.data == target:
            new_node = Node(data)
            new_node.next = temp.next
            temp.next = new_node
            return
        temp = temp.next
        if temp == self.head:
            break
    print(f"Element {target} not found.")
def insert_before(self, target, data):
    if not self.head:
        print("List is empty.")
        return
    if self.head.data == target:
        self.insert_at_beginning(data)
        return
    prev = self.head
    temp = self.head.next
    while temp != self.head:
        if temp.data == target:
            new_node = Node(data)
            new_node.next = temp
            prev.next = new_node
            return
        prev = temp
        temp = temp.next
    print(f"Element {target} not found.")
def delete_at_beginning(self):
    if not self.head:
        return
    if self.head.next == self.head:
        self.head = None
        return
```

```
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = self.head.next
        self.head = self.head.next
    def delete_at_end(self):
        if not self.head:
            return
        if self.head.next == self.head:
            self.head = None
            return
        prev = None
        temp = self.head
        while temp.next != self.head:
            prev = temp
            temp = temp.next
        prev.next = self.head
    def delete_at_index(self, index):
        if not self.head:
            return
        if index == 0:
            self.delete_at_beginning()
            return
        prev = self.head
        temp = self.head.next
        count = 1
        while temp != self.head and count < index:
            prev = temp
            temp = temp.next
            count += 1
        if temp == self.head:
            print("Index out of range.")
            return
        prev.next = temp.next
    def delete_after(self, target):
        temp = self.head
```

```
        while True:
            if temp.data == target:
                if temp.next == self.head:
                    temp.next = temp.next.next
                else:
                    temp.next = temp.next.next
                return
            temp = temp.next
            if temp == self.head:
                break
        print(f"Element {target} not found.")
    def delete_before(self, target):
        if not self.head or self.head.next == self.head:
            return
        prev = None
        curr = self.head
        nxt = curr.next
        if nxt.data == target:
            self.delete_at_beginning()
            return
        while nxt != self.head:
            if nxt.data == target:
                if prev:
                    prev.next = nxt
                return
            prev = curr
            curr = nxt
            nxt = nxt.next
        print(f"Element {target} not found.")
    def traverse(self):
        if not self.head:
            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 size(self):
        if not self.head:
            return 0
        count = 1
        temp = self.head.next
        while temp != self.head:
            count += 1
            temp = temp.next
        return count
    def sort_list(self):
        if not self.head or self.head.next == self.head:
            return
        swapped = True
        while swapped:
            swapped = False
            temp = self.head
            while temp.next != self.head:
                if temp.data > temp.next.data:
                    temp.data, temp.next.data = temp.next.data, temp.data
                    swapped = True
                temp = temp.next
cll = CircularLinkedList()
cll.insert_at_end(50)
cll.insert_at_beginning(20)
cll.insert_at_end(80)
cll.insert_at_index(1, 30)
cll.insert_after(30, 40)
cll.insert_before(50, 45)
print("After Insertions:")
cll.traverse()
cll.delete_at_beginning()
cll.delete_at_end()
```

```
cll.delete_at_index(2)
cll.delete_after(30)
cll.delete_before(80)
print("\nAfter Deletions:")
cll.traverse()
cll.sort_list()
print("\nAfter Sorting:")
cll.traverse()
print(f"\nSize of the list: {cll.size()}")
print("\nFinal Circular Linked List Elements after all operations:")
cll.traverse()
```

7.class StackArray:

```
class StackArray:
    def __init__(self):
        self.stack = []
    def push(self, data):
        self.stack.append(data)
    def pop(self):
        if not self.stack:
            print("Stack is empty!")
            return None
        return self.stack.pop()
    def peek(self):
        if not self.stack:
            print("Stack is empty!")
            return None
        return self.stack[-1]
    def length(self):
        return len(self.stack)
    def display(self):
        print("Stack (top -> bottom):", self.stack[::-1])
stack_arr = StackArray()
stack_arr.push(10)
stack_arr.push(20)
stack_arr.push(30)
print("Array Stack after pushes:")
stack_arr.display()
```

```
print("Peek top element:", stack_arr.peek())
print("Pop top element:", stack_arr.pop())
print("Length of stack:", stack_arr.length())
print("Stack after pop:")
stack_arr.display()
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class StackLinkedList:
    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} onto stack")
  def pop(self):
        if self.is_empty():
            print("Stack is empty")
            return None
        removed = self.top.data
        self.top = self.top.next
        print(f"Popped {removed} from stack")
        return removed
    def peek(self):
        if self.is_empty():
            print("Stack is empty")
            return None
        print(f"Top element is {self.top.data}")
        return self.top.data
    def is_empty(self):
        return self.top is None
    def sort_stack(self):
        if self.is_empty():
            return
```

```
        elements = []
        current = self.top
        while current:
            elements.append(current.data)
            current = current.next
        elements.sort()
        self.top = None
        for data in reversed(elements):
            self.push(data)
        print("Stack sorted")
    def display(self):
        current = self.top
        elements = []
        while current:
            elements.append(current.data)
            current = current.next
        print("Stack:", elements)
stack_ll = StackLinkedList()
stack_ll.push(10)
stack_ll.push(3)
stack_ll.push(7)
stack_ll.display()  # Stack: [7, 3, 10]
stack_ll.peek()     # Top element is 7
stack_ll.pop()      # Popped 7
stack_ll.display()  # Stack: [3, 10]
stack_ll.sort_stack()
stack_ll.display()  # Stack: [3, 10]
```

8.class ArrayQueue:

```
    def __init__(self):
        self.queue = []
    def enqueue(self, data):
        self.queue.append(data)
        print(f"Enqueued {data}")
    def dequeue(self):
        if not self.queue:
            print("Queue is empty")
```

```
            return None
        data = self.queue.pop(0)
        print(f"Dequeued {data}")
        return data
    def search(self, data):
        if data in self.queue:
            index = self.queue.index(data)
            print(f"{data} found at position {index}")
            return index
        print(f"{data} not found in queue")
        return -1
    def sort(self):
        self.queue.sort()
        print("Queue sorted:", self.queue)
    def length(self):
        print("Queue length:", len(self.queue))
        return len(self.queue)
    def display(self):
        print("Queue:", self.queue)
print("=== Array-based Queue ===")
aq = ArrayQueue()
aq.enqueue(10)
aq.enqueue(30)
aq.enqueue(20)
aq.display()
aq.dequeue()
aq.search(30)
aq.sort()
aq.length()
aq.display()
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedListQueue:
    def __init__(self):
```

```
        self.front = None
        self.rear = None
    def enqueue(self, data):
        new_node = Node(data)
        if not self.front:
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
        print(f"Enqueued {data}")
    def dequeue(self):
        if not self.front:
            print("Queue is empty")
            return None
        data = self.front.data
        self.front = self.front.next
        if not self.front:
            self.rear = None
        print(f"Dequeued {data}")
        return data
    def search(self, data):
        current = self.front
        pos = 0
        while current:
            if current.data == data:
                print(f"{data} found at position {pos}")
                return pos
            current = current.next
            pos += 1
        print(f"{data} not found in queue")
        return -1
    def sort(self):
        elements = []
        current = self.front
        while current:
            elements.append(current.data)
```

```
            current = current.next
        elements.sort()
        self.front = self.rear = None
        for data in elements:
            self.enqueue(data)
        print("Queue sorted")
    def length(self):
        count = 0
        current = self.front
        while current:
            count += 1
            current = current.next
        print("Queue length:", count)
        return count
    def display(self):
        elements = []
        current = self.front
        while current:
            elements.append(current.data)
            current = current.next
        print("Queue:", elements)
print("\n=== Linked List-based Queue ===")
lq = LinkedListQueue()
lq.enqueue(50)
lq.enqueue(10)
lq.enqueue(30)
lq.display()
lq.dequeue()
lq.search(30)
lq.sort()
lq.length()
lq.display()
```

12. class CircularQueueArray:

```
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
```

```
        self.front = -1
        self.rear = -1
    def isFull(self):
        return (self.rear + 1) % self.capacity == self.front
    def isEmpty(self):
        return self.front == -1
    def enqueue(self, data):
        if self.isFull():
            print("Queue is full!")
            return
        if self.isEmpty():
            self.front = 0
        self.rear = (self.rear + 1) % self.capacity
        self.queue[self.rear] = data
        print(f"Enqueued: {data}")
    def dequeue(self):
        if self.isEmpty():
            print("Queue is empty!")
            return None
        data = self.queue[self.front]
        if self.front == self.rear:
            self.front = self.rear = -1  # Queue becomes empty
        else:
            self.front = (self.front + 1) % self.capacity
        print(f"Dequeued: {data}")
        return data
    def peek(self):
        if self.isEmpty():
            print("Queue is empty!")
            return None
        return self.queue[self.front]
    def display(self):
        if self.isEmpty():
            print("Queue is empty!")
            return
        print("Circular Queue (front -> rear):", end=" ")
```

```
        i = self.front
        while True:
            print(self.queue[i], end=" ")
            if i == self.rear:
                break
            i = (i + 1) % self.capacity
        print()
cq = CircularQueueArray(5)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.display()
print("Peek:", cq.peek())
cq.dequeue()
cq.display()
cq.enqueue(40)
cq.enqueue(50)
cq.enqueue(60)  # This should wrap around
cq.display()
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class CircularQueueLinkedList:
    def __init__(self):
        self.front = None
        self.rear = None
    def isEmpty(self):
        return self.front is None
    def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = self.rear = new_node
            self.rear.next = self.front  # Circular link
        else:
            self.rear.next = new_node
```

```
            self.rear = new_node
            self.rear.next = self.front
        print(f"Enqueued: {data}")
    def dequeue(self):
        if self.isEmpty():
            print("Queue is empty!")
            return None
        data = self.front.data
        if self.front == self.rear:  # Only one element
            self.front = self.rear = None
        else:
            self.front = self.front.next
            self.rear.next = self.front
        print(f"Dequeued: {data}")
        return data
    def peek(self):
        if self.isEmpty():
            print("Queue is empty!")
            return None
        return self.front.data
    def display(self):
        if self.isEmpty():
            print("Queue is empty!")
            return
        print("Circular Queue (front -> rear):", end=" ")
        temp = self.front
        while True:
            print(temp.data, end=" ")
            temp = temp.next
            if temp == self.front:
                break
        print()
cq_ll = CircularQueueLinkedList()
cq_ll.enqueue(10)
cq_ll.enqueue(20)
cq_ll.enqueue(30)
```

```
cq_ll.display()
print("Peek:", cq_ll.peek())
cq_ll.dequeue()
cq_ll.display()
cq_ll.enqueue(40)
cq_ll.enqueue(50)
cq_ll.display()
```

1.implement binary tree using arrays and linked list and perform operations like insertions deletions searching min max length sorting traversals

```
class ArrayBinaryTree:
    def __init__(self):
        self.tree = []
    def insert(self, value):
        self.tree.append(value)
        print(f"Inserted {value}")
    def delete(self, value):
        if value in self.tree:
            index = self.tree.index(value)
            last = self.tree.pop()
            if index < len(self.tree):
                self.tree[index] = last
            print(f"Deleted {value}")
        else:
            print(f"{value} not found")
    def search(self, value):
        if value in self.tree:
            index = self.tree.index(value)
            print(f"{value} found at index {index}")
            return index
        print(f"{value} not found")
        return -1
    def inorder(self, index=0):
        res = []
        if index < len(self.tree):
            res += self.inorder(2*index+1)
            res.append(self.tree[index])
            res += self.inorder(2*index+2)
```

```
        return res
    def preorder(self, index=0):
        res = []
        if index < len(self.tree):
            res.append(self.tree[index])
            res += self.preorder(2*index+1)
            res += self.preorder(2*index+2)
        return res
    def postorder(self, index=0):
        res = []
        if index < len(self.tree):
            res += self.postorder(2*index+1)
            res += self.postorder(2*index+2)
            res.append(self.tree[index])
        return res
    def level_order(self):
        return self.tree
    def find_min(self):
        if not self.tree:
            return None
        return min(self.tree)
    def find_max(self):
        if not self.tree:
            return None
        return max(self.tree)
    def length(self):
        return len(self.tree)
    def sort(self):
        sorted_list = sorted(self.tree)
        print("Sorted tree elements:", sorted_list)
        return sorted_list
abt = ArrayBinaryTree()
abt.insert(50)
abt.insert(30)
abt.insert(70)
abt.insert(20)
```

```
abt.insert(40)
abt.insert(60)
abt.insert(80)
print("Inorder:", abt.inorder())
print("Preorder:", abt.preorder())
print("Postorder:", abt.postorder())
print("Level-order:", abt.level_order())
print("Min:", abt.find_min())
print("Max:", abt.find_max())
print("Length:", abt.length())
abt.delete(30)
print("After deletion, Level-order:", abt.level_order())
abt.sort()
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class LinkedBinaryTree:
    def __init__(self):
        self.root = None
    def insert(self, data):
        new_node = Node(data)
        if not self.root:
            self.root = new_node
            print(f"Inserted {data} as root")
            return
        queue = [self.root]
        while queue:
            current = queue.pop(0)
            if not current.left:
                current.left = new_node
                print(f"Inserted {data} to left of {current.data}")
                return
            else:
```

```
            queue.append(current.left)
        if not current.right:
            current.right = new_node
            print(f"Inserted {data} to right of {current.data}")
            return
        else:
            queue.append(current.right)
def inorder(self, node=None):
    if node is None:
        node = self.root
    res = []
    if node.left:
        res += self.inorder(node.left)
    res.append(node.data)
    if node.right:
        res += self.inorder(node.right)
    return res
def preorder(self, node=None):
    if node is None:
        node = self.root
    res = [node.data]
    if node.left:
        res += self.preorder(node.left)
    if node.right:
        res += self.preorder(node.right)
    return res
def postorder(self, node=None):
    if node is None:
        node = self.root
    res = []
    if node.left:
        res += self.postorder(node.left)
    if node.right:
        res += self.postorder(node.right)
    res.append(node.data)
    return res
```

```
    def search(self, value):
        if not self.root:
            print(f"{value} not found")
            return False
        queue = [self.root]
        while queue:
            node = queue.pop(0)
            if node.data == value:
                print(f"{value} found")
                return True
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        print(f"{value} not found")
        return False
    def find_min(self):
        elements = self.inorder()
        return min(elements) if elements else None
    def find_max(self):
        elements = self.inorder()
        return max(elements) if elements else None
    def length(self):
        return len(self.inorder())
    def sort(self):
        elements = self.inorder()
        sorted_list = sorted(elements)
        print("Sorted tree elements:", sorted_list)
        return sorted_list
print("\n=== Linked List-based Binary Tree ===")
lbt = LinkedBinaryTree()
for val in [50, 30, 70, 20, 40, 60, 80]:
    lbt.insert(val)
print("Inorder:", lbt.inorder())
print("Preorder:", lbt.preorder())
print("Postorder:", lbt.postorder())
```

```
lbt.search(40)
lbt.search(90)
print("Min:", lbt.find_min())
print("Max:", lbt.find_max())
print("Length:", lbt.length())
lbt.sort()
```

1. construct binary tree and implement pre order inorder and post order traversal in python

```
# Node class for binary tree
class Node:
  def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None
class BinaryTree:
  def __init__(self):
    self.root = None
  def insert(self, data):
    new_node = Node(data)
    if not self.root:
      self.root = new_node
      print(f"Inserted {data} as root")
      return
    queue = [self.root]
    while queue:
      current = queue.pop(0)
      if not current.left:
        current.left = new_node
        print(f"Inserted {data} to left of {current.data}")
        return
      else:
        queue.append(current.left)
      if not current.right:
        current.right = new_node
        print(f"Inserted {data} to right of {current.data}")
        return
      else:
```

```
            queue.append(current.right)
    def preorder(self, node):
        if node is None:
            return []
        return [node.data] + self.preorder(node.left) + self.preorder(node.right)
    def inorder(self, node):
        if node is None:
            return []
        return self.inorder(node.left) + [node.data] + self.inorder(node.right)
    def postorder(self, node):
        if node is None:
            return []
        return self.postorder(node.left) + self.postorder(node.right) + [node.data]
bt = BinaryTree()
bt.insert(10)
bt.insert(20)
bt.insert(30)
bt.insert(40)
bt.insert(50)
print("Preorder Traversal:", bt.preorder(bt.root))   # Root -> Left -> Right
print("Inorder Traversal:", bt.inorder(bt.root))     # Left -> Root -> Right
print("Postorder Traversal:", bt.postorder(bt.root)) # Left -> Right -> Root
```

2. construct a binary tree from inorder and postorder or pre order traversal in python

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
def build_tree_pre_in(preorder, inorder):
    if not preorder or not inorder:
        return None
    root_val = preorder[0]
    root = Node(root_val)
    root_index = inorder.index(root_val)
    root.left = build_tree_pre_in(preorder[1:1+root_index], inorder[:root_index])
    root.right = build_tree_pre_in(preorder[1+root_index:], inorder[root_index+1:])
```

```
    return root
def inorder_traversal(node):
    return inorder_traversal(node.left) + [node.data] + inorder_traversal(node.right) if node else []
def preorder_traversal(node):
    return [node.data] + preorder_traversal(node.left) + preorder_traversal(node.right) if node else []
def postorder_traversal(node):
    return postorder_traversal(node.left) + postorder_traversal(node.right) + [node.data] if node else []
preorder = [10, 20, 40, 50, 30]
inorder  = [40, 20, 50, 10, 30]
root = build_tree_pre_in(preorder, inorder)
print("Inorder:", inorder_traversal(root))
print("Preorder:", preorder_traversal(root))
print("Postorder:", postorder_traversal(root))
def build_tree_post_in(postorder, inorder):
    if not postorder or not inorder:
        return None
    root_val = postorder[-1]
    root = Node(root_val)
    root_index = inorder.index(root_val)
    root.left = build_tree_post_in(postorder[:root_index], inorder[:root_index])
    root.right = build_tree_post_in(postorder[root_index:-1], inorder[root_index+1:])
    return root
postorder = [40, 50, 20, 30, 10]
inorder   = [40, 20, 50, 10, 30]
root = build_tree_post_in(postorder, inorder)
print("Inorder:", inorder_traversal(root))
print("Preorder:", preorder_traversal(root))
print("Postorder:", postorder_traversal(root))
```

3. implement binary tee with insert delete and serach for element operations

```
# Node class for binary tree
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
class BinaryTree:
    def __init__(self):
        self.root = None
    def insert(self, data):
        new_node = Node(data)
        if not self.root:
            self.root = new_node
```

```
            print(f"Inserted {data} as root")
            return
        queue = [self.root]
        while queue:
            current = queue.pop(0)
            if not current.left:
                current.left = new_node
                print(f"Inserted {data} to left of {current.data}")
                return
            else:
                queue.append(current.left)
            if not current.right:
                current.right = new_node
                print(f"Inserted {data} to right of {current.data}")
                return
            else:
                queue.append(current.right)
    def find_with_parent(self, data):
        if not self.root:
            return None, None
        queue = [(self.root, None)]
        while queue:
            node, parent = queue.pop(0)
            if node.data == data:
                return node, parent
            if node.left:
                queue.append((node.left, node))
            if node.right:
                queue.append((node.right, node))
        return None, None

    def delete(self, data):

        if not self.root:
            print("Tree is empty")
            return
        node_to_delete, _ = self.find_with_parent(data)
        if not node_to_delete:
            print(f"{data} not found in the tree")
            return
        queue = [self.root]
        deepest = None
        while queue:
            current = queue.pop(0)
            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)
            deepest = current
        node_to_delete.data = deepest.data
        self.delete_deepest(deepest)
        print(f"Deleted {data} from the tree")
    def delete_deepest(self, d_node):
        queue = [self.root]
        while queue:
            current = queue.pop(0)
            if current.left:
                if current.left == d_node:
                    current.left = None
                    return
                else:
                    queue.append(current.left)
```

```
            if current.right:
                if current.right == d_node:
                    current.right = None
                    return
                else:
                    queue.append(current.right)
    def search(self, data):
        if not self.root:
            return False
        queue = [self.root]
        while queue:
            current = queue.pop(0)
            if current.data == data:
                return True
            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)
        return False
    def inorder(self, node=None):
        if node is None:
            node = self.root
        res = []
        if node.left:
            res += self.inorder(node.left)
        res.append(node.data)
        if node.right:
            res += self.inorder(node.right)
        return res
bt = BinaryTree()
bt.insert(10)
bt.insert(20)
bt.insert(30)
bt.insert(40)
bt.insert(50)
print("Inorder traversal:", bt.inorder())
print("Search 30:", bt.search(30))
print("Search 60:", bt.search(60))
bt.delete(20)
print("Inorder traversal after deleting 20:", bt.inorder())
```

4.bst

```
# Node class representing each node in the BST
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


# Binary Search Tree class
class BinarySearchTree:
    def __init__(self):
        self.root = None

    # ---------- INSERT ----------
    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
```

```
        elif key > root.key:
            root.right = self.insert(root.right, key)
        return root

    # ---------- SEARCH ----------
    def search(self, root, key):
        if root is None:
            return False
        if root.key == key:
            return True
        elif key < root.key:
            return self.search(root.left, key)
        else:
            return self.search(root.right, key)

    # ---------- DELETE ----------
    def delete(self, root, key):
        if root is None:
            return root

        # Find the node to be deleted
        if key < root.key:
            root.left = self.delete(root.left, key)
        elif key > root.key:
            root.right = self.delete(root.right, key)
        else:
            # Node found
            # Case 1: Node with no child or one child
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left
            # Find inorder successor (smallest in the right subtree)
            temp = self.minValueNode(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)

        return root

    # ---------- HELPER FUNCTION ----------
    def minValueNode(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current
    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)
bst = BinarySearchTree()
root = None
root = bst.insert(root, 50)
root = bst.insert(root, 30)
root = bst.insert(root, 70)
root = bst.insert(root, 20)
root = bst.insert(root, 40)
root = bst.insert(root, 60)
root = bst.insert(root, 80)

print("Inorder traversal of BST:")
```

```
bst.inorder(root)
print("\n")
key = 40
print(f"Search {key}:", "Found" if bst.search(root, key) else "Not Found")
key = 30
print(f"\nDeleting {key}...")
root = bst.delete(root, key)
print("Inorder traversal after deletion:")
bst.inorder(root)
print()
```

18.implement heap using priority queue and perform operations like insert delete and traverse sorting max and min height in pyhton

```
import heapq
class PriorityQueue:
    def __init__(self, mode="min"):
        """
        Initialize priority queue.
        mode = "min" for Min-Heap (default)
        mode = "max" for Max-Heap
        """
        self.heap = []
        self.mode = mode.lower()
        print(f"Priority Queue created as {self.mode.upper()}-HEAP")

    # ---------- INSERT ELEMENT ----------
    def push(self, data):
        if self.mode == "max":
            heapq.heappush(self.heap, -data)  # store negative for max-heap
        else:
            heapq.heappush(self.heap, data)
        print(f"Inserted: {data}")

    # ---------- REMOVE ELEMENT ----------
    def pop(self):
        if not self.heap:
            print("Priority Queue is empty!")
            return None
        if self.mode == "max":
            val = -heapq.heappop(self.heap)
        else:
            val = heapq.heappop(self.heap)
        print(f"Removed: {val}")
        return val

    # ---------- VIEW TOP ELEMENT ----------
    def peek(self):
        if not self.heap:
            print("Priority Queue is empty!")
            return None
        if self.mode == "max":
            return -self.heap[0]
        else:
            return self.heap[0]

    # ---------- CHECK IF EMPTY ----------
    def isEmpty(self):
        return len(self.heap) == 0

    # ---------- DISPLAY QUEUE ----------
    def display(self):
        if self.mode == "max":
```

```
            print("Priority Queue (Max-Heap):", [-x for x in self.heap])
        else:
            print("Priority Queue (Min-Heap):", self.heap)

# ----- MIN-HEAP DEMO -----
print("\n----- MIN-HEAP DEMO -----")
minPQ = PriorityQueue(mode="min")
minPQ.push(40)
minPQ.push(10)
minPQ.push(30)
minPQ.push(50)
minPQ.push(20)

print("\nCurrent Min-Heap:")
minPQ.display()
print("Peek (smallest):", minPQ.peek())

minPQ.pop()
minPQ.display()
# ----- MAX-HEAP DEMO -----
print("\n----- MAX-HEAP DEMO -----")
maxPQ = PriorityQueue(mode="max")
maxPQ.push(40)
maxPQ.push(10)
maxPQ.push(30)
maxPQ.push(50)
maxPQ.push(20)

print("\nCurrent Max-Heap:")
maxPQ.display()
print("Peek (largest):", maxPQ.peek())

maxPQ.pop()
maxPQ.display()
```