OCR A Level Computer Science

Your notes

8.2 Algorithms for the Main Data Structures

Contents

Your notes

Stacks

Implementing a Stack

How Do You Program a Stack?

- To recap the main operations of Stacks and how they work, follow this link
- When programming a stack, the main operations listed below must be included in your program.
- The main operations for Stacks are:

Main Operations

Operation	Python	Java

isEmpty()	def isEmpty(self):
	return len(self.stack) == 0
		// Check if the stack is empty
		System.out.println(stack.isEmpty());
		// Output: true

push(value)	# Push some items into the stack
	stack.push(1)
		// Push some items into the stack
		stack.push(1);

peek()	def peek(self):
	if not self.isEmpty():
	return self.stack[-1]
	else:
	return None
		// Peek the top item in the stack
		int topItem = stack.peek();
		System.out.println("Top item: " + topItem);
		// Output: Top item: 3

pop()	# Pop items from the stack
	popped_item1 = stack.pop()
		// Pop items from the stack
		int poppedItem1 = stack.pop();

size()	# Get the size of the stack
	size = len(stack)
		// Get the size of the stack
		int size = stack.size();

	print("Size of the stack:", size) # Output: Size of the stack: 3	System.out.println("Size of the stack: " + size); // Output: Size of the stack: 3

isFull()	def isFull(self):
	    return len(self.stack) == self.capacity
		public boolean isFull() {
		    return (top == capacity - 1);
		}

Your notes

Programming a Stack in Python

Explanation of the Python Code

- **Class Stack**: Defines a Python class, Stack, with a specified capacity and an empty list to store elements.
- **Push Method**: Appends an item to the stack if the stack is not full; otherwise, it prints a message to say that it is not possible.
- **Pop Method**: Removes and returns the top item from the stack if it's not empty; otherwise, it prints a message to say nothing can be popped as it is empty.
- **Peek Method**: Returns the top item of the stack without removing it if the stack is not empty; otherwise, it will print a message to say the stack is empty.
- **Additional Operations**: The code at the bottom includes methods to retrieve the stack size (size), check if the stack is empty (isEmpty), and check if the stack is full (isFull).

```
class Stack:

  def __init__(self, capacity):

    self.capacity = capacity

    self.stack = []

  def push(self, item):

    if len(self.stack) < self.capacity:

      self.stack.append(item)

      print("Pushed item:", item)

    else:

      print("Stack is full. Cannot push item:", item)
```

```
    def pop(self):
        if not self.isEmpty():
            return self.stack.pop()
        else:
            print("Stack is empty. Cannot pop item.")
    def peek(self):
        if not self.isEmpty():
            return self.stack[-1]
        else:
            print("Stack is empty. Cannot peek.")
    def size(self):
        return len(self.stack)
    def isEmpty(self):
        return len(self.stack) == 0
    def isFull(self):
        return len(self.stack) == self.capacity
```

- This code below is used to demonstrate the usage of the operations above with a stack of animals, pushing, popping, peeking, and checking the stack's status.

```
# Create a stack with a capacity of 5
stack = Stack(5)
# Push animals onto the stack
stack.push("Dog")
stack.push("Cat")
stack.push("Elephant")
stack.push("Lion")
stack.push("Tiger")
# Try pushing more animals than the capacity allows
```

Your notes

```
stack.push("Giraffe")  # Output: Stack is full. Cannot push item: Giraffe

# Pop an animal from the stack

popped_animal = stack.pop()

print("Popped animal:", popped_animal)  # Output: Popped animal: Tiger

# Peek at the top animal of the stack

top_animal = stack.peek()

print("Top animal:", top_animal)  # Output: Top animal: Lion

# Get the size of the stack

stack_size = stack.size()

print("Stack size:", stack_size)  # Output: Stack size: 4

# Check if the stack is empty

is_empty = stack.isEmpty()

print("Is stack empty?", is_empty)  # Output: Is stack empty? False

# Check if the stack is full

is_full = stack.isFull()

print("Is stack full?", is_full)  # Output: Is stack full? False
```

Programming a Stack in Java

Explanation of the Java Code

- **AnimalStack Class**: Defines a class named AnimalStack that uses the Stack class to implement a stack for storing animal names with a specified capacity.
- **Push Method**: Adds an animal to the stack if it's not full; otherwise, it prints a message to say that it is not possible.
- **Pop Method**: Removes and returns the top animal from the stack if it's not empty; otherwise, it prints a message to say that it is not possible and returns null.
- **Peek Method**: Returns the top animal without removing it if the stack is not empty; otherwise, it prints a message to say that it is not possible and returns null.
- **Main Method Usage**: Demonstrates the usage of the AnimalStack class with a stack of animals, showcasing pushing, popping, peeking, and checking stack size and status.

Your notes

Your notes

```
import java.util.Stack;

public class AnimalStack {

  private int capacity;

  private Stack<String> stack;

  public AnimalStack(int capacity) {

    this.capacity = capacity;

    this.stack = new Stack<>();

  }

  public void push(String animal) {

    if (stack.size() < capacity) {

      stack.push(animal);

      System.out.println("Pushed item: " + animal);

    } else {

      System.out.println("Stack is full. Cannot push item: " + animal);

    }

  }

  public String pop() {

    if (!isEmpty()) {

      return stack.pop();

    } else {

      System.out.println("Stack is empty. Cannot pop item.");

      return null;

    }

  }

  public String peek() {

    if (!isEmpty()) {

      return stack.peek();
```

```
    } else {
        System.out.println("Stack is empty. Cannot peek.");
        return null;
    }
}
public int size() {
    return stack.size();
}
public boolean isEmpty() {
    return stack.isEmpty();
}
public boolean isFull() {
    return stack.size() == capacity;
}
public static void main(String[] args) {
    AnimalStack stack = new AnimalStack(5);
    stack.push("Dog");
    stack.push("Cat");
    stack.push("Elephant");
    stack.push("Lion");
    stack.push("Tiger");
    stack.push("Giraffe");  // Output: Stack is full. Cannot push item: Giraffe
    String poppedAnimal = stack.pop();
    System.out.println("Popped animal: " + poppedAnimal);  // Output: Popped animal: Tiger
    String topAnimal = stack.peek();
    System.out.println("Top animal: " + topAnimal);  // Output: Top animal: Lion
    int stackSize = stack.size();
```

Your notes

```
        System.out.println("Stack size: " + stackSize);  // Output: Stack size: 4
        boolean isEmpty = stack.isEmpty();
        System.out.println("Is stack empty? " + isEmpty);  // Output: Is stack empty? false
        boolean isFull = stack.isFull();
        System.out.println("Is stack full? " + isFull);  // Output: Is stack full? false
    }
}
```

Your notes

Queues

Your notes

Implementing Linear Queues

How Do You Program a Queue?

- To recap the main operations of Queues and how they work, follow this link
- When programming a queue, the main operations listed below must be included in your program
  - Initialise the queue
  - Enqueue
  - Dequeue
  - Peek

Algorithm to Implement a Linear Queue

- The code below initialises an array and index pointers to service a linear queue
- The constant MAX_SIZE informs the size of the array
- The front index pointer is initialised to 0 and the rear index pointer is initialised to -1
- The rear index pointer will be incremented every time you add an item to the queue, so the initial value will ensure the first item is added into the first position of the array (index 0)
- Once the first item has been added, both front and rear will correctly point to the item in the array

Pseudocode	Python	Java

MAX_SIZE = 6
ARRAY queue[max_size]
front = 0
rear = -1
	MAX_SIZE = 6
	queue[]
	front = 0
	rear = -1
		public class Main {
		    public static final int MAX_SIZE = 6;
		    public static ArrayList<Integer> queue = new ArrayList<>();
		    public static int front = 0;
		    public static int rear = -1;
		
		    public static void main(String[] args) {
		        // Your code here

		} }

Your notes

EnQueue Data

- Before adding an item to the queue you need to check that the queue is not full
- When the rear index pointer is pointing to the final space in the array there is no room to add new items
- A subroutine isFull() is used to carry out the check.
- If the end of the array has not been reached, the rear index pointer is incremented and the new item is added to the queue

Pseudocode	Python	Java

FUNCTION isFull(rear)
  IF(rear+1) == MAX_SIZE THEN
    RETURN True
  ELSE
    RETURN False
  ENDIF
ENDFUNCTION
 	
FUNCTION enqueue(queue, rear, data)
  IF isFull(rear) == True THEN
    PRINT ("Full")
  ELSE
    rear = rear + 1
    queue[rear] = data
  ENDIF
  RETURN rear
	def isFull(rear):
	  if rear + 1 == MAX_SIZE:
	    return True
	  else:
	    return False
	
	def enqueue(queue, rear, data):
	  if isFull(rear):
	    print("Full")
	  else:
	    rear = rear + 1
	    queue[rear] = data
	  return rear
	
	# Example usage
	queue = [None] * MAX_SIZE
		public static boolean isFull(int rear) {
		    if (rear + 1 == MAX_SIZE) {
		      return true;
		    } else {
		      return false;
		    }
		  }
		
		  public static int enqueue(int[] queue, int rear, int data) {
		    if (isFull(rear)) {
		      System.out.println("Full");
		    } else {
		      rear = rear + 1;
		      queue[rear] = data;
		    }
		    return rear;

Your notes

ENDFUNCTION	rear = -1 rear = enqueue(queue, rear, 10) rear = enqueue(queue, rear, 20) rear = enqueue(queue, rear, 30) print(queue) # [10, 20, 30]	}

DeQueue Data

- Before taking an item from the queue you need to make sure that the queue is not empty
- A subroutine isEmpty() can be defined for this purpose
- The subroutine will check whether the front index pointer is ahead of the rear index pointer
- If the queue is not empty the item at the front of the queue is returned and front is incremented by 1

Pseudocode	Python	Java

FUNCTION isEmpty(front, rear)
 	
  IF front > rear THEN
 	
    RETURN True
 	
  ELSE
 	
    RETURN False
 	
  ENDIF
 	
ENDFUNCTION
 	
FUNCTION dequeue(queue, rear, front)
 	
  IF isEmpty(front, rear) THEN
	def isEmpty(front, rear):
	
	  if front > rear:
	
	    return True
	
	  else:
	
	    return False
	
	def dequeue(queue, rear, front):
	
	  if isEmpty(front, rear):
	
	    print("Empty")
	
	    dequeuedItem = None
	
	  else:
	
	    dequeuedItem = queue[front]
		import java.util.Arrays;
		
		public class Main {
		
		  public static boolean isEmpty(int front, int rear) {
		
		    if (front > rear) {
		
		      return true;
		
		    } else {
		
		      return false;
		
		    }
		
		  }

```
    PRINT ("Empty")
    dequeuedItem = Null
  ELSE
    dequeuedItem = queue[front]
    front = front + 1
  ENDIF
  RETURN (dequeuedItem, front)
ENDFUNCTION
```

```
    front = front + 1
  return dequeuedItem, front

# Example usage
MAX_SIZE = 100
queue = [10, 20, 30, 40, 50]
front = 0
rear = len(queue) - 1

dequeuedItem, front = dequeue(queue, rear, front)
print("Dequeued item:", dequeuedItem)
print("Front:", front)

dequeuedItem, front = dequeue(queue, rear, front)
print("Dequeued item:", dequeuedItem)
print("Front:", front)
```

```
  public static Object[] dequeue(int[] queue, int rear, int front) {
    if (isEmpty(front, rear)) {
      System.out.println("Empty");
      Object[] result = { null, front };
      return result;
    } else {
      Object[] result = { queue[front], front + 1 };
      return result;
    }
  }

  public static void main(String[] args) {
    int MAX_SIZE = 100;
    int[] queue = { 10, 20, 30, 40, 50 };
    int front = 0;
    int rear = queue.length - 1;

    Object[] dequeuedResult = dequeue(queue, rear, front);
    Object dequeuedItem = dequeuedResult[0];
    front = (int) dequeuedResult[1];
    System.out.println("Dequeued item: " + dequeuedItem);
    System.out.println("Front: " + front);

    dequeuedResult = dequeue(queue, rear, front);
```

Your notes

		dequeuedItem = dequeuedResult[0];
		front = (int) dequeuedResult[1];
		System.out.println("Dequeued item: " + dequeuedItem);
		System.out.println("Front: " + front);
		}
		}

Your notes

Implementing Circular Queues

Algorithm to Implement a Circular Queue

- When you manipulate the items in a circular queue you must be able to advance the index pointers in such a way that they will reset to 0 once the end has been reached

```
(pointer + 1) MOD MAX_SIZE
```

- This can be used to calculate the new position for the index pointer
- Where the pointer is front or rear and max_size is the maximum number of items in the queue.

Let's imagine there is an array with 7 items.

Initial Position	**Pointer + 1**	**(Pointer + 1) MOD 7**	**New Position**
0	1	1	1
1	2	2	2
2	3	3	3
3	4	4	4
4	5	5	5
5	6	6	6
6	7	0	0

Your notes

- When the index pointer references the last position in the array, position 6, the next position is calculated as (6+1) MOD 7 which gives the result 0
- This allows the index pointer to move back to the reference at the start of the array

Initialise Circular Queue

- The initialisation is the same as a linear queue apart from you initialise the front index to -1 as this will allow you to be certain that you have an empty queue

Pseudocode	Python	Java

MAX_SIZE = 4
 	
ARRAY cQueue[max_size]
 	
front = -1
 	
rear = -1
	MAX_SIZE = 4
	
	cQueue[]
	
	front = -1
	
	rear = -1
		public class CircularQueue {
		
		private static final int MAX_SIZE = 4;
		
		private int[] cQueue = new int[MAX_SIZE];
		
		private int front = -1;
		
		private int rear = -1;
		
		public static void main(String[] args) {
		
		CircularQueue circularQueue = new CircularQueue();
		
		// Use the circularQueue object to perform operations on the circular queue
		
		}
		
		}

Enqueue Circular Queue

- Before an item can be enqueued the array needs to be checked to see if it's full
- It will be full if the next position to be used is already occupied by the item at the front of the queue
- If the queue is not full then the rear index pointer must be adjusted to reference the next free position so that the new item can be added

Pseudocode	Python	Java
MAX_SIZE = 4	MAX_SIZE = 4	public class CircularQueue {

```
ARRAY cQueue[max_size]

front = -1

rear = -1


FUNCTION is_full(front, rear)

   IF (rear + 1) MOD MAX_SIZE == front THEN

    RETURN True

   ELSE

      RETURN False

   ENDIF

ENDFUNCTION



FUNCTION enqueue(queue, front, rear, data)

   IF is_full(front, rear) == True THEN

      PRINT("Queue is full")

   ELSE

      rear = (rear + 1) MOD MAX_SIZE

    queue[rear] = data

      IF front = -1 THEN // First item to be queued

         front = 0

      ENDIF

   ENDIF

   RETURN (front, rear)
```

```
cQueue = [None] * MAX_SIZE

front = -1

rear = -1


def is_full(front, rear):

   if (rear + 1) % MAX_SIZE == front:

      return True

   else:

      return False


def enqueue(queue, front, rear, data):

   if is_full(front, rear):

      print("Queue is full")

   else:

      rear = (rear + 1) % MAX_SIZE

      queue[rear] = data

      if front == -1:  # First item to be queued

         front = 0

   return front, rear
```

```
   private static final int MAX_SIZE = 4;

   private int[] cQueue = new int[MAX_SIZE];

   private int front = -1;

   private int rear = -1;


   public static void main(String[] args) {

      CircularQueue circularQueue = new CircularQueue();

      // Use the circularQueue object to perform operations on the circular queue

   }

}
```

Your notes

ENDFUNCTION		

Your notes

DeQueue Circular Queue

- Before dequeuing the array needs to be checked if it's empty. If the queue is not empty then the item at the front is dequeued
- If this is the only item that was in the queue, the rear and front pointers are reset
- Otherwise the pointer moves to reference the next item in the queue

Pseudocode	Python	Java

MAX_SIZE = 4
ARRAY cQueue[max_size]
front = -1
rear = -1
 	
FUNCTION is_empty(rear)
  IF front == -1 THEN
   RETURN True
  ELSE
    RETURN False
  ENDIF
ENDFUNCTION
 	
FUNCTION dequeue(queue, front, rear)
  IF is_empty(rear) == True THEN
PRINT("Queue is empty - nothing to dequeue")
	def is_empty(front):
	  if front == -1:
	    return True
	  else:
	    return False
	
	def dequeue(queue, front, rear):
	  if is_empty(front):
	    print("Queue is empty - nothing to dequeue")
	    dequeued_item = None
	  else:
	    dequeued_item = queue[front]
	    if front == rear:
	      front = -1
	      rear = -1
	    else:
		public class CircularQueue {
		  private static final int MAX_SIZE = 4;
		  private Object[] cQueue = new Object[MAX_SIZE];
		  private int front = -1;
		  private int rear = -1;
		
		  public static void main(String[] args) {
		    CircularQueue circularQueue = new CircularQueue();
		    // Use the circularQueue object to perform operations on the circular queue
		  }
		
		  private boolean isEmpty(int front) {
		    if (front == -1) {
		      return true;
		    } else {
		      return false;
		    }
		  }

dequeued_item = Null
ELSE
dequeued_item = queue[front]
// Check if the queue is empty
IF front == rear THEN
front = -1
rear = -1
ELSE
front = (front + 1) MOD maxsize
ENDIF
ENDIF
RETURN (dequeued_item, front, rear)
ENDFUNCTION
	front = (front + 1) % MAX_SIZE
	return dequeued_item, front, rear
		private void dequeue(Object[] queue, int front, int rear) {
		if (isEmpty(front)) {
		System.out.println("Queue is empty - nothing to dequeue");
		Object dequeuedItem = null;
		} else {
		Object dequeuedItem = queue[front];
		if (front == rear) {
		front = -1;
		rear = -1;
		} else {
		front = (front + 1) % MAX_SIZE;
		}
		}
		}
		}

Your notes

Implementing Priority Queues

Algorithm to Implement a Priority Queue

- A priority queue is where each item in the queue has a priority
- When new items are added, they are inserted ahead of those with a lower priority and behind items of equal priority
- If you want to implement a priority queue it is best to use OOP
- Encapsulating the data and methods within class definitions provides a neat solution and once written, the class can be reused in other applications that require a queue

Your notes

Pseudocode

```
FUNCTION enqueue(queue, element, priority)
  // Insert the element into the queue based on its priority
  // Higher priority elements are placed towards the front

  IF queue is empty OR priority is higher than the first element's priority
    Insert element at the front of the queue
  ELSE IF priority is lower than or equal to the last element's priority
    Insert element at the end of the queue
  ELSE
    Find the appropriate position in the queue to insert the element based on priority
    Shift the elements to the right to make space for the new element
    Insert the element at the determined position
  ENDIF
ENDFUNCTION


FUNCTION dequeue(queue)
  // Remove and return the element with the highest priority from the queue

  IF queue is empty
    Return an error indicating that the queue is empty
  ELSE
    Remove and return the element from the front of the queue
  ENDIF
ENDFUNCTION
```

Your notes

```
FUNCTION peek(queue)
    // Return the element with the highest priority without removing it from the queue

    IF queue is empty
        Return an error indicating that the queue is empty
    ELSE
        Return the element from the front of the queue
    ENDIF
ENDFUNCTION


FUNCTION is_empty(queue)
    // Check if the queue is empty

    IF queue is empty
        Return True
    ELSE
        Return False
    ENDIF
ENDFUNCTION
```

Python
class PriorityQueue: def __init__(self):

Your notes

```
    self.queue = []

def enqueue(self, element, priority):
    # Insert the element into the queue based on its priority
    # Higher priority elements are placed towards the front

    if len(self.queue) == 0 or priority > self.queue[0][1]:
        self.queue.insert(0, (element, priority))
    elif priority <= self.queue[-1][1]:
        self.queue.append((element, priority))
    else:
        index = 0
        while index < len(self.queue) and priority <= self.queue[index][1]:
            index += 1
        self.queue.insert(index, (element, priority))

def dequeue(self):
    # Remove and return the element with the highest priority from the queue

    if self.is_empty():
        raise IndexError("Queue is empty - nothing to dequeue")
    else:
        return self.queue.pop(0)[0]

def peek(self):
    # Return the element with the highest priority without removing it from the queue
```

Your notes

```
    if self.is_empty():
      raise IndexError("Queue is empty - nothing to peek")
    else:
      return self.queue[0][0]

  def is_empty(self):
    # Check if the queue is empty

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

Java

```
import java.util.ArrayList;
import java.util.List;

class PriorityQueue {
  private List<Element> queue;

  public PriorityQueue() {
    this.queue = new ArrayList<>();
  }

  public void enqueue(Object element, int priority) {
    // Insert the element into the queue based on its priority
    // Higher priority elements are placed towards the front

    if (queue.isEmpty() || priority > queue.get(0).getPriority()) {
```

```
            queue.add(0, new Element(element, priority));
        } else if (priority <= queue.get(queue.size() - 1).getPriority()) {
            queue.add(new Element(element, priority));
        } else {
            int index = 0;
            while (index < queue.size() && priority <= queue.get(index).getPriority()) {
                index++;
            }
            queue.add(index, new Element(element, priority));
        }
    }

    public Object dequeue() {
        // Remove and return the element with the highest priority from the queue

        if (isEmpty()) {
            throw new IndexOutOfBoundsException("Queue is empty - nothing to dequeue");
        } else {
            return queue.remove(0).getElement();
        }
    }

    public Object peek() {
        // Return the element with the highest priority without removing it from the queue

        if (isEmpty()) {
            throw new IndexOutOfBoundsException("Queue is empty - nothing to peek");
```

Your notes

```
} else {
    return
```

Your notes

Linked Lists

Your notes

Implementing a Linked List

How do you Program a Linked List?

- To recap the main operations of Linked Lists and how they work, follow this link
- To program a Linked List you must be able to perform the following operations:
  - Create a Linked List
  - Traverse a Linked List
  - Add data to a Linked List
  - Remove data from a Linked List

Create a Linked List

- Define the class called 'Node' that represents a node in the linked list.
- Each node has 2 fields 'fruit' (this stores the fruit value) and 'next' (this stores a reference to the next node in the list)
- We create 3 nodes and assign fruit values to each node
- To establish a connection between the nodes we set the 'next' field of each node to the appropriate next node

Pseudocode

```
class Node:
   field fruit
   field next

# Create nodes
node1 = Node()
node2 = Node()
node3 = Node()

# Assign fruit values
node1.fruit = "apple"
node2.fruit = "banana"
node3.fruit = "orange"
```

```
# Connect nodes
node1.next = node2
node2.next = node3
node3.next = None
```

Python

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

# Create nodes
node1 = Node("apple")
node2 = Node("banana")
node3 = Node("orange")

# Connect nodes
node1.next = node2
node2.next = node3
```

Java

```
class Node {
    String fruit;
    Node next;

    Node(String fruit) {
        this.fruit = fruit;
        this.next = null;
    }
}

public class LinkedListExample {
    public static void main(String[] args) {
        // Create nodes
        Node node1 = new Node("apple");
        Node node2 = new Node("banana");
        Node node3 = new Node("orange");

        // Connect nodes
        node1.next = node2;
        node2.next = node3
```

Your notes

```
  }
}
```

Your notes

Algorithm to Traverse a Linked List

- We traverse the linked list starting from 'node1'. We output the fruit value of each node and update the 'current' reference to the next node until we reach the end of the list

Pseudocode

```
# Traverse the linked list
current = node1
while current is not None:
   print(current.fruit)
   current = current.next
```

Python

```
# Traverse the linked list
current = node1
while current is not None:
   print(current.fruit)
   current = current.next
```

Java

```
// Traverse the linked list
current = node1;
while (current != null) {
System.out.println(current.fruit);
     current = current.next;
```

Your notes

Algorithm to Add Data to a Linked List

- We create a new node with the new data and check if the list is empty
- If it isn't, we find the last node and add the new node to the end

Pseudocode

```
# Add "orange" to the end of the linked list
new_node = Node()
new_node.fruit = "orange"
new_node.next = None

if node1 is None:
   # If the list was empty, make "orange" the head
   node1 = new_node
else:
   # Find the last node and append "orange" to it
   current = node1
   while current.next is not None:
      current = current.next
   current.next = new_node
```

Python

```python
# Add "orange" to the end of the linked list
new_node = Node("orange")

if node1 is None:
   # If the list was empty, make "orange" the head
   node1 = new_node
else:
   # Find the last node and append "orange" to it
   current = node1
   while current.next is not None:
      current = current.next
   current.next = new_node
```

Java

```java
// Add "orange" to the end of the linked list
Node new_node = new Node("orange");
```

Your notes

```
if (node1 == null) {
// If the list was empty, make "orange" the head
node1 = new_node;
} else {
// Find the last node and append "orange" to it
    current = node1;
    while (current.next != null) {
    current = current.next;
    }
    current.next = new_node;

}
```

Algorithm to Remove Data from a Linked List

- We traverse the list until we find the data to be removed
- If the data to be removed is the first node, we update the head, else we update the node before to bypass the data to be removed

Pseudocode

```
# Find and remove "apple"

while current is not None:

  if current.fruit == "apple":

    if previous is None:

      # If "apple" is the first node, update the head

      node1 = current.next

    else:

      # Remove the node by updating the previous node's next pointer

      previous.next = current.next

    break

  previous = current

  current = current.next
```

Your notes

Python

```
# Find and remove "apple"
while current is not None:
    if current.fruit == "apple":
        if previous is None:
            # If "apple" is the first node, update the head
            node1 = current.next
        else:
            # Remove the node by updating the previous node's next pointer
            previous.next = current.next
        break
    previous = current
    current = current.next
```

Java

```
// Find and remove "apple"
 while (current != null) {
  if (current.fruit.equals("apple")) {
if (previous == null) {
            // If "apple" is the first node, update the head
            node1 = current.next;
           } else {
            // Remove the node by updating the previous node's next pointer
            previous.next = current.next;
            }
            break;
```

```
        }
previous = current;
    current = current.next;
  }
```

Your notes

Trees

Your notes

Implementing a Tree

How Do You Program a Tree?

In the following example:

- The tree data structure is represented by a 'TreeNode' struct which has a 'value' field to store the value of the node and a 'children' field to hold a collection of child nodes
- The 'createTreeNode' function is used to create a new tree node and initialise its value and an empty children collection
- The 'addChild' function is used to add a child to a parent node. It creates a new child node with the specified value and appends it to the 'children' collection of the parent node
- Then the 'createTreeNode' creates the root node and adds children to it using the 'addChild' function. This allows the build up of the tree structure with multiple levels and branches

Pseudocode

```
// Define the Tree Node structure
struct TreeNode {
   value // The value stored in the node
   children // A collection of child nodes
}


// Create a function to create a new tree node
function createTreeNode(value):
   node = new TreeNode()
   node.value = value
   node.children = []
   return node
```

```
// Create the root node of the tree
root = createTreeNode(rootValue)

// Add children to a node
function addChild(parentNode, childValue):
    childNode = createTreeNode(childValue)
    parentNode.children.append(childNode)

// Usage example:
addChild(root, childValue1)
addChild(root, childValue2)
addChild(root, childValue3)

// Add a child to an existing child node
addChild(root.children[0], grandchildValue)

// ...and so on
```

Python

```
# Define the Tree Node class
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

# Create a function to add a child to a node
def add_child(parent_node, child_value):
```

Your notes

Your notes

```
    child_node = TreeNode(child_value)

    parent_node.children.append(child_node)


# Create the root node of the tree

root_value = "Root"  # Replace with the desired value

root = TreeNode(root_value)


# Add children to the root node

child_value1 = "Child 1"  # Replace with the desired value

child_value2 = "Child 2"  # Replace with the desired value

child_value3 = "Child 3"  # Replace with the desired value


add_child(root, child_value1)

add_child(root, child_value2)

add_child(root, child_value3)


# Add a child to an existing child node

grandchild_value = "Grandchild"  # Replace with the desired value

add_child(root.children[0], grandchild_value)


# Usage example:

print(root.value)  # Output: "Root"

print(root.children[0].value)  # Output: "Child 1"

print(root.children[0].children[0].value)  # Output: "Grandchild"
```

Java

```
import java.util.ArrayList;

import java.util.List;


// Define the Tree Node class

class TreeNode {

   String value;

   List<TreeNode> children;


   TreeNode(String value) {

      this.value = value;

      this.children = new ArrayList<>();

   }

}


public class TreeExample {

   public static void main(String[] args) {

      // Create the root node of the tree

      String rootValue = "Root";

      TreeNode root = new TreeNode(rootValue);


      // Add children to the root node

      String childValue1 = "Child 1";

      String childValue2 = "Child 2";

      String childValue3 = "Child 3";


      addChild(root, childValue1);

      addChild(root, childValue2);
```

Your notes

Your notes

```
        addChild(root, childValue3);


        // Add a child to an existing child node

        String grandchildValue = "Grandchild";

        addChild(root.children.get(0), grandchildValue);


        // Usage example:

        System.out.println(root.value);

          // Output: "Root"

        System.out.println(root.children.get(0).value);
        // Output: "Child 1"

        System.out.println(root.children.get(0).children.get(0).value);

        // Output: "Grandchild"

    }


    // Create a method to add a child to a node

    public static void addChild(TreeNode parentNode, String childValue) {

        TreeNode childNode = new TreeNode(childValue);

        parentNode.children.add(childNode);

    }

}
```

Algorithm to Traverse a Tree

Post Order Traversal

- Post-Order traversal follows the order:

1. Left Subtree
2. Right Subtree

3. Root Node

- Using the outline method, nodes are traversed in the order in which you pass them on the right
- You can recap the theory notes on Tree Data Structures here.
- **Note**: The algorithm below shows that this is identical to the other methods of traversing a tree (pre-order and in-order) except for the position of the output statement. You do not require the other two methods for OCR
- *Node* is an object with 3 attributes; *node.data* contains a value, *node.left* contains a pointer to the left child node and *node.right* contains a pointer to the right child node
- As mentioned above, the algorithm will traverse the left subtree, then the right subtree then the root

Your notes

Pseudocode

```
PROCEDURE post_order_traversal(node)
    // Check any nodes to the left of the current node
    IF node.left != Null THEN
        post_order_traversal(node.left)
    ENDIF

    // Check any nodes to the right of the current node
    IF node.right != Null THEN
        post_order_traversal(node.right)
    ENDIF

    // Output the data of the current node
    PRINT(node.data)
ENDPROCEDURE
```

Python

```
def post_order_traversal(node):
```

```
    # Check any nodes to the left of the current node
    if node.left is not None:
        post_order_traversal(node.left)

    # Check any nodes to the right of the current node
    if node.right is not None:
        post_order_traversal(node.right)

    # Output the data of the current node
```

Java

```
class Node {
    int data;
    Node left;
    Node right;
}

class Tree {
    Node root;
}

public class PostOrderTraversal {
    public static void postOrderTraversal(Node node) {
        if (node != null) {
            // Check any nodes to the left of the current node
            if (node.left != null) {
                postOrderTraversal(node.left);
```

Your notes

Your notes

```
        }

        // Check any nodes to the right of the current node
        if (node.right != null) {
            postOrderTraversal(node.right);
        }

        // Output the data of the current node
        System.out.println(node.data);
    }
}

public static void main(String[] args) {
    // Create a sample tree
    Tree tree = new Tree();
    tree.root = new Node();
    tree.root.data = 1;
    tree.root.left = new Node();
    tree.root.left.data = 2;
    tree.root.right = new Node();
    tree.root.right.data = 3;

    // Perform post-order traversal on the tree
    postOrderTraversal(tree.root);
}
}
```

Algorithm to Add Data to a Tree

- In your exam, the exam board quite often uses binary trees in their questions. The algorithm below will use a binary tree for that reason.
- The code defines a binary tree and a function to add a new node to the tree.
- The binary tree is represented using a class called 'Node', where each node has a data value, a left child, and a right child.
- The add_node function takes two parameters: the root of the binary tree and the value of the new node to be added.
  - If the tree is empty (root is 'None'), it creates a new node with the given value and sets it as the root.
  - Otherwise, it performs a traversal using a queue to find the first available position (either the left or right child of a node) to insert the new node.

Pseudocode

```
NODE:
 data: INTEGER
 left: NODE
 right: NODE

FUNCTION add_node(tree, value):
 new_node = NODE()
 new_node.data = value

 IF tree IS NULL:
  tree = new_node
 ELSE:
  current = tree
  WHILE True:
   IF value < current.data:
```

Your notes

Your notes

```
      IF current.left IS NULL:
        current.left = new_node
        BREAK
      ELSE:
        current = current.left
    ELSE:
      IF current.right IS NULL:
        current.right = new_node
        BREAK
      ELSE:
        current = current.right
ENDFUNCTION
```

Python

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

def add_node(root, value):
    if root is None:
        root = Node(value)
    else:
        queue = []
        queue.append(root)
```

```
    while len(queue) > 0:
        current = queue.pop(0)

        if current.left is None:
            current.left = Node(value)
            break
        else:
            queue.append(current.left)

        if current.right is None:
            current.right = Node(value)
            break
        else:
            queue.append(current.right)

    return root

# Example usage
root = Node(1)
root.left = Node(2)
root.right = Node(3)

print("Before adding node:")
# Perform pre-order traversal to display the tree before adding the node
def pre_order_traversal(node):
    if node is not None:
        print(node.data, end=" ")
```

Your notes

```
        pre_order_traversal(node.left)
        pre_order_traversal(node.right)


pre_order_traversal(root)
print()


root = add_node(root, 4)


print("After adding node:")
pre_order_traversal(root)
```

Java

```
import java.util.LinkedList;
import java.util.Queue;


class Node {
    int data;
    Node left;
    Node right;


    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

Your notes

Your notes

```
class BinaryTree {
  Node root;

  public void addNode(int value) {
    if (root == null) {
      root = new Node(value);
    } else {
      Queue<Node> queue = new LinkedList<>();
      queue.offer(root);

      while (!queue.isEmpty()) {
        Node current = queue.poll();

        if (current.left == null) {
          current.left = new Node(value);
          break;
        } else {
          queue.offer(current.left);
        }

        if (current.right == null) {
          current.right = new Node(value);
          break;
        } else {
          queue.offer(current.right);
        }
```

```
        }
    }
}

// Example usage
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);

    System.out.println("Before adding node:");
    // Perform pre-order traversal to display the tree before adding the node
    preOrderTraversal(tree.root);
    System.out.println();

    tree.addNode(4);

    System.out.println("After adding node:");
    preOrderTraversal(tree.root);
}

// Pre-order traversal method to display the tree
public static void preOrderTraversal(Node node) {
    if (node != null) {
        System.out.print(node.data + " ");
        preOrderTraversal(node.left);
```

Your notes

```
      preOrderTraversal(node.right);
    }
  }
}
```

Your notes

Algorithm to Remove Data from a Tree

- Once again, the algorithm below will use a binary tree
- The find_node function is a function that searches for a node with a given value in the tree.
- The remove_node function removes a node with a specified value from the tree, handling three cases:
  - Node has no children (Leaf Node):
    - Remove the node from the tree by setting it to 'None'.
  - Node has one child:
    - Replace the node with its only child.
  - Node has two children:
    - Find the minimum value node in the right subtree (or the maximum value node in the left subtree).
    - Replace the node's data with the data of the replacement node.
    - Recursively remove the replacement node.
- The find_minimum function is a function that helps to find the node with the minimum value in a given subtree.

Pseudocode

```
NODE:
  data: INTEGER
  left: NODE
  right: NODE

FUNCTION find_node(tree, item):
```

```
 IF tree IS NULL:
  RETURN NULL
 ELSE IF tree.data == item:
  RETURN tree
 ELSE IF item < tree.data:
  RETURN find_node(tree.left, item)
 ELSE:
  RETURN find_node(tree.right, item)
 ENDIF

FUNCTION remove_node(tree, item):
 node = find_node(tree, item)

 IF node IS NULL:
  PRINT("Item not found in the tree")
 ELSE:
  IF node.left IS NULL AND node.right IS NULL:
   // Case 1: Node has no children
   // Simply remove the node from the tree
   DELETE node
  ELSE IF node.left IS NULL OR node.right IS NULL:
   // Case 2: Node has only one child
   // Replace the node with its child
   IF node.left IS NOT NULL:
    child = node.left
   ELSE:
    child = node.right
```

Your notes

Your notes

```
      ENDIF

      UPDATE_PARENT(node, child)

      DELETE node

    ELSE:

      // Case 3: Node has two children

      // Find the replacement node (either minimum value in right subtree or maximum value in left subtree)

      replacement = find_minimum(node.right)

      node.data = replacement.data

      remove_node(node.right, replacement.data)

    ENDIF

  ENDIF


ENDFUNCTION
```

Python

```
class Node:

    def __init__(self, data):

        self.data = data

        self.left = None

        self.right = None


def find_node(tree, item):

    if tree is None:

        return None

    elif tree.data == item:

        return tree
```

Your notes

```
    elif item < tree.data:
        return find_node(tree.left, item)
    else:
        return find_node(tree.right, item)


def remove_node(tree, item):
    node = find_node(tree, item)


    if node is None:
        print("Item not found in the tree")
    else:
        if node.left is None and node.right is None:
            # Case 1: Node has no children
            # Simply remove the node from the tree
            node = None
        elif node.left is None or node.right is None:
            # Case 2: Node has only one child
            # Replace the node with its child
            if node.left is not None:
                child = node.left
            else:
                child = node.right
            node.data = child.data
            node.left = child.left
            node.right = child.right
        else:
            # Case 3: Node has two children
```

Your notes

```
        # Find the replacement node (minimum value in right subtree)
        replacement = find_minimum(node.right)
        node.data = replacement.data
        remove_node(node.right, replacement.data)

def find_minimum(node):
    while node.left is not None:
        node = node.left
    return node

# Example usage
tree = Node(4)
tree.left = Node(2)
tree.right = Node(6)
tree.left.left = Node(1)
tree.left.right = Node(3)
tree.right.left = Node(5)
tree.right.right = Node(7)

remove_node(tree, 2)
```

Java

```
class Node {
    int data;
    Node left;
    Node right;
```

```
    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

class BinaryTree {
    Node root;

    public Node findNode(Node tree, int item) {
        if (tree == null) {
            return null;
        } else if (tree.data == item) {
            return tree;
        } else if (item < tree.data) {
            return findNode(tree.left, item);
        } else {
            return findNode(tree.right, item);
        }
    }

    public void removeNode(Node tree, int item) {
        Node node = findNode(tree, item);

        if (node == null) {
            System.out.println("Item not found in the tree");
```

Your notes

```
    } else {
        if (node.left == null && node.right == null) {
            // Case 1: Node has no children
            // Simply remove the node from the tree
            node = null;
        } else if (node.left == null || node.right == null) {
            // Case 2: Node has only one child
            // Replace the node with its child
            Node child = (node.left != null) ? node.left : node.right;
            node.data = child.data;
            node.left = child.left;
            node.right = child.right;
        } else {
            // Case 3: Node has two children
            // Find the replacement node (minimum value in right subtree)
            Node replacement = findMinimum(node.right);
            node.data = replacement.data;
            removeNode(node.right, replacement.data);
        }
    }
}

public Node findMinimum(Node node) {
    while (node.left != null) {
        node = node.left;
    }
    return node;
```

Your notes

```
    }


    // Example usage
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(6);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);
        tree.root.right.left = new Node(5);
        tree.root.right.right = new Node(7);


        tree.removeNode(tree.root, 2);
    }
}
```

Your notes