a) **How to measure the performance of an algorithm?** The performance of an algorithm can be measured using various factors such as time complexity (how long it takes to run), space complexity (how much memory it uses), and the Big O notation (describes the upper bound of an algorithm's time or space complexity).

b) **What is a polynomial? How is it different from a structure?** A polynomial is a mathematical expression consisting of variables and coefficients, involving addition, subtraction, multiplication, and non-negative integer exponents. In programming, a structure is a user-defined data type that can hold different types of data under one name.

c) **What is a balance factor? How is it calculated?** The balance factor in the context of AVL trees is the difference in height between the left and right subtrees of a node. It's calculated by subtracting the height of the right subtree from the height of the left subtree.

d) **What are Abstract Data Types (ADTs)?** ADTs are data types defined by their behavior (operations they can perform) rather than their implementation. They provide a logical description of data and operations without specifying how they're implemented.

e) **What is an ancestor of a node?** In a tree data structure, an ancestor of a node is any node that lies on the path from the root to that particular node.

f) **State the types of graphs.** Graphs can be classified into various types: directed and undirected graphs, weighted and unweighted graphs, cyclic and acyclic graphs, etc.

g) **Differentiate between an array and a structure.** An array is a data structure that stores elements of the same data type in contiguous memory locations, accessed by indices. A structure is a composite data type that can hold elements of different data types under one name.

h) **What is space and time complexity?** Space complexity refers to the amount of memory space required by an algorithm to execute concerning the input size. Time complexity measures the amount of time an algorithm takes to run based on the input size.

i) **What is a pointer to a pointer?** A pointer to a pointer, also known as a double pointer, is a pointer that holds the memory address of another pointer variable. It's often used for dynamically allocating memory and for implementing data structures like linked lists and trees.

j) **What is a spanning tree?** A spanning tree of a connected, undirected graph is a subgraph that is a tree (no cycles) and includes all the vertices of the original graph. It spans all the vertices with the minimum possible number of edges.

a) **How to measure the performance of an algorithm?**

The performance of an algorithm can be assessed through several metrics:

- **Time Complexity:** This measures the amount of time an algorithm takes to complete concerning the input size. It's often expressed using Big O notation (e.g., O(n), O(n^2)) to describe the upper bound of the algorithm's runtime.
- **Space Complexity:** It evaluates the amount of memory space an algorithm uses concerning the input size. Similar to time complexity, it's expressed using Big O notation.
- **Comparative Studies:** Algorithms can be compared by analyzing their performance on different inputs or scenarios, focusing on factors like execution speed and resource consumption.

b) **What is a polynomial? How is it different from a structure?**

- **Polynomial:** In mathematics, a polynomial is an expression consisting of variables and coefficients, combined using operations like addition, subtraction, multiplication, and non-negative integer exponents. For instance, $2x3+5x2−3x+72x^3 + 5x^2 - 3x + 72x3+5x2−3x+7$ is a polynomial.
- **Structure:** In programming, a structure is a user-defined data type that allows bundling different data types together. It's a way to group variables under one name for easier management and manipulation. Unlike a polynomial, which is a mathematical expression, a structure is used in programming to organize and manage data.

c) **What is a balance factor? How is it calculated?**

- **Balance Factor:** In the context of AVL (Adelson-Velsky and Landis) trees, the balance factor of a node is the difference between the heights of its left and right subtrees. It determines whether the tree is balanced or needs balancing operations like rotations.
- **Calculation:** The balance factor of a node NNN is calculated as the difference between the height of its left subtree LLL and the height of its right subtree RRR: Balance Factor(N) = Height(L) - Height(R).

d) **What are Abstract Data Types (ADTs)?**

- **Abstract Data Types:** ADTs are data types defined by their behavior (operations they can perform) rather than their implementation details. They provide a logical description of data and operations without specifying how they are internally implemented. Examples include stacks, queues, lists, and trees.

e) **What is an ancestor of a node?**

- **Ancestor of a Node:** In a tree data structure, an ancestor of a node is any node lying on the path from the root to that particular node. For instance, in a family tree, parents, grandparents, and beyond are ancestors of a specific person.

f) **State the types of graphs.**

- **Types of Graphs:** Graphs can be categorized based on various properties:
    - **Directed and Undirected Graphs:** Edges in directed graphs have directions, while edges in undirected graphs don't.
    - **Weighted and Unweighted Graphs:** Weighted graphs assign values (weights) to edges, while unweighted graphs don't.
    - **Cyclic and Acyclic Graphs:** Cyclic graphs contain cycles (loops), while acyclic graphs don't have cycles.

g) **Differentiate between an array and a structure.**

- **Array:** An array is a data structure that stores elements of the same data type in contiguous memory locations. Elements in an array are accessed using indices.
- **Structure:** A structure is a composite data type that allows storing different data types under a single name. It doesn't necessarily store elements in contiguous memory locations and can hold various types of data.

h) **What is space and time complexity?**

- **Space Complexity:** It refers to the amount of memory space an algorithm requires concerning the input size. It's measured by the maximum amount of memory space used and is often expressed using Big O notation.
- **Time Complexity:** It measures the amount of time an algorithm takes to run concerning the input size. Time complexity is typically expressed using Big O notation and describes the algorithm's behavior as the input size grows.

i) **What is a pointer to a pointer?**

- **Pointer to Pointer:** Also known as a double pointer, it's a pointer that stores the memory address of another pointer variable. In C or C++, it's used for dynamic memory allocation or for creating data structures like linked lists of pointers.

j) **What is a spanning tree?**

- **Spanning Tree:** In a connected, undirected graph, a spanning tree is a subgraph that is a tree (no cycles) and includes all the vertices of the original graph. It spans all the vertices with the minimum possible number of edges necessary to connect them all without forming cycles. It's often used in network design and optimization.

a) **Explain Insertion sort technique with an example.**

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It works by taking one element from the unsorted part of the array and inserting it into its correct position in the sorted part of the array.

Example: Consider an array: [5, 2, 4, 6, 1, 3]

- Start with the second element (index 1) as the key: 2.
- Compare the key with the elements to its left.
- Since 5 > 2, move 5 to the right and insert 2 before 5: [2, 5, 4, 6, 1, 3]
- Move to the next unsorted element (index 2) - key: 4.
- Compare 4 with its adjacent elements to the left.
- Shift elements larger than 4 to the right and insert 4 at the correct position: [2, 4, 5, 6, 1, 3]
- Repeat this process until the entire array is sorted.

b) **What is a circular queue? How is it different from a static queue?**

- **Circular Queue:** A circular queue is a data structure that behaves like a queue but the last element is connected to the first element. This makes use of the entire array by avoiding wastage of space.
- **Difference from a Static Queue:**
    - In a static queue, once the queue becomes full and some elements are dequeued, those empty spaces cannot be used again. In a circular queue, space is reused by connecting the last element back to the first, making it circular.

c) **What is a stack? What are the various applications of a stack? List operations performed on a stack.**

- **Stack:** A stack is an abstract data type that follows the Last In, First Out (LIFO) principle. It has two primary operations: push (adds an element to the top) and pop (removes the top element).
- **Applications of Stack:**
    - Expression evaluation and syntax parsing
    - Function call and recursion
    - Undo mechanisms in software
    - Backtracking algorithms
- **Operations on Stack:**
    - **Push:** Adds an element to the top of the stack.
    - **Pop:** Removes and returns the top element from the stack.
    - **Peek or Top:** Retrieves the top element without removing it.
    - **isEmpty:** Checks if the stack is empty.
    - **isFull:** Checks if the stack is full (applicable in some implementations).

d) **Explain different types of AVL rotations with an example.**

- **AVL Rotations:** AVL rotations are operations used to balance an AVL tree when it becomes unbalanced after an insertion or deletion.
- **Types of Rotations:**
    - **Left Rotation:** Rebalances the tree by rotating nodes to the left.
    - **Right Rotation:** Rebalances the tree by rotating nodes to the right.
    - **Double Left Rotation (Left-Right Rotation):** Combination of left and right rotations.
    - **Double Right Rotation (Right-Left Rotation):** Combination of right and left rotations.
- **Example:** Suppose a tree becomes unbalanced after insertion. Depending on the specific imbalance, rotations (single or double) are performed to restore balance by adjusting node relationships.

e) **Explain various types of Dynamic Memory Allocation functions.**

- **malloc():** Allocates a specified number of bytes of memory.
- **calloc():** Allocates memory for an array of elements and initializes them to zero.
- **realloc():** Resizes the previously allocated memory block.
- **free():** Deallocates the memory previously allocated by malloc, calloc, or realloc functions, making it available for further use.

a) **Function to create and display a doubly linked list:**

cCopy code

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int main() {
    struct Node* head = createNode(1);
    struct Node* second = createNode(2);
    struct Node* third = createNode(3);

    head->next = second;
    second->prev = head;
    second->next = third;
    third->prev = second;

    printf("Doubly Linked List: ");
    displayList(head);

    return 0;
}
```

b) **Recursive functions for tree traversal (inorder, preorder, postorder):**

cCopy code

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void inorder(struct Node* root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

void preorder(struct Node* root) {
    if (root == NULL)
        return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

void postorder(struct Node* root) {
    if (root == NULL)
        return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}

int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
```

```
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Inorder traversal: ");
    inorder(root);
    printf("\nPreorder traversal: ");
    preorder(root);
    printf("\nPostorder traversal: ");
    postorder(root);

    return 0;
}
```

c) **Function to delete the first node from a singly linked list:**

cCopy code

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

void deleteFirstNode(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty, nothing to delete.\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int main() {
    struct Node* head = NULL;

    // Assume list has some elements
    // ... (code to create and add elements to the list)

    printf("Original List: ");
    displayList(head);

    deleteFirstNode(&head);
    printf("\nList after deleting first node: ");
    displayList(head);

    return 0;
}
```

d) **Function to reverse a string using a stack:**

cCopy code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 100

struct Stack {
    int top;
    char items[MAX_SIZE];
};

void push(struct Stack* stack, char c) {
    if (stack->top == MAX_SIZE - 1) {
        printf("Stack Overflow!\n");
        return;
    }
    stack->items[++(stack->top)] = c;
}

char pop(struct Stack* stack) {
    if (stack->top == -1) {
        printf("Stack Underflow!\n");
        return -1;
    }
    return stack->items[(stack->top)--];
}

void reverseString(char* str) {
    struct Stack stack;
    stack.top = -1;

    int len = strlen(str);
    for (int i = 0; i < len; i++) {
        push(&stack, str[i]);
    }

    for (int i = 0; i < len; i++) {
        str[i] = pop(&stack);
    }
}

int main() {
    char str[] = "Hello, World!";
    printf("Original String: %s\n", str);
```

```
    reverseString(str);
    printf("Reversed String: %s\n", str);

    return 0;
}
```

e) **C Program for evaluation of a polynomial:**

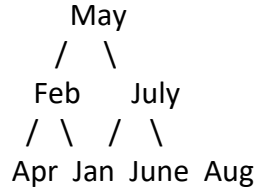
```
cCopy code
#include <stdio.h>

int evaluatePolynomial(int coefficients[], int degree, int x) {
    int result = 0;
    for (int i = 0; i <= degree; i++) {
        result += coefficients[i] * pow(x, i);
    }
    return result;
}

int main() {
    int coefficients[] = {1, 2, 3}; // Example coefficients for x^2 + 2x + 3
    int degree = 2; // Degree of the polynomial
    int x = 5; // Value of x

    int result = evaluatePolynomial(coefficients, degree, x);
    printf("Result of the polynomial for x = %d is: %d\n", x, result);

    return 0;
}
```

a) **Construct an AVL tree for the given sequential data: Jan, Feb, Apr, May, July, Aug, June.**

To construct an AVL tree, the data needs to be ordered in a manner that adheres to the rules of an AVL tree, considering alphabetical order for the months.

Here's the constructed AVL tree based on the given sequential data:

markdownCopy code



This tree maintains the AVL property (balance factor of nodes is <= 1), ensuring it's a balanced binary search tree.

b) **Use merge sort technique on the given data: 45, 85, 96, 78, 34, 12, 49, 38, 18.**

Merge sort is a divide-and-conquer sorting algorithm. Here's the sorted result using merge sort for the provided data:

Copy code
12, 18, 34, 38, 45, 49, 78, 85, 96

c) **Write a 'C' program to create a linked list in which the data part of each node contains individual digits of the numbers.**

Here's a C program that creates a linked list where each node contains individual digits of the numbers:

cCopy code

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int digit) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = digit;
    newNode->next = NULL;
    return newNode;
}

void insertNumber(struct Node** head, int number) {
```

```
    while (number > 0) {
        int digit = number % 10;
        struct Node* newNode = createNode(digit);
        newNode->next = *head;
        *head = newNode;
        number /= 10;
    }
}

void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int main() {
    struct Node* head = NULL;
    int number = 9876; // Example number

    insertNumber(&head, number);

    printf("Linked List with individual digits: ");
    displayList(head);

    return 0;
}
```

d) **What is a circular queue? Explain it with an example.**

A circular queue is a data structure that operates as a queue, but the last element is connected back to the first element, forming a circle. It overcomes the limitations of a regular queue by reusing the empty spaces created after dequeue operations.

An example of a circular queue:

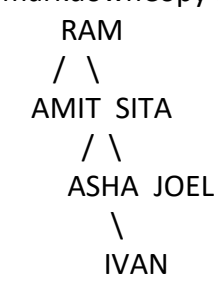
rustCopy code
Front -> [2] -> [3] -> [5] -> [1] -> [ ] -> Rear

In a circular queue, if the queue is full and some elements are dequeued, the new elements can be enqueued into the previously emptied spaces, maintaining a circular structure.

e) **Construct a Binary Search Tree (BST) with the given data: RAM, SITA, AMIT, JOEL, IVAN, ASHA.**

The construction of a BST depends on the ordering of the data. Assuming alphabetical order:

markdownCopy code



This BST satisfies the property of a binary search tree, where the left child is less than the parent, and the right child is greater than the parent.

i) **Directed Graph:**

- A directed graph, also known as a digraph, is a type of graph in which edges have a direction or are represented by arrows indicating the relationship between nodes. Each edge in a directed graph has a starting node (tail) and an ending node (head), indicating a one-way relationship from the tail node to the head node. The edges have a defined direction, and traversal or movement between nodes follows the direction specified by the edges.

ii) **Strict Binary Tree:**

- A strict binary tree is a specific type of binary tree where each node in the tree has exactly zero or two children (never one child). It means every node either has no child nodes (is a leaf) or has exactly two child nodes (left and right child). In a strict binary tree, the absence of a child is represented as a null pointer.

iii) **Cyclic Graph:**

- A cyclic graph is a graph that contains at least one cycle. A cycle in a graph is a path that starts and ends at the same node by traversing through edges, without revisiting any node (except the starting node). In simpler terms, it's a closed path within the graph that forms a loop by visiting a sequence of edges and nodes, ultimately returning to the starting node. A graph is acyclic if it does not contain any cycles; otherwise, it is cyclic.

*******Convert the following expression into postfix i) A/B $ CD * E - A *C ii) (A + B * C -D)/ E $ F

To convert infix expressions to postfix (also known as Reverse Polish Notation), you can use the concept of a stack. Here's how to convert the given expressions:

i) AB$CD∗E−A∗C\frac{A}{B} \$ CD * E - A * CBA$CD∗E−A∗C

Steps:

1. Initialize an empty stack and an empty output string.
2. Start scanning the expression from left to right.
3. If the scanned character is an operand (A, B, C, etc.), add it to the output.
4. If the scanned character is an operator, check its precedence:
    - If the precedence of the current operator is higher than the precedence of the operator at the top of the stack, push it onto the stack.
    - If the precedence of the current operator is lower or equal to the precedence of the operator at the top of the stack, pop operators from the stack and add them to the output until the precedence condition is satisfied, then push the current operator onto the stack.

5. After completing the scanning of the entire expression, pop any remaining operators from the stack and add them to the output.

Let's convert the expression:

lessCopy code
Given infix expression: A/B $ CD * E - A * C

Converted postfix expression: AB/CD$E*A-C*

ii) (A+B∗C−D)E$F\frac{(A + B * C - D)}{E} \$ FE(A+B∗C−D)$F

mathematicaCopy code
Given infix expression: (A + B * C - D)/E $ F

Converted postfix expression: ABC*+D-E/F$

These postfix expressions can be evaluated using a stack-based algorithm or interpreted directly by some programming languages that support postfix notation.