Explain :What is data structure and explain its types

A data structure is a fundamental concept in computer science and programming that refers to a way of organizing and storing data in a computer's memory or storage, such that it can be efficiently accessed, manipulated, and processed. Data structures are essential for solving various computational problems and optimizing the performance of algorithms.

There are several types of data structures, each designed for specific purposes and use cases. Here are some of the most common types:

1. \*\*Arrays\*\*: An array is a collection of elements, each identified by an index or a key. Elements are stored in contiguous memory locations, making it easy to access elements by their index. However, arrays have a fixed size, and inserting or deleting elements can be inefficient.

2. \*\*Linked Lists\*\*: Linked lists consist of nodes, where each node contains data and a reference (or link) to the next node in the list. Linked lists can be singly linked (each node points to the next) or doubly linked (each node points to both the next and the previous). Linked lists are more flexible than arrays in terms of dynamic resizing but may have slower access times.

3. \*\*Stacks\*\*: A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It allows operations like push (add an element to the top) and pop (remove the top element). Stacks are often used for tasks that involve tracking function calls, managing undo/redo functionality, and parsing expressions.

4. \*\*Queues\*\*: A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It supports operations like enqueue (add an element to the rear) and dequeue (remove an element from the front). Queues are used in scenarios where tasks or data must be processed in a specific order, such as task scheduling and breadth-first search algorithms.

5. \*\*Trees\*\*: Trees are hierarchical data structures consisting of nodes, where each node can have zero or more child nodes. A tree structure is widely used for representing hierarchical relationships in data, such as directory structures, organization charts, and XML/HTML parsing. Common types of trees include binary trees, binary search trees, and balanced trees like AVL trees and Red-Black trees.

6. \*\*Graphs\*\*: Graphs are non-linear data structures composed of nodes (vertices) and edges that connect these nodes. They are used to represent complex relationships between objects and entities. Graphs can be directed (edges have a direction) or undirected (edges have no direction). They are crucial in solving problems like route planning, social network analysis, and dependency analysis.

7. \*\*Hash Tables\*\*: A hash table is a data structure that stores key-value pairs. It uses a hash function to map keys to specific locations in an array (called a bucket), making retrieval of values very efficient. Hash tables are commonly used for implementing dictionaries, caches, and databases.

8. \*\*Heaps\*\*: A heap is a specialized tree-based data structure used to maintain a priority queue. Heaps come in two main varieties: max heap (where the parent node is greater than or equal to its child nodes) and min heap (where the parent node is less than or equal to its child nodes). Heaps are valuable in algorithms like heap sort and in finding the maximum or minimum element efficiently.

These are just some of the fundamental data structures in computer science. Choosing the right data structure depends on the specific problem you're trying to solve and the requirements for data access and manipulation efficiency. Understanding data structures is essential for designing efficient algorithms and optimizing the performance of your software applications.

Explain Linear and Non Linear Data Structure…..

Linear and non-linear data structures are two broad categories used to classify data structures based on their organization and relationships between elements. These categories help us understand how data is stored and accessed within a data structure. Here's an explanation of both:

1. \*\*Linear Data Structures\*\*:

Linear data structures are those in which data elements are organized sequentially, one after another, with a clear and linear order. Accessing elements in a linear data structure typically involves traversing the elements one by one, from the beginning to the end or vice versa.

Common examples of linear data structures include:

- \*\*Arrays\*\*: Elements are stored in contiguous memory locations, and they can be accessed using indices.

- \*\*Linked Lists\*\*: Elements are connected in a linear sequence using pointers or references.

- \*\*Stacks\*\*: Elements are arranged in a linear order with a Last-In-First-Out (LIFO) access pattern.

- \*\*Queues\*\*: Elements are organized linearly with a First-In-First-Out (FIFO) access pattern.

Linear data structures are often used when the order of elements matters, and when you need to process data in a specific sequence.

2. \*\*Non-Linear Data Structures\*\*:

Non-linear data structures do not organize elements in a linear sequence. Instead, they allow for more complex relationships and hierarchies among elements. Accessing elements in non-linear data structures usually involves traversing a complex network or structure, following specific paths or connections.

Common examples of non-linear data structures include:

- \*\*Trees\*\*: Elements are organized hierarchically, with a single root node and child nodes branching out from the root. Trees can have various forms, such as binary trees, balanced trees, and more complex tree structures like B-trees or AVL trees.

- \*\*Graphs\*\*: Elements are connected through edges and vertices. Graphs can be directed (edges have a direction) or undirected (edges have no direction). Graphs allow for modeling complex relationships and can have various types, including trees, sparse graphs, and dense graphs.

- \*\*Hash Tables\*\*: While hash tables store data in an array-like structure, they allow for non-linear access to elements based on their hash values. Elements are stored in buckets, and the relationship between keys and buckets is not linear.

Non-linear data structures are used when data relationships are more complex than a simple linear sequence. They are valuable for modeling real-world scenarios where entities interact in intricate ways or when you need to represent hierarchical relationships.

In summary, linear data structures organize elements in a straightforward, sequential manner, making it easy to access them linearly. Non-linear data structures, on the other hand, provide more complex relationships and hierarchies among elements, allowing for modeling of more intricate data scenarios. The choice between linear and non-linear data structures depends on the nature of the data and the specific requirements of the problem you're trying to solve.

What are the advantages of linked list over array

Linked lists and arrays are both data structures with their own advantages and disadvantages, and the choice between them depends on the specific requirements of your application. Here are some advantages of linked lists over arrays:

1. \*\*Dynamic Size\*\*: Linked lists can easily grow or shrink in size during runtime. Unlike arrays, which have a fixed size, linked lists can allocate memory for new elements as needed. This dynamic sizing makes linked lists more flexible when dealing with data of unknown or varying sizes.

2. \*\*Constant-Time Insertions/Deletions\*\*: In a singly or doubly linked list, inserting or deleting an element at the beginning or in the middle of the list can be done in constant time (O(1)), as long as you have a reference to the node where the operation needs to take place. In contrast, inserting or deleting elements in an array often requires shifting elements, which can be O(n), where n is the number of elements in the array.

3. \*\*No Wasted Memory\*\*: Linked lists use memory efficiently because they allocate memory for each element individually. In arrays, you may need to preallocate a larger block of memory than necessary to accommodate potential future elements, which can lead to memory wastage.

4. \*\*Ease of Insertion/Deletion in Sorted Lists\*\*: Linked lists make it easier to insert or delete elements in a sorted list while maintaining the sorted order. In an array, maintaining a sorted order usually requires shifting elements, which can be inefficient.

5. \*\*No Need for Reallocation\*\*: In a dynamic array, if you reach its capacity, you need to allocate a new, larger array and copy the elements from the old array to the new one. Linked lists avoid this overhead because you can simply allocate memory for a new node and update references to insert a new element.

6. \*\*No Overhead for Unused Space\*\*: Arrays may have unused space if they are allocated with a size greater than the number of elements they currently hold. Linked lists only use memory for the elements they contain, avoiding this overhead.

7. \*\*Efficient for Sequential Insertions/Deletions\*\*: When you need to repeatedly insert or delete elements at both ends of the list (e.g., implementing a queue or a stack), linked lists can be more efficient than arrays, especially if the list is frequently modified.

It's important to note that while linked lists have these advantages, they also have some disadvantages compared to arrays. For example, linked lists typically have higher memory overhead due to the storage of pointers or references for each element, and they may not provide efficient random access to elements (unlike arrays, which offer constant-time random access). Therefore, the choice between linked lists and arrays should be based on the specific requirements and trade-offs of your application.

Write a c program to implement queue using array with its function with explaination of code

Certainly! Below is a simple C program that implements a queue using an array. The program includes functions for enqueue (inserting elements at the rear of the queue), dequeue (removing elements from the front of the queue), checking if the queue is empty, and displaying the elements in the queue. I'll provide explanations for each part of the code.

```c

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 10

int queue[MAX\_SIZE];

int front = -1;

int rear = -1;

bool isFull() {

return rear == MAX\_SIZE - 1;

}

bool isEmpty() {

return front == -1;

}

void enqueue(int item) {

if (isFull()) {

printf("Queue is full. Cannot enqueue.\n");

return;

}

if (isEmpty()) {

front = rear = 0;

} else {

rear++;

}

queue[rear] = item;

printf("%d enqueued to the queue.\n", item);

}

int dequeue() {

int item;

if (isEmpty()) {

printf("Queue is empty. Cannot dequeue.\n");

return -1; // Return a sentinel value to indicate an error

}

item = queue[front];

if (front == rear) {

front = rear = -1; // Reset the queue when the last element is dequeued

} else {

front++;

}

return item;

}

void display() {

if (isEmpty()) {

printf("Queue is empty.\n");

return;

}

printf("Queue elements: ");

for (int i = front; i <= rear; i++) {

printf("%d ", queue[i]);

}

printf("\n");

}

int main() {

enqueue(10);

enqueue(20);

enqueue(30);

display();

int item = dequeue();

if (item != -1) {

printf("%d dequeued from the queue.\n", item);

}

display();

return 0;

}

```

Explanation of the code:

1. We start by defining the maximum size of the queue using `#define MAX\_SIZE 10`.

2. We declare the `queue` array to store queue elements and initialize `front` and `rear` to -1 to indicate an empty queue.

3. `isFull()` and `isEmpty()` functions are defined to check if the queue is full or empty, respectively.

4. The `enqueue` function inserts an element at the rear of the queue. It checks if the queue is full and updates the `front` and `rear` pointers accordingly.

5. The `dequeue` function removes an element from the front of the queue and returns it. It checks if the queue is empty and updates the `front` and `rear` pointers.

6. The `display` function shows the elements in the queue.

7. In the `main` function, we enqueue elements (10, 20, 30), display the queue, dequeue an element (if possible), and then display the updated queue.

Compile and run the program to see the queue operations in action. This is a basic example, and in a real-world application, you might want to add more error handling and error-checking to ensure the queue operations are safe and robust.

Write a c program to implement stack using array with its function with explaination of code

Sure! Here's a C program that implements a stack using an array. The program includes functions for pushing elements onto the stack, popping elements from the stack, checking if the stack is empty, and displaying the elements in the stack. I'll provide explanations for each part of the code.

```c

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 10

int stack[MAX\_SIZE];

int top = -1; // Initialize top of the stack to -1

bool isFull() {

return top == MAX\_SIZE - 1;

}

bool isEmpty() {

return top == -1;

}

void push(int item) {

if (isFull()) {

printf("Stack overflow. Cannot push.\n");

return;

}

stack[++top] = item;

printf("%d pushed onto the stack.\n", item);

}

int pop() {

int item;

if (isEmpty()) {

printf("Stack is empty. Cannot pop.\n");

return -1; // Return a sentinel value to indicate an error

}

item = stack[top--];

return item;

}

void display() {

if (isEmpty()) {

printf("Stack is empty.\n");

return;

}

printf("Stack elements: ");

for (int i = 0; i <= top; i++) {

printf("%d ", stack[i]);

}

printf("\n");

}

int main() {

push(10);

push(20);

push(30);

display();

int item = pop();

if (item != -1) {

printf("%d popped from the stack.\n", item);

}

display();

return 0;

}

```

Explanation of the code:

1. We define the maximum size of the stack using `#define MAX\_SIZE 10`.

2. We declare the `stack` array to store stack elements and initialize `top` to -1 to indicate an empty stack.

3. `isFull()` and `isEmpty()` functions are defined to check if the stack is full or empty, respectively.

4. The `push` function adds an element to the top of the stack. It checks if the stack is full before pushing.

5. The `pop` function removes and returns the element from the top of the stack. It checks if the stack is empty before popping.

6. The `display` function shows the elements in the stack.

7. In the `main` function, we push elements (10, 20, 30) onto the stack, display the stack, pop an element (if possible), and then display the updated stack.

Compile and run the program to see the stack operations in action. This is a basic example, and in a real-world application, you might want to add more error handling and error-checking to ensure the stack operations are safe and robust.

Write a short note on Big O notation with example

\*\*Big O notation\*\*, often denoted as O(f(n)), is a mathematical notation used in computer science and mathematics to analyze and describe the performance or complexity of algorithms. It provides an upper bound on the growth rate of the algorithm's time complexity (or space complexity) as a function of the input size, expressed in terms of the worst-case scenario. In simpler terms, it describes how the runtime or resource usage of an algorithm scales as the input size increases.

Here's an explanation of Big O notation with an example:

\*\*Example\*\*:

Consider a simple algorithm that finds the maximum element in an unsorted array of integers.

```c

int findMax(int arr[], int n) {

int max = arr[0];

for (int i = 1; i < n; i++) {

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

```

In this algorithm:

- We initialize `max` with the first element of the array.

- We then iterate through the array, comparing each element with the current maximum (`max`) and updating `max` if we find a larger element.

Now, let's analyze the time complexity of this algorithm using Big O notation:

- The loop runs `n-1` times (once for each element except the first one).

- Inside the loop, we have a constant-time operation (comparing and updating `max`).

Therefore, the time complexity of this algorithm is \*\*O(n)\*\*, where "n" is the size of the input array. This means that the runtime of the algorithm grows linearly with the size of the input. If you double the size of the input array, the algorithm will take roughly twice as long to execute.

Common Big O Notations and Their Meanings:

1. \*\*O(1)\*\*: Constant time complexity. The algorithm's runtime is not dependent on the input size; it performs in a fixed amount of time regardless of the input.

2. \*\*O(log n)\*\*: Logarithmic time complexity. The runtime grows slowly as the input size increases.

3. \*\*O(n)\*\*: Linear time complexity. The runtime grows linearly with the input size.

4. \*\*O(n log n)\*\*: Log-linear time complexity. Common in efficient sorting algorithms like merge sort and quicksort.

5. \*\*O(n^2)\*\*: Quadratic time complexity. The runtime grows quadratically with the input size.

6. \*\*O(2^n)\*\*: Exponential time complexity. The runtime grows very rapidly with the input size and is often considered inefficient.

7. \*\*O(n!)\*\*: Factorial time complexity. The most inefficient scenario, where the runtime grows extremely fast with the input size.

Big O notation is essential for analyzing algorithms, comparing their efficiency, and making informed decisions about which algorithm to choose for a specific problem based on the expected input size. It helps developers and computer scientists reason about the scalability and performance of algorithms without getting bogged down in implementation details.

Write an algorithm to convert inflix expression to postfix using stack

To convert an infix expression to postfix notation using a stack in C, you can use the Shunting Yard algorithm. Here's a step-by-step algorithm along with C code to achieve this:

\*\*Algorithm\*\*:

1. Initialize an empty stack for operators and an empty string (or list) for the postfix expression.

2. Scan the infix expression from left to right.

3. For each character in the input:

- If it is an operand (a digit or variable), add it to the postfix expression.

- If it is an open parenthesis '(', push it onto the operator stack.

- If it is a closing parenthesis ')', pop operators from the stack and append them to the postfix expression until an open parenthesis is encountered. Pop and discard the open parenthesis.

- If it is an operator (+, -, \*, /, etc.), pop operators from the stack and append them to the postfix expression until an operator with lower precedence is encountered or the stack is empty. Then push the current operator onto the stack.

4. After processing all characters in the input, pop any remaining operators from the stack and append them to the postfix expression.

5. The postfix expression is now the desired output.

\*\*C Code\*\*:

```c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

// Define a structure for the stack

struct Stack {

int top;

unsigned capacity;

char\* array;

};

// Function to create a new stack

struct Stack\* createStack(unsigned capacity) {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->capacity = capacity;

stack->top = -1;

stack->array = (char\*)malloc(stack->capacity \* sizeof(char));

return stack;

}

// Function to check if the stack is empty

bool isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element onto the stack

void push(struct Stack\* stack, char item) {

stack->array[++stack->top] = item;

}

// Function to pop an element from the stack

char pop(struct Stack\* stack) {

if (!isEmpty(stack)) {

return stack->array[stack->top--];

}

return '$'; // '$' is used to indicate an empty stack

}

// Function to check the precedence of an operator

int precedence(char op) {

if (op == '+' || op == '-')

return 1;

if (op == '\*' || op == '/')

return 2;

return 0;

}

// Function to convert infix expression to postfix

void infixToPostfix(char\* infix) {

int len = strlen(infix);

struct Stack\* stack = createStack(len);

char postfix[len]; // Output postfix expression

int outputIndex = 0;

for (int i = 0; i < len; i++) {

char currentChar = infix[i];

if (isalnum(currentChar)) {

postfix[outputIndex++] = currentChar;

} else if (currentChar == '(') {

push(stack, currentChar);

} else if (currentChar == ')') {

while (!isEmpty(stack) && stack->array[stack->top] != '(') {

postfix[outputIndex++] = pop(stack);

}

if (!isEmpty(stack) && stack->array[stack->top] == '(') {

pop(stack);

}

} else {

while (!isEmpty(stack) && precedence(currentChar) <= precedence(stack->array[stack->top])) {

postfix[outputIndex++] = pop(stack);

}

push(stack, currentChar);

}

}

// Pop any remaining operators from the stack

while (!isEmpty(stack)) {

postfix[outputIndex++] = pop(stack);

}

// Null-terminate the postfix expression

postfix[outputIndex] = '\0';

printf("Infix Expression: %s\n", infix);

printf("Postfix Expression: %s\n", postfix);

}

int main() {

char infix[] = "A + B \* C - (D / E)";

infixToPostfix(infix);

return 0;

}

```

This C code demonstrates how to convert an infix expression to postfix using the Shunting Yard algorithm. The code defines a stack to hold operators and processes each character in the input infix expression. The result is a postfix expression that maintains the correct order of operations.

Describe Round Robin algorithm for queue

The Round Robin (RR) scheduling algorithm is a widely used time-sharing scheduling algorithm in operating systems and computer systems. It is designed for scheduling tasks in a way that each task is given a fixed time slice or quantum to execute before being moved to the end of the queue to allow other tasks to run. Round Robin scheduling is a preemptive scheduling algorithm, meaning that the currently running task is interrupted when its time quantum expires, and the CPU is given to the next task in the queue.

Here's a description of the Round Robin scheduling algorithm:

1. \*\*Initialization\*\*: Each process in the ready queue is assigned a fixed time quantum, which is typically a small, constant time slice (e.g., 10 milliseconds).

2. \*\*Execution\*\*: The scheduler starts with the first process in the ready queue and allows it to run for its allotted time quantum. If the process completes its execution before the time quantum expires, it voluntarily relinquishes the CPU. If the time quantum expires before the process finishes, it is preempted (interrupted) and moved to the end of the ready queue.

3. \*\*Queue Management\*\*: After a process completes its time quantum or is preempted, the scheduler selects the next process in the queue and gives it the CPU. This process continues until all processes in the ready queue have had a chance to execute. The scheduler keeps cycling through the queue until no processes are left.

4. \*\*Advantages\*\*:

- Round Robin scheduling ensures fairness among processes since each process gets an equal share of CPU time.

- It is suitable for time-sharing systems where multiple users or processes need to execute concurrently.

- Shorter tasks do not have to wait for longer tasks to complete since tasks are preempted regularly.

5. \*\*Disadvantages\*\*:

- The time quantum needs to be chosen carefully. If it's too short, there may be a high context-switching overhead. If it's too long, the system may not be responsive.

- Round Robin may not be the most efficient scheduling algorithm for processes with varying execution times, as shorter tasks still have to wait for the time quantum to expire even if they are done.

6. \*\*Example\*\*:

- Suppose you have three processes, A, B, and C, with time quantum of 10 milliseconds each.

- The scheduler starts with process A and allows it to run for 10 ms.

- After 10 ms, process A is moved to the end of the queue, and process B gets the CPU.

- The cycle continues until all processes complete their execution.

Round Robin scheduling is a simple and fair scheduling algorithm that ensures that all processes get a chance to run. It is commonly used in scenarios where responsiveness and fairness are essential, such as in time-sharing systems, but it may not be the most efficient choice for all types of workloads.

Differentiate between stack and queue

Stacks and queues are both abstract data types that represent collections of elements with specific rules for adding and removing elements. However, they have distinct characteristics and are used for different purposes. Here's a differentiation between stacks and queues:

\*\*1. Data Structure:\*\*

- \*\*Stack\*\*: A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, meaning that the last element added is the first one to be removed.

- \*\*Queue\*\*: A queue is also a linear data structure, but it follows the First-In-First-Out (FIFO) principle, where the first element added is the first one to be removed.

\*\*2. Order of Operations:\*\*

- \*\*Stack\*\*: In a stack, elements are added and removed from one end, often referred to as the "top" of the stack.

- \*\*Queue\*\*: In a queue, elements are added at one end (rear) and removed from the other end (front) of the queue.

\*\*3. Use Cases:\*\*

- \*\*Stack\*\*: Stacks are used in scenarios where the order of processing or retrieval is essential, such as managing function calls in a program (call stack), undo/redo functionality in applications, and solving problems like expression evaluation.

- \*\*Queue\*\*: Queues are used when tasks or data must be processed in a specific order, like scheduling tasks in an operating system, handling print jobs in a printer queue, and implementing breadth-first search algorithms.

\*\*4. Operations:\*\*

- \*\*Stack\*\*: The primary operations on a stack are "push" (to add an element to the top of the stack) and "pop" (to remove the top element). There's also a "peek" operation to view the top element without removing it.

- \*\*Queue\*\*: The primary operations on a queue are "enqueue" (to add an element to the rear of the queue) and "dequeue" (to remove an element from the front of the queue). Similar to stacks, you can also have a "peek" operation to view the front element.

\*\*5. Real-world Analogies:\*\*

- \*\*Stack\*\*: Think of a stack as a stack of plates. You can only add or remove plates from the top of the stack.

- \*\*Queue\*\*: Think of a queue as a line of people waiting for a service. The first person to join the line is the first one to be served.

\*\*6. Data Structures:\*\*

- \*\*Stack\*\*: Stacks can be implemented using arrays or linked lists.

- \*\*Queue\*\*: Queues can also be implemented using arrays or linked lists. There are variations like priority queues and double-ended queues (deque) that have additional features.

\*\*7. Example:\*\*

- \*\*Stack\*\*: To implement a back button in a web browser, where you can go back to the previous web page (LIFO order).

- \*\*Queue\*\*: To manage print jobs in a printer queue, where print jobs are processed in the order they are received (FIFO order).

In summary, stacks and queues are fundamental data structures with distinct behaviors. Stacks are used for managing data in a last-in-first-out (LIFO) manner, while queues are used for managing data in a first-in-first-out (FIFO) manner. The choice between them depends on the specific requirements of the problem you are trying to solve.

Numericals on PostFix,Prefix and Inflix

Sure, here are the examples presented in a question-and-answer format:

\*\*Question 1\*\*: Convert the infix expression (3 + 4) \* 5 to postfix notation and evaluate it.

\*\*Answer 1\*\*:

- \*\*Postfix Notation\*\*: 3 4 + 5 \*

- \*\*Evaluation\*\*:

- Push 3 onto the stack: Stack: [3]

- Push 4 onto the stack: Stack: [3, 4]

- Encountering '+', pop 4 and 3, calculate 3 + 4 = 7, and push the result (7) onto the stack: Stack: [7]

- Push 5 onto the stack: Stack: [7, 5]

- Encountering '\*', pop 5 and 7, calculate 7 \* 5 = 35, and push the result (35) onto the stack: Stack: [35]

- \*\*Result\*\*: The final result is 35.

\*\*Question 2\*\*: Convert the prefix expression \* + 3 4 5 to infix notation and evaluate it.

\*\*Answer 2\*\*:

- \*\*Infix Notation\*\*: (3 + 4) \* 5

- \*\*Evaluation\*\*:

- Start from right to left.

- Encountering 5, push it onto the stack: Stack: [5]

- Encountering 4, push it onto the stack: Stack: [5, 4]

- Encountering '+', pop 4 and 5, calculate 4 + 5 = 9, and push the result (9) onto the stack: Stack: [9]

- Encountering 3, push it onto the stack: Stack: [9, 3]

- Encountering '\*', pop 3 and 9, calculate 3 \* 9 = 27, and push the result (27) onto the stack: Stack: [27]

- \*\*Result\*\*: The final result is 27.

\*\*Question 3\*\*: Convert the postfix expression 3 4 + 5 \* to infix notation and evaluate it.

\*\*Answer 3\*\*:

- \*\*Infix Notation\*\*: (3 + 4) \* 5

- \*\*Evaluation\*\*:

- Start from left to right.

- Encountering 3, push it onto the stack: Stack: [3]

- Encountering 4, push it onto the stack: Stack: [3, 4]

- Encountering '+', pop 4 and 3, calculate 3 + 4 = 7, and push the result (7) onto the stack: Stack: [7]

- Encountering 5, push it onto the stack: Stack: [7, 5]

- Encountering '\*', pop 5 and 7, calculate 7 \* 5 = 35, and push the result (35) onto the stack: Stack: [35]

- \*\*Result\*\*: The final result is 35.

\*\*Question 4\*\*: Convert the infix expression (8 - 3) / (2 + 4) to postfix notation and evaluate it.

\*\*Answer 4\*\*:

- \*\*Postfix Notation\*\*: 8 3 - 2 4 + /

- \*\*Evaluation\*\*:

- Push 8 onto the stack: Stack: [8]

- Push 3 onto the stack: Stack: [8, 3]

- Encountering '-', pop 3 and 8, calculate 8 - 3 = 5, and push the result (5) onto the stack: Stack: [5]

- Push 2 onto the stack: Stack: [5, 2]

- Push 4 onto the stack: Stack: [5, 2, 4]

- Encountering '+', pop 4 and 2, calculate 2 + 4 = 6, and push the result (6) onto the stack: Stack: [5, 6]

- Encountering '/', pop 6 and 5, calculate 5 / 6 = 0.8333 (rounded), and push the result (0.8333) onto the stack: Stack: [0.8333]

- \*\*Result\*\*: The final result is approximately 0.8333.

\*\*Question 5\*\*: Convert the prefix expression / - 8 3 + 2 4 to infix notation and evaluate it.

\*\*Answer 5\*\*:

- \*\*Infix Notation\*\*: (8 - 3) / (2 + 4)

- \*\*Evaluation\*\*:

- Start from right to left.

- Encountering 4, push it onto the stack: Stack: [4]

- Encountering 2, push it onto the stack: Stack: [4, 2]

- Encountering '+', pop 2 and 4, calculate 2 + 4 = 6, and push the result (6) onto the stack: Stack: [6]

- Encountering 3, push it onto the stack: Stack: [6, 3]

- Encountering 8, push it onto the stack: Stack: [6, 3, 8]

- Encountering '-', pop 8 and 3, calculate 8 - 3 = 5, and push the result (5) onto the stack: Stack: [6, 5]

- Encountering '/', pop 5 and 6, calculate 6 / 5 = 1.2, and push the result (1.2) onto the stack: Stack: [1.2]

- \*\*Result\*\*: The final result is approximately 1.2.

\*\*Question 6\*\*: Convert the postfix expression 5 3 2 \* + 4 - to infix notation and evaluate it.

\*\*Answer 6\*\*:

- \*\*Infix Notation\*\*: 5 + (3 \* 2) - 4

- \*\*Evaluation\*\*:

- Start from left to right.

- Encountering 5, push it onto the stack: Stack: [5]

- Encountering 3, push it onto the stack: Stack: [5, 3]

- Encountering 2, push it onto the stack: Stack: [5, 3, 2]

- Encountering '\*', pop 2 and 3, calculate 3 \* 2 = 6, and push the result (6) onto the stack: Stack: [5, 6]

- Encountering '+', pop 6 and 5, calculate 5 + 6 = 11, and push the result (11) onto the stack: Stack: [11]

- Encountering 4, push it onto the stack: Stack: [11, 4]

- Encountering '-', pop 4 and 11, calculate 11 - 4 = 7, and push the result (7) onto the stack: Stack: [7]

- \*\*Result\*\*: The final result is 7.

These examples showcase the conversion and evaluation of expressions in various notations, including infix, prefix, and postfix.