MST Qns Marked as Red 😊

Stack as an ADT (Abstract Data Type)

An **Abstract Data Type (ADT)** is a data structure defined by a set of operations and the behavior of those operations, but **not by its implementation**. The details of how the operations are carried out are hidden from the user, and only the interface is provided.

What is Stack?

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. It supports basic operations such as push (adding an element), pop (removing an element), and peek (viewing the top element without removing it).

Usage:

- Expression evaluation (infix, postfix, and prefix).
- Managing function calls (call stack in programming languages).
- Undo operations in text editors.

Operations of Stack ADT:

1. **Push(element)**: Adds an element to the top of the stack.
2. **Pop()**: Removes and returns the top element of the stack.
3. **Peek() or Top()**: Returns the top element without removing it from the stack.
4. **isEmpty()**: Returns true if the stack is empty, otherwise false.
5. **isFull()**: (For array-based stacks) Returns true if the stack is full, otherwise false.
6. **Size()**: Returns the number of elements in the stack.

Properties of Stack ADT:

- **LIFO**: The stack follows a Last In, First Out order for all operations.
- **No Random Access**: You can only access the top element; direct access to elements deeper in the stack is not possible.

Array Representation of Stack

In the **array representation of a stack**, the stack is implemented using an array to store the elements. The operations like push and pop are performed on the array elements, and the top variable keeps track of the top element in the stack.

Structure:

- **Array**: The stack is backed by an array, which holds the elements of the stack.
- **Top**: A variable top is used to indicate the index of the last element added to the stack (the top of the stack). It starts at -1, indicating the stack is empty.

Operations = Same for stack ^

Advantages:

1. Simple and easy to implement.
2. O(1) time complexity for push, pop, and peek.
3. Efficient memory usage with direct index access.

Disadvantages:

1. Fixed size, leading to overflow or underutilization.
2. Wasted memory if not fully used.
3. No dynamic resizing without reallocation.

linked list representation of stack:

In a **linked list representation of a stack**, the stack is implemented using a linked list. Each element in the stack is a **node**, and the nodes are linked together using pointers.

Each node contains two parts:

1. **Data**: Stores the stack element.
2. **Next Pointer:** Points to the next node in the stack.

Top: The pointer that always points to the last inserted element in the stack (head of the linked list).

Operations = Same for stack ^

This structure allows the stack to grow or shrink dynamically, as memory is allocated or freed for each element. Unlike arrays, there's no fixed size.

Stack (As an Array):

Push Algorithm:

```
1. Start.
2. If top == MAX_SIZE - 1,
       print "Stack Overflow" and exit.
3. Increment top.
4. Set stack[top] = element.
5. Exit.
```

Pop Algorithm:

```
1. Start.
2. If top == -1,
       print "Stack Underflow" and exit.
3. Set element = stack[top].
4. Decrement top.
5. Return element.
6. Exit.
```

Peek Algorithm:

```
1. Start.
2. If top == -1,
       print "Stack Underflow" and exit.
3. Return stack[top].
4. Exit.
```

isEmpty Algorithm:

```
1. Start.
2. If top == -1, return true.
3. Else, return false.
4. Exit.
```

isFull Algorithm:

```
1. Start.
2. If top == MAX_SIZE - 1, return true.
3. Else, return false.
4. Exit.
```

Stack (As Linked List):

Push Algorithm:

```
1. Start
2. If avail == NULL, write "Overflow" and Goto step 8
3. Else, new_node = avail.
4. avail = avail -> next.
5. new_node -> data = data.
6. new_node -> next = top.
7. top = new_node.
8. Exit
```

Pop Algorithm:

```
1. If top == NULL,
        write "Underflow" and Goto 8
2. Else, temp = top.
```

```
3. top = top -> next.
4. data = temp -> data.
5. temp -> next = avail.
6. avail = temp.
7. Return data.
8. Exit
```

Peek Algorithm:

```
1. Start.
2. If top = NULL,
        Write "Stack is Empty”.
        Goto Step 5.
3. Set temp = top.
4. While temp != NULL:
        a. Print temp -> data.
        b. temp = temp -> next.
5. Return or exit.
```

isEmpty Algorithm:

```
1. Start.
1. If top = NULL,
        Write "Stack is Empty”.
        Goto Step 3.
2. Else, Write “Stack is Not Empty”.
3. Exit
```

isFull Algorithm:

```
1. If avail == NULL,
        Write “Stack is Full”.
2. Else,
        Write “Stack is Not Full”.
```

Recursion

Recursion uses the stack to manage function calls.

- Every time a function calls itself (recursive call), the current function's context (arguments, local variables, return address) is pushed onto the stack.
- The stack helps to keep track of the current execution state.
- Once the base case is met, the functions start returning, and the stack is unwound.

1. Direct Recursion:

- Direct recursion occurs when a function calls itself directly within its own definition.
- Example: A function factorial() that calls itself to compute the factorial of a number.

  Example:

```
int factorial(int n) {
        if (n == 0) return 1;
        return n * factorial(n - 1);  // Direct recursion
}
```

- **Usage**: Common in problems like factorial calculation, Fibonacci series, etc.

2. Indirect Recursion:

- Indirect recursion occurs when a function calls another function, which in turn calls the first function.
- Example: Function A() calls function B(), and B() calls A().
- **Example**:

```
void funcA() {
   funcB();
}
```

```
void funcB() {
    funcA();
}
```

- **Usage**: Used in more complex recursive relationships where multiple functions interact.

3. Tail Recursion:

- Tail recursion is when the recursive call is the last statement executed in the function, making it possible for the compiler to optimize the recursion into an iterative loop (Tail Call Optimization).
- **Example**:

```
int factorial(int n, int result = 1) {
    if (n == 0) return result;
    return factorial(n - 1, n * result);  // Tail recursion
}
```

- **Usage**: Efficient in terms of space, as the function doesn't add to the call stack after each call.

Here’s a more detailed explanation of the types of recursion:

---

1. Direct Recursion:

- Direct recursion occurs when a function calls itself directly within its own definition.
- Example: A function factorial() that calls itself to compute the factorial of a number.
- **Example**:

cpp

Copy code

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

- **Usage**: Common in problems like factorial calculation, Fibonacci series, etc.

---

2. Indirect Recursion:

- function **calls another function**, which in turn calls the original function.
- The recursion is indirect because the function doesn't call itself directly.
- **Example**:

```
void funcA() {

    funcB();

}



void funcB() {

funcA();

}
```

Usage: Used in more complex recursive relationships where multiple functions interact.

3. Tail Recursion:

- Tail recursion is when the recursive call is the last statement executed in the function, making it possible for the compiler to optimize the recursion into an iterative loop (Tail Call Optimization).
- **Example**:

```
int factorial(int n, int result = 1) {

    if (n == 0) return result;

    return factorial(n - 1, n * result);

}
```

- **Usage**: Efficient in terms of space, as the function doesn't add to the call stack after each call.

4. Head Recursion:

- Head recursion occurs when the recursive call is made first in the function, followed by some operations after the recursion.
- **Example**:

```
void printNumbers(int n) {

    if (n <= 0) return;

    printNumbers(n - 1);  // Recursive call

    cout << n << " ";      // Operation after recursion

}
```

- **Usage**: Often used when we need to process elements after recursively solving the smaller problem.

5. Mutual Recursion:

- **Mutual recursion** is a specific case of indirect recursion where **two or more functions** call each other in a cyclic manner.
- The functions involved don't have to directly call the original function, but they call each other repeatedly.
- **Example**:

```
void funcA() {

    funcB();

}

void funcB() {

    funcC();

}

void funcC() {

    funcA();

}
```

Usage: Used in problems where multiple entities need to perform recursive calls on each other.

Polish Notations

Polish notation refers to mathematical notation where operators come before their operands. It was introduced by the Polish mathematician Jan Łukasiewicz. There are two types of Polish notation: **Prefix Notation** and **Postfix Notation**.

Polish Notation (Prefix and Postfix)

1. **Prefix (Polish) Notation**:
   - Operators come **before** operands.
   - Example: (A + B) → + A B
   - Conversion: Reverse the expression → convert to postfix → reverse the result.
2. **Postfix (Reverse Polish) Notation**:
   - Operators come **after** operands.
   - Example: (A + B) → A B +
   - Conversion: Use a stack to process the expression.

Tower of Hanoi:

The Tower of Hanoi is a classic recursive problem where you need to move n disks from a source peg to a destination peg, using an auxiliary peg, following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on a larger disk or on an empty peg.
3. The goal is to move all disks to the destination peg.

Recursive Algorithm (in steps):

```
1. Start
2. If n == 1 (only one disk):
     o Move the disk from the source peg to the destination peg.
     o Return.
3. Else:
     o Move n-1 disks from source to auxiliary peg using destination peg.
     o Move the nth disk (largest) from source to destination peg.
     o Move n-1 disks from auxiliary to destination peg using source peg.
4. End
```