SYLLABUS

1.) Stacks: Introduction to stacks:

2.) properties and operations,

3.) implementing stacks using arrays and linked lists,

4.) Applications of stacks in expression evaluation,

5.) backtracking,

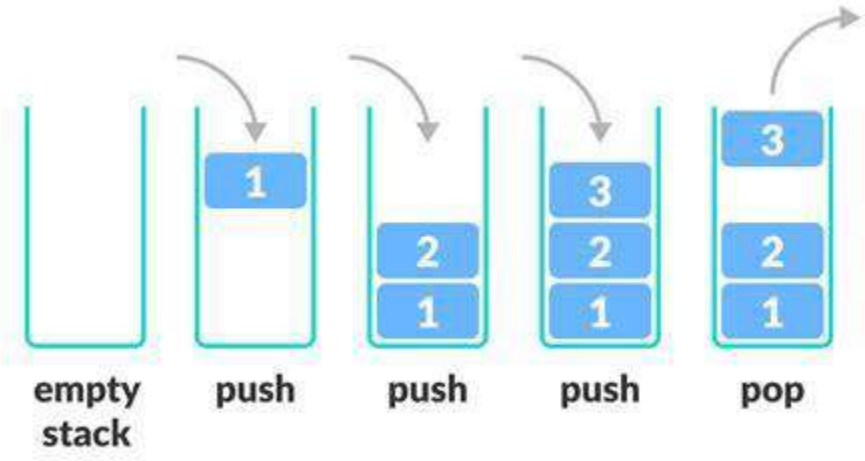
6.) reversing list etc.

1.) INTRODUCTION TO STACK

Certainly! Let's dive into the world of **Stacks**. A stack is a fundamental data structure that follows the **Last-In-First-Out (LIFO)** principle. In other words, the element that is inserted last will come out first. Imagine a stack of plates kept on top of each other: the plate we put last is on top, and we remove the top plate first.

Here are the key points about stacks:

What is a Stack?



- A stack is a **linear data structure** in which the insertion of a new element and removal of an existing element takes place at the same end, represented as the **top** of the stack.
- To implement a stack, we maintain a pointer to the top of the stack, which is the last element to be inserted. We can only access elements from the top of the stack.

Basic Operations on Stack

1. **Push**:
    - Adds an item to the stack.
    - If the stack is full, it results in an **Overflow** condition.
2. **Pop**:
    - Removes an item from the stack.
    - Items are popped in the reversed order in which they were pushed.
    - If the stack is empty, it results in an **Underflow** condition.
3. **Top**:
    - Returns the top element of the stack.
4. **isEmpty**:
    - Returns true if the stack is empty, else false.

Understanding Stack Practically

- Real-life examples of a stack include plates stacked over one another in a canteen. The plate at the top is the first one to be removed, following the LIFO order.

Complexity Analysis

- Time Complexity:
    - Push, Pop, isEmpty, and Top: O(1)
- Types of Stacks:
    - **Fixed Size Stack**: Has a fixed size and cannot grow or shrink dynamically.
    - **Dynamic Size Stack**: Can grow or shrink dynamically (implemented using a linked list).

Remember, stacks are used in various applications, including function calls, undo-redo operations, and managing data efficiently! □□

2.) PROPERTIES AND OPERATIONS OF STACK

Certainly! Let's explore the properties and operations of **stacks**.

Stack Properties

1. **LIFO (Last-In-First-Out)**:
    - The element that is inserted last will come out first.
    - Imagine a stack of plates: the plate we put last is on top, and we remove the top plate first.
2. **Top Element**:
    - The top element of the stack is the last element that was inserted.
    - We can only access elements from the top of the stack.

Basic Operations on Stacks

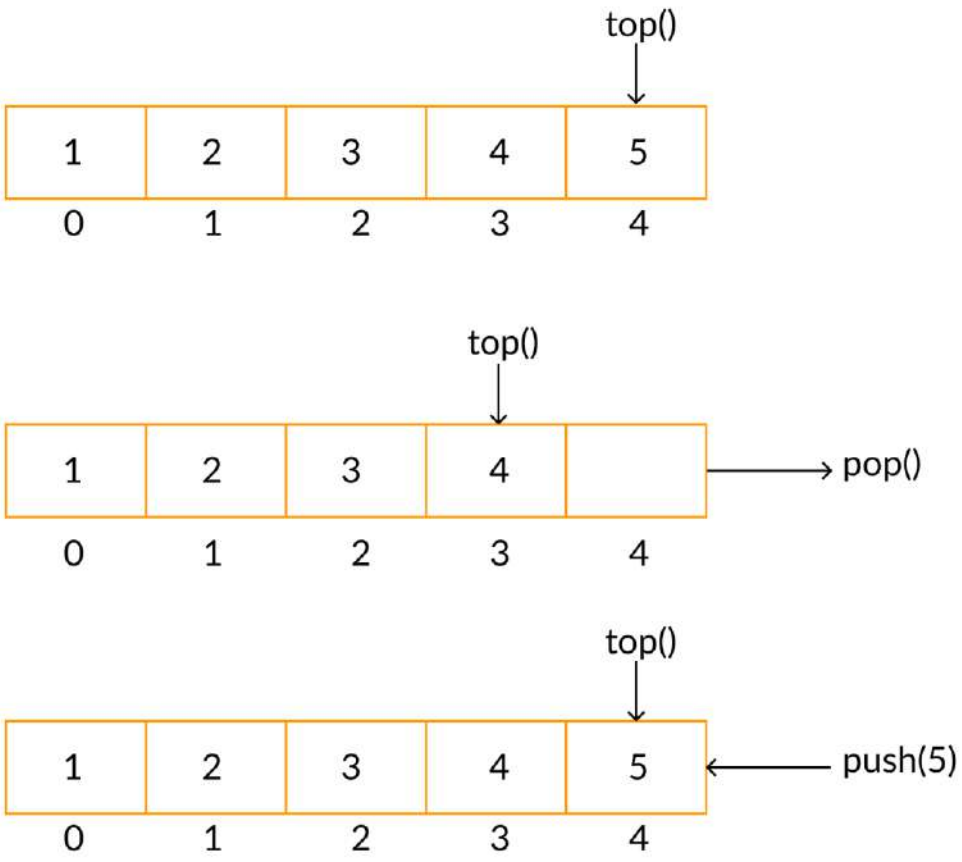
1. **Push**:
    - Adds an item to the stack.
    - If the stack is full, it results in an **Overflow** condition.
2. **Pop**:
    - Removes an item from the stack.
    - Items are popped in the reversed order in which they were pushed.
    - If the stack is empty, it results in an **Underflow** condition.
3. **Top**:
    - Returns the top element of the stack without removing it.
4. **isEmpty**:
    - Returns true if the stack is empty, else false.

3.) IMPLEMENTING STACKS USING ARRAYS AND LINKED LISTS.

Certainly! Let's explore the implementation of **stacks** using both arrays and linked lists.

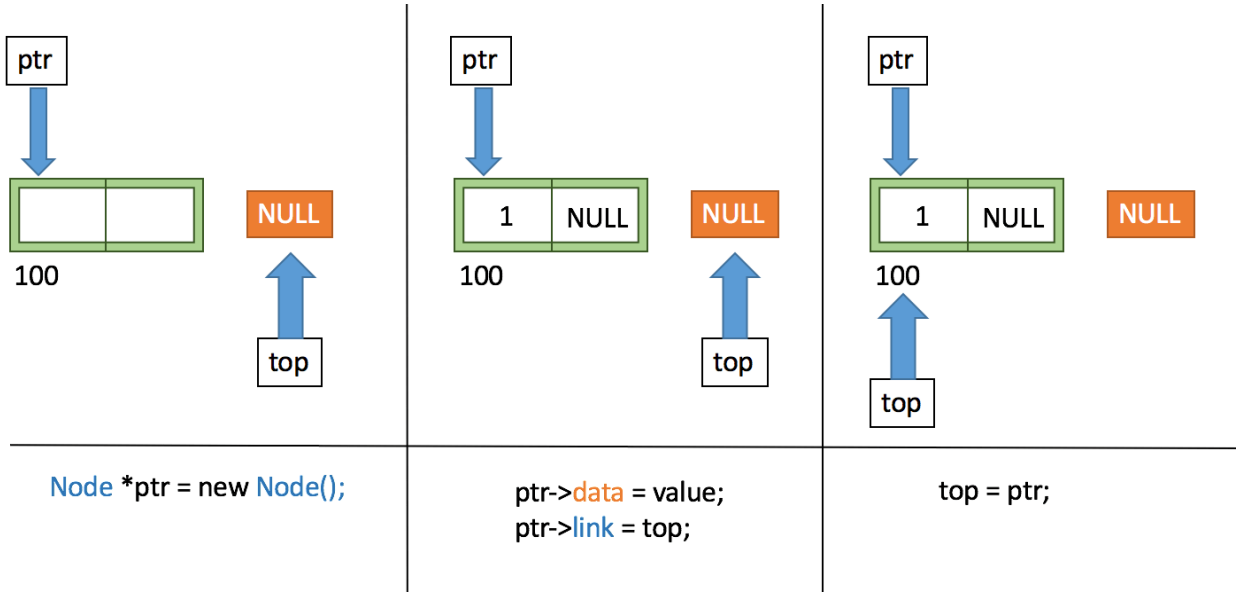
Stack Using Arrays

- A stack can be implemented using a fixed-size array. However, arrays have limitations (fixed size), and the size of the stack must be predetermined.
- Here's an example of implementing a stack using an array in C:



```
#include <stdio.h>
#define MAX_SIZE 100

struct Stack {
    int arr[MAX_SIZE];
    int top;
};

void push(struct Stack* s, int item) {
    if (s->top == MAX_SIZE - 1) {
        printf("Stack overflow!\n");
        return;
    }
    s->arr[++s->top] = item;
}

int pop(struct Stack* s) {
    if (s->top == -1) {
        printf("Stack underflow!\n");
        return -1;
    }
    return s->arr[s->top--];
}

int main() {
    struct Stack myStack;
    myStack.top = -1;

    push(&myStack, 10);
    push(&myStack, 20);

    printf("Popped element: %d\n", pop(&myStack)); // Output: 20

    return 0;
}
```

Stack Using Linked Lists



- A linked list provides dynamic memory allocation and is more flexible for implementing a stack.
- Here's an example of implementing a stack using a singly linked list in C:

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

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

void push(struct Node** head, int item) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = item;
    newNode->next = *head;
    *head = newNode;
}
```

```
int pop(struct Node** head) {
    if (*head == NULL) {
        printf("Stack underflow!\n");
        return -1;
    }
    int poppedValue = (*head)->data;
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
    return poppedValue;
}

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

    push(&myStack, 10);
    push(&myStack, 20);

    printf("Popped element: %d\n", pop(&myStack)); // Output: 20

    return 0;
}
```

Remember, both array-based and linked list-based implementations have their advantages and disadvantages. Choose the one that suits your requirements! □□

5.) Expression Evaluation Using Stacks

Stacks play a crucial role in evaluating expressions, especially arithmetic expressions. Here are some common scenarios where stacks are used:

1. **Infix to Postfix Conversion**:

- Converting an infix expression (e.g., `3 + 4 * 2`) to postfix notation (e.g., `3 4 2 * +`).
- The stack helps maintain the correct order of operators during conversion.

2. **Postfix Expression Evaluation**:
   - Evaluating a postfix expression using a stack.
   - Example: Given `3 4 2 * +`, the result is `11`.
3. **Parentheses Matching**:
   - Using a stack to check if parentheses in an expression are balanced.
   - Example: `(a + b) * (c - d)` has balanced parentheses.
4. **Expression Parsing**:
   - Parsing and evaluating complex expressions involving operators and operands.
   - Stacks help manage the order of evaluation.

Remember, stacks are essential for handling expression evaluation efficiently! □□

5.) BACKTRACKING

Certainly! Let's delve into the world of **backtracking**. Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.

Here are the key points about backtracking:

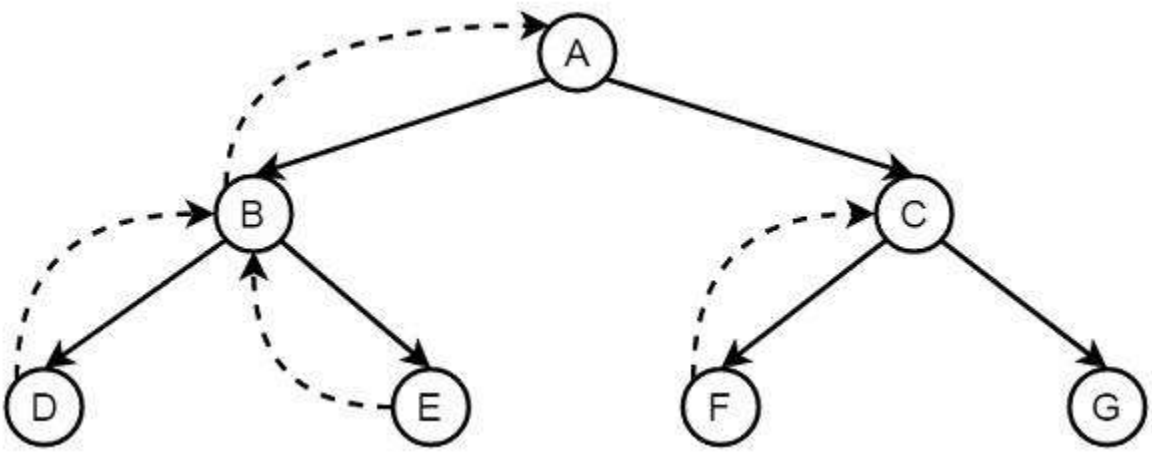
What is Backtracking?

- Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end.

- It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.

How Does a Backtracking Algorithm Work?

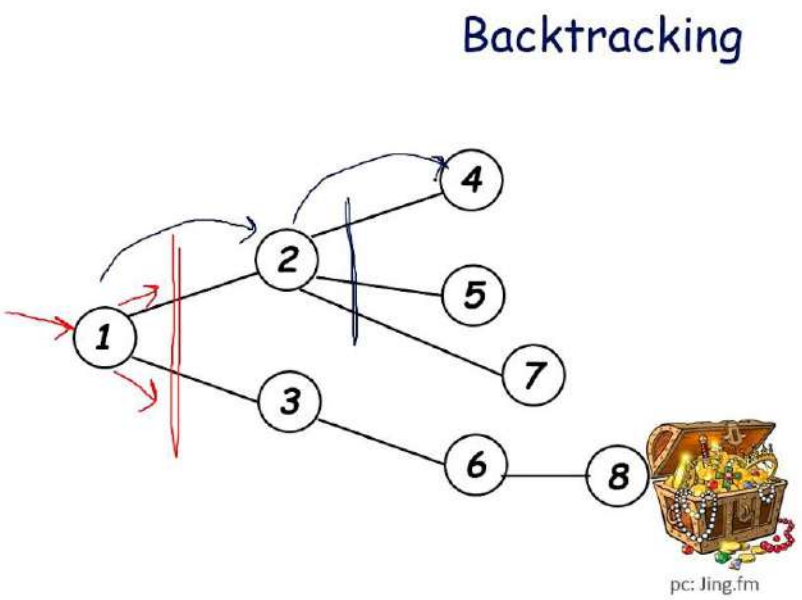
A backtracking algorithm works by recursively exploring all possible solutions to a problem. Here's a general outline of how it works:



1. **Choose an Initial Solution**:
    - Start with an initial solution (e.g., an empty path, an empty configuration, etc.).
2. **Explore All Possible Extensions**:
    - For the current solution, explore all possible extensions (e.g., adding a new element, moving to a neighboring cell, etc.).
3. **Check for a Solution**:
    - If an extension leads to a solution (satisfies the problem constraints), return that solution.
4. **Backtrack If Necessary**:
    - If an extension does not lead to a solution (e.g., violates constraints or reaches a dead end), backtrack to the previous solution and try a different extension.
5. **Repeat Steps 2-4**:

- Continue exploring different paths until a solution is found or all possibilities have been exhausted.

Example of Backtracking Algorithm



Example: Finding the shortest path through a maze

- Input: A maze represented as a 2D array, where 0 represents an open space and 1 represents a wall.
- Algorithm:
  - Start at the starting point.
  - For each of the four possible directions (up, down, left, right), try moving in that direction.
  - If moving in that direction leads to the ending point, return the path taken.
  - If moving in that direction does not lead to the ending point, backtrack to the previous position and try a different direction.

- Repeat the above steps until the ending point is reached or all possible paths have been explored.

When to Use a Backtracking Algorithm?

Backtracking algorithms are best used to solve problems that have the following characteristics:

- There are multiple possible solutions to the problem.
- The solution space can be explored incrementally.
- Constraints or conditions guide the exploration of the solution space.

Remember, backtracking is like solving a puzzle by testing different pieces until they fit together perfectly! □□

6.) Reversing a List Using Arrays

- To reverse an array, we can swap the elements from the outermost positions towards the center.
- Here's an example of reversing an array in C:

```
#include <stdio.h>

void reverseArray(int arr[], int n) {
    int start = 0;
    int end = n - 1;
    while (start < end) {
        // Swap arr[start] and arr[end]
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
```

```
        start++;
        end--;
    }
}

int main() {
    int myArray[] = {10, 20, 30, 40, 50};
    int n = sizeof(myArray) / sizeof(myArray[0]);

    printf("Original array: ");
    for (int i = 0; i < n; ++i) {
        printf("%d ", myArray[i]);
    }
    printf("\n");

    reverseArray(myArray, n);

    printf("Reversed array: ");
    for (int i = 0; i < n; ++i) {
        printf("%d ", myArray[i]);
    }
    printf("\n");

    return 0;
}
```

Reversing a List Using Linked Lists

- To reverse a linked list, we need to change the direction of the pointers.
- Here's an example of reversing a singly linked list in C:

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

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

void reverseLinkedList(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    *head = prev;
}

void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

```
int main() {
    struct Node* myLinkedList = NULL;
    // Create and add nodes to the linked list (not shown here)

    printf("Original linked list: ");
    printLinkedList(myLinkedList);

    reverseLinkedList(&myLinkedList);

    printf("Reversed linked list: ");
    printLinkedList(myLinkedList);

    return 0;
}
```

Remember, reversing a list is a common operation, and understanding both array-based and linked list-based approaches is essential! □□