STACK

A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.

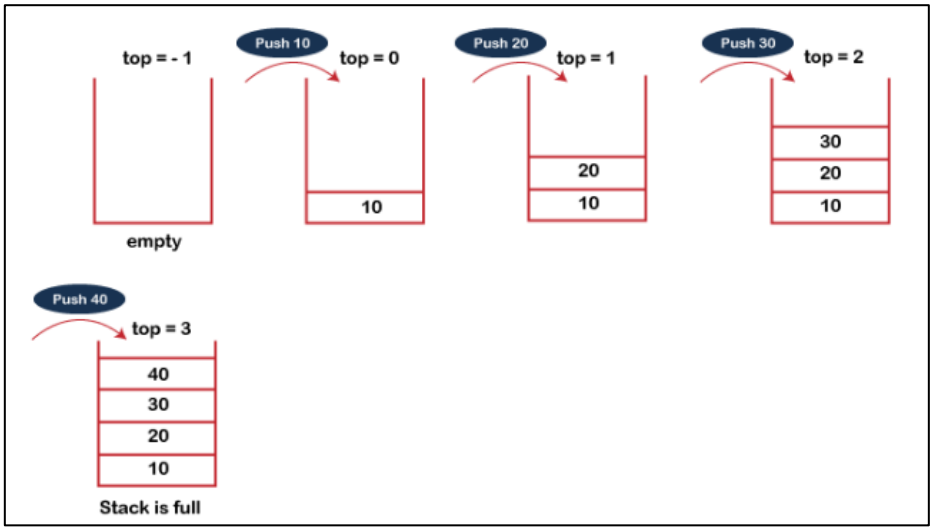
It contains only one pointer top pointer pointing to the topmost element of the stack.

Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.

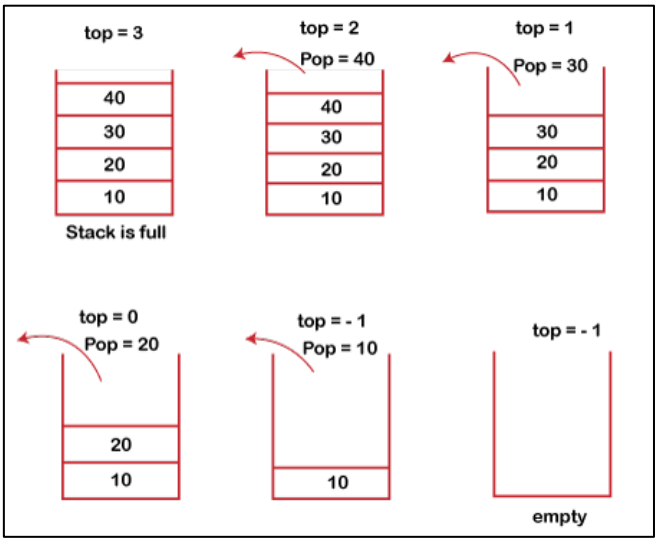
STACK OPERATIONS

The following are some common operations implemented on the stack:

push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.



pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.



isEmpty(): It determines whether the stack is empty or not.

isFull(): It determines whether the stack is full or not.'

peek(): It returns the element at the given position.

STACK APPLICATION

1. **String reversal:** Stack is also used for reversing a string.
2. **Function Call Management:** Stacks are used to manage function calls and returns in programming languages. Each function call is pushed onto the call stack, and when the function returns, its context is popped off the stack.
3. **Expression Evaluation/Expression conversion:** Stacks are used in algorithms for evaluating expressions. For example, in postfix (Reverse Polish Notation) expressions, a stack helps in evaluating the result by processing operators and operands.
4. **Undo Mechanisms:** Many applications, such as text editors, use stacks to implement undo mechanisms. Each action is pushed onto a stack, and undoing an action involves popping from the stack and reverting to the previous state.
5. **Backtracking Algorithms:** In algorithms that involve backtracking (like solving puzzles or mazes), stacks can be used to keep track of the path and states, allowing the algorithm to return to previous states when needed.

STACK IMPLEMENTATION USING ARRAY:

```
#include <iostream>

using namespace std;

class Stack {

private:

   int* arr;

   int capacity;

   int top;


public:

   // Constructor

   Stack(int size) : capacity(size), top(-1) {

      arr = new int[capacity];

   }
```

```
// Destructor
~Stack() {
    delete[] arr;
}

// Check if the stack is empty
bool isEmpty() const {
    return top == -1;
}

// Check if the stack is full
bool isFull() const {
    return top == capacity - 1;
}

// Push an element onto the stack
void push(int value) {
    if (isFull()) {
        cout << "Stack is full";
        return;
    }
    arr[++top] = value;
}

// Pop an element from the stack
int pop() {
    if (isEmpty()) {
        cout << "Stack is empty";
```

```
        return -1; // Assuming -1 indicates an error; choose a suitable error value
    }
    return arr[top--];
  }

  // Peek at the top element of the stack
  int peek() const {
    if (isEmpty()) {
        cout << "Stack is empty";
        return -1; // Assuming -1 indicates an error; choose a suitable error value
    }
    return arr[top];
  }

  // Print the stack
  void print() {
    if (isEmpty()) {
        cout << "Stack is empty";
        return;
    }
    for (int i = 0; i <= top; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
  }
};

// Example usage
int main() {
```

```
    Stack stack(5);

    stack.push(10);
    stack.push(20);
    stack.push(30);

    stack.print(); // Output: 10 20 30

    cout << "Popped: " << stack.pop() << endl; // Output: Popped: 30
    cout << "Top element: " << stack.peek() << endl; // Output: Top element: 20

    stack.print(); // Output: 10 20

    return 0;
}
```

```
Output:

10 20 30
Popped: 30
Top element: 20
10 20
```

TIME COMPLEXITY:

1. Push Operation: O(1)
   Adding an element to the top of the stack is a constant-time operation. It involves incrementing the top index and inserting the new element, both of which are done in constant time.

2. Pop Operation: O(1)
   Removing the top element from the stack is also a constant-time operation. It involves accessing the element at the top index and then decrementing the top index, both of which are done in constant time.

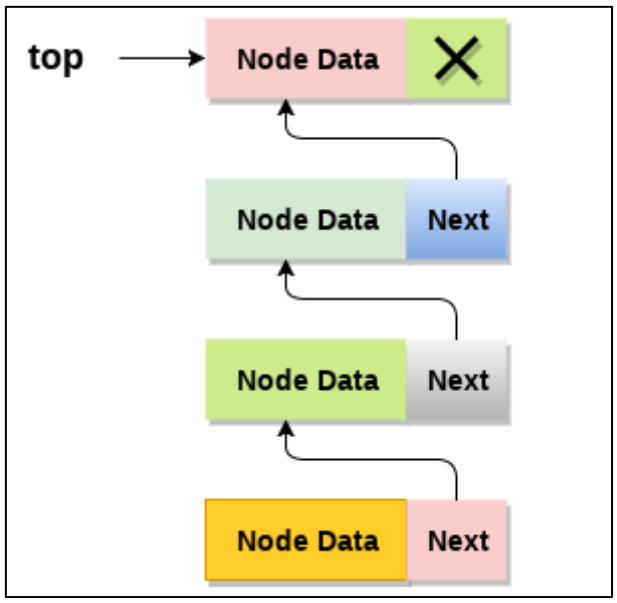
3. Peek Operation: O(1)
   Retrieving the top element without removing it is a constant-time operation. It simply involves accessing the element at the top index.

4. Is Empty Operation: O(1)
   Checking if the stack is empty involves comparing the top index with -1, which is a constant-time operation.

5. Is Full Operation: O(1)
   Checking if the stack is full involves comparing the top index with the maximum index (capacity - 1), which is also a constant-time operation.

STACK IMPLEMENTATION USING LIKED LIST

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically.




```
#include <iostream>


// Node class definition
class Node {
public:
    int data;
    Node* next;
```

```
    // Constructor
    Node(int value) : data(value), next(nullptr) {}
};

// Stack class definition
class Stack {
private:
    Node* top;

public:
    // Constructor
    Stack() : top(nullptr) {}

    // Destructor
    ~Stack() {
        while (!isEmpty()) {
            pop();
        }
    }

    // Push an element onto the stack
    void push(int value) {
        Node* newNode = new Node(value);
        newNode->next = top;
        top = newNode;
    }

    // Pop an element from the stack
    int pop() {
```

```
    if (isEmpty()) {
        std::cout << "Stack is empty" << std::endl;
        return -1; // Return -1 if stack is empty
    }
    Node* temp = top;
    int value = top->data;
    top = top->next;
    delete temp;
    return value;
}

// Peek at the top element of the stack
int peek() const {
    if (isEmpty()) {
        std::cout << "Stack is empty" << std::endl;
        return -1; // Return -1 if stack is empty
    }
    return top->data;
}

// Check if the stack is empty
bool isEmpty() const {
    return top == nullptr;
}

// Check if the stack is full (not applicable for linked list)
bool isFull() const {
    return false; // A linked list stack cannot be full unless out of memory
}
```

```
    // Print the stack
    void print() const {
        Node* current = top;
        if (isEmpty()) {
            std::cout << "Stack is empty" << std::endl;
            return;
        }
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

// Example usage
int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    stack.print(); // Output: 30 20 10

    std::cout << "Popped: " << stack.pop() << std::endl; // Output: Popped: 30
    std::cout << "Top element: " << stack.peek() << std::endl; // Output: Top element: 20
```

```
    stack.print(); // Output: 20 10


    std::cout << "Is stack empty? " << (stack.isEmpty() ? "Yes" : "No") << std::endl; // Output:
No
    std::cout << "Is stack full? " << (stack.isFull() ? "Yes" : "No") << std::endl; // Output: No


    return 0;
}
```

Output:

```
30 20 10
Popped: 30
Top element: 20
20 10
Is stack empty? No
Is stack full? No
```