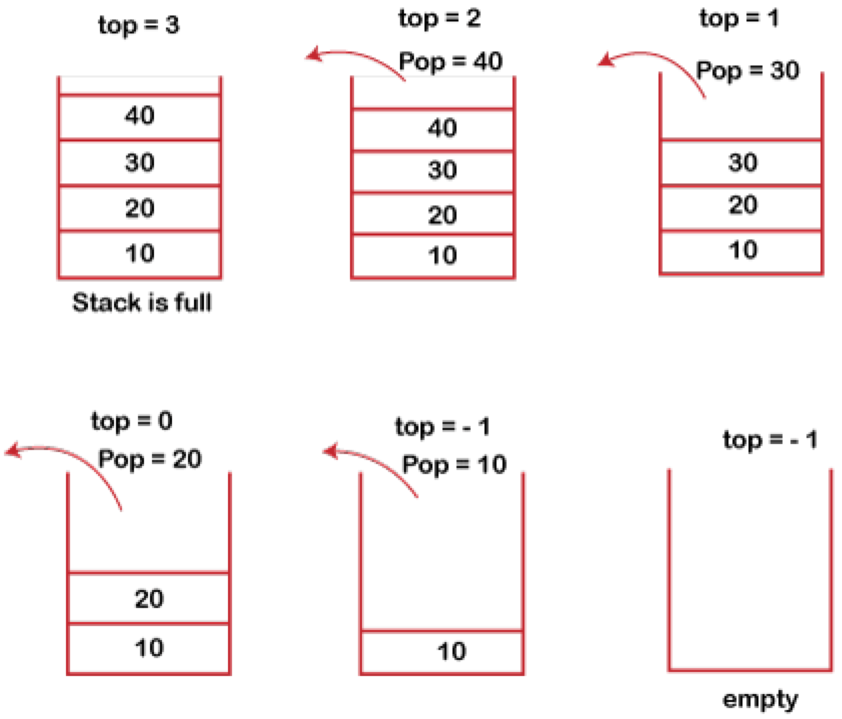

Applications of Stack

The following are the applications of the stack:

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

```
1. int main()
2. {
3.    cout<<"Hello";
4.    cout<<"javaTpoint";
5. }
```

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack. First, we push all the characters of the string in a stack until we reach the null character. After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state. If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

Expression conversion: Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
Infix to prefix
Infix to postfix

Prefix to infix
Prefix to postfix

- **Postfix to infix**
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Algorithms and their complexity analysis

Complexity analysis of different stack operations:

1) push():
This operation pushes an element on top of the stack and the top pointer points to the newly pushed element. It takes one parameter and pushes it onto the stack.
Below is the implementation of push() using Array :

```
#include <bits/dtc++.h>
using namespace std;

class Stack {
public:
    int stack[10];
    int MAX = 10;
    int top;

    Stack() { top = -1; }

    void push(int val)
    {
        // If top is pointing to
        // maximum size of stack
        if (top >= MAX - 1) {

            // Stack is full
            cout << "Stack Overflow\n";
            return;
        }

        // Point top to new top
        top++;

        // Insert new element at top of stack
        stack[top] = val;
        cout << val
             << " pushed into stack successfully !\n";
    }
};
```

```
int main()
{
    Stack st;
    st.push(1);
    st.push(2);
    return 0;
}
```

Output

```
1 pushed into stack successfully !
2 pushed into stack successfully !
```

Complexity Analysis:

- Time Complexity: O(1), In the push function a single element is inserted at the last position. This takes a single memory allocation operation which is done in constant time.
- Auxiliary Space: O(1), As no extra space is being used.

Below is the implementation of push() using Linked List :

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int val)
    {
        data = val;
        next = NULL;
    }
};

class Stack {
public:
    Node* top;

    Stack() { top = NULL; }

    void push(int val)
    {
        // Create new node temp and
        // allocate memory in heap
        Node* temp = new Node(val);

        // If stack is empty
        if (!top) {
            top = temp;
            cout << val
                 << " pushed into stack successfully !\n";
            return;
        }

        temp->next = top;
        top = temp;
        cout << val
             << " pushed into stack successfully !\n";
    }
};

int main()
{
    Stack st;
    st.push(1);
    st.push(2);
    return 0;
}
```

Output

```
1 pushed into stack successfully !
2 pushed into stack successfully !
```

Complexity Analysis:

- Time Complexity: O(1), Only a new node is created and the pointer of the last node is updated. This includes only memory allocation operations. Hence it can be said that insertion is done in constant time.
- Auxiliary Space: O(1), No extra space is used.

2) pop():

This operation removes the topmost element in the stack and returns an error if the stack is already empty.

Below is the implementation of pop() using Array:

- C++

```
#include <bits/stdc++.h>
using namespace std;

class Stack {
public:
    int stack[10];
    int MAX = 10;
    int top;

    Stack() { top = -1; }

    void push(int val)
    {
        // If top is pointing to maximum size of stack
        if (top >= MAX - 1) {

            // Stack is full
            cout << "Stack Overflow\n";
            return;
        }

        // Point top to new top
        top++;

        // Insert new element at top of stack
        stack[top] = val;
        cout << val
             << " pushed into stack successfully !\n";
    }

    void pop()
    {
        // Stack is already empty
        if (top < 0) {
            cout << "Stack Underflow";
        }
        else {
            // Removing top of stack
            int x = stack[top--];
            cout << "Element popped from stack : " << x
                 << "\n";
        }
    }
};

int main()
{
    Stack st;
    st.push(1);

    st.pop();
    st.pop();
    return 0;
}
```

Output

```
1 pushed into stack successfully !
Element popped from stack : 1
Stack Underflow
```

Complexity Analysis:

- Time Complexity: O(1), In array implementation, only an arithmetic operation is performed i.e., the top pointer is decremented by 1. This is a constant time function.
- Auxiliary Space: O(1), No extra space is utilized for deleting an element from the stack.

Below is the implementation of pop() using Linked List :

-

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int val)
    {
        data = val;
        next = NULL;
    }
};

class Stack {
public:
    Node* top;

    Stack() { top = NULL; }

    void push(int val)
    {
        // Create new node temp and allocate memory in heap
        Node* temp = new Node(val);

        // If stack is empty
        if (!top) {
            top = temp;
            cout << val
                 << " pushed into stack successfully !\n";
            return;
    }

    temp->next = top;
    top = temp;
    cout << val
         << " pushed into stack successfully !\n";
}

void pop()
{
    Node* temp;

    // Check for stack underflow
    if (top == NULL) {
        cout << "Stack Underflow\n"
             << endl;
        return;
    }
    else {

        // Assign top to temp
        temp = top;

        cout << "Element popped from stack : "
             << temp->data << "\n";

        // Assign second node to top
        top = top->next;

        // This will automatically destroy
        // the link between first node and second node
```

```
            // Release memory of top node
            // i.e delete the node
            free(temp);
        }
    }
};

int main()
{
    Stack st;
    st.push(1);

    st.pop();
    st.pop();
    return 0;
}
```

Output

```
1 pushed into stack successfully !
Element popped from stack : 1
Stack Underflow
```

Complexity Analysis:

- Time Complexity: O(1), Only the first node is deleted and the top pointer is updated. This is a constant time operation.
- Auxiliary Space: O(1). No extra space is utilized for deleting an element from the stack.

3) peek():

This operation prints the topmost element of the stack.
Below is the Implementation of peek() using Array:

- C++

```
#include <bits/stdc++.h>
using namespace std;

class Stack {
public:
    int stack[10];
    int MAX = 10;
    int top;

    Stack() { top = -1; }

    void push(int val)
    {
        // If top is pointing to maximum size of stack
        if (top >= MAX - 1) {

            // Stack is full
            cout << "Stack Overflow\n";
            return;
        }

        // Point top to new top
        top++;

        // Insert new element at top of stack
        stack[top] = val;
        cout << val
             << " pushed into stack successfully !\n";
    }

    int peek()
    {
        // Stack is already empty then
        // we can't get peek element
        if (top < 0) {
            cout << "Stack is Empty\n";
            return 0;
        }
        else {

            // Retrieving top element from stack
            int x = stack[top];
            return x;
        }
    }
};

int main()
{
    Stack st;
    st.push(1);
    st.push(2);

    cout << "Peek element of stack : "
         << st.peek() << "\n";
    return 0;
}
```

Output

```
1 pushed into stack successfully !
2 pushed into stack successfully !
Peek element of stack : 2
```

Complexity Analysis:

- Time Complexity: O(1), Only a memory address is accessed. This is a constant time operation.
- Auxiliary Space: O(1), No extra space is utilized to access the value.

Below is the implementation of peek() using Linked List :

- C++

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int val)
    {
        data = val;
        next = NULL;
    }
};

class Stack {
public:
    Node* top;

    Stack() { top = NULL; }

    void push(int val)
    {
        // Create new node temp and
        // allocate memory in heap
        Node* temp = new Node(val);

        // If stack is empty
        if (!top) {
            top = temp;
            cout << val
                 << " pushed into stack successfully !\n";
            return;
        }

        temp->next = top;
        top = temp;
        cout << val
             << " pushed into stack successfully !\n";
    }

    bool isEmpty()
    {
        // If top is NULL it means that
        // there are no elements are in stack
        return top == NULL;
    }

    int peek()
    {
        // If stack is not empty,
        // return the top element
        if (!isEmpty())
            return top->data;
        else
            cout << "Stack is Empty\n";
        exit(1);
    }
};

int main()
{
    Stack st;
    st.push(1);
    cout << "Peek element of stack : "
         << st.peek() << "\n";
    return 0;
}
```

Output
1 pushed into stack successfully !
Peek element of stack : 1
Complexity Analysis:

- Time Complexity: O(1). In linked list implementation also a single memory address is accessed. It takes constant time.
- Auxiliary Space: O(1). No extra space is utilized to access the element because only the value in the node at the top pointer is read.

4) isempty():
This operation tells us whether the stack is empty or not.
Below is the implementation of isempty() using Array :

- C++

```
#include <bits/stdc++.h>
using namespace std;

class Stack {
public:
    int stack[10];
    int MAX = 10;
    int top;

    Stack() { top = -1; }

    void push(int val)
    {
        // If top is pointing to
        // maximum size of stack
        if (top >= MAX - 1) {

            // Stack is full
            cout << "Stack Overflow\n";
            return;
        }

        // Point top to new top
        top++;

        // Insert new element at top of stack
        stack[top] = val;
        cout << val
             << " pushed into stack successfully !\n";
    }

    bool isEmpty()
    {
        // If stack is empty return 1
        return (top < 0);
    }
};

int main()
{
    Stack st;
    cout << st.isEmpty();

    return 0;
}
```

Output

```
1
```

Complexity Analysis:

- Time Complexity: O(1), It only performs an arithmetic operation to check if the stack is empty or not.
- Auxiliary Space: O(1), It requires no extra space.

Below is the implementation of isempty() using Linked List :

- C++

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int val)
    {
        data = val;
        next = NULL;
    }
};

class Stack {
public:
    Node* top;

    Stack() { top = NULL; }

    void push(int val)
    {
        // Create new node temp and allocate memory in heap
        Node* temp = new Node(val);

        // If stack is empty
        if (!top) {
            top = temp;
            cout << val
                 << " pushed into stack successfully !\n";
            return;
        }

        temp->next = top;
        top = temp;
        cout << val
             << " pushed into stack successfully !\n";
    }

    bool isEmpty()
    {
        // If top is NULL it means that
        // there are no elements are in stack
        return top == NULL;
    }
};

int main()
{
    Stack st;

    cout << st.isEmpty();
    return 0;
}
```

Output

```
1
```

Complexity Analysis:

- Time Complexity: O(1), It checks if the pointer of the top pointer is Null or not. This operation takes constant time.
- Auxiliary Space: O(1), No extra space is required.

5) size():

This operation returns the current size of the stack.
Below is the implementation of size() using Array:

- C++

```
#include <bits/stdc++.h>
using namespace std;

class Stack {
public:
    int stack[10];
    int MAX = 10;
    int top;

    Stack() { top = -1; }

    void push(int val)
    {
        // If top is pointing to maximum size of stack
        if (top >= MAX - 1) {

            // Stack is full
            cout << "Stack Overflow\n";
            return;
        }

        // Point top to new top
        top++;

        // Insert new element at top of stack
        stack[top] = val;
        cout << val
             << " pushed into stack successfully !\n";
    }
    int size() { return top + 1; }
};
```

```
int main()
{
    Stack st;
    st.push(1);
    st.push(2);

    cout << "The size of the stack is " << st.size()
         << endl;
    return 0;
}
```

Output

```
1 pushed into stack successfully !
```

```
2 pushed into stack successfully !
The size of the stack is 2
```

Complexity Analysis:

- Time Complexity: O(1), because this operation just performs a basic arithmetic operation.
- Auxiliary Space: O(1) NO extra space is required to calculate the value of the top pointer.

Below is the implementation of size() using Linked List:

- C++

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int val)
    {
        data = val;
        next = NULL;
    }
};

class Stack {
public:
    Node* top;
    Node* head;
    int sizeOfStack;
    Stack()
    {
        head = NULL;
        top = NULL;
        sizeOfStack = 0;
    }

    void push(int val)
    {
        // Create new node temp and
        // allocate memory in heap
        Node* temp = new Node(val);
        sizeOfStack += 1;

        // If stack is empty
        if (!top) {
            top = temp;
            return;
        }

        temp->next = top;
        top = temp;
    }

    int size() { return sizeOfStack; }
};

int main()
{
    Stack st;
    st.push(1);
    st.push(3);
    st.push(4);

    cout << "Size of stack : " << st.size();
    return 0;
}
```

Output
Size of stack : 3
Complexity Analysis:

- Time Complexity: O(1), because the size is calculated and updated every time a push or pop operation is performed and is just returned in this function.
- Auxiliary Space: O(1), NO extra space is required to calculate the size of the stack

Expression Conversion and evaluation - corresponding algorithms and complexity analysis:

Evaluate an expression represented by a String. The expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

- ***Infix Notation:*** Operators are written between the operands they operate on, e.g. 3 + 4.
- ***Prefix Notation:*** Operators are written before the operands, e.g + 3 4
- ***Postfix Notation:*** Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for

converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra.
This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to postfix notation. The same algorithm can be modified so that it outputs the result of the evaluation of expression instead of a queue. The trick is using two stacks instead of one, one for operands, and one for operators.

```
1. While there are still tokens to be read in,
   1.1 Get the next token.
   1.2 If the token is:
       1.2.1 A number: push it onto the value stack.
       1.2.2 A variable: get its value, and push onto the value stack.
       1.2.3 A left parenthesis: push it onto the operator stack.
       1.2.4 A right parenthesis:
         1 While the thing on top of the operator stack is not a
           left parenthesis,
             1 Pop the operator from the operator stack.
             2 Pop the value stack twice, getting two operands.
             3 Apply the operator to the operands, in the correct order.
             4 Push the result onto the value stack.
         2 Pop the left parenthesis from the operator stack, and discard it.
       1.2.5 An operator (call it thisOp):
         1 While the operator stack is not empty, and the top thing on the
           operator stack has the same or greater precedence as thisOp,
           1 Pop the operator from the operator stack.
           2 Pop the value stack twice, getting two operands.
```

3 Apply the operator to the operands, in the correct order.
4 Push the result onto the value stack.
2 Push thisOp onto the operator stack.
2. While the operator stack is not empty,
1 Pop the operator from the operator stack.
2 Pop the value stack twice, getting two operands.
3 Apply the operator to the operands, in the correct order.
4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value
stack should have only one value in it, which is the final result.

Implementation: It should be clear that this algorithm runs in linear time – each number or operator is pushed onto and popped from Stack only once.

- C++
- Java
- Python3
- C#
- Javascript