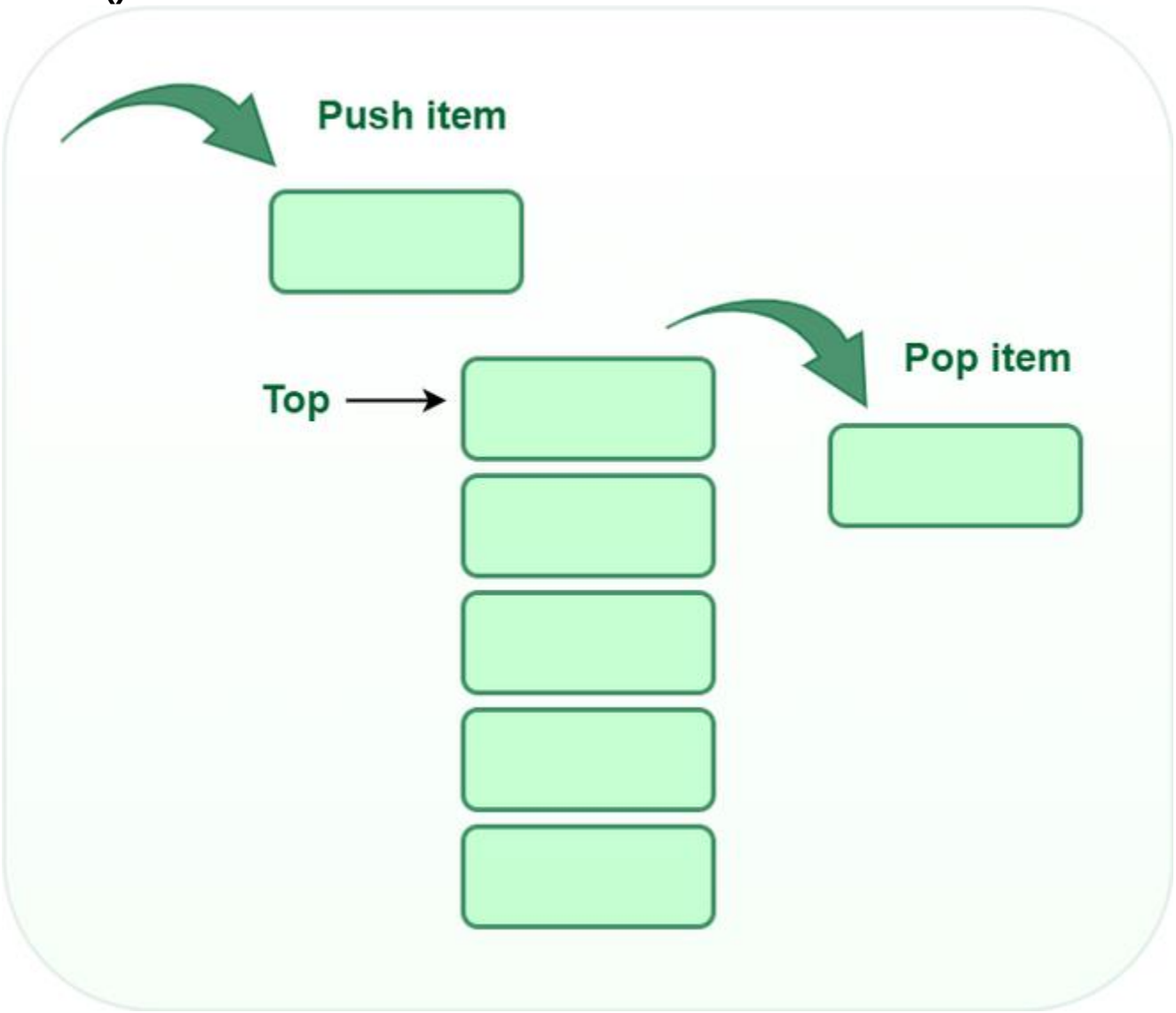
Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations provided to us.

- **push()** to insert an element into the stack
- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if stack is empty else false.
- **IsFull**: Check if the stack is full
- **Peek**: Get the value of the top element without removing it
- **size()** returns the size of stack.



Stack

Push:
Adds an item to the stack. If the stack is full, then it is said to be an **Overflow condition.**

Algorithm for push:

begin

if stack is full

return

endif

else

increment top

stack[top] assign value

end else

end procedure

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

int main(){

    stack<int> s; // creating a stack of integers

    s.push(1); // This pushes 1 to the stack top

    s.push(2); // This pushes 2 to the stack top

    s.push(3); // This pushes 3 to the stack top

    s.push(4); // This pushes 4 to the stack top

    s.push(5); // This pushes 5 to the stack top

    // printing the stack

    while(!s.empty())
    {
```

```
        cout<<s.top()<<" ";
        s.pop();
    }

    // The above loop prints "5 4 3 2 1"
}
```

Pop:

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

Algorithm for pop:

begin

if stack is empty

return

endif

else

store value of stack[top]

decrement top

return value

end else

end procedure

Top:

Returns the top element of the stack.

Algorithm for Top:

begin

return stack[top]

end procedure

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

```
int main(){

    stack<int> s; // creating a stack of integers

    s.push(1); // This pushes 1 to the stack top

    s.push(2); // This pushes 2 to the stack top

    s.push(3); // This pushes 3 to the stack top

    s.push(4); // This pushes 4 to the stack top

    s.push(5); // This pushes 5 to the stack top

    // printing the stack

    while(!s.empty())
    {
        cout<<s.top()<<" ";
        s.pop();
    }

    // The above loop prints "5 4 3 2 1"

    // Now, let us remove elements from the stack using pop function

    s.pop(); // removes 5 from the stack since 5 was present at the top.
Now, the new stack top becomes 4.
    s.pop(); // removes 4 from the stack since 4 was present at the top.
Now, the new stack top becomes 3.
    s.pop(); // removes 3 from the stack since 3 was present at the top.
Now, the new stack top becomes 2.
    s.pop(); // removes 2 from the stack since 2 was present at the top.
Now, the new stack top becomes 1.
    s.pop(); // removes 1 from the stack since 1 was present at the top.
Now, the stack becomes empty.

}
```

3. topElement() / peek()

TopElement / Peek is a function in the stack which is used to extract the element present at the stack top.

In C++, we use the top()function to extract the element present at the stack's top. Given below is the syntax of the top function in the stack.

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

int topElement(stack<int> &s)
{
    return s.top();
}

int main(){

    stack<int> s; // creating a stack of integers

    s.push(1); // This pushes 1 to the stack top
    cout<<topElement(s)<<endl; // Prints 1 since 1 is present at the stack top

    s.push(2); // This pushes 2 to the stack top
    cout<<topElement(s)<<endl; // Prints 2 since 2 is present at the stack top


    s.push(3); // This pushes 3 to the stack top
    cout<<topElement(s)<<endl; // Prints 3 since 3 is present at the stack top

}
```

isEmpty:

Returns true if the stack is empty, else false.

isEmpty is a boolean function in stack definition which is used to check whether the stack is empty or not. It returns true if the stack is empty. Otherwise, it returns false.

Algorithm for isEmpty:

```
begin
 if top < 1
    return true
 else
    return false
end procedure
```

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

bool isEmpty(stack<int> &s){

    bool isStackEmpty = s.empty(); // checking whether stack is empty or not and storing it into isStackEmpty variable

    return isStackEmpty; // returning bool value stored in isStackEmpty
}

int main(){

    stack<int> s;

    // The if - else conditional statements below prints "Stack is empty."
    if(isEmpty(s))
    {
        cout<<"Stack is empty."<<endl;
    }
    else{
        cout<<"Stack is not empty."<<endl;
    }

    s.push(1); //  Inserting value 1 to the stack top

    // The if - else conditional statements below prints "Stack is not empty."
    if(isEmpty(s))
```

```
    {
        cout<<"Stack is empty."<<endl;
    }
    else{
        cout<<"Stack is not empty."<<endl;
    }

}
```

5. isFull()

isFull is a function which is used to check whether the stack has reached its maximum limit of insertion of data or not i.e. if 'maxLimit' is the maximum number of elements that can be stored in the stack and if there are exactly maxLimit number of elements present in the stack currently, then the function isFull() returns true. Otherwise, if the number of elements present in the stack currently are less than 'maxLimit', then isFull() returns false.

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

int maxLimit = 3;

bool isFull(stack<int> &s)
{
    if(s.size() == maxLimit)
    {
        return true; // returns true if s.size() == maxLimit
    }

    return false; // returns false if s.size() != maxLimit
}

int main(){

    stack<int> s;

    s.push(1); // Inserting 1 in the stack
```

```
    s.push(2); // Inserting 2 in the stack

    // The below if - else statement prints "Stack is not full."

    if(isFull(s))
    {
        cout<<"Stack is full."<<endl;
    }
    else{
        cout<<"Stack is not full."<<endl;
    }

    s.push(3); // Inserting 3 in the stack

    // The below if - else statement prints "Stack is full."

    if(isFull(s))
    {
        cout<<"Stack is full."<<endl;
    }
    else{
        cout<<"stack is not full."<<endl;
    }

}
```

6. size()

Size is a function in stack definition which is used to find out the number of elements that are present inside the stack.

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

int main(){

    stack<int> s; // creating a stack of integers
```

```
    cout<<s.size()<<endl; // Prints 0 since the stack is empty

    s.push(1); // This pushes 1 to the stack top
    s.push(2); // This pushes 2 to the stack top
    cout<<s.size()<<endl; // Prints 2 since the stack contains two elements

    s.push(3); // This pushes 3 to the stack top
    cout<<s.size()<<endl; // Prints 3 since the stack contains three
elements

}
```

Understanding stacks practically:

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow the LIFO/FILO order.

Complexity Analysis:

- **Time Complexity**

Operations	Complexity
push()	O(1)
pop()	O(1)
isEmpty()	O(1)
size()	O(1)

Types of Stacks:

- **Fixed Size Stack**: As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.

- **Dynamic Size Stack**: A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

In addition to these two main types, there are several other variations of Stacks, including:

1. **Infix to Postfix Stack**: This type of stack is used to convert infix expressions to postfix expressions.
2. **Expression Evaluation Stack**: This type of stack is used to evaluate postfix expressions.
3. **Recursion Stack**: This type of stack is used to keep track of function calls in a computer program and to return control to the correct function when a function returns.
4. **Memory Management Stack**: This type of stack is used to store the values of the program counter and the values of the registers in a computer program, allowing the program to return to the previous state when a function returns.
5. **Balanced Parenthesis Stack**: This type of stack is used to check the balance of parentheses in an expression.
6. **Undo-Redo Stack**: This type of stack is used in computer programs to allow users to undo and redo actions.

Applications of the stack:

- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward features in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problems, and histogram problems.
- Backtracking is one of the algorithm designing techniques. Some examples of backtracking are the Knight-Tour problem, N-Queen problem, find your way through a maze, and game-like chess or checkers in all these problems we dive into someway if that way is not efficient we come back to the previous state and go into some another path. To get back from a current state we need to store the previous state for that purpose we need a stack.
- In Graph Algorithms like Topological Sorting and Strongly Connected Components

- In Memory management, any modern computer uses a stack as the primary management for a running purpose. Each program that is running in a computer system has its own memory allocations
- String reversal is also another application of stack. Here one by one each character gets inserted into the stack. So the first character of the string is on the bottom of the stack and the last element of a string is on the top of the stack. After Performing the pop operations on the stack we get a string in reverse order.
- Stack also helps in implementing function call in computers. The last called function is always completed first.
- Stacks are also used to implement the undo/redo operation in text editor.

Implementation of Stack:

A stack can be implemented using an array or a linked list. In an array-based implementation, the push operation is implemented by incrementing the index of the top element and storing the new element at that index. The pop operation is implemented by decrementing the index of the top element and returning the value stored at that index. In a linked list-based implementation, the push operation is implemented by creating a new node with the new element and setting the next pointer of the current top node to the new node. The pop operation is implemented by setting the next pointer of the current top node to the next node and returning the value of the current top node.

Stacks are commonly used in computer science for a variety of applications, including the evaluation of expressions, function calls, and memory management. In the evaluation of expressions, a stack can be used to store operands and operators as they are processed. In function calls, a stack can be used to keep track of the order in which functions are called and to return control to the correct function when a function returns. In memory management, a stack can be used to store the values of the program counter and the values of the registers in a computer program, allowing the program to return to the previous state when a function returns.

In conclusion, a Stack is a linear data structure that operates on the LIFO principle and can be implemented using an array or a linked list. The basic

operations that can be performed on a stack include push, pop, and peek, and stacks are commonly used in computer science for a variety of applications, including the evaluation of expressions, function calls, and memory management.There are two ways to implement a stack –

- Using array
- Using linked list

Implementing Stack using Arrays:

Implement Stack using Linked List

C++

```
/* C++ program to implement basic stack

   operations */

#include <bits/stdc++.h>



using namespace std;



#define MAX 1000



class Stack {
```

```
    int top;


public:
    int a[MAX]; // Maximum size of Stack


    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};


bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
```

```
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
```

```
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
```

```
}


// Driver program to test above functions
int main()
{
    class Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";

    //print top element of stack after popping
    cout << "Top element is : " << s.peek() << endl;

    //print all elements in stack :
    cout <<"Elements present in stack : ";
    while(!s.isEmpty())
```

```
    {

        // print top element in stack

        cout << s.peek() <<" ";

        // remove top element in stack

        s.pop();

    }



    return 0;

}
```

Advantages of array implementation:

- Easy to implement.
- Memory is saved as pointers are not involved.

Disadvantages of array implementation:

- It is not dynamic i.e., it doesn't grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
- The total size of the stack must be defined beforehand.

Implementing Stack using Linked List:

C++

```
// C++ program for linked list implementation of stack

#include <bits/stdc++.h>

using namespace std;


// A structure to represent a stack

class StackNode {

public:

    int data;

    StackNode* next;

};


StackNode* newNode(int data)

{

    StackNode* stackNode = new StackNode();

    stackNode->data = data;
```

```
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(StackNode* root)
{
    return !root;
}

void push(StackNode** root, int data)
{
    StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    cout << data << " pushed to stack\n";
}
```

```
int pop(StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);

    return popped;
}

int peek(StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
```

```
}


// Driver code
int main()
{
    StackNode* root = NULL;


    push(&root, 10);
    push(&root, 20);
    push(&root, 30);


    cout << pop(&root) << " popped from stack\n";


    cout << "Top element is " << peek(root) << endl;


    cout <<"Elements present in stack : ";
     //print all elements in stack :
```

```
    while(!isEmpty(root))
    {
        // print top element in stack
        cout << peek(root) <<" ";
        // remove top element in stack
        pop(&root);
    }

    return 0;
}
```

Advantages of Linked List implementation:

- The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- It is used in many virtual machines like JVM.

Disadvantages of Linked List implementation:

- Requires extra memory due to the involvement of pointers.
- Random accessing is not possible in stack.