Stack & Applications

Dr. GC Jana
Assistant Professor

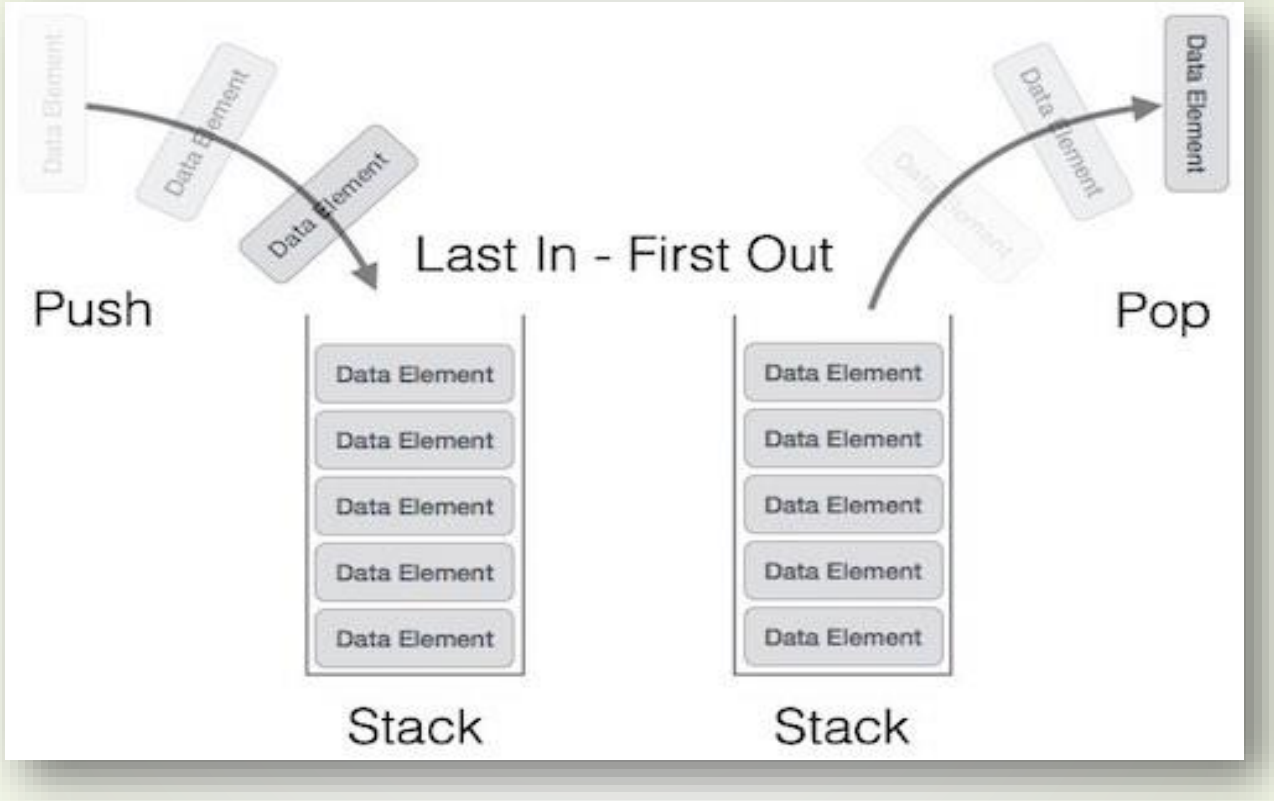
Today's discussion…

- Stack
  - Basic principles
  - Operation of stack
  - Stack using Array
  - Stack using Linked List
  - Applications of stack

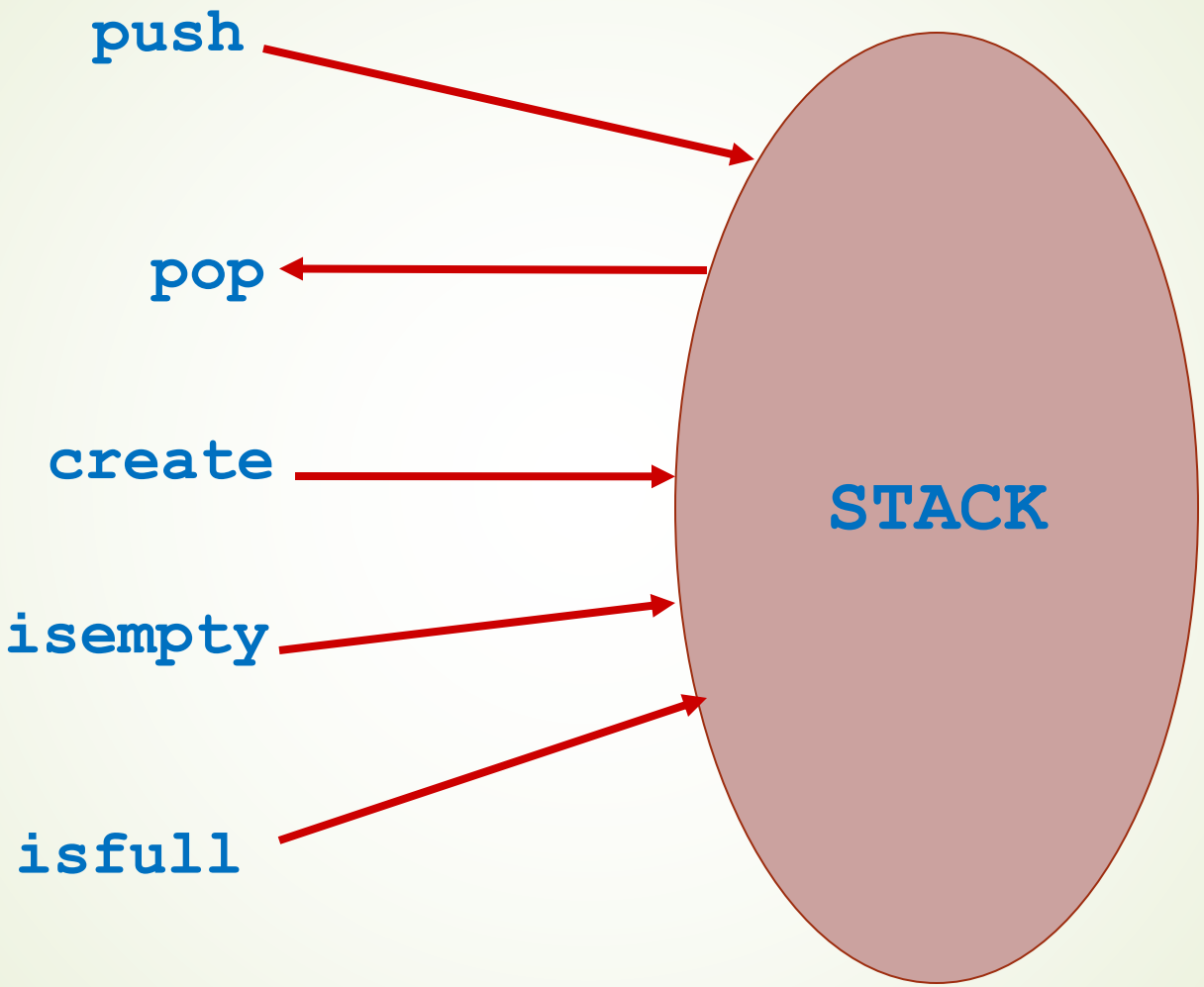
Basic Idea

- A stack is an **Abstract Data Type (ADT)**, commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

Stack Representation



- Can be implemented by means of Array, Structure, Pointers and Linked List.
- Stack can either be a **fixed size or dynamic**.


push
pop
create
isempty
isfull
STACK

STACK: Last-In-First-Out (LIFO)

- `void push (stack *s, int element);`

  /* Insert an element in the stack */

- `int pop (stack *s);`

  /* Remove and return the top element */

- `void create (stack  *s);`

  /* Create a new stack */

- `int isempty (stack *s);`

  /* Check if stack is empty */

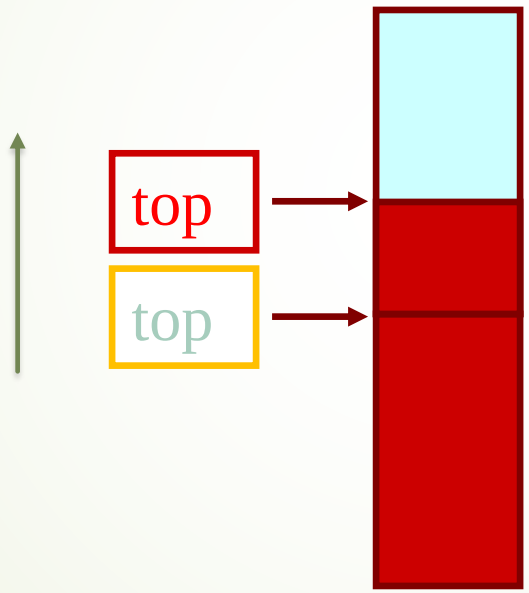
- `int isfull (stack *s);`

  /* Check if stack is full */

Assumption: stack contains integer elements!

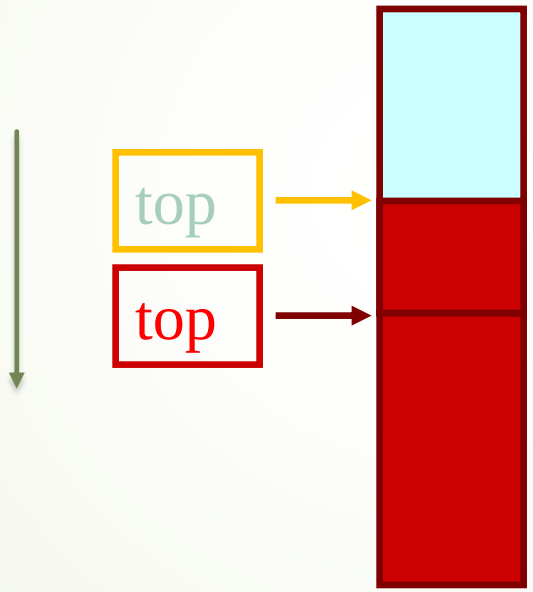
Stack using Array

Push using Stack



PUSH

Pop using Stack

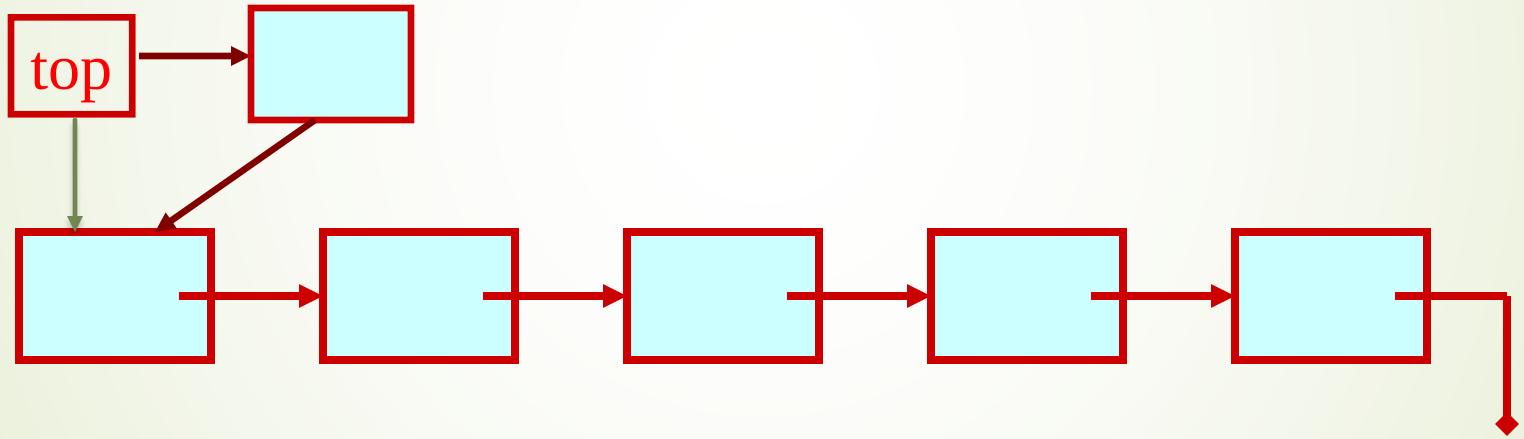


POP

Stack using Linked List

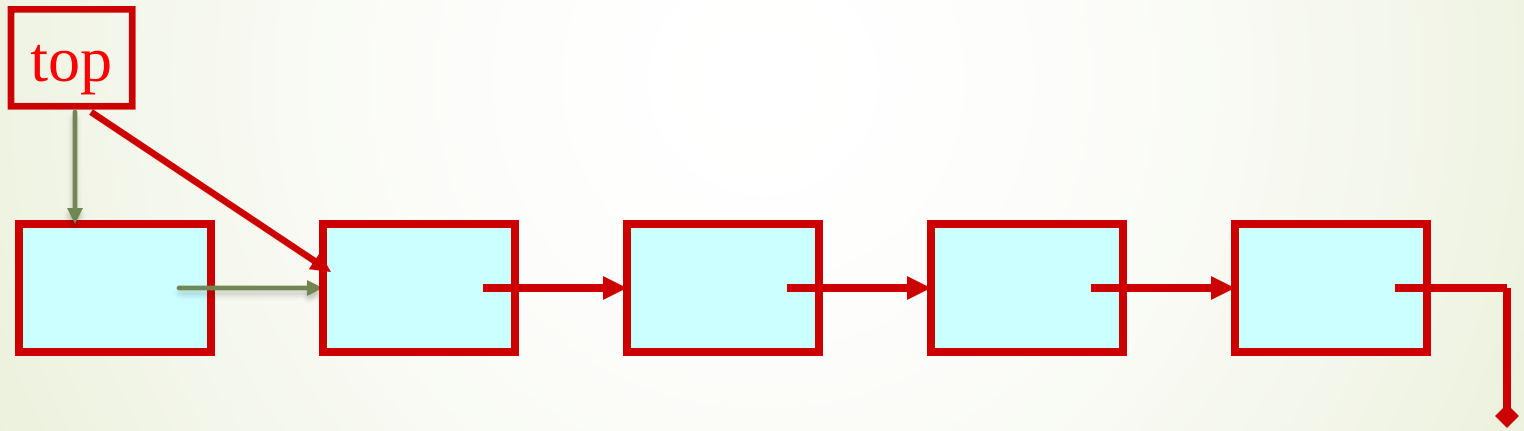
Push using Linked List

PUSH OPERATION



Pop using Linked List

POP OPERATION



Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack)
  - Keep a variable which always points to the “top” of the stack.
    - Contains the array index of the “top” element.

- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable top points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration & Creation

```
#define MAX 1000

class Stack {
    int top;

public:
    int a[MAX];
    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};
```

ARRAY

```
class Node
{
    public:
        int data;
        Node* link;
        // Constructor
        Node(int n)
        {
            this->data = n;
            this->link = NULL;
        }
        Stack() { top=NULL; }
};
```

LINKED LIST

Pushing an element into stack

```
bool push(int x)
{
    if (top >= (MAX - 1))
    {
        cout << "Stack Overflow";
        return false;
    }
    else
    {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
```

ARRAY

```
void push(int data)
{
    Node* temp = new Node(data);
    // Check if stack (heap) is full.

    if (!temp)
    {
        cout << "\nStack Overflow";
        exit(1);
    }

    temp->data = data;
    temp->link = top;
    top = temp;
}
```

LINKED LIST

Popping an element from stack

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

ARRAY

```
void pop()
{
    Node* temp;

    if (top == NULL)
    {
        cout << "\nStack Underflow" << endl;
        exit(1);
    }
    else
    {
        temp = top;
        top = top->link;
        free(temp);
    }
}
```

LINKED LIST

Checking for stack empty

```
bool isEmpty ()
{
   return (top<0);
}
```

ARRAY

```
bool isEmpty ()
{
   return (top==NULL);
}
```

LINKED LIST

Checking for peek

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

ARRAY

```
int peek()
{
    // If stack is not empty ,
    // return the top element

    if (!isEmpty())
        return top->data;
    else
        exit(1);
}
```

LINKED LIST

```
#include <stdio.h>
#define MAXSIZE 100
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 poping
    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 from stack
        s.pop();
    }
    return 0;
}
```

Example: A Stack using an Array

Example: A Stack using Linked List

```
int main()
{
    // Creating a stack
    Stack s;
    // Push the elements of stack
    s.push(11);
    s.push(22);
    s.push(33);
    s.push(44);
    // Display stack elements
    s.display();
    // Print top element of stack
    cout << "\nTop element is " << s.peek() << endl;
    // Delete top elements of stack
    s.pop();
    s.pop();
    // Display stack elements
    s.display();
    // Print top element of stack
    cout << "\nTop element is " << s.peek() << endl;
    return 0;
}
```

```
void display()
{
    Node* temp;
    // Check for stack underflow
    if (top == NULL)
    {
        cout << "\nStack Underflow";
        exit(1);
    }
    else
    {
        temp = top;
        while (temp != NULL)
        {
            cout << temp->data;
            temp = temp->link;
            if (temp != NULL)
                cout << " -> ";
        }
    }
}
```

Complexity Analysis
Time Complexity

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

Stack in C++ STL

The C++ STL offers a built-in class named **stack** for implementing the stack data structure easily and efficiently. This class provides almost all functions needed to perform the standard stack operations like push(), pop(), peek(), remove() etc..

Syntax:

```
stack< data_type > stack_name;

Here,
data_type: This defines the type of data to be
           stored in the stack.
stack_name: This specifies the name of the stack.
```

Some Basic functions of Stack class in C++:

- **empty()** – Returns whether the stack is empty.
- **size()** – Returns the size of the stack.
- **top()** – Returns a reference to the topmost element of the stack.
- **push(g)** – Adds the element 'g' at the top of the stack.
- **pop()** – Deletes the topmost element of the stack.

Stack in C++ STL

```
int main ()
{
    stack <int> s;
    s.push(10);
    s.push(30);
    s.push(20);
    s.push(5);
    s.push(1);

    cout << "The stack is : ";
    showstack(s);

    cout << "\ns.size() : " << s.size();
    cout << "\ns.top() : " << s.top();


    cout << "\ns.pop() : ";
    s.pop();
    showstack(s);

    return 0;
}
```

```
using namespace std;

void showstack(stack <int> s)
{
    while (!s.empty())
    {
        cout << '\t' << s.top();
        s.pop();
    }
    cout << '\n';
}
```

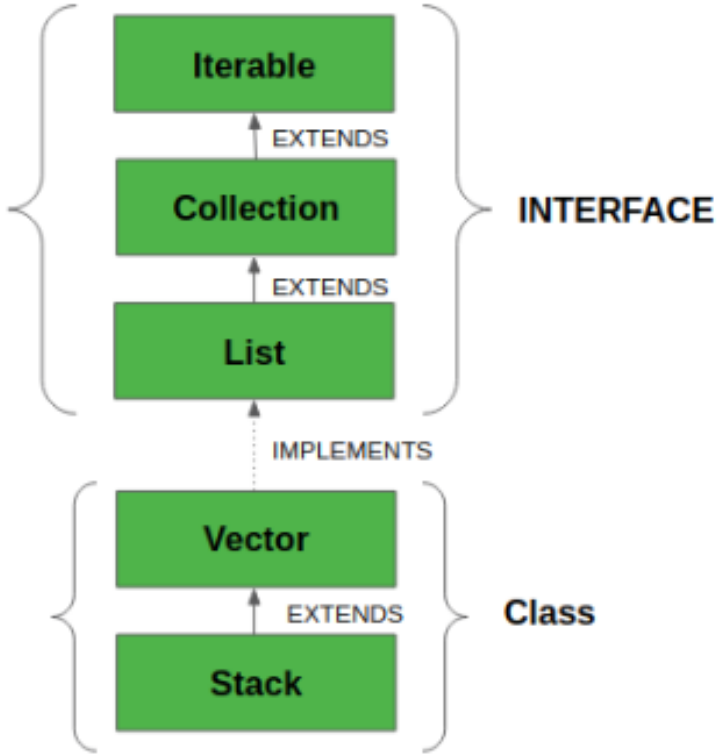
Output

```
The stack is :  1       5       20      30      10

s.size() : 5
s.top() : 1
s.pop() :       5       20      30      10
```

Stack in Java Collection

Java Collection framework provides a Stack class which models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector.

This diagram shows the hierarchy of Stack class:



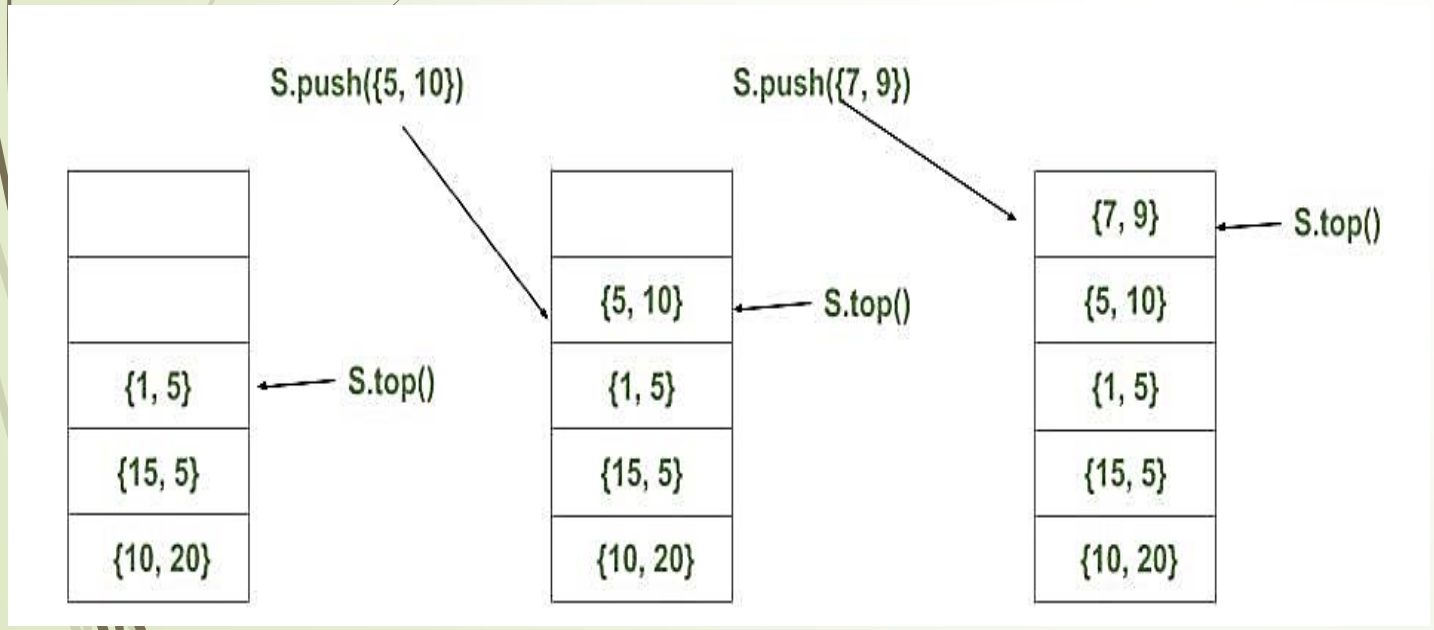
```java
import java.io.*;
import java.util.*;

class Test
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop :");

        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }
```

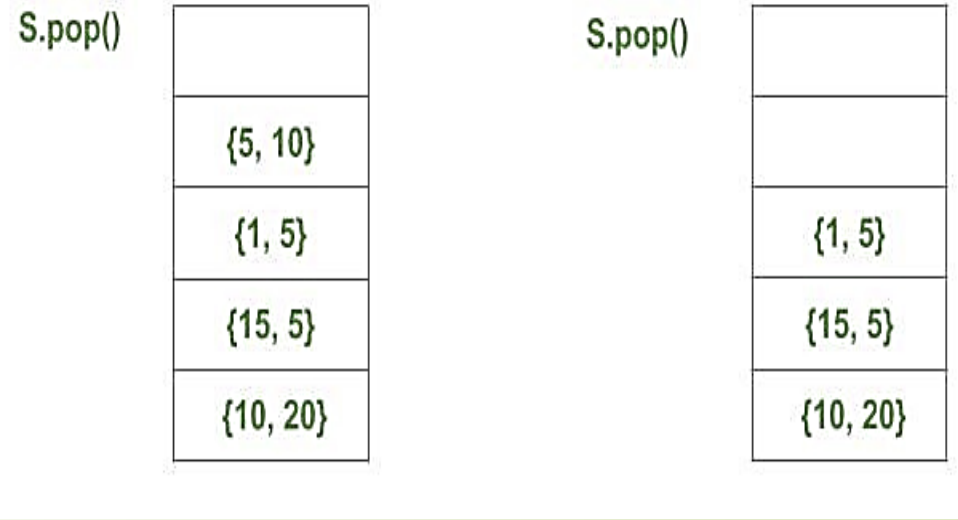
```java
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top : " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer) stack.search(element);

        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position " + pos);
    }

    public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
```

Stack-Pair in STL

- Stack in STL Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

- Pair in STL The pair container is a simple container defined in header consisting of two data elements or objects. The first element is referenced as 'first' and the second element as 'second' and the order is fixed (first, second).





Design a stack that supports getMin() in O(1) time and O(1) extra space

```
int mini(int a, int b) { return a > b ? b : a; }

class MinStack {
public:
    stack<pair<int, int> > s;

    void push(int element)    {
 /* new min will be given element, if stack is empty, else we compare
given element to min at current top of stack*/

        int new_min = s.empty()? element: mini(element, s.top().second);

        // we push the pair of given_element,new_min in s
         s.push({ element, new_min });
    }
```

```
int pop()    {
        int popped;
        if (!s.empty()) {
            // popped has popped number
            popped = s.top().first;
            s.pop();
        }
        else {
            // print a message or throw exception etc
        }
        return popped;
    }


  int minimum()    {
        int min_elem = s.top().second;
        return min_elem;
    }
};
```

```
// Driver code
int main()
{
    MinStack s;

    // Function calls
    s.push(-1);
    s.push(10);
    s.push(-4);
    s.push(0);
    cout << s.minimum() << endl;
    cout << s.pop() << endl;
    cout << s.pop() << endl;
    cout << s.minimum();
    return 0;
}
```

Applications of Stacks

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine
- Function-call abstraction
- Validate XML
- Auxiliary data structure for algorithms
- Evaluation of Arithmetic expressions
- Polish Notation conversion
- To manage the Symbol table in compile design.
- Balanced Parenthesis in arithmetic expressions.

What is stack size estimation?

Stack size estimation is process of determining the amount of memory allocated to the call stack for a particular program or thread. The call stack is a region of memory that operates on a Last In, First Out (LIFO) basis and is used for managing function calls and local variables.

Here are some key points related to stack size estimation:

- **Function Calls:** Each time a function is called, a new stack frame is created, which **includes space for local variables, return addresses, and other information related to the function call.**
- **Recursion:** Recursive functions can **lead to a significant increase in the stack size**, as each recursive call adds a new stack frame. **The depth of recursion directly impacts** the stack size required.
- **Local Variables:** The size of local variables within functions contributes to the overall stack size. **Larger local variables or arrays can consume more stack space**.
- **Thread Stack Size:** In a multithreaded program, **each thread typically has its own stack**. The stack size for each thread is specified during thread creation. **Estimating the required stack size is crucial to prevent stack overflow**.
- **Operating System Limits:** Operating systems often impose limits on the maximum stack size for a thread. Exceeding these limits can result in a stack overflow, leading to program termination.
- **Static Analysis and Tools:** Some programming languages provide static analysis tools that can estimate the stack size requirements based on the code structure and function calls.
- **Platform and Compiler Considerations:** Stack size can vary across different platforms and compilers. It's important for developers to be aware of the characteristics of the target environment and adjust stack sizes accordingly.

Any question?