Chapter 2
Data Structures
Static Stack

By: Dr. Aryaf A. Al-adwan

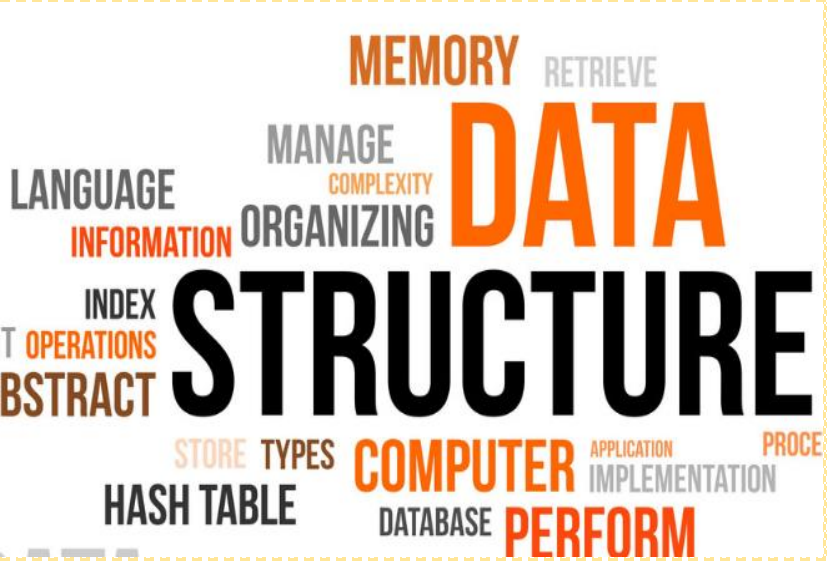
Faculty of Engineering Technology

Computer and Networks Engineering Dept.

Data Structures Course

Outline

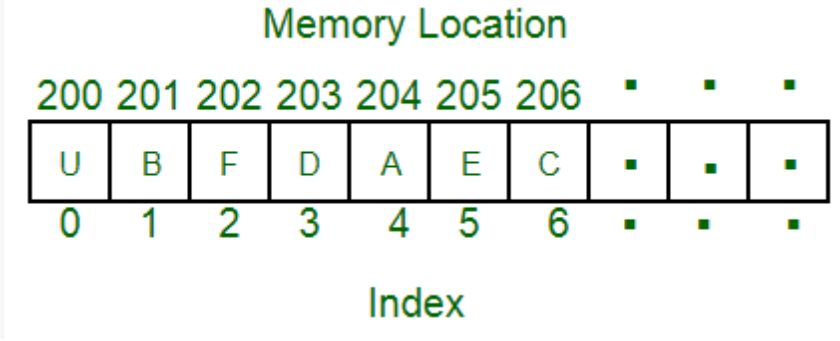
- Why Data Structures?
- Abstract Data Type
- Data Structures Types
- Stack Example
- Stack Data Members and Operations
- Stack Implementation in c++
- Stack Applications



Data Structures

What is Data Structure?

- A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.
- For example, we can store a list of items having the same data-type using the *array* data structure.



- Various Data Structures types are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science for simple as well as complex computations.
- Data structures are used in all areas of computer science such as Artificial Intelligence, graphics, Operating system etc.

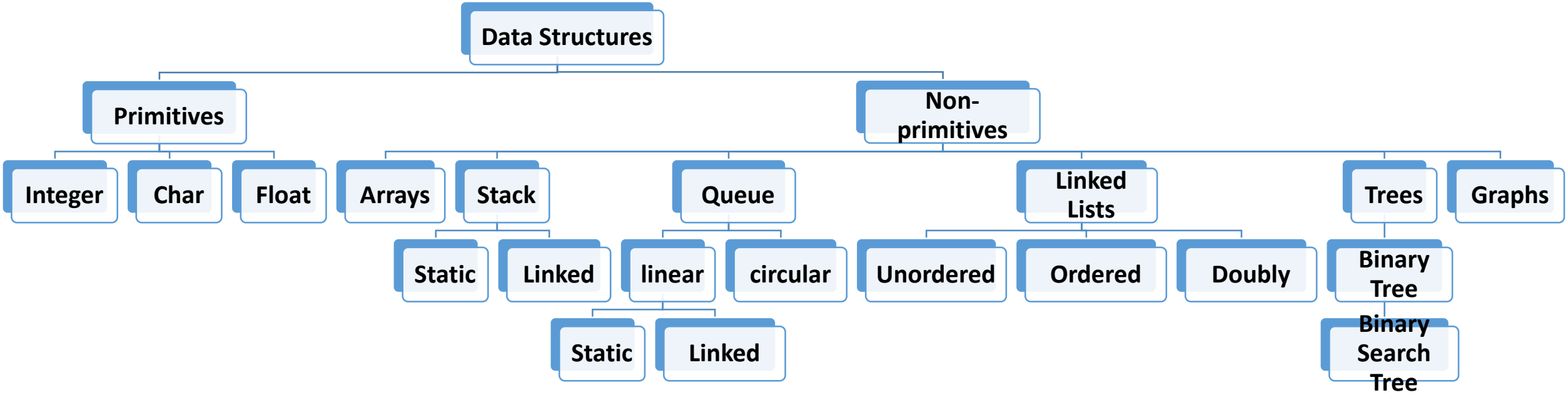
Why Data Structures?

1. Crucial part of several computer algorithms
2. They allow programmers to do data management efficiently
3. Data Structures are the key part of many computer algorithms as they allow the programmers to do data management in an efficient way. A right selection of data structure can enhance the efficiency of computer program or algorithm in a better way.
4. Data Search: Getting a particular record from database should be quick and with optimum use of resources.

Abstract Data Type (ADT)

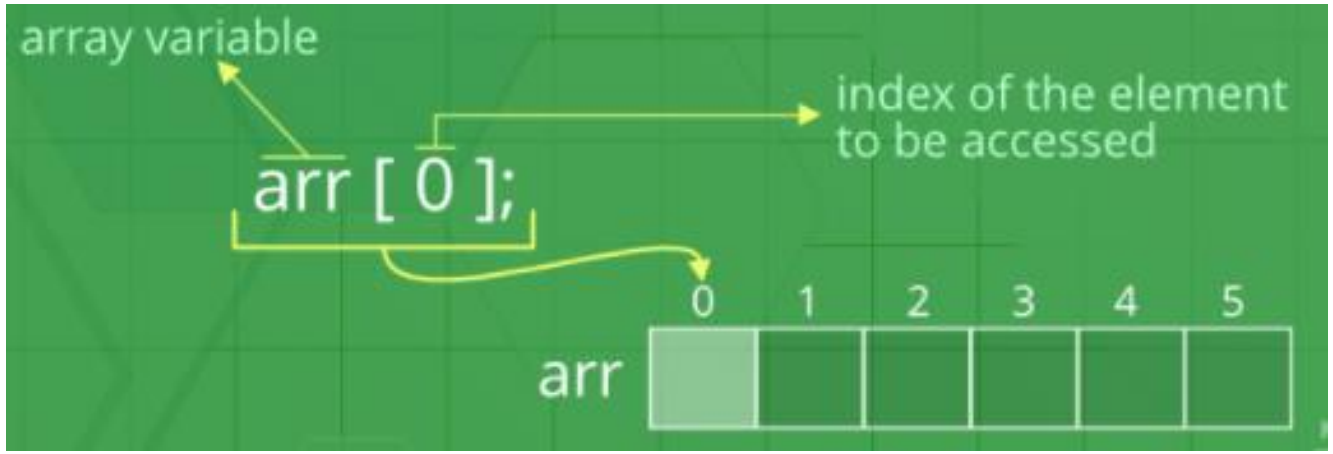
- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
- ADT = data + operations
- Can we consider the class as ADT?
- Program = ADT + Algorithm

Data Structures Types



Arrays

- An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.
- int a[6];



Arrays operations

Basic Operations

Following are the basic operations supported by an array:

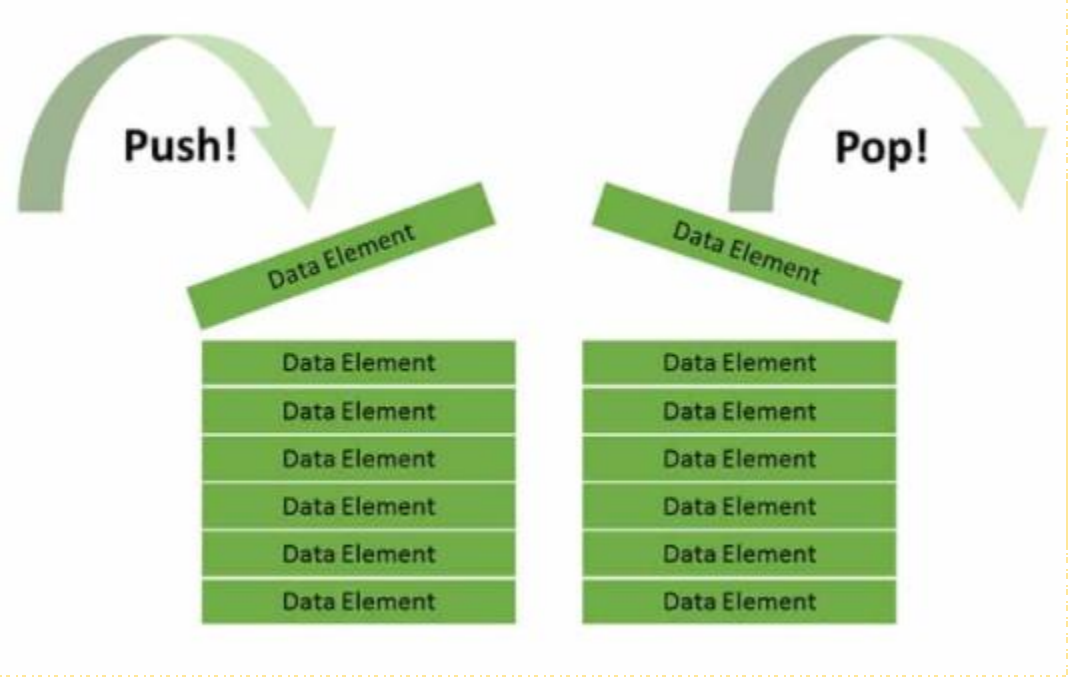
1. **Traverse** − print all the array elements one by one.
2. **Insertion** − Adds an element at the given index.
3. **Deletion** − Deletes an element at the given index.
4. **Search** − Searches an element using the given index or by the value.
5. **Update** − Updates an element at the given index.

int a[6];

Traverse	Insertion	Deletion	Search	Update

Disadvantages of Arrays

- The number of elements to be stored in an array should be known in advance.
- An array is a static structure (which means the array is of fixed size). Once declared the size of the array cannot be modified. The memory which is allocated to it cannot be increased or decreased.
- Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.
- Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.



Static Stack

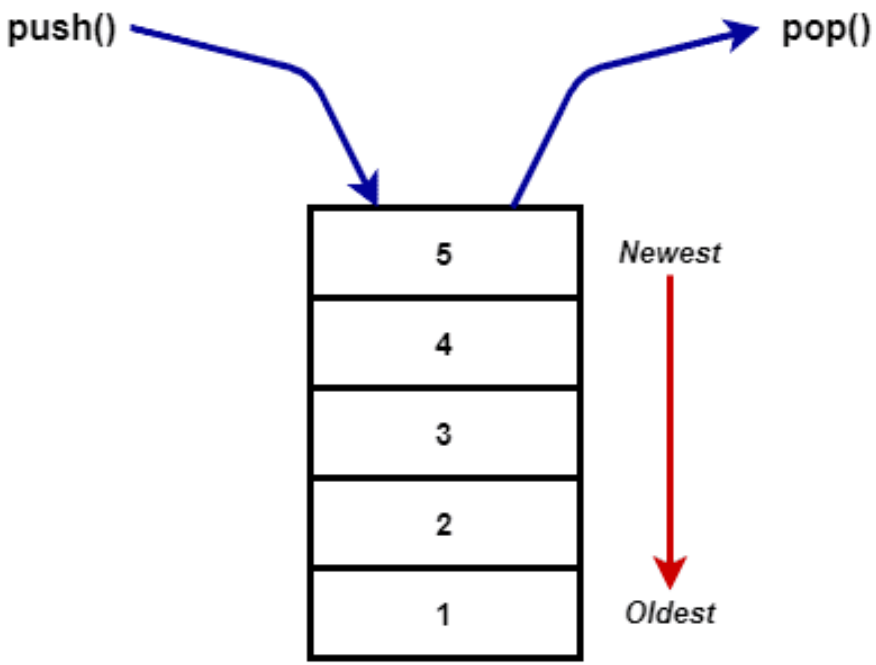
Stack

- 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.

- We can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
- This feature makes it **LIFO** data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first.

Cont.

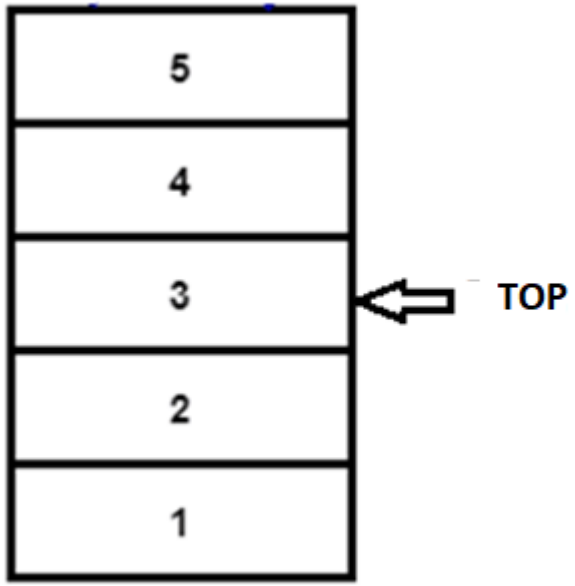
- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



- Stack can be implemented by means of **Array** and **Linked List**. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Data Members

- A template array.
- TOP index which indicates the top of the stack.



Basic Operations

1. **push()** – Pushing (storing) an element on the top of the stack.
2. **pop() –** Removing (accessing) an element from the top of the stack.
3. **peek()** – get the top data element of the stack, without removing it.
4. **isFull()** – check if stack is full.
5. **isEmpty()** – check if stack is empty.

Cont.

Push () Operation	Pop () Operation	isFull () Operation	isEmpty () Operation
Step 1 – Checks if the stack is full. **Step 2** – If the stack is full, produces an error and exit. **Step 3** – If the stack is not full, increments **TOP** to point next empty space. **Step 4** – Adds data element to the stack location, where top is pointing. **Step 5** – Returns success.	**Step 1** – Checks if the stack is empty. **Step 2** – If the stack is empty, produces an error and exit. **Step 3** – If the stack is not empty, accesses the data element at which **TOP** is pointing. **Step 4** – Decreases the value of top by 1. **Step 5** – Returns success.	**Step 1** – Checks if the **TOP** equals the size of the stack. **Step 2** – Return True if yes and return False if no.	**Step 1** – Checks if the **TOP** equals zero. **Step 2** – Return True if yes and return False if no.

Basic Operations

- https://www.youtube.com/watch?v=-KQpk-dIA8s

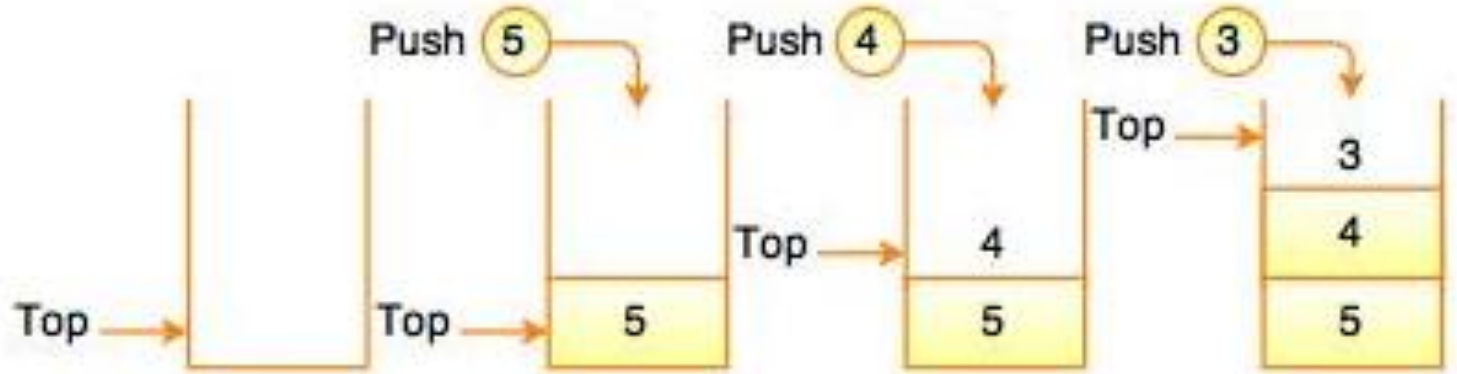


Fig. Insertion of Elements in a Stack

Static Size = 3
TOP = 0
Stack initially is empty
Push(5) ➔ Insert (5) then TOP++ ➔ TOP=1
Push(4) ➔ Insert (4) then TOP++ ➔ TOP=2
Push(3) ➔ Insert (3) then TOP++ ➔ TOP=3
Push(10) ➔ TOP = SIZE ➔ FULL

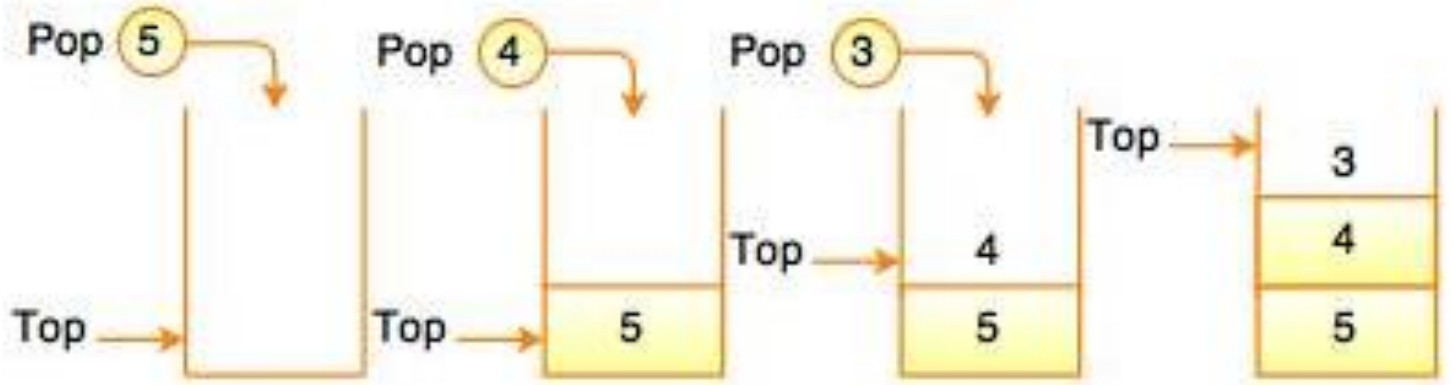
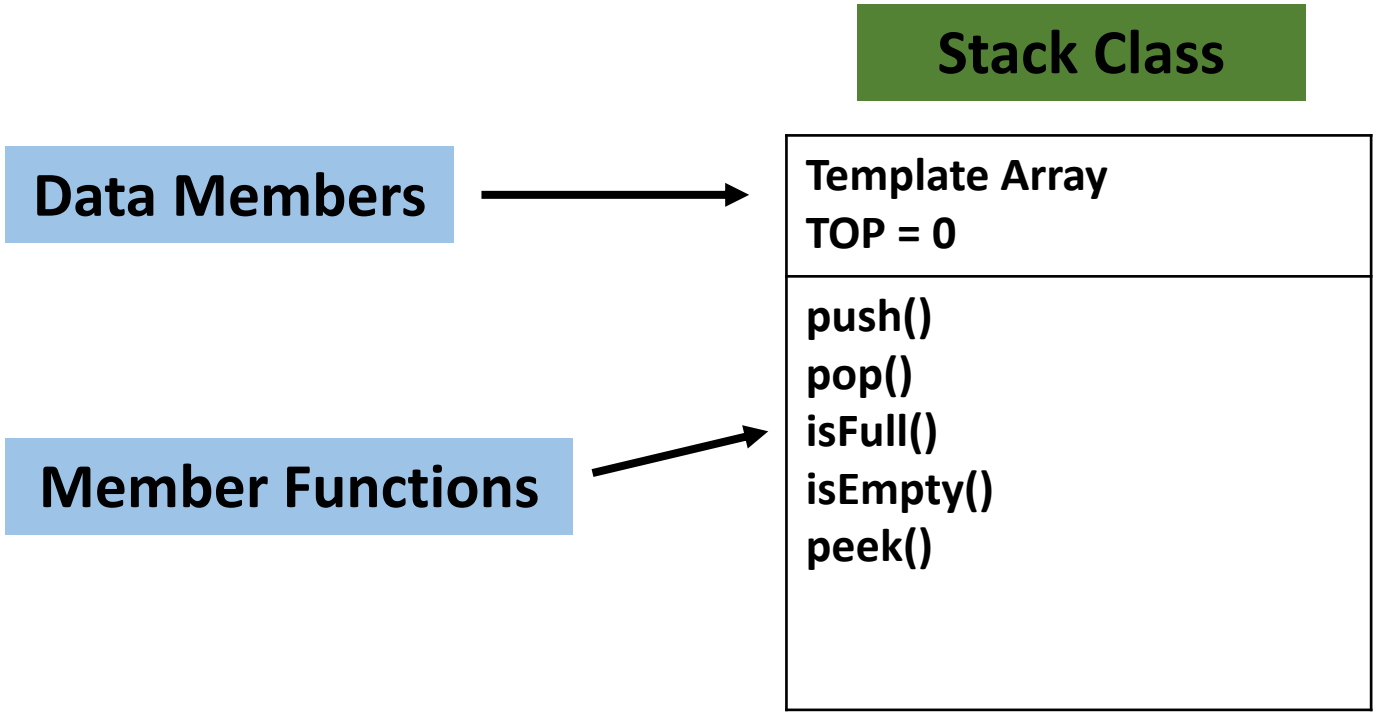


Fig. Deletion of Elements in a Stack

Static Size = 3
TOP = 3
Stack is full
Pop() ➔ TOP - - then return (3) ➔ TOP=2
Pop() ➔ TOP - - then return (4) ➔ TOP=1
Pop() ➔ TOP - - then return (5) ➔ TOP=0
Pop() ➔ TOP = 0➔ Empty

Stack Implementation using c++
Array based implementation



```
#include <iostream>
Using namespace std;
const int SIZE = 3;

template <class T>
class stack
{
 private:
 int top;
 T array[SIZE];
public:
 stack()
 {
 top = 0;
}

 bool isEmpty()
 {
 if(top==0)
 return true;
 else
 return false;
 }
```

```
 bool isFull()
 {
 if(top==SIZE)
 return true;
 else
 return false;
 }
 void push(T);
 T pop();
 T max();
};
template <class T>
void stack <T> ::push(T element)
 {
 if(isFull())
 {
 cout << "Stack is full.\n";
 }
 else
 {
 array[top] = element;
++top;
 }}
```

```
template <class T>
 T stack<T>::pop()
 {
 if(isEmpty())
{
 cout << "Stack is empty.\n";
 return 0; // return null on empty stack
 }
 else
 {
 top--;
 return array[top];
 }
 }
```

```
void main()
{
stack <char> s1;
stack <int> s2;

 s1.push('a');
 s1.push('b');
 s1.push('c');
 s2.push(10);
 s2.push(5);
 s2.push(4);
 cout<<s1.pop()<<endl;
 cout<<s1.pop()<<endl;
 cout<<s1.pop()<<endl;
 cout<<s2.pop()<<endl;
cout<<s2.pop()<<endl;
cout<<s2.pop()<<endl;
}
```

Stack Applications

1. Web browser
2. Static Memory Allocation
3. Printing strings in reverse order
4. Converting from decimal to binary
5. Polish Notations

Polish Notations

- The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, These notations are :
- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

There are three popular methods used for representation of an expression:

Infix	A + B	Operator between operands.
Prefix	+ AB	Operator before operands.
Postfix	AB +	Operator after operands.

- Infix to prefix
- (A + B) * (C + D) ➔ * + A B + C D
- Prefix to infix
- + * A B * C D ➔ A * B + C * D
- Infix to postfix
- (A + B) * (C + D) ➔ A B + C D + *

Converting from infix to postfix using stack

- 2 cases

1. With parenthesis
2. Without parenthesis

Algorithm for Infix to Postfix

Step 1: Consider the next element in the input.

Step 2: If it is operand, display it.

Step 3: If it is opening parenthesis, insert it on stack.

Step 4: If it is an operator, then

1. If stack is empty, insert operator on stack.
2. If the top of stack is opening parenthesis, insert the operator on stack
3. If it has higher priority than the top of stack, insert the operator on stack.
4. Else, delete the operator from the stack and display it, repeat Step 4.

Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.

Step 6: If there is more input, go to Step 1.

Step 7: If there is no more input, delete the remaining operators to output.

End