

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.

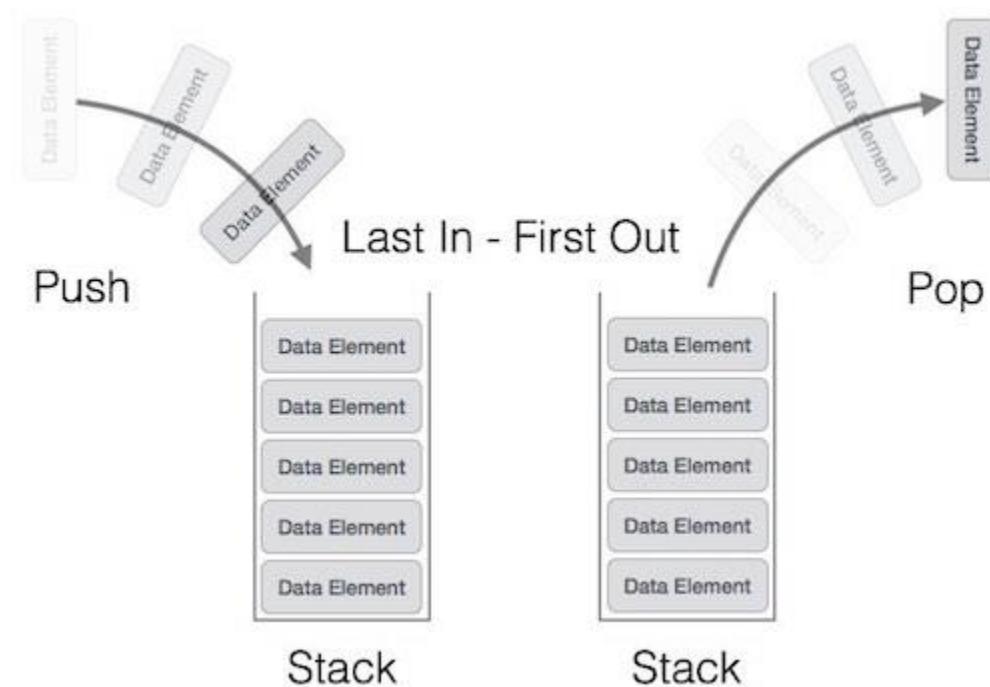


A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

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

Stack Representation

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, 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.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions –

peek()

Algorithm of peek() function –

```
begin procedure peek
    return stack[top]
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {
    return stack[top];
}
```

isfull()

Algorithm of isfull() function –

```

begin procedure isfull

    if top equals to MAXSIZE
        return true
    else
        return false
    endif

end procedure

```

Implementation of isfull() function in C programming language –

Example

```

bool isfull() {
    if(top == MAXSIZE)
        return true;
    else
        return false;
}

```

isempty()

Algorithm of isempty() function –

```

begin procedure isempty

    if top less than 1
        return true
    else
        return false
    endif

end procedure

```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

Example

```

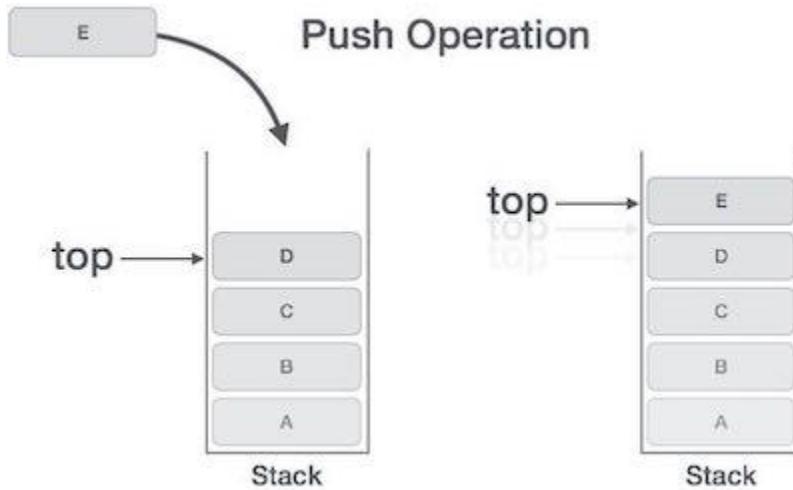
bool isempty() {
    if(top == -1)
        return true;
    else
        return false;
}

```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **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.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data
    if stack is full
        return null
    endif

    top ← top + 1
    stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

Example

```
void push(int data) {
    if(!isFull()) {
        top = top + 1;
```

```

        stack[top] = data;
    } else {
        printf("Could not insert data, Stack is full.\n");
    }
}

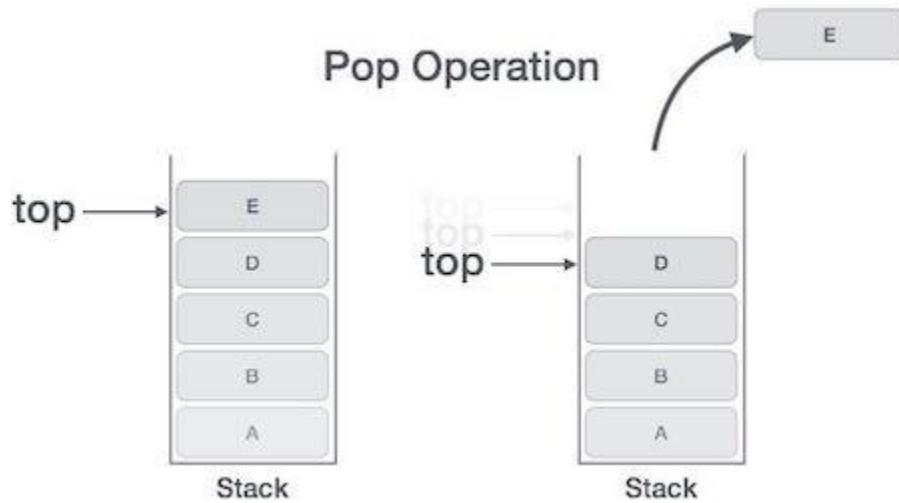
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- **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.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```

begin procedure pop: stack

if stack is empty
    return null
endif

data ← stack[top]
top ← top - 1

```

```

        return data
end procedure

```

Implementation of this algorithm in C, is as follows –

Example

```

int pop(int data) {

    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    } else {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}

```

Implementation of a Stack

```

#include <stdio.h>

int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {

    if(top == -1)
        return 1;
    else
        return 0;
}

int isfull() {

    if(top == MAXSIZE)
        return 1;
    else
        return 0;
}

int peek() {
    return stack[top];
}

int pop() {
    int data;

    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    } else {

```

```

        printf("Could not retrieve data, Stack is empty.\n");
    }
}

int push(int data) {

    if(!isfull()) {
        top = top + 1;
        stack[top] = data;
    } else {
        printf("Could not insert data, Stack is full.\n");
    }
}

int main() {
    // push items on to the stack
    push(3);
    push(5);
    push(9);
    push(1);
    push(12);
    push(15);

    printf("Element at top of the stack: %d\n" ,peek());
    printf("Elements: \n");

    // print stack data
    while(!isempty()) {
        int data = pop();
        printf("%d\n",data);
    }

    printf("Stack full: %s\n" , isfull()?"true":"false");
    printf("Stack empty: %s\n" , isempty()?"true":"false");

    return 0;
}

```

If we compile and run the above program, it will produce the following result –

Output

```

Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true

```

Application of Stacks

- (a) Stacks can be used for expression evaluation.
- (b) Stacks can be used to check parenthesis matching in an expression.
- (c) Stacks can be used for Conversion from one form of expression to another.
- (d) Stacks can be used for Memory Management.
- (e) Stack data structures are used in backtracking problems.

Infix	Prefix	Postfix
$a + b$	$+ b a$	$a b +$
$(a + b) * (c + d)$	$* + d c + b a$	$a b + c d + *$
$b * b - 4 * a * c$	$- * c * a 4 * b b$	$b b * 4 a * c * -$

Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

Algorithm

```
begin procedure peek
    return queue[front]
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {
    return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull

    if rear equals to MAXSIZE
        return true
    else
        return false
    endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
    if(rear == MAXSIZE - 1)
        return true;
    else
        return false;
}
```

isempty()

Algorithm of isempty() function –

Algorithm

```
begin procedure isempty

    if front is less than MIN OR front is greater than rear
        return true
    else
        return false
    endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

Example

```
bool isempty() {
    if(front < 0 || front > rear)
        return true;
    else
        return false;
}
```

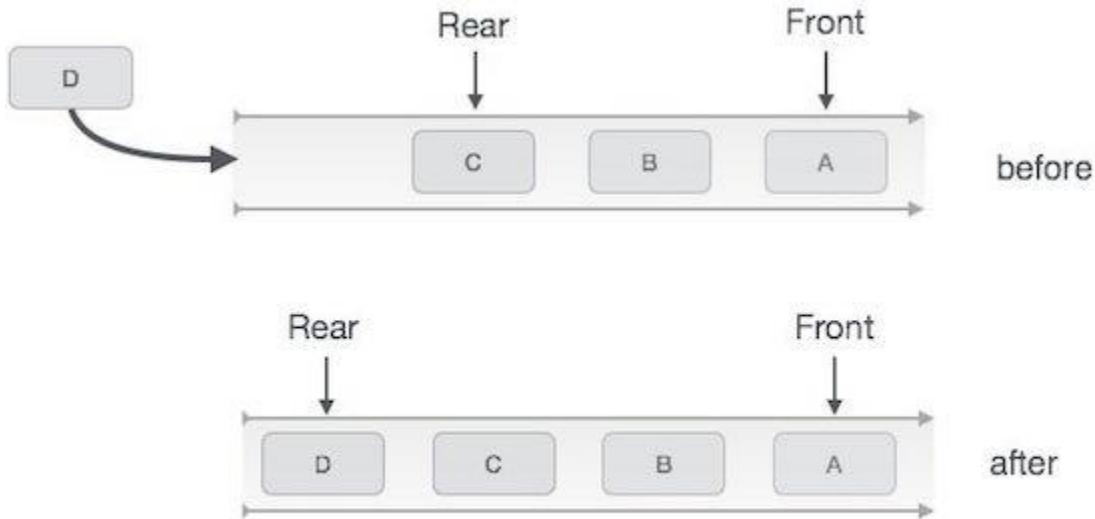
Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.

- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```

procedure enqueue(data)

    if queue is full
        return overflow
    endif

    rear ← rear + 1
    queue[rear] ← data
    return true

end procedure

```

Implementation of enqueue() in C programming language –

Example

```

int enqueue(int data)
    if(isfull())
        return 0;

    rear = rear + 1;

```

```

queue[rear] = data;

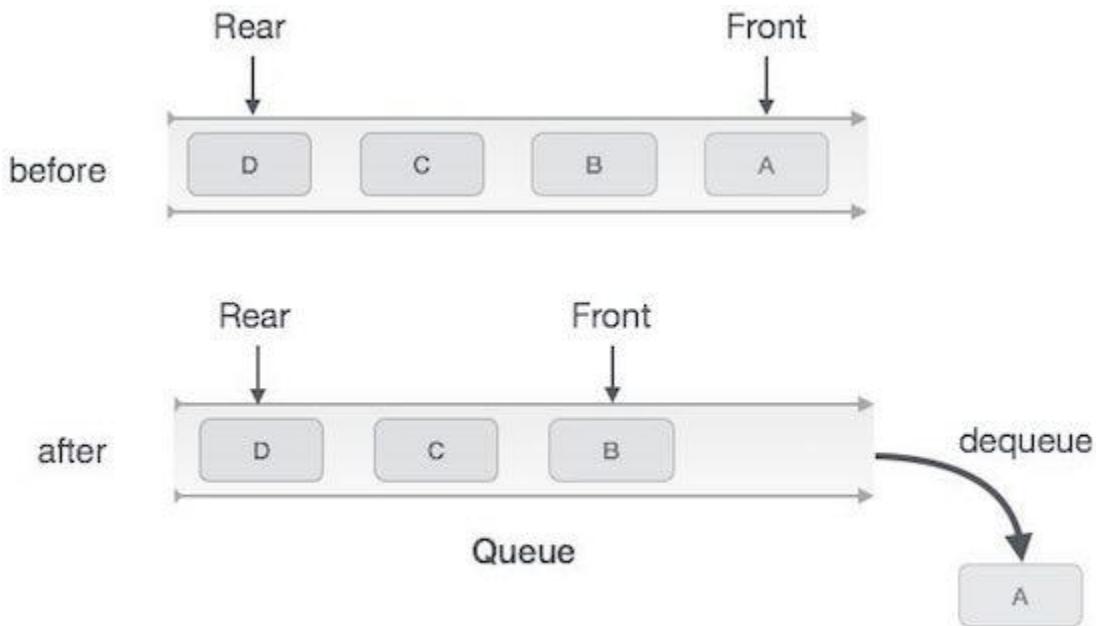
return 1;
end procedure

```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue operation

```

procedure dequeue

if queue is empty
    return underflow
end if

data = queue[front]
front ← front + 1
return true

end procedure

```

Implementation of dequeue() in C programming language –

Example

```
int dequeue() {  
    if(isempty())  
        return 0;  
  
    int data = queue[front];  
    front = front + 1;  
  
    return data;  
}
```

Applications of Queue Data Structure

Queue is used when things don't have to be processed immediately, but have to be processed in **First In First Out**.

- 1) When a resource is shared among multiple consumers.** Examples include CPU scheduling, Disk Scheduling.
- 2) When data is transferred asynchronously** (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
- 3) In Operating systems:**
 - a)** Semaphores
 - b)** FCFS (first come first serve) scheduling, example: FIFO queue
 - c)** Spooling in printers
 - d)** Buffer for devices like keyboard
- 4) In Networks:**
 - a)** Queues in routers/ switches
 - b)** Mail Queues
- 5) Variations:** (Deque, Priority Queue, Doubly Ended Priority Queue)