Linear Data Structure

- Stacks
- Queues

Acknowlege: www.tutorialspoint.com

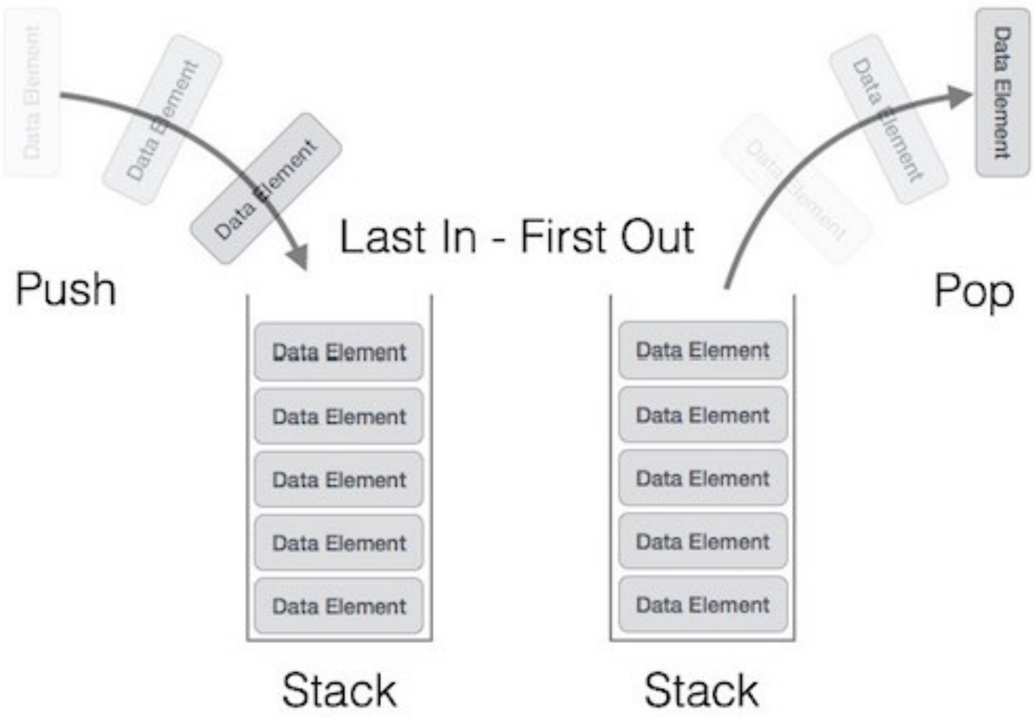
Stacks and Queues

Acknowlege: www.tutorialspoint.com

Stacks: A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top. A stack is a recursive data structure. Here is a structural definition of a Stack: a stack is either empty or it consists of a top and the rest which is a stack;

Stack Representation

The following diagram depicts a stack and its operations –



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 –

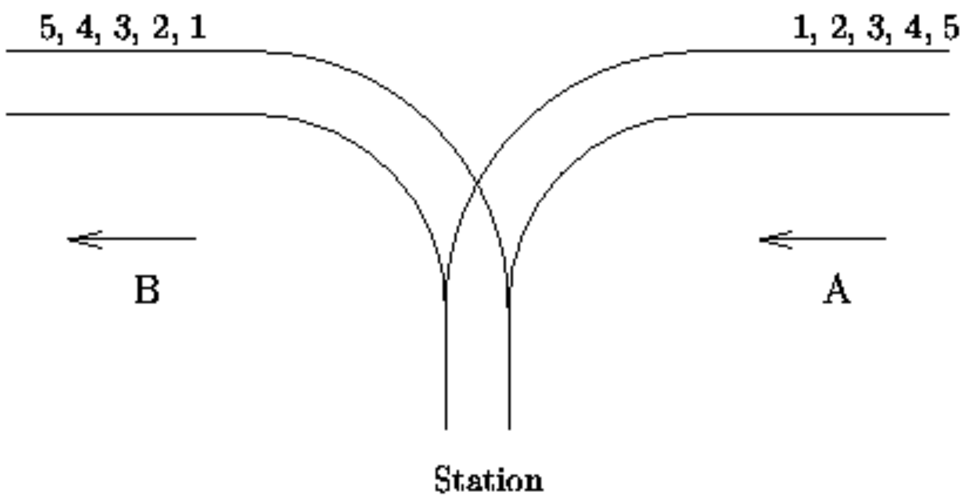
- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

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.

Definition:

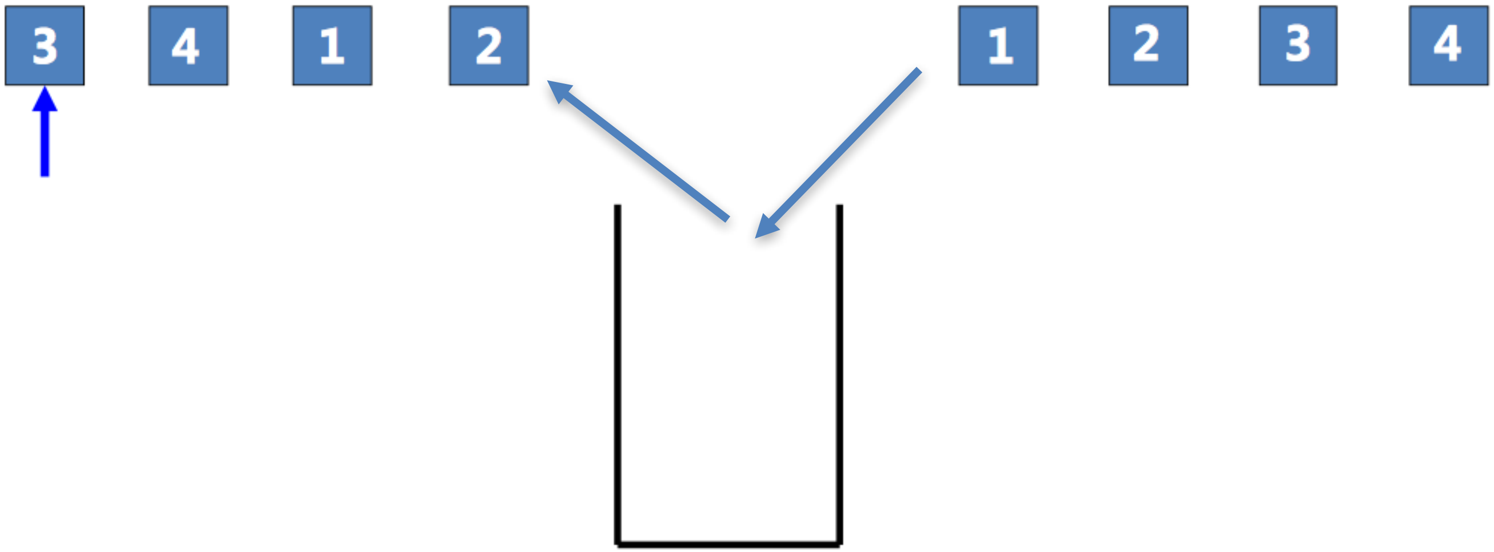
```
template <class T>
class Stack {
    public:
        void clear();           //Clear the items in stack
        bool push(const T item); //push an item on stack

        bool pop(T& item);      //remove an element from the stack
        bool peek(T& item);      //get the top data element of the stack, without removing it
        bool isEmpty();         //check if the stack is empty
        bool isFull();          //check if the stack if full
}
```

Example: There is a famous railway station in PopPush City. Country there is incredibly hilly. The station was built in last century. Unfortunately, funds were extremely limited that time. It was possible to establish only a surface track. Moreover, it turned out that the station could be only a dead-end one (see picture) and due to lack of available space it could have only one track.



Cite: http://poj.org/problem?id=1363

Is this output order legal?



Implementation of Stack:

Array-based Stack

Linked Stack

Array-based Stack:

```
template <class T> class arrStack: public stack <T>{
    prviate:                   //sequential data storage
        int mSize;             // the maxsize of elements stored in the stack
        int top;               // the position of the stack
        T *st;                 // data array
    public:
        arrStack(int size)  // Create a stack with a fixed size
            {
             mSize = size;
             top = -1;
             st = new T[mSize];
            }
        arrStack()             // Create a stack with one element
            {
              top = -1;
            }
        ~arrStack() {delete [] st;}
}
```

Overflow and Underflow

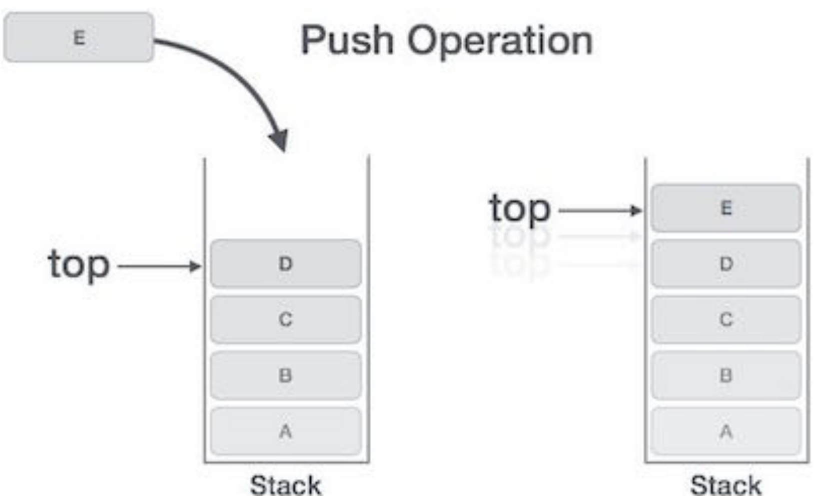
Overflow: Push an element while the stack is full

Underflow: Pop an element while the stack is empty

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.



Push Function:

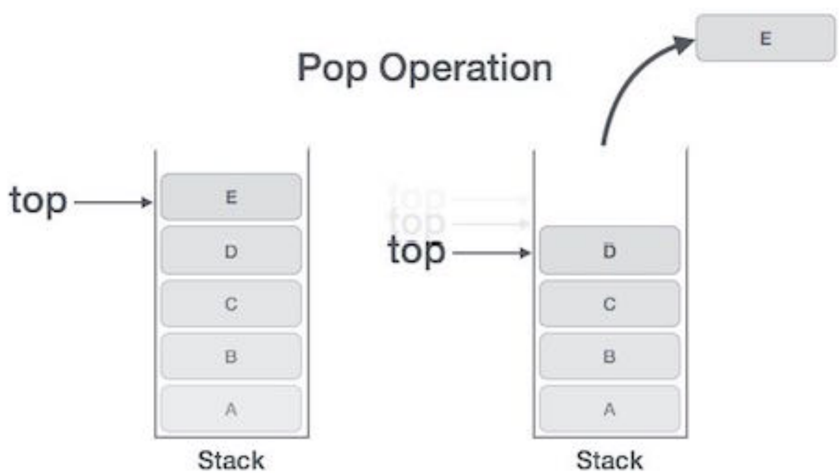
```
bool arrStack<T>::push(const T item)
{
    if(top == mSize - 1)
    {
        cout<<"Stack is full"<<endl;  // the stack is full
        return false;
    }
    else
    {
        st[++top] = item; // push a new element to stack
        return true;
    }
}
```

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 –

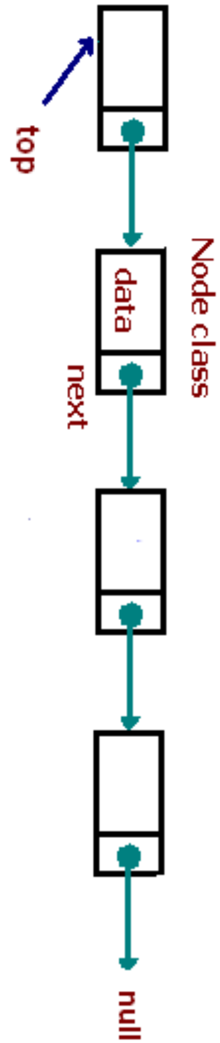
- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Pop Function:

```
bool arrStack<T>::pop(T& item)
{
    if(top == -1)
    {
        cout<<"Stack is empty"<<endl;
        return false;
    }
    else
    {
        item = st[top--]; //return the top element on the stack
        return true;
    }
}
```

Linked Stack:



Data Representation

```
template <class T>
class lnkStack
{
 private:
    Node<T> *top;    // The node pointer to the top node
    int size;        // the number of elements in the stack
 public:
    lnkStack()       // create a stack
        {
            top = NULL;
            size =0;
        }
    ~lnkStack()      // delete a stack
        {
           clear(); // clear the elements on the stack
        }
}
```

Push operation:

```
bool lnkStack<T>::push(const T item)
{
    Node<T> *newND = new Node<T>(item, top); // create a new node on the current top position
    top = newND; // assign the new node to be the top node
    size++;      // increase the size of the stack
    return true;
}
```

Pop an element:

```
bool lnkStack<T>::pop(T& item)
{
    Node<T> *tmp;
    if(size == 0)
    {
        cout<< "The stack is empty, no item pop out"<<endl;
        return false;
    }
    item = top->data; //pop out the top item on the stack
    tmp = top->next;  //assign new top item
    delete top;
    top = tmp;
    size --;
    return true;
}
```

Queues

Queues 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 –

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 enqueing (or storing) data in the queue we take help of **rear** pointer.

Definition:

```
template <class T>
class Queue
{
    public:
        bool enQueue(const T item);
        bool deQueue(T& item);
        bool getFront(T& item);
        bool isEmpty();
        bool isFull();
};
```

Array-based Queue and Linked Queue

Array-based Queue

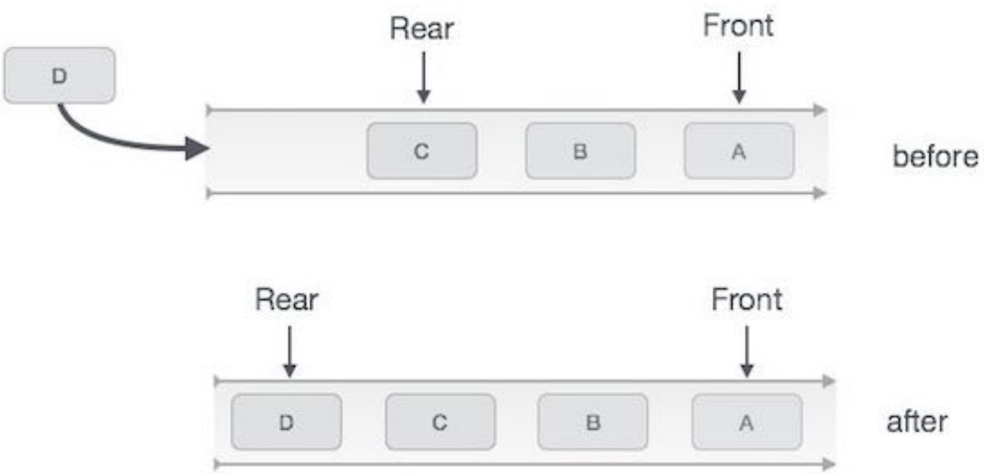
```
template <class T>
class arrQueue
{
    private:
        int mSize; //The size of queue
        int front; //The index of front element
        int rear;  //The index of rear element
        T* qu;       // The data array
    public:
        arrQueue(int size); // create a queue
        ~arrQueue();        // delete a queue
}
```

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.



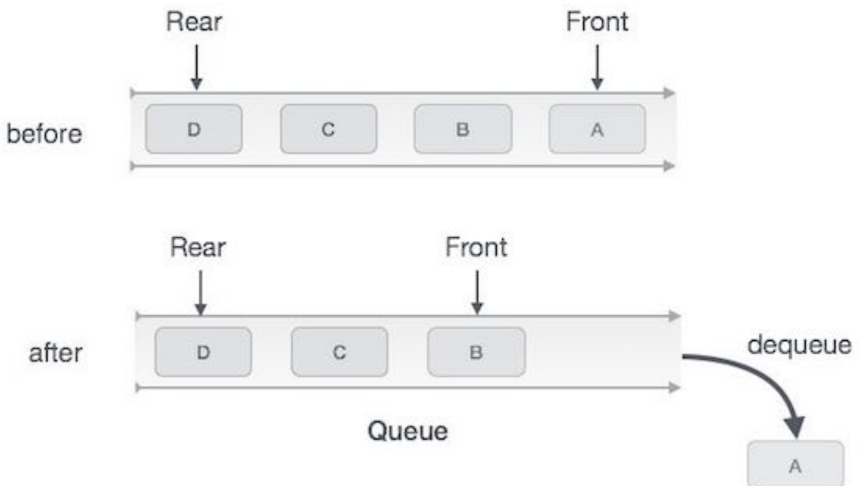
Queue Enqueue

```
bool arrQueue::enQueue(const T item)
{
    if(isFull())          //check if the queue is full
        {
            return false;
        }
    rear = rear + 1;
    qu[rear] = item;
    return true;
}
```

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.



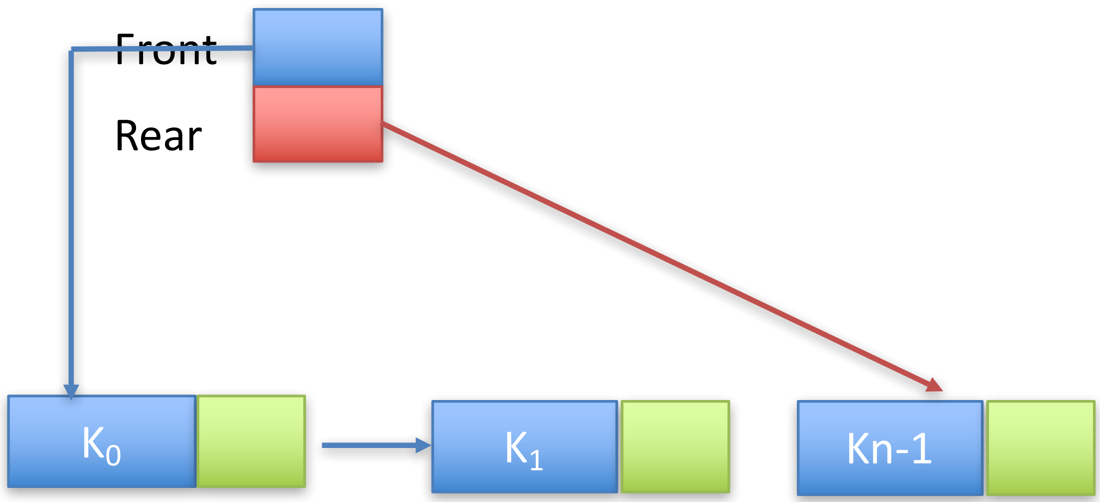
Queue Dequeue

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


bool arrQueue:deQueue(T& item)
{
    if(isempty())
    {
        return false;
    }

    item = qu[front];
    front = front + 1;
}
```

Linked Queue

How to implement the linked queue?

Data Representation



Definition:

```
template <class T>
class lnkQueue
{
    private:
        int size;           // The number of elements in the queue
        Node<T>* front;     // The front node
        Node<T>* rear;      // The rear node
    public:
        lnkQueue(int size); //creat a linked queue
        ~lnkQueue();          // delete the queue
}
```

enQueue operation:

```
bool enQueue(const T item)
{
    if(rear ==NULL)
    {
        front = rear = new Node<T> (item, NULL);
    }
    else
    {
        rear->next = new Node<T>(item, NULL);
        rear = rear->next;
    }
    size++;
    return true;
}
```

deQueue operation:

```
bool deQueue(T& item)
{                               //return the front item and delete it
    Node<T> *tmp;
    item = front->data;
    tmp = front;
    front = front->next;
    delete tmp;
    if(front = NULL)
        rear = NULL;
    size--;
    return true;

}
```