VTC Academy
Think Ahead, Beyond Boundaries

DATA STRUCTURES AND ALGORITHMS

LESSON 3

Stacks and Queues

CONTENT

VTC Academy
Think Ahead, Beyond Boundaries

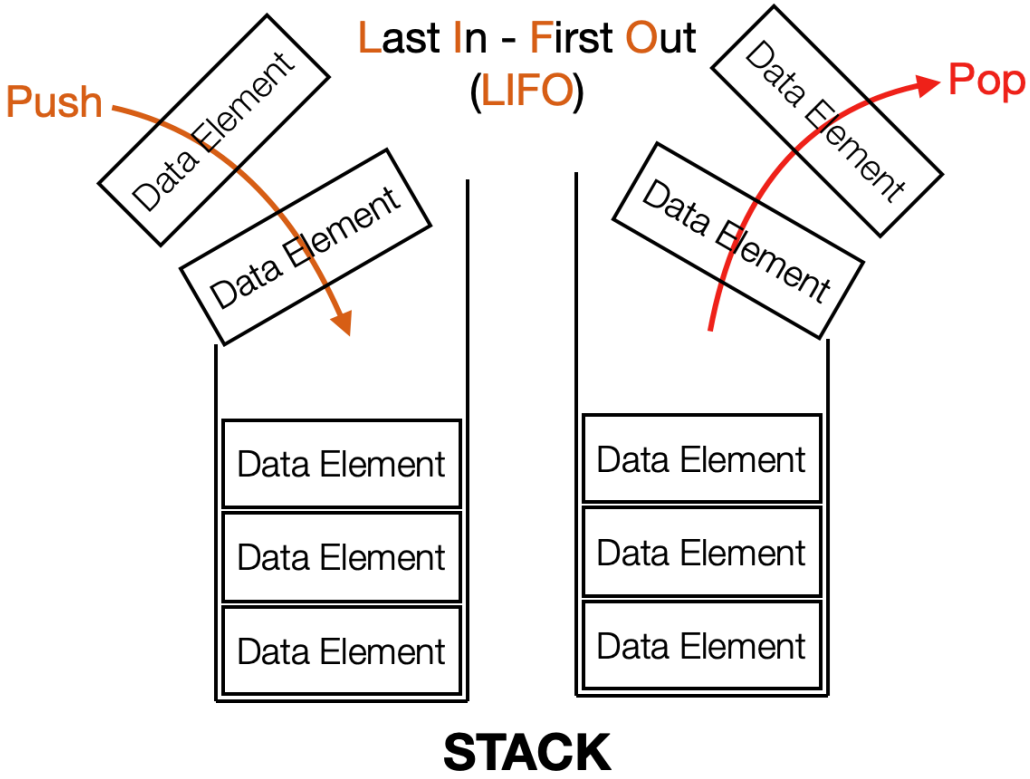
Introduction to Stack

VTC Academy
Think Ahead, Beyond Boundaries

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

Stack Representation

- 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.



Stack Representation

VTC Academy
Think Ahead, Beyond Boundaries

```
//Stack Using Array
struct Stack{
    int data[50];
    int max; //max = 50
    int top;
};

//Stack Using Pointer
struct Stack{
    int *data;
    int max;
    int top;
};

//Stack Using Linked List
struct Stack{
    int data;
    struct Stack *next
};
```

Basic Operations

VTC Academy
Think Ahead, Beyond Boundaries

- 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.

Basic Operations

VTC Academy
Think Ahead, Beyond Boundaries

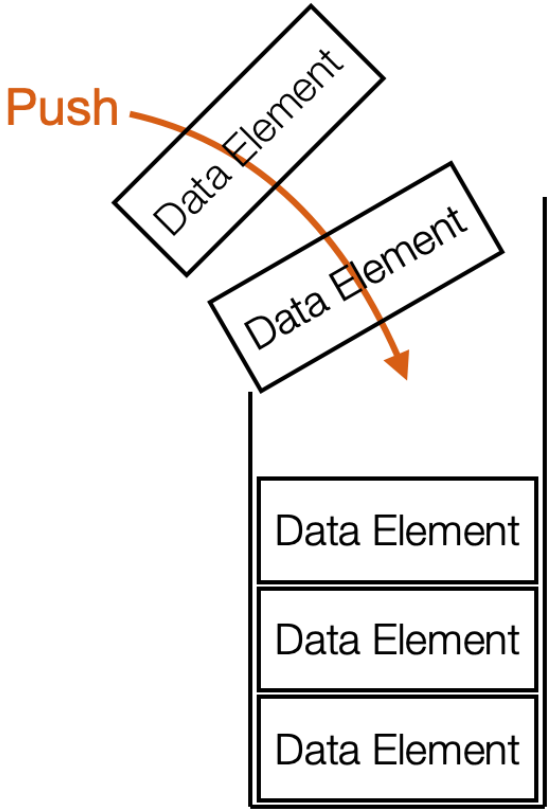
- When data is PUSHed onto stack, to use a stack efficiently, we need to check the status of stack as well.
- For the same purpose, the following functionality is added to stacks –
  - peek() – get the top data element of the stack, without removing it.
  - isFull() – check if stack is full.
  - isEmpty() – check if stack is empty.

Push Operation

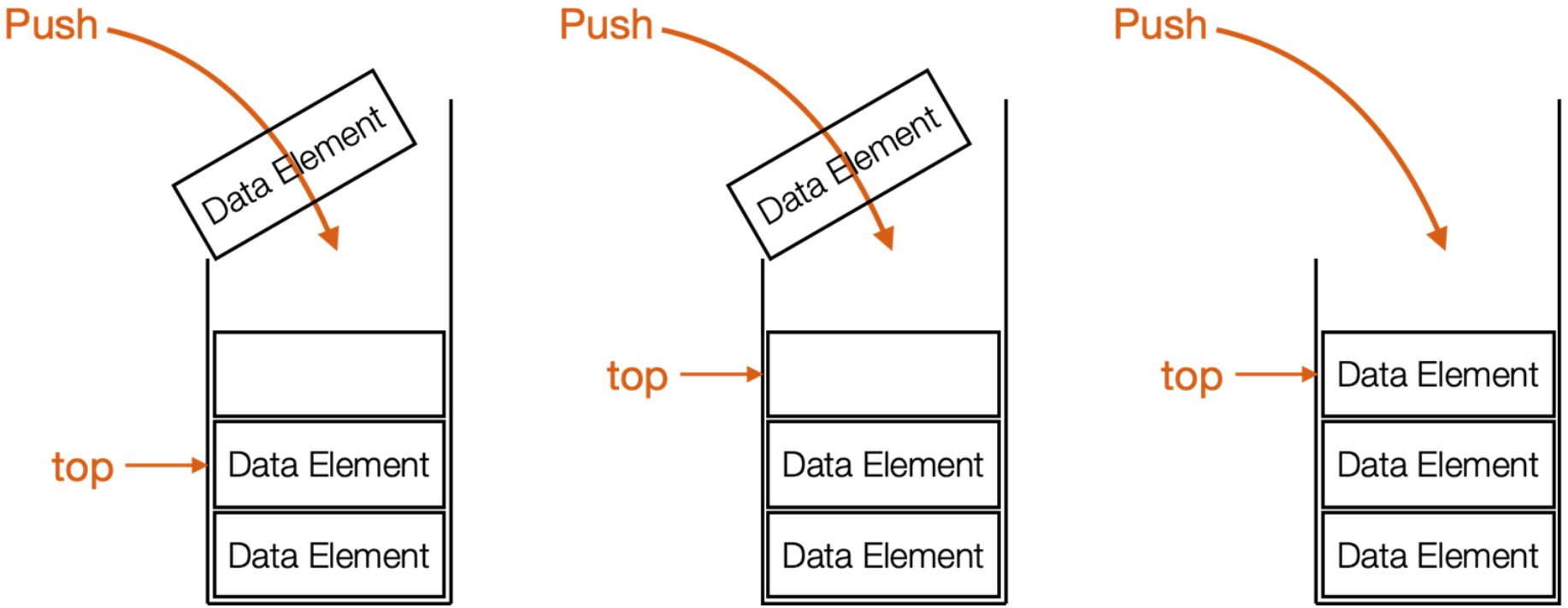
VTC Academy
Think Ahead, Beyond Boundaries

- 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 Operation



Push Operation



Push Operation

VTC Academy
Think Ahead, Beyond Boundaries

```
bool push(Stack* stack, int value){
    //Step 1 - Checks if the stack is full
    if(stack->top == stack->max){
        //Step 2 - If the stack is full, produces an error and exit
        return false;
    }
    //Step 3 - If the stack is not full, increments top to point next empty
space
    stack->top += 1;
    //Step 4 - Adds data element to the stack location, where top is pointing
    stack->data[stack->top] = value;
    //Step 5 - Returns success
    return true;
}
```

Pop Operation

VTC Academy
Think Ahead, Beyond Boundaries

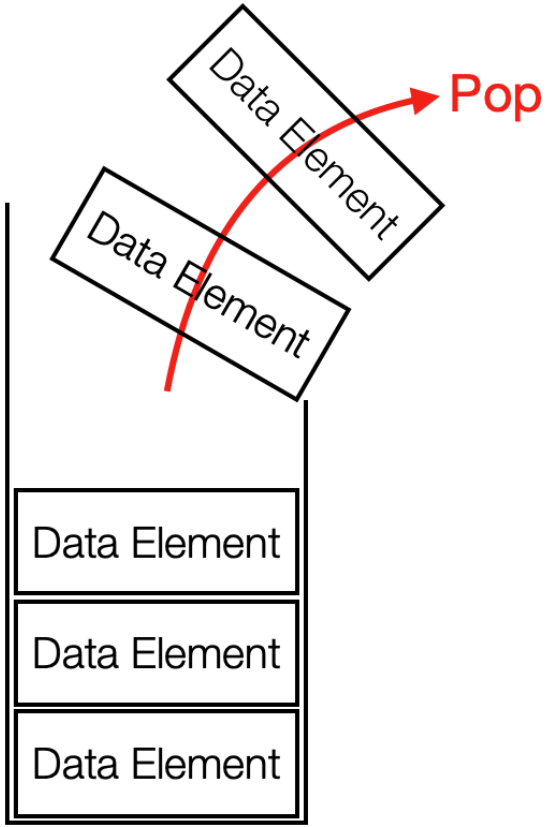
- Accessing the content while removing it from the stack, is known as a Pop Operation.
- In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.
- But in linked-list implementation,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

Pop Operation

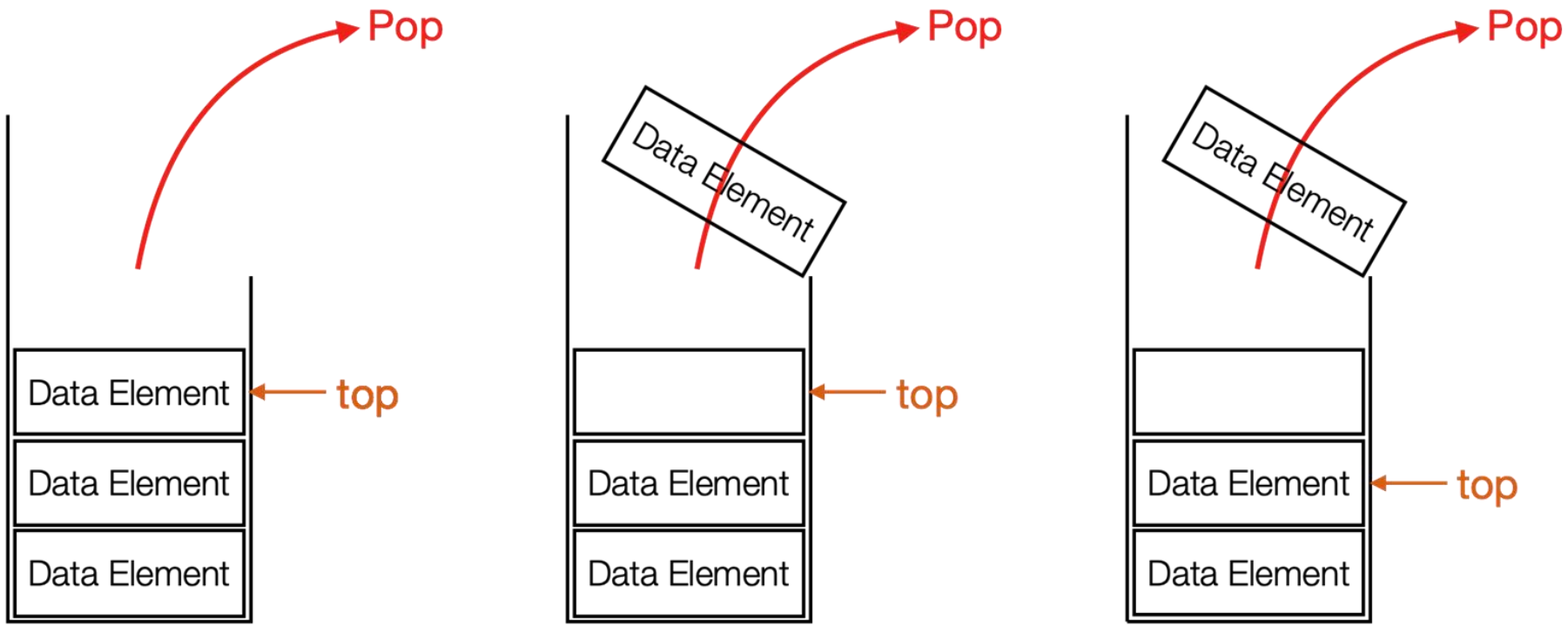
VTC Academy
Think Ahead, Beyond Boundaries

- 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 Operation



Pop Operation



Pop Operation

VTC Academy
Think Ahead, Beyond Boundaries

```
bool pop(Stack* stack, int* topValue){
    //Step 1 - Checks if the stack is empty.
    if(stack->top < 0){
        //Step 2 - If the stack is empty, produces an error and exit.
        return false;
    }
    //Step 3 - If the stack is not empty, accesses the data element at which top
is pointing.
    *topValue = stack->data[stack->top];
    //Step 4 - Decreases the value of top by 1
    stack->top -= 1;
    //Step 5 - Returns success.
    return true;
}
```

Other Pop Operations

VTC Academy
Think Ahead, Beyond Boundaries

```
bool initStack(Stack* stack, int maxElement){
    stack->data = (int*)malloc(sizeof(int)*maxElement);
    stack->max = maxElement;
    stack->top = -1;
    return true;
}
bool peek(Stack stack, int* topValue){
    if(stack.top < 0){
        return false;
    }
    *topValue = stack.data[stack.top];
    return true;
}
bool isFull(Stack stack){
    return stack.top == stack.max;
}
bool isEmpty(Stack stack){
    return stack.top<0;
}
```

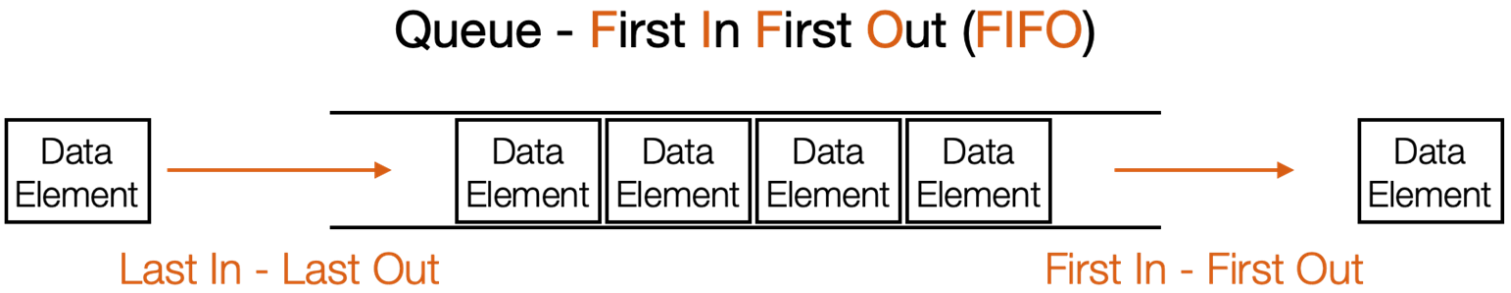
Introduction to Queue

VTC Academy
Think Ahead, Beyond Boundaries

- Queue is an Abstract Data Type, 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.

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.

Queue Representation

VTC Academy
Think Ahead, Beyond Boundaries

- Normal presentation:

```
struct Queue{
    int *data;
    int maxSize;
    int front;
    int rear;
};
```

- With capacity presentation:

```
struct Queue {
    int front;
    int rear;
    int size;
    unsigned capacity;
    int *array;
};
```

Queue Basic Operations

VTC Academy
Think Ahead, Beyond Boundaries

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.
  - 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.
  - 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.

Queue Basic Operations

VTC Academy
Think Ahead, Beyond Boundaries

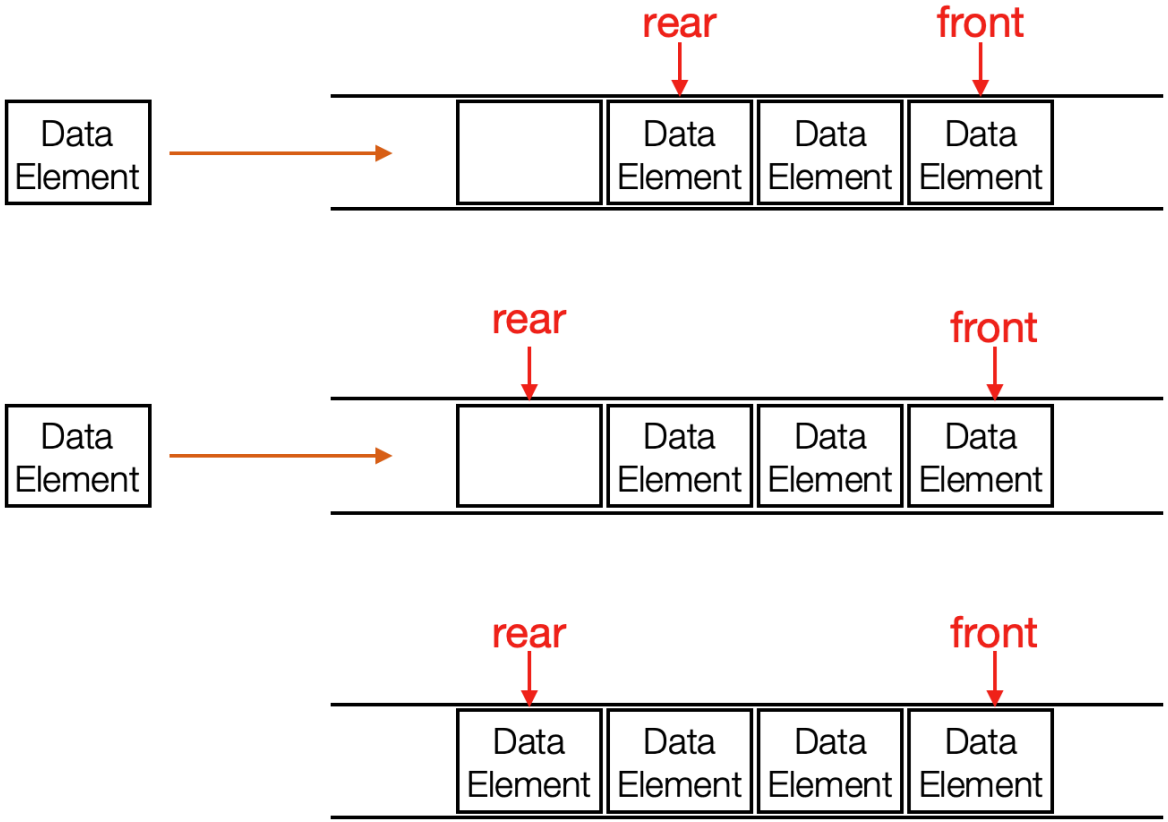
- In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Enqueue Operation

VTC Academy
Think Ahead, Beyond Boundaries

- 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

Enqueue Operation



Enqueue Operation

VTC Academy
Think Ahead, Beyond Boundaries

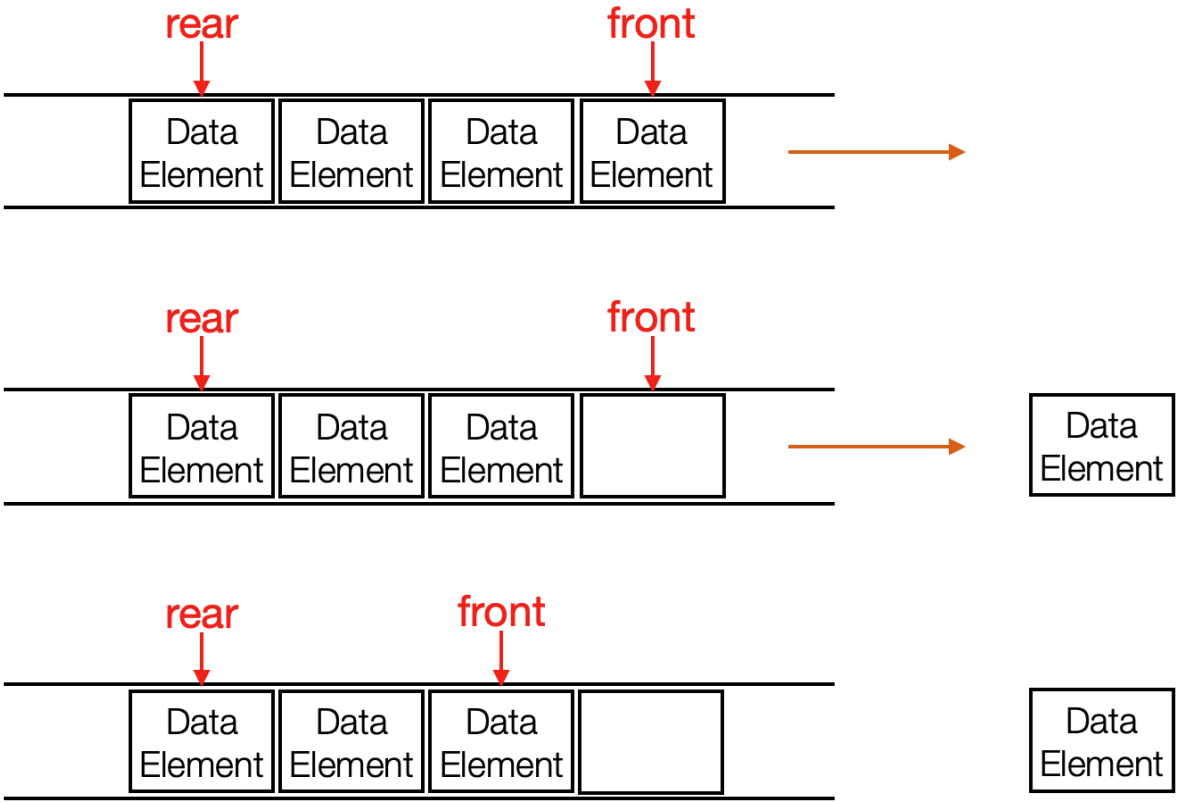
```
bool enqueue(struct Queue* queue, int rearValue){
    //Step 1 - Check if the queue is full
    if(queue->rear==queue->maxSize-1){
        if(queue->front==0){
            //Step 2 - If the queue is full, produce overflow error and exit
            return false;
        }else if(queue->front>0){
            //rear is last element of array but front not is first element of
array -> move data to head of array
            for(int i=0, j=queue->front; j<=queue->rear; i++, j++){
                queue->data[i] = queue->data[j];
            }
            queue->rear -= queue->front;
            queue->front = 0;
        }
    }
    //Step 3 - If the queue is not full, increment rear pointer to point the
next empty space
    queue->rear += 1;
    //Step 4 - Add data element to the queue location, where the rear is
pointing
    queue->data[queue->rear] = rearValue;
    //Step 5 - return success
    return true;
}
```

Dequeue Operation

VTC Academy
Think Ahead, Beyond Boundaries

- 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

Dequeue Operation



Dequeue Operation

VTC Academy
Think Ahead, Beyond Boundaries

```
bool dequeue(struct Queue *queue, int *frontValue){
    //Step 1 - Check if the queue is empty
    if(queue->rear<queue->front){
        //Step 2 - If the queue is empty, produce underflow error and exit.
        return false;
    }
    //Step 3 - If the queue is not empty, access the data where front is
pointing
    *frontValue = queue->data[queue->front];
    //Step 4 - Increment front pointer to point to the next available data
element
    queue->front += 1;
    //Step 5 - Return success
    return true;
}
```

Other Operations of Queue

VTC Academy
Think Ahead, Beyond Boundaries

```
bool initQueue(struct Queue* queue, int maxSize){
    queue->maxSize = maxSize;
    queue->data = (int*)malloc(maxSize*sizeof(int));
    queue->front = 0;
    queue->rear = -1;
    return true;
}
bool peek(struct Queue queue, int *frontValue){
    if(isEmpty(queue)){
        return false;
    }
    *frontValue = queue.data[queue.front];
    return true;
}
bool isFull(struct Queue queue){
    return queue.rear == queue.maxSize-1;
}
bool isEmpty(struct Queue queue){
    return queue.rear < queue.front;
}
```

Difference between Stack and Queue

VTC Academy
Think Ahead, Beyond Boundaries

Stacks	Queues
Stacks are based on the LIFO principle	Queues are based on the FIFO principle
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer - first element inserted in the list, and the rear pointer - the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

Summary

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.
- This feature makes it LIFO data structure. LIFO stands for Last-in-first-out.
- In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.
- Queue is an Abstract Data Type, 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.

VTC Academy
Think Ahead, Beyond Boundaries