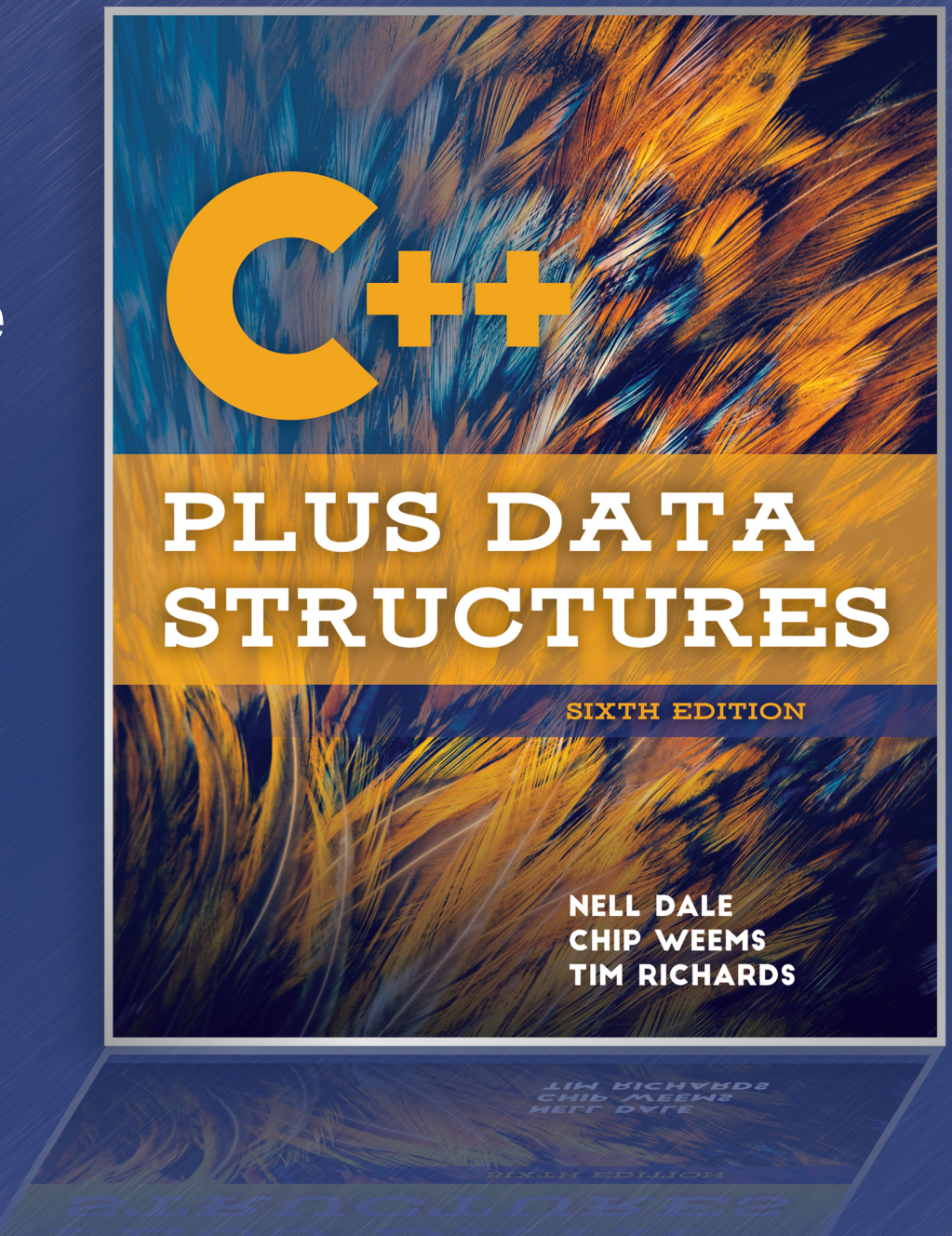
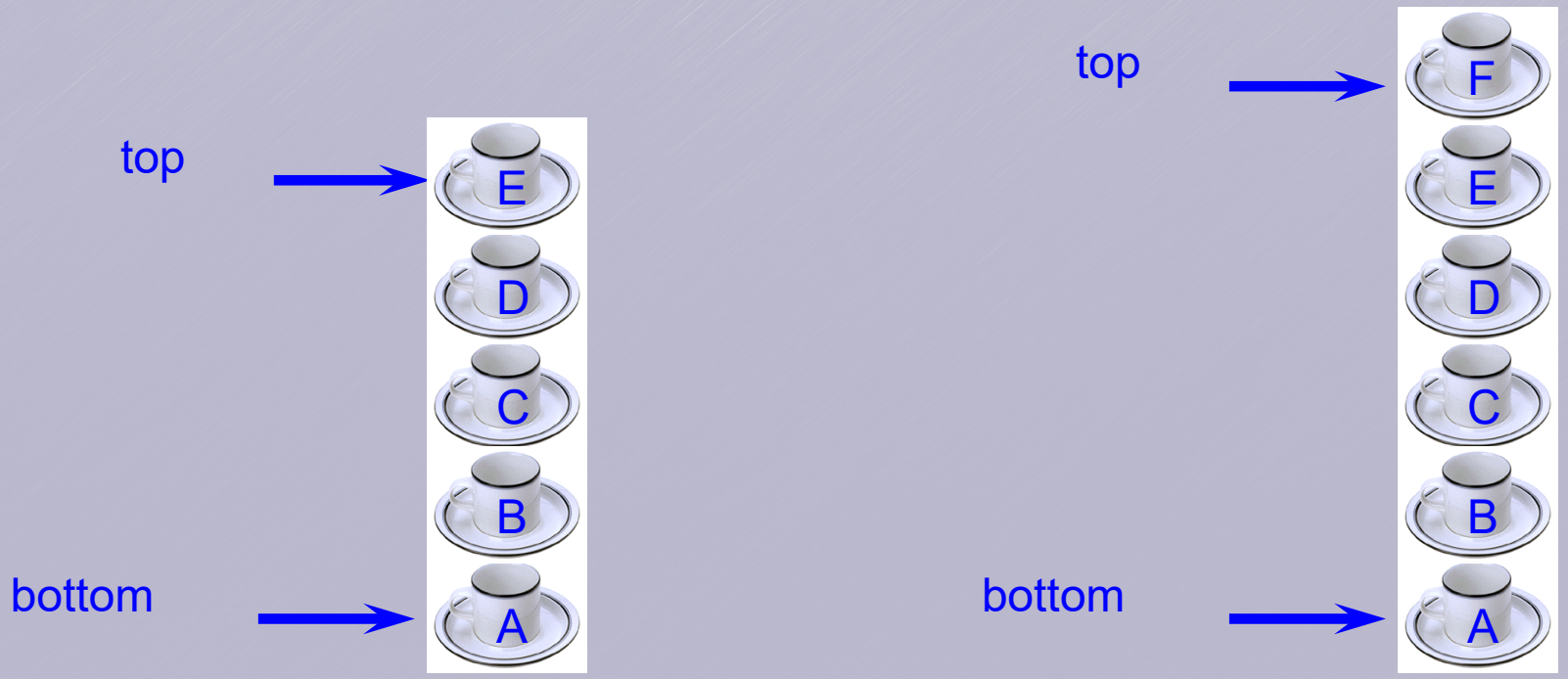
Stack and Queue



Stacks: Logical Level

- **Stack:** An ADT where items are added and removed only from the top of the structure; this behavior is called **LIFO** (**L**ast **I**n, **F**irst **O**ut)
- Items are “ordered” by when they were added to the stack
- Like lists, stacks store homogeneous items

Stack Of Cups



- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO list.

Stacks: Application Level

Stacks have many applications in software engineering. Some examples:

- Tracking the function calls in a program
- Performing syntax analysis on a program
- Traversing structures like trees and graphs
- Some programming languages are entirely stack-based, such as Forth

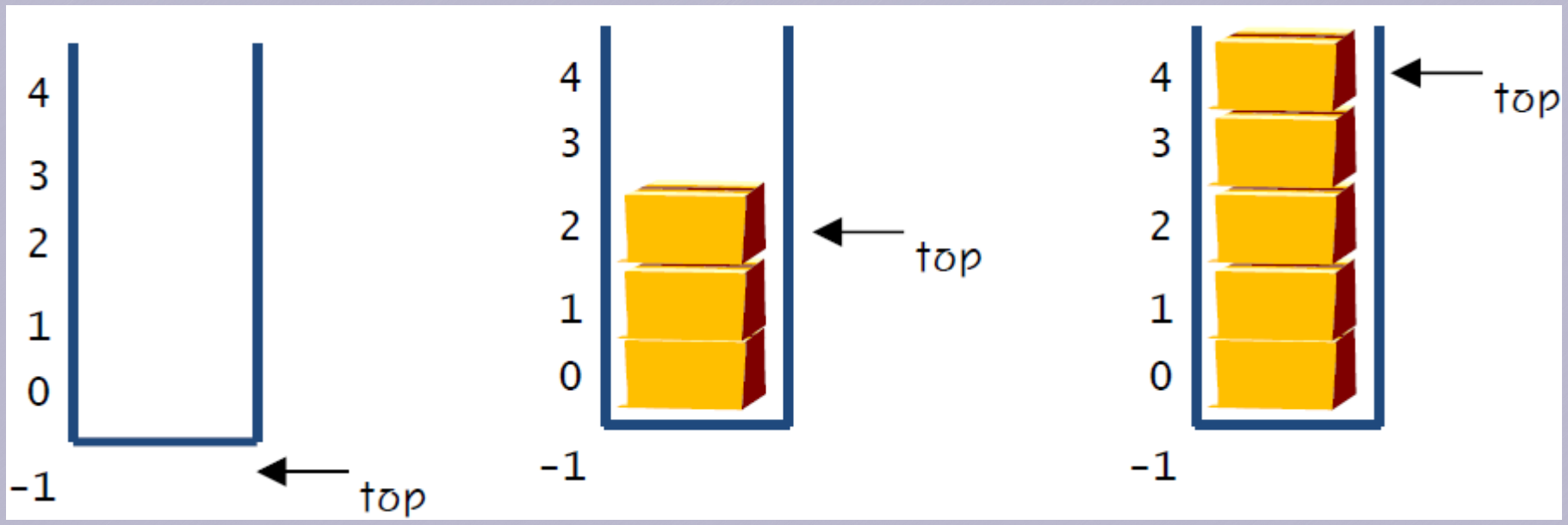
Stack Operations

- **Push:** Adds an item to the top of the stack
- **Pop:** Removes the top item from the stack
- **Top:** Returns the item at the top of the stack but does not remove it
- **IsEmpty:** Returns true if the stack has no items
- **IsFull:** Stacks are logically unbounded, but implementations may be bounded

Stack in an Array

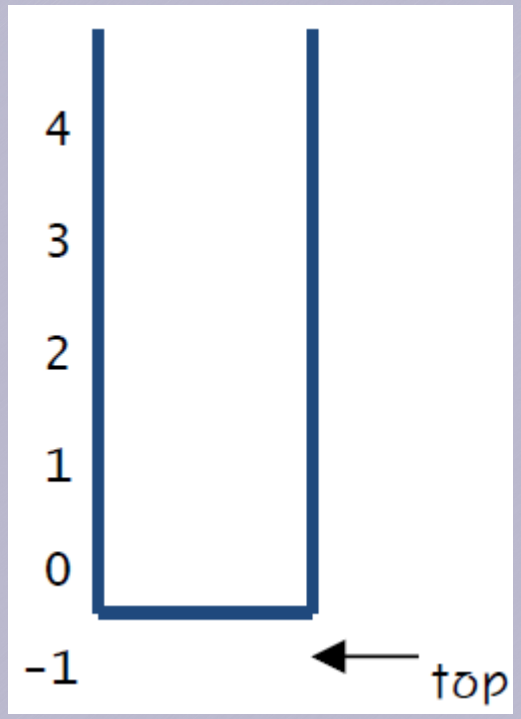
```
#define MAX_STACK 5

class IntStack {
      protected:
            int stack[MAX_STACK] = {0,};
            int top = -1;

      public:
            void push(int x);
            int pop();
            int top();
            int isEmpty();
            int isFull();
};
```

Stack in an Array



Empty **Full**

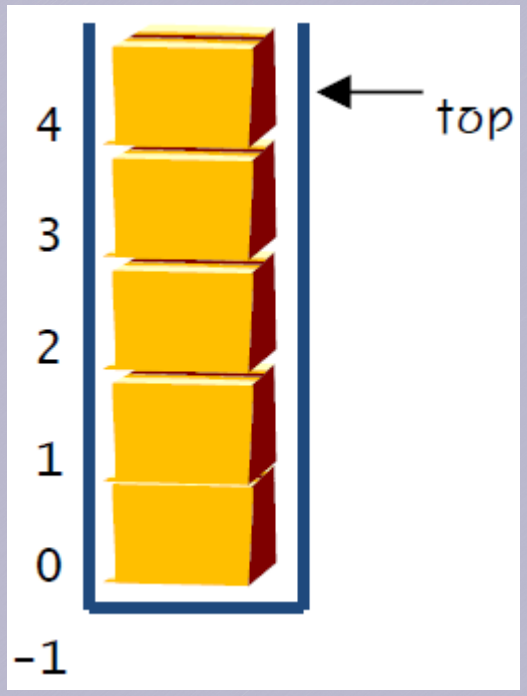
Stack in an Array



Empty

```
int isEmpty(){
        return (top == -1);
}
```

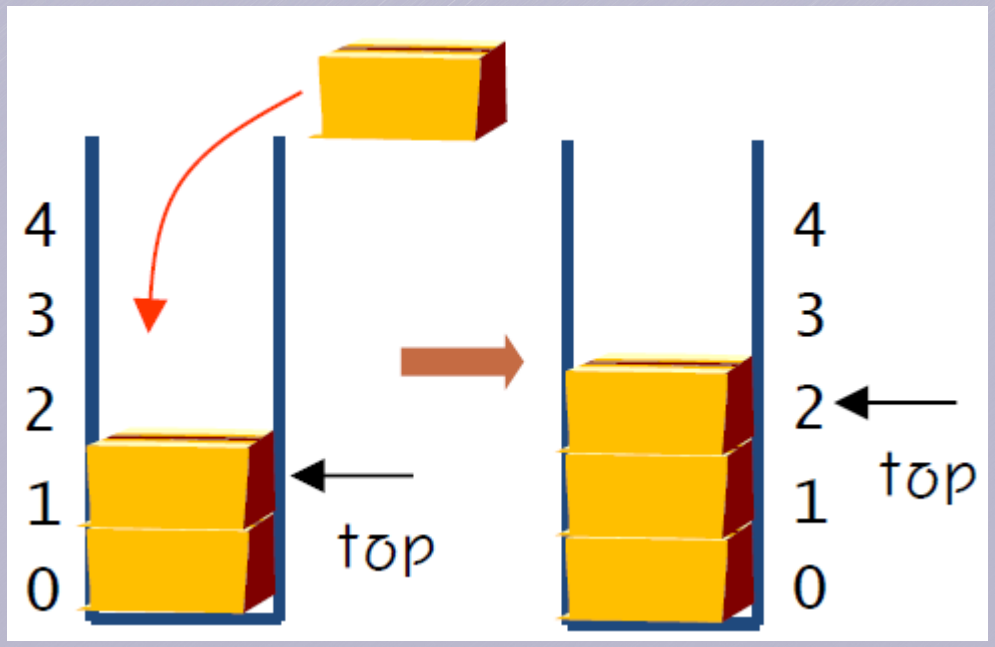
Stack in an Array



Full

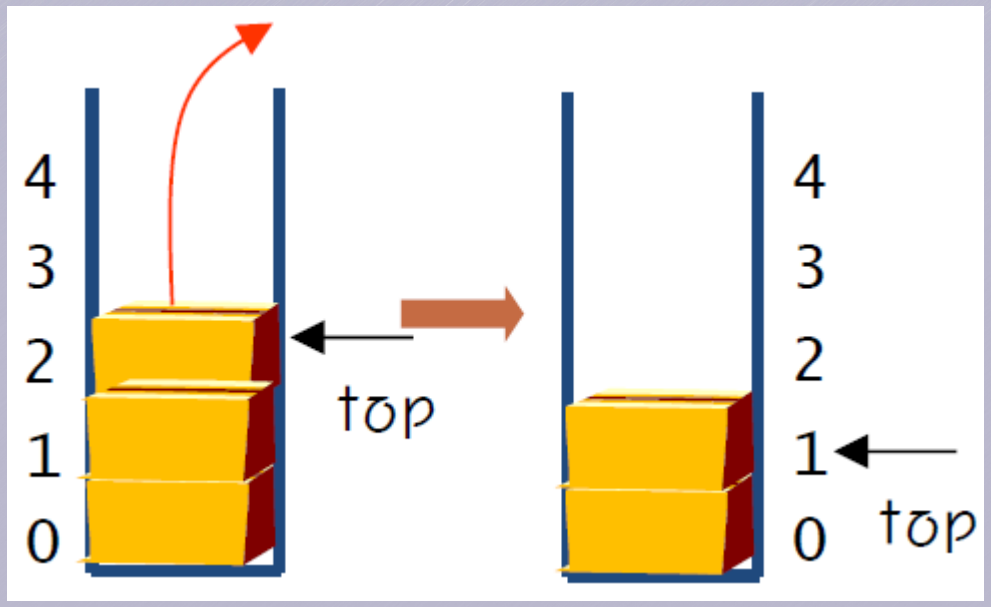
```
int isFull(){
        return (top == MAX_STACK - 1);
}
```

Stack in an Array




```
void push(int x){
        if(isFull()){
                cout << "Stack is Full" << endl;
                exit(1);
        }
        else stack[++top] = x;
}
```

Stack in an Array




```
int pop(){
        if(isEmpty()){
                cout << "Stack is Empty" << endl;
                exit(1);
        }
        else return stack[top--];
}
```

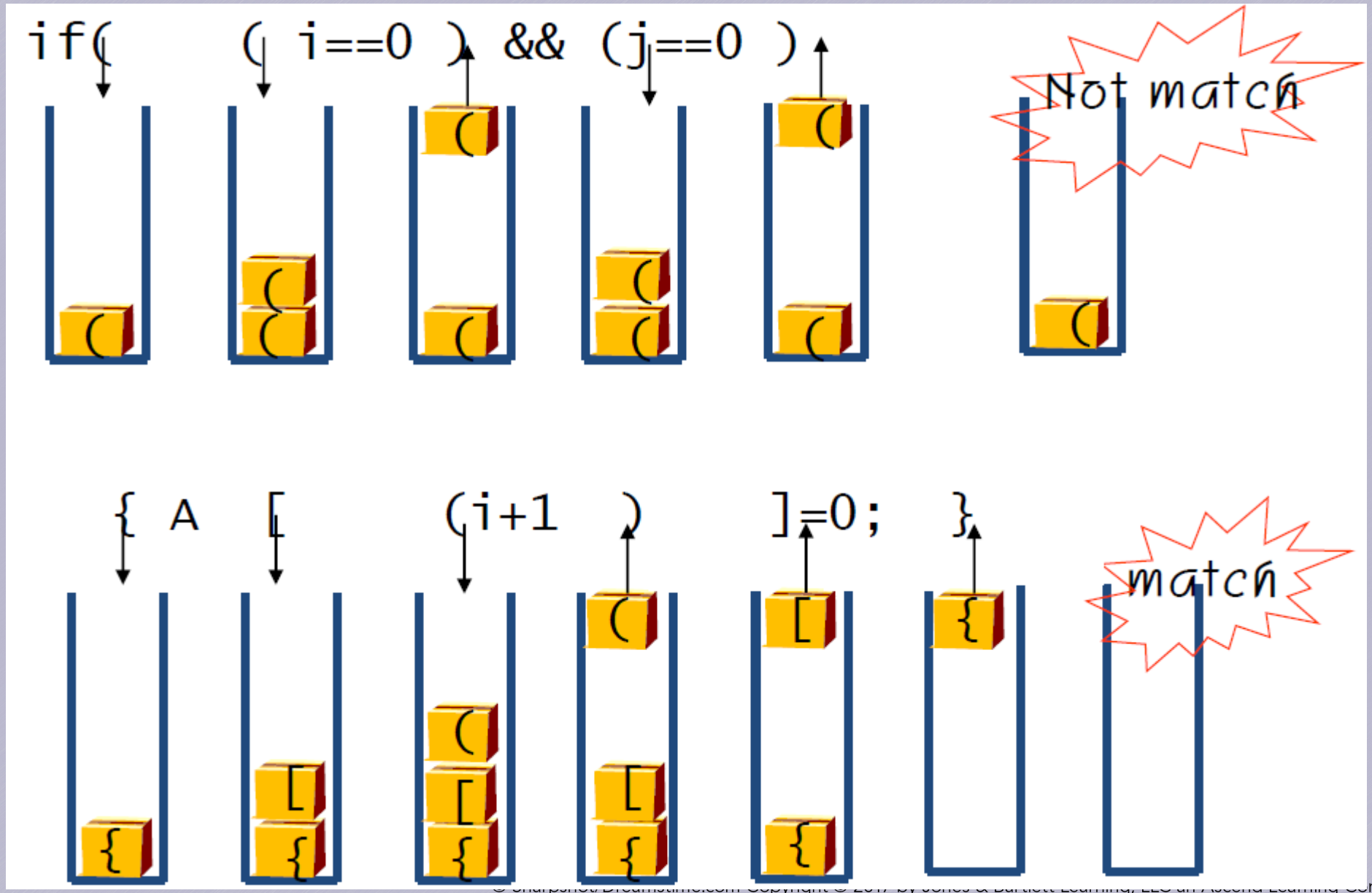
Stack Application

- Parenthesis Matching
- Problem
  - Test whether the left and right parentheses match in a given string
  - '[' and ']', '{' and '}', '(' and ')'
- Examples
  - (a(b)
  - a{b(c[d]e}f)
  - {a[bc](e(fg))}
  - (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

Stack Application

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

Stack Application



Stack Application

```
Algorithm paren_matching(expr)
Input : Expr (A string contains parentheses)
Output : Match(true), Does not match(false)
while(!EndOfExpr) do
       ch <- next character of the Expr
       switch(ch)
              case '(': case '[': case '{': {
                     push (ch)
                     break }
              case ')': case ']': case '}': {
                     if(isEmpty()) then return false
                     else close_ch <- pop()
                     if (ch and close_ch does not match) then
                            return false
                     break }
end while
if (stack empty) return true
else return false
end paren_matching
```

Queues: Logical Level

- **Queue:** An ADT in which elements are added to the rear and removed from the front; this behavior is called **FIFO** (**F**irst **I**n, **F**irst **O**ut)
- Items are homogeneous, like in stacks and lists
- Example: A line of people at a cash register

Queues: Application Level

Like stacks, queues are used in various ways by the OS and other systems:

- Scheduling jobs on the processor
- Buffering data between processes or other systems

Queue Operations

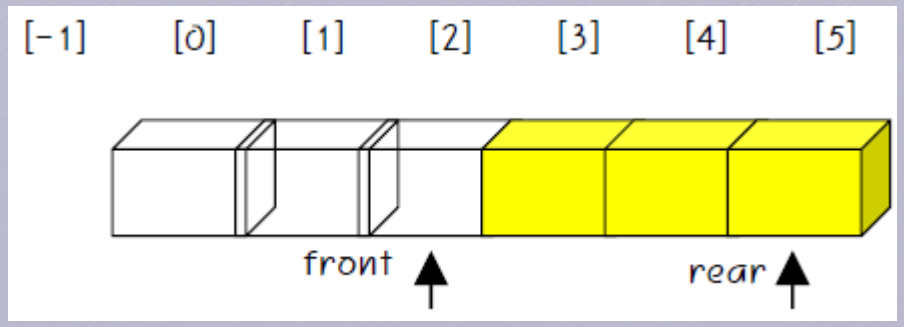
- **Enqueue:** Add an item to the end of the queue
- **Dequeue:** Removes the item at the front of the queue and returns it
- **IsEmpty:** Returns true if the queue is empty
- **IsFull:** Returns true if the queue is full
- **MakeEmpty:** Removes all items from the queue

Queue in an Array

```
#define MAX_QUEUE 5

class IntQueue {
      protected:
            int queue[MAX_QUEUE] = {0,};
            int rear = 0;
            int front = 0;

      public:
            void enQueue(int x);
            int deQueue();
            int isEmpty();
            int isFull();
            void makeEmpty();
};
```

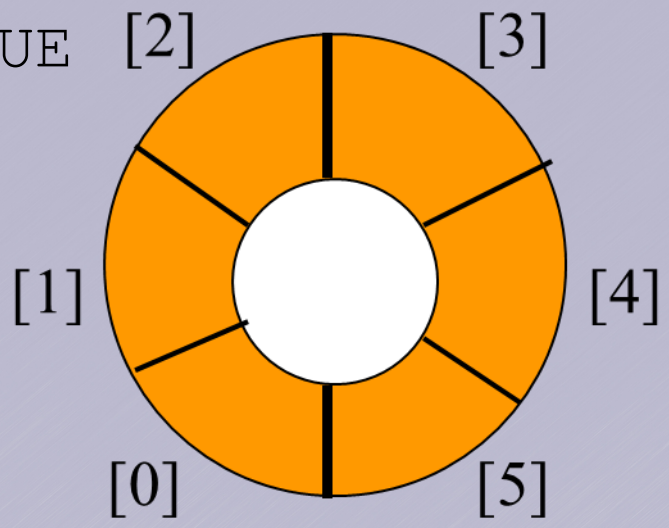
Possible implementations: Linear Queue, **Circular Queue**

Queue in an Array



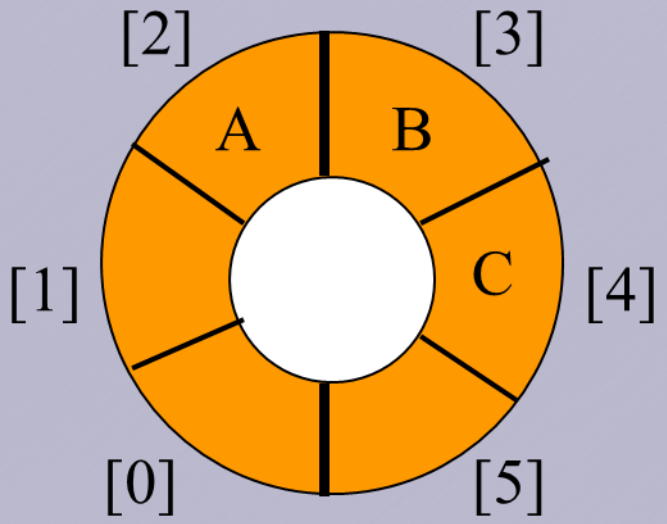
- Linear Queue
  - Increase rear when EnQueue
  - Increase front when DeQueue
  - Adjust front and rear values when it is full

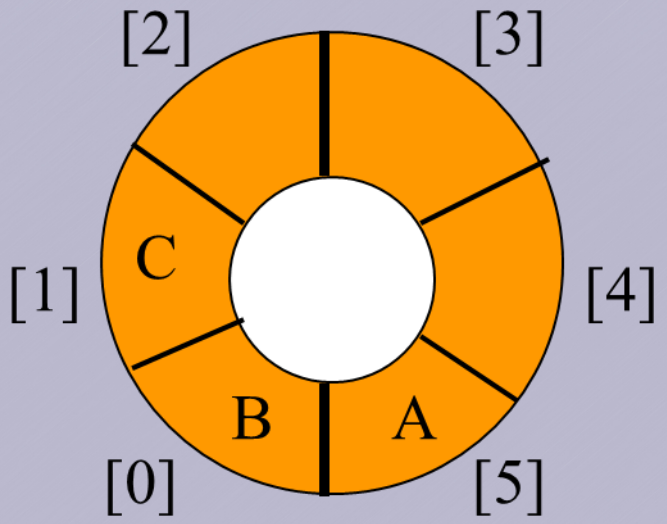
- Solution: Circular Queue
  - `rear = (rear+1) mod MAX_QUEUE`
  - `front = (front+1) mod MAX_QUEUE`



Queue in an Array

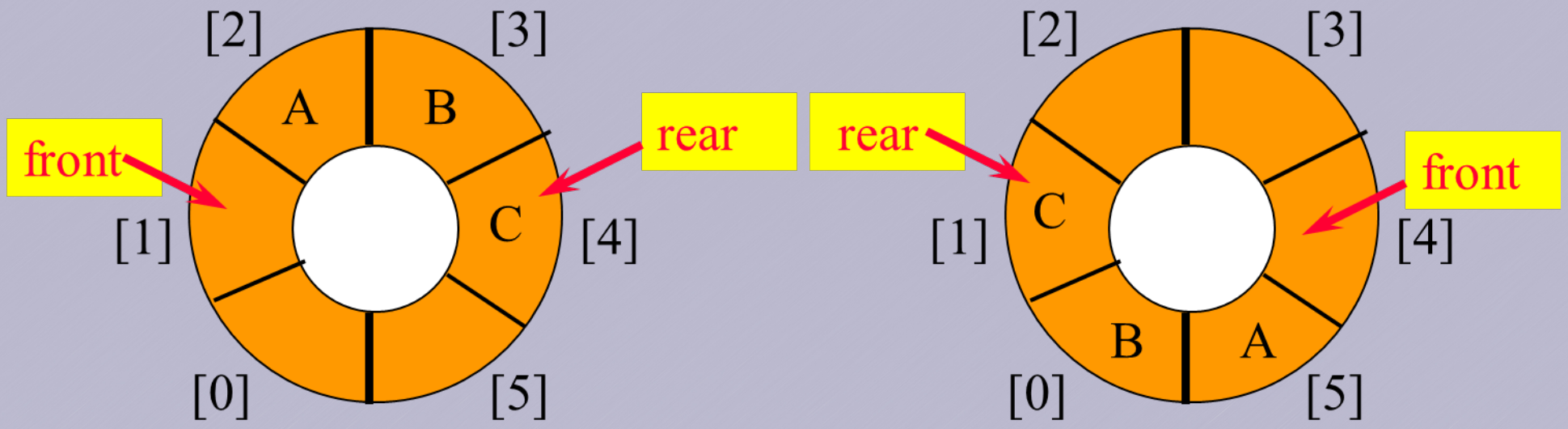
- Example: Possible configuration with 3 elements



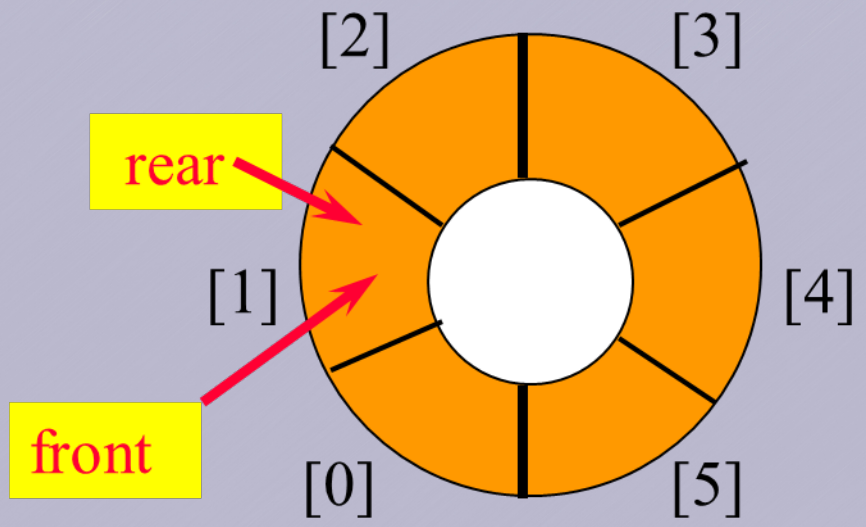


Queue in an Array

- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element

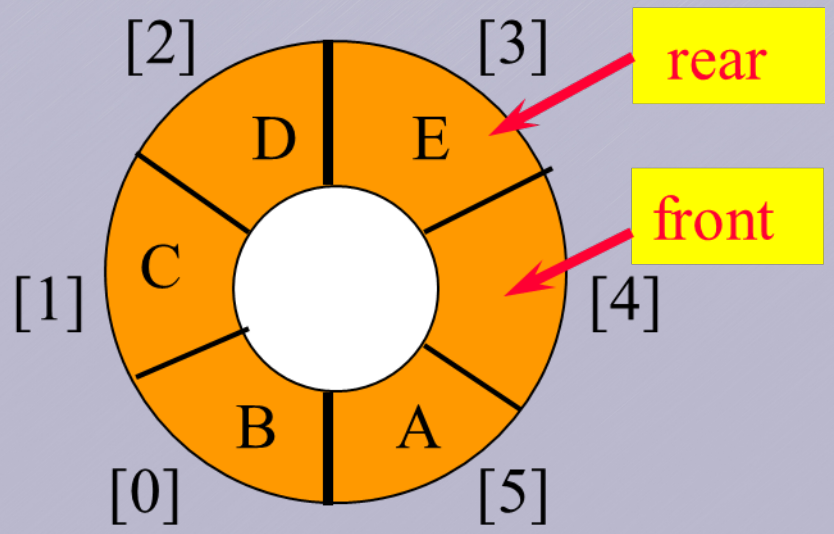


Queue in an Array



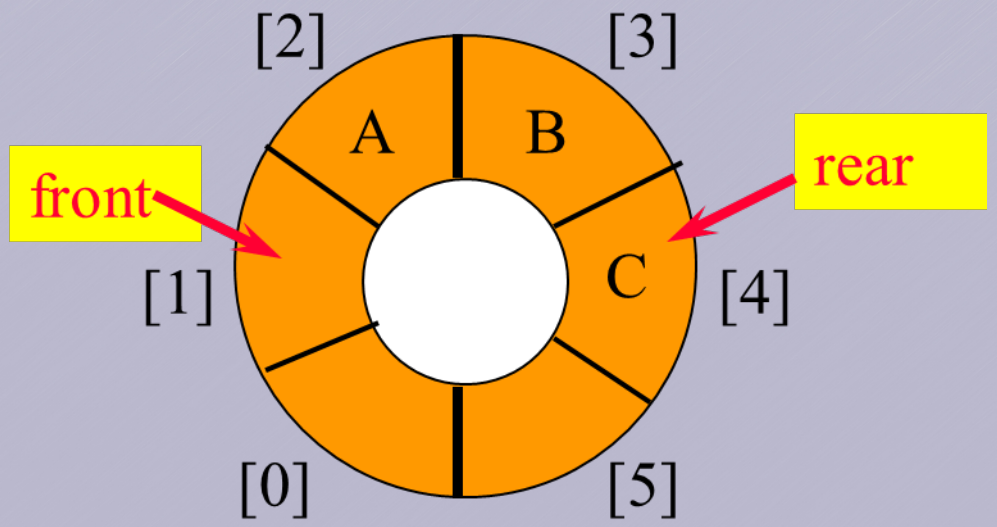

```
int isEmpty(){
        return (front == rear);
}
```

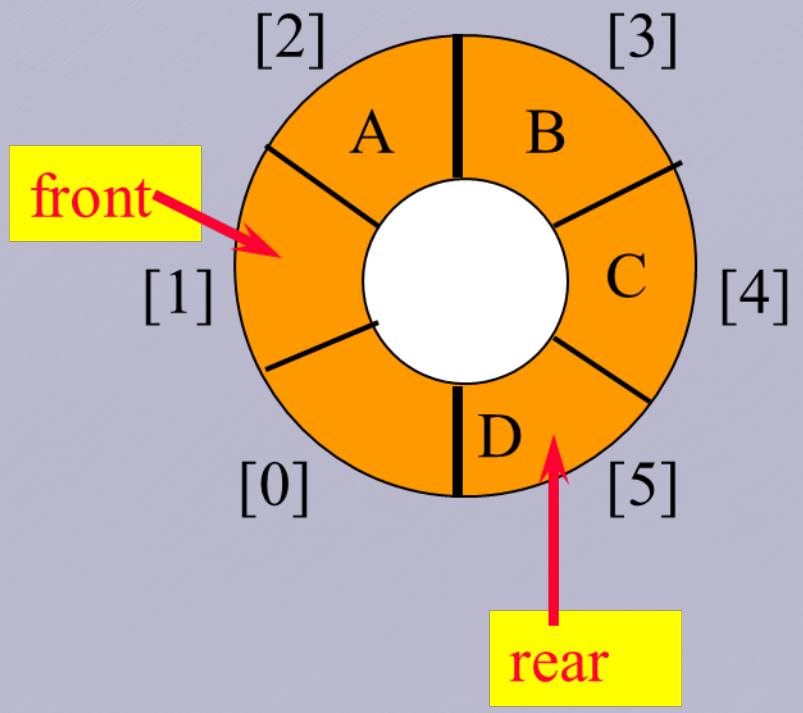
Queue in an Array




```
int isFull(){
        return (front == ((rear + 1) % MAX_QUEUE));
}
```

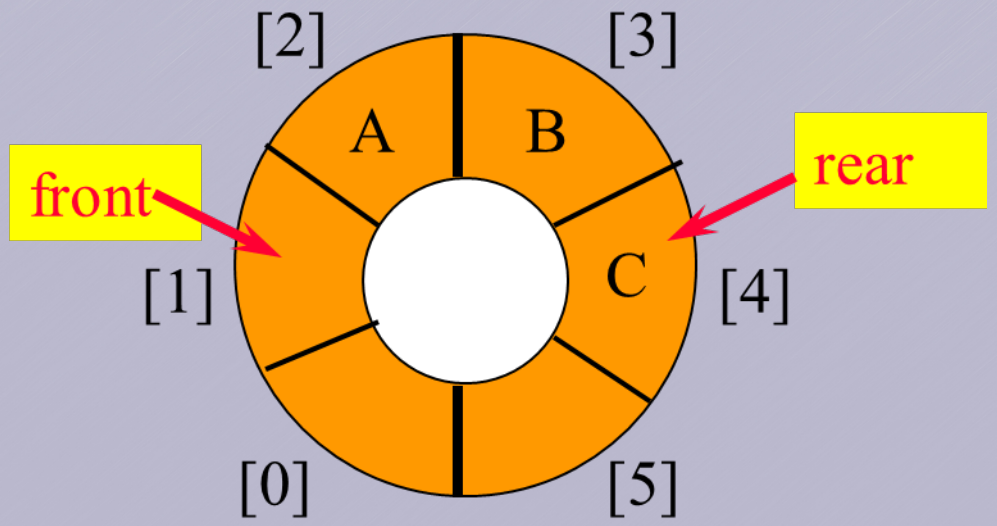
Queue in an Array

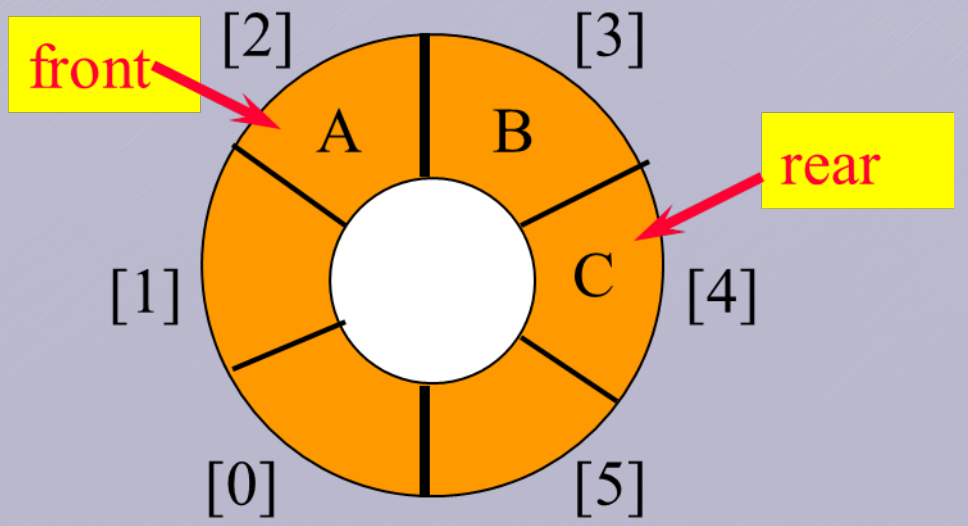





```
void enQueue(int x){
        if (isFull()){
                cout << "Queue is full" << endl;
                exit(1);
        }else{
                rear = (rear + 1) % MAX_QUEUE;
                queue[rear] = x;
        }
}
```

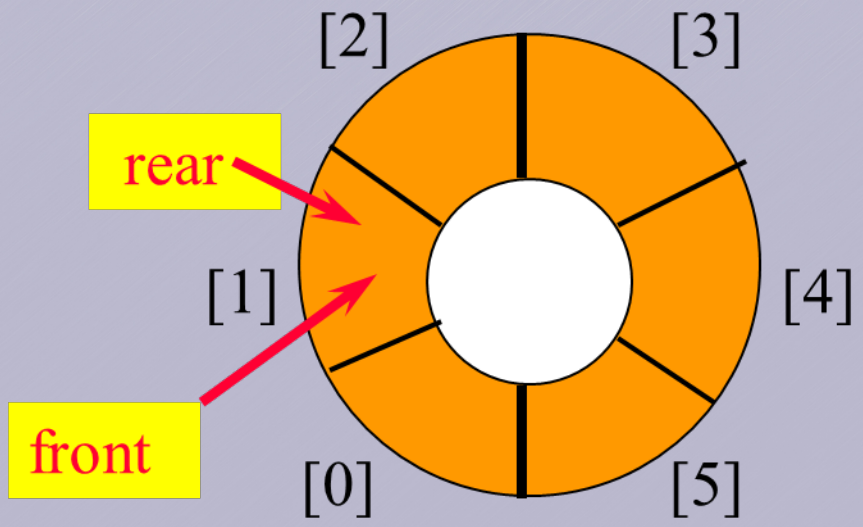
Queue in an Array






```
int deQueue(){
        if (isEmpty()){
                cout << "Queue is empty" << endl;
                exit(1);
        }else{
                front = (front + 1) % MAX_QUEUE;
                return queue[front];
        }
}
```

Queue in an Array




```
void makeEmpty(){
        front = rear = 0;
}
```