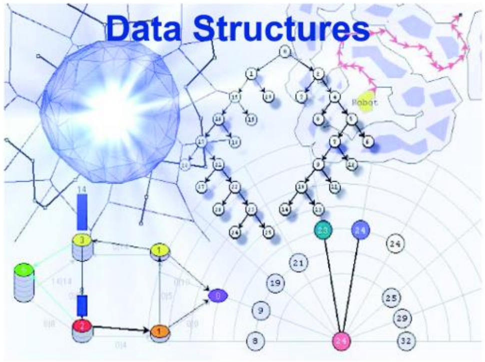
BBM 201
DATA STRUCTURES

Lecture 5:
Stacks and Queues



2018-2019 Fall

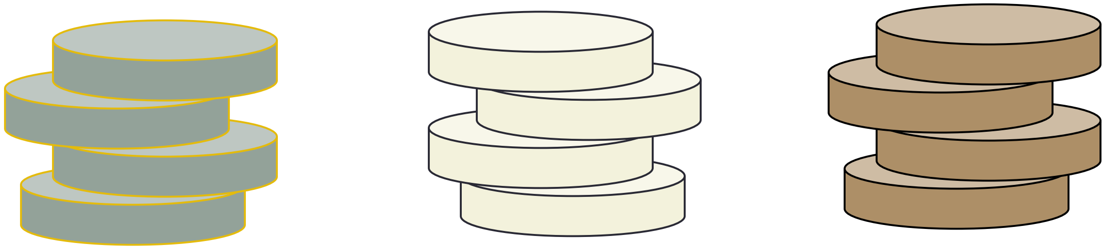

BUS STOP
6 9
13 15 60
96 291 294 297

Stacks

- A list on which insertion and deletion can be performed.
  - **Based on Last-in-First-out (LIFO)**

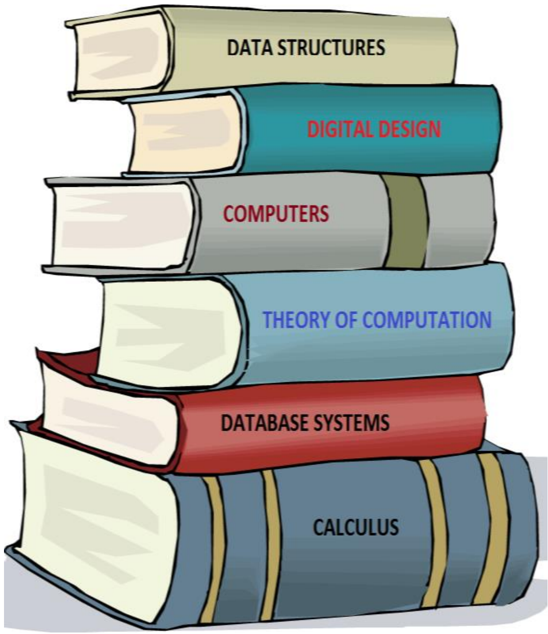
- Stacks are used for a number of applications:
  - Converting a decimal number into binary
  - Program execution
  - Parsing
  - Evaluating postfix expressions
  - Towers of Hanoi

  …



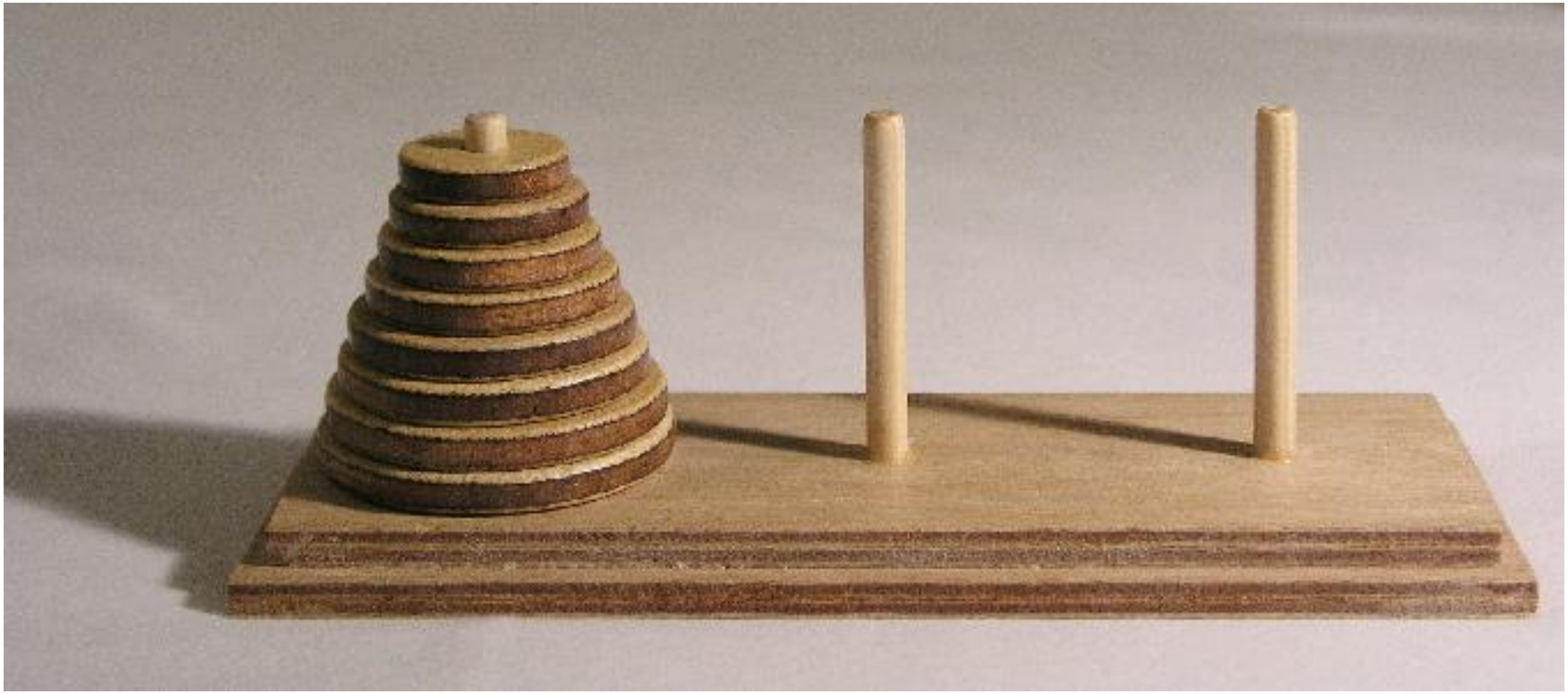
Stacks

A stack is an ordered lists in which insertions and deletions are made at one end called the ***top***.



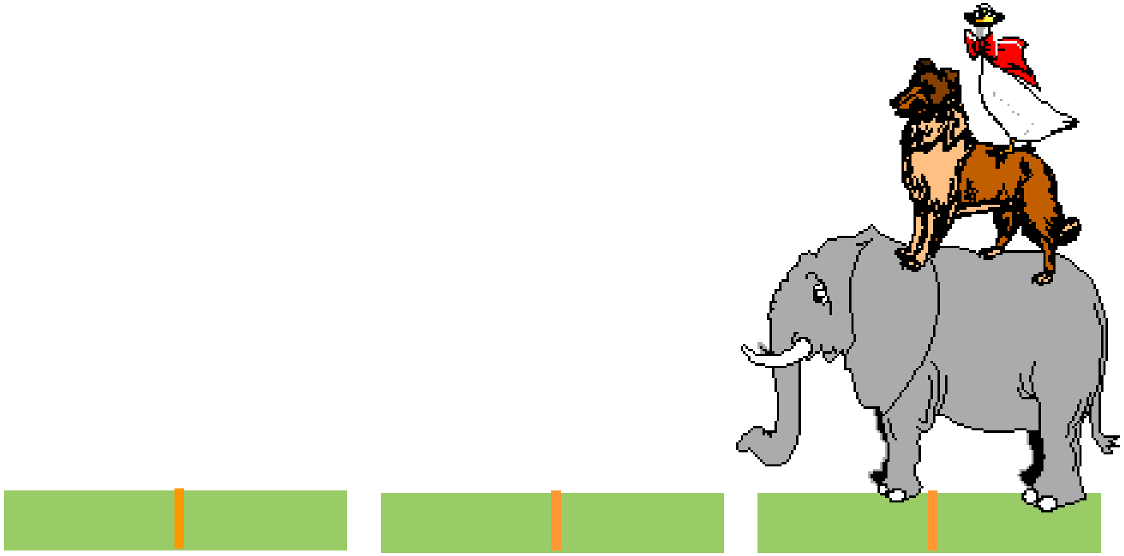
Stacks

Towers of Hanoi

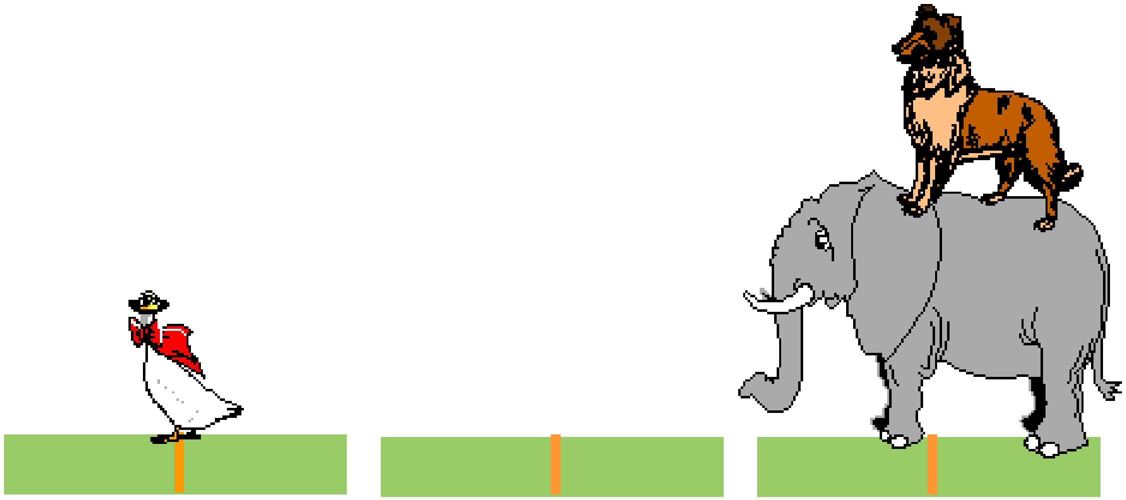


Object of the game is to move all the disks (animals) over to Tower 3.
But you cannot place a larger disk onto a smaller disk.

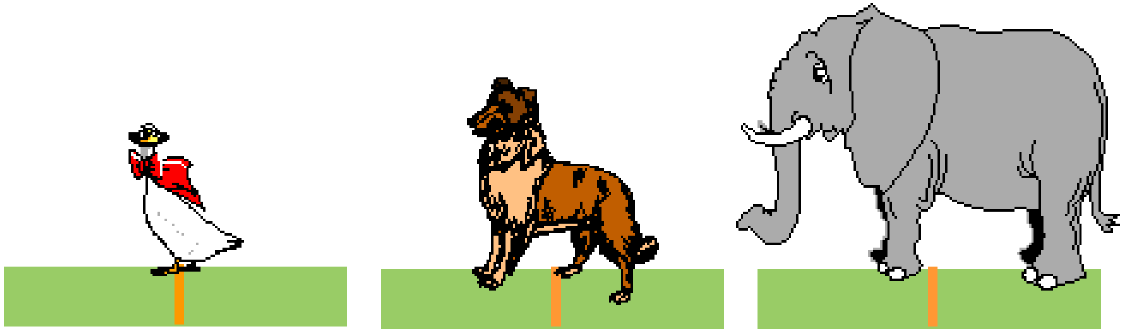
Towers of Hanoi



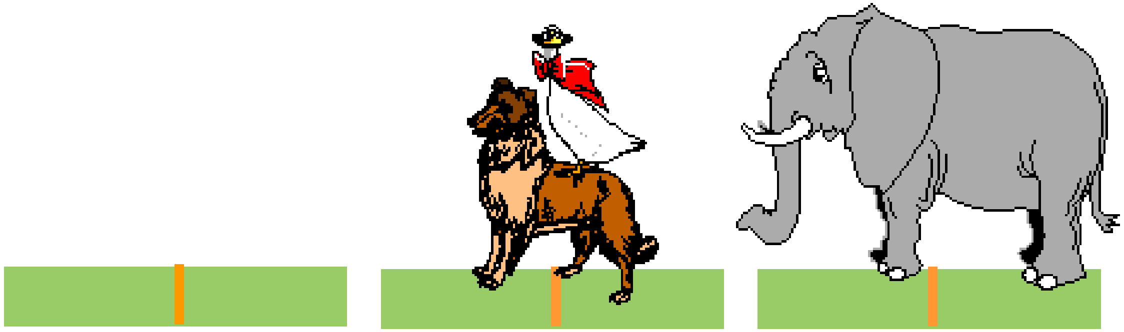
Towers of Hanoi



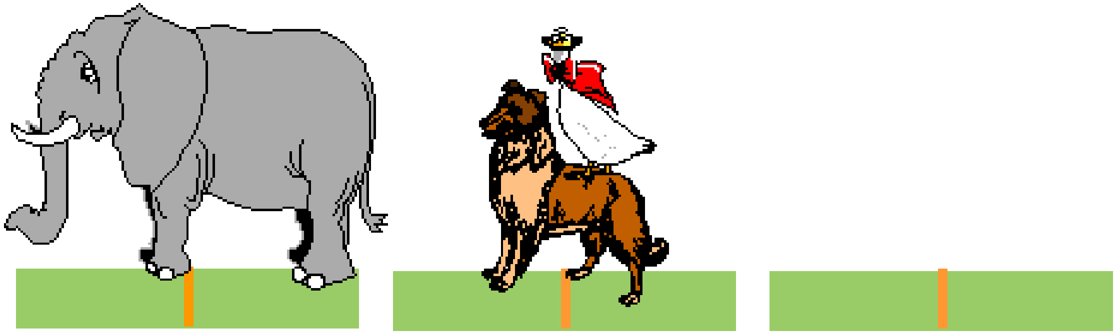
Towers of Hanoi



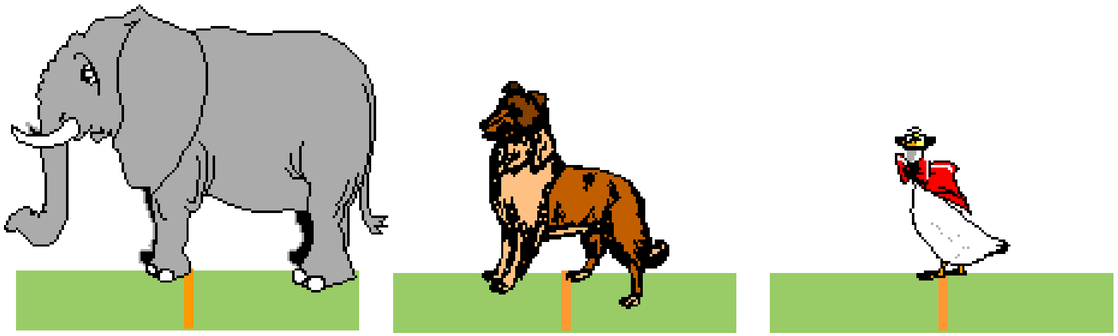
Towers of Hanoi



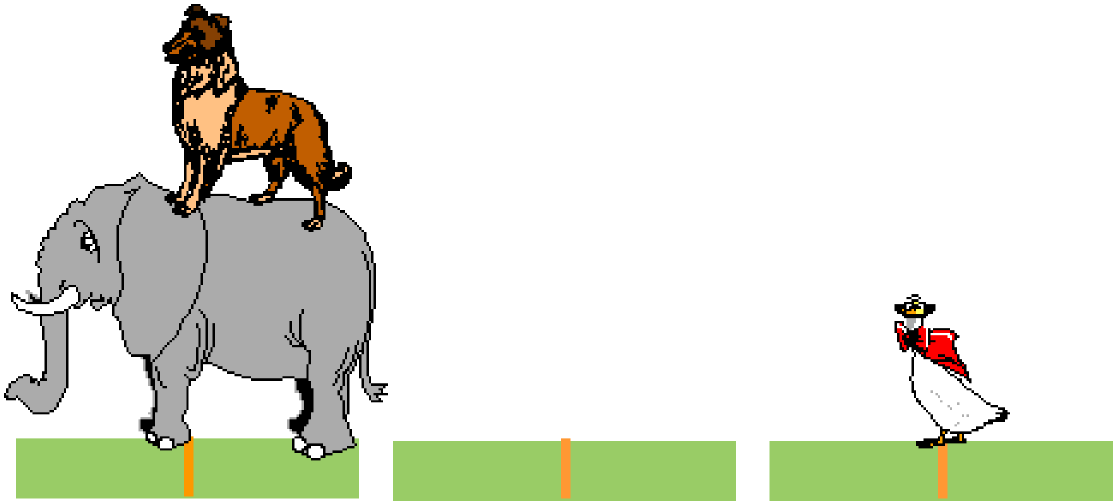
Towers of Hanoi



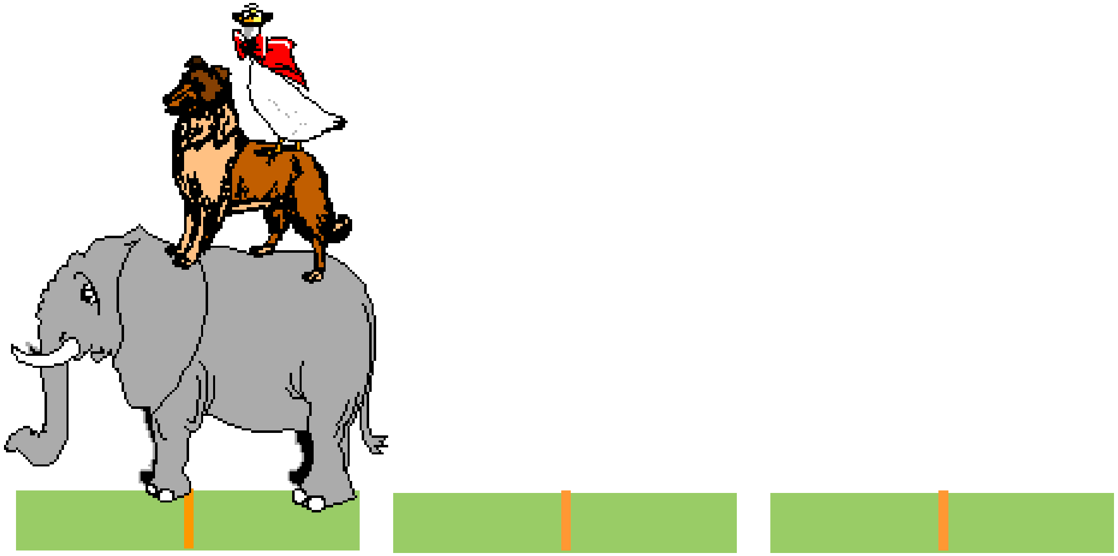
Towers of Hanoi



Towers of Hanoi



Towers of Hanoi



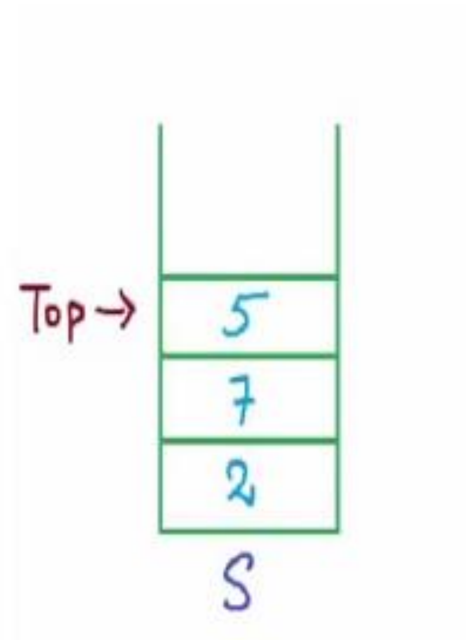
Stack Operations

1. Pop()
2. Push(x)
3. Top()
4. IsEmpty()

- An insertion (of, say x) is called push operation and removing the most recent element from stack is called pop operation.
- Top returns the element at the top of the stack.
- IsEmpty returns true if the stack is empty, otherwise returns false.

All of these take constant time - O(1)

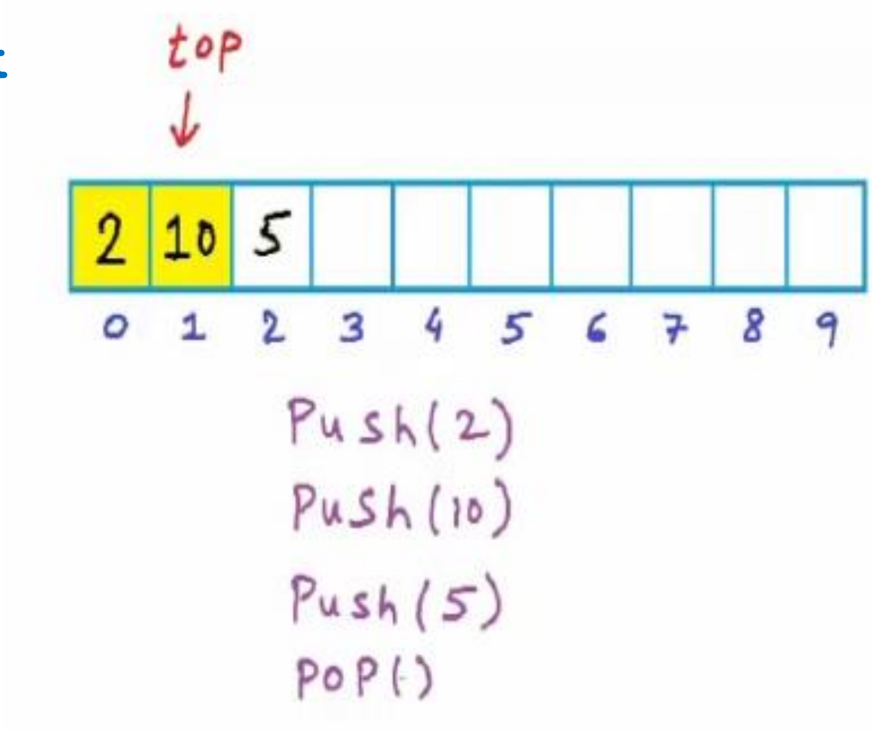
Example



- Push(2)
- Push(10)
- Pop()
- Push(7)
- Push(5)
- Top(): 5
- IsEmpty(): False

Array implementation of stack (pseudocode)

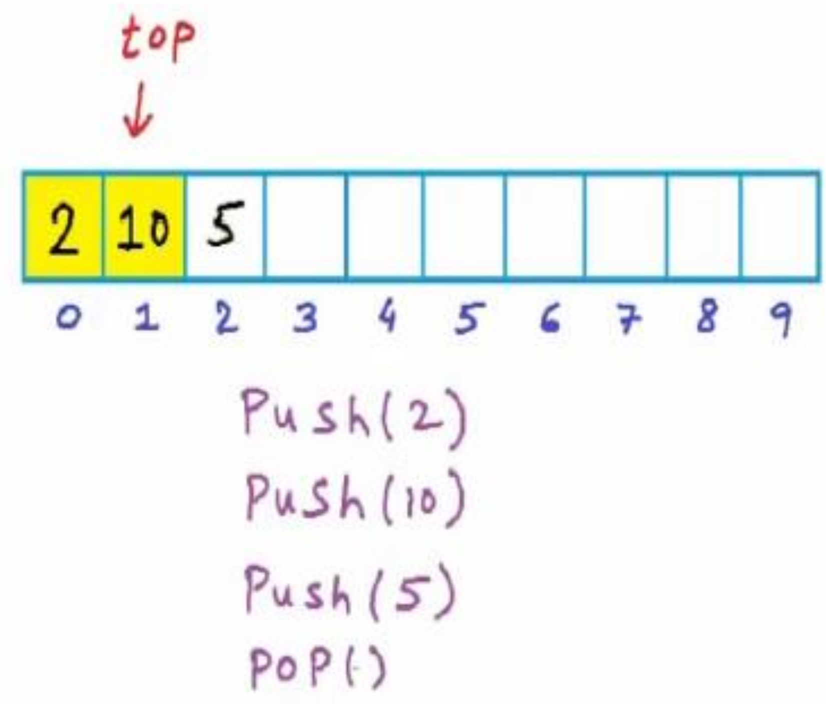
```
int A[10]
top ← -1 //empty stack
Push(x)
{
   top ← top + 1
   A[top]← x
}
Pop()
{
   top ← top – 1
}
```



For an empty stack, top is set to -1.
In push function, we increment top.
In pop, we decrement top by 1.

Array implementation of stack (pseudocode)

```
Top()
{
   return A[top]
}
IsEmpty()
{
   if(top == -1)
      return true
   else
      return false
}
```



Stack

Data Structure

```
#define MAX_STACK_SIZE 100

typedef struct{
            int VALUE;
          }element;

element stack[MAX_STACK_SIZE];
int top=-1;
```

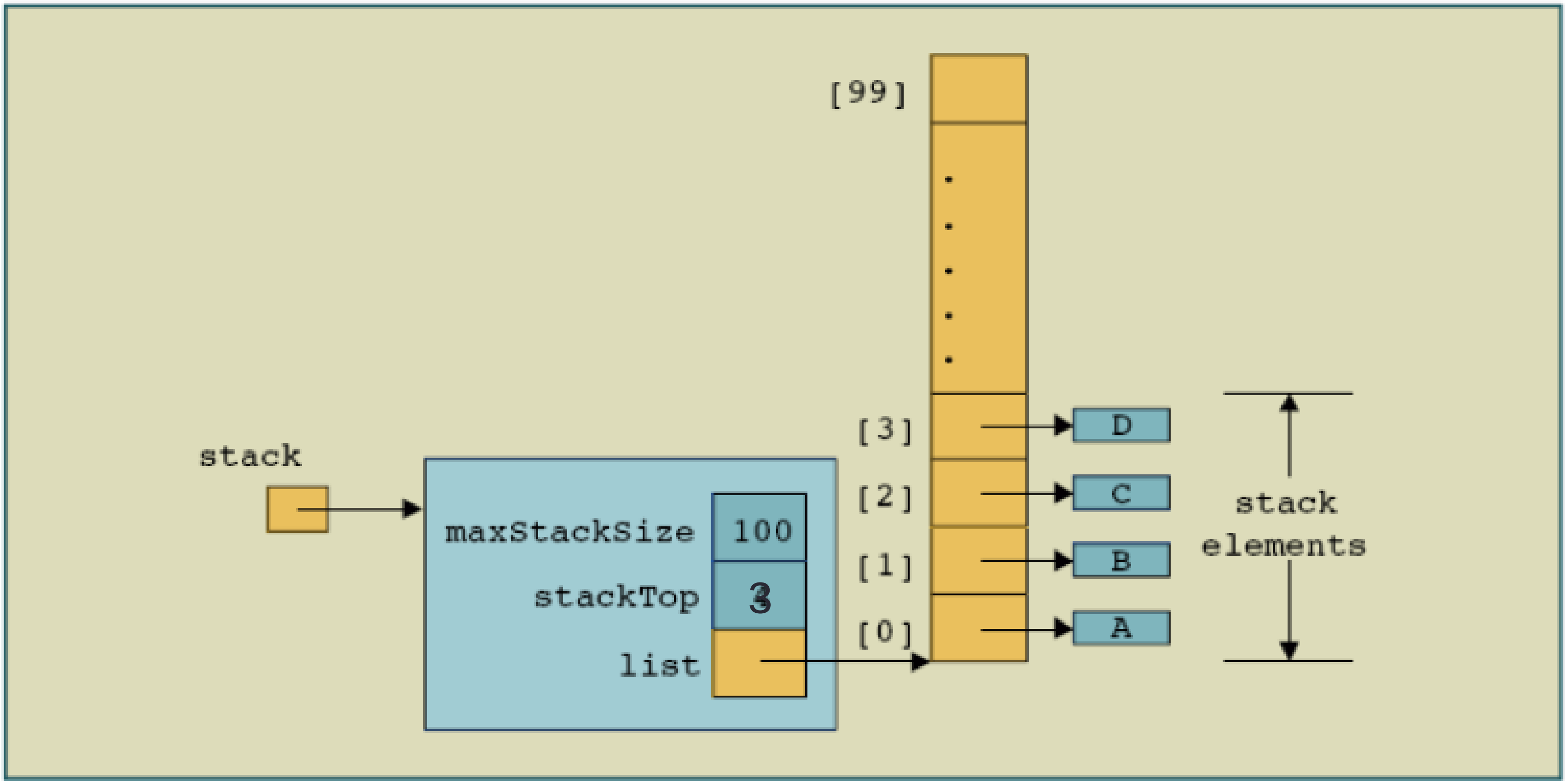
Push Stack

```
void push (element item)
{
   if(top >= MAX_STACK_SIZE-1){
         isFull();
         return;
   }
   stack[++top]=item;
}
```

Pop Stack

```
element pop()
{
    if(top==-1)
        return empty_stack();
    return stack[top--];
}
```

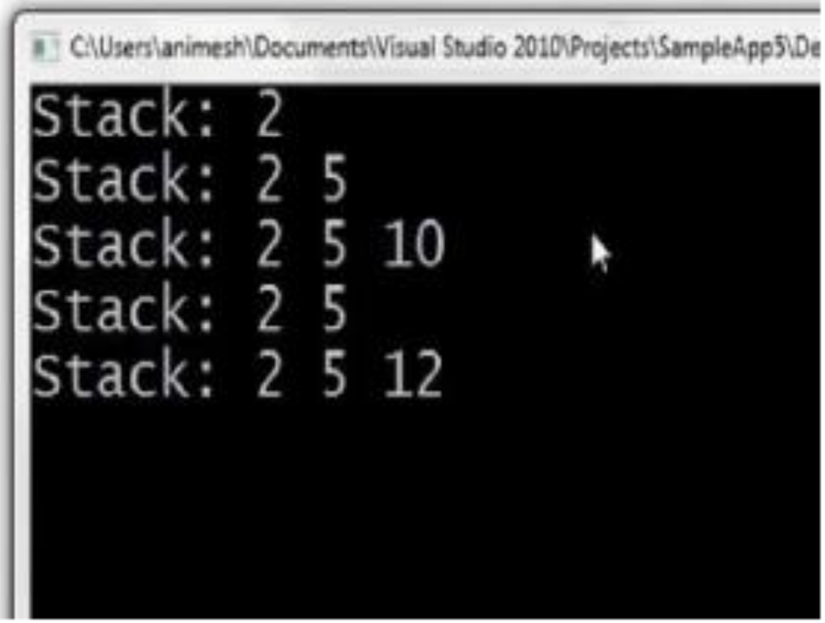
Implementation of Stacks Using Arrays



More array implementation

```
// Stack - Array based implementation.
#include<stdio.h>
#define MAX_SIZE 101
int A[MAX_SIZE];
int top = -1;
void Push(int x) {
    if(top == MAX_SIZE -1) {
        printf("Error: stack overflow\n");
        return;
    }
    A[++top] = x;
}
void Pop() {
    if(top == -1) {
        printf("Error: No element to pop\n");
        return;
    }
    top--;
}
int Top() {
    return A[top];
}
int main() {
}
```

```c
void Print() {
    int i;
    printf("Stack: ");
    for(i = 0;i<=top;i++)
        printf("%d ",A[i]);
    printf("\n");
}
int main() {
    Push(2);Print();
    Push(5);Print();
    Push(10);Print();
    Pop();Print();
    Push(12);Print();
}
```



Check For Balanced Parentheses using Stack

Expression	Balanced?
(A+B)	
{(A+B)+(C+D)}	
{(x+y)*(z)	
[2*3]+(A)]	
{a+z)	

Check For Balanced Parantheses using Stack

Expression	Balanced?
( )	Yes
{ ( ) ( ) }	Yes
{ ( ) ( )	No
[ ] ( )]	No
{)	No

The count of opening should be equal to the count of closings.
AND
Any parenthesis opened last should be closed first.

Idea: Create an empty list

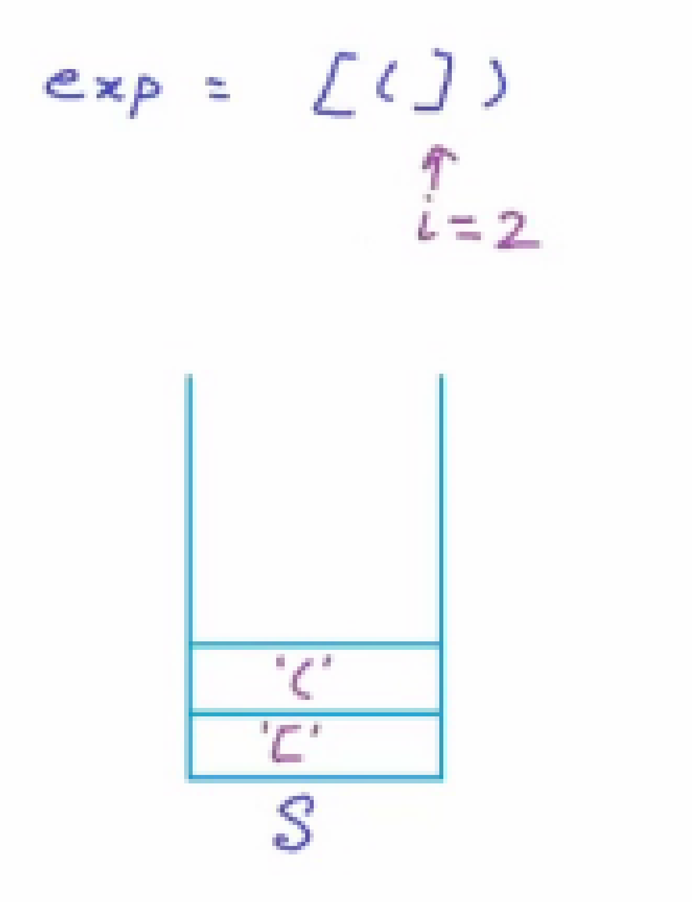
- Scan from left to right
  - If opening symbol, add it to the list
    - Push it into the stack
  - If closing symbol, remove last opening symbol of the same type
    - using Pop from the stack
  - Should end with an empty list

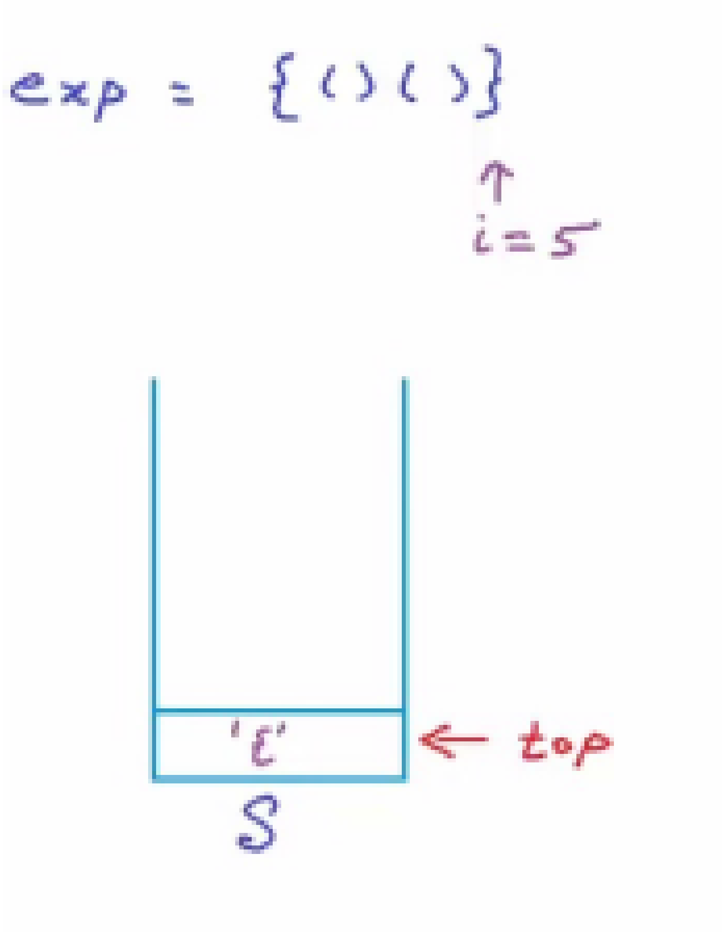
Check For Balanced Parantheses: Pseudocode

```
CheckBalancedParanthesis(exp){
    n← length(exp)
    Create a stack: S
    for i← 0 to n-1{
        if exp[i] is '(' or '{' or '['
            Push(exp[i])
        else if exp[i] is ')' or '}' or ']'{
            if (S is empty or
                top does not pair with exp[i])
              return false
            else
              pop()
        }
    }
    return S is empty?
}
```

Create a stack of characters and scan this string
by using push if the character is an opening parenthesis and
by using pop if the character is a closing parenthesis. (See next slide)

Examples



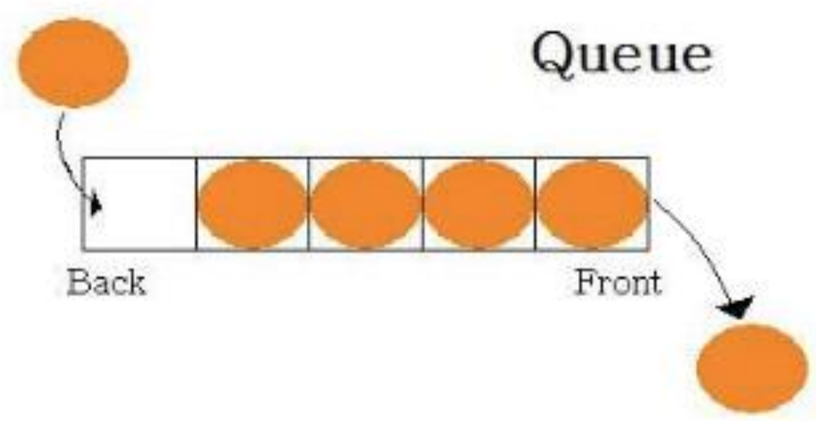


The pseudo code will return false.

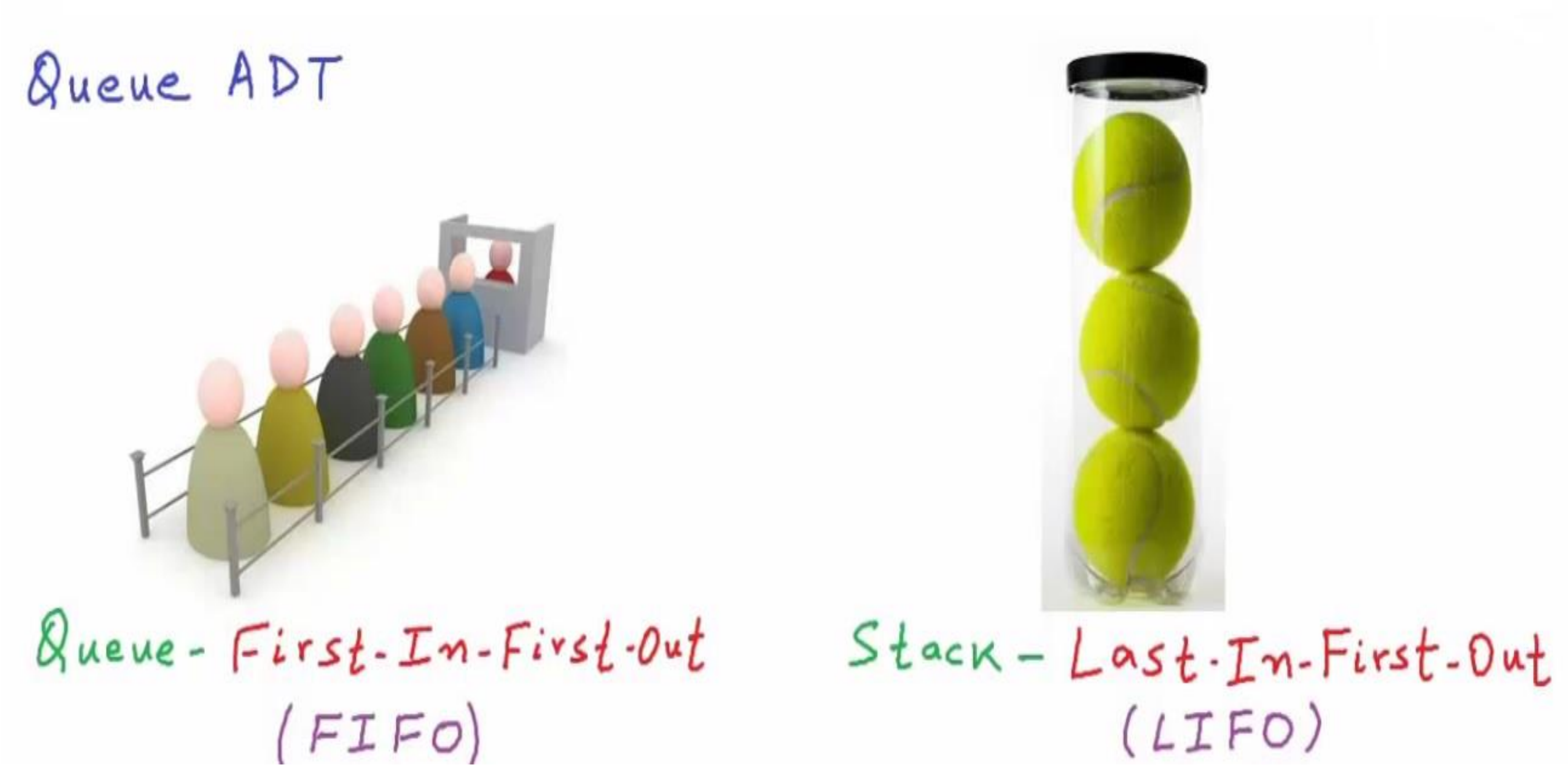
The pseudo code will return true.

Queues

- A queue is an ordered list on which
  - all insertions take place at one end called the **rear/back** and
  - all deletions take place at the opposite end called the **front**.
  - Based on **First-in-First-out (FIFO)**

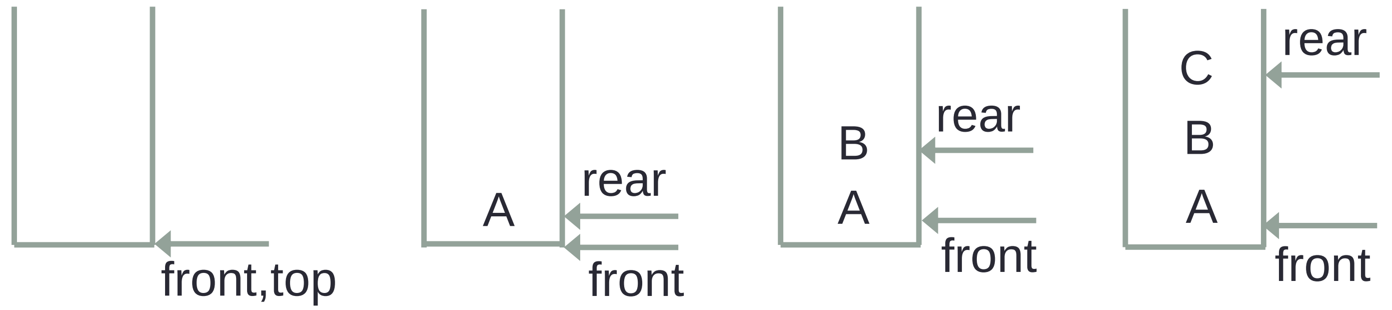


Comparison of Queue and Stack




© Alamy

Queues



Queue is a list with the restriction that insertion can be made at one end (rear)
And deletion can be made at other end (front).

Built-in Operations for Queue

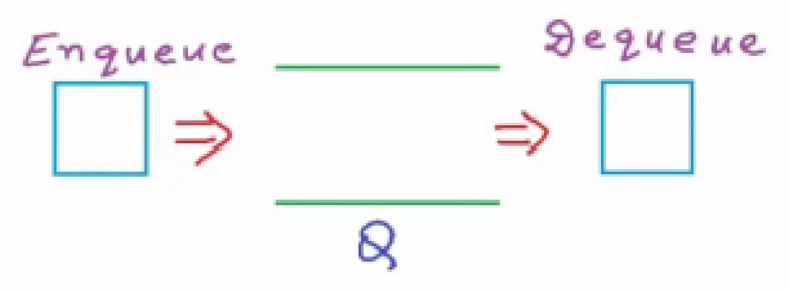
Enqueue(x) or Push(x)

Dequeue() or Pop()

Front(): Returns the element in the front without removing it.

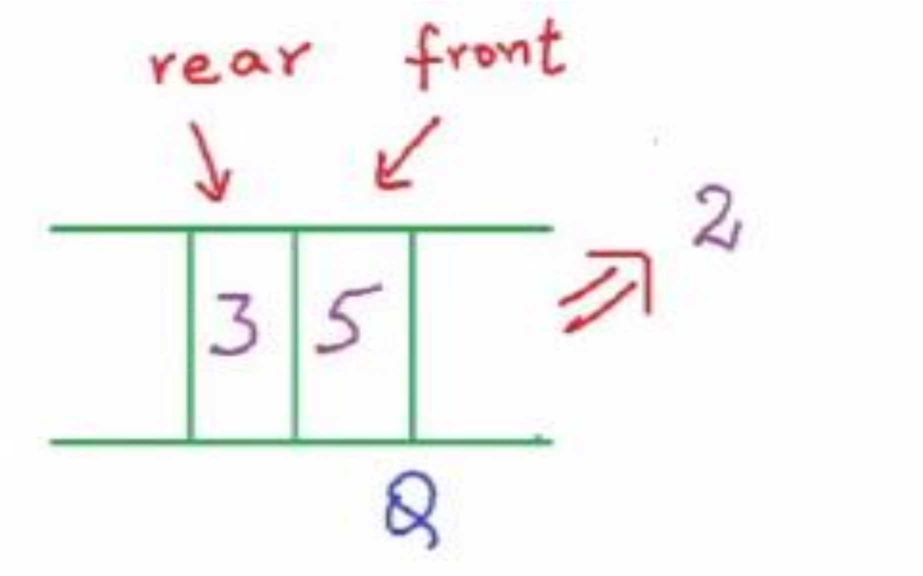
IsEmpty(): Returns true or false as an answer.

IsFull()



Each operation takes constant time, therefore has O(1) time complexity.

Example



Enqueue(2)

Enqueue(5)

Enqueue(3)

Dequeue() → 2

Front() → 5

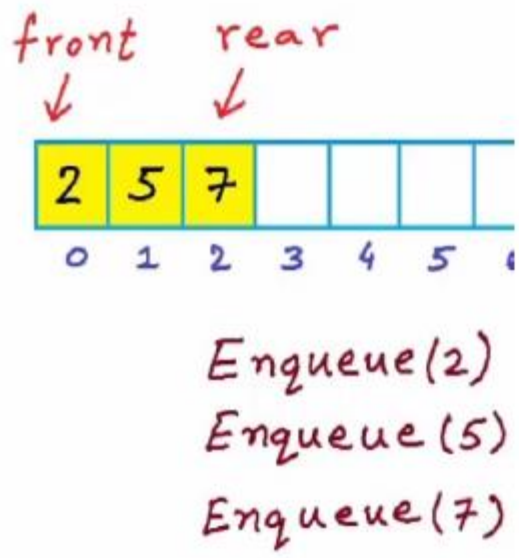
IsEmpty() → False

Applications:

- Printer queue
- Process scheduling

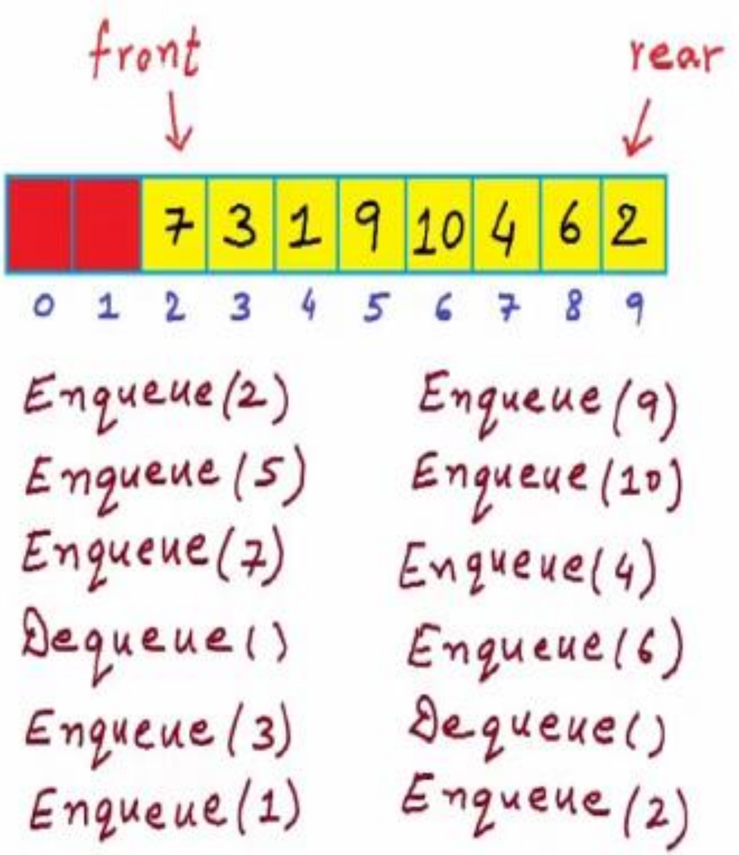
Array implementation of queue (Pseudocode)

```
int A[10]
front ← -1
rear ← -1
IsEmpty(){
    if (front == -1 && rear == -1)
        return true
    else
        return false}
Enqueue(x){
    if IsFull()
        return
    else if IsEmpty()
        front ← rear ← 0
    else
        rear ← rear+1
    A[rear]← x
}
```



Array implementation of queue (Pseudocode)

```
Dequeue(){
    if IsEmpty(){
        return
    else if (front == rear)
        front ← rear ← -1
    else{
        front ← front+1
}
```



At this stage, we cannot Enqueue an element anymore.

Queue

Data Structure

```
#define MAX_QUEUE_SIZE 100

typedef struct{
        int value;
}element;

element queue[MAX_QUEUE_SIZE];
int front=-1;
int rear=-1;
```

Add Queue

```
void addq( element item)
{
   if(rear == MAX_QUEUE_SIZE-1){
         isFull();
         return;
    }
    queue[++rear]=item;
}
```

Delete Queue

```
element deleteq(element item)
{
   if(front == rear)
      return isEmpty();
   return queue[++front];
}
```

Circular Queue

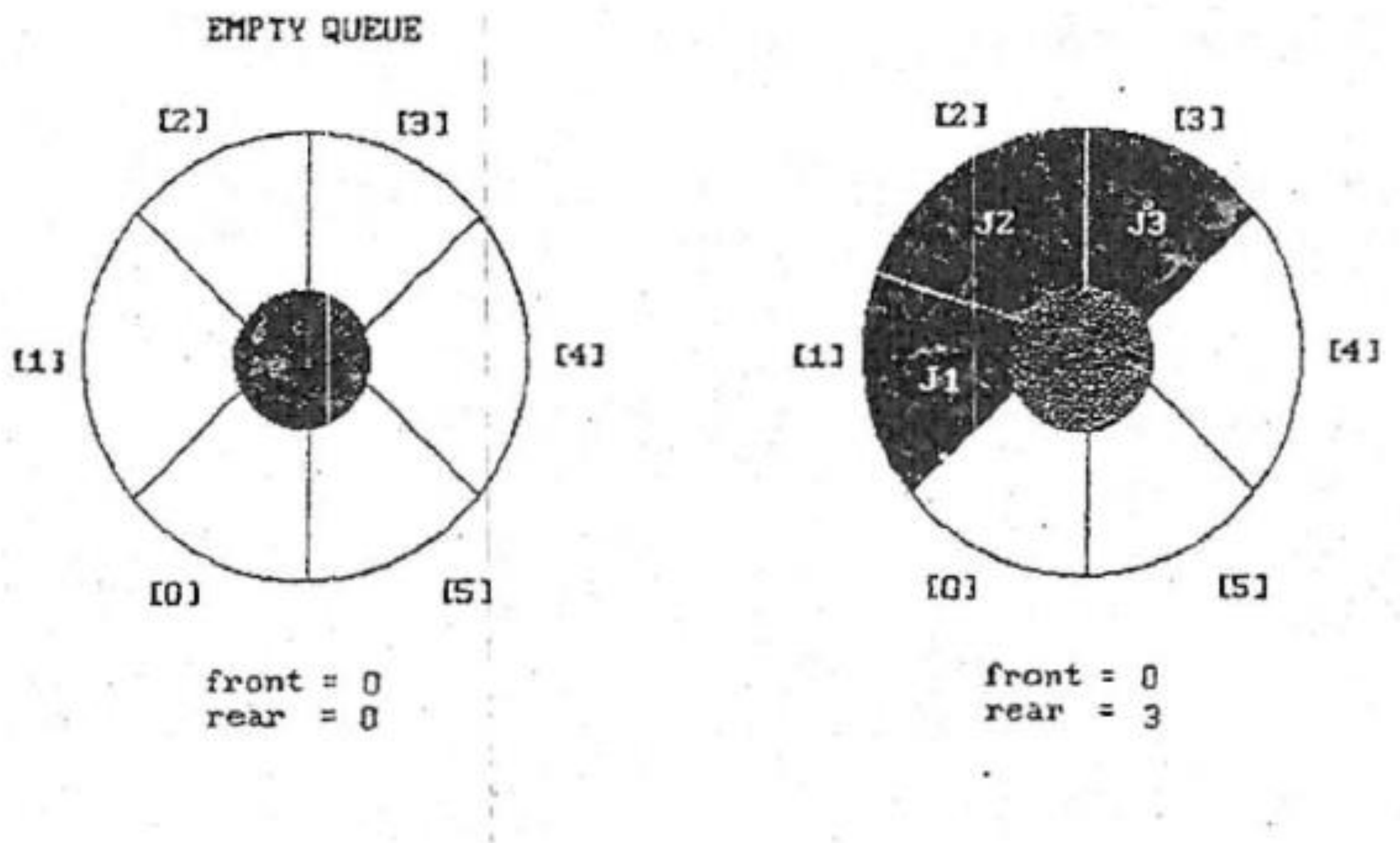
- When the queue is full
  (the rear index equals to MAX_QUEUE_SIZE)
  - We should move the entire queue to the left
  - Recalculate the rear

Shifting an array is time-consuming!

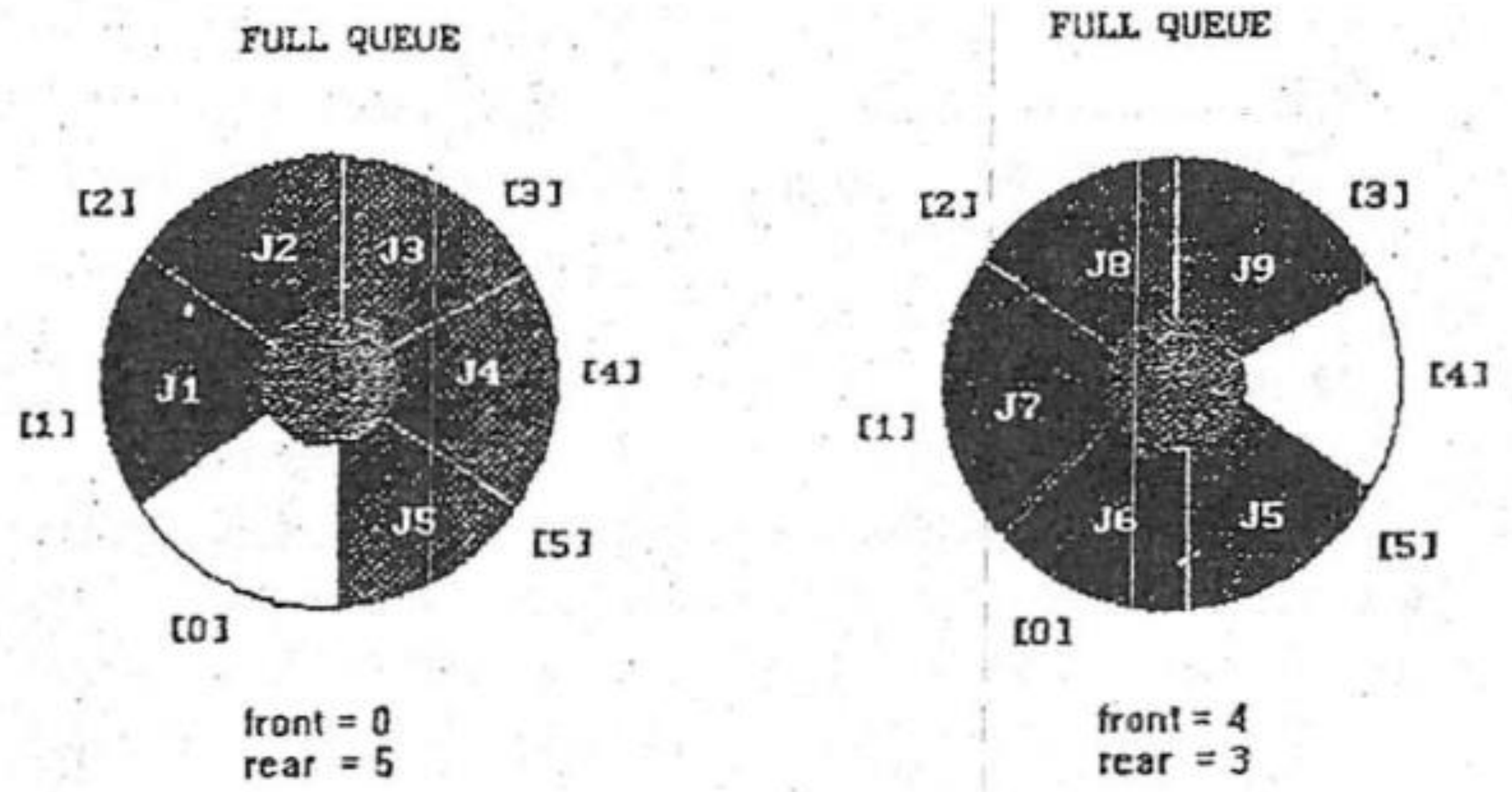
- O(MAX_QUEUE_SIZE)

Circular Queue

- More efficient queue representation

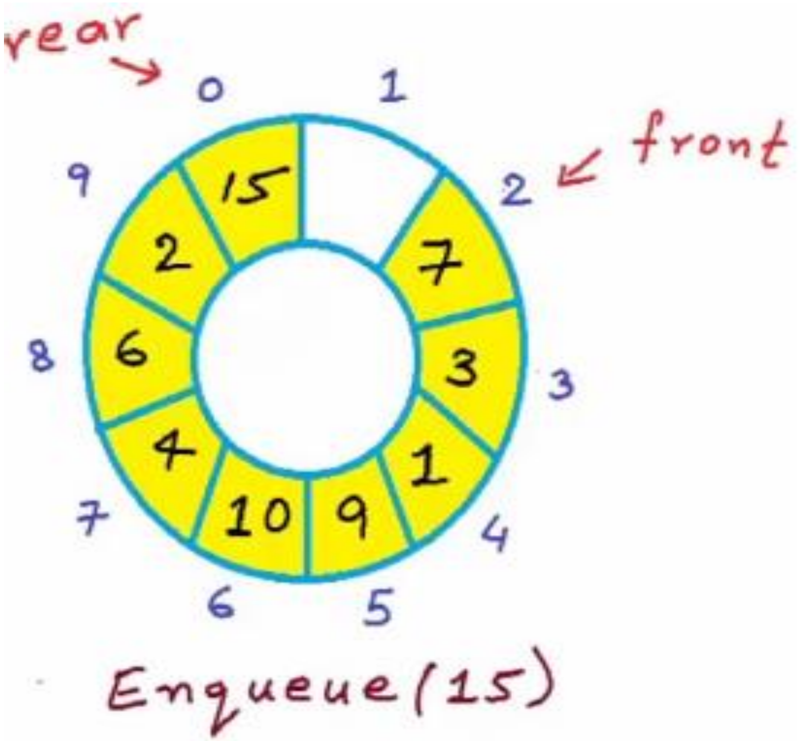


Full Circular Queue



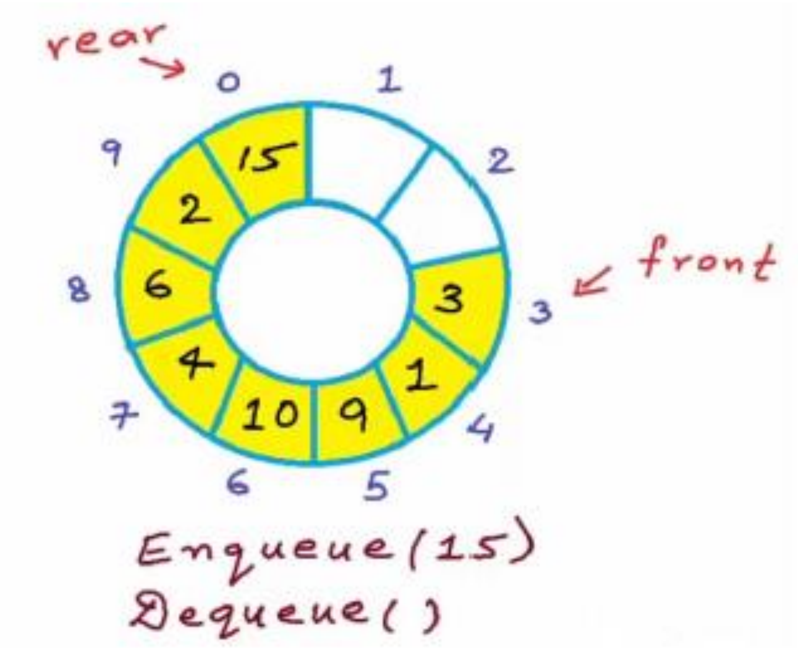
Enqueue for circular array (Pseudocode)

```
Current position = i
Next position = (i+1)% N
previous position = (i+N-1)%N
Enqueue(x){
    if (rear+1)%N == front
        return
    else if IsEmpty()
        front ← rear ← 0
    else
        rear ← (rear+1)%N
    A[rear]← x
}
```



Dequeue for circular array (Pseudocode)

```
Deqeue(x){
    if IsEmpty()
        return
    else if(front == rear)
        front ← rear ← -1
    else
        front ← (front+1)%N
}
```



Add Circular Queue

```
void addcircularq( element item)
{
    rear=(rear+1)% MAX_QUEUE_SIZE;
    if(front == rear){
        isFull(rear);
        return;
    }
    queue[rear]=item;
}
```

Delete Circular Queue

```
void deletecircularq()
{
	if(front == rear)
	  return isEmpty();
	front=(front+1)% MAX_QUEUE_SIZE;
	return queue[front];
}
```