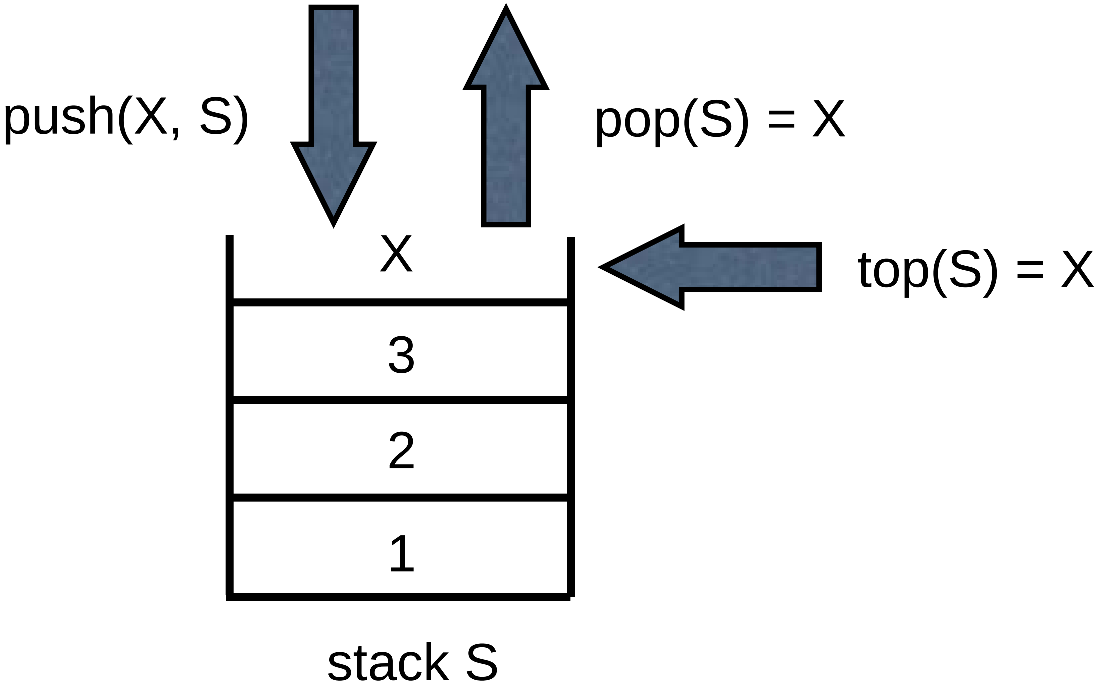
# Data Structure: Stack

Yongjun Park

Hanyang University

# Stack ADT

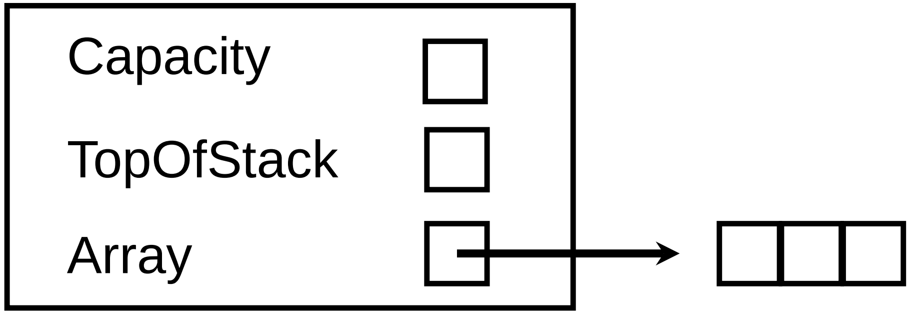
- An ordered list in which insertions and deletions can be performed at one end of the list
- operations
  - push(X, S): insert X in the list S
  - pop(S): deletes the most recently inserted element from S



push(1, S) → push(2, S) → push(3, S) → push(X, S)

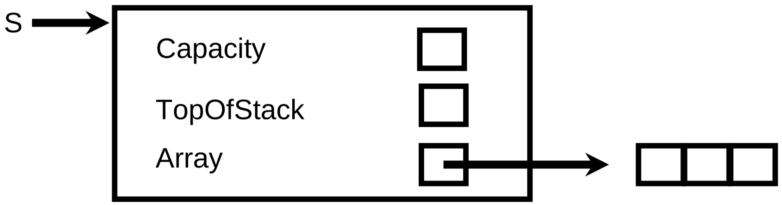
# Stack ADT: array implementation

```
typedef struct StackRecord *Stack;

struct StackRecord
{
    int Capacity;
    int TopOfStack;
    ElementType *Array;
};
```



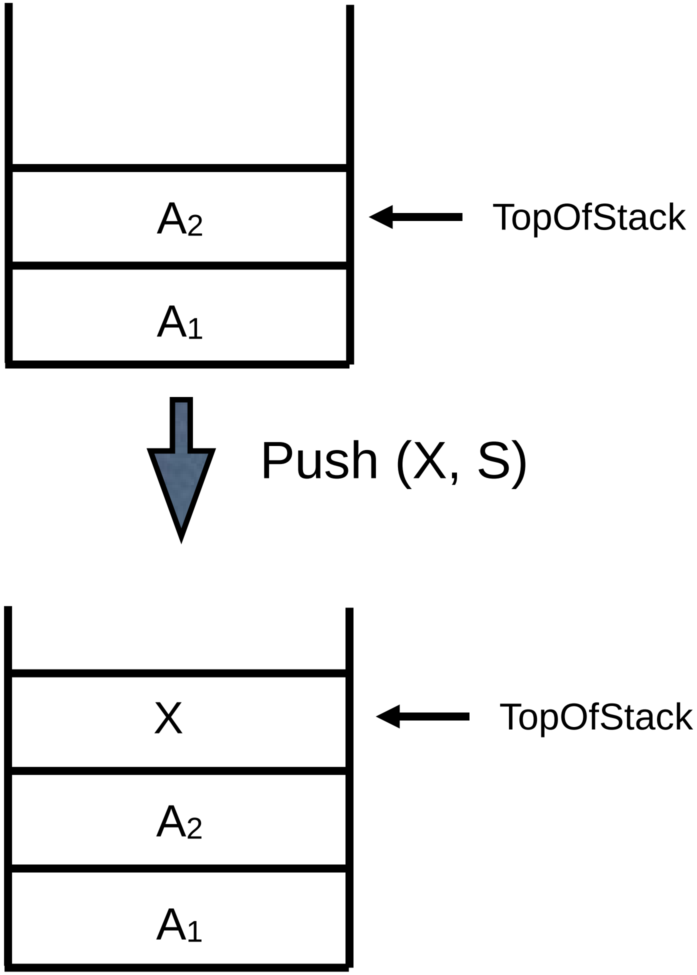
# Stack ADT: array implementation

```
#define EmptyTOS ( -1 )

Stack  CreateStack( int MaxElements )
{
    Stack S;

    S = malloc( sizeof( struct StackRecord ) );
    if( S == NULL )
          FatalError( “Out of space!!!” );

    S->Array = malloc( sizeof( ElementType ) * MaxElements );
    if( S->Array == NULL )
          FatalError( “Out of space!!!” );

    S->Capacity = MaxElements;
    S->TopOfStack = EmptyTOS;

    return S;
}
```



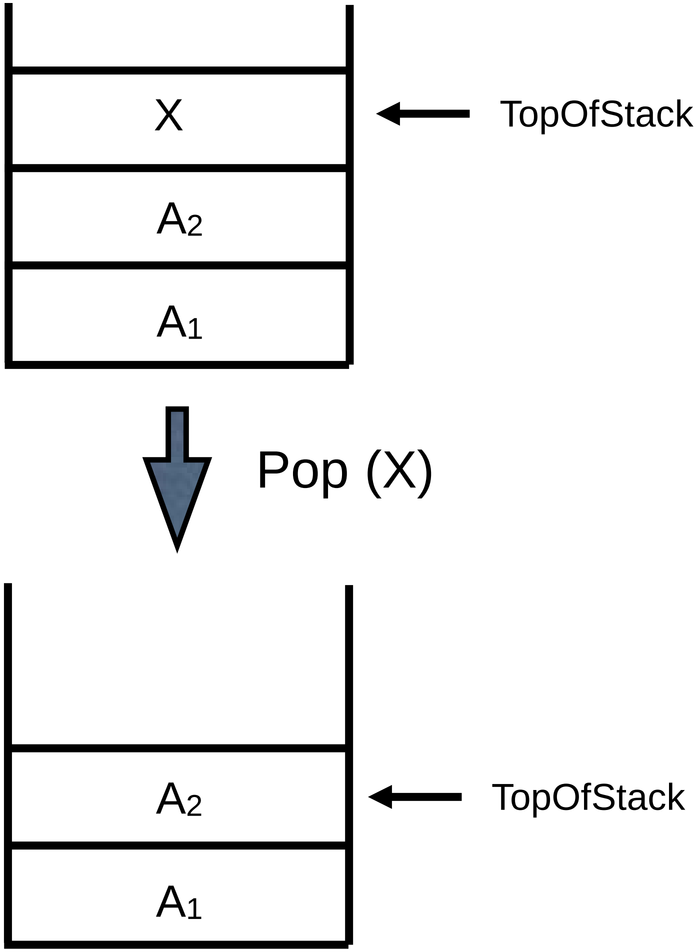
# Stack ADT: array implementation

```
void  Push( ElementType X, Stack S )
{
    if( IsFull( S ) )
        Error( “Full stack” );
    else
        S->Array[ ++S->TopOfStack ] = X;
}
```



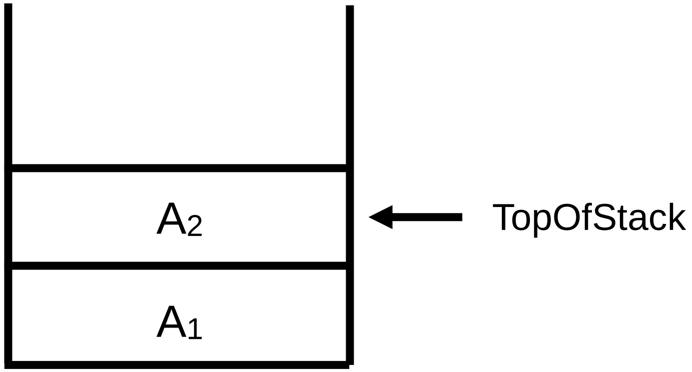
# Stack ADT: array implementation

```
void  Pop( Stack S )
{
    if( IsEmpty( S ) )
        Error( “Empty stack” );
    else
        S->TopOfStack--;
}
```

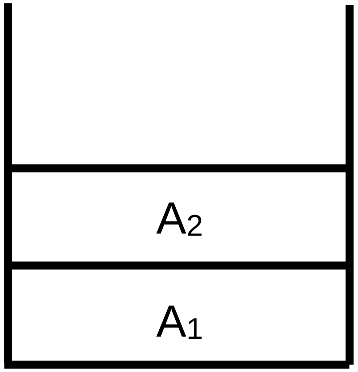


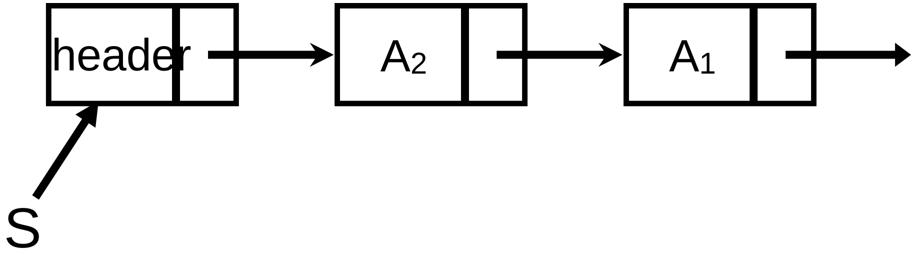
# Stack ADT: array implementation

```
ElementType  Top( Stack S)
{
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack ];

    Error( “Empty stack” );
    return 0;
}
```



# Stack ADT: linked list implementation

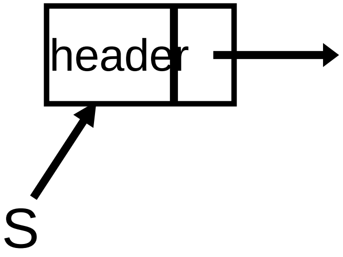




```
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode Stack;

struct Node{
        ElementType Element;
        PtrToNode Next;
        };
```

# Stack ADT: linked list implementation

```
Stack CreateStack (){

                Stack S;
                S = malloc(sizeof (struct Node));

                if (S==NULL)
                           FatalError("Out of space !!!");

                S -> Next = NULL;
                return S;
}
```

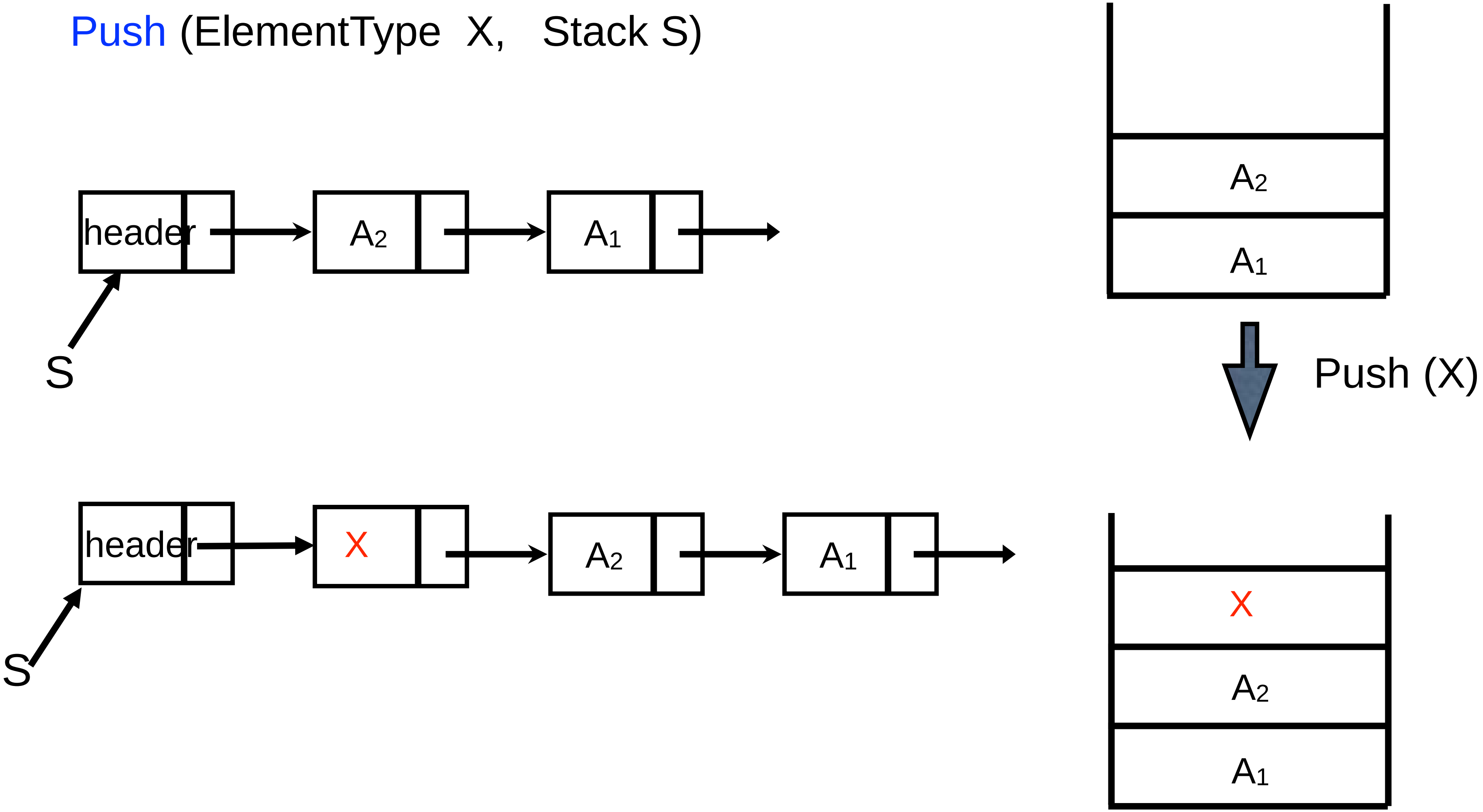


# Stack ADT: linked list implementation

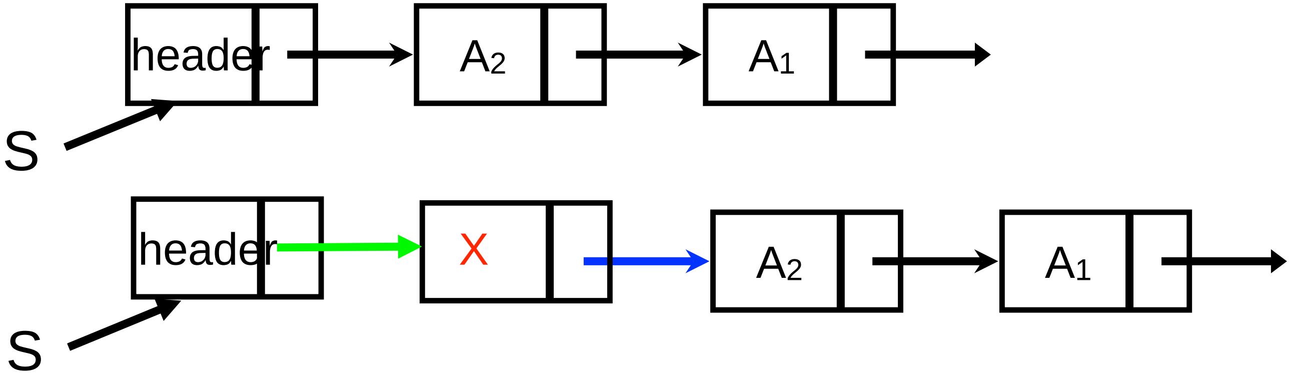
```
void MakeEmpty(Stack S)  {

        if  (S == NULL)
                Error (“No stack exists”);
        else
                while( !IsEmpty(S))
                    Pop(S);
}
```

# Stack ADT: linked list implementation

Push (ElementType X, Stack S)



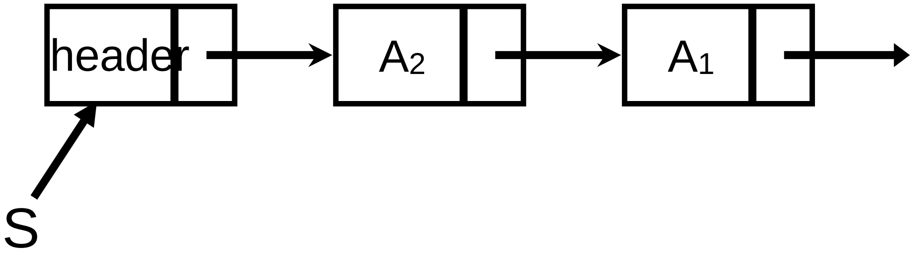
# Stack ADT: linked list implementation

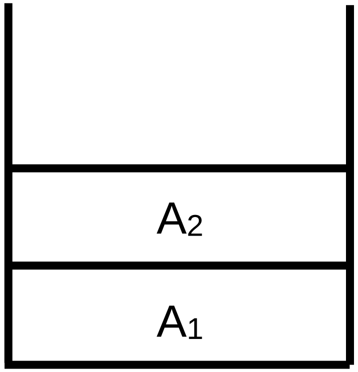


```
void Push (ElementType  X, Stack S)  {

            PtrToNode TmpCell;
            TmpCell = malloc (sizeof (struct Node));

            if (TmpCell ==NULL)  {
                        FatalError(“Out of space !!!”);
            } else {
                        TmpCell -> Element = X;
                        TmpCell -> Next = S -> Next;
                        S -> Next = TmpCell;
            }
}
```

# Stack ADT: linked list implementation

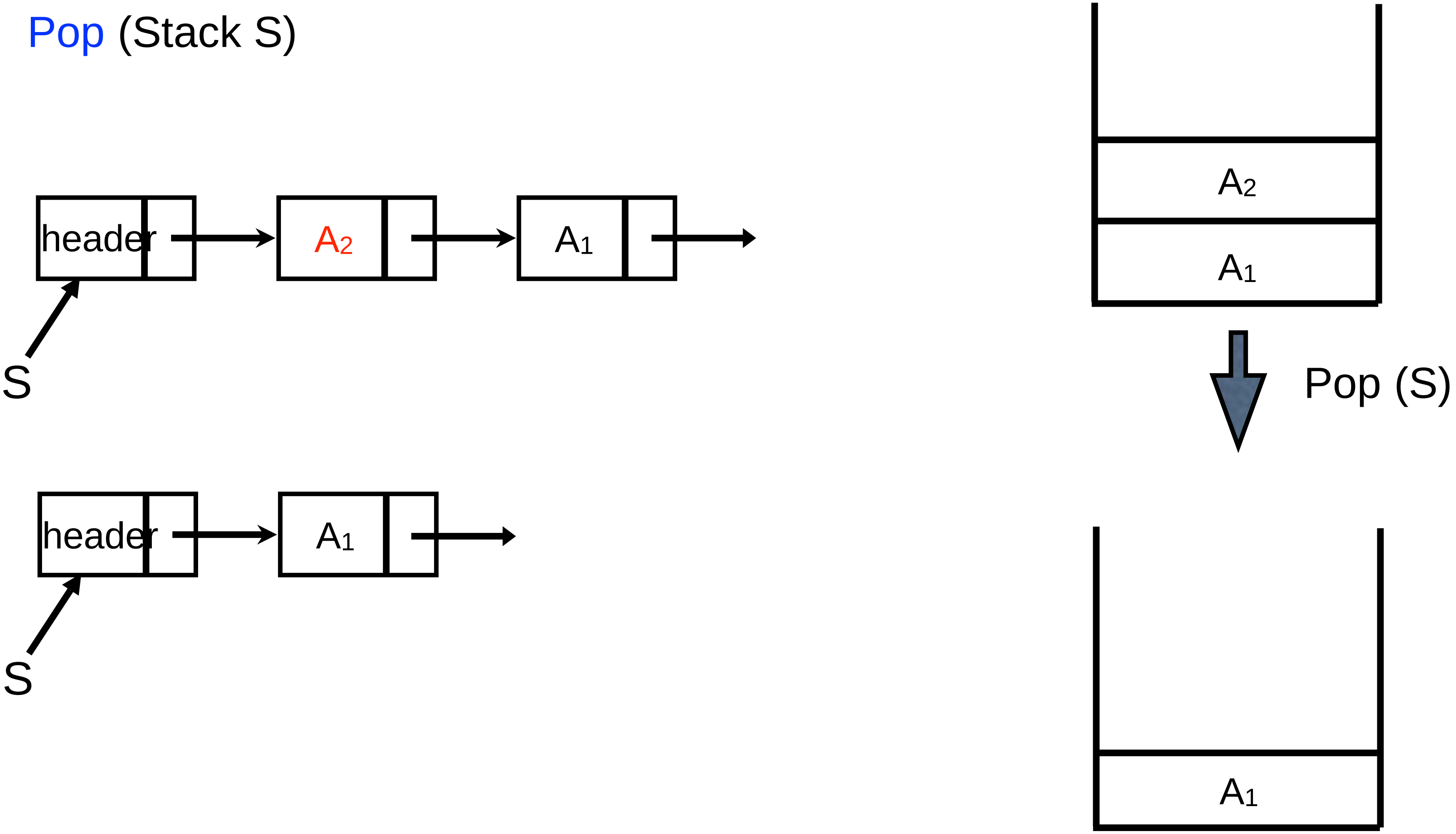
```
ElementType Top (Stack S)  {

        if (!IsEmpty(S))
                return    S->Next->Element;

        Error ("Empty stack");
        return 0;
}
```





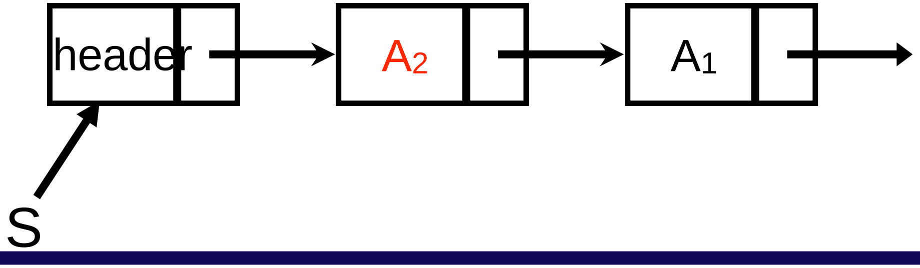
# Stack ADT: linked list implementation

Pop (Stack S)



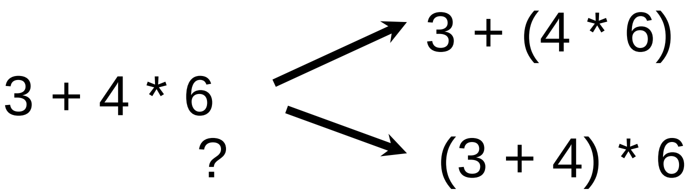
# Stack ADT: linked list implementation

```
void Pop (Stack S) {

        PtrToNode FirstCell;

        if (IsEmpty(S))
                Error(“Empty stack”);
        else{
                FirstCell = S->Next;
                S->Next = S->Next->Next;
                free(FirstCell);
        }
}
```



# infix, prefix, and postfix notation

infix



3 + 4 * 6

?

3 + (4 * 6)

(3 + 4) * 6

# infix, prefix, and postfix notation

infix

$$\begin{tikzcd} & 3 + (4 * 6) \\ 3 + 4 * 6 \; ? \arrow[ur] \arrow[dr] & \\ & (3 + 4) * 6 \end{tikzcd}$$

prefix

$(3 + 4) * 6 \longrightarrow * + 3\ 4\ 6$

$3 + (4 * 6) \longrightarrow + 3 * 4\ 6$

postfix

$(3 + 4) * 6 \longrightarrow 3\ 4 + 6 *$

$3 + (4 * 6) \longrightarrow 3\ 4\ 6 * +$

# postfix evaluation

$$7 \quad \underbrace{2 \quad 3 \quad *}_{2 * 3 = 6} \quad - \quad 4 \quad \uparrow \quad 9 \quad 3 \quad / \quad +$$

$$\underbrace{7 \quad 6 \quad -}_{7 - 6 = 1} \quad 4 \quad \uparrow \quad 9 \quad 3 \quad / \quad +$$

$$\underbrace{1 \quad 4 \quad \uparrow}_{1^4 = 1} \quad 9 \quad 3 \quad / \quad +$$

$$1 \quad \underbrace{9 \quad 3 \quad /}_{9 / 3 = 3} \quad +$$

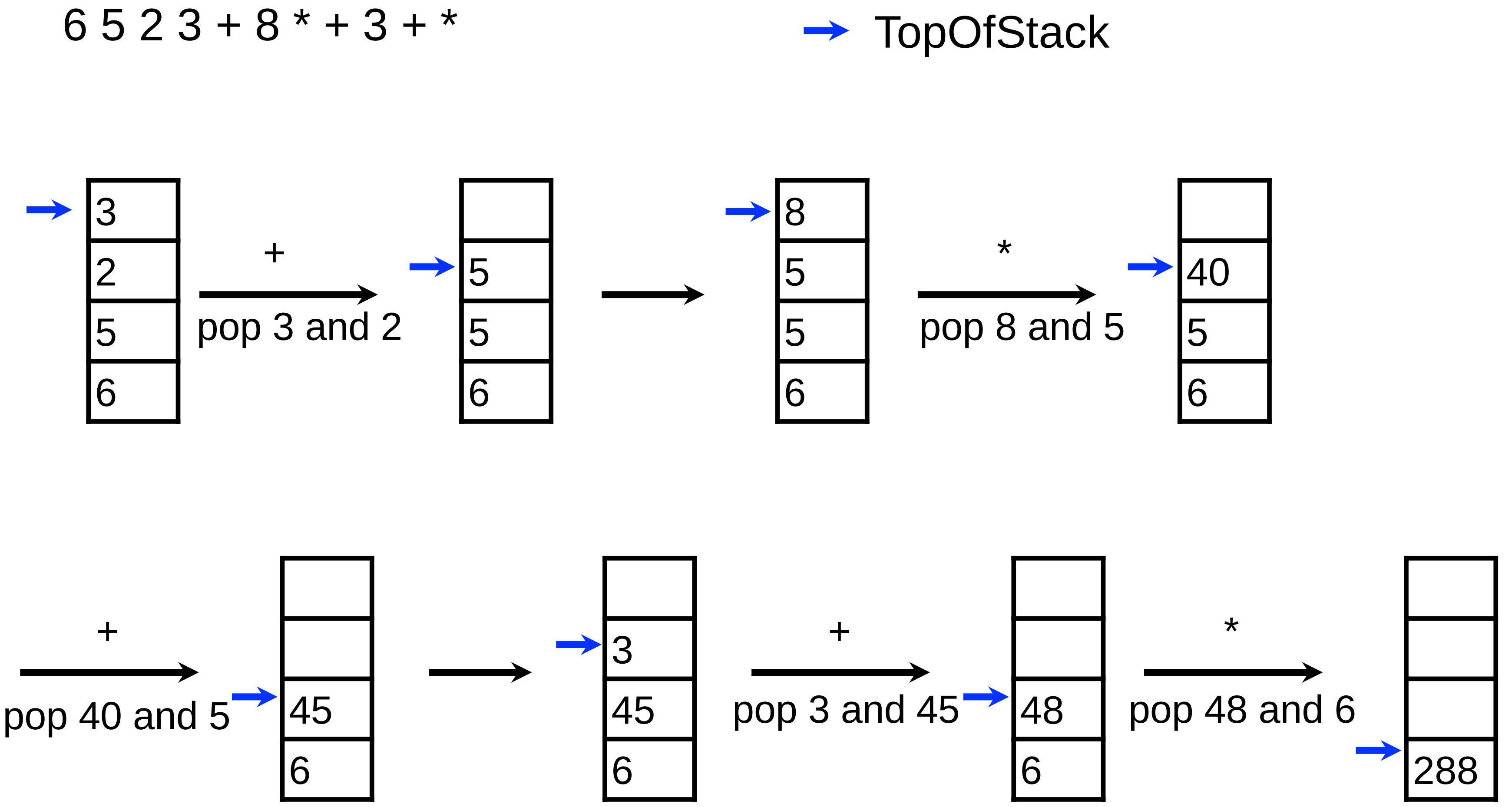
$$\underbrace{1 \quad 3 \quad +}_{1 + 3 = 4}$$

# Stack ADT: postfix evaluation

- scan left-to-right
- place the operands on a stack until an operator is found
- perform operations by popping two elements in the stack w hen an operator is found

# Stack ADT: postfix evaluation

6 5 2 3 + 8 * + 3 + *



# Stack ADT: prefix evaluation

how can we evaluate prefix expression?

- + * 2 3 + 5 4 9

# Stack ADT: translation of infix to postfix

3 + 4 * 6 → 3 4 6 * +

(3 + 4) * 6 → 3 4 + 6 *

3 + (4 * 6) → 3 4 6 * +

# Stack ADT: translation of infix to postfix

When you meet an operand, print it.

3 + 4 * 6 ⟶ 3 4 6 * +

(3 + 4) * 6 ⟶ 3 4 + 6 *

3 + (4 * 6) ⟶ 3 4 6 * +

# Stack ADT: translation of infix to postfix

When you meet an operand, print it.

When you meet an operator, push it as long as the precedence of the operator at the top of the stack is less than the precedence of the incoming operator. Otherwise, pop the top in the stack and print it.

3 + 4 * 6 → 3 4 6 * +

3 + 4 * 6 + 5 → 3 4 6 * + 5 +

# Stack ADT: translation of infix to postfix

When you meet an operand, print it.

When you meet an operator, push it as long as the precedence of the operator at the top of the stack is less than the precedence of the incoming operator. Otherwise, pop the top in the stack and print it.

When you meet the left parenthesis, push it in the stack.

When you meet the right parenthesis, pop all the operators until we reach the corresponding left parenthesis.

(3 + 4) * 6 ⟶ 3 4 + 6 *

# Stack ADT: translation of infix to postfix

When you meet an operand, print it.

When you meet an operator, push it as long as the precedence of the operator at the top of the stack is less than the precedence of the incoming operator. Otherwise, pop the top in the stack and print it.

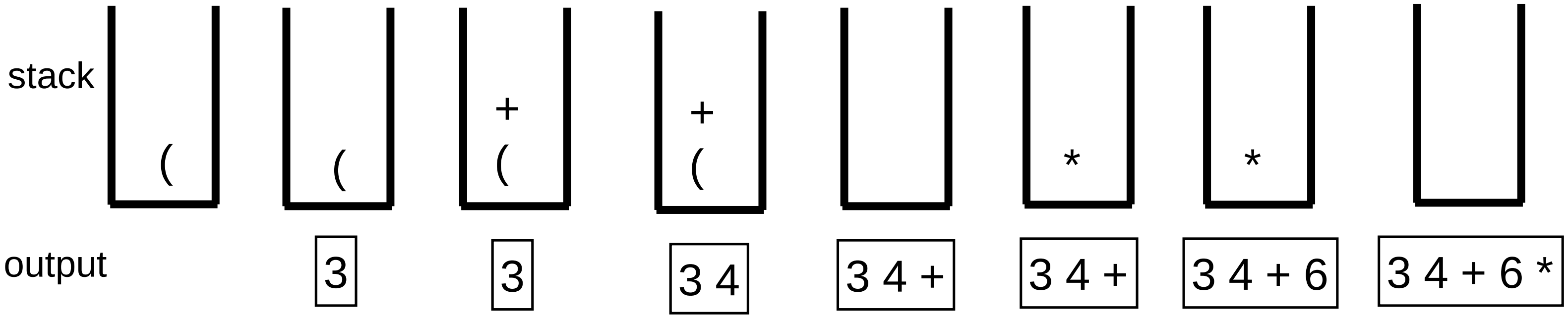
When you meet the left parenthesis, push it in the stack.

When you meet the right parenthesis, pop all the operators until we reach the corresponding left parenthesis.

When you reach the end of expression, pop all the operators from the stack.

# Stack ADT: translation of infix to postfix

(3 + 4) * 6 ⟶ 3 4 + 6 *



stack

(

(

+
(

+
(

*

*

output

3

3

3 4

3 4 +

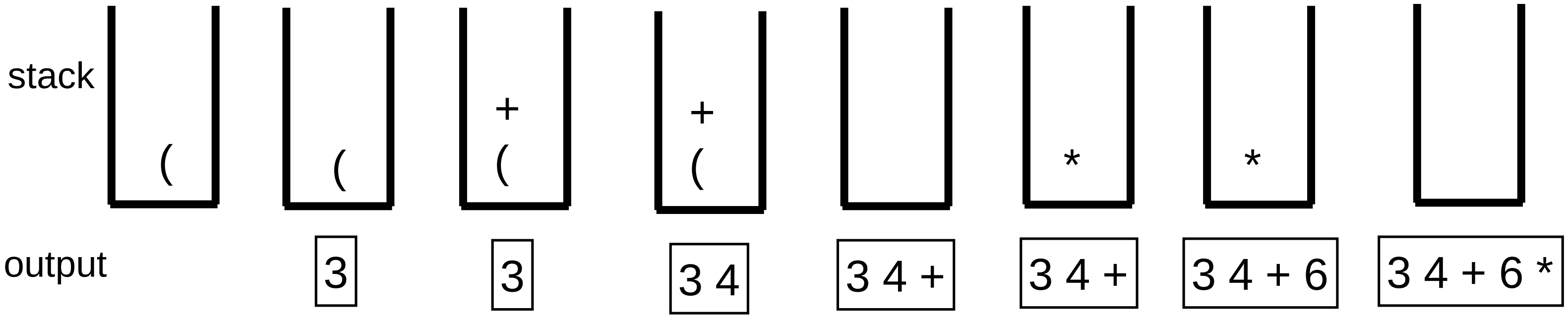
3 4 +

3 4 + 6

3 4 + 6 *

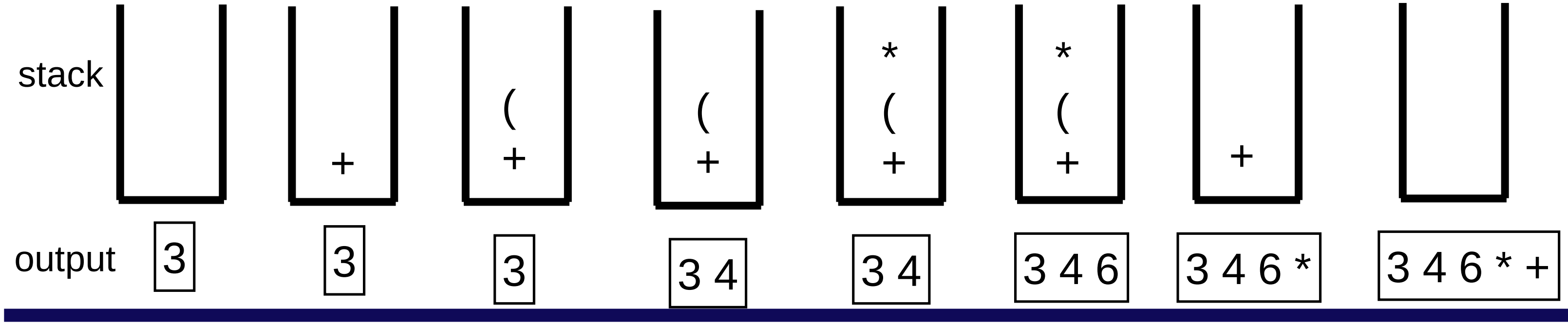
# Stack ADT: translation of infix to postfix

(3 + 4) * 6 → 3 4 + 6 *



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| stack | ( | ( | +<br>( | +<br>( | | * | * | |
| output | | 3 | 3 | 3 4 | 3 4 + | 3 4 + | 3 4 + 6 | 3 4 + 6 * |

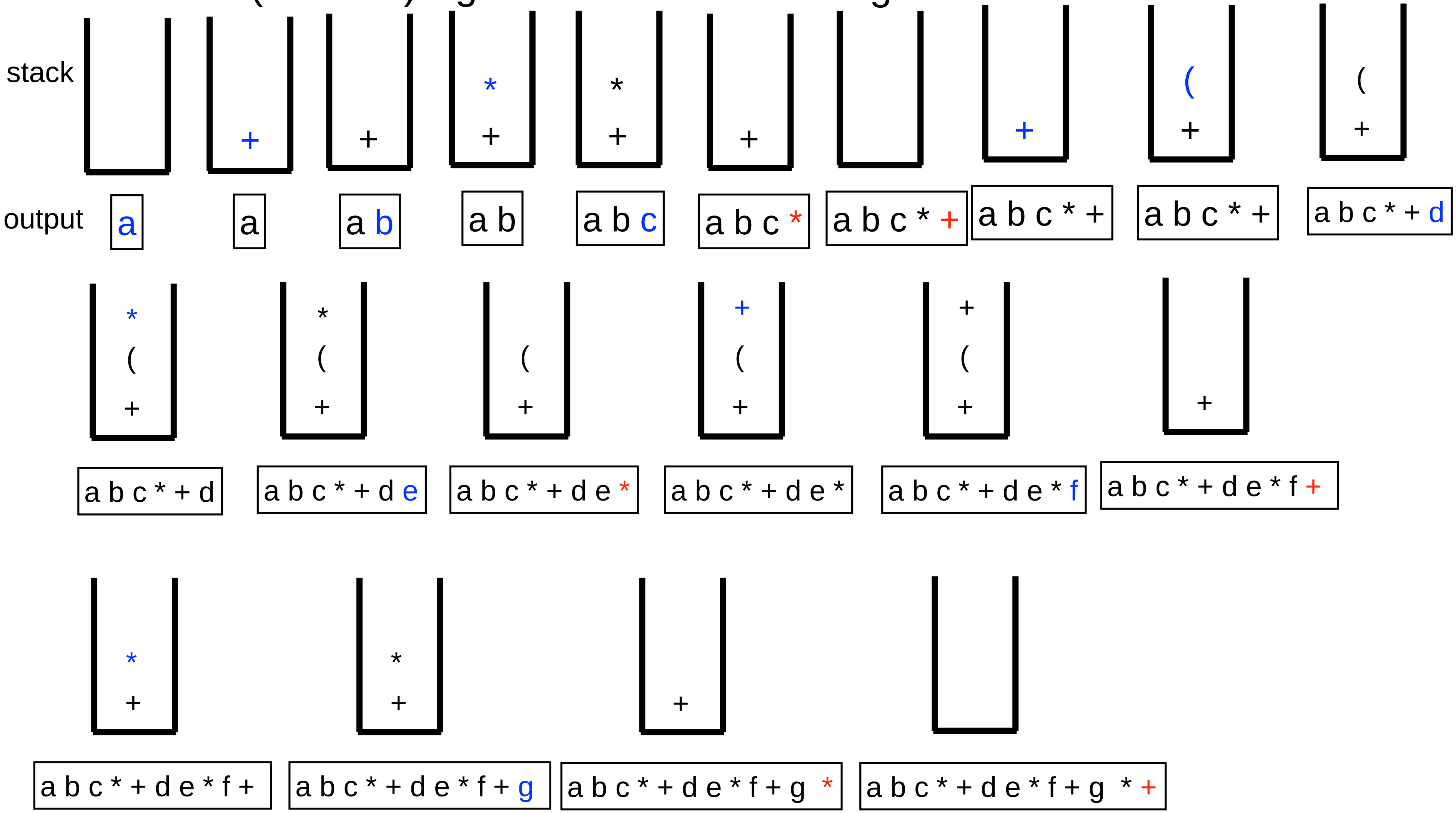
3 + (4 * 6) → 3 4 6 * +



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| stack | | + | (<br>+ | (<br>+ | *<br>(<br>+ | *<br>(<br>+ | + | |
| output | 3 | 3 | 3 | 3 4 | 3 4 | 3 4 6 | 3 4 6 * | 3 4 6 * + |

# Stack ADT: translation of infix to postfix

a + b * c + (d * e + f) * g → a b c * + d e * f + g * +



stack

output

a

+

a

+

a b

*
+

a b

*
+

a b c

+

a b c *

a b c * +

+

a b c * +

(
+

a b c * +

(
+

a b c * + d

*
(
+

a b c * + d

*
(
+

a b c * + d e

(
+

a b c * + d e *

+
(
+

a b c * + d e *

+
(
+

a b c * + d e * f

+

a b c * + d e * f +

*
+

a b c * + d e * f +

*
+

a b c * + d e * f + g

+

a b c * + d e * f + g *

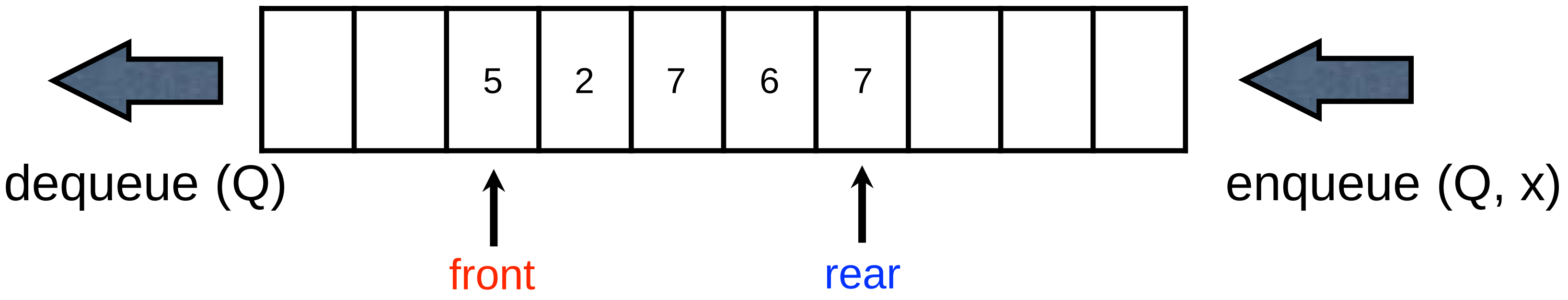
a b c * + d e * f + g * +

# Stack ADT: translation of infix to postfix

| infix | postfix |
|---|---|
| 2 + 3 * 4 | 2 3 4 * + |
| a * b + 5 | a b * 5 + |
| ( 1 + 2 ) * 7 | 1 2 + 7 * |
| a * b / c | a b * c / |
| ( ( a / ( b - c + d ) ) * ( e - a ) * c | a b c - d + / e a - * c * |
| a/b-c+d*e-a*c | a b / c - d e * + a c * - |

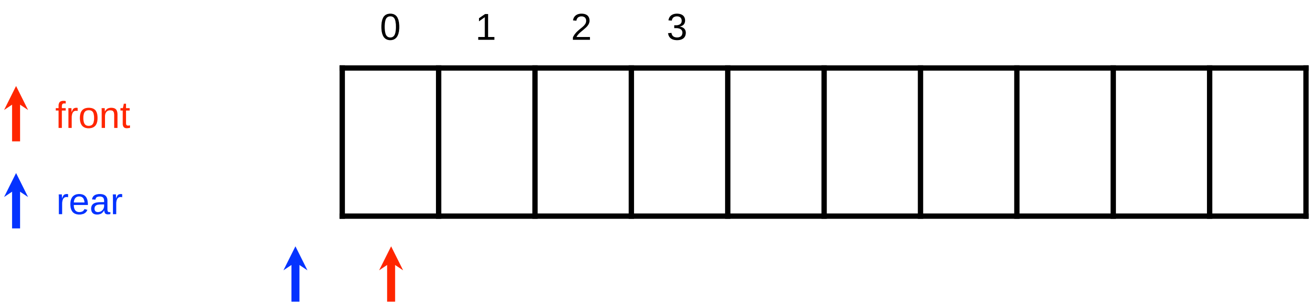
# Queue ADT

- a list that insertion is done at one end, whereas deletion is performed at the other end
- operations
  - enqueue: inserts an element at the end of the list
  - dequeue: deletes the element at the start of the list



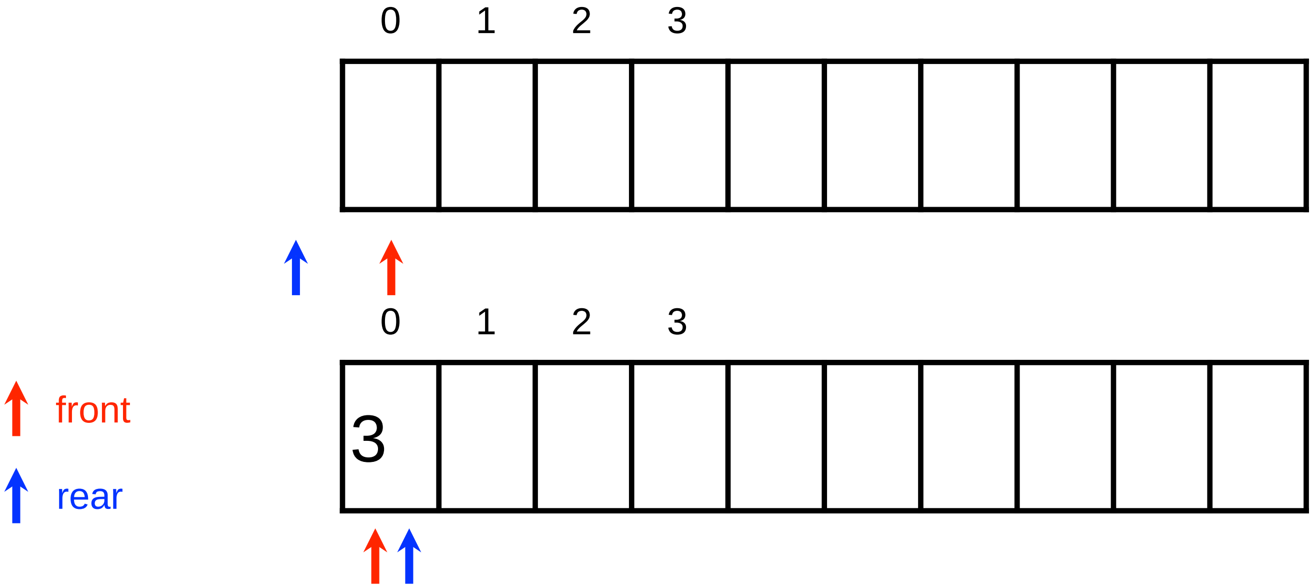
# Queue ADT

```
#define MAX_QUEUE_SIZE 100
typedef struct {
        int key;
} element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = 0;
```



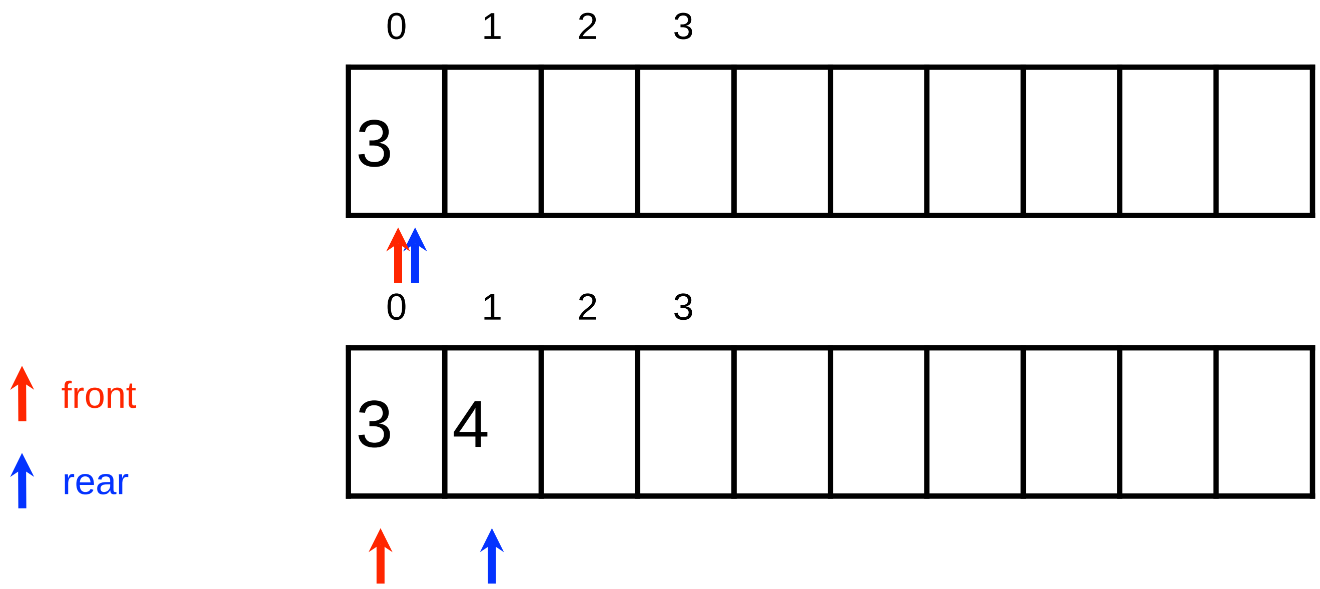
# Queue ADT

```
void Enqueue(element item, element* queue) {

    if (rear == MAX_QUEUE_SIZE - 1)
        queue_full();           /* error */
    queue[++rear] = item;
}
```



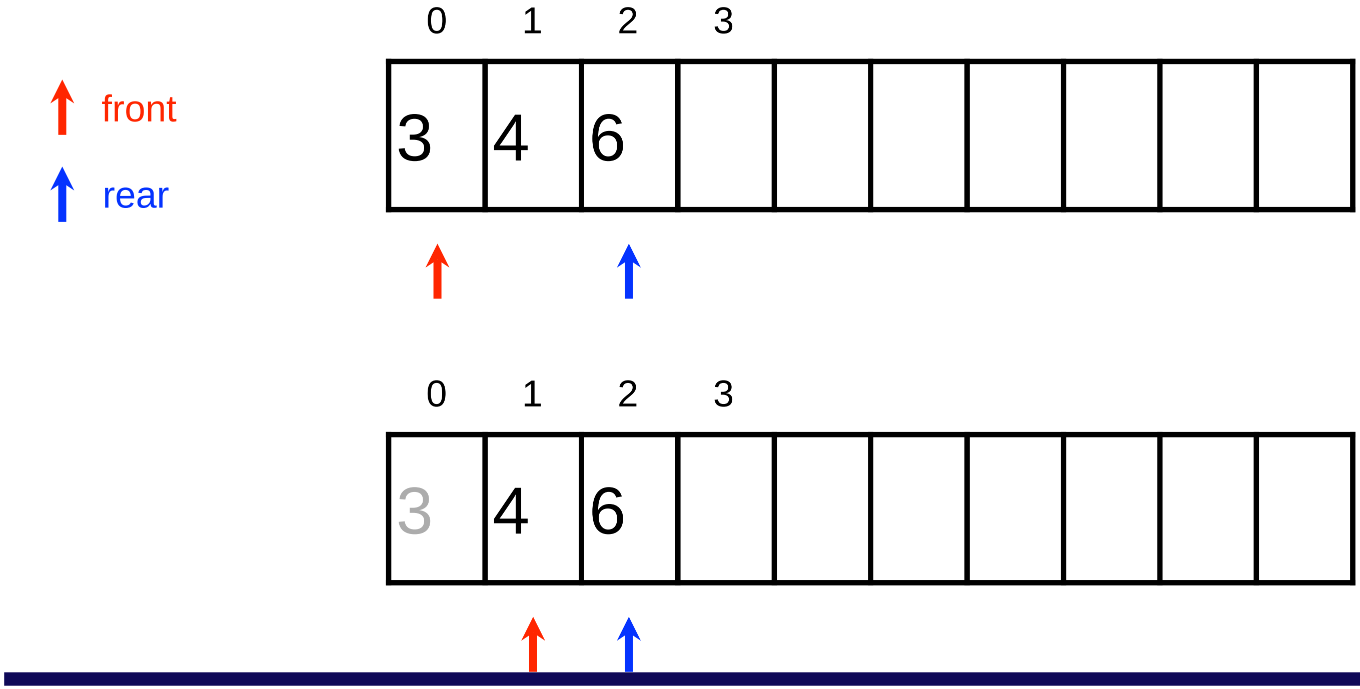
# Queue ADT

```
void Enqueue(element item, element* queue) {


    if (rear == MAX_QUEUE_SIZE - 1)
        queue_full();           /* error */
    queue[++rear] = item;
}
```

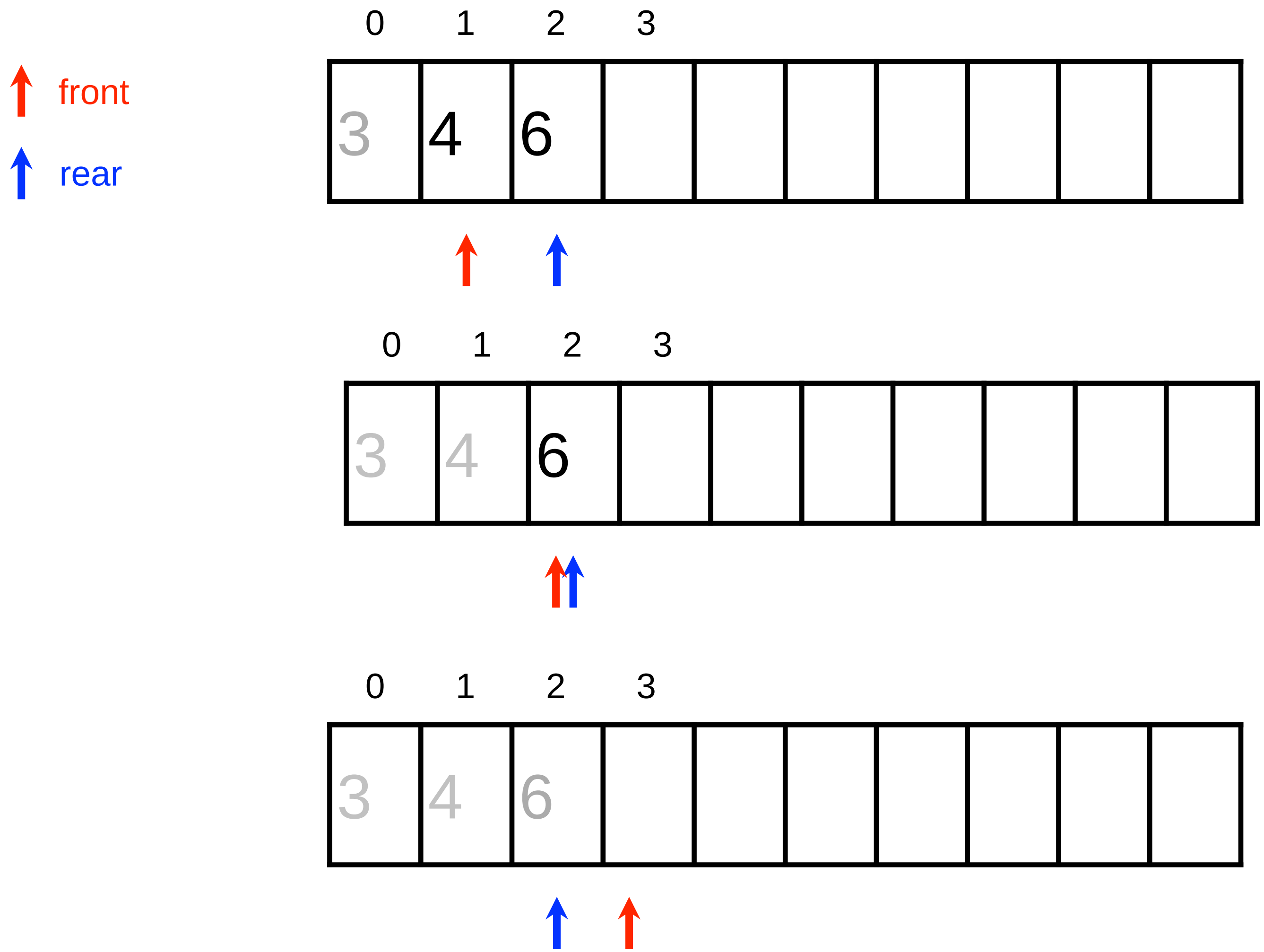


# Queue ADT

```
element Dequeue(element* queue) {
        if (front > rear)               /* empty */
                queue_empty();          /* error */
        return queue[front++];
}
```

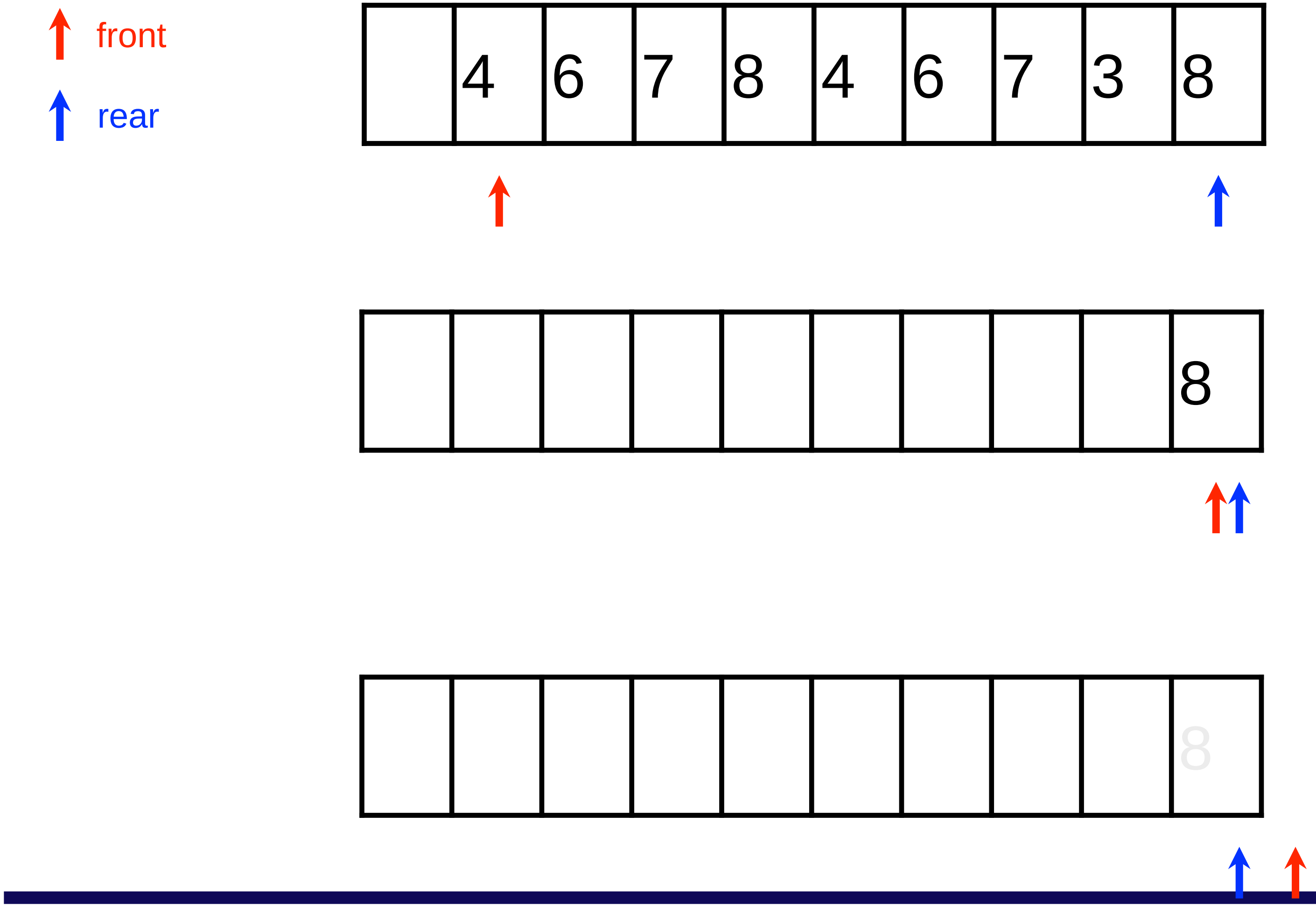


# Queue ADT



What is the condition for empty state?

# Queue ADT



# Queue ADT

Example [job scheduling]

- in the operating system which does not use priorities, jobs are processed in the order they enter the system
- insertion and deletion from a sequential queue

| front | rear | Q[0] | Q[1] | Q[2] | Q[3] | comments |
|---|---|---|---|---|---|---|
| 0 | -1 | | | | | queue is empty |
| 0 | 0 | Job1 | | | | Job1 is added |
| 0 | 1 | Job1 | Job2 | | | Job2 is added |
| 0 | 2 | Job1 | Job2 | Job3 | | Job3 is added |
| 1 | 2 | | Job2 | Job3 | | Job1 is deleted |
| 2 | 2 | | | Job3 | | Job2 is deleted |

# Queue ADT

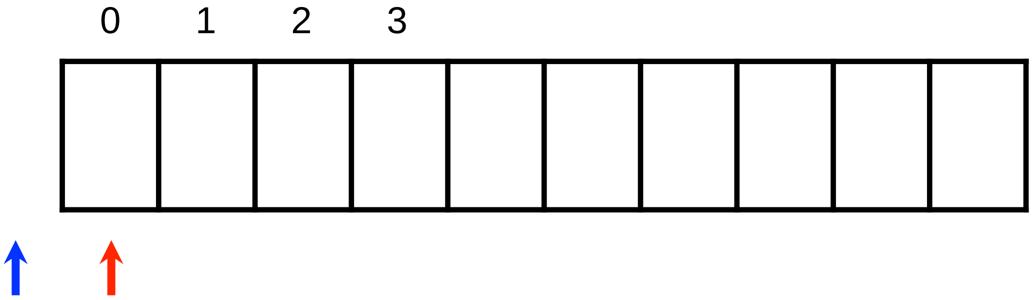
Example [job scheduling]

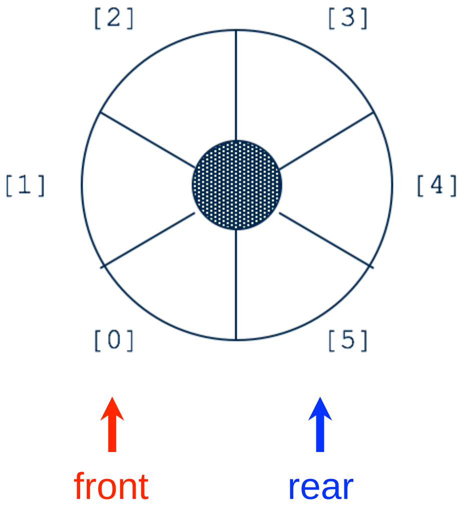
- in the operating system which does not use priorities, jobs are processed in the order they enter the system
- insertion and deletion from a sequential queue

| front | rear | Q[0] | Q[1] | Q[2] | Q[3] | comments |
|---|---|---|---|---|---|---|
| -1 | -1 | | | | | queue is empty |
| -1 | 0 | Job1 | | | | Job1 is added |
| -1 | 1 | Job1 | Job2 | | | Job2 is added |
| -1 | 2 | Job1 | Job2 | Job3 | | Job3 is added |
| 0 | 2 | | Job2 | Job3 | | Job1 is deleted |
| 1 | 2 | | | Job3 | | Job2 is deleted |

# Queue ADT: circular queue

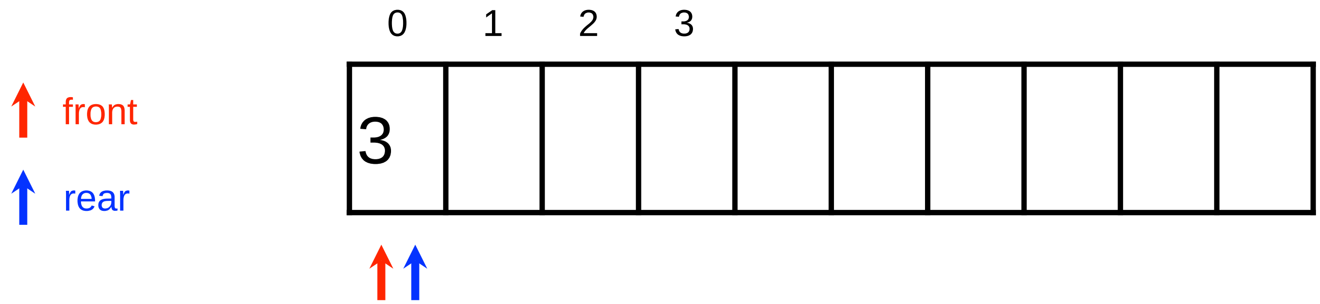
- when front or rear gets to the end of the array, it is wrapped around to the beginning

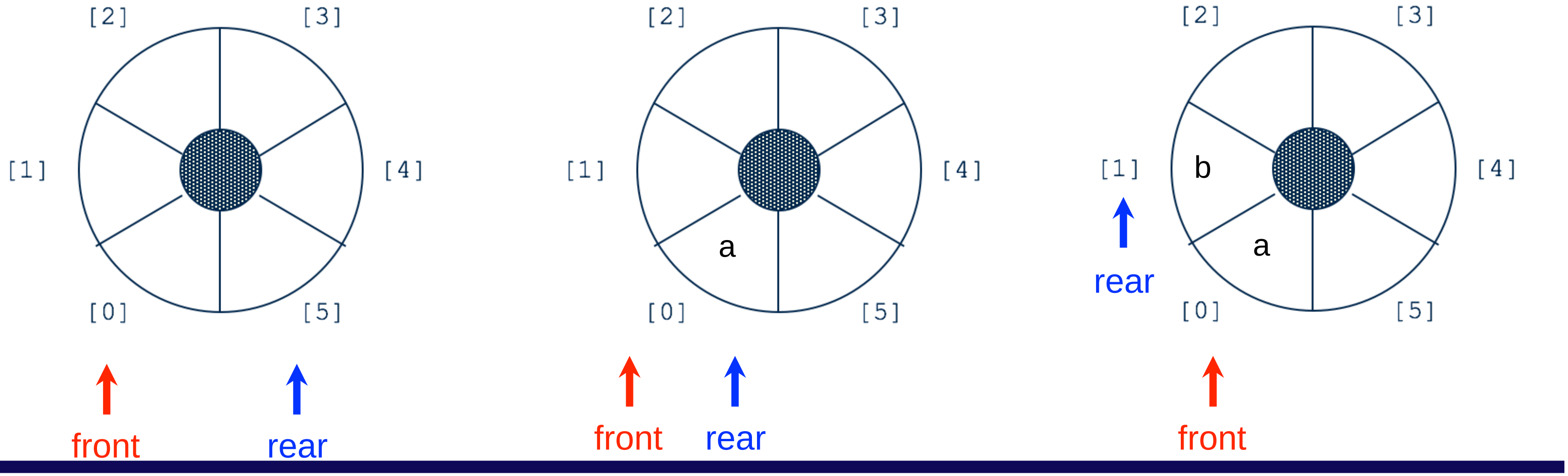




# Queue ADT: circular queue

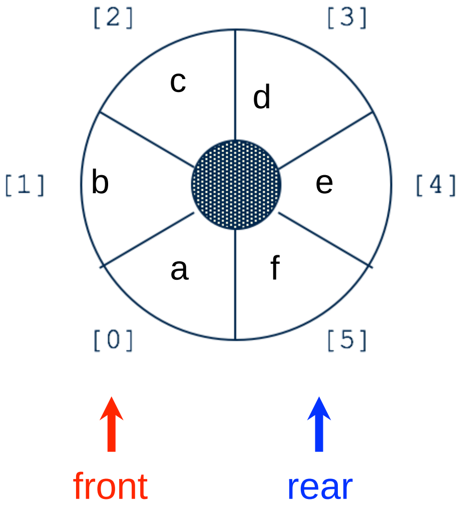
- when front or rear gets to the end of the array, it is wrapped around to the beginning



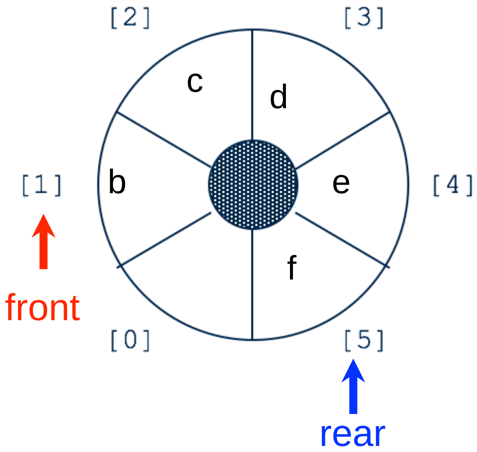


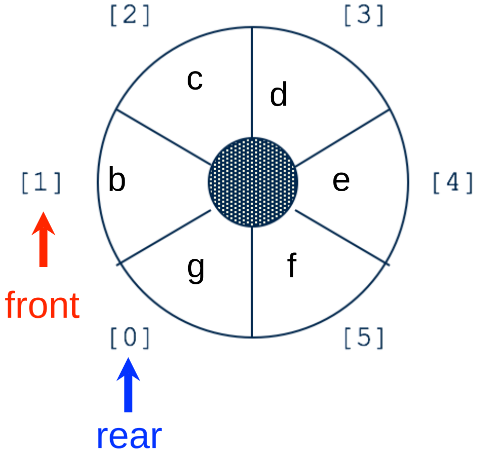
# Queue ADT: circular queue

- when circular queue is full



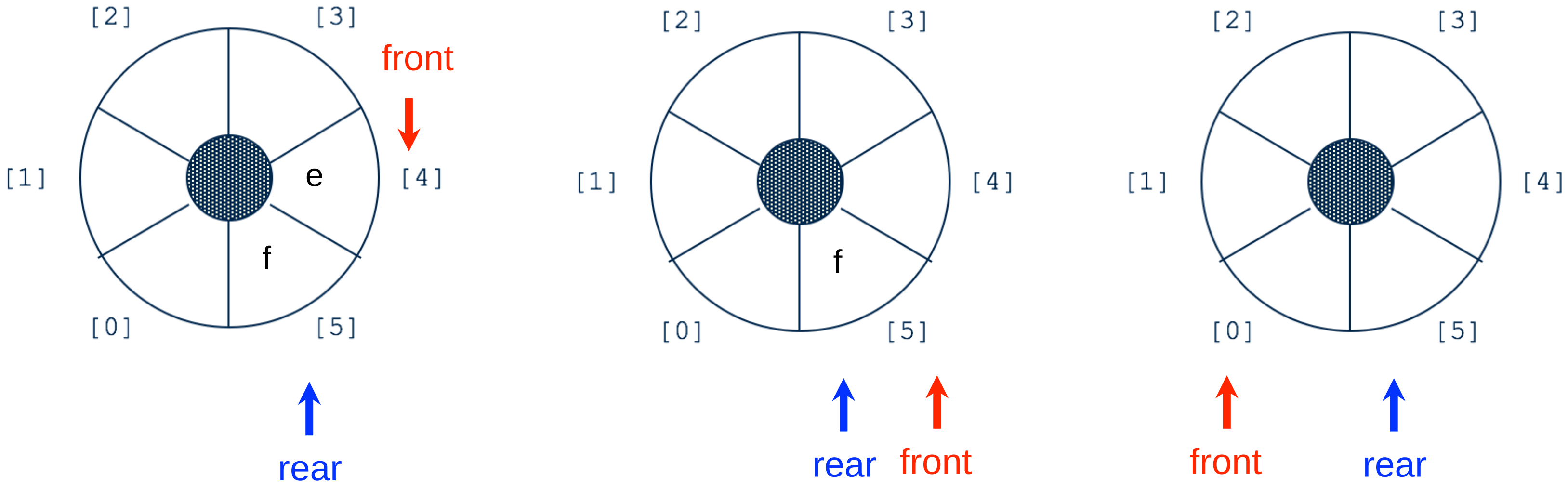
```
struct QueueRecord{

    int Capacity;
    int Front;
    int Rear;
    int Size;
    ElementType *Array;
};
```





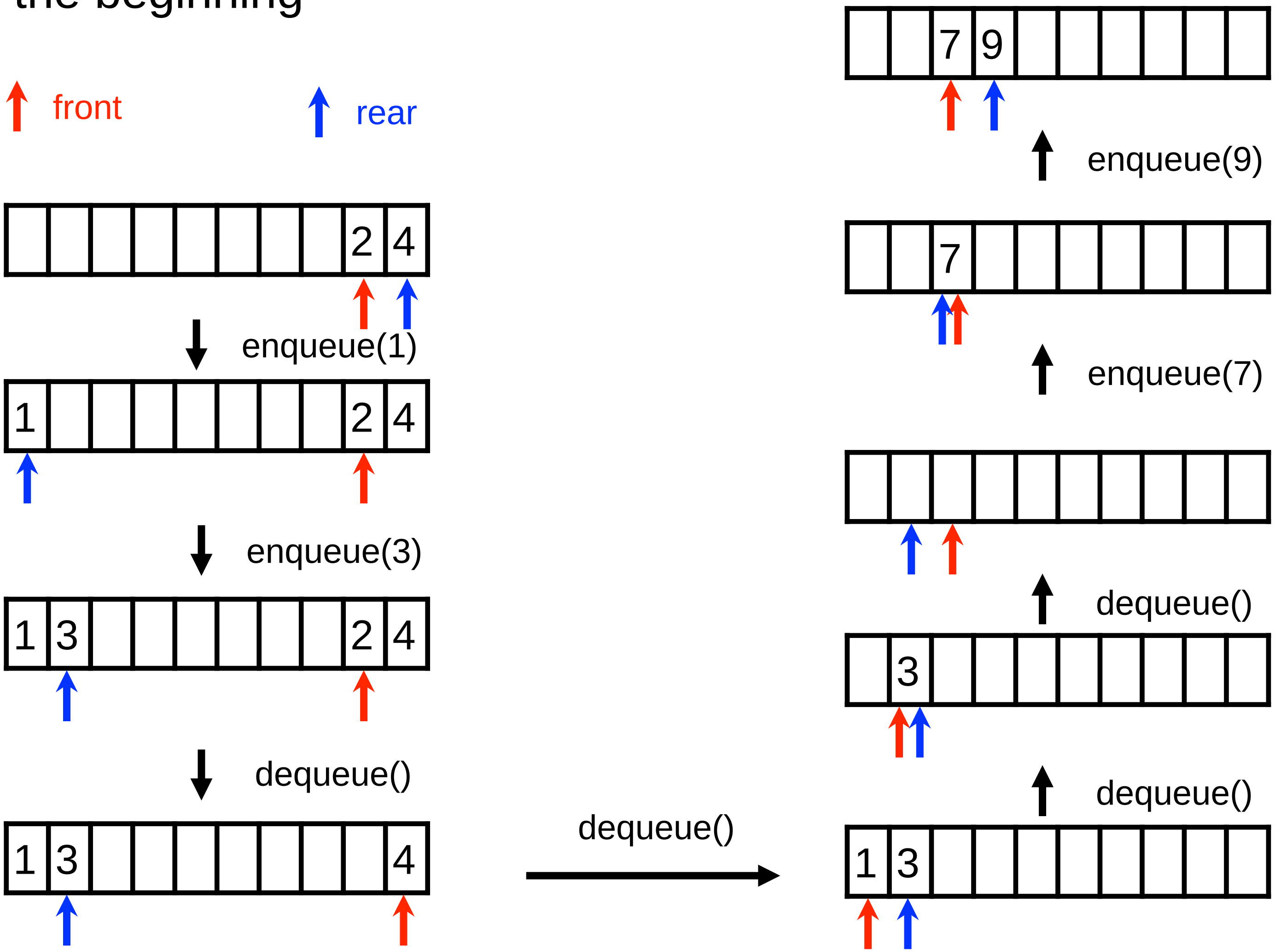
# Queue ADT: circular queue

- dequeue()



# Queue ADT: array implementation

- when front or rear gets to the end of the array, it is wrapped around to the beginning

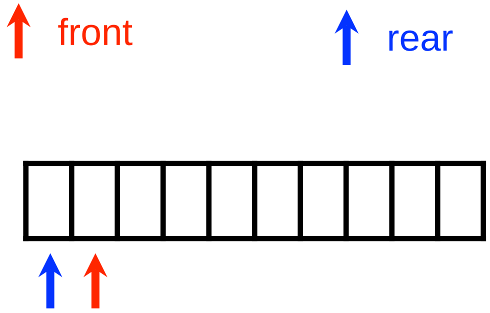


# Queue ADT: array implementation

```
struct QueueRecord;
typedef struct QueueRecord *Queue;

struct QueueRecord{

    int Capacity;
    int Front;
    int Rear;
    int Size;
    ElementType *Array;
};
```

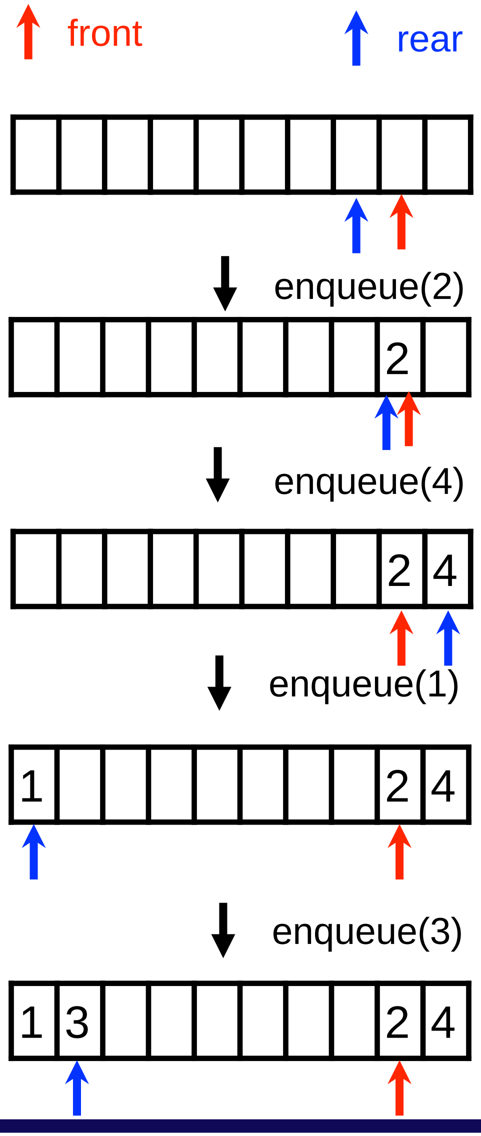
# Queue ADT: array implementation

```
void MakeEmpty (Queue Q){
    Q -> Size = 0;
    Q -> Front = 1;
    Q -> Rear = 0;
}
```



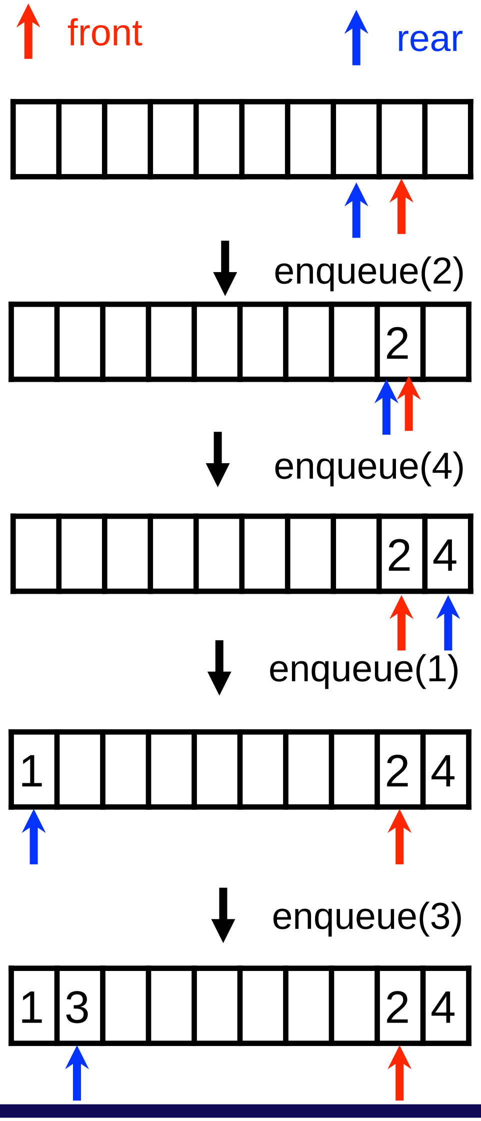
# Queue ADT: array implementation

```
static int Succ(int Value, Queue Q){
     if (++Value == Q->Capacity)
               Value = 0;
     return Value;
}

void Enqueue (ElementType X, Queue Q){
     if (IsFull(Q))
            Error("Full queue");
     else {
            Q -> Size ++;
            Q -> Rear = Succ(Q->Rear, Q);
            Q -> Array[Q->Rear] = X;
     }
}
```



# Queue ADT: array implementation

```
void Enqueue (ElementType X, Queue Q){
    if (IsFull(Q))
        Error(“Full queue”);
    else {
        Q -> Size ++;
        Q -> Rear = (Q->Rear + 1) % Q ->Capacity;
        Q -> Array[Q->Rear] = X;
    }
}
```


front
rear
enqueue(2)
2
enqueue(4)
2
4
enqueue(1)
1
2
4
enqueue(3)
1
3
2
4