ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY
한-베정보통신기술대학교

Nhân bản – Phụng sự – Khai phóng

Stacks and Queues

Data Structures & Algorithms

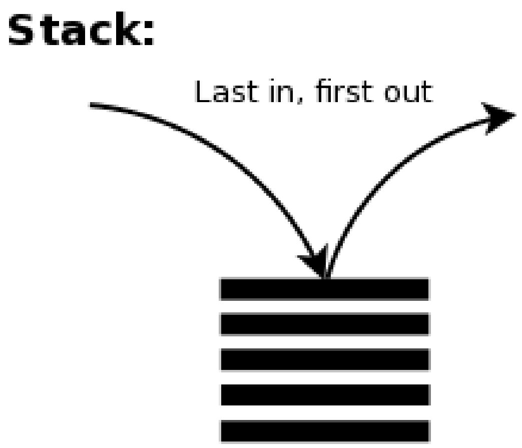
CONTENT

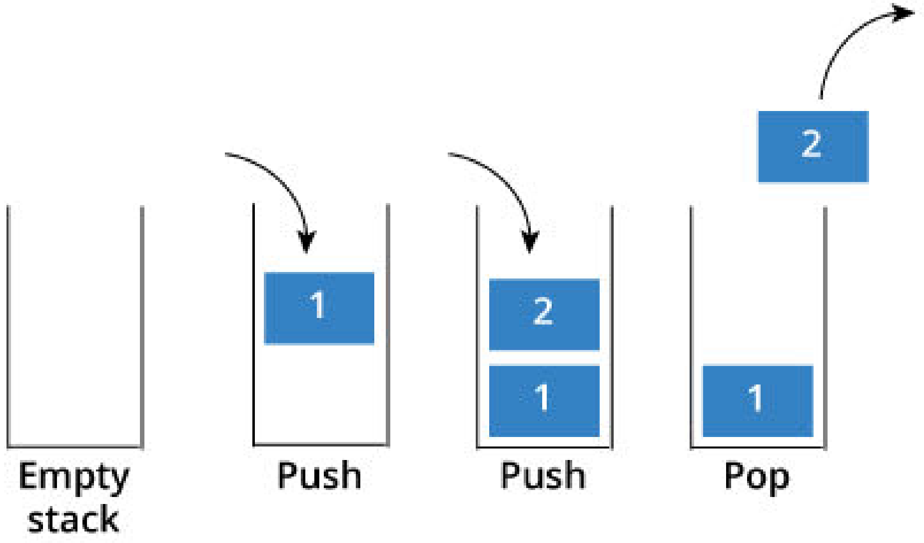
- Stacks
- Queues

CONTENT

- **Stacks**
  - Introduction to Stacks
  - Array representation of Stacks
  - Linked representation of Stacks
  - Applications of Stacks

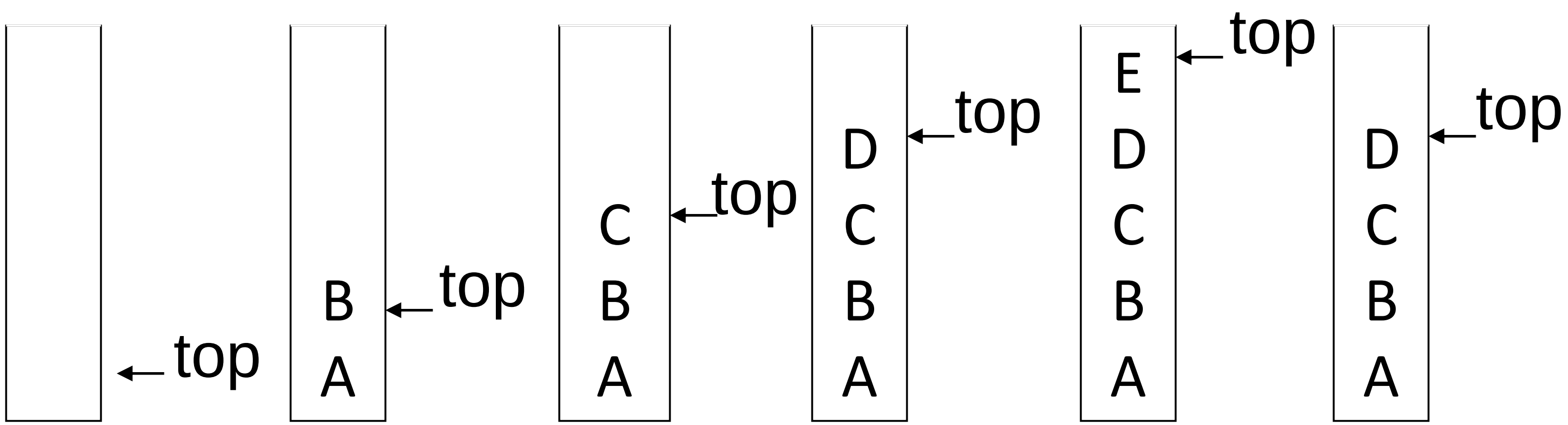
- **Introduction to Stacks**
  - Stack is a linear data structure
  - Elements in a stack are added/deleted only from one end (called **top**)
  - Stack is called a **LIFO** (Last-In-First-Out) data structure
  - Operations on a stack: **push**(), **pop**()





Stacks can be implemented using either **arrays** or **linked lists**

- **Introduction to Stacks**
  - Last-In-First-Out (LIFO)



- **Introduction to Stacks**
  - Example of function call

```
#include <stdio.h>
main(){
        int x;
        x = fact(5);
}

int fact(int n){
        if (n>1)
                return n*fact(n-1);
        else
                return 1;
}
```

X = ?

invoke fact(5)
invoke fact(4)
invoke fact(3)
invoke fact(2)
invoke fact(1)
return from fact(1) = 1
return from fact(2) = 2
return from fact(3) = 6
return from fact(4) = 24
return from fact(5) = 120

- **Array representation of stacks**
  - Variable **top** stores the address of the topmost element of the stack, the element will be added to or deleted from **top**
  - Variable MAX is used to store the maximum number of elements that the stack can hold.

    ⇨ **top** = -1: the stack is empty;

    top = MAX–1: the stack is full
  - Example:

A	AB	ABC	ABCD	ABCDE					
0	1	2	3	**Top = 4**	5	6	7	8	9

  - **top** = 4, so insertions/deletions will be done at this position.
  - five more elements can still be stored.

- **Array representation of stacks**

```
#define MAX 100
typedef  struct {
            int key;
            /* other fields */
} element;

element   stack[MAX];
```

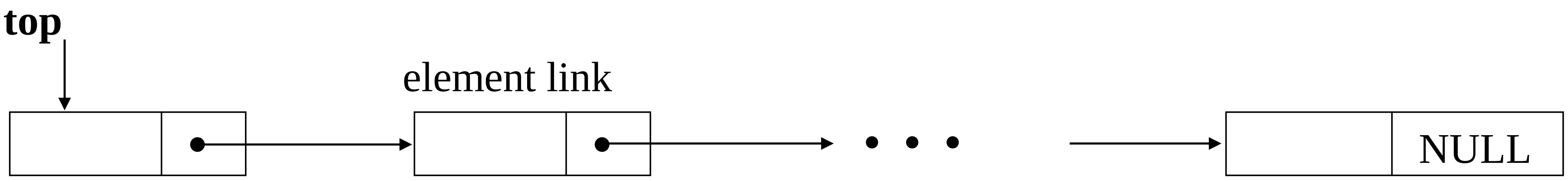
- void **push**(int top, element item){

```
    if (top == MAX-1)  return stack_full( );
    stack[++top] = item;
}
```

- element **pop**(){

```
    if (top == -1) return stack_empty( );
    return stack[top--];
}
```

- **Linked representation of Stacks**



- Every node has two parts: data & the address of the next node
- The start pointer of the linked list is used as **top**.
- Additions/deletions are done at the node pointed by **top**.
- **top** = NULL: the stack is empty

- **Linked representation of Stacks**
  - Declarations

```
typedef struct stack *stack_pointer;
typedef struct stack {
        element item;
        stack_pointer link;
};
```

```
typedef struct {
        int key;
        /* other fields */
} element;
```

  - Boundary conditions
    - **top** = NULL iff the stack is empty;
    - IS_EMPTY(temp) iff the stack is empty;
    - IS_FULL(temp) iff the memory is full

- **Linked representation of Stacks**

```
void push(stack_pointer *top, element itemp){
        /* add an element to the top of the stack */
        stack_pointer temp =  (stack_pointer) malloc (sizeof (stack));
        if (IS_FULL(temp)) {
                fprintf(stderr, " The memory is full\n");
                exit(1);
        }
        temp->item = itemp;
        temp->link = *top;
        *top= temp;
}
```

- **Linked representation of Stacks**

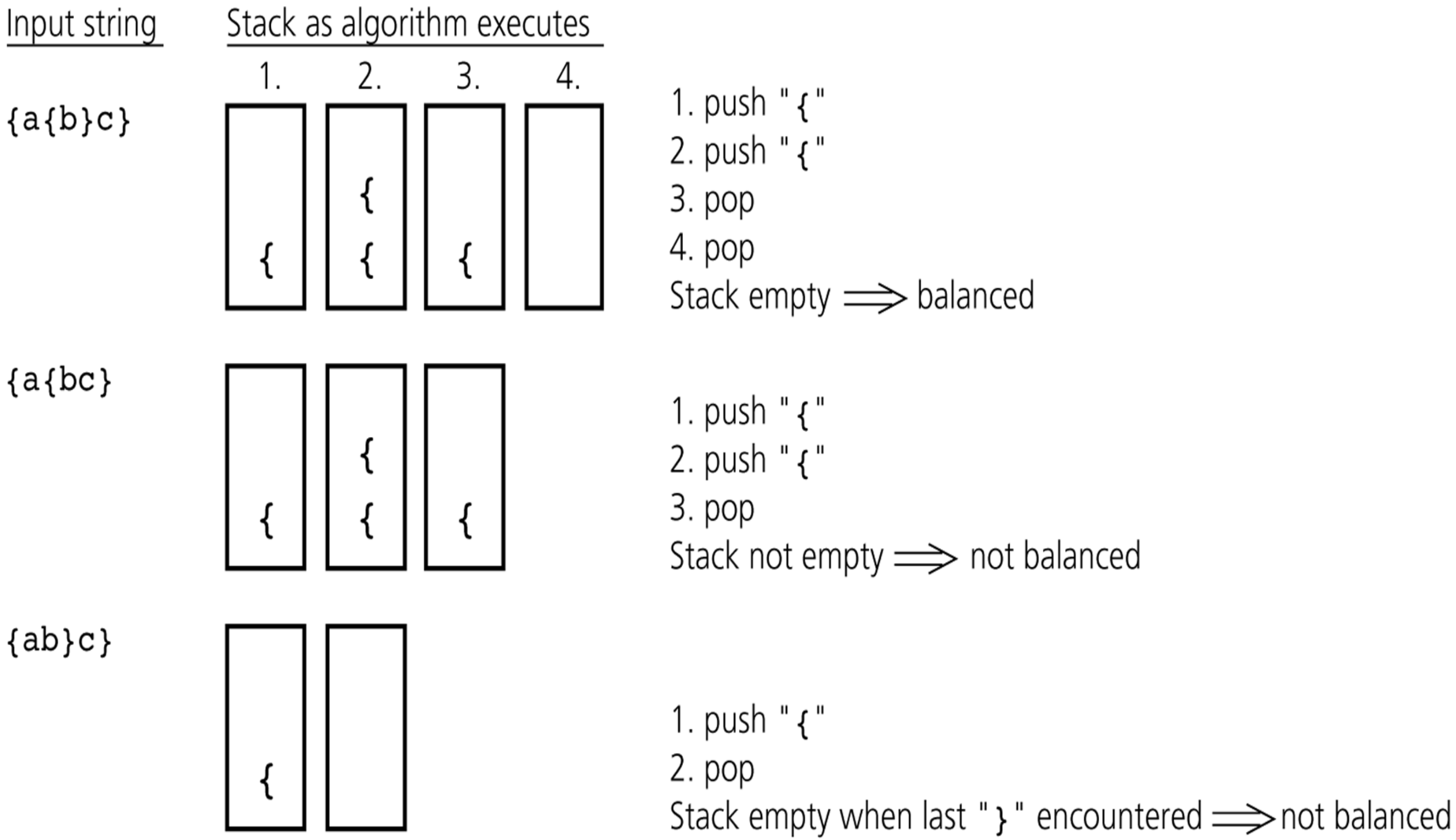
```
element pop(stack_pointer *top) {
        /* delete an element from the stack */
        stack_pointer temp = *top;
        if (IS_EMPTY(temp))  {
                fprintf(stderr,  "The stack is empty\n");
                exit(1);
        }
        element itemp;
        itemp = temp->item;
        *top = temp->link;
        free(temp);
        return itemp;
}
```

- **Comparing representations**
  - Array representation of Stacks
    - Fixed size (cannot grow and shrink dynamically)
  - Linked representation of Stacks
    - May need to perform realloc() calls when the currently allocated size is exceeded
    - But push and pop operations can be very fast

- **Applications of Stacks**
  - Reversing a list
  - Parentheses checker
  - Matching parentheses and HTML Tags
  - Conversion of an infix expression into a postfix expression
  - Evaluation of a postfix expression
  - Conversion of an infix expression into a prefix expression
  - Evaluation of a prefix expression
  - Recursion
  - Tower of Hanoi
  - …

- Applications of Stacks - **Checking for Balanced Braces**
  - A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces
    - abc{defg{ijk}{l{mn}}op}qr
  - An example of unbalanced braces
    - abc{def}}{ghij{kl}m
  - Requirements for balanced braces
    - Each time we encounter a "}", it matches an already encountered "{"
    - When we reach the end of the string, we have matched each "{"

- Applications of Stacks - **Checking for Balanced Braces**



- Applications of Stacks - **Algebraic Expressions**
  - To evaluate an infix expression *//infix: operator in b/w operands*

1. Convert the infix expression to postfix form
2. Evaluate the postfix expression *//**post**fix: operator **after** operands; similarly we have **pre**fix: operator **before** operands*

Infix Expression	Postfix Expression	Prefix Expression
5 + 2 * 3	5 2 3 * +	+ 5 * 2 3
5 * 2 + 3	5 2 * 3 +	+ * 5 2 3
5 * (2 + 3) - 4	5 2 3 + * 4 -	- * 5 + 2 3 4
(5-2)/3	???	???
5/3+2*4	???	???
((1-2)*3+4)/5	???	???

- Applications of Stacks - **Algebraic Expressions**

  - Infix notation is easy to read for humans
  - Pre-/postfix notation is easier to parse for a machine
  - The big advantage in pre-/postfix notation is that there never arise any questions like operator precedence

- Applications of Stacks - **Algebraic Expressions**
  - Evaluating Postfix Expressions
    - When an operand is entered, the calculator
      - Pushes it onto a stack
    - When an operator is entered, the calculator
      - Applies it to the top two operands of the stack
      - Pops the operands from the stack
      - Pushes the result of the operation on the stack

- Applications of Stacks - **Algebraic Expressions**
  - Evaluating Postfix Expressions: 2 3 4 + *

Key entered	Calculator action		After stack operation: Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

- Applications of Stacks - **Algebraic Expressions**
  - Converting Infix Expressions to Postfix Expressions
    - Read the infix expression
      - When an operand is entered, append it to the end of postfix expression
      - When an ’(‘ is entered, push it into the stack
      - When an ’)‘ is entered, move operators from the stack to the end of postfix expression until ’(‘
      - When an operator is entered, push it into the stack
    - Move the operators in the stack to the end of postfix expression

- Applications of Stacks - **Algebraic Expressions**
  - Converting Infix Expressions to Postfix Expressions

ch	Stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

a - (b + c * d)/ e
⇨ a b c d * + e / -

- Applications of Stacks - **Algebraic Expressions**
  - Converting Infix Expressions to Postfix Expressions

```
for (each character ch in the infix expression) {
   switch (ch) {
     case operand:   // append operand to end of postfixExpr
         postfixExpr=postfixExpr+ch;   break;
     case '(':       // save '(' on stack
         aStack.push(ch);   break;
     case ')':    // pop stack until matching '(', and remove '('
         while (top of stack is not '(') {
             postfixExpr=postfixExpr+(top of stack);
             aStack.pop();
         }
         aStack.pop();   break;
```

- Applications of Stacks - **Algebraic Expressions**
    - Converting Infix Expressions to Postfix Expressions

```
   case operator:
          aStack.push();    break;     // save new operator
  } } // end of switch and for

  // append the operators in the stack to postfixExpr
  while (!isStack.isEmpty()) {
        postfixExpr=postfixExpr + (top of stack);
        aStack.pop();
  }
```

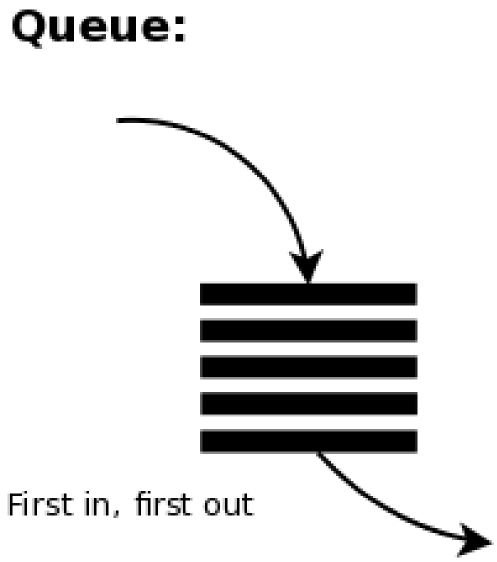
- Applications of Stacks - **Algebraic Expressions**
  - Benefits about converting from infix to postfix
    - Operands always stay in the same order with respect to one another
    - An operator will move only "to the right" with respect to the operands
    - All parentheses are removed

- **The Relationship Between Stacks and Recursion**
  - A strong relationship exists between recursion and stacks
  - Typically, stacks are used by compilers to implement recursive methods
    - During execution, each recursive call generates an activation record that is pushed onto a stack
    - We can get **stack overflow** error if a function makes too many recursive calls
  - Stacks can be used to implement a non recursive version of a recursive algorithm

CONTENT

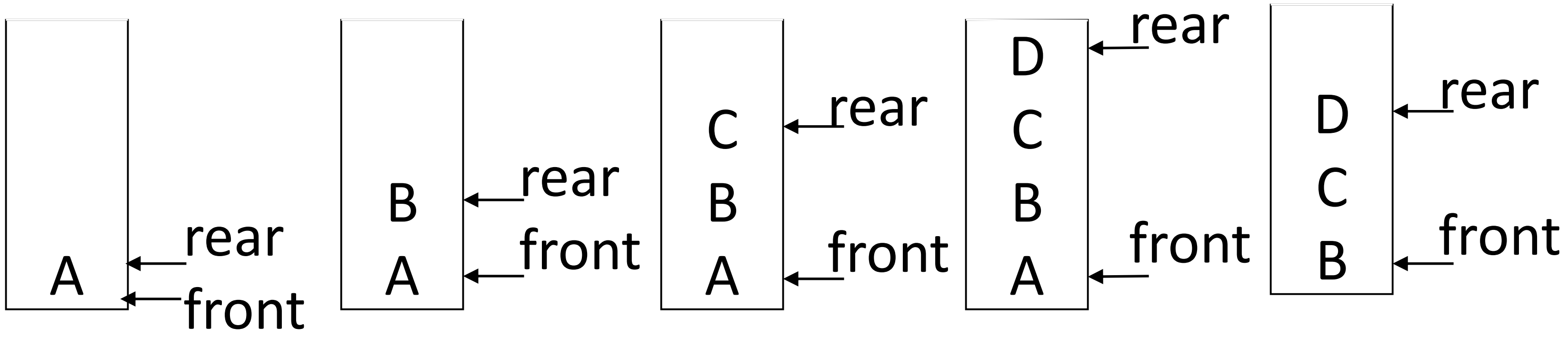
- **Queues**
  - Introduction to Queues
  - Array representation of Queues
  - Linked representation of Queues
  - Applications of Queues

- **Introduction to Queues**
  - The elements are added at one end (called **rear**) and deleted from the other end (called **front**).
  - Queue is a FIFO (First-In, First-Out) data structure
  - Operations on a queue: **add**(), **delete**()



✍ Queues can be implemented by using **arrays** or **linked lists**.

- **Introduction to Queues**
  - First-In-First-Out (FIFO) list



- **Array representation of Queues**
  - Every queue has **front** and **rear** variables that point to the position from where additions/deletions can be done
  - Operations on Queues:

Queue (front = 0, rear = 5):

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Queue after addition of a new element with value 45 (front = 0, rear = 6)

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Queue after deletion of an element with value 12 (front = 1, rear = 6):

	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

- **Array representation of Queues**

```
# define MAX  100
typedef struct {
            int key;
            /* other fields */
 } element;
 element  queue[MAX];
```

- Initially, **front = rear = -1**
- Queues is empty, **front = = rear**
- Queues is full, **rear == MAX-1**

- void **add**(int rear, element item) {

```
 if (rear == MAX - 1) return queue_full( );
 queue [++rear] = item;
}
```

- element **delete**(int front, int rear) {

```
  if ( front == rear) return queue_empty( );
  return queue [front++];
}
```

- Array representation of Queues - **Circular array**

  Problem:

  - The two pointers only increments, never decrements.
  - We eventually fall off the right end of the array.

  ⇨This problem can be solved by periodically moving the elements to the left, to make room on the right end.

- **Types of Queues - Circular Queues**
  - In linear queues, insertions can be done at the REAR
    deletions are done from the FRONT

FRONT = 0 and REAR = 9

54	9	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

FRONT = 2 and REAR = 9

		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Queue after two successive deletions

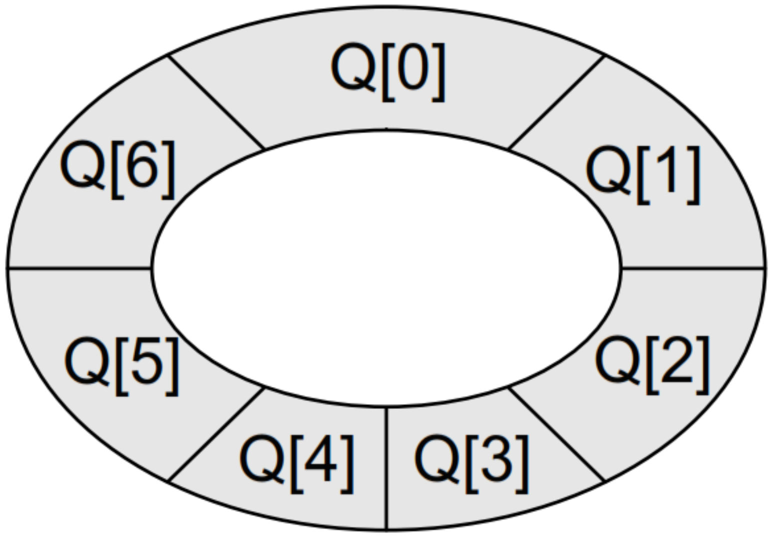
Suppose we want to insert a new element, even though there is space available, the overflow condition still exists because the condition REAR = MAX – 1 still holds true (⇦ a major drawback of a linear queue).

⇨ use a circular queue

- **Types of Queues - Circular Queues**
  - In the circular queue, the first index comes right after the last index
  - The circular queue will be full only when FRONT= 0 and REAR = MAX–1.

⇨ A circular queue is implemented in the same manner as a linear queue is implemented.

⇨ The only difference will be in the code that performs **insertion** and **deletion**



- **Types of Queues - Circular Queues**
  - **Insertion:** check for three conditions
    - (FRONT = 0 and REAR = MAX – 1) ⇨ the circular queue is full (a)
    - (REAR != MAX – 1) ⇨ REAR will be incremented, the value will be inserted (b)
    - (FRONT != 0 and REAR = MAX – 1) ⇨ the queue is not full ⇨ set REAR = 0 and insert the new element there (c)

(a)

90	49	7	18	14	36	45	21	99	72
FRONT = 0	1	2	3	4	5	6	7	8	REAR = 9

(b)

90	49	7	18	14	36	45	21	99	
FRONT = 0	1	2	3	4	5	6	7	REAR = 8	9

Increment rear so that it points to location 9 and insert the value here

(c)

		7	18	14	36	45	21	80	81
0	1	FRONT = 2	3	4	5	6	7	8	REAR = 9

Set REAR = 0 and insert the value here

• Types of Queues - Circular Queues

• Insertion

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Algorithm to insert an element in a circular queue

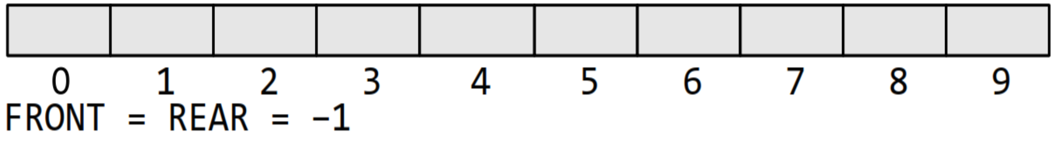
- **Types of Queues - Circular Queues**
  - **Deletion:** check for three conditions
    - (FRONT = -1) ⇨ No elements in the queue
    - (queue is not empty and FRONT=REAR)

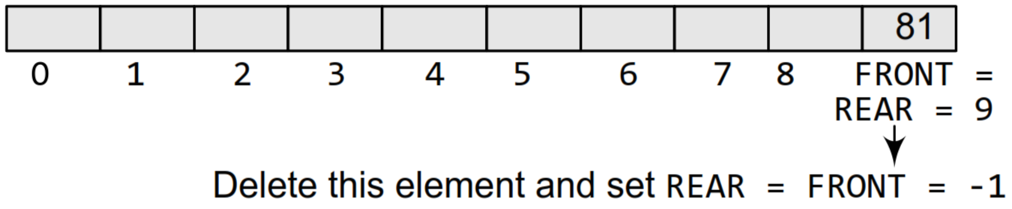
      ⇨ deleting the element at the FRONT, the queue becomes empty

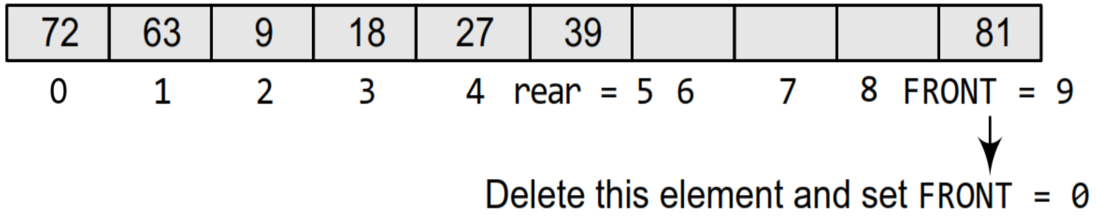
      ⇨ FRONT and REAR are set to –1
    - (queue is not empty and FRONT=MAX–1)

      ⇨ deleting the element at the FRONT

      ⇨ FRONT is set to 0





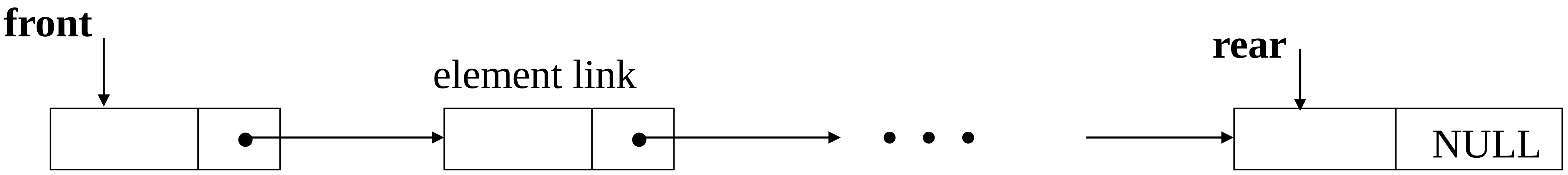


- **Types of Queues - Circular Queues**
  - **Deletion**

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

Algorithm to delete an element from a circular queue

• Linked representation of Queues

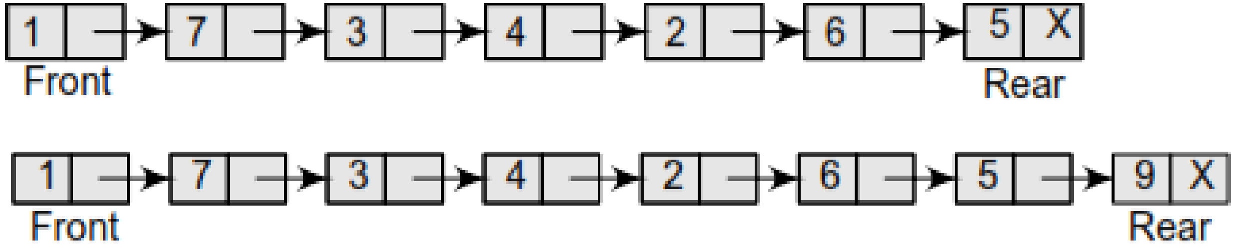


- Every element has two parts: data & the address of the next element
- The start pointer of the linked list is used as **front**. The **rear** pointer store the address of the last element in the queue.
- Additions will be done at the rear, deletions will be done at the front.
- **front** = **rear** = NULL, the queue is empty.

- **Linked representation of Queues**

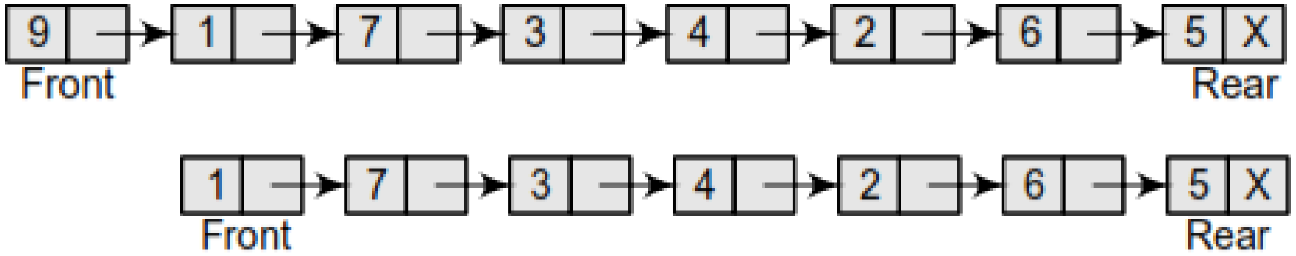
  2 basic operations:

  - **add**: inserts an element to the end of the queue
  - **delete**: removes an element from the front or the start of the queue



add

delete



- **Linked representation of Queues**
  - Declarations

```
typedef struct queue *queue_pointer;
typedef struct queue {
        element item;
        queue_pointer link;
};
```

  - Boundary conditions
    - **front** = NULL iff the queue is empty;
    - IS_EMPTY(temp) iff the queue is empty;
    - IS_FULL(temp) iff the memory is full

- **Linked representation of Queues**
  - Add to the **rear** of a linked queue

```
void addQ(queue_pointer *front, queue_pointer *rear, element item){
        /* add an element to the rear of the queue */
        queue_pointer temp =  (queue_pointer) malloc(sizeof (queue));
        if (IS_FULL(temp)) {
                fprintf(stderr, " The memory is full\n");
                exit(1);
        }
        temp->item = item;
        temp->link = NULL;
        if (*front)        (*rear) -> link = temp;
        else *front = temp;      /* the queue is empty */
        *rear = temp;
}
```

- **Linked representation of Queues**
  - Delete from the **front** of a linked queue

```
element deleteQ(queue_pointer *front) {
        /* delete an element from the queue */
         queue_pointer temp = *front;
              if (IS_EMPTY(*front))  {
             fprintf(stderr,  “The queue is empty\n”);
             exit(1);
        }

        element itemp;
        itemp = temp->item;
        *front = temp->link;
        free(temp);
        return itemp;
}
```

- **Comparing representations**
  - Array representation of Queues
    - A statically allocated array
      - Prevents the enqueue operation from adding an item to the queue if the array is full
    - A resizable array or a reference-based implementation
      - Does not impose this restriction on the enqueue operation
  - Linked representation of Queues
    - A linked list implementation
      - More efficient; no size limit

- **Applications of Queues**
  - Job scheduling
  - Waiting lists for a single shared resource like printer, disk, CPU.
  - Transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., fileIO, sockets.
  - Buffers on MP3 players and portable CD players, iPod playlist.
  - Playlist to add songs to the end, play from the front of the list.
  - Operating system for handling interrupts.
  - …

- **Introduction to Queues**
  - Example of Job scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

- Applications of Queues - **Recognizing Palindromes**
  - A palindrome
    - A string of characters that reads the same from left to right as its does from right to left
  - To recognize a palindrome, a queue can be used in conjunction with a stack
    - A stack reverses the order of occurrences
    - A queue preserves the order of occurrences
  - A nonrecursive recognition algorithm for palindromes
    - As you traverse the character string from left to right, insert each character into both a queue and a stack
    - Compare the characters at the front of the queue and the top of the stack

- Applications of Queues - **Recognizing Palindromes**

  - String: rotator

  - Queue:

r	o	t	a	t	o	r

front ... rear

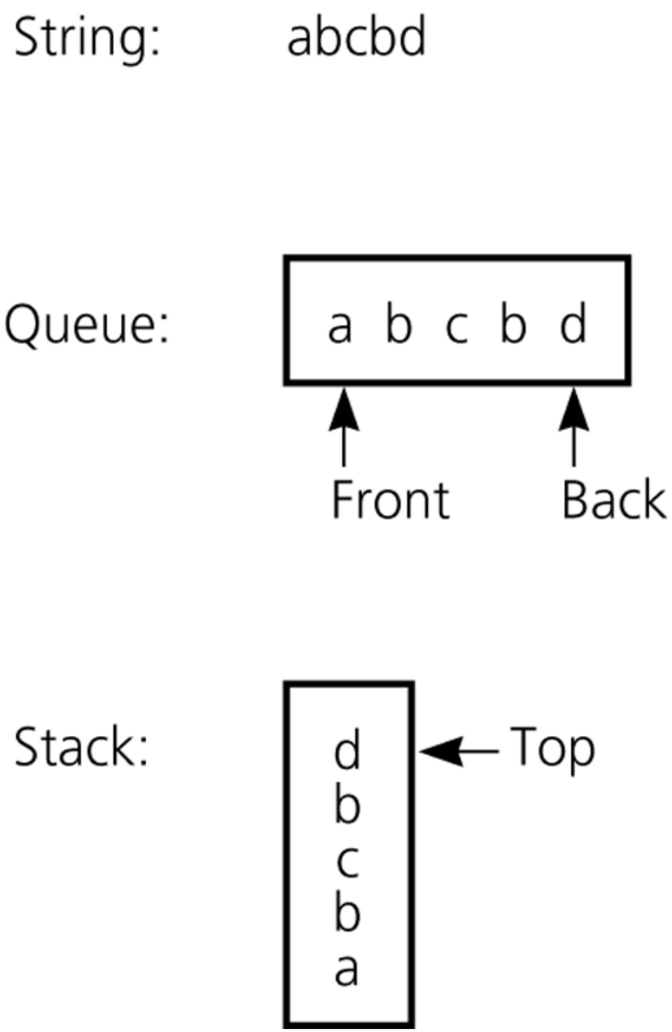
  - Stack:

top →

r
o
t
a
t
o
r

The results of inserting a string into both a queue and a stack

- Applications of Queues - **Recognizing Palindromes**



The results of inserting a string into both a queue and a stack

SUMMARY

- Stacks
- Queues

Nhân bản – Phụng sự – Khai phóng

Enjoy the Course…!