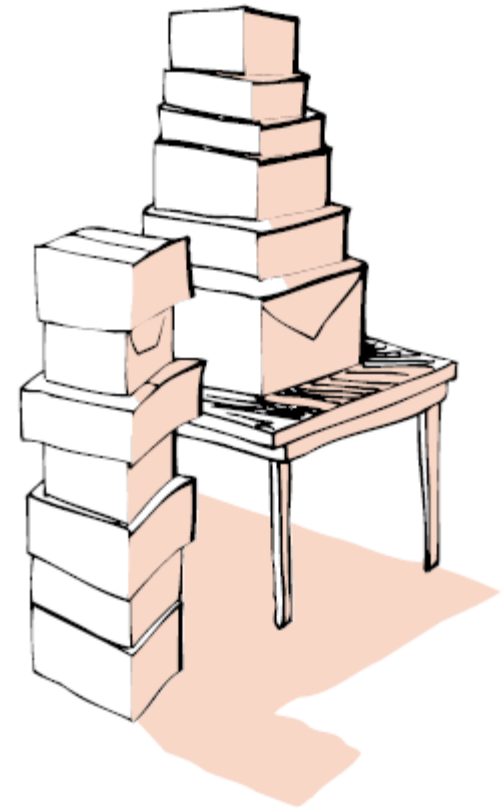


The background features several large, flowing, abstract shapes in shades of purple, green, and blue. Interspersed among these are numerous small, yellow, triangular shapes pointing in various directions, creating a dynamic and celebratory feel.

# **Data Structure & Algorithm Basic Lab – week 4+5**

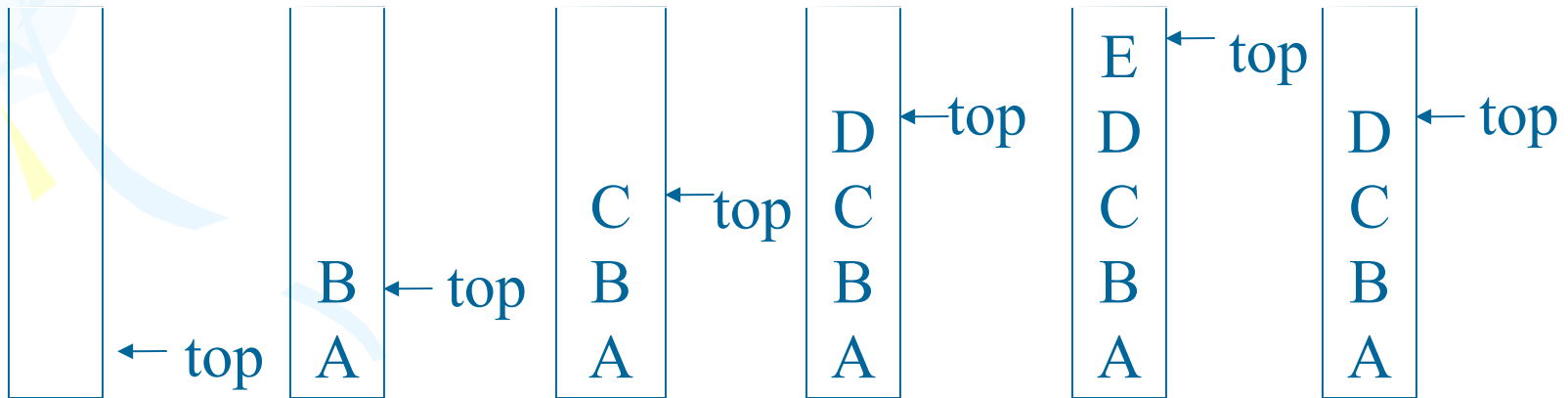
# Topics of this week

- Data structure: Stack
  - Implementation of stack using array
  - Implementation of stack using linked list
- Data structure Queue
  - Implementation of circular queue using array
  - Implementation of queue using linked list
- Exercises on Stack and Queue



# Stack

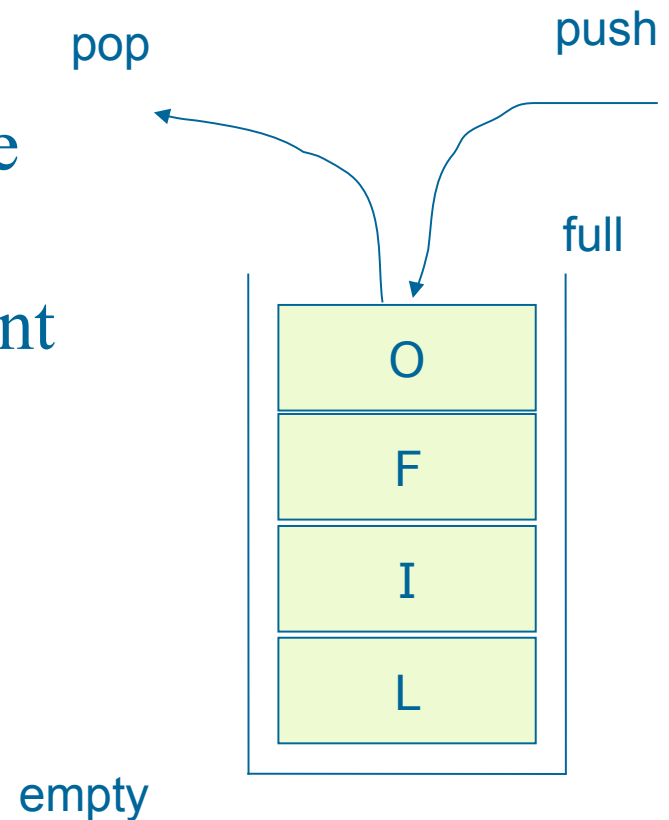
- A stack is a **linear data structure** which can be **accessed only at one of its ends** for storing and retrieving data.
- A LIFO (Last In First Out) structure



Inserting and deleting elements in a stack

# Operations on a stack

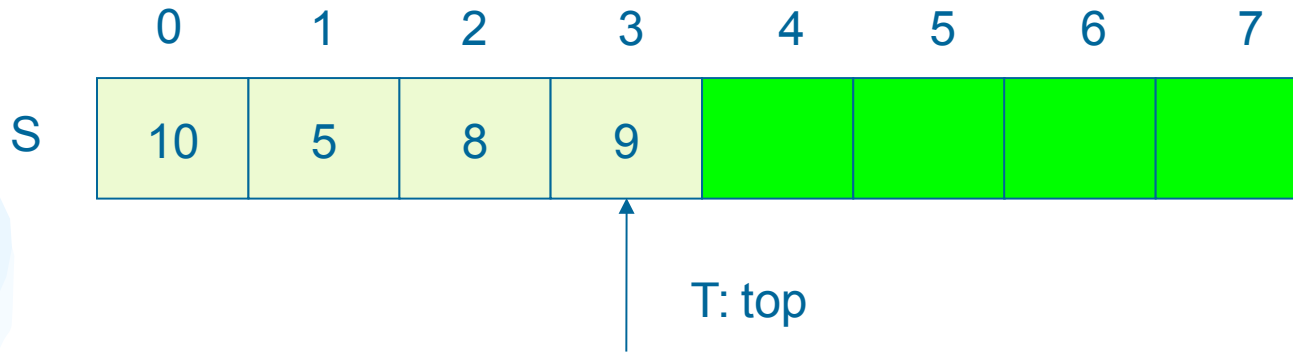
- *initialize(stack)* --- clear the stack
- *empty(stack)* --- check to see if the stack is empty
- *full(stack)* --- check to see if the stack is full
- *push(el,stack)* --- put the element *el* on the top of the stack
- *pop(stack)* --- take the topmost element from the stack
- How to implement a stack?



# Separate implementation from specification

- **INTERFACE:** specify the allowed operations
- **IMPLEMENTATION:** provide code for operations
- **CLIENT:** code that uses them.
- Could use either array or linked list to implement stack
- Client can work at higher level of abstraction

# Implementation using array



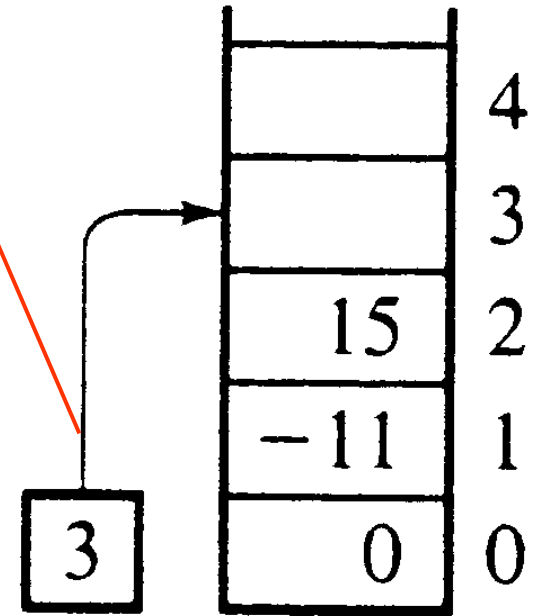
- Each element is stored as an array's element.
- stack is empty:  $top = 0$
- stack is full:  $top = Max\_Element$

# Stack specification (stack.h)

```
#define Max 50
typedef int Etype;
typedef Etype StackType[Max];
int top;

void Initialize(StackType stack);
int empty(StackType stack);
int full(StackType stack);
void push(Etype el, StackType stack);
Etype pop(StackType stack);
```

Top of stack



(a)

# array implementation of stack (stack.c)

```
Initialize(StackType stack)
```

```
{  
    top = 0;  
}
```

```
empty(StackType stack)
```

```
{  
    return top == 0;  
}
```

```
full(StackType stack)
```

```
{  
    return top == Max;  
}
```

```
push(Etype el, StackType stack)
```

```
{  
    if (full(*stack))  
        printf("stack overflow");  
    else stack[top++] = el;  
}
```

```
Etype pop(StackType stack)
```

```
{  
    if (empty(stack))  
        printf("stack underflow");  
    else return stack[--top];  
}
```



# stack implementation using structure

- Implementation (c): stack is declared as a *structure* with two fields: one for storage, one for keeping track of the topmost position

```
#define Max 50
typedef int Eltype;
typedef struct StackRec {
    Eltype storage[Max];
    int top;
};
typedef struct StackRec StackType;
```

# stack implementation using structure

```
Initialize(StackType *stack)
{
    (*stack).top=0;
}
empty(StackType stack)
{
    return stack.top ==0;
}
full(StackType stack)
{
    return stack.top == Max;
    (*stack).top];;
}
```

```
push(Etype el, StackType *stack)
{
    if (full(*stack))
        printf("stack overflow");
    else (*stack).storage[
        (*stack).top++]=el;
}
Etype pop(StackType *stack)
{
    if (empty(*stack))
        printf("stack underflow");
    else return
        (*stack).storage[--
}
}
```



# Compile file with library

You've got `stack.h`, `stack.c` and `test.c`

You need to insert this line:

```
#include "stack.h"
```

into `stack.c` and `test.c`

```
gcc -c stack.c
```

```
gcc -c test.c
```

```
gcc -o test.out test.o stack.o
```

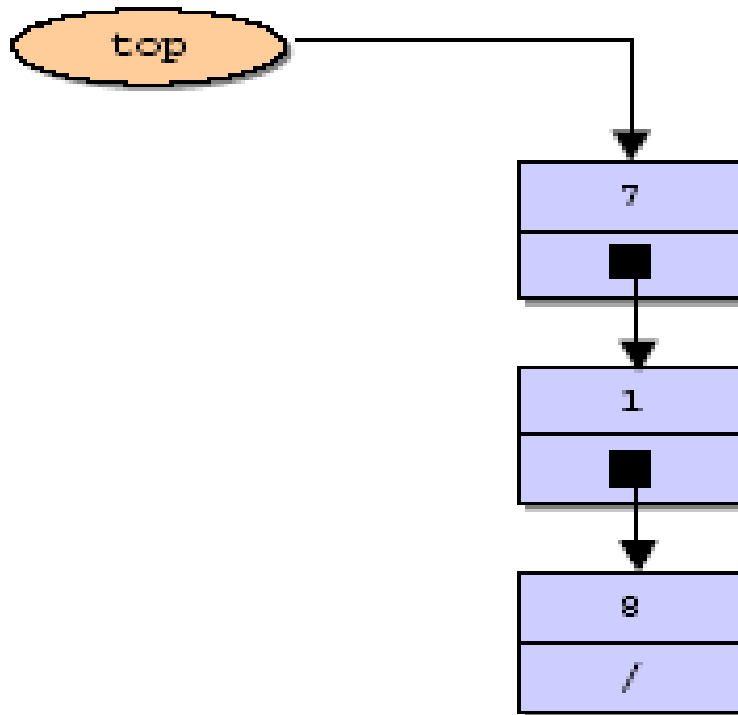
# Simple program using stack

- Write a program that convert a positive integer in decimal form to binary form using library stack you have developed.
- Extend this program to transform from decimal to hexadecimal base.

# Implementation using linked list

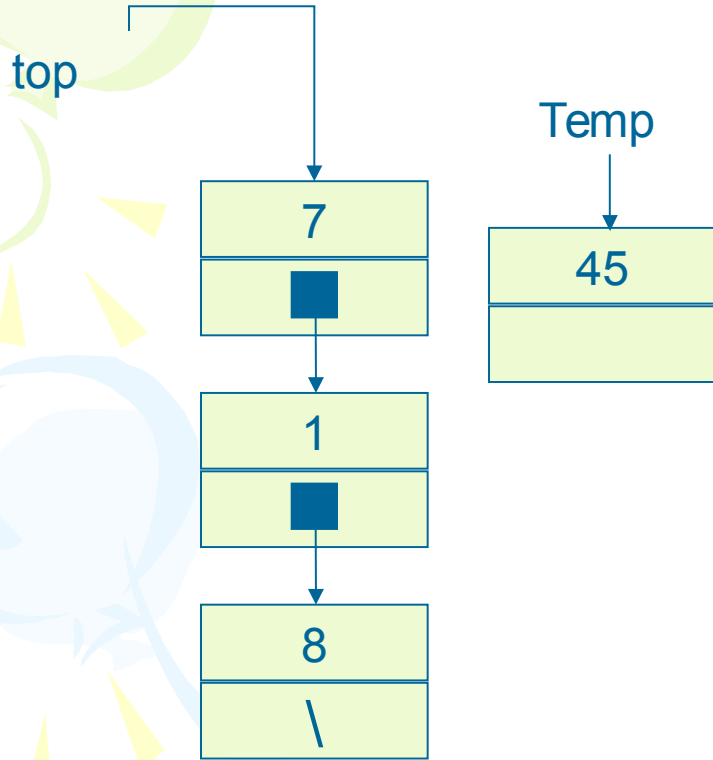
- Implementation of stacks using linked lists are very simple
- The difference between a normal linked list and a stack using a linked list is that some of the linked list operations are not available for stacks
- Being a stack we have only one insert operation called `push()`.
  - In many ways push is the same as insert in the front
- We have also one delete operation called `pop()`
  - This operation is the same as the operation delete from the front

# Pictorial view of stack



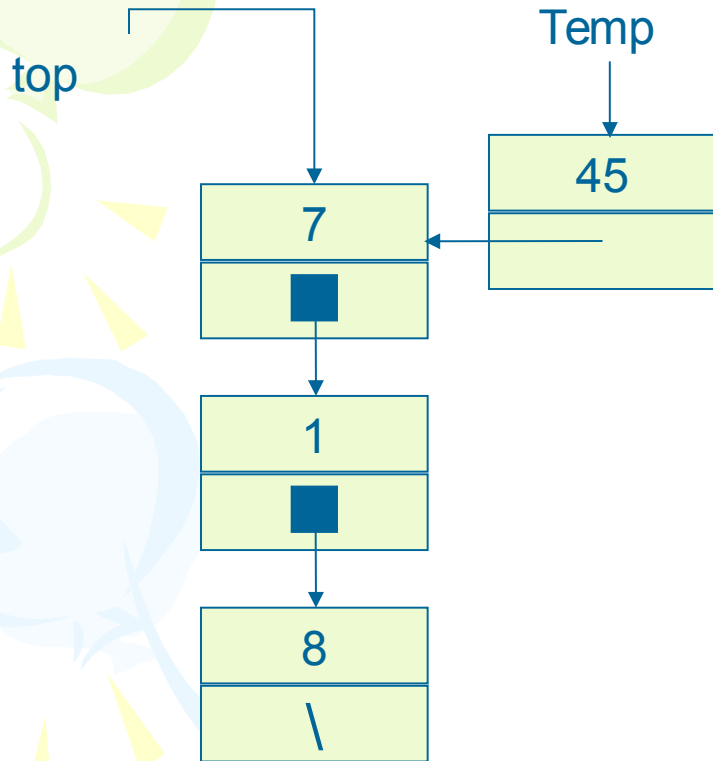
```
struct node {  
    int data;  
    struct node *link;  
};
```

# Push



```
struct node *push(struct node *p, int value)
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct
    node));
    if(temp==NULL) {
        printf("No Memory available Error\n");
        exit(0);
    }
    temp->data = value;
    temp->link = p;
    p = temp;
    return(p);
}
```

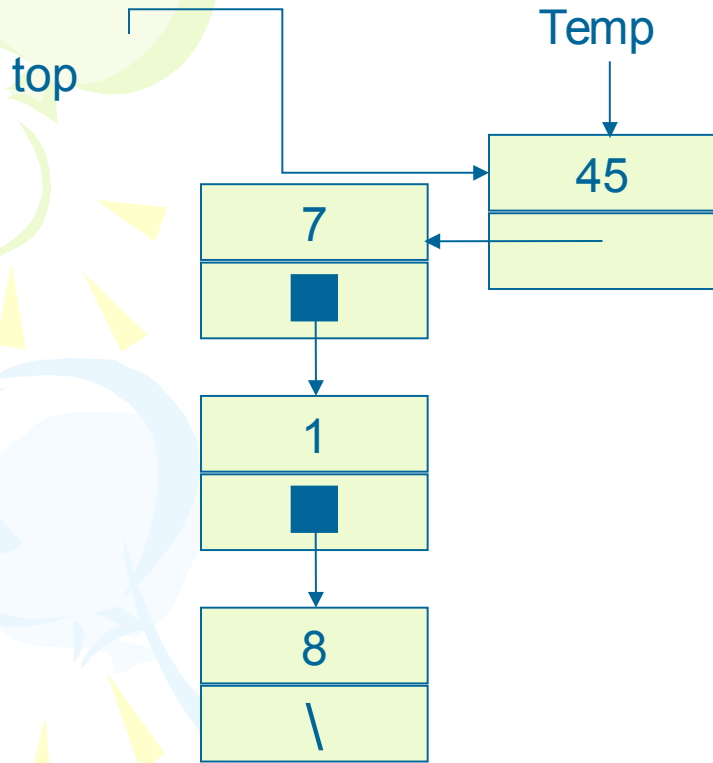
# Push



```
struct node *push(struct node *p, int value)
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct
    node));
    if(temp==NULL) {
        printf("No Memory available Error\n");
        exit(0);
    }
    temp->data = value;
    temp->link = p;
    p = temp;
    return(p);
}
```

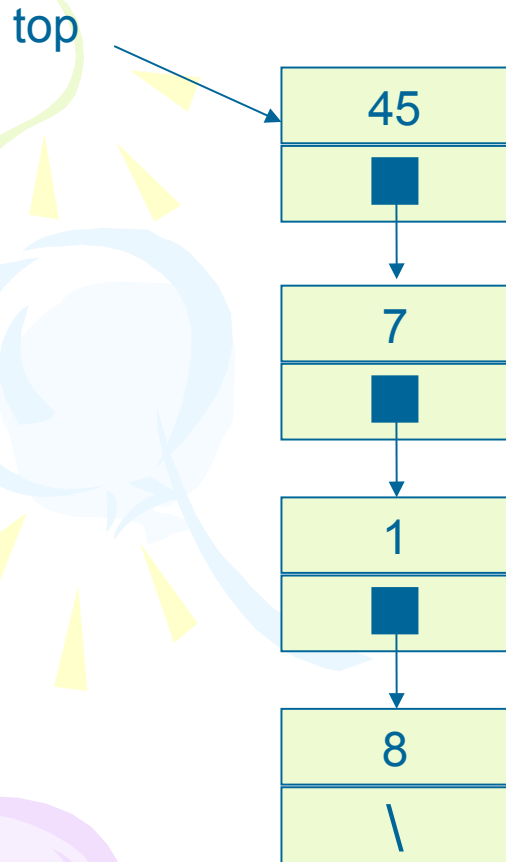


# Push



```
struct node *push(struct node *p, int value)
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct
    node));
    if(temp==NULL) {
        printf("No Memory available Error\n");
        exit(0);
    }
    temp->data = value;
    temp->link = p;
    p = temp;
    return(p);
}
```

# Pop (linked list)



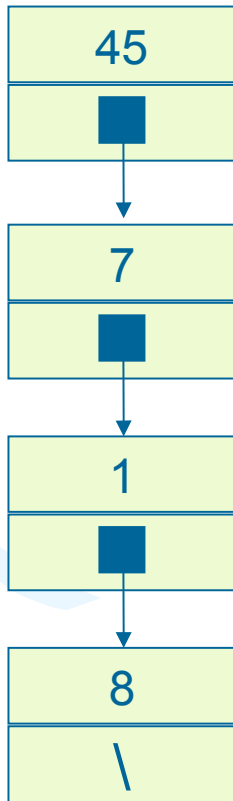
```
struct node *pop(struct node *p, int
                 *value)
{
    struct node *temp;
    if(p==NULL)
    {
        printf(" The stack is empty can
        not pop Error\n");
        exit(0);
    }
    *value = p->data;
    temp = p;
    p = p->link;
    free(temp);
    return(p);
}
```

Value at top element need to be save before pop operation

# Pop (linked list)

top

Temp

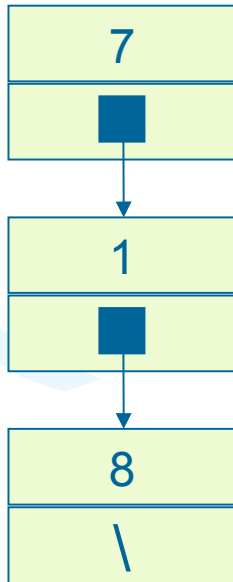


```
struct node *pop(struct node *p, int
                 *value)
{
    struct node *temp;
    if (p==NULL)
    {
        printf(" The stack is empty can
        not pop Error\n");
        exit(0);
    }
    *value = p->data;
    temp = p;
    p = p->link;
    free (temp);
    return (p);
}
```

# Pop (linked list)

top

Temp



```
struct node *pop(struct node *p, int
                 *value)
{
    struct node *temp;
    if(p==NULL)
    {
        printf(" The stack is empty can
        not pop Error\n");
        exit(0);
    }
    *value = p->data;
    temp = p;
    p = p->link;
    free(temp);
    return(p);
}
```

A decorative graphic on the left side of the slide features three balloons: a light green one at the top, a light blue one in the middle, and a light purple one at the bottom. Each balloon is attached to a streamer that trails off to the right. Small yellow triangular shapes are scattered around the balloons, resembling confetti or streamer tassels.

# Attention

- Remember to implement FreeStack Function in your lib.

# Using stack in program

```
# include <stdio.h>
# include <stdlib.h>
void main()
{
    struct node *top = NULL;
    int n,value;
    do
    {
        do
        {
            printf("Enter the element
to be pushed\n");
            scanf("%d",&value);
            top = push(top,value);
            printf("Enter 1 to
continue\n");
            scanf("%d",&n);
        } while(n == 1);
```

```
printf("Enter 1 to pop an element\n");
scanf("%d",&n);
while( n == 1)
{
    top = pop(top,&value);
    printf("The value popped is
%d\n",value);
    printf("Enter 1 to pop an element\n");
    scanf("%d",&n);
}
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
}
```

# Using stack in program

```
printf("Enter 1 to pop an element\n");
scanf("%d",&n);
while( n == 1)
{
    top = pop(top,&value);
    printf("The value popped is %d\n",value);
    printf("Enter 1 to pop an element\n");
    scanf("%d",&n);
}
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
}
```



# Exercises

- Test the "stack" type that you've defined in a program that read from user a string, then reverse it.



# BTVN1: Adding very large numbers

- Treat these numbers as strings of numerals, store the numbers corresponding to these numerals on two stacks, and then perform addition by popping numbers from the stacks

2		9		1
3		2		4
7	+	6	=	3
8		5		6
				1

8732 + 5629 = 14361

# Adding very large numbers: detail algorithm

*Read the numerals of the first number and store the numbers corresponding to them on one stack;*

*Read the numerals of the second number and store the numbers corresponding to them on another stack;*

**result**=0;

*while at least one stack is not empty*

*pop a number from each non-empty stack and add them;*

*push the sum (minus 10 if necessary) on the result stack;*

*store carry in **result**;*

*push carry on the result stack if it is not zero;*

*pop numbers from the result stack and display them;*

# Exercise 4.1 Stack using array

- We assume that you make a mobile phone's address book.
- Declare a structure "Address" that can hold at least "name", "telephone number" and "e-mail address".
- Write a program that copies data of an address book from a file to another file using a stack. First, read data of the address book from the file and push them on a stack. Then pop data from the stack and write them to the file in the order of popped. In other words, data read first should be read out last and data read last should be read out first.

# Exercise 4-2: Conversion to Reverse Polish Notation Using Stacks

- Write a program that converts an expression in the infix notation to an expression in the reverse polish notation. An expression consists of single-digit positive numbers (from 1 to 9) and four operators (+, -, \*, /). Read an expression in the infix notation from the standard input, convert it to the reverse polish notation, and output an expression to the standard output. Refer to the textbook for more details about the Reverse Polish Notation.
- For example,

$3+5*4$

is input, the following will be output.

$3\ 5\ 4\ *\ +$

A decorative background on the left side of the slide features a light green balloon at the top, a light blue balloon in the middle, and a light purple balloon at the bottom. Yellow streamers and triangular flags are scattered around the balloons.

# STACK.h

```
void STACKinit(int);  
int STACKempty();  
void STACKpush(Item);  
Item STACKpop();
```

# STACK.c

```
#include <stdlib.h>
#include "Item.h"
#include "STACK.h"
static Item *s;
static int N;
void STACKinit(int maxN)
    { s = malloc(maxN*sizeof(Item)); N = 0; }
int STACKempty()
    { return N == 0; }
void STACKpush(Item item)
    { s[N++] = item; }
Item STACKpop()
    { return s[--N]; }
```

# Solution

```
#include <stdio.h>
#include <string.h>
#include "Item.h"
#include "STACK.h"
main(int argc, char *argv[])
{ char *a = argv[1]; int i, N = strlen(a);
  STACKinit(N);
  for (i = 0; i < N; i++)
  {
    if (a[i] == ')')
      printf("%c ", STACKpop());
    if ((a[i] == '+') || (a[i] == '*'))
      STACKpush(a[i]);
    if ((a[i] >= '0') && (a[i] <= '9'))
      printf("%c ", a[i]);
  }
  printf("\n");
}
```

# Postfix expression evaluation

- Write a program that reads any postfix expression involving multiplication and addition of interger.
- For example
- `./posteval 5 4 + 6 * => 54`



A decorative background on the left side of the slide featuring a green balloon at the top, a blue balloon in the middle, and a purple balloon at the bottom. Yellow streamers and triangular flags are scattered around the balloons.

# STACK.H

```
void STACKinit(int);  
int STACKempty();  
void STACKpush(Item);  
Item STACKpop();
```

# Solution

```
#include <stdio.h>
#include <string.h>
#include "Item.h"
#include "STACK.h"
main(int argc, char *argv[])
{ char *a = argv[1]; int i, N = strlen(a);
  STACKinit(N);
  for (i = 0; i < N; i++)
  {
    if (a[i] == '+')
      STACKpush(STACKpop()+STACKpop());
    if (a[i] == '*')
      STACKpush(STACKpop()*STACKpop());
    if ((a[i] >= '0') && (a[i] <= '9'))
      STACKpush(0);
    while ((a[i] >= '0') && (a[i] <= '9'))
      STACKpush(10*STACKpop() + (a[i++]-'0'));
  }
  printf("%d \n", STACKpop());
}
```

# Solution: polish.h

```
#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#define EMPTY 0
#define FULL 10000
struct data {
    enum {operator, value} kind;
    union {
        char op;
        int val;
    } u;
};
typedef struct data data;
typedef enum {false, true} boolean;
struct elem {
    data d;
    struct elem *next;
};
/* an element on the stack */
```

# Solution: polish.h

```
typedef struct elem elem;
struct stack {
    int cnt; /* a count of the elements */
    elem *top; /* ptr to the top element */
};
typedef struct stack stack;
boolean empty(const stack *stk);
int evaluate(stack *polish);
void fill(stack *stk, const char *str);
boolean full(const stack *stk);
void initialize(stack *stk);
data pop(stack *stk);
void prn_data(data *dp);
void prn_stack(stack *stk);
void push(data d, stack *stk);
data top(stack *stk);
```

# Solution: eval.c

```
#include "polish.h"
int evaluate(stack *polish)
{
    data    d, d1, d2;
    stack  eval;
    initialize(&eval);
    while (!empty(polish)) {
        d = pop(polish);
        switch (d.kind) {
            case value:
                push(d, &eval);
                break;
            case operator:
                d2 = pop(&eval);
                d1 = pop(&eval);
                d.kind = value;
                /* begin overwriting d */
```

```
                switch (d.u.op) {
                    case '+':
                        d.u.val = d1.u.val + d2.u.val;
                        break;
                    case '-':
                        d.u.val = d1.u.val - d2.u.val;
                        break;
                    case '*':
                        d.u.val = d1.u.val * d2.u.val;
                }
                push(d, &eval);
            }
        }
        d = pop(&eval);
        return d.u.val;
    }
}
```

# Bài tập về nhà (2 tuần)

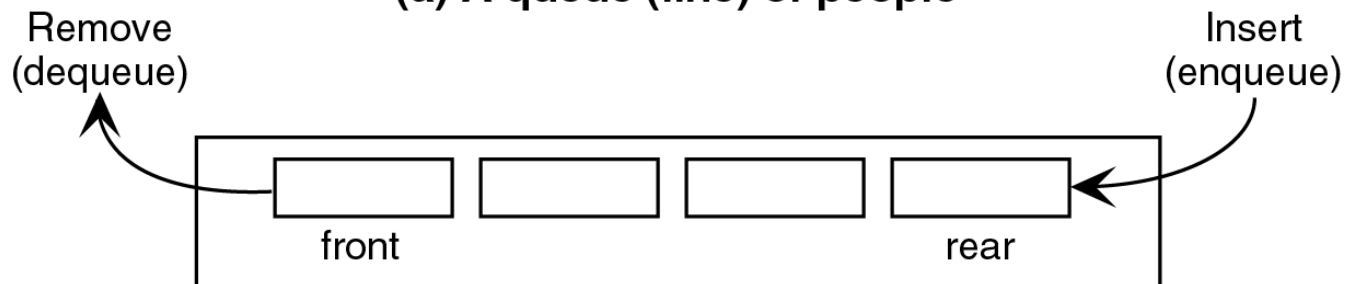
- Nghiên cứu các thuật toán nhân số lớn.
- Định hướng: Phân đoạn số lớn thành các cụm số có 3 chữ số (hoặc  $k$  chữ số tùy ý). Sau đó áp dụng phép nhân thông thường và cộng.

# Queue

- A queue is a waiting line
- Both ends are used: one for adding elements and one for removing them.
- Data is inserted (enqueued) at the rear, and removed (dequeued) at the front



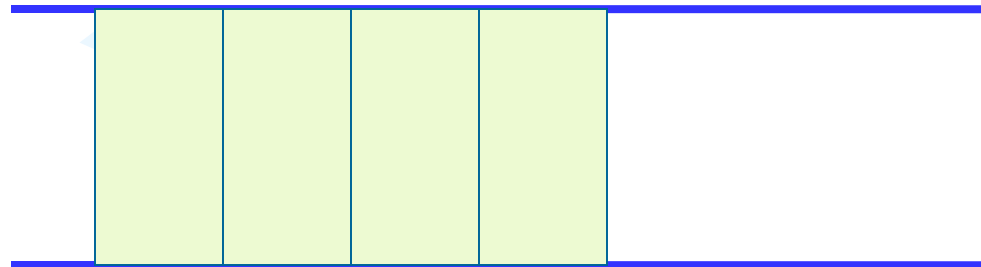
(a) A queue (line) of people



(b) A computer queue

# Data structure FIFO

- Queue items are removed in exactly the same order as they were added to the queue
  - FIFO structure: First in, First out



front rear



# Operations on queue

- *Queue*  $\text{CreateQ}(\text{max\_queue\_size}) ::=$   
create an empty queue whose  
maximum size is  
*max\_queue\_size*
- *Boolean*  $\text{IsFullQ}(\text{queue}, \text{max\_queue\_size}) ::=$   
**if**(number of elements in *queue* ==  
*max\_queue\_size*)  
**return** *TRUE*  
**else return** *FALSE*

# Operations on queue

- *Queue* EnQ(*queue*, *item*) ::=  
    **if** (IsFullQ(*queue*)) *queue\_full*  
    **else** insert *item* at rear of *queue* and  
return *queue*
- *Boolean* IsEmptyQ(*queue*) ::=  
    **if** (*queue*  
==CreateQ(*max\_queue\_size*))  
    **return TRUE**  
    **else return FALSE**
- *Element* DeQ(*queue*) ::=  
    **if** (IsEmptyQ(*queue*)) **return**  
    **else** remove and return the *item* at  
front of *queue*.

# Implementation using array and structure

```
#define MaxLength 100
typedef ... ElementType;
typedef struct {
    ElementType Elements[MaxLength];
    //Store the elements
    int Front, Rear;
} Queue;
```

# Initialize and check the status

```
void MakeNull_Queue (Queue *Q) {  
    Q->Front=-1;  
    Q->Rear=-1;  
}
```

```
int Empty_Queue (Queue Q) {  
    return Q.Front==-1;  
}
```

```
int Full_Queue (Queue Q) {  
    return (Q.Rear-Q.Front+1)==MaxLength;  
}
```

# Enqueue

```
void EnQueue (ElementType X, Queue *Q) {  
    if (!Full_Queue (*Q)) {  
        if (Empty_Queue (*Q)) Q->Front=0;  
        Q->Rear=Q->Rear+1;  
        Q->Element [Q->Rear]=X;  
    }  
    else printf ("Queue is full!");  
}
```

# Dequeue

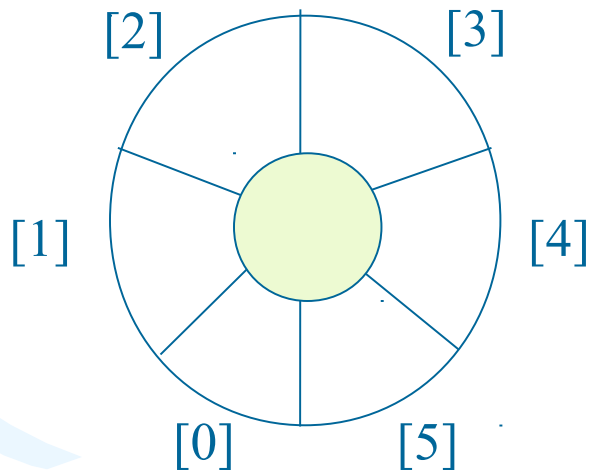
```
void DeQueue (Queue *Q) {  
    if (!Empty_Queue (*Q)) {  
        Q->Front=Q->Front+1;  
        if (Q->Front > Q->Rear)  
            MakeNull_Queue (Q);  
        // Queue become empty  
    }  
    else printf ("Queue is empty!");  
}
```

## Implementation 2: regard an array as a circular queue

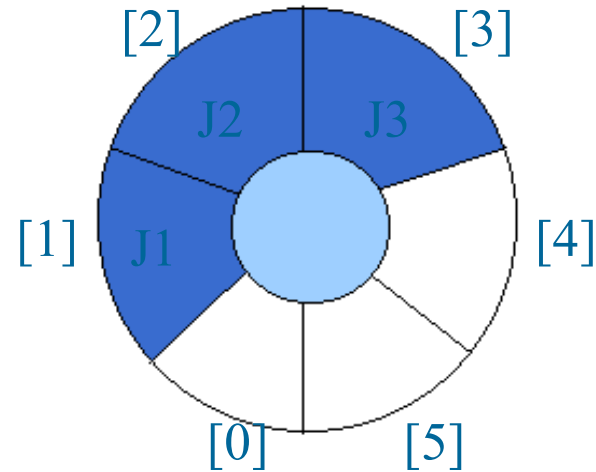
front: one position counterclockwise from the first element

rear: current end

EMPTY QUEUE



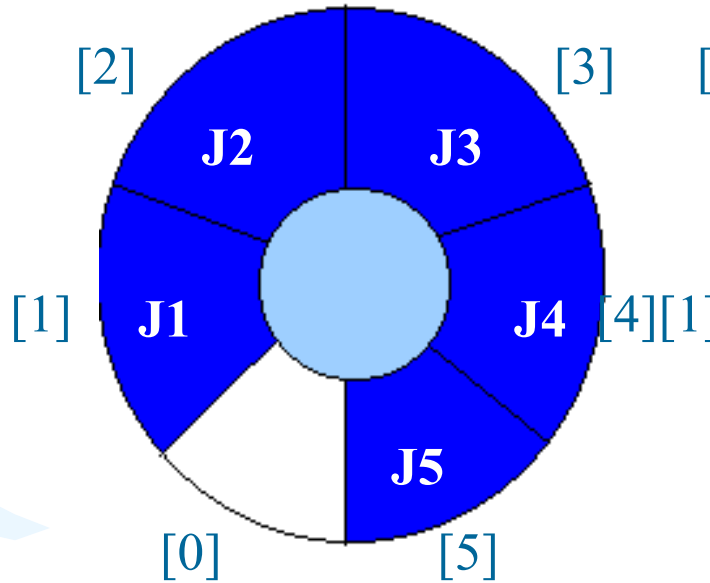
**front = 0**  
**rear = 0**



**front = 0**  
**rear = 3**

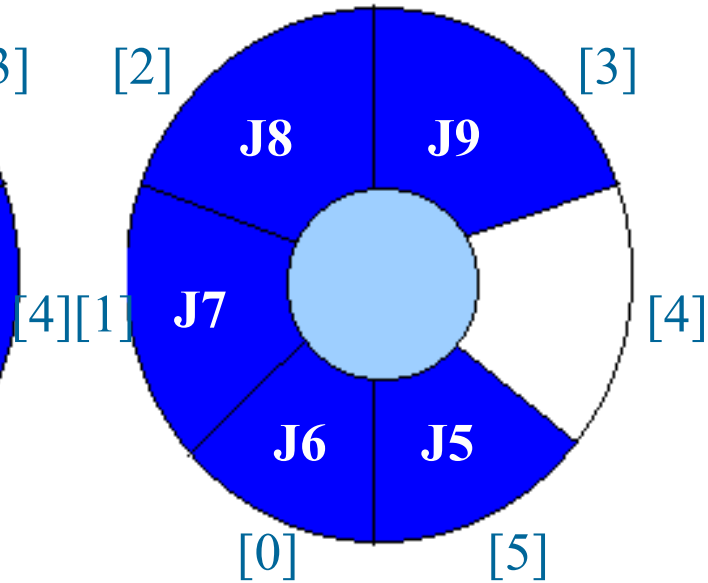
**Problem:** one space is left when queue is full

FULL QUEUE



**front = 0**  
**rear = 5**

FULL QUEUE



**front = 4**  
**rear = 3**



# Queue is full or not?

```
int Full_Queue (Queue Q) {  
    return (Q.Rear-Q.Front+1) %  
        MaxLength==0;  
}
```

# Dequeue

```
void DeQueue (Queue *Q) {  
    if (!Empty_Queue (*Q)) {  
        //if queue contain only one element  
        if (Q->Front==Q->Rear) MakeNull_Queue (Q);  
        else Q->Front=(Q->Front+1) % MaxLength;  
    }  
    else printf("Queue is empty!");  
}
```

# Enqueue

```
void EnQueue(ElementType X, Queue *Q) {  
    if (!Full_Queue(*Q)) {  
        if (Empty_Queue(*Q)) Q->Front=0;  
        Q->Rear=(Q->Rear+1) % MaxLength;  
        Q->Elements[Q->Rear]=X;  
    } else printf("Queue is full!");  
}
```

# Implementation using a List

- Exercise: A Queue, is a list specific. Implement operations on queue by reusing implemented operations of list.

# Implementation using a List

```
typedef ... ElementType;
typedef struct Node{
    ElementType Element;
    Node* Next; //pointer to next element
};
typedef Node* Position;
typedef struct{
    Position Front, Rear;
} Queue;
```

# Initialize an empty queue

```
void MakeNullQueue (Queue *Q) {  
    Position Header;  
    Header=(Node*)malloc(sizeof(Node));  
    //Allocation Header  
    Header->Next=NULL;  
    Q->Front=Header;  
    Q->Rear=Header;  
}
```

# Is-Empty

```
int EmptyQueue (Queue Q) {  
    return (Q.Front==Q.Rear) ;  
}
```

# EnQueue

```
void EnQueue (ElementType X, Queue *Q) {  
    Q->Rear->Next=  
        (Node*) malloc (sizeof (Node) );  
    Q->Rear=Q->Rear->Next;  
    Q->Rear->Element=X;  
    Q->Rear->Next=NULL;  
}
```



# DeqQueue

```
void DeQueue (Queue *Q) {  
    if (!Empty_Queue (Q) ) {  
        Position T;  
        T=Q->Front;  
        Q->Front=Q->Front->Next;  
        free (T) ;  
    }  
    else printf ("Error: Queue is empty.");  
}
```

# Exercise 4-3: Queues Using Lists

- We assume that you write a mobile phone's address book.
- Declare a structure "Address" that can hold at least "name", "telephone number" and "e-mail address".
- Write a program that copies data of an address book from the file to other file using a queue. First, read data of the address book from the file and add them to the queue. Then retrieve data from the queue and write them to the file in the order of retrieved. In other words, data read in first should be read out first and data read in last should be read out last.



# Exercises

- Make a queue that holds integers. The size of the queue is fixed to 10.
- Read integers separated by spaces from the standard input, and add them to the queue. When the program reads the 11th integer, the queue is already full. So the program removes the first integer and adds the 11th integer. Print the removed integer to the standard output.
- Process all the integers in this way.

# Exercise: Booking management

- A plane has 50 rows of seat: A B C D E F
- By using a queue, write a Air ticket Booking management program with a menu for adding, canceling, modifying requests about ticket from client.
- A ticket request has the following fields:
  - Flight Number
  - Name of client
  - Booking Time
  - Quantity.
  - Seat type: W/C/N (W: C,F C: A, D, N: B,E)
- The result is: Accept or Refuse/Wait. If accept – the system reserve a seat for each ticket end inform the users. The time field can be the system time at the moment of input.

# Queue Exercise 4EF

- Simulate a computer that process computing request from OS's programs.
- Configuration Input:
  - Number of parallel process it can run
  - Memory capacity
- Program has the menu:
  - Create new program (with a given amount of necessary memory and ID)
  - Kill a program
  - Show the status of running and waitting processes.

# BTVN: Mô phỏng quầy phục vụ tại Ngân hàng

- Thông thường tại các ngân hàng sẽ có các bàn phục vụ khách hàng – thực hiện các dịch vụ rút tiền, gửi tiền.
- Lập chương trình mô phỏng hoạt động này tại một nhà băng với tham số là số quầy phục vụ thay đổi (thực chất đây là các hàng đợi)
- Chương trình cung cấp menu để thêm khách đến nhà băng
  - nhập thời gian (9 – 9h->10h, 10 – 10-11h)
  - Thời gian phục vụ một khách là 15 phút
- Chương trình ra: Phân khách vào quầy số mấy – thời gian chờ bao lâu.
- Chương trình thống kê xem thời gian khách phải chờ là bao lâu.
  - Tổng số khách
  - Tổng thời gian chờ
  - Trung bình

# Hướng dẫn:

- Có thể đưa số liệu khách hàng đến chi nhánh ngân hàng trong file theo cấu trúc sau:

• Thời gian	Số lượng
• 9:00	2
• 9:10	1
• 9:25	3
• 9:40	2

# Giao diện

- Ngân hàng BIDV- Hà Thành 17 Tạ Quang Bửu
- Số quầy mô phỏng: 2
- 9:00 2
- Khách thứ nhất vào hàng 1 đợi : 0
- Khách thứ hai vào hàng 2 đợi : 0
- 9:10 1
- Khách thứ nhất vào hàng 1 đợi : 5 (được phục vụ tại 9:15)
- 9:25 3
- Khách thứ nhất vào hàng 2 đợi : 0
- Khách thứ hai vào hàng 1 đợi: 5
- Khách thứ ba vào hàng 2 đợi: 15
- ...
- Trung bình: XYZ khách – thời gian đợi tổng thể, trung bình



# Appendix

- 2 +1 bài (trên lớp Computer)

# Another implementation using array

- Queue CreateQ(*max\_queue\_size*) ::= 

```
# define MAX_QUEUE_SIZE 100
typedef struct {
    int key; /* other fields */
} element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
Boolean IsEmpty(queue) ::= front == rear
Boolean IsFullQ(queue) ::= rear ==
MAX_QUEUE_SIZE-1
```

# Enqueue

- ```
void enq(int *rear, element item)
{
    /* add an item to the queue */
    if (*rear == MAX_QUEUE_SIZE_1) {
        queue_full( );
        return;
    }
    queue [++*rear] = item;
}
```

# Dequeue

- `element deq(int *front, int rear)`  
{  
    if ( \*front == rear)  
        return queue\_empty( );  
    /\* return an error key \*/  
    return queue [++ \*front];  
}

# Enqueue

```
void addq(int front, int *rear, element item)
{
    *rear = (*rear + 1) % MAX_QUEUE_SIZE;
    if (front == *rear) /* reset rear and print
                        error */
        return;
}
queue[*rear] = item;
}
```

# Dequeue

```
element deleteq(int* front, int rear)
{
    element item;

    if (*front == rear)
        return queue_empty( );
    /* queue_empty returns an error key */

    *front = (*front+1) % MAX_QUEUE_SIZE;
    return queue[*front];
}
```