

PowerPoint presentation on ...

**Array**

## What is Arrays ?

- An array is a group of consecutive memory locations with same name and data type.
- Simple variable is a single memory location with unique name and a type. But an Array is collection of different adjacent memory locations. All these memory locations have one collective name and type.
- The memory locations in the array are known as **elements** of array. The total number of elements in the array is called **length**.
- The elements of array is accessed with reference to its position in array, that is call **index** or **subscript**.

## Advantages / Uses of Arrays

- Arrays can store a large number of value with single name.
- Arrays are used to process many value easily and quickly.
- The values stored in an array can be sorted easily.
- The search process can be applied on arrays easily.

## **Types of Arrays:**

- One-Dimensional Array
- Two-Dimensional Array
- Multi-Dimensional Array

# One-D Array

A type of array in which all elements are arranged in the form of a list is known as **1-D array** or **single dimensional array** or **linear list**.

## Declaring 1-D Array:

data\_type identifier[length];

e.g: int marks[5];

- **Data\_type:** Data type of values to be stored in the array.
- **Identifier:** Name of the array.
- **Length:** Number of elements.

0 1

int marks



# One-D array Initialization

The process of assigning values to array elements at the time of array declaration is called **array initialization**.

## Syntax:

data\_type identifier[length]={ List of values };      e.g. 

70	54	82	96	54	92
----	----	----	----	----	----

,96,49};

- **Data\_type:** Data type of values to be stored in the array.
- **Identifier:** Name of the array.
- **Length:** Number of elements
- **List of values:** Values to initialize the array. Initializing values must be constant

## Accessing element of array

- **Individual Element:**

```
Array_name[index];
```

- **Using Loop:**

```
int marks[5];  
for(int i=0;i<=4;i++)  
marks[i]=i;
```

## Searching In Array

Searching is a process of finding the required data in the array. Searching becomes more important when the length of the array is very large.

There are two techniques to searching elements in array as follows:

- Sequential search
- Binary search



# Sequential Search

Sequential search is also known as **linear** or **serial search**. It follows the following step to search a value in array.

- Visit the first element of array and compare its value with required value.
- If the value of array matches with the desired value, the search is complete.
- If the value of array does not match, move to next element and repeat same process.

# Binary Search

Binary search is a quicker method of searching for value in the array. Binary search is very quick but it can only search an sorted array. It cannot be applied on an unsorted array.

- It locates the middle element of array and compare with desired number.
- If they are equal, search is successful and the index of middle element is returned.
- If they are not equal, it reduces the search to half of the array.
- If the search number is less than the middle element, it searches the first half of array.

Otherwise it searches the second half of the array. The process continues until the required number is found or loop completes without successful search.

# Sorting Arrays

Sorting is a process of arranging the value of array in a particular order.  
An array can be sorted in two order.

- Ascending Order

- Descending Order

12	25	33	37	48
48	37	33	25	12

# Techniques Of Sorting Array

There are two techniques of sorting array:

- Selection Sort
- Bubble Sort

# Selection Sort

Selection sort is a technique that sort an array. It selects an element in the array and moves it to its proper position. Selection sort works as follows:

1. Find the minimum value in the list.
2. Swap it with value in the first position.
3. Sort the remainder of the list excluding the first value.

# Bubble Sort

Bubble Sort is also known as **exchange sort**. It repeatedly visits the array and compares two items at a time. It works as follows:

- Compare adjacent element. If the first is greater than the second, swap them.
- Repeat this for each pair of adjacent element, starting with the first two and ending with the last two. (at this point last element should be greatest).
- Repeat the step for all elements except the last one.
- Keep repeating for one fewer element each time until there are no pairs to compare.

# Two-D Arrays

Two-D array can be considered as table that consists of rows and columns. Each element in 2-D array is referred with the help of two indexes. One index indicates row and second indicates the column.

## Declaring 2-D Array:

Data\_type Identifier[row][column];

e.g: int arr[4][3];

- **Data\_type:** Data type of values to be stored in the array.
- **Identifier:** Name of the array.
- **Rows :** # of Rows in the table of array.
- **Column :** # of Columns in the table of array.

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2
3,0	3,1	3,2

## Two-D array Initialization

The two-D array can also be initialized at the time of declaration. Initialization is performed by assigning the initial values in braces separated by commas.

Some important points :

- The elements of each row are enclosed within braces and separated by comma.
- All rows are enclosed within braces.
- For number arrays, if all elements are not specified , the unspecified elements are initialized by zero.



# Two-D array Intialization

## Syntax:

```
int arr[4][3]={ {12,5,22},
```

Column  
indexes

```
{95,3,41},
```

```
{77,6,53},
```

```
{84,59,62} }
```

Row  
indexes

0

1

2

0

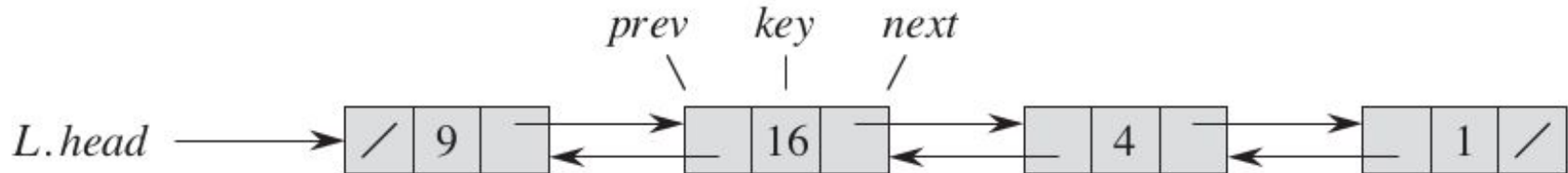
2

3

	0	1	2
0	12	5	22
1	95	3	41
2	77	6	53
3	84	59	62

# Linked List

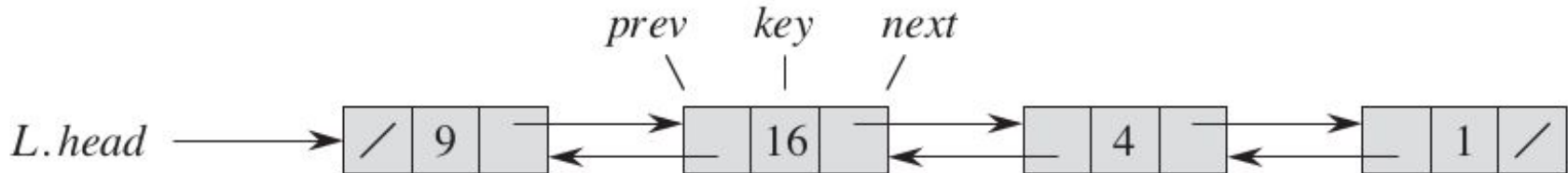
- A linked list is a data structure in which the objects are arranged in a linear order.
- Linked List provide a simple and flexible representation for dynamic sets which is not available in arrays due to fixed size.
- Linked Lists also support operations such as Search, Insertion and Deletion.



# Doubly Linked List

- Each element of a doubly linked list  $L$  is an object with an attribute *key* and two other pointer attributes: *next* and *prev*.
- If  $x.\text{prev} = \text{NIL}$ , the element  $x$  has no predecessor and is therefore the first element, or *head*, of the list.

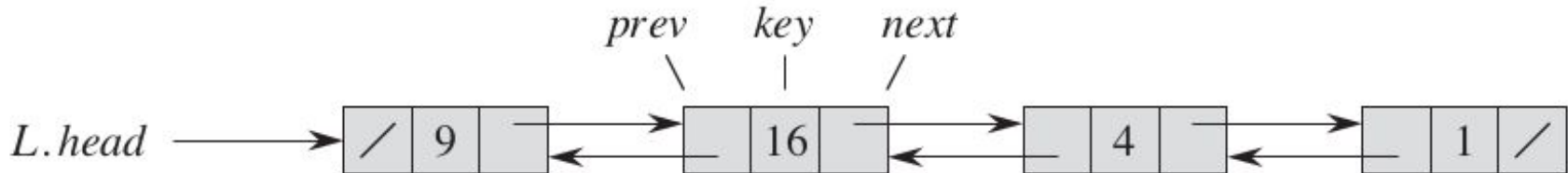
```
1 struct Node
2 {
3     int key;
4     Node *prev;
5     Node *next;
6 };
```



# Doubly Linked List

- If  $x.next = \text{NIL}$ , the element  $x$  has no successor and is therefore the last element, or *tail*, of the list.
- An attribute  $L.head$  points to the first element of the list. If  $L.head = \text{NIL}$ , the list is empty.

```
1 struct Node
2 {
3     int key;
4     Node *prev;
5     Node *next;
6 };
```



# Different Types of Linked Lists

In a **Singly Linked List**, the *prev* pointer is not present in any element.

```
1  struct Node
2  {
3      int key;
4      Node *next;
5  };
```

In a **Circular list**, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head.

```
1  tail->next = head;
2  head->prev = tail;
```

# Searching

- *List-Search*(*L*, *k*) finds the first element with key *k* in list *L* by a simple linear search, returning a pointer to this element. If no such element is present, it returns NIL.
- To search a list of *n* objects, the *List-Search* procedure takes  $\Theta(n)$  time in the worst case, since it may have to search the entire list.

LIST-SEARCH(*L*, *k*)

```
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

```
1  Node *List_Search(List L, int k)
2  {
3      Node *x = L.head;
4      while (x != nullptr && x->key != k)
5          x = x->next;
6      return x;
7  }
```

# Insertion

- *LIST-INSERT* procedure “splices”  $x$  onto the front of the linked list.
- The running time for *LIST-INSERT* on a list of  $n$  elements is  $O(1)$ .

*LIST-INSERT*( $L, x$ )

```
1   $x.next = L.head$ 
2  if  $L.head \neq \text{NIL}$ 
3       $L.head.prev = x$ 
4   $L.head = x$ 
5   $x.prev = \text{NIL}$ 
```

```
1  void List_Insert(List L, Node *x)
2  {
3      x->next = L.head;
4      if (L.head != nullptr)
5          L.head->prev = x;
6      L.head = x;
7      x->prev = nullptr;
8  }
```

# Deletion

- *LIST-DELETE* removes an element  $x$  from a linked list  $L$ . It must be given a pointer to  $x$ , and it then “splices”  $x$  out of the list by updating pointers. If we wish to delete an element with a given key, we must first call *LIST-SEARCH* to retrieve a pointer to the element.
- *LIST-DELETE* runs in  $O(1)$  time, but if we search first, then  $\Theta(n)$  time is required.

*LIST-DELETE*( $L, x$ )

```
1  if  $x.prev \neq \text{NIL}$ 
2       $x.prev.next = x.next$ 
3  else  $L.head = x.next$ 
4  if  $x.next \neq \text{NIL}$ 
5       $x.next.prev = x.prev$ 
```

```
1  void List_Delete(List L, Node *x)
2  {
3      if (x->prev != nullptr)
4          x->prev->next = x->next;
5      else
6          L.head = x->next;
7      if (x->next != nullptr)
8          x->next->prev = x->prev;
9  }
```



# STACKS & QUEUES

DATA  
STRUCTURE

```
graph TD; A[DATA STRUCTURE] --> B[LINEAR DATA STRUCTURE]; A --> C[NON LINEAR DATA STRUCTURE]; B --> D[ARRAY]; B --> E[QUEUE]; B --> F[STACK];
```

LINEAR DATA  
STRUCTURE

NON LINEAR  
DATA  
STRUCTURE

ARRAY

QUEUE

**STACK**

# What is Linear Data Structure

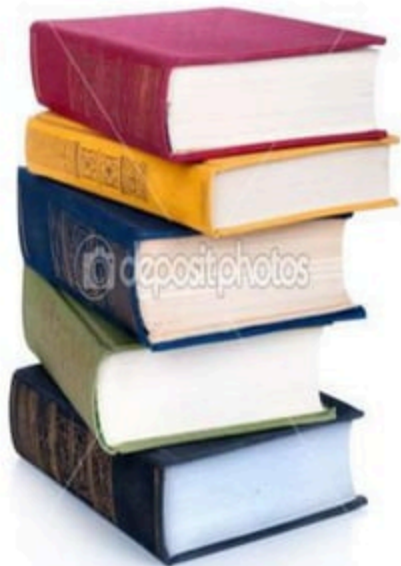


- In linear data structure, data is arranged in linear sequence.
- Data items can be traversed in a single run.
- In linear data structure elements are accessed or placed in contiguous(together in sequence) memory location.

# WHAT Is *stack*

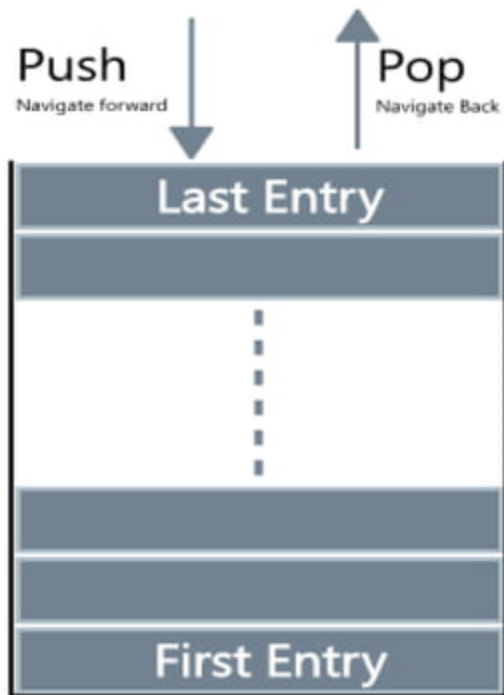
- A stack is called a last-in-first-out (LIFO) collection. This means that the last thing we added (pushed) is the first thing that gets pulled (popped) off.
- A stack is a sequence of items that are accessible at only one end of the sequence.

# EXAMPLES OF STACK:



# Operations that can be performed on STACK:

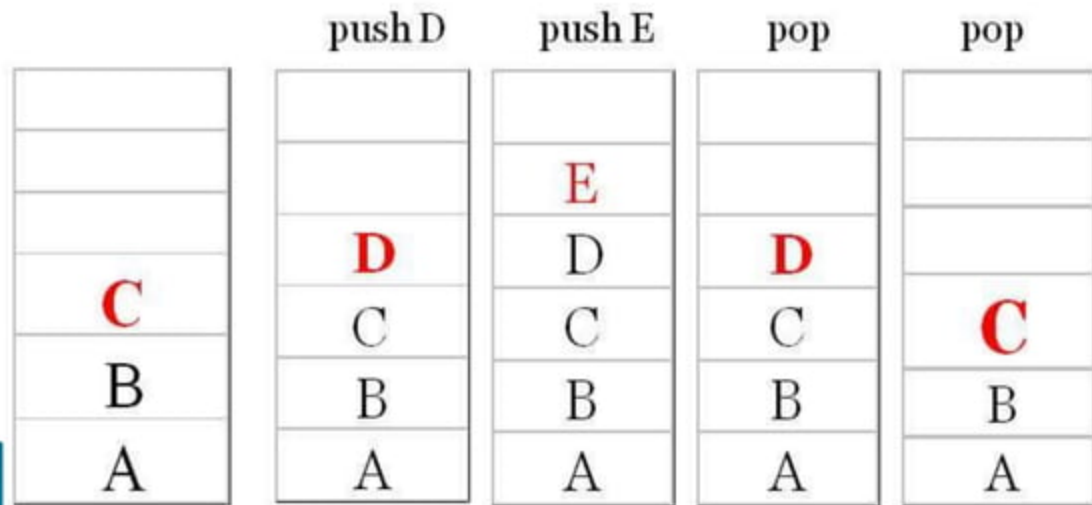
- PUSH.
- POP.



PUSH : It is used to insert items into the stack.

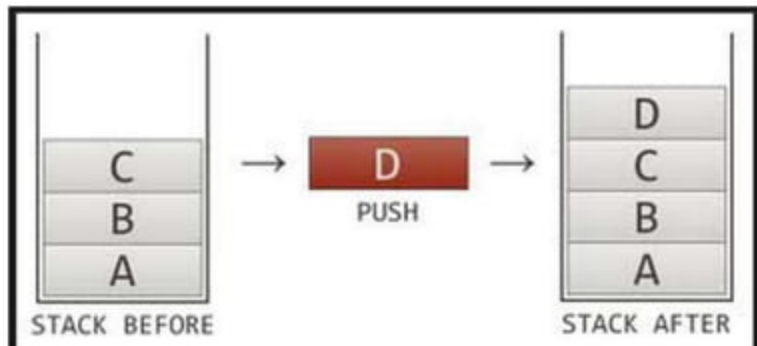
POP: It is used to delete items from stack.

TOP: It represents the current location of data in stack.



# ALGORITHM OF INSERTION IN STACK: (PUSH)

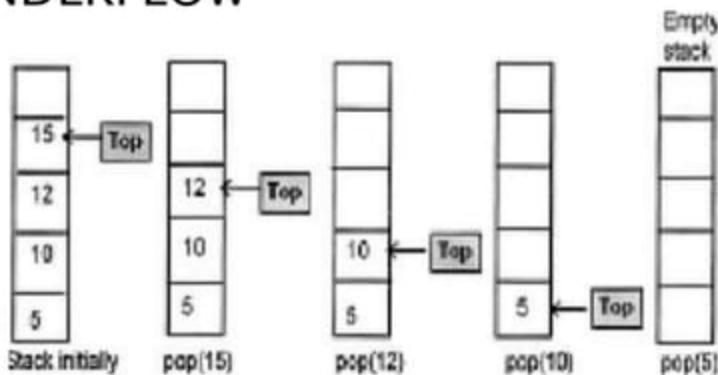
1. Insertion(a,top,item,max)
2. If top=max then  
print 'STACK OVERFLOW'  
exit  
else
3. top=top+1  
end if
4. a[top]=item
5. Exit





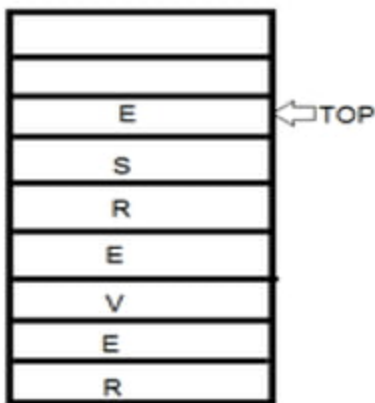
# ALGORITHM OF DELETION IN STACK: (POP)

1. Deletion(a,top,item)
2. If  $top=0$  then  
    print 'STACK UNDERFLOW'  
    exit  
    else
3. item=a[top]  
    end if
4.  $top=top-1$
5. Exit



# ALGORITHM OF DISPLAY IN STACK:

1. Display(top, i, a[i])
2. If top=0 then  
Print 'STACK EMPTY'  
Exit  
Else
3. For i=top to 0  
Print a[i]  
End for
4. exit



STACK

# APPLICATIONS OF STACKS ARE:

## I. Reversing Strings:

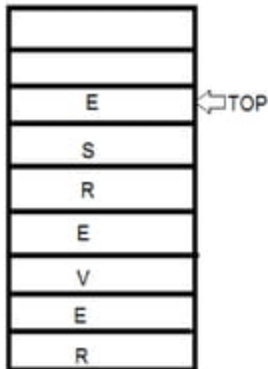
- A simple application of stack is reversing strings. To reverse a string , the characters of string are pushed onto the stack one by one as the string is read from left to right.
- Once all the characters of string are pushed onto stack, they are popped one by one. Since the character last pushed in comes out first, subsequent pop operation results in the reversal of the string.

# For example:

To reverse the string 'REVERSE' the string is read from left to right and its characters are pushed . LIKE:


STRING IS:

REVERSE



STACK

## II. Checking the validity of an expression containing nested parenthesis:

- Stacks are also used to check whether a given arithmetic expressions containing nested parenthesis is properly parenthesized.
  - The program for checking the validity of an expression verifies that for each left parenthesis braces or bracket ,there is a corresponding closing symbol and symbols are appropriately nested.
- 

For example:

VALID INPUTS	INVALID INPUTS
{ }	{ ( }
( { [ ] } )	( [ ( ( ) ] )
{ [ ] ( ) }	{ } [ ] )
[ { ( { } [ ] ( { } ) ] ]	[ { ) } ( [ ] } ]

### III. Evaluating arithmetic expressions:

#### INFIX notation:

The general way of writing arithmetic expressions is known as infix notation.

e.g,  $(a+b)$

#### PREFIX notation:

e.g,  $+AB$

#### POSTFIX notation:

e.g:  $AB+$

## Conversion of INFIX to POSTFIX conversion:

Example:  $2+(4-1)*3$

step1

$2+41-*3$

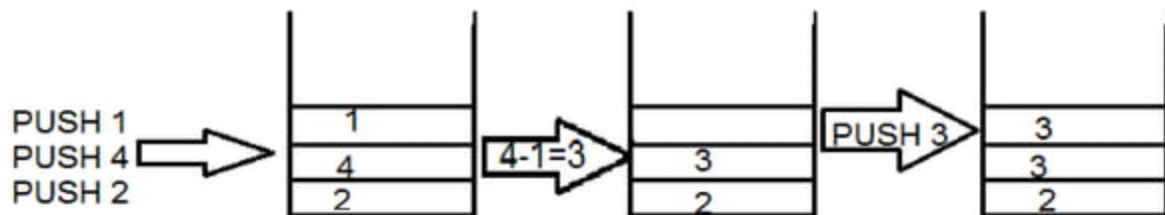
step2

$2+41-3*$

step3

$241-3*+$

step4





# CONVERSION OF INFIX INTO POSTFIX


$2+(4-1)*3$  into  $241-3*+$

CURRENT SYMBOL	ACTION PERFORMED	STACK STATUS	POSTFIX EXPRESSION
(	PUSH C	C	2
2			2
+	PUSH +	(+	2
(	PUSH (	(+(	24
4			24
-	PUSH -	(+(-	241
1	POP		241-
)		(+	241-
*	PUSH *	(+*	241-
3			241-3
	POP *		241-3*
	POP +		241-3*+
)			

# Queues

- ▶ Queue is an ADT data structure similar to stack, except that the first item to be inserted is the first one to be removed.
- ▶ This mechanism is called First-In-First-Out (FIFO).
- ▶ Placing an item in a queue is called “insertion or enqueue”, which is done at the end of the queue called “rear”.
- ▶ Removing an item from a queue is called “deletion or dequeue”, which is done at the other end of the queue called “front”.
- ▶ Some of the applications are : printer queue, keystroke queue, etc.

## Operations On A Queue

1. To insert an element in queue
  2. Delete an element from queue
- 

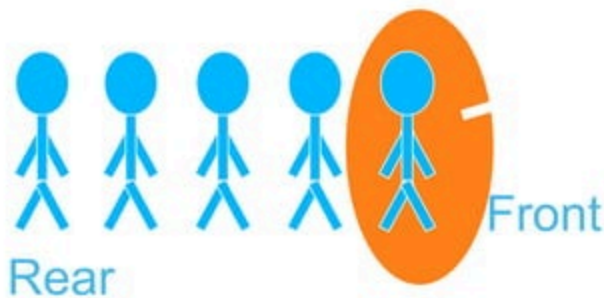
# The Queue Operation

Placing an item in a queue is called “insertion or **enqueue**”, which is done at the end of the queue called “**rear**”.



# The Queue Operation

Removing an item from a queue is called “deletion or **dequeue**”, which is done at the other end of the queue called “**front**”.



## Algorithm QINSERT (ITEM)

1.If (rear = maxsize-1 )

    print ("queue overflow") and return

2.Else

    rear = rear + 1

    Queue [rear] = item

## Algorithm QDELETE ()

1.If (front =rear)

print "queue empty" and return

2. Else

Front = front + 1

item = queue [front];

Return item

# Queue Applications

- Real life examples
  - ✓ Waiting in line
  - ✓ Waiting on hold for tech support
- Applications related to Computer Science
  - ✓ Round robin scheduling
  - ✓ Job scheduling (FIFO Scheduling)
  - ✓ Key board buffer



## 3 states of the queue

1. Queue is empty

$\text{FRONT} = \text{REAR}$

2. Queue is full

$\text{REAR} = N$

3. Queue contains element  $\geq 1$

$\text{FRONT} < \text{REAR}$


$\text{NO. OF ELEMENT} = \text{REAR} - \text{FRONT} + 1$

# Representation Of Queues

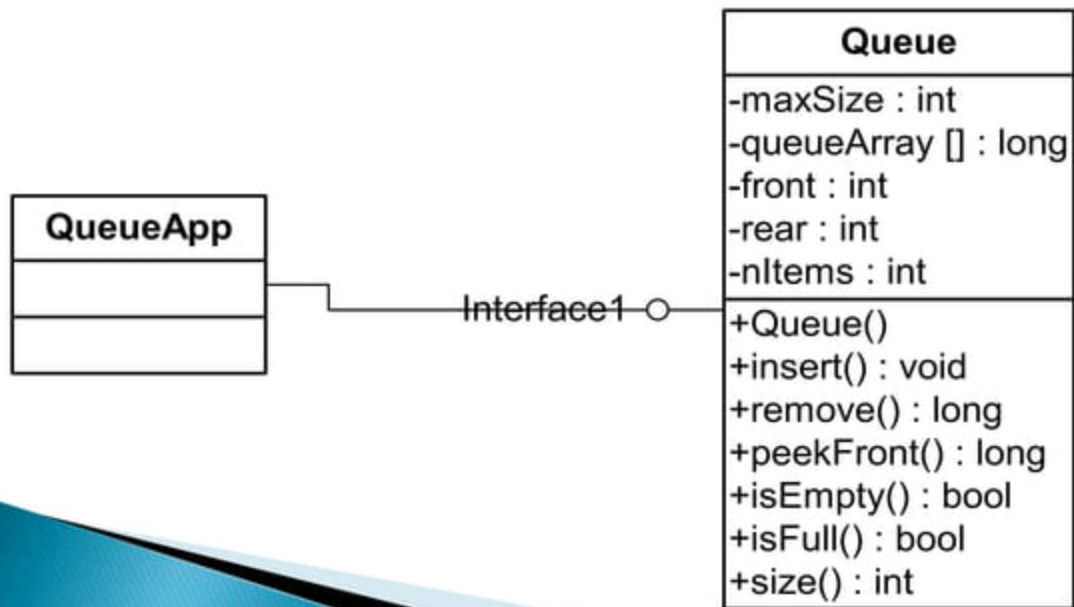
1. Using an array
2. Using linked list



# Circular Queue

- ▶ To solve this problem, queues implement wrapping around. Such queues are called Circular Queues.
  - ▶ Both the front and the rear pointers wrap around to the beginning of the array.
  - ▶ It is also called as “Ring buffer”.
  - ▶ Items can inserted and deleted from a queue in  $O(1)$  time.
- 


# Queue Example




# Queue sample code

- ▶ C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Queue\queue.java


# Various Queues

- ▶ Normal queue (FIFO)
  - ▶ Circular Queue (Normal Queue)
  - ▶ Double-ended Queue (Deque)
  - ▶ Priority Queue
- 

# Deque

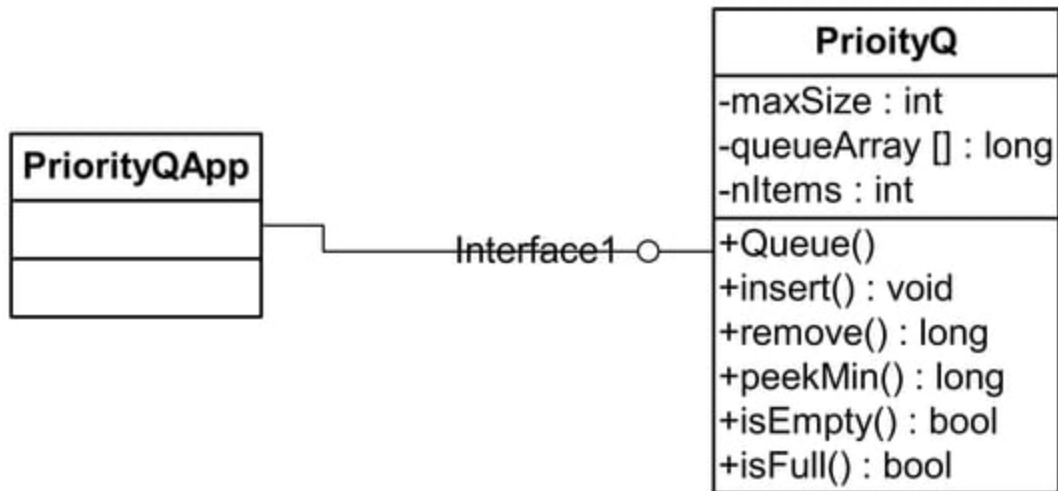
- ▶ It is a double-ended queue.
  - ▶ Items can be inserted and deleted from either ends.
  - ▶ More versatile data structure than stack or queue.
  - ▶ E.g. policy-based application (e.g. low priority go to the end, high go to the front)
  - ▶ In a case where you want to sort the queue once in a while, **What sorting algorithm will you use?**
- 

# Priority Queues

- ▶ More specialized data structure.
  - ▶ Similar to Queue, having front and rear.
  - ▶ Items are removed from the front.
  - ▶ Items are ordered by key value so that the item with the lowest key (or highest) is always at the front.
  - ▶ Items are inserted in proper position to maintain the order.
  - ▶ Let's discuss complexity
- 



# Priority Queue Example



# Priority Queues

- ▶ Used in multitasking operating system.
- ▶ They are generally represented using “heap” data structure.
- ▶ Insertion runs in  $O(n)$  time, deletion in  $O(1)$  time.
- ▶ [C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\PriorityQ\priorityQ.java](#)

# Parsing Arithmetic Expressions

- ▶  $2 + 3$  •  $2\ 3\ +$
- ▶  $2 + 4 * 5$  •  $2\ 4\ 5\ *\ +$
- ▶  $((2 + 4) * 7) + 3 * (9 - 5)$  •  $2\ 4\ +\ 7\ *\ 3\ 9\ 5\ -\ *\ +$
- ▶ Infix vs postfix
- ▶ Why do we want to do this transformation?

# Infix to postfix

- ▶ Read ch from input until empty
  - If ch is arg , output = output + arg
  - If ch is "(", push '(';
  - If ch is op and higher than top push ch
  - If ch is ")" or end of input,
    - output = output + pop() until empty or top is "("
  - Read next input
- ▶ C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Postfix\postfix.java

# Postfix eval

▶  $5 + 2 * 3 \rightarrow 5\ 2\ 3\ *\ +$

▶ Algorithm

- While input is not empty
- If ch is number , push (ch)
- Else
  - Pop (a)
  - Pop(b)
  - Eval (ch, a, b)

▶ C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Postfix\postfix.java

# Quick XML Review

- ▶ XML – Wave of the future

## Another Real world example

- ▶ `<?xml version = "1.0"?>`
- ▶ `<!-- An author -->`
- ▶ `<author>`
- ▶     `<name gender = "male">`
- ▶         `<first> Art </first>`
- ▶         `<last> Gittleman </last>`
- ▶     `</name>`
- ▶ `</author>`

THANK  
YOU





# Recursion

# Recursive Methods Must Eventually Terminate

*A recursive method must have  
at least one base, or stopping, case.*

- A base case does not execute a recursive call
  - stops the recursion
- Each successive call to itself must be a "smaller version of itself"
  - an argument that describes a smaller problem
    - a base case is eventually reached

# Factorial ( $N!$ )

- $N! = (N-1)! * N$  [for  $N > 1$ ]
- $1! = 1$
- $3!$ 
  - $= 2! * 3$
  - $= (1! * 2) * 3$
  - $= 1 * 2 * 3$
- Recursive design:
  - Decomposition:  $(N-1)!$
  - Composition:  $* N$
  - Base case:  $1!$

# factorial Method

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; // composition
    else // base case
        fact = 1;

    return fact;
}
```

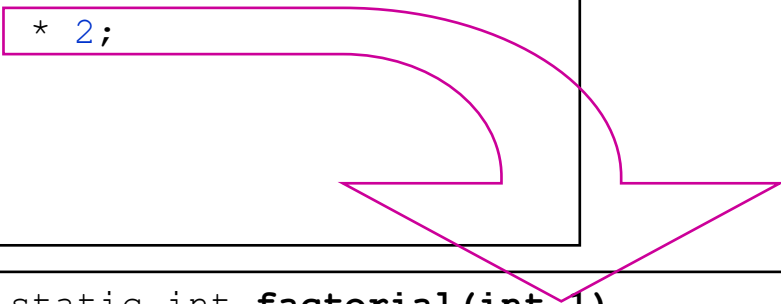
```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

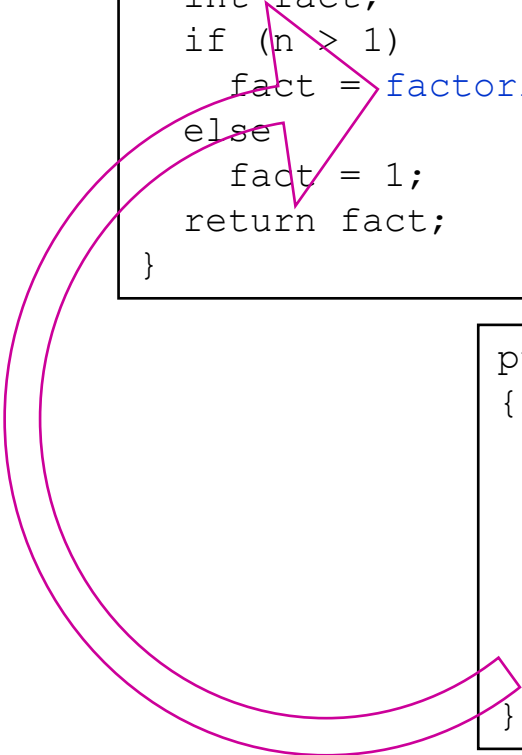


```
public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

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

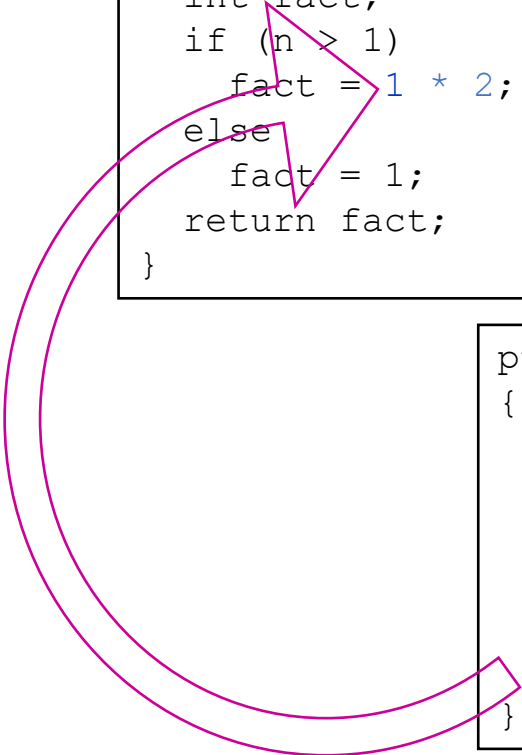




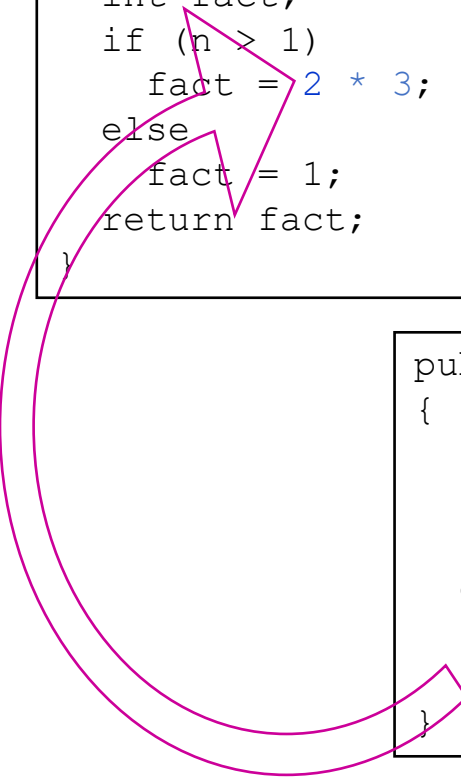
```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return fact;
}
```

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



```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return fact;
}
```



```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return 2;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return 6;
}
```

SR No.	Recursion	Iteration
1)	Terminates when the base case becomes true.	Terminates when the condition becomes false.
2)	Used with functions.	Used with loops.
3)	Every recursive call needs extra space in the stack memory.	Every iteration does not require any extra space.
4)	Smaller code size.	Larger code size.

# Advantages

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals etc. For such problems, it is preferred to write recursive code. We can write such codes iteratively with the help of a stack data structure.