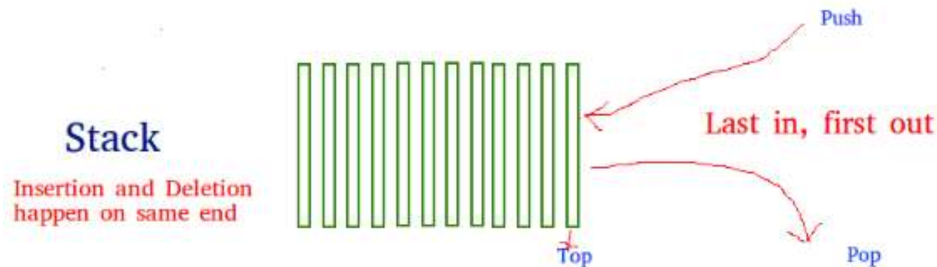
# Stack

# Lecture content:

- Stacks definition
- Fundamental operations
- How to understand a stack practically?
- Mainly basic operations are performed in the stack
- Advantages and Dis-Advantages of stack
- Implementation of stack

# <u>Stacks definition:</u>

**Stack** is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). A stack is a ***limited access data structure*** (**<u>restricted</u>** )- elements can be added and removed from the stack *only at the top.*



**<u>fundamental operations</u>** :

pushing and popping of items at the top of the stack.

## How to understand a stack practically?

There are many real life examples of stack.

Consider the simple example of plates stacked over one another in canteen.

The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottom position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

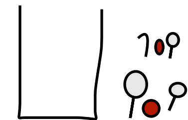
# Mainly basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

- **Peek or Top**: Returns the top element without removing it. Return null or zero if the stack is empty.

- **isEmpty:** Returns true if stack is empty, else false.

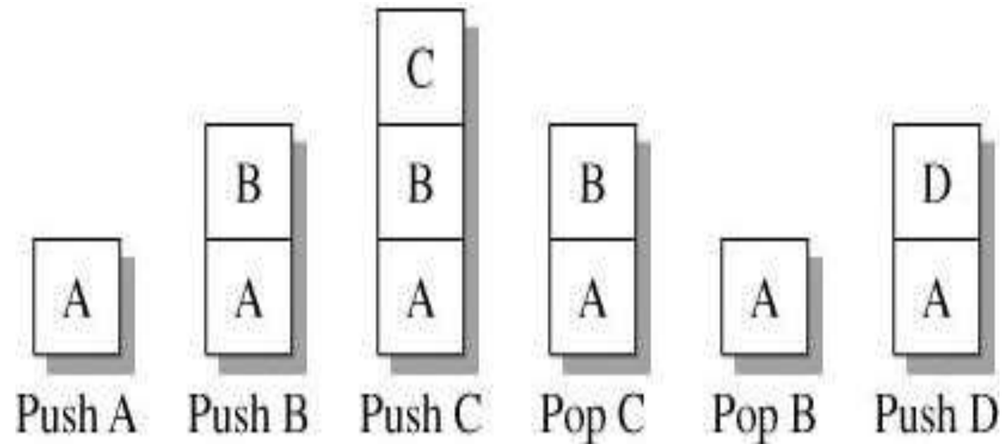


| Method | Return Value | Stack Contents |
|---|---|---|
| push(5) | – | (5) |
| push(3) | – | (5, 3) |
| size() | 2 | (5, 3) |
| pop() | 3 | (5) |
| isEmpty() | false | (5) |
| pop() | 5 | () |
| isEmpty() | true | () |
| pop() | null | () |
| push(7) | – | (7) |
| push(9) | – | (7, 9) |
| top() | 9 | (7, 9) |
| push(4) | – | (7, 9, 4) |
| size() | 3 | (7, 9, 4) |
| pop() | 4 | (7, 9) |
| push(6) | – | (7, 9, 6) |
| push(8) | – | (7, 9, 6, 8) |
| pop() | 8 | (7, 9, 6) |

# Stack Operations (continued)

- An item removed (pop) from the stack is the last element that was added
  (push) to the stack. A stack has LIFO (last-in-first-out) ordering.

  push A
  push B
  push C
  pop
  pop
  pop
  push D




- Stack inserts followed by stack deletions reverse the order of the elements

## Advantages of stack

-Stacks are last in first out so two major applications that Stacks accomplish are things like State and reversing. What I mean by state is a particular setting or structure that a program has at a point in time. Say you are writing a notepad application.

-A good use for a stack would be the undo/redo.

## Dis-Advantage of stack

Stack can not be randomly accessed .

## Implementation of stack:

There are two ways to implement a stack:

-Using array - Using linked list

**Array Implementation:**

```
class ArrayStack
{
    public static final int CAPACITY=1000;
    private String[ ] data;
    private int t=-1;
    public ArrayStack()
      // constructs stack with default capacity
    { this(CAPACITY); }
```

```
public ArrayStack(int c)
  // constructs stack with given capacity
{       data = (String[ ])  new String[c];  }
public int size()
{         return (t + 1); }

public boolean isEmpty()
{         return (t==-1); }
public void push(String e)
{        if (size() == data.length)
                  System.out.println("Stack is full");
        data[++t] = e;  }
```

```
public String top()
{
    if (isEmpty())   return null;    return data[t];
}

public String pop()
  {        if (isEmpty())
           return null;
      String answer = data[t];
      data[t] = null;   t=t-1;
      return answer;
  } } // end class
```

```java
//main
ArrayStack stack1=new ArrayStack(20);

stack1.push("Makkah");
stack1.push("Madinah");
stack1.push("Taief");

System.out.println(stack1.pop());
System.out.println(stack1.pop());
```

```
Run:
Taief
Madinah
```

# Linked list Implementation:

```
class Node
{
        public String elem;      public Node next;
        public Node(String s , Node n)
        {   elem=s;   next=n;   }}

class SLink
{
  protected Node head;      protected long size;
  public SLink()
  {       head=null;    size=0;    }
  public void addFirst(Node v)
  {       v.next=head;
     head=v;        size++;    }
```

```
public String removefirst()
   {
      if(head==null)              return null;
      String M=head.elem;         head=head.next;
      size=size-1;         return  M;
   }
public void displaylist()    {
      Node cur=head;
      for(int k=0 ; k<size ; k++)
      {
         System.out.print(cur.elem + "  ");
         cur=cur.next;
      }
      System.out.println();    }}
```

# main

```
Node N1=new Node("ATL",null);
S1.addFirst(N1);
N1=new Node("MSP",null);
S1.addFirst(N1);
N1=new Node("LAX",null);
S1.addFirst(N1);
S1.displaylist();
System.out.println();
System.out.println(S1.removefirst());
System.out.println();
S1.displaylist();
```

```
run:
LAX  MSP  ATL

LAX

MSP  ATL
```

# lecture 7: Stack applications

# Lecture content:

- Stack Applications
  1. Redo-undo features at many places like editors, Photoshop.
  2. Forward and backward feature in web browsers.
  3. Balancing of symbols.
  4. Infix to Postfix /Prefix conversion.
  5. Recursion and the Runtime Stack.
  6. Used in many algorithms like tree traversals

**Application 1:**

Text editors usually provide an “undo” mechanism that cancels recent editing operations and reverts to former states of a document.

This undo operation can be accomplished by keeping text changes in a stack.

**Application 2:**

Internet Web browsers store the addresses of recently visited sites on a stack.

Each time a user visits a new site, it’s address “pushed” onto the stack.
The browser then allows the user to “pop” back to previously visited sites using the “back” button.

## Application 3:

**Check for balanced parentheses in an expression**

Given an expression string exp , write a program to examine whether the pairs and the orders of "{","}","(",")","[","]" are correct in exp.

For example, the program should print true for exp = "[()]{}{[()()]()}" and false for exp = "[(])"

**Algorithm:**

1) Declare a character stack S.
2) Now traverse the expression string exp.

a) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.

b) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and
if the popped character is the matching starting bracket then fine
else parenthesis are
not balanced.

3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

# Application 4:

## Infix to Postfix /Prefix conversion

**Infix expression:** The expression of the form a op b. When an operator is in-between every pair of operands.

**Infix notation: X + Y**

**Postfix expression:** The expression of the form a b op. When an operator is followed for every pair of operands.

Postfix notation (also known as "Reverse Polish notation"): **X Y +**

**Prefix expression:** The expression of the form op a b. When an operators are written before their operands.

Prefix notation (also known as "Polish notation"): **+ X Y**

**Why postfix representation of the expression?**

The compiler scans the expression either from left to right or from right to left.

Consider the below expression:

a+b*c+d

The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is: abc*+d+. The postfix/prefix expressions can be evaluated easily using a stack.

**Algorithm (infix → postfix)**

**1.** Scan the infix expression from left to right.
**2.** If the scanned character is an operand, output it.
**3.** Else,
**3.1** If the precedence of the scanned operator is greater than the precedence of the operator
in the stack(or the stack is empty), push it.
**3.2** Else, Pop the operator from the stack until the precedence of the scanned operator is
less-equal to the precedence of the operator residing on the top of the stack. Push the
scanned operator to the stack.
**4.** If the scanned character is an '(', push it to the stack.
**5.** If the scanned character is an ')', pop and output from the stack until an '(' is encountered.
**6.** Repeat steps 2-6 until infix expression is scanned.
**7.** Pop and output from the stack until it is not empty.

# Infix to Postfix Conversion

**Example 1:** convert the expression 3+4*5/6 to postfix

3+4*5/6
Stack:
Output:

3+4*5/6
Stack:
Output: 3

3+4*5 / 6
Stack: +
Output: 3

3+4*5 / 6

Stack: +

Output: 3 4

3+4*5/6

Stack: + *

Output: 3 4

3+4*5/6

Stack: + *

Output: 3 4 5

3+4*5/6

Stack: +

Output: 3 4 5 *

3+4*5/6

Stack: + /

Output: 3 4 5 *

3+4*5/6

Stack: + /

Output: 3 4 5 * 6

3+4*5/6

Stack: +

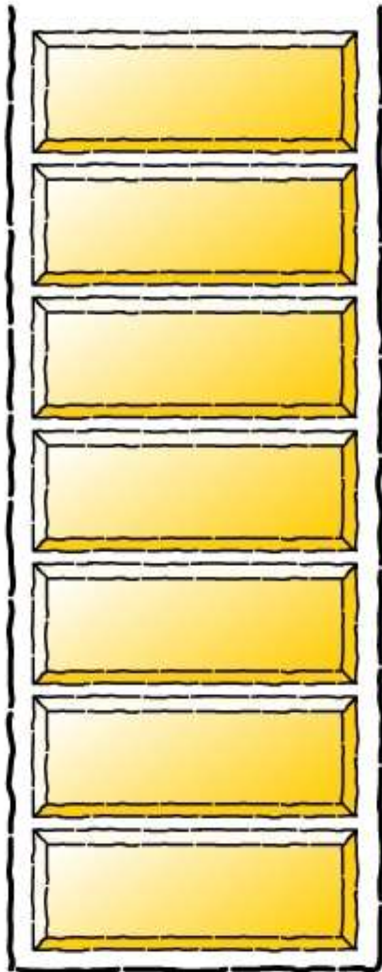
Output: 3 4 5 * 6 /

3+4*5/6

Stack:

Output: 3 4 5 * 6 / +

# Infix to postfix conversion

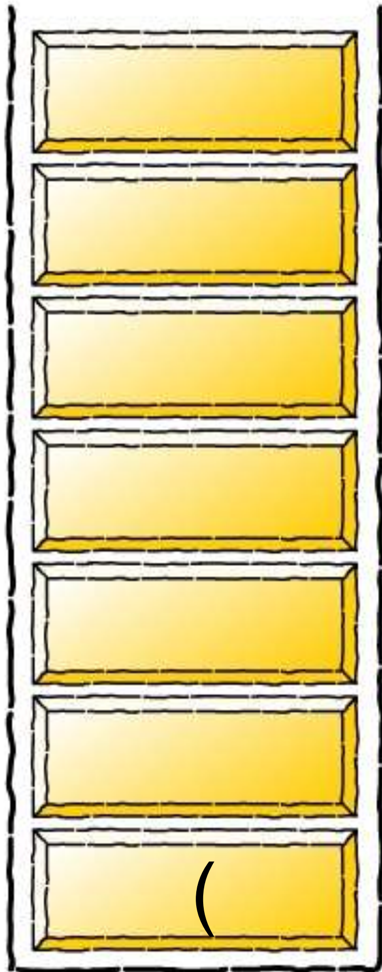


infixVect

( a + b - c ) * d – ( e + f )

postfixVect

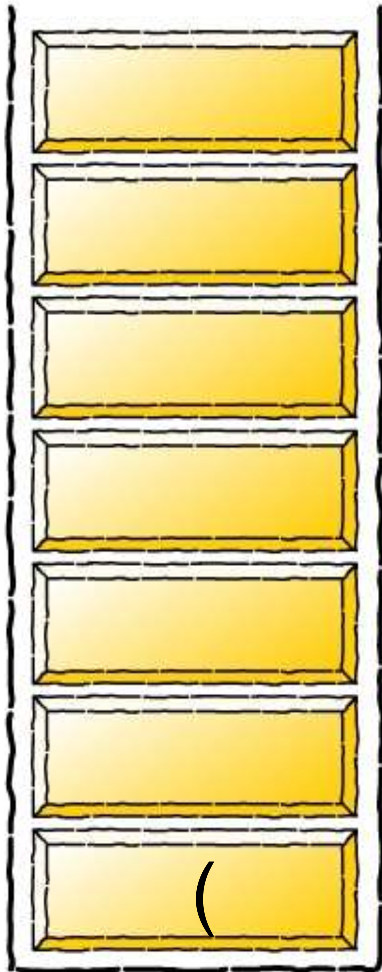
stackVect




infixVect

a + b - c ) * d – ( e + f )

postfixVect



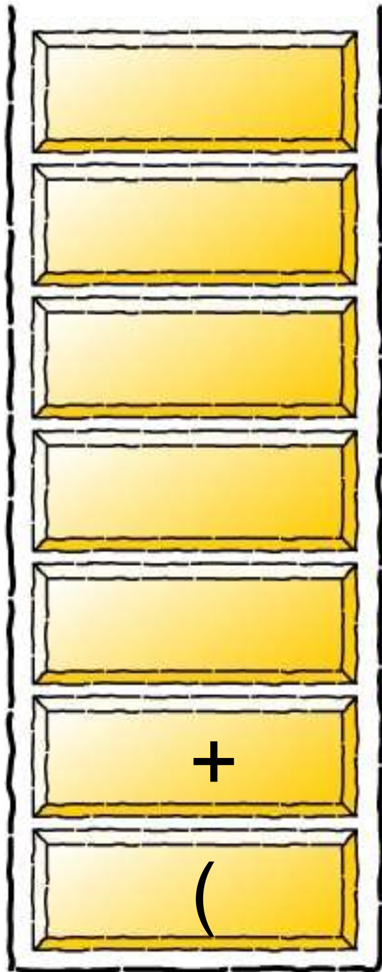

infixVect

+ b - c ) * d – ( e + f )

postfixVect

a

stackVect



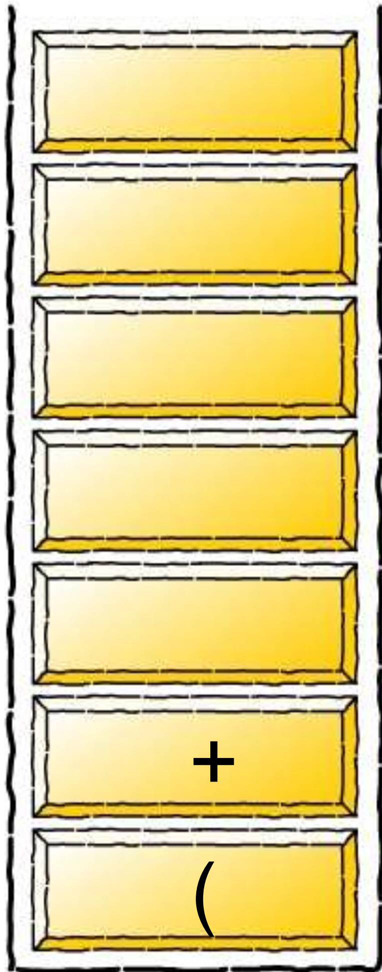

infixVect

b - c ) * d – ( e + f )

postfixVect

a

stackVect



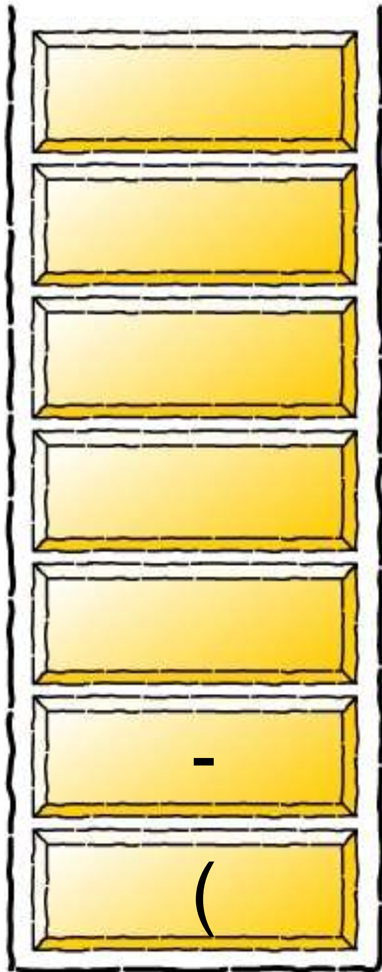

infixVect

- c ) * d – ( e + f )

postfixVect

a b

stackVect

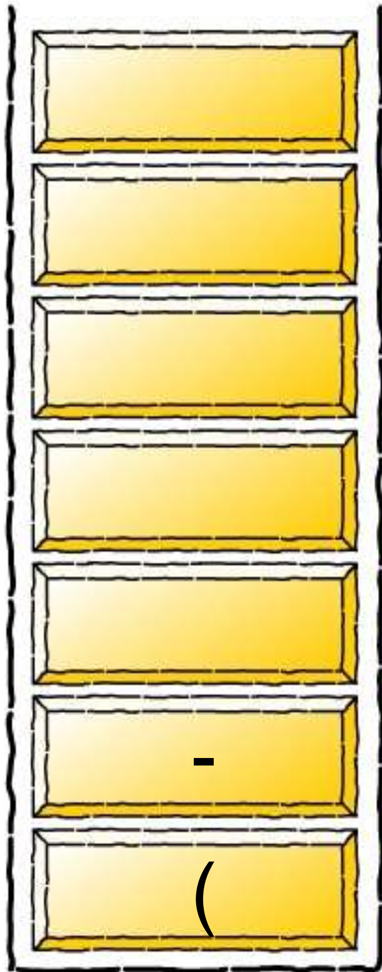



infixVect

c ) * d – ( e + f )

postfixVect

a b +



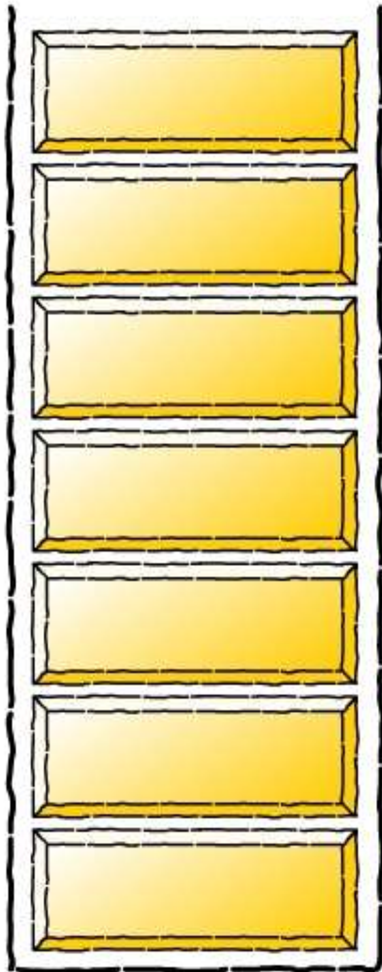

infixVect

) * d – ( e + f )

postfixVect

a b + c

stackVect



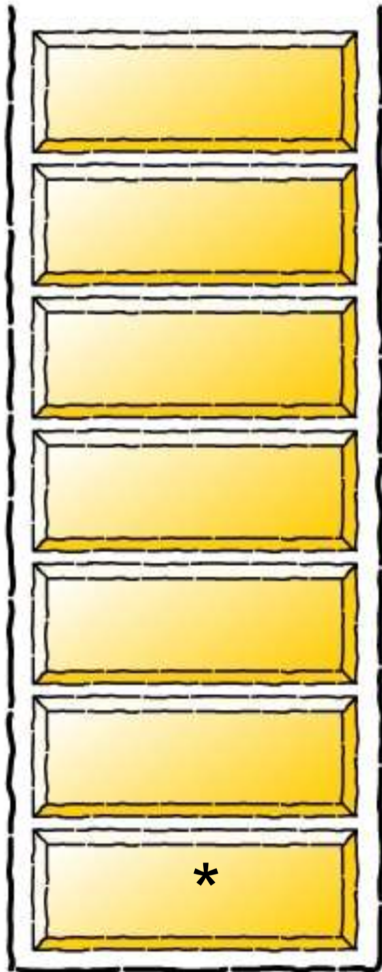
infixVect

* d – ( e + f )

postfixVect

a b + c -

stackVect



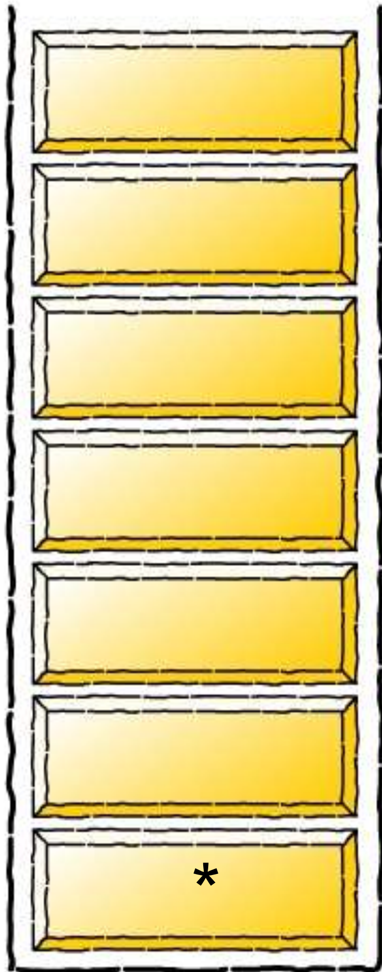

infixVect

d – ( e + f )

postfixVect

a b + c -

stackVect



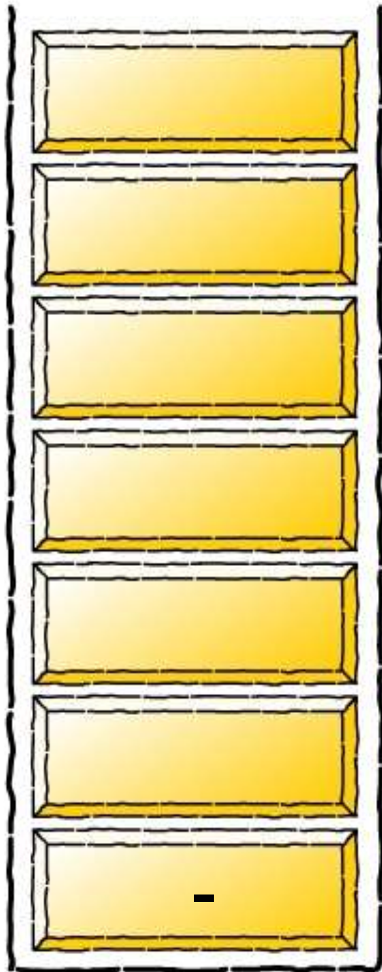

infixVect

– ( e + f )

postfixVect

a b + c - d

stackVect

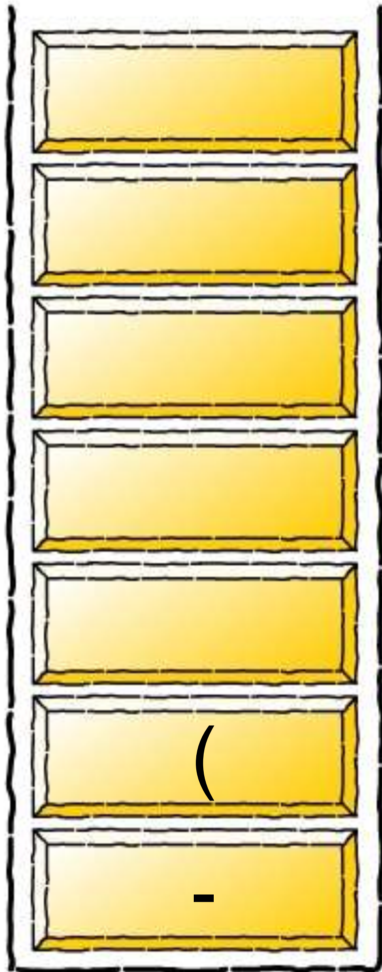



infixVect

( e + f )

postfixVect

a b + c – d *



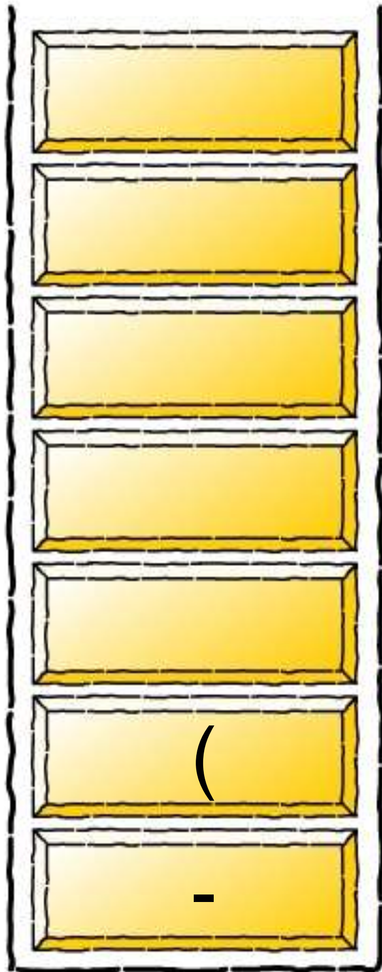

infixVect

e + f )

postfixVect

a b + c – d *

stackVect



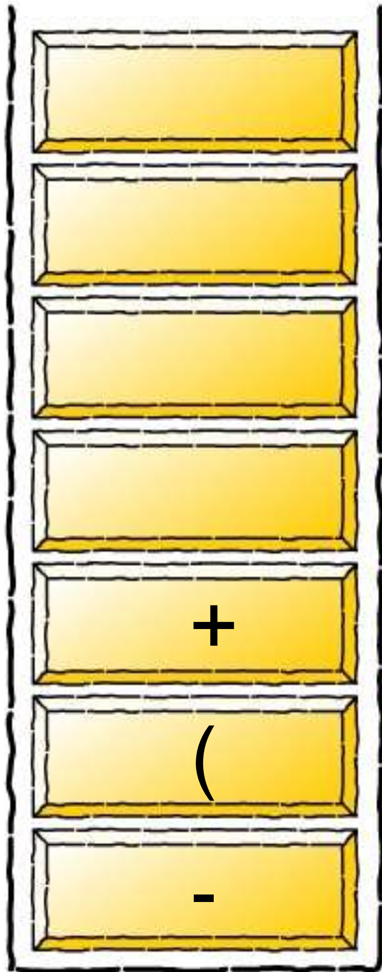

infixVect

+ f )

postfixVect

a b + c – d * e

stackVect



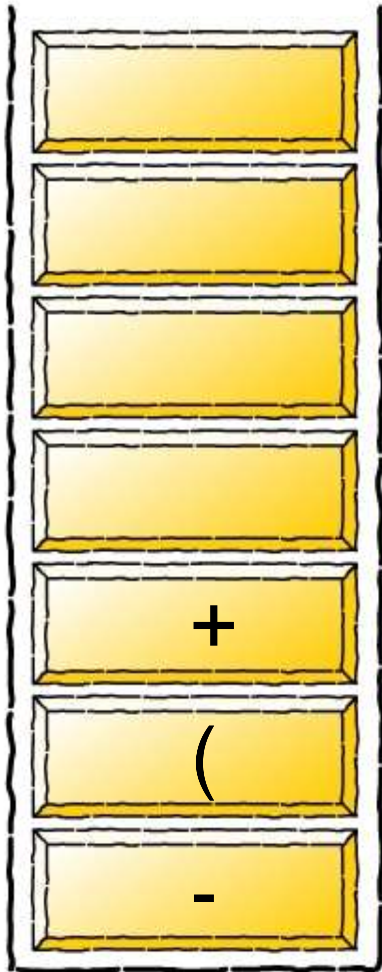

infixVect

f )

postfixVect

a b + c – d * e

stackVect



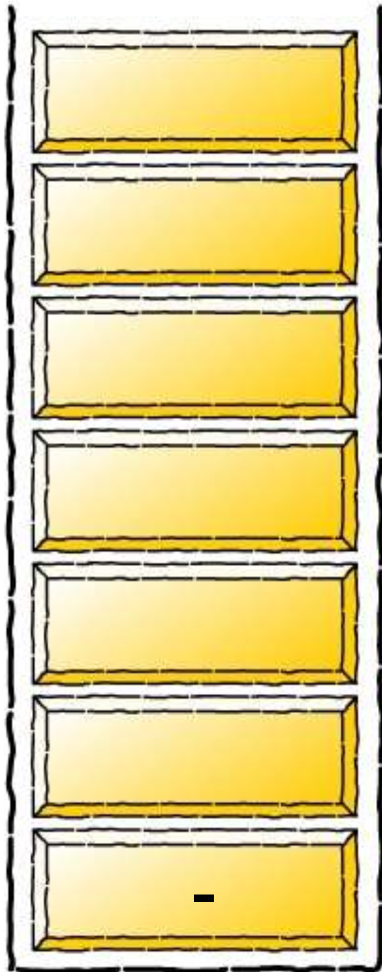

infixVect

)

postfixVect

a b + c – d * e f

stackVect

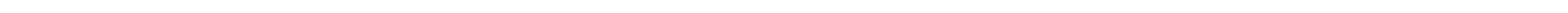



infixVect

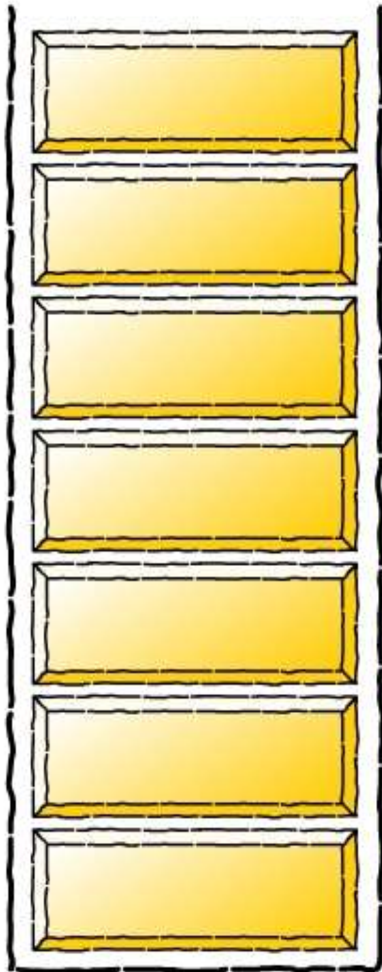
postfixVect

a b + c – d * e f

+



stackVect



infixVect

postfixVect

## Convert the following expression from infix → postfix:

(300+23)*(43-21)/(84+7)

**solution:**

Output: 300 23 + 43 21 - * 84 7 + /

## Infix to prefix conversion

Expression = **(A+B^C)*D+E^5**

**Step 1.** Reverse the infix expression.

**5^E+D*)C^B+A(**

**Step 2.** Make Every '(' as ')' and every ')' as '('

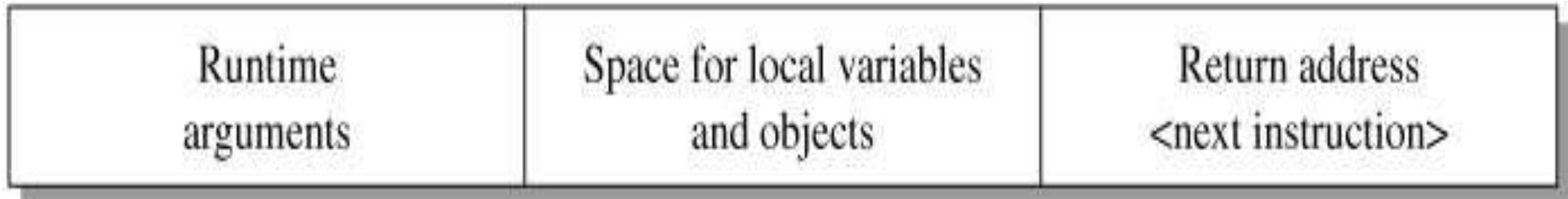
**5^E+D*(C^B+A)**

**Step 3.** Convert expression to postfix form.

## Application 5:

### Recursion and the Runtime Stack

- For method execution, the runtime system creates an activation record for the parameters and return address.




Activation Record

The record is pushed on a runtime stack when the method is called and then popped from the stack when the method finishes execution. The program continues at the return address.

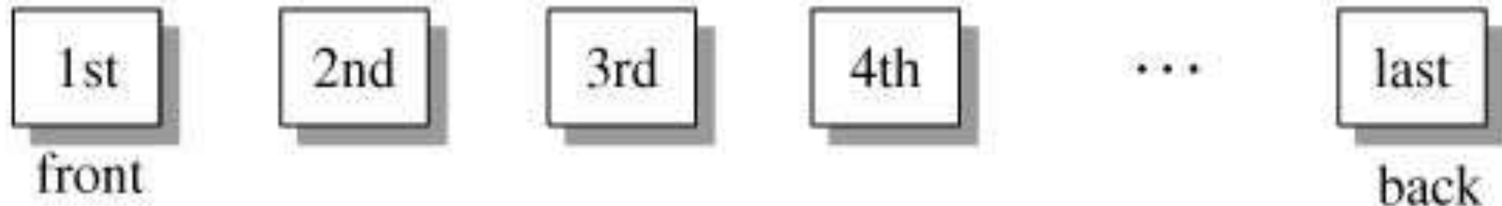
# lecture 8: queue

# Lecture content:

- **Queue Definition**
- **queue processing**
- **Building a Queue Class**
- **Queues in a Computer System**
- **Implementing a Queue Class**
- **Queue Structure**
- **Queue Class Methods**
- **Fundamental operations**

# Queue Definition:

- A **queue** is a list of items that allows for access only at the two ends of the sequence, called the front and back of the queue. An item enters at the back and exits from the front.



## ➢ Queue definition

Queue means ‘waiting line’, which is very similar to queues in real life:-

Such that:

1. **a queue of people standing in an airport’s check-in gate;**
2. **a queue of cars waiting for green light in a road in the city;**
3. **a queue of customers waiting to be served in a bank’s counter, etc.**

In programming, **queue** is a data structure that holds elements prior to processing, similar to queues in real-life scenarios. Let’s consider a queue holds a list of waiting customers in a bank’s counter. Each customer is served one after another, follow the order they appear or registered. The first customer comes is served first, and after him is the 2nd, the 3rd, and so on.

- When serving a customer is done, he or she lefts the counter (removed from the queue), and the next customer is picked to be served next.

- Other customers come later are added to the end of the queue.

- This processing is called First In First Out or FIFO.

## Queue definition cont.

In computer science, a **queue** is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principle (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue,* and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a First-In-First-Out (FIFO) data structure.

In a FIFO data structure, the first element added to the queue will be the first one to be removed.
This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed.

Often a *peek* or *front* operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

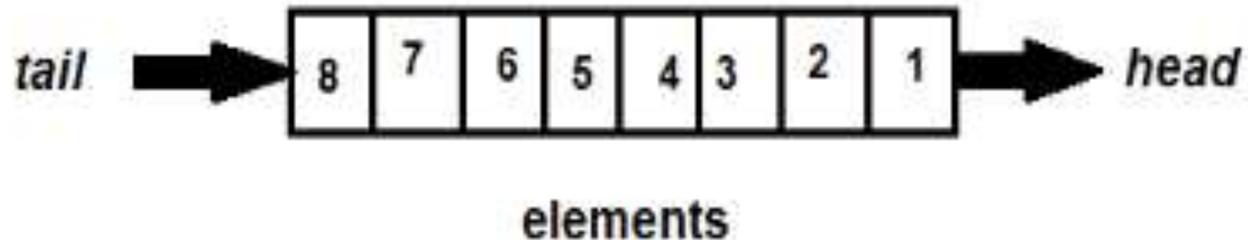
## ➢Fundamental operations

| | |
|---|---|
| enqueue(e) | Adds element e to the back of queue. |
| dequeue() | Removes and returns the first element from the queue (or null if the queue is empty). |
| first() | Returns the first element of the queue, without removing it (or null if the queue is empty). |
| size() | Returns the number of elements in the queue |
| isEmpty() | Return a boolean if the queue is empty. |

| **Method** | **Return Value** | first $\leftarrow Q \leftarrow$ last |
| --- | --- | --- |
| enqueue(5) | – | (5) |
| enqueue(3) | – | (5, 3) |
| size( ) | 2 | (5, 3) |
| dequeue( ) | 5 | (3) |
| isEmpty( ) | false | (3) |
| dequeue( ) | 3 | ( ) |
| isEmpty( ) | true | ( ) |
| dequeue( ) | null | ( ) |
| enqueue(7) | – | (7) |
| enqueue(9) | – | (7, 9) |
| first( ) | 7 | (7, 9) |
| enqueue(4) | – | (7, 9, 4) |

## ➢ Characteristics of Queue

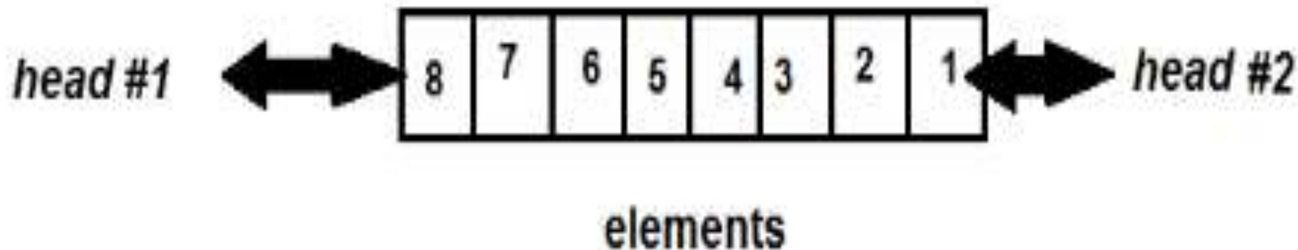
Basically, a queue has a head and a tail.
New elements are added to the tail, and to-be-processed elements are picked from the head.
The following picture illustrates a typical queue:




Elements in the queue are maintained by their insertion order.

## ➢ Characteristics of Queue

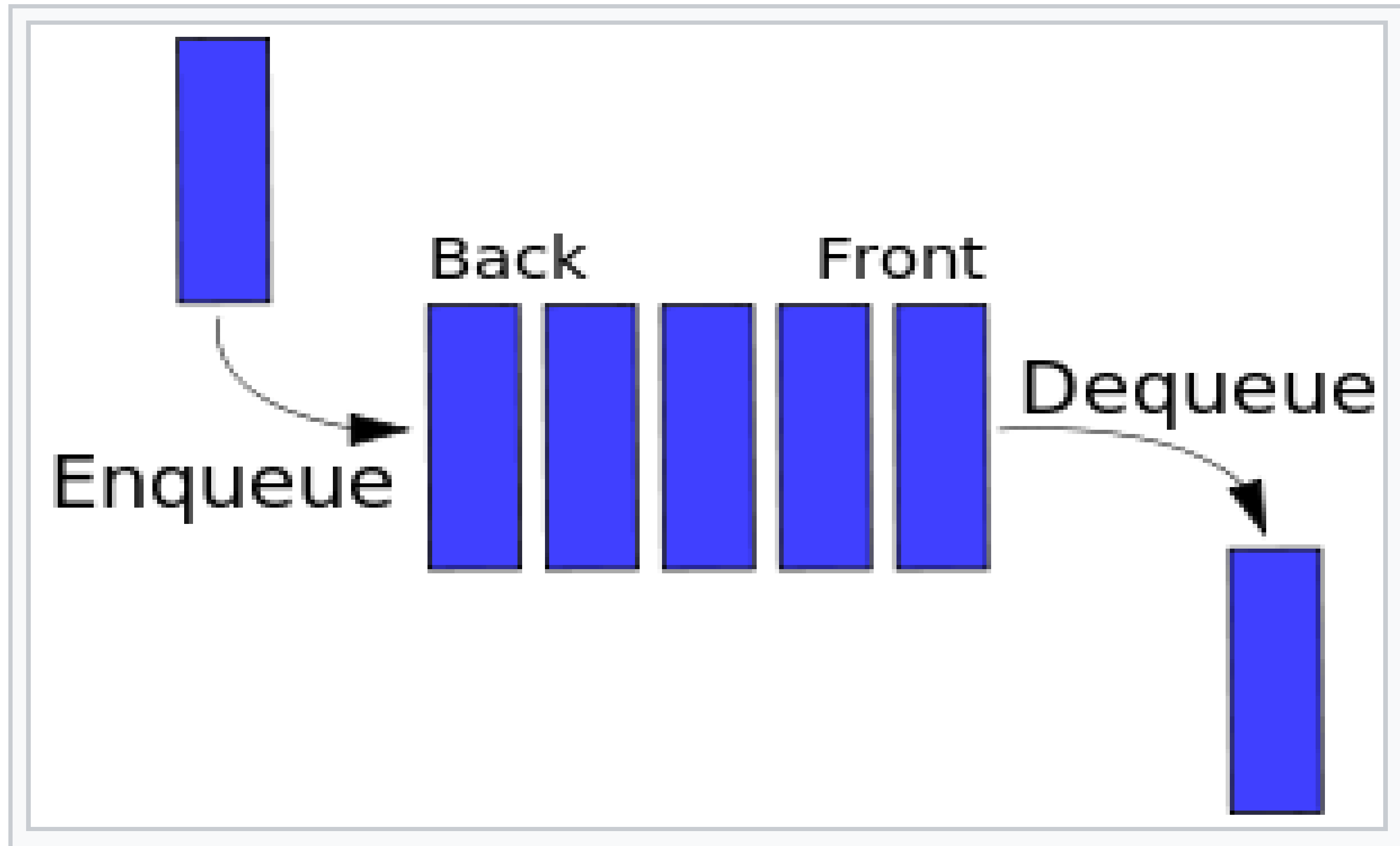
Another kind of queue is double ended queue, or deque.
**A deque** has two heads, allowing elements to be added or removed from both ends. The following picture illustrates this kind of queue:



# queue processing

- This queue processing is called First In First Out or FIFO.
- In a FIFO data structure, the first element added to the queue will be the first one to be removed.

- This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed.


Back
Front
Dequeue
Enqueue

# Queues in a Computer System

- When a process (program) requires a certain resource
  - ***printer***
  - ***disk access on a network***
  - ***characters in a keyboard buffer***
- Queue Manipulation Operations
  - `isEmpty()`: returns `true` or `false`
  - `first()`: returns copy of value at front
  - `add(v)`: adds a new value at rear of queue
  - `remove()`: removes, returns value at front

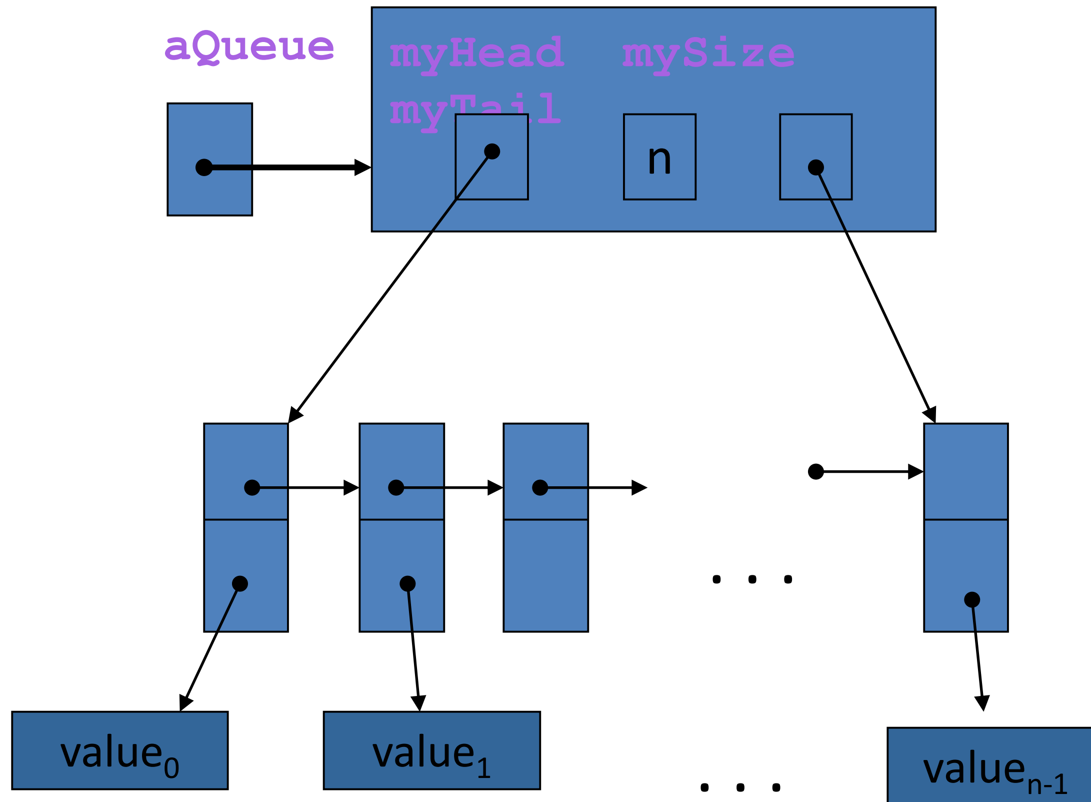
# Implementing a `Queue` Class

- Implement as a `LinkedList` attribute value
  insertions and deletions from either end are efficient, occur in constant O(1) time
  - good choice
- Implement as an `ArrayList` attribute
  - poor choice
  - adding values at one end, removing at other end require multiple shifts

# Implementing a `Queue` Class

- Build a `Queue` from scratch
  - build a linked structure to store the queue elements
- Attributes required
  - handle for the head node
  - handle for tail node
  - integer to store number of values in the queue
  - use `SinglyLinkedNode` class, source code

# Queue Structure



# `Queue` Class Methods

- Constructor
  - set `myHead`, `myTail` to null
  - set `mySize` to zero
- `isEmpty()`
  - return results of comparison `mySize == 0`
- `front()`
  - return `myHead.getValue()`

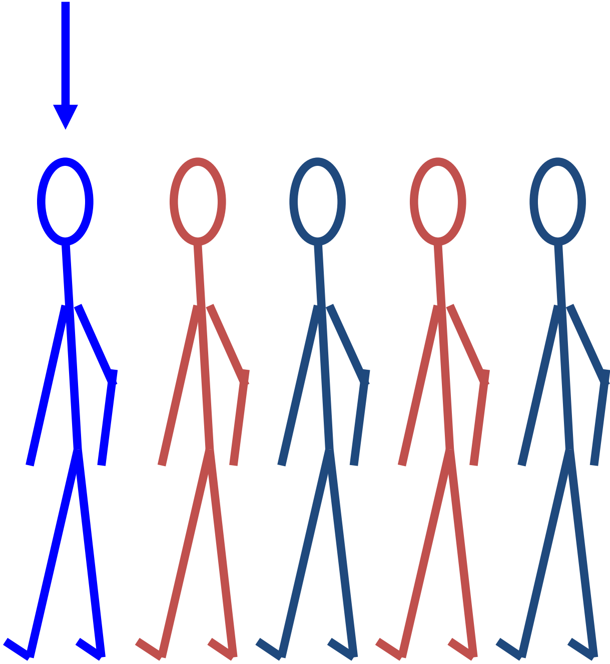
    // unless empty

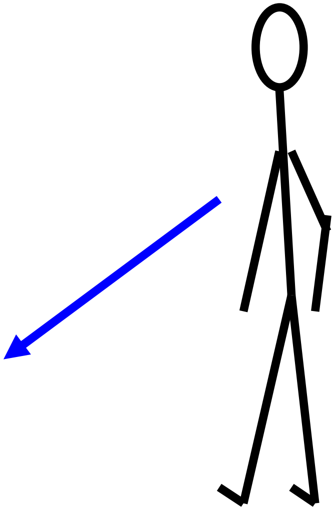
# `Queue` Class Methods

- `add()`
  - create new node, update attribute variables
  - if queue is empty, must also update `myHead`
- `remove()`
  - must check if class not empty
    otherwise …
  - save handle to first object
  - adjust head to refer to node
  - update `mySize`

# Adding an element

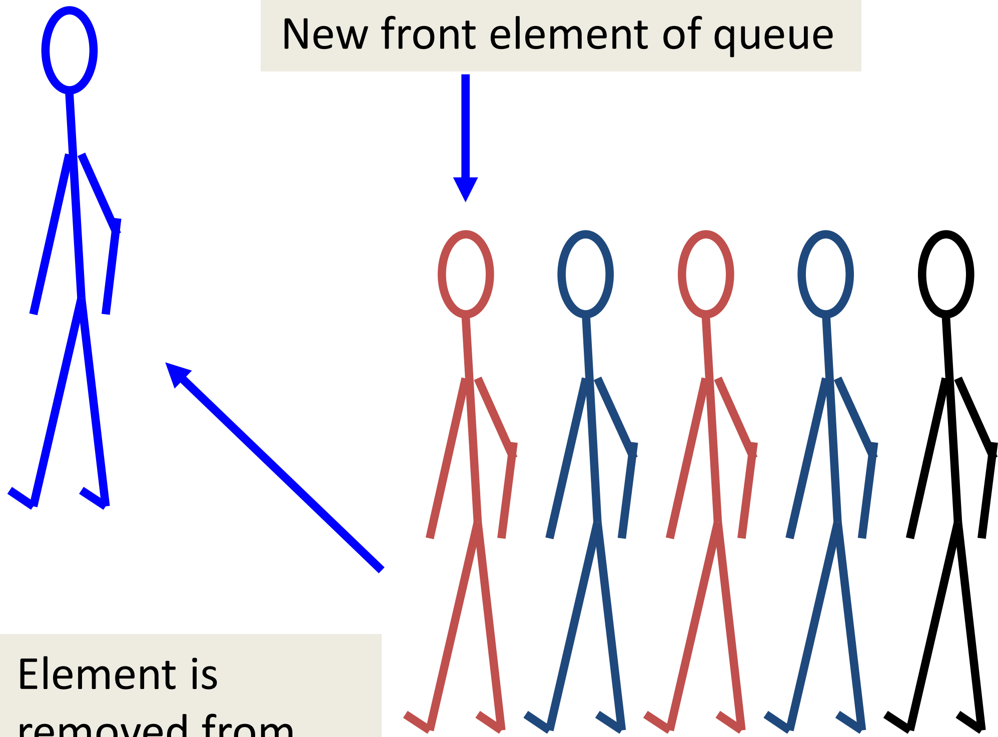
Front of queue





New element is added to the rear of the queue

# Removing an element




Element is removed from the front of the queue

**Array-Based Queue Implementation:**

```
class ArrayQueue
{    private int[ ] data;
    public static final int CAPACITY=1000;
    private int f = 0;      public int size = 0;
    public ArrayQueue()
    {              this(CAPACITY);            }

public ArrayQueue(int capacity)
{       data = new int[capacity]; }
public boolean isEmpty()
{       return (size == 0); }
```

## Enqueue method:

-add a new element to the back of the queue.

-To determine the proper مناسب index at which to place the new element.

avail = (f + size ) % data.length;

```
public void enqueue(int e)
{
    if (size == data.length)
        System.out.println("Queue is full");
    int avail = (f + size ) % data.length;
    data[avail] = e;
    size ++;
}
```

```
public int first()
{
    if (isEmpty())      return 0;    return data[f];
}

public int dequeue()
  {       if (isEmpty())            return 0;
    int answer = data[f];        //data[f] = null;
    f = (f + 1) % data.length;
    size = size -1;
    return answer;
  }}
```

```
public class Qq
{
   public static void main(String[] args)
   {         ArrayQueue Q1=new ArrayQueue(4);
Q1.enqueue(155);       Q1.enqueue(100); Q1.enqueue(123);     Q1.enqueue(247);

      System.out.println( "deleted elem: " + Q1.dequeue());
      System.out.println( "deleted elem: " + Q1.dequeue());

      Q1.enqueue(331);       Q1.enqueue(779);

      System.out.println( "deleted elem: " + Q1.dequeue());
      System.out.println( "deleted elem: " + Q1.dequeue());
      System.out.println( "deleted elem: " + Q1.dequeue());
      System.out.println( "deleted elem: " + Q1.dequeue());
   }}
```

```
run:
deleted elem: 155
deleted elem: 100
deleted elem: 123
deleted elem: 247
deleted elem: 331
deleted elem: 779
```

# Compare between Stack & Queue ?

| Comparison aspect | Stack | Queue |
| --- | --- | --- |
| Working principle | stack uses LIFO (last in first out) | Queue uses FIFO (First in first out) |
| Structure | Stack has only one end open for pushing and popping the data elements | Queue has both ends open for enqueuing and dequeuing the data elements |
| **example** | The stack is a stack of CD's where you can take out and put in CD through the top of the stack of CDs. | The queue is a queue for Theatre tickets where the person standing in the first place, i.e., front of the queue will be served first. |
| Number of pointers used | One | Two (In simple queue case) |
| Operations performed | Push and Pop | Enqueue and dequeue |
| Examination of empty condition | Top == -1 | Front == -1 |
| Implementation | Simpler | Comparatively complex |