
# COMP242

# Data Structure

# Lectures Note: Stacks and Queues

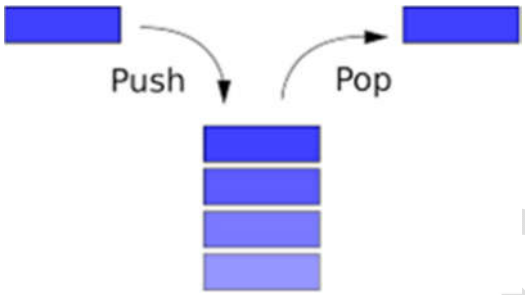
Prepared by: Dr. Mamoun Nawahdah

**2016/2017**

# Stacks

**Stack** is an abstract data type that serves as a collection of elements, with two principal operations:

- **push** adds an element to the collection;
- **pop** removes the last element that was added.




- **Last In, First Out ➔ LIFO**

**ABSTRACT DATA TYPE: STACK**

**DATA**

- A collection of objects in reverse chronological order and having the same data type

**OPERATIONS**

| PSEUDOCODE | UML | DESCRIPTION |
|---|---|---|
| `push(newEntry)` | `+push(newEntry: T): void` | Task: Adds a new entry to the top of the stack.<br>Input: `newEntry` is the new entry.<br>Output: None. |
| `pop()` | `+pop(): T` | Task: Removes and returns the stack's top entry.<br>Input: None.<br>Output: Returns the stack's top entry. Throws an exception if the stack is empty before the operation. |
| `peek()` | `+peek(): T` | Task: Retrieves the stack's top entry without changing the stack in any way.<br>Input: None.<br>Output: Returns the stack's top entry. Throws an exception if the stack is empty. |
| `isEmpty()` | `+isEmpty(): boolean` | Task: Detects whether the stack is empty.<br>Input: None.<br>Output: Returns true if the stack is empty. |
| `clear()` | `+clear(): void` | Task: Removes all entries from the stack.<br>Input: None.<br>Output: None. |

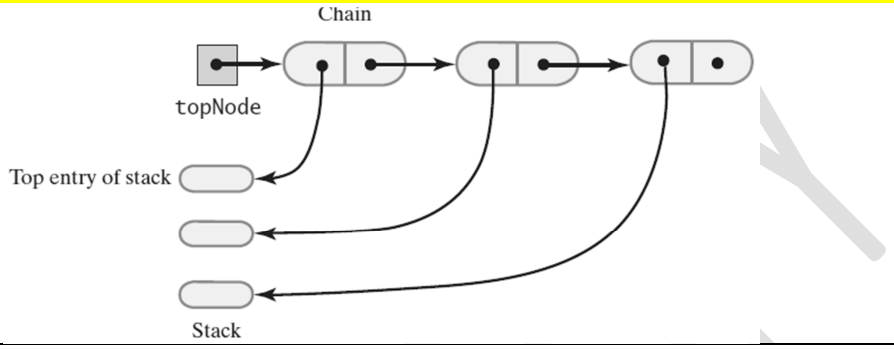
## Single Linked List Implementation:

Each of the following operation involves top of stack

- **push**
- **pop**
- **peek**

**Head or Tail for topNode??**

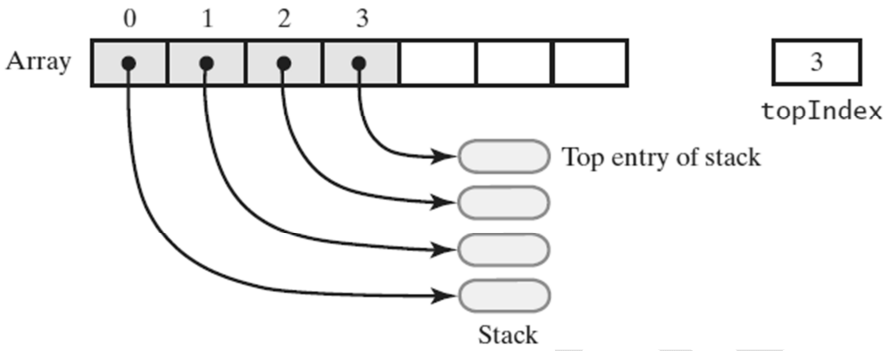
Head of linked list easiest, fastest to access ➔ Let this be the top of the stack




```
public class LinkedStack<T extends Comparable<T>> {
   private Node<T> topNode;

   public void push(T data) {
      Node<T> newNode = new Node<T>(data);
      newNode.setNext(topNode);
      topNode = newNode;
   }

   public Node<T> pop() {
      Node<T> toDel = topNode;
      if(topNode != null)
         topNode = topNode.getNext();
      return toDel;
   }

   public Node<T> peek() {   return topNode; }

   public int length() {
      int length = 0;
      Node<T> curr = topNode;
      while (curr != null) {
         length++;
         curr = curr.getNext();
      }
      return length;
   }

   public boolean isEmpty() {    return (topNode == null);    }

   public  void clear() {  topNode = null;  }
}
```

## Array-Based Implementation:

- End of the array easiest to access
  - Let this be top of stack
  - Let first entry be bottom of stack




```
public class ArrayStack <T> {
   private Object[] s;
   private int n=-1;

   public ArrayStack(int capacity){
      s = new Object[capacity];
   }

   public boolean isEmpty(){ return n ==-1;}
   public int getN(){ return n;}

   public void push(T data){
      s[++n] = data;
   }

   public Object pop(){
      if(!isEmpty())
         return s[n--];
      return null;
   }

   public String toString() {
      String res = "Top-->";
      for(int i=n; i>=0;i--)
         res+="["+s[i]+"]-->";
      return res+"Null";
   }
}
```

# Iteration (Optional)

**Design challenge:** Support iteration over stack items by client, without revealing the internal representation of the stack.

- **Java solution**. Make stack implement the **java.lang.Iterable** interface.

Q. What is an `Iterable` ?
A. Has a method that returns an `Iterator`.

Iterable interface

```
public interface Iterable<Item> {
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator` ?
A. Has methods `hasNext()` and `next()`.
Q. Why make data structures `Iterable` ?
A. Java supports elegant client code.

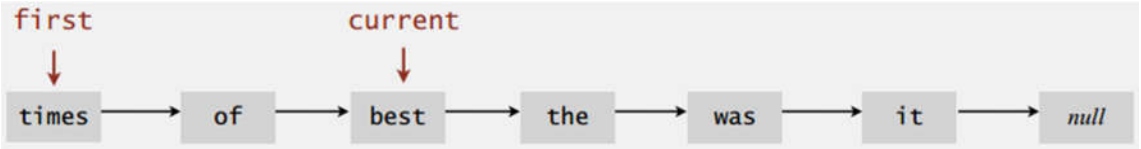
Iterator interface

```
public interface Iterator<Item> {
   boolean hasNext();
   Item next();
   void remove();   <-- optional; use at your own risk
}
```

```
import java.util.Iterator;
public class LinkedStack<T extends Comparable<T>> implements Iterable<T> {
   :
   public Iterator<T> iterator(){
      return new ListIterator();
   }

   private class ListIterator implements Iterator<T>{
      private Node<T> curr = topNode;
      public boolean hasNext(){return curr!=null;}
      public void remove(){}
      public T next(){
         T t = curr.data;
         curr = curr.next;
         return t;
      }
   }
}
```

```
Iterator<String> itt = ls.iterator();
while (itt.hasNext())
   System.out.println(itt.next());
```

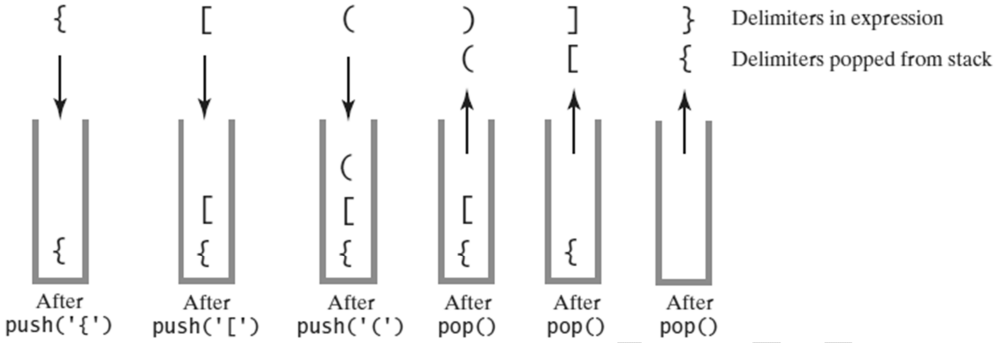
```
for(String s: ls)
   System.out.println(s);
```



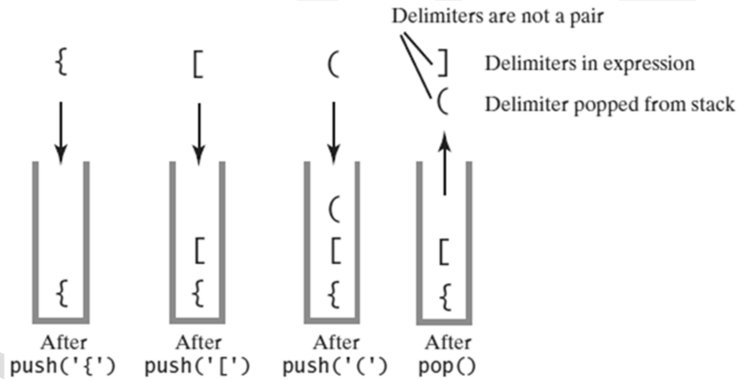
# Balanced Delimiters

**Problem**: Find out if delimiters ( "[{(]})" ) are paired correctly ➔ Compilers

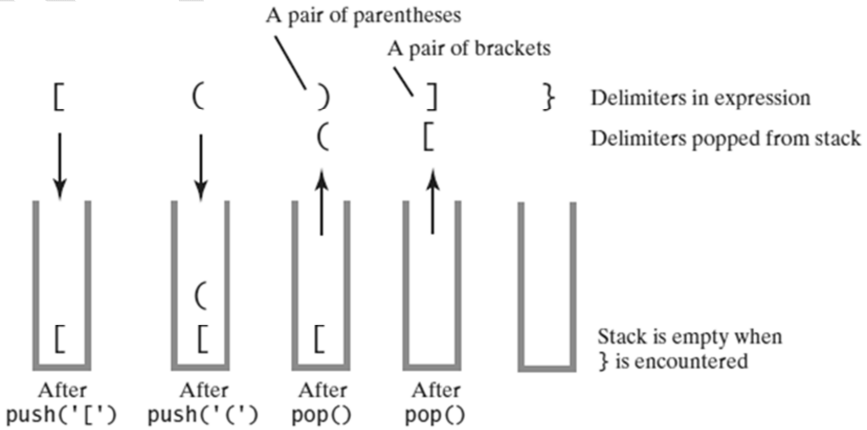
**Example 1:** The contents of a stack during the scan of an expression that contains the **balanced delimiters { [ ( ) ] }**



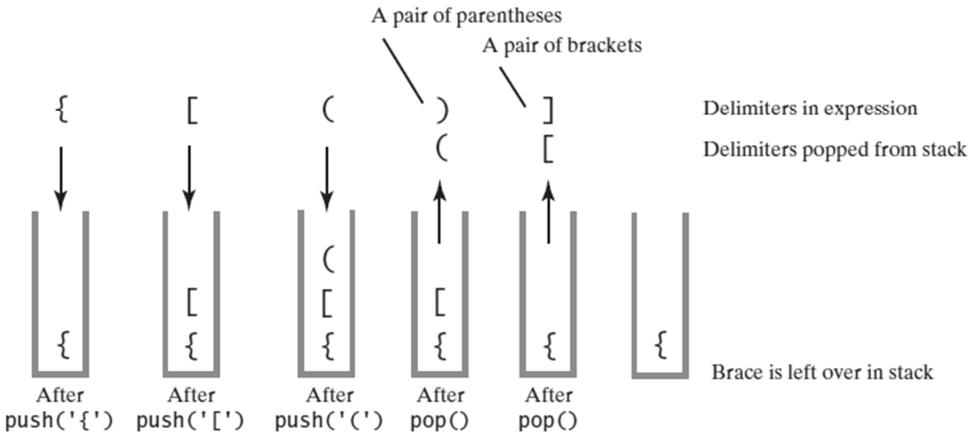

**Example 2:** The contents of a stack during the scan of an expression that contains the **unbalanced delimiters { [ ( ] ) }**




**Example 3:** The contents of a stack during the scan of an expression that contains the **unbalanced delimiters [ ( ) ] }**



**Example 4:** The contents of a stack during the scan of an expression that contains the **unbalanced delimiters { [ ( ) ]**




## Algorithm to process balanced expression:

```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets,
// and braces in an expression are paired correctly.
isBalanced = true
while ((isBalanced == true) and not at end of expression) {
   nextCharacter = next character in expression
   switch (nextCharacter) {
      case '(': case '[': case '{':
         Push nextCharacter onto stack
         break

      case ')': case ']': case '}':
         if (stack is empty)
            isBalanced = false
         else {
             openDelimiter = top entry of stack
             Pop stack
             isBalanced = true or false according to whether openDelimiter
                          and nextCharacter are a pair of delimiters
          }
          break
   }
}
if (stack is not empty)  isBalanced = false
return isBalanced
```

**H.W. implement check balance algorithm using linked list/array stacks**

# Processing Algebraic Expressions

- **Infix**: each binary operator appears between its operands *a + b*
- **Prefix**: each binary operator appears before its operands *+ a b*
- **Postfix**: each binary operator appears after its operands *a b +*

## Evaluate infix expressions:

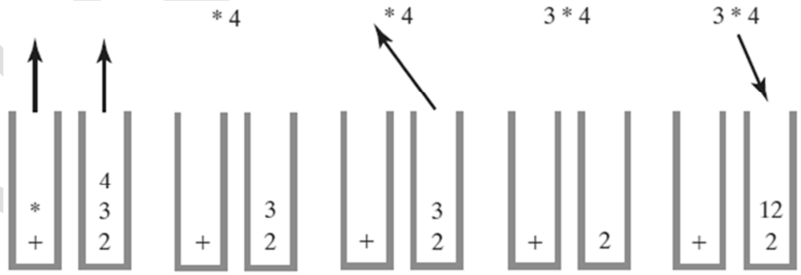

**Two-stack algorithm.** [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

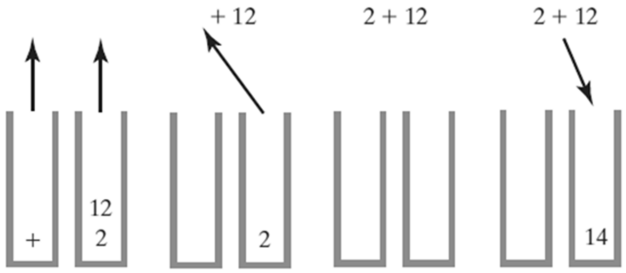
**Example: evaluate a + b * c when a is 2, b is 3, and c is 4:**

**Step 1**: Fill the two stacks until reaching the end of the expression:


**Step 2**: performing the multiplication:




**Step 3**: performing the addition:



**Algorithm to evaluate infix expression:**

```
Algorithm evaluateInfix(infix)
operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process) {
   nextCharacter = next nonblank character of infix
   switch (nextCharacter) {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                precedence of nextCharacter <= precedence of operatorStack.peek()) {
            // Execute operator at top of operatorStack
            topOperator = operatorStack.pop()
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in topOperator and its operands
                     operandOne and operandTwo
            valueStack.push(result)
         }
         operatorStack.push(nextCharacter)
         break
      case '(' :
         operatorStack.push(nextCharacter)
         break
      case ')' : // Stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(') {
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in topOperator and its operands
                     operandOne and operandTwo
            valueStack.push(result)
            topOperator = operatorStack.pop()
         }
         break
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty()) {
  topOperator = operatorStack.pop()
  operandTwo = valueStack.pop()
  operandOne = valueStack.pop()
  result = the result of the operation in topOperator and its operands
           operandOne and operandTwo
  valueStack.push(result)
}
return valueStack.peek()
```

## Infix to Postfix Conversion

| | |
|---|---|
| • Operand | Append each operand to the end of the output expression. |
| • Operator ^ | Push ^ onto the stack. |
| • Operator +, -, *, or / | Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack. |
| • Open parenthesis | Push ( onto the stack. |
| • Close parenthesis | Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses. |

**Example 1:** Converting the **infix** expression **a + b * c** to **postfix** form

| Next Character in Infix Expression | Postfix Form | Operator Stack (bottom to top) |
|---|---|---|
| *a* | *a* | |
| + | *a* | + |
| *b* | *a b* | + |
| * | *a b* | + * |
| *c* | *a b c* | + * |
| | *a b c* * | + |
| | *a b c* * + | |

**Example 2:** Successive Operators with Same Precedence: **a - b + c**

| Next Character in Infix Expression | Postfix Form | Operator Stack (bottom to top) |
|---|---|---|
| *a* | *a* | |
| − | *a* | − |
| *b* | *a b* | − |
| + | *a b* − | |
| | *a b* − | + |
| *c* | *a b* − *c* | + |
| | *a b* − *c* + | |

**Example 3:** Successive Operators with Same Precedence: **a ^ b ^ c**

| Next Character in Infix Expression | Postfix Form | Operator Stack (bottom to top) |
|---|---|---|
| *a* | *a* | |
| ^ | *a* | ^ |
| *b* | *a b* | ^ |
| ^ | *a b* | ^^ |
| *c* | *a b c* | ^^ |
| | *a b c* ^ | ^ |
| | *a b c* ^ ^ | |

**Example 4:** The steps in converting the infix expression **a / b * ( c + ( d – e ) )** to postfix form

| Next Character from Infix Expression | Postfix Form | Operator Stack (bottom to top) |
|---|---|---|
| *a* | *a* | |
| / | *a* | / |
| *b* | *a b* | / |
| * | *a b* / | |
| | *a b* / | * |
| ( | *a b* / | * ( |
| *c* | *a b* / *c* | * ( |
| + | *a b* / *c* | * (+ |
| ( | *a b* / *c* | * (+ ( |
| *d* | *a b* / *c d* | * (+ ( |
| – | *a b* / *c d* | * (+ (– |
| *e* | *a b* / *c d e* | * (+ (– |
| ) | *a b* / *c d e* – | * (+ ( |
| | *a b* / *c d e* – | * (+ |
| ) | *a b* / *c d e* – + | * ( |
| | *a b* / *c d e* – + | * |
| | *a b* / *c d e* – + * | |

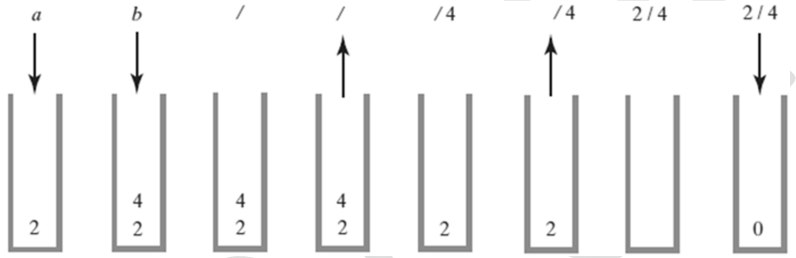
**Infix-to-postfix Algorithm:**

```
Algorithm convertToPostfix(infix)
operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse) {
   nextCharacter = next nonblank character of infix
   switch (nextCharacter) {

      case variable:
         Append nextCharacter to postfix
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                precedence of nextCharacter <= precedence of operatorStack.peek()){
            Append operatorStack.peek() to postfix
            operatorStack.pop()
         }
         operatorStack.push(nextCharacter)
         break
      case '( ' :
         operatorStack.push(nextCharacter)
         break
      case ')' : // Stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(') {
              Append topOperator to postfix
              topOperator = operatorStack.pop()
          }
          break
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty()) {
   topOperator = operatorStack.pop()
   Append topOperator to postfix
}
return postfix
```

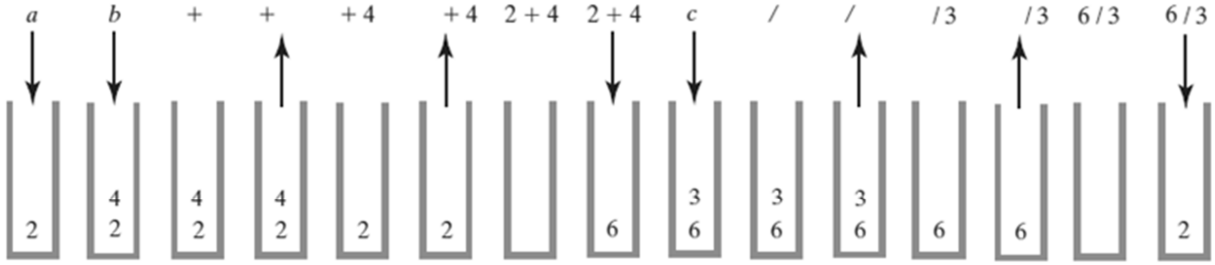
## Evaluating Postfix Expressions

- When an **operand** is seen, it is **pushed** onto a stack.
- When an **operator** is seen, the appropriate numbers of **operands** are **popped** from the stack, the operator is **evaluated**, and the result is **pushed** back onto the stack.
  - Note that the **1st** item popped becomes the (right hand side) **rhs** parameter to the binary operator and that the **2nd** item popped is the (left hand side) **lhs** parameter; thus **parameters are popped in reverse order**.
  - For addition and multiplication, the order does not matter, but for subtraction and division, it does.
- When the complete postfix expression is evaluated, the result should be a single item on the stack that represents the answer.

**Example 1:** The stack during the evaluation of the postfix expression ***a b /*** when *a* is **2** and **b** is 4




**Example 2:** The stack during the evaluation of the postfix expression **a b + c /** when ***a*** is ***2***, ***b*** is ***4***, and ***c*** is ***3***




**Self exercises:**

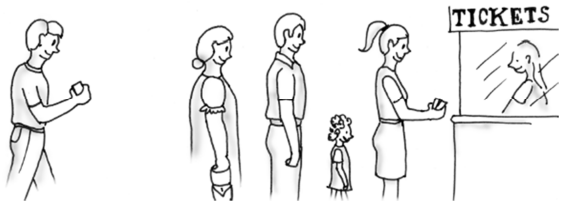
- **2 3 4 + * 6 -** ➔ **8.0**
- **2 3 + 7 9 / -** ➔ **4.222**
- **10 2 8 * + 3 -** ➔ **23.0**
- **1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -** ➔ **-8.67**

**Algorithm for evaluating postfix expressions.**

```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
    nextCharacter = next nonblank character of postfix
    switch (nextCharacter)
    {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' : case '^' :
         operandTwo = valueStack.pop()
         operandOne = valueStack.pop()
         result = the result of the operation in nextCharacter and its operands
                  operandOne and operandTwo
         valueStack.push(result)
         break
      default: break // Ignore unexpected characters
    }
}
```

# Queues

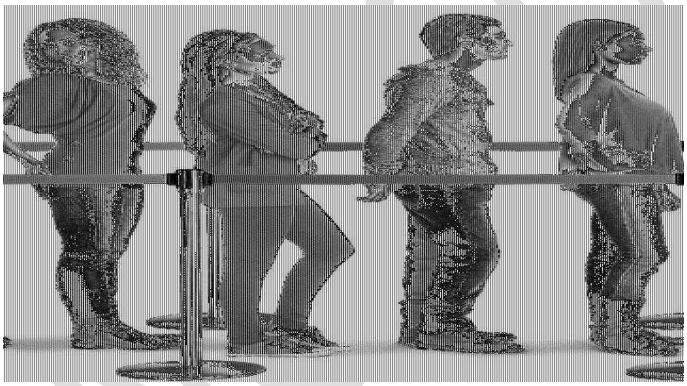
- A **queue** is another name for a **waiting line**:




- Used within operating systems and to simulate real-world events.
  - Come into play whenever processes or events must wait
- Entries organized **first-in**, **first-out**.

**Terminology**

- Item added first, or earliest, is at the front of the queue
- Item added most recently is at the back of the queue
- Additions to a software queue must occur at its back.
- Client can look at or remove only the entry at the front of the queue



**Tail**
**Last**
**Back**

**FIFO: First In First Out**

**Head**
**First**
**Front**

**The ADT Queue**

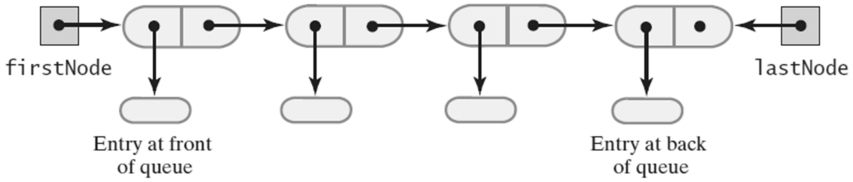
DATA

- A collection of objects in chronological order and having the same data type

OPERATIONS

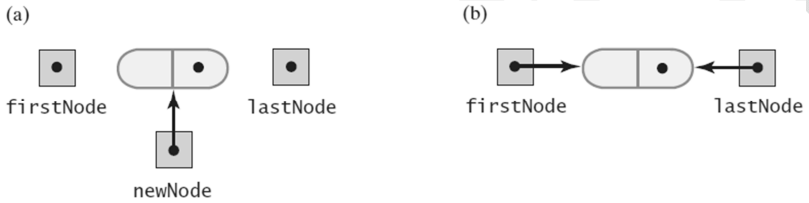
| PSEUDOCODE | UML | DESCRIPTION |
|---|---|---|
| `enqueue(newEntry)` | `+enqueue(newEntry: integer): void` | Task: Adds a new entry to the back of the queue. |
| `dequeue()` | `+dequeue(): T` | Task: Removes and returns the entry at the front of the queue. |
| `getFront()` | `+getFront(): T` | Task: Retrieves the queue's front entry without changing the queue in any way. |
| `isEmpty()` | `+isEmpty(): boolean` | Task: Detects whether the queue is empty. |
| `clear()` | `+clear(): void` | Task: Removes all entries from the queue. |

## Linked-list Representation of a Queue

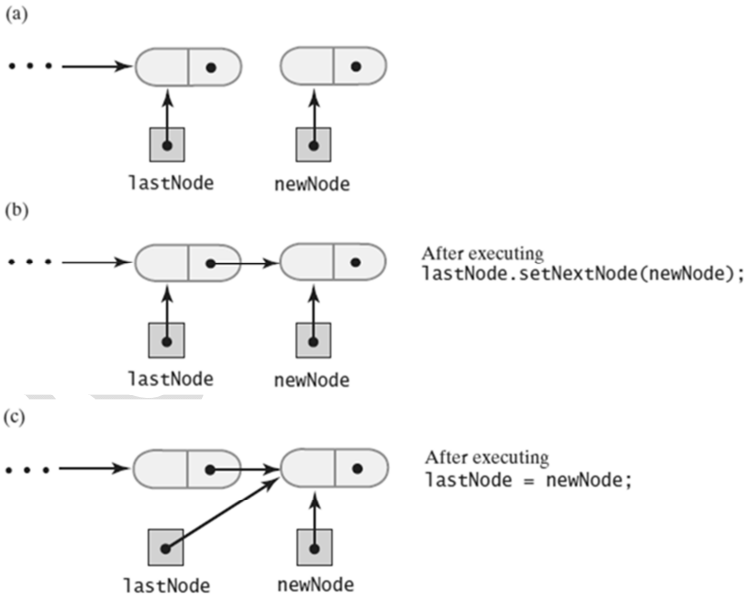



```
public class linkedQueue <T extends Comparable<T>> {
   private Node<T> first;
   private Node<T> last;

   public boolean isEmpty(){   return (first==null) && (last==null);   }
   public void clear(){
      first = null;
      last = null;
   }
}
```

- The definition of **enqueue** Performance is **O(1)**:
  - Adding a new node to an empty chain




  - Adding a new node to the end of a nonempty chain that has a **tail** reference

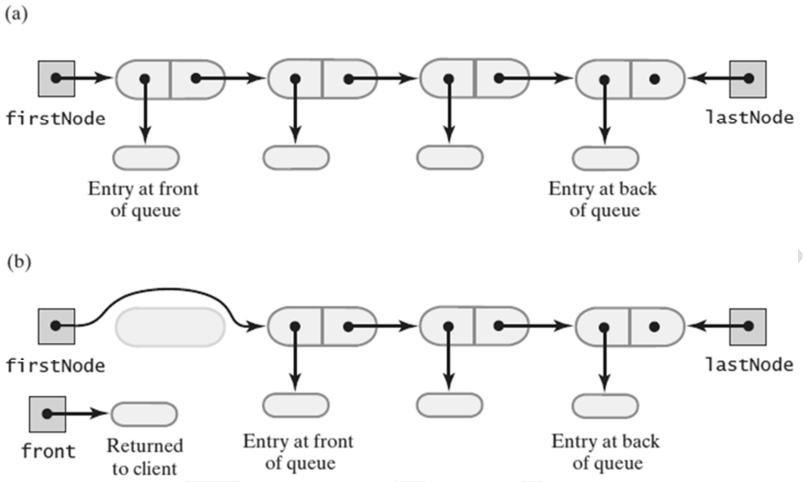



```
public void enqueue(T data){
   Node<T> newNode = new Node<T>(data);
   if(isEmpty())
      first=newNode;
   else
      last.next = newNode;
   last = newNode;
}
```

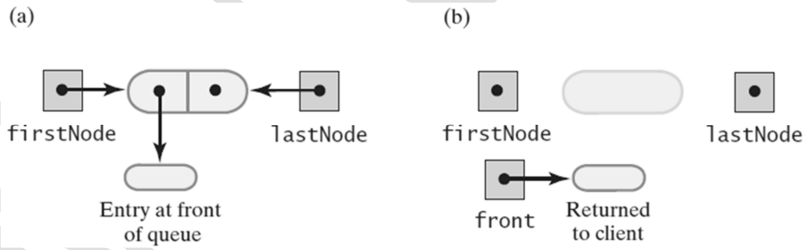
- Retrieving the front entry:

```
public T getFront(){
   if(!isEmpty())
      return first.data;
   return null;
}
```

- Removing the front entry (**dequeue**):
  - A queue of more than one entry:




  - A queue of one entry:

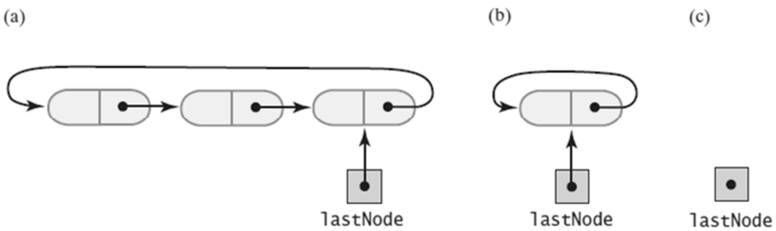



```
public T dequeue(){
   T front = getFront();
   if(!isEmpty())
      first = first.next;
   if(first==null)
      last = null;
   return front;
}
```

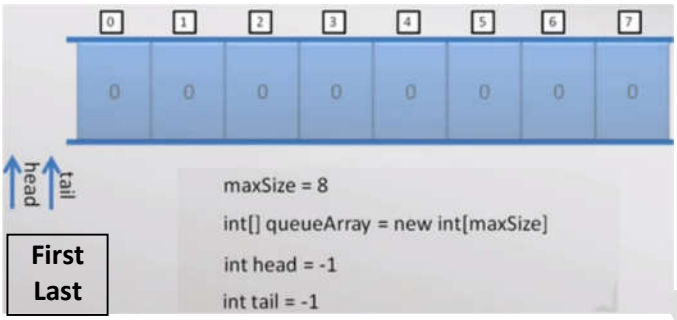
## Circular Linked Implementations of a Queue

A circular linked chain with an external reference to its last node that

a) has more than one node; b) has one node; c) is empty

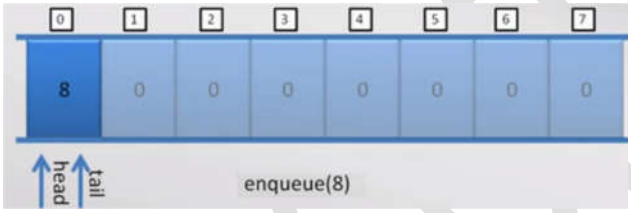


## Array implementation of a Queue



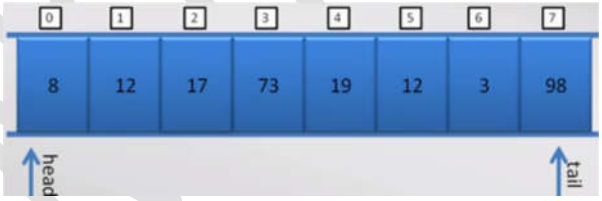

- **enqueue**(): add new item at after **last** (**tail**).
- **dequeue**(): remove item from **first** (**head**).

enqueue(8)



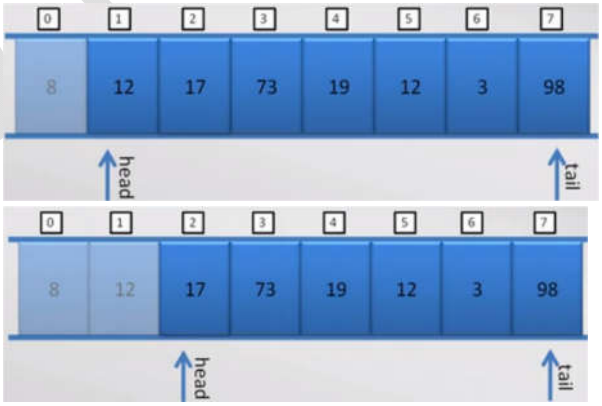

enqueue (12)


After a number of enqueues:



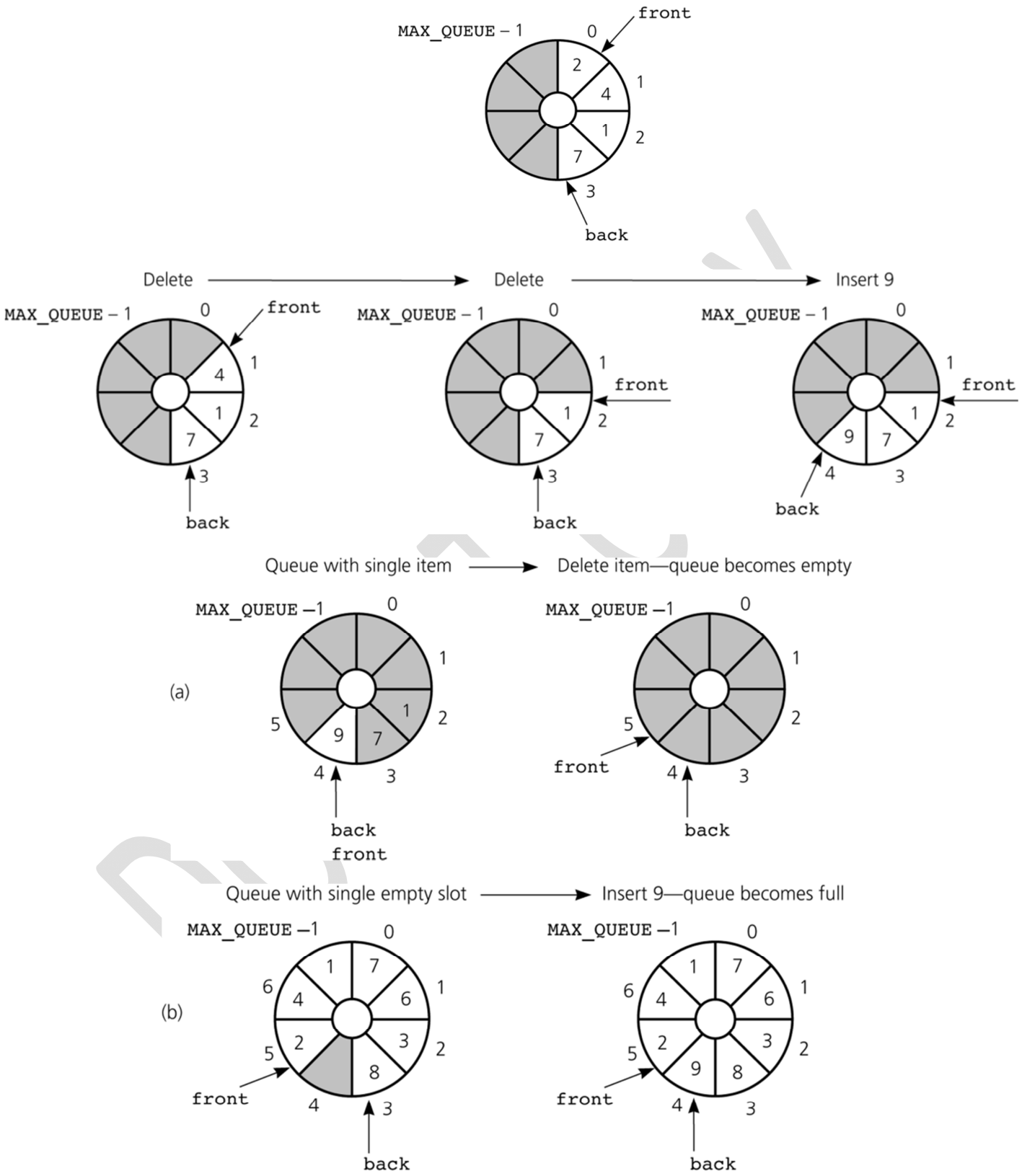

dequeue(): returns the item pointed by **head** and advances **head** pointer

dequeue()




enqueue (27) ?? How to advance **tail**?? We have space at the beginning?? Shift??

## Circular Queue



- **To detect circular queue-full and queue-empty conditions**
  - Keep a **count** of the queue items

- **To initialize the circular queue, set:**
  - front to -1
  - back to -1
  - count to 0

- **Inserting into a circular queue:**

```
If(count < MAX_QUEUE)  // free
        back = (++back) % MAX_QUEUE;
        items[back] = newItem;
        ++count;
        If(count == 1) // first item
                front = back;
```

- **Deleting from a circular queue:**

```
If(count > 0) // not empty
        front = (++front) % MAX_QUEUE;
        --count;
        If(count == 0)  // empty
                front = back = -1
```

**HW: Queue implementations using linked List and Arrays.**

## DE Queue (Double Ended Queue)

Allows add/remove elements from both head/tail.