Vietnam National University of HCMC

International University

School of Computer Science and Engineering

# Data Structures and Algorithms
# ★ Stack & Queue ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn

https://vichithanh.github.io

SCAN ME

# Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

**Mid-Term**

7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

**Final-Exam**

**10 LABS**

# What is a stack?

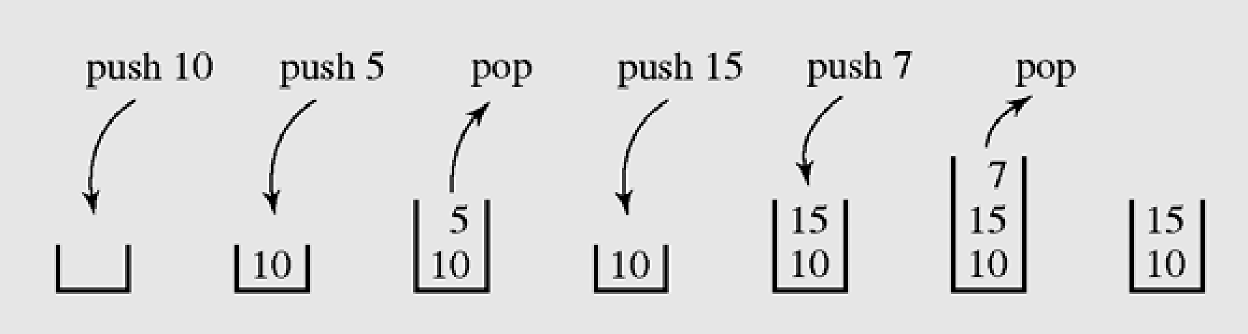
- A **stack** is a linear data structure that can be accessed only at one of its ends for storing and retrieving data
- A stack is a Last In, First Out (LIFO) data structure
- Anything added to the stack goes on the “top” of the stack
- Anything removed from the stack is taken from the “top” of the stack
- Things are removed in the reverse order from that in which they were inserted

# Stack - Introduction

- Accessible item ?
  - Last inserted item
  - Last in, first out (LIFO)
- Operations
  - Push ?
  - Pop ?
  - Peek / Top?
- Properties
  - Stack Overflow (Full)
  - Stack Underflow (Empty)

# Operations on a stack

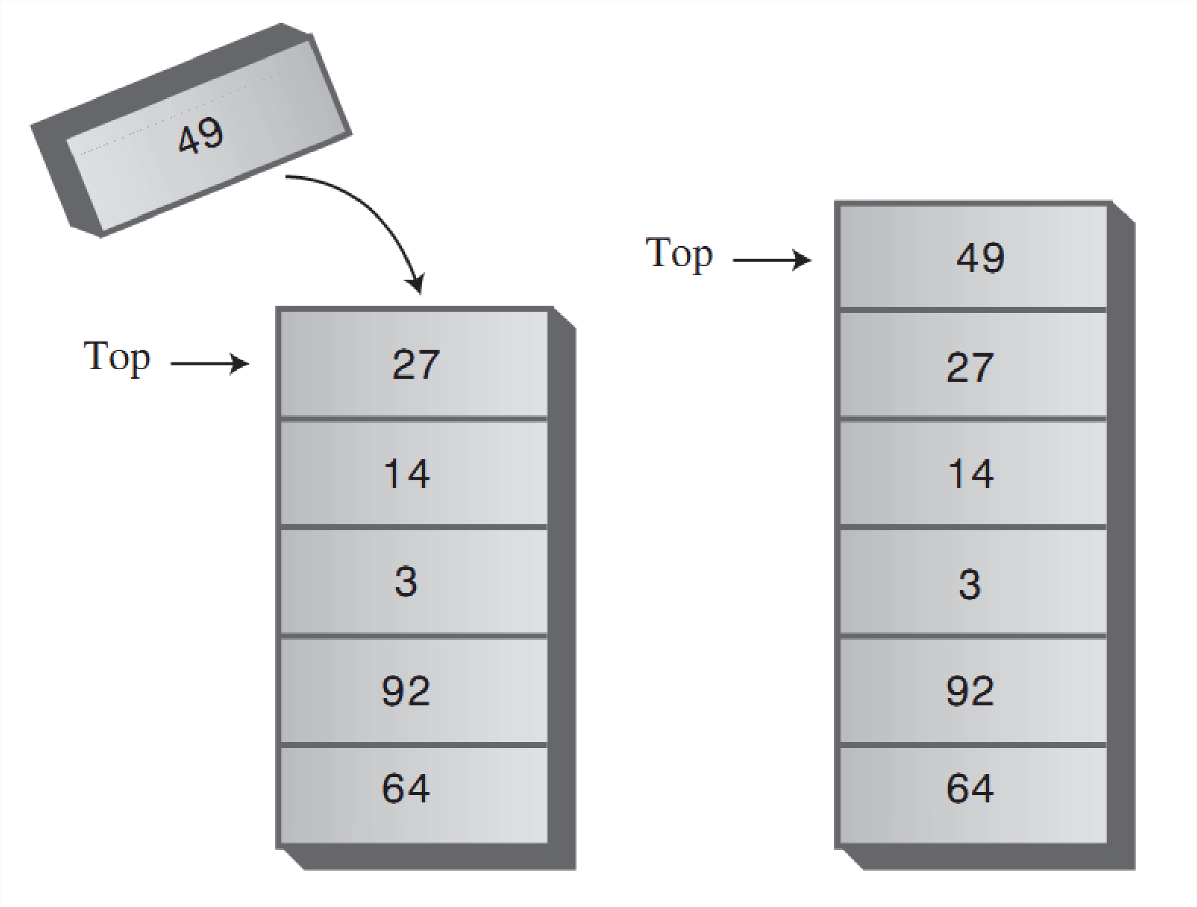
- The following operations are needed to properly manage a stack:
  - *clear()* – Clear the stack
  - *isEmpty()* – Check to see if the stack is empty
  - *push(el)* – Put the element *el* on the top of the stack
  - *pop()* – Take the topmost element from the stack
  - *top()* – Return the topmost element in the stack without removing it



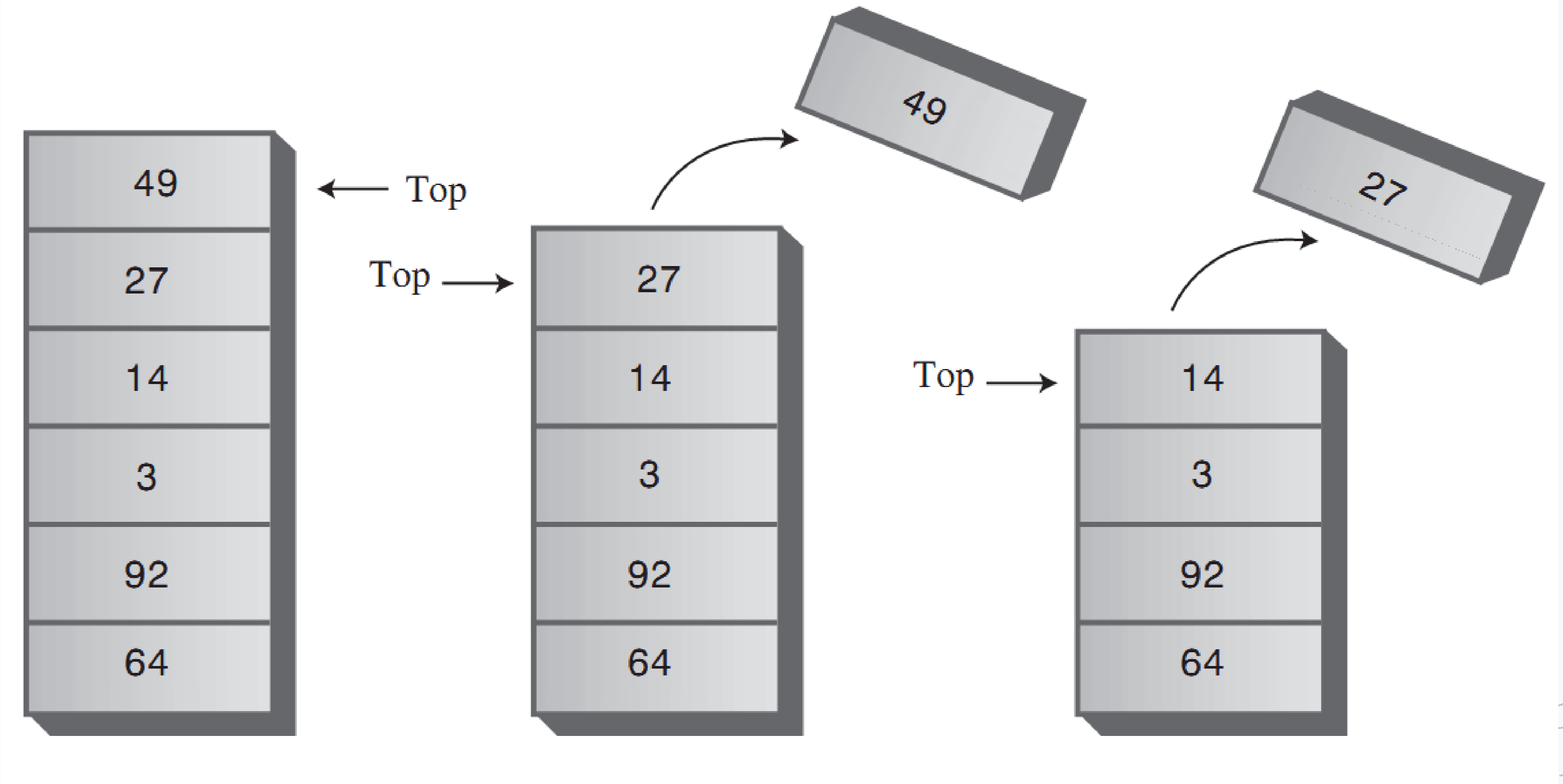
# Stack info

- Info must be managed?
  - Stack size (is full?)
  - Number of element (is empty?)
  - Top item (accessible item)

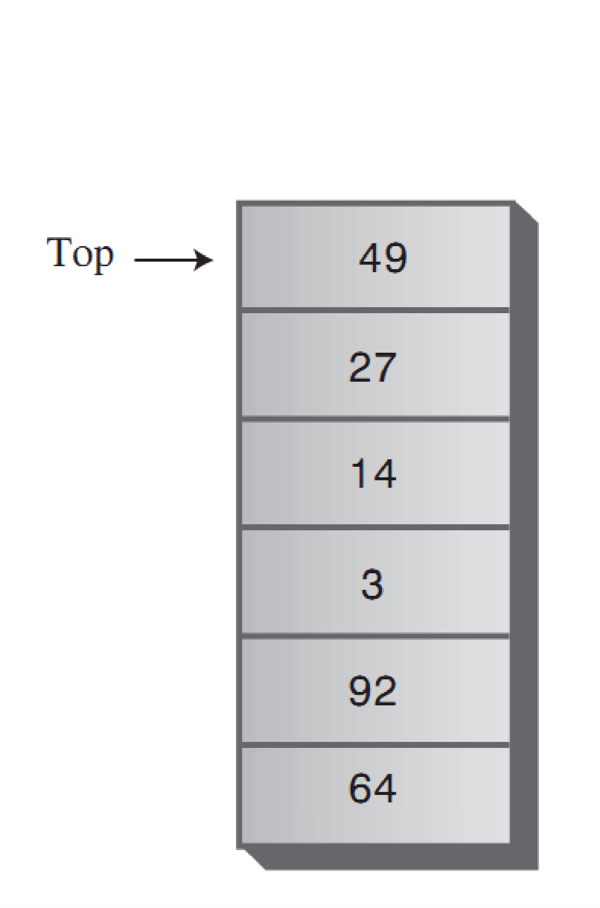
# Operations – Push (visual demo)



# Operations – Pop (visual demo)



# Operations – Peek (visual demo)



# Check Stack?

### Empty

- top = ?
  - -1

### Full

- top = ?
  - maxSize - 1

# Stack Exceptions

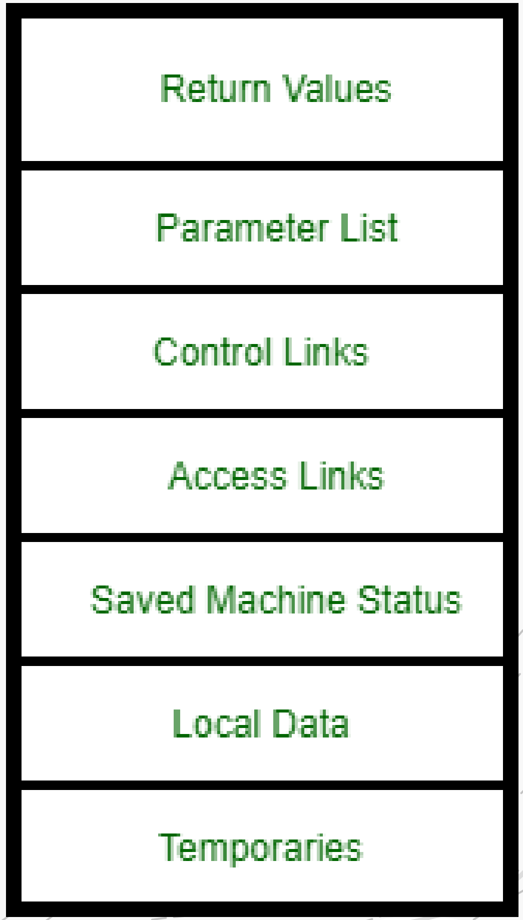
- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an **StackEmptyException**.

# Applications of Stacks

- Stacks are used for:
  - Any sort of nesting (such as parentheses)
  - Evaluating arithmetic expressions (and other sorts of expression)
  - Implementing function or method calls
  - Keeping track of previous choices (as in backtracking)
  - Keeping track of choices yet to be made (as in creating a maze)
  - Undo sequence in a text editor.
  - Auxiliary data structure for algorithms
  - Component of other data structures

# Stack in computer memory

- **How does a stack in memory actually work?**
- Each time a method is called, an **activation record (AR)** is allocated for it. This record usually contains the following information:
  - Parameters and local variables used in the called method.
  - A dynamic link, which is a pointer to the caller's activation record.
  - Return address to resume control by the caller, the address of the caller’s instruction immediately following the call.
  - Return value  for a method not declared as void. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.



Return Values

Parameter List

Control Links

Access Links

Saved Machine Status

Local Data

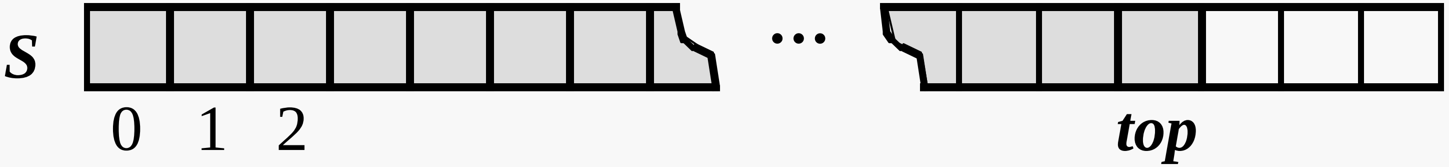
Temporaries

# Stack in computer memory

- **How does a stack in memory actually work?**
- Each new activation record is placed on the top of the run-time stack
- When a method terminates, its activation record is removed from the top of the run-time stack
- Thus, the first AR placed onto the stack is the last one removed

# Array-based Stack - 1

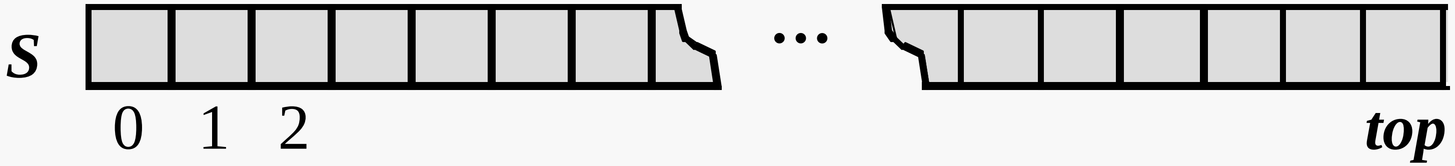
- A simple way of implementing the Stack ADT (abstract data type) uses an array
- We add elements from left to right
- A variable top keeps track of the index of the top element



Array-based stack

# Array-based Stack - 2

- The array storing the stack elements may become full
- A push operation will then throw a **FullStackException**
    - Limitation of the array-based  implementation
    - Not intrinsic to the Stack ADT



**Array-based stack may become full**

# Array implementation of a stack

```
class ArrayStack {
  protected  Object[] a; int top, max;
  public ArrayStack() {
     this(50);
  }
  public ArrayStack(int max1) {
     max = max1;
     a =  new Object[max];
     top = -1;
  }
  protected  boolean grow() {
     int max1 = max + max/2;
     Object[] a1 = new Object[max1];
     if(a1 == null) return(false);
     for(int i =0; i<=top; i++) {
         a1[i] = a[i];
     }
     a = a1;
     return(true);
  }
  public boolean isEmpty() {
     return(top==-1);
  }
```

```
public boolean isEmpty() {
  return(top==-1);
}
public boolean isFull() {
  return(top==max-1);
}
public void clear() {
  top=-1;
}
public void push(Object x) {
  if(isFull() && !grow()) return;
  a[++top] = x;
}
public Object top() throws EmptyStackException {
  if(isEmpty()) throw new EmptyStackException();
  return(a[top]);
}
public Object pop() throws EmptyStackException {
  if(isEmpty()) throw new EmptyStackException();
  Object x = a[top];
  top--;
  return(x);
}
```

# Stack - Application

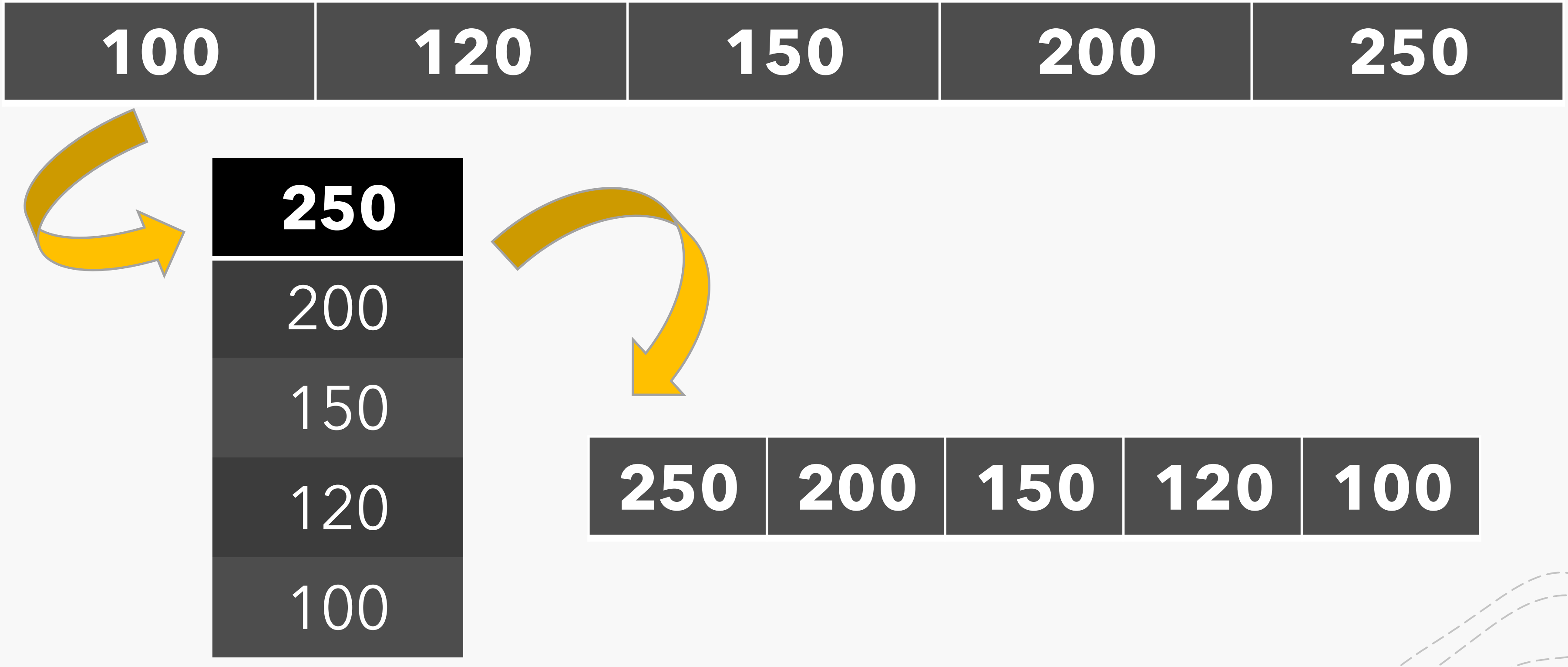
- Reversing an array/ a word

| | | | | |
|---|---|---|---|---|
| **100** | **120** | **150** | **200** | **250** |

- Delimiter Matching
  - 100 * (100 – 50) ➔ Correct
  - [100 * (100 – 50)] /2 ➔ Correct
  - [100 * (100 – 50)} /2 ➔ Incorrect, error on }
  - [100 * (100 – 50) /2 ➔ Incorrect, error on [
  - (100 * (100 – 50))) /2 ➔ Incorrect, error on )

# How would you do it?

- Reversing an array



| | | | | |
|---|---|---|---|---|
| 100 | 120 | 150 | 200 | 250 |

| |
|---|
| 250 |
| 200 |
| 150 |
| 120 |
| 100 |

| | | | | |
|---|---|---|---|---|
| 250 | 200 | 150 | 120 | 100 |

# Validate expression using stack - 1

- We consider arithmetic expressions that may contain various pairs of grouping symbols, such as
    - Parentheses: “(” and “)”
    - Braces: “{” and “}”
    - Brackets: “[” and “]”
- Each opening symbol must match its corresponding closing symbol. For example,
    - a left bracket, “[,” must match a corresponding right bracket, “],” as in the following expression
    - [(5+x)−(y+z)].
- The following examples further illustrate this concept:
    - Correct: ( )(( )){([( )])}
    - Correct: ((( )(( )){([( )])}))
    - Incorrect: )(( )){([( )])}
    - Incorrect: ({[ ])}
    - Incorrect: (

# Validate expression using stack - 2

- **An Algorithm for Matching Delimiters:**
- An important task when processing arithmetic expressions is to make sure their delimiting symbols match up correctly.
- We can use a stack to perform this task with a single left-to-right scan of the original string.
- Each time we encounter an opening symbol, we push that symbol onto the stack, and each time we encounter a closing symbol, we pop a symbol from the stack (assuming it is not empty) and check that these two symbols form a valid pair.
- If we reach the end of the expression and the stack is empty, then the original expression was properly matched.
- Otherwise, there must be an opening delimiter on the stack without a matching symbol. If the length of the original expression is n, the algorithm will make at most n calls to push and n calls to pop.

# How would you do it?

- Delimiter Matching

(a * [b – c]) / d

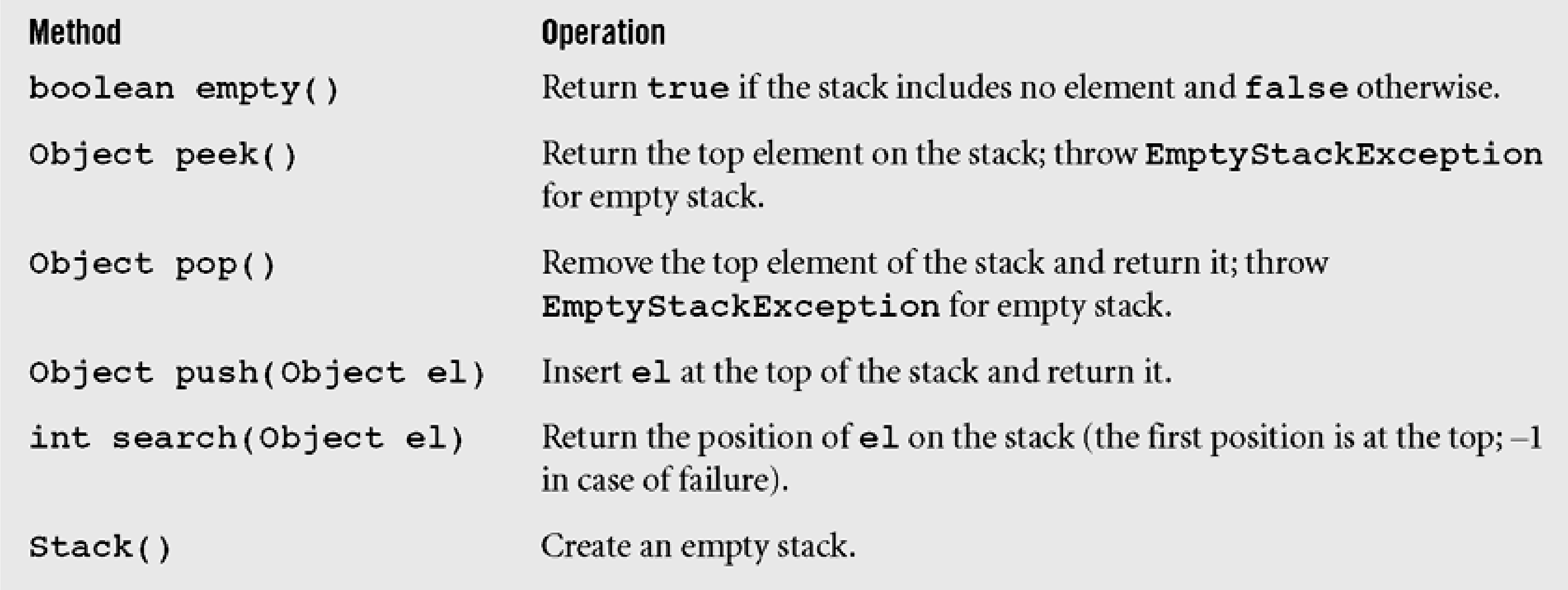
| Character Read | Stack contents |
|---|---|
| ( | ( |
| a | ( |
| * | ( |
| [ | ([ |
| b | ([ |
| - | ([ |
| c | ([ |
| ] | ( |
| ) | |
| / | |
| d | |

# Matching Parentheses and HTML Tags

- Another application of matching delimiters is in the validation of markup languages such as HTML or XML. HTML is the standard format for hyperlinked documents on the Internet and XML is an extensible markup language used for a variety of structured data sets.
- In an HTML document, portions of text are delimited by ***HTML tags***. A simple opening HTML tag has the form "**`<name>`**" and the corresponding closing tag has the form "**`</name>`**". For example, the **`<body>`** tag has the matching **`</body>`** tag at the close of that document.
- Ideally, an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags. We can use stack to check whether an HTML document is valid or not.

# Stack class in Java

- A Stack class implemented in the **java.util** package is an extension of class Vector to which one constructor and five methods are added.



| Method | Operation |
|---|---|
| `boolean empty()` | Return `true` if the stack includes no element and `false` otherwise. |
| `Object peek()` | Return the top element on the stack; throw `EmptyStackException` for empty stack. |
| `Object pop()` | Remove the top element of the stack and return it; throw `EmptyStackException` for empty stack. |
| `Object push(Object el)` | Insert `el` at the top of the stack and return it. |
| `int search(Object el)` | Return the position of `el` on the stack (the first position is at the top; –1 in case of failure). |
| `Stack()` | Create an empty stack. |

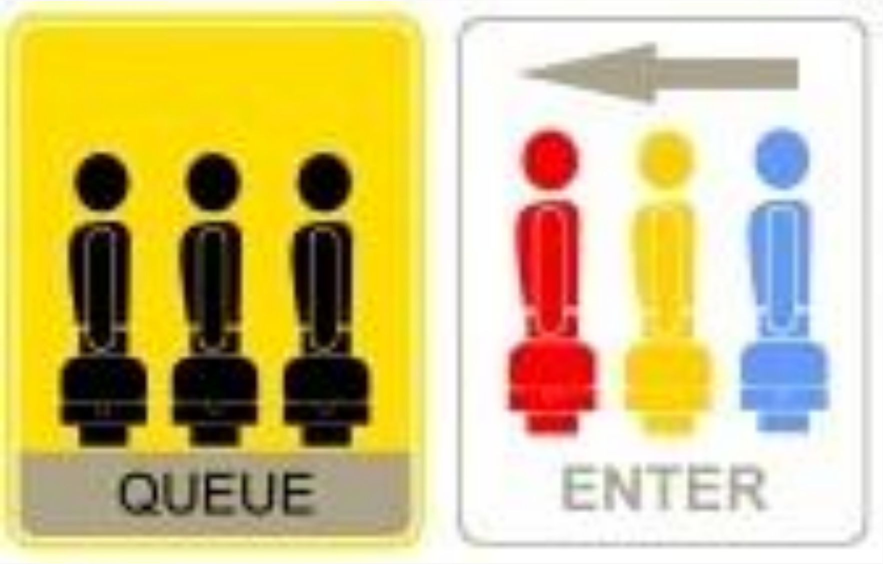
# Summary

- A stack is a linear data structure that can be accessed at only one of its ends for storing and retrieving data.
- A stack is called a **LIFO** structure: **Last in / First out.**

# QUEUE

# Queue - Introduction

- Accessible item ?
  - First inserted item
  - First in, first out (FIFO)
- Tail / Head of queue
- Operations
  - Insert / Enqueue
  - Remove / Dequeue
- Properties
  - Full
  - Empty



# What is a queue?

- A queue is a waiting line that grows by adding elements to its end and shrinks by taking elements from its front
- A queue is a structure in which both ends are used:
  - One for adding new elements
  - One for removing them
- A queue is a **FIFO** structure: **First In/ First Out**

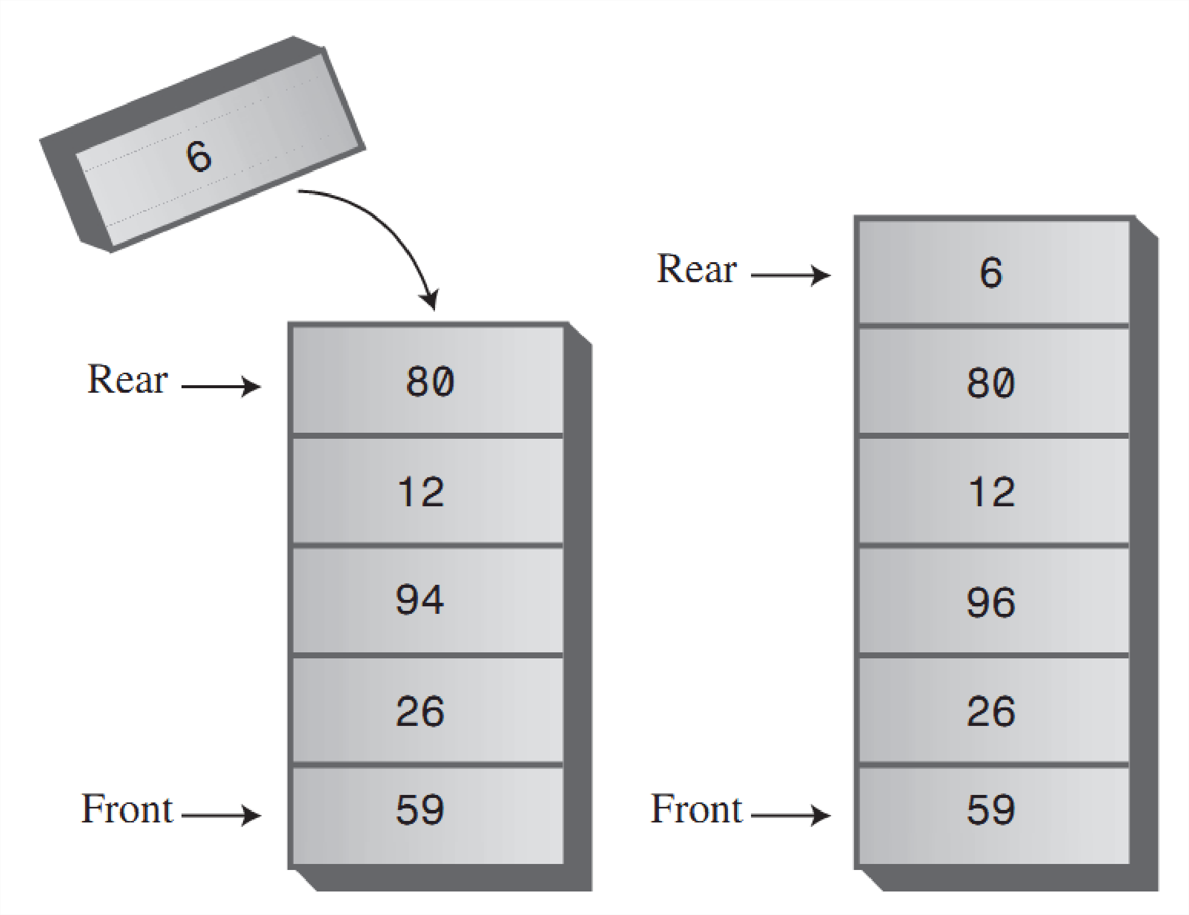
# Queue info

- Info must be managed?
  - Queue size (is full?)
  - Number of element (is empty?)
  - Head / tail item (accessible item)

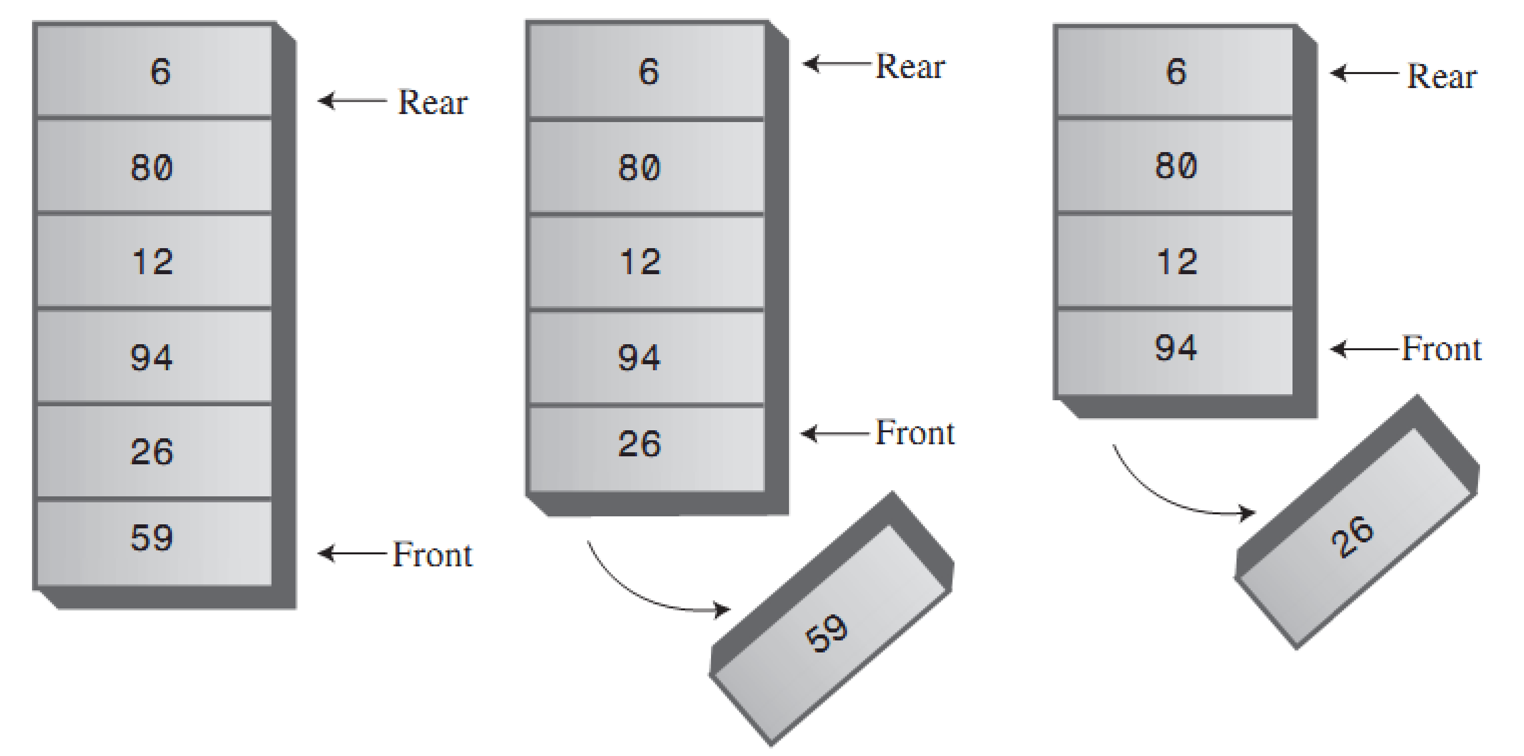
# Queue operations

- The following operations are needed to properly manage a queue:
  - *clear()* – Clear the queue
  - *isEmpty()* – Check to see if the queue is empty
  - *enqueue(el)* – Put the element *el* at the end of the queue
  - *dequeue()* – Take the first element from the queue
  - *front()* – Return the first element in the queue without removing it
    - sometimes the ***peek()*** is named instead of ***front()***
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

# Operations - Enqueue



# Operations - Dequeue



# Queue example

- enqueue(5)
- enqueue(3)
- dequeue()
- enqueue(6)
- dequeue()
- front()
- dequeue()
- dequeue()
- isEmpty()
- enqueue(9)
- enqueue(7)
- size()
- enqueue(4)
- dequeue()

# Queue example

| Operation | Output | Queue |
|---|---|---|
| enqueue(5) | - | (5) |
| enqueue(3) | - | (5, 3) |
| dequeue() | 5 | (3) |
| enqueue(6) | - | (3, 6) |
| dequeue() | 3 | (6) |
| front() | 6 | (6) |
| dequeue() | 6 | () |
| dequeue() | *“error”* | () |
| isEmpty() | *true* | () |
| enqueue(9) | - | (9) |
| enqueue(7) | - | (9, 7) |
| size() | 2 | (9, 7) |
| enqueue(4) | - | (9, 7, 4) |
| dequeue() | 9 | (7, 4) |

# Check queue

- dequeue/ peek/ front
  - Empty queue?
- enqueue
  - Full queue?
  - Size of queue?

# Efficiency of queue

- Enqueue?
- Dequeue?
- ➔ O(1)

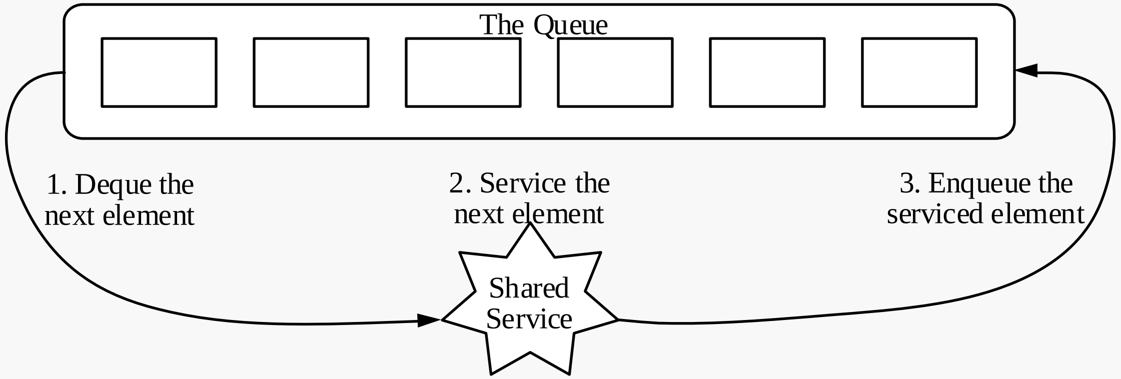
# Application

- Queue is used when things don’t have to be processed immediately, but must be processed in First In First Out order. This property of Queue makes it also useful in the following applications:
- Direct applications
  - Waiting lists
  - Access to shared resources (e.g., printer)
- Multiprogramming
  - Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

# Application: Round Robin Schedulers

We can implement a round robin scheduler using a queue, *Q*, by repeatedly performing the following steps:

1. *e* = *Q*.dequeue()
2. Service element *e*
3. *Q*.enqueue(*e*)


The Queue
1. Deque the next element
2. Service the next element
3. Enqueue the serviced element
Shared Service

**Round Robin Schedulers**

Round-robin (RR) is one of the simplest scheduling algorithms for processes in an operating system, which assigns time slices to each process in equal portions and in circular order, handling all processes without priority. Round-robin scheduling is both simple and easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks.

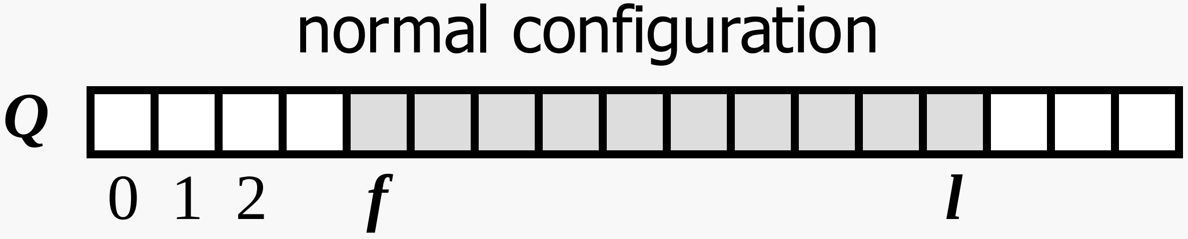
The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn

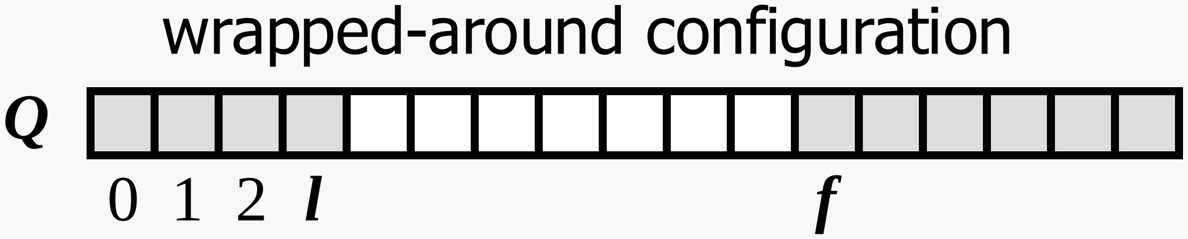
# Array-based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the first and last

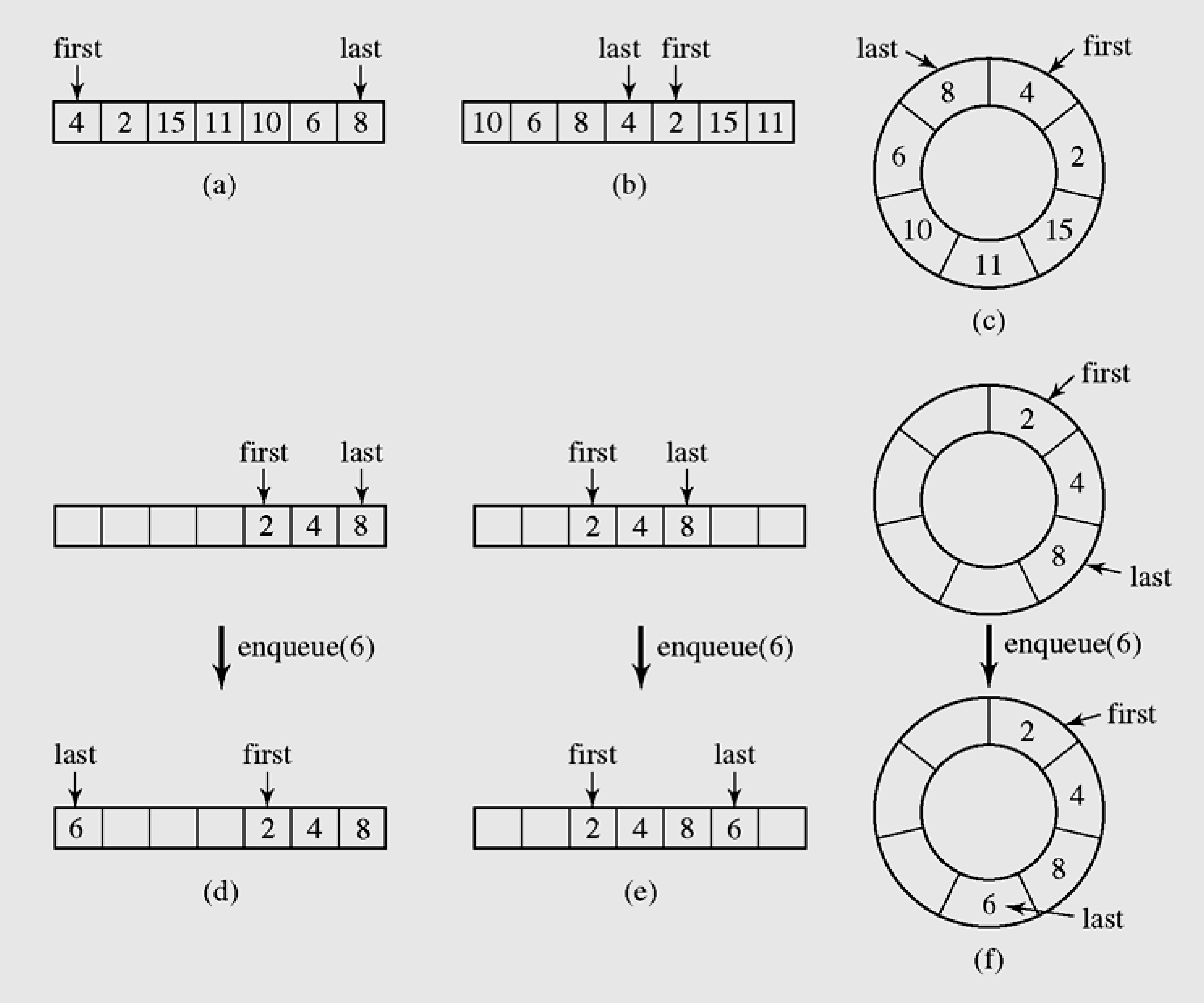
  $f$ index of the front element

  $l$ index of the last element





# Array-based Queue



# Array implementation of a queue

```
class ArrayQueue {
    protected Object [] a;
    protected int max;
    protected int first, last;

    public ArrayQueue() {
      this(10);
    }

    public ArrayQueue(int max1) {
      max = max1;
        a = new Object[max];
        first = last = -1;
    }

    public boolean isEmpty() {
      return(first == -1);
    }

    public boolean isFull() {
      return((first == 0 &&
        last == max-1) || first == last+1);
    }
```

```
private boolean grow() {
    int i,j;

    int max1 = max + max/2;

    Object [] a1 = new Object[max1];

    if(a1 == null) return(false);

    if(last >= first)
      for(i = first; i <= last; i++) a1[i - first] = a[i];
    else {
      for(i = first; i < max; i++) a1[i - first] = a[i];
      i = max - first;
      for(j = 0; j <= last; j++) a1[i + j]= a[j];
    }
    a = a1;
    first = 0;
    last = max - 1;
    max = max1;
    return(true);
}
```

# Array implementation of a queue

```
void enqueue(Object x) {
    if(isFull() && !grow()) return;

    if(last == max-1 || last == -1) {
       a[0] = x; last = 0;
       if(first == -1) first = 0;
    }
    else   a[++last] = x;
}

Object front() throws Exception {
    if(isEmpty()) throw new Exception();

    return(a[first]);
}
```

```
public Object dequeue() throws Exception {
      if(isEmpty()) throw new Exception();

      Object x = a[first];

      if(first == last) { // only one element
         first = last = -1;}
      else
        if(first == max-1)
            first = 0;
         else
            first++;

      return(x);
  }
```

# Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in Java class

```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public Object front() throws EmptyQueueException;
  public void enqueue(Object o);
  public Object dequeue()throws EmptyQueueException;
}
```

# Double-Ended Queues (Deque)

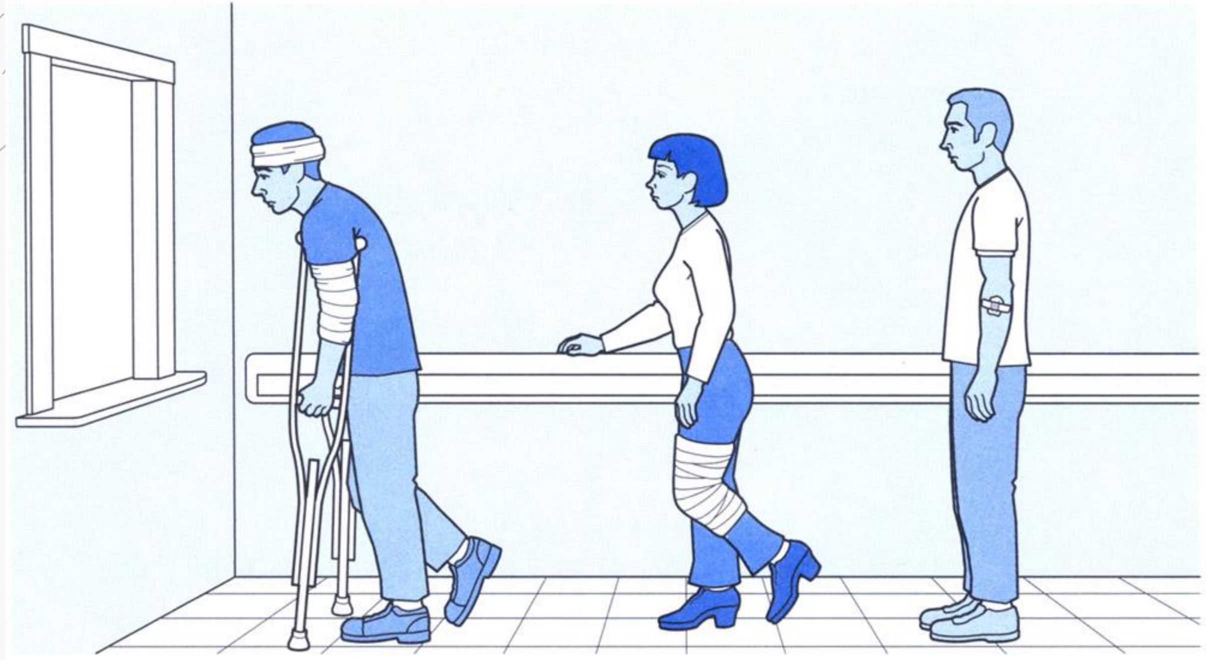
- A queue-like data structure that supports insertion and deletion at both the front and the back of the queue.
- Such a structure is called a **double-ended queue**, or **deque**, which is usually pronounced “***deck***” to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation “D.Q.”
- The deque abstract data type is more general than both the stack and the queue ADTs. The extra generality can be useful in some applications.
- For example, we described a restaurant using a queue to maintain a waitlist. Occasionally, the first person might be removed from the queue only to find that a table was not available; typically, the restaurant will reinsert the person at the first position in the queue. It may also be that a customer at the end of the queue may grow impatient and leave the restaurant.

# Double-Ended Queues (Deque)

- The deque abstract data type is richer than both the stack and the queue ADTs. To provide a symmetrical abstraction, the deque ADT is defined to support the following update methods:
  - **addFirst(e)**: Insert a new element e at the front of the deque.
  - **addLast(e)**: Insert a new element e at the back of the deque.
  - **removeFirst()**: Remove and return the first element of the deque (or null if the deque is empty).
  - **removeLast()**: Remove and return the last element of the deque (or null if the deque is empty).
- Additionally, the deque ADT will include the following accessors:
  - **first()**: Returns the first element of the deque, without removing it (or null if the deque is empty).
  - **last()**: Returns the last element of the deque, without removing it (or null if the deque is empty).
  - **size()**: Returns the number of elements in the deque.
  - **isEmpty()**: Returns a boolean indicating whether the deque is empty.

# Priority queue

# Priority queue

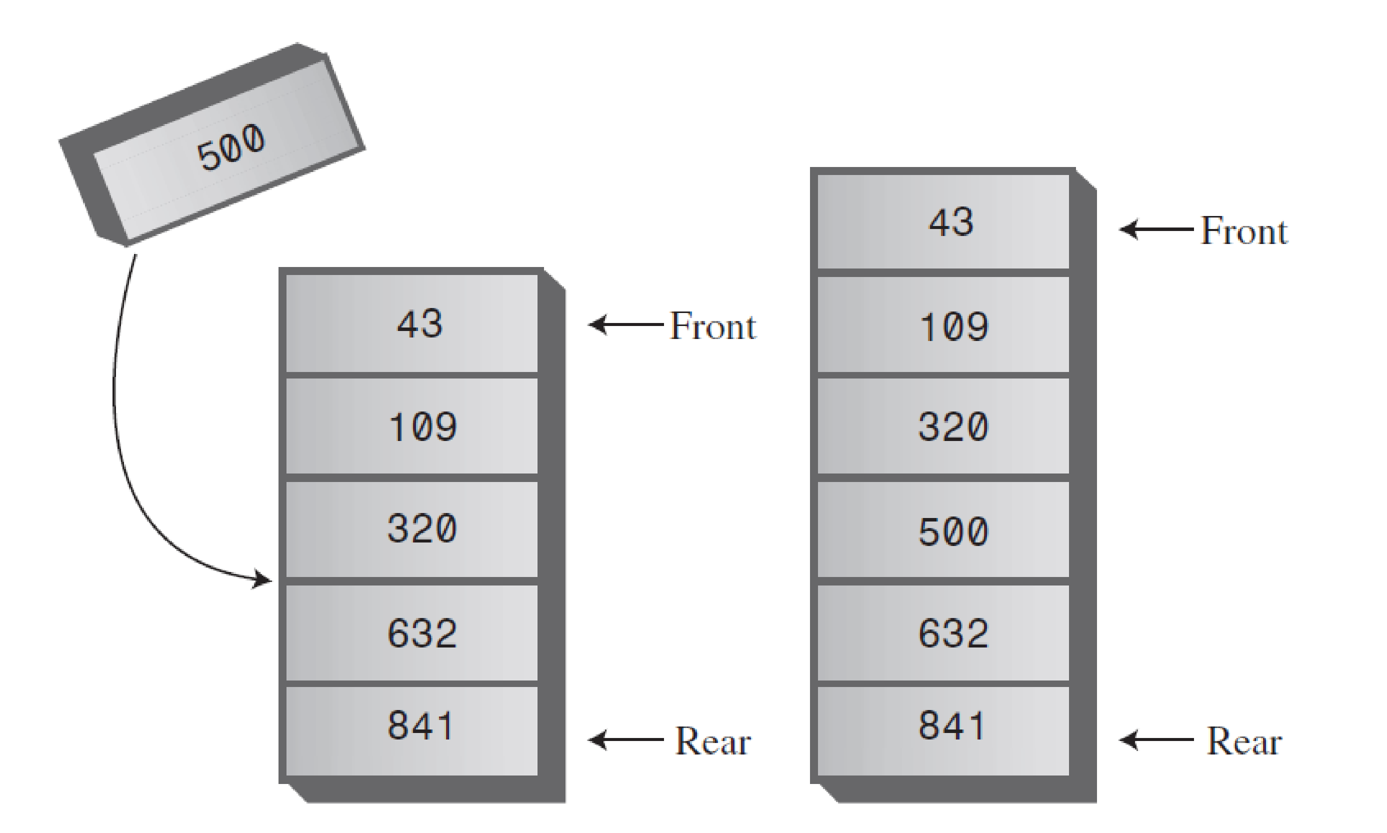


- Head / tail
- Enqueue / Dequeue with criteria
- E.g, dequeue
  - Highest value, or
  - Most severe patient, …
- Ascending-priority / descending-priority queue

# Operation

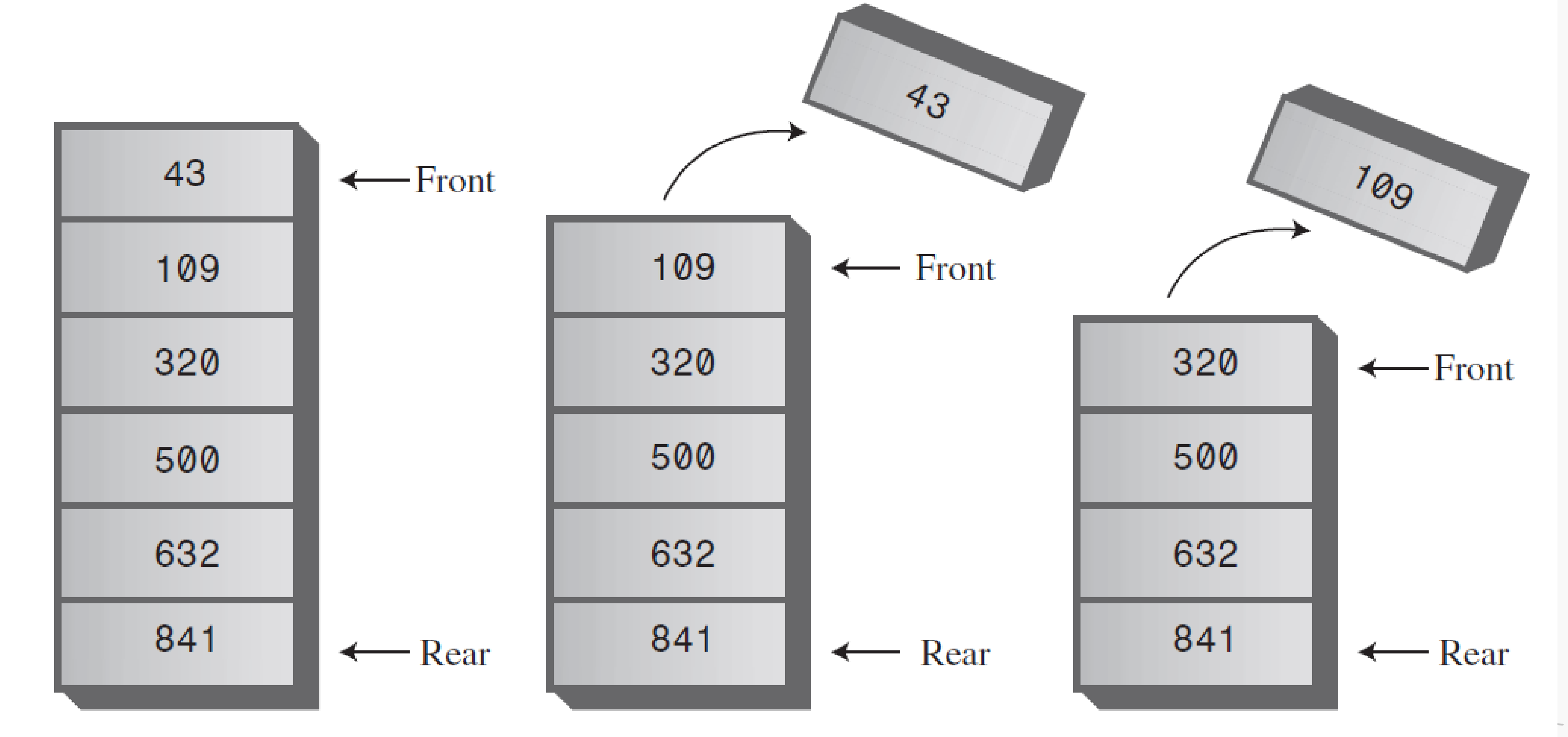
- Enqueue
- Dequeue
- Peak

# Priority queue - Enqueue



New item inserted in priority queue

# Priority queue - Dequeue



Two items removed from front of priority queue

# Priority Queue

- A **priority queue** can be assigned to enable a particular process, or event, to be executed out of sequence without affecting overall system operation
- FIFO rule is broken in these situations
- In priority queues, elements are dequeued according to their priority and their current queue position

# Efficiency of Priority queue

- Insertion ?
  - If use ARRAY: O(N)

- Deletion ?
  - O(1)

# Array implementation of a priority queue

```
class PriorityQueue {
    protected  float [] a; int top, max;

    public PriorityQueue() {
        this(50);
    }

    public PriorityQueue(int max1) {
        max = max1;
        a =  new float[max];
        top = -1;
    }

    protected  boolean grow() {
        int max1 = max + max/2;
        float [] a1 = new float[max1];
        if(a1 == null) return(false);
        for(int i =0; i<=top; i++)
             a1[i] = a[i];
         a = a1;
        return(true);
    }
```

```
    public boolean isEmpty()
      { return(top == -1);}

    public boolean isFull()
      { return(top == max-1);}

    public void clear()
      { top = -1;}

    public void enqueue(float x)
      { if(isFull() && !grow()) return;
        if(top == -1)
           { a[0] = x; top = 0;
             return;
           }
        int i = top;
        while(i >= 0 && x < a[i])
           {  a[i+1] = a[i];
              i--;
           }
        a[i+1] = x;  top++;
      }
```

# Array implementation of a priority queue

```
public float front() {
  assert(!isEmpty());
  return(a[top]);
}

public float dequeue() {
  assert(!isEmpty());
  float x = a[top];
  top--;
  return(x);
}
```

# Question

**One difference between a priority queue and an ordered array is that**

a. the lowest-priority item cannot be extracted easily from the array as it can from the priority queue.

b. the array must be ordered while the priority queue need not be.

c. the highest priority item can be extracted easily from the priority queue but not from the array.

d. All of the above.

# Questions

**Which of the following best describes the Last-In-First-Out (LIFO) principle in a stack?**

- a) The first element added to the stack is the first element removed.
- b) The last element added to the stack is the first element removed.
- c) Elements in the stack are removed in random order.
- d) Elements in the stack are removed in ascending order.

# Questions

**Suppose you have a stack containing the elements [10, 20, 30, 40, 50], and you perform the following operations in sequence: push(60), pop(), push(70), push(80), and pop(). What will be the content of the stack after these operations?**

- a) [10, 20, 30, 40, 50]
- b) [10, 20, 30, 40, 50, 60, 70]
- c) [10, 20, 30, 40, 70]
- d) [10, 20, 30, 40, 50, 70]

# Questions

**In a queue, which operation adds an element to the back (end) of the queue?**

- a) Dequeue
- b) Front
- c) Enqueue
- d) Rear

# Questions

**Suppose you have a queue containing the elements [10, 20, 30, 40, 50], and you perform the following operations in sequence: enqueue(60), dequeue(), enqueue(70), enqueue(80), and dequeue(). What will be the content of the queue after these operations?**

a) [10, 20, 30, 40, 50]

b) [20, 30, 40, 50, 60, 70, 80]

c) [30, 40, 50, 60, 70, 80]

d) [50, 60, 70, 80]

# Summary

- A queue is a waiting line that grows by adding elements to its end and shrinks by taking elements from its front.
- A queue is a FIFO structure: First in / First out.
- In queuing theory, various scenarios are analysed, and models are built that use queues for processing requests or other information in a predetermined sequence (order).
- A priority queue can be assigned to enable a particular process, or event, to be executed out of sequence without affecting overall system operation.
- In priority queues, elements are dequeued according to their priority and their current queue position.

Vietnam National University of HCMC

International University

School of Computer Science and Engineering

# THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn

https://vichithanh.github.io

SCAN ME