香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 7: Stack and queue

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Stack

- A stack stores a set S of elements that have two constrained updates:
  - Push(e): add a new element to S
  - Pop(): removes the most recently added element from S
- Stack follows:
  - The Last-In-First-Out (LIFO) property
  - We can only add/remove/examine from one end (called the "top").
  - Consider the trays/dishes in canteens

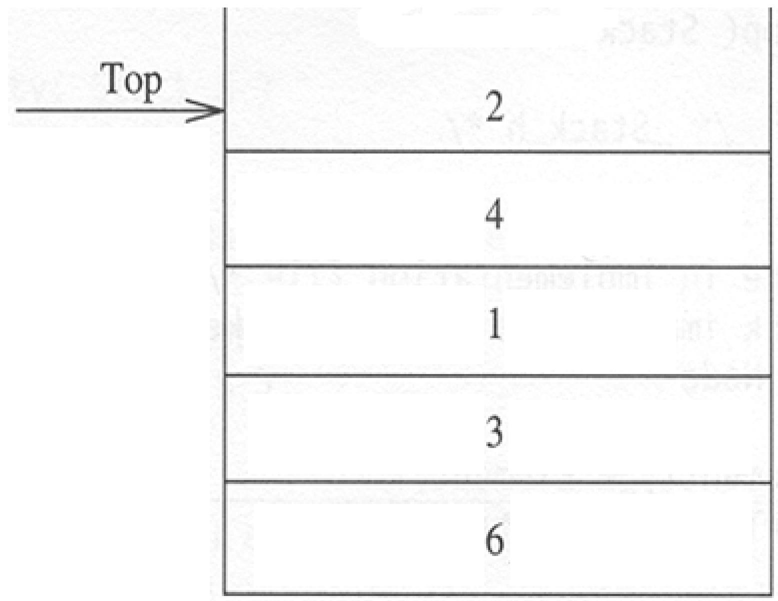
Stack

- Access from the top

- Basic operations
  - pop ()
  - push (i)

  - makeEmpty ()
  - top ()
  - isEmpty()

- Implementation
  - Linked list
  - Arrays



Implementation of Stack using Linked List

```
class Node {
        Node next;
        Object element;
}

class Stack {
        Node next;
}
```

Implementation of Stack using Linked List

- Create an empty stack

```
public Stack() {                //constructor
      this.next = null;
}
```

- Push onto a stack

```
void push(Object x) {
            Node tmpNode = new Node();
            tmpNode.element = x;
            tmpNode.next = this.next;
            this.next = tmpNode;
}
```

Implementation of Stack using Linked List

- Return top element in a stack

```
public Object top( ) {
        if (!isEmpty())
                        return next.element;
        else {
                        return null;
        }
}
```

- Pop from a stack

```
public Object pop() {
   Node firstNode = null;
   if (isEmpty()) {
              return null;
   } else {
              firstNode = this.next;
              this.next = firstNode.next;
              return firstNode.element;
   }
}
```

Implementation of Stack using Array

```
class Stack {
        final static int MIN_STACK_SIZE = 5;
        int topOfStack = -1;
        Object[ ] array;
}
```

Implementation of Stack using Array

- Stack creation

```
public Stack (int maxElements) {
    int capacity = maxElements;

    if (maxElements < MIN_STACK_SIZE)
        capacity = MIN_STACK_SIZE;

    array = new Object[capacity];
}
```

Implementation of Stack using Array

- Test for full stack

```
public boolean isFull() {
        return (topOfStack == array.length - 1);
}
```

- Push an element onto the stack

```
public boolean push(Object x) {
        if (isFull())
                return false;
        else
                array[++topOfStack] = x;
        return true;
}
```

Implementation of Stack using Array

- Return top of stack

```
public Object top() {
        if (!isEmpty())
                return array[topOfStack];
        else
                return null;
}
```

- Pop element from stack

```
public Object pop() {
        if (!isEmpty())
                return array[topOfStack--];
        else
                return null;
}
```

Comparison of these two implementation

- Using list saves space

- Using array is faster. Why?
  - Two reasons:
    - Memory allocation
    - Continuous memory can be loaded into cache

Stack applications

Balanced Symbol Checking

- In programing languages, there are many instances when symbols must be balanced
  - E.g., { } , [ ] , ( )

- Stack can be used for checking if the symbols are balanced
  - Balanced
    - (){[]}
    - ({{}})
    - ({[]})
  - Unbalanced
    - (]
    - (){([])}]
    - ()[[]{}

C code example

```c
int sum = 0;
for(int i=0; i<n; i++){
    sum += array[i];
}
return sum;
```

Balanced Symbol Checking

- Observation
  - If the next symbol is the opening symbol, e.g., (, [, {.
    - It will not result in unbalanced symbols
  - If the next symbol is the closing symbol, e.g.,),], }.
    - It must match the last symbol
    - E.g., if the next symbol is), then the last symbol must be (

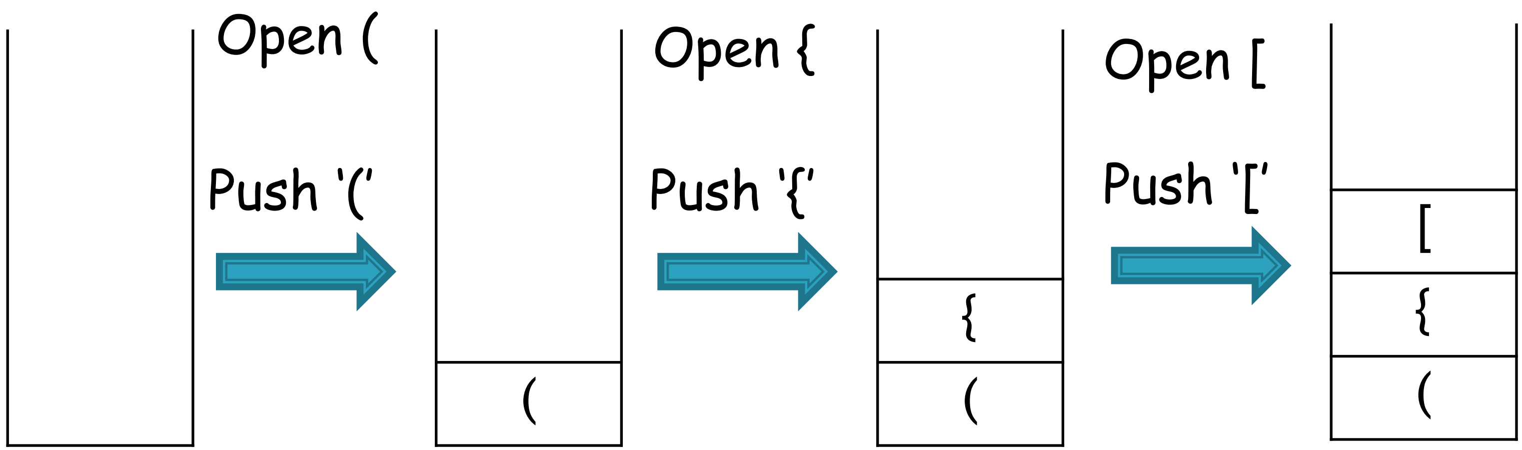
Balanced Symbol Checking Algorithm

- Step 1: make an empty stack
- Step 2: read the symbols from the input text
  - If the symbol is an opening symbol, push it onto the stack
  - If it is a closing symbol
    - If the stack is empty: return false
    - Otherwise, pop from the stack. If the symbol popped does not match the closing symbol, return false
- Step 3: at the end, if the stack is not empty, return false (unbalanced), else return true (balanced)

A Running Example

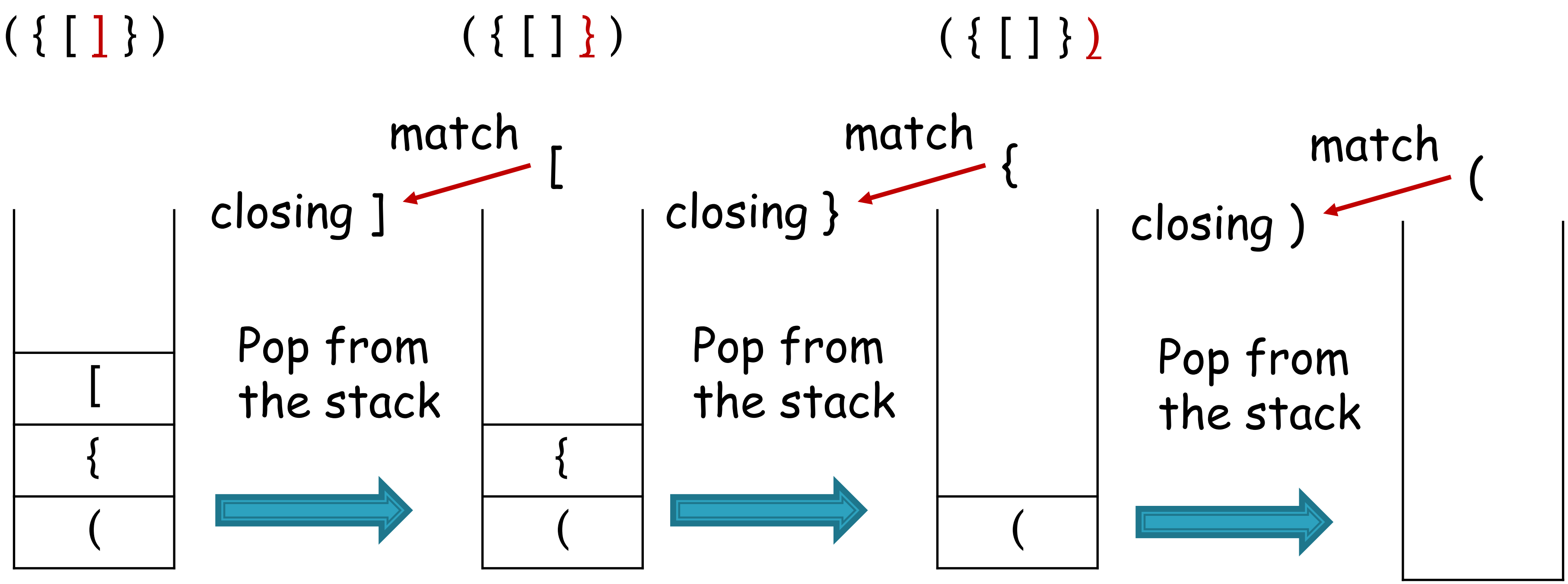
- Given an input symbol list: ({ [ ] }),
  - check if the symbols are balanced: show the status of the stack after each symbol checking

({ [ ] })　　　　({ [ ] })　　　　({ [ ] })



A Running Example (cont.)

- Given an input symbol list: ({[]}),
  - check if the symbols are balanced: Show the status of the stack after each symbol checking



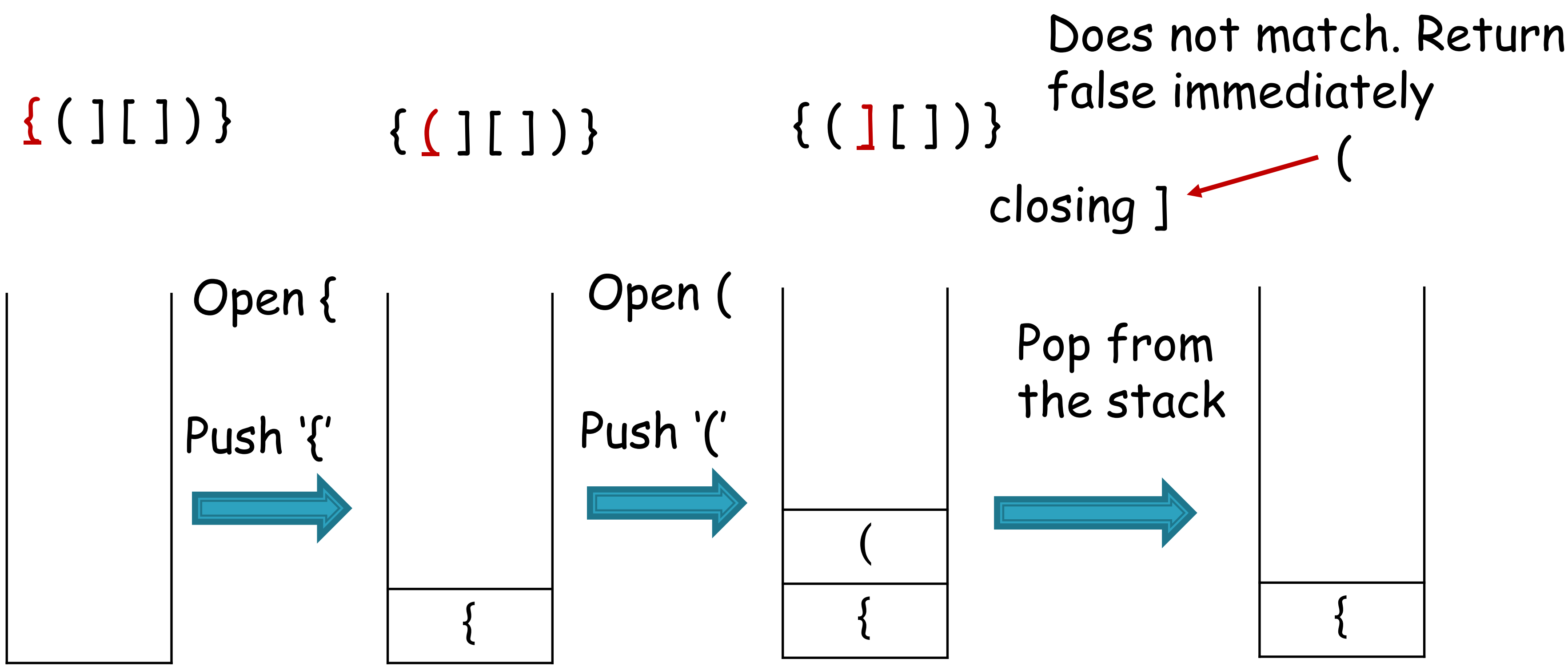
- After checking all symbols, the stack is empty: return true

Practice

- Given an input symbol list: { ( ] [ ]) },
  - check if the symbols are balanced
  - show the status of the stack after each symbol checking
- Check if the symbol list ( ) [[ ] { } is balanced
  - Show the status of the stack after each symbol checking

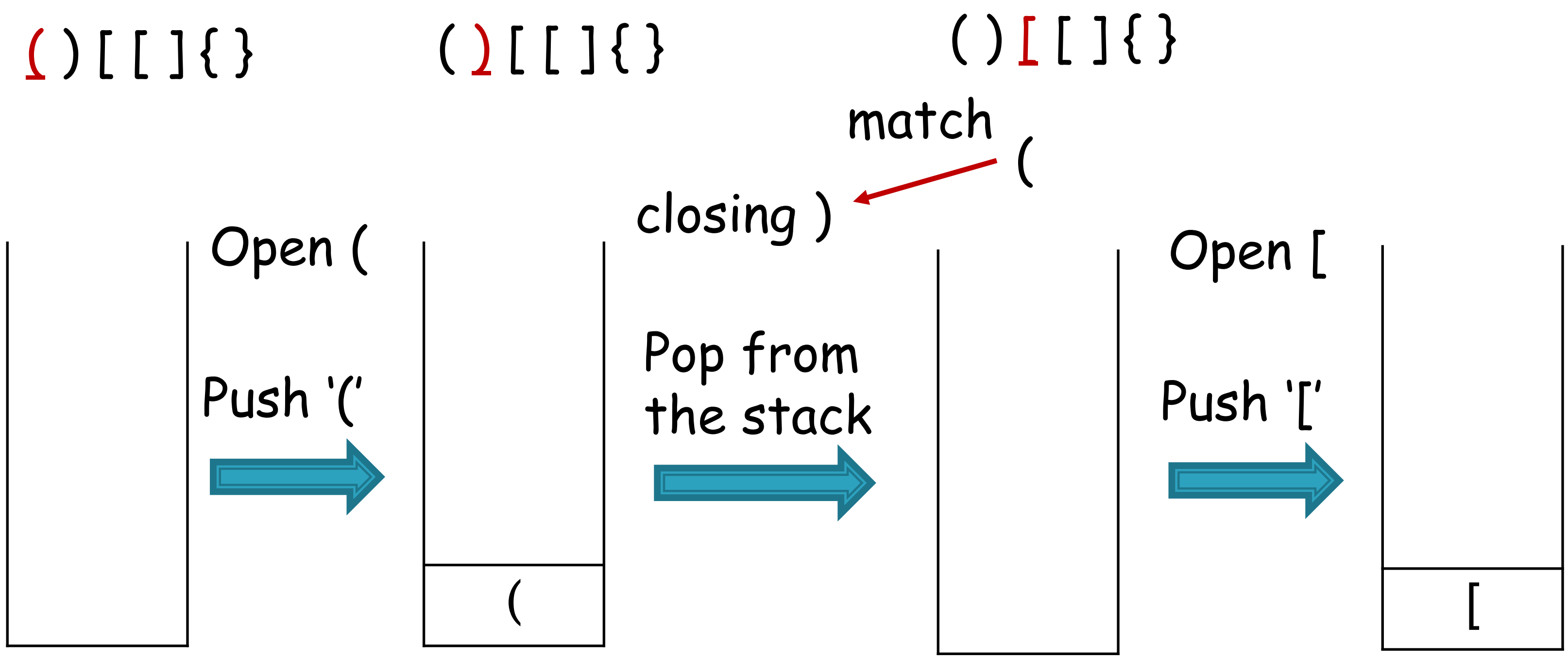
Practice

- Check if the symbol list { ( ] [ ]) } is balanced
  - Show the status of the stack after each symbol checking



Practice (Cont.)

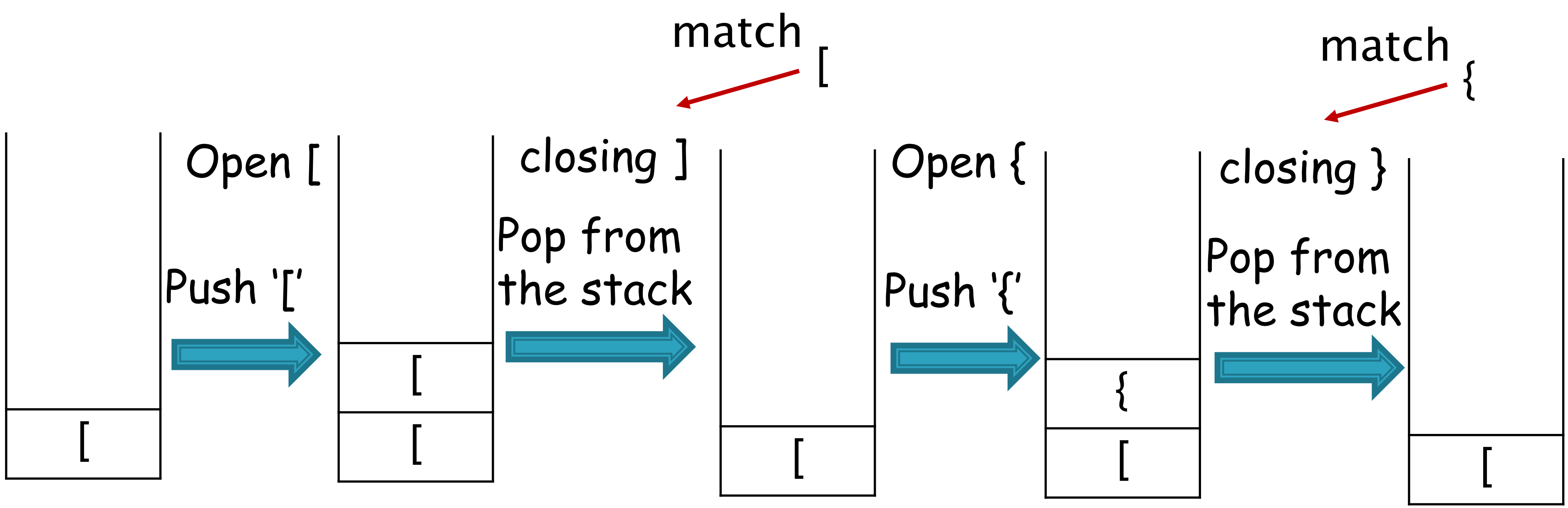
- Check if the symbol list ()[[]{} is balanced
  - Show the status of the stack after each symbol checking



Practice (Cont.)

- Check if the symbol list ( ) [[ ] { } is balanced
  - Show the status of the stack after each symbol checking

( ) [[ ] { }   ( ) [[ ] { }   ( ) [[ ] { }   ( ) [[ ] { }



- After examining all symbols, the stack is not empty, so return false

Application of Stack (ii)

- Evaluation of expressions
  - The representation and evaluation of expressions are of great interest to computer scientists
  - How we usually write expressions
    - *a/b - c+d*e - a*c*
  - If we examine the above expression, notice that they contain:
    - operators: +, -, *, /
    - operands: *a*, *b*, *c*, *d*, *e*
  - How the expressions are interpreted?
    - Precedence rule + associative rule

Expressions: What We Learnt

- Precedence of operators: The order in which the operators are performed:
  - Precedence:
    - * and / have the same precedence, + and – have the same precedence
    - * and / have higher prededence than + and -
  - (((a / b) - c) + (d * e)) - (a * c)
- Associative rule of operators:
  - +, -, * and / are left-associative (from left to right)
- Parentheses can be used to override precedence:
  - Expressions are always evaluated from the innermost parenthesized expression, e.g., a * (b + c)

Representations of Expressions

- Consider the four binary operators +, -, * and /
- The standard way (when we write expressions): **Infix Expressions**
  - A binary operator is placed in-between its two operands
  - Con: need to use parentheses and precedence rules to evaluate expressions
- When a program executes an expression: **Postfix Expressions**
  - Each operator appears after its operands
  - Pro: precedence has been considered when the postfix expression is generated. No parentheses

We leave a space here to distinguish two operands 2 and 3 and one operand 23

Infix	Postfix
2 + 3 * 4	2 3 4 * +
2 * 3 + 4	2 3 * 4 +
2 * 3 * 4	2 3 * 4 *
(2 + 3) * 4	2 3 + 4 *
a / b - c + d * e - a * c	a b / c - d e * + a c * -

How to Derive the Postfix?

- 7/(2+3)*4
  - According to the definition, operator should appear after operand 7/(2+3) and 4 should be put before *, so the postfix L for the expression is:
    - L: "Postfix for 7/(2+3)" 4 *
  - We got a smaller problem. What is the postfix L' for 7/(2+3)?
    - 7 and postfix of (2+3) should appear before /
    - L': 7 "postfix for (2+3)" /
  - What is the postfix L" for (2+3)?
    - 2 3 +
  - => Postfix for L' is: 7 2 3 + /
  - => Postfix for L is: 7 2 3 + / 4 *

Practice

- What is the postfix expression for the following expressions?
  - 2*(3+2*4)
  - Hint: according to the definition, operator should appear after operand. 2 and (3+2*4) are the operand of *, so they should appear before *

Answer: The operand of * is 2 and (3+2*4), let's first denote the postfix expression of (3+2*4) as x. Then, the postfix expression will be 2 x *. Now consider 3+2*4. The operand of + is 3 and 2*4.

Let's denote the postfix of 2*4 as y, and the postfix expression of 3+2*4 becomes 3 y +. Now consider the postfix expression of 2*4, we know it is 2 4 * according to its definition. So y=2 4 *. As x = 3 y +, putting y to the equation, we have x = 3 2 4 * +.

Putting back x to the postfix expression, we have the postfix expression of 2*(3+2*4) is: 2 3 2 4 * + *

Evaluating Postfix Expression

- We can use the previous recursive idea to derive the postfix expression

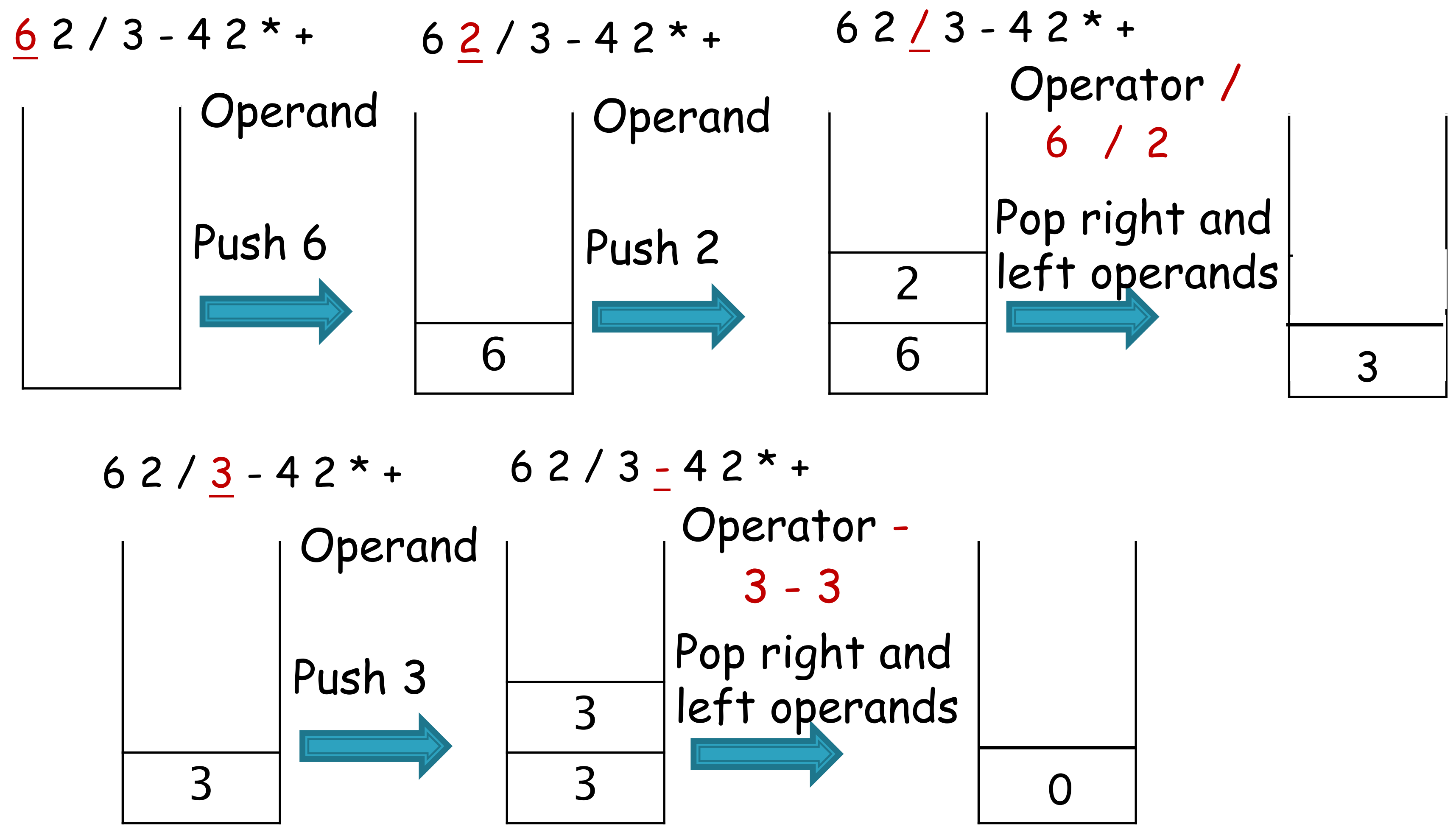
- Given a postfix expression
  - How to evaluate the postfix expression?

Postfix Evaluation Algorithm

- Here, we only consider evaluation of expressions using the four binary operators +, -, * and /
  - Create a stack
  - Scan the postfix expression from left-to-right
    - If an operand is encountered, push to the stack
    - If an operator is encountered
      - pop the stack for the right hand operand
      - pop the stack for the left hand operand
      - apply the operator to the two operands
      - push the result onto the stack
  - When the postfix expression has been scanned, the result is kept on the top of the stack.

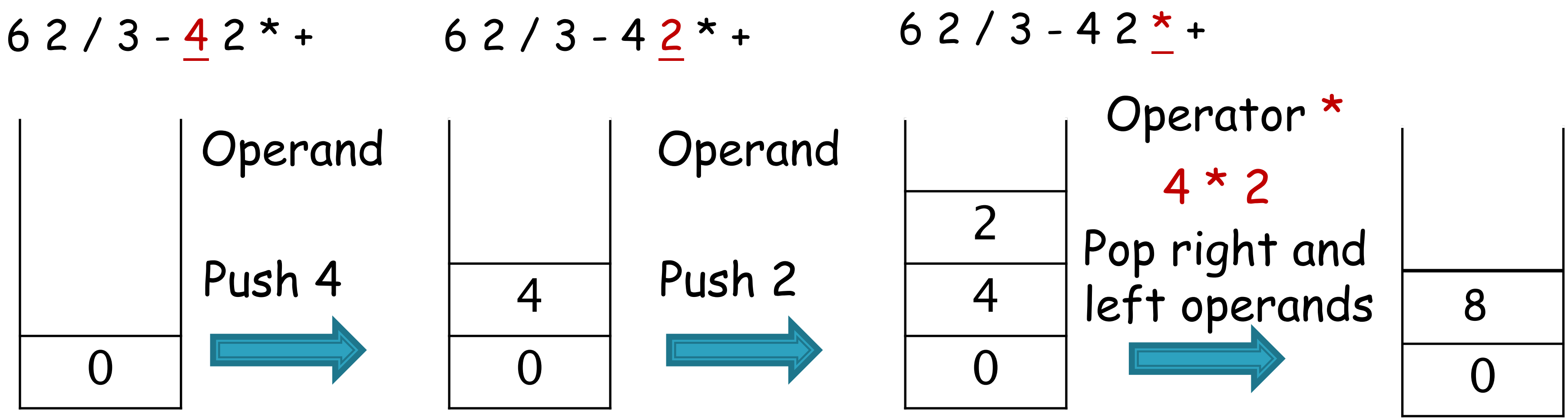
A Running Example

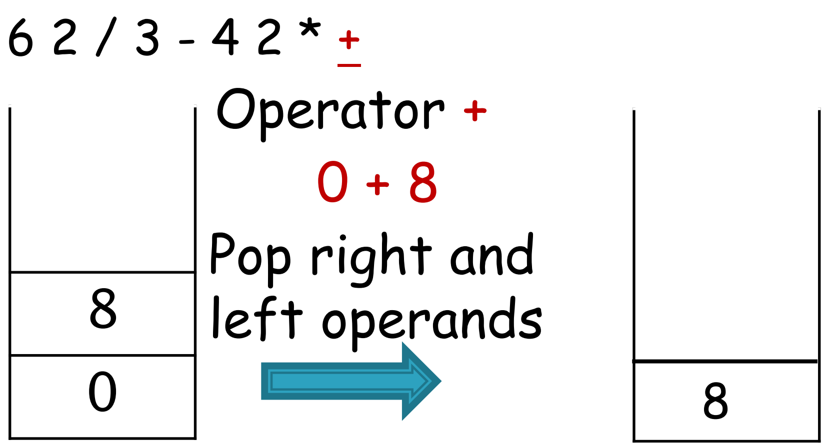
- Evaluate the postfix expression: 6 2 / 3 - 4 2 * +



A Running Example (cont.)

- Evaluate the postfix expression: 6 2 / 3 - 4 2 * +

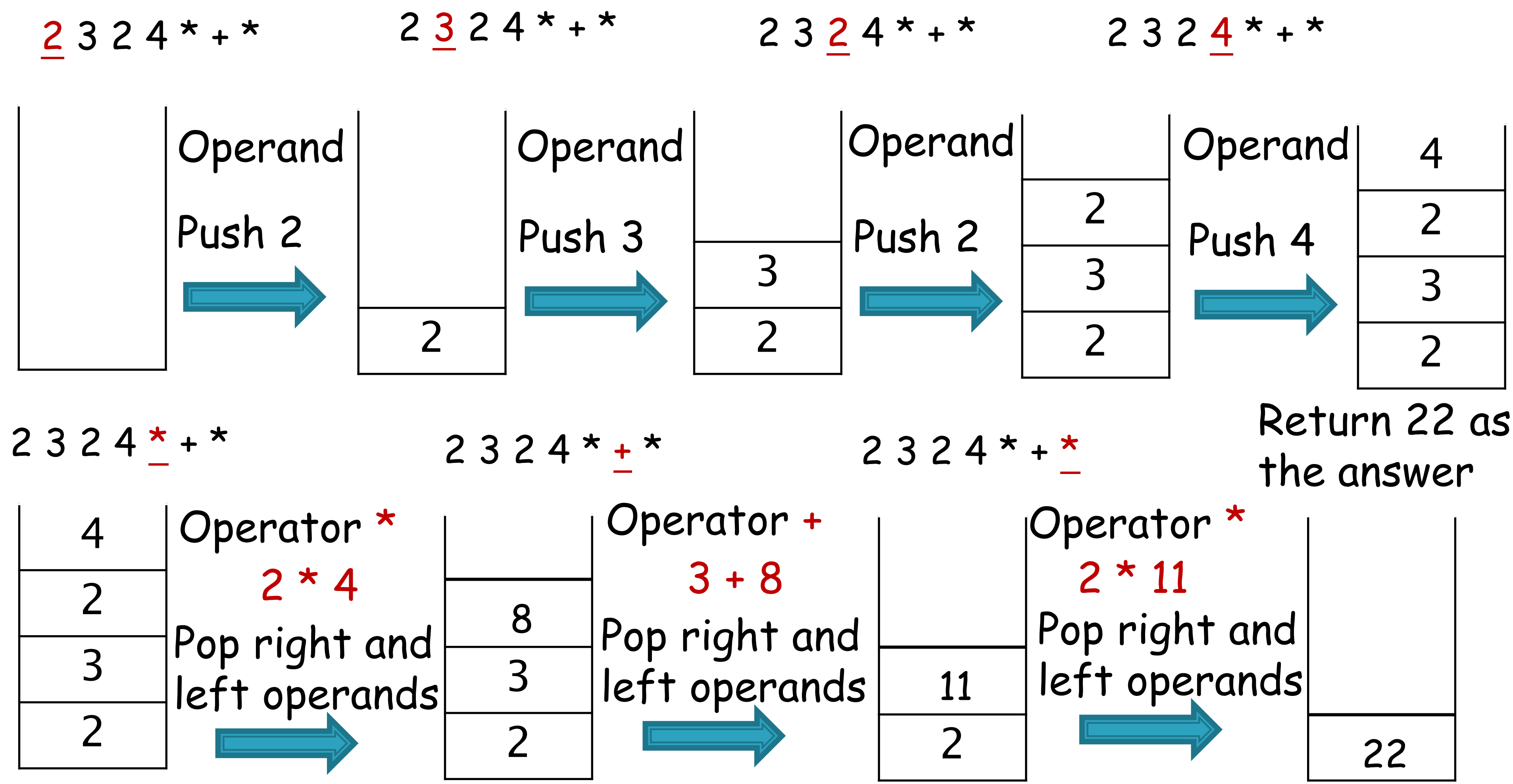




Return 8 as the answer

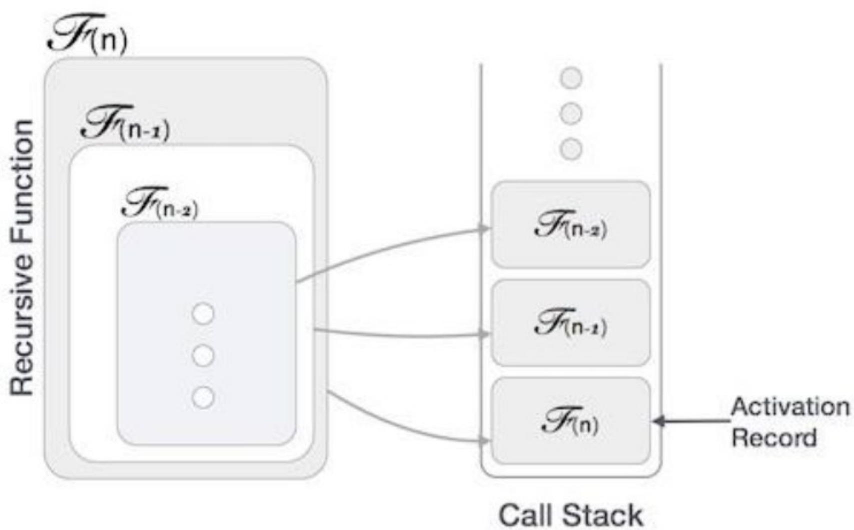
Practice

- Evaluate the postfix expression: 2 3 2 4 * + *



Stack applications

- Call functions
  - E.g., recursive functions



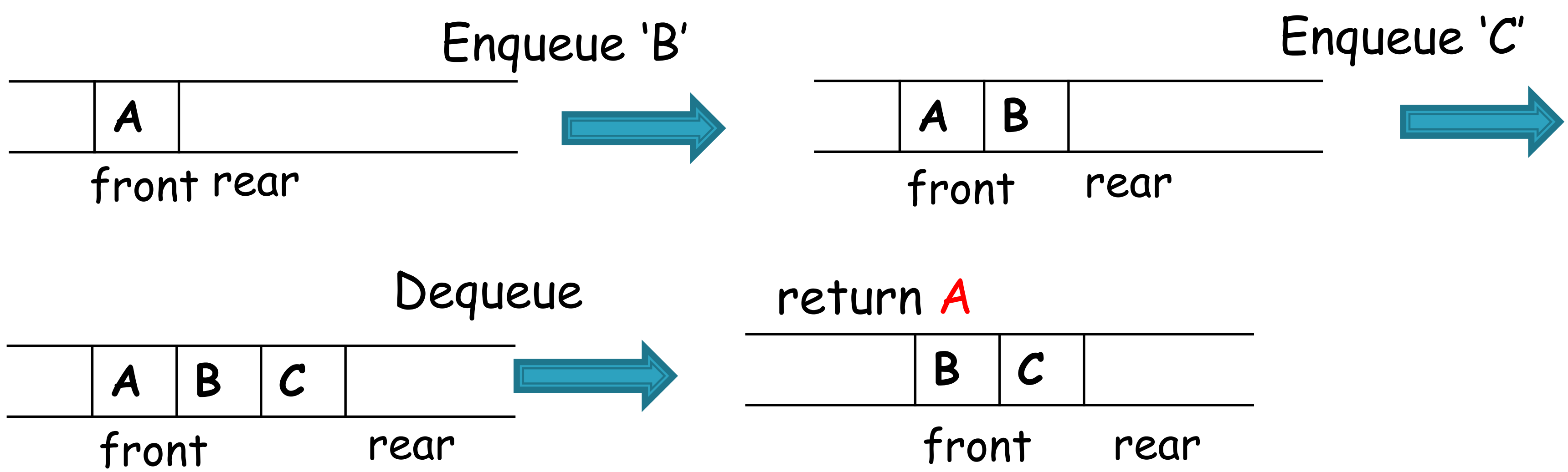
Queue

- The queue on a set S of n elements supports two constrained update operations:
  - Enqueue(e): insert a new element e into S
  - Dequeue: remove the least recently inserted element from S, and return it

- Queue follows:
  - First-In-First-Out (FIFO): The first element being enqueued into a queue is the first element dequeued
  - We add from one end, called the rear, and delete from the other end, called the front

Example: Updates to Queues

- 1. A queue Q with only one element 'A'
- 2. Enqueue 'B' to Q
- 3. Enqueue 'C' to Q
- 4. Deqeue from Q



Queue Implementations

- Option 1: Linked list
- Option 2: Arrays

Queue Design with Linked List

- CreateQueue: create a linked list L

Algorithm: *CreateQueue(capacity)*

```
Q<-Allocate new memory for queue
Q.L <- allocate new memory for list
return Q
```

- Enqueue: add e to the end of the linked list

Algorithm: Enqueue(Q,e)

```
insert(Q.L,e)
```

- Dequene: retrieve the first element, i.e., $L.head.next.element$, and delete the first node, i.e., $L.head.next$.

Algorithm: Dequeue(Q)

```
if isEmpty(Q.L) error “Queue empty”
frontNode = Q.L.head.next
frontElement = frontNode.element.
delete(Q.L, frontNode)
return frontElement
```

Queue Design with Linked List

- isEmpty: return true if the linked list is empty, otherwise return false.

Algorithm: ***isEmpty(Q)***

```
return isEmpty(Q.L)
```

- isFull: always return NO

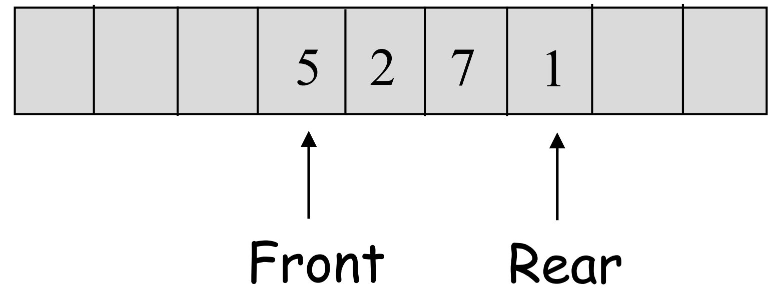
Algorithm: ***isFull(Q)***

```
return false
```

Queue using array

- Array implementation

```
public class Queue_Array {
        int capacity;
        int front, rear, size;
        ElementType[] array;
}
```



Operations:

- To enqueue an element X

  Increment Size and Rear, then
  set Queue[Rear] = X;

- To dequeue an element

  Return the value of Queue[Front],
  decrease Size, and then
  increment Front.

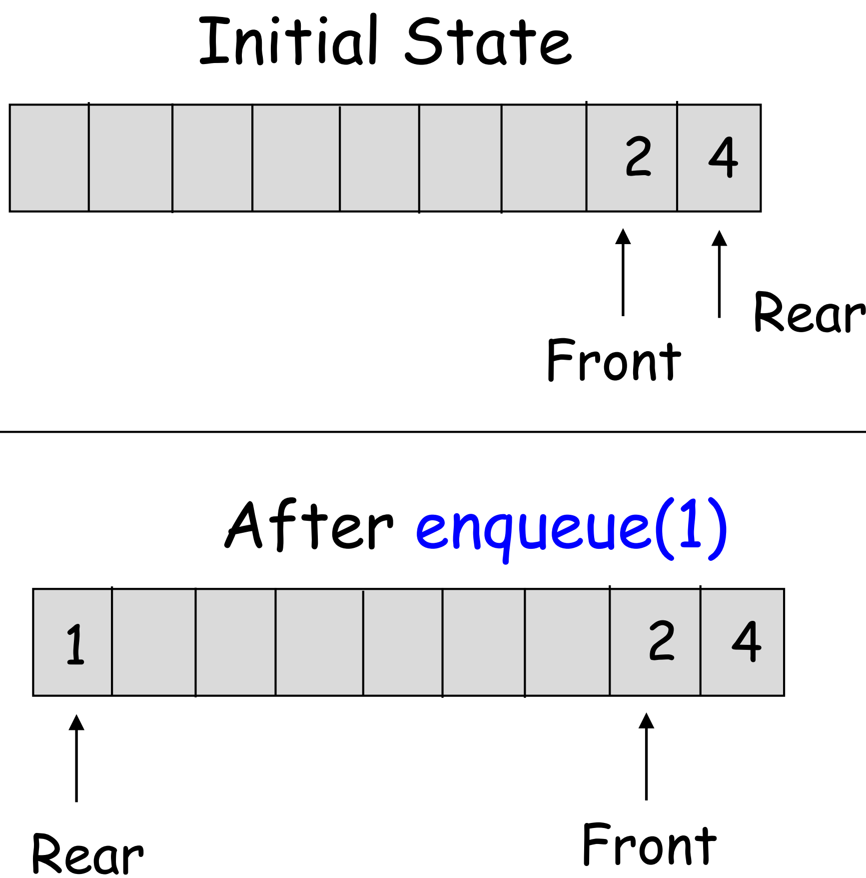
Queue using array

- Problem: may run out of rooms

  (1) Keep front always at 0 by shifting the contents up the queue, but this solution is inefficient

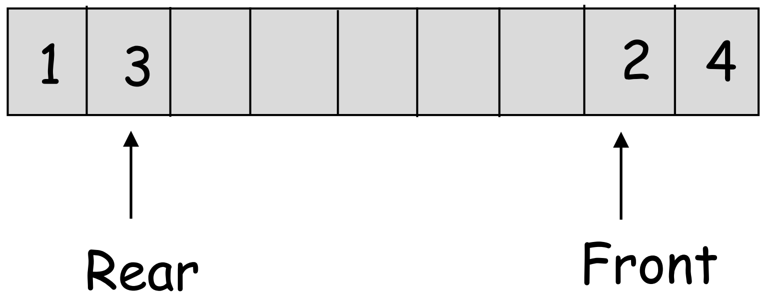
  (2) Use a circular queue (wrap around & use a length variable to keep track of the queue length)

Circular Array Implementation

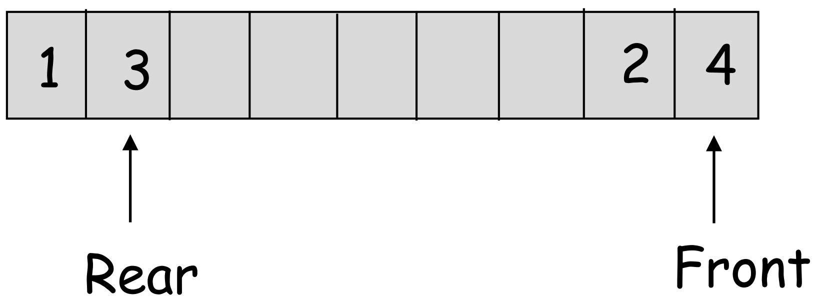


Circular Array Implementation

After enqueue(3)

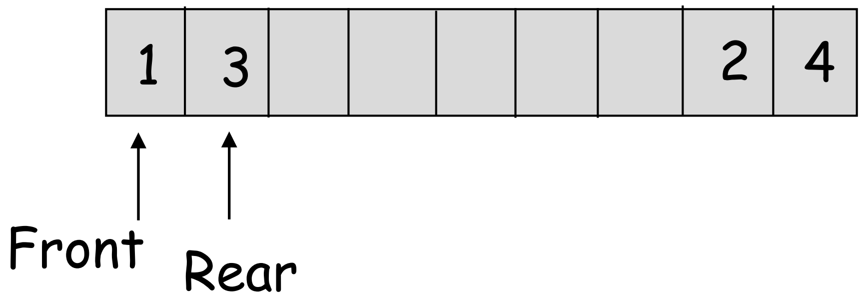


After dequeue, which returns 2

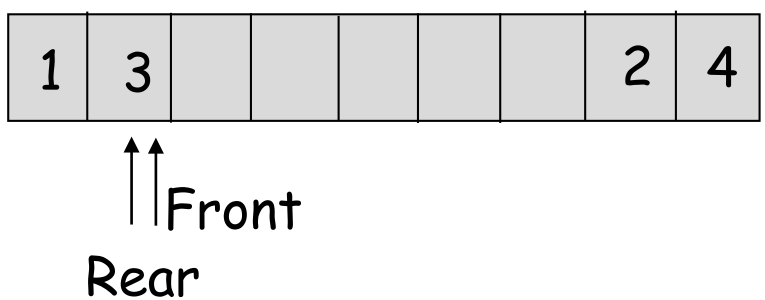


Circular Array Implementation

After dequeue, which returns 4



After dequeue, which returns 1



Queue implementation

```
public boolean enqueue(Object x) {
        if (isFull())
                return false;
        else {
                size++;
                rear = (rear + 1) % capacity;
                array[rear] = x;
                return true;
        }
}
```

Queue implementation

```
public ElementType dequeue() {
        int first;
        if (isEmpty())
                return null;
        else {
                size--;
                first = front;
                front = (front + 1) % capacity;
                return array[first];
        }
}
```

Applications of Queues

- Print jobs
  - Jobs are arranged in order of arrival
- Computer networks
  - Users are given access to the file server on a first-come first-served basis
- Real-life waiting lines

Recommended reading

- Reading
  - Chapter 10, textbook

- Next lectures
  - Trees