INT202
Complexity of Algorithms
Data Structures

XJTLU/SAT/INT

SEM2 AY2020-2021

Review

- The actual run time is difficult to assess.
  - Input, CPU frequency, RAM, data transmission speed, programs preempting resources
- The size of the problem is often the most important factor in determining algorithm running time.
  - Define the time complexity of the algorithm
  - Growth rate, asymptotic notations

Data Structures

- Algorithmic computations require data (information) upon which to operate.

- The speed of algorithmic computations is (partially) dependent on efficient use of this data.

- *Data structures* are specific methods for storing and accessing for information.

- We study different kinds of data structures as one kind may be more appropriate than another depending upon the *type* of data stored, and *how* we need to use it for a particular algorithm.

D

Data Structures: Stacks

A stack is a *Last-In, First-Out* (LIFO) data structure.

- Items can be inserted into a stack at any time (assuming no stack-overflow).

- We only have direct access to the last element that was inserted.

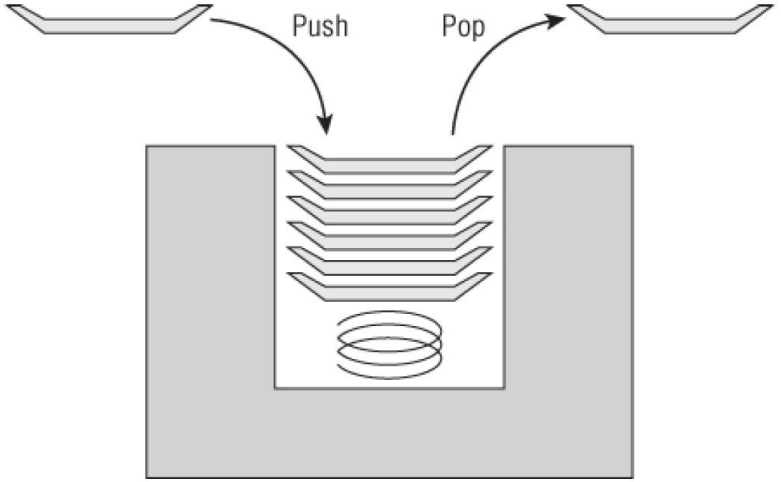
For example, Web browsers store recently visited web addresses on a stack.

Stack Abstract Data Type

A *stack* is an Abstract Data Type (ADT) and supports the following methods:

- *push(Obj)* : Insert object *Obj* onto the top of the stack.
- *pop()* : Remove (and return) the object from the top of the stack. An error occurs if the stack is empty.

In addition to *push* and *pop* there are also typically methods like:

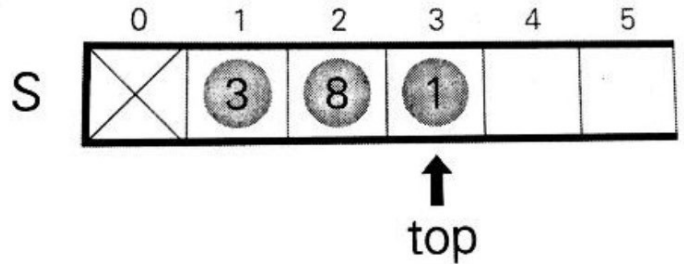


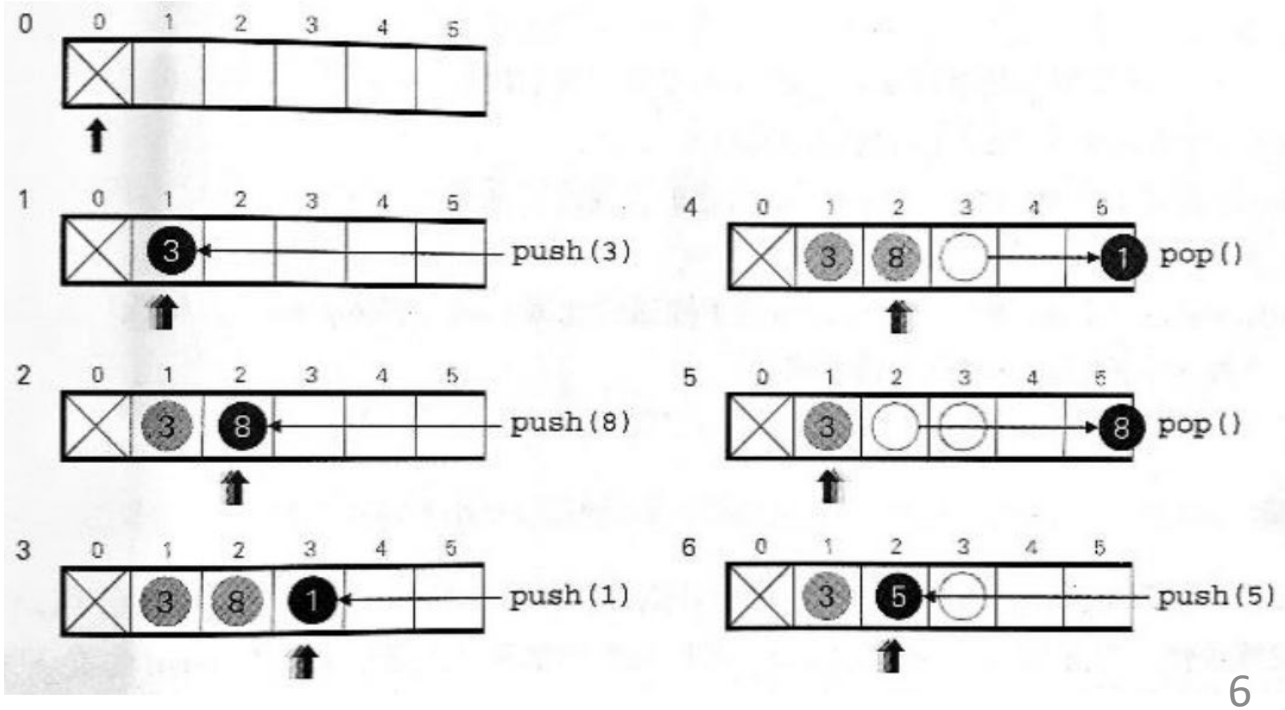
- *initialize() : initialize a stack*
- *isEmpty()* : returns a **true** if stack is empty, **false** otherwise.
- *isFull()* : returns a true if stack is full, false otherwise.

Stack: Push and Pop methods

```
PUSH(Obj)
1   ▷ Check to see if stack is full
2   if size() == N
3       then indicate stack-full error occurred
4   else t ← t + 1
5            S[t] ← Obj
```

```
POP()
1   ▷ Check to see if stack is empty
2   if isEmpty()
3       then indicate "stack empty" error occurred
4   else Obj ← S[t]
5            S[t] ← null
6            t ← t − 1
7   return Obj
```





Stacks: Applications

1. Important in *run-time environments* of modern procedural languages (e.g., C, C++, Java).

2. Evaluating arithmetic expressions can be performed using a stack if they are given using *postfix* notation (Reverse Polish notation (RPN), named for its developer Jan Łukasiewicz).

Stacks: Applications

Problem : Reversing an Array

The following pseudocode shows this algorithm:

```
ReverseArray(Data: values[])
        // Push the values from the array onto the stack.
        Stack: stack = New Stack
        For i = 0 To <length of values> - 1
            stack.Push(values[i])
        Next i
        // Pop the items off the stack into the array.
        For i = 0 To <length of values> - 1
            values[i] = stack.Pop()
        Next i
End ReverseArray
```

Stacks: Applications

Problem : Reverse Polish notation

In a standard arithmetic expression we write operands interspersed with the arithmetic operations, e.g. x + y where x and y are operands and the + addition operator is applied to add y to x . This is also referred to as infix notation.

With infix notation it is important to understand the precedence of operations. For example, which operation is performed first in the expression $4 + 3 * 9$?

Stacks: Applications

Problem : Reverse Polish notation

In postfix notation the operands precede the arithmetic operations, e.g. x y +, x y z + $*$ or x y + z $*$. When you encounter an operator, it applies to the previous two operands in the list, in that order.

e.g.: You'd want to represent an expression like $-x$, i.e. the unary negation symbol, as something like x -1 $*$ using postfix notation

Stacks: Applications

Problem : Reverse Polish notation

x y +, x y z + ∗, x y + z ∗

In the first example above we simply take x and y and add them together.

In the second we take y and add z to it, then finish by multiplying y + z by x .

The third expression computes (x + y) ∗ z.

Stacks: Applications

Problem : Reverse Polish notation

► The postfix expression

x y w z / − ∗

gets translated into the expression

x ∗ (y − w/z)

whereas

x y w /z − ∗

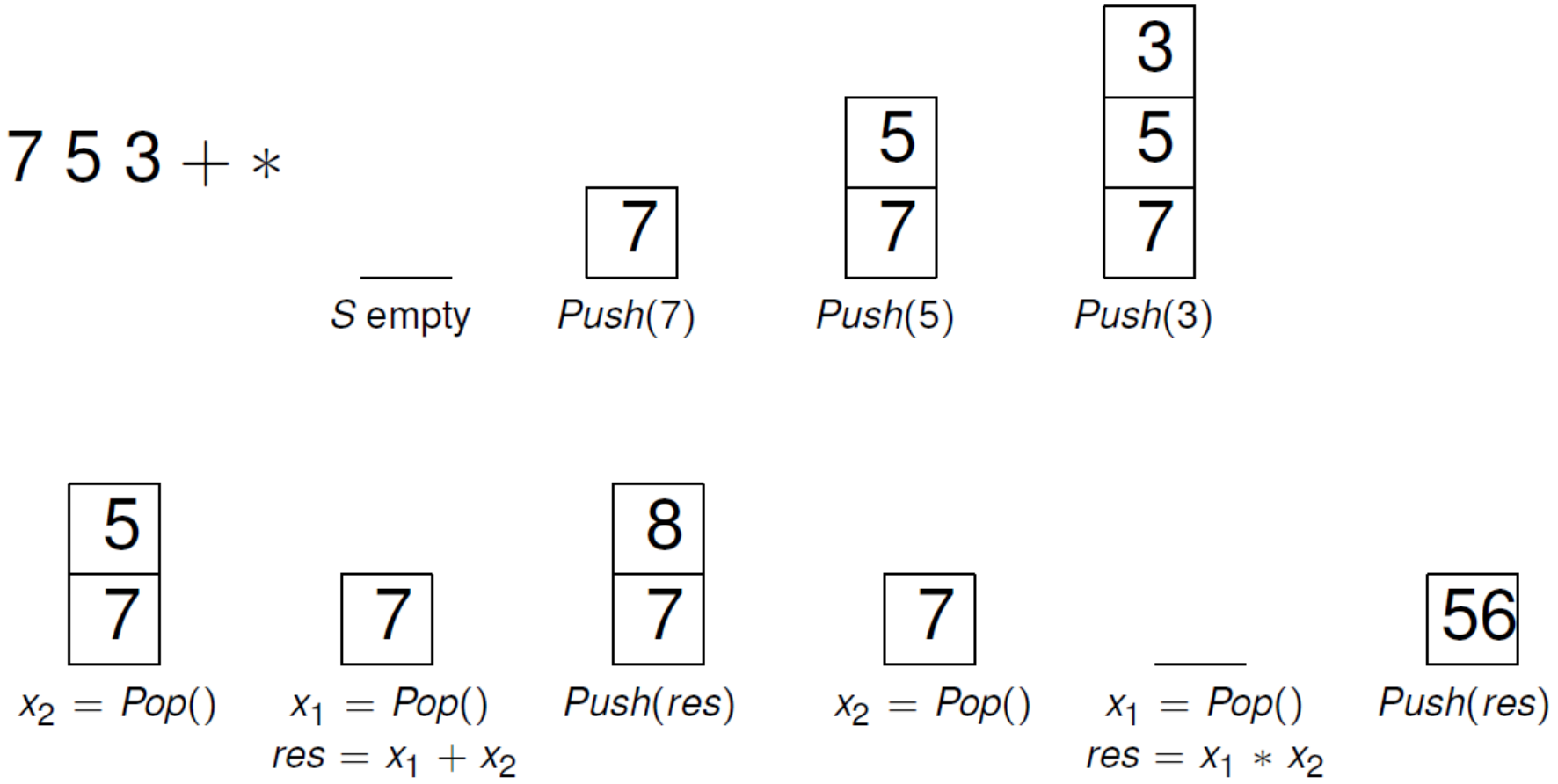
means

x ∗ (y/w − z).

► Given an expression in postfix notation we can use a stack to obtain the result.

Stacks: Applications

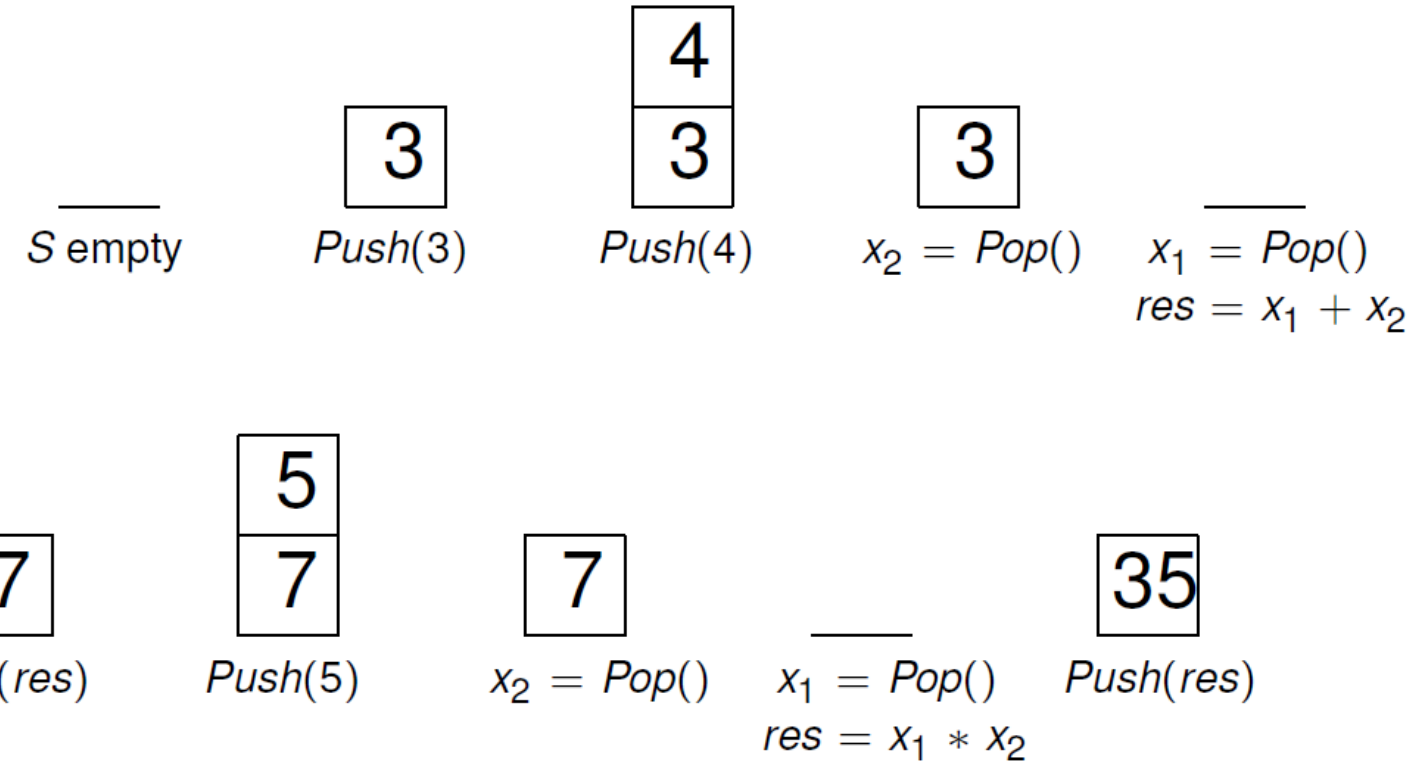
Problem : Reverse Polish notation



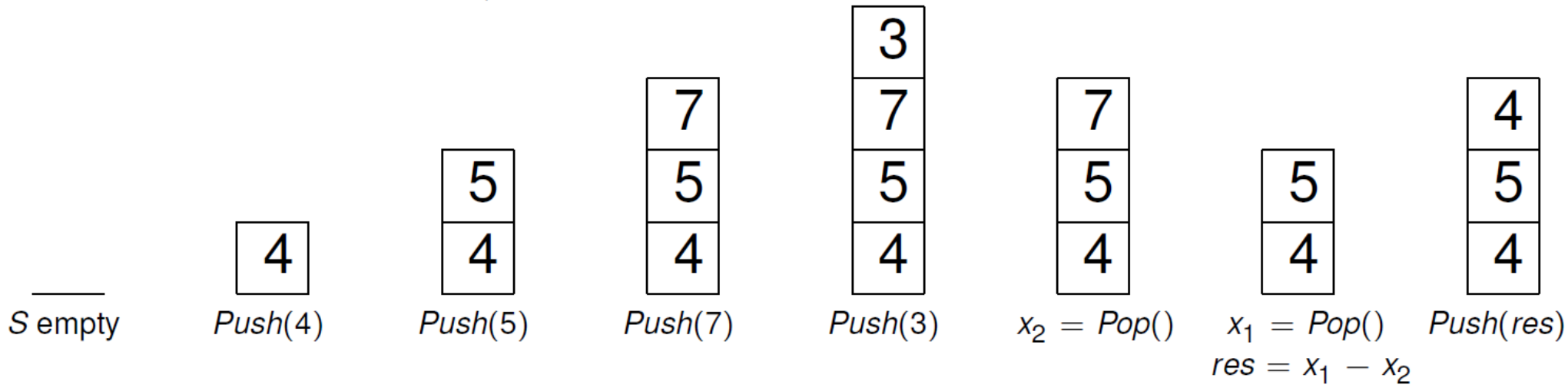
Stacks: Applications

Problem : Reverse Polish notation

$3\ 4\ +\ 5\ *$



$4\ 5\ 7\ 3\ -\ *\ +\ 3\ /$


S empty
Push(4)
Push(5)
Push(7)
Push(3)
x2 = Pop()
x1 = Pop()
res = x1 − x2
Push(res)
x2 = Pop()
x1 = Pop()
res = x1 ∗ x2
Push(res)
x2 = Pop()
x1 = Pop()
res = x1 + x2
Push(res)
Push(3)
x2 = Pop()
x1 = Pop()
res = x1 / x2
Push(res)

Stacks: Applications

Problem : Reverse Polish notation

► Using postfix notation, there is no operator precedence, the order (from left to right) of the operators in the postfix notation is the order in which they are executed.

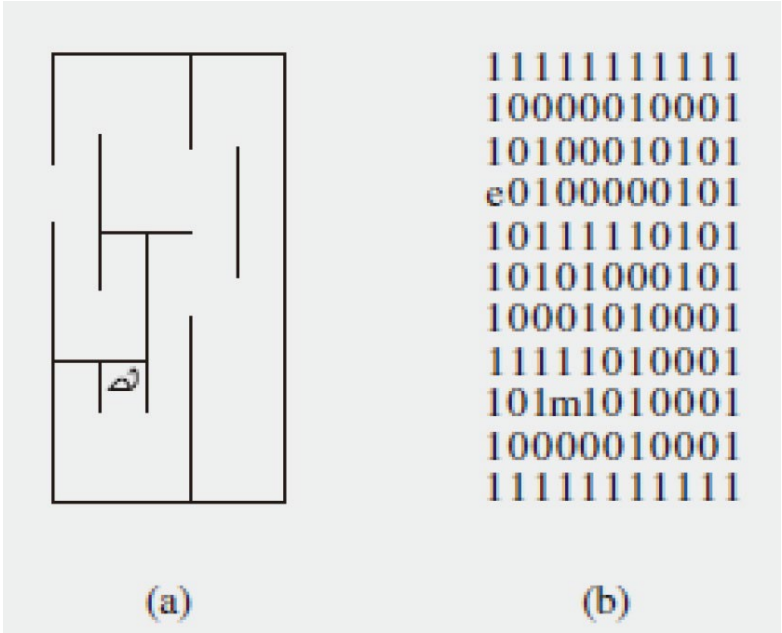
► Postfix notation really mimics the way in which we go about computing an expression given in the “usual” (infix) notation.

► Postfix notation removes the need for brackets (provided we don’t confuse the subtraction operator and the “-" symbol that appears in a negative number, so correct formatting and processing of numbers is important if read by a machine).

Stacks: Applications

Problem : Exiting a Maze.

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze.



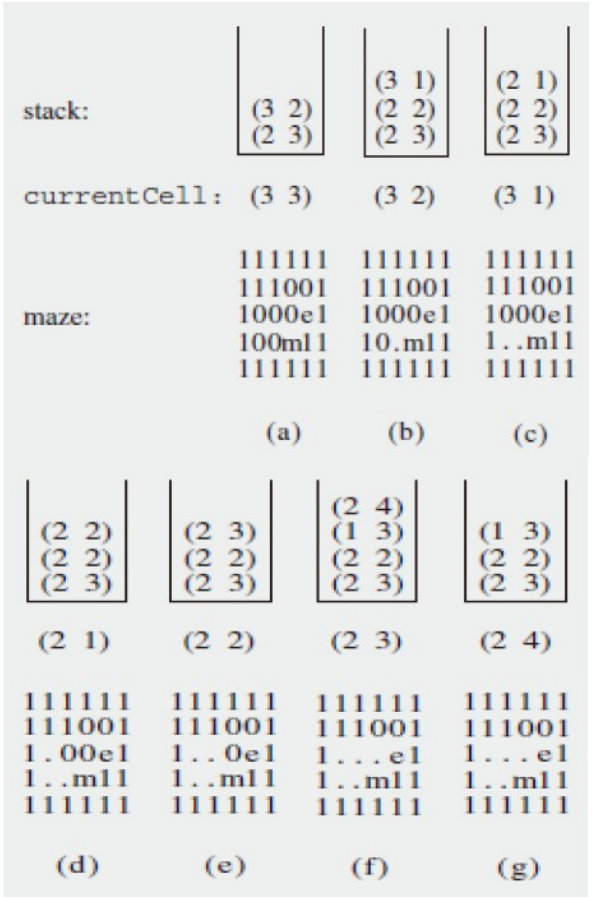
Write a program to solve the above problem by using stacks.

Stacks: Applications

Problem : Exiting a Maze.

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze.

```
  012345
0111111
1111001
21000e1
3100m11
4111111
```



Write a program to solve the above problem by using stacks.

```
exitMaze()
initialize stack, exitCell, entryCell, currentCell = entryCell;
while currentCell is not exitCell
        mark currentCell as visited;
        push onto the stack the unvisited neighbors
of currentCell;
         If stack is empty
               failure;
          else pop off a cell from the stack and make it
currentCell;
success;
```

Data Structures: Queues

- A queue is a *First-In, First-Out* (FIFO) data structure, like queues you're used to in the real world.
- Objects can be inserted (at the rear) into a queue at *any* time, but only the element at the front of the queue can be removed.
- An example of a queue is a list of *jobs* sent to a printer for printing waiting lines
- We say that elements *enter* the queue at the *rear* and are *removed* from the front.

Queue ADT

A queue ADT supports the following fundamental methods:

- *enqueue(Obj)*: inserts object *Obje*t the *rear* of the queue.
- *dequeue()*: removes and returns the object from the *front* of the queue. An error occurs if the queue is *empty*.

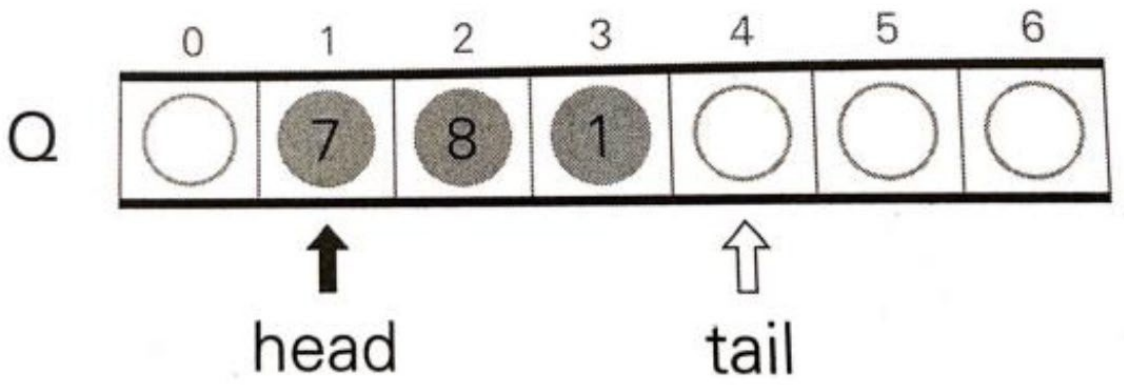
In addition to *enqueue()* and *dequeue()* there are also:

- *size(): Return the number of objects in the queue.*
- *isEmpty()*: Returns true if queue is empty, and false otherwise.
- isFull(): Returns true if queue is full, and false otherwise.
- front(): Return, but do not remove, the object at the front of the queue. An error is returned if the queue is empty.

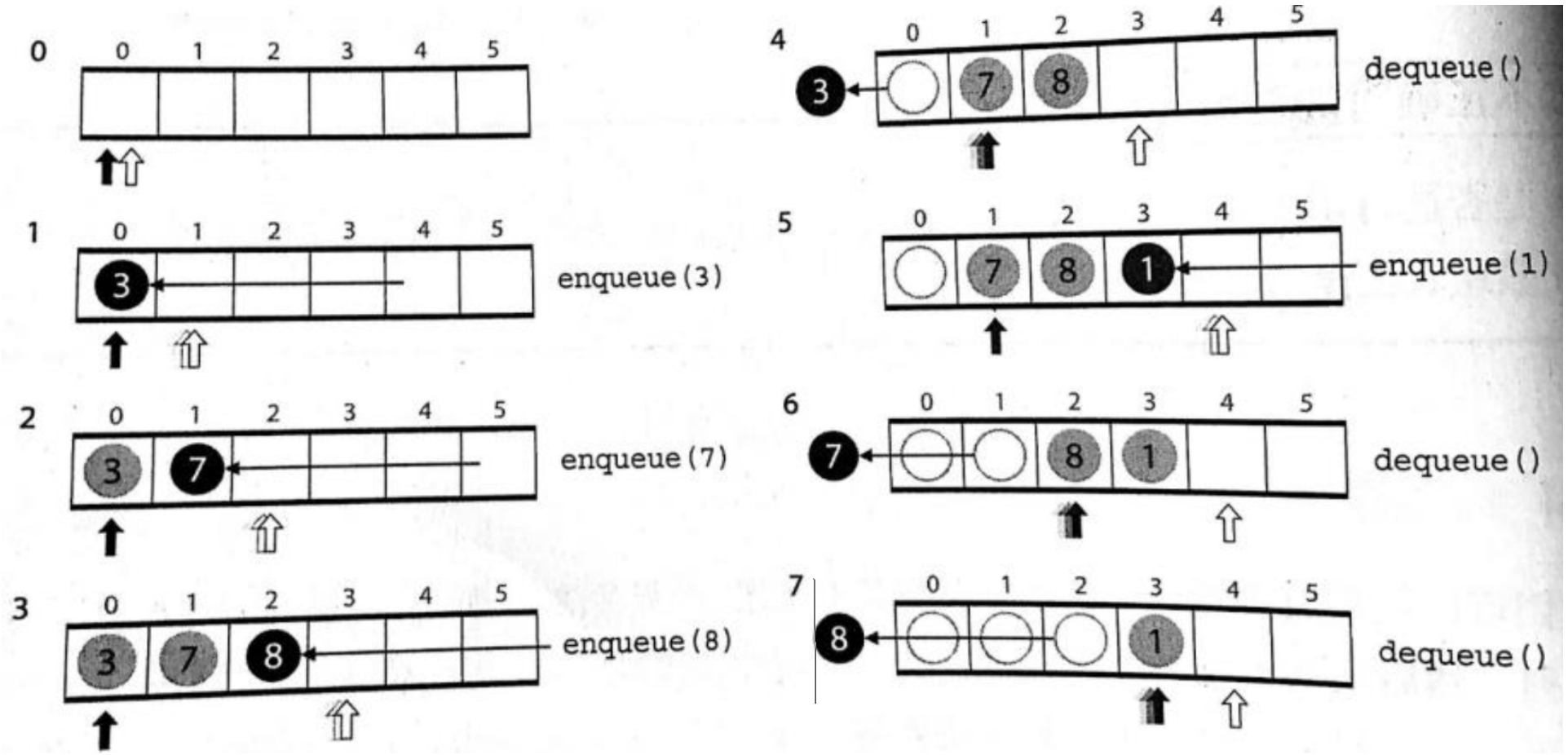
Queue ADT

A queue ADT supports the following fundamental methods:

- *enqueue(Obj)*: inserts object *Obje*t the *rear* of the queue.
- *dequeue()*: removes and returns the object from the *front* of the queue. An error occurs if the queue is *empty*.



Queue ADT

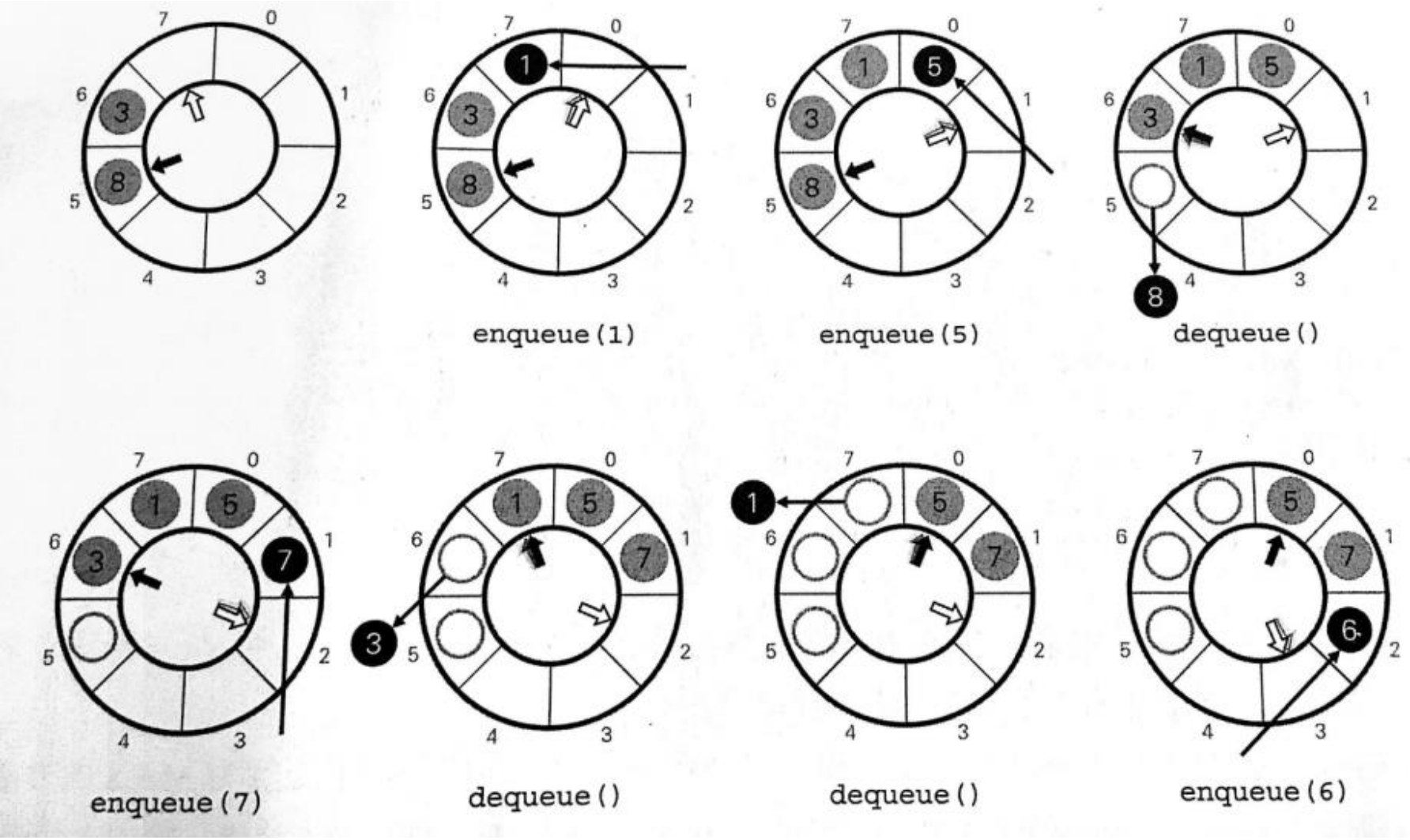


Moving to the end of the array

->out of memory

Queue ADT

A circular queue holding the values 3 and 8



Queue and Multiprogramming

► Multiprogramming achieves a limited form of parallelism.
► Allows multiple tasks or threads to be run at the same time.
► For example, a computer (or program) might use multiple threads, one of which is responsible for catching mouse clicks, whilst another may be responsible for drawing animations on the screen.
► When designing programs (or operating systems) that use multiple threads, we must not allow a single thread to monopolize the CPU.
► One solution is to use a queue to allocate CPU time to threads in a round robin protocol. Here we can use a queue that holds the collection of threads. The thread at the beginning of the queue is removed, allocated some portion of the CPU time, and then replaced at the rear of the queue.

Data Structures: List ADT

- A *list* is a collection of items, with each item being stored in a *node* which contains a *data* field and a *pointer* to the next element in a list.
- Data can be inserted *anywhere* in the list by inserting a new node into the list and reassigning pointers.
- A list ADT supports: *referring*, *update* (both *insert* and *delete*) as well as *searching* methods.
- We can implement the list ADT as either a *singly*-, or *doubly*-linked list.

List ADT: Referring methods

A list ADT supports the following *referring* methods:

- *first()*: Return position of first element; error occurs if list S is empty.
- *last()*: Return the position of the last element; error occurs if list S is empty.
- *isFirst(p)*: Return **true** if element *p* is first item in list, **false** otherwise.
- *isLast(p)*: Return **true** is element *p* is last element in list, **false** otherwise.
- *before(p)*: Return the position of the element in S preceding the one at position *p*; error if *p* is first element.
- *after(p)*: Return the position of the element in S following the one at position *p*; error if *p* is last element.

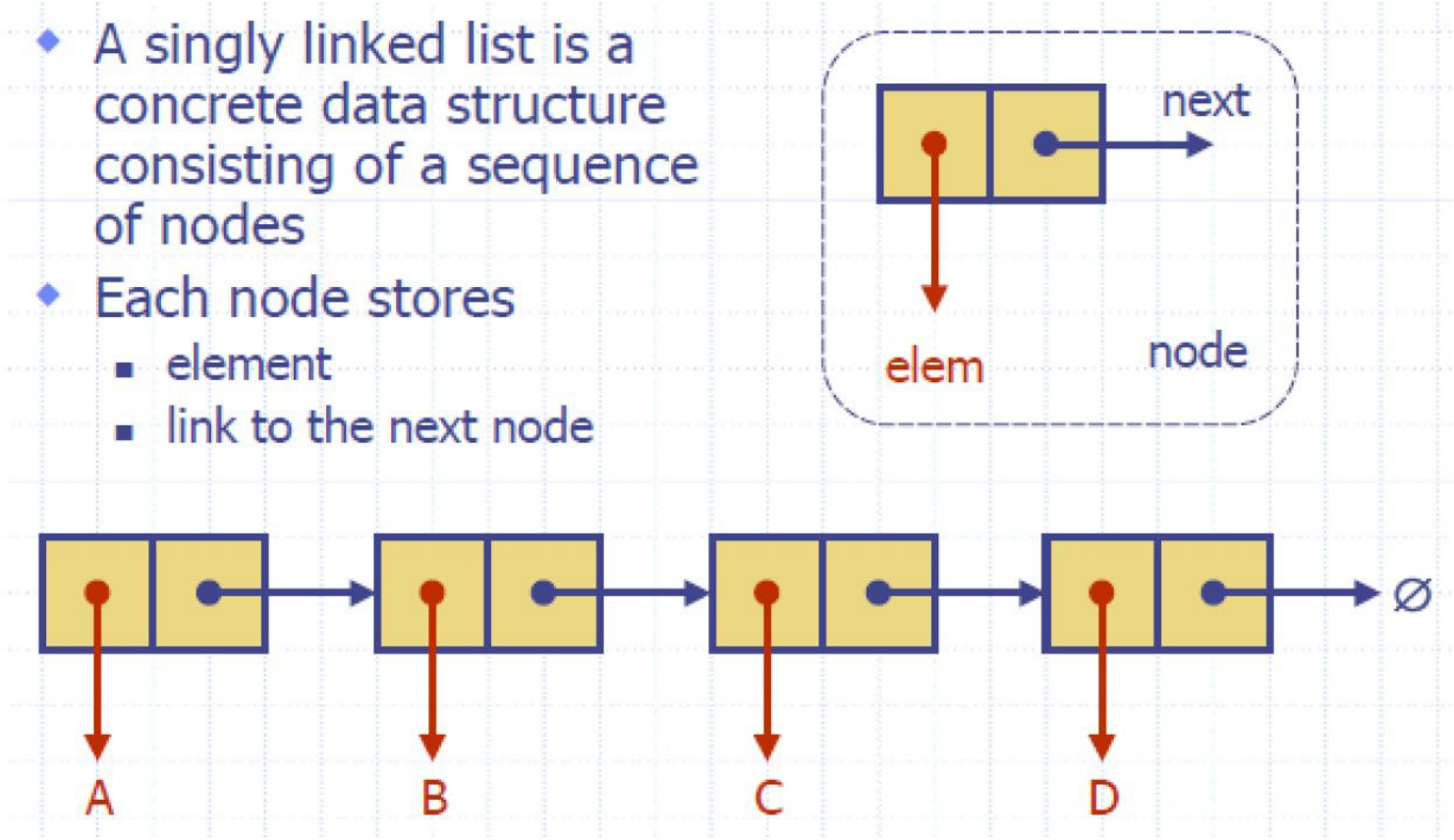
List ADT: Update methods

A list ADT supports the following *update* methods:

- *replaceElement(p,e)*: *p* - position, *e* - element.
- *swapElements(p,q)*: *p,q* - positions.
- *insertFirst(e)*: *e* - element.
- *insertLast(e)*: *e* - element.
- *insertBefore(p,e)*: *p* - position, *e* - element.
- *insertAfter(p,e)*: *p* - position, *e* - element.
- *remove(p)*: *p* - position.

Linked List

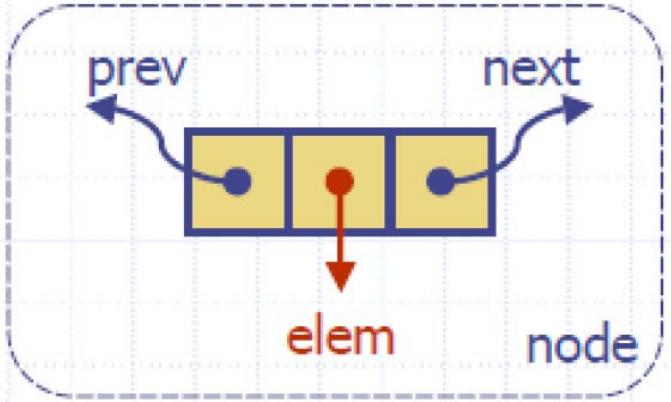
A *node* in a *singly*-linked list stores a *next* link pointing to next element in list (**null** if element is last element).

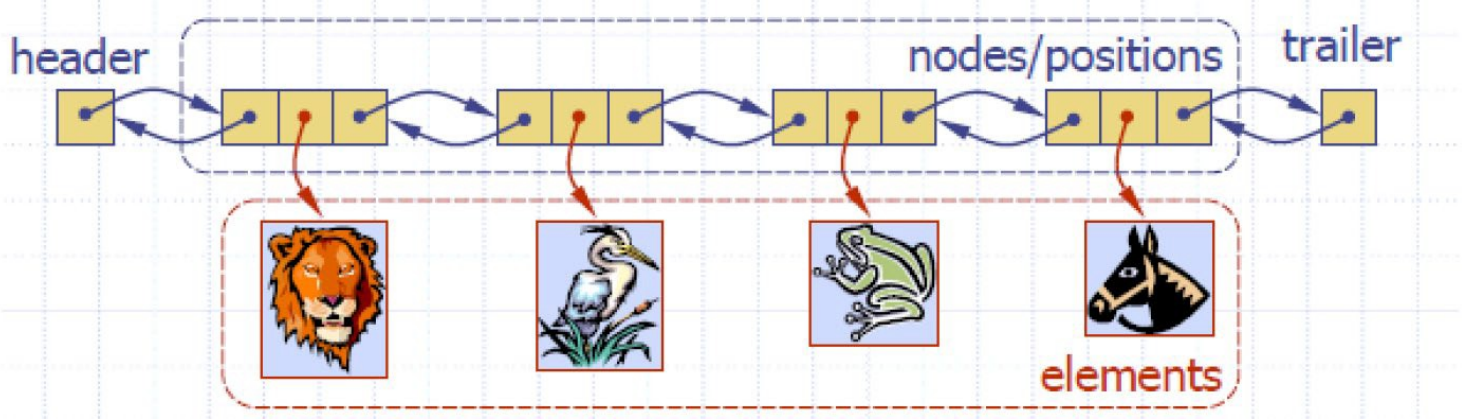


Linked List

A *node* in a *doubly*-linked list stores two links: a *next* link, pointing to the next element in list, and a *prev* link, pointing to the previous element in the list.

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



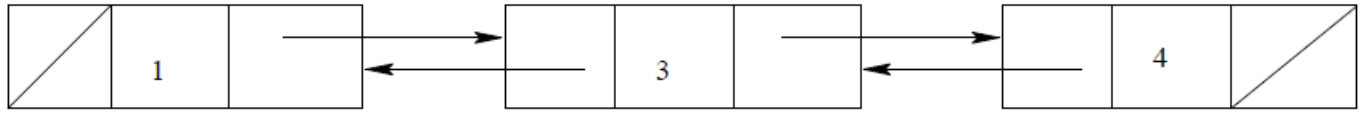


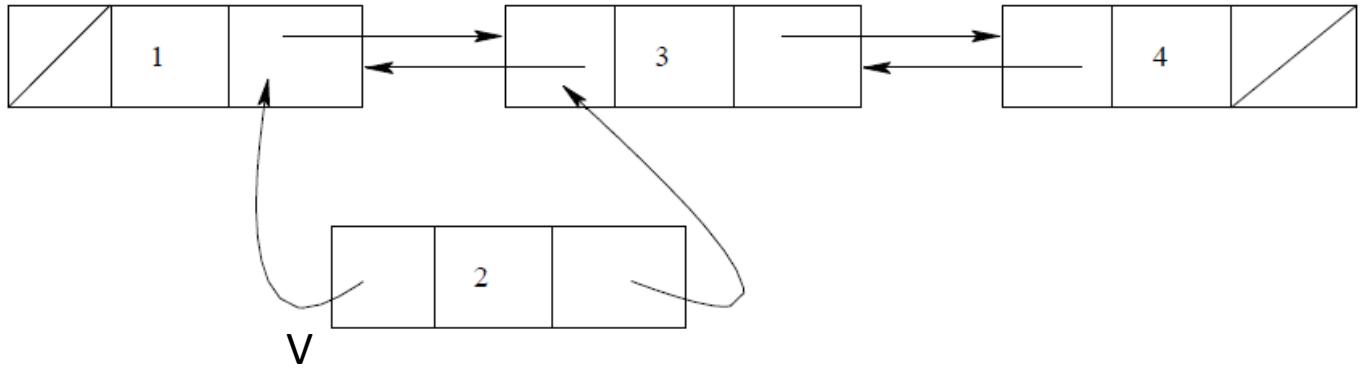
From now on, we will concentrate on doubly-linked lists.

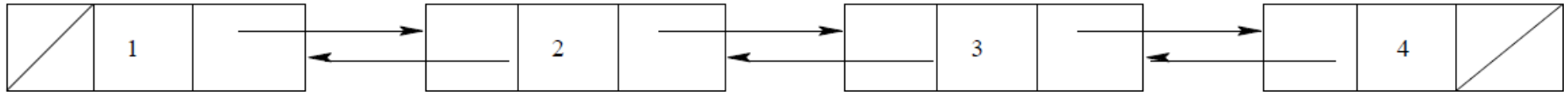
List update: Element insertion

How to insert an element?

Example: *insertAfter(1,e)*





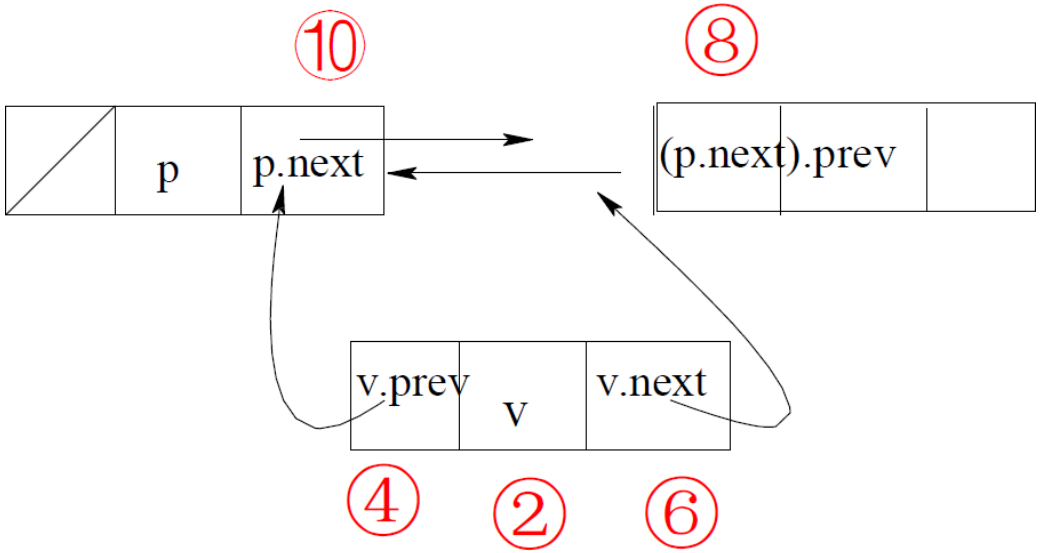


List update: Element insertion

Pseudo-code for *insertAfter(p,e)* :

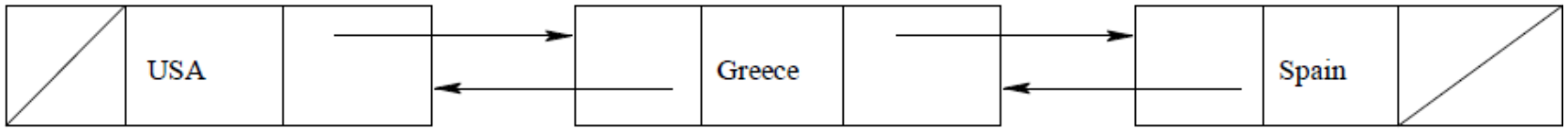
INSERTAFTER(*p,e*)

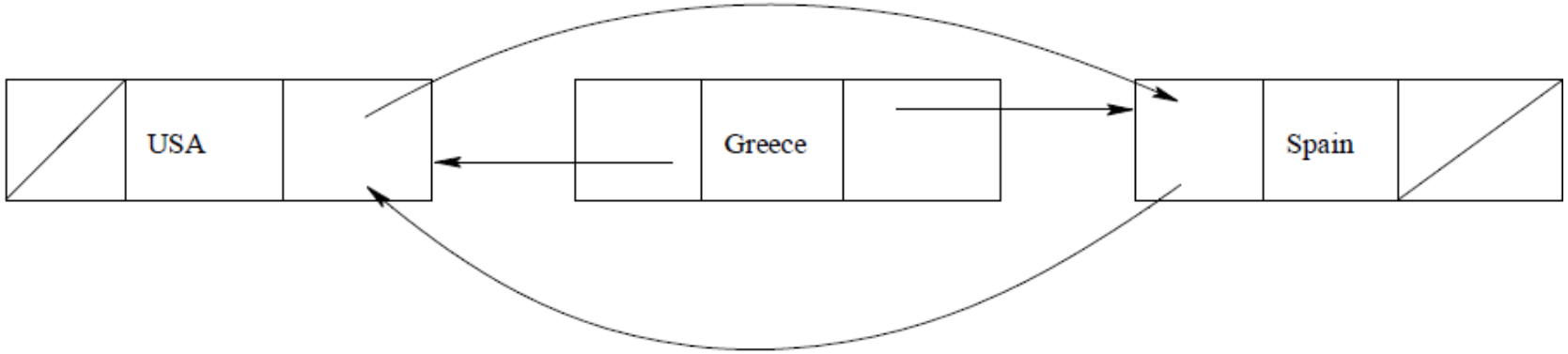
```
    //Create a new node v
2   v.element ← e
    //Link v to its predecessor
4   v.prev ← p
    //Link v to its successor
6   v.next ← p.next
    //Link p's old successor to v
8   (p.next).prev ← v
    //Link p to its new successor v
10  p.next ← v
11  return v
```

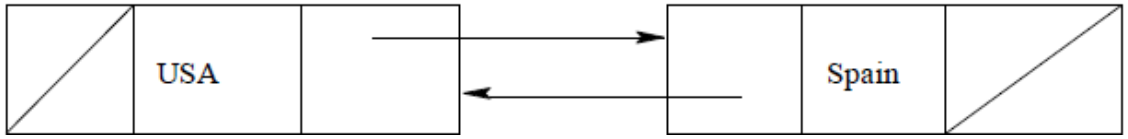


List update: Element removal

Example: *remove(2)*







List update: Element removal

The pseudo-code for *remove(p)*:

REMOVE(*p*)

```
    //Assign a temporary variable to hold return value
2   t ← p.element
    //Unlink p from list
4   (p.prev).next ← p.next
5   (p.next).prev ← p.prev
    //invalidate p
7   p.prev ← null
8   p.next ← null
9   return t
```