Chapter 3 *

Abstract Data Types (ADT)

Abstract Data Types

- ADT: set of objects together with a set of operations
- Mathematical abstraction ; no mention in ADT of *how* the set of operations is implemented
- Examples: Objects such as *lists*, *sets*, and *graphs*, along with their operations, can be viewed as ADTs
- The C++ *class* allows for the implementation of ADTs, with appropriate hiding of implementation details
- Any implementation changes should not affect the use of the ADT (completely transparent to the rest of the programme)

Usual Data Types

Integer, Double, Boolean

Implementation details of these are not known

Some operations on these can be defined: *add*, *multiply*, *read*, *write*

The programmer just uses these operations without knowing their implementation.

The List ADT

- We will deal with a general list of the form A_0, A_1, A_2, . . ., A_{N-1}

Popular operations on Lists:

- *printList* and *makeEmpty*,
- *find* (returns the position of the first occurrence of an item)
- *Access/findkth* (*read*/*change*) an element at the beginning/end/ith position.
- *Add* an element at the beginning/end/ith position
- *Delete* an element at the beginning/end/ith position

Array Implementation of a list

- A List can be implemented as an array
- Need to know the maximum number of elements in the list at the start of the program (The C++ *vector* class is different)
- Adding/Deleting an element can take $O(n)$ operations if the list has n elements.
- Accessing/changing an element anywhere takes $O(1)$ operations independent of n

Adding an element at the beginning

Suppose first position (A[0]) stores the current size of the list

Actual number of elements currsize + 1

Adding at the beginning:

- Move all elements one position behind
- Add at position 1; Increment the current size by 1

```
For (j = A[0]+1;  j > 0; j--)
    A[j] = A[j-1];
A[1] = new element;
A[0]→ A[0]+1;
```

Complexity: O(n)

Adding an element at the End

Add the element at the end

Increment current size by 1;

A[A[0]+1] = new element;

A[0]$\rightarrow$ A[0]+1;

Complexity: O(1)

Adding at kth position

Move all elements one position behind, from kth position onwards;

Add the element at the kth position

Increment current size by 1;

```
For (j = A[0]+1;  j > k; j--)
    A[j] = A[j-1];
A[k] = new element;
A[0]→ A[0]+1;
```

Complexity: O(n-k)

Deleting an Element

Deleting at the beginning:

Move all elements one position ahead;

Decrement the current size by 1

```
For (j = 1;  j  < A[0] ; j++)
    A[j] = A[j+1];
A[0]→ A[0]-1;
```

Complexity: O(n)

Deleting at the End

Delete the element at the end

Decrement current size by 1;

A[0]$\rightarrow$ A[0]-1;

Complexity: O(1)

Deleting at the kth position

Move all elements one position ahead, from (k+1)th position onwards;

Decrement the current size by 1;

```
For (j = k;  j < A[0]+1; j++)
      A[j] = A[j+1];
A[0]→ A[0]-1;
```

Complexity: O(n-k)

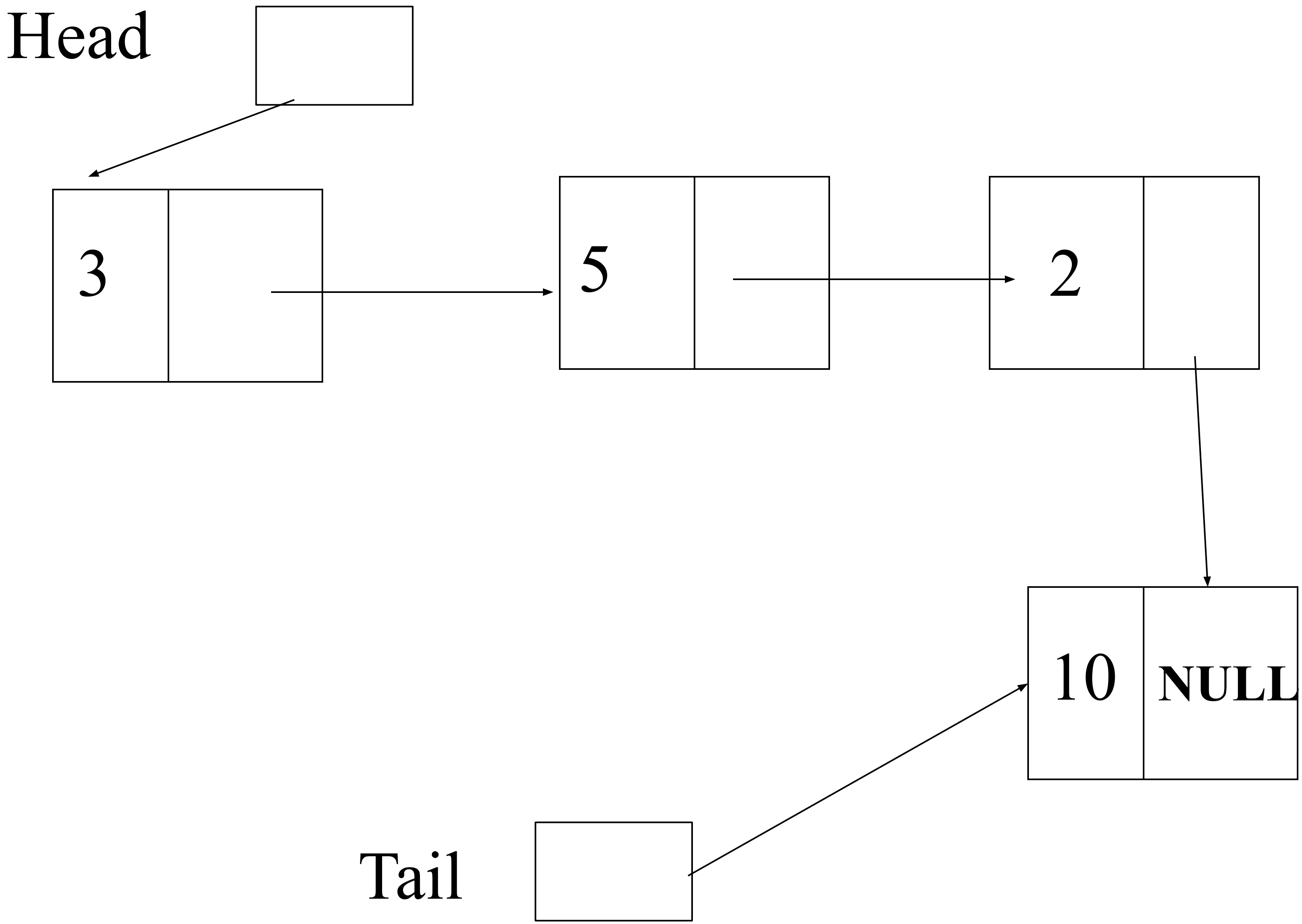
Accessing an Element at the kth position

```
A[k];
```

O(1) operation;

Linked List

- To avoid the linear cost of insertion and deletion, we need to ensure that the list contiguous storage
- Otherwise, entire parts of the list will need to be moved.
- A Linked List consists of a sequence of nodes
- A node in the linked list consists of two elements
  - Value of the corresponding element in the list
  - Pointer to the next element nextptr.
- The pointer is a null pointer nullptr for last element
- May be a special node at the beginning, known as the header node


Head
3
5
2
10
NULL
Tail

Adding an element at the beginning

Create a new node;

Element in the node has the same value as the new element;

Node pointer points to the first element (non-header)

Pointer from the header points to new node;

```
ListNode* newptr = new newNode
newptr →value = val;
newptr →next = head;
head→ newptr;
```

O(1)

Adding an element at the end

- Create a new node;
- Element in the node has the same value as the new element;
- Node pointer points to the last element;
- Pointer from the last element points to new node;

```
ListNode* newptr = new newNode;
newptr →value = val;
Tail→ nextPtr = newPtr;
tail = newPtr;
```

O(1)

Accessing an element

```
ListNode* current = head;
while (current →value != val && current →next!= 0)
     current = current →next;
if (current →value == val )
    return true;
else
    return false;
```

Complexity: O(n)

May store a pointer for the special positions, e.g., last, but then update it as necessary;

Adding at the kth position

```
ListNode* newptr = new newNode;
newptr →value = val;
ListNode* current = head;
for (count = 1; count<k; count++ && current != 0)
      current = current →next;
if (count == k ) {
      newptr →next = current → next;
      current → next = newptr;
     return true;}
else
     return false;
```

Complexity: O(k)

Deleting at the kth position

```
ListNode* current = head;
ListNode* prevPtr;
for (count = 1; count<k; count++ && current != 0) {
      prevPtr = current;
      current = current →next;   }
if (count == k ) {
      prevPtr →next = current → next;
      delete current;
      return true;
}
else
      return false;
```

Complexity: O(k)

Delete at the kth position

Complexity depends on access complexity

O(1) for deleting first element;

O(1) or O(n) for deleting the last element;
(depending on whether tail pointer exists or not)

O(k) for any other element;

Advantage of Linked List

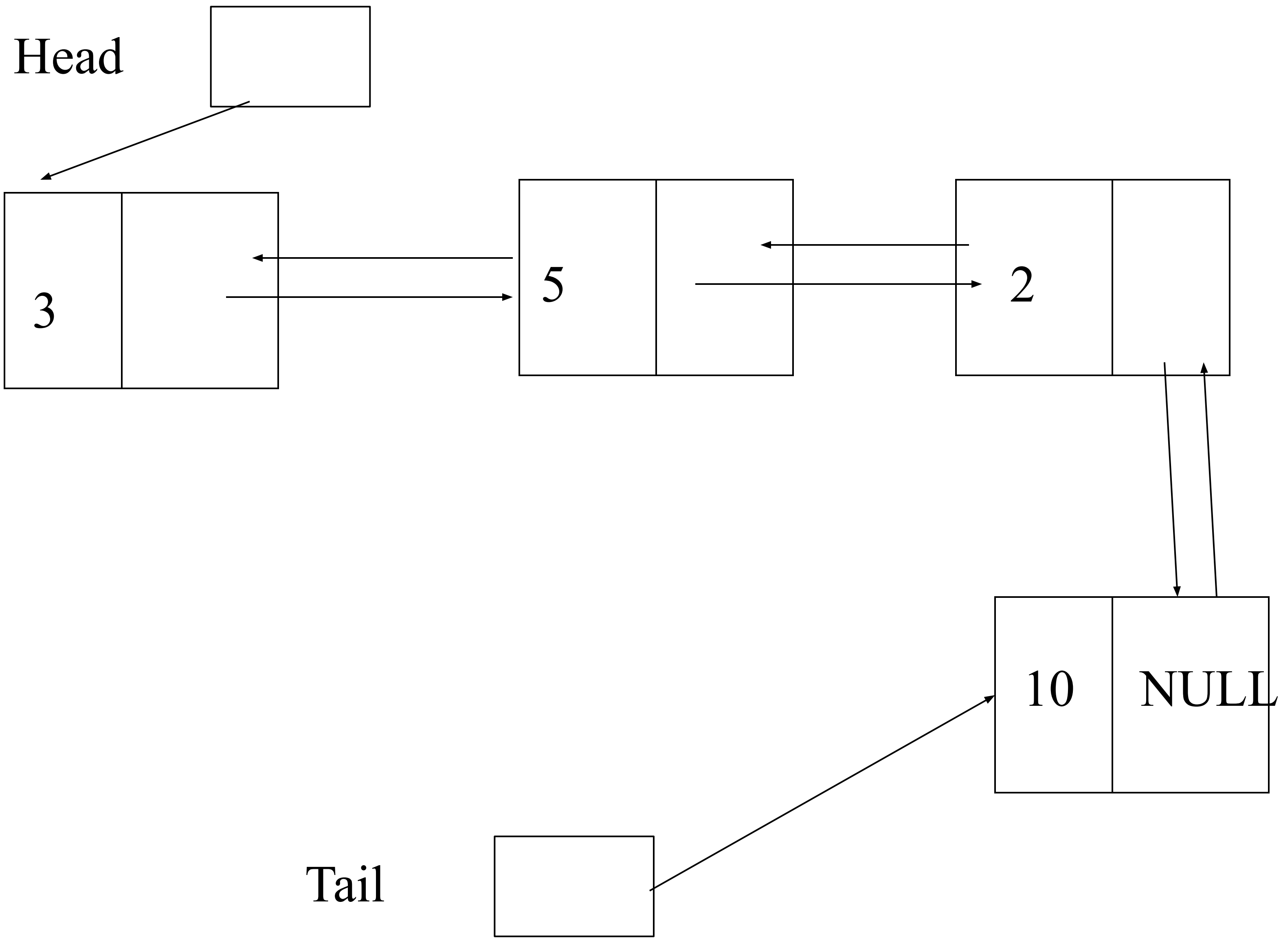
No need not know the maximum number of elements ahead of time.

Doubly Linked List

A node contains pointers to previous and next element

One can move in both directions

Complexities?


Head
3
5
2
10
NULL
Tail

Advantages Of DLL:

- Reversing the doubly linked list is very easy.
- It can allocate or reallocate memory easily during its execution.
- As with a singly linked list, it is the easiest data structure to implement.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node to be deleted but in the doubly linked list, it only required the pointer which is to be deleted.

Disadvantages Of DLL:

- It uses extra memory when compared to the array and singly linked list.
- Since elements in memory are stored randomly, the elements are accessed sequentially; no direct access is allowed.

Uses of DLLs

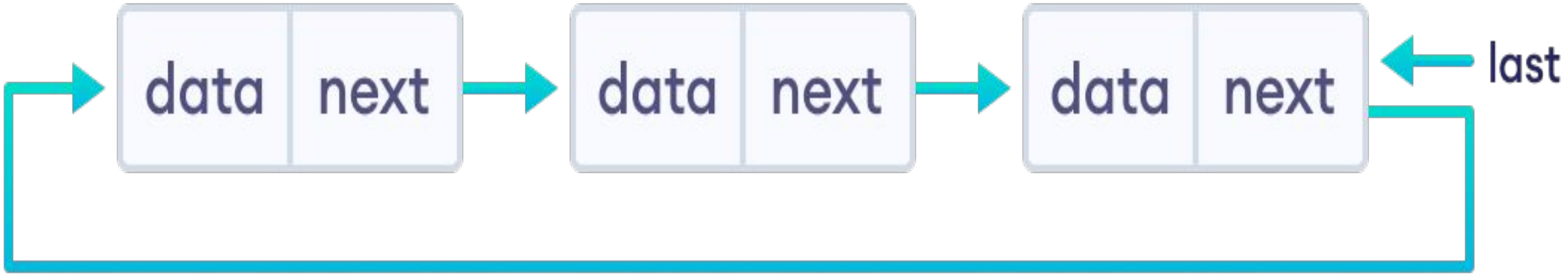
- Used in the navigation systems where front and back navigation are required.
- Used by the browser to implement backward and forward navigation of visited web pages (back and forward buttons)
- Used by various applications to implement undo and redo functionality.
- Also in many operating systems, the **thread scheduler** (the thing that chooses what process needs to run at which time) maintains a doubly-linked list of all processes running at that time. Other data structures like stacks, Hash Tables, Binary trees can also be constructed or programmed using a doubly-linked list.

Circular Lists

- The last node points to the first one

<u>Operations</u>:

- Insertion / deletion: front, end, some specific point



Circular Lists

Avantages:

- We can go to any node and traverse from any node. We just need to stop when we visit the same node again.
- As the last node points to the first node, going to the first node from the last node just takes a single step

Applications of Circular Linked Lists

It is used in multiplayer games to give a chance to each player to play the game.

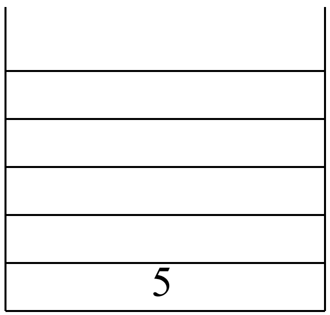
Multiple running applications can be placed in a circular linked list on an operating system. The OS keeps on iterating over these applications giving them time slices.

Stacks

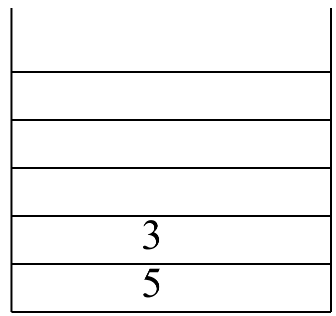
- A Stack is a list where you can access add or delete elements at one end only.
- Stacks are called ``last in first out'' (LIFO), the last added element among the current ones can be accessed or deleted.
- All of these operations take constant time.
  - Deletion is ``pop''
  - Addition is ``push''
- You can also find out whether the stack is empty but nothing else.

EXAMPLE

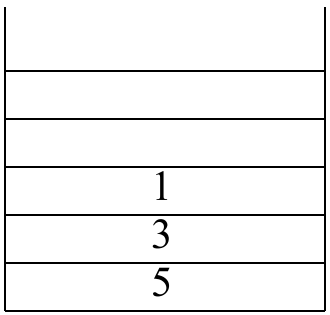
Push 5, 3, 1, Pop, Pop, Push 7



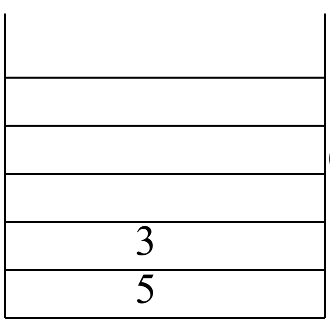
Push 5



Push 3

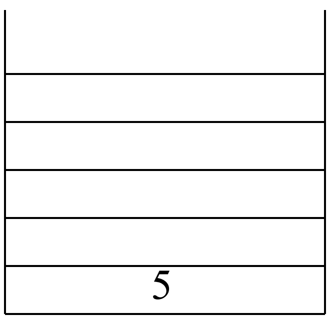


Push 1



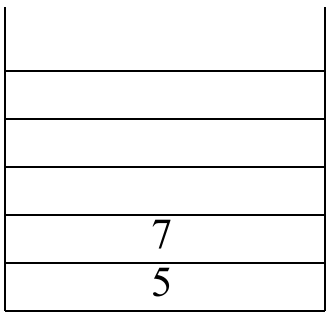
Get 1

Pop



Get 3

Pop



Push 7

Function Calls

When a subroutine calls another subroutine (including itself) it must first store its register contents.

So, it pushes the register contents into a stack.

The new function uses the registers.

When the program returns to the old function, pop the stack.

Nested Functions and Stack Overflow

Stack Implementation using an Array

Need to know the maximum number of elements

Delete/Access/Add at the end

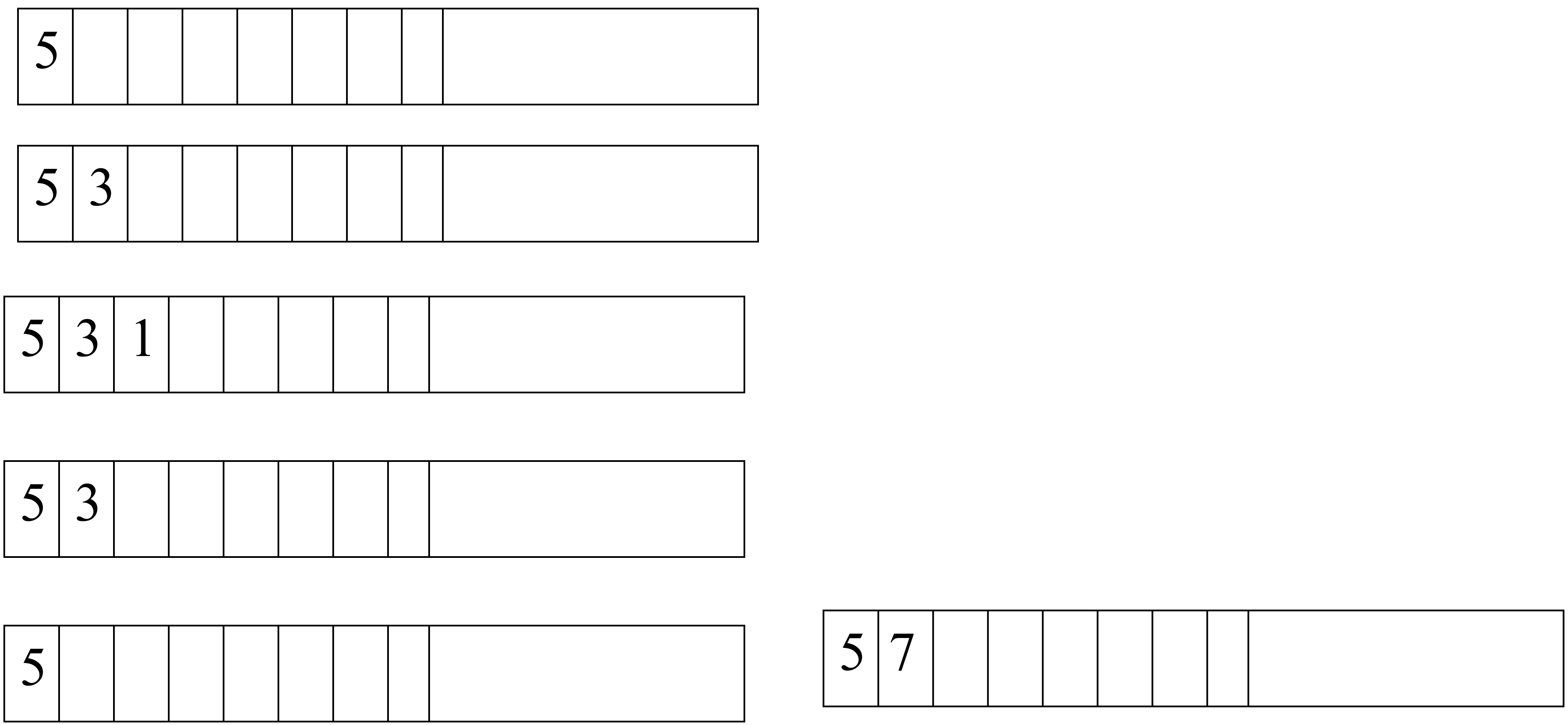
O(1) operation.

Store array length in the 0^{th} position.

Delete/Access/Add at A[array length+1]

Example

Push 5, 3, 1, Pop, Pop, Push 7



Stack Implementation using a Linked List

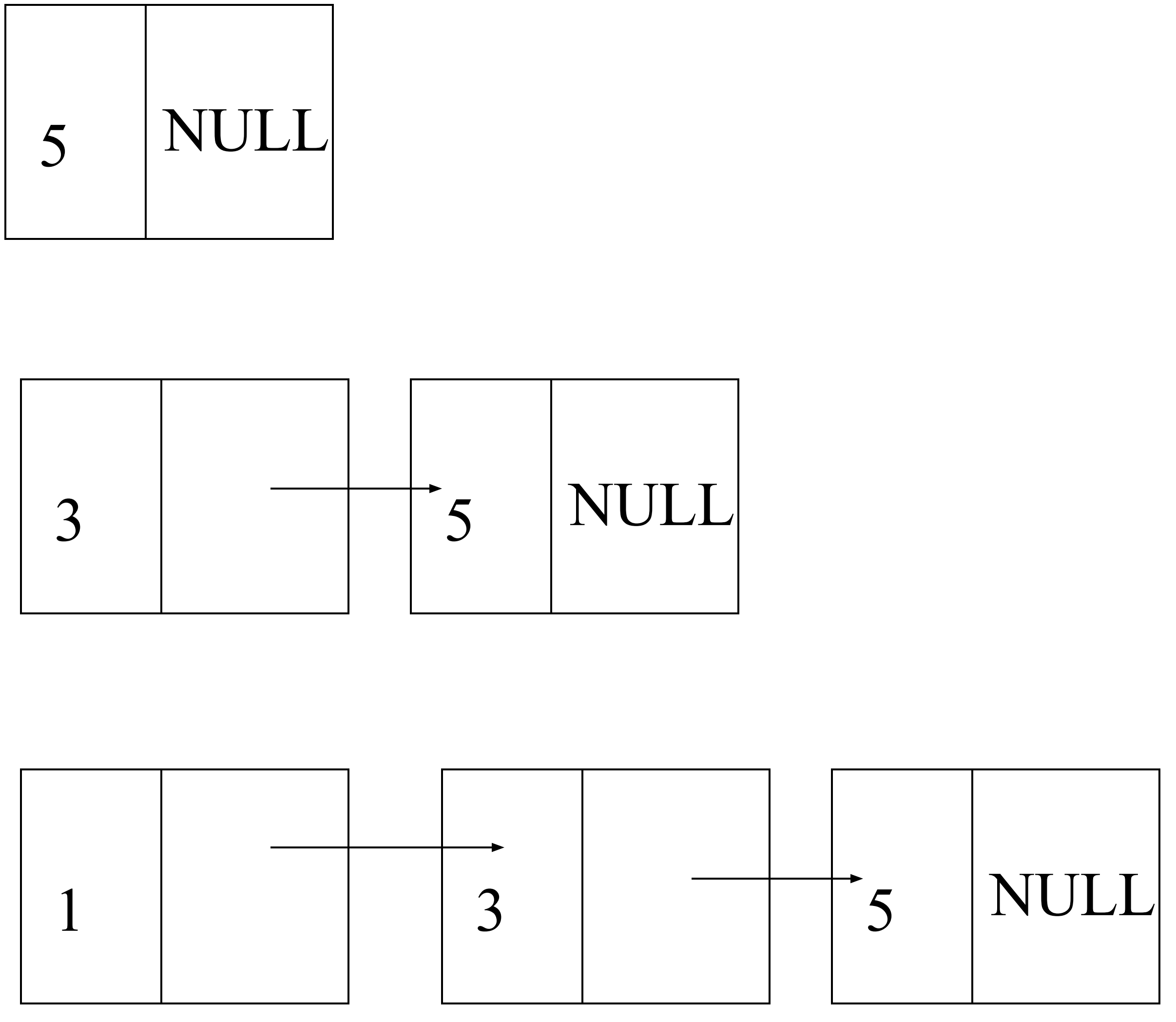
Need not know the maximum size

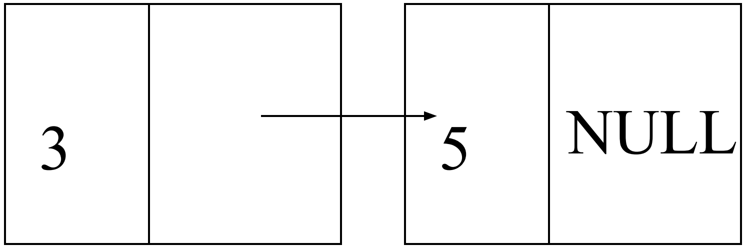
Need not have a special header.

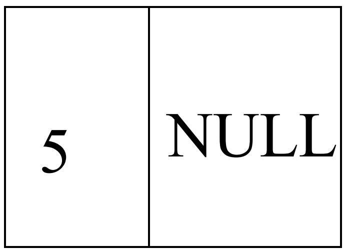
Add/Access/Delete in the beginning, O(1)

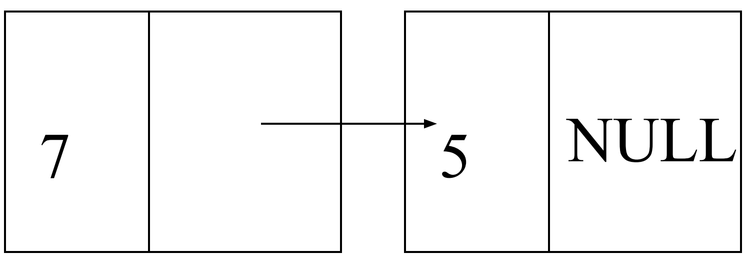
Need several memory access, deletions.

Push 5, 3, 1, Pop, Pop, Push 7




3
5
NULL


5
NULL


7
5
NULL

Uses of Stacks: Symbol Matching

Braces, parentheses, brackets, begin, ends must match each other

[{ [( )] }]

[{]}()

Easy check using stacks

Start from beginning of the file.

Push the beginning symbols you wish to match, ignore the rest.

Push brace, parentheses, brackets, ignore the alphabets

Whenever you encounter a right symbol, pop an element from the stack. If stack is empty, then error.

Otherwise, if the popped element is the corresponding left symbol, then fine, else there is an error.

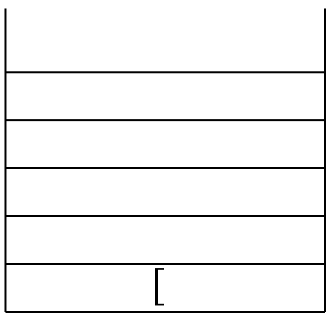
What is the complexity of the operation? **O(n)**

EXAMPLE

Check brace, bracket parentheses matching

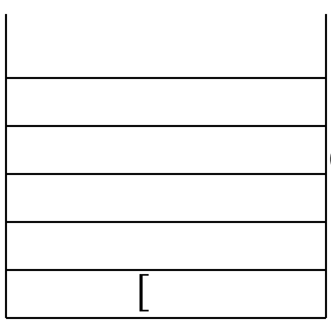
[a+b{1*2]9*1}+(2-1)

Push [, Push {, Pop, Pop, Push (, Pop



Push [

Push {



Get {

Pop Expect?

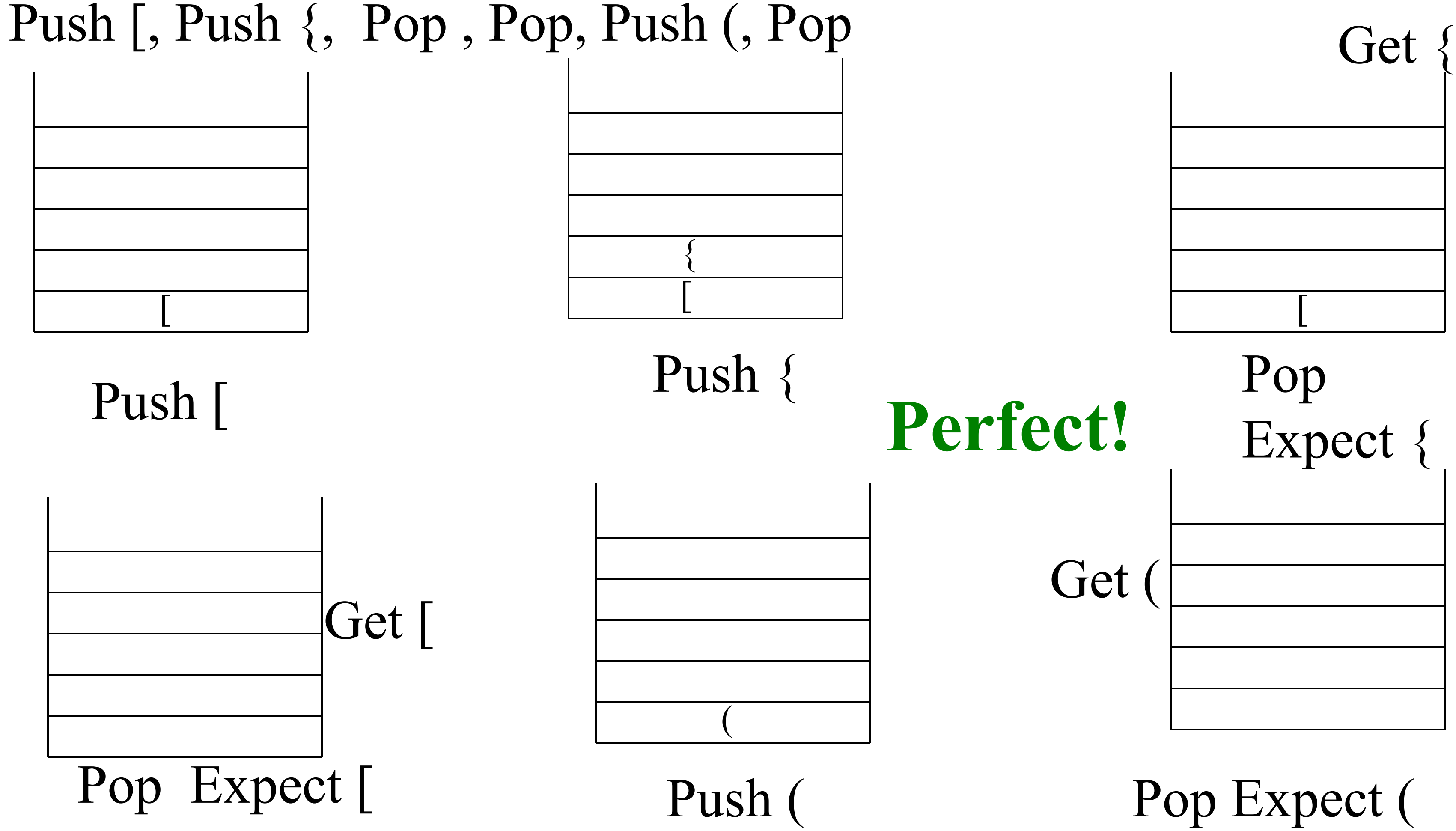
Oops! Something wrong, was expecting [

EXAMPLE

Check brace, bracket parentheses matching

[a+b{1*2}9*1]+(2-1)

Push [, Push {, Pop , Pop, Push (, Pop



Evaluation of arithmetic expressions

A*B + C

A*(B + C)

A+B+C*D

A*D+B+C*E

To do the correct operation, the calculator needs to know the priority of operands

We don't need any priority nor parentheses if the operation is expressed in postfix or reverse polish notation.

Postfix

A*B + C = AB*C+

A*(B + C) = A(B + C)* = ABC+*

A+B+C*D = AB+CD*+

A*D+B+C*E = AD*B+CE*+

Suppose the expression is in postfix, how do we compute the value? (N.B.: Postfix notation already takes into account operand priorities.)

Computation of a Postfix Expression

Whenever you see a number push it in the stack.

Whenever you see an operator,

- Pop 2 numbers,
- Apply the operator to the 2 numbers
- Push the result

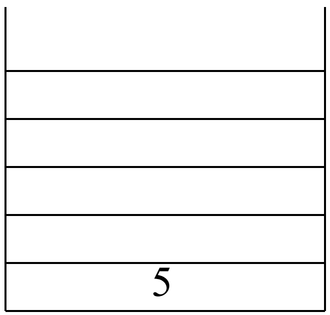
At the end Pop to get the answer

Complexity? **O(n)**

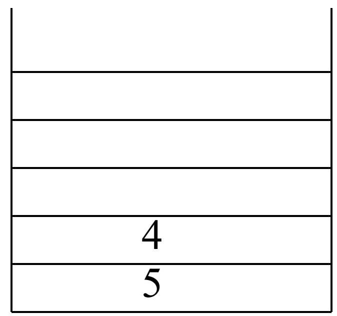
EXAMPLE

Compute 5*4+6+7*2 Result: 40

Postfix: 5 4*6+7 2*+



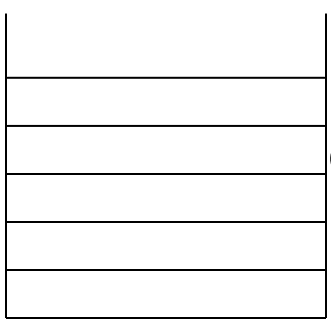
Push 5



Push 4

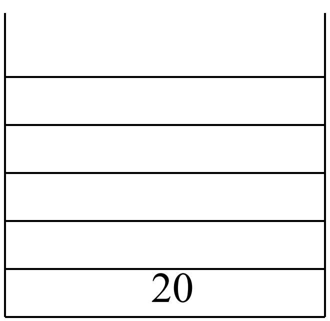
Get 4

Pop

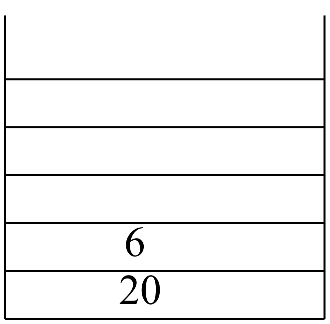


Get 5

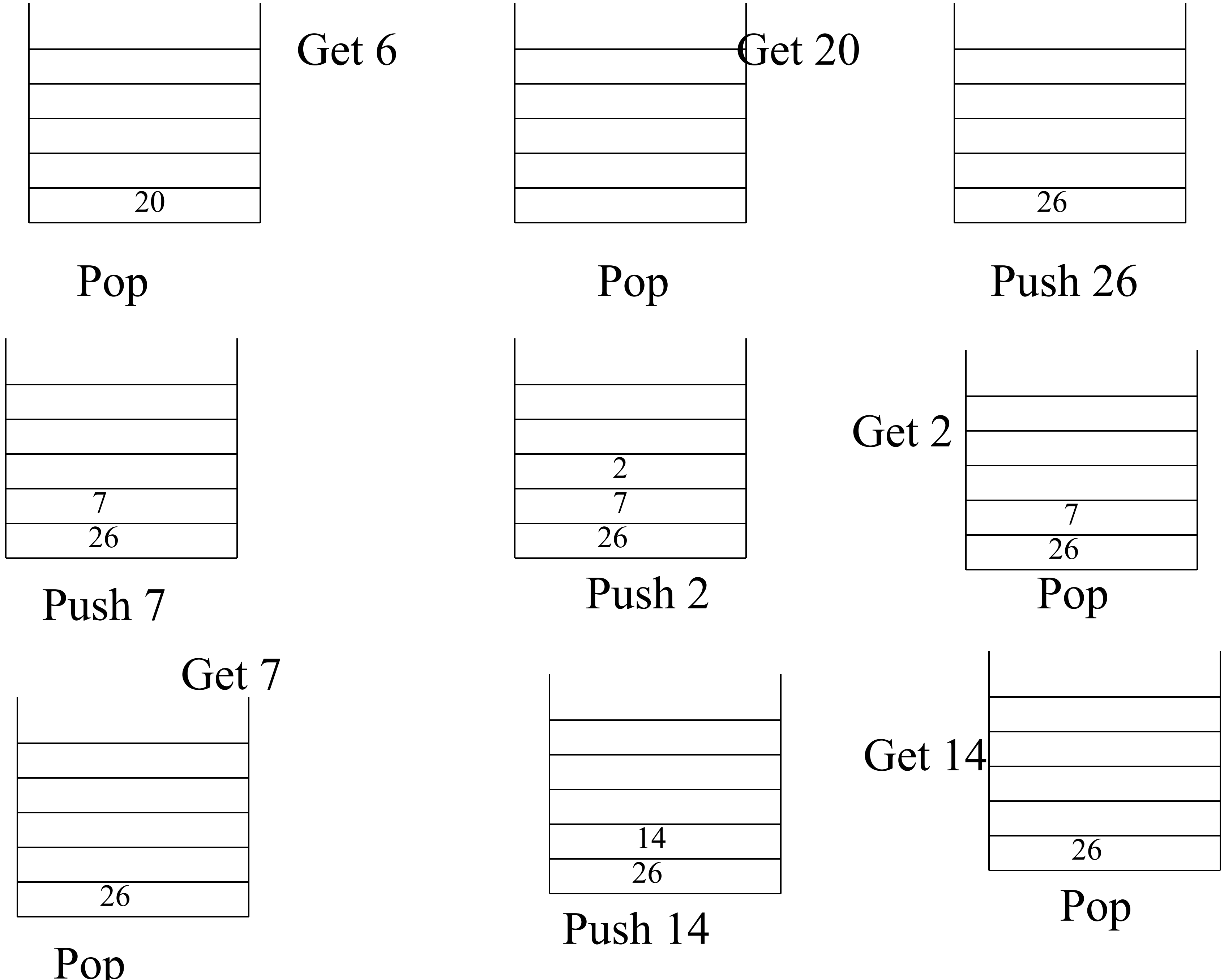
Pop



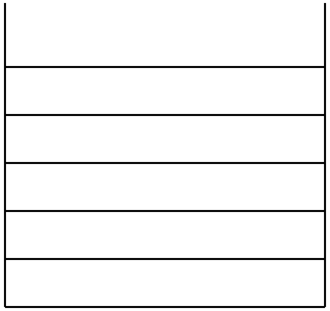
Push 20



Push 6

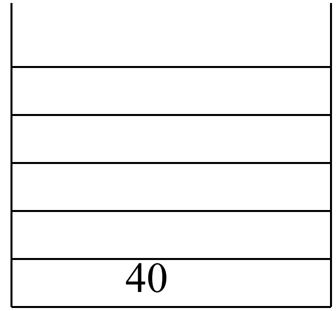


Get 6
20
Pop
Get 20
Pop
26
Push 26
7
26
Push 7
2
7
26
Push 2
Get 2
7
26
Pop
Get 7
26
Pop
14
26
Push 14
Get 14
26
Pop



Get 26

Pop

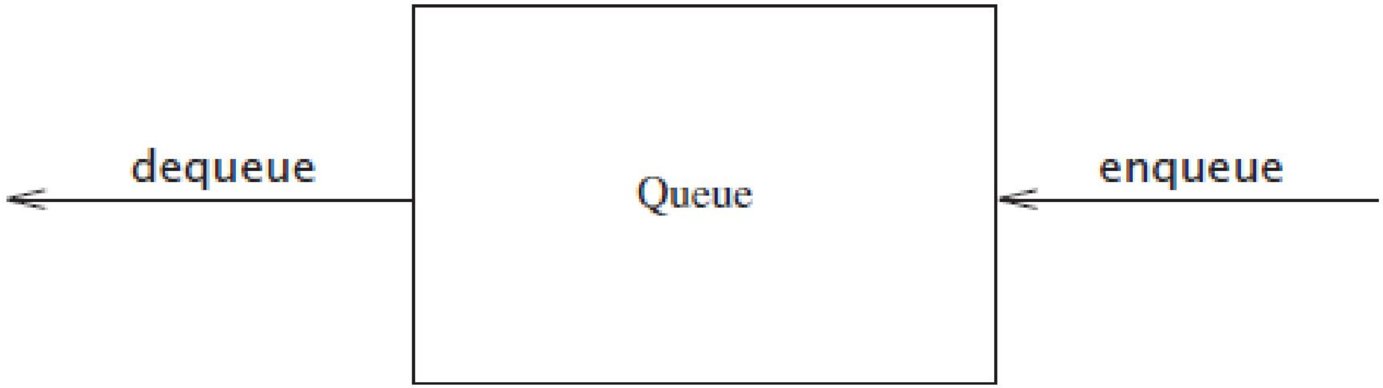


Push 40

Pop to get answer

Queues

- A Queue is a Linked List where an element is inserted at the end (back), and deleted from the beginning (front) of the list
- Thus insertion and deletion work at different ends
- This type of service is called First in First out



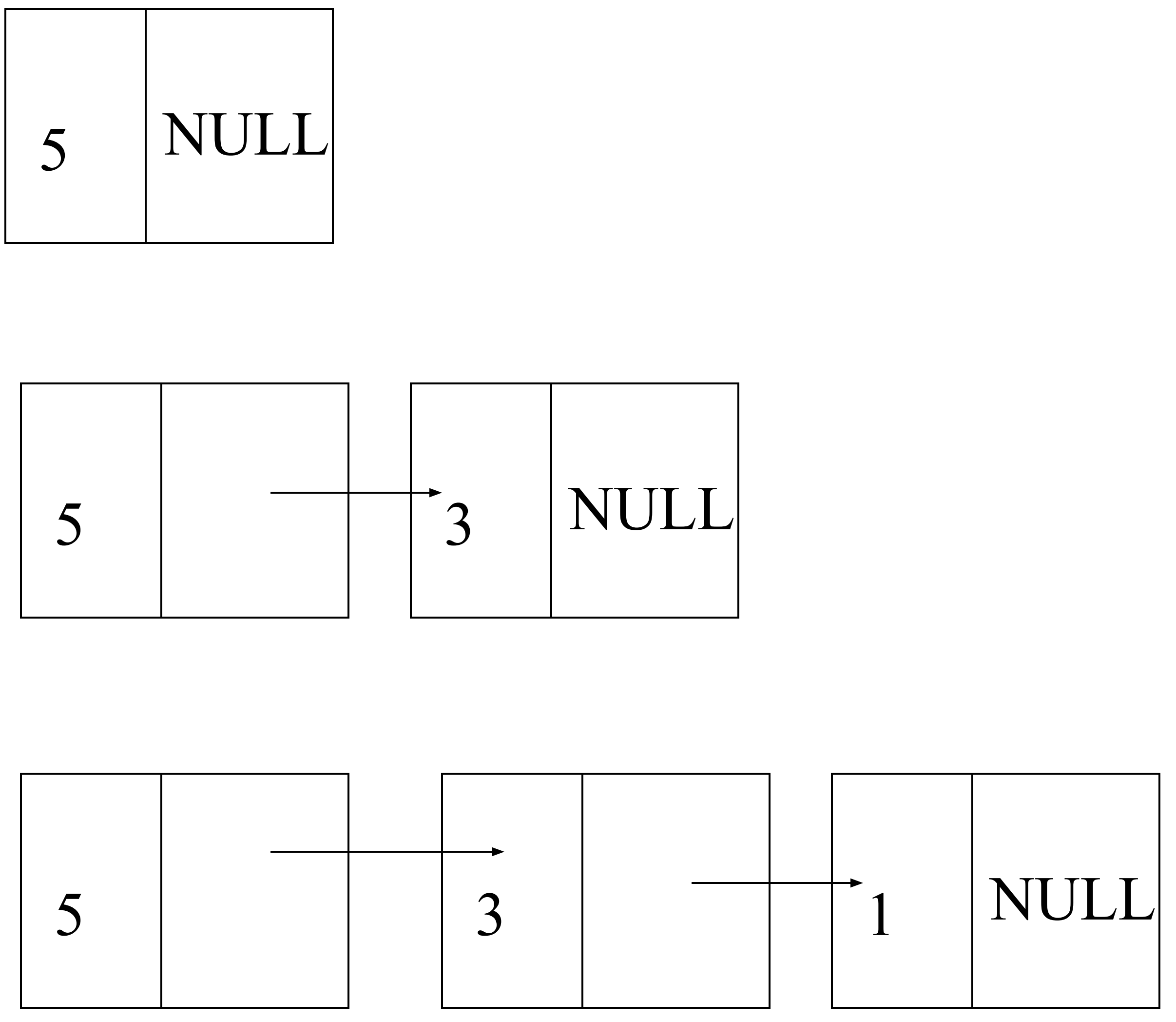
Deletion and Insertion must be **O(1)**

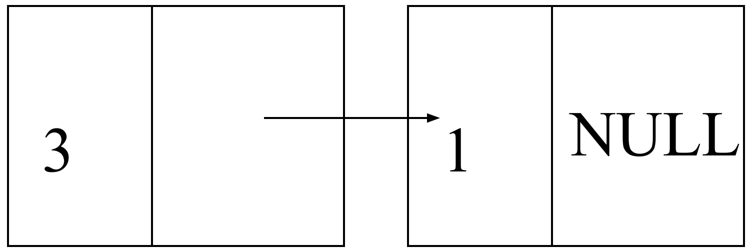
Linked List Implementation of a Queue

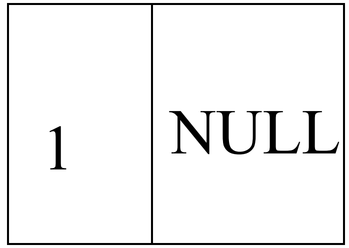
- Insert at the *Back* of the linked list
  - Maintain a pointer to the last element
  - Whenever there is insertion, update this pointer *Back* to point to the newly inserted element.
- Delete from the beginning of the linked list. Use the *Front* pointer

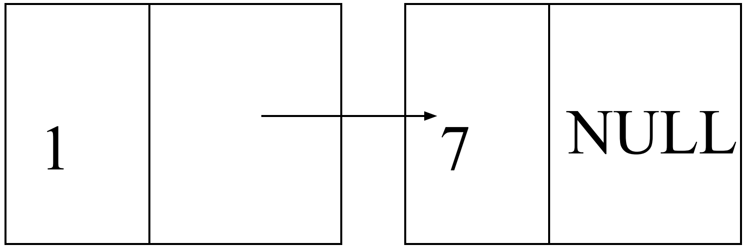
Both insertion and deletion are O(1)

Insert 5, 3, 1, Delete, Delete, Insert 7




3
1
NULL


1
NULL


1
7
NULL

Array Implementation of a Queue

Using arrays:

- Insertion at the back has complexity O(1)
- But deletion at the front has complexity O(n)

Can this be improved?

Solution: Use a circular array implementation

Implementing Queues using circular arrays

Maintain the length of the queue as a separate variable, not as the first element of the array (for convenience).

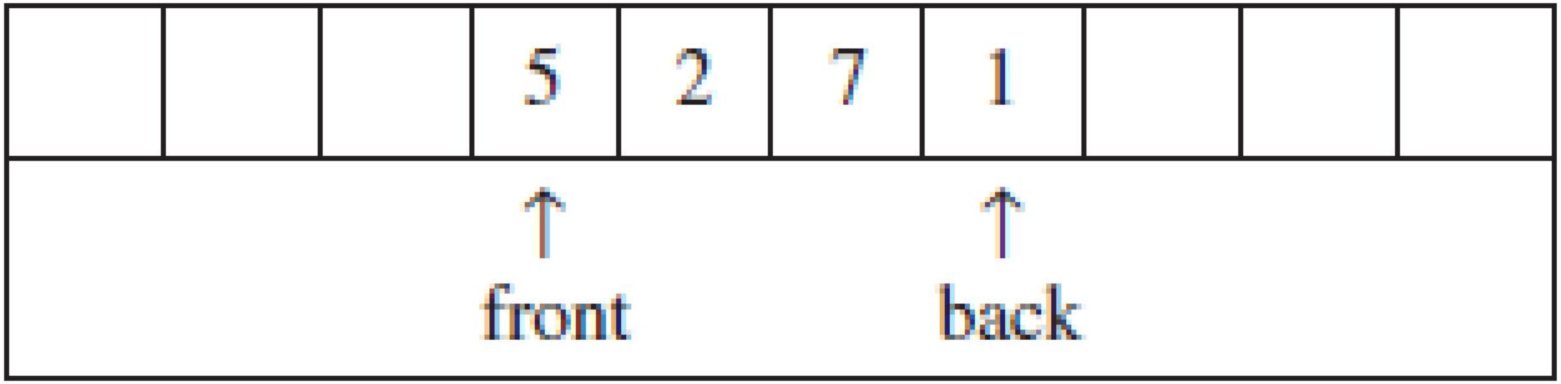
Start inserting from the beginning of the array.

- Insert at the end of the current list (*Back*)

When an element is deleted from the beginning (*Front*), DO NOT move all elements forward

- Just mark the position as blank.

Any problem?



We will soon reach the end of the array even though there are spaces in the beginning to insert in.

We roll back to the beginning.

When the last element is at the end of the array, we insert a new element at the beginning of the array

Implementation Routine

Maintain two positions:

Front, Back

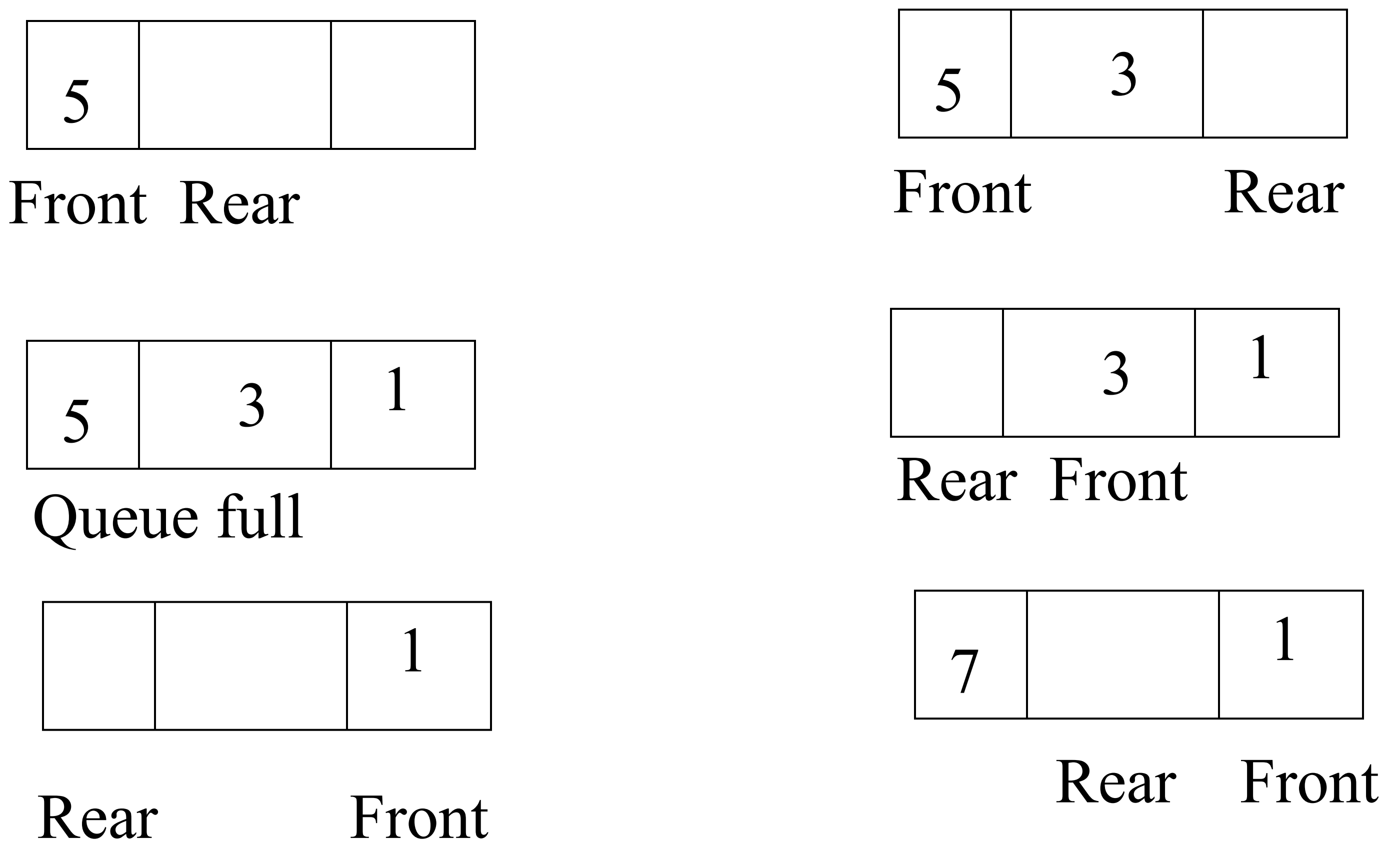
Maintain Queue Size

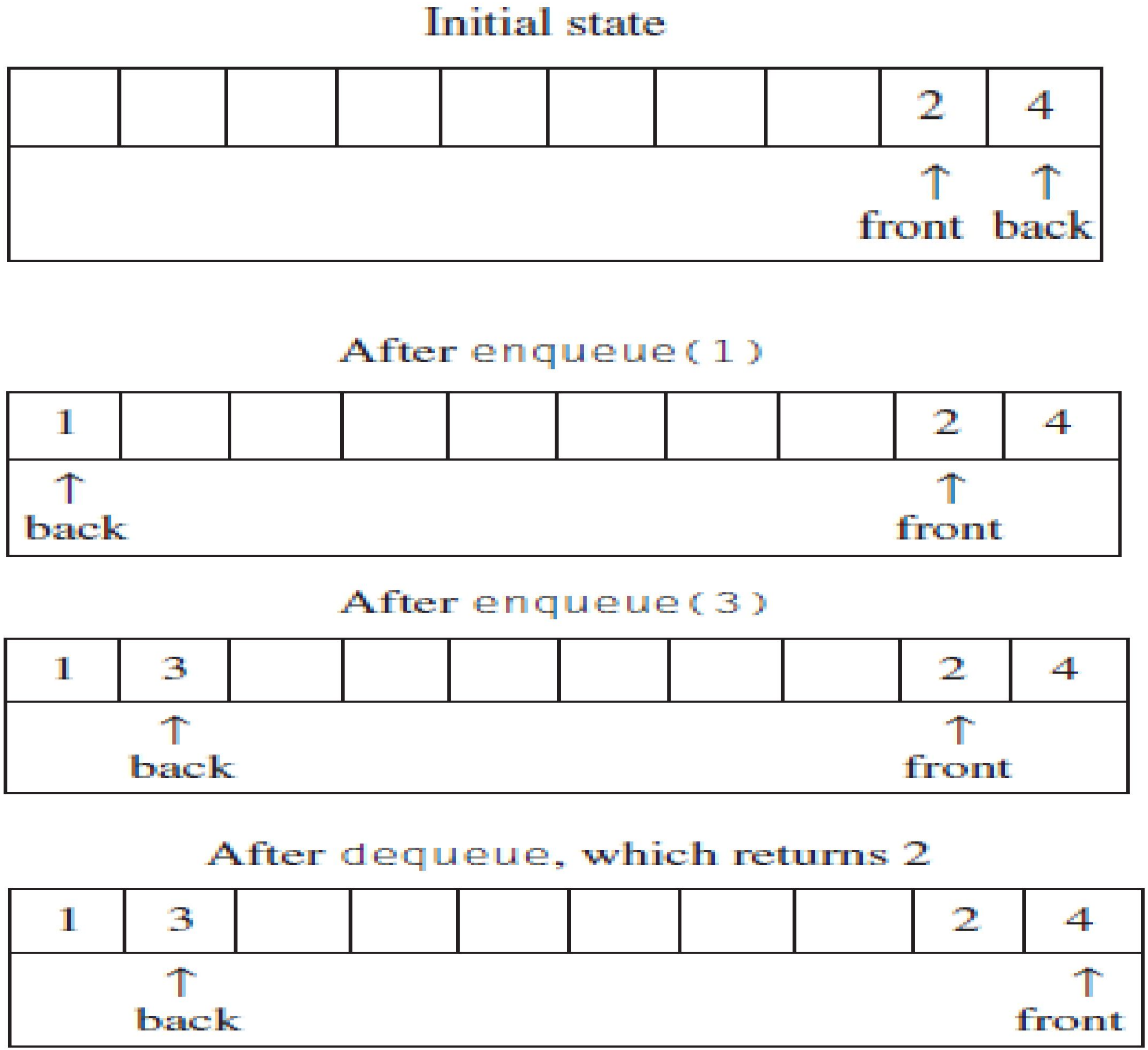
Initially, Front = 0, Back = 0, Queue Size = 0

EXAMPLE

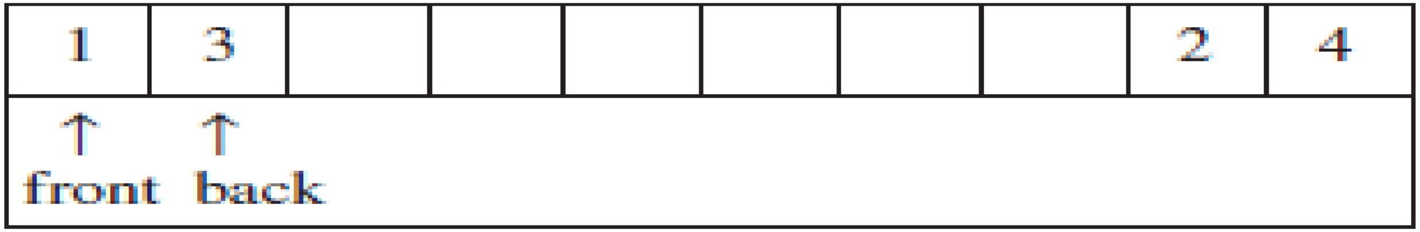
EXAMPLE

Insert 5, 3, 1, Delete, Delete, Insert 7

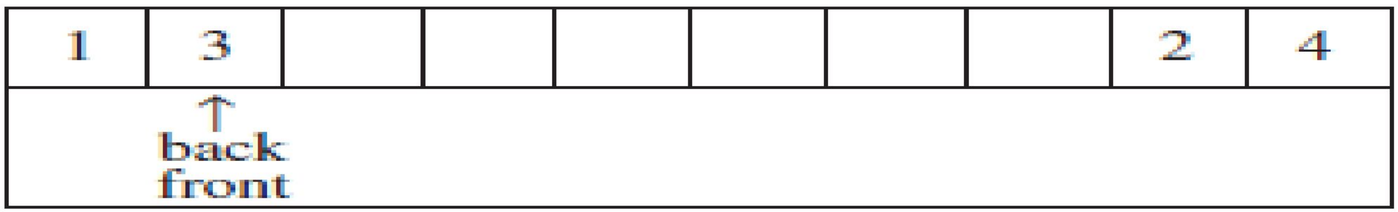



Initial state
2
4
front
back
After enqueue(1)
1
2
4
back
front
After enqueue(3)
1
3
2
4
back
front
After dequeue, which returns 2
1
3
2
4
back
front

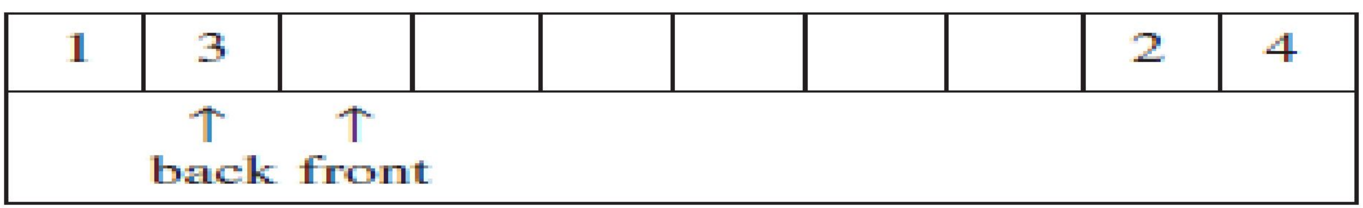
After `dequeue`, which returns 4



After `dequeue`, which returns 1



After `dequeue`, which returns 3
and makes the queue empty



Deletion

If Queue Size = 0, conclude empty queue,

 no deletion

Else

{

Decrement Queue Size;

Retrieve A[Front];

Change Front to (Front + 1) Modulo Array Size;

}

Insertion

If Queue Size = Array Size, conclude Full queue

 no insertion

Else

{

Increment Queue Size;

A[Back] = new element;

Change Back to (Back+1) Modulo Array Size;

}