# ANÁLISIS Y DISEÑO DE ALGORITMOS I
# Periodo I – 2023

Jesús Aranda

Universidad del Valle
Escuela de Ingeniería de Sistemas y Computación

Este documento es una adaptación del material original del profesor Oscar Bedoya

Pilas

Colas

Listas enlazadas

Listas doblemente enlazadas

Árboles

# Estructuras de datos

## Pila

Una pila es una estructura de datos tipo LIFO (Last In First Out), por lo que el último elemento insertado será el primero en ser borrado

Operaciones básicas:

STACK-EMPTY(S)

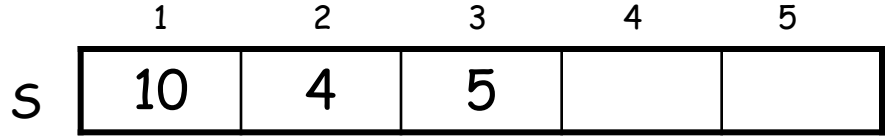
PUSH(S,x)

POP(S)

# Estructuras de datos

## Pila

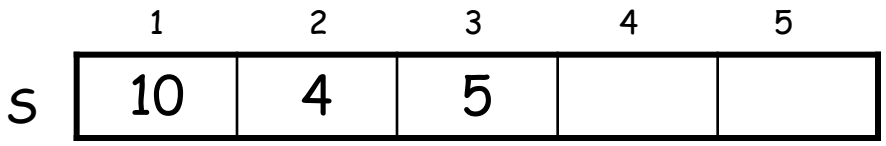
Una forma de implementar la pila es por medio de un arreglo unidimensional




Esto supone varios aspectos:

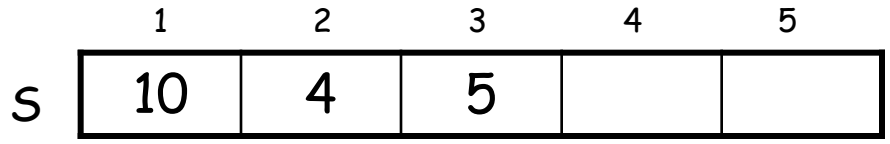
- La pila tiene una capacidad limitada
- Se cuenta con un atributo adicional, llamado top[S], que almacena el índice en el arreglo que guarda el último valor, esto es, el tope de la pila
- Cuando la pila esté vacía, top[S]=0

# Estructuras de datos




**top[S]=3**

# Estructuras de datos

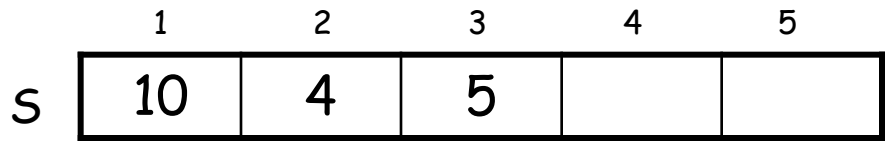



**top[S]=3**

Indique lo que sucede después de cada instrucción, siendo S la pila que se muestre arriba:

STACK-EMPTY(S)
PUSH(S,4)
PUSH(S,12)
PUSH(S,7)

# Estructuras de datos

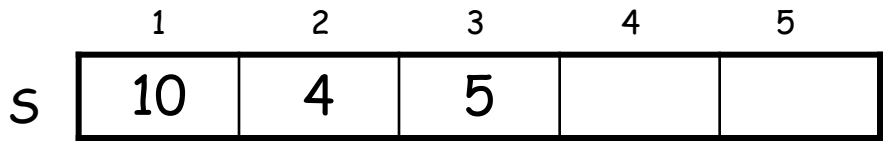



**top[S]=3**

Indique lo que sucede después de cada instrucción, siendo S la pila que se muestre arriba:

STACK-EMPTY(S)
PUSH(S,4)
PUSH(S,12)
PUSH(S,7) ← Overflow - desbordamiento en su capacidad máxima

# Estructuras de datos




**top[S]=3**

Indique lo que sucede después de cada instrucción, siendo S la pila que se muestre arriba:

STACK-EMPTY(S)
POP(S)
POP(S)
POP(S)
STACK-EMPTY(S)
POP(S) ← Underflow

# Estructuras de datos

Indique un algoritmo para cada operación básica, acompañado de su respectiva complejidad usando la notación O

- STACK-EMPTY(S), retorna true o false
- PUSH(S,x), adiciona x al tope, no devuelve ningún valor
- POP(S), borra el elemento que esté en el tope y devuelve ese valor

# Estructuras de datos

```
STACK-EMPTY(S)
1  if (top[S]==0)
2   then return true
3   else return false
```

# Estructuras de datos

```
STACK-EMPTY(S)
1  if (top[S]==0)
2   then return true
3   else return false
```

T(n)=O(1), tiempo constante

# Estructuras de datos

**PUSH(S,x)**

```
1   if S.top == S.size
2         error "overflow"
3   S.top = S.top + 1
4   S[S.top] = x
```

**POP**(S)

```
1   if STACK-EMPTY(S)
2          error "underflow"
3    else
4           S.top  = S.top -1
5           return S[S.top + 1]
```

**T(n)=O(1), tiempo constante**

# Estructuras de datos

## Cola

Una cola es una estructura de datos tipo FIFO (First In First Out), por lo que el primer elemento que es insertado, es el primero en ser borrado

Operaciones básicas:

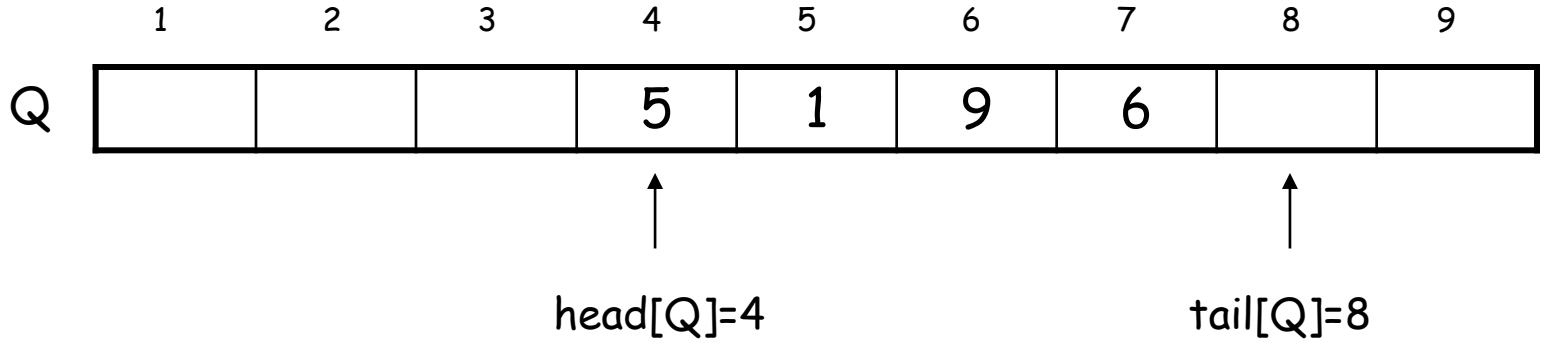
ENQUEUE(Q,x)

DEQUEUE(Q)

# Estructuras de datos

## Cola

Una forma de implementar la cola es por medio de un arreglo unidimensional



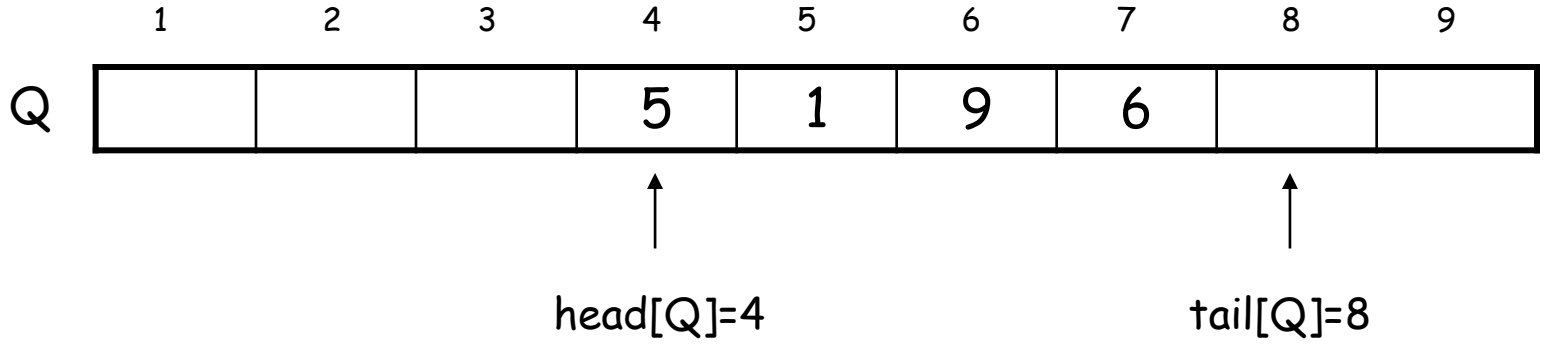

Esto supone varios aspectos:

- La cola tiene una capacidad limitada
- Se cuenta con dos atributos adicionales, head[Q] que guarda el indice de la cabeza y tail[Q] que apunta al siguiente lugar en el cual será insertado un elemento

# Estructuras de datos

## Cola

Una forma de implementar la cola es por medio de un arreglo unidimensional




ENQUEUE(Q,2)

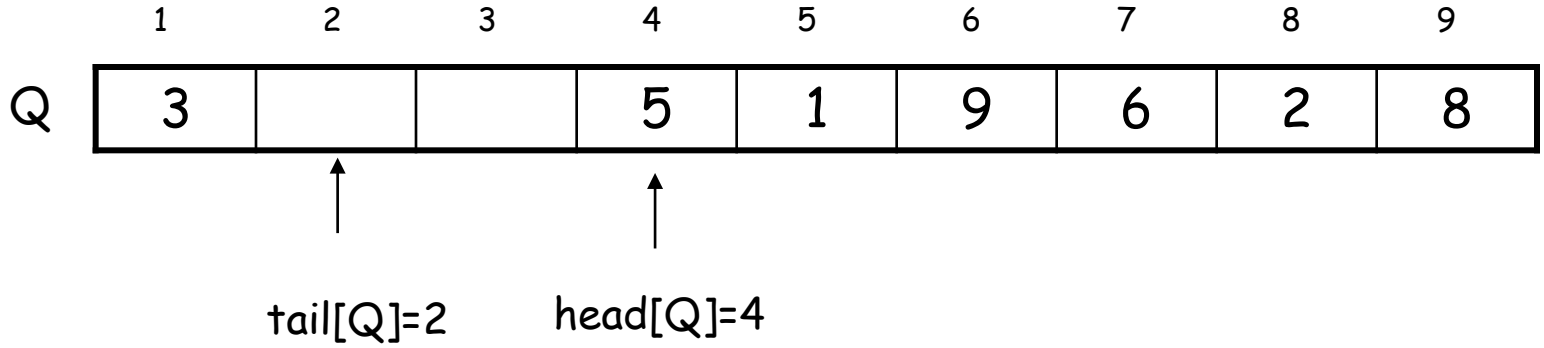
ENQUEUE(Q,8)

ENQUEUE(Q,3)

# Estructuras de datos

## Cola

Una forma de implementar la cola es por medio de un arreglo unidimensional




ENQUEUE(Q,2)

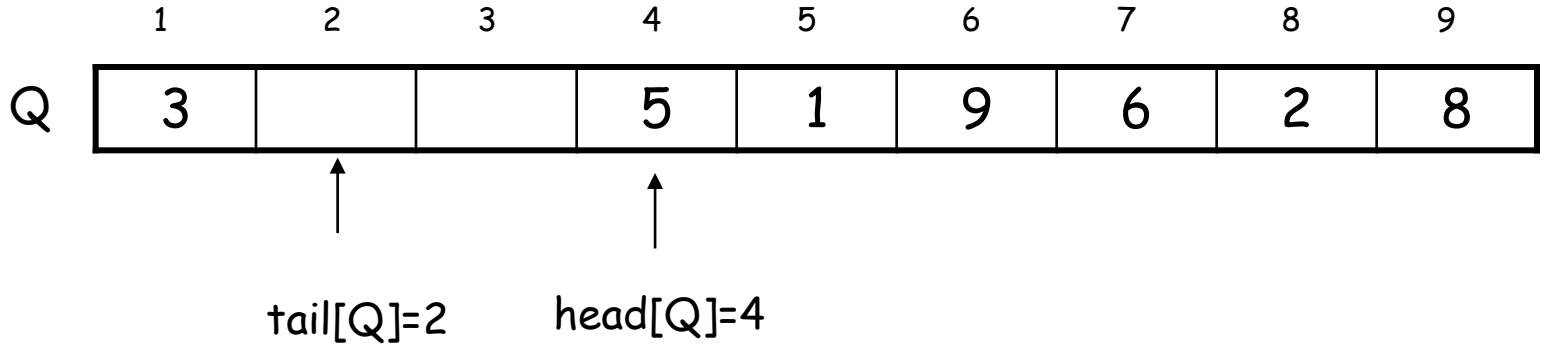
ENQUEUE(Q,8) ← Si se llega al final del arreglo, se intenta insertar en la posición 1

ENQUEUE(Q,3)

# Estructuras de datos

## Cola

Una forma de implementar la cola es por medio de un arreglo unidimensional

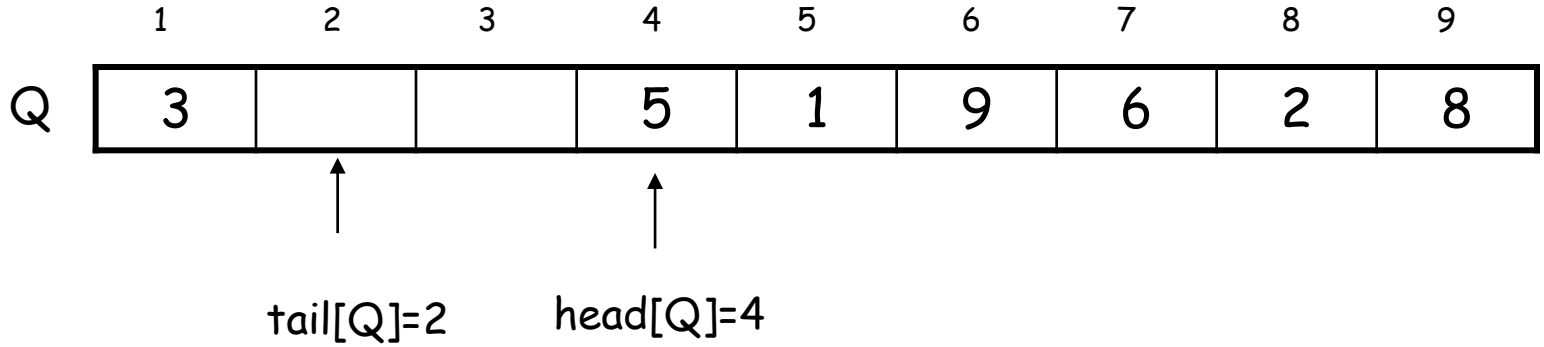



Cómo sabe que la cola está llena?

# Estructuras de datos

## Cola

Una forma de implementar la cola es por medio de un arreglo unidimensional



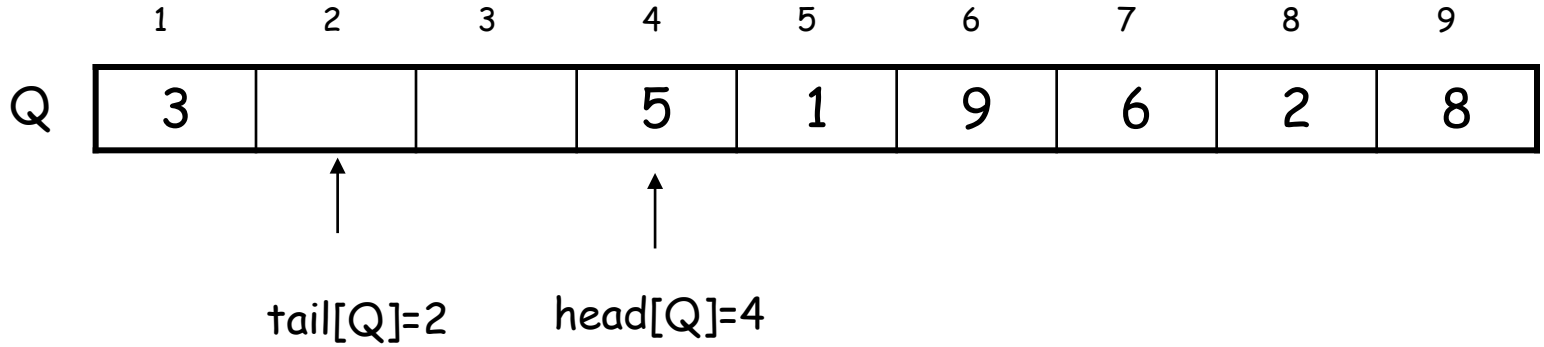

Cómo sabe que la cola está llena?

tail[Q] + 1 =head[Q]

# Estructuras de datos

## Cola

Una forma de implementar la cola es por medio de un arreglo unidimensional




Inicialmente tail[Q]=head[Q]=1

# Estructuras de datos

Indique un algoritmo para cada operación básica, acompañado de su respectiva complejidad usando la notación *O*

- ENQUEUE(Q,x)
- DEQUEUE(Q)

# Estructuras de datos

Indique un algoritmo para cada operación básica, acompañado de su respectiva complejidad usando la notación *O*

```
ENQUEUE (Q,x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3           Q.tail = 1
4  else  Q.tail = Q.tail + 1
```

T(n)=*O*(1), tiempo constante

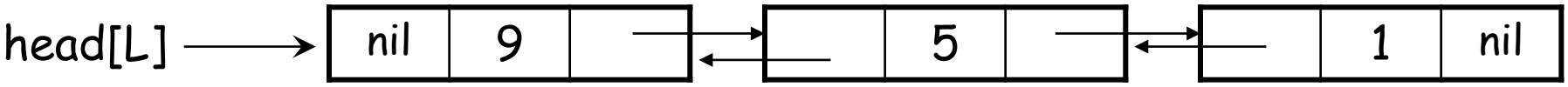
```
DEQUEUE (Q)
1   x = Q[Q.head]
2  if Q.head == Q.length
3           Q.head= 1
4  else  Q.head = Q.head + 1
5  return x
```

T(n)=*O*(1), tiempo constante

# Estructuras de datos

## Listas doblemente enlazadas

Es una estructura de datos en la cual los objetos son organizados en un orden lineal. A diferencia de los arreglos, el orden en las listas está dado por un puntero a cada objeto

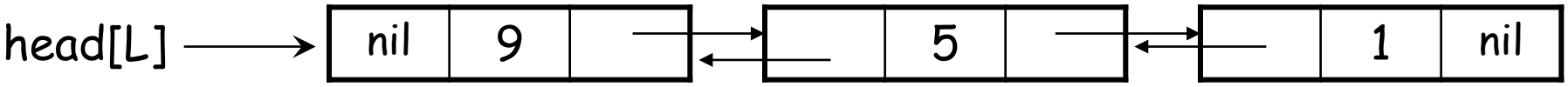



- Cada nodo en una lista doblemente enlazada tiene 3 campos: prev, key y next
- Se tiene además un puntero al primer nodo

# Estructuras de datos

## Listas doblemente enlazadas

Es una estructura de datos en la cual los objetos son organizados en un orden lineal. A diferencia de los arreglos, el orden en las listas está dado por un puntero a cada objeto




Operaciones

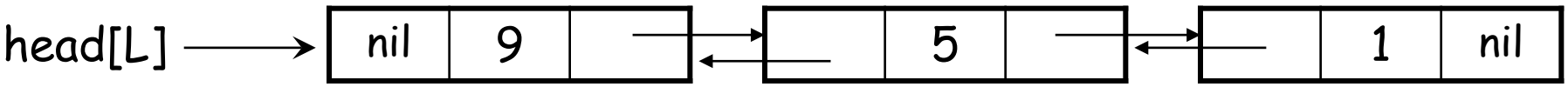
- LIST-INSERT(L,x): inserta x en la cabeza de la lista. x es un nodo tal que key[x]=k, y prev=next=nil
- LIST-DELETE(L,x): donde x es el nodo que se desea borrar
- LIST-SEARCH(L,k): busca el primer nodo que tiene llave k y retorna un puntero a ese nodo

# Estructuras de datos

*LIST-SEARCH busca el primer nodo que tiene llave k y retorna un puntero a ese nodo*

```
LIST-SEARCH(L,k)
1. x←head[L]
2. while x≠nil and key[x]≠k
3.         x←next[x]
4.  return x
```

¿Cuál es la complejidad en el peor caso?

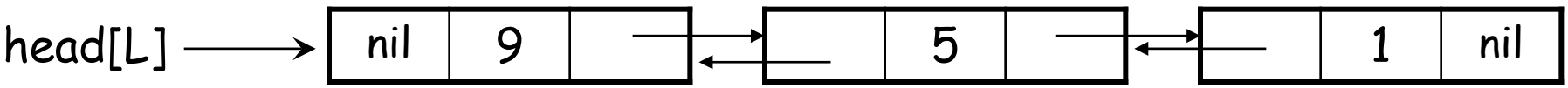


# Estructuras de datos

*LIST-SEARCH busca el primer nodo que tiene llave k y retorna un puntero a ese nodo*

```
LIST-SEARCH(L,k)
1. x←head[L]
2. while x≠nil and key[x]≠k
3.         x←next[x]
4.  return x
```

En el peor caso será O(n)



# Estructuras de datos

Indique el resultado de realizar las siguientes operaciones:

prev[z]=nil

next[z]=nil

key[z]=10

LIST-INSERT(L,z)

prev[w]=nil

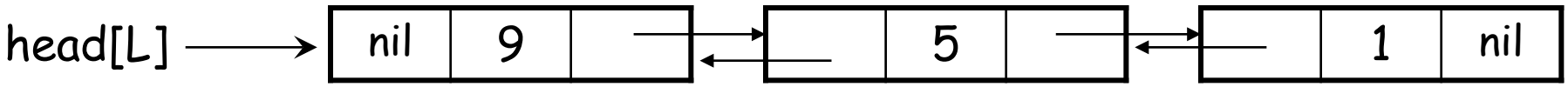
next[w]=nil

key[w]=8

LIST-INSERT(L,w)

x=LIST-SEARCH(L,10)

LIST-DELETE(L,x)



# Estructuras de datos

Indique el algoritmo para las siguiente operaciones y muestre su complejidad en el peor caso:

- LIST-INSERT(L,x): inserta x en la cabeza de la lista. x es un nodo tal que key[x]=k, y prev=next=nil

- LIST-DELETE(L,x): donde x es el nodo que se desea borrar

# Estructuras de datos

Indique el algoritmo para las siguiente operaciones y muestre su complejidad en el peor caso:

• LIST-INSERT(L,x): inserta x en la cabeza de la lista. x es un nodo tal que key[x]=k, y prev=next=nil

• LIST-DELETE(L,x): donde x es el nodo que se desea borrar

LIST-INSERT(L, x)

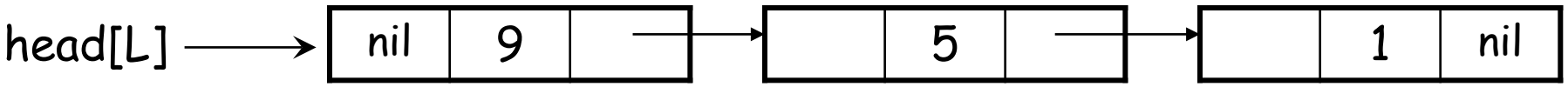
1. x.next ← L.head
2. if L.head ≠ NIL
3. L.head.prev ← x
4. L.head ← x
5. x.prev ← NIL

LIST-DELETE(L, x)

1. if x.prev ≠ NIL
2. x.prev.next ← x.next
3. else L.head ← x.next
4. if x.next ≠ NIL
5. x.next.prev ← x.prev

# Estructuras de datos

## Listas simplemente enlazada




Operaciones

- LIST-INSERT(L,x): inserta x al final de la lista. x es un nodo tal que key[x]=k, y prev=next=nil
- LIST-DELETE(L): donde x es el nodo al final de la lista
- LIST-SEARCH(L,k): busca el primer nodo que tiene llave k y retorna un puntero a ese nodo

# Estructuras de datos

## Árboles

Cada nodo tiene los campos p, left y right para almacenar los punteros al padre, hijo izquierdo e hijo derecho. Además, se tiene el campo key.

Si p[x]=nil entonces x es raíz

Si el nodo x no tiene hijo izquierdo entonces left[x]=nil

Si left[x]=right[x]=nil entonces x es una hoja

# Estructuras de datos

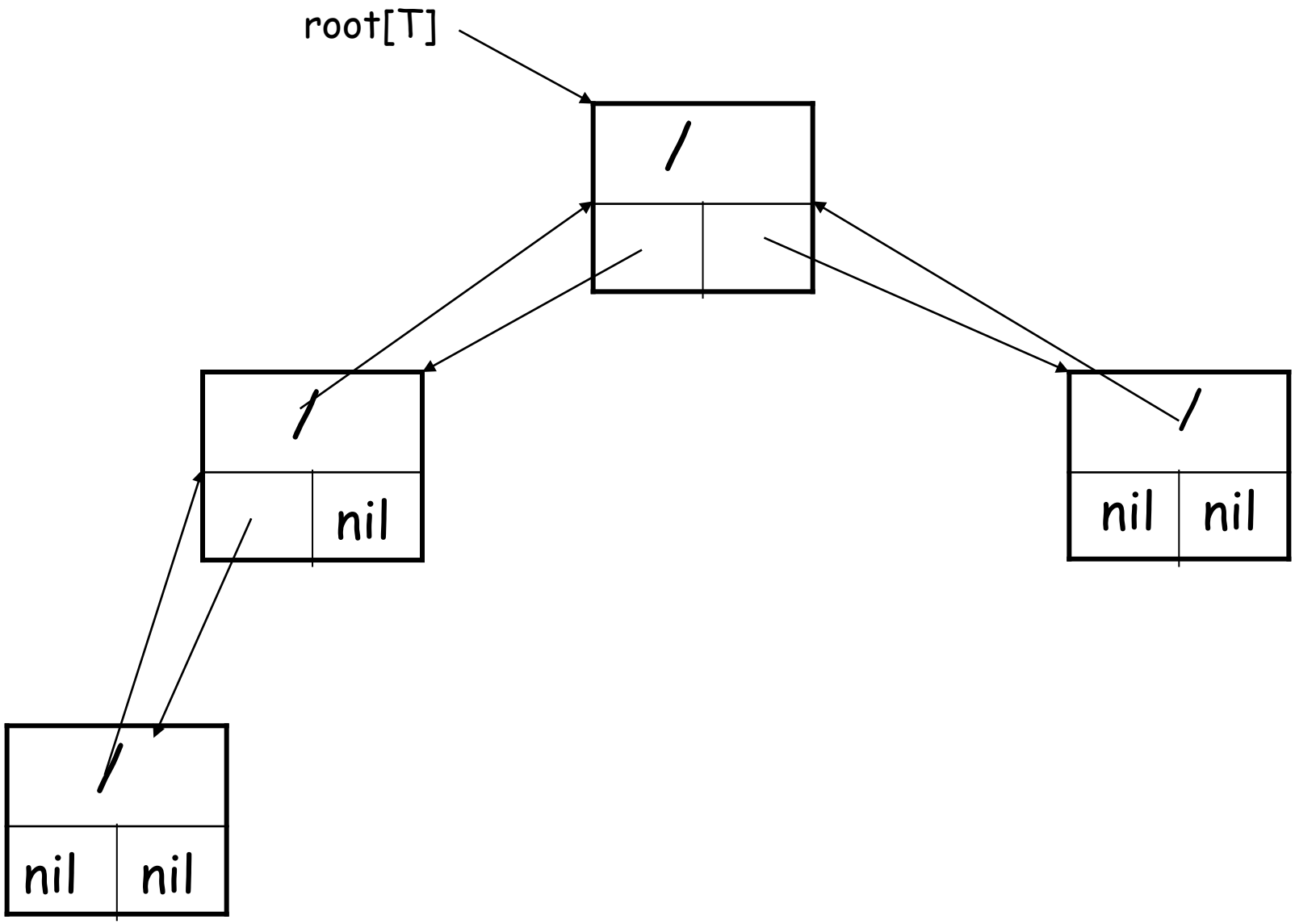
## Árboles

Cada nodo tiene los campos p, left y right para almacenar los punteros al padre, hijo izquierdo e hijo derecho. Además, se tiene el campo key.

root[T] es el apuntador a la raíz del árbol

Si root[T]=nil entonces el árbol está vacío

# Estructuras de datos



# Estructuras de datos

Desarrolle un algoritmo que permita imprimir las llaves de un árbol binario, tal que en el peor caso sea O(n)