

SEPARATAS DE CURSO

ESTRUCTURA DE DATOS

ESIS - FAIN – UNJBG

AUTOR: ING. ISRAEL NAZARETH CHAPARRO CRUZ

Presentación

La presente separata, pese a su corta extensión de 23 hojas, constituye el material desarrollado durante casi 50 horas lectivas del curso de Estructura de Datos turno mañana en el semestre académico 2022-I. Su artesanal elaboración es un proceso paso a paso en vivo durante clase, desarrollando los algoritmos que permiten implementar las principales operaciones en las estructuras de datos: Arreglos, Matrices, Listas, Pilas, Colas y Árboles Binarios. Se han descrito casi todas las estructuras de datos utilizando punteros y memoria dinámica (características del lenguaje de programación C++) excepto las colas, cuya implementación ha sido realizada utilizando arreglos, debido a que: 1) implementarlas usando punteros y memoria dinámica sería lo mismo que implementar una lista enlazada 2) para entrenar al estudiante en la implementación de estas estructuras en lenguajes de programación que no permiten el uso de punteros. El material bibliográfico en referencia ha sido el libro *Introduction to Algorithms* de *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein* considerado la biblia de los algoritmos. Sin embargo muchos supuestos difieren del material en referencia debido a que fueron adaptados para ser más didácticos. La potencialidad de la presente separata es contar con un material que enlaza algoritmos con casos de estudio en las operaciones de las estructuras de datos, para esto, el color es usado para enlazar ambas partes. Una versión digital puede ser encontrada en <https://github.com/ichaparroc/IA-UNJBG-2022/blob/main/README.md>

El Autor

Punteros: C++ ← potencial, diferencia

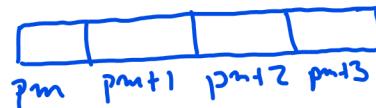
- Señalar cosas ↑.

- C++ POC todo es un objeto.

Punteros C++: Señalar objetos. (Def 1)

Objetos > 1 byte → posiciones correlativas

int = 4 bytes



cout << "Plataforma de " << sizeof(int)*8 << "bits";

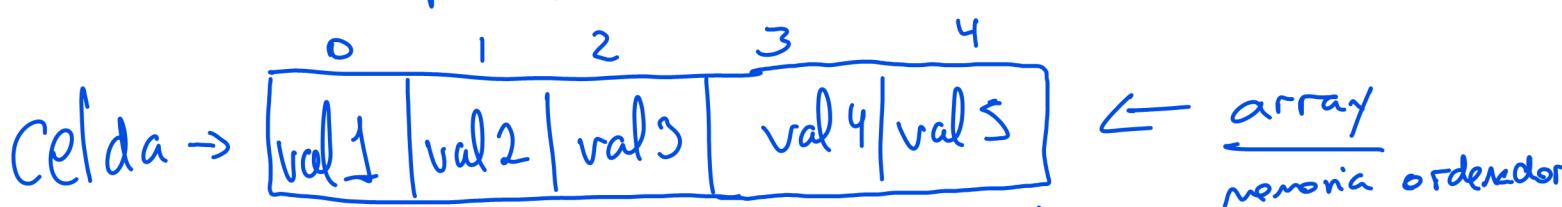
cout << "Plataforma de " << sizeof(int)*8 << "bits";

Puntero: tipo especial de objeto que contiene una dirección de memoria.

→ char, int, float, array ← definido al declarar

(Def 2).

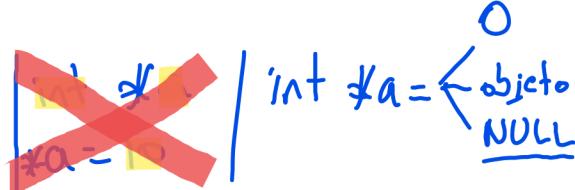
tipo de objeto.



índice = 0

celda[0] = celda[índice]

Valor Inicial : → int *pInt
char *pChar
struct stPunto *pPunto | int a.
a=10



int a= 123 → dirección
int a= dirección → dirección

Declaración: <tipo> *<identificador> | int *x, y;

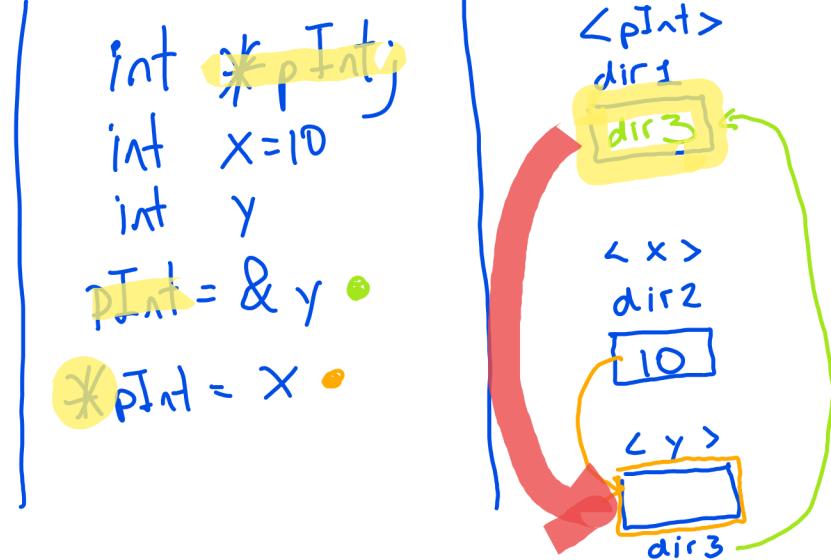
int *x, y; → puntero int

Dirección Objeto:



Objeto Apuntado por objeto

* → declarar (int *pA)
* → modificar (*a = 10)
* → dirección



Arreglos y Punteros:

int array[10] :

- Declara puntero constante del mismo tipo que los elementos.
- Reserva memoria + elementos del arreglo reservas consecutivas.
- Se inicializa apuntando al primer elemento.
- Compilador asocia a los elementos [1]

Asignación

```

int *p, *q;
int aj;
q = &a;
p = q;

```

①

Aritmetica:

```

int vector[10];
int *p, *q;
p = vector;
q = &vector[4];
cout << q - p << endl;

```

② 4

Comparaciones:

```

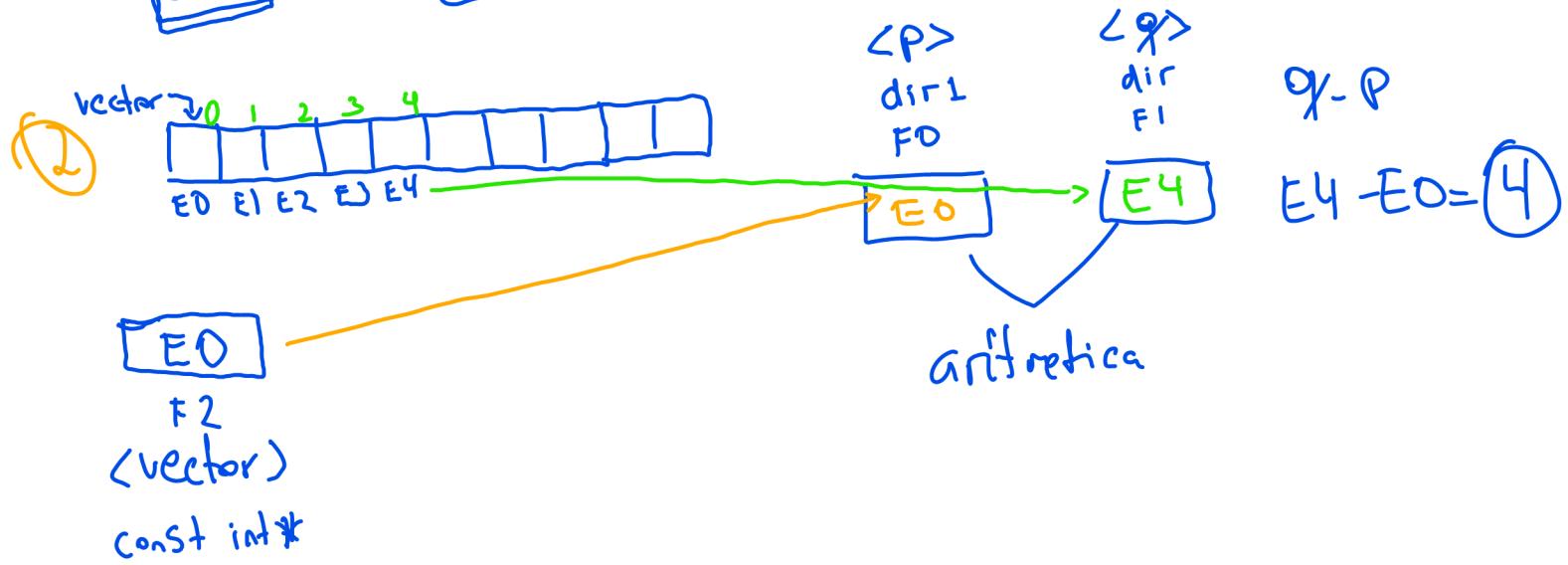
< ≤ ≥ > == !=
if(p != NULL)
if(p)
for (int i=NULL; (float*)NULL;

```

③



0xE0
0xE1
hex



Punteros Genericos:

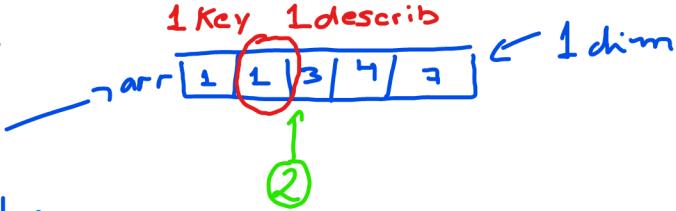
```

void *v;
int a
v=&a
cout << (int*)v;
float b
v=&b
cout << *(float*)v;
    
```

< v >
dirx

Estructuras de Datos Elementales:

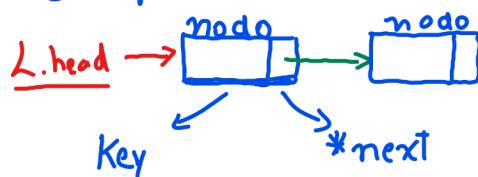
- Edd simples utilizan punteros.
- Listas enlazadas, Pilas, Colas, Árboles.



1o ~ Listas Enlazadas: Dinamica

- Array → Orden x indice.
- Lista → Orden x puntero.

- Simplemente Enlazadas, Dblemente Enlazadas.

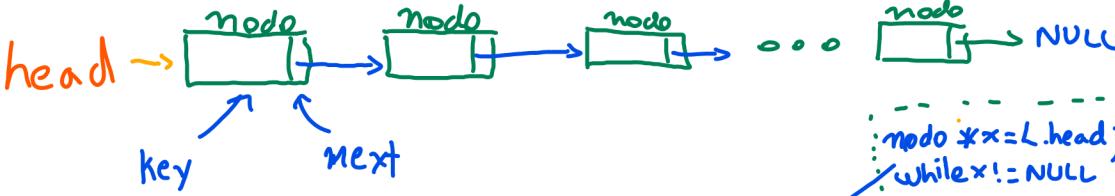


Key
↓
forma identif modo.

- Una lista puede tener varias formas, ordenadas o desordenadas, circulares o no.

- Ordenada: head = min
tail = max
- Circular: tail.next = head

1o1o ~ Simplemente Enlazada.



head apunta a la cabeza
NULL sig. no hay nodo.

Struct modo
{
int key;
nodo *next;};

```
class lista_simple
{
public:
    nodo *head;
    void imprimir();
}
```

head → NULL
x

head → [1] → [2] → [3] ...
x → x.next

1,2,3
head → [1] → [2] ... → [100] → NULL
x → x.next

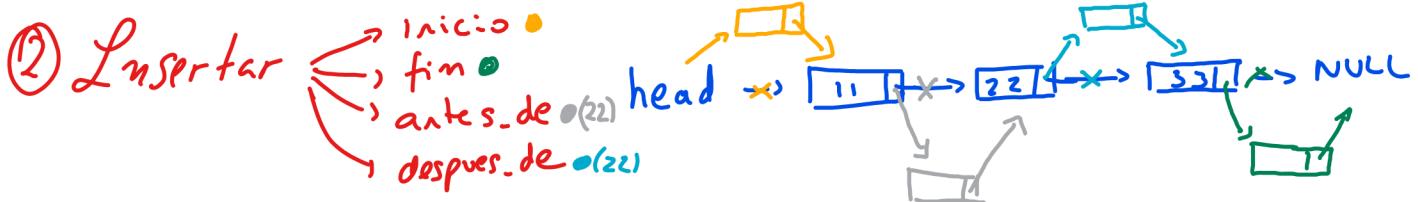
... → 1,2,...,100
x → x.next

① imprimir:
L.head →
x → x.next → x → ... → NULL
L.head->key
L.head->next
L.head-L.head->next

a) No existan elementos o nodos.
L.head → NULL
x = NULL

b) Existen elementos. x apunta a un nodo
cout << x.key << endl;
x ≠ NULL

c) Ya he recorrido y x.next apunta a NULL.
x = x.next
x = NULL



a) inserta_inicio (k) →

- Lista está vacía.

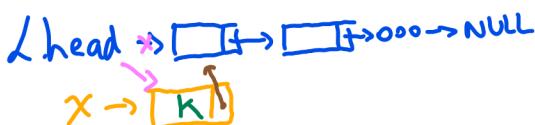
```

nodo *x = new nodo;
x.key = k;
x.next = L.head;
L.head = x;

```



- Lista tiene elementos ✓

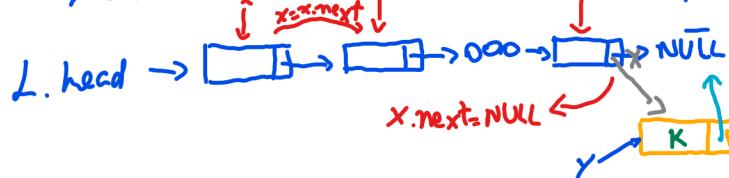


b) Insertar_fin (k) →

- Lista está vacía

L.head → NULL

- Lista tiene elementos



c) Eliminar (k)

elimina primera aparición

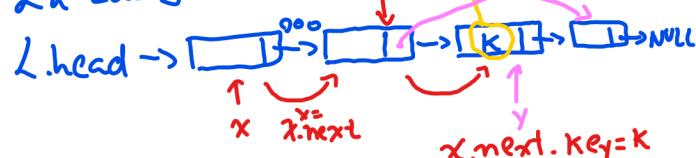
- Lista vacía

L.head → NULL ✓

- La cabeza es K



- La cabeza no es K



```

if (L.head == NULL)
    L.inserta_inicio(k)
else
    nodo *x = L.head
    while (x.next != NULL)
        x = x.next
    nodo *y = new nodo
    y.key = k
    y.next = x.next
    x.next = y

```

```

if (L.head != NULL)
{
    nodo *x = L.head
    if (x.key == k)
    {
        L.head = x.next
        delete x
    }
    else
    {
        if (!esta_k(x))
            while (x.next != NULL & x.next.key != k)
                x = x.next
        if (x.next != NULL)
            {
                nodo *y = x.next
                x.next = x.next.next
                delete y
            }
    }
}

```

no está k

Encontré k

d) buscar_key(k)

$L.\text{head} \rightarrow \text{NULL}$

$L.\text{head} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \text{NULL}$

P: $x == \text{NULL}$ (ya no hay nodos) \rightarrow ya no buscar
 q: $x.\text{key} == k$ (ya lo encontré) \rightarrow ya no buscar

$p \vee q$

```

nodo *x=L.head
while(x!=NULL & x.key!=k)
    x=x.next
    if(x==NULL)
        return 1
    else
        return 0
    }
}

```

buscar
 \downarrow
 $\sim(p \vee q)$
 $\sim p \wedge \sim q$

2.~ Listas Circulares

Son listas enlazadas cuya propiedad es que el último elemento está enlazado al primero.

$Lc.\text{head} \rightarrow \boxed{\quad} \leftarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \text{NULL}$

Simple link
doble link

Tener cuidado
con los
BUCLAS.

a) buscar_key(int k)

- lista vacía $\rightarrow \text{NULL}$

ningún enlace es NULL

- Existe elemento

$Lc.\text{head} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \dots \rightarrow \boxed{k} \rightarrow \dots$

- No existe el elemento

$Lc.\text{head} \rightarrow \boxed{\quad} \rightarrow \dots \rightarrow \boxed{\quad}$

- Dos elementos

$Lc.\text{head} \rightarrow \boxed{\quad} \rightarrow \boxed{\quad} \rightarrow \dots$

```

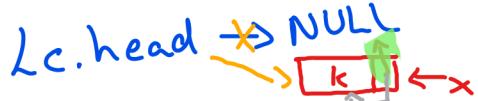
nodo *x=Lc.head
if(x==NULL)
    return 0;
else
{
    while(x->next!=Lc.head & x.key!=k)
        x=x->next
    if(x.key==k)
        return 1
    else
        return 0
}

```

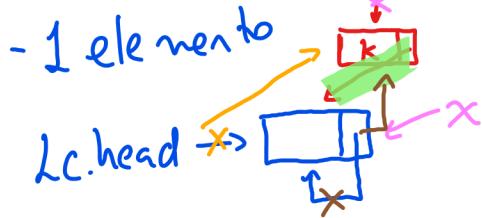
- P: $x.\text{next} == \text{Lc.head}$ (estoy en el último elemento)
- Q: $x.\text{key} == k$ (encontré el elemento)

b) `inserta_inicio(k)`

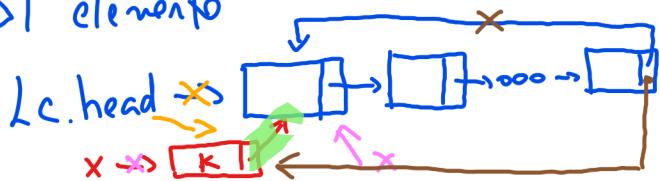
- Lista vacía



- Lista no vacía



-> 1 elemento



```

nodo *x = new nodo •
x.Key = k •
x.next = Lc.head •
Lc.head = x •
if (x.next == NULL)
    x.next = x •

else
    x = x.next •
    while (x.next != Lc.head.next)
        x = x.next
    x.next = Lc.head •

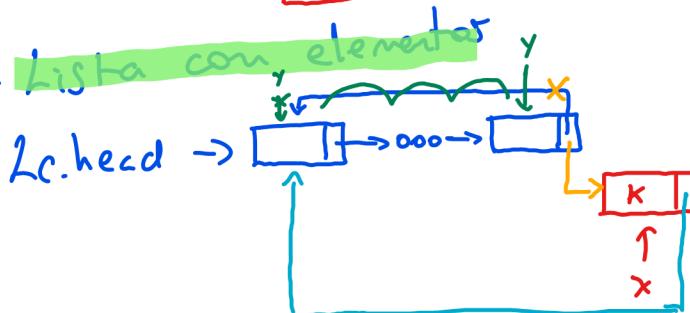
```

c) Insertar - fin (k)

- Listas vacías



- Lista com elementos



```

node *x = new node •
x.key = k •
if (Lc.head == NULL)
{
    Lc.head = x •
    x.next = x •
}
else
{
    node *y = Lc.head
    while (y.next != Lc.head)
        y = y.next
    y.next = x •
    x.next = Lc.head •
}

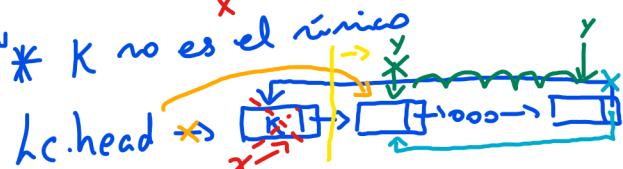
```

d) eliminar_key(k)

- Lista está vacía
 $Lc.head \rightarrow NULL$

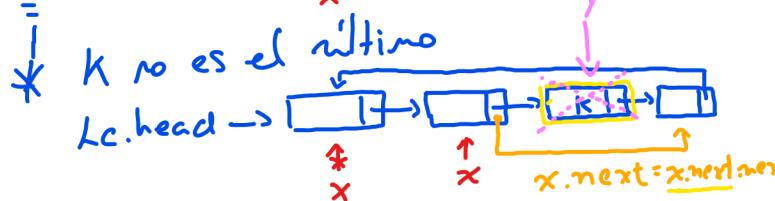
- K está al inicio

* K es el único
 $Lc.head \rightarrow [K] \rightarrow Lc.head \rightarrow NULL$



- K no está al inicio

* K es el último
 $Lc.head \rightarrow [] \rightarrow [] \rightarrow [K] \rightarrow NULL$



* K no es el último
 $Lc.head \rightarrow [] \rightarrow [] \rightarrow [K] \rightarrow []$
 $x.next = x.next.next$

```

if(Lc.head != NULL)
{
    nodo *x = Lc.head
    if(Lc.head.key == k)
    {
        if(Lc.head.next == Lc.head)
        {
            Lc.head = NULL
            delete x
        }
        else
        {
            Lc.head = Lc.head.next
            nodo *y = Lc.head
            while(y.next != x)
                y = y.next
            y.next = Lc.head
            delete x
        }
    }
    else
    {
        while(x.next != Lc.head &
              x.next.key != k)
            x = x.next
        if(x.next != Lc.head)
        {
            nodo *y = x.next
            x.next = x.next.next
            delete y
        }
    }
}

```

3~ Lista doblemente enlazada:

Pueden ser recorridas en ambas direcciones debido a que cada nodo contiene punteros "prev" y "next", adicionalmente la lista mantiene punteros en la cabeza ("head") y cola ("tail").

$Ld.head \rightarrow \text{NULL} \leftarrow Ld.tail$ \Leftarrow Lista doble vacía

$Ld.head \rightarrow \boxed{\text{K}}$ $\leftarrow Ld.tail$ \Leftarrow Lista doble con 1 nodo

$Ld.head \rightarrow \boxed{\text{K}} \rightarrow \boxed{\text{L}} \rightarrow \boxed{\text{M}} \rightarrow \boxed{\text{N}} \rightarrow \boxed{\text{O}} \rightarrow \boxed{\text{P}} \leftarrow Ld.tail$ \Leftarrow Lista doble con n nodos

Struct nodo_ld

```
{
    int Key;
    nodo_ld *prev;
    nodo_ld *next;
}
```

```
class lista_ld
{
public:    this->head <->
            nodo_ld *head;
            nodo_ld *tail;
            this->tail <->
            void inserta_inicio(int K);
}
```

"lista_ld.h"

\hookrightarrow struct nodo_ld
 \hookrightarrow class lista_ld

a) insertar_fin(K)

Lista vacía

$Ld.head \rightarrow \text{NULL} \leftarrow Ld.tail$

Lista con elementos

$Ld.head \rightarrow \boxed{\text{K}} \rightarrow \boxed{\text{L}} \rightarrow \boxed{\text{M}} \rightarrow \boxed{\text{N}} \rightarrow \boxed{\text{O}} \rightarrow \boxed{\text{P}} \leftarrow Ld.tail$

```
nodo_ld *x = new nodo_ld;
x.key = K;
x.next = NULL;
x.prev = Ld.tail;
Ld.tail = x;
if(Ld.tail.prev != NULL)
    Ld.tail.prev.next = x;
else
    Ld.head = x;
```

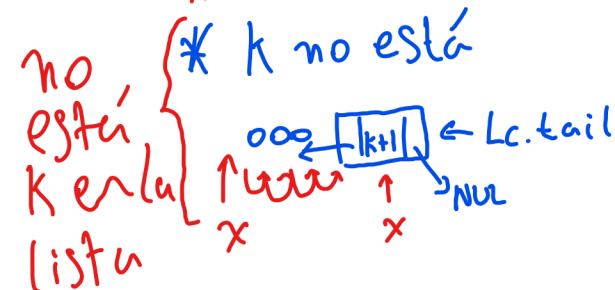
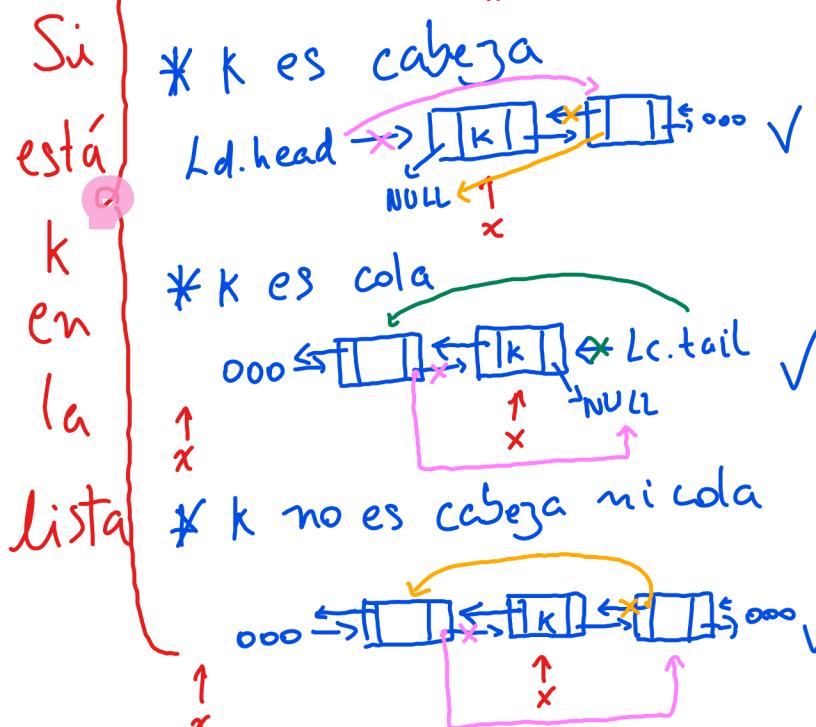
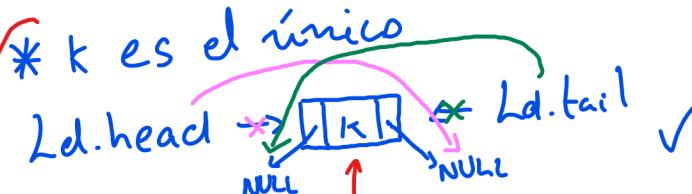
b) buscar_key(k) Este algoritmo puede ser idéntico al buscar_key(k) de una lista simple.

c) eliminar_key(k)

- Lista vacía

$Ld.head \rightarrow NULL \leftarrow Ld.tail$

- Lista no vacía

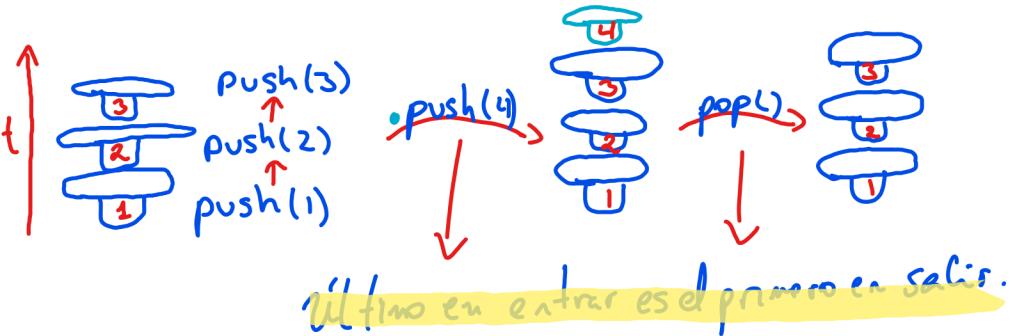
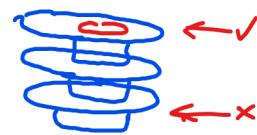


```

if (Lc.head != NULL)
{
    nodo *x = Ld.head
    while (x != Lc.tail & x.key != k)
        x = x.next
    if (x.key == k)
    {
        if (x == Ld.head)
            Ld.head = Ld.head.next
        if (x == Lc.tail)
            Lc.tail = Lc.tail.prev
        else
            if (x.next != NULL)
                x.next.prev = x.prev
            if (x.prev != NULL)
                x.prev.next = x.next
            delete x
    }
}

```

Pilas (Stack). LIFO (last in first out)



Implementación

↳ Arreglos

↳ Listas

Arreglos:

```
class Stack_array {
public:
    int top;
    #define stack [MAX_STACK];
    push (int x);
    int pop ();
    bool empty();
}
```



Estado inicial



push(6)



push(9)



pop()

Stack_array()

```
{top = -1;
bool empty()
if (top == -1)
    return TRUE
else
    return FALSE}
```

push (int x) O(1)

```
if (top < MAX_STACK)
    top++;
    stack[top] = x;
else
    cout << "overflow";
```

int pop()

```
if (top == -1)
    cout << "underflow"
    return 0
else
    top--;
    return stack[top+1]
```

top = 1

top = 0

Return

Listas:

```
struct nodo_s
int key;
nodo_s *prev;
```

class stack_list

```
public:
    nodo_s *top;
    push (int x);
    nodo_s pop();
    bool empty();
}
```

top → NULL

push (6)

NULL ← $\boxed{6}$ ← top

push (8)

NULL ← $\boxed{6}$ ← $\boxed{8}$ ← top

pop()

NULL ← $\boxed{6}$ ← top

trick

top ≈ tail

push ≈ insert-fin O(1)

pop ≈ elimina-fin O(1)

```

push(int x) O(1)
nodo_s *p = new nodo_s;
p.key = x;
p.prev = top;
top = p;

```

```

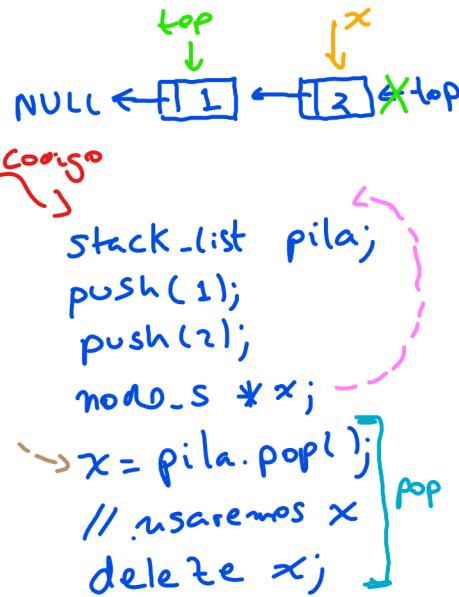
bool empty() {
    if (top == NULL)
        return TRUE;
    else
        return FALSE;
}

```

```

nodo_s *pop() O(1)
if (top == NULL)
    cout << "underflow";
    return NULL;
else
    nodo_s *x = top;
    top = top.prev;
    return x;
}

```



Si el nodo solo almacena un elemento, podríamos eliminarlo y retornar solo el dato

Evaluación de Expresiones:

Infija: Operador al medio : $3+4$

Prefija: Operador después de operandos : $+34$

Posfija: Operador antes de operandos : $34+$

↳ Evaluación más rápida que la notación infija. Parentesis no son requeridos.

1) Crear una pila para almacenar operandos

2) Escanear la expresión

- Si es un operando, push.

- Si es un operador, 2x pop, eval, push.

3) Cuando termina la expresión, pop.

String operando : [a-z]
 operador : ^ * / + -

Difícil de evaluar para las computadoras debido al trabajo adicional de decidir la precedencia.

Ex: $1+2*2^2$

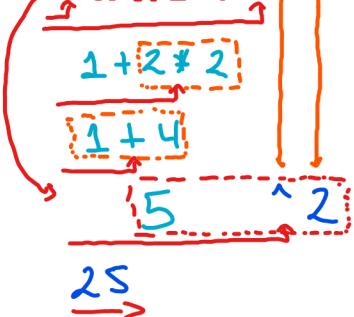
$\approx O(Kn)$
↑
#oper

$1+2*4$

$1+8=9$

Puede requerir paréntesis

Ex: $(1+2*2)^2$



Ex: $122*+2^4$
abc d

1	push	1	O(n)
2	push	1 2	
2	push	1 2 2	
*	2pop, eval, push	1 4	
+	2pop, eval, push	5	
2	push	5 2	
^	2pop, eval, push	25	

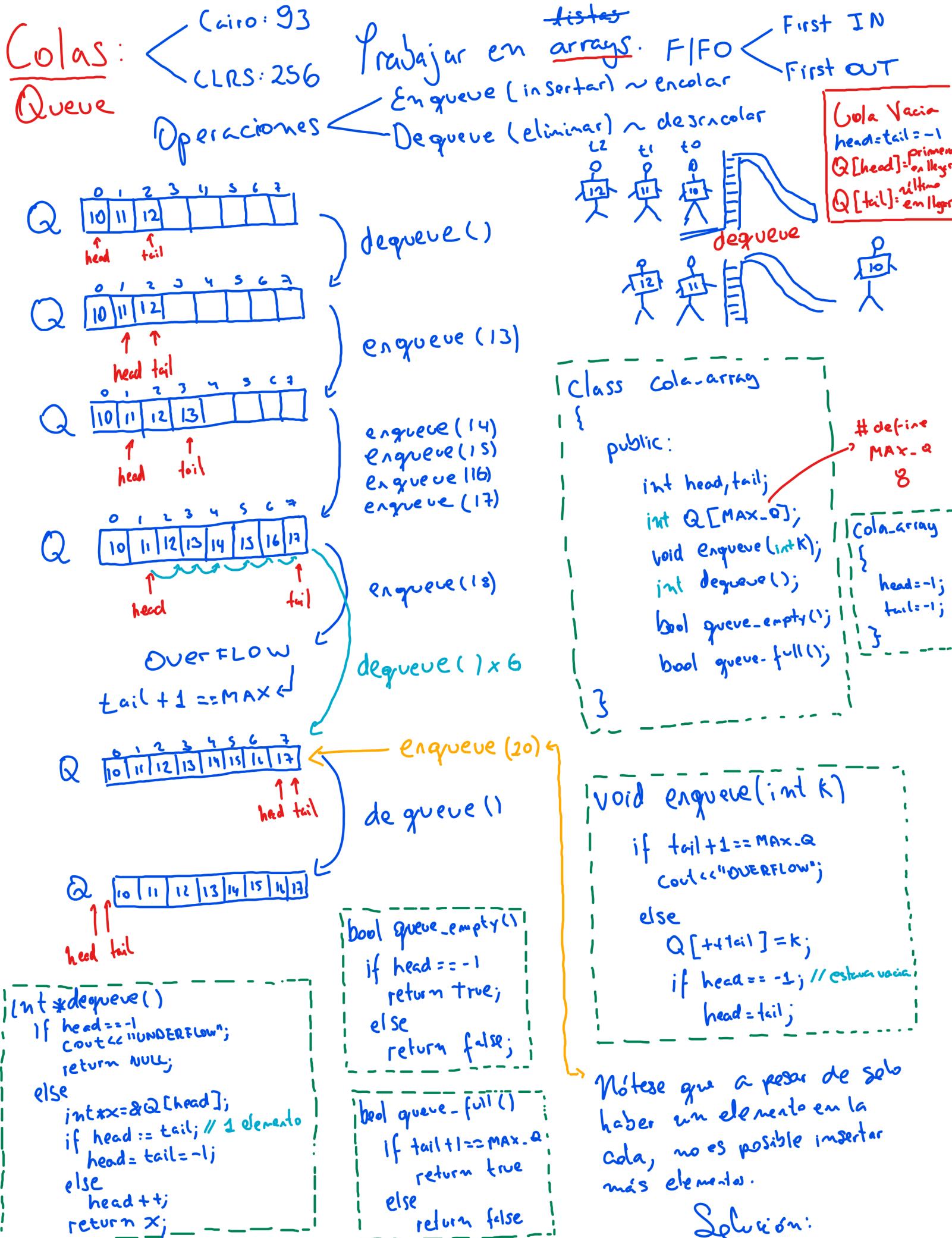
Convertir expresión Infija a Posfija:

<u>a</u>	<u>+</u>	<u>b</u>	tokens
operando	operador		prec

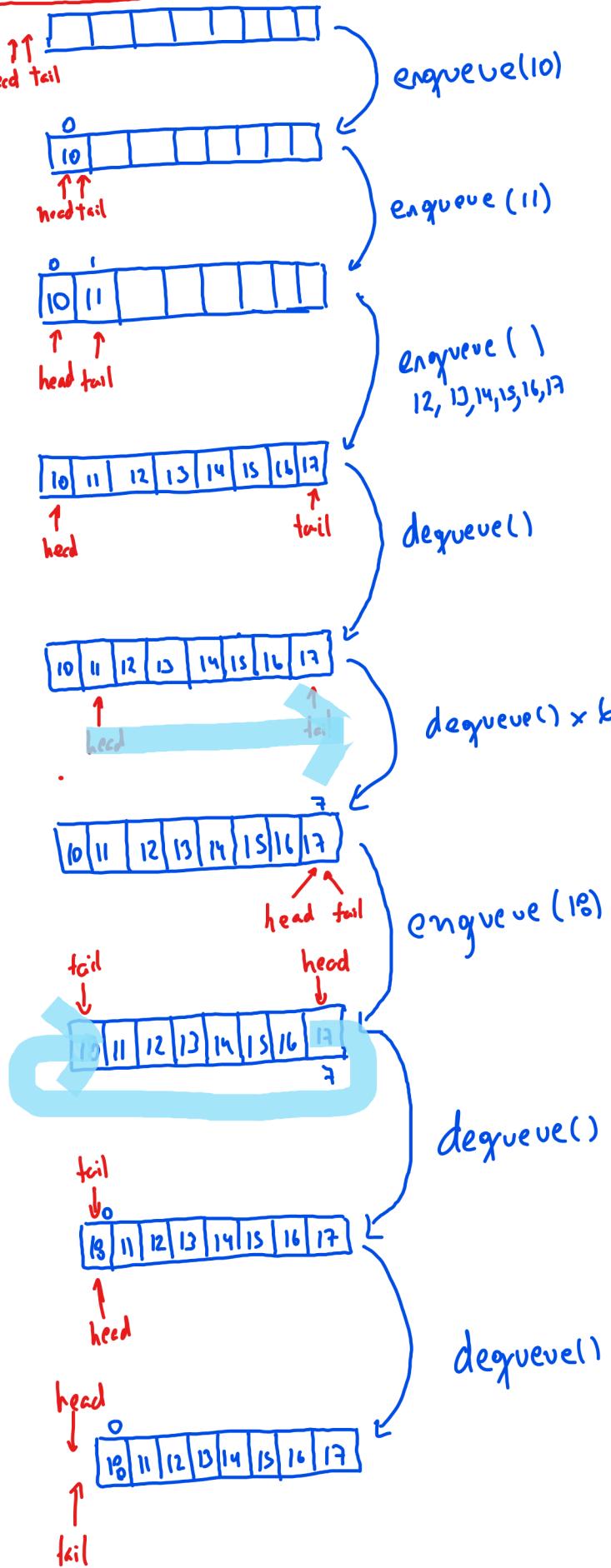
```

i<-0 // infija
j<-0 // posfija
while (infija[i] != '\0')
    token = infija[i++]
    if (token >='a' and token <='z') // es operando
        posfija[j++] = token
    else if (token == '(')
        pila.push(token)
    else if (token == ')')
        aux=pila.pop();
        while (aux != '(')
            posfija[j++]=aux;
            aux=pila.pop();
    else // es un operador
        while (!pila.empty() and
               prec(token) <= pila.prec_top()) // prec de top
            if (token == '^' and pila.prec_top() == 3)
                break;
            posfija[j++]=pila.pop();
        pila.push(token);
    while (!pila.empty())
        posfija[j++]=pila.pop();
    posfija[j]='\0';
    cout<<posfija;

```



Colas Circulares: Aprovechan todo el espacio en un array.



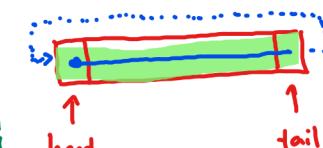
```
bool queue-empty()
{
    return head == -1;
}
```

```
bool queue-full()
```

```
return (tail+1 == head) || (head == 0 & tail+1 == MAX_Q);
```

```
void enqueue(k)
```

```
if tail+1 == MAX_Q
    tail = -1
Q[tail+1] = k
if head == -1
    head = tail
```



```
int dequeue()
```

```
int x = Q[head];
if head == tail // solo 1 elemento
    head = -1;
    tail = -1;
else
    if head+1 == MAX_Q
        head = -1
    head++;
return x;
```

#define
MAX_Q 8

```
class ColaCircularArray
```

```
≈
```

```
class ColaArray
```

enqueue y dequeue
incluyen verificación de OF, UF.

enqueue y dequeue

Suponen, asumen no estar en OF, UF
es decir; se debe verificar OF y UF
antes de llamarlos.

Esta vez SIEMPRE
debo verificar OF, UF
antes de enqueue.

Doble Cola: Generalización que permite insertar e eliminar por cualquiera de los dos extremos (head, tail).

↑ enqueve, dequeve
asumimos no Ø, UF

Variantes:

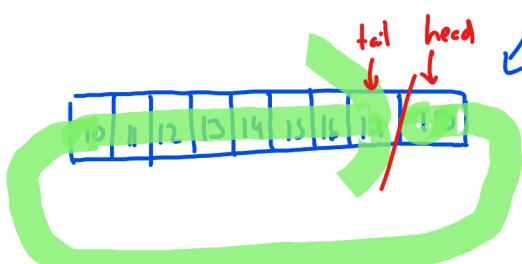
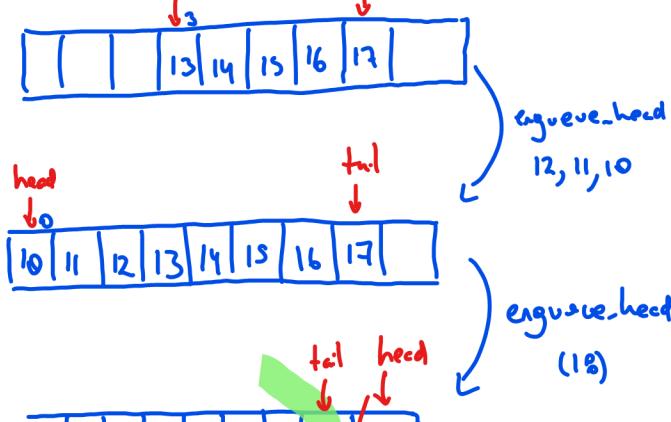
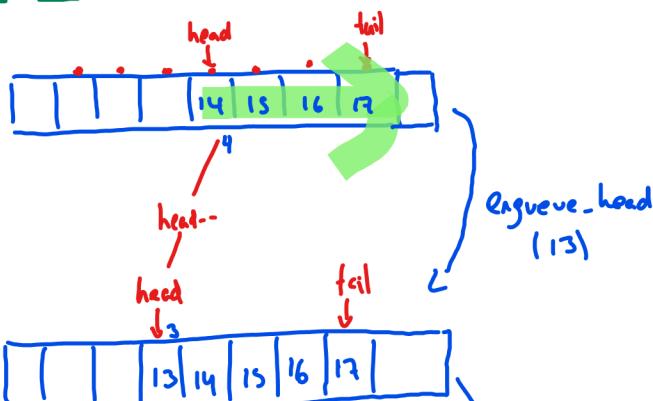
- Doble cola con **entrada restringida**.

↳ Elimina por cualquier extremo
Inserta solo por la coda (tail).

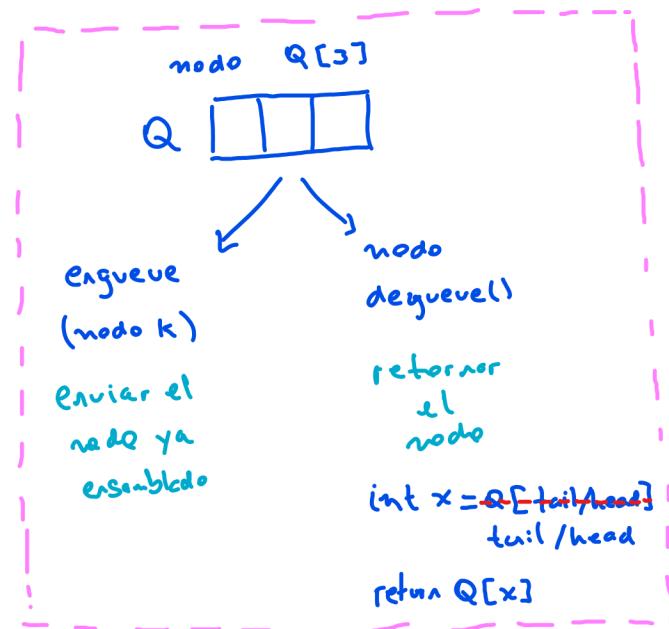
- Doble cola con **salida restringida**.

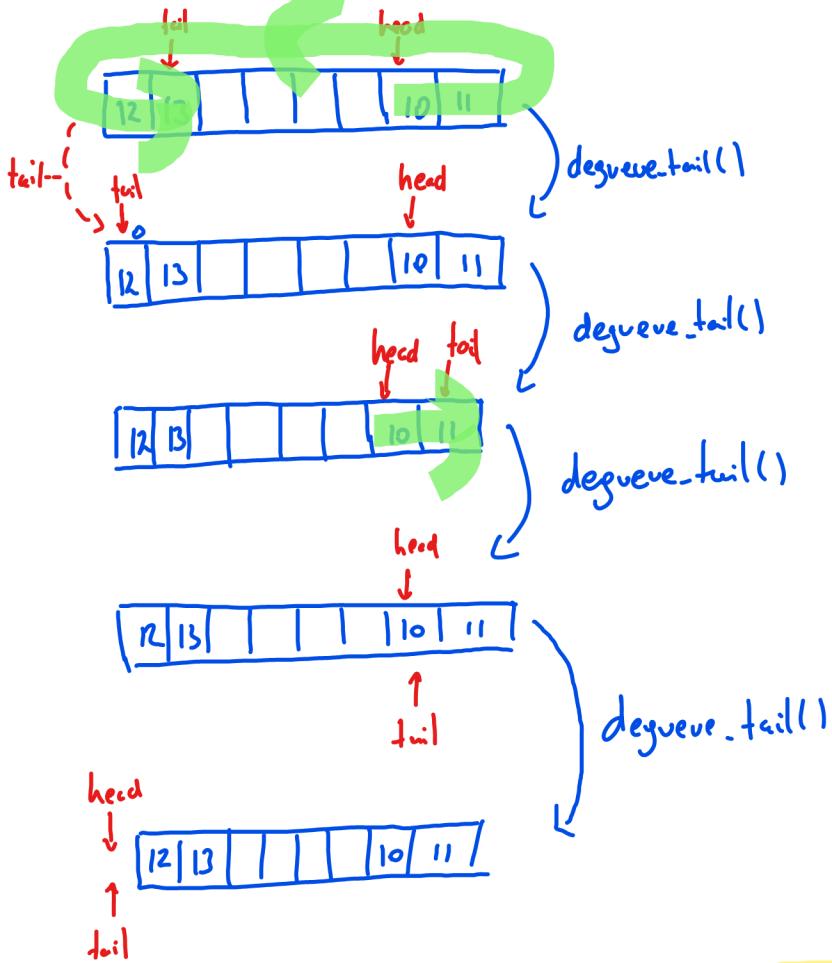
↳ Insertar por cualquier extremo.
Eliminar solo por la cabeza (head).

```
class cola-doble-circular-array
{
    class cola-circular-array
    {
        enqueve = enqueve_tail;
        dequeve = dequeve_head;
    }
    void enqueve_head(int k);
    int dequeve_tail();
}
```



```
void enqueve_head(int k)
{
    if head == -1 // cola vacia
        enqueve_tail(k);
    else
        head--;
    if head == -1;
        head = MAX_Q-1;
    Q[head] = k;
}
```





```

int dequeue_tail()
{
    int x = Q[tail];
    if head == tail // solo 1 elem
        head = -1; tail = -1;
    else
        tail--;
    if tail == -1
        tail = MAX_Q-1;
    return x;
}

```

Árboles:

No Lineales

Árboles

Ls Búsqueda Binaria

Binaria

Ls Balanceados (AVL, RB)

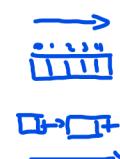
Ls Trie. (Information Retrieval)

Ls Strings.

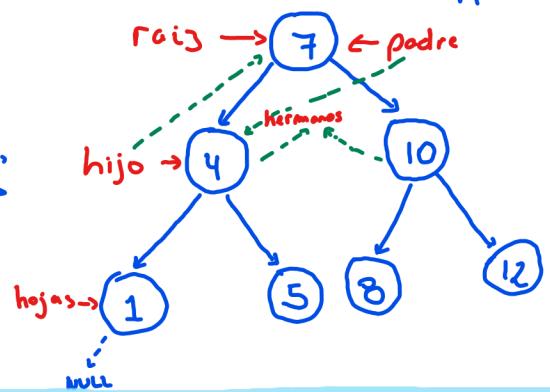
Lineales

Arreglos

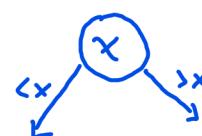
Listas



Árbol



Árboles de Búsqueda Binaria

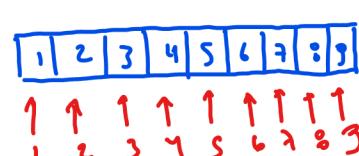
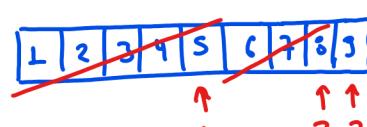
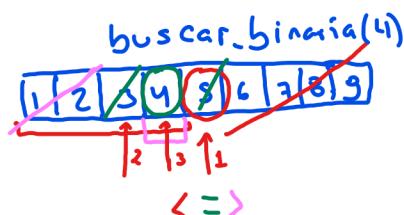


Búsqueda Binaria.

Lineal. Ordenada.

→ Arreglo

Lista X Búsqueda Binaria



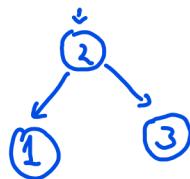
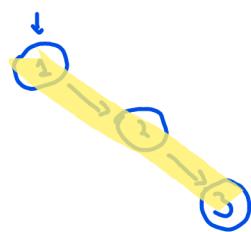
buscar_binaria(9)

consultas = 3 → O(log₂n)

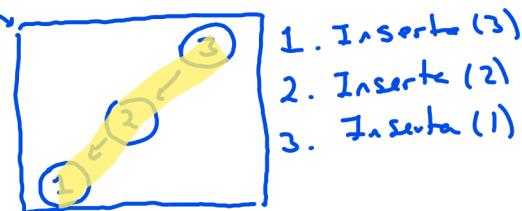
buscar_lineal(9)

consultas = 9 → O(n)

Complejidad



1. Mismos datos
2. Los datos fueron insertados en orden **diferente**.

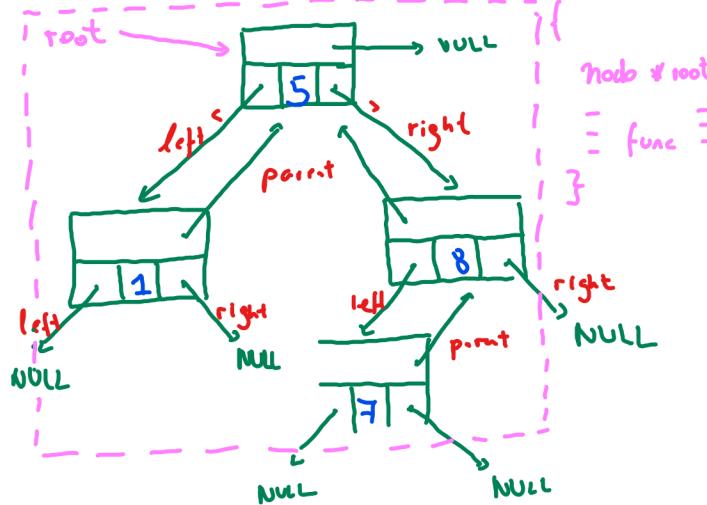


Árbol Lineal no balanceado.

Soluc. AVL, RBTree.



max	1
1	3
2	7
:	:
h	$h^{2+1}-1$



Altura :

0	0
1	1
2	2
:	:
h	$h^{2+1}-1$

data, performance
place

Recorrido Árbol (walk)

2 + 1
+ 2 1
2 1 +

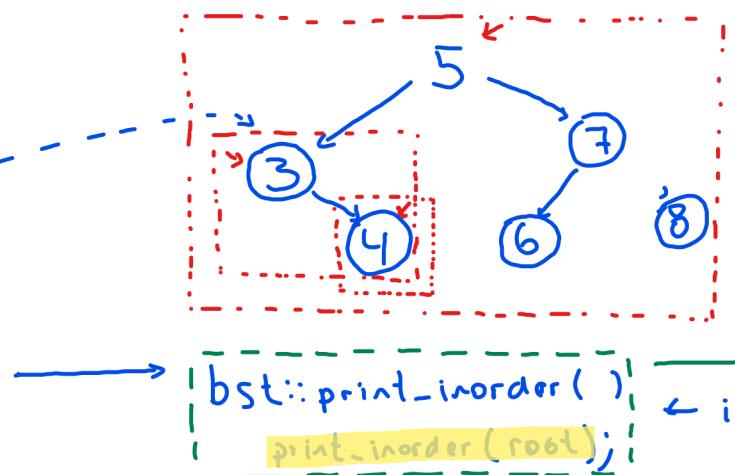
infixa
prefija
postfixa



inorder left, key, right
preorder key, left, right
postorder left, right, key

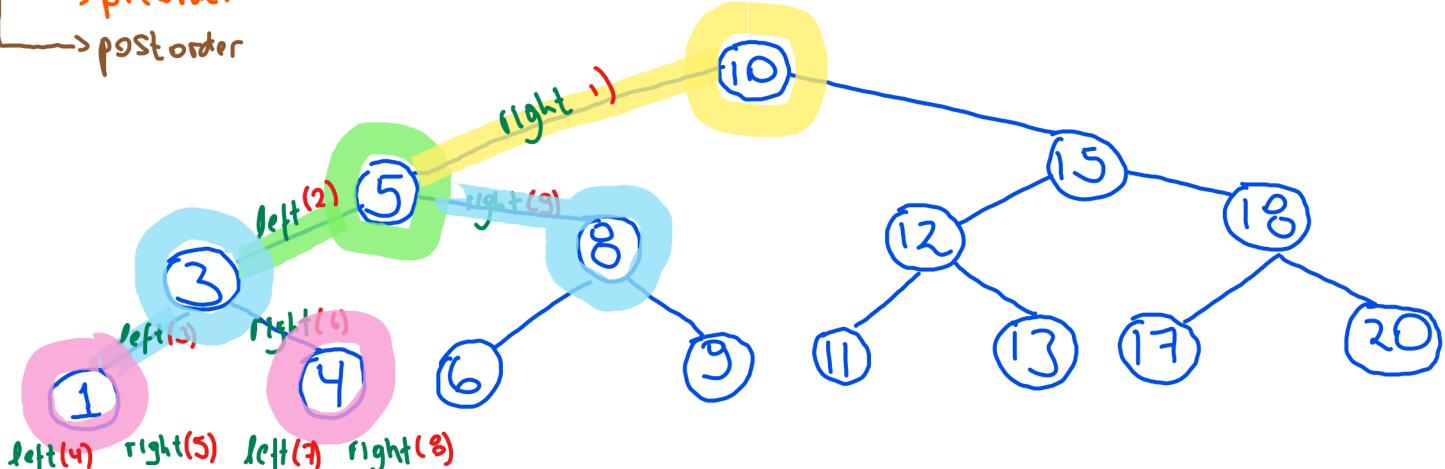
```
recursive
1 print_inorder(x)
1 if(x!=NULL)
1   print_inorder(x.left);
1   cout << x.key;
1   print_inorder(x.right);
2 12.1..... 312.1..... 233.1.....
```

inorder: de forma ordenada.
preorder
postorder

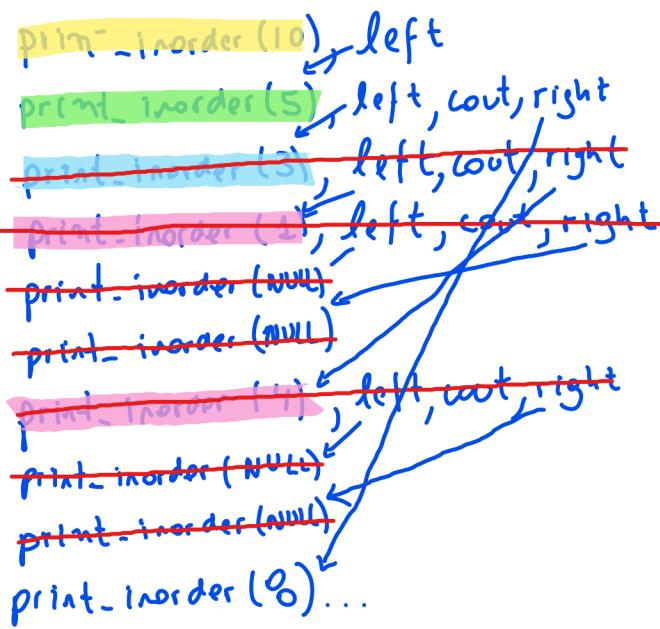


bst arbol;
arbol.print();

interface



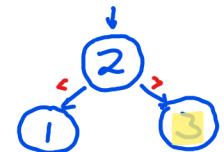
Pila



: 1 3 4 5

```
tree_node *Search_recursive(tree_node *x, int k)
```

```
if (x == NULL or x.key == k)
    return x
if (k < x.key)
    return search_recursive(x.left, k)
else
    return search_recursive(x.right, k)
```



interface

```
tree_node *Search_recursive(int k)
return search_recursive(root, k)
```

```
tree_node *Search_iterative(int k)
```

```
tree_node *x = root;
while (x != NULL and x.key != k)
    if (k < x.key)
        x = x->left;
    else
        x = x->right;
return x
```

→ main()

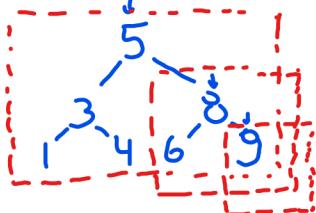
```
tree_node *result;
result = arbol.search(k);
if (result == NULL)
    cout << result.key;
else
    cout << "no encontrado";
```

```

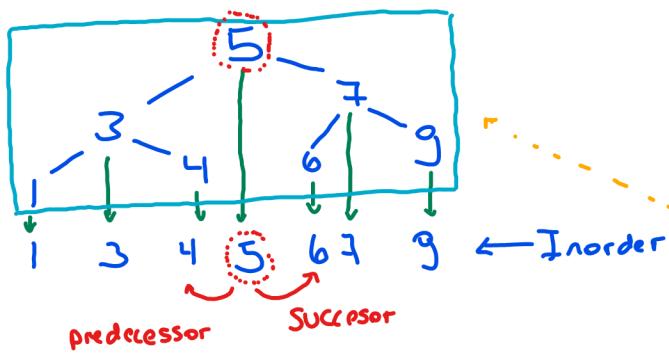
tree-node *max()
{
    tree-node *x = root;
    while(x.right != NULL)
        x = x.right;
    return x;
}

```

para max/min seguir el puntero right/left hasta antes de NULL.



Successor y predecessor:



Dado bst, el ^{successor} se def. _{predecessor}

como:

- El nodo ^{siguiente} en un recorrido _{anterior} inorder.
- El nodo con el valor ^{menor} _{mayor} ^{más} _{menos} grande que $x.key$.

Dos Casos

si $right=NULL$ $successor(x) = predecessor(x)$

si $right \neq NULL$
 $left$ kft

Es el ^{min} _{max} de $x.right$

Es el menor antecesor
^{mayor} cuyo hijo _{derecho} izquierdo es antecesor de x .

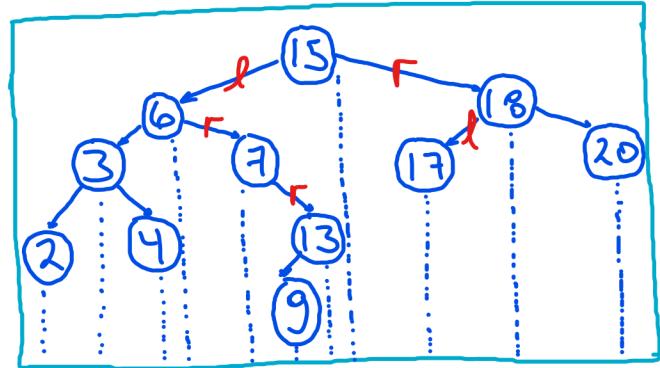
≈ recorrer hacia arriba hasta el primer nodo cuyo hijo _{derecho} izquierdo es antecesor de x .

```
tree-node *successor(x)
predecessor(x)
```

```
if(x.right!=NULL)
    left
    return min(x.right);
    max(x.left);
```

else

```
y = x.parent;
while(y!=NULL and x==y.right)
    x=y
    y=y.parent
return y
```



Inorder: 2 3 4 6 7 9 13 15 17 18 20

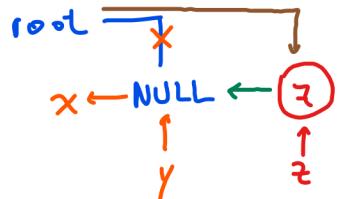
Mientras recorro _{hacia arriba} y el antecesor es _{valido} pero es hijo _{derecho} _{izquierdo}

Inserción - Al igual que en los procedimientos de búsqueda (binaria).
 - Empezamos en la raíz del árbol y descendemos hasta encontrar un hijo NULL para reemplazarlo.

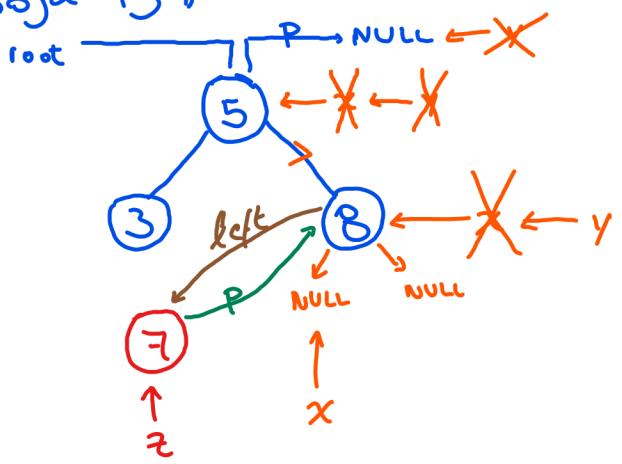
insert (int k)

```
tree_node *z = new tree_node();
z.key = k
x = root // inicio de búsqueda
y = NULL // parent de z
while (x != NULL) {
    if (z.key < x.key)
        x = x.left
    else
        x = x.right
    z.parent = y
    if (y == NULL)
        root = z // Árbol vacío
    else if (z.key < y.key)
        y.left = z
    else
        y.right = z
```

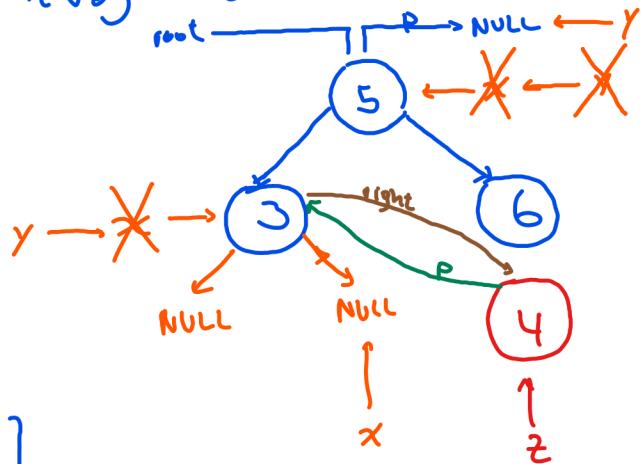
- Árbol vacío



- Hijo izquierdo



- Hijo derecho



Mnemotecnia: ("lógica")

1- x apunta al conocido NULL
 y apunta al parente de x }
 mientras x != NULL

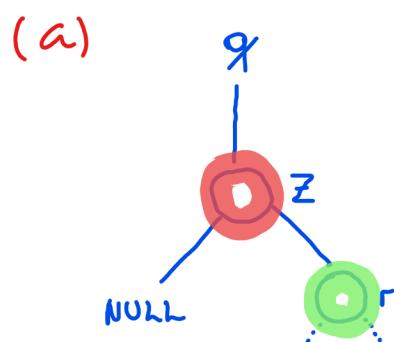
2- hijo (nuevo) reconozca al parente (y)
 3- parente (y) reconozca al hijo (nuevo).

↳ left <
 ↳ right >

x → root
 y → NULL

(caso base)

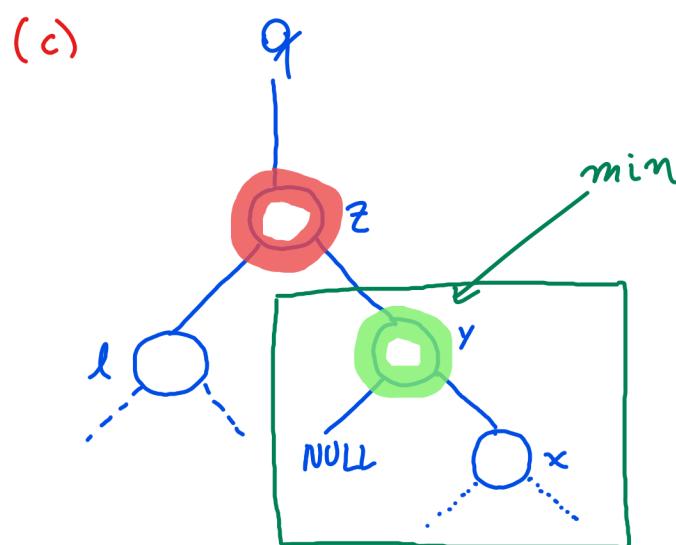
Eliminación:  ← nodo a eliminar  ← reemplazo



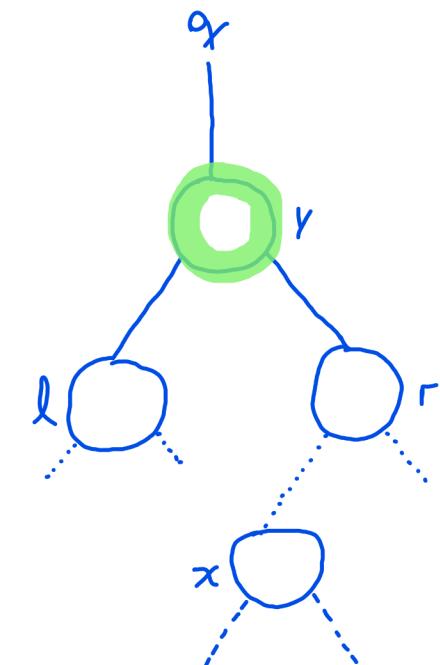
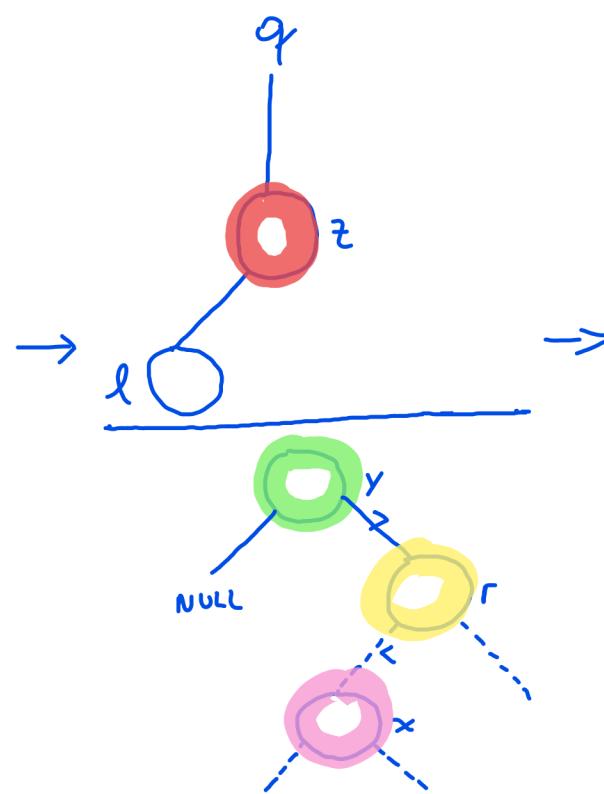
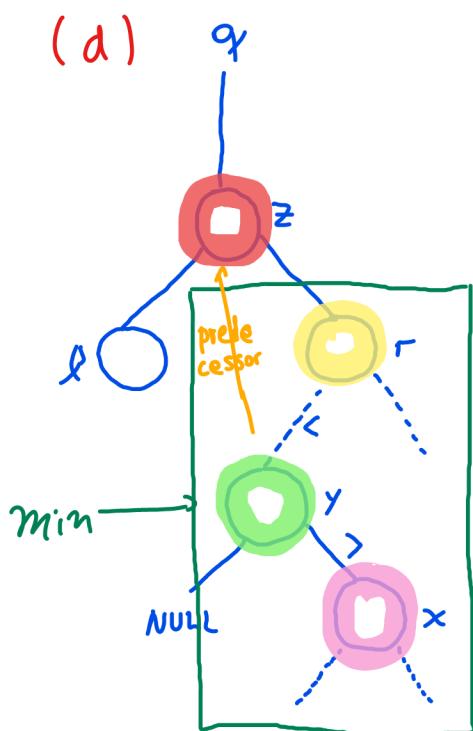
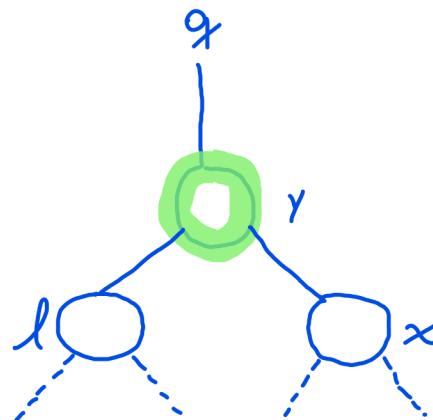
$z.left = \text{NULL}$



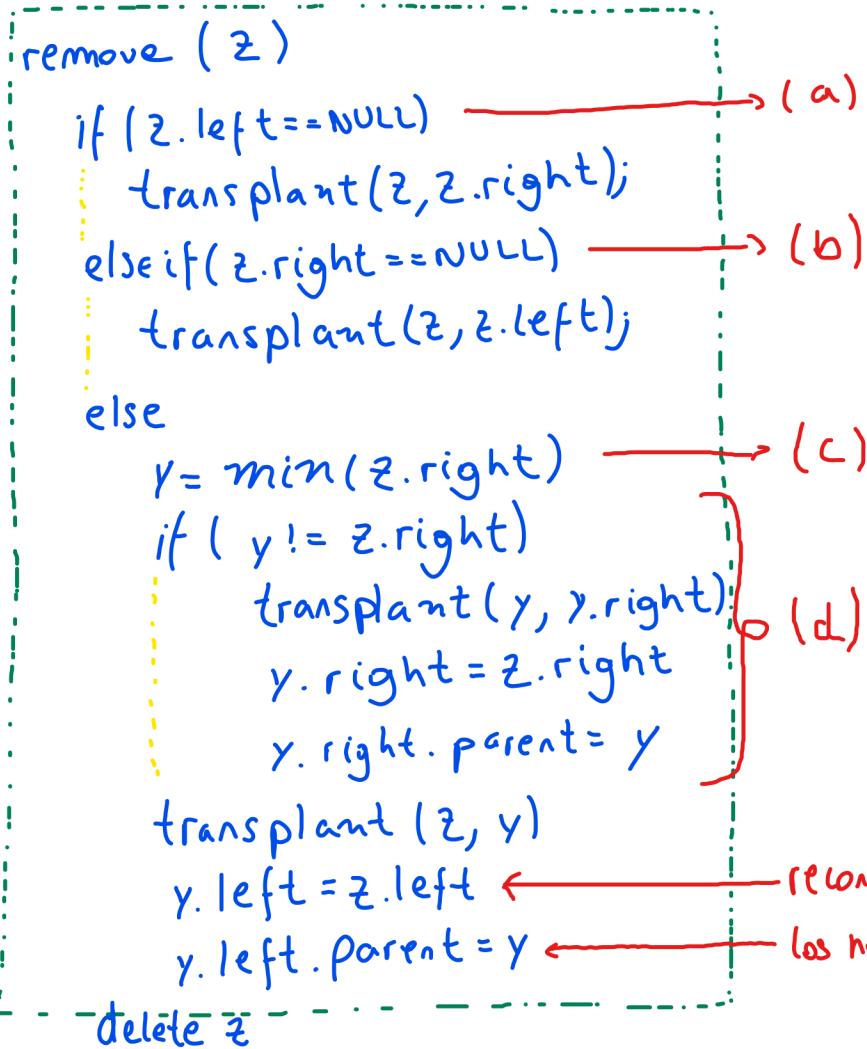
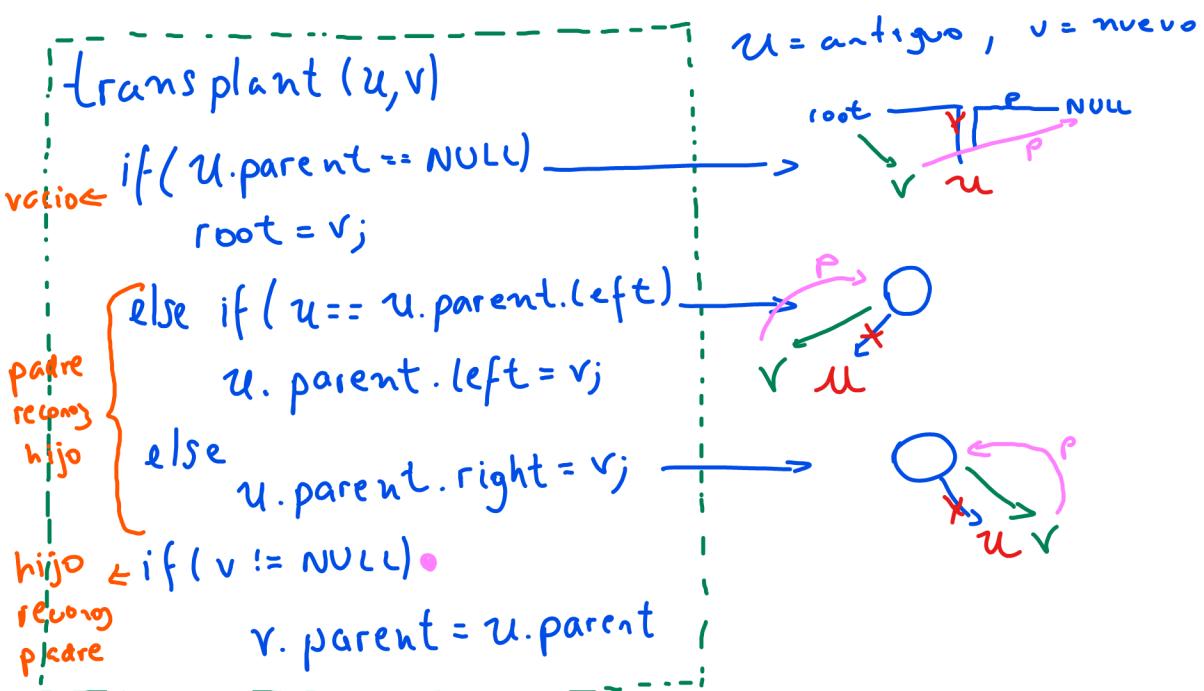
$z.right = \text{NULL}$



if $y == z.right \wedge y.left == \text{NULL}$)



Transplante: - Reemplaza un subárbol como hijo de su padre con otro subárbol.
 - El padre reconoce a su nuevo hijo.



End