Punteros: C++ ← potencial, diferencia

- Señalar cosas ⇧.
- C++ POO todo es un objeto.

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

Objetos > 1byte ⇝ posiciones correlativas

int = 4bytes

1bit < 0, 2^1 [0 1] (2^1-1)

8bits < ..... 2^8 [0 255] (2^8-1)

byte ← microprec.

C/byte tiene dirección de memoria.

S.O. 32bits = 2^{32} = 4GB

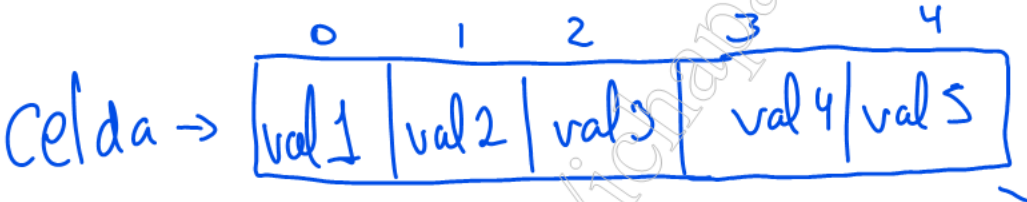
S.O. 64bits = 2^{64} = 16EB

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

Puntero: Tipo especial de objeto que contiene una dirección de memoria. (Def 2).

⇝ Char, int, float, array ← definido al declarar

tipo de objeto.



← array

memoria ordenador

indice = 0 ≃ → puntero

celda[0] = celda[indice]

Valor Inicial: → int *pInt | char *pChar | struct stPunto *pPunto

int a. a=10 | ~~int *a *a=10~~ | int *a = 0 ← objeto NULL

dirección

int a= 123

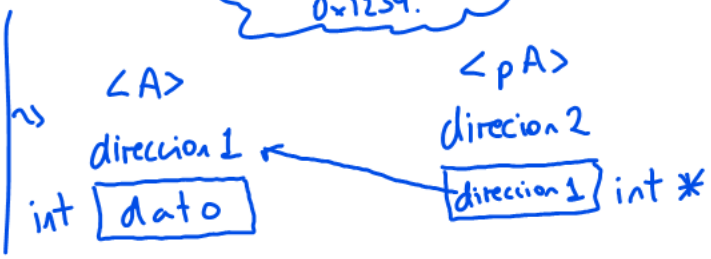
dirección

dirección ← puntero

Declaración: <tipo> *<identificador> | ~~int* x, y;~~ (puntero, int)

Dirección Objeto:


```
int A;
int *pA;
pA = &A
```



Objeto apuntado por objeto

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

```
int *pInt;
int x=10
int y
pInt = &y
*pInt = x
```

Arreglos y Punteros:

int array[10]:
- Declara puntero constante del mismo tipo que los elementos.
- Reservar memoria # elementos del arreglo posiciones consecutivas.
- Se inicializa apuntando al primer elemento.
- Compilador asociara a los elemento. [1]

Asignación

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

①

Aritmetica:

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

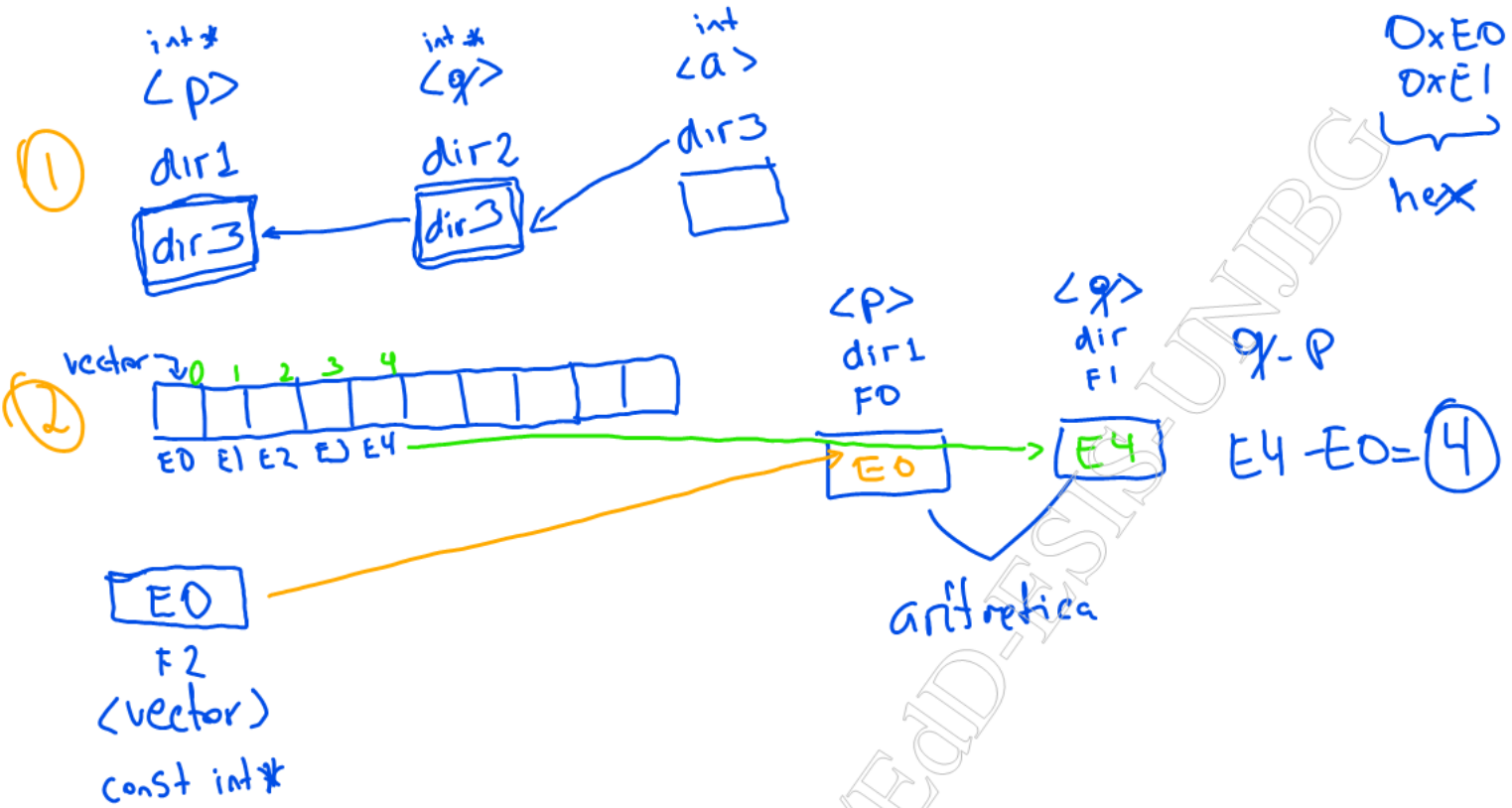
4 ②

Comparaciones:

< ≤ ≥ > == !=

```
if(p != NULL)
if(p)
```

```
gpr (int *)NULL != (float*) NULL
```
③



Punteros Genericos:

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

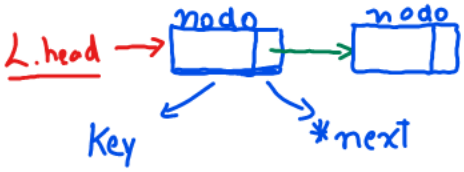
Estructuras de Datos Elementales:

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

1º~ Listas Enlazadas: Dinamica

- sin reservar memoria anticipada. ~~int array[10];~~
- facilidad en el orden.
- Array -> Orden x indice.
- Lista -> Orden x puntero.
- Simplemente Enlazadas, Doblemente Enlazadas.

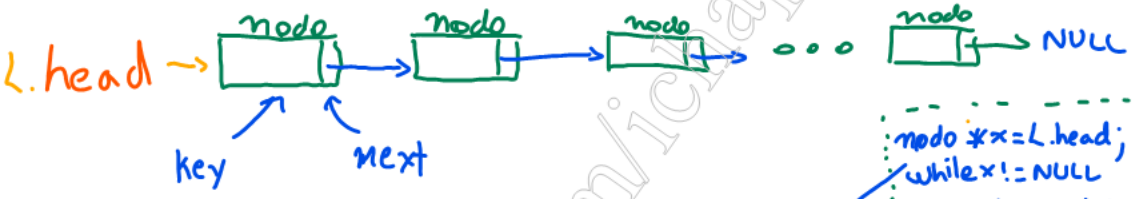
Key ↓ forma identif nodo.



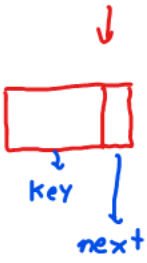
- Una lista puede tener varias formas, ordenadas o desordenadas, circulares o no.
- Ordenada: head = min, tail = max
- Circular: tail.next = head

1o1~ Simplemente Enlazada.

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



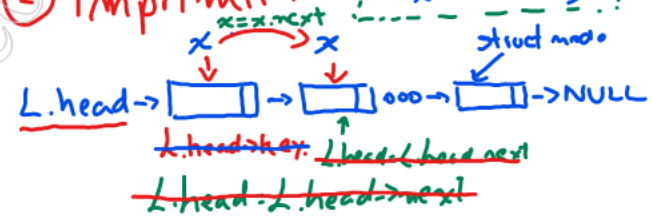

```
struct nodo
{
    int key;
    nodo *next;
};
```

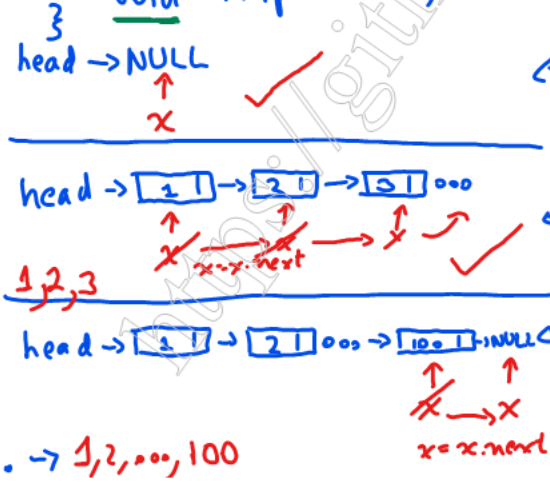



```
lista_simple Li;
class lista_simple
{
public:
    nodo *head;   -> this->head
    void imprimir();
};
```

1) imprimir:

```
nodo *x=L.head;
while x!=NULL
    cout<<x.key;
    x=x.next;
```





a) No existan elementos o nodos. ← L.head -> NULL x==NULL

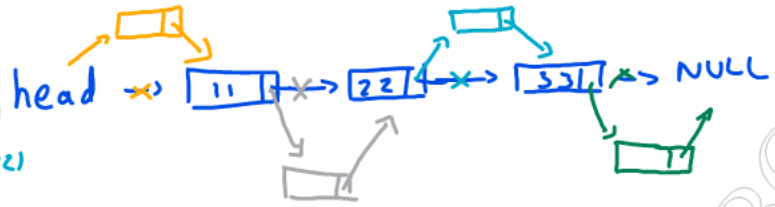
b) Existen elementos. x apunta a un nodo ← cout<<x.key<<endl; x≠NULL

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

. -> 1, 2, ..., 100

② Insertar

- inicio
- fin
- antes_de (22)
- despues_de (22)



a) inserta_inicio(K)

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

- Lista está vacia.

- Lista tiene elementos. ✓

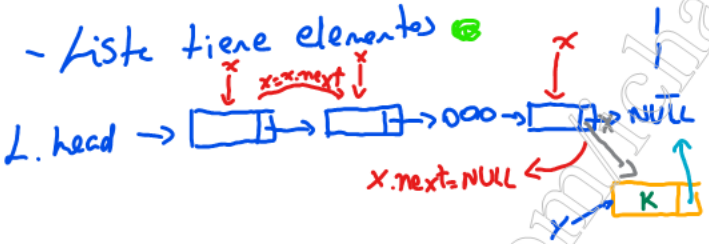
b) Insertar_fin(K)

```
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
```

- Lista está vacia

L.head → NULL

- Lista tiene elementos



c) Eliminar(K) ← elimina primera aparición

```
if(L.head != NULL)
{   nodo *x = L.head
    if(x.key == k)
    {
        L.head = x.next
        delete x
    }
    else
    {
        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
```

F no está k

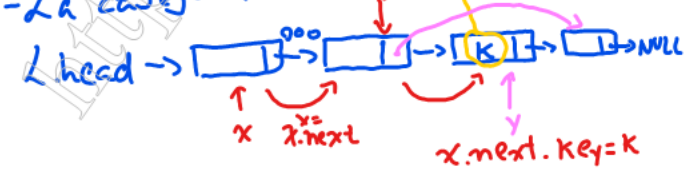
F encontré k

- Lista vacia

L.head → NULL ✓

- La cabeza es K

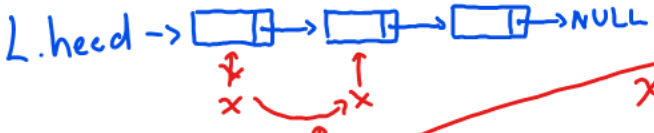
- La cabeza no es K



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

d) buscar_key(k) — Si hay al menos uno → 0 no encontré / 1 encontré

L.head -> NULL



p: x == NULL (ya no hay nodos) → ya no buscar
q: x.key == k (ya lo encontré) → ya no buscar

p ∨ q

x = x.next

buscar

~(p ∨ q)

~p ∧ ~q

2.- Listas Circulares

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

Simple link
doble link

Tener cuidado con los BUCLES.

a) buscar_key(int k)

- Lista vacía

Lc.head -> NULL

ningún enlace es NULL

- Existe elemento

- No existe el elemento

- Dos elementos

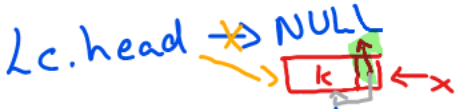

```
nodo *x = Lc.head
if (x == NULL)
    return 0;
else
{
    while (x.next != Lc.head ∧ x.key != k)
        x = x->next
    if (x.key == k)
        return 1;
    else
        return 0
```

p: x.next==Lc.head (estoy en el último elemento)

q: x.key==k (encontré el elemento)

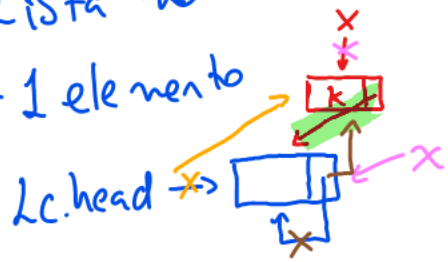
b) inserta_inicio(k)

- Lista vacia

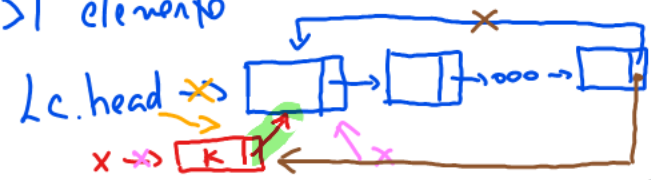


- Lista no vacia

- 1 elemento



->1 elemento



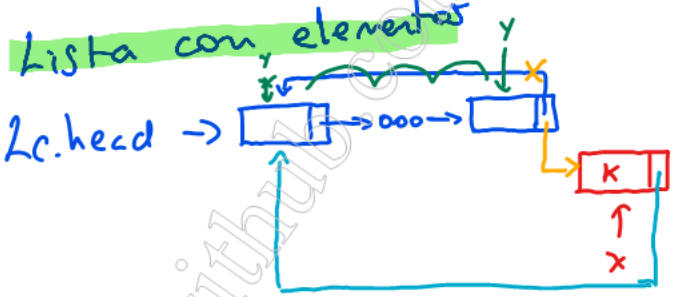

```
nodo *x = new nodo
x.key = k
x.next = Lc.head
Lc.head = x
if(x.next == NULL)
    x.next = x
else
    x = x.next
    while(x.next != Lc.head.next)
        x = x.next
    x.next = Lc.head
```

c) insertar_fim(k)

- Lista vacia

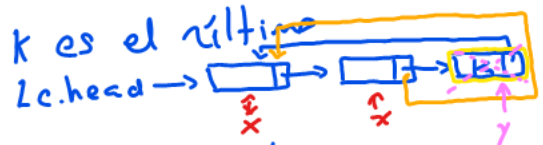
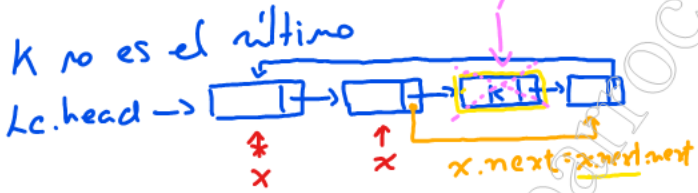
✓

- Lista con, elementos




```
nodo *x = new node
x.key = k
if(Lc.head == NULL)
{
    Lc.head = x
    x.next = x
}
else
{
    nodo *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 -> NULL ✓
- k está al inicio
  * k es el único ✓
  Lc.head -> [K] ~> Lc.head -> NULL
  * k no es el único
  
- k no está al inicio
  * k es el último
  
  * k no es el último
  


```
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 -> NULL <- Ld.tail ⇐ Lista doble vacía

Ld.head -> <- Ld.tail ⇐ Lista doble con 1 nodo

⇐ Lista doble con n nodos

```
struct nodo_ld
{
    int key;
    ==========
    nodo_ld *prev;
    nodo_ld *next;
};
```

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

"lista_ld.h"
- struct nodo_ld
- class lista_ld

a) insertar_fin(K)

Lista vacía

Lista con elementos


```
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 vacia

Ld.head -> NULL ← Ld.tail

- Lista no vacia

Si está k en la lista

* k es el único

✓

* k es cabeza

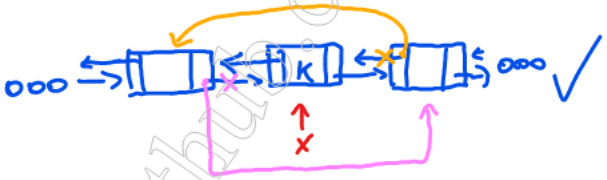
✓

* k es cola

✓

x

* k no es cabeza ni cola



✓

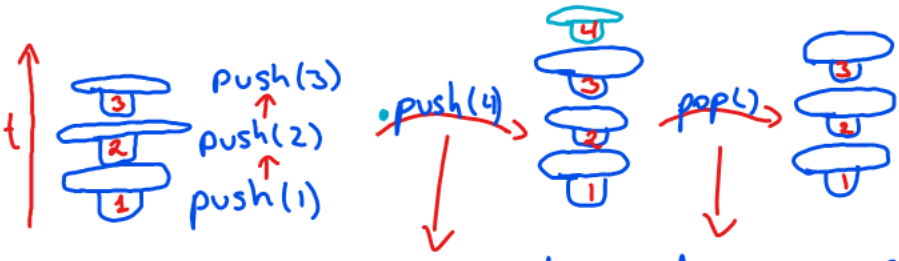
x

no está k en la lista

* k no está


```
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
        else if (x == Ld.tail)
            Ld.tail = Ld.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

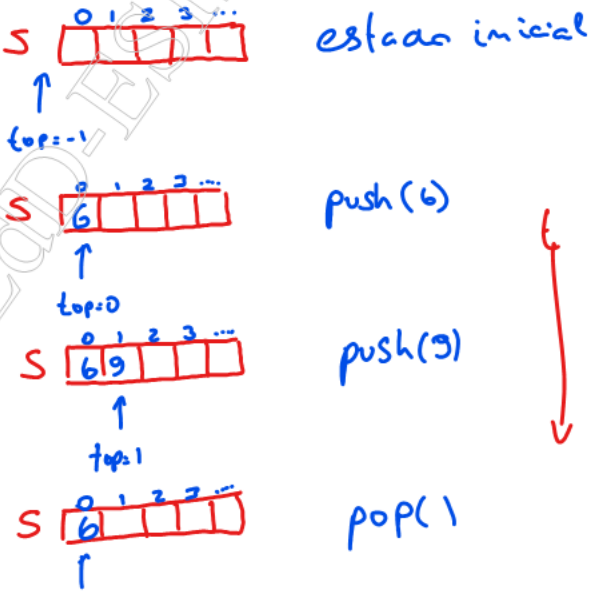


Implementación
- Arreglos
- Listas

último en entrar es el primero en salir.

Arreglos:

```
Class stack_array {
  public:
    int top;          // -1
    stack [MAX_STACK]; // #DEFINE
    push ( int x);
    int pop ( );
    bool empty();
}
```




```
Stack_array()
  top=-1;
```

```
push(int x)                    O(1)
  if(top+1 < MAX_STACK)
    top++;
    stack[top]=x;
  else
    cout <<"overflow";
```



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


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

Listas:

```
Struct nodo_s
  int key;
  nodo_s *prev;
```

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

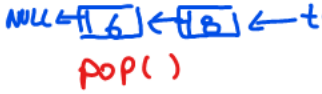
```
stack_list()
  top=NULL
```

top → NULL

push(6)

push(8)

pop()



trick

top ≈ tail

push ≈ inserta_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

código

```
stack_list pila;
push(1);
push(2);
nodo_s *x;
x = pila.pop();      pop
// usaremos x
delete x;
```

Evaluación de Expresiones:

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

Infija: Operador al medio: 3+4 → Difícil de evaluar para las computadoras debido al trabajo adicional de decidir la precedencia.

Prefija: Operador después de operandos: +34

Posfija: Operador antes de operandos: 34+

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

eval_posfija

1) Crear una pila para almacenar operandos
2) Escanear la expresión
   - Si es un operando, push.
   - Si es un operador, 2× pop, eval, push.
3) Cuando termina la expresión, pop.

≃ O(kn) (k = # oper)

Ex: 1+2*2^2
1+2*4
1+8 = 9

Puede requerir paréntesis

Ex: (1+2*2)^2
1+2*2
1+4
5^2
25

Ex: 1 2 2 * + 2 ^
(a b c, d)

1	push	1
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

O(n)

Convertir expresión Infija a Posfija:

a + b tokens

operando operador

prec

^ 3

*/ 2

+ - 1

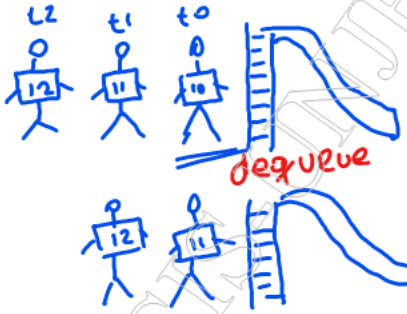
```
i←0 // infija
j←0 // posfija
while(infija[i] != '\0')
    token = infija[i++];
    if(token >='a' and token <='z') // es operando
                                    [a-z]
       posfija[j++] = token;
    else if (token == '(')
       pila.push(token);
    else if (token == ')')
       aux=pila.pop();
       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: Queue

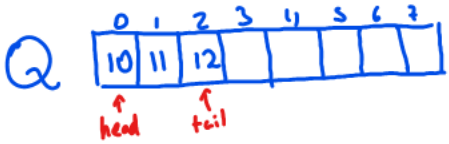
Cairo: 93
CLRS: 256

Trabajar en ~~listas~~ arrays. FIFO: First IN, First OUT

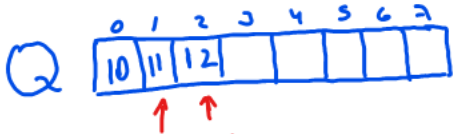
Operaciones:
- Enqueue (insertar) ~ encolar
- Dequeue (eliminar) ~ desencolar



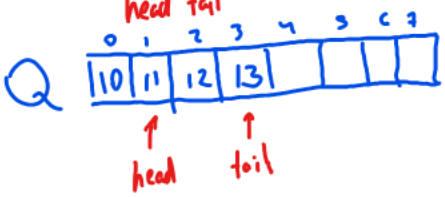
Cola Vacia
head=tail=-1
Q[head]: primero en llegar
Q[tail]: último en llegar



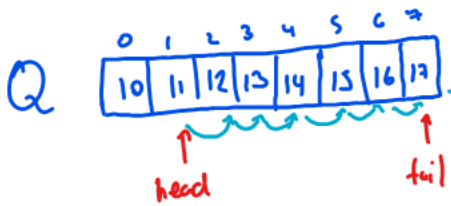
dequeue()



enqueue(13)



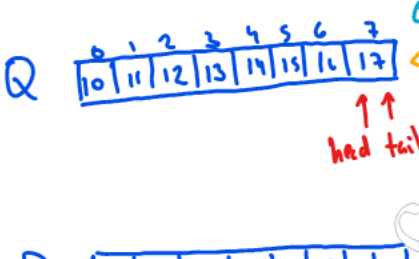
enqueue(14)
enqueue(15)
enqueue(16)
enqueue(17)



enqueue(18)

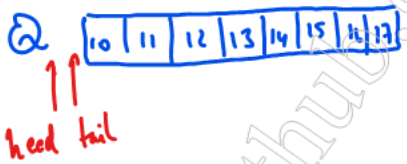
OVERFLOW
tail+1 == MAX

dequeue() x 6



enqueue(20)

dequeue()




```
class cola_array
{
public:
    int head,tail;
    int Q[MAX_Q];
    void enqueue(int K);
    int dequeue();
    bool queue_empty();
    bool queue_full();
};
```

```
#define MAX_Q 8
```

```
cola_array
{
    head=-1;
    tail=-1;
}
```

```
void enqueue(int K)
    if tail+1==MAX_Q
        cout<<"OVERFLOW";
    else
        Q[++tail]=k;
        if head==-1; // estaba vacia
            head=tail;
```

```
int *dequeue()
    if head==-1
        cout<<"UNDERFLOW";
        return NULL;
    else
        int *x=&Q[head];
        if head==tail; // 1 elemento
            head=tail=-1;
        else
            head++;
        return x;
```

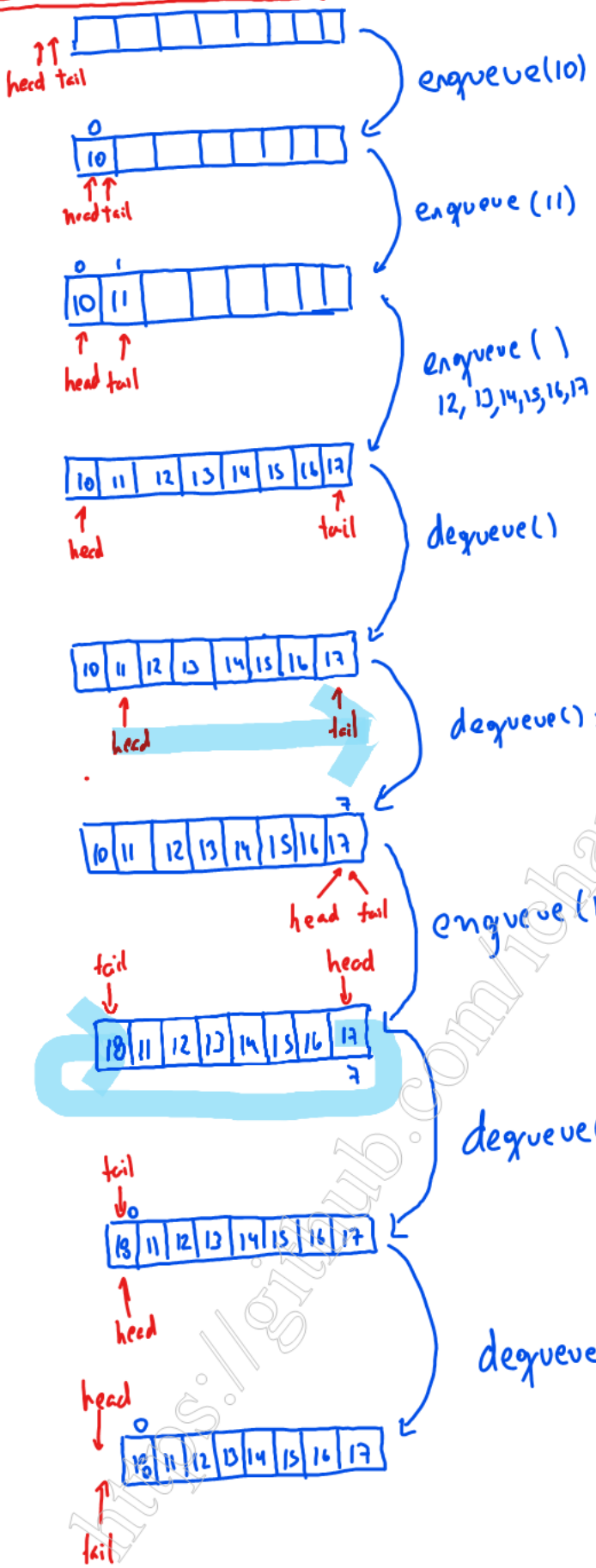
```
bool queue_empty()
    if head==-1
        return true;
    else
        return false;
```

```
bool queue_full()
    if tail+1==MAX_Q
        return true
    else
        return false
```

Nótese que a pesar de solo haber un elemento en la cola, no es posible insertar más elementos.

Solución:

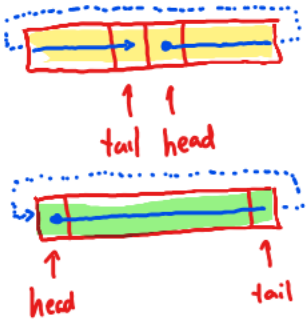
Colas Circulares: Aprovechan todo el espacio en un array.




```
bool queue_empty()
  return head == -1
```

Esta vez SIEMPRE debo verificar OF,UF antes de en/dequeue.

```
bool queue_full()
  return (tail+1 == head) v (head == 0 ^ tail+1 == MAX_Q)
```




```
void enqueue(k)
  if tail+1 = MAX_Q
    tail = -1
  Q[++tail] = 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 cola_circular_array
  ≃
class cola_array
```

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.

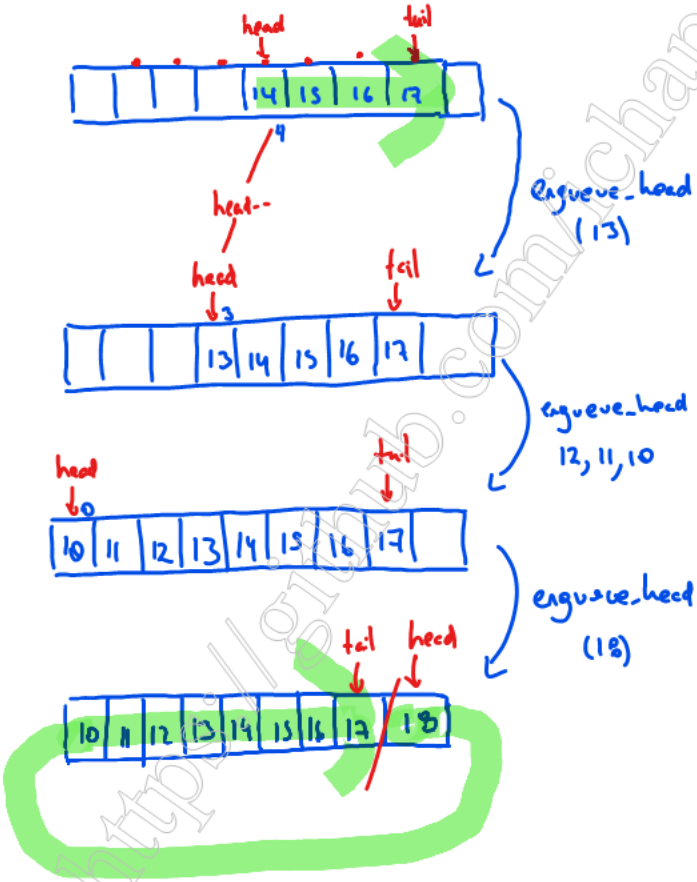
Doble Cola (circular): Generalización, permite insertar e eliminar por cualquiera de los dos extremos (head, tail).

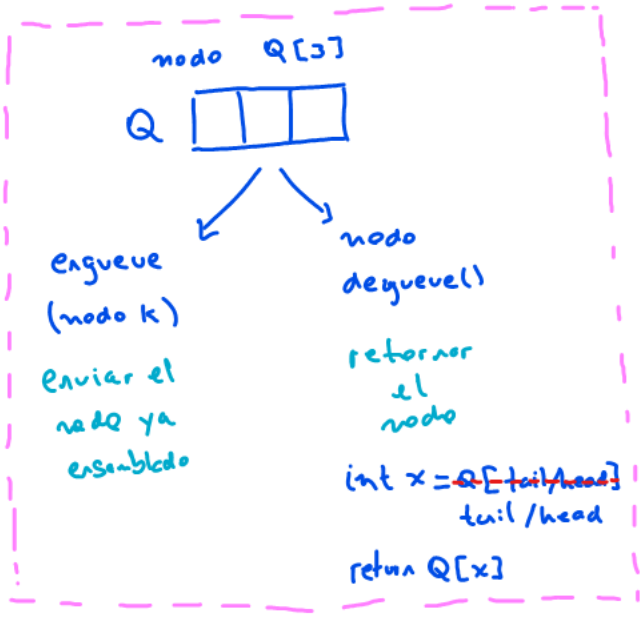
∀ enqueue, dequeue asumimos no OF, UF

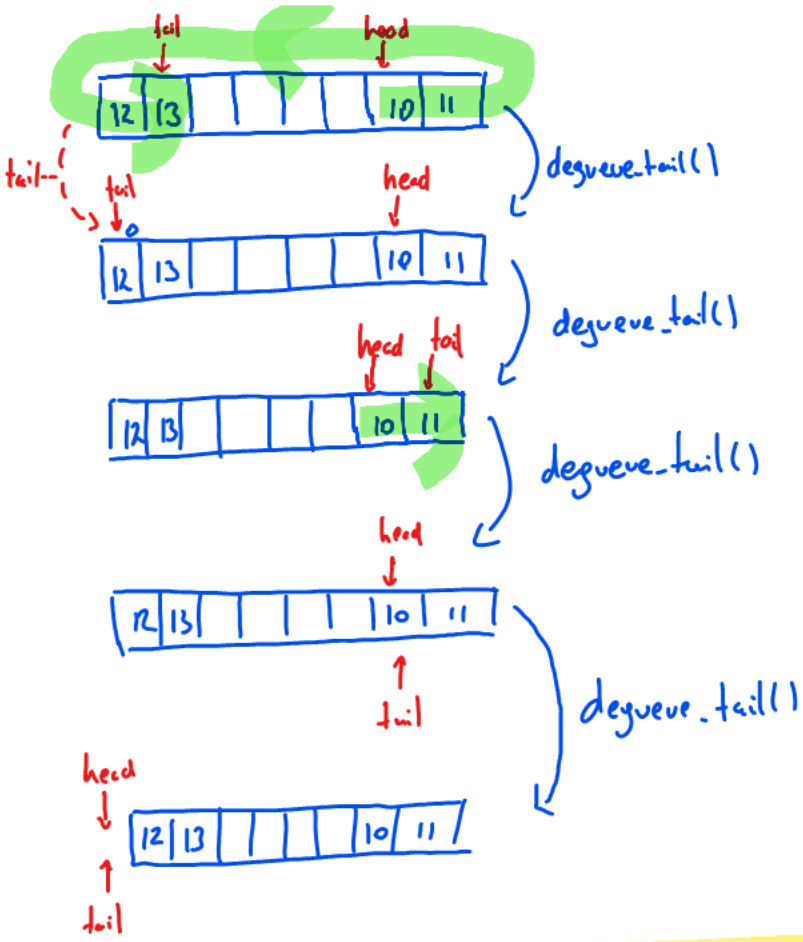
Variantes:
- Doble cola con entrada restringida.
  - Elimina por cualquier extremo
  - Inserta solo por la cola (tail).
- Doble cola con salida restringida.
  - Inserta por cualquier extremos.
  - Eliminar solo por la cabeza (head).

```
class cola_doble_circular_array
        ~
class cola_circular_array
   enqueue = enqueue_tail
   dequeue = dequeue_head
        +
   void enqueue_head(int k);
   int dequeue_tail();
```

```
void enqueue_head(int k)
  if head == -1 // cola vacia
     enqueue_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:

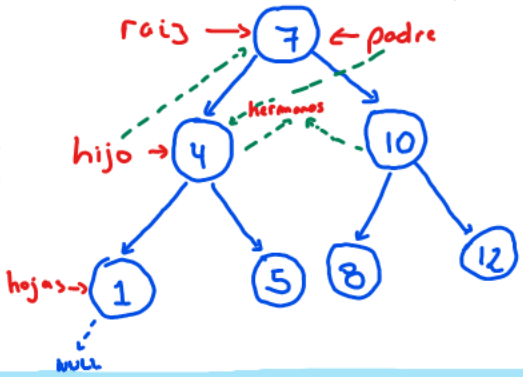
No Lineales

- Árboles
  - Búsqueda Binaria
  - Balanceados (AVL, RB)
  - Trie (Information Retrieval)
    - Strings.

Lineales

- Arreglos
- Listas

Árbol



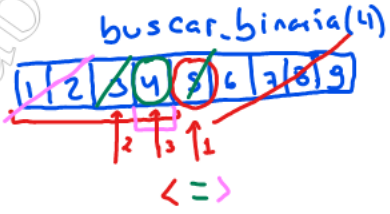

```
class nodo {
  int key;          // único
  nodo * left;
  nodo * right;
  nodo * parent;
  balanced
}
```

Búsqueda Binaria.

Lineal. Ordenada.

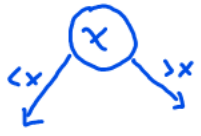
↳ Arreglo

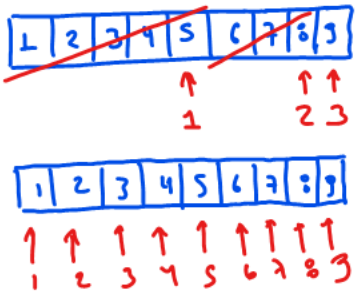
Lista X Busqueda Binaria



consultas = 3

Árboles de Búsqueda Binaria





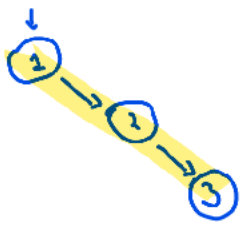
buscar_binaria(9) → balanceado → $O(\log_2 n)$

consultas = 3

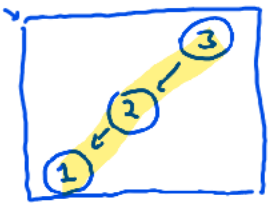
buscar_lineal(9) → $O(n)$

consultas = 9

Complejidad



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



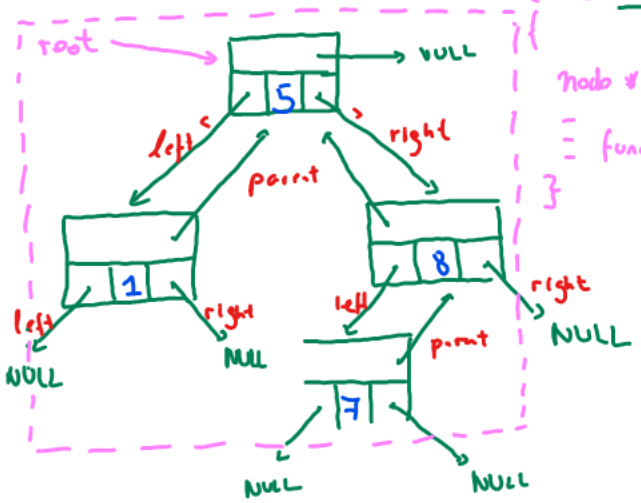
1. Inserta (3)
2. Inserta (2)
3. Inserta (1)

Árbol Lineal — nó balanceado.

Soluc. AVL, RBTree.

Altura :

	max
0	1
1	3
2	7
⋮	⋮
h	$h^{2+1}-1$




```
class bst
{
  nodo * root
  = func =
};
```

Recorrido Árbol (walk)

2 + 1 — infija
+ 2 1 — prefija
2 1 + — postfija

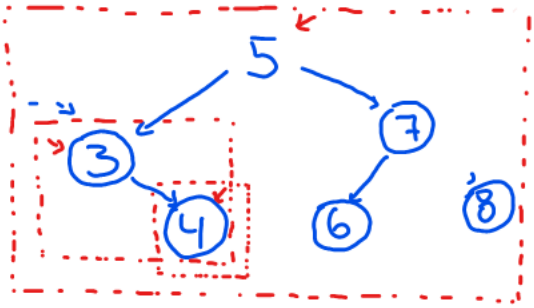
inorder left, key, right 2+1
preorder key, left, right +21
postorder left, right, key 1+

recursiva

```
print_inorder(x)
  if(x != NULL)
    print_inorder(x.left);
    cout << x.key;
    print_inorder(x.right);
```

1 2 1
3 1 2
2 3 3

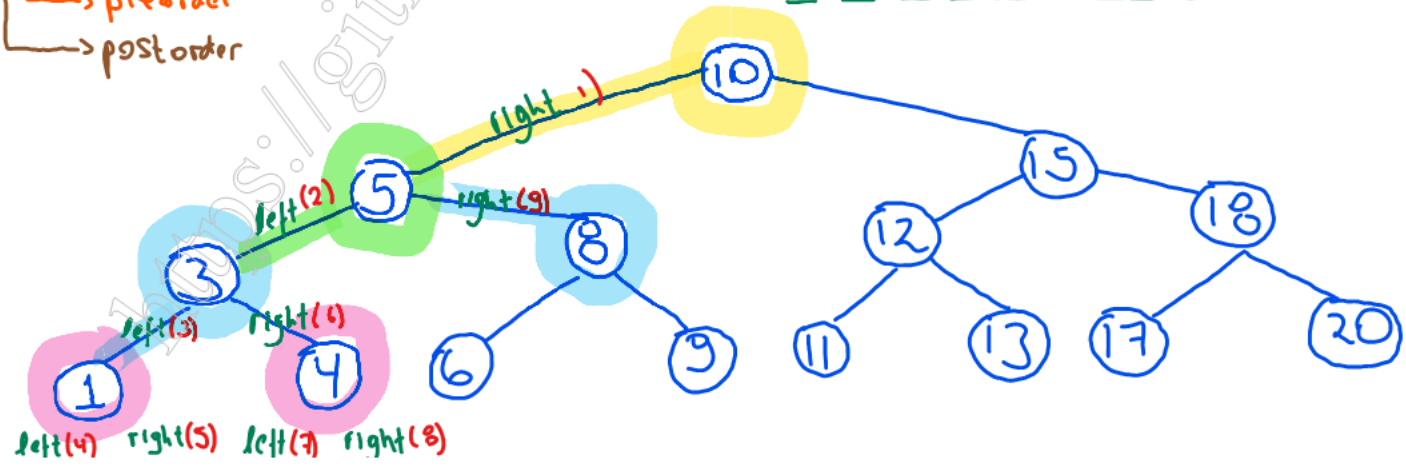
- inorder: de forma ordenada.
- preorder
- postorder




```
bst arbol;
arbol.print_();
```

```
bst::print_inorder( )
  print_inorder(root);
```

← interface



Pila

print_inorder(10), left

print_inorder(5), left, cout, right

print_inorder(3), left, cout, right

print_inorder(1), left, cout, right

print_inorder(NULL)

print_inorder(NULL)

print_inorder(4), left, cout, right

print_inorder(NULL)

print_inorder(NULL)

print_inorder(8)...

: 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)
```

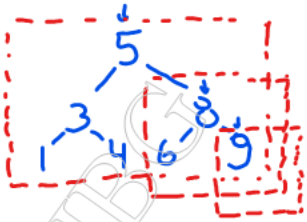
```
tree_node *search_iterative(int k)
    tree_node *x = root;
    while (x != NULL and x.key != k)
        if (k < x.key)
            x = x->left;
        else
            x = x->right;
    return x
```

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

print_node

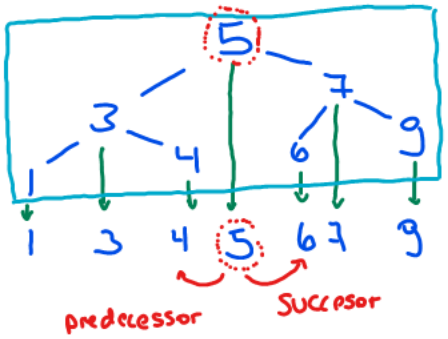
```
tree_node *max()   (min)
    tree_node *x = root;
    while (x.right != NULL)   (left)
        x = x.right;
    return x;
```

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



Successor y predecessor:

Dado bst, el successor (predecessor) se def. como:
- El nodo siguiente (anterior) en un recorrido inorder.
- El nodo con el valor menor (mayor) más (menos) grande que x.key.



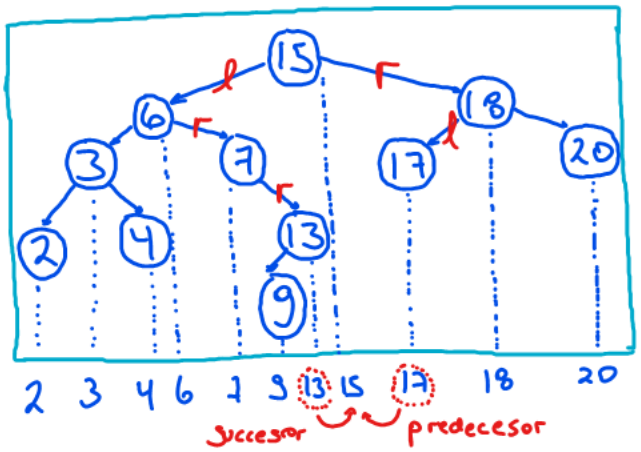
Dos Casos

si right = NULL (left) → successor(x) / predecessor(x)

si right ≠ NULL (left) → Es el min (max) de x.right (left)

Es el menor (mayor) antecesor cuyo hijo izquierdo (derecho) es antecesor de x.

≈ recorrer hacia arriba hasta el primer nodo cuyo hijo izquierdo (derecho) es antecesor de x.



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

13: Succesor — 17: predecesor

```
tree_node *Succesor(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)   (left)
            x = y
            y = y.parent
        return y
```

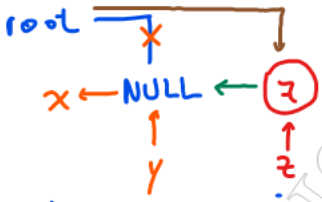
Mientras recorro hacia arriba y el antecesor es valido pero es hijo derecho (izquierdo).

Inserción

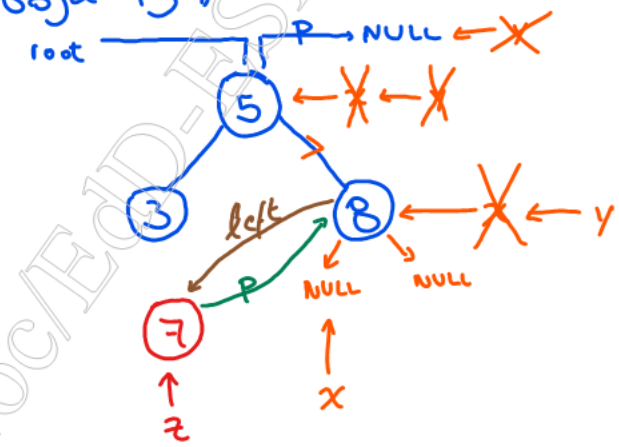
- Al igual que en los procedimientos de búsqueda (binaria).
- Empezamos en la raiz del árbol y descendemos hasta encontrar un hijo NULL para reemplazarlo.

```
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)
        y = x
        if(z.key < x.key)
            x = x.left
        else
            x = x.right
    z.parent = y
    if(y == NULL)
        root = z // árbol vacio
    else if(z.key < y.key)
        y.left = z
    else
        y.right = z
```

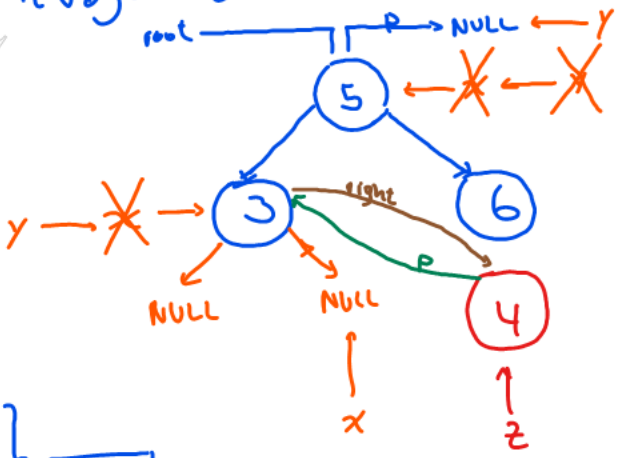
- Árbol vacio



- Hoja izquierda



- Hoja derecha.



Mnemotecnia: ("lógica")

1.- x apunta al contenido NULL
y apunta al padre de x
mientras x != NULL

→ Caso base

2.- hijo(nuevo) reconoce al padre(y)

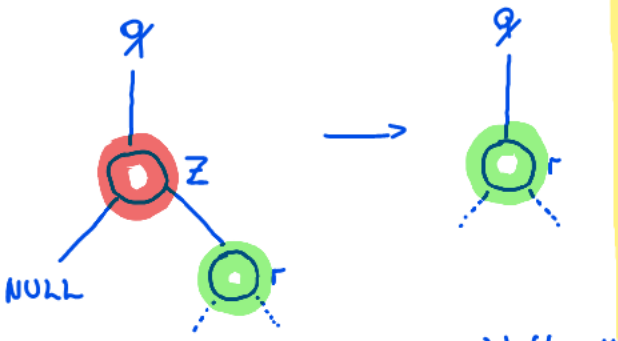
3.- padre(y) reconoce al hijo(nuevo).

- left <
- right >

x → root
y → NULL

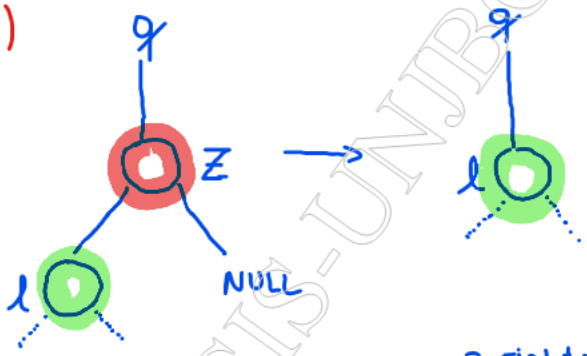
Eliminación: ← nodo a eliminar ← reemplazo

(a)



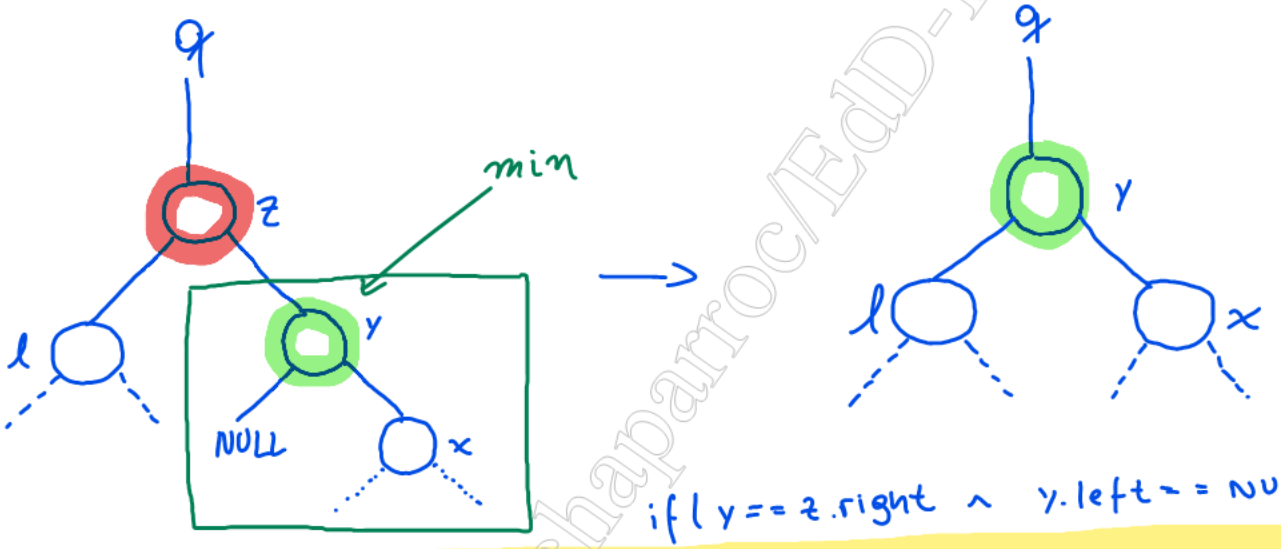
z.left = NULL

(b)



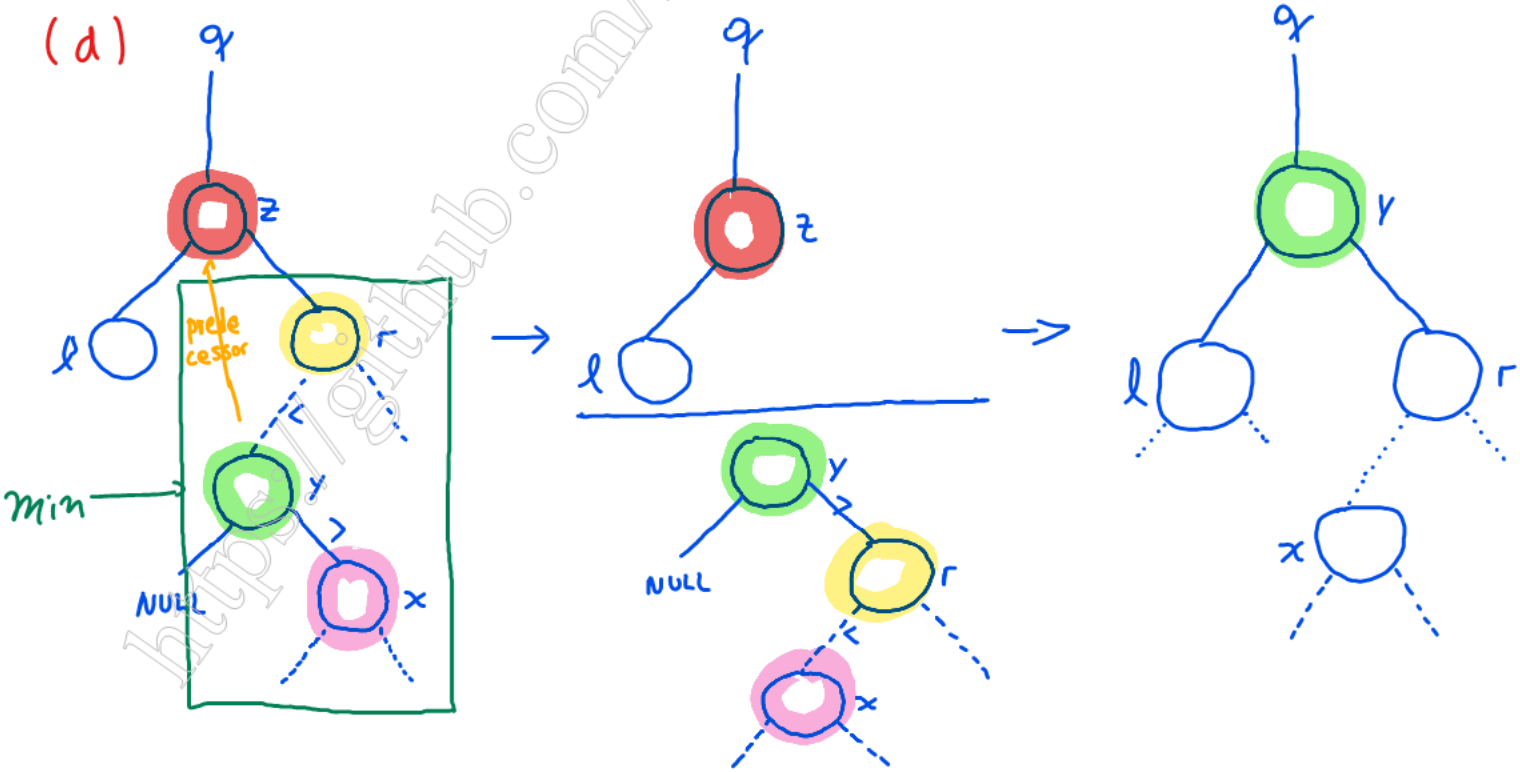
z.right = NULL

(c)



if(y == z.right ∧ y.left == NULL)

(d)



Transplante:

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

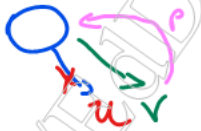
```
transplant(u,v)
if(u.parent == NULL)
    root = v;
else if(u == u.parent.left)
    u.parent.left = v;
else
    u.parent.right = v;
if(v != NULL)
    v.parent = u.parent
```

vacío

padre reconoce hijo

hijo reconoce padre

u = antiguo, v = nuevo




```
remove(z)
if(z.left == NULL)                    (a)
    transplant(z, z.right);
else if(z.right == NULL)              (b)
    transplant(z, z.left);
else
    y = min(z.right)                  (c)
    if(y != z.right)                  (d)
        transplant(y, y.right);
        y.right = z.right
        y.right.parent = y
    transplant(z, y)
    y.left = z.left                   reconozca a los huerfanos
    y.left.parent = y                 los huerfanos reconozcan al padre
delete z
```

End