# Árboles

EEDD - GRADO EN ING. INFORMATICA UCO

## Árboles binarios de búsqueda

# Changelog

17/3/25:

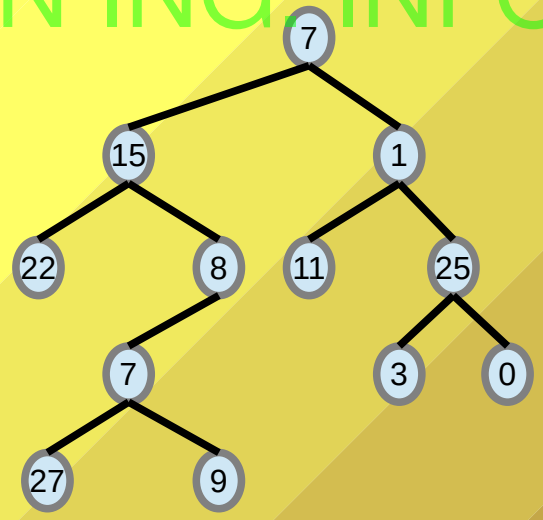
- Actualizada post-cond. del BSTree[T]::create(v:T).
- Reformateados algunos alogoritmos para mantener coherencia.

EEDD - GRADO EN ING. INFORMATICA UCO

# Motivación

- Inconvenientes de los Árboles binarios.
  - Dada una secuencia de entrada no está definido cómo crear el árbol correspondiente.
  - La complejidad de la búsqueda de un item en el árbol es O(N).






¿Dónde insertar 4?
¿Está el valor 3?

# Contenidos

- Árboles binarios de búsqueda.
  - Definición.
  - Operación de búsqueda.
  - Operación de inserción.
  - Operación de borrado.

# Árboles binarios de búsqueda

- Definición:
  - Es un árbol binario que mantiene la invariante:
    - En cada subárbol, la raíz es mayor que todos sus descendientes propios izquierdos y menor que todos sus descendientes propios derechos.

# Árboles binarios de búsqueda

- Ejemplo
  - Crea un BSTree que almacene la secuencia de valores: {10,5,20,1,9,15,25,7,22,27,6,8}

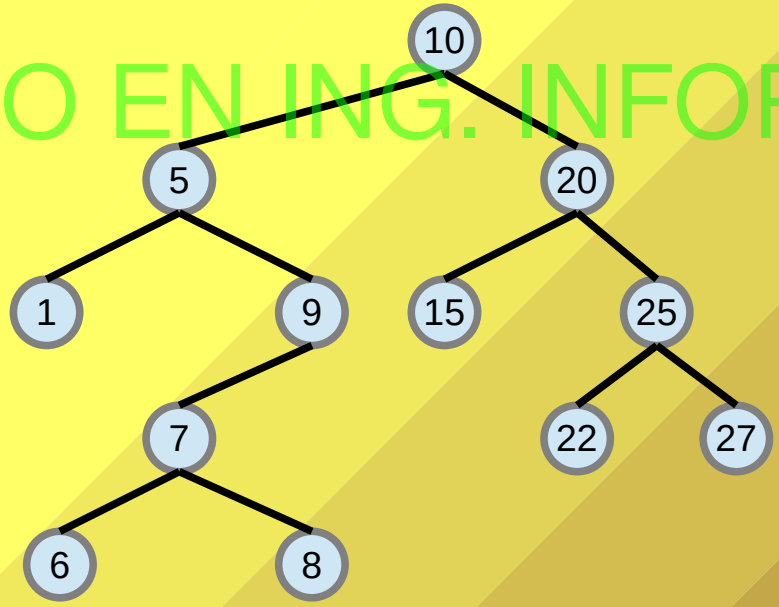


**¿Cuál sería el recorrido “en orden”?**

# Árboles binarios de búsqueda

- Ejemplo
  - Crea un BSTree que almacene la secuencia de valores: {10,5,20,1,9,15,25,7,22,27,6,8}






**¿Cuál sería el recorrido "en orden"?: 1 5 6 7 8 9 10 15 20 22 25 27**

# Árboles binarios de búsqueda

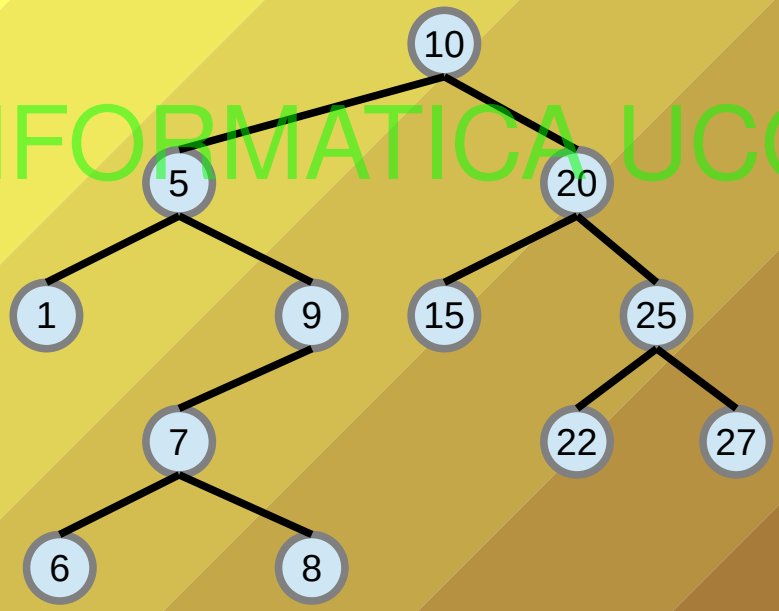
- Inserción en orden usando un árbol Binario.

```
Algorithm insert(Var t:BTree; v:T)
Begin




End.
```





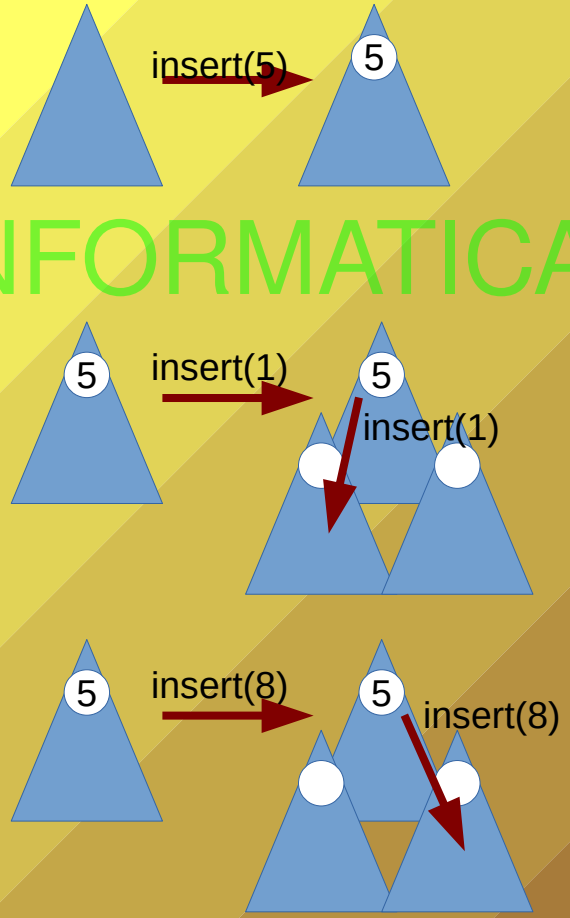
# Árboles binarios de búsqueda

- Inserción en orden usando un árbol Binario.

```
Algorithm insert(Var t:BTree; v:T)
Begin
  If t.is_empty() Then      //Case 0
    t.create_root(v)
  Else If v<t.item() Then //Case 1
    If t.left().is_empty() Then
      t.set_left(BTree[T].create(v))
    Else
      insert(t.left(), v)
    End-If
  Else If v>t.item() Then //Case 2
    If t.right().is_empty() Then
      t.set_right(BTree[T].create(v))
    Else
      insert(t.right(), v)
    End-If
  End-If
End.
```

Complejidad
O(  )





# Árboles binarios de búsqueda

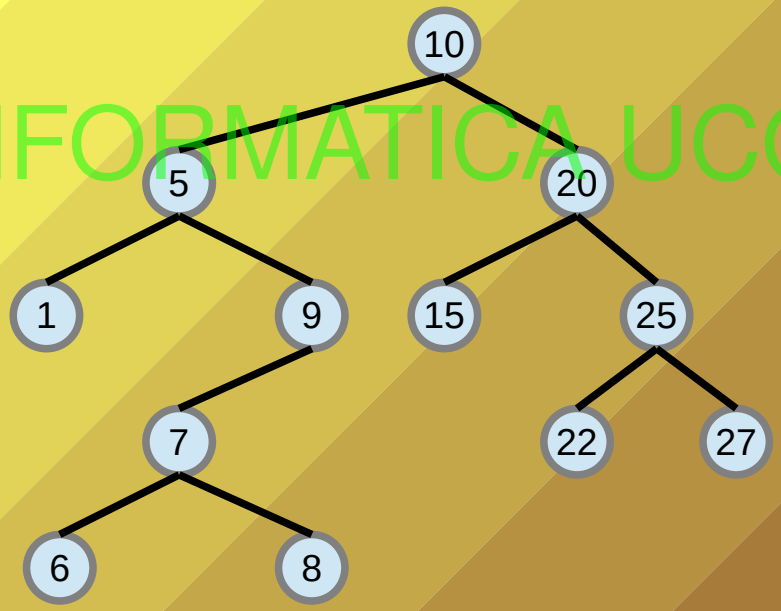
- Búsqueda en orden usando un árbol Binario.

```
Algorithm has(t:BTree, v:T):Bool
Var retVal:Bool
Begin




End.
```

search(8)

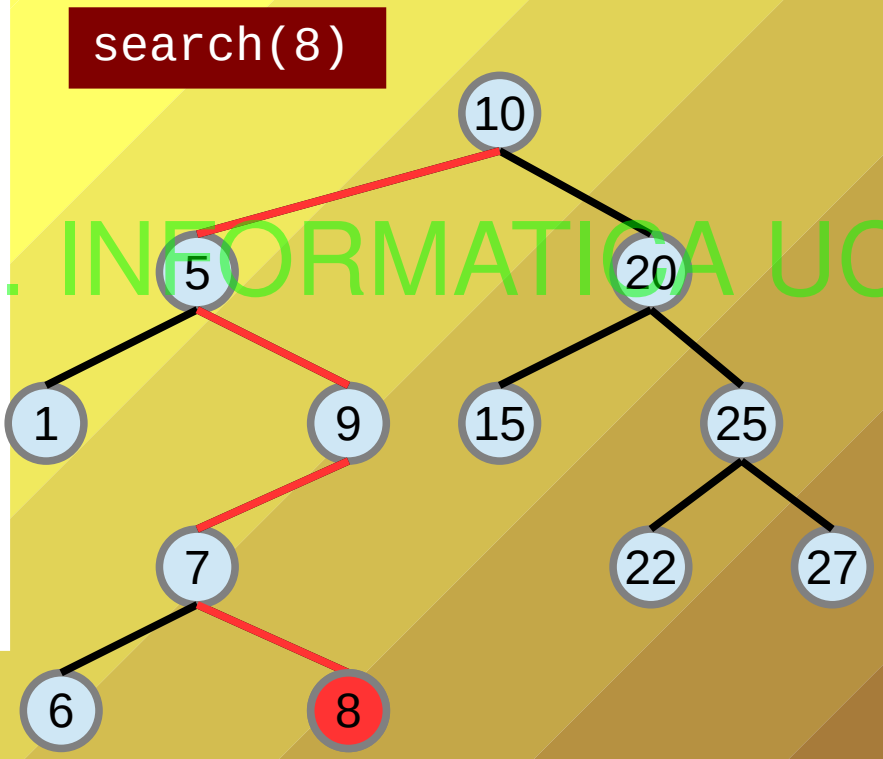


# Árboles binarios de búsqueda

- Búsqueda en orden usando un árbol Binario.

```
Algorithm has(t:BTree, v:T):Bool
Var retVal:Bool
Begin
  retVal ← False
  If Not t.is_empty() Then
     If v<t.item() Then
        retVal ← has(t.left(), v)
     Else If v>t.item() Then
        retVal ← has(t.right(), v)
     Else
        retVal ← True
     End-If
  End-If
  Return retVal
End.
```

Complejidad
O(  )



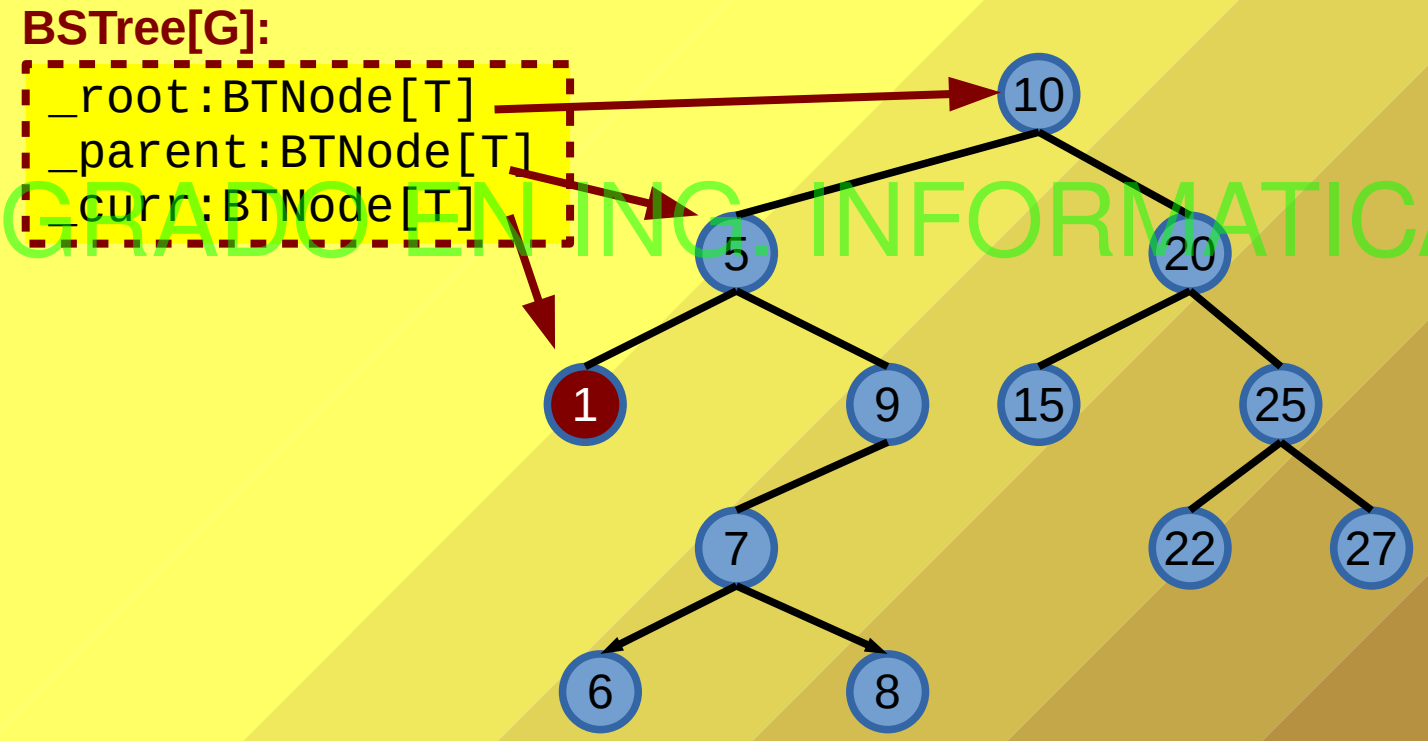
# Árboles binarios de búsqueda

- TAD BSTree[T]: especificación.

  - **Makers:**
    - create() //makes an empty tree.
      - post-c: not currentExists()
    - create(v:T) //makes an leaf tree.
      - post-c: currentExists() and current()=v
  - **Observers:**
    - has(v:T):Bool
    - currentExists():Bool
    - current():T
      - Pre-c: currentExists()
  - **Modifiers:**
    - search(v:T):Bool
      - Post-c:  not retVal or (currentExists() and current()=v)
      - Post-c: retVal or not currentExists()
    - insert(v:T)
      - Post-c: has(v) and current()=v
    - remove()
      - Pre-c: currentExists()
      - Post-c: not has(old.current()) and not currentExists()
    - **Invariant**:
      - isEmpty() or “in-fix traversal makes a sorted sequence of items”.

# Árboles binarios de búsqueda

- TAD BSTree[T]: diseño

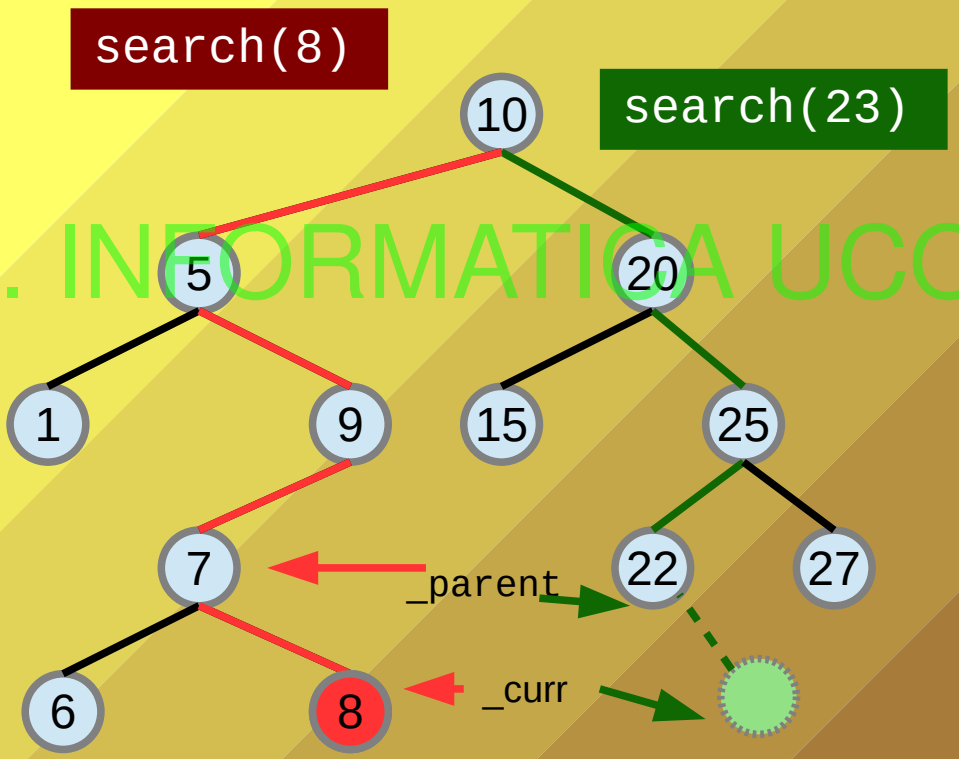


# Árboles binarios de búsqueda

- Search:

```
BSTree[T]::search(it:T): Boolean
var:
  found:Boolean
begin
  found ← False
  _curr ← _root
  _parent ← Void
  while _curr <> Void and not found do
    if _curr.item() = it then
      found ← True
    else
      _parent ← _curr
      if _curr.item() > it then
        _curr ← _curr.left()
      else
        _curr ← _curr.right()
      endif
    endif
  end-while
  return found
end.
```





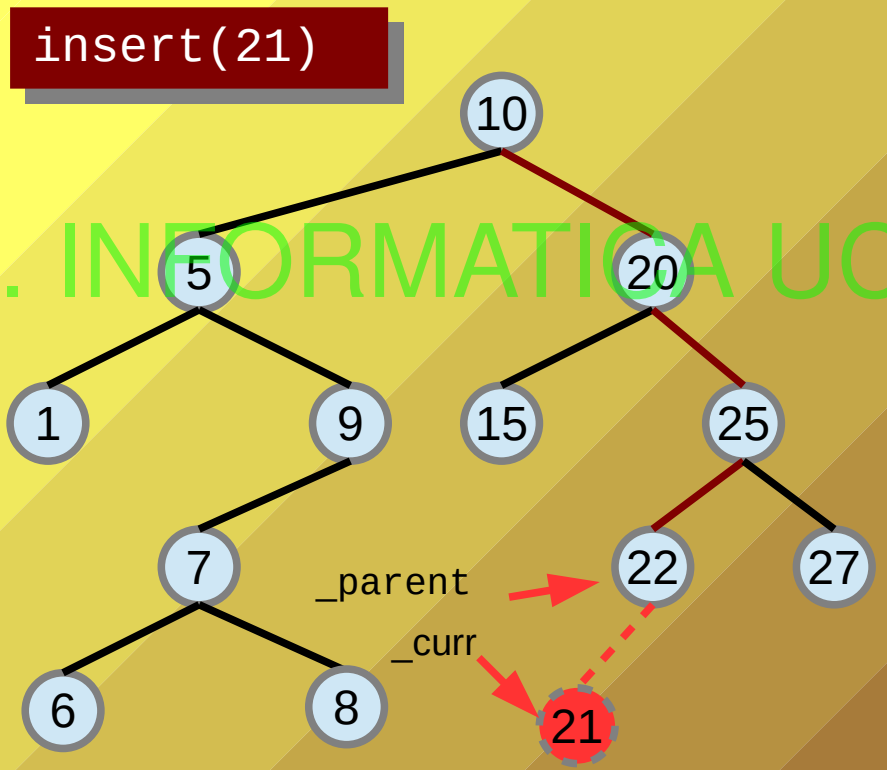

**¿Cuál es la complejidad algorítmica?**

# Árboles binarios de búsqueda

- Insert:

```
BSTree[T]::insert(it:G)
Begin
  If isEmpty() Then
    create_root(it)
    _curr ← _root
    _parent ← Void
  Else If Not search(it) Then
    _curr ← BTNode(it,Void,Void)
    If (_parent.item()>it) Then
      _parent.setLeft(_curr)
    Else
      _parent.setRight(_curr)
    End-If
  End-If
End.
```

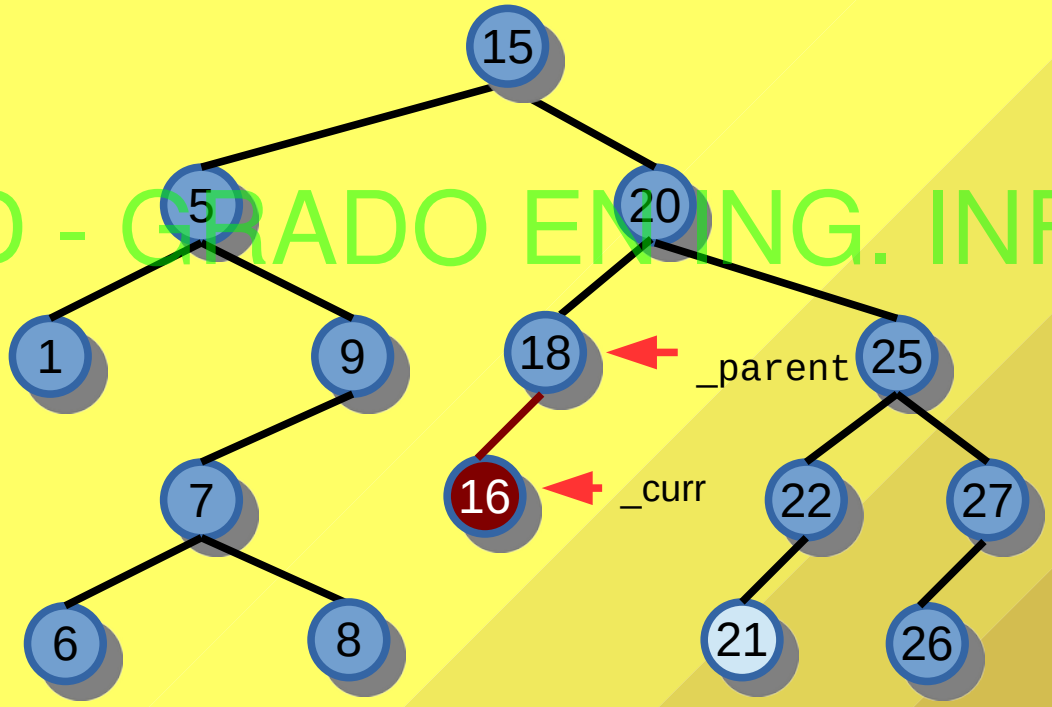





**¿Cuál es la complejidad algorítmica?**

# Árboles binarios de búsqueda

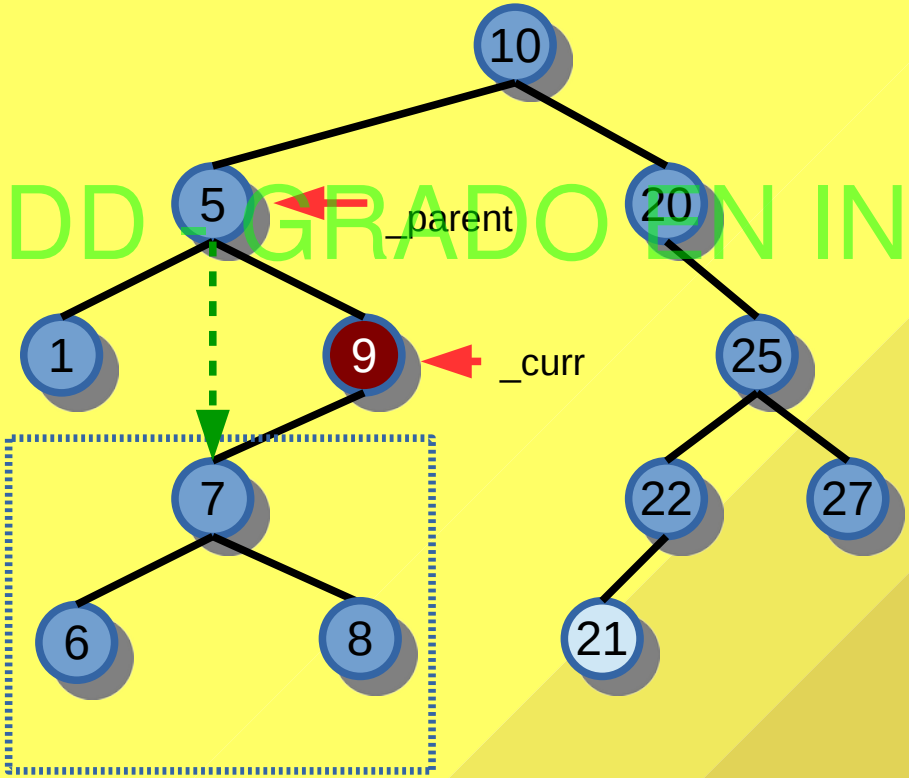
- Remove: Caso 0.




```
search(16)
remove() // _curr es una hoja.
If _parent <> Void Then
  If _parent.left()=_cur Then
    _parent.setLeft(Void)
  Else
    _parent.setRight(Void)
Else
  _root ← Void //árbol vacío.
```

# Árboles binarios de búsqueda

- Remove: Caso 1.

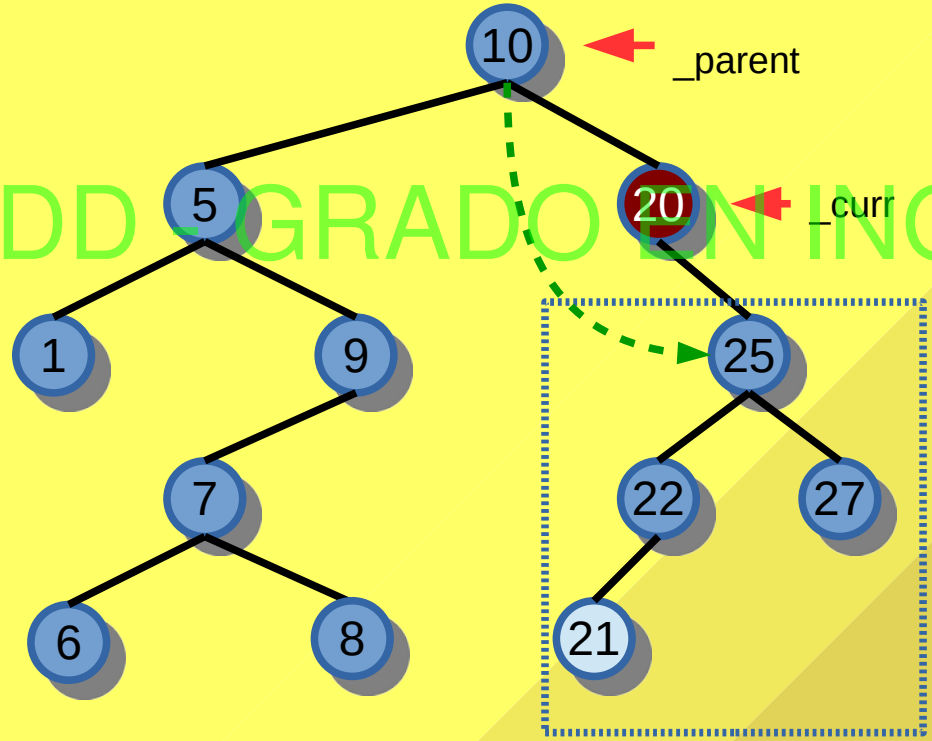



```
search(9)
remove():
//_curr sólo tiene hijo izquierdo.

If _parent <> Void Then
  If _parent.left()=_curr Then
    _parent.setLeft(_curr.left())
  Else
    _parent.setRight(_curr.left())
  End-If
Else
   _root <- _curr.left()
End-If
```

# Árboles binarios de búsqueda

- Remove: Caso 2.

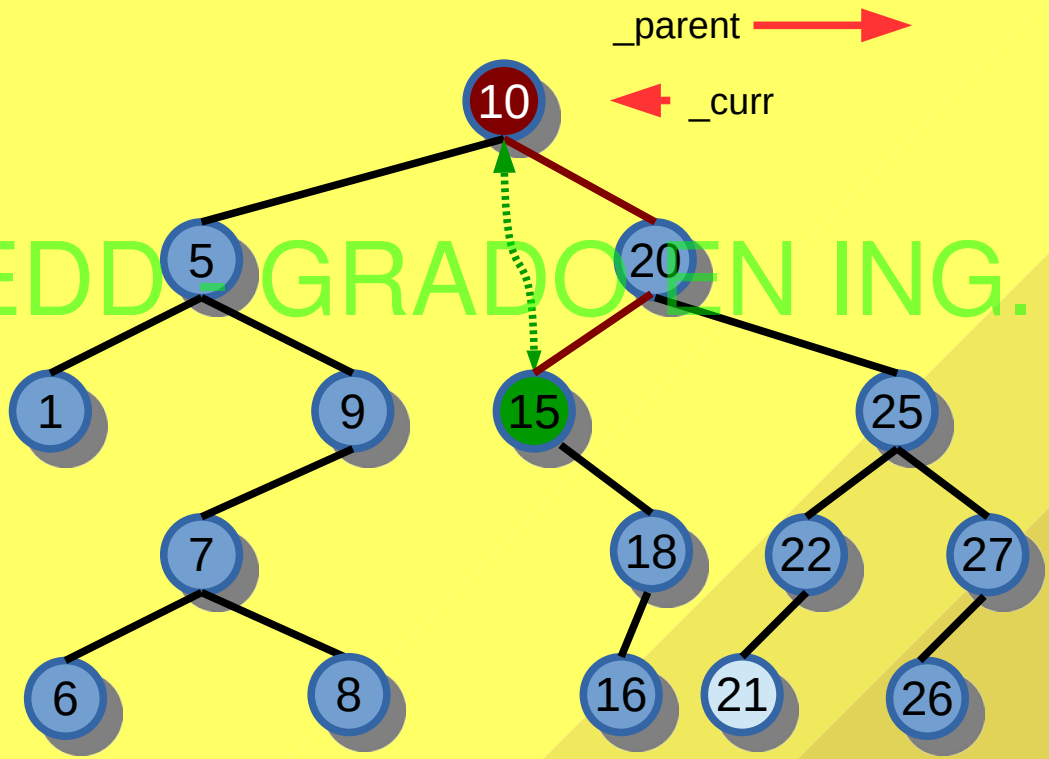



```
search(9)
remove():
//_curr sólo tiene hijo derecho.

If _parent <> Void Then
  If _parent.left()=_curr Then
    _parent.setLeft(_curr.right())
  Else
    _parent.setRight(_curr.right())
  End-If
Else
   _root <- _curr.right()
End-If
```

# Árboles binarios de búsqueda

- Remove: Caso 3 (paso 1).





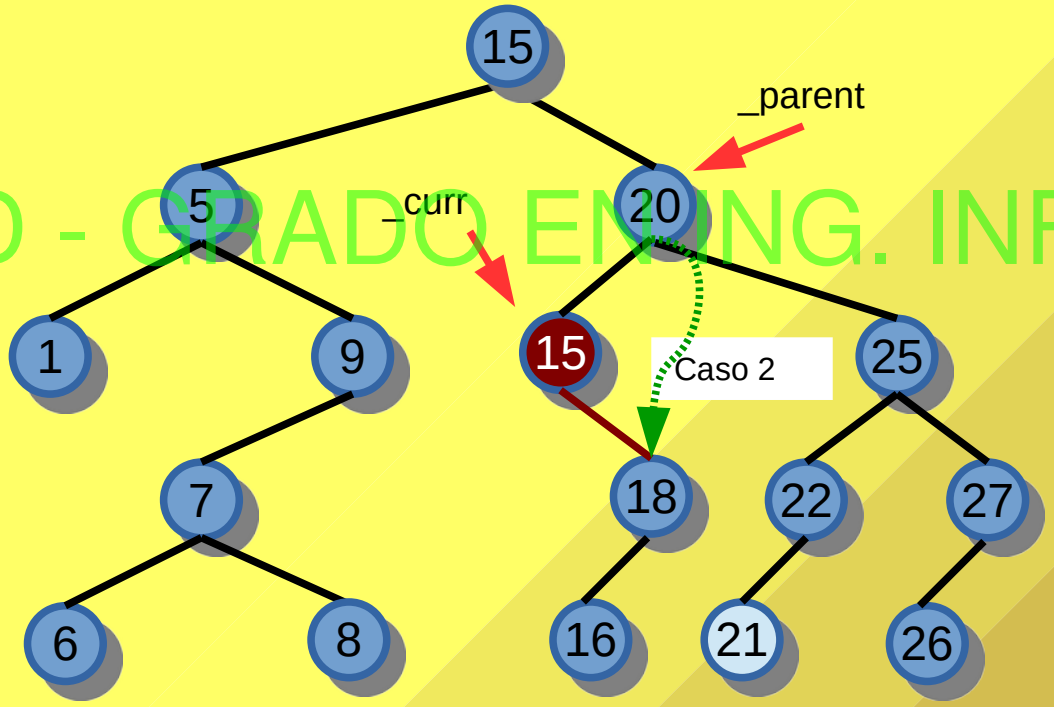

search(10)
remove():
_curr tiene dos hijos.
Mover el cursor al sucesor (predecesor) en orden.
Actualizar la clave con el sucesor (predecesor).
Eliminar el nodo sucesor (predecesor).

```
findInOrderSuccessor():
Pre-c: hasRight()
Begin
  _parent ← _curr
  _curr ← _curr.right()
  While _curr→hasLeft() Do
    _parent ← _curr
    _curr ← _curr.left()
  End-While
End.
```

O(  )

# Árboles binarios de búsqueda

- Remove: Caso 3 (paso 2).




```
search(10)
remove()
```

recursión

```
remove()
sólo casos 0 o 2 (1 si
usamos el predecesor en
orden)
```

# Árboles binarios de búsqueda

- Remove:

```
BSTRee[T]::Remove():
Var
  replaceWithSubTree:Boolean #Is there a subtree to use?
  subTree: BSTNode[T] #The root node of the subtree to use.
  tmp: BSTNode[T]
Begin
  replaceWithSubTree ← True
  If _curr.left() = Void And _curr.right()=Void Then
0   subTree ← Void # Use an empty tree.
  Else If _curr.right() = Void Then
1   subTree ← _curr.left() #Use the left subtree.
  Else If _curr.left() = Void Then
2   subTree ← _curr.righ() #Use the right subtree.
  Else
3   replaceWithSubTree ← False #None subtree can be used.

  If repalceWithSubTree Then
    If _parent = Void Then
       _root ← subTree
    Else If _parent.right() = _curr Then
       _parent.setRight(subTree)
    Else
       _parent.setLeft(subTree)
    _curr ← Void
  Else
    tmp ← _curr
    findInOrderSuccessor() #move the cursor to the successor
    tmp.setItem(_curr.item())
    remove()
  End-if
End.
```

O( )

# Árboles Binarios de Búsqueda

- Resumiendo:
  - Son árboles binarios donde la inserción se hace respetando un orden de padres a hijos.
  - El recorrido “en-orden” sigue una secuencia ordenada de nodos.
  - El orden permite localizar un nodo con O(H) en vez de O(N).
  - Las operaciones de inserción/borrado tienen complejidad O(H).
  - La mejora tiene sentido si $H \approx Log_2(N)$.

# Referencias

- Lecturas recomendadas:
  - Caps. 10, 11 y 12 de “Estructuras de Datos”, A. Carmona y otros. U. de Córdoba. 1999.
  - Caps 9 y 13.5 de *“Data structures and software develpment in an object oriented domain”*, Tremblay J.P. y Cheston, G.A. Prentice-Hall, 2001.
  - Wikipedia:
    - Binary Search Tree: en.wikipedia.org/wiki/Binary_search_tree