

# Árboles

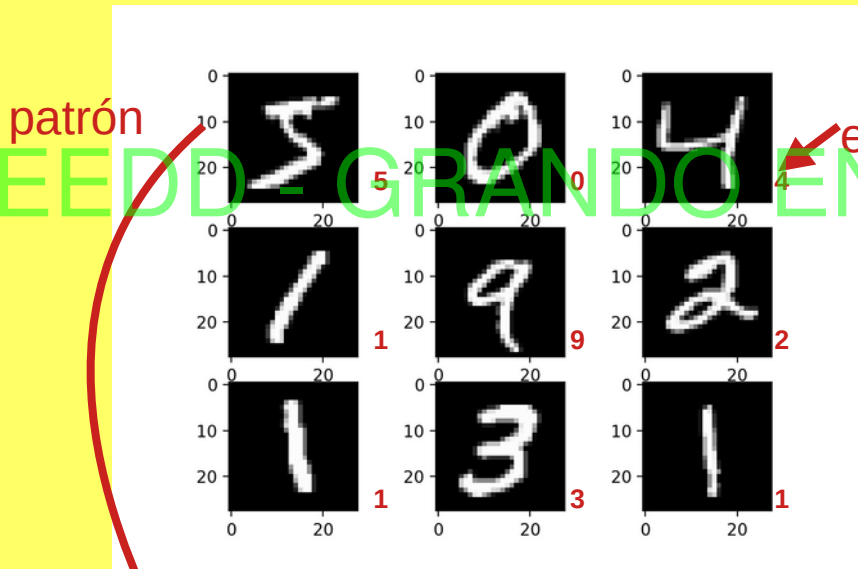
EEDD - GRANDO EN ING. INFORMÁTICA - UCO

Árbol binario

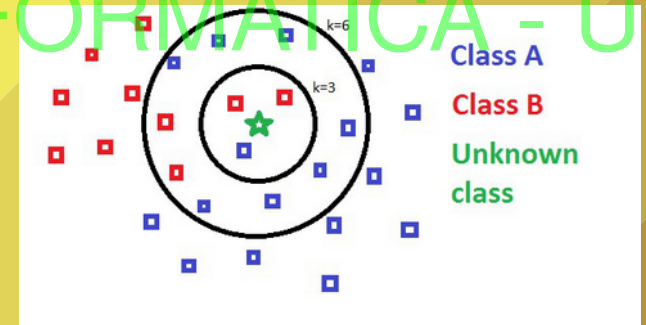
# Motivación

- El problema del vecino más cercano.

Dataset anotado



¿qué etiqueta tendrá?



Usando un array de [patrones| etiquetas] para representar el Dataset, ¿qué complejidad tendría buscar el vecino más cercano? ¿se podrá hacer mejor?

16x16 = 256 dimensiones  
0010000100111001 ... 01100

# Contenidos

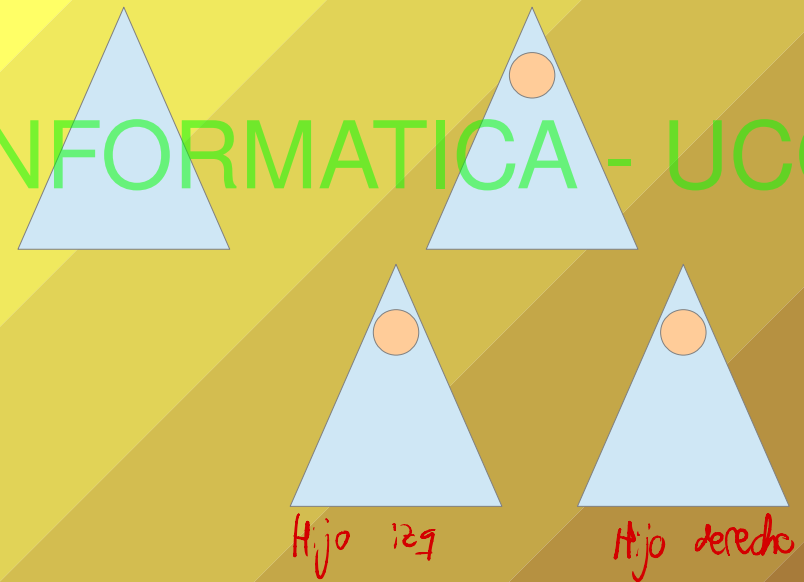
- **Definición de Árbol Binario.**
- Especificación del TAD BinaryTree.
- Implementación enlazada.
- Recorridos.
- Plegado/desplegado.

EEDD - GRANDO EN ING. INFORMÁTICA - UCO

# Árboles binarios

Grado  $m_{\max} = 2$

- Es un árbol vacío o un árbol que almacena un ítem de datos y dos sub árboles binarios .
- ADT BinaryTree[T]:
  - **Makers:**
    - create():BinaryTree[T] //Makes an empty binary tree.
      - Post-c: isEmpty()
    - create(v:T):BinaryTree[T] //Makes an binary tree (leaf).
      - Post-c: not isEmpty()
      - Post-c: item()==v
      - left().isEmpty()
      - right().isEmpty()
  - **Observers:**
    - Bool isEmpty() //is an empty tree?
    - item ():T //The root's item.
      - Pre-c: not isEmpty()
    - left():BinaryTree[T] // the left subtree.
      - Pre-c: not isEmpty()
    - right():BinaryTree[T] //the right subtree.
      - Pre-c: not isEmpty()

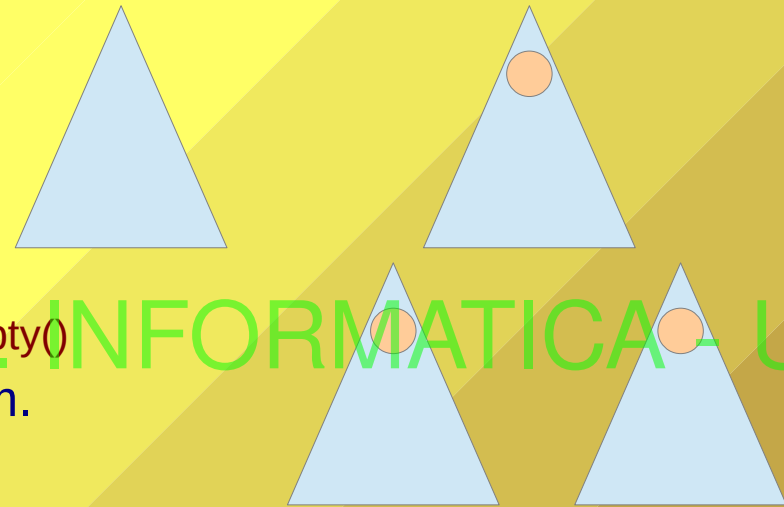


# Árboles binarios

- ADT BinaryTree[T]:

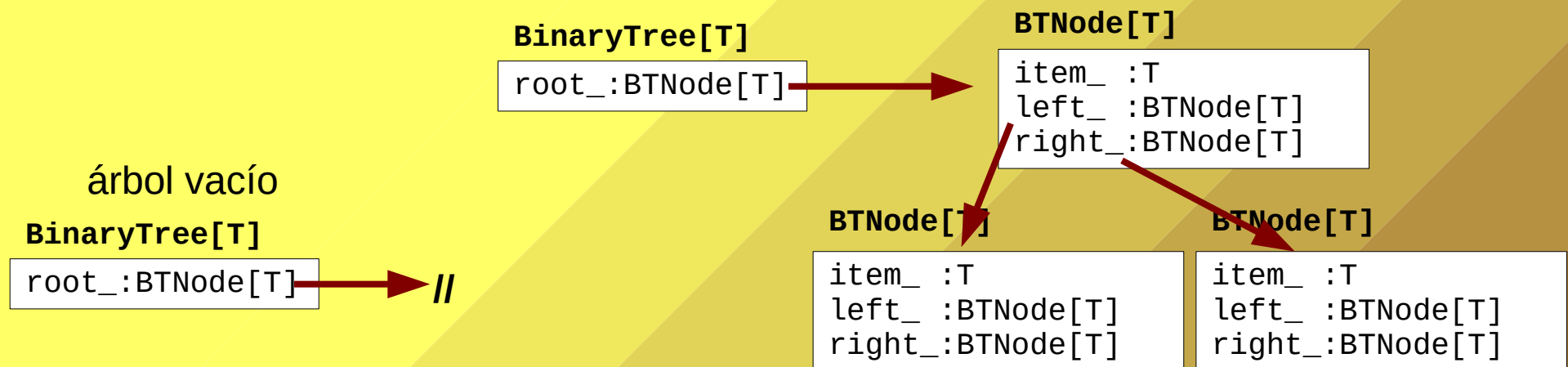
- **Modifiers:**

- createRoot(v:T) //create the Root.
      - Pre-c: isEmpty()
      - Post-c: not isEmpty() and item()==v
      - Post-c: left().isEmpty() and right().isEmpty()
    - setItem(v:T) //Set the root's data item.
      - Pre-c: not isEmpty()
      - Post-c: item()==v
    - setLeft(t:BinaryTree[T]) //attach a tree as left child.
      - Pre-c: not isEmpty()
      - Post-c: left()==t
    - setRight(t:BinaryTree[T]) //attach a tree as right child.
      - Pre-c: not isEmpty()
      - Post-c: right()==t



# Árboles binarios

- Diseño con nodos enlazados.
  - Usar un TAD auxiliar `BTNode[T]`:
    - `BTNode` para modelar los nodos del árbol.
    - `BinaryTree` para modelar un árbol.



# Árboles binarios

- **ADT BTreeNode[T]:**

- **Constructors:**

- create(item:T) //Create a leaf node.
      - Post-c: item() = item
      - Post-c: left() = Void
      - Post-c: right()=Void

- **Observers:**

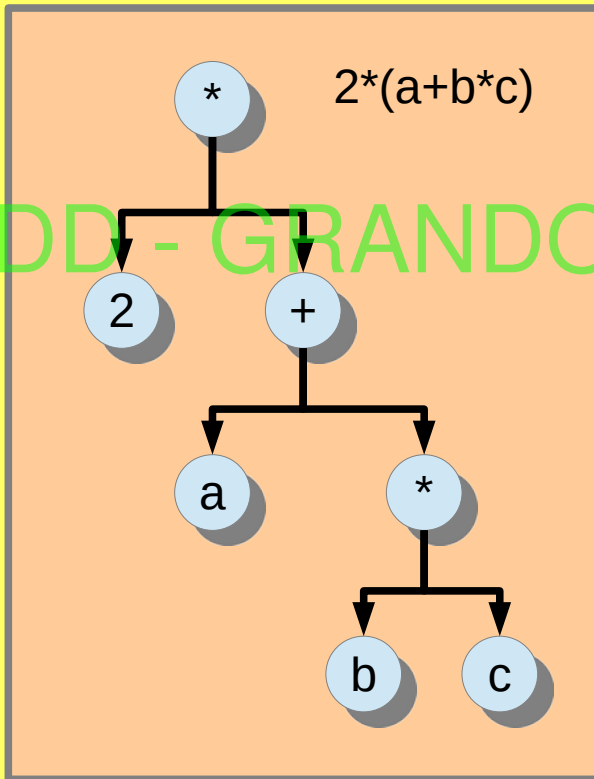
- item():T //O(1)
    - left():BTreeNode[T] //O(1)
    - right():BTreeNode[T] //O(1)

- **Modifiers:**

- setItem(item:T) //O(1)
      - Post-c: item()=item
    - setLeft(l:BTreeNode[T]) //O(1)
      - Post-c: left()=l
    - setRight(r:BTreeNode[T]) //O(1)
      - Post-c: right()=r

# Árboles binarios

- Fold.



```
tree : []
tree : [' T tree tree ' ]
      p  izq  der
```

Ejemplo:

```
"['*']['2']['+']['a']['*']['b']['c']"]
```

¿Un algoritmo para plegar esto?

**Algorithm** BinaryTree[T]::fold (out:Stream)

**Begin**

**If** isEmpty() **Then**  
    out.write('[]')

**Else**

    out.write('[ ')  
    item().fold(out)  
    out.write(' ')  
    left().fold(out)  
    out.write(' ')  
    right().fold(out)  
    out.write(' ]')

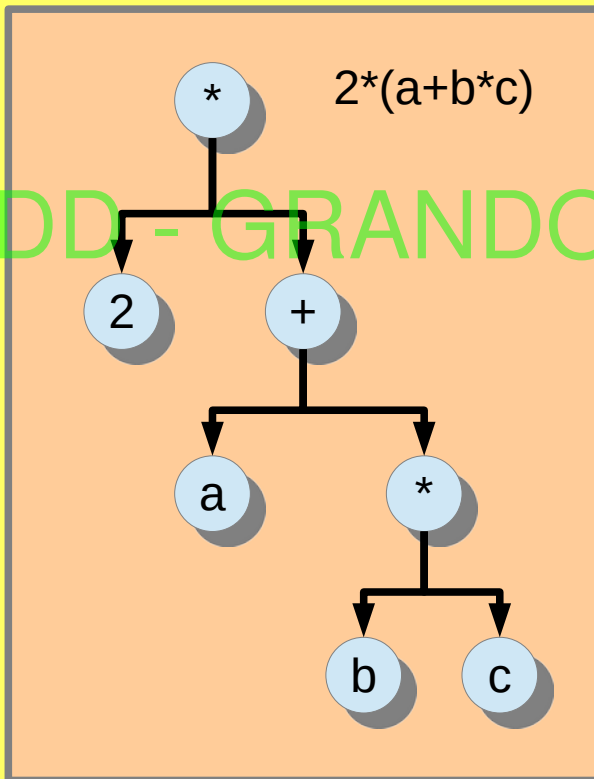
**End-If**

**End.**



# Árboles binarios

- Unfold.



```
tree : []  
tree : [' T tree tree ' ]
```

**Ejemplo:**

```
“[ '*' [ '2' [] ] [ '+' [ 'a' [] ] [ '*' [ 'b' [] ] [ 'c' [] ] ] ]”
```

**Algorithm** BinaryTree[T]:create(in:Stream): BinaryTree[T]

**Var** token: String, t:BinaryTree[T]

**Begin**

t ← BTree[T]:create() #make an empty tree.

in.read(token)

**If** token = '[' **Then**

t.createRoot( T::create(in) )

t.setLeft(BinaryTree[T]:create(in))

t.setRight(BinaryTree[T]:create(in))

in.read(token)

**If** token <> ']' **Then**

**ERROR**

**Else If** token <> '[' **Then**

**ERROR**

**End-If**

**Return** t

**End.**

# Árboles binarios

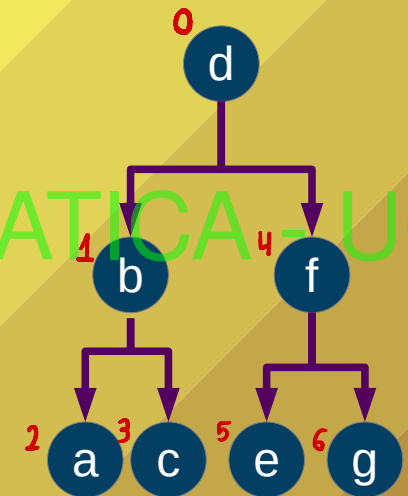
- Recorridos en profundidad: Pre-orden

```
preorderTraversal(t:BinaryTree, var p:Process):Bool
Begin
  retV <- True
  If not t.isEmpty() Then
    retV <- p.apply(t.item())
    retV <- retV And preorderTraversal(t.left(),p)
    retV <- retV And preorderTraversal(t.right(),p)
  End-If
  Return retV
End.
```

*if (true - true) = true  
if (false - true) = false*

**Process como C++ lambda:**  
auto p = [&out](T const& value){  
 out << value << ' '; return true;}  
**Process como un C++ "Functor":**  
struct OutputProcess {  
 OutputProcess(std::ostream& out): out\_(out) {}  
 bool operator()(T const& value) {  
 out\_ << value << ' '; return true;  
 }  
 std::ostream& out\_  
};

*padre-izquierdo-derecho (abajo)*



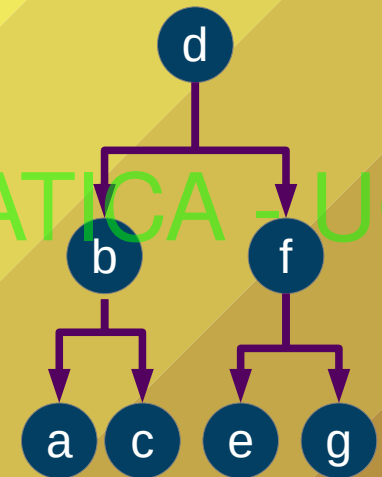
¿preorder?:  
dbacfeeg

# Árboles binarios

- Recorridos en profundidad: En-orden.

```
inorderTraversal(t:BinaryTree, var p:Process):Bool
Begin
  retV <- True
  If not t.isEmpty() Then
    retV <- inorderTraversal(t.left(),p)
    retV <- retV And p.apply(t.item())
    retV <- retV And inorderTraversal(t.right(),p)
  End-If
  Return retV
End.
```

izq-padre-derecho (abajo)

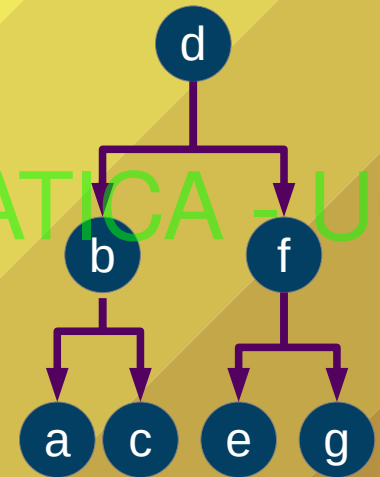


# Árboles binarios

- Recorridos en profundidad: Post-orden.

```
postorderTraversal(t:BinaryTree, var p:Process):Bool
Begin
  retV <- True
  If not t.isEmpty() Then
    retV <- postorderTraversal(t.left(),p)
    retV <- retV And postorderTraversal(t.right(),p)
    retV <- retV And p.apply(t.item())
  End-If
  Return retV
End.
```

*izquierdo - derecho - padre (abajo)*



# Árboles binarios

- Recorridos en anchura.

```
breadthFirstTraversal(t:BinaryTree[G], var p:Process):Bool O(n)
```

```
Var
```

```
q      : Queue[ BinaryTree[G] ] almacena binary tree
```

```
keep_going: Bool
```

```
subtree: BinaryTree[G]
```

```
Begin
```

```
keep_going ← True
```

```
q.enqueue(t) encolo árbol
```

```
While Not q.isEmpty() And keep_going Do
```

```
    subtree ← q.front() coge 1er árbol entero
```

```
    q.dequeue()
```

```
    If not subtree.isEmpty() Then
```

```
        keep_going ← p.apply(subtree.item()) se convierte en padre
```

```
        q.enqueue(subtree.left()) encolo al hijo izquierdo
```

```
        q.enqueue(subtree.right()) encolo al hijo derecho
```

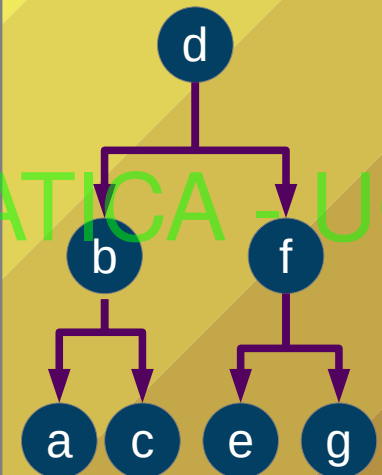
```
    End-If
```

```
End-While
```

```
Return keep_going
```

```
End.
```

*padre-izquierdo-derecho (arriba)*

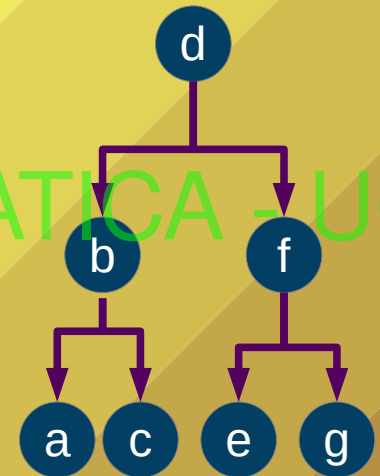


bfs:  
dbfaceg

# Árboles binarios

- Recorrido en profundidad (iterativo). → preorden  
¿postorden?

```
depthFirstTraversal(t:BinaryTree[G], var p:Process):Bool
Var
  s          : Stack[ BinaryTree[G] ]
  keep_going: Bool
  subtree: BinaryTree[G]
Begin
  keep_going ← True
  s.push(t)
  While Not s.isEmpty() And keep_going Do
    subtree ← s.top()
    s.pop()
    If Not subtree.isEmpty() Then
      keep_going ← p.apply(subtree.item())
      s.push(subtree.right())
      s.push(subtree.left())
    End-If
  End-While
  Return keep_going
End.
```



dfs:  
dbacfeg

# Árboles binarios

- ¿Almacena el árbol un valor dado?.

```
Algorithm has(t:Btree[G], key:G):Bool
//0( )
Begin
  Return Not xxxxxxTraversal(t,
    NotEqual(key))
End.
```

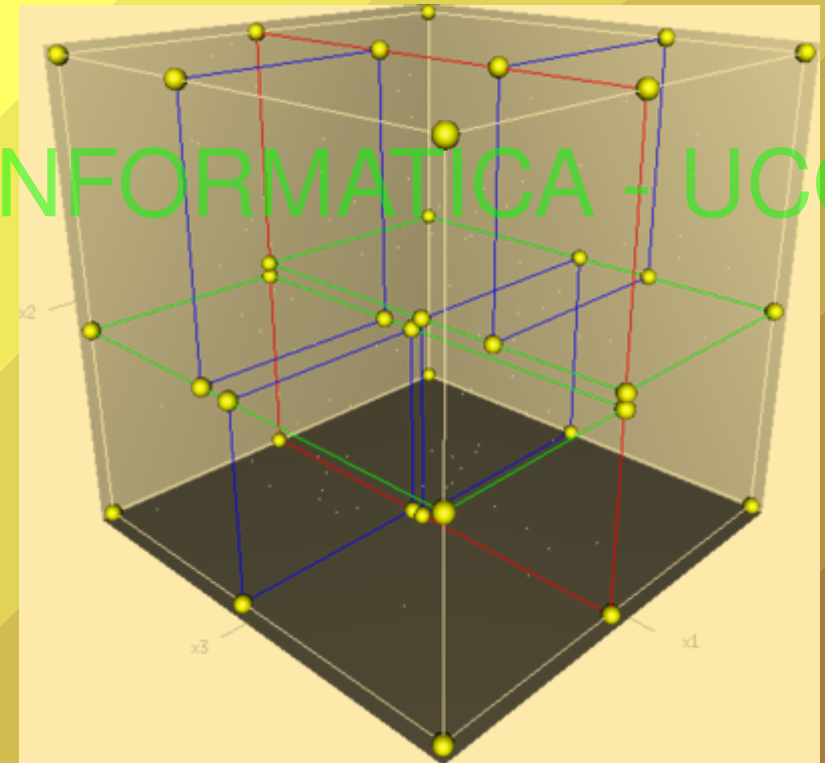
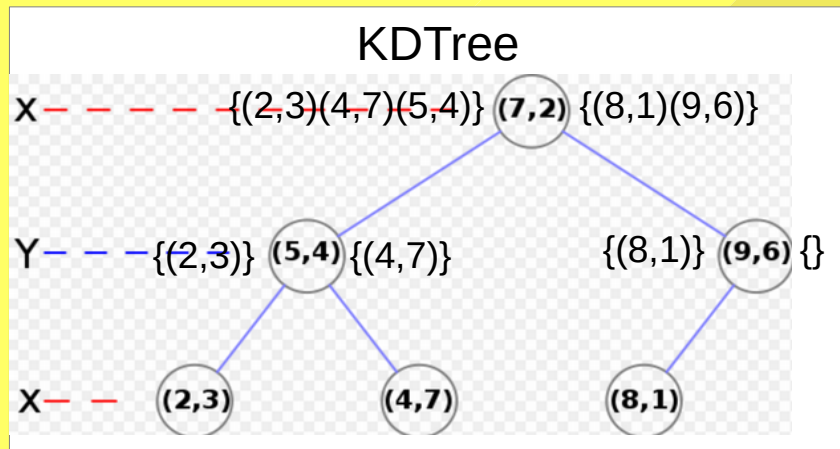
```
ADT NotEqual extend Process:
create(ref:T)
  post-c: refValue()=ref
refValue():T //Gets the value to be compared with.

apply(v:T):Bool
Begin
  Return refValue()<>v
End.
```



# Motivación

- El problema del vecino más cercano.





# Resumiendo

- El árbol binario es un tipo de árbol que está vacío o tiene a lo sumo dos sub árboles: izquierdo y derecho.
- Se establecen tres tipos de recorridos en profundidad: “pre orden”, “en orden” y “post orden”.
- También se puede recorrer en anchura (por niveles).
- Localizar un ítem es una operación  $O(N)$ .
- Obtener el máximo/mínimo es una operación  $O(N)$ .
- La representación enlazada usa el TAD auxiliar BTNode.
- Resuelve el problema del vecino más cercano con  $O(\log N)$  en promedio (KDTre)

# 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 development in an object oriented domain*”, Tremblay J.P. y Cheston, G.A. Prentice-Hall, 2001.
  - Wikipedia.