Árboles

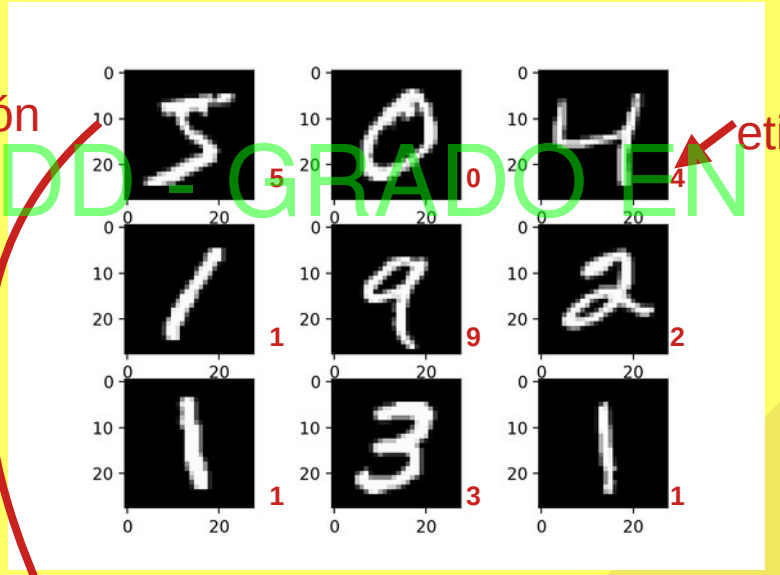
EEDD - GRADO EN ING. INFORMATICA - UCO

Árbol binario

Motivación

- El problema del vecino más cercano.

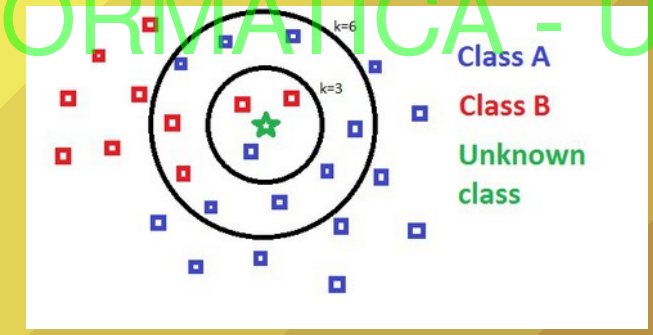
Dataset anotado



16x16 = 256 dimensiones
0010000100111001 ... 01100

¿  ?

¿qué etiqueta tendrá?



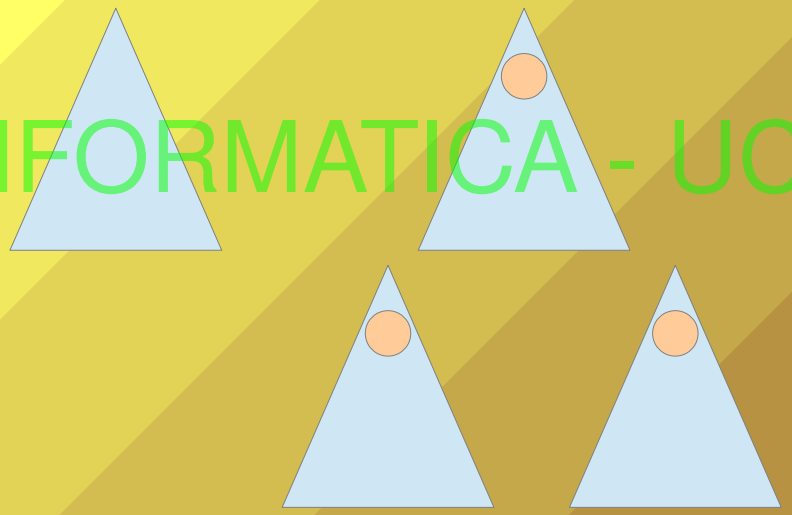
Usando un array de [patrones| etiquetas] para representar el Dataset, ¿qué complejidad tendría buscar el vecino más cercano? ¿se podrá hacer mejor?

Contenidos

- **Definición de Árbol Binario.**
- Especificación del TAD BinaryTree.
- Diseño con nodos enlazados.
- Recorridos.
- Plegado/desplegado.

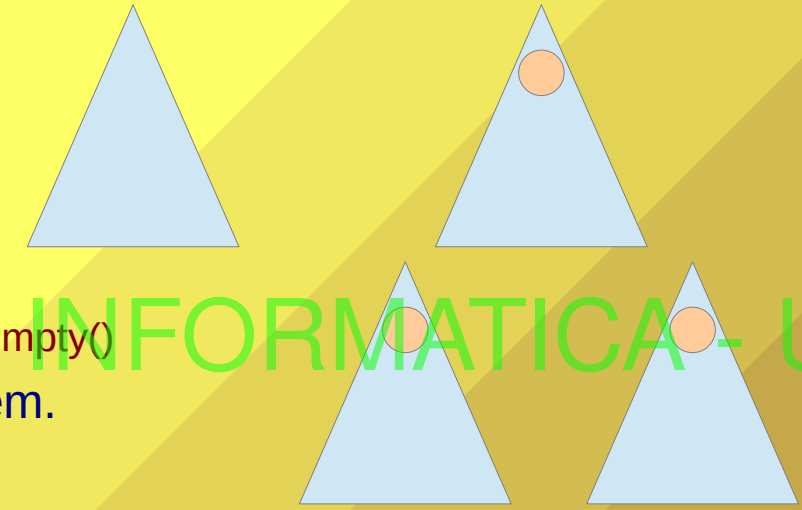
Árboles binarios

- Es un árbol vacío o un árbol que almacena un item 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()



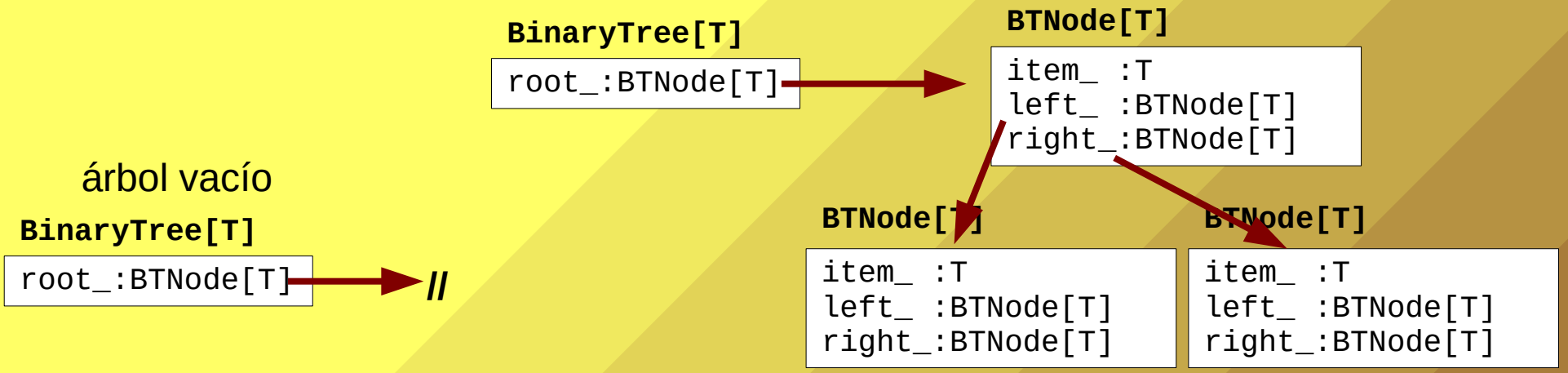
Árboles binarios

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

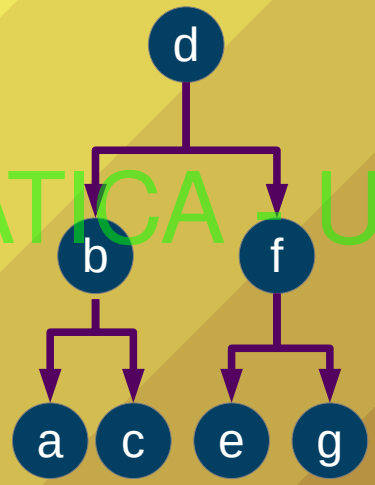
Árboles binarios

- Recorrido del árbol.
  - Recorrido en profundidad.
    - Recorrido recursivo infijo, prefijo y postfijo.
    - Recorrido iterativo infijo.
  - Recorrido en amplitud.

Árboles binarios

- Recorridos en profundidad: prefijo.

```
preorderTraversal(t:BinaryTree[T], var p:Process[T]):Bool
Begin



End.
```

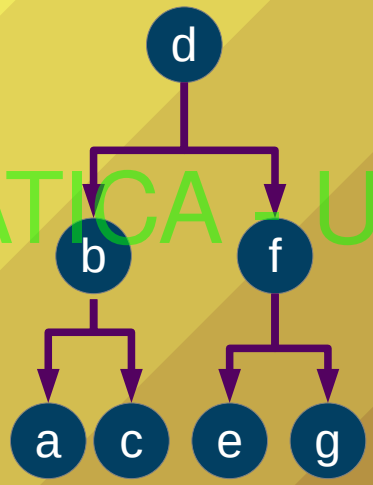

```
ADT Process:
bool apply(v:T) //Realiza un cómputo y devuelve true si el
                  recorrido debe seguir, false si debe parar.
```

¿preorder?:

Árboles binarios

- Recorridos en profundidad: prefijo.

```
preorderTraversal(t:BinaryTree[T], var p:Process[T]):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.
```

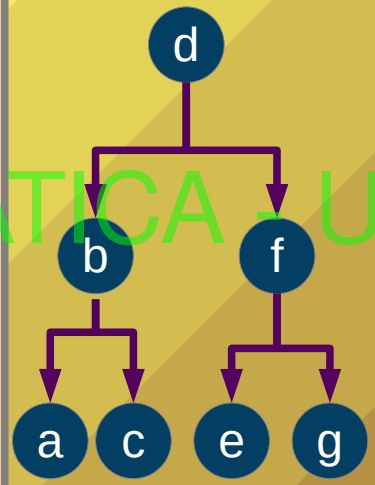

```
Process como C++ lambda:
auto p = [&out](T const& value)-> bool {
   out << value << ' '; return true;}

Process como un C++ "Functor":
class OutputProcess {
  OutputProcess(std::ostream& out): out_(out) {}
  bool operator()(T const& value) {
    out_ << value << ' '; return true;
  }
  std::ostream& out_
};
```

¿preorder?:
dbacfeg

Árboles binarios

- Recorridos en profundidad: infijo.

```
inorderTraversal(t:BinaryTree[T], var p:Process[T]):Bool
Begin




End.
```

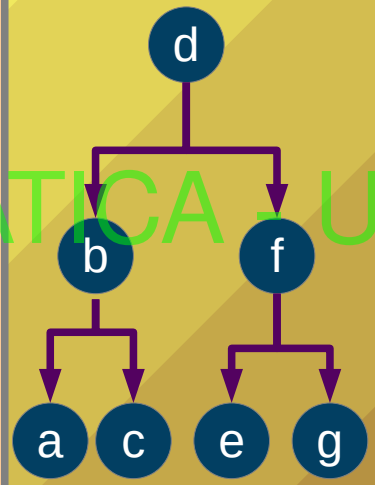




```
¿inorder?:
```

Árboles binarios

- Recorridos en profundidad: infijo.

```
inorderTraversal(t:BinaryTree[T], var p:Process[T]):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.
```

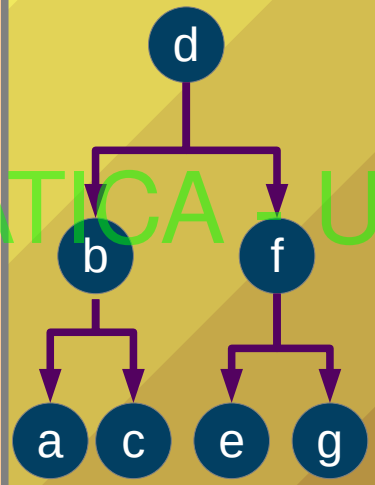




```
¿inorder?:
abcdefg
```

Árboles binarios

- Recorridos en profundidad: postfijo.

```
postorderTraversal(t:BinaryTree[T], var p:Process[T]):Bool
Begin




End.
```

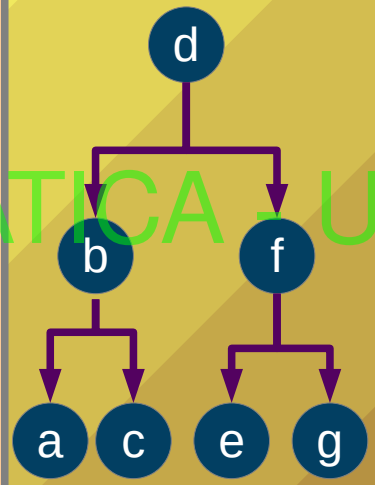




```
¿postorder?:
```

Árboles binarios

- Recorridos en profundidad: postfijo.

```
postorderTraversal(t:BinaryTree[T], var p:Process[T]):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.
```

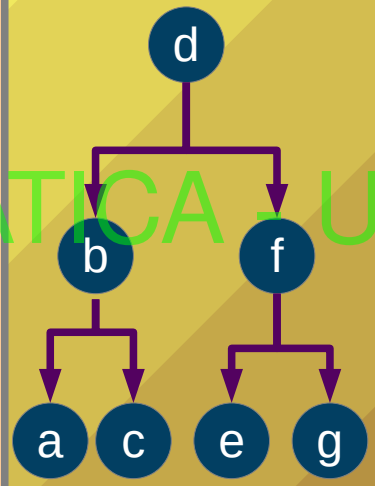



¿postorder?:
acbegfd

Árboles binarios

- Recorridos en anchura.

```
breadthFirstTraversal(t:BinaryTree[T], var p:Process[T]):Bool
Var


Begin









End.
```

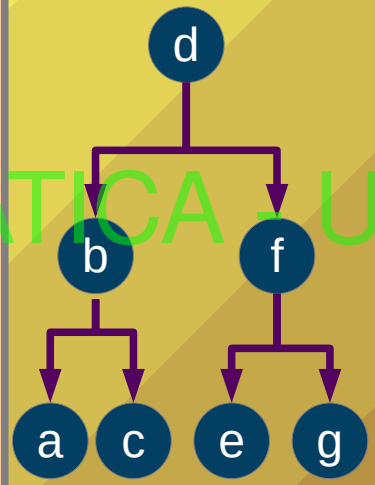




```
bfs:
```

Árboles binarios

- Recorridos en anchura.

```
breadthFirstTraversal(t:BinaryTree[T], var p:Process[T]):Bool
Var
   q          : Queue[ BinaryTree[T] ]
   keep_going: Bool
   subtree    : BinaryTree[T]
Begin
   keep_going ← True
   q.enque(t)
   While Not q.isEmpty() And keep_going Do
      subtree ← q.front()
      q.deque()
      If not subtree.isEmpty() Then
         keep_going ← p.apply(subtree.item())
         q.enque(subtree.left())
         q.enque(subtree.right())
      End-If
   End-While
   Return keep_going
End.
```

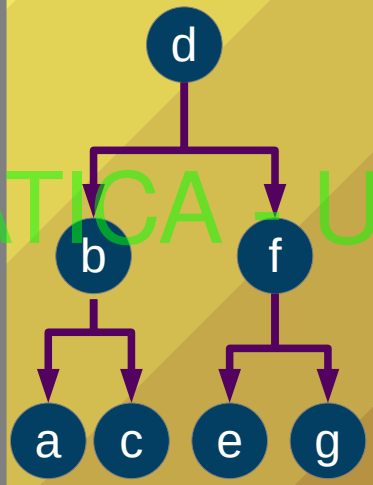




```
bfs:
dbfaceg
```

Árboles binarios

- Recorrido en profundidad (iterativo).

```
depthFirstTraversal(t:BinaryTree[T], var p:Process[T]):Bool
Var


Begin









End.
```

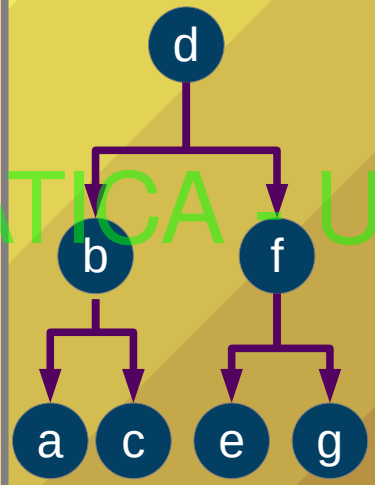



```
dfs:
```

Árboles binarios

- Recorrido en profundidad (iterativo).

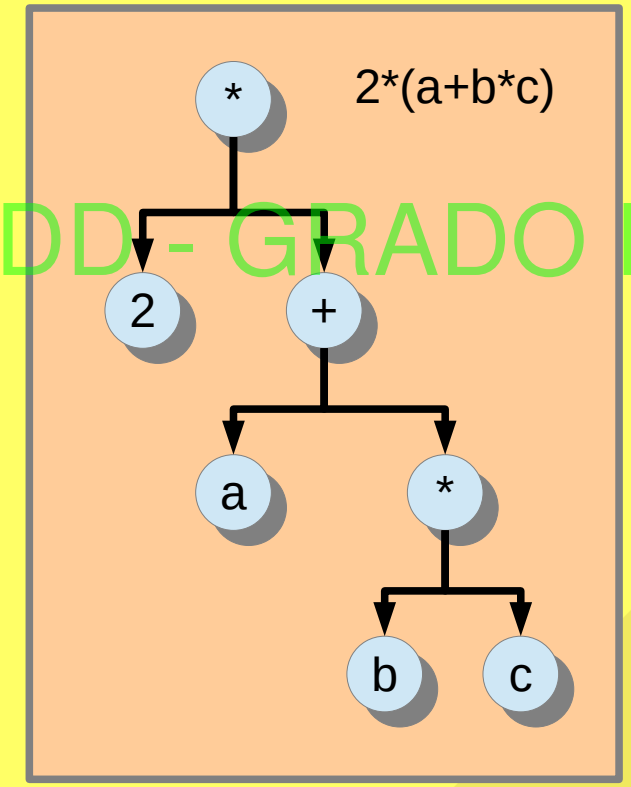
```
depthFirstTraversal(t:BinaryTree[T], var p:Process[T]):Bool
Var
   s          : Stack[ BinaryTree[T] ]
   keep_going: Bool
   subtree    : BinaryTree[T]
Begin
   keep_going ← True
   s.push(t)
   While Not s.isEmpty() And keep_going Do
      subtree ← s.top()
      s.pop()
      If Not subtree.isEmpty() 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

- Fold.



tree : ‘[]’
tree : ‘[‘ T tree tree ‘]’

Ejemplo:
“[‘*’ [‘2’ [] []] [‘+’ [‘a’ [] []] [‘*’ [‘b’ [] []] [‘c’ [] []]]]]”

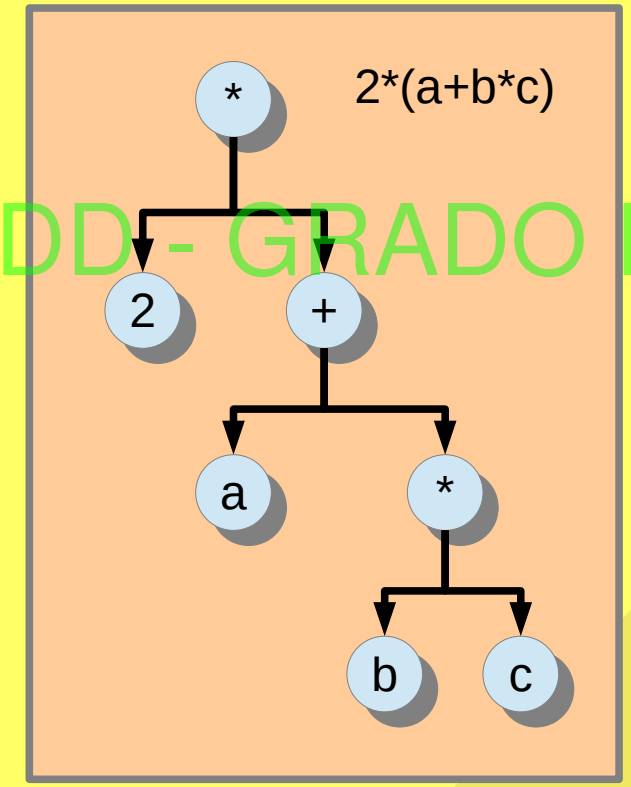
¿Un algoritmo para plegar esto?

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

End.

Árboles binarios

- Fold.



tree : ‘[]’
tree : ‘[‘ T tree tree ‘]’

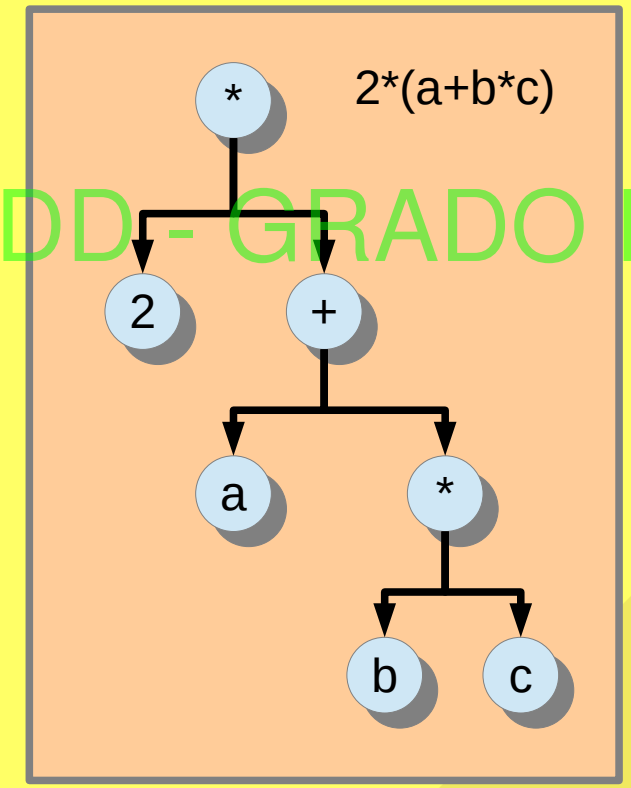
Ejemplo:
“[‘*’ [‘2’ [] []] [‘+’ [‘a’ [] []] [‘*’ [‘b’ [] []] [‘c’ [] []]]]]”

¿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’ [] []]]]]”

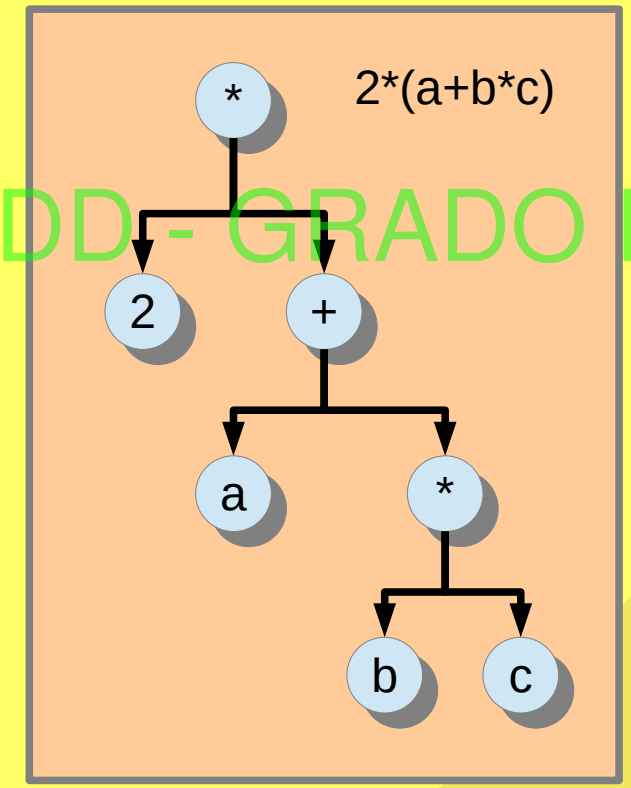
Algorithm BinaryTree[T]::create(in:Stream): BinaryTree[T]

Begin

End.

Árboles binarios

- Unfold.



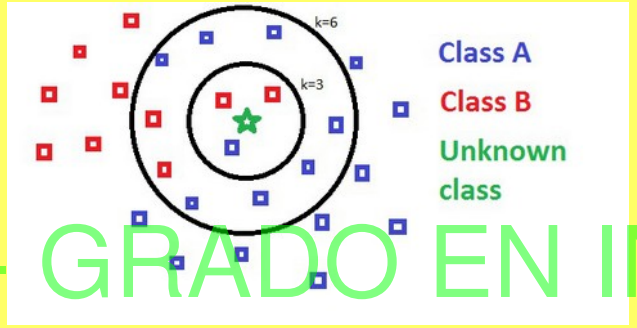
tree : '[]'
tree : '[' T tree tree ']'

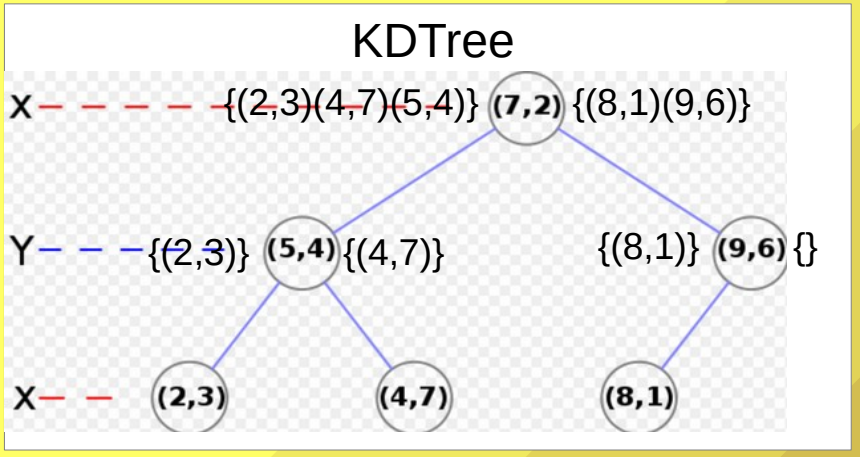
Ejemplo:
"['*' ['2' [] []] ['+' ['a' [] []] ['*' ['b' [] []] ['c' [] []]]]]"

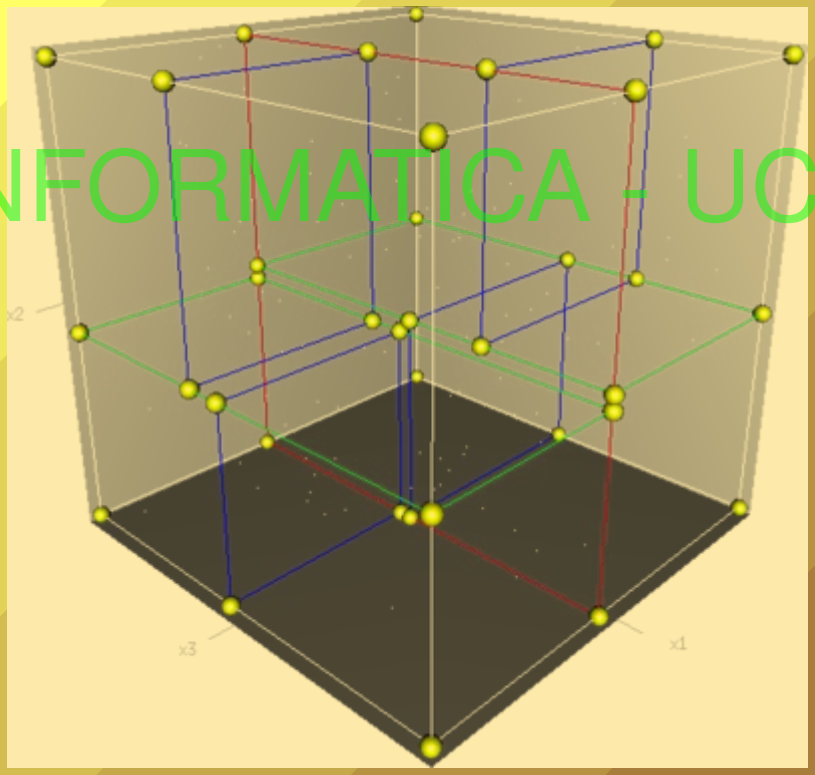
```
Algorithm BinaryTree[T]::create(in:Stream): BinaryTree[T]
Var token: String, t:BinaryTree[T]
Begin
   t ← BTree[T]::create() #make an empty tree.
   in.read(token)
   If token = '[' Then
      t.createRoot( T::create(in) )
      t.setLeft(BinaryTree[T]:create(in))
      t.setRight(BinaryTree[T]:create(in))
      in.read(token)
      If token <> ']' Then
         ERROR
   Else If token <> '[]' Then
      ERROR
   End-If
   Return t
End.
```

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 item 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 (KDTree)

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.