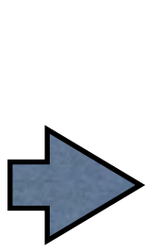
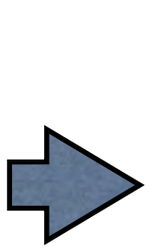
Bounded Exhaustive Testing

Generación Automática de Tests • 2018

Repaso: Random Testing



Programa bajo Test



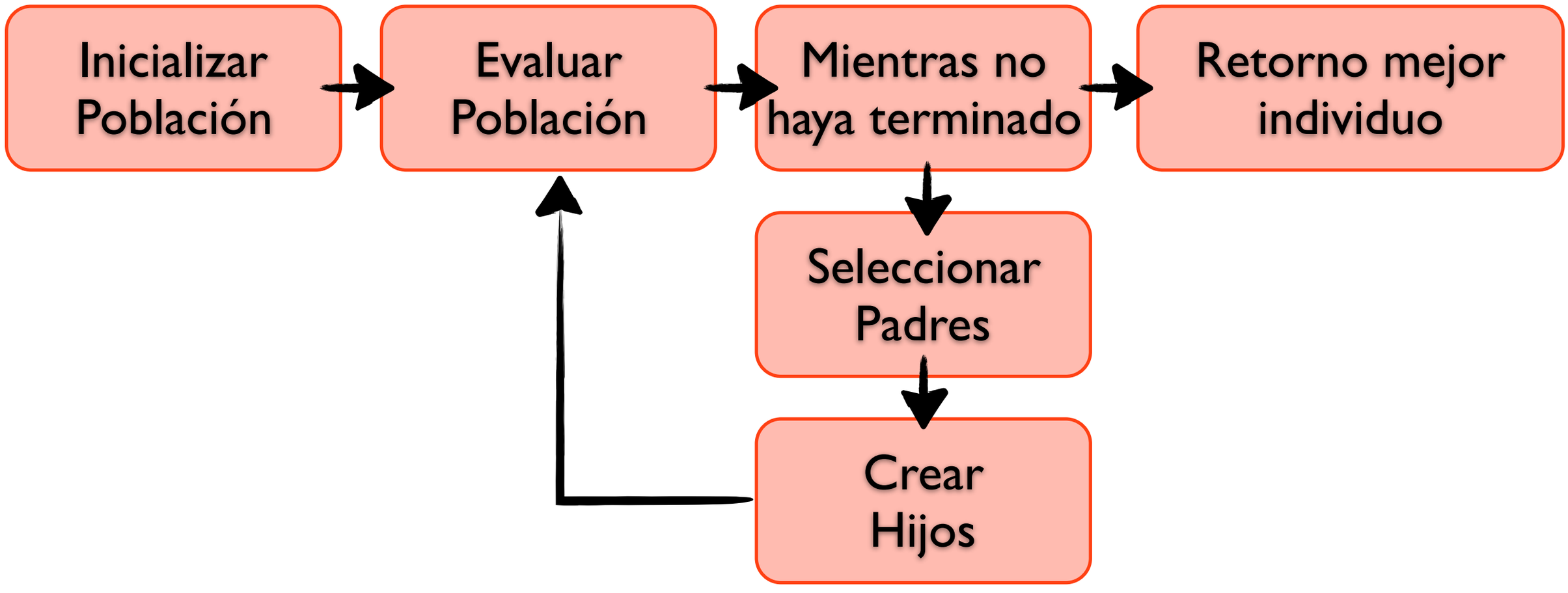
Oráculo

- Es una búsqueda (casi) completamente no guiada
- Si la técnica sistemática no es mejor que random testing, entonces no es valiosa
- Barata & fácil de implementar
- Funciona bastante bien en muchos casos

Repaso: Concolic Testing

- Ejecuta concretamente el test pero guarda la path condition
- Utiliza un constraint solver para crear nuevos inputs

Repaso: Search-Based Testing



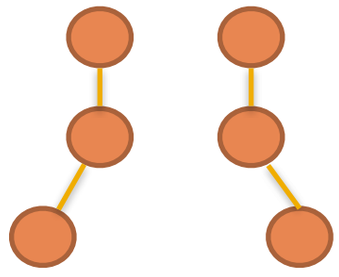
Bounded Exhaustive Testing

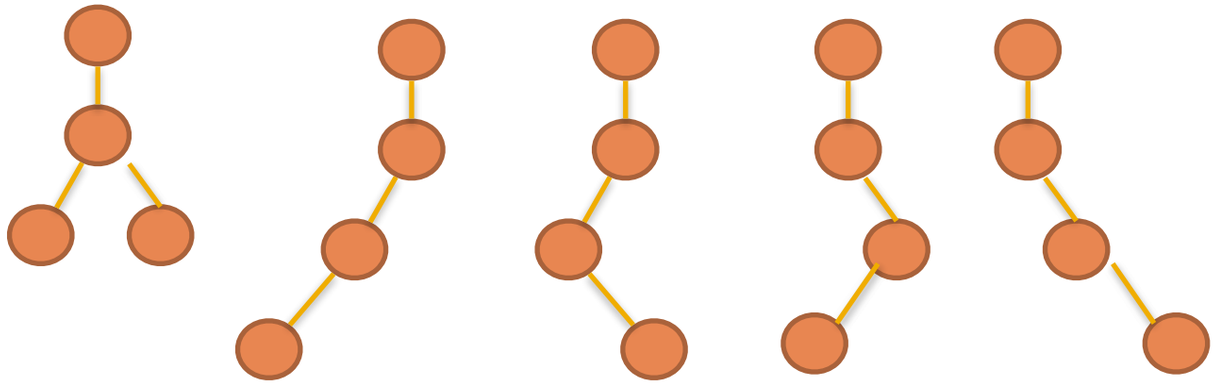
- Idea: Para una función *f*, generar *todos* los inputs (hasta un tamaño dado)
- También conocido como "*Bounded verification*" o "*Bounded Testing*"
- Asume que Ejecutar f es *barato*

Exhaustive Inputs

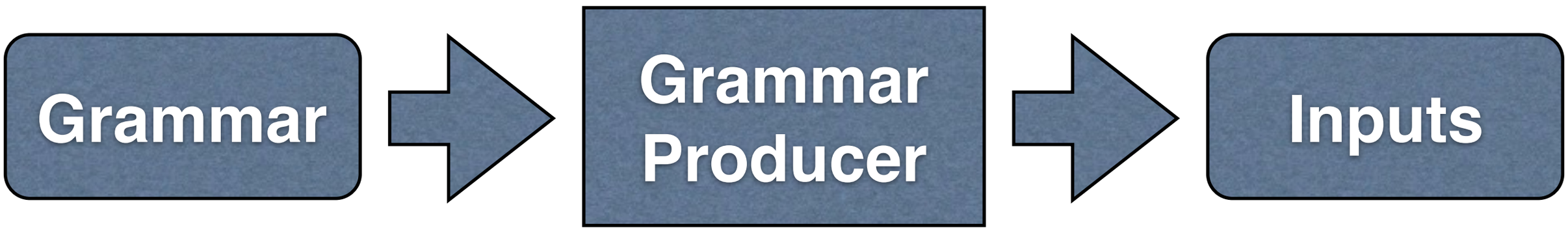
- Necesitamos identificar qué significa que dos inputs sean *equivalentes*

- Eso (generalmente) significa centrarnos en la e*structura en lugar de en el* contenido





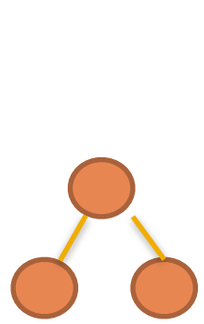
Grammar-Based Testing

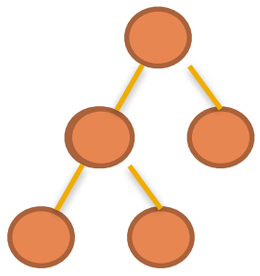


- Enumerar *todas las "cadenas"* de una gramática
- Comenzar por las producciones *más cortas*

Grammar-Based Testing

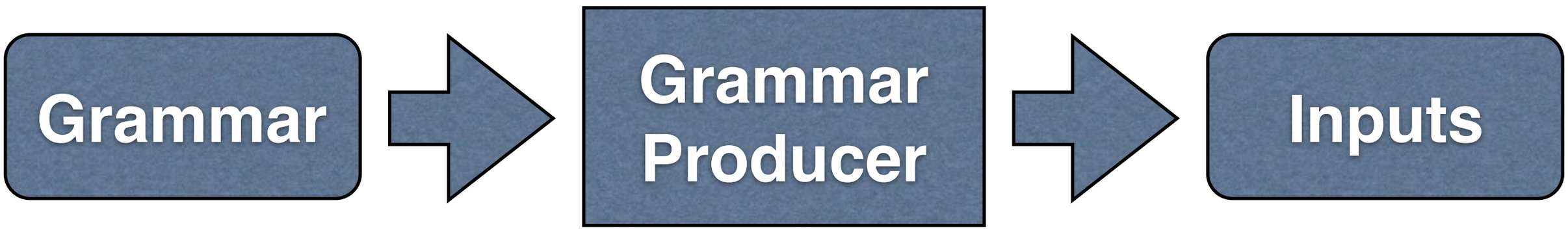
- Queremos construir árboles binarios:
  - x
  - (x x)
  - ((x x) x), etc…





- ¿Qué gramática podemos definir para generar todos los árboles binarios posibles?

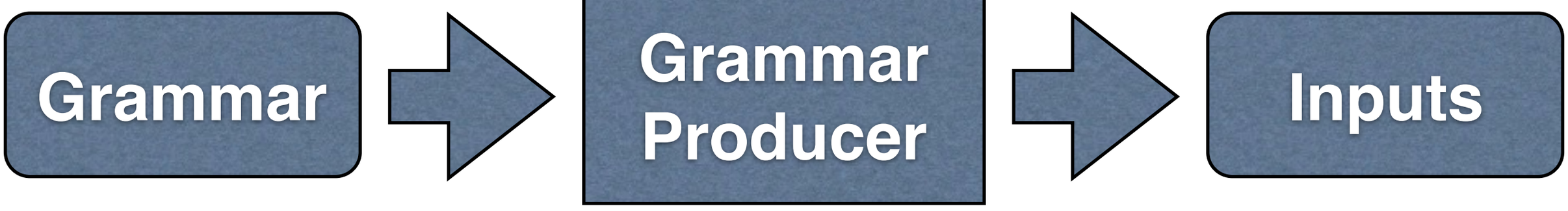
Grammar Producer

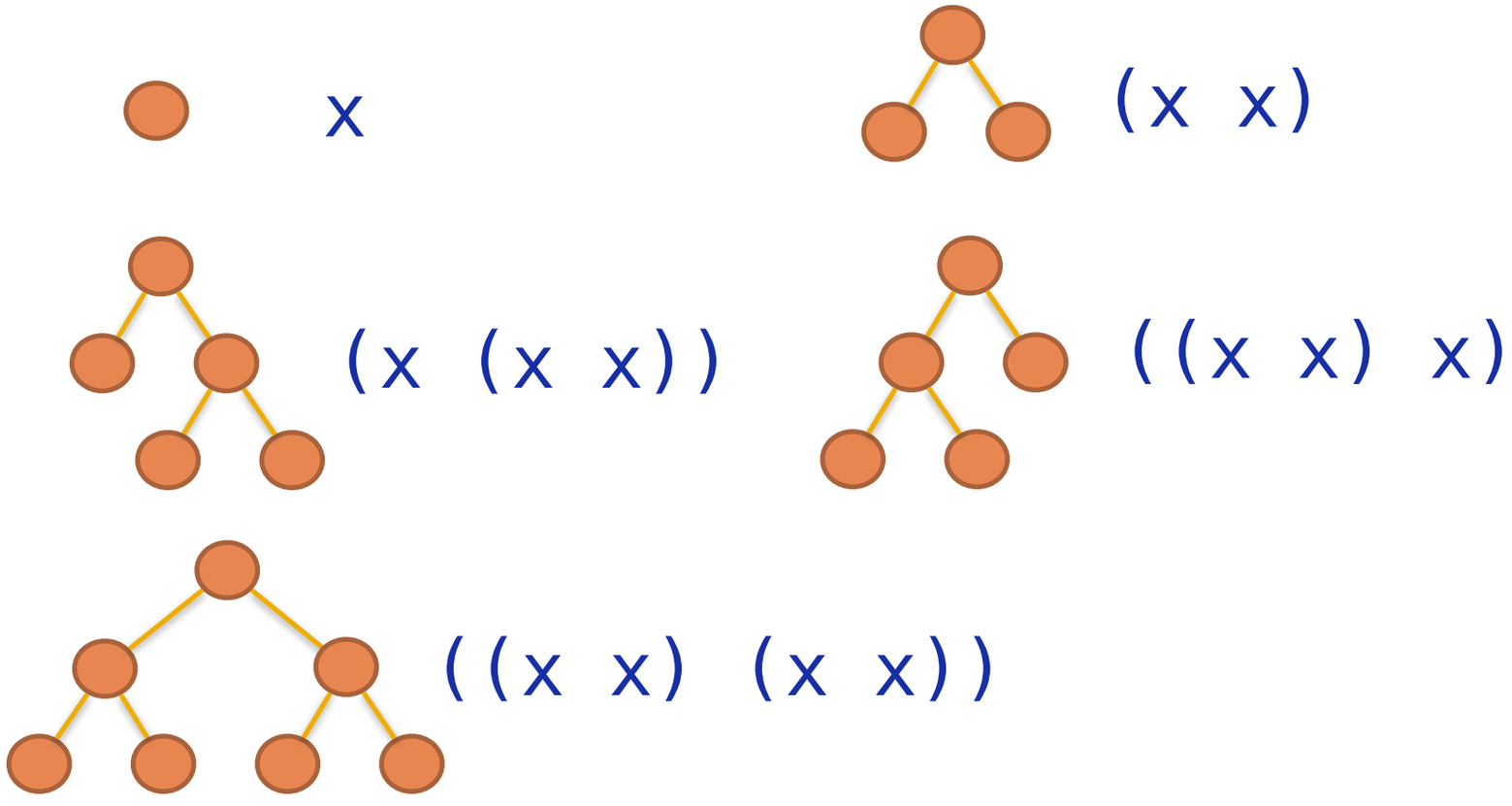



```
grammar = [
    ("$START", "$TREE"),

    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),

    ("$LEAF",  "x"),
]
```

```
x
(x x)
((x x) x)
(x (x x))
((x x) (x x))
(((x x) x) x)
((x (x x)) x)
(x ((x x) x))
(x (x (x x)))
```

Grammar Producer






```
x
(x x)
((x x) x)
(x (x x))
((x x) (x x))
(((x x) x) x)
((x (x x)) x)
(x ((x x) x))
(x (x (x x)))
```

Expand and Enqueue

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
$START
```

Expand and Enqueue

```
grammar = [
        ("$START", "$TREE"),
        ("$TREE",  "$LEAF"),
        ("$TREE",  "($TREE $TREE)"),
        ("$LEAF",  "x"),
    ]
```

$START

$TREE

Expand and Enqueue

```
grammar = [
        ("$START", "$TREE"),
        ("$TREE",  "$LEAF"),
        ("$TREE",  "($TREE $TREE)"),
        ("$LEAF",  "x"),
    ]
```

$START

$TREE

$LEAF ($TREE $TREE)

Expand and Enqueue

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

$START

$TREE

$LEAF

x ($TREE $TREE)

Expand and Enqueue

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

$START

$TREE

$LEAF

x

($TREE $TREE)

Expand and Enqueue

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
$START
$TREE
$LEAF
x
($TREE $TREE)
```

```
($LEAF $TREE) ($TREE $LEAF) (($TREE $TREE) $TREE) ($TRE
```

Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
```

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
(x $TREE) ($TREE $LEAF) ($LEAF ($TREE $TREE)) (($TREE
```

Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
```

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
(x $LEAF) ($TREE $LEAF)(x ($TREE $TREE))($LEAF ($TRE
```

Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
(x $LEAF)
```

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
(x x)($TREE $LEAF)(x ($TREE $TREE))($LEAF ($TREE $TR
```

Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
(x $LEAF)
(x x)
```

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
($TREE $LEAF)(x ($TREE $TREE))($LEAF ($TREE $TREE))
```

Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
(x $LEAF)
(x x)
($TREE $LEAF)
```

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
($LEAF $LEAF)(x ($TREE $TREE))($LEAF ($TREE $TREE))(
```

Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
(x $LEAF)
(x x)
($TREE $LEAF)
($LEAF $LEAF)
```

```
grammar = [
        ("$START", "$TREE"),
        ("$TREE",  "$LEAF"),
        ("$TREE",  "($TREE $TREE)"),
        ("$LEAF",  "x"),
]
```

```
($LEAF x)(x ($TREE $TREE))($LEAF ($TREE $TREE))(($TR
```

Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
(x $LEAF)
(x x)
($TREE $LEAF)
($LEAF $LEAF)
($LEAF x)
```

```
grammar = [
    ("$START", "$TREE"),
    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),
    ("$LEAF",  "x"),
]
```

```
(x ($TREE $TREE))($LEAF ($TREE $TREE))(($TREE $TREE)
```

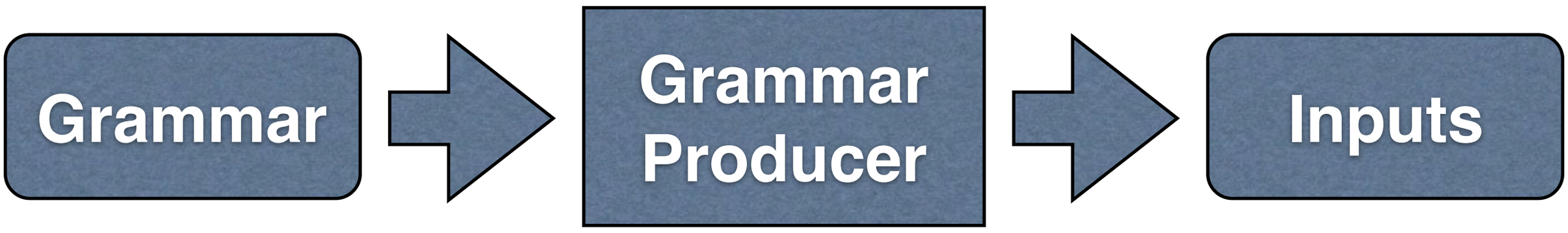
Expand and Enqueue

```
$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
(x $LEAF)
(x x)
($TREE $LEAF)
($LEAF $LEAF)
($LEAF x)
(x ($TREE $TREE))
```

```
grammar = [
  ("$START", "$TREE"),
  ("$TREE",  "$LEAF"),
  ("$TREE",  "($TREE $TREE)"),
  ("$LEAF",  "x"),
```

```
(x ($LEAF $TREE)) (x ($TREE $LEAF)) (($TREE $TREE) $T
```

Grammar Producer




```
grammar = [
    ("$START", "$TREE"),

    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),

    ("$LEAF",  "x"),
]
```

nuestra gramática

```
grammar = [
    ("$START", "$TREE"),

    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),

    ("$LEAF",  "x"),
]
```

```
# Producer, using a priority queue
# to prioritize shorter expansions
def produce(grammar, start = None):

    if start is None:
        # Use first symbol
        start = grammar[0][0]

    # Enqueue starting symbol
    pq = Queue.PriorityQueue()
    pq.put((0, start))                          encolar elemento
    seen = set()
```

```
    ("$START", "$TREE"),

    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),

    ("$LEAF",  "x"),
]

# Producer, using a priority queue
# to prioritize shorter expansions
def produce(grammar, start = None):

    if start is None:
        # Use first symbol
        start = grammar[0][0]

# Enqueue starting symbol
pq = Queue.PriorityQueue()
pq.put((0, start))
seen = set()

while not pq.empty():
    # Get current term
    (prio, term) = pq.get()
    yield term
```

retornar primer elemento

```
    # Enqueue starting symbol
    pq = Queue.PriorityQueue()
    pq.put((0, start))
    seen = set()

    while not pq.empty():
        # Get current term
        (prio, term) = pq.get()
        yield term

        # Produce possible expansions
        instance = 0
        for symbol in symbols(term):
            for rule in rules(grammar, symbol):
                new_term = apply(term, rule, instance)
                if new_term in seen:
                    continue

                # Favor short strings
                prio = len(new_term)
                pq.put((prio, new_term))
                seen.add(new_term)

        instance = instance + 1
```

aplico todas las reglas

y luego encolo las expansiones

Best Priority

- Queremos producir cadenas *cortas* antes que *largas*
- Queremos favorecer a *terminales* sobre *no terminales*
- Necesitamos anticipar cuanto una variable se expandirá (larga o corta)
- Heurística:

```
prio = len(new_term)
       + new_term.count('$') * len(new_term) / 2
pq.put((prio, new_term))
```

Helpers

```
# Aplica RULE sobre TERM, reeemplazando el símbolo
# INSTANCE
def apply(term, rule, instance = 0):
    (old, new) = rule
    index = 0
    for i in range(0, instance):
        index = term[index:].find('$') + 1
    return (term[:index] +
            term[index:].replace(old, new, 1))

assert apply("$F", ("$F", "bar")) == "bar"
assert apply("$F $F", ("$F", "bar"), 0) == "bar $F"
assert apply("$F $F", ("$F", "bar"), 1) == "$F bar"
```

Helpers

```
# Retorna la lista de símbolos en TERM
def symbols(term):
    return re.findall("\$[A-Za-z]*", term)

assert symbols("$FOO $BAR") == ["$FOO", "$BAR"]
```

Helpers

```
# Retorna las reglas en GRAMMAR que reemplazan SYMBOL
def rules(grammar, symbol):
    r = [];
    for (old, new) in grammar:
        if old == symbol:
            r.append((old, new))
    return r

assert rules([("$F", "bar"), ("$B", "baz")], "$F")
           == [("$F", "bar")]
```

Demo

```
$ python GrammarProducer.py
```

Producciones

$START
$TREE
$LEAF
x
($TREE $TREE)
($LEAF $TREE)
(x $TREE)
(x $LEAF)
(x x)
($LEAF $LEAF)
($LEAF x)
($TREE $LEAF)
($TREE x)
(($TREE $TREE) x)
(($LEAF $TREE) x)
((x $TREE) x)
((x $LEAF) x)
((x x) x)
(($LEAF $LEAF) x)

(x (x ((x $LEAF) x)))
(x (x ((x x) x)))
(x (x (($LEAF $LEAF) x)))
(x (x (($LEAF x) x)))
(x (x (($TREE $LEAF) x)))
(x (x (($TREE x) x)))
(x (x (x ($TREE $TREE))))
(x (x (x ($LEAF $TREE))))
(x (x (x (x $TREE))))
(x (x (x (x $LEAF))))
(x (x (x (x x))))
(x (x (x ($LEAF $LEAF))))
(x (x (x ($LEAF x))))
(x (x (x ($TREE $LEAF))))
(x (x (x ($TREE x))))
($LEAF ($LEAF ($TREE $TREE)))
($LEAF ($LEAF ($LEAF $TREE)))
($LEAF ($LEAF ($LEAF $LEAF)))
($LEAF (($TREE $TREE) $TREE))
($LEAF (($LEAF $TREE) $TREE))

x
(x x)
((x x) x)
(x (x x))
((x x) (x x))
(((x x) x) x)
((x (x x)) x)
(x ((x x) x))
(x (x (x x)))
((x x) (x (x x)))
(((x x) (x x)) x)
(((x x) x) (x x))
((((x x) x) x) x)
(((x (x x)) x) x)
((x (x x)) (x x))
((x ((x x) x)) x)
((x (x (x x))) x)
((x x) ((x x) x))
(x ((x x) (x x)))
(x (((x x) x) x))
(x ((x (x x)) x))
(x (x ((x x) x)))
(x (x (x (x x))))

Únicamente Terminales

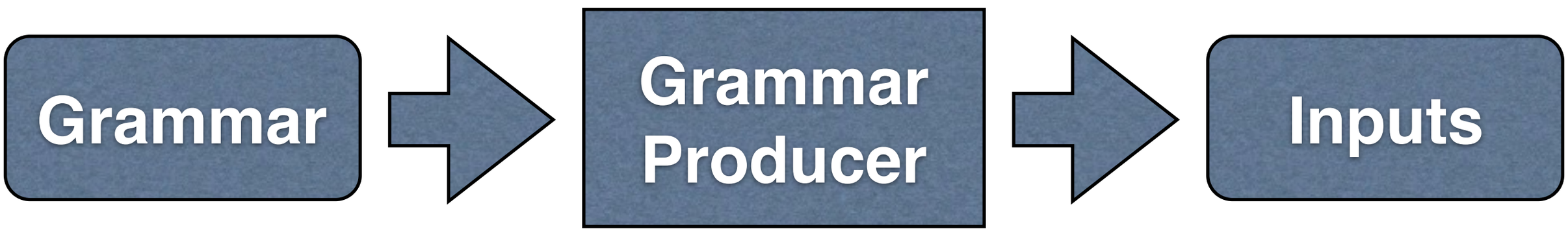
```
grammar = [
    ("$DOCUMENT", "$DOCTYPE$HTML"),

    ("$DOCTYPE", '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"\n' + \
        '"http://www.w3.org/TR/html4/strict.dtd">\n'),

    ("$HTML", "<HTML>$HEAD$BODY</HTML>\n"),

    ("$HEAD", "<HEAD>$TITLE</HEAD>\n"),
    ("$TITLE", "<TITLE>A generated document</TITLE>\n"),

    ("$BODY", "<BODY>$DIVS</BODY>\n"),

    ("$DIVS", "$DIV"),
    ("$DIVS", "$DIV\n$DIVS"),

    ("$DIV", "$HEADER\n$LIST"),

    ("$HEADER", "<H1>A header.</H1>"),
    ("$HEADER", "<H1>Another header.</H1>"),

    ("$LIST", "<UL>$ITEMS</UL>"),
    ("$LIST", "<OL>$ITEMS</OL>"),

    ("$ITEMS", "$ITEM"),
    ("$ITEMS", "$ITEM$ITEMS"),

    ("$ITEM", "<LI>$TEXT</LI>\n"),

    ("$TEXT", "An item"),
    ("$TEXT", "Another item"),
]
```

“Exhaustive” HTML

Demo

```
$ python HTMLTester.py
```

Grammar-Based Testing

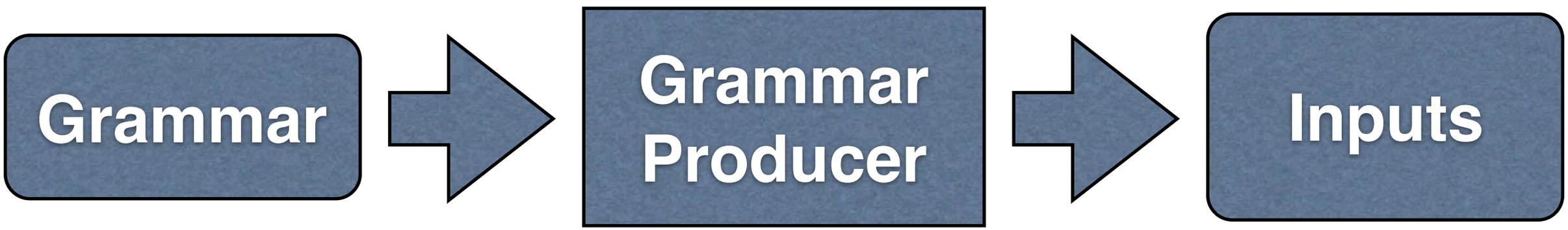


- Dado una gramática generamos inputs.
- Quizás algunas características de los inputs no son fácilmente capturables con una gramática

Algunos problemas

- Generar los árboles binarios con los elementos 1,2 y 3
  - Usamos una extensión de la gramática presentada anteriormente
- ¿Qué hacemos si queremos generar los *árboles binarios de búsqueda* con elementos 1,2,3?

Filtered Grammar-Based Testing



Ej:binary trees

Ej:arboles binarios de búsqueda

Recap: Cubrimientos (Dimensiones)

- De Código (Líneas, Branches, ejes, etc)
- De “Fallas” (Score: Mutantes Muertos / Totales)
- **Lo Nuevo:** De “Instancias/ Estructuras” (Inputs no isomorfos)

¿Qué son Inputs No-Isomorfos?

- Para Stack, todas estas producciones generan instancias isomorfas:
  - empty
  - pop(push(x, empty))
  - pop(pop(push(x,push(y,empty))
- Nos interesa el cubrimiento de instancias no isomorfas (i.e. inputs con estructura distinta)

Filtered Grammar-Based Testing

Limitaciones:

- Cantidad muy baja de instancias que cumplen el filtro (predicado)
- Inexistencia de una gramática para generar los inputs (sólo el predicado)
- Pocas instancias no-isomorfas

RepOk: Invariantes de Representación

```
class Node {
   Node left;
   Node right;
}

class BinTree {
   Node root;

  boolean repOK() {
    if this.root==null
      return true;

    Set<Node> visited = new HashSet<Node>();
    List<Node> toVisit = new LinkedList<Node>();
    toVisit.add(this.root);
```

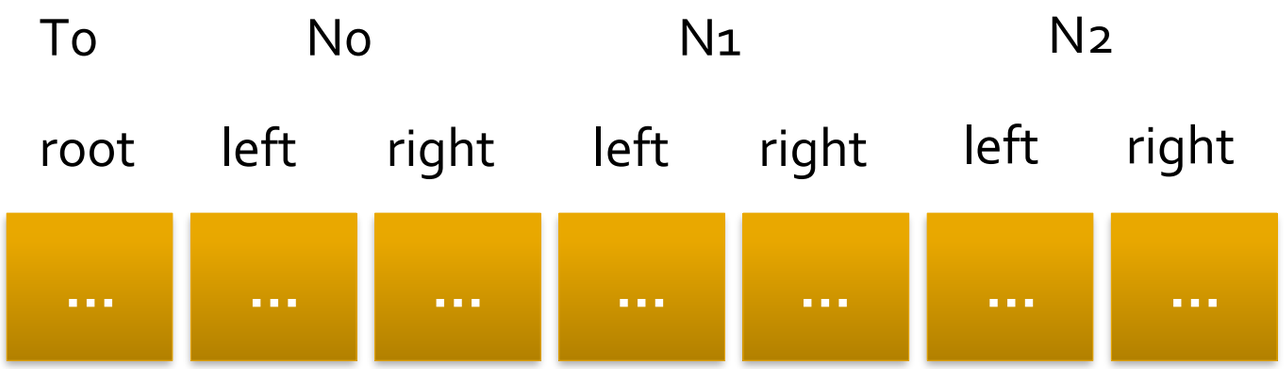
```
 Node root;

boolean repOK() {
  if this.root==null
    return true;

  Set<Node> visited = new HashSet<Node>();
  List<Node> toVisit = new LinkedList<Node>();
  toVisit.add(this.root);

  while !toVisit.isEmpty() {
      Node curr = toVisit.removeFirst();
      if visited.contains(curr)
         return false;
      if (curr.left!=null)
        toVisit.add(curr.left);
      if (curr.right!=null)
        toVisit.add(curr.right)
    visited.add(curr);
  }
  return true;
}
```

Generación Basada en Predicados

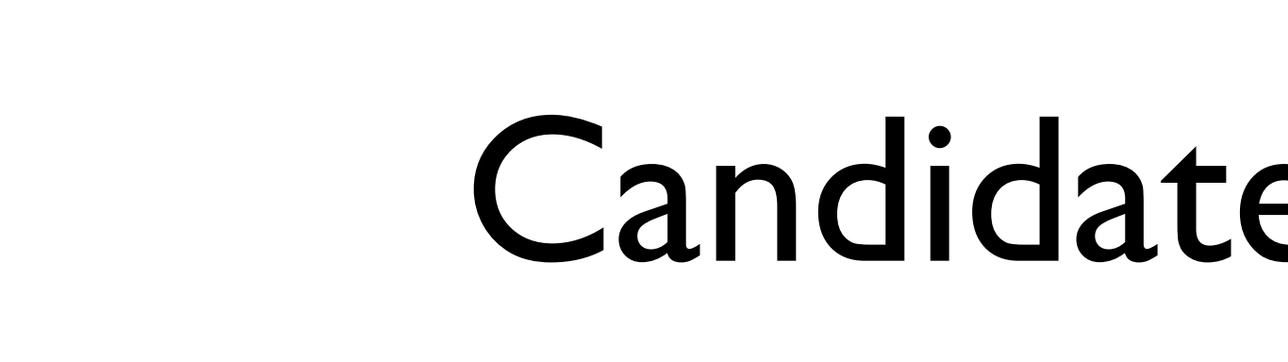
- ¿Cómo podemos generar instancias del BinTree tal que …
  - …que cumplan el repOk(),y además
  - …sean no-isomorfas?

Finitización (Scope)

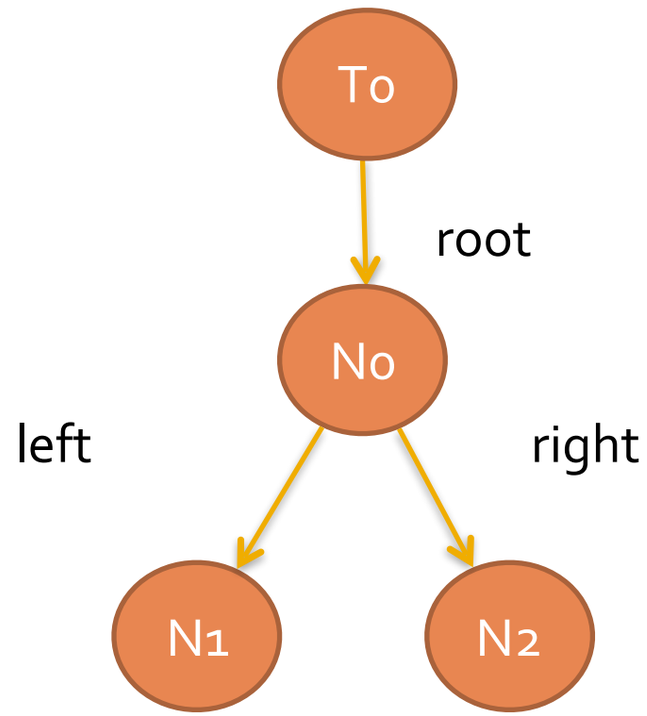
- Definimos un tamaño máximo
- Ejemplo, a lo sumo:
  - 1 objeto BinTree (null<T0)
  - 3 objetos Node (null<N0<N1<N2)



Candidate Vector

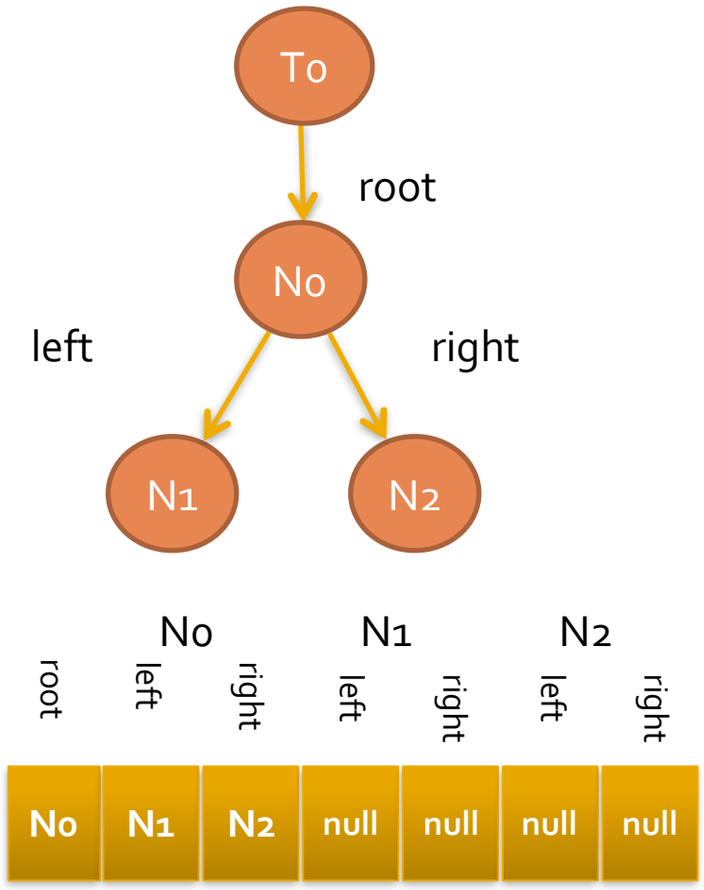


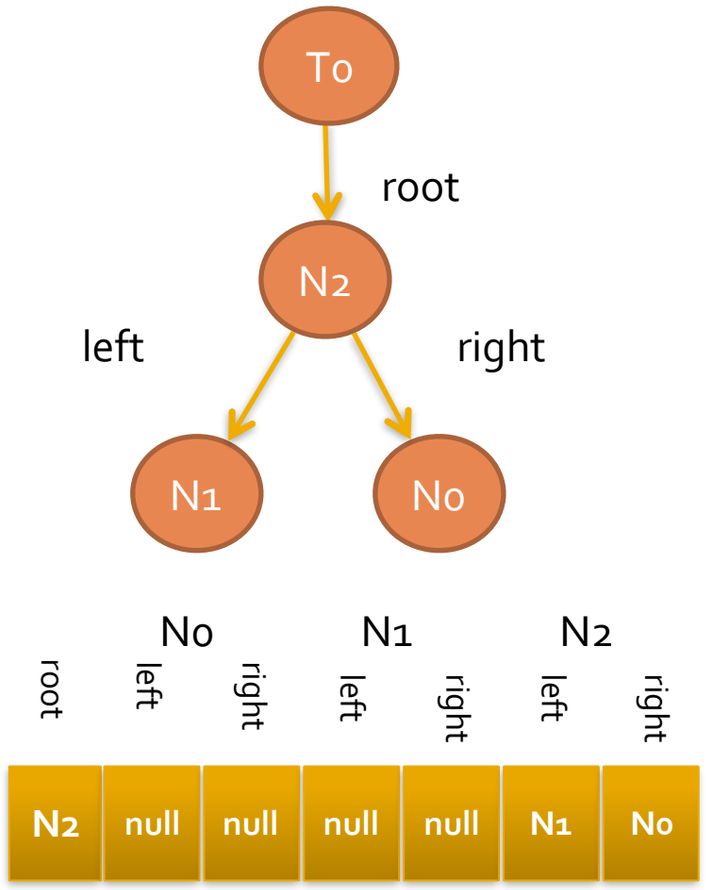
Representación Vectorial

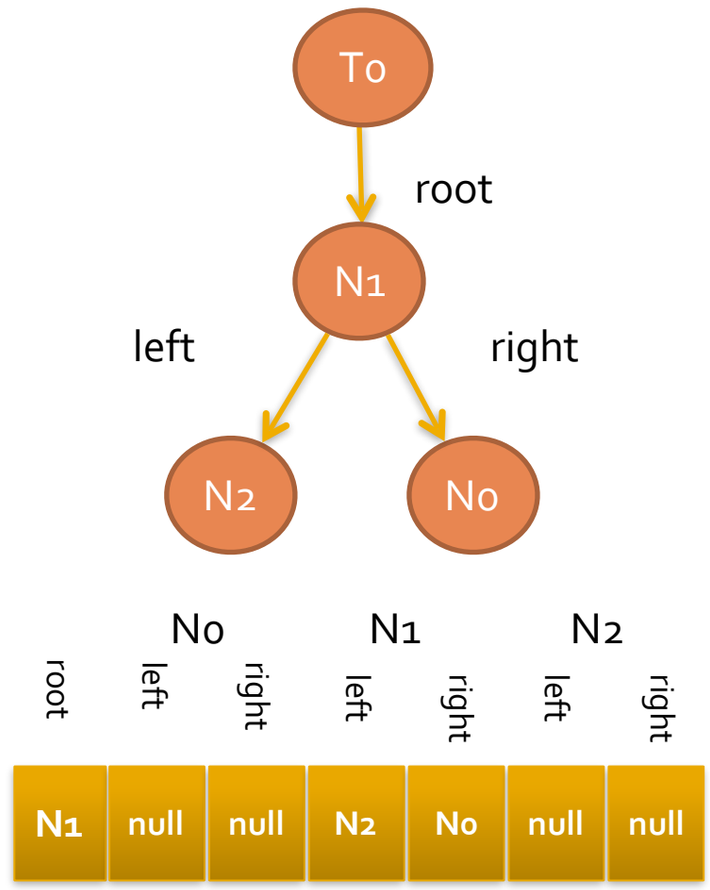


Representación Gráfica

Instancias Isomorfas

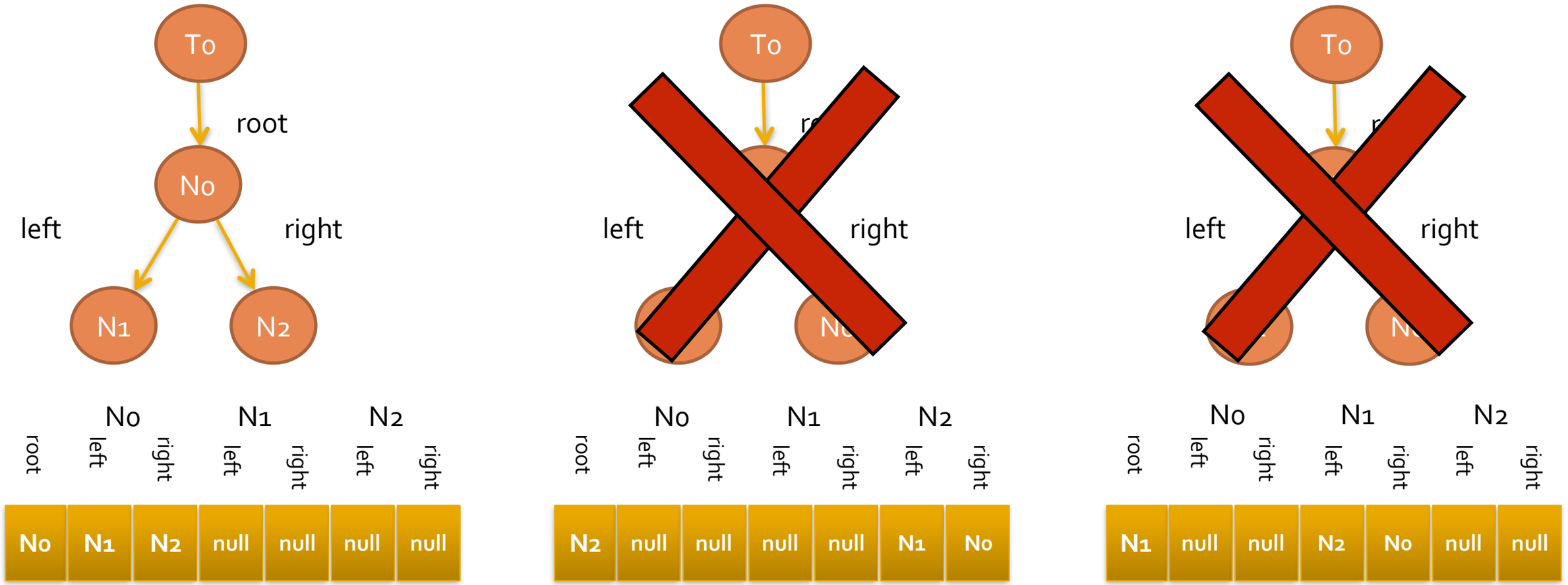






- ¿Cómo podemos evitar enumerar instancias isomorfas de candidatos (simetrías)?

Rotura de Simetrías

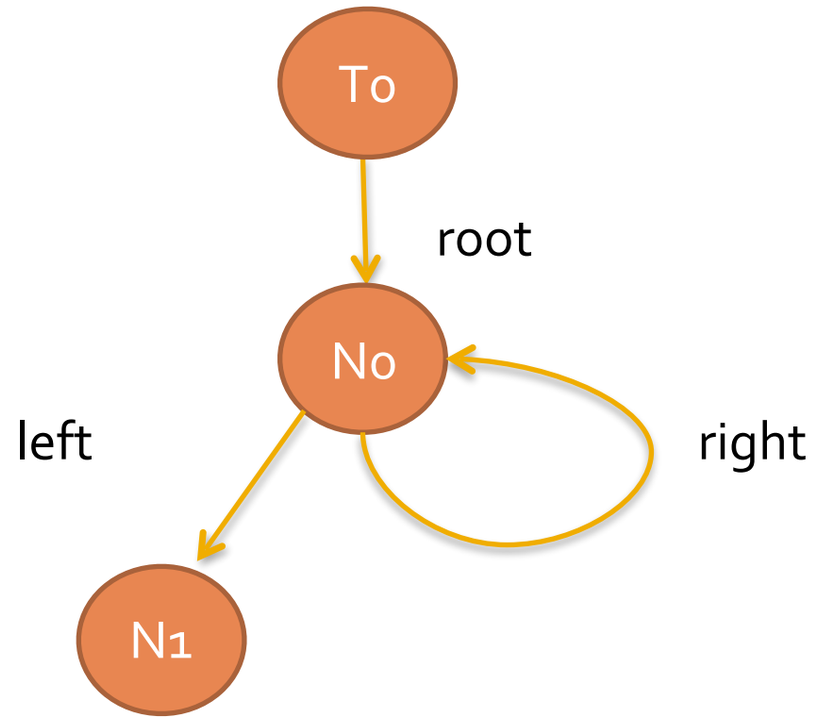


- El índice de un elemento *e* no puede ser mayor a *k+1*, donde *k* es el mayor de los índices de todos los elementos del mismo tipo que *e*

Rotura de Simetrías

- Rotura de simetrías (eliminación de estructuras simétricas) mediante el uso de una regla muy simple:
  - *"Durante la búsqueda, en la construcción de un vector candidato, se permite a lo sumo un objeto 'no tocado' de cada dominio"*
  - *El índice de un elemento e no puede ser mayor a k+1, donde k es el mayor de los índices de todos los elementos del mismo tipo que e*

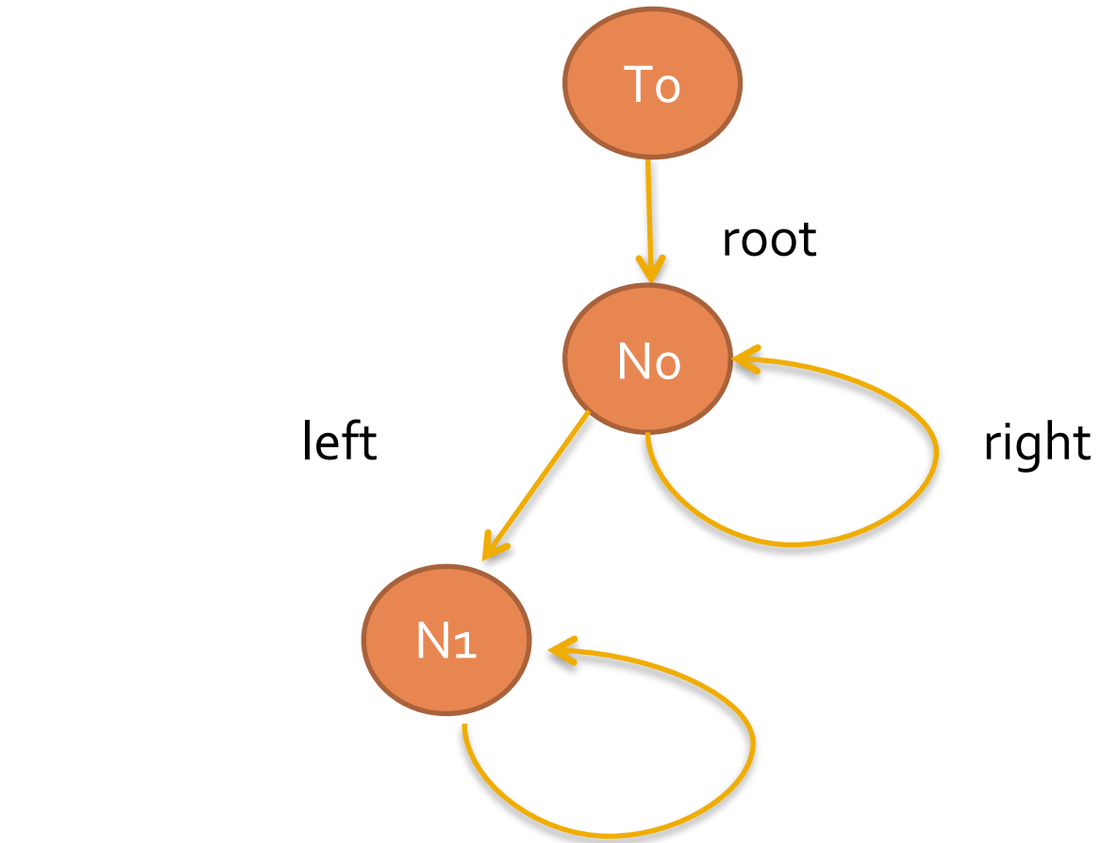
Generación Basada en Predicados



repOk()==false

To		No		N1		N2	
root	left	right	left	right	left	right	
No	**N1**	**No**	**null**	**null**	**null**	**null**	

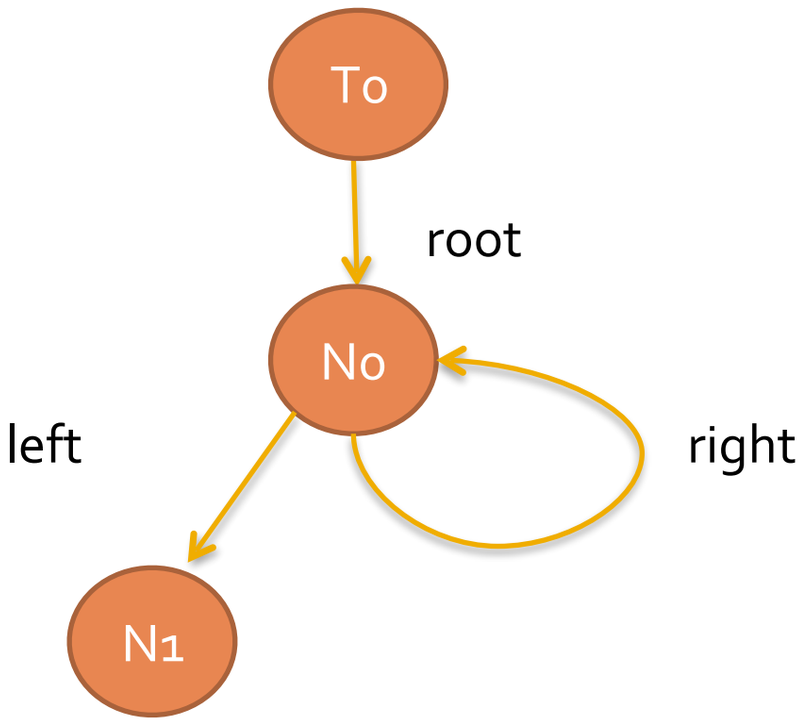
Generación Basada en Predicados

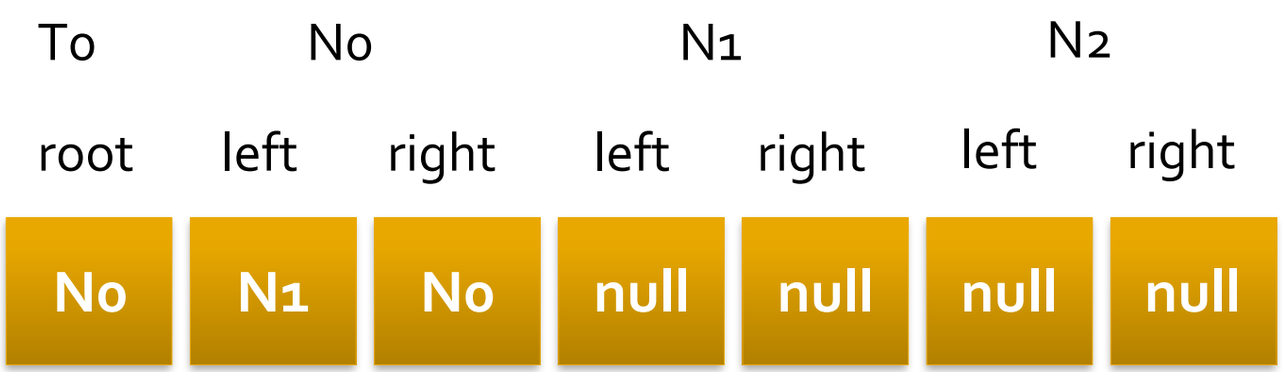


repOk()==false

To		No		N1		N2	
root	left	right	left	right	left	right	
No	**N1**	**No**	**N1**	**null**	**null**	**null**	

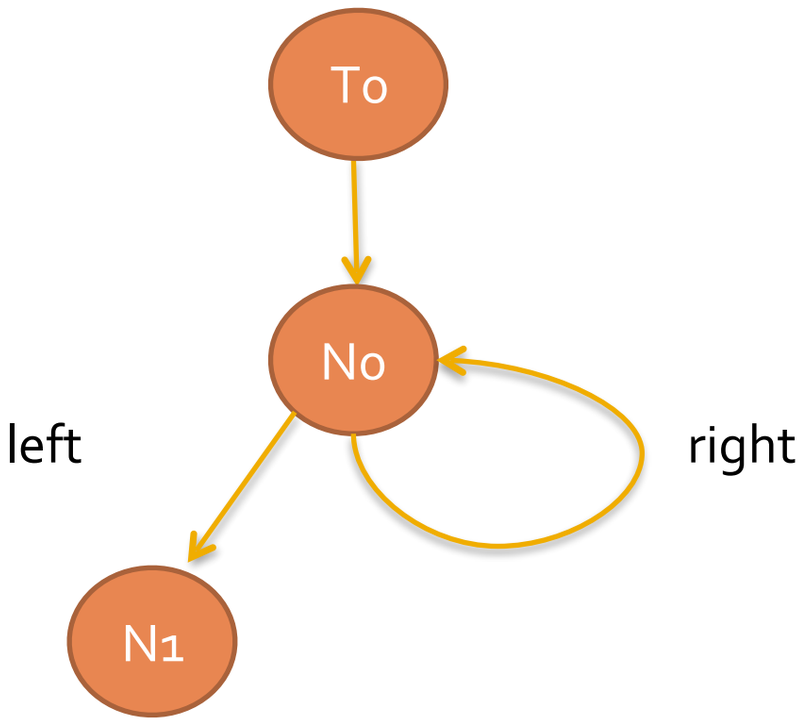
Generación Basada en Predicados

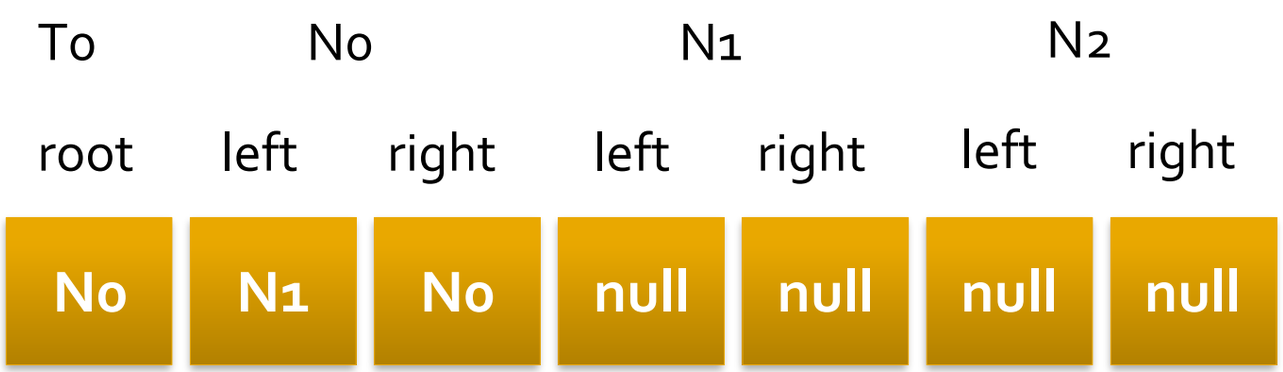




- ¿Cómo puedo evitar iterar sobre regiones que no impactan en el repOK()?

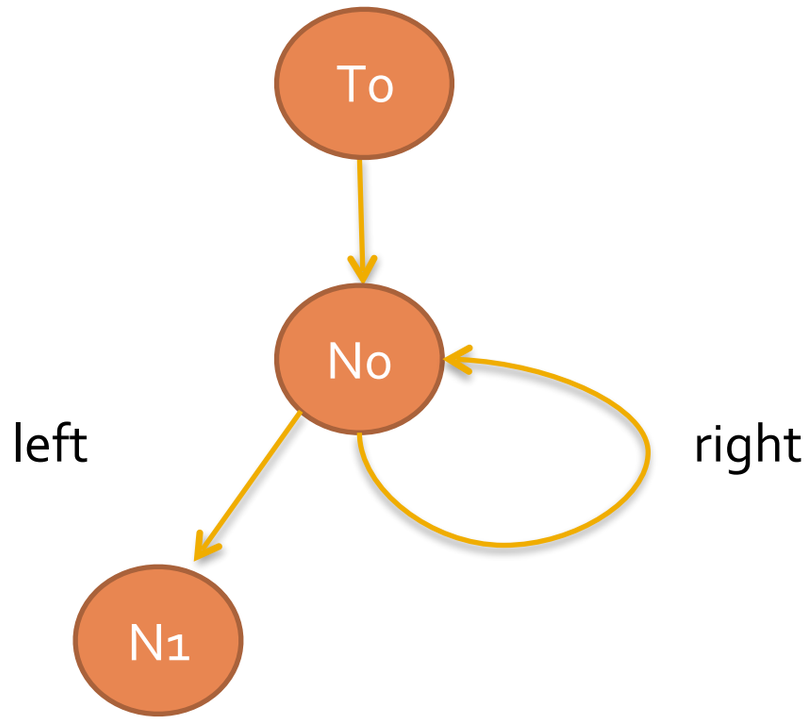
Field Ordering





- Monitoreamos la ejecución de repOK()
- Almacenamos el orden de lectura de los campos de los objetos (=Field Ordering)

Field Ordering



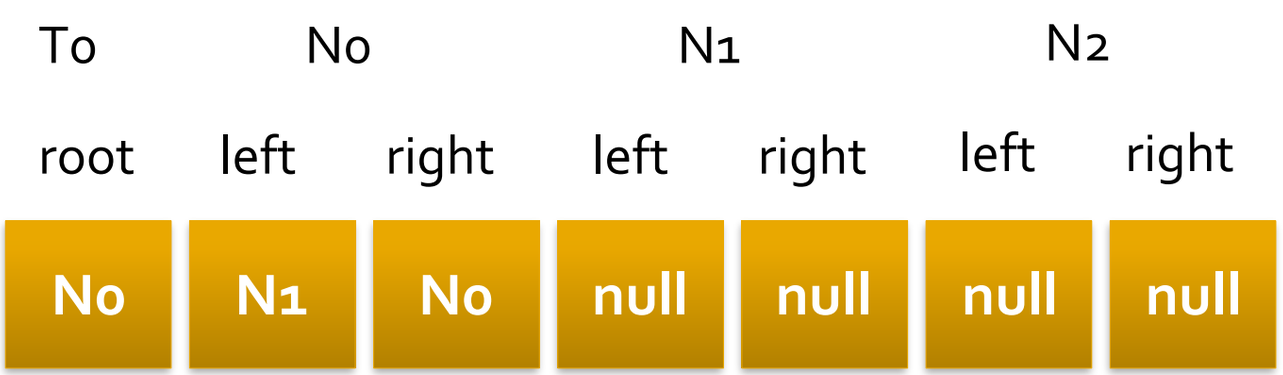
To		No		N1		N2	
root	left	right	left	right	left	right	
No	**N1**	**No**	**null**	**null**	**null**	**null**	

1. 2. 3.

Field Ordering:
Al ejecutar repOK()
se acceden a los
campos de los objetos
en el siguiente orden:

1) To.root
2) No.left
3) No.right

Field Ordering

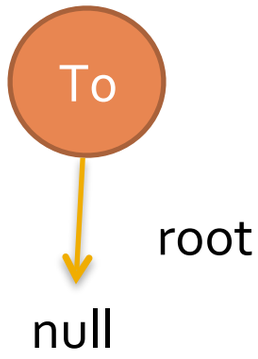


Siempre incremento el último campo en el field ordering

Si no puedo incrementarlo, backtraqueo al anterior campo en el field ordering hasta que puedo incrementar alguno

Generando Candidatos

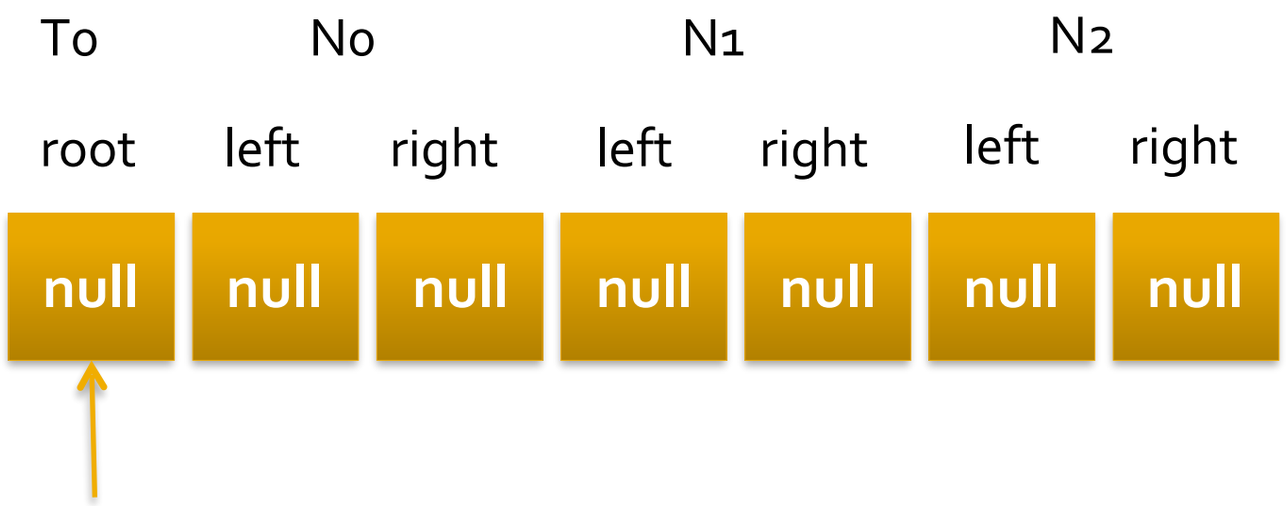
Representación Gráfica



Field Ordering

1) To.root

Candidate Vector

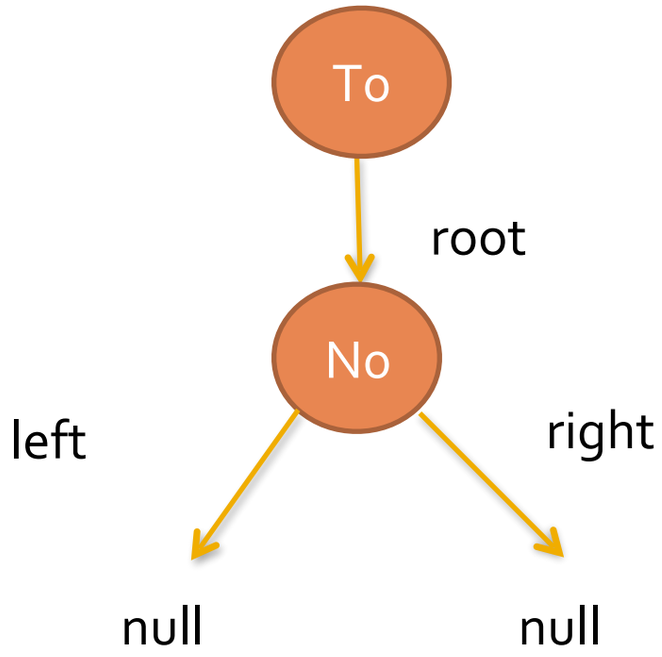


this.repOk()==true

inc To.root

Generando Candidatos

Representación Gráfica



Field Ordering

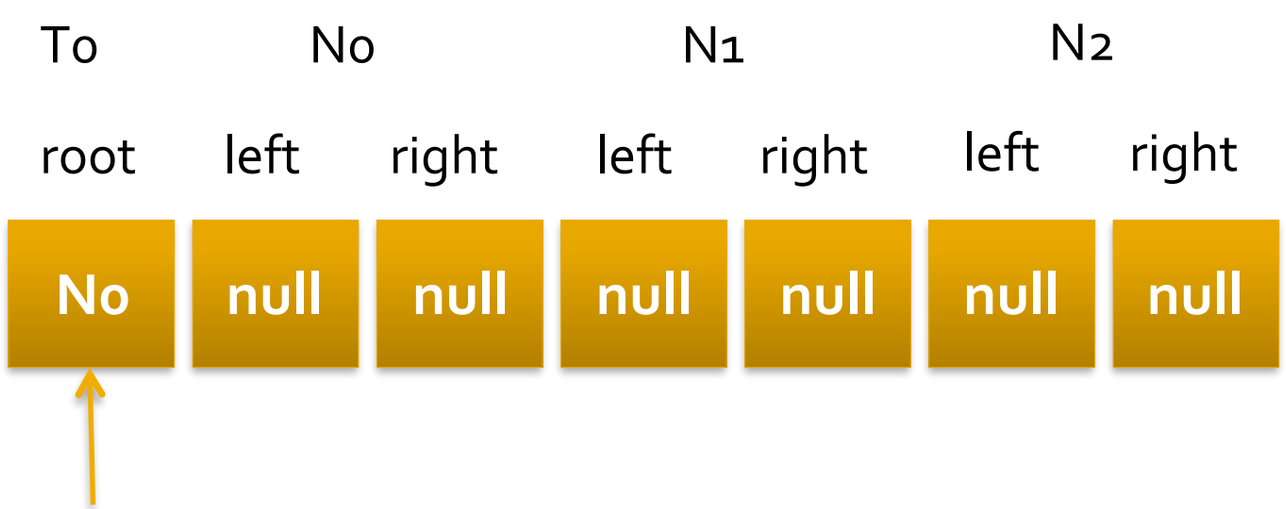
1) To.root

2) No.left

3) No.right ←

this.repOk()==true

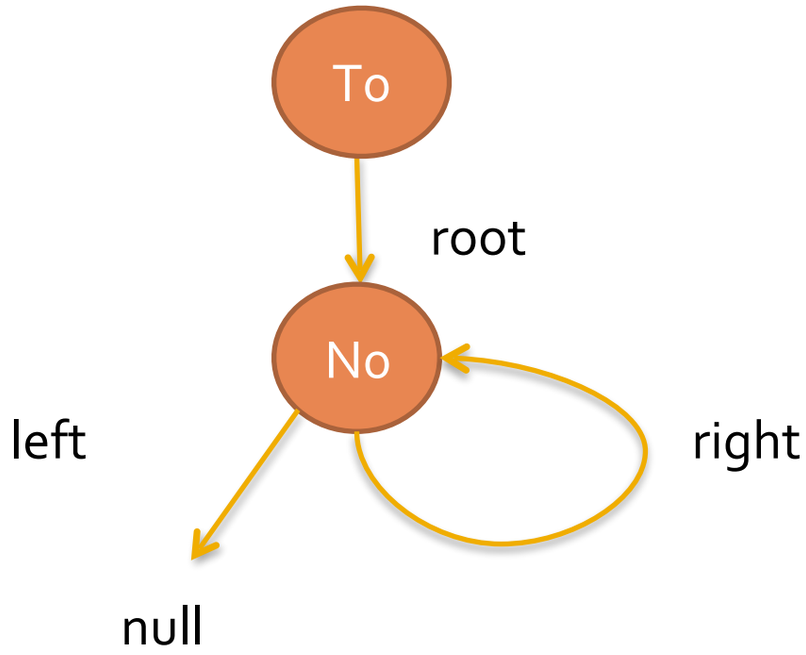
Candidate Vector



inc No.right

Generando Candidatos

Representación Gráfica



Field Ordering

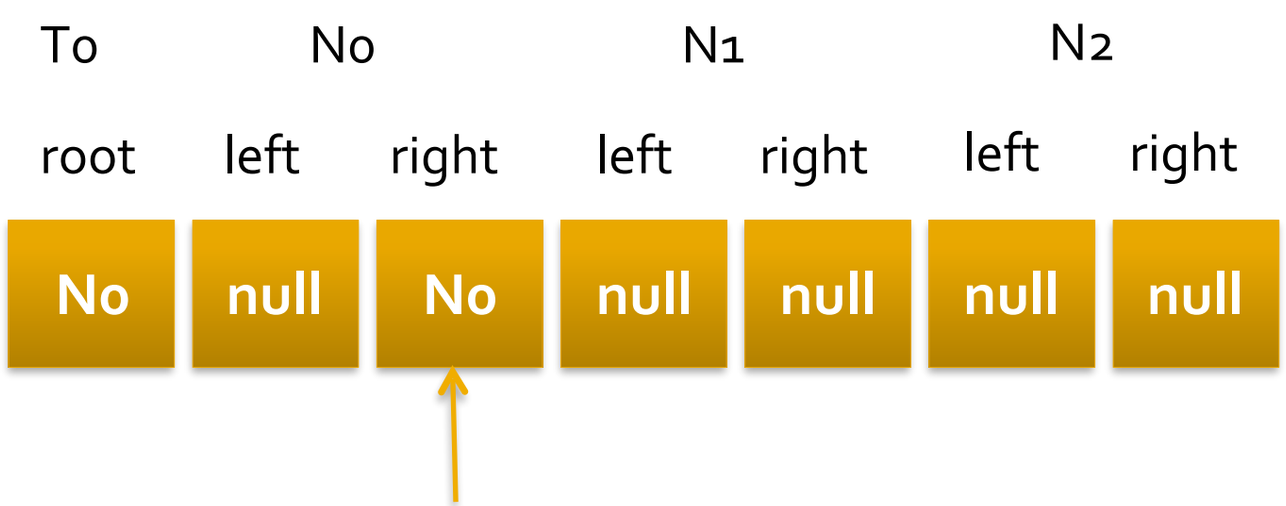
1) To.root

2) No.left

3) No.right ←

this.repOk()==false

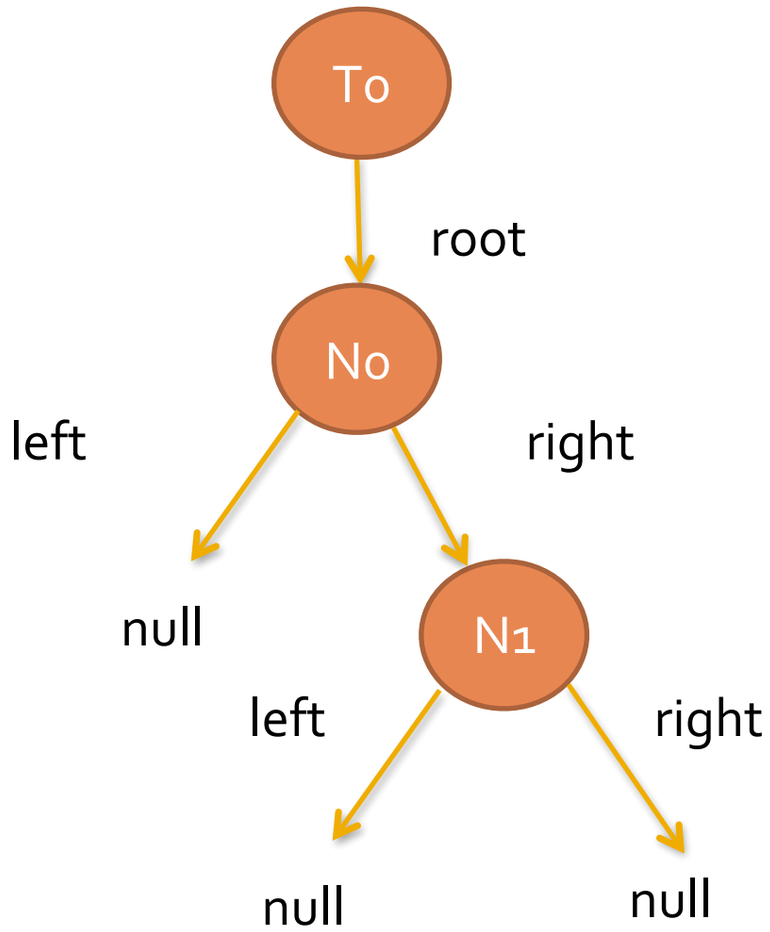
Candidate Vector



inc No.right

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

2) No.left

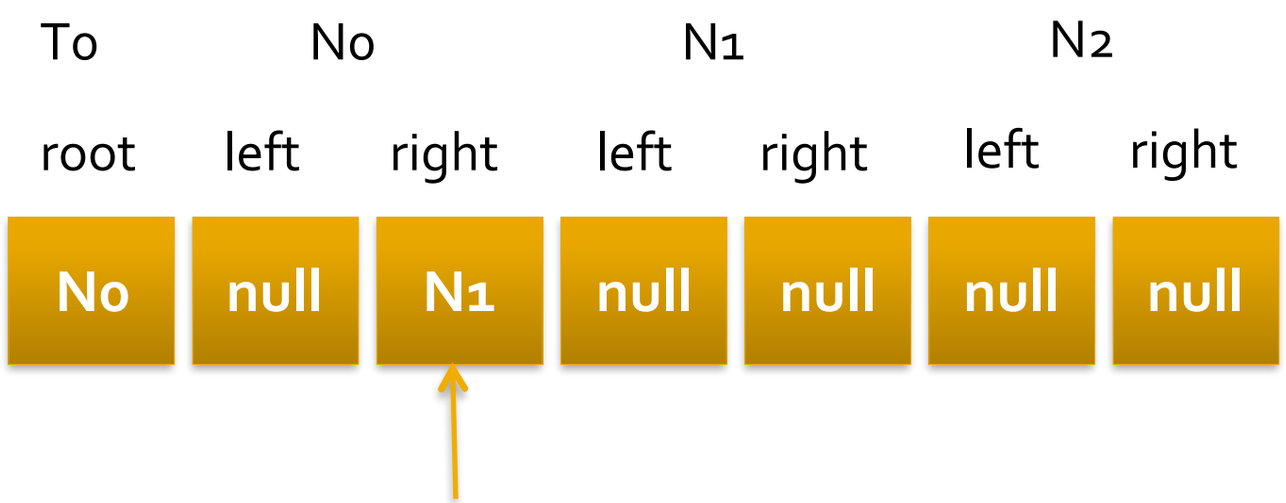
3) No.right

4) N1.left

5) N1.right ←

this.repOk()==true

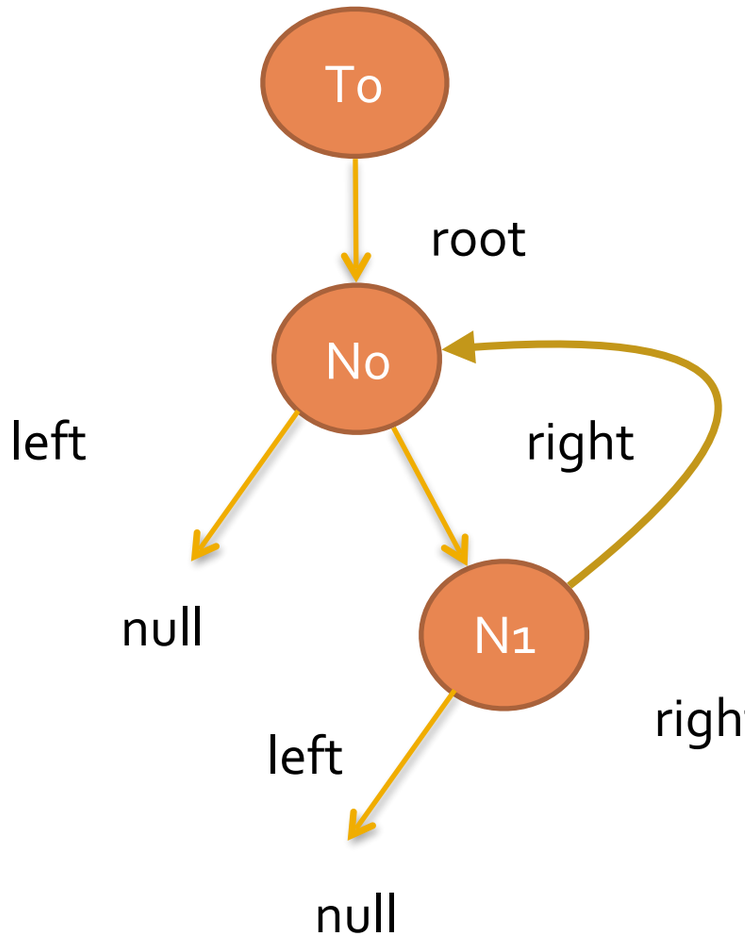
Candidate Vector



inc N1.right

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

2) No.left

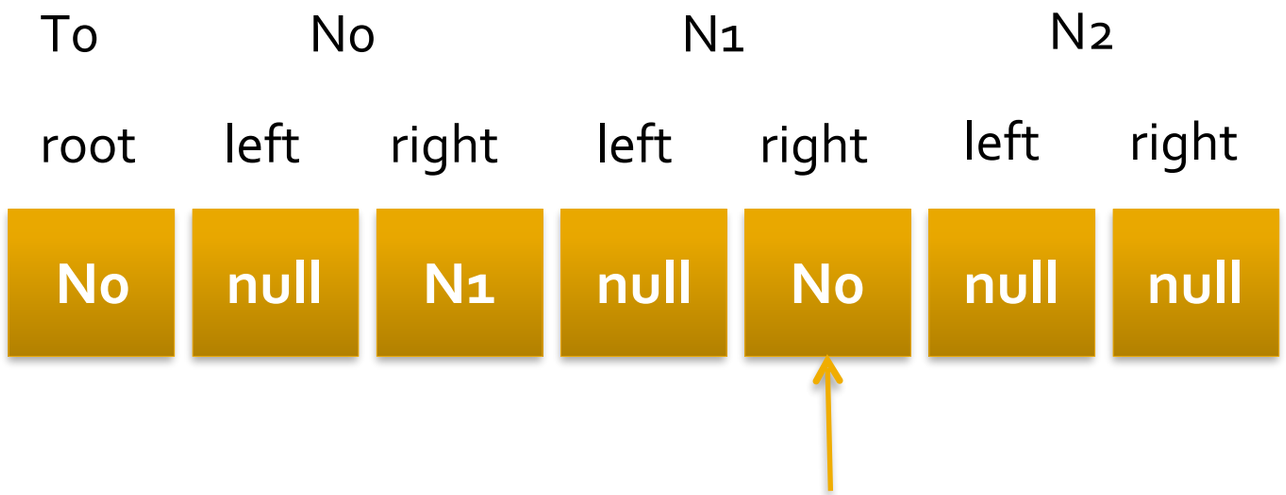
3) No.right

4) N1.left

5) N1.right ←

this.repOk()==false

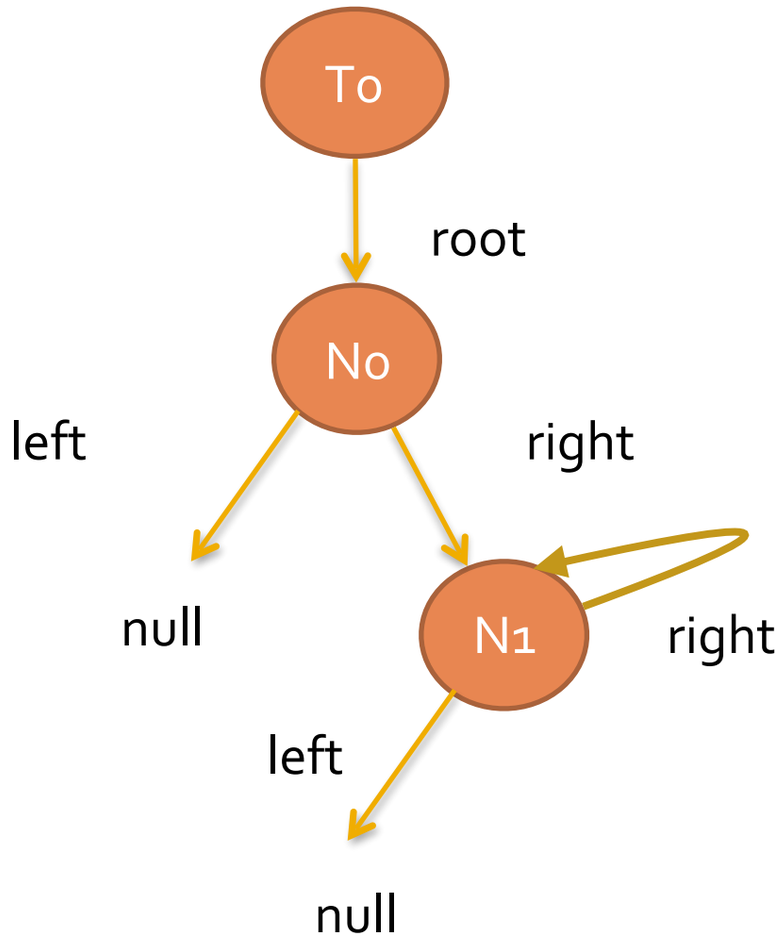
Candidate Vector



inc N1.right

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

2) No.left

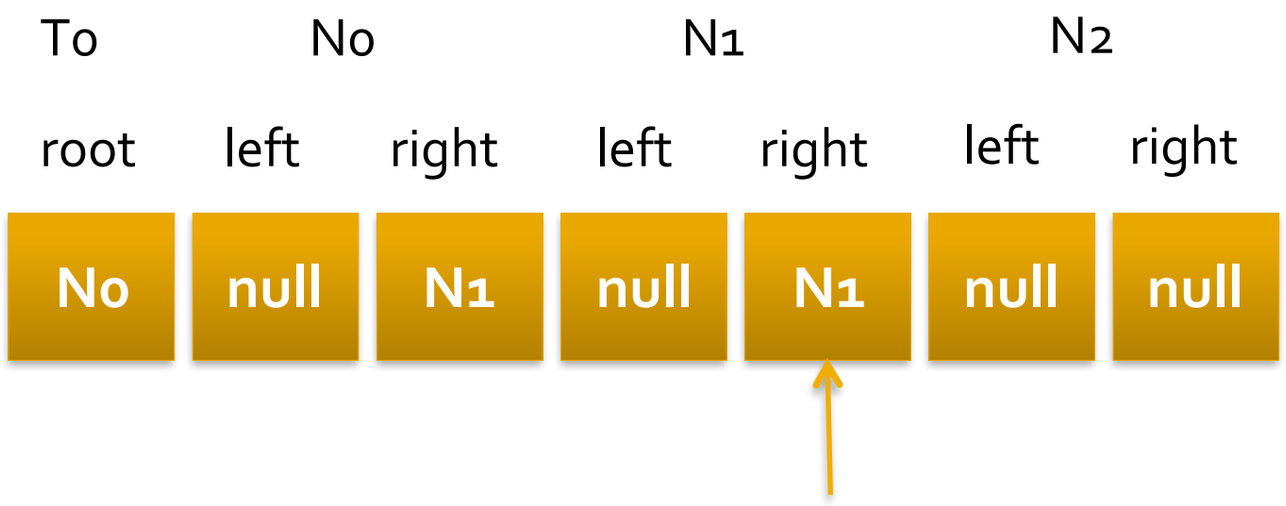
3) No.right

4) N1.left

5) N1.right ←

this.repOk()==false

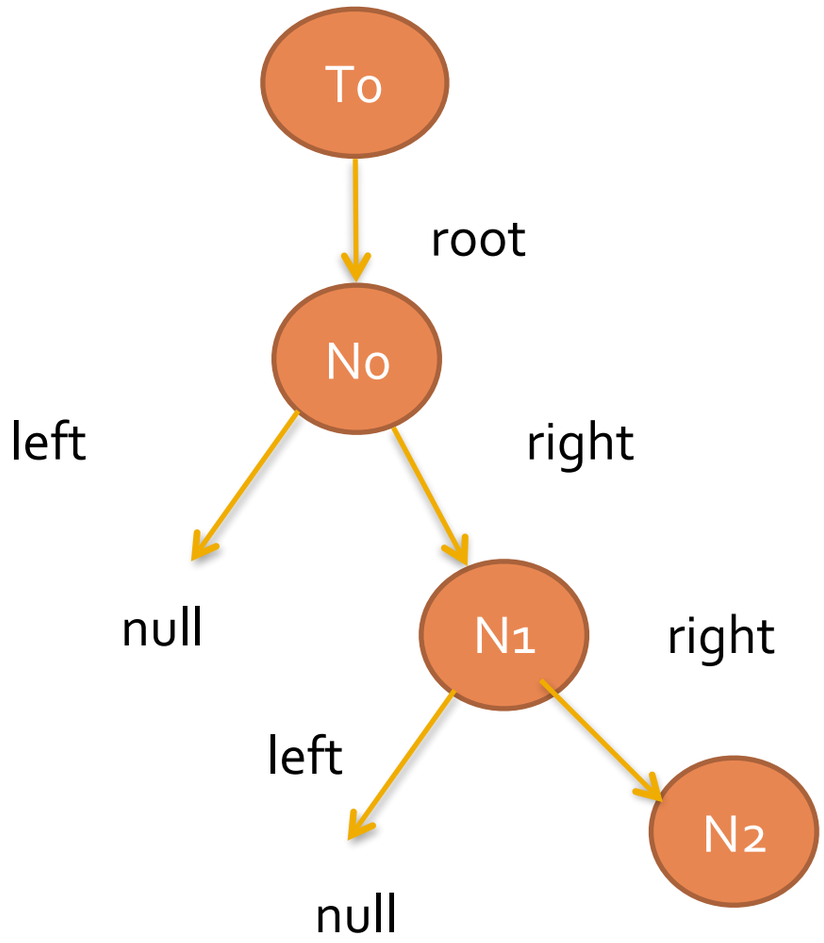
Candidate Vector



inc N1.right

Generando Candidatos

Representación Gráfica

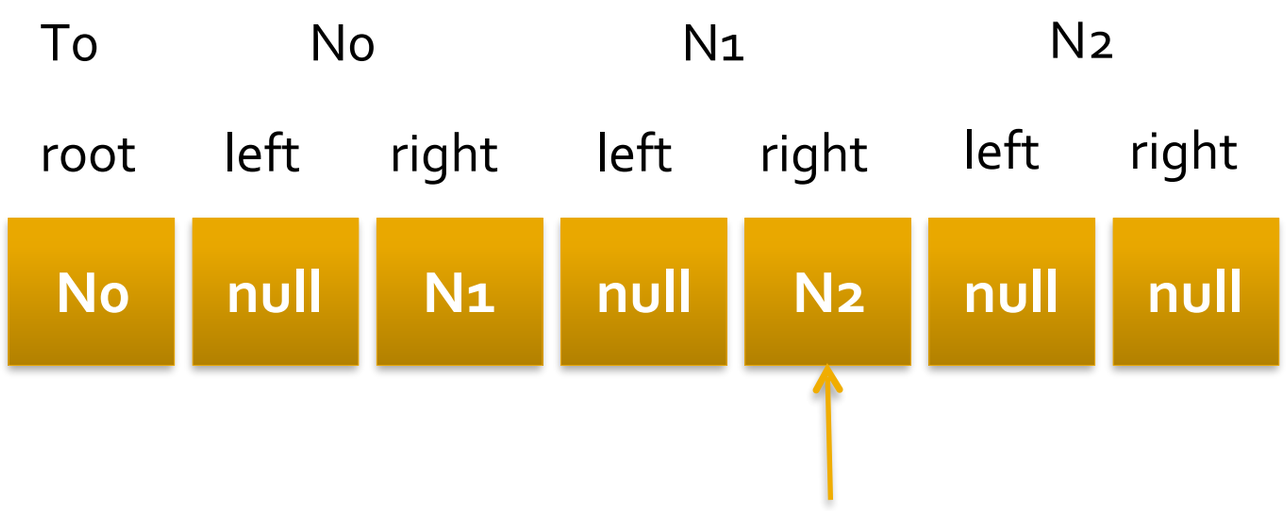


Field Ordering

1) To.root
2) No.left
3) No.right
4) N1.left
5) N1.right
6) N2.left
7) N2.right ←

this.repOk()==true

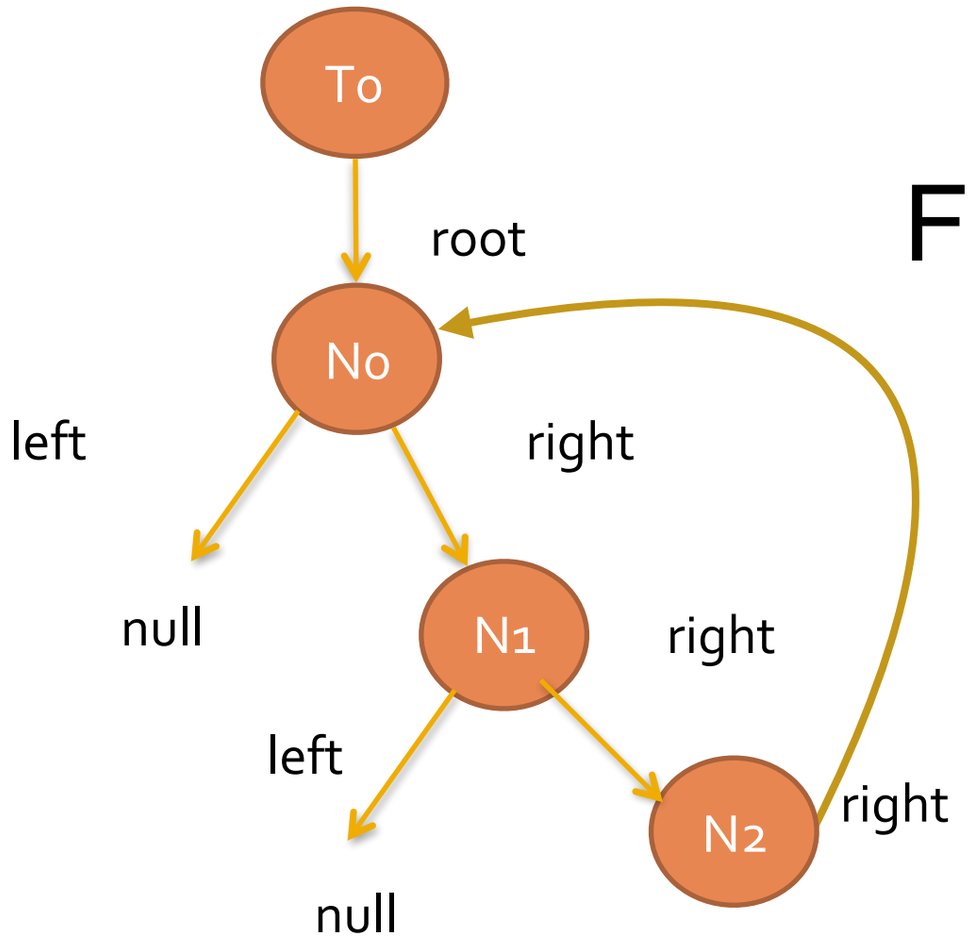
Candidate Vector



inc N2.right

Generando Candidatos

Representación Gráfica

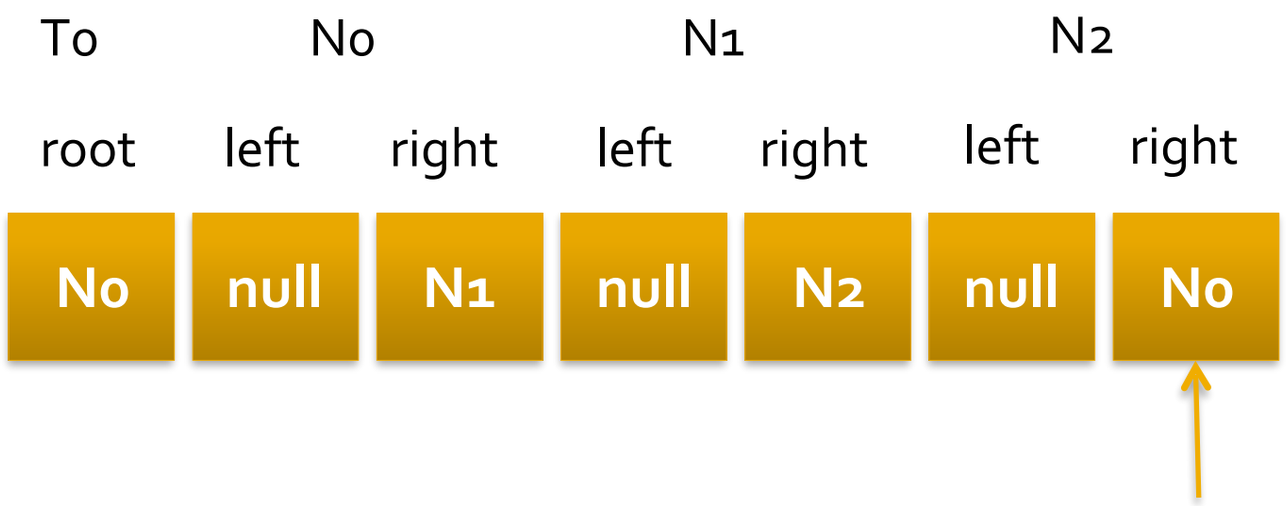


Field Ordering

1) To.root
2) No.left
3) No.right
4) N1.left
5) N1.right
6) N2.left
7) N2.right ←

this.repOk()==false

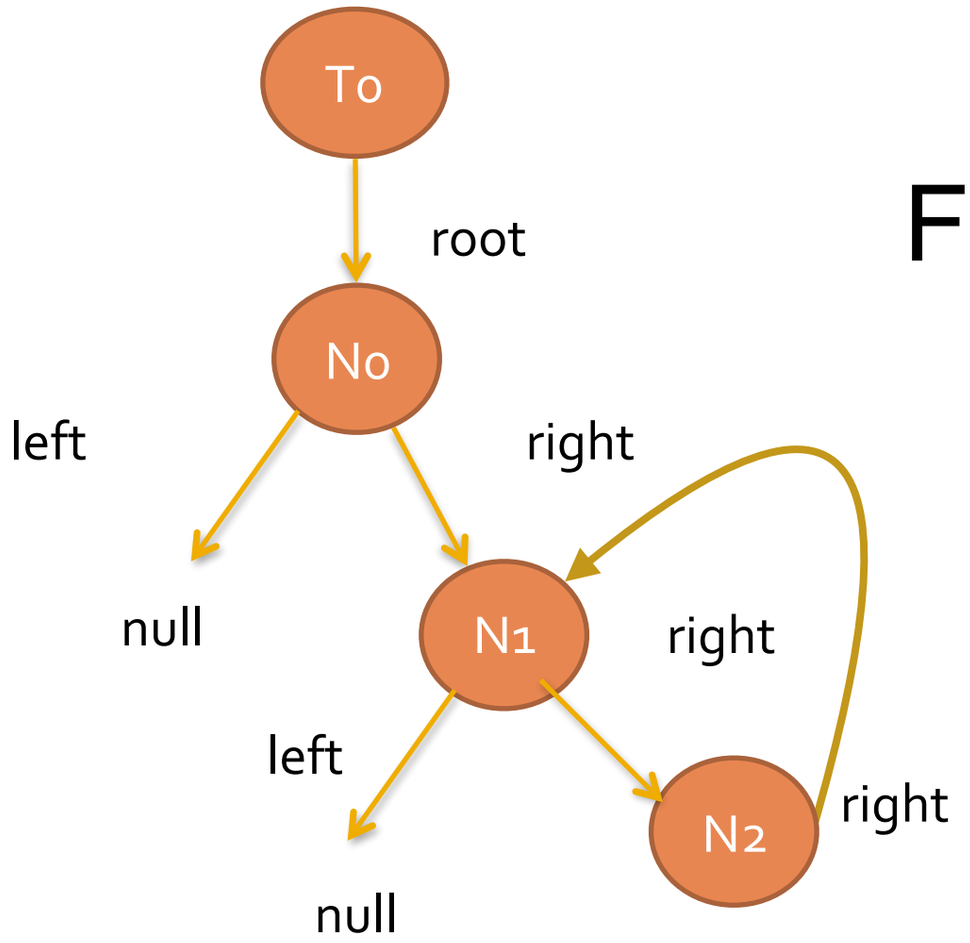
Candidate Vector



inc N2.right

Generando Candidatos

Representación Gráfica

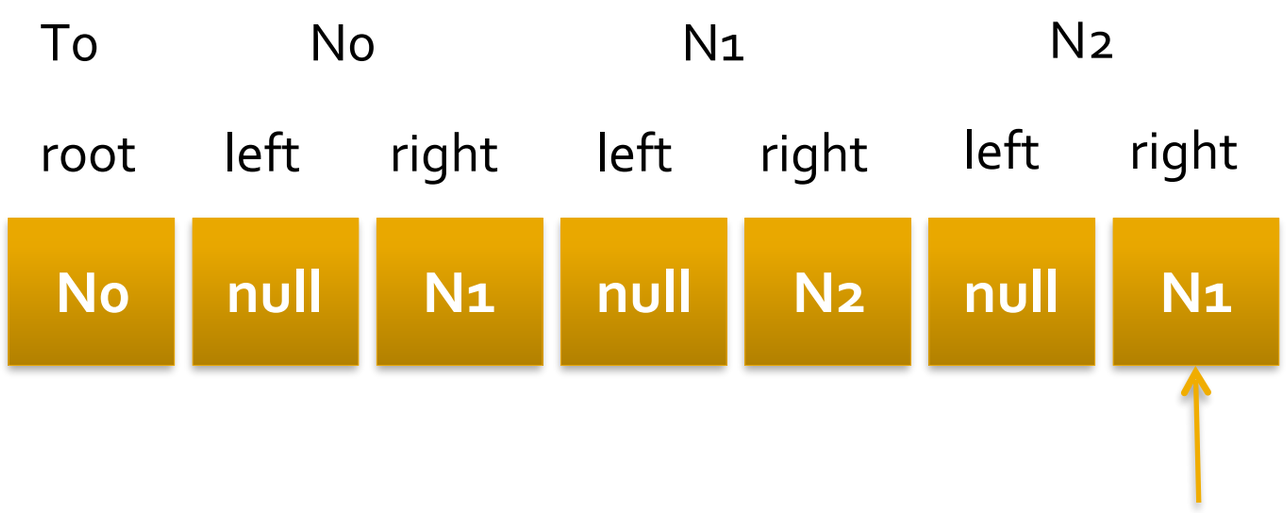


Field Ordering

1) To.root
2) No.left
3) No.right
4) N1.left
5) N1.right
6) N2.left
7) N2.right ←

this.repOk()==false

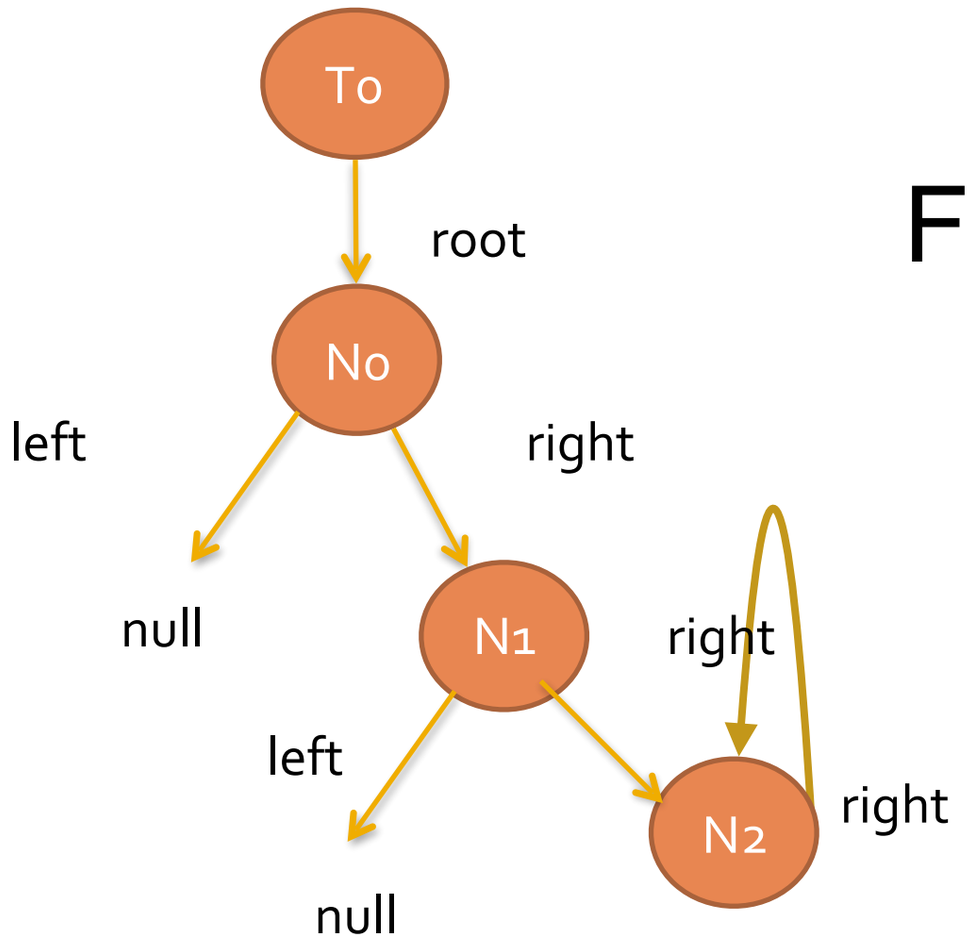
Candidate Vector



inc N2.right

Generando Candidatos

Representación Gráfica

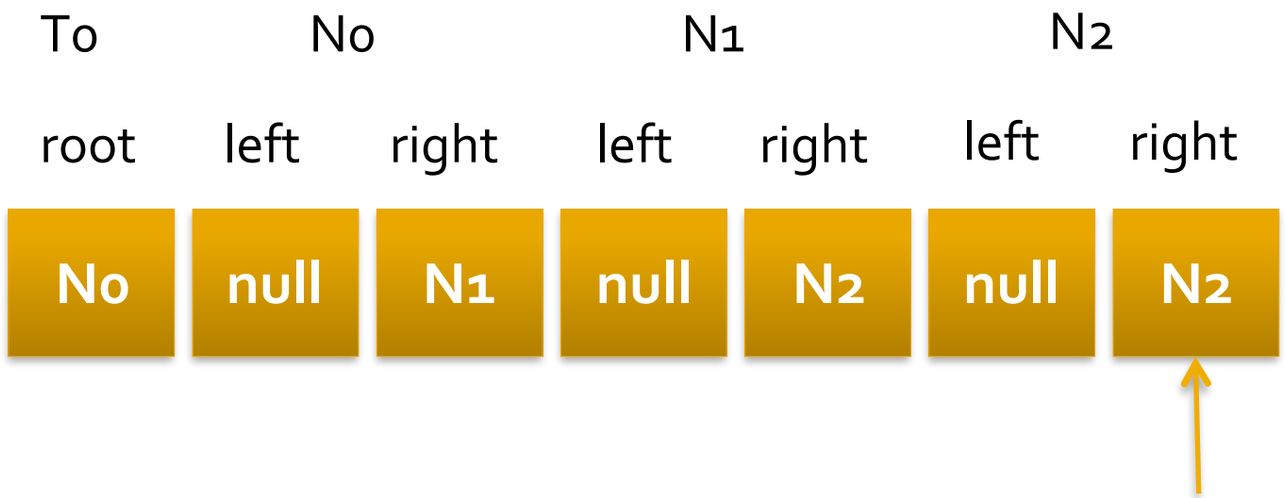


Field Ordering

1) To.root
2) No.left
3) No.right
4) N1.left
5) N1.right
6) N2.left ←
7) N2.right

this.repOk()==false

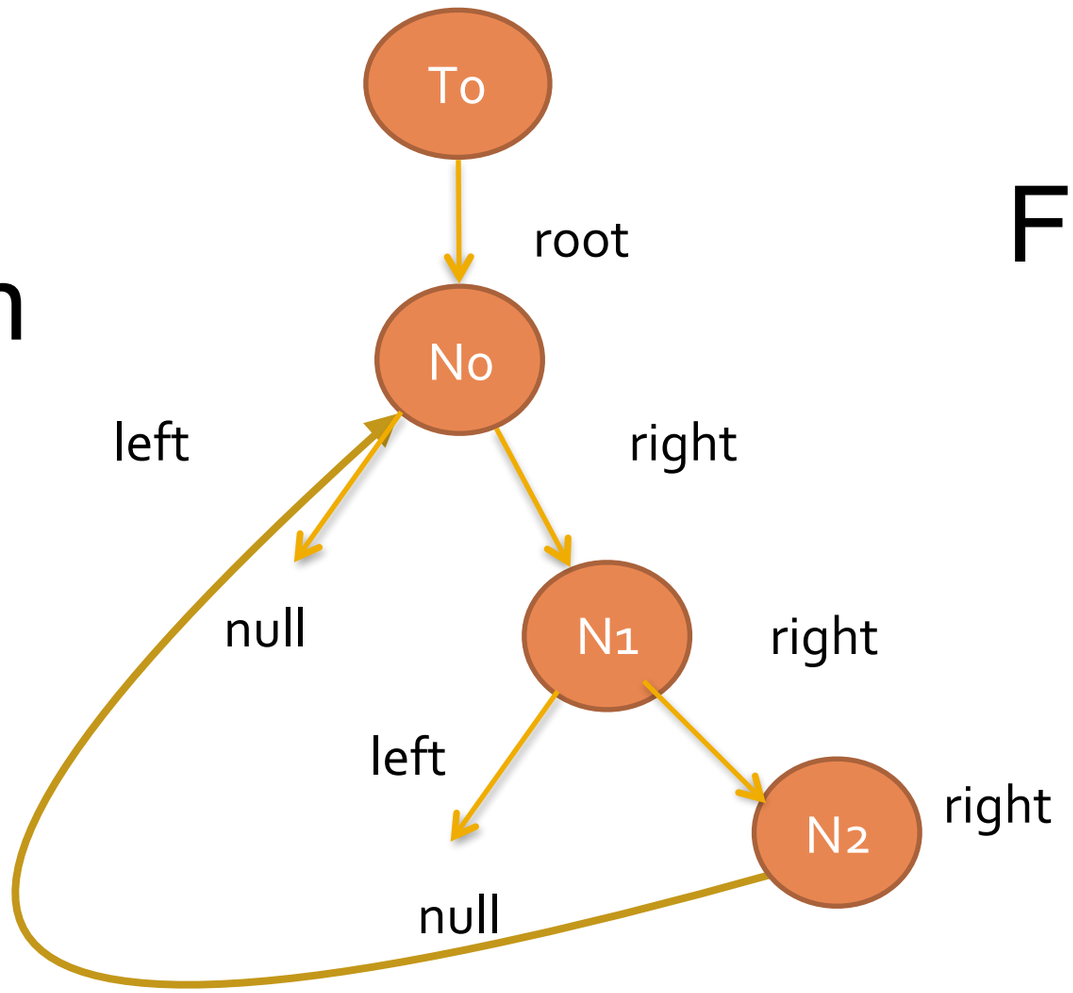
Candidate Vector



Backtrack a N2.left

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

2) No.left

3) No.right

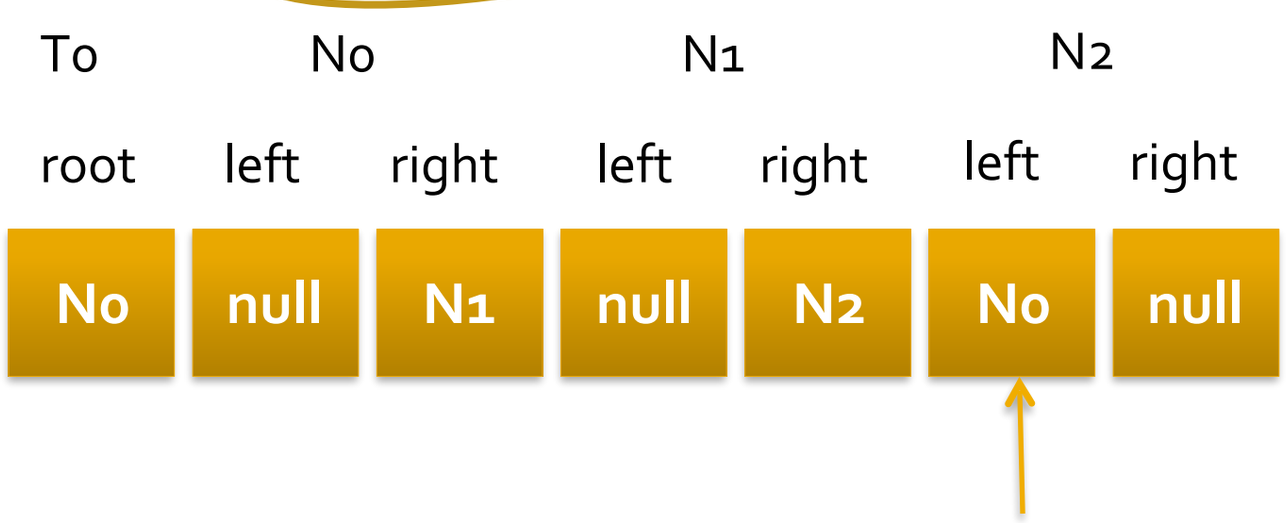
4) N1.left

5) N1.right

6) N2.left ←

this.repOk()==false

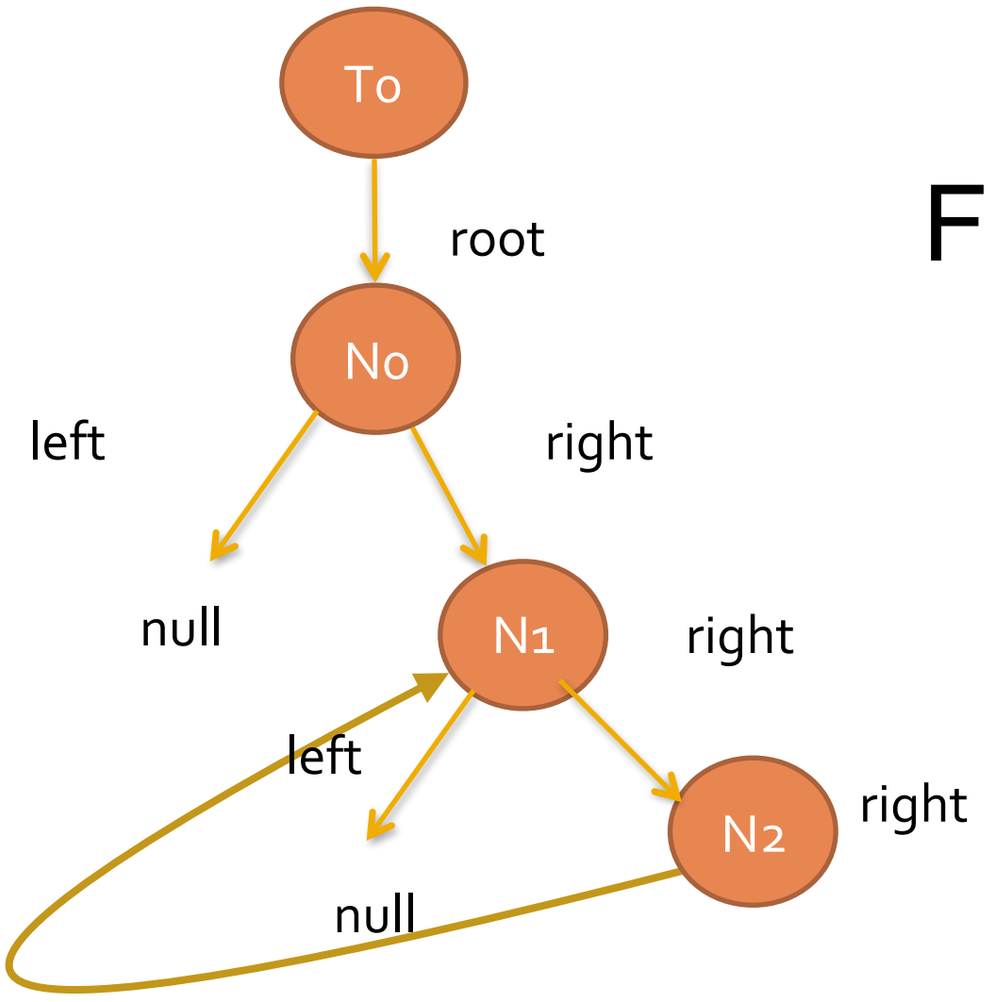
Candidate Vector



inc N2.left

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

2) No.left

3) No.right

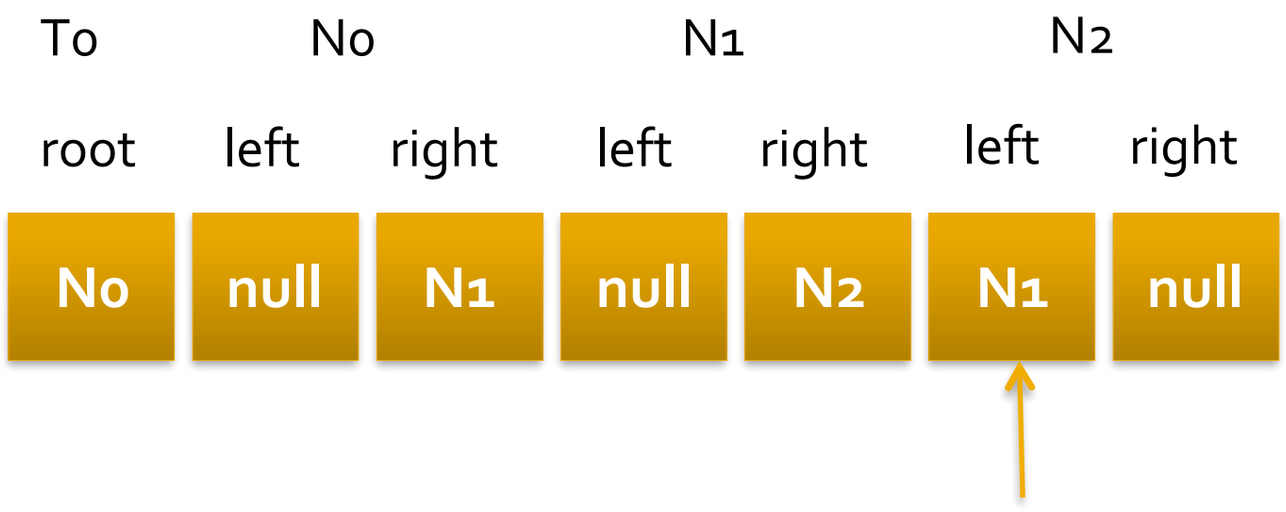
4) N1.left

5) N1.right

6) N2.left ←

this.repOk()==false

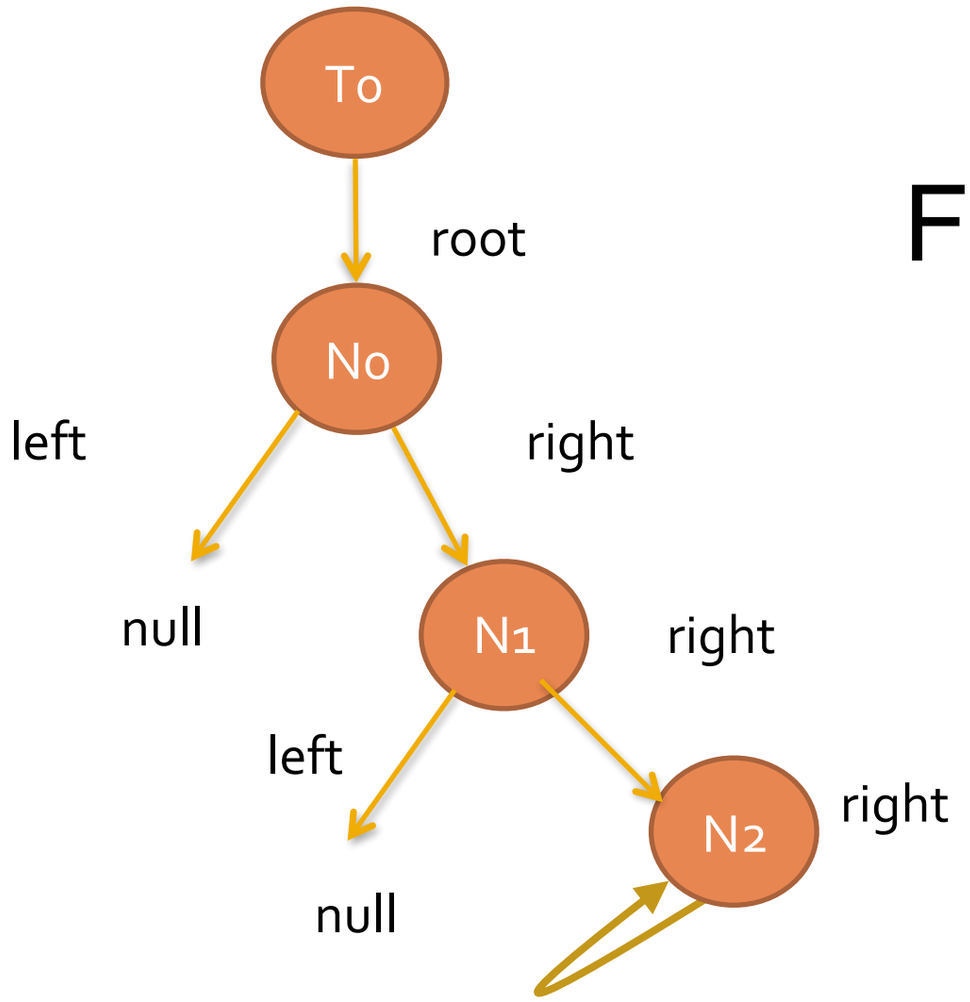
Candidate Vector



inc N2.left

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

2) No.left

3) No.right

4) N1.left ←

5) N1.right

6) N2.left

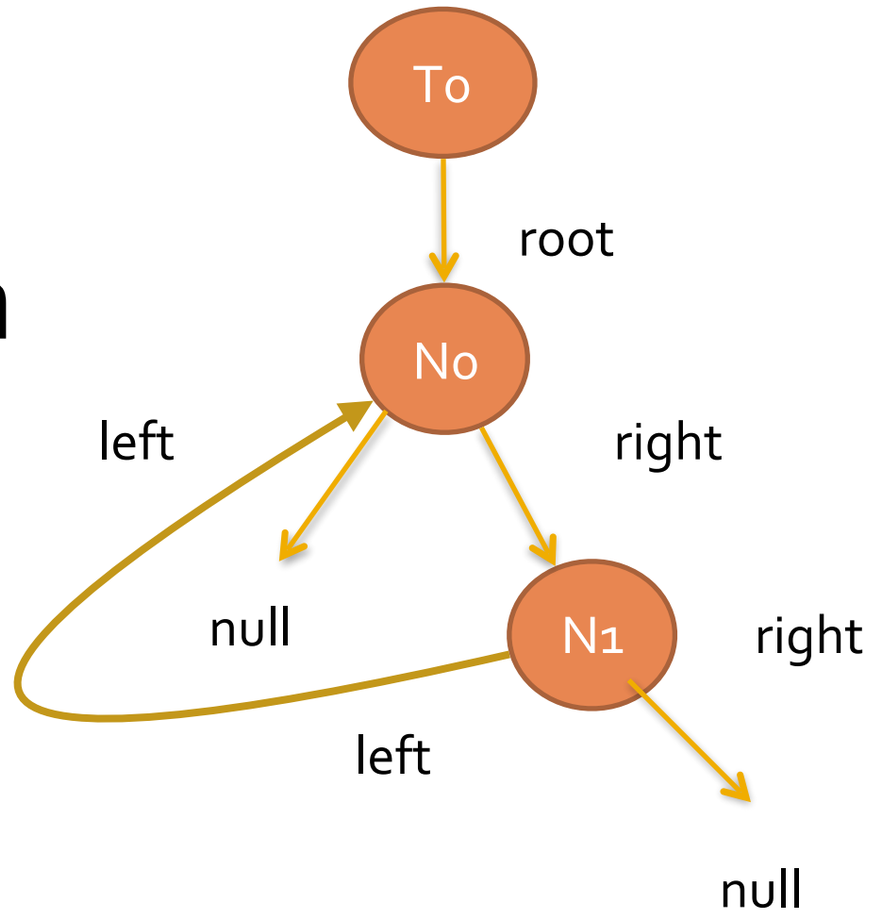
this.repOk()==false

Candidate Vector

backtrack a N1.left

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

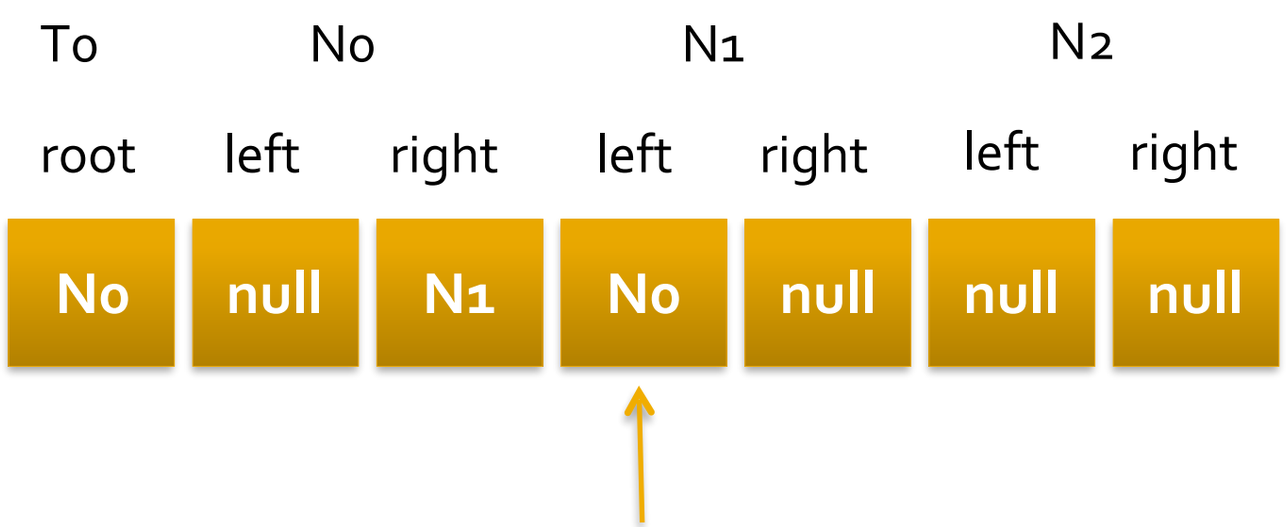
2) No.left

3) No.right

4) N1.left ←

this.repOk()==false

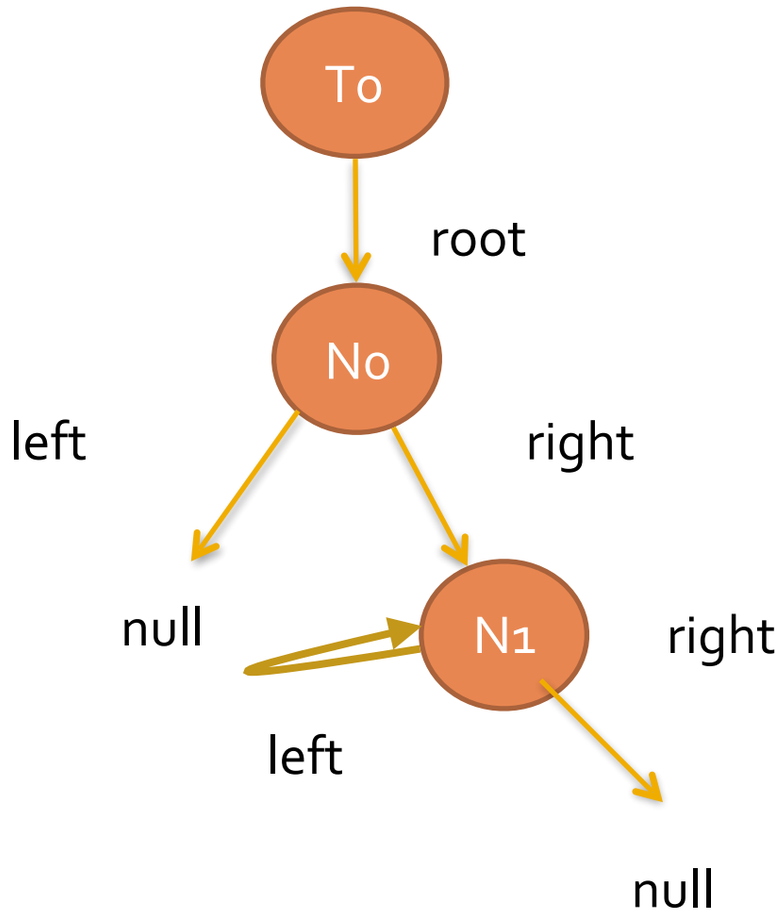
Candidate Vector



inc N1.left

Generando Candidatos

Representación Gráfica



Field Ordering

1) To.root

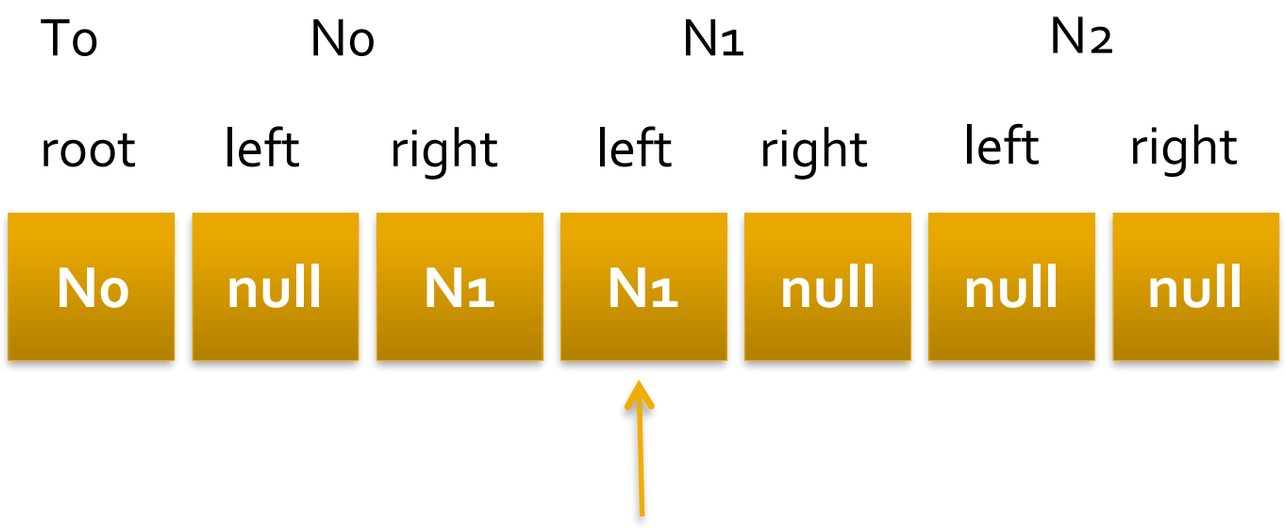
2) No.left

3) No.right

4) N1.left ←

this.repOk()==false

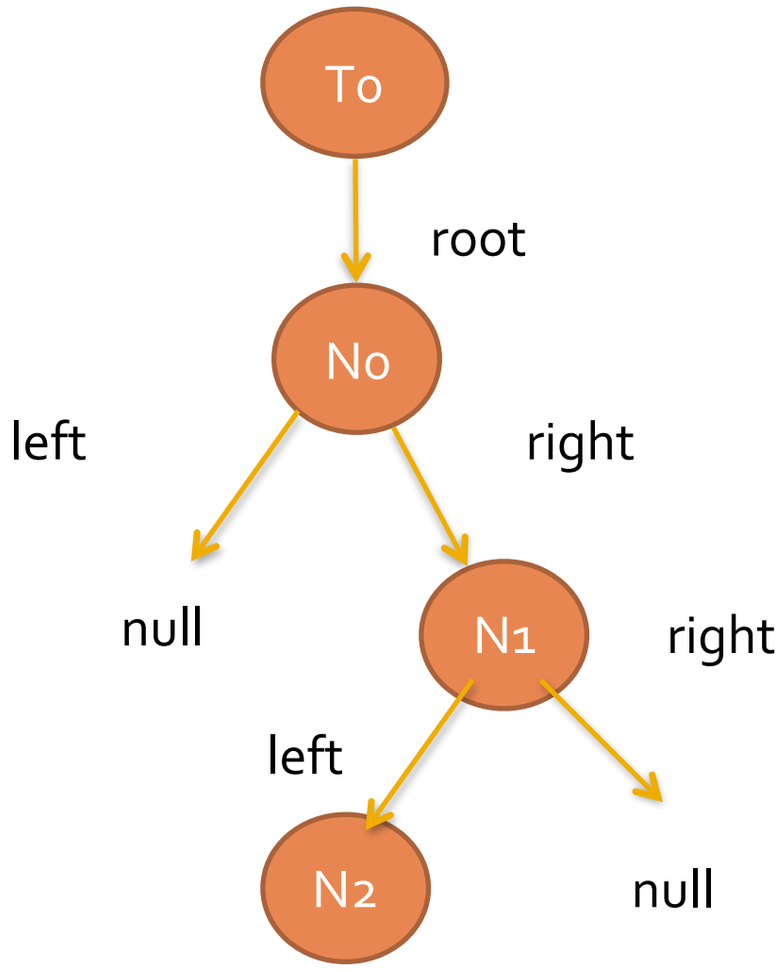
Candidate Vector



inc N1.left

Generando Candidatos

Representación Gráfica

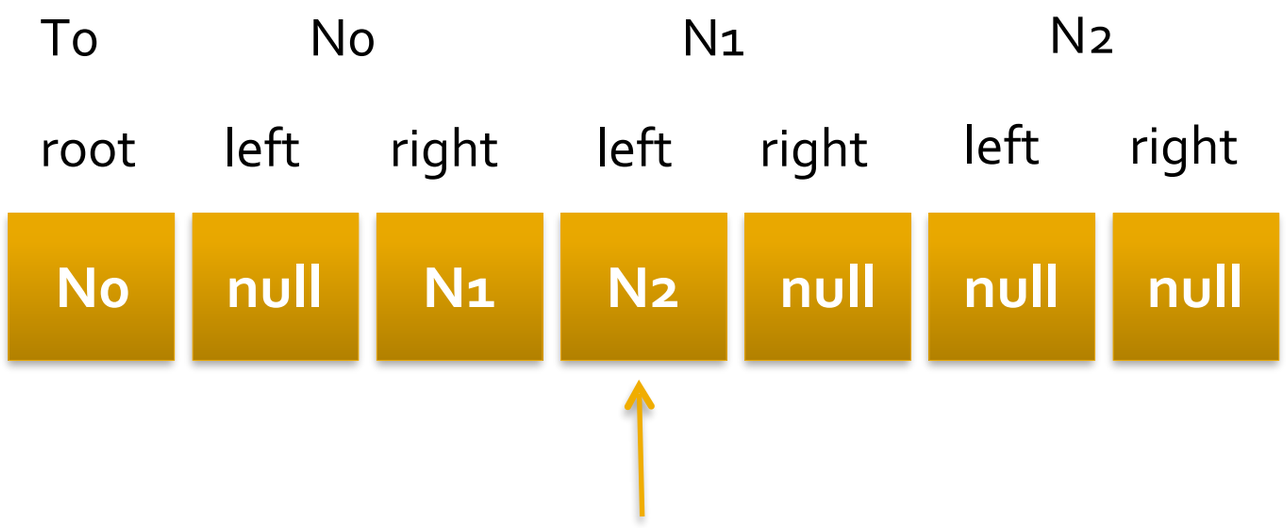


Field Ordering

1) To.root
2) No.left
3) No.right
4) N1.left
5) N1.right
6) N2.left
7) N2.right ←

this.repOk()==true

Candidate Vector



inc N2.right

Generación Basada en Predicados

```
def generator(repOK, scope):

  # inicializamos el vector con el primer candidato
  vector = init(scope)
```

Generación Basada en Predicados

```
def generator(repOK, scope):

  # inicializamos el vector con el primer candidato
  vector = init(scope)

  while True:
    # ejecutar repOK() y obtener field ordering
    fOrder, ret_val = exec repOK(vector)
    # retorno el candidato si satisface repOK()
    if ret_val==True:
      yield vector
```

```
def generator(repOK, scope):

  # inicializamos el vector con el primer candidato
  vector = init(scope)

  while True:
    # ejecutar repOK() y obtener field ordering
    fOrder, ret_val = exec repOK(vector)
    # retorno el candidato si satisface repOK()
    if ret_val==True:
      yield vector

    # Obtengo el próximo candidato
    fIndex=fOrder.removeLast()
    while not incField(vector,fIndex):
      # Si ya no puedo incrementar el field,
      # hacemos backtracking
      if len(fOrder)>0:
        fIndex = fOrder.removeLast()
```

```
    # inicializamos el vector con el primer candidato
    vector = init(scope)

    while True:
      # ejecutar repOK() y obtener field ordering
      fOrder, ret_val = exec repOK(vector)
      # retorno el candidato si satisface repOK()
      if ret_val==True:
        yield vector

      # Obtengo el próximo candidato
      fIndex=fOrder.removeLast()
      while not incField(vector,fIndex):
        # Si ya no puedo incrementar el field,
        # hacemos backtracking
        if len(fOrder)>0:
          fIndex = fOrder.removeLast()
        else:
          # Si ya no se puede hacer backtracking,
          # no existen más candidatos no isomorfos
          return # causa StopIteration
```

```
def incField(vector,fIndex):
    # Incrementa el campo fIndex
    # Aplica el mecanismo de ruptura de simetría
```

Generación Basada en Predicados

- **Únicamente extendemos *estructuras válidas***
  Si una estructura es *inválida*, no la extendemos
- **Solamente cambiamos *campos accesibles*.**
  Si el predicado *no accede a un campo*, no tratamos de alterar sus valores
- **Evitamos instancias *isomórficas*.**
  Usamos el vector candidato para identificarlas

¿Qué es más fácil?

- ¿Especificar una gramática?

```
grammar = [
      ("$START", "$TREE"),
      ("$TREE",  "$LEAF"),
      ("$TREE",  "($TREE $TREE)"),
      ("$LEAF",  "x"),
]
```

- ¿Especificar un predicado (filtro)?

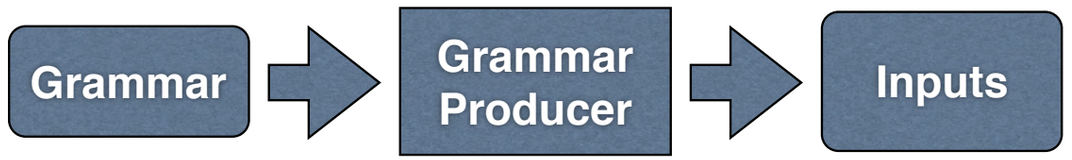
```
public boolean repOk() {
  if (this.root == null) return true;
  if (this.hasCycles()) return false;
  return true;
}
```

Grammar Producer


```
grammar = [
    ("$START", "$TREE"),

    ("$TREE",  "$LEAF"),
    ("$TREE",  "($TREE $TREE)"),

    ("$LEAF",  "x"),
]
```

```
x
(x x)
((x x) x)
(x (x x))
((x x) (x x))
(((x x) x) x)
((x (x x)) x)
(x ((x x) x))
(x (x (x x)))
```

Filtered Grammar-Based Testing

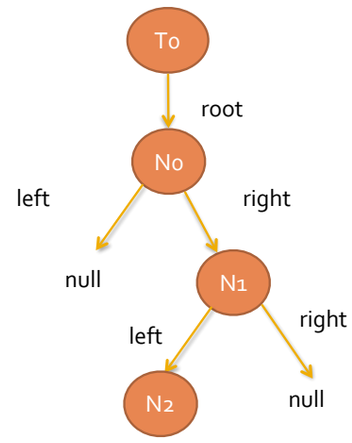


Ej:binary trees

Ej:arboles binarios de búsqueda

Generando Candidatos

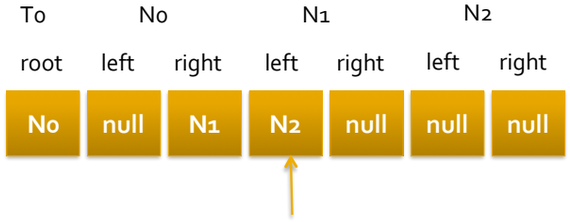
Representación Gráfica



Field Ordering

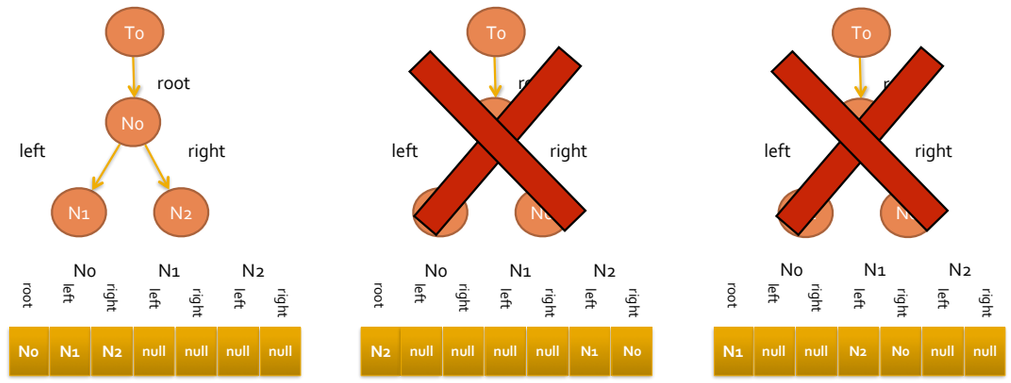
1) To.root
2) No.left
3) No.right
4) N1.left
5) N1.right
6) N2.left
7) N2.right

Candidate Vector



this.repOk()==true
(backtrack)

Rotura de Simetrías



- El índice de un elemento *e* no puede ser mayor a *k+1*, donde *k* es el mayor de los índices de todos los elementos del mismo tipo que *e*