

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Recorrido en inorden iterativo (2)

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Objetivo

- Aplicar técnicas de transformación de programas:

recursiva

```
void inorder(NodePointer &node) {  
    if (node != nullptr) {  
        inorder(node→left);  
        visit(node→elem);  
        inorder(node→right);  
    }  
}
```



iterativa.

```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        st.pop();  
        visit(x→elem);  
        descend_and_push(x→right, st);  
    }  
}
```

Objetivo: A partir de una función recursiva construir una función iterativa.

Transformación de funciones recursivas finales

Recordatorio: funciones recursivas

- Esquema general de una función recursiva simple:

Ya que solo se hace una única llamada recursiva.

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
        post_recursivo();  
    }  
}
```

suponemos que en pre_recursivo asignamos a la x otro valor distinto para acercarnos al caso base.

- Una función es **recursiva final** (o recursiva de cola) si finaliza justo después de la llamada recursiva.
- Es decir, si no se realiza ninguna acción en **post_recursivo()**.

Transformación a iterativo

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

previo();

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();
```

no se cumple la condición
del caso base.

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```


Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();  
[suponemos ¬es_caso_base(x)]
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```


Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos es_caso_base(x)}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

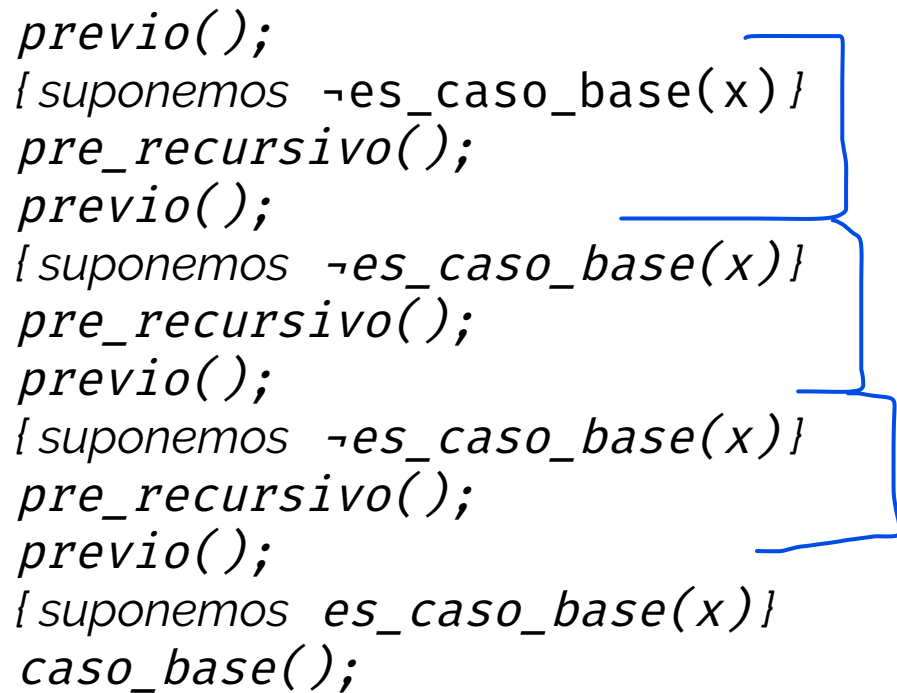
Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos es_caso_base(x)}  
caso_base();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursoivo();  
previo();  
{suponemos es_caso_base(x)}  
caso_base();
```



```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformación a iterativo

```
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();  
[suponemos ¬es_caso_base(x)]  
pre_recursoivo();  
previo();  
[suponemos es_caso_base(x)]  
caso_base();
```

```
void f(x) {  
    previo();  
    while (!es_caso_base(x)) {  
        pre_recursoivo();  
        previo();  
    }  
    caso_base();  
}
```

Transformación a iterativo

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```



Transformación.

```
void f(x) {  
    previo();  
    while (!es_caso_base(x)) {  
        pre_recursoivo();  
        previo();  
    }  
    caso_base();  
}
```

Transformación de in-order

Recorrido en inorden

```
void inorder(NodePointer node) {  
    if (node ≠ nullptr) {  
        inorder(node→left);  
        visit(node);  
        inorder(node→right);  
    }  
}
```



Dos funciones auxiliares

19

```
inorder_stack(stack<NodePointer> &st)
```

Desapila todos los elementos de st, y para cada uno de ellos:

- Visita su raíz.
- Realiza un recorrido en inorden de su hijo derecho

```
void inorder_stack(stack<NodePointer> &st) {  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder(current->right);  
        inorder_stack(st);  
    }  
}
```

recursiva

Dos funciones auxiliares

gen porque es una generalización del inorden.

```
inorder_gen(NodePointer node, stack<NodePointer> &st)
```

- Realiza un recorrido en inorden de node.
- Llama a `inorder_stack` pasándole `st` como parámetro.

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    inorder(node);  
    inorder_stack(st);  
}
```

Si `st` es una pila vacía, entonces `inorder_gen()` hace lo mismo que `inorder()`

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    inorder(node);  
    inorder_stack(st);  
}
```



Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    inorder(node);  
    inorder_stack(st);  
}
```

vamos a intercambiar ambos

```
void inorder(NodePointer node) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right);  
    }  
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right);  
    }  
    inorder_stack(st);  
}
```

```
void inorder(NodePointer node) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right);  
    }  
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right);  
    }  
    inorder_stack(st);  
}
```

El siguiente paso es querer mover el inorder dentro del if.

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right);  
    } else {  
  
    }  
    inorder_stack(st);  
}
```



Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right);  
        st.push(node);  
        inorder_stack(st);  
    } else {  
        inorder_stack(st);  
    }  
}
```

Así, inorder_stack hace el resto del trabajo.

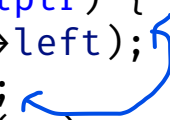
inorder_stack(stack<NodePointer> st)

Desapila todos los elementos de st, y para cada uno de ellos:

- Visita su raíz.
- Realiza un recorrido en inorden de su hijo derecho

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        st.push(node);  
        inorder_stack(st);  
    } else {  
        inorder_stack(st);  
    }  
}
```



Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        st.push(node);  
        inorder(node->left);  
        inorder_stack(st);  
    }  
    else {  
        inorder_stack(st);  
    }  
}
```

`inorder_gen(NodePointer node, stack<NodePointer> st)`

- Realiza un recorrido en inorden de node.
- Llama a `inorder_stack` pasándole `st` como parámetro.

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        st.push(node);  
        inorder_gen(node->left, st);  
    } else {  
        inorder_stack(st);  
    }  
}
```

¡Es recursiva final!

Esto ya lo podríamos pasar a una función iterativa.

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        st.push(node);  
        node = node->left;  
        inorder_gen(node, st);  
    } else {  
        inorder_stack(st);  
    }  
}
```



Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node == nullptr) {  
        inorder_stack(st);  
    } else {  
        st.push(node);  
        node = node->left;  
        inorder_gen(node, st);  
    }  
}
```



Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node == nullptr) {  
        inorder_stack(st);  
    } else {  
        st.push(node);  
        node = node->left;  
        inorder_gen(node, st);  
    }  
}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```



```
void f(x) {  
    previo();  
    while (!es_caso_base(x)) {  
        pre_recursoivo();  
        previo();  
    }  
    caso_base();  
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    inorder_stack(st);  
}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```



```
void f(x) {  
    previo();  
    while (!es_caso_base(x)) {  
        pre_recursoivo();  
        previo();  
    }  
    caso_base();  
}
```


Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    inorder_stack(st);  
}
```



Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    inorder_stack(st);  
}
```

Esta función es recursiva.

```
void inorder_stack(stack<NodePointer> &st) {  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder(current->right);  
        inorder_stack(st);  
    }  
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder(current->right);  
        inorder_stack(st);  
    }  
}
```

```
void inorder_stack(stack<NodePointer> &st) {  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder(current->right);  
        inorder_stack(st);  
    }  
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder(current->right);  
        inorder_stack(st);  
    }  
}
```

`inorder_gen(NodePointer node, stack<NodePointer> st)`

- Realiza un recorrido en inorden de node.
- Llama a `inorder_stack` pasándole `st` como parámetro.

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node→left;  
    }  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder_gen(current→right, st);  
    }  
}
```

¡Es recursiva final!

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        node = current->right;  
        inorder_gen(node, st);  
    }  
}
```

¡Es recursiva final!

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    if (st.empty()) {  
  
    } else {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        node = current->right;  
        inorder_gen(node, st);  
    }  
}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursoivo();  
        f(x);  
    }  
}
```

Transformando inorder_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    while (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        node = current->right;  
        while (node != nullptr) {  
            st.push(node);  
            node = node->left;  
        }  
    }  
}
```

iEs iterativa!

Versión iterativa

```
stack<NodePointer> st;  
NodePointer node = root;  
  
while (node != nullptr) {  
    st.push(node);  
    node = node→left;  
}  
while (!st.empty()) {  
    NodePointer current = st.top();  
    st.pop();  
    visit(current);  
    node = current→right;  
    while (node != nullptr) {  
        st.push(node);  
        node = node→left;  
    }  
}
```



Comparación

```
stack<NodePointer> st;  
NodePointer node = root;  
  
while (node != nullptr) {  
    st.push(node);  
    node = node→left;  
}  
  
while (!st.empty()) {  
    NodePointer current = st.top();  
    st.pop();  
    visit(current);  
    node = current→right;  
    while (node != nullptr) {  
        st.push(node);  
        node = node→left;  
    }  
}
```

```
stack<NodePointer> st;  
descend_and_push(root, st);  
  
while (!st.empty()) {  
    NodePointer x = st.top();  
    st.pop();  
    visit(x);  
    descend_and_push(x→right, st);  
}
```