Reto 4: Árboles

Francisco David Charte Luque Ignacio Cordón Castillo

1 Inorden no recursivo

Diseñar un procedimiento inorden no recursivo a imagen y semejanza del procedimiento preorden no recursivo que el profesor diseñó en la clase.

La idea seguida en el procedimiento de inorden iterativo es acumular en una pila cada elemento pendiente de mostrar, e ir descendiendo en los hijos izquierda de cada nodo hasta encontrar una hoja. Entonces, se imprime la hoja y se comienza a ascender por el árbol, mostrando los padres y entrando en los hijos derecha. Cada vez que se pasa por un nodo nuevo (que no se haya visitado antes), se reinicia la búsqueda en los hijos izquierda.

A continuación mostramos la implementación del método en C++:

```
void inordenNR(const ArbolBinario<int>& a){
    ArbolBinario<int>::Nodo actual;
    stack<ArbolBinario<int>::Nodo> p;
    bool subiendo = false;

    actual=a.raiz();
    p.push(ArbolBinario<int>::nodo_nulo);
    p.push(actual);

    while (actual != ArbolBinario<int>::nodo_nulo){
        if (a.izquierda(actual) != ArbolBinario<int>::nodo_nulo && !subiendo){
            // Pasamos a manejar el hijo izquierdo
            actual = a.izquierda(actual);
            p.push(actual);
        } else {
            cout << a.etiqueta(actual) << ' ';
            p.pop();
            subiendo = true;

            if (a.derecha(actual) != ArbolBinario<int>::nodo_nulo) {
```

```
                // Pasamos a manejar el hijo derecho
                actual = a.derecha(actual);
                p.push(actual);
                subiendo = false;
            } else {
                // Trataremos de saltar al hermano
                actual = p.top();
            }
        }
    }
}
```

2 Codificación de árbol binario

Dar un procedimiento para guardar un árbol binario en disco de forma que se recupere la estructura jerárquica de forma unívoca usando el mínimo número de centinelas que veais posible.

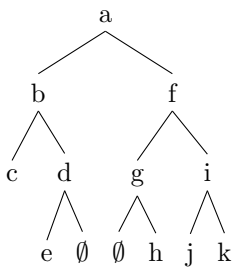
El método escogido consiste en especificar, para cada nodo del árbol binario, la manera en que se estructuran sus hijos. Es decir, en la codificación podremos conocer si un nodo tiene dos hijos, sólo el izquierdo, solo el derecho o ninguno. Para ello, se recorrerá el árbol en preorden, añadiendo después de cada nodo, en caso de ser necesario, uno de los tres centinelas siguientes:

i. `<` Le falta el hijo izquierdo

ii. `>` Le falta el hijo derecho

iii. `-` No tiene hijos

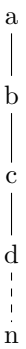
No se hará empleo de ningún centinela si el nodo tiene ambos hijos, ni tampoco tendrá centinela el último nodo, pues este siempre será `-`. Veamos un ejemplo: para el árbol siguiente



se obtendrá la codificación `abc-d>e-fg<h-ij-k`.

Para esta codificación, se tiene que en el peor caso posible para n nodos(aquel en que a cada nodo le falta un hijo, y por tanto cada nodo va acompañado de un centinela), se usan $n-1$ centinelas; esto es, se tiene que se emplean $2n-1$ posiciones (entre centinelas y etiquetas)

Ilustración del peor caso en que se toman todos los nodos excepto la raíz como hijos izquierdos del padre:



En la que la codificación sería: `a>b>c>d>...n`

El procedimiento se ha desarrollado en dos partes: la primera ejecuta una preparación de los datos, para un manejo recursivo por la segunda. Es conveniente aclarar que algoritmo 2 trabaja directamente con los datos, no con una copia, es decir, está modificando en cada ejecución el árbol `nuevo` y las pilas `hijos` y `nodos` que se pasan desde el algoritmo 1. Además, para evitar añadir complejidad innecesaria a la descripción de los algoritmos, se han identificado los tipos de dato del nodo del árbol y el de la etiqueta (se realizan asignaciones de etiquetas a nodos).

Algoritmo 1 Recuperado del árbol (I)

Entrada:
`bin_tree`, árbol binario leído codificado

Crear pila `nodos`, pila `hijos`
`centinelas` ← {`-`,`<`,`>`}

para todo `elemento` *en* `bin_tree`, excepto el último
  si `elemento` ∈ `centinelas` **entonces**
    Apilar `elemento` en `hijos`
  en otro caso
    Apilar `elemento` en `nodos`
    si `posterior` ∉ `centinelas` **entonces**
      Apilar * en `hijos`
    fin si
  fin si
  `posterior` ← siguiente a `elemento`
fin para
Apilar último elemento *en* `bin_tree` en `nodos`
Apilar `-` en `hijos`

Crear árbol `nuevo`
Raíz de `nuevo` ← tope de `nodos`
Sacar tope de `nodos`
Llamar al algoritmo 2 con parámetros {raíz de `nodos`, `nodos`, `hijos`}

devolver `nuevo`

Algoritmo 2 Recuperado del árbol (II)

Entrada:

`actual`, nodo sobre el que añadir descendientes
`nodos`, pila de nodos
`hijos`, pila de centinelas

1: `centinela` ← tope de `hijos`
2: Sacar tope de `hijos`
3: **si** `centinela` ≠ `-` **entonces**
4: **si** `centinela` ≠ `<` **entonces**
5: Hijo izquierda de `actual` ← tope de `nodos`
6: Sacar tope de `nodos`
7: Llamar al algoritmo 2 con parámetros {hijo izquierda de `actual`, `nodos`, `hijos`}
8: **fin si**

9: **si** `centinela` ≠ `>` **entonces**
10: Hijo derecha de `actual` ← tope de `nodos`
11: Sacar tope de `nodos`
12: Llamar al algoritmo 2 con parámetros {hijo derecha de `actual`, `nodos`, `hijos`}
13: **fin si**
14: **fin si**