Estructuras de Datos y Algoritmos
Grado en Ingeniería Informática
Universidad de Zaragoza – curso 2024-2025

Sesión de problemas 4
Árboles binarios de búsqueda

Objetivos

- Implementación (en pseudocódigo) de operaciones con ABBs.
- Diseño de estructuras de datos basadas en árboles binarios de búsqueda.

1. Implementación de operaciones `k_ésimo` e `insertar`

Dado un árbol binario de búsqueda (ABB, es decir, un árbol binario tal que todo nodo es mayor o igual que los nodos –si existen– de su subárbol izquierdo y menor que los nodos –si existen– de su subárbol derecho), una operación necesaria a veces es conocer la k-ésima clave más pequeña del árbol. Es decir, si el árbol contiene las claves $c_1 \leq c_2 \leq \ldots c_{k-1} \leq c_k \leq c_{k+1} \leq \ldots \leq c_n$, se trata de devolver c_k. Para hacerlo eficientemente, se almacena en cada nodo, además de la clave, un campo que guarda el número de nodos de su subárbol izquierdo.

Dados los tipos de datos que se definen a continuación:

```
tipos abb = ↑nodo;
        nodo = registro
                   laClave: clave;
                   numIzq: natural;
                   izq, der: abb
               freg
               {numIzq de un nodo guarda el número de nodos del subárbol izq. de ese nodo}
```

diseña las siguientes operaciones:

```
función k_ésimo(a: abb; k: natural) devuelve clave
{Precondición: a es un abb no vacío de n datos de tipo clave
(que pueden compararse con los operadores '=','<','>','≤','≥'), y además 1 ≤ k ≤ n}
{Postcondición: devuelve el valor de la k-ésima clave más pequeña de a}

procedimiento insertar(e/s a: abb; ent c: clave)
{Precondición: a es un abb}
{Postcondición: se añade al árbol a un ejemplar de c (exista o no otro igual)}
```

2. Implementación de la función `esABB`

Implementa la función `esABB` que devuelva un booleano indicando si el árbol `a` dado como parámetro (que es cualquier árbol binario de números naturales) es un árbol binario de búsqueda.

```
función esABB(a: arbin) devuelve booleano
{Precondición: a es un abb de números naturales
{Postcondición: devuelve cierto si y sólo sí el árbol a es un árbol binario de búsqueda}
```

3. Estructura de datos eficiente para colección ordenada por dos criterios

Tenemos una colección muy grande de datos de alumnos que nos interesa ordenar por dos campos: nombre y número de matrícula. Debe ser posible recorrer en orden creciente los nodos según su nombre y también recorrer en orden creciente los nodos según su número de matrícula.

Propón una estructura de datos que no malgaste espacio en memoria y para la que sea posible realizar de manera eficiente las operaciones típicas de mantenimiento y consulta de la colección. Para ello, **define los tipos de datos necesarios para esta estructura de datos y escribe un algoritmo que, trabajando con la estructura propuesta, escriba en pantalla los datos ordenados por nombre de alumno**.

4. Implementación de operaciones sobre árboles binarios de intervalos

En un árbol binario de intervalos (denominado *abi*) cada nodo n contiene los extremos inferior y superior de un intervalo cerrado de números con extremos enteros y también el máximo de los límites superiores de los intervalos almacenados en todos los nodos del subárbol que tiene a n por raíz. El árbol se organiza como un árbol binario de búsqueda, usando como clave de búsqueda los extremos inferiores de los intervalos (es decir, los intervalos almacenados en el subárbol izquierdo de un nodo tienen extremos inferiores menores o iguales que el extremo inferior de ese nodo, mientras que los intervalos almacenados en el subárbol derecho de un nodo tienen extremos inferiores mayores que el extremo inferior de ese nodo).

Completa la implementación del siguiente módulo, que incluye la operación de inserción y una operación de búsqueda de un intervalo cualquiera (si existe) almacenado en el árbol que tenga intersección no vacía con uno dado.

```
módulo abis
exporta
    tipo abi
    procedimiento crear(sal a: abi)
    {crea un abi vacío}
    procedimiento insertar(e/s a: abi; ent inf, sup: entero)
    {Precondición: el árbol a es un abi y además inf ≤ sup}
    {Postcondición: se añade al árbol a el intervalo [inf, sup], de manera que a
    sigue siendo un abi}
    procedimiento buscar(ent a: abi; ent infin, supin: entero; sal éxito: booleano;
                         sal infout, supout: entero)
    {Precondición: a es un abi y además infin ≤ supin}
    {Postcondición: si existe [inf, sup] en a tal que [inf, sup]∩[infin, supin] ≠ ∅ entonces
    devuelve éxito=verdad, infout = inf y supout = sup; si no, devuelve éxito = falso}
implementación
    tipos abi = ↑unDato;
          unDato = registro
                       inf, sup, max: entero;
                       izq, dch: abi
                   freg

    ... {A COMPLETAR}
fin
```