95.12 Algoritmos y Programación II

Práctica 7: árboles

Notas preliminares

- El objetivo de esta práctica es introducir distintas clases de estructuras de datos arbóreas y algoritmos para manipularlas.
- Los ejercicios marcados con el símbolo ♣ constituyen un subconjunto mínimo de ejercitación. No obstante, recomendamos fuertemente realizar todos los ejercicios.

---

1. Árboles binarios

Ejercicio 1

Dada una posible implementación dinámica de árboles binarios en C++,

```
template<typename T>
class bintree {
        T *t_;
        int n_;
        bintree *izq_;
        bintree *der_;
public:
        // ...
};
```

Hallar una relación matemática simple entre el número de punteros *null* y la cantidad de nodos del árbol, n. Mostrar, además, que esta relación no depende de la forma del árbol. Discutir, a partir de este resultado, cuánto espacio se "desperdicia" en punteros *null*.

Ejercicio 2 ♣

(a) Proponer un algoritmo recursivo que, dado un árbol binario de n nodos, imprima la clave de cada nodo del árbol. Además, la complejidad debe ser una $O(n)$.

(b) Proponer un algoritmo no recursivo que, dado un árbol binario de n nodos, imprima la clave de cada nodo del árbol. Usar una pila como estructura de datos auxiliar. Además, la complejidad debe ser una $O(n)$.

(c) Proponer un algoritmo que, dado un árbol binario de n nodos, imprima la clave de cada nodo del árbol. Además, sólo se permite usar una cantidad de espacio de almacenamiento constante (es decir, la cantidad de memoria extra a usar -sin contar la memoria consumida por la representación del árbol- no debe depender de n). No se permite modificar el árbol, ni siquiera temporalmente, durante el proceso. La complejidad resultante debe ser una $O(n)$.

Nota: se puede suponer que la implementación provee un método `up()` que, dado un nodo o subárbol, devuelve el padre de ese nodo en tiempo constante.

Ejercicio 3 ♣

Probar que, conociendo los recorridos *preorder* e *inorder* de un árbol binario arbitrario sin elementos repetidos, se puede reconstruir la estructura original del mismo.

¿Puede afirmarse un resultado similar conociendo los recorridos en *preorder* y *postorder* (en vez de *inorder*)? ¿Y *postorder* e *inorder*?

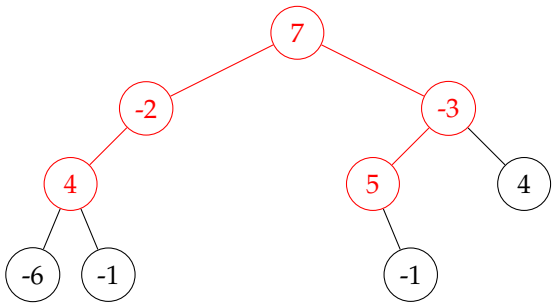
Ejercicio 4

Dado un árbol binario de enteros t y dos nodos $u, v \in t$, llamamos *peso de camino* entre u y v, notado $\pi(u, v)$, a la suma del valor asociado a cada nodo en el camino que une a u con v, incluyendo a ellos mismos. Cuando es $u = v$, $\pi(u, v)$ es el valor asociado a u.

(a) Diseñar un algoritmo que, dado un árbol binario t de n enteros, determine el máximo peso de camino π_t entre dos nodos cualesquiera de t, y calcular luego su complejidad temporal. Puesto en términos formales, se desea calcular lo siguiente:

$$\pi_t = \text{máx}\,\{\pi(u, v) \;/\; u, v \in t\}$$

Por ejemplo, para el árbol t mostrado abajo, el algoritmo debe devolver $\pi_t = 11$, que corresponde al camino formado por los nodos marcados en rojo.



(b) Pensar cómo mejorar el algoritmo anterior para que corra en tiempo $O(n)$ y no pase más de una vez por cada nodo de t.
Pista: considerar el uso de la técnica de generalización de funciones.

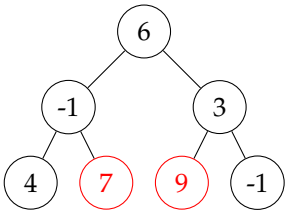
Ejercicio 5 ♣

Sea t un árbol binario completo de n enteros.

(a) Proponer un algoritmo iterativo que corra en tiempo $\Theta(n)$ para encontrar la mínima distancia δ_t entre dos nodos adyacentes de t. En otras palabras, se debe calcular lo siguiente:

$$\delta_t = \text{mín}\,\{|u - v| \;/\; u \text{ y } v \text{ son nodos adyacentes en } t\}$$

Por ejemplo, para el árbol t mostrado abajo, el algoritmo debe devolver $\delta_t = 2$, que corresponde a los nodos marcados en rojo.



(b) Proponer un algoritmo que implemente la técnica de dividir y conquistar para calcular δ_t. Se debe mantener la misma complejidad temporal que en el caso anterior.

(c) Calcular la complejidad espacial de ambos algoritmos. ¿Cuál resulta más eficiente?

Ejercicio 6

Supongamos la siguiente declaración de árboles binarios:

```
template<typename T>
class bintree {
    T *t_;
    bintree *left_;
    bintree *right_;
public:
    bool is_subtree_of(const bintree<T>&) const;
    // ...
};
```

En esta implementación, los árboles vacíos son representados mediante instancias de la clase con todos sus atributos nulos.

Implementar, en C++, el método `bool is_subtree_of(const bintree<T>&) const` que permite determinar si el árbol apuntado por `this` es subárbol del árbol pasado como argumento. Dado un árbol binario t con subárboles izquierdo y derecho t_i y t_d respectivamente, decimos que un árbol binario s es *subárbol* de t si $s = t$ o s es subárbol de t_i o de t_d.

Puede asumirse que el tipo paramétrico `T` tiene sobrecargado el operador `==`.

2. Árboles binarios de búsqueda y balanceados

Ejercicio 7 ♣

(a) Para cada secuencia de claves, dibujar el árbol binario de búsqueda que resulta de insertar las claves, una por una, en un árbol inicialmente vacío:

(i) $\{1, 2, 3, 4, 5, 6, 7\}$;

(ii) $\{4, 2, 1, 3, 6, 5, 7\}$;

(iii) $\{1, 6, 7, 2, 4, 3, 5\}$.

(b) Repetir el ejercicio anterior usando árboles AVL.

Ejercicio 8

(a) Proponer un algoritmo que reciba un árbol AVL t y dos valores, a y b, $a \leq b$, y visite todas las claves $x \in t$ tales que $a \leq x \leq b$. El tiempo de corrida del algoritmo debe ser $O(k + \log n)$, donde k es el número de claves visitadas y n el número total de elementos en el árbol.

(b) Analizar cómo cambiaría la complejidad si t fuese un árbol binario de búsqueda arbitrario en lugar de un árbol AVL.

(c) Analizar cómo adaptar el algoritmo para soportar como entrada árboles binarios arbitrarios. ¿Qué pasa con la complejidad en este caso?

Ejercicio 9 ♣

(a) Proponer la interfaz de una clase C++ para representar árboles AVL.

(b) Escribir en dicha clase un método de complejidad sublineal que permita calcular la altura de un árbol AVL (i.e., la mayor de las distancias de los caminos que unen la raíz con cada una de las hojas).

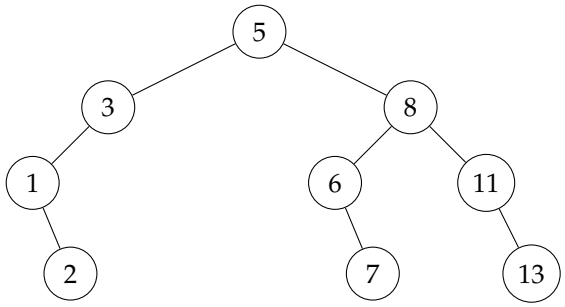
Ejercicio 10

(a) Diseñar un algoritmo que reciba un arreglo $A[1 \dots n]$ de números enteros, ordenado ascendentemente, y construya, en tiempo lineal, un árbol binario de búsqueda conteniendo esa información. El árbol generado deberá ser balanceado, es decir, es necesario que la profundidad de todas las hojas sea l o $l-1$ para algún $l \in \mathbb{N}$.

(b) Explicar qué cambios se necesitaría introducir en el algoritmo si se requiriese obtener un árbol completo (i.e., todas las hojas a profundidad l o $l-1$ y estando las hojas del nivel l ocupando las posiciones consecutivas desde más a la izquierda, sin dejar huecos).

Ejercicio 11

Dado un árbol binario t y dos nodos $u, v \in t$, el *ancestro común más cercano* de u y v, notado $\text{LCA}_t(u, v)$ (del inglés *lowest common ancestor*), es el nodo w más profundo en t tal que u y v son descendientes de w. Por ejemplo, en el árbol binario de búsqueda t de la figura, se tiene que $\text{LCA}_t(7, 11) = 8$ y que $\text{LCA}_t(1, 13) = 5$:



Supongamos la siguiente declaración de árboles binarios de búsqueda:

```
template<typename T>
class bst {
        T *t_;
        bst *left_;
        bst *right_;
public:
        T lca(const T&, const T&) const;
        // ...
};
```

En esta implementación, los árboles vacíos son representados mediante instancias de la clase con todos sus atributos nulos.

a) Implementar, en C++, el método `T lca(const T&, const T&) const` que permite encontrar el ancestro común más cercano de dos nodos dados. Puede asumirse que el tipo paramétrico `T` tiene sobrecargados los operadores de comparación.

b) Calcular la complejidad temporal de peor caso de este algoritmo en función de la cantidad de nodos del árbol, n.

c) Si el dominio de este algoritmo se restringiese a árboles AVL, ¿la complejidad calculada en el ítem anterior se vería afectada?

Ejercicio 12 ♣

Analizar si cada una de las siguientes aseveraciones es verdadera o falsa, justificando cuidadosamente las respuestas dadas:

(a) Sean a_0 y a_1 dos árboles AVL de números reales. Entonces, existe por lo menos un número r tal que alguno de los árboles t_0 o t_1 es AVL, donde t_i es el árbol con r como raíz, a_i como subárbol izquierdo y a_{1-i} como subárbol derecho ($i = 0, 1$).

(b) Sea t un árbol binario de n nodos y sea $\kappa_i(t)$ la cantidad de nodos en el i-ésimo nivel de t, $1 \leq i \leq h(t)$. Entonces, es posible encontrar por lo menos un nivel i tal que $\kappa_i(t) \in \Theta(n)$.

(c) La cantidad de hojas de cualquier árbol binario de búsqueda de n nodos es $\Theta(n)$.

(d) La cantidad de hojas de cualquier árbol AVL de n nodos es $\Theta(n)$. **Pista**: es verdadero pero sumamente difícil de probar. Recomendado para pensar en las vacaciones ☺.

(e) Dado un árbol AVL arbitrario, la diferencia de tamaño (i.e., cantidad de nodos) entre sus subárboles es $O(1)$.

(f) Sea t un árbol AVL de n nodos y sea e un elemento no presente en t. La siguiente recurrencia permite caracterizar la máxima cantidad de comparaciones realizadas al buscar a e en t:

$$T(n) = T(n/2) + O(1)$$

(g) Sea t_h un árbol binario de altura h en el que el subárbol izquierdo es el árbol completo de altura $h-2$, C_{h-2}, y el subárbol derecho es el árbol de Fibonacci de altura $h-1$, F_{h-1}. Sea $\delta(h) = ||C_{h-2}| - |F_{h-1}||$ la diferencia de tamaño entre los subárboles de t_h. Entonces, $\delta(h) \in O(\phi^h)$, en donde $\phi = \frac{1+\sqrt{5}}{2}$ es el número de oro. **Pista:** recordar que $|F_h| = \text{fib}_{h+1} - 1$.

(h) Si t es un árbol binario tal que su recorrido preorder está ordenado crecientemente, todo nivel de t está ordenado crecientemente.

(i) Si t es un árbol binario de búsqueda de $n > 2$ nodos, el recorrido preorder de t no puede estar ordenado crecientemente.

Ejercicio 13

(a) Probar que, dado un árbol binario de búsqueda arbitrario t, es posible reconstruir a t a partir de un arreglo $P[1 \dots n]$ que representa su recorrido preorder.

(b) Probar que, dado un árbol binario de búsqueda arbitrario t, es posible reconstruir a t a partir del arreglo L que representa su recorrido por niveles.

(c) Probar que no existe ningún algoritmo que, dados n números, construya en tiempo $O(n)$ un árbol binario de búsqueda conteniendo esos números.

Ejercicio 14

Un *árbol RN* es un árbol binario cuyos nodos pueden ser rojos o negros y que además verifica las siguientes condiciones:

- Todas sus hojas son de color negro,
- Los nodos rojos pueden tener únicamente hijos negros, y
- Cada camino desde la raíz hasta cualquiera de las hojas tiene la misma cantidad de nodos negros.

(a) Proponer un algortimo que utilice la técnica de dividir y conquistar para determinar si un árbol binario es RN. Asumir que se cuenta con una función `color()` para conocer el color de un nodo en tiempo constante.

(b) Calcular la complejidad temporal de peor caso del algoritmo anterior.

Ejercicio 15

Sea t un árbol AVL arbitrario, y sea $P[1 \dots n]$ un arreglo conteniendo el recorrido preorder de t. Diseñar un algoritmo que, dado e y el arreglo P, determine si $e \in t$ en tiempo estrictamente inferior que $O(n)$. Indicar claramente la complejidad temporal del algoritmo desarrollado.

Ejercicio 16 ♣

(a) Proponer un algoritmo que, dado un árbol binario arbitrario t de n nodos, determine si t es un árbol de búsqueda en tiempo $O(n)$.

(b) Proponer un algoritmo que, dado un árbol binario arbitrario t de n nodos, determine si t es un árbol AVL en tiempo $O(n)$.

Pista: para ambos ítems, considerar el uso de la técnica de generalización de funciones.

Ejercicio 17

Sea t un árbol binario y sea $\delta : \mathbb{N} \to \mathbb{N}$. Decimos que t es un *árbol-δ* si se verifican simultáneamente las siguientes condiciones:

- t es un árbol binario de búsqueda,
- La diferencia entre la cantidad de nodos de los subárboles de t es a lo sumo $\delta(|t|)$, donde $|t|$ indica la cantidad de nodos de t, y
- Ambos subárboles de t son a su vez árboles-δ.

Sea $\delta_k(n) = k$ (i.e., la función constante k). Dado un árbol-δ_k t de n nodos, ¿es posible implementar un algoritmo de búsqueda sobre t en tiempo $O(\log n)$?

Ejercicio 18

Sean $V =$  y $W =$ dos árboles binarios. Consideremos la familia de árboles $\{T_n\}_{n \geq 1}$, en donde $T_1 = \bullet$ y, para $n \geq 2$, T_n se obtiene agregando dos hijos a todas las hojas de T_{n-1}: W a izquierda y V a derecha.

(a) Dar una expresión en función de n para $|T_n|$, la cantidad de nodos de T_n.

(b) Analizar si la altura de T_n es $\Theta(\log |T_n|)$.

(c) Encontrar todos los valores de $n \in \mathbb{N}$ para los cuales T_n satisface la condición de balance de los árboles AVL.

Ejercicio 19

Sea $A(h)$ la cantidad de árboles AVL estructuralmente distintos de altura h.

(a) Proponer una recurrencia que defina $A(h)$.

(b) Probar que $A(h) \in O(3^{2^h})$.