Algoritmos y Estructuras de Datos (VJ1215) - Universitat Jaume I
Examen final - 2023/2024 - Primera parte

6 de junio de 2024

Nombre:

El examen final consta de dos partes con una duración de dos horas cada una. Este es el enunciado de la primera parte, que suma 5 puntos. La prueba es individual. No puedes consultar libros, apuntes ni dispositivos electrónicos. Al finalizar entrega tu solución junto con el enunciado (no es necesario que entregues todas las hojas). Pon tu nombre en cada hoja que entregues.

En los ejercicios 1 y 3 escribe tu solución empleando el lenguaje C++.

Ejercicio 1 — 1 punto

Estamos utilizando una lista simplemente enlazada para realizar una implementación del Tipo Abstracto de Datos **Conjunto** a la que se ha añadido la posibilidad de consultar y eliminar el mínimo eficientemente. Para ello, tenemos los siguientes atributos, y no puedes añadir otros atributos en `Conjunto` ni en `Conjunto::Nodo` (los puntos suspensivos corresponden a posibles declaraciones de métodos):

```
class Conjunto {
   struct Nodo {
      int dato;
      Nodo * siguiente;
      ...
   };
   Nodo * primero;
   ...
 public:
   ...
};
```

Los datos se guardan de modo tal que el coste temporal en el peor caso de la operación `consultarMinimo` es $O(1)$, el de `eliminarMinimo` es $O(1)$, el de `insertar` es $O(n)$, el de `eliminar` es $O(n)$ y el de `buscar` es $O(n)$, siendo n la talla del conjunto. No se guardan datos repetidos. Piensa cómo conseguir que esas operaciones tengan esos costes y tenlo en cuenta para resolver lo que se pide a continuación.

Añade a la clase `Conjunto` un método:

```
Conjunto unir(const Conjunto & c2) const
```

de modo tal que `c1.unir(c2)` devuelva un nuevo conjunto que contenga tanto los elementos de `c1` como los elementos de `c2`, sin repeticiones. Los conjuntos `c1` y `c2` no deben modificarse. Como casos particulares, tanto `c1` como `c2` como el resultado pueden ser conjuntos vacíos.

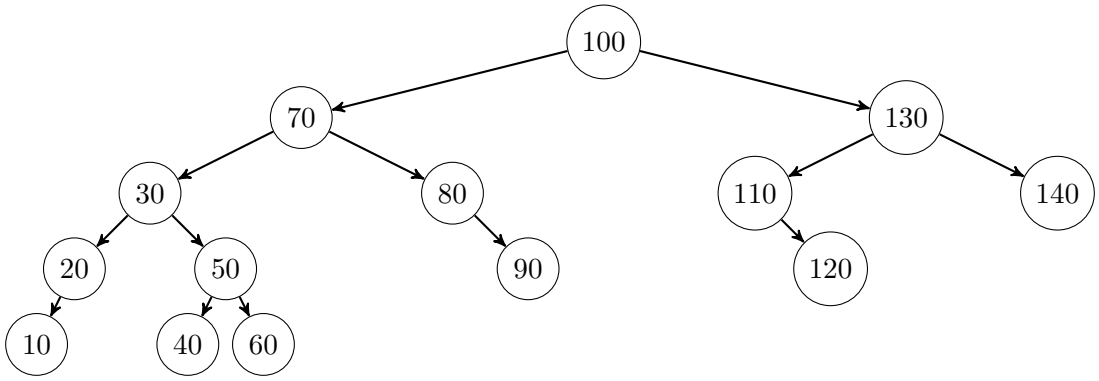
No es necesario que implementes los constructores de `Conjunto` y de `Conjunto::Nodo` pero, si haces uso de otras funciones, debes implementarlas.

Para que tu solución sea válida, debe ser eficiente. Indica cuál es el siguiente coste de tu solución en función de a y b, siendo a la talla del primer conjunto y b la talla del segundo. No es necesario que lo justifiques.

Coste temporal en el peor caso	$O($ $)$

Ejercicio 2 — 0,5 puntos

Aplica el algoritmo de inserción de los árboles AVL para insertar el dato 65 en el siguiente árbol. Indica qué rotaciones se realizan (diciendo si son a derecha o a izquierda y en qué nodo) y dibuja cómo queda el árbol tras cada rotación simple. Si se realiza alguna rotación doble, dibuja por separado el resultado de cada una de las dos rotaciones simples de que consta.



Ejercicio 3 — 2 puntos

Estamos utilizando un árbol binario de búsqueda (no AVL) para realizar una implementación del Tipo Abstracto de Datos **Conjunto**, con datos de tipo real, que permita realizar las siguientes operaciones: *insertar*, *eliminar*, *buscar*, *consultarMínimo* y *eliminarMínimo*. No se guardan datos repetidos. Tenemos los siguientes atributos, y no puedes añadir otros atributos en `Conjunto` ni en `Conjunto::Nodo` (los puntos suspensivos corresponden a posibles declaraciones de métodos):

```
class Conjunto {
   struct Nodo {
      float dato;
      Nodo * izquierdo;
      Nodo * derecho;
      ...
   };
   Nodo * raiz;
   Nodo * minimo;
   ...
 public:
   ...
};
```

El atributo `minimo` sirve para que el coste temporal de la operación *consultarMínimo* en el peor caso sea O(1).

Sin utilizar ningún bucle, utilizando recursión, implementa el método `void eliminarMinimo()` que elimine el mínimo de los elementos del conjunto. Si el conjunto está vacío, no debe cambiar nada. Puedes utilizar otros métodos o funciones solamente si los implementas también sin utilizar ningún bucle.

Indica cuáles son los siguientes costes de tu solución en función de n, siendo n la talla del conjunto (es decir, la cantidad de nodos del árbol), y cuáles serían los costes si el árbol fuese AVL y se añadiese lo necesario para mantenerlo balanceado. No es necesario que lo justifiques.

	sin ser AVL	**si fuese AVL**
Coste temporal en el mejor caso	$O($ $)$	$O($ $)$
Coste temporal en el peor caso	$O($ $)$	$O($ $)$

Ejercicio 4 — 0,5 puntos

Analiza el coste temporal en el peor caso del algoritmo de Huffman cuando se aplica a un alfabeto que tiene n caracteres, suponiendo que la cola de prioridad que utiliza el algoritmo se implementa empleando un vector no ordenado. Explica cómo obtienes el resultado. ¿Cómo podríamos mejorar ese resultado? Justifica tu respuesta.

Ejercicio 5 — 0,5 puntos

Explica brevemente qué harías para buscar un dato en un montículo binario de mínimos (*min-heap*). Dí cuál sería el coste temporal de esa operación en el mejor y en el peor caso, y en qué podrían consistir esos casos.

Ejercicio 6 — 0,5 puntos

Teorema Maestro: La solución de la ecuación $T(N) = aT(N/b) + \theta(N^k \log^p N)$, con $a \geq 1$, $b > 1$ y $p \geq 0$, es

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{si } a > b^k \\ O(N^k \log^{p+1} N) & \text{si } a = b^k \\ O(N^k \log^p N) & \text{si } a < b^k \end{cases}$$

El siguiente algoritmo determina si un dato aparece en un vector ordenado. Utilizando el teorema anterior, analiza el coste temporal del algoritmo en el peor caso en función de n, siendo n la talla del vector que se le pasa en la primera llamada. Explica (i) qué ecuación recursiva obtienes para $T(N)$ y por qué, y (ii) a qué solución llegas aplicando el teorema a partir de esa ecuación y cómo.

```
bool buscar(const vector<float> & v, float dato) {
   if (v.size() == 0)
      return false;
   if (v.size() == 1)
      return v[0] == dato;
   int mitad = v.size() / 2;
   if (v[mitad] > dato) {
      vector<float> izquierda(mitad);
      for (int i = 0; i < mitad; i++)
         izquierda[i] = v[i];
      return buscar(izquierda, dato);
   }
   if (v[mitad] < dato) {
      vector<float> derecha(v.size() - mitad - 1);
      for (int i = mitad + 1, j = 0; i < v.size(); i++, j++)
         derecha[j] = v[i];
      return buscar(derecha, dato);
   }
   return true;
}
```