Algoritmos y Estructuras de Datos (VJ1215) - Universitat Jaume I
Evaluación continua - 2023/2024

17 de noviembre de 2023

Nombre:

La duración de esta prueba es de dos horas. Consta de 6 ejercicios que suman 5 puntos (que se pueden sumar a los puntos obtenidos en la segunda parte en el examen final). 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 todo lo que entregues. En los ejercicios 1, 3 y 4 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.

En teoría de conjuntos, la diferencia simétrica de dos conjuntos es una operación cuyo resultado contiene a aquellos elementos que pertenecen a cada uno de los conjuntos iniciales, pero no a ambos a la vez. Añade a la clase `Conjunto` un método `void diferenciaSimetrica(const Conjunto &)`, de modo tal que `c1.diferenciaSimetrica(c2)` deje en `c1` el resultado de obtener la diferencia simétrica de ambos conjuntos. Por tanto, el método debe eliminar de `c1` los datos que estaban a la vez en `c1` y `c2`, y también debe añadir a `c1` los datos de `c2` que no estaban en `c1`. Como casos particulares, tanto `c1` inicialmente como `c2` como `c1` al finalizar pueden ser conjuntos vacíos.

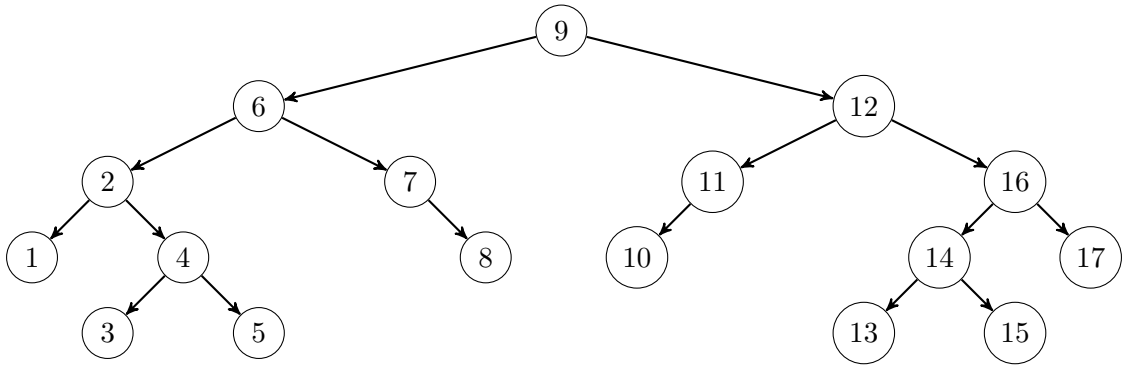
Para que tu solución sea válida, debe ser eficiente. Implementa también los constructores de `Conjunto` y de `Conjunto::Nodo` necesarios para que tu solución funcione correctamente. Si haces uso de otras funciones, debes implementarlas también.

Indica cuál es el coste temporal en el peor caso 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 eliminación de los árboles AVL para eliminar el 9 en el siguiente árbol. Indica qué rotaciones se realizan (diciendo si son a derecha o a izquierda y dónde) 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**. No se guardan datos repetidos. Tenemos los siguientes atributos, que tienen el significado habitual, 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;
   ...
 public:
   ...
};
```

a) **[1 punto]** Sin utilizar ningún bucle, implementa el método `void eliminarHojas()` que elimine todos los nodos que, al empezar a ejecutar el método, no tienen hijos. Si haces uso de otras funciones, debes implementarlas también sin utilizar ningún bucle.

Por ejemplo, si tuviésemos el árbol del ejercicio 2, habría que eliminar los nodos que contienen los valores 1, 3, 5, 8, 10, 13, 15 y 17. Si tuviésemos un árbol vacío, no debería cambiar.

b) **[1 punto]** Implementa de nuevo el método del apartado anterior, esta vez sin utilizar recursión. Si haces uso de otras funciones, debes implementarlas también sin utilizar recursión. Si lo necesitas, solamente en este apartado, puedes hacer uso en C++ de `stack`, `queue` y/o `priority_queue`, sin tener que implementarlas. Si no recuerdas los nombres de sus operaciones básicas, puedes ponerlos en castellano.

Indica cuáles son los siguientes costes de tus dos soluciones en función de n, siendo n la cantidad de nodos del árbol. Justifica solamente a qué se debe ese coste espacial y cuándo se puede dar el peor caso.

	Apartado a	**Apartado b**
Coste temporal en el peor caso	$O($ $)$	$O($ $)$
Coste espacial en el peor caso sin contar el propio árbol	$O($ $)$	$O($ $)$

Ejercicio 4 — 0,5 puntos

Un programador está implementando la siguiente clase `Huffmann` en las prácticas de la asignatura:

```
class Huffman{
   struct Nodo{
      char   caracter;
      float  frecuencia;
      Nodo * izquierdo;
      Nodo * derecho;
      Nodo * padre;
      char   bit;
      Nodo(char, float);
   };
   Nodo * raiz;
   map<char, Nodo *> hojas;
 public:
   Huffman(const vector< pair<char, float> > &);
   string codificar(const string &) const;
   string decodificar(const string &) const;
};
```

Ayúdale, completando lo que le falta en la implementación del método `codificar` que aparece a continuación, para que ese método reciba una cadena de caracteres a codificar y devuelva, en forma de cadena de ceros y unos, el resultado de codificarla empleando el árbol de Huffman obtenido en el constructor. Puedes suponer que la cadena a codificar no contiene ningún carácter que no estuviese en la tabla de carácteres y frecuencias que recibió el constructor. Escribe tu solución en C++, indicando qué pondrías en los puntos suspensivos dentro del bucle. No puedes modificar nada más ni añadir código en otros sitios.

```
string Huffman::codificar(const string & s) const {
   string resultado;
   for (char c : s) {
      Nodo * n = hojas.at(c);
      ...
   }
   return resultado;
}
```

Ejercicio 5 — 0,5 puntos

En un vector, a partir de la posición 1 inclusive, tenemos un montículo binario de mínimos (*min-heap*).

a) ¿Cómo implementarías una operación que disminuya la prioridad de un elemento en el montículo, recibiendo como argumentos su posición en el vector y su nueva prioridad?

b) Ilustra tu solución aplicándola al siguiente montículo para reducir a 25 la prioridad del elemento que actualmente ocupa, con prioridad 50, la posición 9. Dibuja el árbol representado por el vector y cómo evoluciona paso a paso hasta el estado final.

0	1	2	3	4	5	6	7	8	9	10
	10	30	20	40	60	100	90	80	50	70

c) Indica cuál es el coste temporal en el peor y en el mejor caso de tu solución en función de n, siendo n la talla del montículo. Justifica brevemente tus respuestas diciendo cuándo se pueden dar el peor y el mejor caso.

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

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 encuentra el máximo en un vector de talla $n \geq 1$. Utilizando el Teorema Maestro:

a) **[0,25 puntos]** Analiza su coste temporal en el peor caso en función de n.

b) **[0,25 puntos]** Analiza su coste temporal en el mejor caso en función de n.

En cada apartado, 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.

```
int maximo(const vector<int> & v, int inicio, int fin) {
// Devuelve el maximo valor encontrado entre las posiciones inicio y fin, ambas inclusive
   if (fin - inicio <= 3) {
      int resultado = v[inicio];
      for (int i = inicio + 1; i <= fin; i++)
         if (v[i] > resultado)
            resultado = v[i];
      return resultado;
   } else {
      int medio = (inicio + fin) / 2;
      int cuartoIzquierda = (inicio + medio) / 2;
      int cuartoDerecha = (medio + 1 + fin) / 2;
      int resultado = maximo(v, inicio, cuartoIzquierda);
      for (int i = cuartoIzquierda + 1; i <= cuartoDerecha; i++)
         if (v[i] > resultado)
            resultado = v[i];
      if (maximo(v, cuartoDerecha + 1, fin) > resultado)
         resultado = maximo(v, cuartoDerecha + 1, fin);
      return resultado;
   }
}

int maximo(const vector<int> & v) {
   return maximo(v, 0, v.size() - 1);
}
```