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

23 de enero 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, 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.

Añade a la clase `Conjunto` un método:

```
Conjunto diferenciaSimetrica(const Conjunto & c2) const
```

de modo tal que `c1.diferenciaSimetrica(c2)` devuelva un nuevo conjunto que contenga tanto los elementos que pertenecen al conjunto `c1` y no pertenecen al conjunto `c2`, como los elementos que pertenecen a `c2` y no pertenecen a `c1`. 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

En un árbol AVL insertamos, en este orden, los datos 1, 2, 6, 4, 5 y 3. La primera inserción se realiza en un árbol vacío y las demás se realizan en el resultado de la inserción anterior. Dibuja el árbol tras cada inserción.

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, 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;
      int equipo;
      Nodo * izquierdo;
      Nodo * derecho;
      ...
   };
   Nodo * raiz;
   ...
 public:
   ...
};
```

El atributo `equipo` es de utilidad en el videojuego que estamos desarrollando. Los demás atributos tienen el significado habitual, siendo el atributo `dato` (no el atributo `equipo`) el que determina dónde se sitúan los nodos.

a) **[1 punto]** Sin utilizar ningún bucle, implementa un método:

```
bool mismoEquipo(int p) const
```

que devuelva *true* si, y solo si, todos los nodos que están a profundidad menor o igual que p tienen en el atributo `equipo` el mismo valor que tiene el nodo raíz del árbol.

Supón que el árbol no está vacío y que $p \geq 0$. La raíz está a profundidad 0. Si existen, los hijos de la raíz están a profundidad 1, los nietos de la raíz están a profundidad 2, etcétera.

Si haces uso de otras funciones, debes implementarlas también sin utilizar ningún bucle.

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 p (no de la talla del árbol). No es necesario que lo justifiques.

	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

Considera la siguiente clase `Huffmann`, que has implementado 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;
};
```

Añade a esa clase los siguientes dos métodos:

```
string codificarCaracter(char c) const
char decodificarCaracter(const string & s) const
```

El primero debe devolver, en forma de cadena de ceros y unos, el resultado de codificar un solo carácter empleando el árbol de Huffman obtenido en el constructor. El segundo debe servir para recuperar el carácter original a partir de dicha cadena. Puedes suponer que el carácter a codificar estaba en la tabla de caracteres y frecuencias que recibió el constructor. No puedes utilizar los otros métodos ni añadir código en otros sitios.

Ejercicio 5 — 0,5 puntos

El siguiente vector representa un montículo binario de mínimos (*min-heap*), cuya raíz se encuentra en la posición 1 del vector (la posición 0 no se utiliza).

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

Supón que, a partir de ese estado, se realiza una eliminación del mínimo, seguida de una inserción del dato 25. Dibuja el árbol representado por el vector inicialmente y, en el orden en que sucede al realizar esas operaciones, el árbol resultante cada vez que se hace un intercambio de datos.

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 $n \geq 1$. Analiza su coste temporal en el mejor caso en función de n utilizando el teorema anterior, y 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);

      if (maximo(v, cuartoIzquierda + 1, medio) > resultado)
         resultado = maximo(v, cuartoIzquierda + 1, medio);

      if (maximo(v, medio + 1, cuartoDerecha) > resultado)
         resultado = maximo(v, medio + 1, cuartoDerecha);

      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);
}
```