Algoritmos y Estructuras de Datos (VJ1215) - Universitat Jaume I
Examen final - 2022/2023 - Primera parte

24 de enero de 2023

Nombre:

Esta prueba 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 2 y 4:

- Escribe tus soluciones empleando el lenguaje C++.
- No puedes utilizar ninguna clase de las bibliotecas de C++ (`vector`, `stack`, `queue`, `priority_queue`, `map`, etc.). Lo que necesites, lo has de implementar.
- Asegúrate de tratar bien todos los casos.

Ejercicio 1 — 0,5 puntos

El siguiente algoritmo recursivo averigua si un dato aparece en un vector ordenado:

```
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;
}
```

Analiza los costes que se piden a continuación en función de n, siendo n la talla del vector que se le pasa en la primera llamada. En los apartados d y e se pide cuáles serían esos costes si el vector se pasase siempre por valor cambiando `const vector<float> & v` por `vector<float> v`, sin cambiar nada más.

Justifica tus respuestas diciendo, para cada coste, qué cálculo realizas para obtener el resultado y por qué. En particular, si interviene algún sumatorio en el cálculo, indica de cuál se trata.

a) Coste temporal en el peor caso	$O($ $)$
b) Coste espacial en el peor caso	$O($ $)$
c) Coste espacial en el peor caso sin contar el coste de los vectores	$O($ $)$
d) Coste temporal en el peor caso pasando v por valor	$O($ $)$
e) Coste espacial en el peor caso pasando v por valor	$O($ $)$

Ejercicio 2 — 1,5 puntos

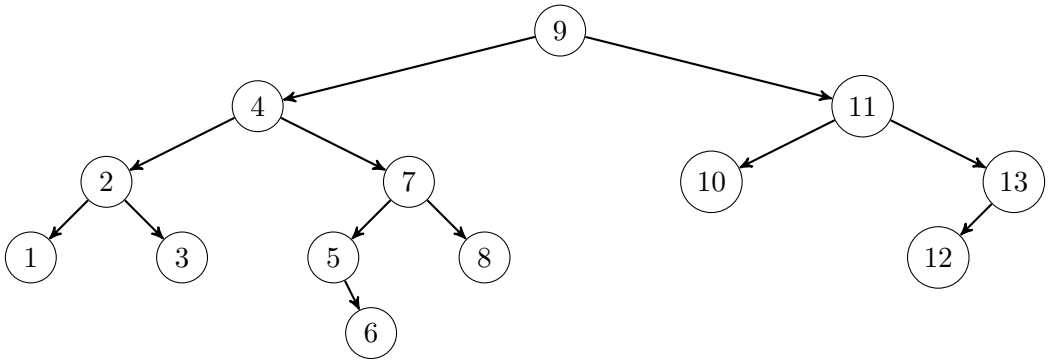
Estamos utilizando una lista simplemente enlazada para realizar una implementación del Tipo Abstracto de Datos **Lista** con datos de tipo real. Tenemos los siguientes atributos, y no puedes añadir otros atributos en `Lista` ni en `Lista::Nodo` (los puntos suspensivos corresponden a posibles declaraciones de métodos):

```
class Lista {
   struct Nodo {
      float dato;
      Nodo * siguiente;
      ...
   };
   Nodo * primero;
   int talla;
   ...
 public:
   ...
};
```

Utilizando recursión, sin utilizar ningún bucle, implementa el método `void insertar(float dato, int i)` que inserta el dato en la posición i-ésima de la lista, sabiendo que $0 \leq i \leq n$, siendo n la talla de la lista antes de la inserción (si i vale 0, el dato se situará en la primera posición; si i vale n, el dato se situará en la última posición). Implementa también los constructores de `Lista` y de `Lista::Nodo` necesarios para que tu solución funcione correctamente. Puedes utilizar otros métodos o funciones solamente si los implementas también sin utilizar ningún bucle.

Ejercicio 3 — 0,5 puntos

Aplica el algoritmo de eliminación de los árboles AVL para eliminar el 10 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 4 — 1,5 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*, *consultarMáximo* y *eliminarMáximo*. 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 * maximo;
   ...
 public:
   ...
};
```

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

Sin utilizar recursión, implementa el método `void eliminarMaximo()` que elimine el máximo 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 recursión.

Indica en esta hoja 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 5 — 1 punto

En la siguiente página se proporciona una implementación del algoritmo que, dados $N \geq 2$ puntos 2D, calcula la distancia entre los dos puntos más próximos empleando Divide y Vencerás. La implementación obtiene el resultado correcto pero su coste temporal no es el de una buena implementación del algoritmo.

a) **[0,5 puntos]** Utilizando el Teorema Maestro, analiza el coste temporal en el peor caso de esa implementación, en función de la cantidad de puntos N. 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.

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

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

b) **[0,5 puntos]** Escribe en C++ los cambios que harías en esa implementación para mejorar ese coste temporal, indicando claramente su posición (puedes responder sobre el propio código o hacer referencia a los números de línea). No es necesario que demuestras cuál es el coste temporal de tu solución.

```
typedef pair<float, float> Punto;
float distanciaAlCuadrado(const Punto & p1, const Punto & p2) {
   float a = p2.first  - p1.first;
   float b = p2.second - p1.second;
   return a * a + b * b;
}
bool compararY(const Punto & p1, const Punto & p2) {
   return p1.second < p2.second;
}
bool buscar(const vector<Punto> & v, const Punto & p) {
   for (int i = 0; i < v.size(); i++)
      if (v[i] == p)
         return true;
   return false;
}
float distanciaAlCuadradoMinima(const vector<Punto> & puntosOrdenX, const vector<Punto> & puntosOrdenY) {
   int talla = puntosOrdenX.size();
   if (talla == 2)
      return distanciaAlCuadrado(puntosOrdenX[0], puntosOrdenX[1]);
   if (talla == 3)
      return min({distanciaAlCuadrado(puntosOrdenX[0], puntosOrdenX[1]),
                  distanciaAlCuadrado(puntosOrdenX[0], puntosOrdenX[2]),
                  distanciaAlCuadrado(puntosOrdenX[1], puntosOrdenX[2])});
   int tallaIzquierda = talla / 2, tallaDerecha = talla - tallaIzquierda;
   vector<Punto> izquierdaX(tallaIzquierda), derechaX(tallaDerecha),
                 izquierdaY(tallaIzquierda), derechaY(tallaDerecha);
   for (int i = 0; i < tallaIzquierda; i++)
      izquierdaX[i] = puntosOrdenX[i];
   for (int i = 0; i < tallaDerecha; i++)
      derechaX[i] = puntosOrdenX[i + tallaIzquierda];
   for (int i = 0, j = 0, k = 0; i < talla; i++)
      if (buscar(izquierdaX, puntosOrdenY[i]))
         izquierdaY[j++] = puntosOrdenY[i];
      else
         derechaY[k++] = puntosOrdenY[i];
   float minima = min(distanciaAlCuadradoMinima(izquierdaX, izquierdaY),
                      distanciaAlCuadradoMinima(derechaX, derechaY));
   vector<Punto> centro;
   float frontera = izquierdaX[tallaIzquierda - 1].first;
   for (int i = 0; i < talla; i++)
      if( abs(frontera - puntosOrdenY[i].first)  < minima )
         centro.push_back(puntosOrdenY[i]);
   for (int i = 0; i < centro.size() - 1; i++)
      for (int j = i + 1; j < centro.size()
               && centro[j].second - centro[i].second < minima; j++)
         minima = min(minima, distanciaAlCuadrado(centro[i], centro[j]));
   return minima;
}
float distanciaMinima(const vector<Punto> & puntos) {
   vector<Punto> puntosOrdenX(puntos), puntosOrdenY(puntos);
   sort(puntosOrdenX.begin(), puntosOrdenX.end());
   for (int i = 0; i < puntosOrdenX.size() - 1; i++)
      if (puntosOrdenX[i] == puntosOrdenX[i + 1])
         return 0;
   sort(puntosOrdenY.begin(), puntosOrdenY.end(), compararY);
   return sqrt(distanciaAlCuadradoMinima(puntosOrdenX, puntosOrdenY));
}
```