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

8 de junio 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 3, escribe tus soluciones empleando el lenguaje C++ y asegúrate de tratar bien todos los casos.

Ejercicio 1 — 0,5 puntos

El siguiente algoritmo recursivo busca un dato en un vector. Si lo encuentra, devuelve la posición en la que lo ha encontrado. Si no lo encuentra, devuelve -1.

```
1  int buscarPosicion(const vector<float> & v, float dato, int inicio, int fin) {
2     // Busca el dato entre las posiciones inicio y fin de v, ambas inclusive
3     if (inicio == fin)
4        if (v[inicio] == dato)
5           return inicio;
6        else
7           return -1;
8
9     vector<float> copia(v.size());
10    for (int i = 0; i < v.size(); i++)
11       copia[i] = v[i];
12
13    int medio = (inicio + fin) / 2;
14    int resultadoIzquierda = buscarPosicion(copia, dato, inicio, medio);
15
16    if (resultadoIzquierda != -1)
17       return resultadoIzquierda;
18    else
19       return buscarPosicion(copia, dato, medio + 1, fin);
20 }
21
22 int buscarPosicion(const vector<float> & v, float dato) {
23    if (v.size() == 0)
24       return -1;
25    return buscarPosicion(v, dato, 0, v.size() - 1);
26 }
```

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 el apartado e se pide cuál sería el coste temporal en el peor caso si el vector se pasase siempre por valor cambiando en ambas funciones `const vector<float> & v` por `vector<float> v`, sin cambiar nada más.

Justifica tus respuestas explicando (i) en qué consisten el peor caso y el mejor caso en este algoritmo y (ii) qué cálculos realizas para obtener el resultado. En particular, si interviene algún sumatorio en los cálculos, indica de qué sumatorio se trata y por qué.

a) Coste temporal en el peor caso	$O($ $)$
b) Coste temporal en el mejor caso	$O($ $)$
c) Coste espacial en el peor caso	$O($ $)$
d) Coste espacial en el mejor caso	$O($ $)$
e) Coste temporal 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 **Conjunto** con datos de tipo real. 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 * 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. Teniendo en cuenta eso, añade a la clase `Conjunto` un método `void eliminarComunes(const Conjunto &)`, de modo tal que `c1.eliminarComunes(c2)` elimine de `c1` todos los datos que aparezcan también en `c2`. No se debe modificar `c2`. En el caso particular de que no haya ningún dato que esté en ambos conjuntos, no se debe modificar `c1`. Para que tu solución sea válida, debe ser eficiente.

Puedes utilizar otros métodos o funciones solamente si los implementas también. No puedes utilizar ninguna clase de las bibliotecas de C++ (`vector`, `queue`, etc.).

Indica en esta hoja 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 3 — 1,5 puntos

Estamos utilizando un árbol binario de búsqueda (no AVL) para realizar una implementación del Tipo Abstracto de Datos **Conjunto**. Tenemos los siguientes atributos (los puntos suspensivos corresponden a posibles declaraciones de métodos) y no puedes añadir otros atributos en `Conjunto` ni en `Conjunto::Nodo`:

```
class Conjunto {
   struct Nodo {
      int dato;
      Nodo * izquierdo;
      Nodo * derecho;
      int tallaIzquierdo;
      int tallaDerecho;
      ...
   };
   Nodo * raiz;
   ...
 public:
   ...
};
```

En cada nodo, el valor del atributo `tallaIzquierdo` es la cantidad de nodos de su subárbol izquierdo y el valor del atributo `tallaDerecho` es la cantidad de nodos de su subárbol derecho. Debes hacer lo necesario para mantener también esos atributos actualizados.

Sin utilizar recursión, implementa el método `void insertar(int dato)` que inserta el dato en el conjunto si no estaba ya en él. Si el dato ya estaba en el conjunto, no debe cambiar nada. Si haces uso de otros métodos, debes implementarlos también sin utilizar recursión. Implementa también los constructores de `Conjunto` y de `Conjunto::Nodo` necesarios para que tu solución funcione correctamente.

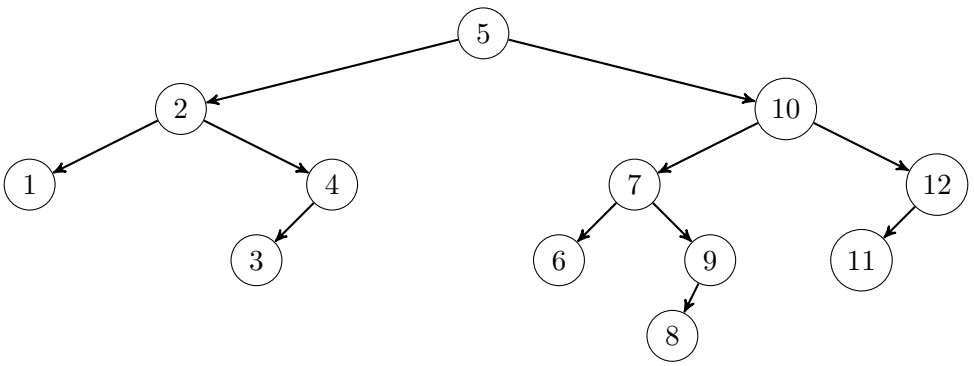
Si lo necesitas, puedes hacer uso en C++ de `stack`, `queue` y/o `priority_queue`, sin tener que implementarlas. Si no recuerdas los nombres de las operaciones básicas, puedes ponerlos en castellano.

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. Justifica tus respuestas diciendo cuando se pueden dar el peor y el mejor caso.

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

Ejercicio 4 — 0,5 puntos

Aplica el algoritmo de eliminación de los árboles AVL para eliminar el 1 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.



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

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

En los apartados a, b y c utiliza este teorema para analizar el coste temporal que se pide 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 y cómo.

a) **[0,25 puntos]** Analiza el coste temporal en el peor caso del siguiente algoritmo para encontrar el máximo en un vector de talla $n \geq 1$.

```
int maximo(const vector<int> & v, int fin) {
   if (fin == 0)
      return v[0];
   int medio = fin / 2;
   int aux = maximo(v, medio);
   for (int i = medio + 1; i <= fin; i++)
      aux = max(aux, v[i]);
   return aux;
}
int maximo(const vector<int> & v) {
   return maximo(v, v.size() - 1);
}
```

b) **[0,25 puntos]** Analiza el coste temporal en el peor caso del siguiente algoritmo para encontrar el máximo en un vector de talla $n \geq 1$.

```
int maximo(const vector<int> & v, int fin) {
   if (fin == 0)
      return v[0];
   int medio = fin / 2;
   int aux = v[medio + 1];
   for (int i = medio + 2; i <= fin; i++)
      aux = max(aux, v[i]);
   if (aux >= maximo(v, medio))
      return aux;
   return maximo(v, medio);
}
int maximo(const vector<int> & v) {
   return maximo(v, v.size() - 1);
}
```

c) **[0,25 puntos]** 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.

Analiza el coste temporal en el peor caso de esa implementación, en función de la cantidad de puntos n.

d) **[0,25 puntos]** Escribe en C++ los cambios que harías en la implementación del apartado c para mejorar su coste temporal, indicando claramente su posición (puedes responder sobre el propio código o hacer referencia a los números de línea). No es necesario que demuestres 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;
}
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 (puntosOrdenY[i] < derechaX[0])
         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(); 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));
}
```