EXAMEN-MAYO-2021-SOL.pdf

Taurux

Programación II

1º Grado en Ingeniería Informática

Facultad de Informática
Universidad de A Coruña

EJERCICIO 1 (1,5 PUNTOS)
Dados los siguientes supuestos prácticos decidir: (1) cuál es la MEJOR estructura de datos, y (2) su MEJOR implementación para resolver el problema (para ser considerada correcta, AMBAS respuestas deberán estar justificadas):

1. En el Servicio de Salud están recibiendo continuamente nuevas dosis de vacunas. Para evitar que caduquen es imprescindible que se usen primero las vacunas que han sido recibidas hace más tiempo. ¿Qué estructura de datos deberemos utilizar para gestionar el almacenamiento de las dosis?

   La mejor implementación posible es una cola, pues posee una estructura FIFO (first in, first out). Dicha cola debería de ser dinámica ya que, como se reciben nuevas dosis continuamente, no sabemos el número máximo de dosis que tendremos de forma simultánea.

2. El Servicio de Salud desea enviar una carta al domicilio y un SMS al móvil de cada paciente a los que se va a administrar la vacuna para recordarle el lugar y la hora en que debe presentarse ¿Qué estructura de datos deberemos utilizar para que podamos recuperar rápidamente esta información a partir del número de la tarjeta sanitaria?

   Ya que necesitaremos hacer muchas búsquedas, la mejor opción sería utilizar un árbol binario de búsqueda, pues dicha operación es muy poco costosa en este tipo de árbol. También sería correcto hacer uso de un árbol equilibrado.

3. Para garantizar la distancia interpersonal durante los exámenes, la Facultad ha asignado varias aulas al examen de cada asignatura. Cada aula tiene una capacidad máxima y los estudiantes elegirán en una web el aula en la que van a hacer el examen (de entre aquellas en las que queden sitios libres). Antes de un examen queremos obtener el listado de los estudiantes, ordenado alfabéticamente por nombre, que se han asignado a cada aula. ¿Qué estructura de datos deberemos utilizar para gestionar este listado?

   Deberíamos de utilizar una lista ordenada (pues se indica que el listado va a estar ordenado alfabéticamente), dicha lista debería de ser estática, pues a priori ya conocemos el número máximo de alumnos que pueden estar almacenados en la lista.

EJERCICIO 2 (1 PUNTO)
Contestar Verdadero o Falso y explicar el porqué a las siguientes preguntas (para ser considerada correcta, la respuesta deberá estar justificada):

1. En el TAD Cola los elementos se organizan de forma circular.

   Falso. Si bien es cierto que en un TAD cola se pueden almacenar de forma circular (véase el caso del uso de un array circular para la implementación estática), no siempre se da dicha situación, por ejemplo podríamos implementar una cola dinámica con un puntero al primer nodo y otro al último.

2. En una implementación dinámica de las listas es imposible tener acceso eficiente al último elemento de la lista.

   Falso. Normalmente en una implementación dinámica de las listas habría que recorrer toda la lista para acceder al último elemento, lo cual no sería eficiente, sin embargo esto no es así en el caso de una lista dinámica doblemente enlazada, ya que cuenta con un puntero que apunta al último elemento de la lista.

3. En una pila los elementos se extraen en orden inverso al de entrada.

   Verdadero. Una pila tiene una estructura LIFO (last in, first out), es decir, el último elemento en entrar a la pila será el último en salir.

4. Los árboles completos son árboles binarios de búsqueda equilibrados.

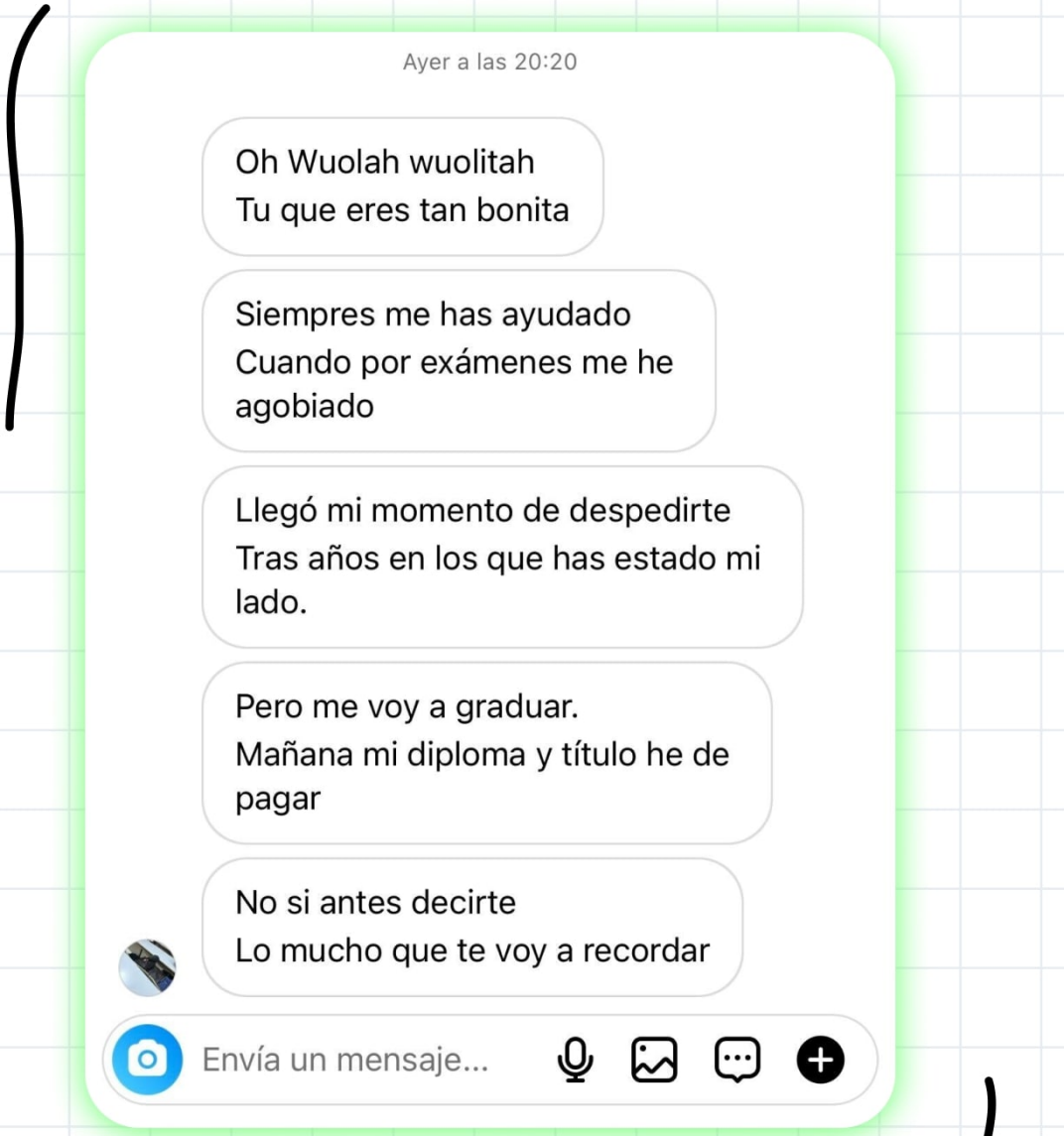
   Verdadero. Un árbol completo cumple las condiciones de un árbol equilibrado. Es verdad que no todos los árboles equilibrados son completos, pero todos los completos son equilibrados.

EJERCICIO 3 (0,75 PUNTOS)
En el siguiente código hay tres errores. Identifica cada uno de ellos, indica por qué se trata de un error y propón el correspondiente código corregido (basta con reescribir la parte del código afectada).

```
01  #include
02  #include
03  typedef struct tNode* tPosL;
04  struct tNode{
        int datum;
        tPosL next;
    };
05  void corrects (int max, tPosL L){
06      struct tNode i;
07          for(i=L; i!=NULL; i=i->next) {
08              if (i->datum > max)
09                  i->datum = max;
10              printf("%d ", i->datum);
11          }
12  }
13  int main(){
14      tPosL q, L;
15          L = malloc(sizeof(struct tNode));
16          printf("Type first value: "); scanf("%d",&L->datum);
17          q = malloc(sizeof(struct tNode));
18          q = L;
19          for(int i=2; i<=10; i++){
20              q->next = malloc(sizeof(struct tNode));
```

Que no te escriban poemas de amor cuando terminen la carrera 🚩🚩🚩🚩🚩🚩

(a nosotros por suerte nos pasa)



WUOLAH

```
21                          q = q->next;
22                          printf("Type next value: "); scanf("%d", &q->datum);
23                  }
24                  q->next = NULL;
25                  corrects(5,&L);
26      }
```

1. En la función hay que pasar por referencia o por valor, pero la forma en la que se muestra es incorrecta, ya que no completa ninguna de las dos.
2. La línea 6 debería ser tPosL i, no struct tNode.
3. La línea 17 sobra por completo.

EJERCICIO 4 (1,25 PUNTOS)

El tipo abstracto de datos (TAD) tBinTree sirve para representar árboles binarios de búsqueda de enteros positivos y para manipularlo solamente disponemos de la siguiente interfaz:

```
#define TNULL …
typedef unsigned int tItemT;
typedef … tBinTree;

void createEmptyTree(tBinTree *T);
bool buildTree(tBinTree LTree, tItemT d, tBinTree Rtree, tBinTree *T);
tBinTree leftChild(tBinTree T);
tBinTree rightChild(tBinTree T);
tItemT root(tBinTree T);
bool isEmptyTree(tBinTree T);
```

NOTA: Precondición común a leftChild, rightChild y root: árbol no vacío.

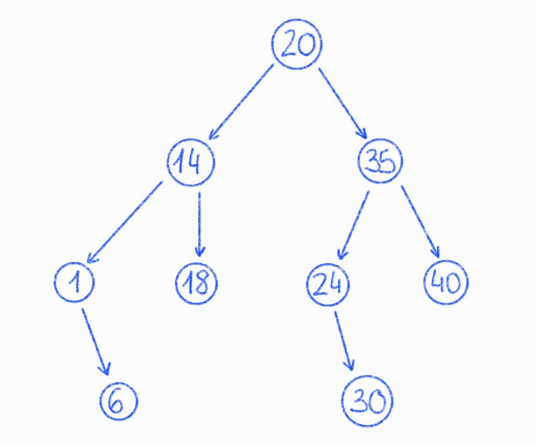
A) (0,75 PUNTOS)

1) Determinar qué OPERACIÓN hace la función WhatItDoes.

```
int WhatItDoes(tBinTree A){
        if (isEmptyTree(A))
                return 0;
        else if (root(A)%2!= 0)
                        return(WhatItDoes (leftChild (A)) + WhatItDoes (rightChild
                (A)));
                else
                return(root(A) + WhatItDoes (leftChild (A)) + WhatItDoes (rightChild
                (A)));
```

La función WhatItDoes suma todas los números no impares almacenados en un árbol.

2) Aplícala al árbol de la figura y obtén su resultado.



Aplicada al árbol de la figura, la función WhatItDoes devolverá 152 tras ser ejecutada.

B) (0,5 PUNTOS)

1) Identificar qué tipo de recorrido nos permitiría mostrar las claves del árbol de la figura en este orden: 20-14-1-6-18-35-24-30-40.

   El tipo de recorrido que permitiría mostrar las claves del árbol de la figura en el orden especificado es Preorden.

2) Aplicando el siguiente pseudocódigo al mismo árbol, ¿cuál sería la secuencia de claves?

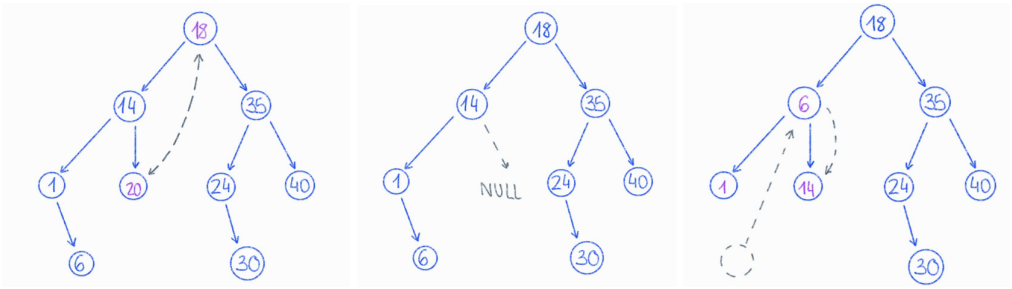
```
Procedimiento recorrido (tBinTree)
Inicio
    si no es árbol vacío (tBinTree) entonces
        recorrido (hijo izquierdo (tBinTree))
        recorrido (hijo derecho (tBinTree))
        imprime (raíz (tBinTree))
Fin
```

   El pseudocódigo mostrado representa un recorrido tipo Posorden, por lo que la secuencia resultante será la siguiente: 6-1-18-14-30-24-40-35-20.

3) Suponiendo que el árbol de la figura es AVL, elimina la clave 20, y muestra el árbol resultado identificando cualquier transformación necesaria y explicándola paso a paso.

   1. En primer lugar, como vamos a eliminar un nodo con 2 hijos, lo intercambiamos con su antecesor.
   2. A continuación eliminamos el nodo y hacemos que el puntero apunte a NULL, ya que el nodo eliminado no tiene hijos.

3. Por último, para mantener el árbol equilibrado se realiza una rotación de tipo LR en los nodos 14, 1 y 6.



EJERCICIO 5 (5,5 PUNTOS)

A)

```
// Tipos de datos
#define Q_MAX 100
#define MAX_PAR_LENGTH 32
#define NULLQ -1

typedef char tCode;

typedef struct ItemQ{
   tCode code;
   char parameter1[MAX_PAR_LENGTH];
   char parameter2[MAX_PAR_LENGTH];
   char parameter3[MAX_PAR_LENGTH];
}tItemQ;

typedef int tPosQ;

typedef struct Queue{
   tItemQ data[Q_MAX];
   tPosQ front;
   tPosQ rear;
}tQueue;

// Funciones
int addOne(int i){
   if (i == Q_MAX - 1)
      return 0;
   else
      return ++i;
}

bool isEmptyQueue(tQueue Q){
   return (Q.front == addOne(addOne(Q.rear)) );
}
```

```
bool enqueue(tItemQ d, tQueue *Q){
    if (Q->rear == addOne(addOne(Q->rear)) )
        return false;
    else{
        Q->rear == addOne(Q->rear);
        Q->data[Q->rear] = d;
        return true;
    }
}

void dequeue(tQueue *Q){
    Q->font = addOne(Q->front);
}
```