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?
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?
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?

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.
2. En una implementación dinámica de las listas es imposible tener acceso eficiente al último elemento de la lista.
3. En una pila los elementos se extraen en orden inverso al de entrada.
4. Los árboles completos son árboles binarios de búsqueda 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 <stdio.h>
02  #include <stdlib.h>

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

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:

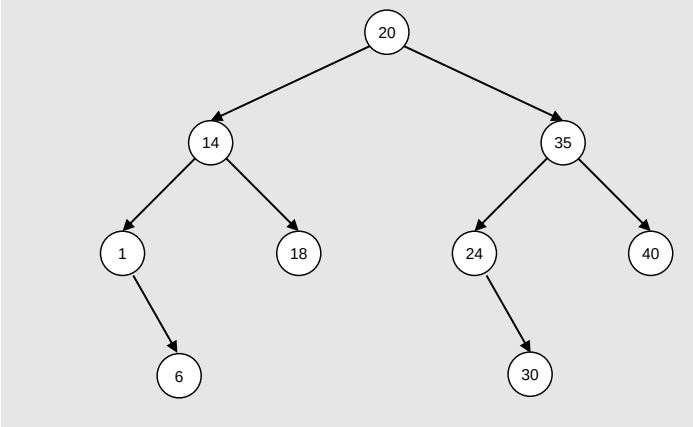
```
#define TNULL …
typedef unsigned int tItemT;
typedef … tBinTree;

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)

Se pide: 1) determinar qué OPERACIÓN hace la función WhatItDoes; 2) Aplícala al árbol de la figura y obtén su resultado.

```
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)));
}
```



B) (0,5 PUNTOS)

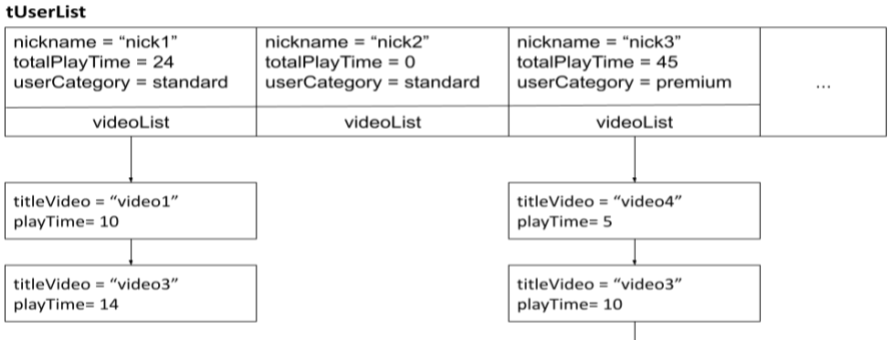
- Identificar qué tipo de recorrido nos permitiría mostrar las claves del árbol de la figura en este orden: 20-14-1-6-18-35-24-30-40.
- Aplicando el siguiente seudocódigo al mismo árbol, ¿cuál sería la secuencia de claves?

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

- 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.

EJERCICIO 5 (5,5 PUNTOS)

En la práctica 2 se implementó un sistema para gestionar la plataforma VIMFIC de vídeo bajo demanda. El sistema emplea dos estructuras de datos (UserList y VideoList) para almacenar conjuntamente toda la información asociada, respectivamente, a usuarios y reproducciones. tal y como se representa en la siguiente figura:



Partiendo del mismo problema, en este ejercicio se proponen una serie de modificaciones.

A) (3 PUNTOS)

Cuando se ponga el sistema en explotación en un entorno real, leer las peticiones de los usuarios desde un fichero dejará de ser viable. En su lugar, se utilizará un TAD Cola (RequestQueue) para almacenar y gestionar dichas peticiones. En concreto, se ha decidido emplear una implementación de **cola estática circular de 100 posiciones**.

Se pide, en primer lugar, realizar la nueva definición de tipos de datos de la cola:

`MAX_PAR_LENGTH`	constante para representar la máxima longitud (32) de las cadenas de caracteres de los parámetros de las peticiones
`tQueue`	representa una cola de peticiones
`tCode`	tipo de la petición recibida (`char`)
`tItemQ`	datos de un elemento de la cola (una petición), compuesto por: `code`: de tipo `tCode`, representa la clase de la petición `parameter1`: de tipo cadena de caracteres `parameter2`: de tipo cadena de caracteres `parameter3`: de tipo cadena de caracteres
`tPosQ`	posición de un elemento de la cola
`NULLQ`	constante para representar posiciones nulas en la cola

Además, partiendo de la nueva definición de tipos de datos, se pide realizar la implementación de las siguientes operaciones:

- **`isEmptyQueue (tQueue) -> bool`**
  Determina si la cola está vacía.
- **`enqueue (tItemQ, tQueue) -> tQueue, bool`**
  Inserta un nuevo elemento (tItemQ) en la cola. Devuelve falso si no hay memoria suficiente para realizar la operación.
- **`dequeue (tQueue) -> tQueue`**
  Elimina el elemento que está en el frente de la cola. PreCD: la cola no está vacía.

B) (2,5 PUNTOS)

Los propietarios de la plataforma VIMFIC quieren lanzar una promoción para fidelizar a los mejores clientes de categoría standard. Para ello, se pondrá a valor 0 el contador totalPlayTime de aquellos que cumplan dos condiciones:

1. Que tengan entre 0 y 30 minutos menos de *totalPlayTime* que el cliente *standard* que más tiempo de reproducción tenga.

2. Que, además, hayan visualizado más de 10 vídeos

Se pide implementar la operación

applyPromo (tUserList) —> tUserList

que lleve a cabo el proceso descrito. El manejo de todos los TADs implicados se hará utilizando **EXCLUSIVAMENTE** los tipos y las operaciones de las interfaces mostrados en los ANEXOS.

ANEXO I

Tipos de datos	
`tUserList`	Representa una lista de usuarios **ordenada por sus nicks**
`tUserCategory`	Categoría de usuario (tipo enumerado: {`standard`, `premium`})

`tUserItem`	Datos de un elemento de la lista (un usuario). Compuesto por:
	`nickname`: de tipo `string`
	`totalPlayTime`: de tipo `integer`
	`userCategory`: de tipo `tUserCategory`
	`videoList`: de tipo `tVideoList`

`tUserPos`	Posición de un elemento de la lista de usuarios
`NULL_USER`	Constante usada para indicar posiciones nulas de la lista

Operaciones (Precondición común (salvo createEmptyList): la lista debe estar inicializada)

- **`createEmptyList (tUserList) → tUserList`**
  Crea una lista vacía. PostCD: La lista queda inicializada y no contiene elementos.
- **`isEmptyList (tUserList) → bool`**
  Determina si la lista está vacía.
- **`first (tUserList) → tUserPos`**
  Devuelve la posición del primer elemento de la lista. PreCD: La lista no está vacía.
- **`last (tUserList) → tUserPos`**
  Devuelve la posición del último elemento de la lista. PreCD: La lista no está vacía.
- **`next (tUserPos, tUserList) → tUserPos`**
  Devuelve la posición en la lista del elemento siguiente al de la posición indicada (o NULL_USER si no tiene siguiente). PreCD: La posición es válida.
- **`previous (tUserPos, tUserList) → tUserPos`**
  Devuelve la posición en la lista del elemento anterior al de la posición indicada (o NULL_USER si no tiene anterior). PreCD: La posición es válida..
- **`getItem (tUserPos, tUserList) → tUserItem`**

Devuelve el contenido del elemento de la lista de la posición indicada. PreCD: La posición es válida.

- **`updateItem (tUserItem, tUserPos, tUserList) → tUserList`**
  Modifica el contenido del elemento de la posición indicada. PreCD: La posición es válida.

ANEXO II

Tipos de datos	
`tVideoList`	Representa una lista de vídeos (no ordenada)

`tVideoItem`	Datos de un vídeo. Contendrá los campos:
	`titleVideo` de tipo `string`
	`playTime` de tipo `integer`

`tVideoPos`	Posición de un elemento de la lista de vídeos
`NULL_VIDEO`	Constante usada para indicar posiciones nulas de la lista

Operaciones (Precondición común (salvo createEmptyListV): la lista debe estar inicializada)

- **`createEmptyListV (tVideoList)→ tVideoList`**
  Crea una lista vacía. PostCD: La lista queda inicializada y no contiene elementos.
- **`isEmptyListV (tVideoList) → bool`**
  Determina si la lista está vacía.
- **`firstV (tVideoList) → tVideoPos`**
  Devuelve la posición del primer elemento de la lista. PreCD: La lista no está vacía.
- **`lastV (tVideoList) → tVideoPos`**
  Devuelve la posición del último elemento de la lista. PreCD: La lista no está vacía.
- **`nextV (tVideoPos, tVideoLiist)→ tVideoPos`**
  Devuelve la posición en la lista del elemento siguiente al de la posición indicada (o NULL_ VIDEO si no tiene siguiente). PreCD: La posición es válida.
- **`previousV (tVideoPos, tVideoList) → tVideoPos`**
  Devuelve la posición en la lista del elemento anterior al de la posición indicada (o NULL_ VIDEO si no tiene anterior). PreCD: La posición es válida.
- **`getItemV (tVideoPos, tVideoList) → tVideoItem`**
  Devuelve el contenido del elemento de la lista de la posición indicada. PreCD: La posición es válida.
- **`updateItemV (tVideoItem, tVideoPos, tVideoList) → tVideoList`**
  Modifica el contenido del elemento de la posición indicada. PreCD: La posición es válida.