Nombre y Apellido: ..... N° Legajo: .....

Segundo Parcial de Programación Imperativa

11/11/2022

- **<u>Condición mínima de aprobación: Sumar 5 (cinco) puntos</u>**
- **Se tendrá en cuenta en la calificación el ESTILO y la EFICIENCIA de los algoritmos.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, <u>no serán aceptados</u>.**
- **En caso de necesitar usar malloc o similar, no validar que retorne distinto de NULL.**
- **No es necesario escribir los #include.**
- **No se responderán preguntas durante el parcial.**

Ejercicio 1 (1,5 puntos)

Dada una **lista de enteros** cuya definición de tipos es la siguiente:

```
typedef struct node * TList;

typedef struct node {
      int elem;
      struct node * tail;
} TNode;
```

y donde una lista vacía se representa con el valor `NULL`.

Escribir una función **<u>recursiva</u>** `sortedList` que reciba como único parámetro una lista y **<u>elimine de la misma</u> aquellos elementos que hagan que la lista no esté ordenada en forma ascendente** (cada elemento debe ser menor al siguiente) teniendo en cuenta que el **<u>último elemento</u> está correctamente ubicado** (no se lo debe borrar).

La función no debe crear una nueva lista, sino eliminar elementos de la lista recibida.

<u>No definir funciones ni macros auxiliares</u>
<u>No usar variables static</u>
<u>No usar ciclos dentro de la función</u>

<u>Ejemplos:</u>

- Si recibe la lista vacía, retorna la lista vacía
- Si recibe la lista 1 → 2 → 3 → 4 → 5 no cambia nada
- Si recibe la lista 2 → 3 → 4 → 1 la lista queda únicamente con el elemento 1
- Si recibe la lista 2 → 1 → 4 → 3 la lista queda 1 → 3
- Si recibe la lista 1 → 3 → 3 → 4 → 2 la lista queda 1 → 2
- Si recibe la lista 3 → 3 → 3 la lista queda únicamente con el elemento 3
- Si recibe la lista 3 → 5 → 2 → 4 → 3 → 6 la lista queda 2 → 3 → 6

Ejercicio 2 (1 punto)

Ejercicio 2.1

Indicar la salida estándar del siguiente programa

```c
#include <stdio.h>

// Calcula la longitud en un parametro de salida
void
stringLen(const char *s, int *length)
{
    static init = 0;
    if (init == 0) {
        *length = 0;
         init = 1;
     }

     if ( *s != 0 )
     {
         (*length) ++;
         stringLen(s+1, length);
     }

     return;
}

int
main(void) {
    int len1, len2;
    stringLen("123456", &len1);

    stringLen("123456", &len2);
    printf("long=%d %d \n", len1, len2);

    return 0;
}
```

Elija al menos una respuesta correcta.

- (A) 6 6
- (B) 0 0
- (C) 6 basura
- (D) El programa aborta y no imprime nada
- (E) Ninguna de las opciones anteriores

Ejercicio 2.2

Indicar si en las siguientes líneas se producen errores de **ejecución**

```c
1   #include <stdio.h>
2
3   typedef struct {
4       int a;
5       int b;
6   } Coordenada;
7
8   typedef Coordenada * PCoord;
9
10  typedef struct
11  {
12     char * nombre;
13     Coordenada punto1;
14     PCoord punto2;
15  } TipoX;
16
17  int
18  main(void)
19  {
20        TipoX var1;
21        var1.nombre = "nombre1";
22        var1.punto1.a = 10;
23        var1.punto1.b = 30;
24        var1.punto2->a = 20;
25        var1.punto2->b = 50;
26
27        return 0;
28  }
```

Elija al menos una respuesta correcta.

- (A) 21
- (B) 22
- (C) 23
- (D) 24
- (E) 25
- (F) No hay errores de ejecución

Ejercicio 3 (2,5 puntos)

Se desea implementar un TAD que permita mantener y consultar un **diccionario de sinónimos**.
El TAD permitirá indicar para una palabra uno o más sinónimos.
La relación entre una palabra y sus sinónimos no es simétrica.
No se eliminarán palabras ni sinónimos del TAD.
Se asume que todas las palabras a insertar no serán extensas..

Completar en el archivo **`dicSynADT.c`** que aparece debajo las funciones pedidas. En caso de ser necesario se pueden definir funciones auxiliares.

No se pueden modificar ni el .h ni los structs definidos.

dicSynADT.h

```
typedef struct dicSynCDT * dicSynADT;

/* Retorna un nuevo diccionario de sinónimos. Al inicio está vacío.
** No hay límite de capacidad.
*/
dicSynADT newDicSyn();

/* Agrega una relación entre una palabra y su sinónimo
** Se asegura que tanto word como synonymous no son NULL ni vacíos
** Ver ejemplo de uso
*/
void add(dicSynADT dict, const char * word, const char * synonymous);

/* Retorna cuántas palabras hay en el diccionario. Se cuentan tanto las palabras
** como los sinónimos (los parámetros word y synonymous de la función anterior)
 */
size_t size(const dicSynADT dict);

/* Dada una palabra retorna un vector con una copia de todos sus sinónimos
** ordenados alfabéticamente, con NULL como marca de final
*/
char ** synonyms(dicSynADT dict, const char * word);

/* Retorna un vector con las palabras y sinónimos del diccionario (ver ejemplo)
** El vector está ordenado en forma ascendente y contiene NULL como marca de
** final.
** Si no hay palabras, retorna un vector que sólo contiene NULL
*/
char ** words(const dicSynADT dict);

/* Libera toda la memoria reservada */
void freeDict(dicSynADT dict);
```

dicSynADT.c parcial

```
typedef struct node {
   char * synonym;
   struct node * tail;
} node;

typedef struct wordWithSyns {
   char * word;
   node * syns;
   struct wordWithSyns * tail;
} wordWithSyns;

struct dicSynCDT {
   size_t size;     // cantidad de palabras
   wordWithSyns * first;
};

dicSynADT newDicSyn() {
   // Completar
}

size_t size(const dicSynADT dict) {
   return dict->size:
}
```

```
/*
** Busca el nodo con la palabra word. Si no estaba lo agrega en forma ascendente
** y hace que node apunte al nuevo nodo.
** Si ya estaba entonces node apunta al nodo que contiene word
*/
static wordWithSyns * addWord( wordWithSyns * first, const char * word,
     wordWithSyns ** node, int * flag {
   // NO COMPLETAR
}

void add(dicSynADT dict, const char * word, const char * synonymous) {
   // Primero veo si hay que agregar word. Me retorna en un parámetro
   // de salida una referencia al nodo, y en un flag retorna 0 si estaba o 1
   // si lo agregó
   wordWithSyns * theWord;
   int flag = 0;
   dict->first = addWord(dict->first, word, &theWord, &flag);
   dict->size += flag;

   // En theWord tenemos el puntero al nodo con la palabra word
   // Completar el resto de la función
 dict->first = addWord(dict->first, synonymous, &theWord, &flag);
   dict->size += flag;

}

char ** synonyms(dicSynADT dict, const char * word) {
   // Completar
}

char ** words(const dicSynADT dict)  {
   // Completar
}

void freeDict(dicSynADT dict)  {
   // NO completar
}
```

Programa de testeo

```
#include "synADT.h"

static void freeAll(char ** v) {
  for(size_t i=0; v[i] != NULL; i++) {
     free(v[i]);
  }
  free(v);
}

int
main(void) {
   dicSynADT dic = newDictSyn();
   char ** aux;

   aux = words(dic);
   assert(aux[0] == NULL);
   free(aux);

   aux = synonyms(dic, "casa");
   assert(aux[0] == NULL);
   free(aux);

   char word[50];
```

```
    strcpy(word, "vivienda");
    add(dic, "casa", word);
    assert(size(dic)==2);           // "casa" y "vivienda"

    strcpy(word, "Hogar");          // Puede venir en minúscula o mayúscula
    add(dic, "casa", word);
    assert(size(dic)==3);           // "casa", "Hogar" y "vivienda"

    strcpy(word, "almondiga");
    add(dic, "albondiga", word);
    add(dic, "casa", "hogar");           // No realiza ningún cambio

    aux = synonyms(dic, "Hogar");
    assert(aux[0] == NULL);
    free(aux);

    assert(size(dic)==5);

    aux = synonyms(dic, "CASA");    // "CASA" es lo mismo que "casa"
    assert(strcmp(aux[0], "Hogar")==0);
    assert(strcmp(aux[1], "vivienda")==0);
    assert(aux[2] == NULL);
    freeAll(aux);

    aux = words(dic);
    assert(strcmp(aux[0], "albondiga")==0);
    assert(strcmp(aux[1], "almondiga")==0);
    assert(strcmp(aux[2], "casa")==0);
    assert(strcmp(aux[3], "Hogar")==0);
    assert(strcmp(aux[4], "vivienda")==0);
    assert(aux[5] == NULL);
    freeAll(aux);

    add(dic, "hogar", "casa");         // Ahora "casa" es sinónimo de "Hogar",
                                  // pero el size no cambia, ya estaba "Hogar"
    assert(size(dic)==5);
    aux = synonyms(dic, "hogar");
    assert(strcmp(aux[0], "casa")==0);
    assert(aux[1] == NULL);
    freeAll(aux);

    freeDict(dic);

    puts("OK, Correcto, Bien");
    return 0;
}
```

Ejercicio 4 (2,5 puntos)

Dado el siguiente contrato para un TAD que almacena **elementos genéricos**, donde un elemento puede ser de cualquier tipo (entero, double, estructura, string, etc.)

```
typedef struct collectionCDT * collectionADT;

typedef .. elemType;     // Tipo de elemento a insertar

/* Crea una nueva colección de elementos genéricos
** Inicialmente la colección está vacía
** Cada elemento a insertar será de tipo elemType
** No hay un límite para la cantidad de elementos a insertar
*/
collectionADT newCollection( ¿? );

/* Retorna cuántos elementos hay insertados */
```

```
int elementCount(collectionADT c);

/* Almacena un elemento en la posición pos.
** Si había un elemento en esa posición, lo pisa con elem
*/
void putElement(collectionADT c, elemType elem, size_t pos);

/* Elimina el elemento en la posición pos.
** Si no hay elementos en pos, no hace nada */
void deleteElement(collectionADT c, size_t pos);

/* Retorna la posición en el cual está insertado el elemento,
** o -1 si no lo encuentra
*/
int getPosition(collectionADT c, elemType elem);

/* Libera todos los recursos reservados por el TAD */
void freeCollection(collectionADT c);
```

Donde **¿? en una lista de parámetros indica que usted (programador) debe definir cuáles son los parámetros necesarios para esa función**, en base a las características del TAD.

Se pide:

a) **Completar la definición de `struct colectionCDT`**
b) Escribir la función `newCollecction`
c) Escribir la función `putElement`
d) Escribir la función `getPosition`

El resto de las funciones ya están implementadas

DEFINIR TODOS LOS TIPOS DE DATOS QUE SE VAYAN A USAR EN LAS FUNCIONES DEL TAD

SE ASUME QUE LA COLECCIÓN TENDRÁ UN BAJO PORCENTAJE DE POSICIONES LIBRES

LA FUNCIÓN `putElement` DEBE SER LO MÁS EFICIENTE POSIBLE

Ejercicio 5 (2,5 puntos)

La **Biblia** consta de **76 libros**, y cada libro está **compuesto a su vez por versículos**. Algunos libros tienen pocos versículos, por ejemplo 21, otros tienen más de 2000 (el libro Salmos contiene 2146 versículos). Cada versículo es un texto que puede ser corto o tener cientos de caracteres.

Se creará un TAD para **almacenar la Biblia completa**, sabiendo que:

- **Se la ingresará completa pero no necesariamente en orden** (utilizando la función **`addVerse`**). Por ejemplo una persona puede estar agregando los primeros versículos del libro 1 (Génesis) mientras otra persona ingresa los últimos versículos del libro 6 (Josué) y otro los capítulos intermedios del libro 42 (Lucas).
- Se harán **muchas consultas** a la Biblia (utilizando la función **`getVerse`**) **por un versículo en particular**, para lo cual se indicará el número de libro y el número de versículo.

```
typedef struct bibleCDT * bibleADT;

bibleADT newBible();
```

```
/*
** Agrega un versículo a la Biblia. Si ya estaba ese número de versículo en ese
** número de libro, no lo agrega ni modifica y retorna 0. Si lo agregó retorna 1
** bookNbr: número de libro
** verseNbr: número de versículo
*/
int addVerse(bibleADT bible, size_t bookNbr, size_t verseNbr, const char * verse);

/*
** Retorna una copia de un versículo o NULL si no existe.
** bookNbr: número de libro
** verseNbr: número de versículo
*/
char * getVerse(bibleADT bible, size_t bookNbr, size_t verseNbr);


/* Libera todos los recursos reservados por el TAD */
void freeBible(bibleADT bible);
```

El objetivo principal es que la función `getVerse` sea lo más eficiente posible.

Se pide: **Implementar el TAD completo**

Ejemplo de uso. En este ejemplo se usan **sólo versículos cortos**.

```
int
main(void) {
    bibleADT b = newBible();
    assert(getVerse(b, 1, 1)==NULL);

    char aux[2000];
    strcpy(aux, "En el principio creo Dios los cielos y la tierra.");
    assert(addVerse(b, 1, 1, aux)==1);

    strcpy(aux, "Y atardecio y amanecio: dia tercero.");
    assert(addVerse(b, 1, 13, aux)==1);

    assert(addVerse(b, 1, 13, "Amaos los unos a los otros")==0); // Ya estaba

    strcpy(aux, "los contados de la tribu de Dan fueron sesenta y dos mil setecientos.");
    assert(addVerse(b, 4, 39, aux)==1);

    assert(addVerse(b, 4, 46,
      "fueron todos los contados seiscientos tres mil quinientos cincuenta.")==1);

    char * v = getVerse(b, 4, 45);
    assert(v==NULL);

    v = getVerse(b, 4, 39);
    assert(strncmp(v, "los con", 7)==0);
    free(v);

    freeBible(b);

    puts("Aleluya !");
    return 0;
}
```