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

Segundo Parcial de Programación Imperativa (72.31)

09/11/2018

	Ejercicio 1	*Ejercicio 2*	*Ejercicio 3*	*Nota*	*Firma Docente*
Entregado				-----	
Calificación	2/	4/	4/		

- **Condición mínima de aprobación: Sumar 5 puntos**
- **Se tendrá en cuenta en la calificación el ESTILO y la EFICIENCIA de los algoritmos.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, no serán aceptados.**
- **Puede entregarse en lápiz.**
- **No usar variables globales ni static.**
- **En caso de necesitar usar malloc o similar, no validar que retorne distinto de NULL.**
- **No es necesario escribir los #include.**
- **Escribir en cada hoja Apellido, Legajo, Número de hoja y Total de hojas entregadas.**
- **Realizar los ejercicios en hojas separadas.**

Ejercicio 1:

Escribir una función recursiva **`sumMatch`** que reciba como único parámetro un vector de enteros mayores o iguales a cero, donde la marca de final es un número negativo. La función debe retornar cero si los elementos del vector están "apareados por sumas", y distinto de cero si no, de acuerdo a los siguientes ejemplos:

Para los siguientes vectores la función retorna cero:

```
int v1[]={1,2,3,6,4,3,2,9,1,2,3,-1};     // 1+2+3=6, 4+3+2=9, 1+2=3
int v2[]={1,1,2,4,5,5,14,-1};            // 14=5+5+4, 2=1+1
int v3[]={3,3,-1};                       // 3=3
int v4[]={1,1,2,2,90,90,-1};             // 90=90, 2=2, 1=1
int v5[]={1,0,1,2,2,2,6,-1};             // 6=2+2+2, 1=0+1
int v6[]={-1};
int v7[]={0,-1};
```

Para los siguientes vectores la función retorna distinto cero:

```
int v1[]={1,1,2,3,6,4,3,2,9,1,2,3,-1};
int v2[]={1,1,2,4,5,5,14,10,-1};
int v3[]={4,3,-1};
int v4[]={1,-1};
int v5[]={1,0,1,2,1,2,2,2,-1};
```

No usar variables `static` ni funciones o macros auxiliares

Ejercicio 2:

El siguiente es un TAD de "**multiSet**". Un multi-set es un conjunto de elementos sin orden pero donde cada elemento puede aparecer más de una vez.

`multiSetADT.h`

```
typedef struct multiSetCDT * multiSetADT;

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

/**
** Retorna 0 si los elementos son iguales, negativo si e1 es "menor" que e2 y positivo
** si e1 es "mayor" que e2
*/
static int compare (elemType e1, elemType e2) {
      ...
}

/* Retorna un nuevo multiSet de elementos genéricos. Al inicio está vacío */
multiSetADT newMultiSet(¿?);

/* Inserta un elemento. Retorna cuántas veces está elem en el conjunto
** luego de haberlo insertado (p.e. si es la primera inserción retorna 1).
*/
unsigned int add(multiSetADT multiSet, elemType elem);

/* Retorna cuántas veces aparece el elemento en el multiSet */
unsigned int count(const multiSetADT multiSet, elemType elem);

/* Retorna la cantidad de elementos distintos que hay en el multiSet */
unsigned int size(const multiSetADT multiSet);

/* Retorna el elemento que aparece más veces. Si hay más de uno
** con esa condición, retorna cualquiera de los dos.
** Pre-condición: el multiSet no debe estar vacío
*/
elemType maxElement(const multiSetADT multiSet);

/* Retorna un vector con los distintos elementos (sin repetir) que hay en el multiSet */
elemType * values(const multiSetADT multiSet);
```

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.

Implementar el TAD completo (el `multiSetCDT.c`).

Ejercicio 3:

Se desea implementar un TAD que permita operaciones de **pila** y de **cola**, para lo cual se tiene el siguiente contrato:

collectionADT.h

```
typedef struct collectionCDT * collectionADT;

typedef int elemType;    // Tipo de elemento a insertar, por defecto int

/**
** Retorna 0 si los elementos son iguales, negativo si e1 es "menor" que e2 y
** positivo si e1 es "mayor" que e2
*/
static int compare (elemType e1, elemType e2) {
      return e1 - e2;
}

/* Retorna un nuevo conjunto de elementos genéricos. Al inicio está vacío.
** No hay límite de capacidad.
*/
collectionADT newCollection();

/* Retorna cuántos elementos hay en la colección */
unsigned int size(const collectionADT collection);

/* Operación de Cola: Encola un elemento, agregándolo al final de la colección */
void enqueue(collectionADT collection, elemType elem);

/* Operación de Cola: Desencola el primer elemento de la colección.
** Precondición: la colección no está vacía.
*/
elemType dequeue(collectionADT collection);

/* Operación de Pila: Apila un elemento, agregándolo al tope (o principio)
** de la colección
*/
void push(collectionADT collection, elemType elem);

/* Operación de Pila: Desapila el elemento que está en el tope (el primero)
** de la colección. Precondición: la colección no está vacía.
*/
elemType pop(collectionADT collection);

/* Libera toda la memoria reservada */
void freeCollection(collectionADT collection);
```

Implementar el TAD completo, teniendo en cuenta que TODAS las operaciones para agregar o quitar elementos deben ser lo más eficiente posible.

Ejemplo de uso:

```
#include "collectionADT.h"

int
main(void) {
   collectionADT c = newCollection(); // una colección de elemType, en este caso de int
   push(c, 1);
   push(c, 3);
   push(c, 5);
   int a = dequeue(c);     // a vale 5
   enqueue(c, 7);
   int b = pop(c);         // b vale 3
   int d = dequeue(c);     // d vale 1
   int e = pop(c);         // e vale 7
   int f = dequeue(c);     // ERROR
   return 0;
}
```

Ejercicio 3

```
struct node{
    elemType elem;
    struct node * tail;
};
typedef struct node * TNode;
struct collectionCDT{
    TNode first;
    TNode last;
    size_t size;
};

collectionADT newCollection (void){
     return calloc (1,sizeof (struct collectionCDT)),
}
void enqueue (collectionADT collection, elemType elem){
     TNode aux = calloc (1, sizeof (struct collectionCDT);
     aux -> elem = elem;
     collection -> size ++;
     if(collection ->first ==NULL){
          collection ->first = aux
          collection ->last = aux;
          return ;
     }
     collection ->last ->tail = aux;
     collection ->last;
 }
 elemType dequeue(collectionADT collection){
      if(collection ->first == NULL)
            exit(1);
      elemType elem = collection ->first ->elem;
      TNode aux = collection ->first;
      collection -> size --;
      if(aux == collection ->last){
          collection ->first = NULL;
          collection ->last = NULL;
          return;
      }
     collection ->first = aux->tail;
     free(aux);
      return elem;
 }
```

```
void push(collectionADT collection, elemType elem){
    TNode aux = calloc(1, sizeof(struct collectionCDT));
    aux->elem = elem;
    collection->size++;
    if(collection->first == NULL){
        collection->first = aux;
        collection->last = aux;
        return;
    }
    aux->tail = collection->first;
    collection->first = aux;
}
elemType pop(collectionADT collection){
    if(collection->first == NULL)
        exit(1);
    elemType elem = collection->first->elem;
    TNode aux = collection->first;
    collection->size--;
    if(aux == collection->last){
        collection->first = NULL;
        collection->last = NULL;
        return;
    }
    collection->first = aux->tail;
    free(aux);
    return elem;
}
void freeRec(TNode node){
    if(node==NULL)
        return;
    freeRec(node->tail);
    free(node);
}
void freeCollection(collectionADT collection){
    freeRec(collection->first);
    free(collection);
}
```

Ejercicio 2

```
struct set {
    elemType elem;
    int quantity;
    struct set * tail;
}
typedef struct set * TSet;
struct multiSetCDT {
    TSet set;
    size_t size;
};

multiSetADT newMultiSet(void){
    return calloc(1, sizeof(struct multiSetCDT));
}
static TSet addRec(TSet set, elemType elem, size_t size, int * cont){
    int c;
    if(set == NULL || (c = compare(elem, set->elem)) <0){
        TSet aux = malloc(sizeof(struct set));
        aux->elem = elem
        aux -> quantity = 1;
        aux -> tail = set;
        (*cont) = aux->quantity;
        (*size) ++;
        return aux;
    }
    if(c==0){
        set -> quantity += 1;
        (*cont) = set ->quantity;
        return set;
    }
    set -> tail = addRec(set -> tail, elem, size, cont);
    return set;
  }
unsigned int add(multiSetADT multiset, elemType elem){
    int cont;
    multiSet -> set = addRec(multiSet -> set, elem, &(multiSet -> size), &cont);
    return cont;
}
static int countRec(TSet set, elemType elem){
    int c;
    if( set == NULL || (c= compare(elem, set->elem))<0)
        return 0;
    if(c==0)
      return set -> quantity;
    return countRec(set ->tail, elem);
}
```

```
unsigned int count (const multiSetADT multiSet, elemType elem){
    return countRec (multiSet -> set , elem);
}
unsigned int size (const multiSetADT multiSet){
     return multiSet -> size;
}
void maxElemRec (const TSet set, int * max, elemType * maxElem){
    if (set == NULL)
       return;
    if (set ->quantity >(*max)){
        (*max) = set -> quantity;
        (*maxElem) = set -> elem;
    }
    maxElemRec (set->tail , max, maxElem);
    return;
}
elemType maxElement (const multiSetADT multiSet){
     int max =0;
     elemType maxElem;
      maxElemRec (multiSet -> set, &max , &maxElem);
      return maxElem;
 }
void valuesRec (Tset set, elemType *elems){
    if (set == NULL)
        return;
    valuesRec (set ->tail, elems +1);
    elems[0] = set -> elem;
    return;
 }
elemType * values (const multiSetADT multiSet){
   elemType * elems = malloc ( multiSet -> size * (sizeof (elemType));
    valuesRec (multiSet ->set, elems);
    return elems;
}
```

Ejercicio 1

```
int sumMatch (int * vec){
   if(vec[0]== -1)
      return 0;
   int aux = sumMatch (vec + 1);
   if(aux == 0)
      return  vec[0];
   return aux - vec[0];
}
```