Trabajo Práctico 2
Programación en C

Tecnología Digital II

1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre distintas estructuras de datos simples y construir un conjunto de casos de test para comprobar su correcto funcionamiento. Las funciones a implementar se realizarán sobre dos tipos de datos. Por un lado `strings` de C, es decir, cadenas de caracteres terminadas en *null*, y por otro un nuevo tipo denominado `trainDance`. Internamente, el tipo `trainDance` utilizará una lista circular para guardar datos. Estos datos serán nombres como `strings` y números enteros como identificadores. La idea es que el tipo `trainDance` modele un conjunto de personas bailando el baile del trencito. El conjunto de personas se tomará unas de las otras hasta armar un tren de personas, donde el primero toma también al último. En la estructura cada `nodo` de la lista circular estará compuesto por el nombre de la persona y un identificador numérico.

> El trabajo práctico debe realizarse en grupos de tres personas. Tienen tres semanas para realizar la totalidad de los ejercicios. **La fecha de entrega límite es el 5 de junio hasta las 23:59.** Se solicita no realizar consultas del trabajo práctico por los foros públicos. Limitar las preguntas al foro privado creado para tal fin.

2. Tipo de datos: `trainDance`

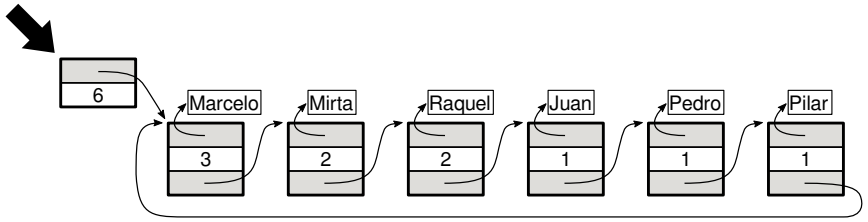
Se define a partir de las siguientes estructuras:

```
struct node {
    char* name;
    int group;
    struct node* next;
};
```

```
struct trainDance {
    struct node* first;
    int count;
};
```

La estructura `trainDance`, contiene un puntero a un dato de tipo `node`, que corresponde al nodo de entrada de la lista circular. Al ser una lista circular, no hay primer nodo, así que podemos decir que el nodo de entrada es simplemente un nodo particular de la lista. Esto nodos construyen una cadena simplemente enlazada de nodos donde el último de la lista, vuelve a apuntar al primero. Además, la estructura `trainDance` cuenta con un contador de la cantidad de nodos en la lista (`count`).
La estructura `node`, por otro lado, contiene un puntero a la *string* nombre (`name`), un entero de 32 bits denominado `group` y un puntero al nodo siguiente. Considerar que en todo momento la lista tendrá un siguiente, por lo que el puntero jamas será cero.

A continuación se ilustra un ejemplo de la estructura:



La lista circular modela personas bailando, por lo que para agregar una persona al baile, dos personas contiguas en el trencito deben invitarla a bailar. La función `trainDanceAddToDance` busca modelar este comportamiento, ya que como entrada toma dos personas contiguas en la lista y una tercera para agregar entre estas dos.
Por otro lado, para quitar un nodo de la lista, o quitar una persona del baile se utilizará la función `trainDanceImTired`, que toma como parámetro el nombre de una persona y la quita del baile (ya que está cansada).

En el ejemplo se ilustra un trencito de 6 personas. El mismo comienza con `Marcelo` y termina con `Pilar`. Al ser circular, `Pilar` se vuelve a conectar con el primero de la lista, es decir `Marcelo`.
En la estructura principal, identificada por la flecha gorda, se tiene un puntero al nodo de entrada de la lista, y un número que indica la cantidad de personas en el baile.

3. Enunciado

Ejercicio 1

Implementar las siguientes funciones sobre *strings*.

- `char* strDuplicate(char* src)`
  Duplica un *string*. Debe contar la cantidad de caracteres totales de *src* y solicitar la memoria equivalente. Luego, debe copiar todos los caracteres a esta nueva área de memoria. Además, como valor de retorno se debe retornar el puntero al nuevo string.
- `int strCompare(char* s1, char* s2)`
  Compara dos *strings* en orden lexicográfico[1]. Debe retornar:
  - 0 si son iguales
  - 1 si `s1` $<$ `s2`
  - -1 si `s2` $<$ `s1`

  Ejemplos:
  `strCompare("ala","perro")` $\rightarrow$ `1`
  `strCompare("casa","cal")` $\rightarrow$ `-1`
  `strCompare("topo","top")` $\rightarrow$ `-1`
  `strCompare("pato","pato")` $\rightarrow$ `0`

- `char* strConcatenate(char* src1, char* src2)`
  La función toma dos *strings* `src1` y `src2`, y retorna una nueva *string* que contiene una copia de todos los caracteres de `src1` seguidos de los caracteres de `src2`. Además, la memoria de `src1` y `src2` debe ser liberada.

  Ejemplos:
  `strConcatenate("mos","tacho")` $\rightarrow$ `"mostacho"`
  `strConcatenate("elefant","e")` $\rightarrow$ `"elefante"`
  `strConcatenate("tucan","")` $\rightarrow$ `"tucan"`

Ejercicio 2

Este ejercicio consiste en implementar funciones sobre el tipo de datos `trainDance`. Para simplificar el desarrollo, se provee un conjunto de funciones útiles ya implementadas.

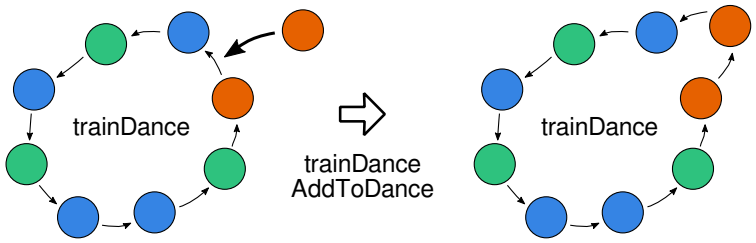
- `struct trainDance* trainDanceNew(char* name1, int group1, char* name2, int group2)`
  Construye un trencito mínimo de dos personas. No puede existir un tren más chico que dos personas. Los nombres pasados por parámetro son copiados.

[1] `https://es.wikipedia.org/wiki/Orden_lexicografico`

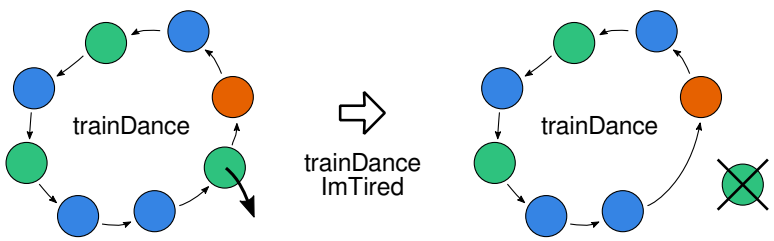
- `int trainDanceGroupCount(struct trainDance* train, int group)`
  Cuenta la cantidad de personas para un determinado grupo.
- `void trainDanceDelete(struct trainDance* train)`
  Libera toda la memoria solicitada para el trencito.
- `void trainDancePrint(struct trainDance* train)`
  Imprime en pantalla el trencito pasado por parámetro.

Se pide entonces implementar las siguientes funciones:

- `struct node* trainDanceGetNode(struct trainDance* train, char* name)`
  Dado un nombre (`name`), retorna el nodo asociado a este nombre. En caso de no existir debe retornar cero. En caso de existir dos nombres iguales, debe retornar el primero que encuentre.
- `char* trainDanceGetNames(struct trainDance* train, char* separator)`
  Construye una *string* con los nombres en todos los nodos de la lista, utilizando la *string* `separador` entre cada par de nombres.
  Suponiendo un separador `"-"`, para el caso del ejemplo del enunciado el resultado sería:
  `Marcelo-Mirta-Raquel-Juan-Pedro-Pilar`
- `int trainDanceAddToDance(struct trainDance* train, char* name1, char* name2, char* nameNew, int groupNew)`
  Para agregar una persona al baile, dos personas ubicadas consecutivamente deben invitarlo. La función toma dos nombres, y de estar estos ubicados consecutivamente. Se agregará al baile a la persona de nombre `nameNew` y se le asignará el grupo `groupNew`. En caso que las personas de nombres `name1` y `name2` no se encuentren consecutivamente ubicadas en la lista y en este orden, no se agregará nada. Tener en cuenta que la *string* `nameNew` pasada por parámetro debe ser copiada.

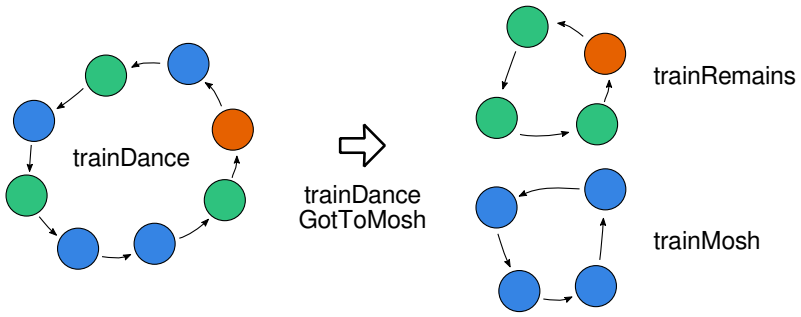


- `int trainDanceImTired(struct trainDance* train, char* name)`
  Cuando una persona se siente cansada de bailar, se puede ir del baile. Para esto la función toma el nombre de la persona cansada, la busca dentro de la lista y la quita. Se debe tener en cuenta que toda la memoria ocupada por el nodo y el nombre deben ser liberados. Además, puede pasar que la persona cansada sea la apuntada como nodo de entrada de la lista circular. En ese caso de debe hacer que el puntero al nodo de entrada sea el siguiente no cansado dentro de la lista.



- `void trainDanceGotToMosh(struct trainDance** train, struct trainDance** trainMosh, struct trainDance** trainRemains, int groupMosh)`
  Esta función modela el comportamiento de hacer *pogo*. En este caso se toma como parámetro

tres dobles punteros. El primero, corresponde al `trainDance` original, el segundo contendrá un nuevo `trainDance` de todas las personas que se suman al *pogo* y el último contendrá las personas restantes. Para decidir quien entra en el *pogo*, se utilza el campo `group`. Estos serán todos los que tengan el mismo `group` que el número `groupMosh` pasado por parámetro.
Existen algunas restricciones a armar un *pogo*, como también para que exista un trencito. Un trencito puede tener como mínimo dos personas, y un *pogo* no puede dejar al trencito sin personas. Por lo tanto, el mínimo para armar un *pogo*, es un tren de 4 personas, donde 2 arman un nuevo trencito de *pogo*, y las dos persoans restantes quedan en el trencito de restantes. En cualquier otro caso más chico, la función debe retornar cero en los parámetros `trainMosh` y `trainRemains`. Considerar que en caso que finalmente se arme el *pogo*, el trencito original se borra y se retorna cero en el parametro `train`.



Ejercicio 3

A continuación, se enumera un conjunto mínimo de casos de test que deben implementar dentro del archivo `main`:

- `strDuplicate`
  1. String vacío.
  2. String de un carácter.
  3. String que incluya todos los caracteres válidos distintos de cero.
- `strCompare`
  1. Dos string vacíos.
  2. Dos string de un carácter.
  3. Strings iguales hasta un carácter (hacer `cmpStr(s1,s2)` y `cmpStr(s2,s1)`).
  4. Dos strings diferentes (hacer `cmpStr(s1,s2)` y `cmpStr(s2,s1)`).
- `strConcatenate`
  1. Un string vacío y un string de 3 caracteres.
  2. Un string de 3 caracteres y un string vacío.
  3. Dos strings de 1 caracter.
  4. Dos strings de 5 caracteres.
- `trainDanceGetNode`
  1. Obtener el primero de la lista.
  2. Obtener el último de la lista.
  3. Buscar uno que no este en la lista.
- `trainDanceGetNames`
  1. Obtener los nombres de un baile de 2 personas.
  2. Obtener los nombres de un baile de 3 personas.

3. Obtener los nombres de un baile de 4 personas.

- `trainDanceAddToDance`

  1. Agregar una persona la principio de la lista.
  2. Agregar una persona al final de la lista.
  3. Agregar una persona en cualquier parte de la lista.

- `trainDanceImTired`

  1. Quitar personas de un baile de solo 2.
  2. Quitar el último de la lista, con al menos 3 personas.
  3. Quitar el primero de la lista, con al menos 3 personas.

- `trainDanceGotToMosh`

  1. Un baile de 2 personas y solo uno para hacer pogo.
  2. Un baile de 3 personas y dos para hacer pogo.
  3. Un baile de 4 personas, dos para pogo y dos de resto.

Entregable

Para este trabajo práctico no deberán entregar un informe. Sin embargo, deben agregar comentarios en el código que expliquen su solución. No deben comentar qué es lo que hace cada una de las instrucciones sino cuáles son las ideas principales del código implementado y por qué resuelve cada uno de los problemas.
La entrega debe contar con el mismo contenido que fue dado para realizarlo más lo que ustedes hayan agregado, habiendo modificado **solamente** los archivos `trencito.c` y `main.c`. Es requisito para aprobar entregar el código correctamente comentado.