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 `stings` de C, es decir, cadenas de caracteres terminadas en *null*, y por otro un nuevo tipo denominado `container`. Internamente, el tipo `container` utilizará listas para guardar datos, particularmente almacenará `strings` ordenados en conjuntos.

> El trabajo práctico debe realizarse en grupos de tres personas. Tienen dos semanas para realizar la totalidad de los ejercicios. **La fecha de entrega límite es el domingo 31 de octubre 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: `container`

Se define a partir de las siguientes estructuras:

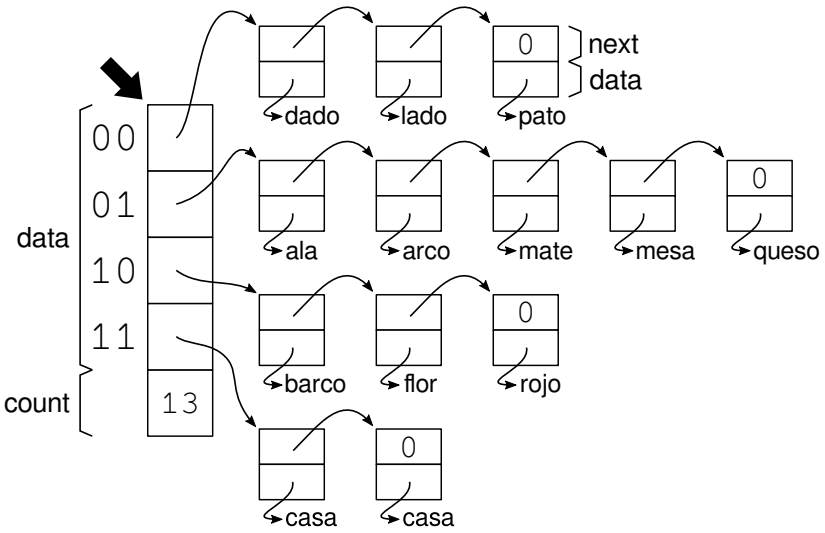
```
struct node {
    struct node* next;
    char* data;
};
```

```
struct container {
    struct node* data[4];
    int count;
};
```

La estructura `container`, contiene un arreglo de 4 punteros a datos de tipo `node`. Estos construyen una cadena simplemente enlazada de nodos que deben estar ordenados según el dato que almacenan. Además, la estructura `container` cuenta con un contador de la cantidad de datos almacenados (`count`). La estructura debe contener en todo momento una copia de los datos, es decir, cuando un dato es agregado, este debe ser copiado, y cuando la estructura es borrada, se deben borrar los datos que esta contiene.

La estructura `node`, por otro lado, contiene un puntero al dato (una *string* de C) y un puntero al nodo siguiente, que en caso de ser el último deberá ser cero.

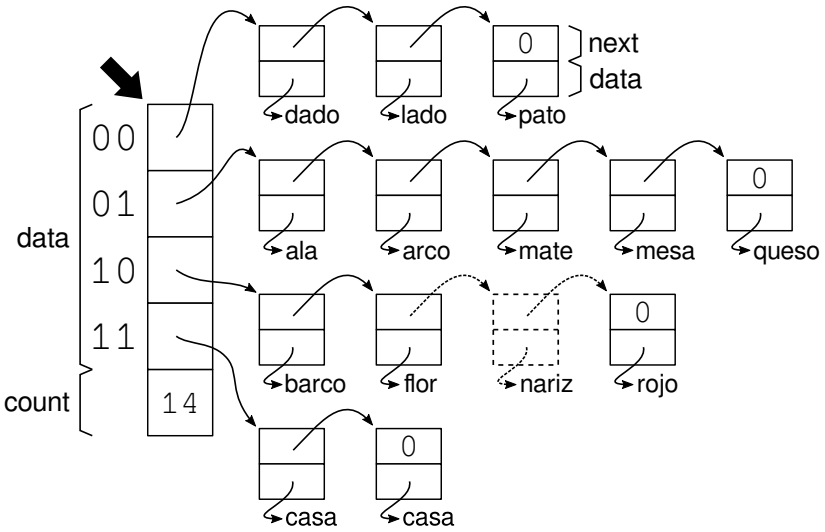
A continuación se ilustra un ejemplo de la estructura:



Los *strings* que se agregan al `container` se agregan sobre una de las 4 listas existentes. Para determinar sobre qué lista se debe agregar, se toma el primer byte del *string* y sobre este byte se identifica el par de bits menos significativos. Los bits se usan como índice en el arreglo de 4 posiciones. Por ejemplo, dado el string `"arco"` que comienza con `"a"`, entonces el byte que corresponda a esta letra es 97_{d}. Si se toman los últimos dos bits se obtiene 01_{b}, resultando en la posición `1` del arreglo `data` dentro de la estructura `container`.

Por último, una vez seleccionada la lista (o *slot*), se debe agregar el *string* en la lista de forma ordenada. Para esto se debe buscar el lugar donde agregar el nuevo nodo de la lista y el dato utilizando la función `cmpStr` implementada previamente. A partir de esta función es posible comparar dos *strings* y establecer una relación de orden entre los mismos.

Ejemplo:



Se agrega el *string* `"nariz"`. Como comienza con `"n"`, corresponde a la tercera lista. Se recorre la lista hasta encontrar dónde debe agregarse. En este caso debe ir entre `"flor"` y `"rojo"`. Por último, se incrementa el contador de datos.

3. Enunciado

Ejercicio 1

Implementar las siguientes funciones sobre *strings*.

1. `int dupStr(char* src, char** dst)`
   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. El puntero al nuevo string se almacenará en el doble puntero `dst`. Además, como valor de retorno se debe retornar el tamaño del *string*.

2. `int cmpStr(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:
   `cmpStr("ala","perro")` $\rightarrow$ `1`
   `cmpStr("casa","cal")` $\rightarrow$ `-1`
   `cmpStr("topo","top")` $\rightarrow$ `-1`
   `cmpStr("pato","pato")` $\rightarrow$ `0`

3. `void split(char* source, int count, char** s1, char** s2)`
   A partir del *string* en `source`, genera dos nuevos strings `s1` y `s2`. `s1` debe contener los primeros `count` caracteres del string *source*, mientas que `s2` debe contener los caracteres restantes. La memoria del *string* `source` pasado por parámetro debe ser liberada. En caso que `count` supere la cantidad de caracteres totales de `source`, se debe retornar en `s2` un *string* vacío. El parámetro `count` es siempre un número positivo.

   Ejemplos:
   `split("mostacho",3,&s1,&s2)` $\rightarrow$ `s1="mos"`, `s2="tacho"`
   `split("elefante",0,&s1,&s2)` $\rightarrow$ `s1=""`, `s2="elefante"`
   `split("tucan",10,&s1,&s2)` $\rightarrow$ `s1="tucan"`, `s2=""`

Ejercicio 2

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

- `struct container* newContainer()`
  Solicita memoria e inicializa una estructura `container` vacía.
- `void deleteContainer(struct container* c)`
  Borra toda la memoria solicitada dentro de un `container`.
- `void printContainer(struct container* c)`
  Imprime en pantalla el container pasado por parámetro.

Se pide entonces implementar las siguientes funciones:

1. `void sortedAdd(struct container* c, char* value)`
   Agregar el *string* `value` pasado por parámetro dentro del `container`, respetando los invariantes de la estructura indicados anteriormente.
2. `int contains(struct container* c, char* value)`
   Determina la cantidad de veces que `value` está definido dentro de `container`. Debe retonar la cantidad de apariciones del dato.
   Sea el siguiente ejemplo en el que se tiene inicialmente la estructura:

```
0 -> [dado]->[lado]->[pato]->0
1 -> [ala]->[arco]->[mate]->[mesa]->[queso]->0
2 -> [barco]->[flor]->[nariz]->[rojo]->0
3 -> [casa]->[casa]->0

contains(c,"pero") → 0
contains(c,"arco") → 1
contains(c,"casa") → 2
```

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

3. `int inverseDelete(struct container* c, char* value)`
   Borra todos los datos del *slot* al que pertenece el string `value`, menos todas las copias que existan del string `value`. Debe retonar la cantidad de datos que fueron borrados.
   Sea el siguiente ejemplo en el que se tiene inicialmente la estructura:

   ```
   0 -> [dado]->[lado]->[pato]->0
   1 -> [ala]->[arco]->[mate]->[mesa]->[queso]->0
   2 -> [barco]->[flor]->[nariz]->[rojo]->0
   3 -> [casa]->[casa]->0
   ```

   Se llama a: `inverseDelete(c,"mate")` $\rightarrow$ `4`
   La función retorna 4, ya que se borran los valores `"ala"`, `"arco"`, `"mesa"`, `"queso"`, dejando a la estructura de la siguiente forma:

   ```
   0 -> [dado]->[lado]->[pato]->0
   1 -> [mate]->0
   2 -> [barco]->[flor]->[nariz]->[rojo]->0
   3 -> [casa]->[casa]->0
   ```

Ejercicio 3

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

- `dupStr`
  1. String vacío.
  2. String de un carácter.
  3. String que incluya todos los caracteres válidos distintos de cero.
- `cmpStr`
  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)`).
- `split`
  1. Un string vacío y diferentes valores de `count`.
  2. Un string de un carácter y diferentes valores de `count`.
  3. Un string de múltiples caracteres y diferentes valores de `count`.
- `sortedAdd`
  1. Agregar un dato por lista.
  2. Sobre el primer caso, agregar un dato mayor a todos por lista.
  3. Sobre el primer caso, agregar un dato menor a todos por lista.
  4. Sobre el primer caso, agregar un dato que ordenado termine entre dos elementos.
- `contains`
  1. Dada una estructura, consultar si existe un dato que se ubique como último dato de alguna de las listas.
  2. Dada una estructura, consultar si existe un dato que se ubique como primer dato de alguna de las listas.
  3. Dada una estructura, consultar si existe un dato que se ubique en el medio de una lista.
  4. Dada una estructura, consultar si existe un dato que no esté en la lista.

- `inverseDelete`

    1. Para un dato que se ubique al comienzo de una lista.
    2. Para un dato que se ubique al final de una lista.
    3. Para un dato que se ubique entre dos nodos de una lista.

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 `container.c` y `main.c`. Es requisito para aprobar entregar el código correctamente comentado.