Trabajo Práctico 1
Buscador de Documentos

Organización del Computador II

Primer Cuatrimestre de 2020

1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre distintas estructuras de datos. Consideraremos los tipos básicos `Int`, `Float` y `String`, además de tres estructuras de datos más complejas denominadas `Document`, `List` y `Tree`.

Las funciones sobre `Int` y `Float` permitirán operar sobre estos valores para copiarlos, compararlos e imprimirlos. Las funciones de `String` serán operaciones sobre cadenas de caracteres terminadas en cero.

El tipo de datos `Document` es un arreglo de datos con un tipo asociado, es decir, guarda un vector de datos de diferentes tipos. La estructura de `List` permitirá crear, agregar, borrar e imprimir elementos de una lista doblemente enlazada. Mientras que la estructura `Tree` implementa un árbol binario de búsqueda que almacena claves y valores de diferente tipo.

Todas las estructuras pueden contener datos de cualquier tipo. Por esta razón se agrega a cada estructura un campo que indica el tipo del dato almacenado. Para poder conocer las funciones básicas asociadas a cada tipo, se provee un conjunto de funciones que retornan el puntero a la función correspondiente. Las siguientes son las funciones que permiten obtener la función asociada a cada tipo para una acción dada:

- `funcCmp_t* getCompareFunction(type_t t)`
- `funcClone_t* getCloneFunction(type_t t)`
- `funcDelete_t* getDeleteFunction(type_t t)`
- `funcPrint_t* getPrintFunction(type_t t)`

Cada una de estas funciones toma un tipo de datos y retorna la función asociada a la acción indicada. Los tipos de las funciones de retorno son:

- Función comparar: `typedef int32_t (funcCmp_t)(void*, void*);`
- Función duplicar: `typedef void* (funcClone_t)(void*);`
- Función borrar: `typedef void (funcDelete_t)(void*);`
- Función imprimir: `typedef void (funcPrint_t)(void*, FILE*);`

Los valores posibles de entrada como `type_t` son:

```
typedef enum e_type {
    TypeNone = 0,
    TypeInt = 1,
    TypeFloat = 2,
    TypeString = 3,
    TypeDocument = 4
} type_t;
```

2. Estructuras

En esta sección se describiran las estructuras de datos a utilizar.

Estructura `Int`

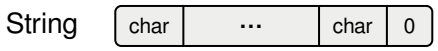
Representa un puntero a un entero con signo de 32 bits.

Estructura `Float`

Este tipo de datos es similar a `Int`, pero utilizando datos en punto flotante de precisión simple.

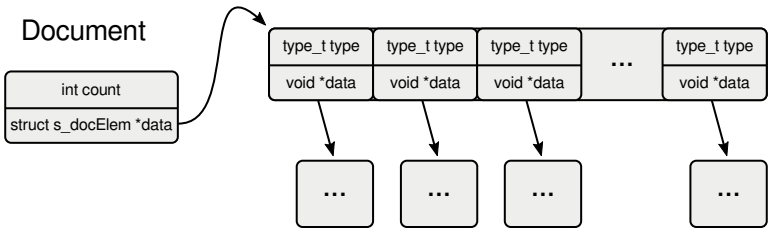
Estructura `String`

Este tipo no es una estructura en sí misma sino un puntero a un *string* de C terminado con el caracter nulo `'\0'`.



Estructura `Document`

Implementa un tipo denominado documento que representa un vector de datos de diferentes tipos, ya sean, *string*, enteros de 32 bits, valores en punto flotante de precisión simple u otros documentos. La estructura `document_t` contiene un puntero al arreglo identificado como *values* y la cantidad de elementos como *count*. Cada valor del arreglo es de tipo `docElem_t` y contiene un puntero al dato almacenado y su tipo.

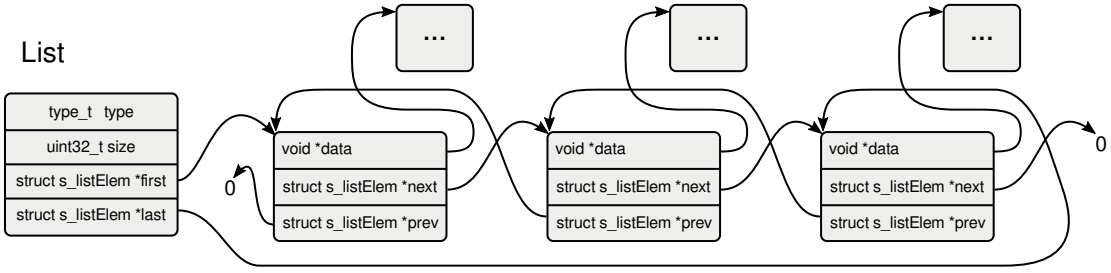



```
typedef struct s_docElem {
    type_t type;
    void* data;
} docElem_t;

typedef struct s_document {
    int count;
    struct s_docElem* values;
} document_t;
```

Estructura `List`

Implementa una lista doblemente enlazada de nodos. La estructura `lista_t` contiene un puntero al primer y último elemento de la lista, mientras que cada elemento es un nodo de tipo `listElem_t` que contiene un puntero a su siguiente, anterior y al dato almacenado. Este último es de tipo `void*`, permitiendo referenciar cualquier tipo de datos.



```
typedef struct s_listElem {
    void* data;
    struct s_listElem* next;
    struct s_listElem* prev;
} listElem_t;

typedef struct s_list {
    type_t   type;
    uint32_t size;
    struct s_listElem* first;
    struct s_listElem* last;
} list_t;
```

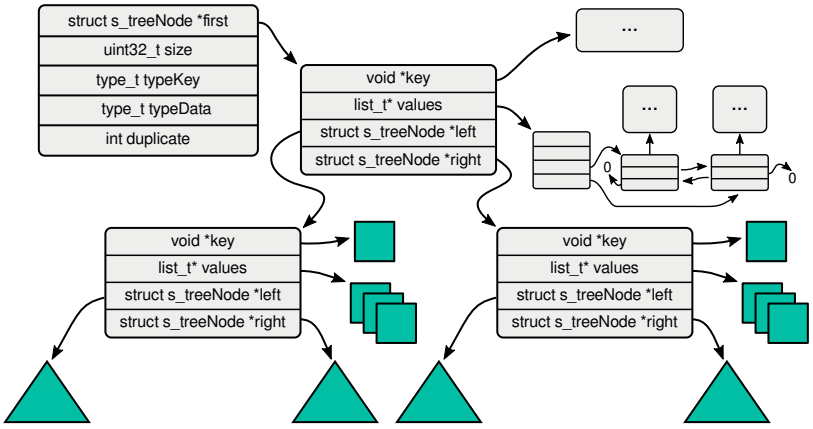
Estructura `Tree`

Esta estructura implementa un árbol binario de búsqueda de claves (Key) y valores (Values). Contiene dos campos con los tipos de datos asociados a la Key (*typeKey*) y a los Values (*typeData*).

La idea básica es que cada vez que se agregue un dato nuevo, este se agregará en un nodo donde algún puntero a derecha o izquierda sea cero. Para encontrar el lugar donde debe ser agregado el dato, se buscará la key en el árbol respetando el invariante que todas las claves almacenadas en el subarbol derecho son más grandes que la clave almacenada en el nodo y todas las claves almacenadas en el subarbol izquierdo son más chicas que la clave almacenada en el nodo. Este invariante se respeta para todos los nodos del árbol.

Si la clave pasada por parámetro a la hora de agregar un nuevo dato ya existe, entonces el dato será agregado a la lista de values del nodo que contiene la clave. Si al borrar datos, un nodo se queda sin valores almacenados (es decir si la lista queda vacía), el nodo seguirá existiendo. Solo será borrado cuando se borre todo el árbol.

Además la estructura soporta poder o no agregar duplicados (es decir, dos datos con el mismo Key).

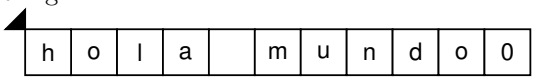



```
typedef struct s_tree {
    struct s_treeNode* first;
    uint32_t size;
    type_t typeKey;
    int    duplicate;
    type_t typeData;
} tree_t;

typedef struct s_treeNode {
    void* key;
    list_t* values;
    struct s_treeNode* left;
    struct s_treeNode* right;
} treeNode_t;
```

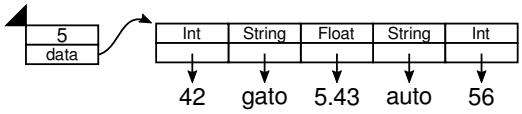
Ejemplos

- String:



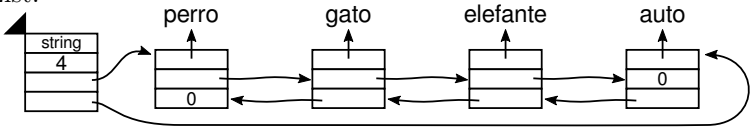
  Print: `hola mundo`

- Document:



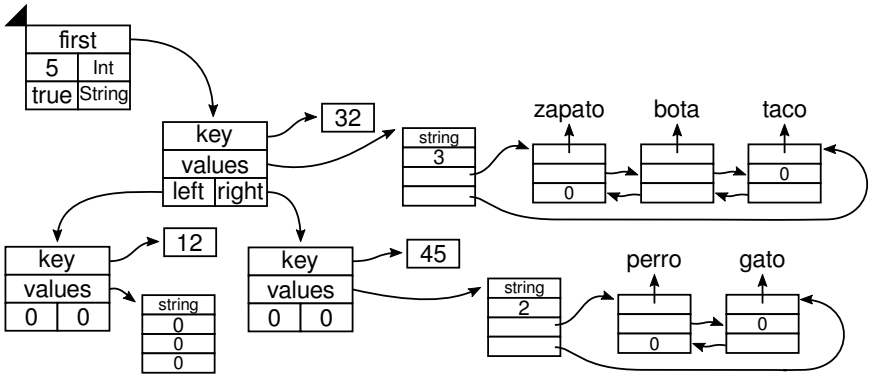
  Print: `{42,gato,5.43,auto,56}`

- List:



  Print: `[perro,gato,elefante,auto]`

- Tree:



  Print: `(12)->[](32)->[zapato,bota,taco](45)->[perro,gato]`

3. Enunciado

El trabajo práctico consiste en implementar algunas funciones para operar sobre las estructuras de datos dadas. Algunas funciones son provistas por la cátedra.

A continuación se detallan las funciones a implementar en lenguaje ensamblador.

- `int32_t floatCmp(float* a, float* b)`(10 Inst.)
- `float* floatClone(float* a)`(8 Inst.)
- `void floatDelete(float* a)`(1 Inst.)
- `char* strClone(char* a)`(19 Inst.)
- `uint32_t strLen(char* a)`(7 Inst.)
- `int32_t strCmp(char* a, char* b)`(25 Inst.)
- `void strDelete(char* a)`(1 Inst.)
- `void strPrint(char* a, FILE* pFile)`(18 Inst.)
- `document_t* docClone(document_t* a)`(39 Inst.)
- `void docDelete(document_t* a)`(23 Inst.)
- `void listAdd(list_t* l, void* data)`(54 Inst.)

- `int treeInsert(tree_t* tree, void* key, void* data)`(82 Inst.)
- `void treePrint(tree_t* tree, FILE* pFile)`(45 Inst.)

Además las siguientes funciones deben ser implementadas en lenguaje C.

- `void listRemove(list_t* l, void* data)`(20 Líneas)
- `list_t* treeGet(tree_t* tree, void* key)`(11 Líneas)
- `int treeRemove(tree_t* tree, void* key, void* data))`(12 Líneas)

Recordar que cualquier función auxiliar que desee implementar debe estar en lenguaje ensamblador o C según corresponda el lenguaje solicitado. A modo de referencia, se indica entre paréntesis la cantidad de instrucciones o líneas de código necesarias para resolver cada una de las funciones.

Compilación y Testeo

Para compilar el código y poder correr las pruebas cortas deberá ejecutar `make main` y luego `./runMain.sh`. En cambio, para compilar el código y correr las pruebas intensivas deberá ejecutar `./runTester.sh`.

Pruebas cortas

Deberá construirse un programa de prueba modificando el archivo `main.c` para que realice las acciones detalladas a continuación, una después de la otra:

1- **Caso document**

- Crear un documento con dos valores enteros, dos valores en punto flotante y dos strings.
- Clonar el documento creado.
- Imprimir ambos documentos.
- Borrar ambos documentos.

2- **Caso list**

- Crear una lista vacía de tipo string
- Agregar exactamente 10 strings cualquiera.
- Crear una lista vacía de tipo float
- Agregar exactamente 5 valores en punto flotante cualquiera.
- Clonar ambas listas.
- Imprimir ambas listas.
- Borrar todas las listas creadas.

3- **Caso tree**

- Crear un tree que permita duplicados con claves de tipo Int y datos de tipo String.
- Agregar los siguientes datos respetando el orden:
  - Key: 24 Data: “papanatas”
  - Key: 34 Data: “rima”
  - Key: 24 Data: “buscabullas”
  - Key: 11 Data: “musica”
  - Key: 31 Data: “Pikachu”
  - Key: 11 Data: “Bulbasaur”
  - Key: -2 Data: “Charmander”
- Crear otro tree y agregar los mismos datos que el caso anterior pero en el orden inverso.
- Imprimir ambos tree.
- Borrar los tree creados.

El programa puede correrse con `./runMain.sh` para verificar que no se pierde memoria ni se realizan accesos incorrectos a la misma.

Pruebas intensivas (Testing)

Entregamos también una serie de *tests* o pruebas intensivas para que pueda verificarse el buen funcionamiento del código de manera automática. Para correr el testing se debe ejecutar `./runTester.sh`, que compilará el *tester* y correrá todos los tests de la cátedra. Un test consiste en la creación, inserción, eliminación, ejecución de funciones e impresión en archivos de alguna estructura implementada. Luego de cada test, el *script* comparará los archivos generados por su TP con las soluciones correctas provistas por la cátedra. También será probada la correcta administración de la memoria dinámica.

Notas

- Toda la memoria dinámica reservada usando la función `malloc` debe ser correctamente liberada, utilizando la función `free`.
- Para el manejo de archivos se recomienda usar las funciones de `C`: `fopen`, `fprintf`, `fwrite`, `fputc`, `fputs` y `fclose`.
- Para el manejo de strings **no está permitido** usar las funciones de `C`: `strlen`, `strcpy`, `strcmp` y `strdup`.
- Para correr los tests deben tener instalado *Valgrind* (en Ubuntu: `sudo apt-get install valgrind`).
- Para corregir los TPs usaremos los mismos tests que les fueron entregados. Es condición necesaria para la aprobación que el TP supere correctamente todos los tests.

Archivos

Se entregan los siguientes archivos:

- `lib.h`: Contiene la definición de las estructuras y de las funciones a realizar. No debe ser modificado.
- `lib.asm`: Archivo a completar con la implementación de las funciones en lenguaje ensamblador.
- `lib.c`: Archivo a completar con la implementación de las funciones en lenguaje C.
- `Makefile`: Contiene las instrucciones para ensamblar y compilar el código.
- `main.c`: Archivo donde escribir los ejercicios para las pruebas cortas (`runMain.sh`).
- `tester.c`: Archivo del tester de la cátedra. No debe ser modificado.
- `runMain.sh`: Script para ejecutar el test simple (pruebas cortas). No debe ser modificado.
- `runTester.sh`: Script para ejecutar todos los tests intensivos. No debe ser modificado.
- `salida.caso*.catedra.txt`: Archivos de salida para comparar con sus salidas. No debe ser modificado.

4. Informe y forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar el mismo contenido que fue dado para realizarlo, habiendo modificado **solamente** los archivos `lib.asm`, `lib.c` y `main.c`.

La fecha de entrega de este trabajo es Martes 29/09. Deberá ser entregado a través de un repositorio GIT almacenado en `https://git.exactas.uba.ar` respetando el protocolo enviado por mail y publicado en la página web de la materia (`https://campus.exactas.uba.ar/`). A la hora de subir su trabajo al GIT es fuertemente recomendable que ejecuten la operación `make clean` borrando todos los binarios e impidiendo que estos sean cargados al repositorio.
Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes
`orga2-doc@dc.uba.ar`.

5. Anexo

Esta sección describe la operatoria de todas las funciones, ya sean implementadas por la cátedra o solicitadas como parte del trabajo práctico.

Estructura `Int`

- `int32_t intCmp(int32_t* a, int32_t* b)`
  Compara los valores de `a` y `b`, y retorna:
  - 0 si son iguales
  - 1 si `a` $<$ `b`
  - -1 si `b` $<$ `a`
- `int32_t* intClone(int32_t* a)`
  Solicita 4 bytes de memoria donde copiar el dato almacenado en `a`.
- `void intDelete(int32_t* a)`
  Libera la memoria que contiene el dato pasado por parámetro. Equivalente a la función *free*.
- `void intPrint(int32_t* a, FILE* pFile)`
  Imprime el valor entero en el *stream* indicado a través de `pFile`.

Estructura `Float`

- `int32_t floatCmp(float* a, float* b)`
  Compara los valores de `a` y `b`, y retorna:
  - 0 si son iguales
  - 1 si `a` $<$ `b`
  - -1 si `b` $<$ `a`
- `float* floatClone(float* a)`
  Solicita 4 bytes de memoria donde copiar el dato almacenado en `a`.
- `void floatDelete(float* a)`
  Libera la memoria que contiene el dato pasado por parámetro. Equivalente a la función *free*.
- `void floatPrint(float* a, FILE* pFile)`
  Imprime el valor de punto flotante en el *stream* indicado a través de `pFile`.

Estructura `String`

- `char* strClone(char* a)`
  Genera una copia del *string* pasado por parámetro. El puntero pasado siempre es válido aunque podría corresponderse a la *string* vacía.
- `uint32_t strLen(char* a)`
  Retorna la cantidad de caracteres distintos de cero del *string* pasado por parámetro.
- `int32_t strCmp(char* a, char* b)`
  Compara dos *strings* en orden lexicográfico[1]. Debe retornar:
  - 0 si son iguales
  - 1 si `a` $<$ `b`
  - -1 si `b` $<$ `a`
- `void strDelete(char* a)`
  Borra el *string* pasado por parámetro. Esta función es equivalente a la función `free`.
- `void strPrint(char* a, FILE* pFile)`
  Escribe el *string* en el *stream* indicado a través de `pFile`. Si el *string* es vacío debe escribir "`NULL`".

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

Estructura `Document`

- `document_t* docNew(int32_t size, ... )`
  Crea un nuevo documento tomando una cantidad variable de parámetros. El primero es la cantidad de datos que tendrá el documento, para cada dato se indicará su tipo y el dato en sí mismo. Los siguientes parámetros se leerán de a pares, primero corresponderá al tipo y el siguiente al puntero al dato; tantas veces como datos se agreguen al documento. Para todos los datos se generará una copia del dato pasado por parámetro. Para esto será necesario utilizar la función clone asociada al tipo. Ejemplo: `docNew(3, TypeInt, &integer, TypeString, &text, TypeFloat, &valueFloat)`. Este llamado creará un documento de tres datos, el primero será un entero, el siguiente una *string* y el último un valor en punto flotante.
- `int32_t docCmp(document_t* a, document_t* b)`
  Compara dos documentos usando las siguientes reglas:

  1. Compara el tamaño de los documentos en cantidad de datos.
  2. Si tienen la misma cantidad de datos. Compara los tipos de los datos uno a uno en orden de aparición en los documentos.
  3. Si tienen los mismos tipos de datos. Compara los datos uno a uno en orden de aparición en los documentos, usando para esto la función de comparación asociada a cada tipo.
  4. Si pasa todas estas comparaciones entonces los documentos son iguales.

  El resultado de la función se debe reflejar según los siguientes casos.

  · 0 si son iguales

  · 1 si `a` $<$ `b`

  · -1 si `b` $<$ `a`
- `document_t* docClone(document_t* a)`
  Genera una copia del documento junto con todos sus datos. Para esto, debe llamar a las funciones clone de cada uno de los tipos de los datos que integran el documento.
- `void docDelete(document_t* a)`
  Borra un documento, esto incluye borrar todos los datos que contiene utilizando las funciones delete asociadas a cada tipo de los datos.
- `void docPrint(document_t* a, FILE* pFile)`
  Imprime un documento, para esto llama a las funciones imprimir de cada tipo dentro del documento. El formato es: `{`dato_0`, ... ,` dato_n`}` donde dato_0 a dato_n son el resultado de llamar a la función print de cada tipo de datos.

Estructura `List`

- `list_t* listNew(type_t t)`
  Crea una nueva `list_t` vacía donde los punteros a `first` y `last` estén inicializados en cero. Además toma el tipo de datos que almacenará la lista, este dato será utilizado por las diferentes funciones para obtener las funciones que permiten operar con los datos almacenados.
- `void listAdd(list_t* l, void* data)`
  Agrega un nuevo nodo que almacene `data`, respetando el orden dado por la función de comparación para el tipo de datos de la lista. Esta función no debe hacer una copia del dato.
- `void listRemove(list_t* l, void* data)`
  Borra todos los nodos de la lista cuyo dato sea igual al contenido de `data` según la función de comparación asociada al tipo de datos almacenado en la lista.
- `list_t* listClone(list_t* l)`
  Construye una copia de la lista, copiando todos los elementos de la misma mediante la función clone asociada al tipo de datos almacenado en la lista.
- `void listDelete(list_t* l)`
  Borra la lista completa con todos sus nodos. Para borrar los datos utiliza la función delete asociada al tipo de datos almacenado en la lista.

- `void listPrint(list_t* l, FILE* pFile)`
  Escribe en el *stream* indicado por `pFile` la lista almacenada en `l`. Para cada dato llama a la función `fp`, y si esta es cero, escribe el puntero al dato con el formato `"%p"`. El formato de la lista será: `[`x_0`,...,`x_{n-1}`]`, suponiendo que x_i es el resultado de escribir el i-ésimo elemento.

Estructura `Tree`

- `tree_t* treeNew(type_t typeKey, type_t typeData, int duplicate)`
  Crea e inicializa un nuevo arbol. Inicialmente el puntero al primer nodo debe ser cero.
- `int treeInsert(tree_t* tree, void* key, void* data)`
  Inserta un nuevo dato al árbol. Para esto recorre los nodos, utilizando la key en el nodo para comparar hasta encontrar uno cuyo puntero a *left* o *right* sea cero. En ese caso agrega un nuevo nodo con los valores de Key y Data asociados. Si encuentra un nodo que tenga el mismo key, agrega el dato como parte de la lista del nodo existente. En cualquier caso siempre clona los datos pasados por parámetro de ser necesario. Si el valor duplicate es distinto de cero, permitirá que las listas de valores almacenen mas de un dato. Si duplicate es cero, las listas podrán tener como máximo un solo dato. El valor de retorno de esta función es 1 si el dato fue agregado y cero en el caso contrario.
- `list_t* treeGet(tree_t* tree, void* key)`
  Recorre el árbol buscando los datos asociados a la key pasada por parámetro. Si encuentra un nodo con la Key correspondiente, entonces retorna una copia completa de la lista contenida en el nodo.
- `void treeRemove(tree_t* tree, void* key, void* data))`
  Borra el dato pasado por parámetro del árbol. Para esto debe recorrer el árbol utilizando la key y una vez en el nodo, recorrer la lista y borrar el dato en cuestión. La lista dentro del nodo puede quedar en cero, pero la key debe seguir siendo válida. Esta no debe ser borrada cuando se borra un dato del árbol.
- `void treeDelete(tree_t* tree)`
  Borra todo el árbol, para esto recorre recursivamente todos los nodos del arbol. Tanto para borrar las listas como para borrar las keys debe utilizar la función delete asociada al tipo de datos que se esta procesando.
- `void treePrint(tree_t* tree, FILE* pFile)`
  Recorre recursivamente todos los nodos del árbol. El árbol deberá recorrerse de forma *inorder*, es decir, primero los menores y luego los mayores. El formato será el siguiente: `(key)->[`dato_0`,...,`dato_n`]`