Trabajo Práctico 1
Listas colgantes

Organización del Computador 2

Segundo Cuatrimestre 2013

1. Introducción

El objetivo de este trabajo práctico es implementar un conjunto de funciones sobre una estructura recursiva que representa listas de listas. Una lista posee un puntero al primer nodo. Un nodo contiene un nodo siguiente y un nodo hijo, además de un valor de 64 bits y un tipo. Para simplificar este trabajo, los nodos hijos pueden tener hijos pero no tendrán siguientes. Además, todos los nodos de una lista dada tendran valores del mismo tipo.

Las funciones a implementar permiten crear listas, recorrerlas y borrarlas. Además de estas operaciones, se deben proveer dos funciones avanzadas, *filtrar* y *colapsar*. La primera recorre todos los nodos base filtrando los que cumplan una cierta condicion. La segunda recorre todos los nodos base colapsando los que cumplan una cierta condición; la acción de colapsar un nodo base implica reemplazarlo por uno nuevo, cuyo valor depende de sus descendientes que luego se borran.

Los ejercicios a realizar para este trabajo práctico estarán divididos en tres secciones. En la primera deberán completar las funciones que permiten manipular el tipo lista. La segunda sección corresponde a un conjunto de funciones muy sencillas que operan sobre tipos de datos básicos. Estas funciones serán utilizadas por el tipo lista para las operaciones avanzadas. En la última sección deben construir un programa en `C` que arme determinadas listas y ejecute algunas de las funciones ejemplo.

1.1. Tipos lista_colgante_t y nodo_t

La estructura de la lista colgante está compuesta por un bloque de memoria que contiene un puntero al primero de una lista enlazada de nodos. Lo más importante son los nodos, de tipo nodo_t, que apuntan al siguiente elemento y a su hijo (que puede tener otro hijo, pero no un siguiente), de un valor de 64 bits, y por último de un entero que indica su tipo. El valor puede usarse para almacenar los tipos `int`, `double` o `string` de C. En la figura 1 se puede ver un ejemplo para una lista colgante de ints que contiene 5 valores.

```
typedef enum tipo_elementos { Integer, Double, String } tipo_elementos;

typedef union valor_e {
    int i;
    double d;
    char *s;
} __attribute__((__packed__)) valor_e;
```

```
typedef struct lista_colgante_t {
    struct node_t *primero;
} __attribute__((__packed__)) lista_colgante_t;

typedef struct nodo_t {
tipo_elementos tipo;
struct nodo_t *siguiente;
struct nodo_t *hijo;
valor_e valor;
} __attribute__((__packed__)) nodo_t;
```

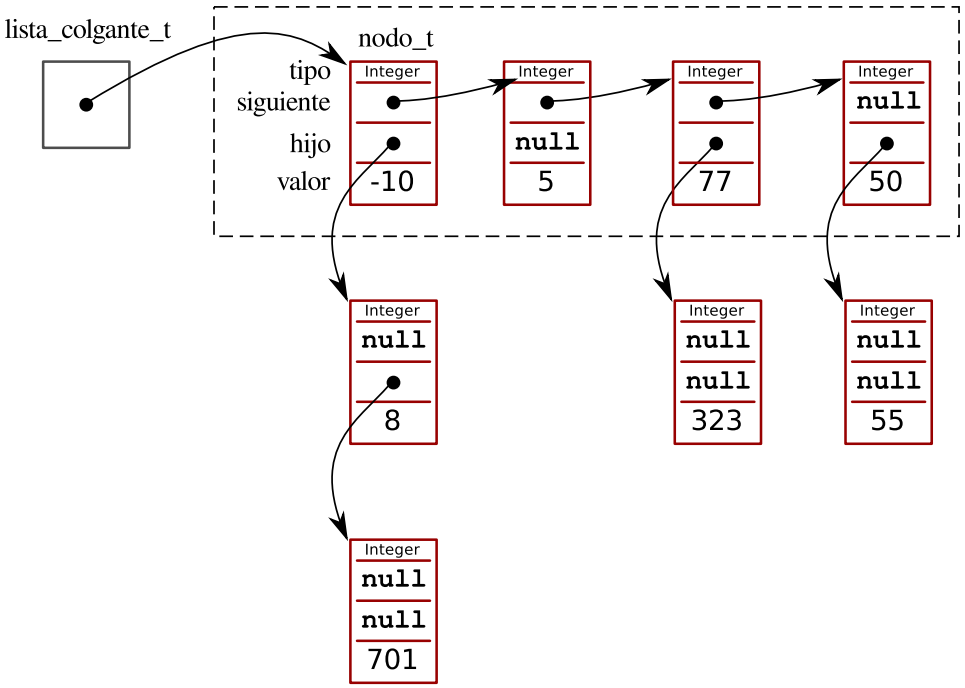

Figura 1: Ejemplo de una estructura lista colgante de `int`s.

Funciones de lista_colgante_t y derivados

- `lista_colgante_t* lista_colgante_t *lista_crear()`
  Crea una nueva lista vacía.

- `nodo_t* nodo_crear(tipo_elementos tipo, valor_e valor)` Crea un nodo nuevo con valor valor, del tipo tipo, sin siguiente ni hijo. En el caso de que el nodo sea de tipo String la lista deberá encargarse de liberar la memoria cuando sea necesario.

- `void lista_borrar(lista_colgante_t *self)`
  Borra la lista `self` y todos sus nodos internos. En el caso de las listas de strings, se debe liberar la memoria apuntada por los nodos

- `void lista_concatenar(lista_colgante_t *self, nodo_t *siguiente)`
  Agrega el nodo siguiente al final de la lista self.

- `void lista_colgar_descendiente(lista_colgante_t *self, uint posicion, nodo_t *hijo)` Agrega el nodo hijo en la lista self, yendo a través del puntero al siguiente hasta la posicion posicion (0-based) y ubica a hijo al final de la cadena de hijos.

- `void lista_imprimir(lista_colgante_t *self, char *nombre_archivo)`
  Debe agregar al archivo pasado por parámetro una línea con la lista self. El archivo se debe abrir en modo append, de modo que las nuevas líneas sean adicionadas al archivo original.
  Cada nodo base de la lista se imprimirá entre `{` y `}` . Debe dejarse un espacio entre las llaves y el contenido de la columna. Dentro de las llaves, cada nodo descendiente se imprime seguido del anterior, entre `[` y `]`. Debe dejarse un espacio entre cada corchete y el contenido del nodo, y tambien entre cada `}` y `{`. El valor impreso dependerá del tipo del nodo, ya sea, un `Integer`, un `Double` o una `String` de C. Si la lista está vacía se debera imprimir `<vacia>`.

  Ejemplo: `lista_imprimir(l, "lista.txt")` para la lista de la figura 1 imprime en `lista.txt`:

  ```
  { [ -10 ][ 8 ][ 701 ] } { [ 5 ] } { [ 77 ][ 323 ] } { [ 50 ][ 55] }
  ```

Funciones avanzadas sobre listas

Previo a presentar el comportamiento de las funciones avanzadas cabe mencionar las siguientes definiciones de tipos.

```
typedef enum boolean { False=0, True=1 } boolean;
```

La misma sirve para indicar un valor de verdad.

```
typedef boolean (*nodo_bool_method)(nodo* self);
```

Representa al tipo de los punteros a funciones que reciben un nodo como argumento y devuelven un booleano.

```
typedef valor_e (*nodo_value_method)(valor_e vA, valor_e vB);
```

Representa al tipo de los punteros a funciones que reciben dos valores como argumento y devuelven otro valor como resultado. Las funciones avanzadas a implementar son las siguientes:

- `void lista_filtrar(lista_colgante_t *self, nodo_bool_method test_method)`
  Toma una lista y una función que devuelve booleanos. Evalúa la función booleana para cada nodo base de la lista. Si el resultado es "false" entonces conserva intacto el elemento evaluado y pasa al nodo siguiente. En caso contrario borra el nodo y a sus descendientes.

- `void lista_colapsar(lista_colgante_t *self,`
  `nodo_bool_method test_method, nodo_value_method join_method)`
  La función `lista_colapsar` toma una lista y dos funciones, que aplicará a los nodos. Para cada nodo base, primero aplica test_method. En caso de que devuelva "False" lo deja intacto. En caso contrario, deberá *colapsar* la cadena de descendientes. Para ello, se creará un nuevo elemento, con el valor del nodo base. Luego se iterará por cada descendiente, evaluando join_method con el valor actual del nuevo nodo y el del nodo descendiente. El resultado de cada evaluación se guardara en el nodo nuevo. Notar que en caso de que el nodo colapsado no tenga hijos, no se aplicará join_method. No olvidar liberar la memoria correspondiente a los nodos que desaparzcan.

Funciones adicionales

Las funciones avanzadas requieren como parámetros otras funciones. A continuación se enumeran las únicas 6 funciones que serán utilizadas como parámetros. Las tres primeras se usarán para las funciones filtrar y colapsar, mientras que las tres restantes sólo para la función colapsar.

- `boolean tiene_ceros_en_decimal(nodo_t *n)`
  Toma un puntero a un nodo de *int*s y retorna un valor de verdad que indica si en ese nodo el número al representarse en decimal posee algun 0.
- `boolean parte_decimal_mayor_que_un_medio(nodo_t *n)`
  Toma un puntero a un nodo de *double*s y retorna un valor de verdad que indica si en ese nodo la parte fraccionaria del número es mayor que 0.5.
- `boolean tiene_numeros(nodo_t *n)`
  Toma un nodo de *string*s de C y retorna un valor de verdad que indica si en ese nodo la cadena de caracteres contiene numeros.
- `valor_e raiz_cuadrada_del_producto(valor_e valorA, valor_e valorB);`
  Toma dos valores que contienen enteros y retorna un valor que representa a la raíz cuadrada del producto valorA * valorB.
- `valor_e raiz_de_la_suma(valor_e valorA, valor_e valorB)`
  Toma dos valores que contienen doubles y retorna un valor que representa a la raíz cuadrada de la suma de valorA y valorB.
- `valor_e revolver_primeras_5(valor_e vA, valor_e vB)`
  Toma dos valores que contienen strings y retorna una **nueva** string. Esta nueva cadena tendrá longitud maxima 10 caracteres, y es el resultado de ir colocando una letra de cada string a la vez. En caso de que alguna de las dos cadenas tenga tamaño menor a 5, se completarán hasta llegar a 10 caracteres con los de la otra. Ej. Revolver de "balsamo" y "hora" da como resultado "bhaolrsamo".

2. Enunciado

A lo largo del trabajo deberá implementar todas las funciones en assembler, para un listado completo se enumeran a continuación.

- Funciones de listas y nodos
  - `lista_colgante_t *lista_crear()`
  - `nodo_t* nodo_crear(tipo_elementos tipo, valor_elemento valor)`
  - `void lista_borrar(lista_colgante_t *self)`
  - `void lista_concatenar(lista_colgante_t *self, nodo_t *siguiente)`
  - `void lista_colgar_descendiente(lista_colgante_t *self, uint posicion, nodo_t *hijo)`
  - `void lista_imprimir(lista_colgante_t *self, char *nombre_archivo);`
- Funciones avanzadas de listas
  - `void lista_filtrar(lista_colgante_t *self, nodo_bool_method test_method)`
  - `void lista_colapsar(lista_colgante_t *self, nodo_bool_method test_method, nodo_value_method join_method))`
- Las funciones adicionales antes especificadas

Para implementar todo lo anterior de una manera concisa y fácil de entender, se utilizarán funciones auxiliares. Se enumeran aquí las mínimas requeridas. Queda a criterio de cada uno como utilizarlas para implementar lo anterior llamando a éstas.

- `void nodo_borrar_con_hijos(nodo_t *self)`.
  Borra toda una columna de nodos
- `nodo_t* nodo_acceder(nodo_t *self, uint posicion)`.
  Devuelve el nodo base en la posicion posicion.
- `void nodo_concatenar(nodo_t **self, nodo_t *siguiente)`
  Concatena un puntero a puntero de nodo con otro nodo
- `nodo_t** nodo_ultimo(nodo_t **self_p)`
  Recibe un puntero a puntero de nodo y devuelve el ultimo de puntero a puntero de nodo de la cadena.
- `nodo_t *nodo_ultimo_hijo(nodo_t *self)`
  Idem anterior pero para los descendientes, usa puntero a nodo directamente.
- `void valor_imprimir(tipo_elementos tipo, valor_elemento valor, FILE *stream)`
  Imprime entre `[` y `]` en el archivo stream el valor recibido.
- `void nodo_imprimir_columna(nodo_t *self, FILE *stream)`
  Imprime entre `{` y `}` en el archivo stream la lista de los descendientes de un nodo dado.
- `void nodo_colapsar(nodo_t **self_pointer, nodo_value_method join_method)`
  Recibe un puntero a puntero a un nodo base y se encarga de colapsar todos sus descendientes.

Compilación y testeo

Para compilar el código, deberá ejecutar `make main` y `make tester` según corresponda.

Prueba corta

Deberá construir un programa de prueba (`main.c`) que realice las acciones detalladas a continuación. La idea del ejercicio es verificar manualmente que las funciones que vaya implementando funcionen correctamente. Corra el programa con

```
$> ./pruebacorta.sh
```

para verificar que no se pierde memoria ni se realizan accesos incorrectos a la misma. Recordar siempre borrar la lista luego de usarla.

1- Crear una lista nueva vacía, imprimirla y borrarla. Para esto deberá implementar las funciones `lista_crear` y `lista_imprimir` y `lista_borrar` en assembler. No es necesario que implemente completamente imprimir y borrar, ya que no utiliza nodos, puede dejar eso para el siguiente punto.

2- Crear una lista nueva y un nodo de valor entero 10. Concatenar el nodo a la lista. Deberá implementar `nodo_crear` y `lista_concatenar`, además de ahora sí completar imprimir y borrar.

3- Crear una lista nueva, un nodo de valor `"hola"` y otro de valor `"chau"`. Concatenar el primero a la lista y colgar el segundo del primero. Para esto necesitará `lista_colgar_descendiente`

Hasta aquí todas las funciones de listas fueron implementadas en assembler. Los siguientes puntos deberá realizarlos implementando filtrar y colapsar en C. Para evitar un problema de nombres, las versiones en C de las funciones deben llamarse `lista_filtrar_c` y `lista_colapsar_c`.

4- Crear una lista como la de la figura 1 y aplicarle la función filtrar (de C), pasando como parámetro la función que verifica si tiene ceros en su representacion decimal. Imprimir la lista.

5- Crear una lista como la de la figura 1 y aplicarle la función colapsar (de C), pasando como parámetro la función que verifica si tiene ceros en su representación decimal y la que calcula la raiz cuadrada del producto. Imprimir la lista.

Realizar el mismo procedimiento para una lista de nodos de tipo double y otra de strings, usando `parte_decimal_mayor_que_un_medio` / `raiz_de_la_suma` para la lista de doubles y `tiene_numeros` / `revolver_primeras_5` para la de strings.

Por último, implemente las funciones lista_filtrar y lista_colapsar en assembler, y corra los tests que se especifican en la siguiente sección.

Testing

En un ataque de bondad, hemos decidido entregarle una serie de *tests* o pruebas para que usted mismo pueda verificar el buen funcionamiento de su código de manera automática.

Luego de compilar, puede ejecutar `./tests.sh` y eso correrá todos los tests de la cátedra con su código. Además, para un testeo más especifico, puede usar `./testuno.sh [-c] -f <archivo_test>`. Un test consiste en la creación, inserción, eliminación e impresión en archivo de una gran cantidad de listas. 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.

Archivos

Se entregan los siguientes archivos:

- `main.c`: Es el archivo principal donde escribir los ejercicios del punto 1.
- `tester.cpp`, `test-o-matic.cpp/h`: Son los archivos del tester. No debe modificarlos.
- `base.h`: Definiciones de estructuras y tipos básicos.
- `lista_colgante.h`: Contiene la definición de la estructura de la lista, los elementos y las funciones que debe completar.
- `lista_colgante.asm`: Archivo a completar con su código.
- `Makefile`: Contiene las instrucciones para compilar el código.
- `tests.sh`: Es el script que corre todos los test con distintas entradas.
- `pruebacorta.sh`: Es el script que corre los tests simples.

Notas:

a) Todas las funciones deben estar en lenguaje ensamblador. Cualquier función extra, también debe estar hecha en lenguaje ensamblador.

b) Toda la memoria dinámica reservada por la función `malloc` debe ser correctamente liberada, utilizando la función `free`.

c) Para el manejo de archivos se recomienda usar las funciones de C: `fopen`, `fread`, `fwrite`, `fclose`, `fseek`, `ftell`, `fprintf`, etc.

d) Para poder correr los test, se debe tener instalado *Valgrind* (En ubuntu: `sudo apt-get install valgrind`).

e) Para corregir los TPs usaremos los mismos tests que les fueron entregados. El criterio de aprobación es que el TP supere correctamente todos los tests.

3. Resolución informe y forma de entrega

Este trabajo práctico consta carácter de **individual**. No deberán entregar un informe. En cambio, deberán entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado solo archivo `lista_colgante.asm`, y agregado un archivo `main.c` con el segundo ejercicio resuelto.

Este trabajo deberá entregarse como máximo a las 00:00 hs del día 9 de Septiembre, a través de la página web. El sistema sólo aceptará entregas de trabajos hasta las 00:00 hs del día de entrega.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.