TP Árbol De Intervalos

Cassinerio Marcos - Cerruti Lautaro

Junio 7, 2020*

*Updated June 8, 2020

1 Introducción

El objetivo de este trabajo fue implementar un árbol AVL de Intervalos. Adicionalmente implementamos un intérprete para poder utilizar el árbol y sus funciones desde la entrada estándar.

2 Compilado y ejecucion

Para compilar el proyecto abrimos una terminal, y una vez ubicados en el directorio del proyecto, ejecutamos el comando `make`. Esto nos generará el ejecutable del intérprete.

El mismo lo corremos con:

```
./interprete
```

Este programa nos permitirá ingresar comandos para manipular un árbol. Los comando aceptados son:

- `i [a, b]`: inserta el intervalo [a, b] en el árbol. Ej: `i [5.4, 6.3]`.
- `e [a, b]`: elimina el intervalo [a, b] del árbol. Ej: `e [2.2, 1e4]`.
- `? [a, b]`: interseca el intervalo [a, b] con los intervalos del árbol. Ej: `? [-1, 0.5]`.
- `dfs`: imprime los intervalos del árbol con recorrido primero en profundidad.
- `bfs`: imprime los intervalos del árbol con recorrido primero a lo ancho.
- `salir`: destruye el árbol y termina el programa.

Los primeros 3 comandos tienen que respetar el formato `caracterOperacion [a, b]`.

3 Organizacion de los archivos

El programa se divide en 4 partes: Intervalo, Árbol de Intervalos, Cola (Queue) e intérprete
Por un lado tenemos la implementación y declaración de Queue en los archivos `queue.c` y `queue.h` respectivamente.
Por otro lado tenemos Intervalo hecho de la misma manera, en los archivos `intervalo.c` y `intervalo.h`.
Luego tenemos al Árbol de Intervalos que hace uso de las dependecias queue e intervalo. Su implementación y declaración se encuentra en los archivos `itree.c` y `itree.h`.
Finalmente tenemos en el archivo `interprete.c` el intéprete que es nuestra interfaz del progragama para manipular los Árboles de Intervalos.

4 Implementaciones y estructuras

4.1 Queue

La implementación de Queue esta basada en una Lista Doblemente Enlazada Circular, definida de la siguiente forma:

```
struct _QNode {
  void *dato;
  struct _QNode *ant;
  struct _QNode *sig;
};

typedef struct _QNode QNode;

typedef QNode *Queue;
```

En su cabecera declaramos sus únicas 3 funciones:

- `queue_crear`
- `queue_pop`
- `queue_push`
- `queue_destruir`

La función `queue_destruir` nunca es utilizada, pero para que la implentación de la Cola sea general se vio la necesidad de hacerla.
Todas las implementaciones se encuentran en `queue.c`.

4.2 Intervalo

La declaración de intervalo es la siguiente:

```
struct _Intervalo{
  double extremoIzq;
  double extremoDer;
};

typedef struct _Intervalo Intervalo;
```

En su cabecera declaramos las siguientes funciones:

- `intervalo_crear`
- `intervalo_destruir`
- `intervalo_extremo_izq`
- `intervalo_extremo_der`
- `intervalo_valido`
- `intervalo_interseca`
- `intervalo_imprimir`

Sus implentaciones se encuentran en `intervalo.c`.

4.3 ITree

El Árbol de Intervalos se encuentra definido de la siguiente manera:

```
struct _INodo {
  Intervalo *intervalo;
  double maxExtremoDer;
  int altura;
  struct _INodo *izq;
  struct _INodo *der;
};

typedef struct _INodo INodo;

typedef INodo *ITree;
```

En su archivo cabecera se encuentran declaradas las siguientes funciones:

- `itree_crear`
- `itree_destruir`
- `itree_altura`
- `itree_insertar`
- `itree_eliminar`
- `itree_intersecar`
- `itree_recorrer_dfs`
- `itree_recorrer_bfs`
- `itree_aplicar_a_intervalo`

Sus implementaciones se encuentran en el archivo `itree.c`, junto con las implementaciones de las funciones:

- `itree_nuevo_nodo`
- `itree_calcular_max_extremo_der`
- `itree_calcular_altura`
- `itree_actualizar_datos`
- `rotacion_izquierda`
- `rotacion_derecha`
- `itree_balance`
- `itree_balancear`
- `itree_obtener_menor`

4.4 Interprete

El interprete se encuentra el main del programa, este se encarga de leer la entrada estandar, validarla y ejecutar las funciones de ITree e intervalo según la entrada. En este archivo estan implementadas las siguientes funciones:

- `leer_cadena`
- `obtener_operacion`

5 Desarrollo y complicaciones

Cuando empezamos con la implementacion del arbol nos dimos cuenta que para obtener la altura de un nodo podiamos crear una función que recorria el arbol y retornaba la altura del nodo o almacenar en cada nodo un nuevo campo que fuera su altura. Decidimos ir por la segunda opción ya que cada vez que quisieramos acceder a la altura de un nodo, con una funcion habia que recorrer todo el arbol, mientras que almacenandola, era de forma directa. Esto era una ventaja a la hora de calcular los balances de los nodos debido a que nos ahorraba recorrer varios subarboles.

Cuando estabamos haciendo las funciones necesarias para el árbol de intervalos necesitamos utilizar algunas funciones auxiliares, pero como no queriamos que estas fueran utilizadas por fuera de las funciones que las llamaban, decidimos no declararlas en la cabecera y de esta forma, cuando se importara la misma, no tendrían acceso a esas funciones.

Mientras trabajabamos en las funciones de Árbol de Intervalos, nos dimos cuenta que podiamos modularizar los intervalos como una estructura para facilitar el agregado, el borrado y la impresion de los mismos, y para mantener una mejor organización del código.

Luego de haber implementado las funciones basicas de intervalo, empezamos a reemplazar los 2 doubles que habiamos declarado en los ITree por Intervalos, pero allí encontramos un problema, que fue que al querer acceder a los extremos del intervalo no podiamos porque la declaracion de intervalos estaba en `intervalo.c` y nosotros importabamos `intervalo.h`. Como los extremos de los intervalos no son modificados despues de crearlo, nos parecio la mejor opcion crear 2 funciones en Intervalo para obtener cada uno de los extremos.

Luego seguimos refactorizando lo ya hecho del árbol.

Realizamos estos cambios hasta llegar a la impresion del arbol con BFS. Esta la habiamos dejado para el final por la necesidad de implementar una Cola. La hicimos en su propio modulo ya que la implementamos con un funcionamiento general.

Habiendo terminado estas 3 partes con todas sus implementaciones, nos pusimos a trabajar en el interprete. El mismo tuvo una version inicial, que parecia funcionar perfectamente pero probando algunos casos leia comandos que no eran validos.

Para facilitar algunas cosas de leer la entrada, decidimos hacer nuestra propia función para leerla, ya que queriamos que la misma devolviera el string leido.

Luego para validar la entrada, decidimos hacer una funcion, que en principio la hicimos con una implementacion que no nos convencía, ya que recorriamos todo el string caracter por caracter, validando cada uno de ellos, y habia casos compliacados de validar y hasta algunos que no nos dimos cuenta de hacerlo. Cuando habiamos terminado la version inicial, seguimos investigando sobre algunas funciones, y en especial nos llamo la atención la función `strtod` y su caracteristica de almacenar el puntero al caracter que no se pudo transformar dentro del string, y esto nos dio una idea para implementar la validación sin necesidad de recorrer todo el string. Implementamos la misma y fue la que dejamos como versión final de la validación.

6 Bibliografia

https://en.wikipedia.org/wiki/AVL_tree
https://www.geeksforgeeks.org/avl-tree-set-2-deletion/
http://www.cplusplus.com/reference/cstdio/scanf/
https://www.tutorialspoint.com/c_standard_library/c_function_gets.htm
https://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm
http://www.cplusplus.com/reference/cstdio/sscanf/
https://stackoverflow.com/questions/27973759/return-value-of-strtod-if-string-equals-to-zero
https://stackoverflow.com/questions/277772/avoid-trailing-zeroes-in-printf