Trabajo Práctico Final Julio/Agosto

Licenciatura en Ciencias de la Computación

Estructuras de datos y algoritmos I

Cátedra: Federico Severino Guimpel, Mauro Lucci, Martín Ceresa, Emilio López, Valentina Bini

Implementación de conjuntos numéricos

Tomás Scalbi

2020

Índice

1. Introducción

1.1. Motivación del trabajo

Objetivo El objetivo del trabajo es lograr implementar un intérprete de conjuntos numéricos que soporte las operaciones propuestas:

- Creación por extensión.
- Creación por compresión.
- Unión.
- Intersección.
- Complemento.
- Resta.

El interprete además soporta los siguientes comandos:

- Imprimir conjunto.
- Imprimir menu.

La correcta sintaxis de cada comando está bien descripta en el desarrollo del informe.

1.2. Estructuras de datos

Decidí implementar los conjuntos numéricos modificando un poco la estructura de los árboles de intervalose que usé en el trabajo anterior. Para el manejo de alias, la elección fue mucho más simple y me fuí directo por una tabla hash con a'rea de rebalse, donde el área de rebalse fuese un árbol AVL.

1.3. Funcionamiento

Mi programa al que llamo intérprete, toma comandos ingresados a través de la entrada estandar. Si el comando es válido, realiza la acción correspondiente, caso contrario devuelve un mensaje de error que intenta ser preciso y aportar la información que necesita el usuario para poder utilizarlo correctamente.

Para compilar el programa y ejecutarlo, hay que pararse sobre la carpeta y ejecutar el comando make en la consola, se generará un archivo al que llamé interprete.out.

Al igual que los trabajos anteriores incluí las funciones de cleanLin y cleanWin para limpiar los *.o* y *.out*.

1.4. Personal

Personalmente este trabajo fue el que más me costó de todos por el hecho de ser individual. Encontré complicaciones a la hora de elegir las estrucuturas de datos y decidir sobre distintas maneras de hacer una misma cosa. El hecho de estar comprometido con un compañero me resultaba muy motivador para trabajar.

2. Carpeta de entrega

En la carpeta de entrega se tiene un makefile principal, este archivo pdf, y las carpetas con los recursos para compilar todo el programa.

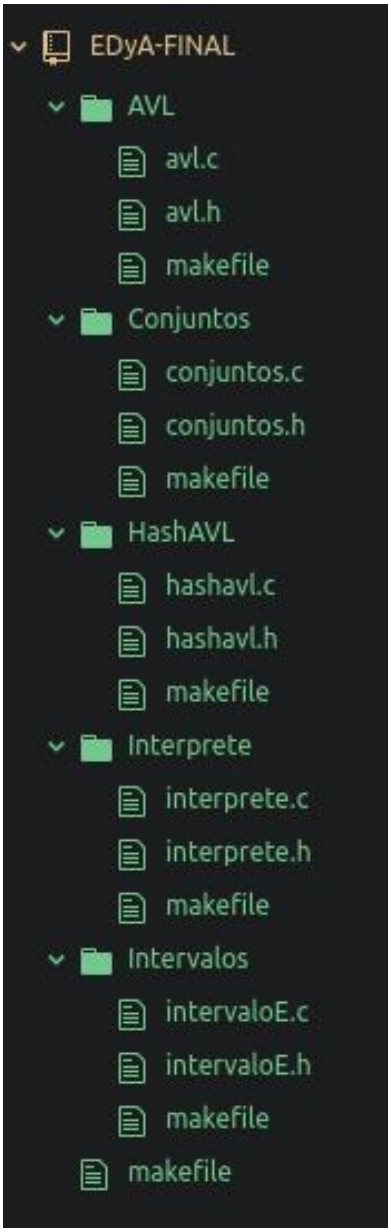

Figura 1: Carpeta de entrega

2.1. Carpetas

Cada carpeta tiene 3 archivos dentro, un makefile, un '.c'con implementaciones y un '.h'prototipos y explicaciones. Luego cuando se ejecute el comando make, se creará en cada carpeta un '.o'que se utilizará para la compilación del ejecutable final.

2.2. makefile

Cada carpeta tiene un archivo makefile que dentro tiene una sola instrucción, que es crear el '. o' correspondiente. Para este makefile hize uso de nuevas herramientas, .PHONY principalmente para ejecutar recetas comandos sin chequear sus dependencias. Y el uso de una 'lista' de directorios para ejecutar los make de cada subdirectorios.[1]

[1] `https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html`

3. Intérprete

3.1. Alias

Los alias aceptados son alfanuméricos de longitud indefinida, es decir letras de la a-Z sin tilde y digitios del 0-9. Se distingue entre mayusculas y la ñ no es aceptada.

3.2. Lectura de entrada

A diferencia del trabajo pasado que con mi compañero habíamos decidido utilizar un scanf para leer toda la entrada y parsearla. En este trabajo decidí leer la entrada manualmente caracter por caracter con 2 funciones auxiliares que creé. La motivación tras de esto fué quitar la dependencia del tamaño del buffer. Las funciones empleadas para leer la entrada son:

```
1     char* leer_entrada ();
2     char* aumentar_tamanio (char* buffer, int *tamAnterior);
3
```

Estas funciones están bien comentadas en el código, pero básicamente se lee la entrada caracter por caracter y se va almacenando en un puntero a caracteres alocado dinámicamente. Cuando la cantidad de caracteres superará el tamaño del buffer, se llama a la función aumentar_tamanio que crea un buffer nuevo con malloc, alocando el doble de tamaño y copiando los elementos del primero al nuevo.

Luego el buffer viejo se libera. También es necesario que se pase el tamaño anterior como un puntero para que, en la función leer_entrada quede actualizado el tamaño del buffer por si es necesario *agrandarlo* nuevamente.

3.3. Mensajes de error

Por como se realizó el interprete, los mensajes de error son **específicos**. Mantuve siempre la estrucutra de if−else de forma tal que los mensajes de error estén siempre en la cláusula else, o en su defecto en la cláusula default de switch.

Para manejar de forma más legible y sin repetir codigo decidí crear la función mensaje_error que dadas 2 strings, la primera siendo un identificador para el codigo de error, y la segunda información adicional para describir el error, imprime el mensaje de error adecuado.

Separé los **tipos de errores** en dos categorías, la primera son **errores de sintaxis** en el intérprete, es decir que se ingresen comandos incompletos, inválidos, etc.. Y la segunda son **errores de uso** del intérprete que solamente identifiqué uno así, el error que se genera cuando se hace referencia a un conjunto que no fue declarado anteriormente.

Los códigos estan codificados de la siguiente forma: "f123". El primer caracter es una letra que indica el tipo de error. Las letras disponibles son 'f'*errores de formato* y 'u'*errores de uso*. Luego el resto del código se transforma a int y se pasa por un switch que contendrá el mesaje a imprimir.

Debido a esta implementación, se reportará el primer error que se encuentre, aunque haya mas a lo largo de la entrada.

3.4. Comandos aceptados

Los comandos aceptados y su sintaxis es la misma que la propuesta por la cátedra. Además del comando 'menu'que imprime un menu práctico sobre los comandos disponibles. También decidí dejar la alternativa de que el nombre de la variable en la creación de conjuntos por compresión sea mas de un solo caracter, pudiendo ser la siguiente entrada valida: 'a = {???sdfasd : 5 <= ???sdfasd <= 10}'.
Los comandos disponibles cuentan con flexibilidad de espacios entre 'palabras', considerandose cada palabra separada por al menos un espacio. **Lista de comandos**, donde sea que haya un espacio en la siguiente lista, en realidad pueden haber ilimitados.

- ' salir ': salir del intérprete.
- ' imprimir A ': imprimir el conjunto A.
- ' imprimir2D A ': imprimir el conjunto A en modo árbol.
- ' A = {var : k_0 <= var <= k_1} ' : crear por compresión el conjunto A.
- ' A = {k_0,k_1,k_2,...,k_n} ': crear por extensión el conjunto A.
- ' A = B | C ': A es la unión entre B y C.
- ' A = B C ': A es la intersección entre B y C.
- ' A = B - C ': A es la resta entre B y C.
- ' A = B ': A es el complemento de B.
- ' A = B ': A es igual a B.

En todos los casos en los que se este definiendo A, si ya existía un conjunto con dicho alias, este se pisa.

3.5. Dificultades

Parseo En la primera implementación del intérprete que realizé, se utilizaba la función de string.h strtok. Tuve que cambiarme a otra función puesto que con esta última, se hizo muy complejo para modularizar ciertas funciones del intérprete. La función strtok modifica la cadena original insertando un terminador cuando encuentra el string que se quiere delimitar. El problema de esto es que si se quiere seguir obteniendo tokens en una funcion auxiliar es muy incómodo puesto que no se tiene la referencia de donde sigue la string que estamos tokenizando.

Por lo tanto buscando un poco por internet dí con la función char * strsep(char **stringp, const char *delim); de string.h. Esta función tambien modifica la cadena original pero con la diferencia que se sigue mantienendo el resto de la string tras el delimitador. Este comportamiento me permite modularizar las secciones del interprete, quedando este mucho mas entendible y legible.

Pero al cambiar a strsep me encontré con 2 problemas. El primero es que strsep no proporciona la flexibilidad de espacios que tenía antes con strtok, y el segundo es que según la versión del compilador que utilizase, me devolvia un error diciendo que strsep no esta declarada en string.h. Por lo tanto para solucionar estos problemas, realize mi propia implementacion de la funcion strsep basada en otra que encontre en internet.[2]

Implementación encontrada:

```c
#include <string.h>

char *strsep(char **stringp, const char *delim) {
  char *rv = *stringp;
  if (rv) {
    *stringp += strcspn(*stringp, delim);
    if (**stringp)
      *(*stringp)++ = '\0';
    else
      *stringp = 0; }
  return rv;
}

```

Mi implementación:

```c
char *strsep(char **cadenaParsear, const char *delim) {
  char *cadenaDevolver = *cadenaParsear;
  if (cadenaDevolver) {
    // Se adelanta el puntero cadenaParsear hasta que se encuetre con algo
    // diferente del delimitador.
    while (**cadenaParsear == delim[0]){
      ++(cadenaDevolver);
      ++(*cadenaParsear);
    }
    // Se adelanta hasta que se vuelva a encontrar con el delimitador.
    *cadenaParsear += strcspn(*cadenaParsear, delim);
    // Se chequea que el ultimo caracter sea distinto de NULL.
    if (**cadenaParsear != '\0') {
      // Se corta alli la cadena.
      **cadenaParsear = '\0';
      // Se adelanta uno.
      ++(*cadenaParsear);
      while (**cadenaParsear == delim[0])
        ++(*cadenaParsear);
    }
    else
        *cadenaParsear = NULL;

    if (strcmp(cadenaDevolver, "") == 0)
      cadenaDevolver = NULL;
  }

  return cadenaDevolver;
}

```

[2] https://stackoverflow.com/questions/58244300/getting-the-error-undefined-reference-to-strsep-with-clang-and-mingw

4. Conjuntos

4.1. Elección de estructura de datos

Árbol de intervalos disjuntos Como dije en la introducción, la estructura de datos que decidí implementar fue la de árbol de intervalos disjuntos. Al tener que elegir una estructura para implementar conjuntos numéricos, lo primero que se me vino a la cabeza fue la estrucutura del trabajo anterior, la de **Árboles de Intervalos**. La diferencia principal con la estructura anterior es que ahora al tener que implementar conjuntos, no tendría sentido almacenar intervalos que colisionen, por lo tanto los intervalos del árbol deberían ser disjuntos.
Estuve algunos días decidiendo si implementar esta estructura o bien ir con un listas, pero realmente me gustó más la idea de que un conjunto sea un árbol y fue la que más sentido tuvo en un comienzo para mi.
Un árbol de intervalos disjuntos representa un conjunto numérico. La declaración de la estructura, como está en el archivo de cabecera de la implementacion de conjuntos es la siguiente:

```
/**
 * Estructura que representa cada nodo (o subarbol) de un arbol de intervalos.
 * Cada nodo tiene:
 *  - un intervalo.
 *  - un int que representa la altura de ese arbol.
 *  - 2 apuntadores a 2 nodos que representan los hijos del subarbol actual
 */

typedef struct _ITreeNodo {
  IntervaloE intervalo;
  int altura;
  struct _ITreeNodo *left;
  struct _ITreeNodo *right;
} ITreeNodo;

/**
 * Estructura de ITree.
 * El ITree (o arbol de intervalos) es un tipo de AVLTree que trabaja
 * especificamente con intervalos disjuntos.
 */
typedef ITreeNodo *ITree;

```

4.2. Desarrollo y dificultades

Muchas de las funciones de AVLTree ya las tenía hechas del trabajo anterior y para no extender el informe no voy a incluirlas, de todas formas estas están bien comentadas en los archivos. Lo único que modifiqué de la estructura del trabajo anterior fue quitar el campo maximoExtremoDerecho de cada nodo puesto que no era necesario. Este campo se usaba para la función de intersección con un intervalo, pero ahora que el árbol es de intervalos disjuntos, es más simple buscarla.

Inserción La inserción en este tipo de árboles ahora debería ser un poco diferente puesto que los intervalos del árbol deben ser disjuntos. Para lograr esto creé una nueva función:

```
void itree_insertar (ITree *arbol, IntervaloE);

```

Esta elimina todas las colisiones del árbol con el intervalo pasado como parámetro y luego utiliza la vieja función de inserción para insertar en ese árbol modificado, el intervalo que contiene la unión de todos los intervalos con los que se intersecó.
También fue interesante ver, la idea de que los intervalos por ejemplo [1,2] y [3,4] se puedan unir en uno solo el [1,4]. Esta funcionalidad está bien comentada en el código.

Intersecar Para el funcionamiento de itree_insertar fue necesario una función de intersección:

```
ITree itree_intersecar (ITree, IntervaloE);

```

Esta función busca si dentro del árbol hay por lo menos un intervalo que interseque con el intervalo parámetro. En caso de ser así devuelve un subarbol cuyo primer nodo interseca con el intervalo parámetro. En caso contrario devuelve NULL.
Era necesario que esta interseccion se haga de a uno puesto que de esta forma se puede ir actualizando el intervalo que acumula las colisiones y eliminando los nodos con intervalos colisionados de forma correcta manteniendo la propiedad de árboles AVL.

Union Para realizar la unión entre dos conjuntos, es decir entre 2 árboles de intervalos disjuntos, empleé dos funciones. La primera:

```
ITree itree_unir (ITree arbol1, ITree arbol2);

```

Es la encargada de hacer la unión, primero chequea los casos trviales de vacío/universo. Luego si el caso es no trivial, realiza una copia de forma efectiva con la función que llame itree_copiar del árbol más alto y se llama a la segunda función:

```
void itree_dfs_union (ITree arbol, ITree *arbolU);

```

Que tomará el árbol más alto como *arbulU*, el árbol más bajo como el *arbol* e insertará cada nodo de *arbol* en *arbolU* con la función de inserción explicada más arriba, de esta manera el conjunto resultado *arbolU* será la unión de ambos árboles.

Complemento Para la función de complemento de un conjunto, empleé 2 funciones. La primera:

```
ITree itree_complemento (ITree origen);

```

Esta función es la encargada de obtener el complemento, primero se chequean los casos triviales que están bien comentados en el código. Luego, en el caso de que el conjunto sea no vacío ni el universo...
Me interesa explicar un poco esta sección:

```
// Se sabe con seguridad que el arbol origen es != universo y != vacio.
IntervaloE ant = intervaloE_crear (INT_MIN, INT_MIN);
// Le doy este valor al intervalo anterior a sabiendas que la funcion
// recursiva debajo, al ser inorder, comenzara por el nodo mas a la
// izquierda.
itree_complemento_aux(origen, &nuevoArbol, &ant);
// Ahora ant contiene el valor del ultimo intervalo.
if (ant.extDer != INT_MAX)
    itree_insercion (&nuevoArbol,intervaloE_crear(ant.extDer + 1,INT_MAX));

```

Se crea un intervalo auxiliar al que llamo 'ant' que comenzará con el valor del mínimo int. Este valor será pasado a la función auxiliar:

```
void itree_complemento_aux (ITree origen, ITree *destino, IntervaloE *ant);

```

La idea con la que funciona este función es que 'ant' sea el intervalo anterior al nodo que se está analizando, puesto que el complemento se puede calcular con el extremo derecho del anterior, y el extremo izquierdo del actual. Entonces esta función estaría *arrastrando* el valor del intervalo anterior a medida que la recorre, creando los intervalos que componen el conjunto complemento, e insertándolos en el árbol resultado.
Ademas de esta forma, al finalizar la función recursiva, como a itree_complemento_aux se le pasó la dirección de memoria del intervalo 'ant', ahora este contendrá el valor del último intervalo, por lo tanto se puede calcular su relación con el ma'ximo int y así determinar si hay que agregar o no, un último intervalo al conjunto resultado.

itree_dfs_origen_destino Considero necesario antes de explicar como implementé las siguientes 2 operaciones, la función auxiliar que estas emplean:

```
void itree_dfs_origen_destino (ITree origen1, ITree origen2, FuncionAux aux,
                               ITree *destino){
                               if (!itree_es_vacio (origen1)){
    itree_dfs_origen_destino (origen1->left, origen2, aux, destino);
    aux(origen2, origen1->intervalo, destino);
    itree_dfs_origen_destino (origen1->right, origen2, aux, destino);
  }
}

```

Donde FuncionAux es del tipo:

```
typedef void (*FuncionAux) (ITree arbol2, IntervaloE intervalo, ITree *destino);

```

Se toman dos árboles considerados origen1 y origen2. Una función auxiliar que tomará el ÁRBOL origen2 y lo recorrerá realizando comparaciones con un INTERVALO (perteneciente a el ÁRBOL origen1) y almacenando los resultados en el ÁRBOL destino. Y un árbol destino que será el que almacene el resultado final.
La función recorre el árbol origen1 y aplica la función auxiliar con cada intervalo de este, al árbol origen2, y almacenando los resultados en el árbol destino.

Interseccion Para la función de intersección entre 2 conjuntos, es decir entre 2 árboles de intervalos disjuntos, también se emplean 3 funciones.

```
ITree itree_interseccion (ITree arbol1, ITree arbol2);
void itree_dfs_origen_destino (ITree origen1, ITree origen2, FuncionAux aux,
                               ITree *destino);
void itree_intersecarV (ITree arbolAInt, IntervaloE intervalo, ITree *arbolRes);

```

La primera función es la encargada de realizar la intersección, primeramente chequea los casos triviales, que están bien comentados en el código. Luego una vez establecido que ninguno de los conjuntos es vacío ni el universo, se pasa a determinar cuál es el árbol más alto y cual es el más bajo.
Luego se llama a itree_dfs_origen_destino para que recorra el árbol más bajo, aplicando la función itree_intersecarV con cada intervalo de este, a el árbol más alto, y almacenando los resultados en el árbol destino.
itree_intersecarV, logra que el árbol destino termine siendo un acumulador de intersecciones entre un árbol y un intervalo. En esta implementación, se está recorriendo uno de los árboles(arbolMasBajo) con itree_dfs_origen_destino, aplicando itree_intersecarV con cada nodo de este (arbolMasBajo) y el otro arbol (arbolMasAlto).
De esta forma el árbol resultado termina acumulando la intersección entre ambos árboles.

Resta La implementación de la resta entre 2 conjuntos es algo similar a la de intersección. Para lograrlo se emplean también 3 funciones.

```
ITree itree_resta (ITree arbol1, ITree arbol2);
void itree_dfs_origen_destino (ITree origen1, ITree origen2, FuncionAux aux,
                               ITree *destino);
void itree_resta_aux (ITree arbol1, IntervaloE intervalo, ITree *destino);

```

La primera es la encargada de realizar la operación, primeramente chequea que no se trate de los casos trivales y de ser así, los soluciona rápidamente, estos casos están bien comentados en el codigo.
Luego, de no ser así se llama a itree_dfs_origen_destino para que recorra el primer árbol, aplicando la función itree_resta_aux con cada intervalo de este, al segundo árbol, almacenando los resultados en el árbol destino.
La idea con la que implementé **itree_resta_aux** es bastante visual:

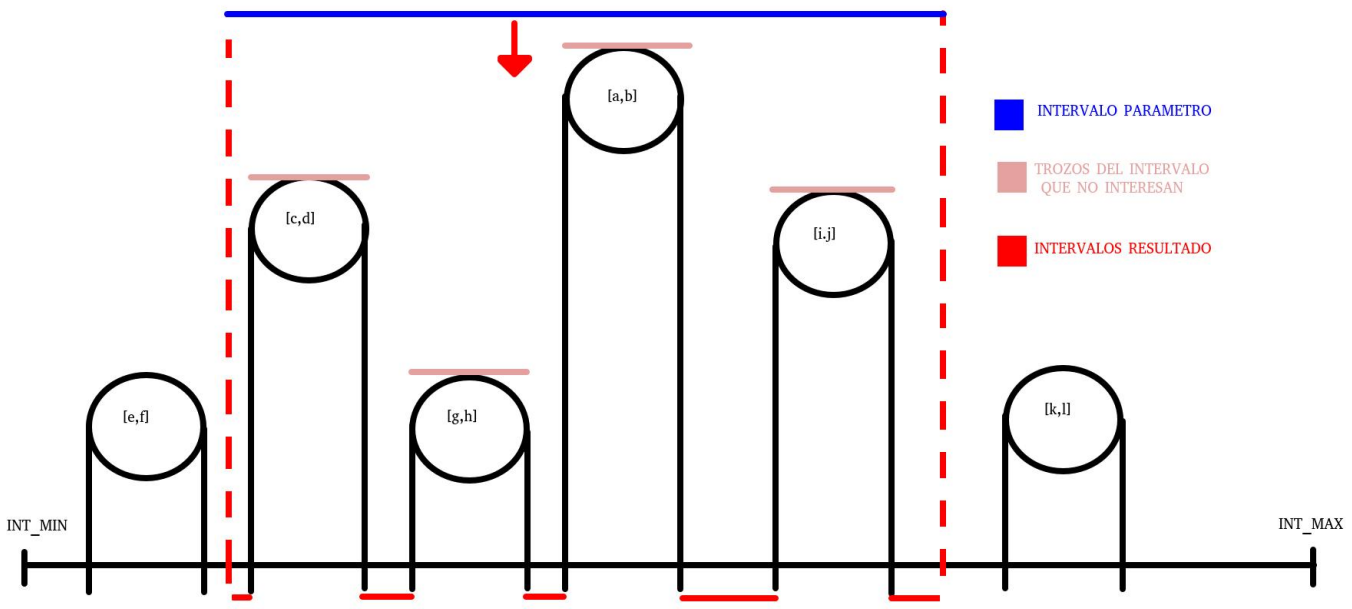

Figura 2: Representacion visual de itree_resta_aux

Todos los intervalos resultado se insertan en el árbol resultado. Y se ejecuta esta función con cada intervalo del primer árbol sobre el segundo.

5. Tabla Hash, manejo de alias

5.1. Elección de la estructura de datos

Tabla Hash En este caso, la elección de la estructura de datos fue más directa, de primeras quedó claro que se trataría de una tabla hash cuyas claves serían los alias y los datos serían conjuntos.

```
typedef struct {
  AVLClavesTree *tabla;
  unsigned numElems;
  unsigned capacidad;
  FuncionHash hash;
} TablaHash;

```

Área de rebalse Para manejar las colisiones elegí el método de área de rebalse, donde el área de rebalse es un arbol AVL. De esta manera de haber muchas colisiones, el tiempo de búsqueda se ve mejorado frente a si el área de rebalse fuese de listas. Elegí este método frente a direccionamiento abierto porque fué con el que más seguro me sentí, aunque según entiendo, para este caso direccionamiento abierto sería más efectivo puesto que nunca hay que realizar eliminaciones en la tabla hash, y este es su eslabón débil.

```
typedef struct _AVLClavesNodo {
  char* clave;
  ITree conjunto;
  int altura;
  struct _AVLClavesNodo *left;
  struct _AVLClavesNodo *right;
} AVLClavesNodo;

typedef AVLClavesNodo *AVLClavesTree;

```

5.2. Desarrollo y dificultades

Para realizar las funciones de tabla hash, me base en el archivo que está subido en el campus como ejemplo. Decidí no implementar la tabla con punteros void y hacerla específica para mi caso.

Funcion de hasheo Para la función de hasheo, el problema era hashear las claves, que son strings. Para esto primeramente se me ocurrió sumar los asciis de los caracteres de la clave. Pero me dí cuenta que es una función de hasheo bastante mala puesto que las claves que tengan los mismos caracteres tendrían el mismo id y habría muchas colisiones.
Por lo tanto habría que hacer que un caracter en la primer posición tenga un valor distinto que ese mismo caracter pero en otra posición. Teniendo esto en mente, buscando por internet me encontré con un post sobre este mismo tema [3], en el que una de las respuestas propone la siguiente función:

```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}

```

La cual adapté a:

```
unsigned int hasheo(char *alias, unsigned capacidad) {
  // Funcion de hash de Dan Bernstein.
  // Declarado como unsigned para que sea siempre positivo.
  unsigned int hash = 5381;

  for(int i = 0; alias[i]; i++)
    hash = (33 * hash) + alias[i];
  return hash %capacidad;
}

```

[3] https://stackoverflow.com/questions/7666509/hash-function-for-string

Y se puede ver que se está teniendo en cuenta el problema que había surgido cuando solo se suman los valores ascii. Además al multiplicar por números progresivamente grandes, se alcanza el overflow de int rápidamente resultando en una distribución mas homogenea.

AVLTree para área de rebalse Primeramente había implementado el área de rebalse con listas simplemente enlazadas, por lo tanto ya tenía una lógica planeada para esta implementación. Las funciones de AVLTree en su mayoría estaban implementadas del trabajo pasado y para no extender el informe no voy a meterme en ellas.

Las funciones más importantes y nuevas que valen la pena explicar son 2, la primera:

```
void avlClavesTree_insertar (AVLClavesTree *arbolClaves, ITree conjunto,
                             char* clave);

```

El cambio principal de esta función con respecto a la función de inserción realizada en el trabajo anterior, además de que la información que lleva el nodo es diferente, es que si se encuentra un nodo con la clave que se está tratando de insertar, se libera el conjunto que estaba allí antes y se coloca el nuevo conjunto en su lugar. De esta forma, insertar estaría siempre pisando los valores anteriores, de esta manera es que tiene sentido la reasignación de conjuntos en el intérprete.

La segunda función, es la función de busqueda:

```
Contenedor avlClavesTree_buscar (AVLClavesTree arbol, char* clave);

```

Como se puede ver, devuelve un tipo de dato que hasta ahora no nombré, y esto es porque es un tipo de dato que se usa solo para referenciar conjuntos de la tabla hash.

```
typedef struct Contenedor {
  ITree conjunto;
} *Contenedor;

```

Este tipo de dato surge para resolver la ambigüedad que surge entre conjunto vacío y un valor que no está en la tabla hash. El conjunto vacío es un árbol vacío es decir, NULL, y una casilla vacía de la tabla hash también es NULL, por lo tanto cuando se busque un conjunto, esta función no podría simplemente retornar un apuntador al conjunto puesto que si este fuese vacío, se confundiría con una casilla vacía. Por lo tanto, la función para buscar en una casilla retorna un Contenedor, de esta forma cuando lo que llega de la función es NULL se sabe ciertamente que la casilla esta vacía, es decir que no hay ningún conjunto con el alias buscado.

Crear Para crear la tabla hash, primero se pide memoria para la estructura principal y cada una de sus casillas, luego se inicializa cada casilla como un *AvlClavesTree* vacío, y finalmente se retorna la tabla.

Inserción Para insertar en la tabla hash, primero se calcula con la función de hasheo el id asociado a la clave. Luego se corrobora si existe algún conjunto en esa casilla, si no hay ninguno, significa que se ocupará una nueva casilla por lo tanto se aumenta en 1 el número de elementos en la tabla.
Luego simplemente se inserta en la estructura de área de rebalse.

Búsqueda Primero se hashea la clave y se obtiene el número de casilla asociada, y luego simplemente retorna lo que retorne la función anteriormente explicada avlClavesTree_buscar. De esta forma se puede discernir entre un conjunto vacío y una casilla vacía.

Destruir Para destruir específicamente esta tabla, se recorre primero el array de casillas hasta que no queden elementos utilizados en la tabla. Cuando se encuentra con un AVLClavesTree diferente de vacío, se destruye y se cambia su valor a NULL.
Luego una vez liberadas todas las casillas, se procede a liberar el arreglo y finalmente la tabla.

6. Bibliografía

Referencias

[1] **Geeksforgeeks** *Imprimir árbol binario en horizontal*, Consultado en Mayo-Junio 2020
`https://www.geeksforgeeks.org/print-binary-tree-2-dimensions/`

[2] **Geeksforgeeks** *Eliminación en AVLTree*, Consultado en Mayo-Junio 2020
`https://www.geeksforgeeks.org/avl-tree-set-2-deletion/?ref=lbp`

[3] **GNU** *PHONY targets*
`https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html`

[4] **StackOverflow** *strsep*
`https://stackoverflow.com/questions/58244300/getting-the-error-undefined-reference-to-strsep-with-clang-an`

[5] **StackOverflow** *hash function for string*
`https://stackoverflow.com/questions/7666509/hash-function-for-string`