Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Estructura de Datos y Algoritmo I

Trabajo Integrador Final y Ejercicios Adicionales

Alumno:
Franco I. Vallejos Vigier

Legajo: 2952/1

Julio de 2020

Introducción

En este trabajo fue de suma importancia tener una idea clara y concisa de los objetivos planteados por el enunciado provisto por la cátedra.Desde un prematuro comienzo había que tener una división de objetivos, ya sea, el desarrollo de una interfaz, la correcta elección e implementación de diversas estructuras de datos que en estrecha interrelación con una retroalimentación de errores conformarían un amigable intérprete para el usuario.En esta interrelación aceitada sería primordial, antes que nada, plantear un marco metodológico el cual empezaría con un fuerte arraigo teórico orientado hacia las estructuras de datos estudiadas en la materia. Fundamentalmente, en mi proyecto, la estructura Hash y las listas simplemente enlazadas cuyo por qué de elección estará ricamente justificada en un apartado propio.
Luego de poder absorber este marco teórico, he planteado un plan de acción en el cual desde un primer momento trabajaríamos sobre una sintaxis prematura, muy arraigada a la consigna, demasiado sensible al contexto y sin retroalimentación de errores. Para luego pasar a una etapa donde optariamos por una manera eficaz tanto como eficiente de almacenar y recuperar los datos para operar con ellos. En cuanto a las operaciones me incliné por empezar en los casos bases de las mismas y que se cumplan sus respectivas propiedades. Por ejemplo en el caso de la operación unión y de la operación intersección la propiedad idempotencia.

Contents

PART I

Interfaz

1.1 Desarrollo de la interfaz

Desde un comienzo del proyecto y utilizando una sintaxis o una entrada de datos muy endeble como para poder pasar a otro estadío del proyecto y ocuparme del resguardo como de la recuperación de los datos. En esta interfaz hacía solamente uso de la función 'strstr' como se adjunta en el ejemplo.

```
int comando_int (char* operacion, char* alias){
 if(strstr(alias,"imprimir "))
  return 7;
 if(strstr(operacion,",") || strcmp(operacion, " {}") == 0 ||
     conjunto_un_elem(operacion) == 0){
  return 1;
 }
 if(strstr(operacion, " : "))
  return 2;
 if(strstr(operacion," | "))
  return 3;
 if(strstr(operacion," & "))
  return 4;
 if(strstr(operacion," - "))
  return 5;
 if(strstr(operacion,"~"))
  return 6;
 if(strstr(alias,"salir"))
  return 8;
 return 9;
}
```

Es loco pensar que una interfaz se puede sostener solamente con esta función y con un simple mensaje de error de sintaxis si en la entrada no se encontraba la subcadena que definía a cada operación (por ejemplo la subcadena " & " es indispensable para la definición de la operación intersección). Por eso remarco que esta primitiva interfaz era muy permisiva y permitía casos por el cual el programa seguía corriendo de una manera errónea sin importar si estaba bien la entrada.
Una vez terminada la maqueta del proyecto donde había una primera funcionalidad, decidí abocarme a abatir este problema. Dónde las correcciones más problemáticas y minuciosas tendrían que darse en la sintaxis de la definición del conjunto por extensión y en la definición por comprensión. Dado que estas definiciones son ricas en contenido ya sea porque los números pueden exceder el overflow permitido por el rango de los enteros, problemas de llaves, que las cadenas que conforman los números contengan otro carácter no numérico por ejemplo "2412A2", problemas con la coincidencia de los nombres de las variables llámese 'alias', problemas con separadores llámese ',' o '<=', entre otras...
Para el desarrollo de una nueva interfaz fue esencial el pre-análisis de los datos ingresados. En las operaciones unión, intersección, resta y complemento fue relativamente fácil detectar errores de sintaxis ya que para mí estas operaciones son del tipo:
AliasC = AliasA cadena AliasB

Explicando un poco AliasC es dónde se deposita el resultado de la operacion definida por la cadena (“ | “, “ & “, “ - “, “∼” respectivamente) que toma como operandos a AliasA y AliasB. Los posibles errores en estas operaciones son del tipo:

Tipo A:

Indefinición de conjuntos, AliasA y/o AliasB no definido con anterioridad.

Solución:

El problema se soluciona simplemente con el hecho de buscar en nuestra estructura de datos donde almacenamos nuestros operandos el AliasX de los mismos y si obtenemos un resultado no nulo a esta búsqueda en ambos casos se puede decir que nuestros operadores están listos para llevar a cabo alguna operación.

Tipo B:

Subcadena definidora inválida, que la cadena que define la operación no se encuentre entre las anteriores mencionadas.

Solución:

El problema se soluciona de una manera similar al problema tipo A, solamente comparamos nuestra subcadena definidora(cadena) con las anteriormente mencionadas y en el caso de que nuestra búsqueda sea fructífera llevamos a cabo la operación, en caso negativo notificamos al usuario del inconveniente.

Ahora, retomando las operaciones dónde se hacen definiciones de conjuntos encaremos la definición por extención. Recordando su sintaxis definida por los casos
Alias = {x1,x2,...,xn}
Alias = {}
Sus habituales problemas pueden ser:

Tipo C:

Problemas de llaves, este problema no solo deriva en que no se abra con el cáracter ‘{‘ o se cierre correctamente con su opuesto ‘}’ la declaración del conjunto, independientemente si sea vacío o no. Sino también en qué otra cadena puede ocupar el lugar de estas llaves. Ejemplo : Alias = cadenaA1,2,3,4cadenaB (con cadenaA distinto a ‘{‘ y cadenaB distinto a ‘}’

Solución:

El problema es fácilmente evitable, dado que si analizamos una cadena con una correcta definición de un conjunto bajo esta declaración. Las llaves siempre van a tener la misma posición con respecto al largo de la cadena, es decir la primera llave va a tener la posición de carácter 1 y la segunda va a mantener la posición en el largo de la cadena.

Tipo D:

Problemas de comas múltiples, al usar el carácter ‘,’ como divisorio entre cada número perteneciente al conjunto es factible pensar en un error como el siguiente. Ejemplo:
Alias = {1„2,3,5,-12„}

Solución:

Dado que en la función para obtener las cadenas que representan los números uso la función strtok() con carácter divisorio ‘,’ este problema es fácilmente solucionado porque cuando encuentra dos comas juntas la cadena fraccionada resultante es nula. Por lo cual esta cadena no es transformada en un número entero entre los límites.

Tipo E:

Problemas de mal ingreso de números, obviamente en una primera instancia al procesar nuestros números que integran nuestro conjunto estos están representados con una cadena de carácteres. La misma no es impoluta y puede contener otros caracteres que no sean dígitos. Ejemplo:
Alias = {2,3ka2,as1,4}

Solución:

Estos problemas son curiosos en sí mismos. Pero evitables con el simple hecho de que una vez que nosotros tenemos nuestra supuesta cadena que representa un número entero la tenemos que recorrer validando que cada carácter que la conforma sea un dígito de [0-9], y en el caso de los números negativos que el símbolo ‘-’ encabece la cadena.

Tipo F:

Problemas de overflow en los números, obviamente al transformar nuestra cadena previamente corregida está va a representar un número dentro del rango del tipo de dato entero y cualquier cadena que represente un número por fuera de este rango hace que el conjunto definido sea inválido.
Rango int: [-2147483648, 2147483647]
Ejemplo:
Alias = {1,2,5,-12,2200000000}

Solución:

Para los problemas tipo F, primero hay que hacerse una pregunta ¿Cómo poder decir que un número A supera a un número B, siendo que B es el límite tolerado?. He decidido usar un tipo de variable numérica que abarque en su totalidad el intervalo de las variables enteras del tipo int, este tipo de variable para evitar problemas es long long. Por lo cual me es mucho más fácil comparar si un número excede el límite de los enteros(inferior o superior) con una sola comparación usando los límites proporcionados por la librería limits.h

¿Por qué hice énfasis primero en los problemas presentes en la definición por extensión de un conjunto? Porque ahora me es mucho más sencillo explicar los posibles errores de sintaxis en la definición por comprensión dado que comparten los errores Tipo C, E y F. Refresquemos un poco la sintaxis de esta última definición...
Alias = {var : limiteinf <= var <= limitesup}
Sumado a los errores previamente mencionados a estos tres se le suman los tipo:

Tipo G:

Problemas de coincidencia de variables, este problema se da porque la declaración de las variables definidas para el conjunto no son coincidentes. Es decir sus nombres(var) no son iguales.

Tipo H:

Problemas de signos, como en la sintaxis estrictamente solo hace uso de los signos '<='. No se admite otra cadena que la suplante o su mera ausencia.

Solución G y H:

Para los problemas G y H, recaemos nuevamente en la gran funcionalidad presente en la función strtok(), ya que solo basta con dividir o fraccionar nuestra declaración del conjunto usando como separador el caracter ' ' y comparar la igualdad de los nombres de variables(var) y los signos.

Tipo I:

Problema de nulidad de definición, por pedido estricto de la consigna, si el limite superior de la variable(limitesup) es menor al límite inferior(limiteinf) el conjunto definido es considerado nulo.

Solución:

Solo es necesario una comparacion entre los límites en la definición.

Como dato de color, cabe destacar que el orden de estos errores no es azaroso. Sino representa un orden de prioridad en la resolución de warnings en mí sistema de errores en 'cascada' que le brinda un detallado informe al usuario sobre entradas problemáticas, el cual detallaré en el apartado de Retroalimentación de Errores(ver 1.4).

1.2 Formatos de alias

Dado que desde un comienzo a mí input lo fracciono mediante el carácter '=' entonces obtengo dos variables:
char* alias;
char* operacion;
Dónde la primera va almacenar el lado izquierdo de nuestro input y la segunda el derecho. Tomando como referencia el carácter central '='.
Entonces me es muy sencillo crear un Conjunto cuyo Alias sea el ingresado,

sólo es una cuestión de asignación y de asegurarse que la o las funciones Key que van a aplicar una función hash a nuestro alias lo haga correctamente. Dado esto tanto los alias de los conjuntos como la declaración de variables aceptan con fidelidad palabras compuestas por la concatenación de los siguientes abecedarios [A-Z],[a-z],[0,9]. Haciendo la salvedad obvia de la cadena vacía o nula.
Ejemplos:
Abb - AbbC - AAAA - aaaa - A9 - a0 - 100

1.3 Sintaxis

La sintaxis empleada en mí programa no dista de la pedida en la consigna del trabajo práctico respetando puntos, comas y referencias a alias. Dando a entender al usuario cuando éste hace un input erroneo.

Declaraciones

Definición de conjunto por extensión

Alias = {x1,x2,...,xn} con Alias como una cadena arbitraria especificada en la sección 2.2 Formatos de alias, dónde x1 a xn son cadenas(char*) que representan números en el rango entero.
Alias = {} denota al conjunto vacio.

Definición de conjunto por comprensión

Alias = {var : limiteinf <= var <= limitesup} con Alias tanto como var cadenas arbitrarias soportadas por el programa, con limiteinf y limitesup cadenas númericas dónde los números que estas representan se encuentran en el rango de los enteros obviamente respetando la transitividad. Si limiteinf > limitesup es otra forma de denotar al conjunto vacío.

Definición de la operación Unión

Alias = AliasA | AliasB con Alias una cadena arbitraria bajo el regimen de soporte, y AliasA como AliasB declaraciones previas a conjuntos ya definidos. (*).

Definición de la operación Intersección

Alias = AliasA & AliasB con aclaraciones idénticas (ver *).

Definición de la operación Resta

Alias = AliasA - AliasB con aclaraciones idénticas (ver *).

Definición de la operación Complemento

Alias = ~AliasA con aclaraciones idénticas (ver *) con la salvedad que en esta definición solo entra en juego un único operando con definición previa.

1.4 Retroalimentación de errores

Retomando la funcionalidad de mi programa contra errores es esencial explicar la manera de notificación cuando se está enfrente de un input erróneo. Haciendo énfasis en las operaciones de definición de conjuntos tanto por extensión como por comprensión ya que son las que presentan, por lo visto anteriormente, mayor cantidad de errores posibles. Obviamente sin subestimar a las restantes declaraciones, pero dado que son del tipo: AliasC = AliasA cadena AliasB, con las consideraciones (ver *). Pueden presentar muy escasos errores y en el caso de tenerlos se solucionan fácilmente contrastando el input ingresado contra la tabla de sintaxis(ver 1.3) esto no quita el hecho de que el usuario va a recibir un mensaje de error notificando el mismo.
Volviendo a las definiciones de los conjuntos previamente, en la sección 1.1 Desarrollo de Interfaz, hicimos una breve introducción a este sistema de retroalimentación adjudicándole el nombre de sistema "cascada". Explicando brevemente cómo trabaja este sistema solamente se resume en los siguiente. Dada una cadena errónea con uno o más errores, siempre el primer mensaje o notificación de error va a ser el de más alta jerarquía implicando que si este error se soluciona puede generar un efecto cascada solucionando los errores más abajo en la jerarquía pero no así de manera inversa. Con un ejemplo sería más fácil su comprensión, veámoslo.
Ejemplos en definición por extensión:

```
Input:
AliasA = {2
Output:
Error del tipo {x1,x2,...,xn'l', con l distinto a '}'
```

Se aprecia que es el único error en la sintaxis.

```
Input:
AliasA = {hola}
Output:
Error del tipo {x1,'x',...,xn} con x distinto a un número entero
```

Nuevamente es el único error gramátical.

```
Input:
AliasA = 2
Output:
Error del tipo 'l'x1,x2,...,xn}, con l distinto a '{'
Error del tipo {x1,x2,...,xn'l', con l distinto a '}'
Error del tipo {x1,'x',...,xn} con x distinto a un número entero
```

A esto me refiero cuando hablo de un sistema de errores cascada puesto que el carácter "2" es una cadena que representa fielmente un número pero dado que por el parseo y la ausencia de llaves esta cadena no puede ser representada como tal ya que hay errores que están más alto en la jerarquía. Ahora bien si solamente corregiríamos los dos primeros errores el tercer error también automáticamente estaría corregido, este es el efecto "cascada" mencionado.
Remitámonos también a un ejemplo de la retroalimentación en la definición por comprensión de un conjunto:

```
Input:
AliasA = { : -1 <= <= 2}
Output:
Error de sintaxis del tipo {‘alias1’ : x1 <= ‘alias2’ <= x2}
Error de sintaxis del tipo {alias : x1 ‘x’ alias ‘y’ x2} con
x e y distintos a ‘<=’
Error de sintaxis del tipo {alias : ‘x’ <= alias <= ‘y’} con
x e y no números enteros
```

De una manera similar a lo explicado en el tercer ejemplo sumado a que la cadena nula no es soportada en mi programa (véase 2.1) El problema más importante sería la inexistencia de la declaración del nombre de las variables puesto solucionado esto se podría parsear con normalidad la cadena omitiendo los errores siguientes y así definir el conjunto. Cabe aclarar que cada error definido en la sección 2.1 posee un detallado mensaje de retroalimentación, a excepción de los errores Tipo B que solo están acompañados por un simple mensaje de "ERROR DE SINTAXIS" textual.
En la siguiente página se adjunta una tabla que relaciona los errores con su devolución. Recordemos que los errores siguen una jerarquía.

Error	Retroalimentación
A	” ‘Alias‘ NO EXISTE COMO OPERANDO”.
B	“ERROR DE SINTAXIS”
C	"Error del tipo ‘l‘x1,x2,...,xn}, con l distinto a ‘{’" "Error del tipo {x1,x2,...,xn’l’, con l distinto a ‘}’" "Error de sintaxis en la primera llave de definicion ’{’alias : x1 <= alias <= x2}" "Error de sintaxis en la segunda llave de definicion {alias : x1 <= alias <= x2’}’".
D	Las comas múltiples se solucionan automáticamente.
E	"Error del tipo x1,’x’,...,xn con x distinto a un número entero" "Error de sintaxis del tipo alias : ‘x’ <= alias <= ‘y’ con x e y no numeros enteros".
F	"Error del tipo x1,’x’,...,xn, con x fuera de rango INT" "Error de sintaxis del tipo alias : ’x’ <= alias <= y con x fuera de rango INT" "Error de sintaxis del tipo alias : x <= alias <= ’y’ con y fuera de rango INT".
G	"Error de sintaxis del tipo ’alias1’ : x1 <= ’alias2’ <= x2".
H	"Error de sintaxis del tipo alias : x1 ’x’ alias ’y’ x2 con x e y distinto a ’<=’".
I	"VACIO POR COMPRENSION".No es error puesto que se sigue operando.

PART II

Estructura de Datos y Dificultades

2.1 Estructura de almacenamiento troncal Hash

El primer tema a abordar en esta sección es, ¿Cuál estructura de datos adoptar que nos permita un rápido acceso y rápido ingreso de información?, obviamente contemplando el hecho de una reiterada y extensa utilización del programa. Analizando la estructura de árbol dada en la materia, podríamos haber utilizado una función que dado un alias nos estratifique un árbol binario de búsqueda. Esta opción fue rápidamente descartada por el simple hecho de que dado un sucesivo y reiterado uso de las declaraciones de conjuntos el ingreso de datos a nuestra estructura sería muy costoso. Pero en algo estábamos en lo correcto necesitábamos una función que transforme a nuestro alias en un número, en un costo, en un índice y así poder indexarlo a una estructura de datos. De una manera sencilla y poco problemática he decidido implementar una estructura Hash a través de una lista simplemente enlazada(como área de rebalse).Pero ahora me encontraba en la disyuntiva de encontrar una función Hash que provoque el menor número de colisiones posibles. Pero antes de todo esto había que seleccionar u optar por un largo para nuestra tabla, dado que entre el '0' y la 'z' cuyos códigos ascii son el 48 y el 122 respectivamente hay un intervalo de 74 carácteres. Por picardía a esta cantidad la divido por 2 obteniendo un número primo el 37(que nos ayudará más tarde cuando implementemos una función Hash por división). Ahora recapitulemos un poco y veamos lo que es una función Hash por división, recordemos que son las funciones del tipo h(k) = k mod M, dónde M es el tamaño de la tabla. Analicemos un poco este número M y su importancia, si M fuera par todas las claves pares serían asignadas o indexadas a localizaciones pares, de una manera similar si M fuera impar. Una manera simple para solucionar este problema es que M sea primo y he aquí la importancia del largo 37. Ya que si M es primo nos garantiza que antes de la primera colisión todas las celdas habrán sido visitadas 1 vez. Veamos la función hash:

```
int codigo_ascii(char a) {
     return a - '0';
}

int hasheo_uno(char* palabra) {
  int largo = strlen(palabra);
  int suma = 0;
  for (int i = 0; i < largo; i++) {
    suma += codigo_ascii(palabra[i]) * pow(2, i);
  }
  return (suma % 37);
}
```

Podemos apreciar la simpleza de la función hasheo_uno(), pero aún así la característica de una buena función Hash es su baja tasa de colisiones, y a modo de desafío personal me decidí a implementar un Hash doble por el cual me aboqué a la función hasheo_dos() (además con dos índices las listas serían menores y con un área de rebalse conformado por estructuras LSE el tiempo de búsqueda sería menor).

```c
int hasheo_dos(char* palabra) {
  int largo = strlen(palabra);
  int k = 0, suma = 0;
  for (int i = 0; i < largo; i++) {
    if (i != 0) {
      suma += codigo_ascii(palabra[i]) * pow(2, k++);
    }
  }
  return suma % 37;
}
```

Donde hasheo_dos() es igual a hasheo_uno() solamente que la key devuelta es con un desplazamiento a la izquierda del alias original. Ejemplo:
hasheo_uno(aba) = hasheo_dos(aaba)
Ya que hasheo_dos() hace el mismo calculo que realiza hasheo_uno() solamente que lo omite para el primer caracter de la cadena generando una diferencia en potencias de 2.
De esta manera con estas dos funciones Hash similares, las colisiones serían significativamente menores.
Ahora la pregunta sería ¿Cómo resolver una colisión?. Dado que por la dinámica del programa no hay que eliminar conjuntos de nuestra estructura de almacenamiento Hash, solamente hay que guardarlos para tenerlos como posibles operandos o a lo sumo redefinirlos. Por eso opte por un sistema de encadenamiento separado o hashing abierto, por el cual en cada localización de la tabla brindada por las dos funciones Hash se construiría una lista simplemente enlazada de registros donde su data vendría ser determinada por un puntero a la siguiente estructura:

```c
typedef struct _Conjunto {
  char* alias;
  GList lista;
  GList intervaloLista;
  int vacio;
}* Conjunto;
```

De esta manera tendríamos en cada posición de la tabla inicializada una lista simplemente enlazada de conjuntos, ahora analicemos en profundidad esta estructura.
A mi parecer es fácil entender que la variable alias va a almacenar el nombre del conjunto mientras que la bandera entera vacío nos va a servir como un indicador sobre el estado del conjunto en cuanto a qué si es vacío o no. Seguro se estará preguntando sobre la existencia de las dos listas simplemente enlazadas presentes en la declaración de la estructura. La primera lista va a ser funcional cuando se declara un conjunto por extensión almacenando en ella una data tipo int.
Ejemplo:

```
Input:
Alias = {1,4,6,7}
Valor de lista:
Alias->lista = 7->6->4->1
```

Se muestra el inverso de la lista porque siempre se agrega el numero al comienzo de la LSE para amortizar costos de ingreso.
Bien, la segunda lista va a tener primacía en la declaración por comprensión de un conjunto y en la devolución de los resultados de las operaciones dado que esta lista va a ser una lista simplemente enlazada de la siguiente estructura:

```
typedef struct _Intervalo {
  long long inicio;
  long long ultimo;
  int esVacio;
  long long cardinalidad;
} Intervalo;
```

Dónde es intuitivo que en esta estructura desglosamos los componentes de un intervalo matemático almacenando su extremo inferior, extremo superior, cardinalidad y si es vacío. Veamos un ejemplo de cómo participa esta lista de intervalos a la hora de declarar un conjunto por comprensión:

```
Input:
AliasA = {var : -10 <= var <= 10}
Valor de intervaloLista:
AliasA->intervaloLista = (Intervalo) [-10,10] -> NULL
```

Estos ejemplos son intuitivos para poder entender un poco como desde un comienzo se desglosa la información proporcionada por el usuario en las definiciones de los conjuntos, si bien podría haber transformado todo en una lista intervalos desde un comienzo y expresar las declaraciones por extensión {x1,x2,...,xn} de la forma [x1,x1]->[x2,x2]->...->[xn,xn]. Yo quería respetar la naturaleza del conjunto definido por extensión y no "transformarlo" desde un comienzo a comprensión. De esta manera si un conjunto definido por extensión no es redefinido por el usuario(y pasado a comprensión) o siendo revalorizado por una operación(por ejemplo AliasA = AliasA | AliasB), la GList lista va a mantener su integridad.

2.2 ¿LSE? Sí. ¿POR QUÉ?

En este apartado nos dedicaremos a justificar el uso de esta estructura de datos como principal precursor del motor de las operaciones. Si bien, se podrían haber utilizado otras estructuras como la empleada en el segundo que TP, el árbol AVL de intervalos, estructura que hasta hoy en día pienso que sería la óptima y la predilecta para este trabajo práctico. Pero la simpleza de la lista simplemente enlazada, la facilidad para su manipulación, la posibilidad de una trazabilidad más simple de errores, seguramente a cambio de optimización y gasto de recursos terminaron inclinando la balanza hacia la LSE. Aunque en un arbol de intervalo existen operaciones sumamente ligeras en cuanto al costo, sumado al hecho de que estos árboles solamente iban a estar cargados de intervalos totalmente disjuntos(por eso por ejemplo la operación Complemento se resolvería en un solo recorrido de árbol). Además teniendo en consideración que las operaciones básicas de una lista simplemente enlazada, ya sea buscar tanto como recorrer tienen una complejidad O(n) mientras que el ingreso si este

se da al comienzo es O(1), no me parecía una mala base por dónde empezar a desarrollar el programa. Por lo cual nuestra estructura de datos para operar en el meollo de cada operación vendría a ser definida por las siguiente estructura:

```
typedef struct _GNodo {
  void* data;
  struct _GNodo* next;
} GNodo;

typedef GNodo* GList ;
```

Donde void* data vendría a estar definida con la anteriormente mencionada struct _ Intervalo (véase 2.1)

PART III

Operaciones

3.1 Unión

En sus comienzos la operación Unión estaba pensada como una composición de dos funciones la primera que dividía la lista de intervalos del primer operando en simples variables del tipo intervalo, aquí es donde entraba en juego la segunda función que solamente se encargaba de tomar un solo elemento de la primera lista de intervalos y unirlo a la segunda teniendo cuidado con los solapamientos. Esta estrategia de divide y conquistarás en un comienzo fue fructífera pero presentaba problemas cuando la segunda lista de intervalos correspondiente al segundo operador poseía intervalos con solapamiento además de problemas de gestión de memoria.
Pensé en unir ambas listas al momento de operar así se simplificaría la operación dado que sólo tendría que operar sobre la misma lista pero ahora tenía otro problema: los intervalos cuya intersección no es vacía están dispersos por toda la lista. Esto era fácil de solucionar sólo bastaba con ordenar los intervalos de una manera creciente tomando como referencia el inicio del mismo(ordenar tomando como factor el límite inferior de menor a mayor) esto no sólo traía beneficios al juntar los intervalos que podían presentar solapamiento sino también al momento de juzgar si dos intervalos tienen overlap ya que sólo tendría que hacer una comparación con los extremos superiores puesto que el ordenamiento me garantizaba que el extremo inferior de un elemento anterior en la lista sería menor a una posterior en la lista. Ahora sólo bastaba con recorrer la lista en búsqueda de superposiciones, pero al ser una lista de intervalos enteros(discretos) cabe recordar que si tenemos dos intervalos del tipo [z, x] y [x+1,w] su unión es [z,w].
Ejemplo:

```
Input:
AliasA = {var : 2 <= var <= 8}
AliasB = {var : 9 <= var <= 14}
AliasC = AliasA | AliasB
imprimir AliasC

Output:
2:14
```

Por eso en la lógica no solo hacía falta chequear las superposiciones de los intervalos sino también había que verificar el extremo superior del primer intervalo en la lista no sea inmediato inferior al extremo inferior del segundo intervalo presente en la lista. Verificado esto serían intervalos disjuntos de caso contrario conforman un único intervalo en el campo de los números enteros dentro de las posibilidades de representación del tipo de variable int.

3.2 Intersección

Siguiendo la línea de pensamiento de la operación Unión, era imprescindible partir de la premisa de que ambos conjuntos estén compuestos por intervalos disjuntos. Para no recurrir a un sistema de chequeo de intersecciones de complejidad O(m*n) con m el largo de la primera lista y n el largo de la segunda(dado que habría que contrastar cada intervalo perteneciente a la primera lista contra la totalidad de la segunda lista). Este problema, cómo está desarrollado el programa, es omitible dado que todo conjunto se define desde un comienzo como una sucesión de intervalos disjuntos y al tener asegurado que mi función Unión devolvería una lista intervalos disjuntos, este problema no prosperaría. De una manera similar que en la anterior operación era necesario tener las listas de los operandos ordenadas con respecto al extremo inferior de los intervalos.
Siguiendo un poco con el desarrollo, antes que nada, dado dos intervalos si existe intersección entre ellos(llámese [x,y] ,[z,w]), el intervalo que los interseca sería el mínimo de los extremos superiores y el máximo de los extremos inferiores. Obviamente moviéndonos con la premisa de la existencia de intersección. Continuando con la línea explicación nos quedaría contrastar cada nodo de la primera lista con la segunda, jugando solamente con los extremos de cada nodo de las lista. Dado que si un intervalo de la primera lista presenta solapamiento con uno de la segunda y el nodo que representa esa intersección es válido(el inicio menor o igual al final) automáticamente se toma como parte de resultado(porque son listas compuestas por conjuntos disjuntos) avanzando así en la primera lista. De caso contrario se sigue buscando intersección con el siguiente nodo de la segunda. Pero surgió un problema cuando no había intersecciones 'totales' (denominé intersección total cuando un operando estaba totalmente incluído en el otro).
Ejemplo:

```
Intersección Total:
[1,3] & [0,4] = [1,3]
```

La primera implementación se veía capaz solamente de poder resolver intersecciones de este tipo, pero no las que denominé 'parciales' o con 'remanente'. ¿A qué llamo intersección con 'remanente'? Es cuando un solo intervalo de la primera lista interseca a dos o más intervalos de la segunda lista(por eso 'remanente' ya que la parte que 'sobra' o que no está contenida en la primera intersección interseca a otro intervalo).
Ejemplo:

```
Intersección Remanente:
[0,3] & [0,0],[2,4] = [0,0], [2,3]
```

Como podemos ver [0,3] interseca a [0,0], pero deja un remanente que es [1,3] que interseca a [2,4]. Este problema fue complejo de solucionar ya que tendría que pensar en todas las posibilidades en las cuales se generaran un remanente, sobre todo con los extremos superiores de cada intervalo.

```
Remanentes:

Tipo A:
        [x___________y]
   [w_________z]
Tipo B:
        [x___________y]
               [w__________z]
Tipo C:
        [x___________y]
            [w____z]
```

Así los remanentes también no son estáticos, es decir que solamente se expresa el intervalo [x,y] contra el [w,z] sino que estos pueden permutar. Para solucionar este problema fue de esencial importancia mantener una relación de orden entre los extremos superiores de los intervalos(no de los extremos inferiores porque recordemos que las listas ya estaban ordenadas con respecto a este parámetro con anterioridad) así saber cuándo podemos hacer un cambio de índice para proseguir con la búsqueda tomando como variable otro intervalo en ambas listas.

3.3 Resta

Dado que por la definición formal, en la teoría de conjuntos, la operación resta depende de la definición de complemento y viceversa. Por lo cual me encontraba en la disyuntiva de si hacer que la operación resta dependiera de la de complemento o al revés. Me incliné por la segunda opción, definir una función resta y hacer que la definición de la operación complemento sea:

```
Complemento de A = ~A = Universal - A.
```

Lo he intentado de 2 maneras, ambas funcionales. Como primera opción viable quise explotar al máximo que las listas de intervalos a restar estuviesen conformadas por intervalos disjuntos.
Por definición de resta se sabe que, son todos los elementos que pertenecen al primer conjunto que quedan excluidos del segundo. Por eso se me había ideado una manera que dada una lista de intervalos A y un solo intervalo B, te haga la resta del intervalo B y de la lista A. Esto no tiene mucho sentido, ¿no?. Pero si te digo que el intervalo solitario B pertenece a nuestro primer operando y la lista A son los intervalos del segundo operando?. Como el resultado son todos los elementos pertenecientes a ese solo intervalo B que no están en la lista A. Dado que la lista de nuestro primer operando, que contiene a nuestro intervalo B, son intervalos disjuntos la operación intervalo B - lista A es parte del resultado final de una hipotética operación lista B - lista A, con intervalo B contenido en lista B. En definitiva es mucho menos costosa que la segunda manera.

La segunda manera la cual decidí dejar el trabajo por una simple razón, es mucho más intuitiva a la hora de entender, a la hora de juzgar y la hora de evaluar. Empecemos con la operacion Conjunto A - Conjunto B(con una lista A y una lista B que representa sus intervalos respectivamente), el principio

aplicado es facil, restamos a la lista A cada intervalo de la lista B, siguiendo la definición de manual de la operación resta. Pero estos resultados no pertenecen al resultado final, ni siquiera son resultados parciales como en la anterior forma. Sino que requieren de más procesamiento aún... y ese procesamiento viene de la mano de la operación ya hecha, intersección. Dado que por cada intervalo M presente en nuesto conjunto B vamos a obtener una 'pseudo-resta' del Conjunto A contra ese intervalo M, el resultado son elementos perteneciente al Conjunto A y no pertenecientes a ese intervalo M, a esa pseudo-resta la llamamos C. Vamos a tener tantas pseudo-restas C como intervalos tenga nuestro Conjunto B, todas estas pseudo-restas C son elementos pertenecientes a nuestro Conjunto A pero a su vez no pertenecen a un intervalo que compone a nuestro Conjunto B. Pensando en la definición de intersección, si interseco todos las pseudo-restas C obtengo una sucesión de intervalos pertenecientes al Conjunto A que no pertenece a ningún intervalo que compone a mi Conjunto B(por ende a B), llegando al resultado de la resta del Conjunto A y el Conjunto B.
Ahora, ¿Cómo guardar todas las pseudo-restas C de una manera eficaz?. He decidido que a cada C la voy a guardar en una pila, costo de guardado O(1). Luego para su procesamiento voy a intersecar los elementos en la pila sacando de a pares los resultados parciales(tipo pseudo-resta C), volviendo a introducir el resultado en la misma y hasta que la pila no tenga un único elemento no deja de intersecar su contenido. Este elemento es el resultado de la operación. Veamos un ejemplo:

```
Conjunto A = [2,6],[8,12]
Conjunto B = [1,2],[5,9]
Pila = NULL

 A - [1,2] = [3,6],[8,12] = C. Pila = C -> NULL
 A - [5,9] = [2,4],[10,12] = C'. Pila = C'->C->NULL

Saco los operandos de la pila
con una sucesión de stack.top(),stack.pop().

C' & C = [3,4],[10,12] -> Resultado de la operación
```

Espero que este breve ejemplo clarifique un poco el funcionamiento de este segundo método. Se aprecia a simple vista que el costo es más elevado que el anterior método. Pero obteniendo el mismo resultado.

3.4 Complemento

Como habíamos acordado con anterioridad esta función no tiene una definición perse propia sino que es, gracias a sus propiedades, la sustracción del universal contra el conjunto a hacer el complemento. Sin mucho más que agregar, hay un pequeño detalle en esta función. Dado que nunca se define el Universal con anterioridad me tomé la licencia de definirlo yo mismo, es decir, que el alias de conjunto "universal" está estrictamente reservado. Y este conjunto será inicializado la primera vez que se haga la operación Complemento. Por eso queda en el usuario no redefinirlo de caso contrario la operación complemento traería problemas. Cabe destacar que el universal está disponible también para

usar en otras operaciones sin definición previa, después de ser llevada a cabo la operación Complemento al menos 1 vez.

PART IV

Complementos

4.1 Makefile, compilación y ejecución

La compilación se lleva a cabo con el comando make, detallando que el archivo a compilar es el ./main creando el ejecutable.

4.2 Bibliografía y material de consulta

. Apuntes teórico de la asignatura
. El lenguaje de programación C, 2da edición. Kernighan & Ritchie

4.3 Curiosidades

4.3.1 Encarpetado

Sinceramente, me hubiese gustado darle un encarpetado más prolijo al proyecto dándole mayor organización. Pero me fue complicado llevarlo a cabo, creo yo, por un tema en relación a las dependencias. Dado que cuando encarpetaba el proyecto de manera local no reconocía ciertas funciones básicas de otros archivos que estaban agregados con el comando #include "../path". Desde ya sería la principal tarea pensando en un aspecto a mejorar.

4.3.2 if()...

En tres ocasiones tuve que usar estructuras if() vacías cuando nos referimos a la captura de datos por teclado ya sea por la función fgets() o scanf(). Dado que cuando compilaba con el make, este tiraba el siguiente error:

```
Ignoring return value of ‘scanf’, declared with attribute warn_unused_result
Ignoring return value of ‘fgets’, declared with attribute warn_unused_result
```

Dado que intente poner el cast (void) delante de las funciones, cosa que no funcionó. Intenté colocarlas en estructuras if() seguidas de un ";" y el error persistía en el make, no me quedó otra opción que poner llaves.

```
if (fgets(alias, LIMITE, stdin)){}
char* operacion = NULL;
int instruccion = 0;
 if (strlen(alias) == LIMITE - 1) {
  printf("Overflow de entrada...\n");
  if (scanf("%*[^\n]")){}
  if (scanf("%*c")){}
  free (alias);
  continue;
 }
```

Desconozco el por qué del origen de este error dado que a mis compañeros no les ocurrió. Pero buscando en internet, un poco, este error es frecuente.

PART V

Ejercicios Adicionales

5.1 Ejercicio Greedy

Luego de absorber la teoría de algoritmos Greedy, dada por la cátedra, y acompañado de la idea proveída por:
https://en.wikipedia.org/wiki/Activity_selection_problem
Me propuse hacer un algoritmo Greedy de manual, es decir, con todas sus funciones como están definidas en el pdf del campus. Primero antes de definir las soluciones ya sean parcial o temporal, hay que analizar la estructura de los tiempos. Dado que en un conjunto de tareas o actividades, a mí en partícular no me va a interesar el tiempo de inicio de las mismas sino el tiempo de parada. Y para seleccionar una solución temporal, esta tendría que ser aquella actividad que termine antes que todas las demás, por eso abrimos el algoritmo con la línea:

```
int greedy_de_actividades(GList grilla) {
 grilla = glist_selection_sort(grilla);
```

Ahora necesitamos inicializar la solucion_Parcial como set_vacio(), por eso a mi Glist solucion la igualo a nulo y como curiosidad inicializo una variable que sería su cardinal, se podría hablar de dos soluciones del tipo parcial... pero como tienen una estrecha relación y una depende de la otra podríamos decir o acotar que son una:

```
 GList solucion = initialization_glist();
 int cantidadActividades = 0;
```

Ya inicializada nuestra solución parcial, es turno de la temporal, que sería el primer elemento de nuestra lista ordenada. Tomando este primer elemento como un "candidato" al saber que es el que tiene el menor tiempo de parada sabemos que cumpliría con la función es_solucion_parcial() por lo cual se lo agrego a nuestra solucion parcial:

```
 if (grilla != NULL){
  Intervalo* primeraAct = get_data_glist(grilla, 0);
  solucion = prepend_glist(solucion, primeraAct);
 }
```

Ahora, según la definición necesitamos dos funciones !set_es_vacio(C) && !es_solucion(solucion_Parcial) dentro de un bucle.
!set_es_vacio(C) sería el equivalente a !empty_glist(grilla). Dado que cumplen la misma función.
Pensemos cuando una solución parcial es solución...Cuando su cardinal es mayor a cero, indiscutidamente:
es_solucion(solucion_Parcial) tiene su equivalente en (cantidadActividades $>$ 0) siendo cantidadActividades el cardinal de mi solucion_Parcial "solucion" y su negación sería !(cantidadActividades $>$ 0):

```
 while (!empty_glist(grilla) && !(cantidadActividades > 0) && indiceJ <
     largoGrilla) {
```

Según la definición de Greedy, necesitamos seleccionar los candidatos del conjunto. Yo lo hago de una manera directa, entonces mi función seleccion(C)

dado que la lista esta ordenada de menor a mayor por el tiempo de parada, solo hay que seguir un orden ascendente en la lista:

```
Intervalo* actPosJ = get_data_glist(grilla, indiceJ);
Intervalo* actPosI = get_data_glist(grilla, indiceI);
```

Siguiendo un poco, necesitamos una función que nos diga si nuestro candidato "actPosJ", ahora solucion temporal, es una solución parcial. Para ello tengo que verificar si actPosJ es un intervalo disjunto a su inmediato anterior, solo comparamos si el inicio de actPosJ es mayor o igual al de su inmediato anterior ya que la lista está ordenada por los extremos superiores.

```
if (actPosJ->inicio >= actPosI->ultimo){
```

Si es solución parcial, a nuestra solución temporal actPosJ la agrego a nuestra solución parcial "solucion" y como actPosJ es solución parcial igualo mi indiceI a mi indiceJ y avanzo en indiceJ

```
  solucion = prepend_glist(solucion, actPosJ);
  indiceI = indiceJ;
  indiceJ++;
}
```

Si no es solución parcial, significa que hay solapamiento en nuestros nodos, por lo cual no se pueden desarrollar ambas a vez por lo cual solo avanzo en mi indiceJ

```
  else {
     indiceJ++;
  }
}
```

Salido del bucle actualizo el cardinal de mi solucion_Parcial:

```
cantidadActividades = largo_glist(solucion);
```

Recordando mi función en el condicional del bucle while, !(cantidadActividades > 0) que es el equivalente a !es_solucion(solucion_Parcial). Debo aplicarla en un control de flujo if(), según la receta, retornando el cardinal del set_vacio el 0:

```
if (!(cantidadActividades > 0)) {
  glist_destruir_int(solucion);
  return 0;
}
```

Si no retornamos el cardinal de la solucion_Parcial "solucion":

```
glist_destruir_int(solucion);
return cantidadActividades;
}
```

Asi espero haber respetado la consigna del enunciado y que esta breve explicación sumada a la presente en el mismo código, comentada, clarifique la funcionalidad.

Intenté que el algoritmo posea todas las funciones que son propias de los algoritmos Greedy.

5.2 Rehashing

Lo más complicado de este ejercicio fue el adaptarme a una implementación ya hecha, pero luego de leerla la pude comprender y desarrollar un plan de acción para abordar el ejercicio. Pasemos a la explicación... Antes que nada me pareció más fácil crear una nueva tabla con la capacidad igual al doble de la anterior, respetando el argumento invariante de la función hash:

```
void tablahash_redimensionar(TablaHash * tabla) {
 TablaHash* nTabla = tablahash_crear(tabla->capacidad * 2, tabla->hash);
```

Luego tendríamos que hacer un recorrido similar al que se hace cuando se inicializa la tabla hash en busca de casillas no vacías para su posterior agregado en la tabla nueva:

```
 for (unsigned idi = 0; idi < tabla->capacidad; idi++) {
  if (tabla->tabla[idi].clave != NULL) {
   CasillaHash cas = tabla->tabla[idi];
   tablahash_insertar(nTabla, &(*((int *)cas.clave)), &(*((int *)cas.dato)));
  }
 }
```

Para finalizar solo tendría que pisar el contenido de la vieja tabla (redundantemente también este campo se llama tabla), cambiar la dirección de los apuntadores y liberar el puntero de la nueva tabla. Quedando actualizada la vieja:

```
 free(tabla->tabla);
 *tabla = *nTabla;
 free(nTabla);
}
```

En el main() se especifican los casos de testeo sobre el redimensionamiento de la tabla.