INFORME TRABAJO PRACTICO FINAL

- **Estructuras utilizadas:**

Tabla Hash: Decidí utilizar esta estructura para almacenar los conjuntos ya que brinda la posibilidad de buscarlos rápidamente, para poder, en el caso de este programa, operar con ellos. Dentro de ella se encuentran:

  - Las casillas de la tabla.
  - La capacidad de la tabla.
  - La función Hash.

Dentro de las casillas de la tabla se encuentran:

  - Un puntero void que representa a la clave
  - (SList) Una lista enlazada que representa al área de rebalse (en caso de que no se esté operando con el primer nodo).

Dentro de la estructura SNodo se encuentran:

  - Un puntero char que representa al alias del conjunto.
  - (CList) Una lista enlazada que representa al conjunto de enteros.
  - Un puntero SNodo al siguiente nodo.
  - Un puntero CNodo que representa al último nodo del conjunto, para así al agregar un nuevo nodo, no haya que recorrer la lista entera.

Dentro de la estructura CNodo se encuentran:

  - Un puntero void que representa al dato del nodo.
  - Un entero que representa al tipo de dato que se encuentra en el puntero anterior (NÚMERO, INTERVALO o VACIO).
  - Un puntero CNodo al siguiente nodo.

Estructura intervalo: Esta estructura es utilizada para almacenar intervalos dentro de los nodos de CList. Dentro de ella se encuentran:

  - Un entero que representa al extremo izquierdo.
  - Un entero que representa al extremo derecho.

- **No marcar el ultimo nodo de la lista que representa el área de rebalse:**

Este nodo no es marcado ya que como el programa puede sobrescribir conjuntos, hay que chequear todo nodo siempre antes de insertar el nuevo conjunto, ya que si se sobrescribe, este no irá en un nuevo nodo, por lo tanto, no necesariamente estará al final.

- **Clave utilizada:**

Decidí utilizar al alias de cada conjunto para conseguir la clave. Esta se define como: $\sum_{k=0}^{n} 3.alias[k]$, donde n es la longitud del alias menos uno.

- **Formatos de los alias:**

El programa soporta alias sin espacios, para ingresar conjuntos, o para operar con ellos, ya que va a leer hasta que encuentre uno. Ej. A, B1, AyC, B&D1, A-B, etc.

- **Variables en el ingreso de comandos por compresión:**

El programa soporta variables las cuales no sean numéricas y sean de un solo carácter. Ej. x, y, z, etc.

- **Implementación de algoritmo insertion sort y función que elimina elementos duplicados** (1)**:**

La función que elimina elementos duplicados es necesaria en mi trabajo ya que es la única forma de asegurarnos de que no haya elementos duplicados.

Decidí implementar el algoritmo insertion sort ya que, al tener a los conjuntos por extensión ordenados desde un principio, me facilito la construcción de los algoritmos de las operaciones del programa.

- **Conjuntos vacios:**

El programa soporta conjuntos vacios, recibidos de la forma: alias = {}.

Además, estos conjuntos también van a aparecer en los casos donde haya: intersección vacía, resta vacía y complemento vacio (este último tomando como universo el intervalo [INT_MIN, INT_MAX]). Luego, todas las operaciones soportan trabajar con conjuntos vacios, y al momento de imprimir uno de estos, se imprimirá “conjunto vacio”.

- **Mensajes de error:**

El programa devolverá un mensaje de error en los siguientes casos:

  - Ingresos de conjuntos por extensión o compresión.

    En caso de que estos ingresos sean erróneos, se imprimirá un error en el cual se muestra como se deben ingresar dichos conjuntos.

- Operaciones (unión, intersección, resta y complemento).

  Como en el caso anterior, en caso de que estos ingresos sean erróneos, se imprimirá un error en el cual se muestra como deben ser ingresados.

  Si el anterior error no pasa, pero se quiere operar con alias los cuales no tienen relación con ningún conjunto, se imprimirá un mensaje indicando cual es dicho alias.

- Casos inválidos.

  Si el programa no detecta que lo ingresado tiene relación con alguno de los anteriores casos, se imprimirá “comando invalido”.

- **Complicación con el algoritmo unión:**

En un principio diseñe y escribí el algoritmo completo, pero luego había casos que no estaban contemplados, por lo que la función no devolvía el resultado esperado. Por lo tanto, decidí volver a diseñar la función esta vez pensando caso por caso. Luego de probar un caso y que funcionara, pasaría al siguiente, entonces al llegar al último caso y resolverlo, la función estaría terminada.

Esta misma estrategia fue usada en el algoritmo de intersección y complemento.

- **Complicación doble free:**

En las funciones de las operaciones, en algunos casos, cuando tenía que agregar un nuevo nodo a la lista, lo que hacía era directamente agregar el dato de alguna de las listas con las que estaba operando, en lugar de hacer un nuevo malloc. Por lo tanto al destruir la lista, se hacía free a un dato el cual ya estaba liberado. Cuando me di cuenta del error lo que hice fue hacer un malloc en los casos en los que no lo hacía, guardar el dato en ese espacio de memoria, y luego agregarlo a la lista.

- **Compilación:**

El archivo makefile adjuntado a la entrega se encarga de compilar el programa con el comando ‘$ make interprete’, y también de limpiar los archivos objeto generados, con el comando ‘$ make clean’.

El programa se corre con el comando ‘$ ./interprete n’ donde n es un numero natural que representa el tamaño de la tabla. Decidí que el tamaño se pase de esta manera para que así se elija el tamaño adecuado según el uso que se le va a dar al programa. En caso de que no se acompañe al comando con un número, se imprimirá un error; y esto mismo pasará si la variable es un número menor o igual a cero o directamente no es número.

- **Utilización del programa:**

Formas de ingresar conjuntos u operar con ellos:

Tomando a y b como enteros.

- Conjuntos por extensión: alias = {a,..,b} (sin espacios entre comas).
- Conjuntos por compresión: alias = {x : a <= x <= b}
  En caso de que a = b, se añadirá a la lista como numero, y en caso de que b < a, se añadirán dos intervalos, [INT_MIN, b] y [a, INT_MAX].
- Conjunto vacio: alias = {}
- Operaciones que relacionan dos conjuntos: alias = alias2 | alias3, donde | también puede ser – o &.
- Operación complemento: alias = ~alias2
- Imprimir conjunto: imprimir alias
- Salir del programa: salir

- **Debugging y problemas de memoria luego de haber terminado el código:**

Con la utilización de valgrind, me di cuenta que había casos en los que quedaba memoria sin liberar. A partir de herramientas que brinda este programa, logre darme cuenta donde estaban estos errores y pude resolverlos.

- En un principio el alias y la clave se obtenían para cualquier tipo de comando recibido, lo que generaba errores de memoria ya que las funciones que hacían este trabajo no estaban preparadas para cualquier tipo de comando. Por lo tanto al encapsular este comportamiento para casos específicos, se resolvió el error.
- No se hacía free de alias en los casos de que este no haya sido utilizado.

El debugging, compilación y ejecución se realizaron en Ubuntu 20.04 (consola virtual desde Windows).

- **Bibliografía:**

(1) https://es.wikipedia.org/wiki/Ordenamiento_por_inserción