Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Estructuras de Datos y Algoritmos I

Trabajo Práctico Final

Alumna:

Sullivan, Katherine

Universidad Nacional de Rosario

2021

A continuación se presenta un informe detallando las decisiones en general y particularidades en el diseño y desarrollo del Trabajo Práctico Final de la asignatura Estructuras de Datos y Algoritmos I.

1 Módulos del programa

El programa se encuentra dividido en 9 módulos:

- `acciones`: engloba las funciones y estructuras particulares de las que se denominan "listas de acciones". Las mismas son utilizadas concretamente para la implementación de las funciones *deshacer* y *rehacer*.
- `andor`: dada la extensión del código de las funciones *and* y *or* y la necesidad de crear una estructura *Argumento* para la ejecución en paralelo que realizan, se decidió crear este módulo especificamente para ellas.
- `arbol`: módulo creado para la implementación de árboles AVL que contienen un idx (atributo que no se repite entre los nodos de un mismo árbol) y un dato que puede repetirse entre los nodos de un mismo árbol pero es el que generará su orden.
- `contacto`: engloba las funciones relativas a la estructura *Contacto*, que se usa como dato para la agenda (además de la estructura misma).
- `impresiones`: a fin de organizar y mantener junta la interacción que el programa hace con el usuario, se creó este módulo que mandeja las impresiones por consola.
- `interprete`: módulo principal del programa, se encarga de las solicitudes de usuario, conteniendo en el una función que interpreta la entrada de usuario y las funciones que realizan las acciones respectivas.
- `slist`: módulo creado para la implementación de lisats simplemente enlazadas.
- `stree`: módulo creado para la implementación de árboles AVL con un solo dato único entre los nodos de un mismo árbol.
- `tablahash`: módulo creado para la implementación de la estructura principal de la agenda, una tabla de hash con manejo de colisiones a través de hashing doble.

2 Estructuras de datos utilizadas

2.1 Tabla Hash

Como estructura principal para la representación de la agenda se decide implementar una tabla de hash.

Al momento de elegir las estructuras de datos con las cuales trabajar en el proyecto, se decidió priorizar el aspecto que se considera el requerimiento más importante de una agenda de contactos: realizar búsquedas rápidas.

Por lo tanto, para poder realizar la operación *buscar* en tiempo constante se decidió implementar una tabla de hash. Sin embargo, si se observa la estructura a continuación se puede notar que no se trata de una tabla de hash usual, pues cuenta con 4 árboles extra. De estos se hablará en la próxima sección.

```
typedef struct {
  CasillaHash *tabla;
  unsigned numElems;
  unsigned capacidad;
  FuncionHash hash;
  FuncionHash hash2;
  Arbol arbol_nombre;
  Arbol arbol_apellido;
  Arbol arbol_edad;
  Arbol arbol_tel;
} TablaHash;
```

Con casilla de hash siendo la estructura

```
  typedef struct {
  char *clave;
  Contacto dato;
  int estado;                    // 0 libre, 1 ocupada, 2 eliminada
} CasillaHash;
```

2.2 Árboles AVL

Los árboles mencionados en la sección anterior son los representados por la siguiente estrctura:

```
typedef struct _Nodo {
  void *dato;
  int idx;
  struct _Nodo *izq;
  struct _Nodo *der;
  int alt;
} Nodo;

typedef Nodo *Arbol;
```

Estos árboles se implementaron con la misma idea de optimizar las búsquedas, solo que esta vez las generadas por las funciones *and* y *or*, pues servirán para realizar búsquedas por atributo con un costo promedio de $O(log\ n)$ donde n es la cantidad de contactos en la agenda (el costo variará de acuerdo a la cantidad de elementos repetidos que se presenten, pudiendo ser mayor).

Además estos árboles proveen una gran ventaja para la función *guardar ordenado* al poder ser recorridos en orden.

Respecto a los otros árboles implementados, su propósito fue el de proveer una manera de no repetir la impresión de contactos repetidos en la función *or*, pues puedo asegurar la no inserción de elementos con claves repetidas.

Su estructura es la que se presnta a continuación:

```
typedef struct _STNodo {
  int idx;
  struct _STNodo *izq;
  struct _STNodo *der;
  int alt;
} STNodo;

typedef STNodo *STree;
```

2.3 Listas simplemente enlazadas

Su creación fue con el fin de tener una estructura de tamaño variable donde se pueda ir almacenando los resultados de las búsquedas en los árboles de atributos.

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
  int cant;
} SNodo;

typedef SNodo *SList;
```

2.4 Listas de acciones

Las listas de acciones son, en esencia, listas doblemente enlazadas con un puntero a su cola. Este tipo de estructura fue la elegida porque provee una forma simple de eliminar tanto del principio (cuando se llega a su capacidad máxima) como del final (cuando se realiza un deshacer o rehacer) e insertar al final (cuando se invoca a *agregar*, *eliminar* o *editar*).

Las acciones son estructuras con la siguiente forma:

```
typedef struct {
  int tipo;                    // 1 agregar - 2 eliminar - 3 editar
  char *nombre;
  char *apellido;
  char **tel;
  int *edad;
} Accion;
```

Y se encuntran dentro de nodos doblemente enlazados para formar la siguiente estructura de lista:

```
typedef struct {
  AccNodo *head;
  AccNodo *tail;
  int elems;
  int cap;
} AccList;
```

2.5 Otras estructuras

Además de los expuestos arriba el programa cuenta con 3 estructuras más.

La primera de ellas es *Contacto*. Esta estructura es la que representa un elemento de la agenda.

Respecto a las otras dos, ellas son *Argumento* y *ArgHilo*. Como se puede deducir de su nombre, estas estructuras sirven el porpósito de funcionar como argumentos para las rutinas.

3 Algoritmos de interés

3.1 Deshacer/Rehacer

Como se hizo mención en las secciones anteriores, las funciones *deshacer* y *rehacer* se apoyan en la estructura lista de acciones.

Antes de pasar a la explicación sobre el comportamiento de estas funciones, se definirá lo que se entiende por "acción contraria". Por "acción contaria" se entiende de agregar, eliminar (ambas con los mismos datos), de eliminar, agregar (ambas con los mismos datos) y de editar, editar pero con el orden de los teléfonos y edades cambiados, es decir, el `accion->tel[0]` pasa a ser `accion->tel[1]` y viceversa, y `accion->edad[0]` pasa a ser `accion->edad[1]` y vicerversa, en la "acción contraria".

Ahora bien, para mantener su comportamiento esperado se siguen las siguientes reglas:

- Al realizarse la operación *agregar*, se añade a la lista de acciones de deshacer una acción de tipo 2 (eliminación) con todos los datos del contacto agregado (con `accion->tel[1] = NULL` y `accion->edad[1] = 0`).
- Al realizarse la operación *eliminar*, se añade a la lista de acciones de deshacer una acción de tipo 1 (inserción) con todos los datos del contacto eliminado (con `accion->tel[1] = NULL` y `accion->edad[1] = 0`).
- Al realizarse la operación *editar*, se añade a la lista de acciones de deshacer una acción de

tipo 3 (edición) con nombre y apellidos como los del contacto editado, en las posiciones 0 de teléfono y edad el télefono y edad viejos, y, en las posiciones 1, los nuevos.

- Al llamarse a las funciones *deshacer* o *rehacer* se realiza la última acción que agregaron a sus listas (eliminándola posteriormente) y se agrega a la otra lista (rehacer en el caso de deshacer, y viceversa) la acción contraria.
- Siempre que se realiza una acciíon de tipo 3 (edición) se toma el teléfono y la edad que se encuentran en la posición 0 de sus respectivos arrays.
- Cada vez que se realice un cambio en la agenda (producido por las tres operaciones que influyen en deshacer y rehacer o por la función *cargar*) se eliminan de rehacer todas las acciones que estaba guardando.

3.2 Guardar ordenado

Gracias a los árboles de atributos, el algoritmo necesario para guardar de manera ordenada es simplemente un recorrido inorder sobre el árbol de atributo indicado que va impriendo en el archivo de salida los contactos indicados por el parámetro *idx*.

3.3 Buscar por suma de edades

El algoritmo utilizado para la función *buscar por suma de edades* sigue la estrategia de programación dinámica bottom-up.

La idea del algoritmo es, en primer lugar, establecer si se puede o no conseguir un subconjunto de edades tales que sumadas den el natural ingresado. Esta parte del algoritmo resulta ser el problema que se conoce como *Subset Sum Problem* y es sobre la que se aplica la estrategia de programación dinámica.

La resolución del *Subset Sum Problem* implementada sigue la siguiente lógica:

Podemos decidir si un conjunto de enteros (en este caso, las edades) puede formar una suma N si vamos recorriendo el conjunto y por cada elemento x ir viendo si se puede obtener N con los anteriores elementos del conjunto o si con los anteriores elementos del conjunto se puede formar

la suma $N - x$ (luego sumando x al conjunto se puede obtener N). Por lo tanto, para evitar el recálculo de algunas instancias que supondría una implementación recursiva, lo que realiza el algoritmo es la construcción de una matriz $(m + 1) * (N + 1)$ (donde m representa la cantidad de elementos del conjunto) de manera bottom-up siguiendo los pasos mostrados a continuación:

1. Todos los elementos de la columna 0 (es decir, para la suma igual a 0) tendrán un valor verdadero (1 en este caso) pues se puede obtener la suma con el conjunto vacío.

2. Salvo por el perteneciente a la columna 0 todas las entradas de la fila 0 tendrán un valor falso (0 en este caso) pues sin elementos no se puede conseguir ninguna otra suma más que 0.

3. Para completar los valores del resto de la tabla se analiza lo siguiente: si la columna en la que se está posee un valor menor a la edad correspondiente a la fila (no se puede usar este elemento para construir el conjunto deseado) simplemente se copia el valor de la fila de arriba, por otro lado, si no sucede lo anterior se verifica si se pudo obtener la suma sin ese elemento (revisando la fila anterior) o si se pudo obtener la suma menos el elemento actual (fijandose en la columna correspondiente de la fila anterior).

Una vez armada esa tabla simplemente se chequea el elemento de la última fila y última columna. Si es 1, luego significa que se puede obtener el conjunto, si es 0, no.

En cuanto a la segunda parte del algoritmo, esta solo se realiza si se determinó que se puede obtener algún subconjunto tal que la suma de sus edades de N y consiste en imprimir uno.

Para su implementación se realiza una especie de backtracking. Este consiste en, empezando desde la última fila y última columna, ir subiendo y determinando si el elemento correspondiente a esa fila es necesario para la construcción del conjunto. Si se determina que es necesario, entonces se imprime y también se cambia de análisis de columna a la columna actual menos el elemento necesario. Para establecer si un elemento es necesario o no se revisa si su fila anterior en la misma columna tiene un 0 o un 1. Si esta posee un 0, el elemento es necesario. Se procede así hasta que la llegar a la columna 0. De esa manera queda impreso un conjunto tal que la suma de sus edades da N.

4 Dificultades encontradas

4.1 Sobre la impresión en la búsqueda de suma de edades

Encontrar una forma óptima de imprimir un conjunto dentro de esta función resultó un desafío. Se pudo crear la solución expuesta en la sección anterior gracias a la lectura sobre el problema conocido como *Perfect Sum Problem*, una variante del *Subset Sum Problem* donde se deben imprimir todos los subconjuntos tales que den la suma requerida.

4.2 Sobre la replicación de la información

Resultó también desafiante el decidir cómo manejar la replicación de la información, especialmente frente a la decisión de creación de los árboles. Estos debían contener el dato del atributo correspondiente pero también una manera de acceder al contacto entero para su impresión. La solución implementada fue usar el mismo puntero a char que se encuentra dentro del contacto en la tabla de hash para los atributos árboles correspondientes a nombres, apellidos y teléfonos y crear un puntero cuya desreferencia sea la edad del contacto para el árbol de edad, además de que todos contaran con el idx que permite acceder al contacto a través de la tabla de hash.

5 Decisiones particulares

5.1 Sobre cuándo reinicializar la lista de acciones de rehacer

Se decidió reinicializar rehacer cuando se produjesen cambios sobre la agenda. Es decir, si se invoca a cualquier función de búsqueda o guardado se sigue pudiendo rehacer los cambios producidos por deshacer.

5.2 Sobre pisar la información de un contacto

Una decisión importante a tomar era qué hacer si un usuario intenta agregar un contacto que ya existe en la agenda. Se decidió que si ese intento es realizado por la función agregar no se va a eliminar el contacto que ya existía sino que se le va a avisar al usuario, en cambio, si se está

intentando agregar un contacto que ya existía mediante la función cargar, entonces sí se pisará la información del contacto que ya existía, pues se puede entender de un cargar que se requiere que queden presentes en la agenda todos los contactos que estaban en el archivo, pero podría suceder que si un usuario está agregando manualmente los contactos este se olvide que ya agregó uno con el mismo nombre.

5.3 Sobre resizing dinámico de la agenda

Al crear una tabla de hash se le debe asignar una capacidad, por lo tanto surge la pregunta de qué se debe hacer si se intentan agregar más contactos de los que permite la capacidad. Se decidió que lo mejor es realizar resizing dinámico cuando se llega al 70% de la capacidad de la tabla (para poder seguir asegurando un costo constante de búsqueda) y se garantiza que empezando con la capacidad dada (31) sólo se llegará a una capacidad no prima cuando se llegue a querer una agenda de 1271 contactos a partir de esa nueva capacidad no se garantiza un efectivo doble hashing porque no existirá coprimalidad entre el paso de la segunda tabla de hash y la capacidad de la tabla. Si se van a cargar más de 441 contactos se recomienda que se incie con una dimensión prima más grande. (Además de en general recomendarse establecer una capacidad lo suficientemente grande, puesto que el resizing resulta costoso).

6 Compilación e invocación

Para la compilación del programa la entrega cuenta con un archivo Makefile. Para producir el archivo ejecutable basta con correr alguno de los siguientes comandos:

- `make`: además de generar un ejecutable para el uso del programa borra todos los archivos objeto de la carpeta actual
- `make all`: ídem make
- `make main`: solo produce el ejecutable main

7 Bibliografía

7.1 Obras consultadas

- Brassard, G. - Bratley, P. (1997) *Fundamentos de la Algoritmia.*
-
- Kumar, V. (2019) *On the prerequisite of Coprimes in Double Hashing.*
- Tenebaum, A. - Augenstein, M. - Langsam, Y. (1993) *Estructuras de Datos con C.*

7.2 Enlaces de interés

- https://www.geeksforgeeks.org/perfect-sum-problem-print-subsets-given-sum/
- https://www.geeksforgeeks.org/subset-sum-problem-dp-25/