Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Estructuras de datos y algoritmos I

Trabajo práctico Final

Alumno:
Petruskevicius Ignacio

Junio 2020

EDyAI - Informe TP Final

Petruskevicius Ignacio, LCC UNR FCEIA 01/06/2020

1. Introducción

Al comienzo del desarrollo del trabajo, fue clave la interpretación del enunciado. En este punto determiné como encararlo, dividirlo en 3 grandes partes que luego trabajé una por una. La primera es la interfaz de el usuario, encargada de leer las entradas del mismo y ejecutarlas de ser posible. Otra, el almacenamiento de los alias de forma que acceder a ellos sea poco costoso y rápido, y por último el almacenado de los conjuntos y su respectivo manejo.

2. Interfaz de usuario:

Esta sección busca responder a las especificaciones del enunciado, leyendo cada uno de los comandos de la forma explicitada, respondiendo a errores de escritura.

Comandos:

- Crear conjunto compresión: alias = {x: num <= x <= num}
- Crear conjunto extensión: alias = {num, num, ..., num}
- Salir: salir
- Imprimir conjunto: imprimir alias
- Unión conjuntos: alias = alias | alias
- Intersección conjuntos: alias = alias & alias
- Resta conjuntos: alias = alias - alias
- Complemento conjunto: alias = ∼alias

Nótese que realice algunas modificaciones en la sintaxis, en el conjunto dado por extensión, entre cada , y un numero introduje un espacio. Y en el conjunto dado por compresión entre la x y : borre el espacio. Estas correcciones me parecieron que seguían con la forma de escritura habitual en matemática.

2.1. Formato alias soportado:

El tipo de alias aceptado es alfanumérico no exclusivamente numérico, es decir, un alias puede contener números pero no solo números. Ejemplos:

A1, A, hola, 1hola1.

2.2. Dificultades y progreso:

Esta sección me generó algunas dificultades, primero al no tener claro como encararla, realice una implementación que andaba, pero resultaba engorrosa y "trabada". Estaba conformada por una gran cantidad de sentencias if anidadas. Por lo cual procedí a eliminarla y pensar en cómo resolverlo. Revisando en internet distintas técnicas o formas de hacerlo, encontré una que me pareció aplicable en este caso. Consta de clasificar cada palabra del comando en "tokens"los cuales son estructuras que contienen meta-datos de cada palabra clasificada. Luego de "tokenizar" el comando, se procede a leerlo, pero, con la información ya obtenida en la etapa anterior resulta más simple detectar errores en la sintaxis y a su vez ejecutar el código. Además se tiene certeza de que las palabras a las que se le asocia un tipo están bien escritas y siguen con las reglas. La última etapa la implementé con un gran switch, el cual analiza la primer palabra y dependiendo del tipo que tenga se procede a ejecutar la acción correspondiente.

3. Almacenamiento de alias:

En la sección anterior, unos de los chequeos que se deben realizar es, si el alias al que se le realizará una operación está, en efecto, almacenado y asociado a un conjunto. Para esto se debe buscar en la estructura de almacenamiento cada alias. Por ende, esta operación debe ser relativamente rápida. Tendiendo esto en cuenta, se me ocurrió utilizar una tabla hash, en la cual almacenar cada alias con su respectivo conjunto asociado. ¿Porque un hash y no una array o un árbol binario de búsqueda? Principalmente, porque el costo de acceder a cada elemento es generalmente una constante, no depende de la cantidad de elementos, de esta forma me aseguro contar con la información necesaria para operar a un bajo costo.

3.1. Implementación:

La tabla hash fue implementada mediante una array de listas simplemente enlazadas. La idea es que en caso de que la función de hasheo retorne para dos valores la misma key, está colisión sea solucionada agregando un nodo más a la lista enlazada. Con una buena elección de función de hasheo y un tamaño de tabla adecuado las comisiones se pueden reducir, llegando al caso de que no afecten significativamente en los costos de búsqueda.

3.1.1. Funcion hash:

Los alias de los conjuntos pueden ser alfanuméricos, por lo tanto esta función debe ser factible para este tipo de cadena de caracteres. Investigando, encontré la siguiente función:

Listing 1: Función hash djb2

```
1 int hash_obtener_key(char *alias) {
2   unsigned long hash = 5381;
3
4   for (int i = 0; alias[i]; i++)
5     hash = (33 * hash) + alias[i];
6
7   return hash;
8 }
```

El algoritmo fue desarrollado por Dan Bernstein, matemático y criptoanalista. La misma se basa en un acumulador al cual se le aplican operaciones (linea 5), notese que las mismas tienen a que el valor que contiene el acumulador crezca tanto que ocacione "overflow", esto es bueno, ya que ayuda a que pequeñas diferencias entre cadenas desarrollen grandes cambios en la key final, logrando así una mayor homogeneidad.

El creador afirma que el multiplicador, en este caso 33 (*) no tiene gran influencia sobre el desempeño de la función, pero que con 33 se obtuvieron buenos resultados, que logré corroborar en el desarrollo del trabajo.

(*) El creador propuso que esta linea sea de la forma

$$hash << 5 + hash + alias[i]$$

para que sea mas eficiente el cálculo cuando se trabaja con grandes números o largas cadenas, pero para el caso y para que sea mas entendible, decidí escribir más explícitamente la operación.

$$hash << 5 + hash = (hash * 2^5 + hash) = hash * 32 + hash = hash * 33$$

3.1.2. Tamaño de tabla:

La relación entre la elección de una buena función de hash y un adecuado tamaño de tabla resulta fundamental para el correcto desempeño de la estructura de datos. En este caso lo importante era pensar en la cantidad de conjuntos que serian almacenados a la vez, esto define el tamaño necesario de tabla. Ya que si el tamaño es menor a la cantidad de espacios disponibles en la tabla, las colisiones serán inevitables y por lo tanto la eficiencia bajara. Pero, si el tamaño de tabla es excesivamente grande, se desperdiciara mucha memoria. Luego de algunas pruebas y pensando en que 1000 conjuntos almacenados sera una cantidad normal, pero pensando en que tal vez se supere, decidí setear el tamaño de la tabla en 1500. Esto permite que una cantidad bastante mayor de conjuntos sea almacenado y aun sera eficiente y rápido, ya que, por ejemplo si almacenamos 3000 conjuntos, las colisiones sera en promedio de 2 elementos, lo cual es rápido de recorrer.

Este tamaño puede ser fácilmente ajustado desde la constante LARGO_TALBA en el main.

4. Almacenamiento de conjuntos:

Una de las partes mas importantes del trabajo, es el almacenamiento y manejo de conjuntos. Estos pueden ser dados de dos formas, por compresión y extensión. Teniendo esto en cuenta, debí encontrar una estructura de datos para almacenarlos, baraje unas cuantas posibilidades, desde una array de estructuras intervalo, una array de arrays de ints, en donde se explicitan los elementos de cada conjunto. Pero pensando, investigando y viendo que en el anterior TP trabajamos con intervalos, pensé en usar el árbol de intervalos que implementamos. Esto es, un conjunto de elementos pasa a ser un árbol de intervalos, así soporta las dos formas de dar un conjunto, ya que si es por extensión cada elemento es un intervalo cuyo inicio y fin coincide.

Las operaciones que se deben soportar son unión, intersección, complemento y resta, a partir de esto, y pensando en como implementar cada una, decidí que era necesario que el árbol contenga intervalos disjuntos. Los principales motivos son:

Se ahorra memoria, al no tener datos repetidos almacenados.

La operación de intersecar un intervalo con el árbol es barata por la mejora que se le hizo en el trabajo anterior a la estructura (almacenar mayor de los hijos). Esto conlleva a mejorar las operaciones que utilizan esta función.

Se simplifican las operaciones, por ejemplo, el complemento es fácilmente resolvible con un solo recorrido del árbol. Además, buscar elementos, insertar y recorrer el árbol pasan a tener menor complejidad, al ser menor la cantidad de nodos en cada árbol.

4.1. Estructura elegida

Formalmente la estructura elegida para almacenar y representar los conjuntos es, un Árbol AVL de intervalos, que tiene la invariante de que todos los intervalos tienen intersección nula entre si. Utilicé el árbol de trabajo anterior, cambiado el tipo del intervalo, que antes era double y ahora son int. Además realicé una modificación, antes, el dato del árbol era un puntero a una estructura intervalo, ahora es directamente la estructura intervalo.

Listing 2: Estructura Intervalo

```
1 typedef struct _Intervalo {
2     int inicio;
3     int final;
4 } Intervalo;
```

4.2. Operaciones de conjuntos:

Para implementar las operaciones pedidas, unión, intersección, complemento y resta, pensé en que rango de cantidad de elementos se van a aplicar. Quiero decir, la entrada es por consola, con un ancho de linea máximo determinado en 1024 caracteres, por lo tanto los conjuntos no tendrán una gran cantidad de elementos, y teniendo en cuenta la modificarían del árbol (intervalos disjuntos), la cantidad de nodos del estará en el rango de los cientos. Por ende la implementación de cada una de las operaciones esta pensada para funcionar eficientemente en ese rango.

- Unión:

  La operación unión tiene asociada una función del árbol llamada **itree_union**, la cual toma dos árboles y devuelve un tercero, siendo este la unión de ambos. Para su implementación, la idea intuitiva fue recorrer ambos árboles insertando en otro árbol los nodos, pero de manera disjunta. A priori pareció una buena opción, pero recorrer un árbol e insertar disjunto cada nodo en otro es mas costoso que copiarlo, ya que se debe verificar si cada nodo tiene intersección y además se debe ubicarlo dentro del árbol, teniendo que recorrer parte del mismo. Entonces, para resolver esto, verifico árbol es el alto (al ser balanceado, esto implica que tiene más nodos) y proceder a copiarlo, de esta manera lo duplico y luego inserto en este nuevo árbol los nodos del otro que tiene menor cantidad de nodos.

- Intersección:

  Para la intersección de conjuntos realicé la función **itree_interseccion**, que al igual que itree_union toma dos árboles y devuelve un tercero siendo este la intersección de ambos.

  Un intervalo, puede tener intersección con más de un nodo de un árbol, por ende la implementación de intersección debe recorrer uno de los dos árboles e insertar en el árbol resultado todas las intersecciones de cada nodo del primer árbol en el segundo. Para realizar esto hice una función que toma un intervalo y un árbol y devuelve un AVLTree conformado por todas las intersecciones del intervalo con dicho árbol. Esta función se llama **itree_todas_las_intersecciones**, la misma busca la intersección de un intervalo con un árbol y cuando la encuentra sigue buscando posibles intersecciones en los hijos del nodo con el que interseca. Esto es gracias a la propiedad del árbol de intervalos disjuntos. Así se ahorra un poco de trabajo innecesario.

  Luego se unen todas las intersecciones de los nodos en un árbol resultado.

- Complemento:

  En este caso, realicé una función que recorre el árbol de forma inorder, al ser este un árbol binario de búsqueda, esta forma de recorrerlo, muestra los intervalos el menor al mayor, de esta forma se puede construir los intervalos que se encuentran entre los intervalos del árbol, obteniendo así el complemento el conjunto representado por el árbol.

  Pensé en construir el árbol complemento mientras se recorre el árbol original, pero teniendo en cuenta que la diferencia en el costo de los dos métodos no es muy grande y que se trabajará con árboles con una cantidad no demasiado grande de nodos, decidí dejar la implementación anterior.

- Resta:

  Para implementar la operación, utilicé las funciones anteriormente definidas, teniendo en cuenta la siguiente propiedad: A - B = $A \cap \overline{B}$

5. Compilación y ejecución

El programa se compila simplemente escribiendo el comando **make** en la consola de comandos, luego se procede a ejecutar el archivo **./main**.

Para limpiar los archivos de compilación y el ejecutable, esta a disposición el comando **make clean**.

6. Bibliografía

Gran parte de la realización del trabajo fue investigación, estas son algunas de las fuentes a las que se recurrió.

- Estructuras de Datos con C - Tenenbaum, Langsam, AugensteinURL
- Introduction to algorithms - Thomas H. Cormen
- https://es.wikipedia.org/wiki/%C3%81rbol_de_intervalo
- Teoría proveída por la cátedra.