Trabajo Práctico Final - EDyAI

Valentina Prato

Agosto 2020

Estructuras

conjunto

La estructura base del programa es la estructura *Elemento*, que se halla en el archivo *conjunto.h*. Esta estructura puede puede representar tanto un valor individual como un intervalo, de la siguiente manera:

```
typedef struct _Elemento {
    int* dato;
    unsigned tipo : 1;
    struct _Elemento* sig;
} Elemento;
```

El campo *tipo* es el que indica si el elemento es individual o un intervalo. Como se le asigna 1 bit de memoria, puede tener valor 0 (elemento individual) o 1 (intervalo). Esta representación no es arbitraria, sino que en muchos casos permite operar sobre el último valor de un elemento sin tener que dividir el código por caso, pudiendo directamente acceder utilizando los punteros $elemento \rightarrow dato[elemento \rightarrow tipo]$.

Con esto, se definió la estructura *Conjunto* como una lista simplemente enlazada de estructuras *Elemento*. Esto se consideró una buena opción por las siguientes razones, entre otras:

- Permite crear conjuntos con una cantidad inicialmente desconocida de elementos (especialmente al crear conjuntos por extensión) sin tener que recorrer todo un string previamente, alocar demasiada memoria o tener que realocar
- Permite insertar elementos individuales en cualquier posición de forma simple para mantener un orden menor a mayor
- Facilita el recorrido y comparación entre conjuntos, además de la unión de elementos consecutivos entre ellos, algo que resultaría bastante mas complejo utilizando por ejemplo árboles

- Simple de usar y ahorra memoria en relación a otras estructuras que utilizan mayor cantidad de punteros

Otro detalle importante es la representación del conjunto vacío y el conjunto enteros. El conjunto vacío se implementó de forma simple: como un intervalo donde el valor inicial es mayor al final. El conjunto de todos los enteros, por el otro lado, no es realmente representable. Esto se debe a que, al utilizar valores *int*, existe un valor mínimo y máximo representable (INT_MIN e INT_MAX) y no existe representación de infinito, y por lo tanto el conjunto de enteros se definió como el intervalo entre estos valores. Uno podría indicar que estos sean considerados $-\infty$ y $+\infty$ respectivamente, pero no se puede diferenciar fácilmente el uso de estos valores como infinitos del uso de ellos como los números que representan, y podría resultar confuso si el usuario crea un intervalo con uno de estos valores y el programa lo imprime como infinito. Por esta razón, los límites siempre se imprimen como los valores que guardan. Esta limitación está aclarada en la ayuda del programa.

chash

Para guardar los conjuntos junto con sus nombres, se utiliza una tabla de hash llamada $ConjHash$ definida en *chash.h*. Para esta tabla se adaptó la implementación de hashing doble para trabajar específicamente con claves $char*$ y datos $Conjunto$. De esta forma se minimiza la cantidad de punteros necesarios.

Además, como el intérprete no incluye comando de eliminación de conjuntos individuales, se consideró innecesario el concepto de casillas eliminadas, y se simplificó y optimizó la implementación al no incluirlas. Por ejemplo, al redimensionar o destruir la tabla se la puede recorrer hasta contar la cantidad de elementos indicada en el campo $numElems$, sin tener que seguir buscando para tomar en cuenta recursos ocupados por casillas eliminadas que no están incluidas en dicho valor.

Para realizar el hashing, se buscó utilizar una función que genere pocas colisiones pero que también sea simple, ya que un usuario que desee utilizar el programa generalmente no creará cientos de conjuntos manualmente y por lo tanto teniendo un tamaño de tabla suficiente no es necesario utilizar una función demasiado compleja. La función utilizada (definida en *shell.c*) es una función polinomial rolling hash [1], que tiene la siguiente forma:

$$hash(s) = \sum_{i=0}^{n} s[i] \cdot p^i \ mod\, m$$

En esta función, m es el tamaño de la tabla y p un número primo generalmente cercano a la cantidad de letras diferentes en el string.

En nuestro programa, un nombre puede tener hasta 94 letras (el máximo que acepta el buffer es 98 sin contar el caracter de nueva línea, aunque esto se puede aumentar facilmente en *shell.c*, y el mínimo para el resto del comando es 4, por ejemplo "=1" o "=a|b") y puede incluir casi cualquier letra del alfabeto inglés menos '=' que separa el nombre del comando (también se eliminan espacios de los extremos pero se permiten en el medio). Es decir, 94 letras diferentes

posibles sin incluir '=' o caracteres no imprimibles [2]. Por lo tanto un nombre puede tener hasta 94 letras diferentes, y el primo más cercano es $p = 97$.

Por otro lado, como la cantidad de colisiones es inversamente proporcional al tamaño de la tabla, se suele usar un número primo grande para esto. Generalmente se recomienda $m = 10^9 + 9$, sin embargo como ya se dijo antes un usuario de nuestro programa normalmente no llega a generar siquiera cientos de conjuntos, así que un número tan grande seria una gran pérdida de espacio. Como nuestra tabla usa doble hashing, el valor de m además de ser primo no debe ser divisible por ningún valor obtenido de la función paso para poder recorrer toda la tabla antes de insertar un conjunto en el peor caso. Nuestra función paso (también definida en *shell.c*) devuelve el valor ASCII de la primera letra de la clave, que puede tener casi cualquier valor entre 33 y 126 [2], por lo tanto se eligió un tamaño bastante mayor a eso que disminuya colisiones sin ocupar demasiado espacio innecesario: $m = 211$.

Errores

A continuación se listan los diferentes mensajes de error que se pueden obtener:

- **comando demasiado largo:** si el comando supera la cantidad de letras aceptadas por el buffer
- **comando inválido:** si el comando escaneado no es ninguno de los aceptados por el intérprete
- **comando vacío:** si no se halló comando luego del '='
- **nombre vacío:** si el escaneo de algún nombre de conjunto resultó en una cadena vacía
- **no existe el conjunto [nombre]:** si no se encontró conjunto en la tabla con el nombre ingresado
- **conjunto inválido:** si no se puede crear el conjunto pedido, ya sea por no poder leer algún elemento como número, porque no se lean suficientes elementos o porque alguno esté fuera de rango.

Compilación y Ejecución

La carpeta del programa incluye un archivo Makefile, que compila el programa facilmente. Este contiene diferentes acciones junto con la compilación:

- **make:** La forma más simple de compilarlo. Simplemente compila el programa con el nombre *shell*, que luego se puede ejecutar como *./shell*.
- **make leaks:** Compila el programa y lo corre con Valgrind para chequear pérdidas de memoria y otros errores. Las últimas pruebas de esto dieron 0 pérdidas.

- **make clean:** Elimina el binario.
- **make tests:** Compila el programa y realiza todos los tests del archivo $Test.pdf$ de Comunidades guardando los resultados en los archivos $ladoIzq.res$ y $ladoDer.res$ en la carpeta $tests$. Luego los compara con $diff$. El archivo $ladoDer.test$ tiene dos definiciones de conjuntos mas que $ladoIzq.test$, así que para balancear se agregaron dos definiciones extra al comienzo de $ladoIzq.test$, $A = \{0\}$ y $B = \{0\}$, que luego son reemplazadas por las definiciones reales. Esto evita que $diff$ indique diferencias por falta de comandos.

Dificultades

Una dificultad encontrada al realizar el trabajo fue el ingreso de cadenas de longitud mayor a la aceptada por el buffer. Al ignorar estos casos, el intérprete procesa pedazos de la cadena como comandos separados hasta leerla completa. Luego de probar diferentes métodos, se decidió buscar en cada escaneo el caracter de nueva línea en el buffer, que indica el final de la cadena ingresada. Si no se encuentra, se avisa que la cadena es muy larga y se sigue escaneando hasta encontrar este caracter para limpiar stdin. Luego se pide un nuevo comando.

Otra dificultad fue la conversión de strings a conjuntos por extensión. A diferencia de los conjuntos por comprensión, que tienen un formato con una cantidad de valores definida y se puede usar fácilmente $sscanf$ para obtenerlos o reconocer errores, en conjuntos por extensión la cantidad de elementos es arbitraria por lo cual no se puede utilizar el mismo método. La separación de elementos se resolvió simplemente con un loop que realiza $strtok$ tomando strings separados por coma como elementos, pero este no fue el mayor conflicto. Esto surgió luego al tener que convertir estos strings en enteros. La función más común para esto es $atoi$, pero esta tenía demasiados problemas para nuestra implementación:

- Al convertir un string no numérico retorna 0 en lugar de dar error, haciendo que se deba verificar si el string era realmente 0 o no
- Al convertir un string que comienza con números pero termina con letras, ignora las letras y convierte simplemente los números iniciales. Esto se considero inaceptable ya que, en conjuntos por comprensión, $sscanf$ no lo permite, y se pensó importante que ambos métodos tengan un comportamiento similar
- Al convertir un valor fuera del rango representable por enteros, simplemente retorna un valor diferente. Esto también sucedía al buscar enteros ($\%d$) con $sscanf$, y se consideró inaceptable en ambos casos (en $sscanf$ se remplazó $\%d$ por $\%ld$ y luego se castea el resultado como int si es un entero válido)

Por estas razones, en lugar de *atoi* se creó una nueva función: *string_to_int*. Esta convierte el string en *long* utilizando *strtol* y luego verifica que se haya convertido todo el string (chequeando un puntero a la última posición que se leyó del string) y que el valor pertenece al rango representable por enteros. De ser así, se devuelve un puntero a *int* con el resultado. En caso contrario, se devuelve $NULL$.

Finalmente, posiblemente la mayor dificultad fue la implementación de la resta. Hay varios casos a tener en cuenta (al restar elementos de un conjunto a un elemento de otro este puede quedar en partes separadas o hasta ser eliminado por completo, o tambien un elemento de uno puede restar a varios elementos de otro) y se quería solicitar la mínima cantidad de memoria necesaria (evitar crear elementos que deban ser eliminados o intervalos que deban terminar como valores individuales). Uno de los detalles de la implementación es el uso de un entero auxiliar *ini* para guardar el valor con el que se cree que empezara el próximo elemento a crear teniendo en cuenta lo que se conoce al momento. Toda la implementación comentada se puede ver en mas detalle en *operaciones.c*. Esta función también ayudó al cálculo del complemento, que se definió como la resta entre los enteros y el conjunto que se pide.

References

[1] Hash de strings con rolling hash: `https://cp-algorithms.com/string/string-hashing.html#toc-tgt-6`

[2] Tabla ASCII: `http://www.asciitable.com/`