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

Trabajo Práctico Final

Palumbo, Matías

Universidad Nacional de Rosario
2020

1. Introducción

A continuación, se detallan las decisiones en general y particularidades en el proceso de diseño y desarrollo del Intérprete. El Trabajo Práctico fue realizado en su totalidad en el sistema operativo Mac OS 10.15.6. Se testeó tanto en Mac OS como en Ubuntu 18.04 LTS. Además, se monitoreó y optimizó el manejo de la memoria corriendo el programa con Valgrind en la misma versión de Ubuntu.

2. Estructuras de Datos Utilizadas

2.1. Conjuntos

Se decidió implementar conjuntos de enteros en C mediante árboles AVL de intervalos. Si bien los elementos de cada conjunto son números enteros en su naturaleza, juntar enteros consecutivos en intervalos permite optimizar y unificar completamente la creación de conjuntos por comprensión y por extensión. Al crear un conjunto por comprensión, simplemente se inserta el intervalo correspondiente en un árbol (o no, si el conjunto ingresado es vacío). Por otro lado, al crear por extensión, se comienza insertando los elementos como intervalos unitarios, y la función que inserta elementos en el árbol une intervalos que se intersequen o sean consecutivos, y posteriormente busca repeticiones de algún intervalo contenido en ese en los subárboles. Esto resulta en que, en un árbol dado, todos los intervalos que lo componen son disjuntos dos a dos, no repitiéndose ningún entero. Además, la impresión de un conjunto es óptima, al estar unidos todos los intervalos de enteros consecutivos.

Implementar conjuntos de esta manera tiene la ventaja de que, siendo n la cantidad de nodos que posee el árbol que representa al conjunto, las operaciones de inserción y eliminación son $O(\log n)$ en tiempo, es decir, proporcionales a la altura del árbol. Como estas dos operaciones son las básicas de un árbol binario, la elección de esta estructura presenta una optimización general en el cálculo de operaciones entre conjuntos, en contraposición con, por ejemplo, listas enlazadas.

2.2. Almacenamiento de Conjuntos en Memoria

Se utilizan Tablas Hash para almacenar los conjuntos ingresados en memoria. Las Tablas Hash implementadas presentan las siguientes características:

- El tamaño inicial de la tabla utilizada en el Intérprete es `HASH_TABLE_DEF_SIZE` $= 32 = 2^5$.
- Una vez que el factor de carga de la tabla supera `LIM_FACTOR_CARGA` $= 0{,}75$, se procede a redimensionarla, duplicando la capacidad de la misma.
- Se utiliza una segunda función de hash en caso de que la casilla accedida esté ocupada, la cual devuelve el valor `INTV_HASH_DOBLE` $= 7$.

Se puede observar que como el tamaño inicial de la tabla (y todos los tamaños resultantes de redimensionarla) son potencias de 2, y el intervalo de sondeo es el número primo 7, siempre se recorrerán todas las casillas de la tabla en busca de un lugar disponible. Efectivamente, tomando el tamaño de la tabla $N = 2^r$ (con $r \geq 5$) en un momento dado, sus divisores son de la forma $2^k,\ 0 \leq k \leq r$, todos pares. Entonces, el tamaño de la tabla y `INTV_HASH_DOBLE` son primos relativos, quedando demostrado.

La elección de Tabla Hash como estructura para almacenar los conjuntos permite acceder rápidamente a la casilla con la clave dada, así como también insertar y eliminar conjuntos eficientemente. A su vez, se permite especificar funciones destructoras de claves y datos para manejar adecuadamente la memoria al eliminar elementos y destruir la tabla.

2.3. Estado

Se definen en el archivo de cabecera "estado.h" dos enum útiles para representar las acciones que el intérprete puede seguir. El primero, `enum EstadoInput`, tiene un valor por cada comando válido del intérprete, es decir, cada tipo de creación, imprimir un conjunto, salir, y realizar cada una de las operaciones. El otro, `enum TipoError`, es utilizado para representar los tipos de errores contemplados en el uso del intérprete, explicados más detalladamente en la Sección 3.2.

Adicionalmente, para mantener centralizadas en un mismo lugar las variables necesitadas por el intérprete, se hace uso de la estructura Estado, la cual guarda, entre otras cosas, los conjuntos con los cuales se opera, y el conjunto a guardar en memoria en caso de haberlo.

3. Particularidades del Intérprete

3.1. Formatos Aceptados para los Alias

A continuación se enumeran los comando considerados válidos en el intérprete. Una observación importante es que no se permite crear conjuntos cuyo primer caracter sea `"~"` o `"{"`. Hacerlo resultará en comandos considerados no válidos. En el siguiente listado se asume que los alias de conjuntos creados son válidos.

- Salir: `salir` (sale del intérprete).
- Imprimir un conjunto: `imprimir alias` (si existe `alias`, lo imprime en orden, expresando como intervalo las secciones del conjunto que puedan serlo; los conjuntos vacíos se imprimen como `{}`).
- Creación de un conjunto por extensión: `alias = {1,2,3,4,5}` (el alias seguido de `" = {"` y una sucesión de enteros separados por coma terminada con `"}"`). Los elementos repetidos se ignoran.
- Creación de un conjunto por comprensión: `alias = {x : n1 <= x <= n2}`, donde `x` puede ser cualquier letra, y `n1` y `n2` son enteros. Si `n1` es mayor que `n2`, el conjunto es considerado vacío.

- Complemento de un conjunto: `alias1 = ~alias2` (si existe `alias2`, calcula su complemento y lo almacena con el nombre `alias1`).
- Intersección entre conjuntos: `alias1 = alias2 & alias3` (si existen `alias2` y `alias3`, calcula su intersección y la almacena en `alias1`).
- Unión entre conjuntos: `alias1 = alias2 | alias3` (si existen `alias2` y `alias3`, calcula su unión y la almacena en `alias1`).
- Resta de un conjunto con otro: `alias1 = alias2 - alias3` (si existen `alias2` y `alias3`, calcula `alias2 - alias3`, y lo almacena en `alias1`).
- Igualación de conjuntos: `alias1 = alias2` (si existe `alias2`, crea una copia de él y la almacena en `alias1`).

3.2. Manejo de Errores

`enum TipoError` presenta los siguientes valores, cada uno permitiendo distinguir distintos tipos de errores:

1. `NoError`: Ausencia de error. Utilizado para inicializar y preparar el Estado.
2. `ComandoNoValido`: Contempla el caso en el que no se respete el formato preestablecido.
3. `ConjuntoInexistente`: Contempla el caso en el que el comando sea válido pero los conjuntos a utilizar no existan.
4. `ElementosNoValidos`: Contempla el caso en el que al crear un conjunto se ingrese algún número que supere los límites de enteros en C.

4. Compilación del Programa

Los archivos necesarios en la compilación del programa y sus dependencias se encuentran detallados en el archivo *makefile*. Para compilar el mismo, sólo es necesario correr cualquiera de los siguientes dos comandos en la terminal:

```
make
```

```
make interprete
```

Luego, el intérprete se inicializa con el comando

```
./interprete
```

5. Dificultades Encontradas e Información Adicional

Las dos grandes dificultades encontradas considero que fueron elegir en un principio la estructura para representar conjuntos, y poder idear una buena función de hash para strings. En cuanto a la elección de estructura para conjuntos, las dos principales opciones que consideré fueron árboles AVL y listas enlazadas. Al principio, comencé a diseñar las funciones de la implementación usando listas, porque había algunas cosas concretas que no creía que pudiera implementar con árboles. Sin embargo, al investigar un poco más encontré que los árboles de altura balanceada es una implementación muy utilizada para conjuntos, y considerablemente más eficiente que las listas enlazadas. Después de eso, me adentré más en los problemas que me traían dificultad con los árboles; particularmente, general el complemento de un conjunto.

Diseñando la función, llegué a la conclusión de que había que calcular los intervalos presentes entre nodos consecutivos (según el recorrido in-order) del árbol, y no encontraba manera de hacerlo. En ese momento, cada nodo del árbol guardaba el extremo derecho máximo del árbol del cual era raíz; esto lo había tomado del Trabajo Práctico 2, y resultaba de utilidad al intersecar conjuntos. Entonces, decidí agregar también el mínimo extremo izquierdo a cada nodo, y descubrí que esto hacía posible los cálculos para calcular el complemento, y también optimizaba aún más la intersección entre conjuntos.

Con respecto a las funciones hash de strings, quería balancear simplicidad y velocidad con una buena distribución de las claves, e investigando, descubrí que alcanzar esto era bastante difícil; una función hash que distribuye mejor las claves generalmente conlleva código más extenso y operaciones más caras. En la página citada en el ítem 5 de la Bibliografía, se presenta el algoritmo de hash llamado *djb2*, y utilizando esto en conjunto con la función presentada en K&R en la Sección 6.6, obtuve una función hash simple y rápida de calcular, que además distribuye las claves uniformemente para el propósito que se le da en este trabajo.

6. Bibliografía

- *The C Programming Language*, Kernighan & Ritchie: Sección 6.6 (Table Lookup), Sección 7.4 (Formatted Input).
- *Fundamentos de Algoritmia*, G. Brassard & . Bratley: Sección 3 (Notación Asintótica).
- Hash Table (Wikipedia): `https://en.wikipedia.org/wiki/Hash_table`
- Double Hashing (Wikipedia): `https://en.wikipedia.org/wiki/Double_hashing`
- Ejemplos de funciones hash: `http://www.cse.yorku.ca/~oz/hash.html`
- Sets (Wikipedia): `https://en.wikipedia.org/wiki/Set_(abstract_data_type)`