UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Análisis de algoritmos y estructura de datos

Laboratorio 4 – Tabla Hash

Alberto Rodríguez Z.

Profesor:	Pablo Schwarzenberg Riveros
Ayudantes:	Javiera Torres
	Javiera Sáez
	Diego Opazo

Santiago - Chile

1-2018

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

CAPÍTULO 1. INTRODUCCIÓN

Uno de los contenidos dentro del curso Algoritmos y Estructura de datos son las tablas de hash, El *hashing* es una técnica ampliamente utilizada para la implementación de diccionarios, son un conjunto de datos para los que solo se requiere contar con funciones para saber si un elemento es miembro de esta tabla y para realizar inserciones y eliminaciones de elementos. Al emplear esta estructura de datos se distinguen dos formas distintas de *hashing: Hashing* abierto y *Hashing* cerrado.

Este laboratorio cuenta con un conjunto de datos de tipo entero que representan un nuevo identificador para la población de un país. Se tiene que implementar un programa que sea eficiente, por lo que se debe implementar una tabla hash abierta con 997 clases y en cada clase se debe implementar una lista simplemente enlazada y un árbol AVL para almacenar los elementos. Para después comparar el rendimiento de cada tabla hash con la estructura de datos usada en cada uno. Se deben reportar gráficos de comparación.

Para lograrlo se utilizarán las tablas de hash abierta, estas permiten que el conjunto sea almacenado en un espacio potencialmente infinito, no se establece un límite para el tamaño del conjunto de datos.

También se trabajará con el lenguaje de programación C, específicamente con ANSI C (estándar de lenguaje publicado por el Instituto Nacional Estadounidense de Estándares). Se trabajará con dos algoritmos uno con una tabla hash implementando las clases con listas enlazadas y otra implementando las clases con árboles AVL.

A continuación, se mencionarán los objetivos de este informe junto a la descripción del problema y como se abordó para solucionarlo.

Objetivos

1.1 Generales:

El propósito central de este proyecto es la implementación de listas enlazadas y árboles AVL en las clases de una tabla hash, para después comparar sus rendimientos.

1.2 Específicos:

1. Aplicación de conceptos clave de tablas hash aprendidos en clase, para poder trabajarlos con listas enlazadas y árboles AVL.
2. Estudiar los algoritmos desde variados aspectos, desde la representación a través de los tipos de datos abstractos hasta los tiempos de ejecución de aquellos algoritmos.
3. Elaborar un programa que respete los estándares ANSI C con el fin de crear un programa eficiente, óptimo y estable.

CAPÍTULO 2. DESCRIPCIÓN DE LA SOLUCIÓN

A continuación, se explicarán las herramientas, técnicas y algoritmos empleados para la elaboración de una solución además del material investigado el cual sirvió como base para resolver el problema.

2.1 Marco teórico

Dentro de los conceptos claves de este informe se encuentran las tablas de hash, estas son estructuras de datos que se asocia a llaves o claves con valores. Es un conjunto de datos que solo requieren contar una clave originada por una función hash de la tabla. La función más eficiente es la de búsqueda, ya que gracias a la clave se puede acceder a una clase específica.

Para emplear esta estructura de datos existen dos formas distintas de hashing:

2.1.1 Hashing abierto

Permite que un conjunto sea almacenado en un espacio potencialmente infinito, por lo que no se establece un límite para el tamaño del conjunto de datos. La tabla hash contiene b clases enumeradas de 0 hasta el b-1.

Se espera que cada celda contenga una cantidad similar de elementos, de modo que cada lista sea corta. Si el conjunto total tiene n elementos, entonces la celda promedio tendrá n/b elementos.

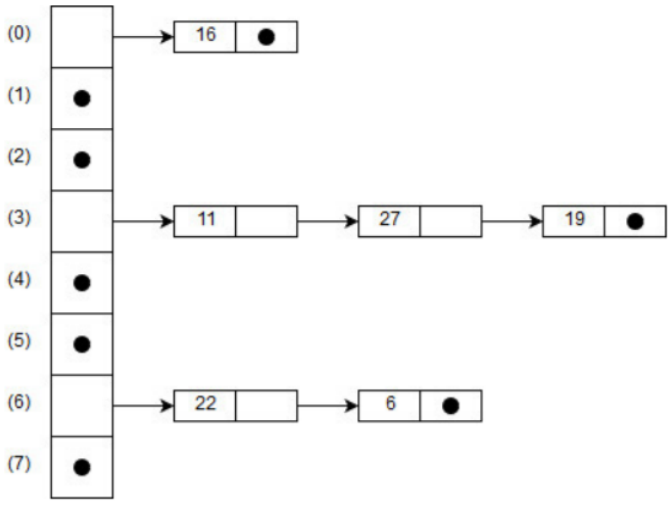

Fig 2.1.1.1 Ejemplo de una tabla hash abierta implementada con listas enlazadas.

2.1.2 Hashing cerrado

Tabla que almacena los elementos del conjunto en la tabla de celdas mismas en lugar de usar la tabla para almacenar una lista de cabeceras. En consecuencia, solo es posible almacenar un elemento en cada celda de la tabla. Para evitar si se ingresa un segundo elemento en una celda, es posible utilizar una estrategia de *rehash*, la cual escoge una serie de posibles de h(x) para el elemento x dentro de la tabla de celdas.

Operación(T, x)	h(x)	tabla[0]	tabla[1]	tabla[2]	tabla[3]	tabla[4]
INSERTAR(T, 5)	0	5	VACÍO	VACÍO	VACÍO	VACÍO
INSERTAR(T, 12)	1	5	12	VACÍO	VACÍO	VACÍO
INSERTAR(T, 42)	1	5	12	42	VACÍO	VACÍO
INSERTAR(T, 8)	4	5	12	42	VACÍO	8
BORRAR(T, 12)	1	5	BORRADO	42	VACÍO	8
BORRAR(T, 5)	0	BORRADO	BORRADO	42	VACÍO	8
INSERTAR(T, 37)	1	BORRADO	37	42	VACÍO	8
BORRAR(T, 8)	4	BORRADO	37	42	VACÍO	BORRADO
INSERTAR(T, 24)	2	BORRADO	37	42	24	BORRADO
INSERTAR(T, 0)	0	0	37	42	24	BORRADO
INSERTAR(T, -6)	3	0	37	42	24	-6
BORRAR(T, 24)	2	0	37	42	BORRADO	-6

Fig 2.1.2.1 Ejemplo de una tabla hash cerrada.

2.2 Herramientas y Técnicas

Las tablas de hash se suelen implementar sobre vectores de una dimensión, aunque tambien puede ser en vectores multi-dimensionales basada en varias clases. Como en el caso de los arrays, las tablas hash proveen un tiempo de búsqueda constante de 0(n) en el peor caso sin importar el numero de elementos de la tabla. Para esta ocasión se implementó una tabla hash mediante arreglos de una dimensión, donde cada posición de este arreglo corresponde a una clase de la tabla, y cada clase apunta a una lista o árbol dependiendo de cual de los dos algoritmos se ve.

Una técnica utilizada fue poder generar la función h(x) mediante una función que obtiene un número aleatorio del 0 al 996 el cual corresponda a la clase en donde irá ese elemento. Todos los arreglos de la tabla apuntaran a nulo excepto donde se agregue a lo menos un elemento.

Una herramienta importante fue como trabajar con “time.h” de la biblioteca estándar del lenguaje de programación C que contiene funciones para manipular y formatear la fecha y hora del sistema. En este laboratorio se usa de forma de cronometro para detectar el tiempo de ejecución de cada operación y para obtener el numero aleatorio que será posteriormente la clase. Las funciones utilizadas son:

- Srand(time(NULL)): función para obtener el numero aleatorio.
- Clock_t clock (void): Devuelve el numero de pulsos del reloj desde que se inició el proceso hasta que termina. Para este laboratorio, el resultado de esta función es un flotante ya que entrega el resultados en segundos.

2.3 Algoritmos y estructura de datos

Estructuras:

Se crean 3 estructuras, una esta incluida en los dos algoritmos, esta es la estructura de la tabla hash, la cual representa lo que contiene cada clase de la tabla, la estructura contiene el número de la clase, o sea el "hash[numero]", la cantidad de nodo de esa clase y a la estructura que apunta, en este caso puede apuntar a una lista enlazada o a un árbol AVL.

```
typedef struct Tabla{
    int numeroClase;
    int cantidadNodo;
    struct Arbol *puntero;
}Hash;

Hash *hash;
```

Fig 2.3.1 Representación de la estructura de la tabla hash.

Las otras estructuras son la de listas enlazadas y la de los árboles AVL. La estructura de listas guarda lo que contiene cada nodo de la lista, es decir, el dato y el puntero al nodo siguiente (que puede ser nulo).

```
typedef struct Lista{
    int dato;
    struct Lista *siguiente;
}List;
```

Fig 2.3.2 Representación de la estructura de la lista enlazada.

La estructura del árbol AVL guarda lo que contiene cada nodo del árbol, es decir, el número del nodo, su equilibrio (que indica si esta equilibrado o no ese nodo) y 3 punteros, uno al padre, otro al hijo izquierdo y otro al hijo derecho, en el caso de que no tenga alguno de estos, el puntero apuntará a nulo.

```
typedef struct Arbol{
    int numeroNodo;
    int equilibrio; //diferencia entre la altura del subarbol izquierdo y derecha de un nodo
    struct Arbol *derecho;
    struct Arbol *izquierdo;
    struct Arbol *padre;
}Nodo;
```

Fig 2.3.3 Representación de la estructura del árbol AVL.

Tabla Hash con lista enlazada:

Algoritmos:

- *void leerTextos( ):*

Función que lee los dos archivos ingresados por el usuario, o sea "datos.txt" que contiene los datos a agregar en las listas de las clases y "operaciones.txt" que contiene las operaciones a realizar en la tabla hash. La salida de esta función es un archivo con las respuestas de cada operación, es decir, si se puedo realizar la operación y el tiempo que demoró en ejecutarse.

Tiempo de ejecución: T(n) = n ^2 + 4n + c
Orden de complejidad O(n^2).

- *void agregarLista(int aleatorio,int dato):*

Función que agrega un nodo a la lista de una clase específica, dependiendo si es el primero nodo en agregar o no.

Tiempo de ejecución: T(n) = n + cn + c
Orden de complejidad: O(n).

- *void buscar(int valor,FILE *archivoSalida):*

Función que busca en una clase determinada el dato ingresado como parámetro, si esta se encuentra escribirá un TRUE en el archivo, sino un FALSE.

Tiempo de ejecución: T(n) = n ^2 + cn+ c
Orden de complejidad: O(n^2).

- *void borrar(int valor,FILE *archivoSalida):*

  Función que borra el nodo donde se encuentra el dato solicitado. En caso del dato está incluido en más de una clase, borrará el primero que encuentre. Si lo borra escribirá un TRUE en el archivo, sino no lo borra (significa que no lo encuentra) entonces escribirá un FALSE.

  Tiempo de ejecución: T(n) = n^2 + cn + c
  Orden de complejidad: O(n^2).

Tabla Hash con árboles AVL:

Algoritmos:

- *void leerTextos( ):*

Función que lee los dos archivos ingresados por el usuario, o sea "datos.txt" que contiene los datos a agregar en los árboles de las clases y "operaciones.txt" que contiene las

operaciones a realizar en la tabla hash. La salida de esta función es un archivo con las respuestas de cada operación, es decir, si se puedo realizar la operación y el tiempo que demoró en ejecutarse.

Tiempo de ejecución: T(n) = n ^2 + 4n + c
Orden de complejidad O(n^2).

- *void agregarArbol(Nodo **raiz,Nodo *nodoNuevo,int numero,int aleatorio)*

Función que agrega un nodo al árbol, primero verifica si ya tiene como mínimo una raíz, después si es mayor o menos que el nodo padre, para ver si es su hijo izquierdo (si es menor) o derecho (si es mayor).

Tiempo de ejecución T(n) = cn+c.
Orden de complejidad: O(n).

- *void buscar(Ficha *inicio)*

Función que busca el dato en el árbol de la clase específica, si lo encuentra escribe en el archivo TRUE sino escribe FALSE.

Tiempo de ejecución: T(n) = n^2 + cn +c.
Orden de complejidad: O(n^2).

- *void eliminar(int valor.FILE *archivoSalida)*

Función que elimina algún nodo del árbol, este nodo es indicado por el usuario. Ya eliminado el nodo la función reordena el árbol, para eso cambia los hijos del nodo eliminado a otro nodo, para después ir a la función *equilibrar* para equilibrar el árbol.

Tiempo de ejecución: T(n) = n^2 + cn + c.
Orden de complejidad: O(n^2).

- *void equilibrar(Nodo **raíz,Nodo *newNodo,int rama,int operación,int aleatorio)*

  Función que verifica si el árbol esta equilibrado o no, si este no está equilibrado lo envía a una de las 4 funciones de rotación según su caso de desequilibrio.

  Tiempo de ejecución: T(n) = cn +c

  Orden de complejidad: O(n).

2.4 ANÁLISIS DE LOS RESULTADOS

Al trabajar en el lenguaje de programación C, una de sus características es el manejo de memoria a través de punteros, la cual para objetivos de este laboratorio nos permite realizar un problema que aparte de resolver el problema, sea mas óptimo en cuanto a tiempos de ejecución, para eso se analizó a través de formulas que permiten calcular cuanto se demora cada ciclo. Con estas formulas podemos determinar la eficiencia de nuestro programa.

Una de las funciones mas importantes del programa es la que leer ambos textos “leerTextos”, ya que debe leer guardar los datos de ambos archivos en las estructuras para después trabajar con ellas. Probando con archivos se consigue tiempo de prueba efectivo, realiza la lectura de forma rápida dando un orden de complejidad óptimo de O(n^2).

Las funciones más importantes para ambos algoritmos son la de insertar, buscar y eliminar, ya que con estas se midieron el tiempo de ejecución para cada una de estas operaciones.

Estas al ser las funciones mas importantes y conociendo el orden de complejidad de cada una respecto a cada estructura (por lo visto en cátedra), se realizaron varias pruebas. Primero se ingresaron 100 datos en las listas y árboles de la tabla hash, y al calcular las operaciones no se vieron mayores diferencias en los tiempos de ejecución. Después se ingresó un archivo que contiene 1 millón de datos, los cuales se ingresaron en las listas y árboles de la tabla con el que se realizaron las 3 operaciones.

Primero se ingresó un archivo llamado “operaciones.txt” que contiene 27 operaciones buscar, 27 de eliminar y 27 de insertar. Con los cuales se calculó el tiempo de ejecución de cada una de las 27 operaciones de cada operación, para después sacar el promedio de tiempo de cada operación:

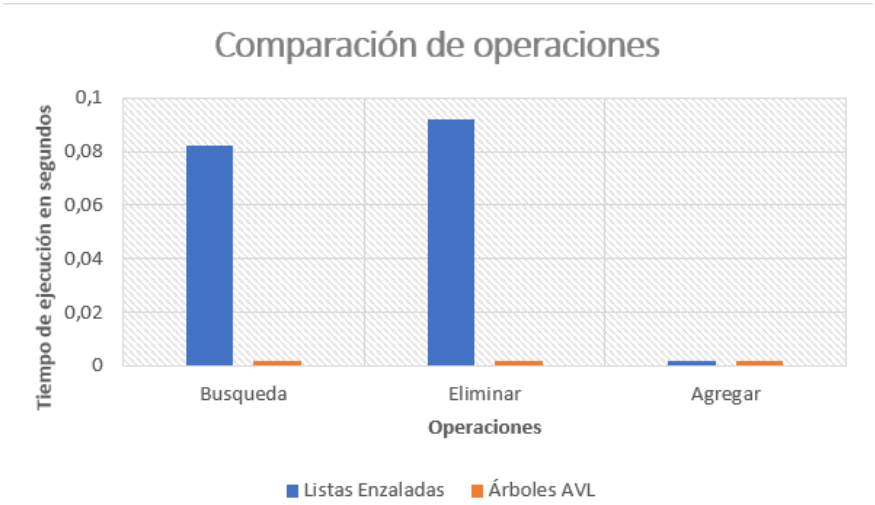

Fig 2.4.1 Grafico de comparación de operaciones.

Como se puede observar en el gráfico, el promedio de tiempo de búsqueda de las listas es bastante más largo que el de los árboles, es decir se cumple lo dicho en cátedra que el tiempo de ejecución de los arboles AVL es mas eficiente que el de las listas enlazadas. En la operación eliminar se puede observar que en las listas enlazadas demora más en eliminar que en los árboles AVL. Terminando en la operación insertar tienen casi el mismo tiempo de ejecución ambas, pero un poco más corta la estructura de listas.

Con lo evaluado anteriormente pudimos verificar que si se cumplen los contenidos vistos en catedra respecto a los tiempos de ejecución de cada estructura de datos. Aunque en la operación eliminar puede que se cumpla de forma distinta.

Retomando los objetivos, podemos ver un logro satisfactorio, implementando correctamente un programa que aborda el enunciado que se pide, excepto como mencionado anteriormente la operación eliminar que en listas dio un tiempo de ejecución más elevado que el de árboles, opuesto a lo mencionado en cátedra.

Dentro de los objetivos específicos podemos ver como logramos estudiar los conceptos claves de tablas hash implementándolo con listas enlazadas y árboles AVL, desarrollando un programa que cumpla estos tipos de datos correctamente, logrando cumplir la gran mayoría del programa inicial. También este programa cumple nuestro objetivo de respetar los estándares de ANSI C compilando en entornos de Windows y Linux. Este programa cumple ser óptimo y estable para recibir al usuario y permitirle el uso de las herramientas creadas.

CAPÍTULO 3. CONCLUSIÓN

Como se mencionó anteriormente en el análisis, el laboratorio tuvo un logro bastante alto ya que se logró estudiar en profundidad y con el tiempo de las tablas hash con sus implementaciones respectivas, logrando plantear algoritmos necesarios para su operación y de manera exitosa, sin errores o fallas, logrando resolver la mayoría de los problemas que eran planteados en el enunciado, es decir logrando comparar las 2 estructuras de datos implementadas.

Gracias a este presente laboratorio fue posible verificar en la praxis como la implementación de las estructura de listas enlazas y árboles AVL dentro de las clases de una tabla hash y en conjunto con las operaciones de buscar, eliminar e insertar sirve para realizar programas eficientes y óptimos, recalcando que un orden de complejidad O(n^2) es uno de los resultados tal vez no más óptimos al ser n-exponencial.

A futuro se espera poder realizar un manual de usuario mas completo ya que si bien se aborda todo lo especificado, siempre se podrían detallarse mejor las instrucciones de compilado y uso del programa juntos a sus limitaciones.

REFERENCIAS

Ryan McVay (2015). *Cómo hacer un gráfico lineal que compare dos cosas en Excel.* (Estados Unidos) Recuperado de: https://techlandia.com/grafico-lineal-compare-cosas-excel-como_187063/.

Daniel López (2011). *Tablas Hash.* (España) Recuperado de: http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/tablash.html.

Jacqueline Köhler Casasempere (2017). *Apuntes de la asignatura Análisis de Algoritmos y Estructuras de Datos*. Chile, Departamento de ingeniería informática (DINF), Usach.