Práctica 5 AED

Ismael Caamano Figueiras, Mateo Díaz Pérez

Resumen

Una estrategia de recolocación cuadrática para una tabla hash con función de suma ponderada de caracteres que componen la clave, resultó ser la apuesta más equilibrada para minimizar el número de comparaciones (aprovechando la complejidad ***O(1)*** de las tablas hash) y el uso de memoria para guardar una muestra de 8000 usuario de nuestra base de datos de juego de tronos.

Palabras clave: *función hash, factor de carga, recolocación, encadenamiento, colisión, clave, lineal, cuadrática, hash, direccionamiento directo, modularidad, coprimos, complejidad computacional*

1. Introduccion

En el siguiente informe se analizará el comportamiento de las funciones de inserción y búsqueda de dos implementaciones de la **tablas hash**, estructuras de datos con tiempo de acceso constante . De este modo, resultan muy útiles para almacenar información, siendo útiles, por ejemplo en bases de datos.

En este caso, se relizará un estudio del número de **colisiones** (intento de inserción en una posición ocupada por un dato) y pasos extra (en recolocación y búsqueda), para 8000 entradas de datos probando con diferentes tamaños para la tabla, funciones y técnicas algorítmicas.

2. Comparativa para el proceso de Insercción

En esta primera sección se abordará la comparativa para la operación de inserción de datos en la tabla hash realizando varias comparaciones modificando varios parámetros como la función hash a utilizar, el factor de carga y la elección de un número coprimo o no comprimo con el tamaño de la tabla, así como también la comparativa entre los dos algoritmos de solución de colisiones, cuyo funcionamiento explicaré a continuación:

- **Encadenamiento:** La tabla hash está formada por listas, de tal manera que cada vez que se produce una colisión el nuevo dato pasa a insertarse en esa lista. Por tanto, no hay operaciones extras para encontrar el hueco en donde colocar el dato, pero la tabla ocupa más memoria y a la hora de la búsqueda hay que hacer operaciones extra para búscar el dato dentro de la lista.
- **Recolocación:** Consiste en analizar la posición en la que se debería insertar el dato y cuando esté ocupada se busca otra avanzando un valor constante (lineal) o variable (cuadrática) de posiciones cada vez que no encuentre un hueco donde insertar el elemento. Esto se repite también en la búsqueda.

Una **colisión** se define como el acontecimiento en el que tras pasar por una función hash dos claves distintas reciben el mismo **hash** (valor número que identifica la posición de la tabla donde se debe insertar si no existe o buscar si ya fue insertado) y por lo tanto la estructura debe elegir como gestionar que dato ocupa esa posición. Por lo tanto es importante tener una función hash que disperse los datos por toda la tabla de forma uniforme para minimizar las colisiones ya que así se aprovechará al máximo el **acceso directo** de la tabla hash.

La función a utilizar también será determinante ya que es la que transforma las claves en cada correspondiente hash, o lo que es equivalente, nos devuelve el `i` para poder acceder al dato en su posición sencillamente a través de `T[i]` (siendo `T` nuestra tabla hash). Ya que una colisión nos obligará a hacer más comparaciones (mayor coste computacional) o reservar más memoria. El análisis de las diferentes funciones hash proporcionadas se verá en más detalle en la **Subsección 2.2**.

2.1. Comparativa de algoritmos con Hash2

En la primera comparativa utilizaremos una función hash (***Hash2()***) que realiza una suma ponderada del valor `ASCII` de cada caracter.

```
suma = (suma * 256 + cad[i]) % N
```

Figura 1: Implementación función ***Hash2()***

Podría verse también como una suma del siguiente polinomio:

$256^{m-1}c_1 + 256^{m-2}c_2 + \cdots + c_m$, Donde m es la longitud del mensaje.

Al cual luego se le aplicará el módulo $\%N$, por tanto si $gcd(256, N) \gg 1$, es decir tienen muchos factores comunes, dichas potencias irán reducíendose hacia valores muy similares, en el caso extremo si $N \equiv 256^x$, luego algunos caracteres no se tendrán en cuenta para el cálculo del **hash**.

Obsérvese en las siguientes tablas que los valores primos de N generan mejores resultados:

HASH2	L = 0,5		L = 0,75		L = 0,8	
N	16000	16001	10600	10601	10000	10007
nColisiones	7875	1707	6676	2391	7375	2416

Cuadro 1: Hash2 Encadenamiento

HASH2	L = 0,4		L = 0,5		L = 0,75	
N	20000	20011	16000	16001	10600	10601
nColisiones	7375	1648	7875	2011	6678	3025
Pasos extra	51350	2894	255982	3838	29113	12682

Cuadro 2: Hash2 Recolocación simple

Para cada **factor de carga** L se han asignados dos tamaños de tabla N, uno con muchos divisores comunes con n y otro número primo, donde n es el total de datos del experimento, en este caso 8000.

Además cabe destacar que los factores de carga de ambas tablas son distintos porque el factor de carga óptimo para Encadenamiento suele ser $L = 0,75$, mientras que para Recolocación suele ser $L = 0,5$, lo que justifica elegir en Cuadro 1 elegir un valor menor y otro mayor al valor central, el óptimo $(0,75)$, mientras en Cuadro 2 el valor central $(0,5)$ y sus valores menor y mayor son diferentes.

Además se puede observar que en recolocación la elección del tamaño de la tabla es mucho más relevante, ya que cada inserción resulta más costosa en tiempo de computo (debido a que los Pasos Extra a veces se disparan).

2.1.1. Conclusión

Por tanto la mejor opción en este caso para minimar el número de colisiones y pasos extra será elegir el mayor tamaño de tabla (siendo este un número primo), para que no comparta divisores con la con la constante de la función hash (en este caso 256) si queremos la mayor eficiencia computacional, o buscar una **alternativa intermedia** con un factor de carga superior si nuestro objetivo es balancear el consumo de memoria junto con la eficiencia computacional.

Ya que como resulta intuitivo, cuanto mayor sea la tabla, si la función Hash es buena, menos colisiones se probocarán ya que hay más espacio disponible para el mismo volumen de datos, ahí entra en juego el interesés especifico y la importancia que le demos a cada factor, alta eficiencia o bajo consumo de memoria. Como veremos en el siguiente apartado, una mala función Hash proboca que el factor de carga de la tabla sea irrelevante.

2.2. Impacto de la función Hash

En esta experimentación se observó que la peor función hash es la función ***Hash1()*** porque es la simplificación de la función de la figura 1 para un $k = 1$, por tanto solo se suman el código `ASCII` de todos los caracteres, raramente puede superar el tamaño de la tabla, por lo tanto la operación $\%N$ no cambia el valor del hash devuelto, provocando que los datos no se dispersen casi nada, limitando así el valor devuelto por la función a la longitud m de la clave.

En la función ***Hash2()***, al aumentar el valor de $k = 256$, los datos se dispersan mucho mas debido a que la suma acumulada supera con más facilidad el valor de N . Aún así, se puede observar que donde mejor funciona es para valores de N primos, por lo que si k fuera primo funcionaría mejor para todos los casos, ya que cumpliría la condición de que ***k, N* sean coprimos**.

Para ***Hash3()***, con $k = 500$, el número de colisiones se vuelve a disparar en los N no primos, ya que, aunque se supera el valor de N en la suma acumulada, tiene muchos más divisores en común con ellos que 256, provocando que los datos se dispersen hacia las mismas zonas.

Como valores de k para el último apartado, los escogimos en función del valor de carga, para que se dispersen de forma equilibrada, ya que para un mismo valor de k,en un valor más pequeño de N es más fácil dar más vueltas a la tabla [1] con esta función hash.

El objetivo es que la constante k de la generalización de la función ***Hash2()*** cumpla las dos condiciones siguientes (para k primo más cercano a $N/256$ se garantizan):

- **Coprimo con N:** Para que los coheficientes del polinomio no se reduzcan rápidamente al realizar la operación módulo.
- **Suficientemente grande:** Al elegir este valor nos aseguramos que con un cadena de dos caracteres promedio de `ASCII`, cuya suma promedio será exactamente 256, igualemos el tamaño de la tabla y podamos «dar una vuelta» a la misma.

Se puede observar (ver Cuadro 3 y 4 a continuación) como para valores no primos de N, elegir un k primo es una buena solución, ya que no necesitan comprobar sus divisores comunes con N.

HASH2	L = 0,5		L = 0,75		L = 0,8	
N	16000	16001	10600	10601	10000	10007
nColisiones H1	6409	6409	6409	6409	6409	6409
nColisiones H2	7875	1707	6676	2391	7375	2416
nColisiones H3,k=500	7972	1726	7894	2400	7982	2471
nColisiones H3,k=N/256	1642	1669	2354	2363	2483	2464
	k = 63		k = 41		k=39	

Cuadro 3: Encadenamiento clave alias

HASH2	L = 0,4		L = 0,5		L = 0,75	
N	20000	20011	16000	16001	10600	10601
nColisiones H1	7231	7231	7231	7231	7231	7231
nPasosExtra H1	25568821	25568821	25568821	25568821	25568821	25568821
nColisiones H2	7375	1648	7875	2011	6678	3025
nPasosExtra H2	51350	2894	255982	3838	29113	12682
nColisiones H3,k=500	7976	1573	7972	2025	7892	3015
nPasosExtra H3,k=500	3187039	2647	1660012	4077	302683	12184
nColisiones H3,k=N/256	1554	1621	1958	1959	2993	3018
nPasosExtra H3,k=N/256	2570	2666	3915	3792	12098	12186
	k = 79		k = 63		k=41	

Cuadro 4: Recolocación clave alias

[1] Esto es un descripción informal para expresar que la suma acumulada supera el valor del módulo y por lo tanto se reduce, por eso vuelve a «empezar» desde valores pequeños, lo que se produce de forma cíclica si la suma es muy elevada.

2.3. Impacto de la clave a utilizar

Para comprovar la importancia de seleccionar una clave adecuada, se probó a realizar la experimentación anterior de los Cuadros 3, 4 modificando la clave que anteriormente era el alias que identificaba a un usuario de la base de datos, aproximadamente entre 10 por el correo electrónico con el que se registró, con el formato: *nombre.apellido1.apellido2@jdt.gal*, de una longitud considerablemente mayor (sobre 27 caracteres).

A partir de los cuadros 5 y 6 que se muestran un poco más abajo deberíamos poder observar una premisa que preveíamos anterioremente: Los resultados de la función Hash1 están limitados por la longitud de la clave, puede observarse entonces que como los correos tienen de media más caracteres, sus resultados deberían ser mejores.

Sin embargo, los resultados son un poco engañosos debido al problema de guardado de claves que nos comentó nuestra profesora el otro día, y que debido a que todas las tablas ya estaban completas decidimos manenerlo sin aplicar la modificación:

Opción Incorrecta (Utilizada en este informe)

```
fscanf(fp, "%[^,] , %s , %s", jugador.nombre, jugador.alias, jugador.correo);
```

Opción Correcta

```
fscanf(fp, "%[^,] , %[^,] , %s", jugador.nombre, jugador.alias, jugador.correo);
```

La parte del código mostrada anteriormente no lee bien los datos guardaos en el archivo, ya que se encuentran separados por comas y no espacios en blanco, por tanto el `%s` sigue hasta encontrar un espacio en blanco o caracter de final de cadena `'\n'`, intentando guardar más caracteres de los que son posibles en el campo alias de la estructura del jugador (que solo admite 15 caracteres), provocando que los demás se guarden en las consecutivas posiciones de memoria, en el mejor de los casos serán las correspondientes al correo y el desecho se guardará en el correo. En cualquier otro caso estaremos haciendo la tabla hash con una clave formada por basura de memoria, ya que nada se habrá guardado en el campo correo del jugador.

Puede observarse que como el contenido de los campos de la estructura no han sido inicializados (ya que su alias o su correo no se igualan a 0), su contenido puede ser incierto:

```
void insercionArchivo(params...) {

    TIPOELEMENTO jugador;

    ⋮

}
```

HASH2	L = 0,5		L = 0,75		L = 0,8	
N	16000	16001	10600	10601	10000	10007
nColisiones H1	6564	6564	6564	6564	6564	6564
nColisiones H2	5201	1893	2714	2490	3420	2652
nColisiones H3,k=500	7677	1852	6676	2535	7811	2653
nColisiones H3,k=N/256	1824	1840	2537	2595	2589	2623
	k = 63		k = 41		k=39	

Cuadro 5: Encadenamiento clave correo

Sin embargo ver unos resultados tan similares a los obtenidos con la clave alias puede indicarnos que realmente si se está guardando el residuo que no pudo guardarse en el alias, dentro del correo, haciendo sus longitudes bastante equivalente ya que ***Hash1()*** genera incluso más colisiones, lo que se explica perfectamente si tenemos en cuenta que al no tener un caracter de final de cadena `'\0'`, la condicion de parada de la ***Hash1()*** es `strlen(clave)` [2], por lo tanto leerá también todo el correo electrónico como parte del alias, y por lo tanto al tener más caracteres la distribución en ***Hash1()*** será mejor

[2] `strlen()` Lee hasta el siguiente final de cadena en memoria que es justo el final del correo electrónico.

HASH2	L = 0,4		L = 0,5		L = 0,75	
N	20000	20011	16000	16001	10600	10601
nColisiones H1	7319	7319	7319	7319	7319	7319
nPasosExtra H1	24674412	24674412	24674412	24674412	24674412	24674412
nColisiones H2	3220	1437	5646	1943	3134	2839
nPasosExtra H2	7915	2350	72277	3723	12047	10520
nColisiones H3,k=500	7804	1488	7749	1914	6217	2913
nPasosExtra H3,k=500	2083534	2439	1145125	3996	146265	11514
nColisiones H3,k=N/256	1577	1556	1841	1867	2925	2841
nPasosExtra H3,k=N/256	2649	2472	3539	3664	10640	11136
	k = 79		k = 63		k=41	

Cuadro 6: Recolocación clave correo

2.3.1. Conclusion

Tras pasar de una clave de longitud aproximada de 30 caracteres (10 del alias y 20 del correo) a unos 15 aproximadamente (residuo del correo que quedaba dentro de la variable de alias) no se vieron cambios significativos, por lo que podemos concluír que hasta ahora una buena función hash puede dar mejores resultados y no necesita que se tome la **longitud de la clave** como un factor relevante una vez que se define una función adecuada (que disperse bien los valores).

2.4. Selección de mejores casos

Para este apartado, escogemos como casos favorables la función ***Hash2()*** para $N = 16001$ y Hash 3 con $k = 500$ y $N = 20011$, del proceso de recolocación con clave correo, ya que presentan muy buenos resultados tanto a nivel de colisiones como en número de pasos extra. Como valores de a elegimos 20, por tomar uno mayor que $a = 1$ (recolocación simple) pero que tenga bastantes divisores, y 41, por ser primo. Los números de colisiones no varían prácticamente debido a que los N ya son primos, no obstante, se puede observar claramente que la cantidad de pasos extra se reduce significativamente al usar la función de recolocación cuadrática, no afectando tanto a las colisiones sino a la redistribución de los datos.

Por lo tanto la **recolocación cuadrática** será una alternativa interesante a tener en cuenta en futuras experimentaciones (para poder seguir optimizando el resultado que se obtiene simplemente al cambiar la constante k como hicimos en el apartado 2.2 o el tamaño de la tabla como hicimos en el apartado 2.1 puediendo llegar a mejores resultados).

Casos	Variables	Simple a=1	Lineal a=20	Lineal a=41	Cuadrática
16001/Hash2	nColisiones	1943	1960	1981	1945
	nPasosExtra	3723	3911	3715	3282
20011/Hash3 k=500	nColisiones	1488	1483	1495	1485
	nPasosExtra	2439	2338	2518	2229

Cuadro 7: Recolocación comparación clave correo

3. Comparativa para el proceso de búsqueda (nPasosExtraB)

Para realizar esta experimentación, se volvieron a buscar los 8000 datos del archivo original en la tabla Hash, con el objetivo de ver el número de pasos extra que se han de realizar antes de encontrar el dato (en encadenamiento hay que iterar la lista comparando cada clave y en recolocación hay que avanzar las posiciones correspondientes comparando cada clave).

HASH2	L = 0,5		L = 0,75		L = 0,8	
N	16000	16001	10600	10601	10000	10007
H1	26402	26402	26402	26402	26402	26402
H2	255982	1995	24001	3093	51350	3038
H3,k=500	1595259	2061	302683	2400	2855225	3156
H3,k=N/256	1933	1940	2991	2988	3214	3169
	k = 63		k = 41		k=39	

Cuadro 8: Encadenamiento clave alias

HASH2	L = 0,4		L = 0,5		L = 0,75	
N	20000	20011	16000	16001	10600	10601
H1	25568821	25568821	25568821	25568821	25568821	25568821
H2	51350	2894	255982	3838	29113	12682
H3,k=500	3187039	2647	1660012	4077	302683	12184
H3,k=N/256	2570	2666	3915	3792	12098	12186
Cuadrática H1	397927	397927	397927	397927	397927	397927
Cuadrática H2	45158	2601	57840	3457	36355	8193
Cuadrática H3,n=500	1243202	2378	512084	3445	42816	7968
Cuadrática H3,k=N/256	2359	2378	3399	3323	7962	7602
	k = 79		k = 63		k=41	

Cuadro 9: Recolocación clave alias, numero de pasos extra

- En la tabla de **Encadenamiento** (Cuadro 8), ***Hash1()*** ofrece un valor muy alto e igual para todos los valores de N, (lo mismo ocurre en recolocación), y para el resto de funciones los mejores valores se sitúan mayormente en el valor de carga igual a **0,75** (juntando los valores de N asociados a la misma carga), confirmando que esta es aproximadamente el valor óptimo para una tabla hash con este tipo de estrategia de recolocación. También afecta el tipo de función Hash ultilizado, en este caso la mejor sería Hash 3 con los valores escogidos.
- En **recolocación** (Cuadro 10), como ya se ha especificado, para ***Hash1()*** no afecta el valor de N, de modo que contemplando el resto de resultados. De nuevo, los mejores valores se obtienen con los tamaños siguen números primos, pero, aún así,juntando los primos y no primos próximos, los mejores resultados se obtendrían entre $0,4$ y 0,5, ya que en algunos casos una carga resulte mejor que la otra (en alguno también gana $0,75$).

4. Conclusión general

Según nuestros criterios hemos decidido optar por una estategia bastante segura, la estrategia de recolocación con $L = 0,5$ para un valor de $k = 63$ y una función ***Hash3()*** Cuadrática, ya que el tamaño total de la tabla será solo de $16K$ datos y no utilizar lista enlazadas el consumo de memoria no es muy elevado, además el número de pasos extras es de 3399.

Para apoyar la afirmación anterior hemos recogido a mayores también datos para el número de colisiones que generó cada una de las estrategias de recolocación cuadŕaticas mostradas en la tabla anterior (Cuadro 10):

HASH2	L = 0,4		L = 0,5		L = 0,75	
N	20000	20011	16000	16001	10600	10601
Cuadrática H1	7215	7215	7215	7215	7215	7215
Cuadrática H2	7377	1660	7875	2000	6791	3046
Cuadrática H3,n=500	7976	1583	7972	2008	7894	3024
Cuadrática H3,k=N/256	1553	1627	1968	1951	2989	2982
	k = 79		k = 63		k=41	

Cuadro 10: Colisiones en recolocación cuadrática clave alias

Las colisiones de utilizar recolocación cuadrática no aumentan practicamente nada a utilizar recolocación linea simple, pero los pasos extras disminuyen al movernos más rápidamente por la matriz y hacer saltos de distintos tamaños (1, 4, 9, 16 ,25...), por lo que hacer el cálculo de i para cada iteración de recolocación sale más rentable que realizar un `strcmp()` ya que hay que realizar m (para una cadena de m caracteres) restas de caracteres para ver si dos arrays son iguales.

En cualquier caso cualquier opción similar será bastante adecuada pero decidimos tomar esta por **simplicidad** de tamaño de tabla (para $16KB \times sizeof(dato)$ de reserva nos aseguraríamos que nos caberían todos los datos) y además el coste de operaciones extra y colisiones no supera el 30 % y 20 % del total de datos respectivamente, considerando un 20 % de **ahorro de memoria** frente a la opción de $L = 0,4$.