ANALISIS DE TABLAS HASH PRÁCTICA 5

Modificación del TAD TablaHash y explicación de la implementación de los contadores:

Lo primero que hay que hacer es modificar los TAD de los dos tipos de tablas de forma que implementen dentro los contadores de colisión, inserción y busqueda, mi propuesta para este ejercicio es la siguiente:

```
typedef struct{
   tipo_jugadorjugador;
   int colisionesHueco;
   int pasosAdicionalesBusquedaHueco;
   int pasosAdicionalesInserccionHueco;
}TablaHash[Tam];
```

Para la tabla con recolocación y

```
typedef struct{
   lista jugadores;
   int colisionesHueco;
   int pasosAdicionalesBusquedaHueco;
   int pasosAdicionalesInserccionHueco;
}TablaHash[Tam];
```

Para la tabla con encadenamiento.

Con esta nueva implementación se puede saber cuantas colisiones, pasos adicionales en la inserción y pasos adicionales en la búsqueda se producen en cada hueco y para calcular el total simplemente hay que sumarlos todos. Una vez modificado todo el código para que sea coherente con los nuevos TAD lo que hay que hacer es examinar los códigos de ambos TAD para ver cuando hay que incrementar los contadores, el resultado fue el siguiente:

Hash con recolocación:

Contador de colisiones: El contador se incrementará en la función _PosicionInsertar() dentro del for en caso de que la i sea uno ya que esto indica que la posición que le correspondía al elemento está ocupada y el programa está buscando otra donde meterlo.

Contador de pasos adicionales en la insercción: El contador se incrementará en la función _PosicionInsertar() dentro del for una vez por cada i > 0 ya que como se comenta en el apartado anterior si la i > 0 el hueco que le corresponde al elemento está ocupado y con cada nueva iteración el programa intenta meter el dato en otro hueco.

Diego Baleirón Pais

Contador de pasos adicionales en la búsqueda: El contador se incrementará en la función _posicionBuscar en cada iteración del bucle for excepto la primera, el motivo es el mismo que en el apartado anterior, si i > 0 quiere decir que el elemento no está en la posición que le corresponde y el programa tiene que buscarlo por el array.

Hash con encadenamiento:

Contador de colisiones: En la función InsertarHash antes de ejecutar la función inserta el programa comprueba si la lista está vacía y en caso de que no lo esté se incrementa el contador.

Contador de pasos adicionales inserción: En la función InsertarHash antes de ejecutar la función inserta se comprueba cual es la longitud de la lista y el programa le suma al contador ese valor.

Contador de pasos adicionales búsqueda: En la función Busqueda dentro del while se incrementa el contador en una unidad por cada iteración en la que la variable p sea distinta del inicio de la lista ya que esta es la posición que le correspondería al dato.

Comprobación de la influencia en el tamaño de las tablas:

Tabla Hash con recolocación:

- El primer tamaño elegido para la tabla fue 20011 un buen tamaño ya que es un número primo y además el factor de carga de la tabla será aproximadamente 0.5. El resultado fue 2588 colisiones y 5548 operaciones adicionales de inserción.

- El segundo tamaño elegido para la tabla fue 50000 para probar la influencia de los números no primos en el tamaño de las tablas. El resultado fue 7010 colisiones y 15956 operaciones adicionales de inserción.

- El tercer tamaño fue 53437 el cual según la teoría es un muy buen valor ya que además de ser un número primo mantiene el factor de carga de la tabla muy bajo. El resultado fue 881 colisiones y 1073 operaciones adicionales de inserción.

- El cuarto tamaño fue 15111 un valor muy malo ya que mantiene un factor de carga alto y además no es primo. El resultado fue 3376 colisiones y 9695 operaciones adicionales de inserción.

- El último tamaño para la tabla con recolocacion fue 15107 un valor que provoca que la tabla tenga un factor de carga relativamente alto pero es primo. El resultado fue 3255 colisiones y 9349 operaciones adicionales de inserción.

Tabla Hash con encaminamiento (mismos valores para los tamaños que en las tablas con recolocación) :

Tamaño tabla	Colisiones	Operaciones adicionales insercción
2011	2201	2608
50000	7010	15957
53437	825	873
15111	2695	3333
15107	2679	3303

Con estos resultados se puede comprobar lo que dice la teoría para ambos tipos de tablas, generalmente para tamaños de tabla mayores habrá menos colisiones y menos operaciones adicionales de inserción (ya que hay un espacio mayor por el que repartir los datos por la tabla) a menos que se usen tamaños que generen muchas claves iguales como 1000,2000,50000 ya que muchos valores irán a parar al mismo sitio en la tabla, además en relación con esto último que acabamos de decir poner usar números primos para el tamaño ayudará a dispersar mejor la tabla y por tanto a provocar menos colisiones y operaciones de reinserción.

Comprobación de la influencia de la función hash:

i)
Tablas hash con recolocación:

Tamaño tabla:	Colisiones función 1	Colisiones función 2	Colisiones función 3 (k = 500)	Colisiones función 3 (k= 7477)
20011	9785	2588	2507	2484
50000	9785	7010	9980	983
53437	9785	881	995	917
15111	9785	3376	3381	3265
15107	9785	3255	3339	3287

Tablas con encadenamiento:

Tamaño tabla:	Colisiones función 1	Colisiones función 2	Colisiones función 3 (k = 500)	Colisiones función 3 (k= 7477)
20011	9272	2051	2105	2106
50000	9272	7010	9980	931
53437	9272	825	900	848
15111	9272	2695	2681	2744
15107	9272	2679	2708	2731

Una vez comprobado el valor de las colisiones para los distintos tamaños se puede decir que la primera función es muy mala para este tamaño de tabla porque todos los nicks tienen un tamaño aproximado por lo que la mayoría de resultados de la función estarán concentrados en un parte de la tabla y además muchas personas comparten nombre o apellidos haciendo que aún se apiñen más los datos. Las funciones 2 y 3 solucionan los problemas anteriores, por una parte se soluciona el problema de los valores centralizados ya que el producto “suma = (suma*K+cad[i])%Tam” permite alcanzar a la variable suma valores muy altos y por tanto la función es capaz de repartir elementos por toda la tabla además con esta implementación no solo importan los caracteres de la clave sino también su posición por lo que se soluciona el problema del apiñamiento de datos con claves permutadas.

Dicho esto siempre que en la función 3 la variable K tenga un valor suficientemente alto ambas funciones tendrán un rendimiento similar para estos tamaños de tabla, sin embargo con tablas mucho más grandes que esta, la función 3 sería una mejor opción, ya que es posible que la función 2 produzca un apiñamiento debido a que su valor para K es fijo y no muy elevado.

ii) Cambio del campo clave, en los siguientes ejemplos se usa el campo correo

Tabla con recolocación:

Tamaño tabla:	Colisiones función 1	Colisiones función 2	Colisiones función 3 (k = 500)	Colisiones función 3 (k= 7477)
20011	9486	2514	2461	2544
50000	9486	1517	9953	970
53437	9486	941	964	917
15111	9486	3328	3338	3287
15107	9486	3334	3300	3279

Tabla con encadenamiento:

Tamaño tabla:	Colisiones función 1	Colisiones función 2	Colisiones función 3 (k = 500)	Colisiones función 3 (k= 7477)
20011	8788	2139	2117	2153
50000	8788	1437	9947	914
53437	8788	888	910	865
15111	8788	2717	2695	2671
15107	8788	2715	3344	2687

En este sub apartado se ve que el cambio en la clave no fue muy relevante excepto en la función 1 ya que el campo correo es más largo que el campo nick y por tanto la función puede colocar los datos en un rango más amplio, si se usan el resto de funciones no vale la pena cambiar de clavee incluso en algunos casos es perjudicial.

Cabe destacar el mal desempeño que tiene la función tres con k = 500 y tamaño tabla = 50000, la similitud entre ambos números causa que la función devuelva valores muy parecidos para todas las entradas provocando que la función 3 tenga un rendimiento aún peor que la función 1, sin embargo este problema es fácil de solucionar, basta con asignarle a k valores primos.

Comprobación de la influencia de la recolocación en las búsquedas:

Para está comprobación se usan tablas hash con recolocación (no tiene sentido comprobar las tablas con encadenamiento ya que no hay recolocación de elementos) con tamaño tabla = 53437 y la función de hash 2.

Número colisiones con recolocación simple	Número colisiones con recolocación lineal (a = 5)	Número colisiones con recolocación lineal (a = 25)	Número colisiones con recolocación cuadrática
881	879	876	875

Viendo los resultados se puede deducir que para este caso concreto la recolocación cuadrática es la más eficiente, pero esto se podía predecir viendo el factor de carga de la tabla, las recolocaciones simple y lineal (con un valor para la variable “a” pequeño) generalmente forman bloques en el interior de la tabla mientras que la recolocación cuadrática es capaz de desperdigar mejor los elementos en la tabla por lo que la probabilidad de ocupar la posición de otro elemento es menor (ya que que se formen bloques quiere decir que hay una gran cantidad de elementos que ocupan posiciones muy cercanas, y las recolocaciones lineales y simples colocan los elementos lo más cerca posible de esos bloques) , el inconveniente de la recolocación cuadrática es que a veces es imposible encontrar posiciones donde poder recolocar las entradas de la tabla pero este problema se soluciona manteniendo un factor de carga <= 0.5 condición que cumple esta tabla de sobra.

Comprobación del numero de acciones adicionales en búsquedas para distinto tamaño, estrategia de recolocación y función de hashing:

Tablas hash con recolocación simple:

Tamaño tabla:	Operaciones adicionales búsqueda función 1 /Tiempo en ms	Operaciones adicionales búsqueda función 2 / Tiempo en ms	Operaciones adicionales búsqueda función 3 (k = 500) / Tiempo en ms
20011	43608695 / 0.550712	5326 / 0.006509	4897 / 0.009567
50000	43608695 / 0.627046	7014 / 0.006060	9981 / 0.005486
53437	43608695 / 0.547282	1072 / 0.006553	1171 / 0.006383
15111	43608695 / 0.563505	10009 / 0.006286	9431 / 0.006248
15107	43608695 / 0.627553	9231 / 0.006014	9598 / 0.006363

Tablas hash con recolocación cuadrática:

Tamaño tabla:	Operaciones adicionales búsqueda función 1 /Tiempo en ms	Operaciones adicionales búsqueda función 2 / Tiempo en ms	Operaciones adicionales búsqueda función 3 (k = 500) / Tiempo en ms
20011	635113 / 0.015134	4585 / 0.006252	4343 / 0.006270
50000	635113 / 0.024805	21014 / 0.006845	7383125 / 0.165810
53437	635113 / 0.016612	1041 / 0.006288	1145 / 0.006456
15111	635113 / 0.019050	6910 / 0.006860	7084 / 0.006402
15107	635113 / 0.017172	7099 / 0.006426	6984 / 0.010293

Tablas hash con encadenamiento

Tamaño tabla:	Operaciones adicionales búsqueda función 1 /Tiempo en ms	Operaciones adicionales búsqueda función 2 / Tiempo en ms	Operaciones adicionales búsqueda función 3 (k = 500) / Tiempo en ms
20011	258356 / 0.012440	5216 / 0.007230	4936 / 0.011237
50000	258356 / 0.010037	31911 / 0.008231	31911 / 0.008231
53437	258356 / 0.015594	1746 / 0.007129	1902 / 0.007639
15111	258356 / 0.010049	258356 / 0.010049	6616 / 0.007990
15107	258356 / 0.009964	6606 / 0.008298	6704 / 0.007636

Diego Baleirón Pais

Con los datos obtenidos se puede ver claramente que el número de operaciones adicionales en la búsqueda es directamente proporcional al tiempo de ejecución lo que es bastante lógico ya que estas operaciones adicionales significan que el elemento que se busca no se encuentra a la primera y hay que buscarlo por toda la tabla hasta encontrarlo. Otro dato interesante que se puede ver en la tabla es que para dos tipos de tablas distintas (distinto tamaño o forma de resolver las colisiones) aunque una tenga más colisiones que otra puede acabar teniendo un tiempo de ejecución menor puesto que las tablas realizan operaciones distintas (o al menos con operadores distintos) unas más costosas que otras, la diferencia de rendimiento que puede causar operar con distintos valores se puede ver claramente en la primera columna de las tres tablas donde las tablas hash, donde las tablas hash solo difieren en tamaño pero su tiempo de ejecución es muy distinto. Por último también hay que tener en cuenta que los datos obtenidos de las dos últimas tablas no son realmente representativos ya que los tiempos son tan bajos que factores como que el programa tardara más en ejecutarse porque la CPU estaba ocupada suponen una parte considerable del tiempo de ejecución.

Teniendo en cuenta todos los apartados anteriores se puede decir que la mejor implementación para este problema es la tabla con recolocación cuadrática con un tamaño de 20011 unidades, de esta forma el factor de carga un poco inferior al 0.5 por lo que no se desperdicia memoria además con ese factor de carga se aprovecha al máximo la recolocación cuadrática permitiendo evitar el apiñamiento (y por tanto evitando colisiones) sin caer en la imposibilidad de recolocar valores, que es el principal problema que tiene este tipo de recolocación.