FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TDA HASH

Nombre y apellido	Lautaro De Nobili
Padron	107394
Email	ldenobili@fi.uba.ar

QUE ES UN HASH?

Tambien conocido como Tabla hash o Mapa hash, es una estructura de datos (una estructura que sirve para el almacenamiento, manejo y operar de elementos) que pide almacenar elementos bajo una clave identificadora. Esta clave es pasada por una funcion de hash para obtener su posicion en esta tabla. Puede haber mas de un elemento igual bajo distintas claves, pero no pueden haber claves repetidas, lo cual implica la unicidad al ubicar el elemento en la busqueda.

Al ser esta clave algo como un DNI para el elemento guardado, resultan mas rapidas y eficientes las operaciones de la tabla, siendo estas comunmente: insertar, quitar y buscar. Extremadamente eficientes, ya que busco en la tabla el indice directamente y vendria a caer justo sobre ese elemento (asumiendo que no se produciria repeticion de indices con distintas claves, lo cual si puede pasar).

Si bien la tabla utiliza la funcion hash para ubicar el elemento en si luego de obtener un indice, puede ser que distintas claves den un mismo indice, es decir, que se tengan que almacenar los elementos en el mismo lugar de la tabla. Estos casos se los denomina “colisiones”. Y hay que contemplaros de manera distinta.

QUE HACER ANTE UNA COLISION?

Bueno, lo primero que debemos hacer es indicar con herramientas de transito a otros vehiculos que ha ocurrido algo para evitar mas incidentes de transito. Ah, me confundi de colision. Volvamos a las coliciones en una Tabla hash.

Para resguardarnos ante una posible colision, debemos elegir que tipo de tabla vamos a usar, siendo estas:

- De direccionamiento abierto (hash cerrado): consiste en guardar los elementos en la misma tabla pero en un lugar distinto (temporalmente) a donde nos indico nuestra funcion hash, ya que el lugar en el que deberia de estar, se encuentra ocupado. Para saber donde debemos colocar nuestro elemento, tenemos que acudir a tecnicas de *probing* (busqueda interna a la tabla). Estas tecnicas pueden ser:
  - probing lineal: ubicar el elemento en el espacio mas proximo al que deberia estar;
  - probing cuadratico: asignar al indice original una sucesion de funciones polinomicas hasta encontrar un espacio vacio;
  - doble hashing: aplicar la funcion de hash nuevamente al indice obtenido previamente hasta encontrar un espacio vacio.

- De direccionamiento cerrado (hash abierto): consiste en encadenar los elementos colisionados en otras estructuras de almacenamiento de datos. El encadenamiento mas utilizado es con listas enlazadas, de la misma manera que se pueden utilizar arboles binarios de busqueda autobalanceados mientras se mantenga un criterio de comparacion entre elementos. Porque soy un distinto, yo use estos ultimos.

Concepto no menor: rehash

Que es rehash? Tambien conocido como redimensionamiento dinamico (palabras con clase), consiste en cambiar el tamanio inicial de la tabla hash para evitar las coliciones y que su complejidad algoritmica se mantenga en O(1), lo que significa que es mas rapido para hacer las operaciones que se requieran.

Este redimensionamiento se procura hacer cuando un porcentaje arbitrario de la tabla ya se encuentra lleno y se agranda el tamanio de la misma, tambien arbitrariamente. Lo mas comun para esto es utilizar una tabla nueva con el tamanio nuevo y reinsertar todos los elementos de la tabla vieja en esta nueva tabla. Otra convencion muy comunmente utilizada es la de aplicar rehash cuando se llena el 50% de la tabla y duplicar el tamanio anterior.

Es una operación muy costosa, por lo que se procura evitar hacerlo, pero hay que encontrar un balance entre los tamanios elegidos y las convenciones para rehash seleccionadas. Menos uso de memoria puede implicar menor eficiencia de la tabla, y mayor eficiencia de la tabla puede implicar mayor uso de memoria (para tener la tabla mas espacio para guardar elementos y evitar colisiones).

- **Indice Rehash**: elegi 0.75 para poder evitar lo mayor posible hacer rehash, ya que es costoso. Asi mismo, al ser el 75% de la tabla la cantidad maxima que voy a querer ocupar, voy a tratar de ocupar la mayor cantidad de casilleros antes de aplicar rehash.
- **Multiplicador Rehash**: esta es la cantidad la cual voy a multiplicar el tamanio de mi hash al hacerle rehash. Elegi 4 veces el anterior ya que no es tan costoso el reservar 4 veces la cantidad de punteros previos como seria volver a hacer rehash. 4 es un buen numero. Intente con 2 veces el tamanio y en algunos casos me volvia a hacer rehash, lo cual no queria.
- **Tamanio minimo inicial de la tabla**: mas de 3, por consigna.

- **ABB's**: si bien mis arboles no son autobalanceados, voy a asumir que no se degeneran en lista. Y aunque lo hiciesen, serian mas eficientes que las listas en el resto de arboles no degenerados.

convenciones

Un gran objetivo de la tabla de hash es poder acceder a los elementos de la misma de la manera mas eficiente, con la menor complejidad algoritmica. Esto implica balancear la manera en la que voy a encadenar los contenedores de elemento y clave, y la forma en la que accedo a ellos. Asi que compare las estructuras que tenia disponibles para usar en estos encadenados: lista y abb (suponiendo que los abb's no degeneran en lista). Como se notara, elegi usar abb's para ello.

Funcion	Lista	ABB
crear	O(1)	O(1)
insertar	O(2n)	O(2logn)
quitar	O(2n)	O(logn)
obtener	O(n)	O(logn)
contiene	O(n)	O(logn)
cantidad	O(1)	O(1)
destruir	O(n)	O(n)

Adicionalmente, el abb.h ya contenia funciones que me serian bastante mas utiles que las disponibles en la lista.h.

mi implementacion

- insertar: en ambos casos recorro el entre los encadenados para ver si la clave ya esta en uso. En el caso de la lista, debo recorrer para encontrarlo. Si lo encuentro ya usado, tengo que recorrer devuelta para conseguir el contenedor y cambiarlo. Si no lo encuentro en uso, ya recorri la lista 1 vez (O(n)), y mi insercion al final de la lista es O(1). En el caso del abb, recorro hasta encontarlo (O(logn)), si lo encuentro lo cambio de una, siendo esto O(1). Si no lo encuentro en uso, recorro de vuelta con O(logn) para encontra la posicion en donde ubicar mi contenedor con el abb_insertar. La suma de ambos en los peores casos son O(2n) y O(2logn), respectivamente.

implementacion de las estructuras de mi implementacion

- **contenedor_t**: almacena el elemento a guardar en el hash junto a su respectiva clave. Estos van a ser los elementos que mis ABB's almacenen;
- **hash_t**: posee de un vector de ABB's, un puntero al destructor a aplicar a los elementos del hash, un contador de cantidad de elementos en la tabla de hash, un contador de la cantidad de arboles en uso, y un contador de punteros a arboles totales. Estos ultimos dos me van a definir el si debo hacer Rehash o no;

- **contenedor_funciones_t**: una estructura fea, no te lo voy a negar. La implemente de forma de compatibilidad entre el iterador de ABB y el iterador de Hash. Esta estructura se pasa como auxiliar al iterador de ABB como auxiliar y va de la mano con "funcion_de_aplicar_funcion" de forma que todos los parametros se cumplan y a su vez se cuenten correctamente las aplicaciones.

funciones internas pt.1

Quiero destacar que muchas de estas funciones fueron implementadas de manera que pudiese aprovechar los iteradores de ABB que ya tenia disponibles. Estas cumplen el proposito de aplicar otras funciones extra y siempre devolver true, de forma que se les sea aplicada la funcion a todos los elementos.

- **comparar_contenedores**: este es mi comparador para los ABB's, de forma que sean comparados en base a sus claves. Siendo estas ultimas strings, le dejo el trabajo de comparar las claves al strcmp de <string.h>;

- **funcion_hash**: tengo entendido que es el corazon del Hash. Mi implementacion consiste en recorrer la clave y sumar sus valores ASCII. A su vez, me guardo auxiliarmente el valor del carácter mas grande. Mi retorno es el resultado de la sumatoria y aplicando el modulo del valor del carácter mas grande;

- **destruir_contenedor**: funcion adaptada para recorrido de ABB. Libera la memoria almacenada para el contenedor_t. En caso de existir y no ser nulo, se le aplica la funcion destructora al elemento almacenado;

- **destruir_abb_y_contenedores**: recorre en preorden el ABB pasado por parametro, aplicandole la funcion destruir_contenedor a cada contenedor. Luego destruye el arbol. No utilice la funcion abb_destruir_todo ya que con el destructor no tenia idea de como guardar el destructor de Hash sin tener que crear un campo adicional en los contenedores.

funciones internas pt.2

Ahora si voy a usar punteos en mis funciones, por la sanidad de ambos.

- **crear_contenedor**: crea un contenedor_t con el elemento pasado y una copia de la clave, cosa que me entere luego de fallar todo en Chanu.

Lamento que mi explicacion del crear_contenedor no este bien ubicada, pero la redacte de apuro ya que fue un cambio de implementacion de ultimo momento (no sabia que habia que guardar las claves en copia).

- **insercion_de_rehash**: funcion adaptada para recorrido de ABB.
  - ✓ Casteo ambos void* a contenedor y hash, respectivamente;
  - ✓ Llamo a hash_insertar pasandole el hash, la clave del contenedor y el elemento del contenedor;
  - ✓ Chequeo que la insercion haya sido exitosa y devuelvo True si lo fue, False si no lo fue.

- **rehash**: funcion tranquila y para nada costosa por suerte.
  - ✓ Creo un hash auxiliar, con el mismo destructor pero con un tamanio inicial de 4 veces que el hash original;
  - ✓ Itero insertando los contenedores de cada ABB de mi hash original en el hash auxiliar. Adicionalmente, voy destruyendo los ABB del hash original sin tocar los contenedores (por eso le paso null como segundo parametro);
  - ✓ Libero los punteros a ABB del hash original;
  - ✓ Le paso todos los campos de mi hash auxiliar al original, incluyendo los punteros a mis nuevos ABB's;
  - ✓ Libero el hash auxiliar (los punteros de este no se tocan para que el hash original los siga usando).

funciones internas pt.3

Ultima funcion interna, lo juro.

- **funcion_de_aplicar_funcion**: lamento el nombre, pero es muy abstracta y sin embargo hace lo que dice que hace.
  - ✓ Casteo mis parametros a contenedor_t y contenedor_funciones_t, respectivamente;
  - ✓ Desarmo mi contenedor de funciones de forma que use la funcion que previamente me pasaron por parametro para hash_con_cada_elemento;
  - ✓ A esa funcion le paso, de mi contenedor de funciones, el mismo hash en el que se esta laburando, el la clave del contenedor y el auxiliar que me pasaron previamente;
  - ✓ El resultado de aplicar la funcion va a un booleano dentro del contenedor de funcion. Este booleano le va a decir al iterador de hash_con_cada_elemento si continuar o no;
  - ✓ Incremento en uno el contador de mi contenedor de funcion, el cual lo inclui en esta estructura por comodidad;
  - ✓ Esta funcion devuelve el opuesto de lo que me devolvio la funcion del contenedor de forma que coincida la condicion de continuar con la del recorrido de ABB;
  - ✓ Nota: en el TDA ABB se recorre hasta que su funcion devuelva false, en el TDA Hash se recorre hasta que su funcion devuelva true, he ahí las cosas raras para hacer ambos recorridos compatibles.

- **hash_crear**:
  - ✓ Reservo la memoria para mi hash_t;
  - ✓ Asigno el destructor a mi nuevo hash;
  - ✓ Asigno la cantidad inicial de ABB's, la cual no va a ser menor a 3 por consigna;
  - ✓ Reservo memoria para los n punteros a abb_t*, siendo n la cantidad inicial del hash.
  - ✓ Devuelvo el hash.

funciones externas pt. 1

Quiero destacar que mi codigo es extenso por las numerosas comprobaciones tanto de los mallocs como de los resultados de las funciones.

me lamento de antemano si mi codigo fue muy bloque…

- **hash_quitar**:
  - ✓ Aplico la funcion hash a la clave recibida. Le aplico el modulo para obtener mi indice en el arbol;
  - ✓ Me creo un contenedor auxiliar con la clave recibida;
  - ✓ Utilizo el ABB quitar con mi contenedor auxiliar, que al usar un comparador de claves, va a coincidirme con el contenedor ya insertado que quiero sacar;
  - ✓ Recibo el contendor a sacar o NULL en caso de no existir tal contenedor;
  - ✓ Si existe el contenedor, lo destruyo y aplico el destructor al elemento.
  - ✓ Reduzco la cantidad de hashes y libero la memoria de mi contenedor auxiliar.

- **hash_obtener**:
  - ✓ Aplico la funcion de hash y obtengo su modulo para conseguir mi indice del arbol a recorrer;
  - ✓ Creo un contenedor auxiliar con la clave buscada;
  - ✓ Utilizo abb_buscar con mi contenedor auxiliar para recibir el contenedor ya dentro del ABB que coincida en clave.
  - ✓ En caso de existir tal contenedor, libero la memoria de mi contenedor auxiliar y devuelvo el elemento del contenedor obtenido.
  - ✓ Si no existe tal contenedor, libero al contenedor auxiliar y devuelvo NULL.

funciones externas pt. 2

- **hash_contiene**:
  - ✓ Aplico la funcion de hash y obtengo su modulo para conseguir mi indice del arbol a recorrer;
  - ✓ Creo un contenedor auxiliar con la clave buscada;
  - ✓ Utilizo abb_buscar con mi contenedor auxiliar para recibir el contenedor ya dentro del ABB que coincida en clave.
  - ✓ En caso de existir tal contenedor, comparo si mis claves coinciden. Si lo hacen, libero mi auxiliar y devuelvo true;
  - ✓ Si no existe tal contenedor o las claves no coinciden, libero al contenedor auxiliar y devuelvo false.

- **hash_cantidad**:
  - ✓ Devuelvo de mi estructura de hash, la cantidad de hashes realizados.

- **hash_destruir**:
  - ✓ Recorro todos mis punteros a ABB's aplicando la funcion de destruir abb junto con sus contenedores. Le paso un destructor no nulo y siendo este el que el usuario me dio;
  - ✓ Se destruyen los ABB's como se detalla en la funcion destruir_abb_y_contenedores;
  - ✓ Libero la memoria de los punteros a los ABB's;
  - ✓ Libero la memoria de mi hash_t.

funciones externas pt. 3

funciones internas pt. 4

- **hash_con_cada_elemento**: uff esta funcion…
  - ✓ Defino mi contador para iterar entre los ABB's del hash;
  - ✓ Creo una estructura contenedor_funciones_t, la cual va a tener la funcion, el hash y el auxiliar, los 3 pasados por parametro. El booleano va a arrancar inicialmente en false por condicion del recorrer de Hash;
  - ✓ Itero entre todos los ABB's del hash con un recorrido, pasandole como funcion a aplicar funcion_de_aplicar_funcion y como auxiliar mi contenedor de funcion. Este recorrido va a pasarle cada contenedor que posea a mi funcion_de_aplicar_funcion de manera tal que la trabaje en base a la funcion que me paso el usuario;
  - ✓ Dentro de la funcion pasada al abb_con_cada_elemento, voy a incrementar mi contador de aplicaciones;
  - ✓ Tanto el abb_con_cada_elemento del ABB actual como el hash_con_cada_elemento van a cortar cuando la funcion pasada a esta ultima devuelva true;
  - ✓ Me guardo auxiliarmente la cantidad de aplicaciones de funcion para poder liberar mi contenedor de funciones;
  - ✓ Libero mi contenedor de funciones;
  - ✓ Devuelvo mi contador auxiliar de aplicaciones de funcion;

funciones externas pt. 5

- hash_insertar:
  - ✓ Me fijo cuantos punteros a ABB estoy usando. Casteo a double mis valores porque al parecer las divisiones de size_t dan siempre 0. Si, en decimales, la cantidad de ABB's en uso es mayor a mi Indice de Rehash, aplico la funcion rehash. Chequeo previo a hacer la insercion para no tener que hacer rehash con un elemento mas;
  - ✓ Creo el contenedor de mi nuevo elemento junto con su clave;
  - ✓ Obtengo el indice del arbol a usar habiendo aplicado la funcion hash y luego el modulo de ese;
  - ✓ Hago un abb_buscar con el arbol en mi indice de arbol y mi nuevo contenedor. Si me devuelve nulo, no habia ningun contenedor bajo esa clave.
  - ✓ Si me devuelve un contenedor, es decir que esa clave ya estaba en uso, le aplico el destructor al elemento de ese contenedor, le asigno el elemento nuevo al mismo contenedor, y libero mi nuevo contenedor porque no lo voy a insertar en ningun abb;

funciones externas pt. 6

- hash_insertar pt. 2:
  - ✓ Si mi abb_buscar devuelve NULL, significa que no tenia ningun elemento bajo esa clave ingresada, asi que continuo de la siguiente manera…;
  - ✓ Existe la posibilidad de que mi indice de arbol me de una posicion de mi vector de ABB's la cual no tenga un arbol creado. Contemplo este caso y si efectivamente no hay un arbol creado, lo creo. Si lo creo, incremento la cantidad de arboles en uso de mi estructura de hash;
  - ✓ Luego de la posible creacion de un nuevo ABB, inserto con abb_insertar, el ABB en ese indice de mi vector de ABB's, y el nuevo contenedor, el elemento bajo la nueva clave;
  - ✓ Si se inserto pipi cucu, incremento la cantidad de hashes;
  - ✓ Devuelvo 0, por convencion, luego de hacer todo eso correctamente.

memoria usada durante las funciones

- hash_crear:
  - ✓ Nuevo hash_t
  - ✓ Punteros a abb_t★ (inicialmente apuntando a NULL)
- hash_insertar:
  - ✓ Nuevo contenedor_t de elemento y clave
  - ✓ Nuevo abb_t★ de requerirlo
- hash_quitar:
  - ✓ Nuevo contenedor_t auxiliar para el abb_quitar
  - × Destructor de elemento si hay
  - × Libero el contenedor_t en el ABB y libero el contenedor_t auxiliar
- hash_contiene y hash_obtener:
  - ✓ Nuevo contenedor_t auxiliar para el abb_buscar
  - × Libero el contenedor_t auxilair
- hash_destruir:
  - × Destructor de elemento si hay, junto a su contenedor_t
  - × Destruyo los ABB's, los punteros a ellos y el hash_t
- hash_con_cada_elemento:
  - ✓ Me reservo un contenedor_funciones_t para mis recorridos
  - × Libero ese contenedor_funciones_t
- rehash:
  - ✓ Nuevo hash_t auxiliar
  - ✓ Punteros a abb_t★ (siendo mas esta vez)
  - ✓ Toda la memoria de insertar en el nuevo hash_t auxiliar
  - × Destruyo los ABB's del viejo hash
  - × Libero los punteros a los abb_t★ del viejo hash
  - ~ Paso los punteros de mi hash auxiliar al viejo hash
  - × Libero la memoria del hash auxiliar