UNIVERSIDAD NACIONAL SAN AGUSTÍN

Facultad de Producción y Servicios

Ciencias de la Computación

Algoritmos y Estructura de Datos

Tema
Hashing

Alumnos:
Luis Sihuinta Pérez
Brian Silva Corrales
Edwar Vargas

26 de noviembre de 2019

HASHING

26 de noviembre de 2019

1. Definiciòn

Una tabla hash es una estructura de datos la cual idealmente es solo un array de tamaño fijo que contiene elementos. Un elemento podría ser una cadena que sirve como clave y miembros dato adicionales.
La tabla hash tiene un tamaño como TableSize la cual es una variable necesaria en una tabla hash.
Cada clave se asigna en el rango de 0 a TableSize - 1. La asignación se llama función hash, la cual debe calcular y asegurar que dos claves distintas se asignen en celdas diferentes. Ya que las celdas son finitas y las claves pueden ser muchas, por lo tanto la función hash busca que las claves se distribuyan de manera uniforme entre las celdas.

2. Funciòn Hash

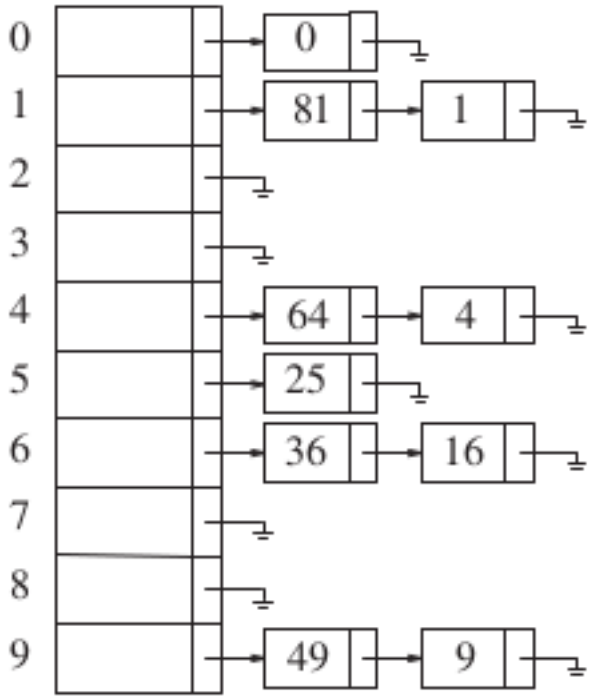
Las funciones Hash nos ayudaran para determinar como ordenar nuestra tabla Hash.
Por ejemplo si las claves de entrada son enteros, podriamos devolver clave mod TableSize. Elegir la función hash debe considerarse cuidadosamente, si el tamaño de la tabla es 10 y las claves terminan en cero, entonces la función hash estándar es una mala elección. Una buena idea es asegurarse que el tamaño de la tabla sea óptimo. Cuando las teclas de entrada son enteros aleatorios, las claves se distribuirian uniformemente.
Mayormente las teclas son cadenas en este caso, una opción es sumar los valores ASCII de los caracteres en la cadena.

```
1
2  int hash( const string & key, int tableSize )
3  {
4        int hashVal = 0;
5
6        for( char ch : key )
7        hashVal += ch;
8
9        return hashVal % tableSize;
10 }
```

Al programar tablas hash se presenta un nuevo problema que son las colisiones y como resolverlas. Probablemente cuando insertemos un elemento y este tiene el mismo valor que un elemento ya insertado, ocurre una colision. A continuaciòn presentaremos algunas formas de solucionar esto.

3. Separate chaining

Comunmente conocida como encadenamiento separado, esta estrategia permite que todos los elementos de una lista sean combinados con el mismo valor(se puede usar la implementacion de lista de la Biblioteca Estandár). Ademas estas lista estas doblemente vinculadas.
En este ejemplo usaremos los 10 cuadrados perfectos y que la funcion hash(x)= x mod 10 (El tamaño de la tabla no es primo pero se usa por simplicidad).



Para insertar algun elemento a la tabla hacemos uso de la funcion hash(x)=x mod 10, con esta funcion insertaria a la lista correspondiende si son duplicados, generalmente se mantiene un miembro de dato adicional, y este miembro se incrementario en caso de una coincidencia. Si el elemento a insertar se nuevo generalmente se inserta al comienzo ya que los elementos que son insertados recientemente tienen mas probabilidad de ser accedidos en el futuro.

Para la implementacion del Separate Chaining se mostrara

```
template <typename HashedObj>
class HashTable
{
public:
    explicit HashTable( int size = 101 );
    bool contains( const HashedObj & x ) const;
    void makeEmplty();
    bool insertar(const HasheObj &x);
    bool insertar(HasheObj &&x);
    bool remove(const HasheObj &x);

```

```
12 private:
13     vector<list<HashedObj>> theLists;
14     int currentSize;
15     // The array of Lists
16     void rehash( );
17     size_t myhash( const HashedObj & x ) const;
18 };
```

La tabla hash solo funciona para objetos que proporcionan un funcion hash y operadores de igualadad(operador == or operador !=), estas tablas no son iguales que un arbol ya que estos funcionan para objetos comparables.

Las funciones hash funcionan mediante plantillas de objeto a funcion.

```
1 template <typename Key>
2 class hash
3 {
4 public:
5     size_t operator() ( const Key & k ) const;
6 };
```

El tipo size_t representa un tipo integral sin signo y presenta el tamano del objeto.
Una clase de implementa un algoritmo de tabla hash y puede ser usado de objetos genericos para generar un tipo integral sin signo y luego escalar el resultado en un indice de una matriz.

```
1  template <>
2  class hash<string>
3  {
4  public:
5      size_t operator()( const string & key )
6      {
7          size_t hashVal = 0;
8          for( char ch : key )
9              hashVal = 37 * hashVal + ch;
10         return hashVal;
11     }
12 };
```

En los siguiente codigos se mostrara en el que la clase empleado puede almanecenar una tabla hash generica, utilizando el nombre el miembro como clave.
La clase employee proporciona operadores de igualdad y un objeto de funcion hash

```
1 size_t myhash( const HashedObj & x ) const
2 {
3     static hash<HashedObj> hf;
4     return hf( x ) % theLists.size( );
5 }
```

```
1 // Example of an Employee class
2 class Employee
3 {
4 public:
5     const string & getName( ) const
6     {   return name; }
7     bool operator==( const Employee & rhs ) const
8     {   return getName( ) == rhs.getName( ); }
```

```
    bool operator!=( const Employee & rhs ) const
    {   return !( *this == rhs; }
// Additional public members not shown
private:
    string name;
    double salary;
    int
    seniority;
    // Additional private members not shown
};
template<>
class hash<Employee>
{
public:
    size_t operator()( const Employee & item )
    {
        static hash<string> hf;
        return hf( item.getName( ) );
    }
};
```

```
void makeEmpty( )
{
    for( auto & thisList : theLists )
    thisList.clear( );
}
bool contains( const HashedObj & x ) const
{
    auto & whichList = theLists[ myhash( x ) ];
    return find( begin( whichList ), end( whichList ), x ) != end( whichList
    );
}
bool remove( const HashedObj & x )
{
    auto & whichList = theLists[ myhash( x ) ];
    auto itr = find( begin( whichList ), end( whichList ), x );
    if( itr == end( whichList ) )
        return false;
    whichList.erase( itr );
        --currentSize;
    return true;
}
```

4. Tables Hash Without Linked Lists

La tecnica Separate Chaining tiene sus desventajas ya que podria relentizar un poco el algoritmo debido al tiempo requerido de asignar nuevas celdas y requereria la implementacion de una segunda estructura de datos. Una solucion para resolver colisiones con listas vinculadas es probar celdas alternativas hasta encontrar alguna vacia, y esto lo logramos con la siguiente funcion h i (x) = (hash (x) + f (i)) mod; y la funcion f(i) es para resolver colisiones , debido a que todos los dato van en la tabla y se necesataria una tabla mas grande.

4.1. Linear Probing

Linear Probing es la funcion linear i, tipicamente f(i)=i esto equivale probar celdas secuenciales hasta encontrar una vacia. En la siguiente figura se insertara (89,18,49,58,69) en una tabla hash usando la misma funcion de antes, y la estrategia de colision.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

La primera colisión ocurre cuando se inserta 49; se coloca en el siguiente lugar disponible, es decir, el lugar 0, que está abierto. La clave 58 choca con 18, 89 y luego 49 antes de que una celda vacía se encuentre a tres de distancia. La colisión para 69 se maneja de manera similar. Mientras la tabla sea grande siempre se puede encontrar un lugar vacio pero el tiempo puede ser bastante grande, pero mientras se va insertando mas elementos comenzaran a formarse bloques ocupadas y llegara un momento en que no habran celdas vacia. Y esto es conocido como agrupamiento primario, significa que cualquiera datos ingresado al cluster requerira varios intentos para resolver la colision. El numero de sondeos que se realizar insercion es

$$12(1 + 1/(1 - a)^2) \tag{1}$$

Y para la busqueda

$$12(1 + 1/(1 - a)) \tag{2}$$

busquedas exitosas.

4.2. Quadratic Probing

El sondeo cuadradito es un metodo de resolucion de colision que elimina la agrupacion primaria. Y eleccion popular la agrupacion primaria es

$$f(i) = i^2 \tag{3}$$

En el siguiente ejemplo muestra el uso esta funcion al insertar (89,18,49,58,69)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Cuando 49 choca con 89, la siguiente posición intentada es una celda de distancia. Esta celda está vacía, por lo que 49 se coloca allí. A continuación, 58 colisiona en la posición 8. Luego se prueba la celda que está lejos, pero se produce otra colisión. Se encuentra una celda vacante en la siguiente celda probada, que es

$$2^2 = 4 \tag{4}$$

de distancia. 58 se coloca así en la celda 2. Lo mismo sucede con 69. Para el sondeo linear es una mala idea que la tabla se llene , por que el rendiemiento se degrada, encambio para el sondeo cuadratico es mas drastico, en la que no hay garantia de encontrar una celda vacia cuando la tabla este lleno mas de la mitad, o incluso antes de que se llene a la mitad si el tamaño de la mesa no es primo. De hecho hemos demostraado de que si la mesa es esta media vacia o si el tamaño de la mesan o es primo no hay garantia de encontrar celdas vacias.
Es importante que la tabla este medio llene y que el tamañano se un numero primo ya que si el tamaño fuera 16, entonces la unica alternativa las ubicaciones estarian a distancias 1 , 4 o 9. La eliminacion estandar no se podria realizar en una tabla hash de sondeo ya que la celda podria probocar una colison en la figura anterior si eliminacion el 89, entonces todas las operaciones de busqueda fallarian. Para este caso de la eliminacion la tabla hash de prueba requiere una eliminacion lenta.

```
template <typename HashedObj>
class HashTable
{
public:
    explicit HashTable( int size = 101 );
    bool contains( const HashedObj & x ) const;
    void makeEmpty( );
    bool insert( const HashedObj & x );
```

```
    bool insert( HashedObj && x );
    bool remove( const HashedObj & x );

    enum EntryType { ACTIVE, EMPTY, DELETED };
private:
    struct HashEntry
    {
        HashedObj element;
        EntryType info;
        HashEntry( const HashedObj & e = HashedObj{ }, EntryType i = EMPTY )
        : element{ e }, info{ i } { }
        HashEntry( HashedObj && e, EntryType i = EMPTY )
        : element{ std::move( e ) }, info{ i } { }
    };
    vector<HashEntry> array;
    int currentSize;
    bool isActive( int currentPos ) const;
    int findPos( const HashedObj & x ) const;
    void rehash( );
    size_t myhash( const HashedObj & x ) const;
};
```

En lugar de una matriz de celdas tenemos una matriz de celda de entrada de tabla hash. La clase anidada HashEntry almaneca el estado de una entrada que es ACTIVE, EMPTY O DELETE para eso usamos un tipo de enumerado estandar.

```
enum  EntryType { ACTIVE, EMPTY, DELETED };
```

En el siguiente codigo se mostrara poner a cada miembro de la celda para cada Contains(x)

```
explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
{
    makeEmpty( );
}
void makeEmpty( )
{
    currentSize = 0;
    for( auto & entry : array )
    entry.info = EMPTY;
}
```

EL siguiente codigo la función de miembro privado findPos realiza la resolución de colisión.

```
bool contains( const HashedObj & x ) const
{
    return isActive( findPos( x ) );
}
int findPos( const HashedObj & x ) const
{
    int offset = 1;
    int currentPos = myhash( x );
    while( array[ currentPos ].info != EMPTY &&
        array[ currentPos ].element != x )
    {
    currentPos += offset; // Compute ith probe
```

```
13      offset += 2;
14      if( currentPos >= array.size( ) )
15          currentPos -= array.size( );
16      }
17      return currentPos;
18  }
19  bool isActive( int currentPos ) const
20  {
21      return array[ currentPos ].info == ACTIVE;
22  }
```

La rutina final es la insercion, si el elemento que queremos insertar ya esta en la tabla no se hace nada de lo contrario colocamos en el lugar sugerido la la funcion findPos()y si la carga exceso en 0.5, la tabla esta llena simplemente lo que hacemos es ampliar la tabla hash este proceso se llama rehashing. Y por ultimo la eliminacion, como sabemos el sondeo cuadrático elimina la agrupación primaria, los elementos que se desplazan hacia la misma posición explorarán las mismas celdas alternativas. Esto se conoce como agrupamiento secundario.

```
1   bool insert( const HashedObj & x )
2   {
3       // Insert x as active
4       int currentPos = findPos( x );
5       if( isActive( currentPos ) )
6           return false;
7       array[ currentPos ].element = x;
8       array[ currentPos ].info = ACTIVE;
9       // Rehash; see Section 5.5
10      if( ++currentSize > array.size( ) / 2 )
11          rehash( );
12      return true;
13  }
14  bool remove( const HashedObj & x )
15  {
16      int currentPos = findPos( x );
17      if( !isActive( currentPos ) )
18          return false;
19      array[ currentPos ].info = DELETED;
20      return true;
21  }
```

4.3. Double Hashing

Para el double hash una solucion popular es

$$f(i) = i * hash_2(x) \tag{5}$$

Esta fórmula dice que aplicamos una segunda función hash a x y sondeamos a una distancia

$$hash_2(x), 2hash_2(x), ..., y \tag{6}$$

así. Una mala idea seria insertar 99 con:

$$hash_2 = xmod9 \tag{7}$$

Ademas es importante ver que cada celda se pueda sondear (esto es importante en la siguiente figura ya que el tamaño no es primo)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Una funcion como

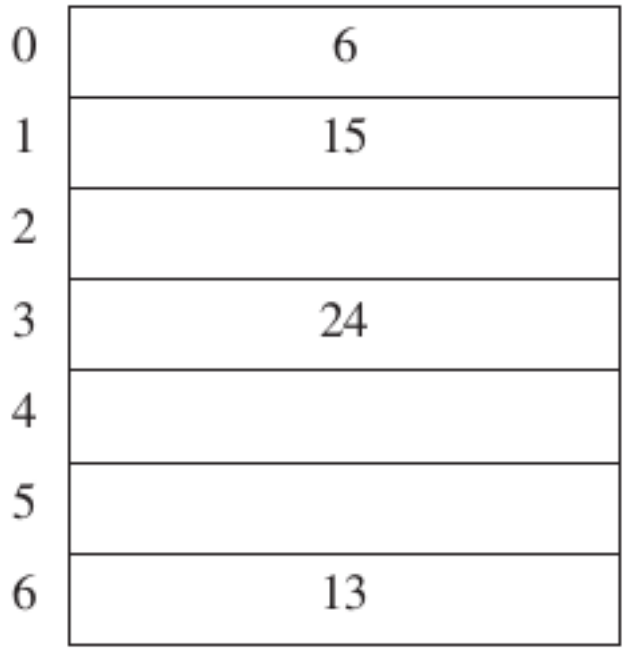
$$hash_2(x) = R(xmodR) \tag{8}$$

funcionaria bien como R un primo menor de el tamaño de la tabla hash.

5. Rehashing

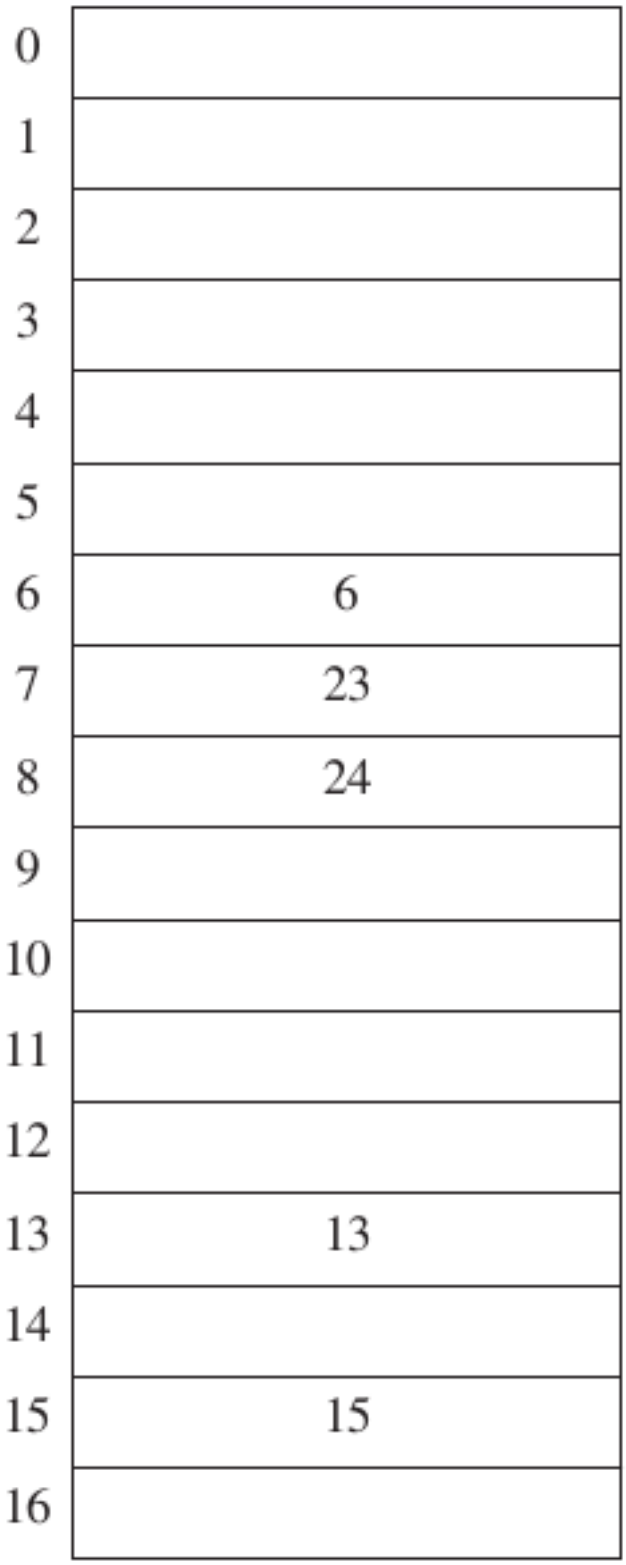
Cuando la tabla hash este medio llena, el tiempo de las operaciones tomara mas tiempo y algunas operaciones fallarian. Lo que se debe hacer es crear otra tabla hash con el doble de tamañno de la otro y escanee toda la tabla hash original, calculando el nuevo valor hash para cada elemento (no eliminado) e insertándolo en la nueva tablaComo ejemplo, suponga que los elementos 13, 15, 24 y 6 se insertan en una tabla hash de sondeo lineal de tamaño 7. La función hash es h (x) = x mod 7. El hash resultante La tabla aparece en la Figura 5.19.

Como ejemplo, suponga que los elementos 13, 15, 24 y 6 se insertan en una tabla hash de sondeo lineal de tamaño 7. La función hash es h (x) = x mod 7. El hash resultante es la siguiente figura.



Si insertamos el 23 en la tabla hash el tamaño seria mas del 70 por ciento lleno. Debido a eso se crea una nueva tabla hash y la nueva funcion seria h(x)= x mod 17. La tabla anterior es escaneada y los elementos nos insertados en la nueva tabla y el tiempo de ejecuacion seira O(N) ya que escanerada todo los elemento de la tabla anterior en realidad esto no seria malo por que sucede con pocas frecuencias.

La nueva tabla creada


0
1
2
3
4
5
6
6
7
23
8
24
9
10
11
12
13
13
14
15
15
16

6. Hash Tables in the Standard Library

En C ++ 11, la Biblioteca estándar incluye implementaciones de tablas hash de conjuntos y mapas como unordered_set y unordered_map, que establece paralelamente enviar y asignar. Los elementos unordered_set deben proporcionar un operator == y una funcion hash. Del mismo modo que las plantillas de conjuntos y mapas se pueden instanciar con un objeto a funcion que proporciona function, unordered_set y unordered_map se pueden instanciar con objetos de función que proporcionar una función hash y un operador de igualdad. Asi como se muestra en el siguiente codigo

```
/**
* Rehashing for quadratic probing hash table.
*/
void rehash( )
{
    vector<HashEntry> oldArray = array;
    // Create new double-sized, empty table
    array.resize( nextPrime( 2 * oldArray.size( ) ) );
    for( auto & entry : array )
        entry.info = EMPTY;
    // Copy table over
    currentSize = 0;
    for( auto & entry : oldArray )
         if( entry.info == ACTIVE )
            insert( std::move( entry.element ) );
}
/**
* Rehashing for separate chaining hash table.
*/
void rehash( )
{
    vector<list<HashedObj>> oldLists = theLists;
    // Create new double-sized, empty table
    theLists.resize( nextPrime( 2 * theLists.size( ) ) );
    for( auto & thisList : theLists )
        thisList.clear( );
    // Copy table over
    currentSize = 0;
    for( auto & thisList : oldLists )
        for( auto & x : thisList )
            insert( std::move( x ) );
}
```

```
class CaseInsensitiveStringHash
{
public:
    size_t operator( ) ( const string & s ) const
    {
        static hash<string> hf;
        return hf( toLower( s ) );
        // toLower implemented elsewhere
    }
     bool operator( ) ( const string & lhs, const string & rhs ) const
    {
```

```
12          return equalsIgnoreCase( lhs, rhs );
13      }
14  };
15  unordered_set<string,CaseInsensitiveStringHash,CaseInsensitiveStringHash> s;
```

7. Hash Tables with Worst-Case O(1) Access

Pero, ¿cuál es el peor caso esperado para una búsqueda que asume una función hash razonablemente bien comportada? En algunas aplicaciones, como el hardware para cachés de memoria, es importante que la búsqueda tenga una cantidad constante de tiempo de finalización. Supongamos que N se conoce, si se nos permite reorganizar los elementos a medida que se insertan, entonces O (1) el peor de los casos es alcanzable para las búsquedas.
A continuaciòn presentaremos el hashing perfecto, y luego dos formas de hashing mas recientes que rivalizan con esquemas de hashing clásicos que han prevalecido durante muchos años.

7.1. Perfect Hashing

Si una implementación de encadenamiento separada pudiera garantizar que cada lista tuviera como máximo un número constante de datos estaria bien . Sabemos que a medida que hagamos más listas, las listas serán más cortas, por lo que, teóricamente, si tenemos suficientes listas,tenemos una alta probabildiad de evitar colisiones .
Aun pueden haber problemas con esto como que el número de listas puede ser excesivamente grande, segundo es que con muchas listas, aún podríamos tener mala suerte.
La idea es que tras analizar una posible funcion que no satisfaga el tiempo de busqueda pasemos a una segunda sin relacion con la primera , y asi sucesivamente con la probabildiad de que no habra colisiones.
El uso de listas no es práctico. Asi que se puede hacer usando solo contenedores, La idea es que debido a que se espera que los contenedores tengan solo un pocos elementos cada uno, la tabla hash que se utiliza para cada contenedor puede ser cuadrática en el tamaño del contenedor.
Al igual que con la idea original, cada tabla de hash secundaria se construirá utilizando una tabla diferente función hash hasta que esté libre de colisión. La tabla hash primaria también se puede construir varias veces si el número de colisiones que se producen es mayor de lo requerido. Esta Esta es la ide a de hashing perfecto.

7.2. Cuckoo Hashing

La idea que se tiene es de usar dos funciones hash en lugar de sólo una. Esto da dos posibles ubicaciones en la tabla hash para cada dato. En una de las variantes de uso común del algoritmo, la tabla hash se divide en dos tablas más pequeñas de igual tamaño, y cada función hash proporciona un índice en una de estas dos tablas.
Este algoritmo en sí funciona: para insertar un nuevo dato x. Usamos la primera función hash, y si la ubicación de la tabla está vacío, el dato se puede colocar.

7.3. Hopscotch Hashing

Hopscotch hashing es un algoritmo que intenta mejorar el linear probing. Debido a la agrupación primaria y secundaria, esta secuencia puede ser larga en promedio a la medida que se carga la tabla y, por lo tanto, muchas mejoras, como el cuadratic probing, doble hashing, y así sucesivamente, se han propuesto para reducir el número de colisiones.

8. Universal Hashing

Aunque las tablas hash son muy eficientes y tienen un costo promedio constante por operación, suponiendo factores de carga apropiados, su análisis y rendimiento dependen del hash función que tiene dos propiedades fundamentales:

1. **La función hash debe ser computable en tiempo constante.**
2. **La función hash debe distribuir sus elementos de manera uniforme entre las ranuras de la matriz.**

En esta sección, discutimos las funciones hash universales, que nos permiten elija la función hash aleatoriamente de tal manera que se cumpla la condición 2 anterior. Nosotros usamos M para la representacion de TableSize
Definicion 1

Una familia H de funciones hash es universal, si para cualquier x = y, el número de funciones hash h en H para el cual h (x) = h (y) es como máximo | H | / M
La definición anterior significa que si elegimos una función hash al azar de una familia universal H, entonces la probabilidad de una colisión entre dos elementos distintos es como máximo 1 / M, y cuando se agrega a una tabla con N elementos, la probabilidad de una colisión en el punto inicial es como máximo N / M, o el factor de carga.
Definicion 2
Una familia H de funciones hash es k-universal, si para cualquier

$$x_1 \neq y_1, x_2 \neq y_2, ..., xk \neq yk, \tag{9}$$

el número de funciones hash h en H para las cuales h (x 1) = h (y 1), h (x 2) = h (y 2),. . ., y h (x k) = h (y k) es como máximo | H | / M^k.

9. Extendible Hashing

Para terminar algunas veces tenemos tantos datos que la memoria principal no puede soportar asi que los guarda en el disco como vimos anteriormente con el B TREE tenemos que considerar la cantidad de accesos al disco para recuperar datos.
Tenemos N registros para almacenar y este N cambia con el tiempo, tambien los registros M caben en un bloque de disco.
Si usamos PROBING HASHING o el SEPARATE CHAINING, las colisiones pueden hacer que se examinen varios bloques en la busqueda.Una alternativa es el hashing extensible, permite realizar una búsqueda en dos accesos de disco. Las inserciones también requieren pocos accesos al disco. En el B TREE en teoría, podríamos elegir tal M que la profundidad del árbol sería 1. Asi las búsqueda después de la primera tendria un acceso al disco debido a que nuestro nodo raiz en teoria podría almacenarse en la memoria principal. El problema son las ramificaciónes son tantas que tomaria tiempo procesar en que hoja esta el dato, si se reduciera el tiempo, tendríamos un esquema práctico y esta es la estrategia de el hashing extensible.

Supongamos que tenemos datos enteros de 6 bits, la raíz del "árbol"tiene 4 punteros dados por los dos bits principales de los datos. Las hojas tienen M = 4 elementos. En cada hoja los dos primeros bits son idénticos el cual es el número entre paréntesis. Al insertar la clave 100100 esta iría a la tercera hoja, pero la tercera hoja ya está llena asi que la dividimos las cuales estan determinadas por los aumentando el tamaño del directorio a 3. Lo cual proporciona tiempos de acceso rápidos para operaciones de inserción y búsqueda en grandes bases de datos. Existe la posibilidad de duplicar claves si hay M duplicados entonces el hashing extendible deja de funcionar y se deberian hacer mas arreglos.