

## Capítulo 7

# Tablas de hash.

### 7.1. Operaciones.

La tabla de hash pertenece a la categoría de diccionarios que son aquellas estructuras de datos y algoritmos que permiten buscar, insertar y descartar elementos.

Si las operaciones se reducen solamente a buscar e insertar se llaman tablas de símbolos.

En diccionarios puros sólo se implementa buscar.

Los diccionarios y tablas pertenecen también a la categoría de conjuntos dinámicos.

### 7.2. Clave.

La información que se desea buscar suele ser una estructura que organiza la información. Uno de los campos de esa estructura se denomina clave, y debe ser única. Sólo una de las estructuras puede tener un determinado valor de la clave.

Si la clave es un string, deben definirse operadores de comparación, los cuales generalmente comparan en forma alfabética.

### 7.3. Tabla de acceso directo.

Un arreglo es una estructura de datos que permite implementar tablas de acceso directo. Es este caso la clave es el índice del arreglo, y existe una posición del arreglo para cada posible clave.

Si el contenido de una celda del arreglo es un puntero a la estructura con los datos asociados a la clave, se puede implementar las operaciones en forma sencilla. Si el elemento asociado a una clave no está presente se lo indica con un puntero de valor NULL.

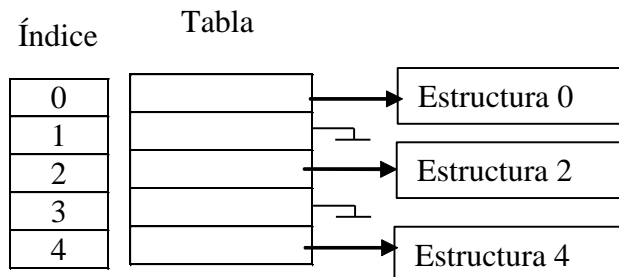


Figura 7.1 Tabla de acceso directo.

```
pnode buscar(int clave)
{
    return (Tabla[clave]);
}

int insertar(int clave, pnode pestructura)
{
    if (Tabla[clave] == NULL ) Tabla[clave]=pestructura; return 0;
    else return (1); // error: ya estaba.
}

int descartar(int clave)
{
    if (Tabla[clave]!= NULL ) {free(Tabla[clave]); Tabla[clave]= NULL ; return 0;}
    else return (1); //error: no estaba.
}
```

Todas las operaciones son  $O(1)$ .

Para emplear esta solución el tamaño del arreglo, debe ser igual al número de claves posibles y éstas deben estar entre 0 y  $N-1$ .

## 7.4. Tablas de Hash.

Si el número de claves almacenadas en la tabla es pequeño en comparación con el número total de claves posibles, la estructura tabla de hash resulta una forma eficiente de implementación.

Ejemplos:

Claves alfanuméricas: Con las letras del abecedario se pueden escribir un gran número de palabras, pero el número de palabras empleadas como identificadores por un programador es muchísimo menor; y se desea almacenar los identificadores del programador solamente.

Claves enteras: el número de RUTs posibles asciende a varios millones, pero el número de RUTs de los alumnos de una carrera no sobrepasa los mil; y se desea almacenar solamente los alumnos de una carrera.

Las tablas de hash se implementan basadas en un arreglo cuyo tamaño debe ser *proporcional* al número de claves almacenadas en la tabla.

## 7.5. Función de hash.

Lo fundamental del método es encontrar una función que a partir de la clave (proveniente de un universo de  $N$  claves posibles), sea ésta numérica o alfanumérica, encuentre un entero sin signo entre 0 y  $B-1$ , si la tabla de hash está formada por  $B$  elementos. Con  $N \gg B$ .

La función de hash muele o desmenuza o hace un picadillo con la clave, de allí el nombre; que es el significado de la palabra inglesa hash (no es un apellido).

Para toda clave  $x$ , perteneciente al Universo, debe estar definida la función de hash,  $h(x)$ , con valores enteros entre 0 y  $(B-1)$ .

La función  $h(x)$  debe distribuir las claves, *lo más equitativamente posible*, entre los  $B$  valores.

Si  $h(x_i) = h(x_j)$  se dice que hay *colisión*, y debe disponerse de un método para resolverla.

Si la función de hash es “buena”, habrán pocas colisiones; si hay  $c$  colisiones en promedio, las operaciones resultarán de complejidad  $O(c)$ , constante; prácticamente independiente de  $B$ . Si todas las claves que se buscan en la tabla, producen colisiones, que es el peor caso, las operaciones resultarán  $O(B)$ , esto en caso de tener  $B$  claves que colisionen en la misma entrada de la tabla.

Las funciones de hash son generadores de números pseudo aleatorios.

### 7.5.1. Funciones de hash para enteros.

Una función bastante simple es dividir la clave por un número primo cercano al número de baldes, y luego sacar módulo  $B$ .

```
int hash(int clave)
{ return clave%B; } //conviene escoger B como un número primo.
```

Si  $B$  es una potencia de dos, la división se efectuará mediante corrimientos a la derecha, con lo cual el valor de hash sólo dependerá de los bits más significativos de la clave. Lo cual tenderá a generar muchas colisiones. Mientras más bits de la clave participen en la formación del valor, mejor será la función de hash.

También es deseable tener una función de hash que para claves *muy parecidas* genere valores de hash con mucha separación, esta propiedad es muy útil en hash lineal. Existe una metodología denominada clase universal de funciones de hash, en la cual  $p$  es un número primo mayor que  $B$ , con  $a$  y  $b$  números enteros, y  $B$  no necesariamente primo.

```
int Uhash(int clave, int a, int b, int p)
{ return ((a*clave+b)%p)%B; }
```

Puede demostrarse que con estas funciones la probabilidad de tener colisión entre dos claves diferentes es menor o igual a  $1/B$ .

Para claves numéricas existen variados procedimientos experimentales. Todos ellos basados en lograr una mezcla de los números que forman la clave, lo más aleatoria posible, mediante corrimientos y operaciones lógicas, que son eficientemente traducidos a lenguaje de máquina.

Cambiando las siguientes definiciones de tipos las funciones para claves enteras se pueden emplear en máquinas de diferente largo de palabra.

La siguiente definición de tipos es para máquinas de 16 bits.

```
typedef unsigned long int u32; //32 bits
typedef unsigned int u16;     //16 bits

/* Mezcla los números de la clave, asumiendo enteros de 32 bits.
 * Robert Jenkin */
u16 inthash(u32 key)
{
    key += (key << 12);    key ^= (key >> 22);
    key += (key << 4);     key ^= (key >> 9);
    key += (key << 10);    key ^= (key >> 2);
    key += (key << 7);     key ^= (key >> 12);
    return (u16) key%B;
}
```

### 7.5.2. Funciones de hash para strings alfanuméricos.

Para claves alfanuméricas puede emplearse la suma de los valores enteros equivalentes de los caracteres, y aplicando módulo  $B$  a la suma total, para entregar un valor de balde válido.

```
unsigned int h(char *s) /* función simple de hash */
{ int hval;
  for (hval =0; *s!='\0';) hval+= *s++;
  return (hval % B);
}
```

Sin embargo no tiene buen comportamiento evaluada experimentalmente.

Una mejor función, también propuesta por Brian Kernighan y Dennis Ritchie:

```
unsigned int stringhash(char *s)
{ int i;
  unsigned int h=0;
  for( i=0; *s; s++) h = 131*h + *s; /* 31 131 1313 13131 131313 .... */
  return (h%B);
}
```

Robert Sedgwick en su libro *Algorithms in C*, propone la siguiente función, que emplea dos multiplicaciones para generar el valor de hash, y que evaluada experimentalmente, tiene muy buen comportamiento.

```
unsigned int RSHash(char* str)
{ unsigned int b  = 378551;
  unsigned int a  = 63689;
  unsigned int hash = 0;
```

```
unsigned int i = 0;

for(i = 0; *str; str++, i++)
{ hash = hash * a + (*str);
  a = a * b;
}
return (hash%B);
}
```

La siguiente función propuesta por Serge Vakulenko, genera mediante dos rotaciones y una resta, por cada carácter del string, el valor aleatorio.

```
unsigned int ROT13Hash (char *s)
{ unsigned int hash = 0;
  unsigned int i = 0;
  for (i = 0; *s; s++, i++) {
    hash += (unsigned char)(*s);
    hash -= (hash << 13) | (hash >> 19);
  }
  return hash%B;
}
```

Con la introducción de algoritmos de encriptación, se han desarrollado nuevos métodos para aleatorizar una clave alfanumérica.

**MD5** es un algoritmo de reducción criptográfico diseñado en 1991 por Ronald Rivest del MIT (Instituto Tecnológico de Massachusetts).

Su uso principal es generar una firma digital de 128 bits (fingerprint) a partir de un documento de largo arbitrario. La cual aplicada a strings podría emplearse como función de hash, seleccionando algunos de los bits de los 128.

A continuación una breve descripción de este algoritmo.

MD5 es un acrónimo de **M**essage **D**igest algorithm **5**, derivado de versiones anteriores (MD4, MD3, etc.). La digestión del mensaje debe entenderse como la generación de un resumen de 128 bits de éste. Está basado en el supuesto de que es poco probable que dos mensajes diferentes tengan la misma firma digital, y menos probable aún producir el mensaje original a partir del conocimiento de la firma.

Sigue empleándose para verificar si un determinado archivo ha sido modificado, pudiendo ser éste un correo, imagen, o un programa ejecutable. El originador del documento puede establecer cuál es su firma digital, y el usuario debería comprobar que su copia tiene la misma huella digital, regenerándola localmente y comparándola con la de la distribución.

Una función de hash es irreversible, si no existe algoritmo que, ejecutado en un tiempo razonable, permita recuperar la cadena original a partir de su valor de hash.

Si el número de valores a los cuales se les aplica la función de hash son mucho mayores que el número de valores que produce la función, se producirán colisiones. En un buen algoritmo de hash, se producirán menores colisiones.

La desincryptación de un valor de salida de la función de hash, consiste en determinar la entrada que produjo ese valor. Para esto deben generarse valores de entrada y obtener el valor de salida de la función de hash y compararlo con el conocido. El método, denominado de fuerza bruta genera todas las combinaciones de las entradas para un largo dado, lo cual puede llevar mucho tiempo. Otro ataque es conocido como el de diccionario, que consiste en probar las palabras previamente almacenadas en un diccionario.

## 7.6. Tipos de tabla.

Se denomina hash abierto o externo o encadenado, a la estructura que resuelve las colisiones mediante una lista. En éstas el número  $n$ , de elementos almacenados en la tabla, podría ser superior al número  $B$  de elementos del arreglo ( $n \geq B$ ).

Se denomina hash cerrado, a las estructuras que almacenan los datos en las mismas casillas del arreglo; debiendo disponerse de un método para determinar si el elemento del arreglo está vacío, ocupado o descartado. En éstas el número  $n$  de elementos almacenados en la tabla no puede ser superior al número  $B$  de elementos del arreglo ( $n \leq B$ ).

### 7.6.1. Hash abierto.

#### 7.6.1.1. Diagrama de la estructura.

Se ilustra una tabla de  $B$  entradas, que puede considerarse como un arreglo de punteros.

En la figura 7.2: la entrada 1 está vacía; la entrada 0 tiene un elemento; la entrada dos, tiene dos elementos, y muestra que las colisiones se resuelven mediante una lista. En la estructura de datos asociada a cada nodo, sólo se ilustra el almacenamiento de la clave.

Se tienen:  $h(\text{Clave}_0) = 0$ ;  $h(\text{Clave}_{B-1}) = B-1$ ;  $h(\text{Clave}_i) = h(\text{Clave}_j) = 2$

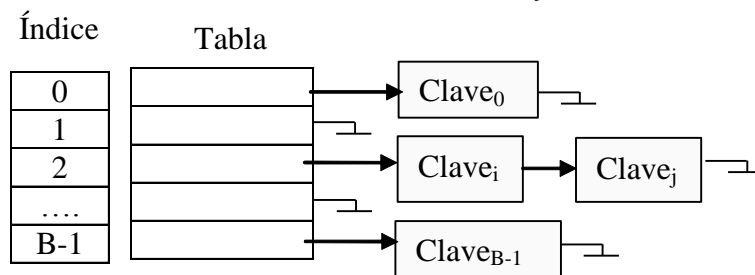


Figura 7.2 Tabla de hash abierto o de encadenamiento directo.

**7.6.1.2. Declaración de tipos, definición de variables.**

```
typedef struct moldecelda
{
    char *nombre;
    struct moldecelda *next;
} tcelda, *pcelda;

#define B 10          /* 10 baldes */
static pcelda hashtablea[B]; /*tabla punteros */
```

La información almacenada en la celda es un string dinámico.

**7.6.1.3. Operaciones en hash abierto.****7.6.1.3.1. Crear tabla vacía.**

```
void makenull(void)
{
    int i;
    for ( i = 0; i < B; i++) hashtablea[i] = NULL;
}
```

**7.6.1.3.2. Función de hash.**

La función  $h(s)$  a partir de los caracteres que forman el string  $s$  genera un número entero sin signo entre 0 y  $B-1$ . Puede usarse alguna de las descritas en 7.5.2.

**7.6.1.3.3. Buscar si un string está en la tabla.**

Calcula el índice;  
Mientras la lista no haya llegado al final:  
    Si la clave del nodo es igual a la buscada: retorna puntero al nodo.  
    Si llegó al final: retorna nulo, que implica que no encontró la clave.

```
pcelda buscar(char *s)
{
    pcelda cp; /* current pointer */

    for (cp = hashtablea[h(s)]; cp!= NULL; cp = cp->next)
        if (strcmp(s, cp->nombre ) == 0) return (cp); /* lo encontró */
    return (NULL);
}
```

**7.6.1.3.4. Insertar string en la tabla.**

Buscar string en la tabla;  
Si no la encuentra:  
    Crea nodo.  
    Si no puede crear el nodo, retorna NULL.

Si pudo crear nodo.  
     Crea string dinámico.  
         Si crea el string. Asocia el string con el nodo.  
         Si no puede crear el string, retorna NULL.  
 Ubica el balde para insertar.  
 Insertar al inicio de la lista.  
 Retorna puntero al insertado. Operación exitosa.  
 Si la encontró: Retorna NULL. Es error de inserción en conjuntos.

La siguiente función solicita espacio en el heap, y a través de strcpy copia el valor del argumento en el espacio recién creado.

```

char *strsave(char *s) /* K.R. pág. 103 */
{ char *p;
  if (( p = malloc(strlen(s) + 1)) != NULL)
    strcpy(p, s);
  return (p);
}

pcelda insertar(char *s)
{ pcelda cp;
  int hval;

  if (( cp = buscar(s)) == NULL)
  {
    cp = (pcelda ) malloc(sizeof (tcelda ));
    if (cp == NULL) return (NULL);
    if (( cp-> nombre = strsave(s)) == NULL ) return (NULL);
    hval = h(cp -> nombre);
    cp -> next = hashtablea[hval];
    hashtablea[hval] = cp;
    return (cp);
  }
  else return (NULL);
}
  
```

#### 7.6.1.3.5. Descartar.

Cálcula índice.

Si no hay elementos ligados al balde, retorna 1.

Si hay elementos:

    Busca en el primer elemento, si lo encuentra liga la lista.

    Si no es el primer elemento recorre la lista, manteniendo un puntero *q* al anterior.

        Si lo encuentra a partir del segundo: Pega la lista, mediante *q*;

    Si estaba en la lista: Liberar espacio; retorna 0, indicando función realizada.

    Si no la encontró: retorna 1, error de descarte.



```

int descartar(char *s)
{
    pcelda q, cp;
    cp = hashtablea[h(s)];
    if (cp != NULL) {
        if (strcmp (s, cp-> nombre ) == 0) /* primero de la lista */
            hashtablea[h(s)] = cp->next;
        else
            for (q=cp, cp = cp ->next; cp != NULL; q = cp, cp = cp ->next )
                if (strcmp (s, cp->nombre ) == 0)
                    { q ->next = cp ->next; break; }
        if (cp != NULL )
            { free((char *)cp -> nombre);
              free ( (pcelda) cp );
              return (0);
            }
        else return (1); //no lo encontró en lista
    }
    else return (1); //balde vacío
}

```

#### 7.6.1.4. Análisis de complejidad en hash abierto.

##### 7.6.1.4.1 Caso ideal.

En un caso ideal, la función  $h$  produce distribución uniforme.

Si se tienen  $n$  elementos en una tabla de  $B$  baldes, se define el factor de carga como:

$$FC = n/B$$

Las listas resultan de largo promedio  $n/B$ .

Las operaciones demandan en promedio:  $O(1 + n/B)$

Considerando de costo 1, la evaluación de la función de hash; y  $n/B$  el recorrer la lista.

Si  $B$  es proporcional a  $n$ , las operaciones resultan de costo constante.

Llenar una tabla con  $n$  items tiene complejidad:  $O(n(1+n/B))$ . Ya que se debe buscar  $n$  veces.

##### 7.6.1.4.2. Distribución binomial:

Con el siguiente modelo:

El experimento de Bernouilli es: introducir una clave en uno de  $B$  baldes.

Se desea introducir  $i$  claves, en el mismo balde, en un conjunto de  $n$  experimentos.

Es decir: Se tienen  $i$  éxitos en  $n$  experimentos Bernouilli con probabilidad  $1/B$ .

Probabilidad de encontrar lista de largo  $i = \text{Binomial}(n, i) \left(\frac{1}{B}\right)^i \left(1 - \frac{1}{B}\right)^{(n-i)}$

Para 100 baldes se ilustran los largos promedios de las listas con 90 y 50 elementos en la tabla. Se aprecia que es baja la probabilidad de encontrar listas de largo mayor que 4.

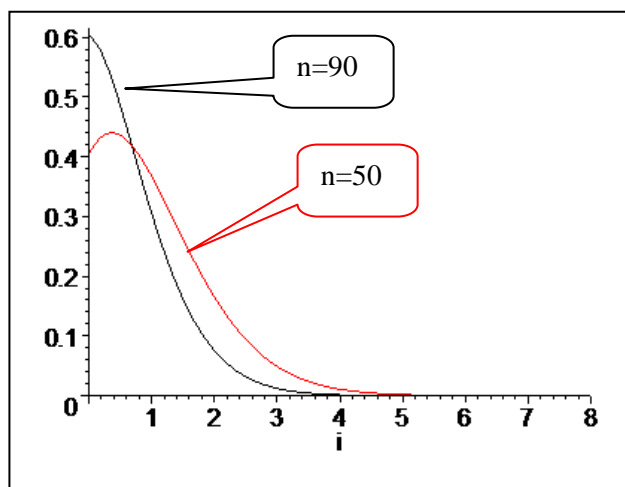


Figura 7.3 Probabilidad de encontrar listas de largo  $i$ . Tabla de 100 baldes.

## 7.6.2. Hash cerrado.

### 7.6.2.1. Estructura de datos.

La tabla es un arreglo de estructuras. Con un campo para la clave y otro para el estado de la celda. En este ejemplo se usan claves enteras.

Usamos el tipo enumerativo `state`, en lugar de códigos numéricos.

```
typedef enum { vacio, ocupado, descartado } state;
```

```
typedef struct moldecelda
```

```
{ int clave;
  state estado;
} tcelda;
```

```
#define B 10 /* 10 celdas */
```

```
static tcelda hashtab[B]; /*tabla de estructura */
```

```
int n; /*ocupados de la tabla
```

Se agrega la variable global  $n$ , para simplificar el diseño de las funciones.

### 7.6.2.2. Colisiones.

Las colisiones implican una estrategia de rehash.

Si la celda  $j$  asociada a  $h(x)$  está ocupada, debe buscarse o insertarse en las siguientes posiciones:

$$h_i(x) \text{ con } i = 1, 2, 3, \dots$$

Si todas las localizaciones están ocupadas, la tabla está llena, y la operación falla.

### 7.6.2.3. Hash lineal.

La más simple forma de rehash es colocar los elementos que colisionan en las posiciones siguientes al valor de hash  $j$ , en forma “ascendente”. Se denomina hash lineal (linear probing) al siguiente conjunto de posiciones:

$$h_i(x) = (h(x) + i) \% B \quad \text{con } i = 1, 2, 3, \dots$$

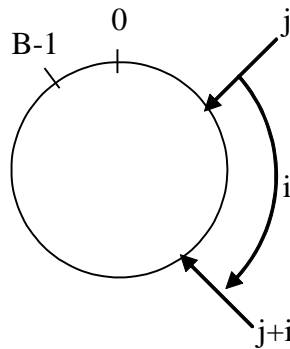


Figura 7.4 Posiciones siguientes para resolver colisiones en forma lineal.

Considerar las posiciones en aritmética módulo  $B$ , implica disponer el arreglo de manera circular, como muestra la Figura 7.4; de esta forma al aumentar  $i$ , la nueva posición  $(j+i) \% B$  podrá llegar a  $j$ , dando la vuelta completa. Si se usa  $(j+i)$  solamente, para posiciones mayores que  $B-1$ , se tendrán posiciones inexistentes.

De esta forma se genera una secuencia de elementos que tienen el mismo valor  $j$  de hash; estos elementos no son necesariamente consecutivos, debido a que algunas celdas pueden estar ocupadas por secuencias producidas por otros valores de hash, tanto anteriores como posteriores al valor  $j$ ; también pueden existir elementos que han sido descartados. En la Figura 7.5, se muestra la secuencia de elementos asociados al valor  $j$  de hash, los elementos descartados se ilustran rellenos de color gris, y los pertenecientes a otros valores de hash se muestran rellenos de color negro.

Si al inicio se marcan las entradas como vacías, la secuencia de  $l$  componentes, iniciada en la posición  $j$ , termina si se encuentra un elemento vacío o si se recorre todo el círculo, según reloj.

El número  $l$  debe ser igual al número de ocupados en la secuencia, que puede ser menor que el número total de ocupados  $n$ , más el número de descartados.

La búsqueda entonces es de complejidad  $O(l)$ . El largo de la secuencia  $l$ , varía dinámicamente con las inserciones y descartes, pero siempre debe cumplirse que  $l$  es menor o igual que  $B$ . El peor caso es  $O(B)$ , que se produce si las  $B$  claves tienen colisiones para el mismo valor de hash, sin embargo esto es difícil que se produzca en la práctica; como se verá más adelante el valor esperado es de complejidad constante.

Si se parte de la posición  $j$ , el número  $k=(j-1+B)\%B$  define el final de un recorrido circular, según reloj, al que se llega después de  $B$  intentos.

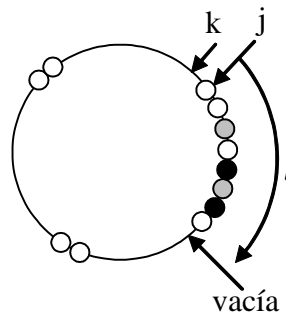


Figura 7.5. Secuencia asociada a colisiones con valor  $j$  de hash.

Si se desea eliminar un ítem de la tabla no se lo puede marcar como vacío, ya que se asumió que esta es una condición para el término de la secuencia de colisiones; debido a esto se agrega el estado descartado.

La búsqueda se detiene cuando se encuentra una posición vacía; pero debe seguir buscando si la posición está descartada, ya que el descarte puede haber roto una cadena de colisiones, asociada a la secuencia.

La inserción debe colocar el ítem en el primer elemento vacío o descartado de la secuencia que comienza en  $j$ ; sin embargo debe recorrer toda la secuencia asociada al valor de hash  $j$ , para determinar si en ésta se encuentra la clave que se desea insertar, ya que no se aceptan claves repetidas. Si se encuentra, en la secuencia la clave que se desea insertar, y la celda está ocupada es un error de inserción. Además no debe insertarse en una tabla llena.

En descarte se recorre la secuencia que comienza en  $j$  buscando la clave, si la encuentra y está marcada como ocupada se la descarta; si no la encuentra se tiene un error en la operación descarte. También debe considerarse la excepción de no descartar en una tabla vacía.

Entonces en hash lineal, las colisiones se resuelven probando en las siguientes posiciones de la tabla. Pero existen numerosas maneras de resolver colisiones en una tabla de hash cerrado.

Si luego de una colisión, el primer elemento que debe seleccionarse para colocar el nuevo ítem puede escogerse de  $(B-1)$  formas, la siguiente posición para colocar la siguiente colisión podrá escogerse de  $(B-2)$  formas; con lo cual pueden tenerse  $(B-1)!$  trayectorias posibles dentro de una tabla. El hash lineal es una de esas formas.

No todas las formas son satisfactorias, ya que algunas pueden producir apilamiento. Esto puede notarse observando que mientras más larga es una secuencia de colisiones, más probables son las colisiones con ella cuando se desee agregar nuevos elementos a la tabla. Adicionalmente, una secuencia larga de colisiones tiene gran probabilidad de colisionar con otras secuencias ya existentes, lo cual tiende a dispersar aún más la secuencia.

Si sólo se implementa buscar e insertar, las funciones resultan más sencillas, y sólo son necesarios los estados vacío y ocupado. Debido a las dificultades que genera la operación descartar en hash cerrado, no suele estar implementada, y si se desea tenerla es preferible emplear tablas de hash abiertas.

#### **7.6.2.4. Otros métodos de resolver colisiones.**

En el hash cuadrático se prueba en incrementos de dos.

$$j = (j + inc + 1) \% B; inc += 2;$$

En hash doble se usa otra función de hash, para calcular el incremento (número entre 0 y  $B-1$ ), para obtener la posición del siguiente intento.

En hash aleatorio, se genera un número aleatorio entre 0 y  $B-1$  para obtener el incremento.

En algunos casos, si las claves que pueden almacenarse en la tabla son fijas, puede encontrarse una función de hash perfecta que no produzca colisiones.

#### **7.6.2.5. Operaciones en tabla de hash cerrado lineal.**

##### **7.6.2.5.1. Crear tabla vacía.**

```
void DejarTablaVacía(void)
{
    int i;
    for ( i = 0; i < B; i++) hashtable[i].estado = vacío;
    n=0;
}
```

##### **7.6.2.5.2. Imprimir ítem y la tabla. Listador.**

```
void PrtItem(int i)
{
    if( i >= 0 && i < B )
        { printf( "Clave=%d Estado=", hashtable[i].clave);
          if( hashtable[i].estado == vacío) printf("vacío\n");
        }
```

```

    else if (hashtab[i].estado==ocupado) printf("ocupado\n");
    else if (hashtab[i].estado==descartado) printf("descartado\n");
    else printf("error en estado\n");
  }
  else printf("Item inválido\n");
}

```

```

void PrtTabla(void)
{ int i;
  if(n>0)
  { printf("Tabla\n");
    for(i=0; i<B; i++) PrtItem(i);
    putchar('\n');
  }
  else printf("Tabla vacía\n");
}

```

#### 7.6.2.5.3. Buscar.

```

int buscar(int clave)
{ int i, last;
  if (n>0)
  { for(i=hash(clave), last=(i-1+B)%B; i!=last && hashtab[i].estado != vacio; i=(i+1)%B)
    { if (hashtab[i].estado == descartado) continue;
      else if (hashtab[i].clave == clave) break; //sólo compara clave si está ocupado
    }
    if (hashtab[i].clave == clave && hashtab[i].estado == ocupado ) return (i);
    else { printf("Error en búsqueda: No encontró %d\n", clave); return (-1);}
  }
  else { printf("Error en búsqueda de clave %d: Tabla vacía\n", clave); return (-2);}
}

```

#### 7.6.2.5.4. Insertar.

```

int insertar(int clave)
{ int i, last, pd=-1; //en pd se almacena posición de primer descartado.
  //Al inicio esa posición no existe.
  if (n<B)
  {
    for(i=hash(clave),last=(i-1+B) % B; i!=last && hashtab[i].estado != vacio; i=(i+1)% B)
    { if (hashtab[i].estado == descartado) {if(pd == -1) pd=i; continue;}
      else if (hashtab[i].clave == clave) break; //sólo compara clave si está ocupado
    }
    if ( hashtab[i].clave == clave && hashtab[i].estado == ocupado )
    { printf("Error en inserción: Clave %d ya estaba en la tabla\n",clave);
      return(1);
    }
  }
  else

```

```

    { if (pd != -1) i=pd; //i apunta al primer descartado
      hashtable[i].clave = clave; hashtable[i].estado = ocupado; n++;
      return(0);
    }
  }
  else {printf("Error en inserción de clave %d. Tabla llena\n", clave); return(2);}
}

```

#### 7.6.2.5.5. Descartar.

```

int descartar(int clave)
{ int i, last;
  if(n!=0)
  {
    for(i=hash(clave),last=(i-1+B) % B; i!=last && hashtable[i].estado != vacio; i=(i+1)% B)
      {if (hashtable[i].estado == descartado) continue;
        else if (hashtable[i].clave == clave) break; //sólo compara clave si está ocupado
      }
    if (hashtable[i].clave == clave && hashtable[i].estado == ocupado )
      { hashtable[i].estado=descartado; n--; return (0);}
    else
      {printf("Error en descarte: No se encuentra activa la clave=%d\n",clave); return (1);}
  }
  else { printf("Error en descarte de clave %d: Tabla vacía\n", clave); return (2);}
}

```

### 7.7. Análisis de complejidad en hash cerrado.

Se asume que todas las claves posibles son igualmente probables, y que la función de hash las distribuye uniformemente en los baldes.

#### 7.7.1. En inserción:

Se desea calcular  $E_{k+1}$ , el valor esperado promedio de los intentos necesarios para insertar una clave en una tabla de  $n$  posiciones, cuando ya hay  $k$  posiciones ocupadas.

La probabilidad de encontrar un balde disponible en el primer intento de inserción es:

$$p_1 = \frac{\text{baldes disponibles primera vez}}{\text{total de baldes primera vez}} = \frac{n-k}{n}$$

La probabilidad de que exactamente se requieran dos intentos, es el producto de tener colisión la primera vez y la de encontrar una posición libre en el segundo intento.

$$p_2 = \text{colisión primera vez} * \frac{\text{disponibles segunda vez}}{\text{total segunda vez}}$$

$$p_2 = \frac{k}{n} * \frac{(n-1) - (k-1)}{(n-1)} = \frac{k}{n} * \frac{n-k}{n-1}$$

En la segunda vez hay  $(n-1)$  casillas disponibles, de las cuales pueden haber  $(k-1)$  ocupadas ya que se está seguro de que la primera estaba ocupada.

Tener colisión en el segundo intento se produce con probabilidad:

$$\frac{k-1}{n-1}$$

La probabilidad de encontrar balde disponible en el tercer intento, puede anotarse:

$$\frac{n-k}{n-2}$$

Entonces:

$$p_3 = \frac{k}{n} * \frac{k-1}{n-1} * \frac{n-k}{n-2}$$

Nótese que el denominador de la última fracción siempre será  $(n-k)$ .

Finalmente la inserción, después de  $i$  intentos:

$$p_i = \frac{k-0}{n-0} * \frac{k-1}{n-1} * \frac{k-2}{n-2} \dots * \frac{k-i+2}{n-i+2} * \frac{n-k}{n-i+1}$$

El numerador de la probabilidad de la tercera colisión es  $k-(3-1)$ . Entonces el denominador de la probabilidad de que la última colisión sea en el intento  $(i-1)$ , es  $k-((i-1)-1)$ .

El denominador de la probabilidad de encontrar un balde disponible en el tercer intento es  $n-(3-1)$ . Entonces el denominador de la probabilidad de encontrar un balde disponible en el intento  $i$ , es  $n-(i-1)$ .

El número esperado de intentos requeridos para insertar una clave, cuando ya hay  $k$  en una tabla de  $n$  posiciones es:

$$E_{k+1} = \sum_{i=1}^{k+1} i p_i$$

También es el número de intentos esperado para insertar la clave  $(k+1)$ .

La sumatoria anterior es compleja de calcular, pero puede resolverse inductivamente:

Poner el primero cuando la tabla está vacía; es decir  $k=0$ , puede anotarse:

$$E_{0+1} = 1 * p_1 = 1 * \frac{n-0}{n} = 1$$

Poner el segundo, cuando ya hay uno en la tabla; es decir con  $k=1$ :



$$E_{1+1} = 1 * p_1 + 2 * p_2 = 1 * \frac{n-1}{n} + 2 * \frac{1}{n} * \frac{n-1}{n-1} = \frac{n+1}{n-1+1}$$

Poner el tercero, cuando ya hay dos, con  $k=2$ :

$$E_{2+1} = 1 * p_1 + 2 * p_2 + 3 * p_3 = 1 * \frac{n-2}{n} + 2 * \frac{2}{n} * \frac{n-2}{n-1} + 3 * \frac{2}{n} * \frac{2-1}{n-1} * \frac{n-2}{n-2}$$

$$E_{2+1} = \frac{n+1}{n-2+1}$$

De lo anterior puede inducirse, que el número esperado de intentos requeridos para insertar una clave, cuando ya hay  $k$  en una tabla de  $n$  posiciones es:

$$E_{k+1} = \frac{n+1}{n-k+1}$$

### 7.7.2. En búsqueda:

Se desea calcular  $E_B$ , el número de intentos necesarios para acceder una clave cualquiera en una tabla en la que hay  $m$  elementos.

Podemos aprovechar el resultado anterior, observando que el número de intentos para buscar un ítem es igual al número de intentos para insertarlo.

Entonces podemos plantear que:

$E_B$  = número promedio de intentos para insertar  $m$  elementos en la tabla.

De antes teníamos que para insertar el primer elemento se requerían  $E_1$  intentos. Para insertar el segundo se requieren  $E_2$  intentos, y así sucesivamente hasta que para insertar el  $m$ avo elemento se requieren  $E_m$  intentos.

Entonces podemos calcular  $E_B$  como el promedio:

$$E_B = \frac{E_1 + E_2 + \dots + E_m}{m} = \frac{1}{m} * \sum_{k=1}^m E_k$$

Reemplazando los valores de  $E_k$ , se obtiene:

$$E_B = \frac{n+1}{m} \left( \frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \dots + \frac{1}{n-m+2} \right)$$

Para evaluar la sumatoria anterior, puede emplearse la siguiente sumatoria, conocida como la función armónica, ya que para ella se conoce, en forma aproximada su suma:

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \approx \ln(n) + \gamma$$

Tenemos que:

$$H_{n-m+1} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n-m+1}$$

Con  $\gamma = 0,577216..$  denominada constante de Euler.

Formando  $H_{n+1}$  y restando el término anterior, en la expresión para  $E_B$ , obtenemos:

$$E_B = \frac{n+1}{m} (H_{n+1} - H_{n-m+1}) = \frac{n+1}{m} (\ln(n+1) - \ln(n-m+1))$$

Agrupando las funciones logaritmo y definiendo el factor de carga como:

$$\alpha = \frac{m}{n+1}$$

Se obtiene:

$$E_B = \frac{n+1}{m} \ln\left(\frac{n+1}{n+1-m}\right) = -\frac{\ln(1-\alpha)}{\alpha}$$

Tabla vacía implica  $\alpha=0$ ; tabla llena implica  $\alpha=n/(n+1) \approx 1$

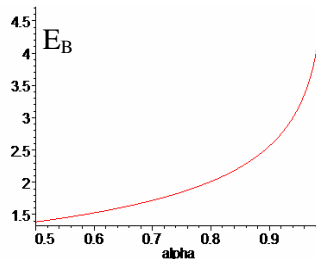


Figura 7.6 Intentos versus Factor de carga. Hash cerrado.

Para tabla casi llena, con  $\alpha=99\%$  se tiene una búsqueda en menos de 5 intentos. Puede decirse que en promedio se requieren 3 ó 4 intentos para buscar un elemento, con tabla casi llena.

## 7.8. Ventajas y desventajas.

La tabla de hash es mejor que los árboles binarios de búsqueda en inserción y búsqueda.

Sin embargo tiene las limitaciones de los métodos estáticos, ya que el tamaño de la tabla debe ser estimada a priori; además la complejidad aumenta si se implementa la operación descarte.

Si el volumen de los datos cambia dinámicamente, creciendo o disminuyendo hasta límites no previsibles, se suelen emplear árboles de búsqueda.

## Problemas resueltos.

### P7.1.

Para una tabla de hash cerrado de 10 posiciones, se tienen las siguientes funciones:

```
void imprime(void)
{ int i;
  for(i=0; i<B; i++)
  {
    if (hashtab[i].estado==vacío)    printf("%2d v ", hashtab[i].clave);
    else if (hashtab[i].estado==descartado) printf("%2d d ", hashtab[i].clave);
    else if (hashtab[i].estado==ocupado) printf("%2d o ", hashtab[i].clave);
    else printf("%2d e ", hashtab[i].clave);
  }
  putchar('\n');
}
```

```
int hash(int key)
{ return key%B; }
```

Determinar que escribe el programa:

```
makenull();
insertar(5); insertar(6); insertar(7); imprime();
insertar(15); insertar(16); insertar(17); imprime();
descartar(6); insertar(25); descartar(7); imprime();
printf("%d e\n", buscar(15));
insertar(14); insertar(4); insertar(24); imprime();
```

Puede efectuar un diagrama con los datos.

Solución.

Un diagrama con los valores y estado del arreglo, a medida que se ejecutan las instrucciones:

Índice	1	2	3	4	5	6
0	vacío	vacío	17 ocupado	17 ocupado	17 ocupado	17 ocupado
1	vacío	vacío	vacío	vacío	vacío	24 ocupado
2	vacío	vacío	vacío	vacío	vacío	vacío
3	vacío	vacío	vacío	vacío	vacío	vacío
4	vacío	vacío	vacío	vacío	vacío	14 ocupado
5	vacío	5 ocupado	5 ocupado	5 ocupado	5 ocupado	5 ocupado
6	vacío	6 ocupado	6 ocupado	6 descartado	25 ocupado	25 ocupado
7	vacío	7 ocupado	7 ocupado	7 ocupado	7 descartado	4 ocupado
8	vacío	vacío	15 ocupado	15 ocupado	15 ocupado	15 ocupado
9	vacío	vacío	16 ocupado	16 ocupado	16 ocupado	16 ocupado

Columna 1: Después de makenull

Columna 2: Después de insertar el 5, 6 y 7.

Columna 3: después de insertar el 15, 16 y 17.

Columna 4: después de descartar el 6.

Columna 5: después de insertar el 25 y descartar el 7.

Columna 6: después de insertar el 14, el 4 y el 24.

Para la parte de resultados impresos:

Algunas observaciones sobre la rutina imprime:

La línea: `printf("%2d e ", hashtable[i].clave);` no se ejecuta nunca.

Se ha **comentado** la impresión de la clave, en caso de estar la celda vacía, ya que makenull, no inicializa el campo clave. En lugar de imprimir la clave, se anota el índice de la celda vacía.

El programa imprimiría:

```
0 v 1 v 2 v 3 v 4 v 5 o 6 o 7 o 8 v 9 v
17 o 1 v 2 v 3 v 4 v 5 o 6 o 7 o 15 o 16 o
17 o 1 v 2 v 3 v 4 v 5 o 25 o 7 d 15 o 16 o
8 e
17 o 24 o 2 v 3 v 14 o 5 o 25 o 4 o 15 o 16 o
```

### **P7.2.**

Para una tabla de hash cerrado de tamaño 9, se tiene la siguiente función de hash:

$$h(x) = (7x + 1) \% 9;$$

Considere la siguiente secuencia de operaciones:

Insertar 13

Insertar 20

Insertar 4

Insertar 8

Descartar 4

Buscar 8

Insertar 16

Indice	Valor	Estado
0		
1		
2		
3		
4		
5		
6		
7		
8		

- a) Muestre el contenido de la tabla después de realizadas las operaciones anteriores, asumiendo linear probing para resolver colisiones. Indicando cuándo se producen colisiones y la razón por la que se almacena en una posición determinada.
- b) Luego de lo anterior, indicar fundamentadamente qué casos de inserciones o búsquedas tienen mayor costo, indicando el número de comparaciones que son necesarias.

**Solución.**

$(13*7+1) \% 9 = 2$ ,  $(20*7+1) \% 9 = 6$ ,  $(4*7+1) \% 9 = 2$ ,  $(8*7+1) \% 9 = 3$ ,  
 $(16*7+1) \% 9 = 5$ ,

Se inserta el 13 en posición 2 de la tabla, ya que estaba vacía. El 20 en la posición 6, que estaba vacía.

Al insertar el 4 se produce colisión en la entrada 2, de la tabla, se almacena en la casilla siguiente. Al insertar el 8 se produce colisión en la entrada 3, de la tabla, se almacena en la casilla siguiente.

Indice	Valor	Estado
0		vacío
1		vacío
2	13	ocupado
3	4	ocupado
4	8	ocupado
5		vacío
6	20	ocupado
7		vacío
8		vacío

Busca primero el 4 en la posición 2, como está ocupada, prueba en la siguiente, lo encuentra y lo descarta. La tabla queda:

Indice	Valor	Estado
0		vacío
1		vacío
2	13	ocupado
3	4	descartado
4	8	ocupado
5		vacío
6	20	ocupado
7		vacío
8		vacío

Al buscar el 8, como la entrada 3 de la tabla no está vacía, se busca en la siguiente y encuentra el valor en la casilla siguiente.

Luego de esto se inserta el 16, en la posición 5, la tabla queda:

Indice	Valor	Estado
0		vacío
1		vacío
2	13	ocupado
3	4	descartado
4	8	ocupado
5	16	ocupado
6	20	ocupado
7		vacío
8		vacío

b) Buscar un valor tal que el retorno de la función de hash tenga valor 2, y que su valor sea diferente de los valores no descartados u ocupados, implica **6** comparaciones, ya que la búsqueda se detiene al encontrar una casilla vacía.

La inserción de un valor tal que el retorno de la función de hash tenga valor 2, implica 2 comparaciones, ya que la búsqueda para insertar se detiene al encontrar una casilla descartada o vacía.

Buscar un valor tal que el retorno de la función de hash tenga valor 3, implica 5 comparaciones, ya que la búsqueda se detiene al encontrar una casilla vacía.

La inserción de un valor tal que el retorno de la función de hash tenga valor 3, implica 1 comparación, ya que la búsqueda se detiene al encontrar una casilla descartada o vacía.

Buscar un valor tal que el retorno de la función de hash tenga valor 4, implica 4 comparaciones, ya que la búsqueda se detiene al encontrar una casilla vacía.

La inserción de un valor tal que el retorno de la función de hash tenga valor 4, implica **4** comparaciones, ya que la búsqueda se detiene al encontrar una casilla descartada o vacía.

Entonces:

La búsqueda de un valor tal que el retorno de la función de hash tenga valor 2 es la que tiene mayor costo en comparaciones, requiere seis comparaciones.

La inserción de un valor tal que el retorno de la función de hash tenga valor 4 es la que tiene mayor costo en comparaciones, requiere cuatro comparaciones.

## Ejercicios propuestos.

### ***E7.1.***

Se desea reemplazar una tabla de hash abierta de B1 baldes, que tiene bastante más que B1 elementos, por otra tabla de hash de B2 baldes. Describir la construcción de la nueva tabla a partir de la vieja.

### ***E7.2. Hash Cerrado.***

a) El descarte de registros produce contaminación y disminuye la eficiencia de la búsqueda en una tabla de hash cerrada. ¿Por qué?

b) Para mejorar la eficiencia se propone mantener un contador del largo de la secuencia de prueba más larga (**csml**).

Modifique los algoritmos, vistos en clases para insertar y buscar, para mejorar la búsqueda con **csml**. ¿Dónde se mantiene el contador?

***E7.3. Para la funciones definidas en 7.6.2.5. Determinar las impresiones.***

```
//Test de las funciones
```

```
void main(void)
```

```
{ DejarTablaVacía(); PrtTabla(); PrtItem(buscar(5)); descartar(5); insertar(5);
```

```
  PrtItem(buscar(5)); PrtItem(buscar(6)); insertar(5); descartar(6); insertar(6);
```

```
  PrtItem(buscar(6)); insertar(7); PrtTabla();
```

```
  DejarTablaVacía();
```

```
  insertar(6); insertar(7); insertar(16); descartar(7); insertar(5); insertar(26); PrtTabla();
```

```
  DejarTablaVacía();
```

```
  insertar(5); insertar(6); insertar(7); insertar(15); descartar(6); descartar(7); insertar(25);
```

```
  insertar(26); PrtTabla();
```

```
}
```

***P7.4. Hash abierto y cerrado.***

Diseñar funciones para una tabla de hash abierto que resuelve las colisiones empleando una tabla de hash cerrado. Ver 11.5 en texto de Cormen.

**Referencias.**

Niklaus Wirth, “Algorithms + Data Structures = Programs”, Prentice-Hall 1975.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. “*Introduction to Algorithms*”, Second Edition. MIT Press and McGraw-Hill, 2001.

## Índice general.

<b>CAPÍTULO 7.....</b>	<b>1</b>
<b>TABLAS DE HASH. ....</b>	<b>1</b>
7.1. OPERACIONES. ....	1
7.2. CLAVE. ....	1
7.3. TABLA DE ACCESO DIRECTO. ....	1
7.4. TABLAS DE HASH. ....	2
7.5. FUNCIÓN DE HASH. ....	2
7.5.1. Funciones de hash para enteros. ....	3
7.5.2. Funciones de hash para strings alfanuméricos. ....	4
7.6. TIPOS DE TABLA. ....	6
7.6.1. HASH ABIERTO. ....	6
7.6.1.1. Diagrama de la estructura. ....	6
7.6.1.2. Declaración de tipos, definición de variables. ....	7
7.6.1.3. Operaciones en hash abierto. ....	7
7.6.1.3.1. Crear tabla vacía. ....	7
7.6.1.3.2. Función de hash. ....	7
7.6.1.3.3. Buscar si un string está en la tabla. ....	7
7.6.1.3.4. Insertar string en la tabla. ....	7
7.6.1.3.5. Descartar. ....	8
7.6.1.4. Análisis de complejidad en hash abierto. ....	9
7.6.1.4.1 Caso ideal. ....	9
7.6.1.4.2. Distribución binomial: ....	9
7.6.2. HASH CERRADO. ....	10
7.6.2.1. Estructura de datos. ....	10
7.6.2.2. Colisiones. ....	10
7.6.2.3. Hash lineal. ....	11
7.6.2.4. Otros métodos de resolver colisiones. ....	13
7.6.2.5. Operaciones en tabla de hash cerrado lineal. ....	13
7.6.2.5.1. Crear tabla vacía. ....	13
7.6.2.5.2. Imprimir ítem y la tabla. Listador. ....	13
7.6.2.5.3. Buscar. ....	14
7.6.2.5.4. Insertar. ....	14
7.6.2.5.5. Descartar. ....	15
7.7. ANÁLISIS DE COMPLEJIDAD EN HASH CERRADO. ....	15
7.7.1. En inserción: ....	15
7.7.2. En búsqueda: ....	17
7.8. VENTAJAS Y DESVENTAJAS. ....	18
PROBLEMAS RESUELTOS. ....	19
P7.1. ....	19
P7.2. ....	20
EJERCICIOS PROPUESTOS. ....	22
E7.1. ....	22
E7.2. Hash Cerrado. ....	22
E7.3. Para la funciones definidas en 7.6.2.5. Determinar las impresiones. ....	23
P7.4. Hash abierto y cerrado. ....	23
REFERENCIAS. ....	23



Tablas de hash	25
ÍNDICE GENERAL. ....	24
ÍNDICE DE FIGURAS.....	25

## Índice de figuras.

FIGURA 7.1 TABLA DE ACCESO DIRECTO.....	1
FIGURA 7.2 TABLA DE HASH ABIERTO O DE ENCADENAMIENTO DIRECTO. ....	6
FIGURA 7.3 PROBABILIDAD DE ENCONTRAR LISTAS DE LARGO I. TABLA DE 100 BALDES. ....	10
FIGURA 7.4 POSICIONES SIGUIENTES PARA RESOLVER COLISIONES EN FORMA LINEAL. ....	11
FIGURA 7.5. SECUENCIA ASOCIADA A COLISIONES CON VALOR $J$ DE HASH. ....	12
FIGURA 7.6 INTENTOS VERSUS FACTOR DE CARGA. HASH CERRADO. ....	18