Tema 7
Tablas Hash

1. Búsqueda interna
2. Tabla hash: idea general
3. Tabla hash: especificación
4. Funciones hash
5. Resolución de colisiones
6. Redispersión
7. Resumen de costes en resolución de colisiones
8. Aplicaciones
9. Ejemplo

1. Búsqueda interna

- La búsqueda interna trabaja con elementos almacenados en memoria principal.

- Se puede realizar sobre estructuras estáticas o dinámicas.

- Métodos más utilizados:
  - Búsqueda secuencial o lineal
  - Búsqueda binaria
  - Árboles de búsqueda
  - Búsqueda por transformación de claves

- **Búsqueda lineal**
  - Consiste en revisar elemento tras elemento hasta encontrar el dato buscado o llegar al final del conjunto de datos disponible.
  - Se pueden distinguir varios casos:
    - Estructura estática (array)
      - Desordenado
      - Ordenado
    - Estructura dinámica (lista enlazada)
      - Desordenada
      - Ordenada
  - En el caso de un conjunto ordenado, se puede restringir la condición de finalización (cuando el elemento analizado es mayor/menor que el buscado)

Búsqueda lineal (análisis)

- ¿Número de comparaciones **c**?

- Conjunto desordenado de N elementos:
  - El elemento no existe => **c**=N comparaciones
  - El elemento existe=> 1<=**c**<=N
  - Valor promedio: $\mathbf{c_{med}=(1+N)/2}$

- **Búsqueda binaria**
  - Se divide el espacio de búsqueda en dos partes, comparando el elemento buscado con el central del array.
  - Sólo sirve para arrays ordenados
  - Con cada iteración, el espacio de búsqueda se reduce a la mitad.

- Búsqueda binaria (análisis)
  - ¿Número de comparaciones **c**?
  - Conjunto ordenado de **N** elementos:
    - Más favorable: el elemento buscado es el central (**c=1**)
    - Más desfavorable: el elemento buscado no existe (**$c=\log_2(N)$**)
    - Caso promedio: **$c_{med}=(1+\log_2(N))/2$**
  - Más eficiente que la búsqueda lineal, pero sólo sirve para array ordenados
  - Si no están ordenados: añadir el coste de la ordenación, que sólo compensa si es necesario hacer muchas búsquedas

- ¿Podemos realizar una búsqueda en un tiempo mejor que O(log n) ?
  - La operación de acceso a memoria en un ordenador se realiza en mucho menos tiempo: se toma una clave (la dirección de la memoria) para insertar o recuperar elementos de datos (la palabra de la memoria) en un tiempo constante O(1).
  - Los tiempos de acceso a la memoria de un ordenador no aumentan con el tamaño de la memoria.

- **Direccionamiento directo**
  - El acceso a la memoria del ordenador es un caso especial de una técnica denominada direccionamiento directo, en la que la clave conduce directamente al elemento buscado.
  - El almacenamiento de datos en arrays es un ejemplo de este método, en el que el índice del array juega el papel de clave.
  - El problema de los esquemas de direccionamiento directo reside en que requieren un **almacenamiento equivalente al rango de todas las posibles claves** y no el proporcional al número de elementos que realmente se almacenan.

- **Búsqueda por transformación de claves**
  - Los dos métodos de búsqueda anteriores (lineal, binaria) permiten encontrar un elemento en un array.
  - El tiempo de búsqueda **c** es función del número de componentes N: **mayor N=> mayor c**
  - Un método alternativo, conocido como **transformación de claves** o **hash**, permite:
    - Aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados.
    - Lograr que el tiempo de búsqueda sea independiente del tamaño del array.

- ¿Qué es una tabla hash?
  - Queremos implementar de forma eficiente las operaciones de inserción, borrado y búsqueda en un conjunto de elementos.
  - Buscamos una implementación en que dichas operaciones se realicen en un tiempo constante en media, independientemente del número de datos.
  - A cada dato se le asigna, mediante una fórmula matemática denominada **función hash**, una posición única en una tabla, con lo que la búsqueda, la inserción y el borrado son inmediatos: **O(1)**.

2. Tabla hash: idea general

- **Ejemplo 1**

Queremos almacenar información relativa a 100 alumnos, con número de matrícula del 1 al 100.

- Si definimos un array de 100 elementos con índices de 1 a 100, podemos usar el índice como identificador de cada alumno.

- **Ejemplo 2**

Queremos almacenar información relativa a 100 empleados, donde la clave de cada empleado es su número de la seguridad social. Con un número de 12 dígitos, tendríamos que definir un vector de 999 999 999 999 elementos para almacenar solamente 100 empleados.

- **Problema**: el coste de memoria sobrepasa al ahorro obtenido por el acceso directo

- **Ejemplo 3**

Queremos guardar un conjunto de registros sobre personas, cada una de ellas representada por su DNI (es un conjunto pequeño, hay menos de 15 personas).

- Para ello utilizamos un vector de 27 componentes, indexadas de 0 a 26, correspondientes a las posibles letras finales.
- Colocamos cada persona en la posición del vector que indica su letra de NIF:
  - **La A corresponde a la posición 0, la B a la 1, etc.**
- Para encontrar una persona basta saber su letra de NIF

- **¿Qué problemas hay?**

2. Tabla hash: idea general

Como suele ocurrir, lo que se gana por un lado se pierde por otro.

En el caso de la tabla hash los inconvenientes son:

- Al tratarse de un array, el **tamaño** de la tabla está **limitado** y **debe fijarse desde el principio**.
- Como las posiciones ocupadas no tienen porque ser consecutivas, el contenido de una tabla hash **no se puede recorrer** de forma directa
- Como la posición de un elemento se calcula de forma matemática, los datos **no** pueden almacenarse **ordenados**.
- Si la función hash proporciona datos duplicados se produce lo que se denomina **colisión**, que consiste en que a dos elementos se les asigna la misma posición en el array.

3. Tabla hash: especificación

Queremos implementar un TAD, identificado mediante claves, en el que inserciones, borrados y búsquedas se realicen de la forma más eficiente posible.

Las tablas hash se aplican cuando el conjunto de claves posibles es mucho mayor que el de claves reales a almacenar.

- Una **tabla hash** es un vector de tamaño prefijado, en el que se almacenan los elementos según sus claves.
- Cada **clave** representa un elemento del conjunto.
- Se necesita una función (**función hash**) que asocie cada clave a una componente del vector.
- Se necesita solucionar el problema de las **colisiones**: cuando varias claves corresponden a la misma componente del vector.

3. Tabla hash: especificación

Informal

TAD TablaHash (VALORES: Atributos almacenados en las posiciones resultado de una transformación Hash de claves; OPERACIONES: Inicializar, Hash, Buscar, Insertar, Eliminar)

Inicializar → TablaHash

Efecto: Crea una tabla hash vacía.

Hash (Clave) → Posicion

Efecto: Devuelve la transformación adecuada a partir de la clave.

Buscar (TablaHash, Clave) → Atributo, Boolean

Efecto: Si la clave está en la tabla, devuelve el atributo correspondiente y un valor lógico llamado error a false. En caso contrario error valdrá true.

Excepción: Error si la clave no está en la tabla.

Insertar (TablaHash, Clave, Atributo) → TablaHash, Boolean

Efecto: Si la posición no está libre, el valor lógico error vale true. En caso contrario en la posición Hash (clave) se almacena el par (clave, atributo).

Excepción: Error si hay colisión, desbordamiento o tabla llena.

Eliminar (TablaHash, Clave) → TablaHash, Boolean

Efecto: Si la clave existe en la tabla, se deja vacía esa posición y se actualiza error a false. En caso contrario, error vale true.

Excepción: Error si la clave no está en la tabla.

4. Funciones hash

Suponemos que la tabla hash está indexada desde 0 a N-1, siendo N el tamaño de la tabla, definido a priori.

- Definimos una función hash:
  - **h: conjunto_de_claves → {0, 1, ..., N-1}**
- Queremos que la función hash:
  - **Sea muy rápida de calcular**
  - **Tenga pocas colisiones, es decir, distribuya las claves de la forma más uniforme posible entre los valores de 0 a N-1**
- Existen varias posibilidades:
  - **Si las claves son enteros**
  - **Si las claves son cadenas de caracteres**

Funciones hash para claves enteras (i)

Función hash por módulo: h(K)=K mod N

- Puede funcionar mal. Por ejemplo, ¿qué ocurre si N=10 y casi todas las claves terminan en 10?
- **Es conveniente elegir un número primo para N**

- Ejemplo:
  - **Si fijamos N=100 y calculamos la clave para K1=7259; K2=9359**

    h(K1)=7259 mod 100 = 59 h(K2)=9359 mod 100 =59 => COLISIÓN
  - **Si fijamos N=97 (número primo próximo a 100):**

    h(K1)=7259 mod 97 = 81 h(K2)=9359 mod 97 = 47 => NO HAY COLISIÓN

Funciones hash para claves enteras (ii)

Función hash cuadrado: h(K)=dígitos centrales(K^2)

- ¿Cuántos dígitos se toman? En función del tamaño de la tabla, suelen tomarse **log(N)**

- Ejemplo:
  - **Si fijamos N=100, log(100)=2 dígitos centrales**
  - **Calculamos la clave para K1=7259 y K2=9359**

    $K1^2$=52693081 h(K1)=93

    $K2^2$=87590881 h(k2)=90

Funciones hash para claves enteras (iii)

Función hash por plegamiento

$$h(K)=dig\_men\_sig((d_1 \dots d_i)+(d_{i+1} \dots d_j)+\dots+(d_l \dots d_n))$$

- K=clave con dígitos $d_1, d_2, \dots, d_n$
- Se divide la clave en partes con el mismo número de dígitos (salvo la última parte, que puede tener menos), se opera con ellas (suma, producto) y se asigna como dirección los **dígitos menos significativos**.
- ¿Cuántos dígitos se toman? En función del tamaño de la tabla, suelen tomarse **log(N)**

- Ejemplo:
  - **Si fijamos N=100 y calculamos la clave para K1=7259 y K2=9359**

    h(K1)=dig_men_sig(72+59)=dig_men_sig(131)=31

    h(K2)=dig_men_sig(93+59)=dig_men_sig(152)=52

Funciones hash para cadenas de caracteres (i)

Método de la división

- **Dada una cadena c:**
  - **Sumar los valores ascii(c[i]) para todo i**
  - **Calcular el resto módulo N**
  - No funciona bien si N es demasiado grande, por ejemplo ¿qué sucede si N=10007, y todas las cadenas tienen 8 o menos caracteres?
  - Hay muchos otros casos en los que da problemas. Por ejemplo, si N=100 y las claves son 'A18', 'A27', 'A36', …, 'A90'
    - SUMA(ascii('A'),ascii('1'),ascii('8'))mod 100=(65+49+56) mod 100 =170 mod 100 =70
    - SUMA(ascii('A'),ascii('2'),ascii('7'))mod 100=(65+50+55) mod 100 =170 mod 100 =70 …

Funciones hash para cadenas de caracteres (ii)

Otros métodos

- **Dada una cadena c: Calcular la suma PONDERADA de los valores ascii(c[i]) para todo i**
  - por ejemplo, interpretando la cadena como un entero en base 256 (o menos, dependiendo del conjunto de caracteres usados)

```
algoritmo h(c: cadena[maxTam]) return 0..maxTam-1;
  i,suma:entero;
  principio
    suma:=0;
    para i:=maxTam-1 hasta 0 hacer
      suma:=(suma*256+ord(c[i])) mod maxTam;
   finpara
   devuelve(suma);
  fin
```

5. Resolución de colisiones

- **Colisión**: se produce cuando la función hash proporciona la misma dirección para dos claves diferentes
  - En el ejemplo de los DNI, las colisiones se producen entre personas con la misma letra de NIF.
- La resolución de colisiones siempre es **COSTOSA**, por lo que debemos evitarlas en la medida de lo posible seleccionando una buena función hash.
  - Solución "drástica": reservar una casilla por clave posible (alto coste en memoria).
  - Analizar alternativas que equilibren uso de memoria y tiempo de búsqueda.

Las dos alternativas más utilizadas son:

- **RECOLOCACIÓN O REASIGNACIÓN**
  - Si la posición de la tabla hash donde debe situarse el elemento está ocupada, se prueba en otra posición.
  - Tres tipos:
    - Recolocación simple
    - Recolocación lineal
    - Recolocación cuadrática
- **ENCADENAMIENTO**
  - Se coloca en cada posición de la tabla hash una lista de todos los elementos que deben situarse en esa posición.

Resolución de colisiones por recolocación (i): Recolocación simple

Si h es la función hash, la recolocación simple consiste en:

- INSERCIÓN: Al insertar una clave c, si la posición h(c) está ocupada, se comprueban las posiciones siguientes:

  $h(c)+1 \pmod N, h(c)+2 \pmod N, h(c)+3 \pmod N, \ldots$

  hasta encontrar una libre, en la que se inserta c
- BÚSQUEDA: Se repite el mismo proceso, es decir, se busca en $h(c)+1 \pmod N, h(c)+2 \pmod N, h(c)+3 \pmod N, \ldots$ hasta encontrar la clave o llegar a una posición vacía, en cuyo caso se entiende que la clave buscada no está.
- **PROBLEMA: ¿qué sucede si elimino elementos?**

Resolución de colisiones por recolocación (ii): Recolocación simple

- **BORRADO**
  - Al borrar una posición podemos hacer imposible encontrar otras claves que se insertaron cuando esa posición estaba ocupada, ¿por qué?
  - **Solución**: **Marcar la posición como borrada**, es decir, que las posiciones tengan una etiqueta que tome los valores: "libre", "ocupada" y "borrada"
- **COSTES**
  - El coste de una inserción, búsqueda o borrado depende del número de posiciones que es necesario examinar (número de ensayos o pruebas)
  - **Problema:** se forman bloques grandes de posiciones en la tabla llenas que están seguidas, y cuando se entra en un bloque hay que recorrerlo entero.

Resolución de colisiones por recolocación (iii): Recolocación lineal

Si h es la función hash, y **a** un número fijo entre **1** y **N-1**, la recolocación lineal (para inserción o búsqueda) consiste en:

- Al insertar una clave c, si la posición h(c) está ocupada, se comprueban las posiciones siguientes:

  $$h(c)+a(\text{mod } N), h(c)+2a(\text{mod } N), \ldots$$

  hasta encontrar una libre, en la que se inserta c
- **PROBLEMA:** puede que no encontremos una posición libre
- **SOLUCIÓN:** elegir un valor de **a primo con N**

Resolución de colisiones por recolocación (iv): Recolocación lineal

- **COSTES**
  - La media del número de ensayos en una tabla con **n** posiciones ocupadas es de $\frac{1}{2}\left[1+\frac{N^2}{(N-n)^2}\right]$
  - La relación entre el número de elementos almacenados, **n**, y el número de posiciones de la tabla, **N**, se denomina **FACTOR DE CARGA L**

$$L = \frac{n}{N}$$

Cuando $L \leq \frac{1}{2}$ (o, lo que es lo mismo, $n \leq \frac{N}{2}$), esta media es constante.

Resolución de colisiones por recolocación (v): Recolocación cuadrática

Si h es la función hash, la recolocación cuadrática (para inserción o búsqueda) consiste en:

- Al insertar una clave c, si la posición h(c) está ocupada, se comprueban las posiciones siguientes:

  $h(c)+1 \pmod N, h(c)+4 \pmod N, h(c)+9 \pmod N, \ldots, h(c)+i^2 \pmod N, \ldots$

  hasta encontrar una libre, en la que se inserta c

- **PROBLEMA**: puede que no encontremos una posición libre, pero esto sólo ocurre si la tabla está casi llena

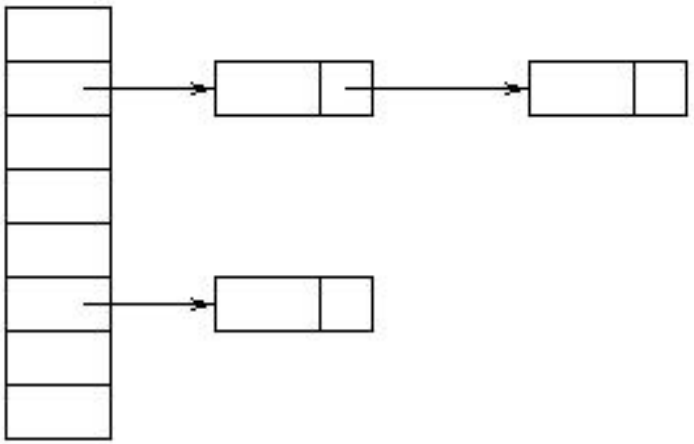
- **SOLUCIÓN:** Mantener $L \leq \frac{1}{2}$ (o, lo que es lo mismo, $n \leq \frac{N}{2}$)

Resolución de colisiones por recolocación (v): Recolocación cuadrática

- **COSTES :**
  - Cuando $L \leq \frac{1}{2}$ (o, lo que es lo mismo, $n \leq \frac{N}{2}$), la media del número de ensayos es constante.
  - No se forman bloques de posiciones llenas

Resolución de colisiones por encadenamiento (i)

- En estas tablas hash, cada posición del vector contiene una **lista enlazada de las claves que corresponden a esa posición**
  - Se almacenan en cada lista todas las claves que "colisionan" (similar a la representación de grafos mediante listas de adyacencia)



¿Es un método eficiente?

¿Cuánto cuestan las inserciones, búsquedas y borrados?

¿En el caso peor? ¿En media?

Resolución de colisiones por encadenamiento (ii)

- **Eficiencia**

  Suponemos que:

  - Tenemos una tabla de tamaño **N** e insertamos **n** claves.
  - La función hash distribuye uniformemente, lo que implica que la longitud media de las listas será de **n/N** elementos.
    - Un incremento de **N** reduce la longitud pero incrementa el espacio de la tabla.
  - La función hash se puede calcular en tiempo constante

Resolución de colisiones por encadenamiento (iii)

- **Eficiencia**
  - **Inserción:** requiere calcular la función hash e insertar al principio de la lista => **CONSTANTE**
  - **Búsqueda y borrado:** requieren calcular la función hash y recorrer una lista, por lo que serán las operaciones más costosas.
    - En el caso peor, la búsqueda costará **L** (=n/N)
    - En media, la búsqueda costará: (1+2+3+… +L)/L=**(L+1)/2**

      Si **n<=N**, la búsqueda cuesta, de media, un tiempo constante

Resolución de colisiones por encadenamiento (iv)

- Como las listas enlazadas pueden contener un número arbitrario de elementos, no existe límite en la capacidad de la tabla.
- Pero si la función hash no distribuye bien las claves, el rendimiento de la tabla hash disminuirá.
- El peor caso (como en un ABB) es aquel en el que se almacenan TODOS los datos en una única lista enlazada: cuando todas las claves se dispersan al mismo elemento en la tabla.
- **Para obtener mayor eficiencia en este tipo de tablas hash, se recomienda un factor de carga L (=n/N) por debajo de 0,75**
- Sería posible mejorar este esquema sustituyendo las N listas de colisión por árboles equilibrados, pero no merece la pena si el factor de carga se mantiene en valores pequeños.

6. Redispersión

- Tanto en una tabla con recolocación como en el caso de encadenamiento, el factor de carga puede mantenerse en valores pequeños mediante una **redispersión**:
  - Cuando el factor de carga (L=n/N) se acerca (o supera en encadenamiento) el valor **uno**, se dobla el tamaño de la tabla empleada para implementar el rango de valores de dispersión.
- La función de dispersión cambia para doblar su alcance, y todas las entradas que están en la tabla se redistribuyen a sus nuevas posiciones en alguna posición o lista de la tabla mayor.
- La redispersión es **costosa pero infrecuente**, así que no incrementa mucho el tiempo de acceso a la tabla.
- Cada vez que el factor de carga se hace excesivo, se reorganiza la tabla y **el factor de carga vuelve a valer ½**.

7. Resumen de costes resolución colisiones

- Suponemos:
  - una **tabla hash de tamaño N**
  - en la que insertamos **n claves**
- **Resolución de colisiones por encadenamiento**
  - **N>=n**
  - **Tiempo**: en media `O(1)`
  - **Memoria**: `n*(tamClave+tamPunt)+N*(tamPunt)`
  - Incremento de N reduce longitud lista pero incrementa espacio
- **Resolución de colisiones por recolocación**
  - **N>=2n**
  - **Tiempo**: en media `O(1)`
  - **Memoria**: `2n*tamClave`
  - Parece peor, pero si las claves son muy pequeñas puede ocupar menos memoria que el encadenamiento

8. Aplicaciones

- Uno de los usos clásicos de las tablas hash es gestionar la **tabla de símbolos de intérpretes y compiladores**.
  - Los nombres de símbolos (p.ej., nombres de variables) son la clave y los datos de símbolos (p.ej., tipo, ubicación, etc.) están contenidos en el objeto almacenado. La necesidad principal es la búsqueda rápida.
  - Si se requiere una lista ordenada de la tabla de símbolos puede ofrecerse en un segundo análisis.

- Otros usos típicos de tablas hash suelen incluir la **gestión de una clase de datos por una clave arbitraria**, como el número de la seguridad social, el número de cuenta o el número de DNI.

9. Ejemplo

Se trata de buscar una estructura de datos adecuada para implementar **polinomios "dispersos"**, es decir, polinomios en los que la mayoría de los coeficientes son cero, por ejemplo: $x^{10000}+24$

- En concreto, trabajaremos con polinomios de cualquier grado, con coeficientes reales y que tengan como máximo 1000 coeficientes distintos de cero (**N=1000**).
- Asumimos que **p** es un polinomio, **a** es un real distinto de 0, **n** un entero no negativo, y **b** es un real.
- **Operaciones que se quieren realizar:**
  - `añadir(p,a,n)` = $p+ax^n$ (suma a p el monomio ax^n)
  - `consultar(p,n)` = b (donde b es el coeficiente de p de grado n)
  - Suma y producto de polinomios

  - **El objetivo es que las funciones de añadir y consultar sean lo más rápidas posibles.**

9. Ejemplo

Una primera opción sería usar listas enlazadas (se deja como ejercicio para quien quiera practicar), pero en este caso el coste de las operaciones añadir y consultar sería lineal.

Para mejorar este coste, se usará una **tabla hash** en la que guardaremos los coeficientes no nulos de un polinomio, junto con el exponente o grado correspondiente.

9. Ejemplo

Usando una tabla hash con encadenamiento para la resolución de colisiones:

```
/*N=primo mayor que número máximo de coeficientes no nulos (1000) */
    #define N 1039
    /* El tipo monomio corresponde a un elemento con coeficiente no
       nulo del polinomio*/
    typedef struct{
        float coeficiente;
        unsigned int exponente;
        ptMonomio sig;
    }monomio;
    typedef struct *monomio ptMonomio;

    /* El tipo polinomio es una tabla hash de monomios, con
       encadenamiento: lista de punteros a monomios */
    typedef ptMonomio polinomio[N];
```

9. Ejemplo

La clave de un monomio es el **exponente.**

- al ser valores enteros, elegimos como **función hash** el módulo (% en lenguaje C) con el tamaño de la tabla N:

```
int h(int exp){
/* esta es la función hash */
   return exp%N;
}
```

- Necesitamos una operación que cree el polinomio vacío (todos los punteros en la tabla apuntando a NULL)

```
void creaVacio(polinomio *p){
   for (int i=0; i<N; i++)
      (*p)[i]=NULL;
}
```

9. Ejemplo

Función de **añadir** monomios (sumárselos) a un polinomio existente:

```
void anhadir(polinomio *p, ptMonomio m) {
/*Pre: p=p0, m!=0 - Post: p=anhadir(p,m->coeficiente, m->exponente)*/
   ptMonomio q;
   /* Calculamos hash(exponente) para obtener posición k */
   int k=h(m->exponente);
   /* m debe ser añadido a la lista que está en p(k) */
    if ((*p)[k]==NULL){
        (*p)[k]=(ptMonomio)malloc(sizeof(struct monomio));
        (*p)[k]->exponente=m->exponente; (*p)[k]->coeficiente=m->coeficiente;
        (*p)[k]->sig=NULL;
       }
    else {
        q=(*p)[k];
        while (q->sig!=NULL && q->exponente!=m->exponente) q=q->sig;
        if (q->exponente==m->exponente) q->coeficiente+=m->coeficiente;
       /* el coeficiente puede quedar 0, se puede modificar para evitarlo */
       else {
          q->sig=(ptMonomio)malloc(sizeof(struct monomio)); q=q->sig;
          q->exponente=m->exponente; q->coeficiente=m->coeficiente;
          q->sig=NULL;
        }
    }
}
```

9. Ejemplo

Función de **consultar**: si la búsqueda no tiene éxito se devuelve 0

```
float consultar(polinomio p, int n) {
   /*Devuelve el coeficiente de grado n de p*/
      /* Calculamos hash(exponente) para obtener posición k */
   int k=h(n);
   ptMonomio q=p[k];
   /* busco el coeficiente de grado n en la lista p(k) */
   while (q!=NULL && q->exponente!=n)
     q=q->sig;
   if (q!=NULL)
      return q->coeficiente;
   else
      return 0.0;
}
```

Coste:

- El coste de las operaciones de **añadir** y **consultar** en esta implementación depende de la longitud de las listas que guardamos en la tabla hash, es decir, de la cantidad de monomios no nulos del polinomio que la función hash envía a la misma posición de la tabla.

- Si hacemos la media entre todos los posibles polinomios, la longitud de estas listas es una constante pequeña. Esto quiere decir que para la mayoría de los polinomios esta representación hará que las operaciones añadir y consultar tarden un tiempo constante.