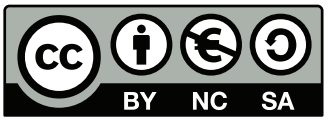
Estructuras de Datos

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM



Estructuras de Datos

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM

Índice

1 Tema 1 - Función de eficiencia.

El proceso que vamos a realiar es extraer la función que marca la eficiencia del código de nuestro programa. Existe una funcion “timer” que me permite ver el tiempo que ha transcurrido en una seccion del codigo que desee.

1.0.1 Notacion O grande.

Se dice que una f(n) = O(g(n)) si a partir de un m, f(n)<= k ůg(n), donde k es una constante. Para usar este tipo de comparación podemos utilizar simplificacion de funciones a su estructura básica.

1.0.2 Hallar la función de eficiencia.

Lo primero es dividir el codigo en trozos, y tenemos en cuenta que si:

1. Los troxos son independientes, la función de la union es O del máximo de las funciones de cada uno.
2. Los trozos son anidados o dependientes, la función de la unión es O del producto de las funciones de cada uno.

Teniendo en cuenta que cada accion cuenta como la unidad, es decir:

```
1 for(int i = 0; i < n; i++){
2   cout << "Lmao";
3 }
```

Serian n iteraciones de valor 1. Sabiendo que la iteracion i++ tambien es una operación pero no tiene sentido tenerlo en cuenta ya que 2n = O(n), de igual forma que la asignacióndel principio del bucle, y la comparación que se realiza, es decir, contaremos las operaciones simples como una única.

1.1 Análisis de sentencias

1.1.1 Bucles

1.1.2 Sentencias IF-ELSE

Es una sencencia en la que usamos la regla del máximo, ocurrirá el if o el else y el máximo de los 2 marca la eficiencia del bloque.

Puede ser que la condición no sea una sentencia elemental, entonces habría que tenerla en cuenta, pero si es una sentencia trivial, la sentencia será O(1)

```
1  if(A[0][0]== 0)
2  {
3    for(i = 0; i < n; i++)
4    {
5      for(j = 0; j < n; j++)
6      A[i][j] = 0;
7    }
8  }
9  else
10 {
11   for(k = 0; k < n; k++)
12   {
13     A[k][k] = 1;
14   }
15 }
```

En este caso, tenemos que tomar máximo entre $O(n^2)$ (lo que cuesta el if) y O(n)(lo que cuesta el else). Por ello, la eficiencia del código es $O(n^2)$.

1.1.3 Bloques de Sentencias

En este caso, si tenemos bloques independientes se va tomando la regla del máximo para todos los bloques

1.1.4 Llamadas a funciones

Hay que mirar cuánto cuesta la llamada a la función, es muy importante para la ejecución del código.

Analizamos un ejemplo de un código de 30 líneas de las transparencias del profesor.

1. Ejemplos
   Cuando en los bucles haya i =2, i*=4,i*=n... entonces la eficiencia será logaritmo en base n de lo que haya dentro del bucle.

2 Tema 2 - Abstracción.

2.0.1 Uso de Template.

Nos permite seleccionar el tipo de dato que vamos a utilizar en tiempo de ejecución. Declarando:

```
template <class T, int n>

class array_n {
private:
    T items[n];
  };
```

Creando un objeto de esta clase de la forma:

```
array_n<int,1000> w
```

Creando un metodo de la forma:

```
template <class T>
T VectorDinamico::componente(int i) const
{
  return datos[i];
}
```

La compilación a la hora de usar templates es distinta a la que estamos acostumbrados. En lugar de hacer un *#include* "/clase.h/'' en el archivo **.cpp**, se incluirá el archivo *#include* "/clase.cpp/'' al final del archivo **.h**.

2.0.2 Clase vector dinámico

Vamos a estudiar una clase que tenga un vector de datos y el número de elementos.Un primer ejemplo con tipo de dato float sería:

```
class VectorDinamico{

  float* datos;
  int ns;
}
```

2.0.3 Iteradores

Pretendemos hacer recorridos mucho más rápido. No volvermos a recorrer vectores haciendo v[i]. Usaremos los punteros para iterar, de la forma:

```
double *v = &a.
double *p;
double * fin;
fin = v+n;
for(p = v; p!= fin; ++p)
  cout << *p << endl;
```

Nota: en un compilador moderno, simplemente activando la optimización de código se consigue el mismo aumento en el rendimiento.

Definiremos incluso un nuevo tipo de dato llamado iterator, haciéndolo de la forma:

```
1  typedef double* iterator;
2  iterator begin(double* v, int n){
3    return v;
4  }
5  iterator end(double* v, int n){
6    return v+n;
7  }
8
9  /**--------------------------*/
10 iterator p;
11 for(p=begin(v,n); p!=end(v,n);++p)
12   cout << *p << endl;
```

2.0.4 Pilas

Son estructuras de datos lineales, secuencias de elementos dispuestos en una dimensión. Tienen estructura *LIFO* (last in, first out)

No usaremos el concepto de posición, sólo trabajaremos con el tope de la estructura. En realidad, sí podemos recorrerla pero debemos salvaguardar los elementos , pues para acceder al siguiente elemento tenemos que borrar el anterior.

- Para insertar un elemento, se inserta siempre al principio de la pila, añadiendo físicamente
- Para borrar un elemento, se elimina y el puntero que había al primero se lleva al segundo
- Como la pila no tiene recorridos, no tienen iteradores.
- Las funciones básicas son Tope, Poner, Quitar, Vaciar.

Hay que recordar que los elementos de la pila se guardan en orden contrario al que fueron insertados. Por tanto si los imprimimos, por ser de tipo *LIFO* salen del primero que hemos insertado al último. En la práctica, usaremos las pilas con Celdas Enlazadas.

```
1  #ifndef __PILA_H__
2  #define __PILA_H__
3
4  typedef char Tbase;
5
```

```
 6 struct CeldaPila{
 7     Tbase elemento;
 8     CeldaPila *sig;
 9 };
10
11 class Pila{
12
13     CeldaPila *primera;
14
15   public:
16
17     Pila();
18     Pila(const Pila& p);
19     ~Pila();
20     Pila& operator= (const Pila& pila);
21
22     void poner(Tbase c);
23     void quitar();
24     Tbase tope() const;
25     bool vacia() const;
26 };
```

2.0.5 Colas

Una cola es una estructura de datos lineal en la que lso elementos se suprimen e insertan por extremos opuestos. Son estructuras *FIFO*. En una cola, si tenemos dos elementos

Usaremos el operador %, que transforma una estructura lineal en una linear circular. Al igual que en las colas, no tengo acceso a los elementos si no destruimos la cola.

3 Resumen general

Este es un resumen de los contenidos de la asignatura Estructuras de Datos. Realizado con el fin de recopilar los conceptos más importantes de la asignatura.

3.1 Eficiencia

Es importante a la hora de evaluar algoritmos conocer su eficiencia, es decir, el tiempo que tardan en completar el algoritmo en tanto a las instrucciones

necesitadas.

La eficiencia la podemos calcular de manera *empírica*, realizando mediciones durante la ejecución del algoritmo. Como *teórica*, basándonos en el número de instrucciones que necesitaría nuestro algoritmo. Nos centraremos en este estudio.

3.1.1 Órdenes de eficiencia

Decimos que un algoritmo tiene un tiempo de ejecución de orden **T(n)**, y estos pueden ser:

-**n**: Lineal
-n^2: Cuadrático
-n^k: Polinómico
-**log(n)**: Logarítmico
-c^n: Exponencial

3.1.2 Notación O(n)

Para simplificar los cálculos de la eficiencia teórica usamos la notación "*O grande*", que usamos para comparar algoritmos.

Esta notación consiste en que, dada una función f(n) que nos determine el tiempo de ejecución de un programa mediante una regla simple. Eliminamos los términos de orden menor y las constantes. Así dada una función $f(n) = n^2+3n+5$ sería $O(n^2)$.

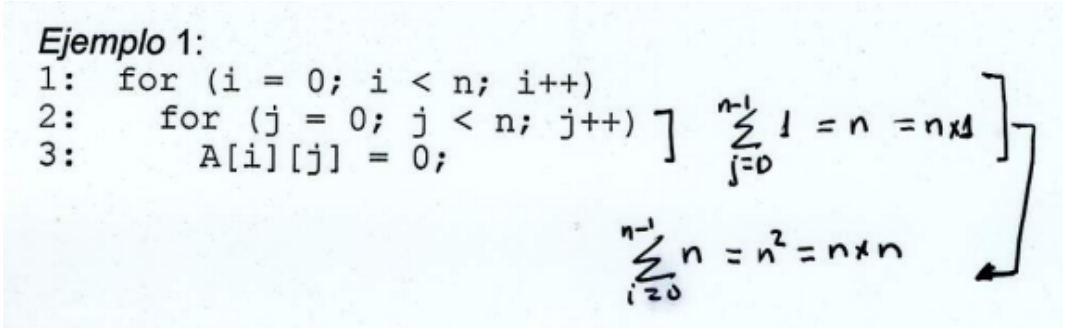
Reglas de O(n)

1. La suma de dos tiempos es O(máx(T_1, T_2))
2. El producto de dos tiempos es O($T_1 \cdot T_2$)

3.1.3 Operaciones

1. **Elementales**: (asignación, lectura, esscritura...)
2. **Bucles**:

El tiempo invertido en el bucle es la suma del tiempo invertido en cada iteración.



3. **Sentencias if/else**:

Cada flujo tiene un tiempo determinado y nos quedamos con el máximo de ellos.

4. **Bloques de sentencias**:

Se aplica la regla de la suma, nos quedamos con el máximo de las sentencias.

5. **Llamadas a funciones**:

Cuesta el O(n) de la propia función a llamar.

Casos especiales:

En un bucle, cuando no recorremos desde la primera iteración 0, hasta la última n-1 hemos de calcular el número de operaciones como número de operaciones en la primera iteración, más numero de operaciones en la última, todo ello dividido por 2 y multiplicado por el número de iteraciones totales.

Ejemplo:

```
1     int n,j; int i=2; int x=0;
2         do {
3             j=1;
4             while(j<=i) {
5                 j=j*2;
6                 x++;
7             }
8             i++;
9         } while(i<=n);
```

Con operaciones elementales en la primera línea, un bucle do-while que a su vez tiene un bucle while anidado, este bucle anidado no se recorre desde j=0 hasta n, sino desde j=0 hasta i. La cantidad de iteraciones a realizar depende del número de iteraciones que haya hecho el bucle en el cual está contenido.

Procedemos entonces a calcular su eficiencia teórica, para las sumatorias usamos el mismo procedimiento: **Número de operaciones en la primera iteración, más numero de operaciones en la última, todo ello dividido por 2 y multiplicado por el número de iteraciones totales.**

$$1+\sum_{i=2}^{i=n}\Big(\sum_{j=0}^{j=log_2(i)} 2\Big) \Rightarrow 1+\sum_{i=2}^{i=n}\frac{2+2}{2}\cdot log_2(i) \Rightarrow 1+\Big(\frac{2\cdot log_2(2)+2\cdot log_2(n)}{2}\Big)\cdot(n-2) \Rightarrow$$

$$\Rightarrow 1+2\cdot log_2(2+n)\cdot(n+2)$$

La notacion O de este código es por tanto **O(n·log$_2$n)**

3.2 Abstracción

Podemos esbozar una idea de lo que es la abstracción considerando esta como la formación de una imagen del objeto a abstraer. Esta imagen la construiremos quedándonos con los detalles y propiedades que posea el objeto, ignorando lo superficial con el fin de mejorar la comprensión de este.

A la hora de programar existen diversos tipos de abstracción. Tanto para un procedimiento, representar una estructura de datos abstracta, como para realizar la iteración sobre un objeto sin importarnos la implementación u organización del mismo.

3.2.1 Conceptos sobre la abstracción

A la hora de representar un TDA hay algunos elementos que merecen ser diferenciados del resto:

Tipo Rep: Es el tipo que representa al dato en sí. En un TDA fecha sería el struct que almacena la información de día, mes, año... Mientras que en otro TDA racional sería un struct con un entero numerador y otro denominador.

Invariante de Representación (IR): Condición que tiene que ser cumplida por todos los datos. Es el caso de, en el TDA número racional, que el denominador siempre tiene que ser distinto de cero, y esto lo cumplen todos los números racionales.

Pero también hay otros conceptos relativos a la implementación de una clase:

Funciones inline: Son aquellas que, por eficiencia, se definen "inline", es decir, donde se produzca una llamada a esa función en el código, en lugar de llamar a la función, "*se pegará*" ese trozo de código allí. Pueden estar definidas dentro de la clase o fuera de ella.

Dentro de la clase:

```
1 class Fecha { \newline
2 int getDia() const {return dia;} \newline
3 };
```

(Recordamos que ese const implica que el objeto no será modificado y que lo usamos para obtener información de este)

Fuera de la clase:

```
1 class Fecha { \newline
2 int getDia() const; \newline
3 };
4
5 inline int Fecha::Dia() const {
6     return dia;
7 }
```

Funciones friend: Cuando necesitamos acceder a algún dato miembro de una clase para operar con él desde otra haremos uso de un mecanismo para saltarnos el control de acceso que habitualmente nos impide esto. Las funciones friend. Se declara en la definición de la clase, precedida de la palabra reservada friend.

```
1 class Fecha { \newline
2     friend ostream & operator<< (ostream &s, const Fecha & f);
3 };
```

Templates: Como ya veremos más adelante, permiten a una clase trabajar sin un tipo de dato predefinido. Su definición se hace del siguiente modo:

```
1 template<class T, int n >
2 class array_n {
3 private:
4     T items[n];
5
6 /* Sería usado como */
7
8 array_n<racional,100> array_racional;
```

Y fuera de la clase se implementarían así:

```
1 template <class T>
2 T array_n::componente(int i) const {
3     return items[i];
4 }
```

3.3 Tipo de Datos Abstracto

Los **TDA** como ya hemos explicado anteriormente, cumplen la función de la abstracción a la hora de representar algún objeto. Veremos los principales TDAs lineales.

3.3.1 Pilas

Se conocen como estructuras LIFO (last in first out) donde cada elemento se introduce a continuación del último insertado, y sólo podemos acceder (o insertar/eliminar) al último miembro que colocamos.

La implementación se puede hacer sencillamente con un vector estático y un índice tope (con los problemas de tamaño que supone) o bien mediante celdas enlazadas que contuviesen el dato en cuestión y un puntero al siguiente elemento.

Ambos casos poseerían las mismas funciones de insertar, borrar, consultar tope, vaciar o comprobar si está vacía...

Pero se implementan de distinta manera, pues en uno se utiliza el número de elementos para insertar para posteriormente aumentarlo, mientras que el otro

reserva memoria para una nueva celda y cambia la dirección a la que apunta el anterior a la de la nueva celda y el puntero de esta a NULL.

3.3.2 Colas

En estas las inserciones y eliminaciones se realizan en extremos opuestos, siguiendo una estructura FIFO (first in first out) donde cada elemento insertado será extraído de esta antes que los posteriores.

Estas también pueden ser implementadas mediante celdas enlazadas (definimos un struct Celda y luego dos punteros a la primera y a la última celda), o en su defecto, por vectores dinámicos (con un puntero a los datos, el número de elementos presentes y reservados y la posición dentro del vector del primero y del último).

Colas con prioridad: Las colas con prioridad son un tipo de dato que contiene una secuencia de valores. Esta es diseñada de manera que se pueda insertar en cualquier lugar en tanto a un valor de prioridad dada, mientras que el acceso o el borrado se produce en un extremo llamado frente.

3.3.3 Listas

Son otro tipo de estructura de datos mucho más generales que estas anteriores. Se caracterizan porque sus elementos son accesibles en todas las posiciones para cualquier tipo de operación.

Su implementación, al igual que pasaba con las colas, puede ser con vectores dinámicos o con celdas enlazadas.

Para facilitar el acceso existen otras variantes de las listas tales como listas circulares (el siguiente al último elemento es el primero), listas con cabecera o listas doblemente enlazadas (un puntero al elemento siguiente y otro al anterior).

3.4 Plantillas e iteradores

3.4.1 Abstracción por parametrización

El concepto de abstracción por parametrización que comentábamos en un principio se ve claramente reflejado en el uso de una plantilla o "*template*", donde hacemos posible que una función trabaje con un tipo de dato cualquiera no definido por defecto.

Uno de los factores más importantes a la hora de implementar las plantillas es cómo tenemos que escribirlas dentro o fuera de la clase (en nuestro caso en el archivo cabecera *.hpp* o en el fuente *.cpp*),

```
1 template<class T, int n >
2 class array_n {
3 private:
4     T items[n];
5
6 /* Sería usado como */
7
8 array_n<racional,100> array_racional;
```

Y fuera de la clase se implementarían así:

```
1 template <class T>
2 T array_n::componente(int i) const {
3     return items[i];
4 }
```

3.4.2 Abstracción por iteración

Pese a esto nos pueden surgir diversos problemas a la hora de acceder a cada uno de los elementos que componen un contenedor. Es aquí donde retomamos el concepto de **abstracción por iteración** y donde surgen por tanto los iteradores, un nuevo tipo de dato abstracto que implementa la idea de indexación de forma similar a los punteros.

Para definir un nuevo tipo de dato que realice la iteración sobre un contenedor podríamos definirlo del siguiente modo:

```
1 typedef double * iterator;
```

Esto nos permitiría referirnos a este iterador para definir funciones que nos devuelvan este tipo. Pero seguiremos teniendo problemas cuando queramos hacer operaciones con parámetros constantes. Por ejemplo cuando queramos escribir un vector no modificaremos su contenido y por tanto lo pasaremos como `const double* v`, esto producirá un error a la hora de desreferenciar un elemento del vector pues el iterador no asegura que el vector no se modifique.

Para solucionar esto hacemos uso de los iteradores constantes (los elementos sobre los que iteramos no serán modificados). Estos se implementan de manera muy similar, añadiendo un const en su definición:

```
1 typedef const double * iterator;
```

Si estamos trabajando con algún tipo de la *STL* podremos aprovecharnos de los iteradores que ya están implementados. Por ejemplo, para el tipo set lo haremos de esta manera:

```
1   std::set<int>::const_iterator c_it;
2
3   std::set<int>::iterator it;
```

Cuando combinamos la abstracción por parametrización con la de iteración hemos de tener especial cuidado en la implementación, pues aquí ya no tendremos un tipo definido al que referirnos "con un sinónimo" como hacíamos con `typedef`, sino que tenemos que hacerlo del siguiente modo:

```
1 template <class T>
2
3 void escribir(const T &contenedor) {
4     typename T::const_iterator p;
5
6     for(p.begin(); p != p.end(); ++p) {
7         cout << *p << endl;
8     }
9 }
```

3.5 Standard Template Library

Una de las librerías de contenedores más importantes es la STL, que implementa de manera eficiente estos contenedores aportándole una versatilidad enorme, permitiendo formar una infinidad de tipos de datos abstractos a partir de estos.

A continuación introduciremos de manera breve algunos de los tipos más importantes presentes en la STL. Su implementación y el conjunto de funciones con el que podemos manejarlos lo podemos consultar en cualquier referencia externa.

3.5.1 Contenedores directos

Garantizan el acceso a cualquier componente a tiempo constante:

Vector dinámico El contenedor **vector**, que implementa un vector dinámico sin tener que preocuparnos de reservar espacio.

3.5.2 Contenedores asociativos

Gestionan los datos mediante claves que les identifican y permiten su recuperación eficiente mediante esta clave.

Conjunto (set) El tipo **set** abstrae la idea de conjunto matemático, permite recoger una colección de valores que no se repiten. Además podemos saber de manera eficiente si un valor está contenido o no en el conjunto.

Bolsa (multiset) Es análogo al conjunto, pero con la diferencia de que **multiset** permite almacenar valores repetidos.

Diccionario (map) El tipo **map** es una estructura que almacena implementando una relación clave-valor, cada clave es única y nos permite recuperar su valor correspondiente mediante el uso de esta clave.

3.5.3 Contenedores secuenciales

Son aquellos que se organizan de manera lineal (primer elemento, segundo, ...) y que quedan identificados por la posición. Entre estos se encuentran el ya mencionado **vector**.

Lista (list) Contenedor que almacena sus elementos en forma de sucesión, permitiendo por tanto el acceso secuencial a este. Cada elemento establece quién es el siguiente de la sucesión.

Pila (stack) Representada como una sucesión de objetos que limita la forma en la que se acceden a estos, así como introducir y borrar elementos, que sólo puede hacerse por un extremo de la secuencia llamada tope.

Cola (queue) Representado como una sucesión de objetos que limita la forma en la que se añaden/eliminan elementos de esta. Así pues sólo podemos introducir elementos por el extremo *final* de la sucesión y recuperar/borrar del extremo opuesto *frente*.

3.6 Árboles

Un árbol es un tipo de estructura consiste en un conjunto de nodos separados por niveles donde cada nodo mantiene una relación con un nodo del nivel inmediatamente superior (hasta llegar a un nodo tope considerado raíz).

Los árboles están fundamentados en la recurrencia, es decir, cada hijo que posea un nodo puede constituir un nuevo subárbol cuya raíz sea ese hijo.

Los árboles para almacenar una información hacen que a cada nodo se le asocia una etiqueta, que puede ser cualquier tipo de dato.

Grado de un nodo y de un árbol: Es el número de subárboles que tiene un nodo. Si el nodo es la raíz hablaremos del grado de un árbol.

Un camino entre dos nodos es la secuencia de nodos que se atraviesan para llegar a un nodo a partir de otro. La longitud se mide como el número de nodos que conforman ese camino menos uno, incluyendo los nodos de salida y llegada (*el número de movimientos necesarios*).

La **altura** de un nodo es el número de niveles que cuelgan por delante de este.

```
1                                o  <-- altura = 2
2                               / \
3               0 = altura --> o   o <-- altura = 1
4                                   \
5                                    o <-- altura = 0
```

3.6.1 Árbol binario

Es un árbol cuyos nodos sólo pueden tener a lo sumo un hijo a la izquierda y otro a la derecha. Donde ser hijo izquierda o hijo derecha es relevante, pues es esto lo que los distingue de un árbol n-ario de grado 2.

Homogéneos: Si todos sus nodos tienen grado 0 o 2 (“no hay ningún hijo suelto”).

Completos: Todos los niveles tienen dos hijos o en su defecto el único nivel con un sólo hijo se encuentra en el último nivel y sería el hijo de la derecha.

Su representación suele hacerse mediante el uso de nodos, definiendo en la parte privada de la clase por un lado el struct nodo que tiene una serie de punteros al padre, hijo izquierda e hijo derecha y la etiqueta de este, y por otro tiene un puntero al nodo raíz.

3.6.2 Recorridos

Los árboles pueden ser recorridos en profundidad de las siguientes maneras recursivas:

Preorden: Visitamos en primer lugar la raíz, luego recorremos el preorden del hijo izquierda y posteriormente el del siguiente hijo a la derecha.

Inorden: Visitamos primero el inorden del hijo izquierda, luego la raíz y luego procedemos con los inorden de los sucesivos hermanos derecha(si los hubieran).

Postorden: Visitamos primero el postorden del hijo izquierda, luego el postorden de cada uno de los hermanos derecha para por último visitar la raíz.

Pero también pueden recorrerse en anchura:

Por niveles: Se recorre de arriba a abajo y de izquierda a derecha empezando por la raíz.

Por lo general, **un árbol no puede ser definido por un único recorrido**.

3.6.3 Árboles especiales

Existen algunos árboles que, por motivos de eficiencia en su tratamiento, se han diseñado con una estructura característica. A continuación veremos algunos ejemplos de ello.

Árboles Binarios de Búsqueda (ABB) En este árbol todos sus nodos cumplen una propiedad, todos los elementos almacenados en el subárbol izquierdo de un nodo son menores o iguales que este nodo, y todos los elementos de la derecha son mayores e iguales que este. Por lo general la desigualdad es estricta pues no hay claves repetidas para los nodos.

Su tiempo de búsqueda es muy eficiente, del orden $O(log_2(n)$. Sin embargo, las inserciones y borrados son muy ineficientes $O(n)$.

Árboles Binarios Parcialmente Ordenados (APO) Este arbol cumple la condición de que la etiqueta de cada nodo es menor o igual que las etiquetas de sus hijos, manteniéndose tan balanceado como sea posible.

Es especialmente útil en los algoritmos de ordenación. Se usa un vector dinámico para su representación donde el elemento en la posición 0 sería la raíz, y el hijo izquierdo de un nodo M[k] si existe estará en la posición M[2k+1] mientras que el hijo derecho estaría en el M[2k+2].

Árboles Equilibrados (AVL) Los árboles AVL (Adison-Velskii Landis) son una versión optimizada de los ABB para poder mejorar las búsquedas en estos. Consiste en que, siendo árboles ABB tienen la propiedad además de que para ningún nodo las alturas de sus subárboles izquierdo y derecho difieran en más de una unidad.

Sus operaciones de inserción y borrado tienen un orden de eficiencia logarítmico.

3.7 Tabla Hash

La tabla hash es una estructura que no opera por comparaciones entre valores clave, sino que para obtener la localización de una clave k usa directamente una función aplicada sobre la clave que nos proporciona la localización exacta donde se encuentra almacenada en la estructura sobre la que estemos trabajando.

La principal complicación de estas funciones es encontrar funciones que cumplan que:

$$f(i) = f(j) \Leftrightarrow i = j$$

Por tanto hay que encontrar funciones hash que generen el menor número posible de colisiones y además diseñar métodos de resolución de colisiones para cuando estas se produzcan.

3.7.1 Funciones hash comunes

Entre los tipos más comunes de funciones hash se encuentran:

1. Truncamiento: Consiste en eliminar dígitos de la clave. Este tiene el problema de que las tablas deben de ser de un tamaño potencia de 10 y puede haber muchas colisiones. Existe una alternativa con el truncamiento a nivel de bit, cuyas tablas podrían adquirir un tamaño potencia de 2.

2. Plegado: Consiste en dividir una clave en dos o más partes y sumarlas. Tiene el mismo inconveniente del tamaño que la función hash anterior.

3. Multiplicación: Similar al plegado, pero en lugar de sumar, se multiplica. Puede combinarse con un truncamiento antes o después del producto para disminuir la probabilidad de las colisiones. Tiene dos variantes, el cuadrado del centro o el centro del cuadrado, ambas relacionadas con la selección de varias cifras y su cuadrado (y viceversa).

4. División - Resto: Consiste en tomar el resto de la división de la clave entre el tamaño de la tabla:

$$h(k) = kmod(M)$$

Es un método simple que no requiere truncamiento y que permite que las tablas sean de tamaño arbitrario. El tamaño de la tabla debe de ser por lo menos igual que el número de claves y la mejor elección es tomar M como un número primo mayor que el número de claves.

3.7.2 Tratamiento de colisiones

Prácticamente todos los tipos de funciones hash producen colisiones, por lo que es igual de importante encontrar un mecanismo para la ubicación de la clave que provocó la colisión de tal modo que después, en una operación de consulta, la búsqueda se realice eficientemente.

Atendiendo a la estructura de datos elegida podemos usar:

Hashing abierto Para estructuras de datos dinámicas (listas). Es usado cuando no conocemos el número de elementos que vamos a ubicar en la tabla hash.

Hay que construir para cada índice de la tabla una lista de claves sinónimas. Para ello usamos una lista dinámica, implementando la tabla como un vector estático de punteros a estas listas o bien como una lista dinámica de punteros.

Se fija el tamaño a priori.

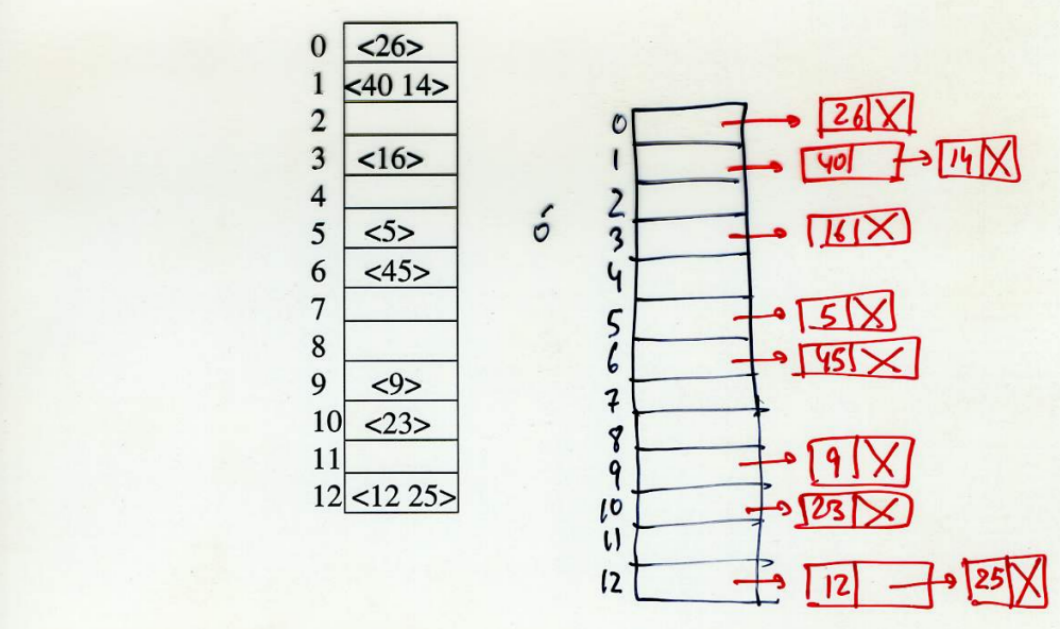
La búsqueda se reduce a recorrer la lista oportuna.

La inserción se utiliza siguiendo algún criterio de orden (LIFO, FIFO...).

Tiene la ventaja de que la tabla puede tener un tamaño inferior al número de registros dada su naturaleza dinámica en la asignación. Pero por otro lado la desventaja que requiere un espacio adicional ocupado por los punteros que se necesitan para mantener las estructuras de tipo lista.

Ejemplo:

El conjunto de números {23,45,16,26,40,14,5,12,9,25} con $f(x) = x$ % 13:



Hashing cerrado Usado en estructuras estáticas (vectores). Cuando sabemos exactamente cuantos elementos se ubicarán en la tabla hash.

Utilizamos un vector como representación, y cuando se producza alguna colisión la resolveremos asignándole otro valor hash a la clave hasta que encontremos un hueco.

Para esta reasignación de casillas tenemos varias posibilidades:

1. **Prueba lineal:** Tal que $h_i(x) = (h(x) + i)$ % M donde i es el número de intento. Produce agrupamientos primarios.
2. **Prueba cuadrática:** $h_i(x) = h(x) + i^2$ donde i es el número de intento. Produce agrupamientos dobles.
3. **Hashing doble:** Usamos una segunda función hash $h'(x)$ con lo que tendríamos $h_i(x) = (h(x) + i \cdot h'(x))$ % M. Esta función $h'(x) nunca podrá

tomar el valor 0.

La búsqueda se hacen siguiendo la misma secuencia que usó la función hash para la inserción.

Una vez que la tabla se llena se debe seleccionar un tamaño de tabla mayor que también sea primo y volver a insertar todas las claves (en sitios distintos, pues el tamaño cambia y por tanto la función hash). Este proceso se llama ***rehashing***.