Lección 14: Funciones de dispersión. Dispersión abierta

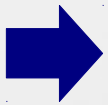
- Motivación
- Tablas de dispersión
- Funciones de dispersión
- Dispersión abierta
- Redispersión

Motivación

Una tabla de símbolos es una estructura de datos usada por compiladores e interpretes donde cada identificador es asociado con información relacionada con su tipo, ámbito y localización

```
int main() {
    int arr[100];
    int nElem = 0;

    arr[0] = 77;
    arr[1] = 99;
    nElem = 2;
    for (int i=0; i<nElem; i++)
        cout << arr[j] << endl;
}
```



nombre	tipo	...
arr	var	...
nElem	var	...
main	fun	...
i	var	...

Motivación

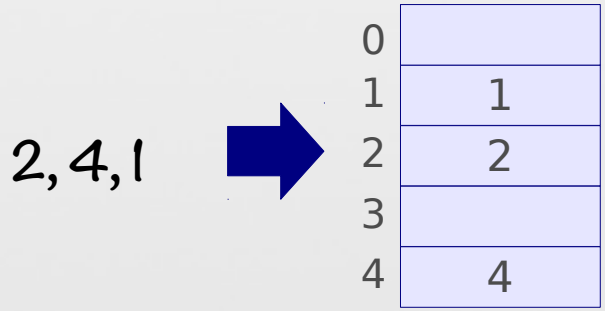
Durante la compilación en el caso de los compiladores, o ejecución en el de los intérpretes, cada vez que aparece una referencia a una variable o función ésta es buscada en la tabla de símbolos. Si no existe la entrada, se añade o se genera un error.

Esta tabla puede ser consultada miles de veces por segundo y su acceso eficiente es vital para una rápida compilación/ejecución.

Un árbol AVL proporciona una solución O(log n) a este problema ¿Puede conseguirse acceso en tiempo constante a esta tabla?

Tablas de dispersión

- Una **tabla de dispersión** es una estructura de datos que usa como soporte un vector de un tamaño predeterminado, y donde el valor de la clave de un dato es usado para calcular el índice de la posición en el vector donde este dato es guardado
- Implementan la funcionalidad de un contenedor asociativo

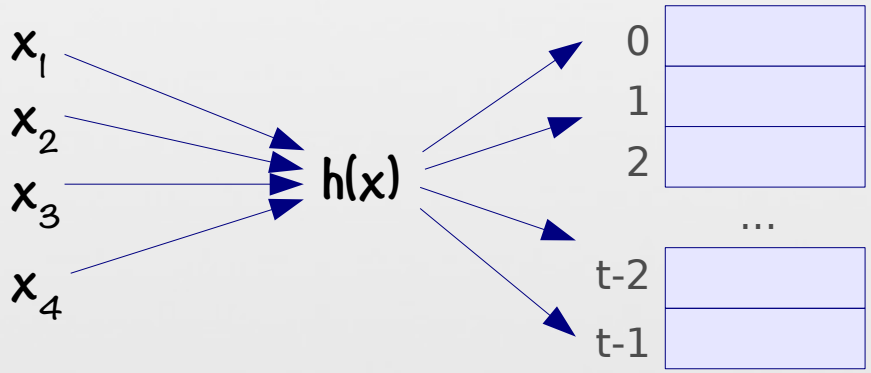


Tablas de dispersión

- Operaciones:
  - Inserción: tabla[clave] = dato
  - Búsqueda: ¿tabla[clave] ≠ nulo?
  - Borrado: tabla[clave] = nulo
- Problema 1: la clave puede no ser entera (p. e. alfanumérica)
- Problema 2: el rango de la clave puede ser muy grande (ej. 0 a 100000000) → la tabla de dispersión sería enorme!

Funciones de dispersión

- Una **función de dispersión** h(x) es una función que mapea cada valor de la clave con una posición de la tabla
- El rango de las claves es normalmente mucho mayor que el rango de posiciones de la tabla



Funciones de dispersión

- Una buena función de dispersión debe:
  - Tener un bajo coste de cálculo
  - Ser determinista → devolver la misma posición siempre para la misma clave
  - Ser uniforme → generar posiciones en todo el rango de posiciones de la tabla
- Es imposible impedir las **colisiones**: $h(x_1)=h(x_2)$ siendo $x_1 \neq x_2$

Ejemplos de pésimas funciones de dispersión:

```
h(x) = rand() % t
h(x) = 0
```

Dispersión por división

- Sencilla y con diferencia la más utilizada:
  - h(x) = x % t
- Funciona mejor si *t* es primo. En caso contrario puede utilizarse:
  - h(x) = (x % p) % t, con p primo tal que p > t

Para x = 4567 y t = 101:
h(x) = 22

Para x = 4567, p = 173 y t = 101
h(x) = (4567 % 173) % 101 = 69

Dispersión por plegamiento

- Buena para claves grandes:
  - Dividir la clave en grupos de dígitos
  - Sumar estos grupos de dígitos
  - Aplicar módulo si es necesario para ajustar al tamaño t

Para x = 23911984 y t = 100:

x= 23|91|19|84 → h(x)=(23 + 91 + 19 + 84) % 100 = 17

Dispersión por mitad del cuadrado

- Buena para números reales:
  - Elevar x al cuadrado y extraer los dígitos centrales que se necesiten

Para x = 4567 y t = 100:

$x^2 = 208\mathbf{57}489 \rightarrow h(x) = 57$

Dispersión para cadenas

- Solución trivial: sumar los caracteres de la cadena
- Muy mala. P. ej. “CASA” y “SACA” generan la misma posición

```
unsigned long dispersion(unsigned char *str) {
    unsigned int hash = 0;
    int c;

    while (c = *str++)
        hash += c;

    return hash;
}
```

Dispersión para cadenas

- Existen soluciones mucho mejores
- Por ejemplo el algoritmo djb2 de Dan Bernstein:

```
unsigned long djb2(unsigned char *str) {
    unsigned long hash = 5381;
    int c;

    while (c = *str++) hash = ((hash << 5) + hash) + c;
    return hash;
}
```

Para t = 101:

djb2(“CASA”) % t = 6383922269 % 101 = 18

djb2(“SACA”) % t = 6384496733 % 101 = 95

Sobre las funciones de dispersión

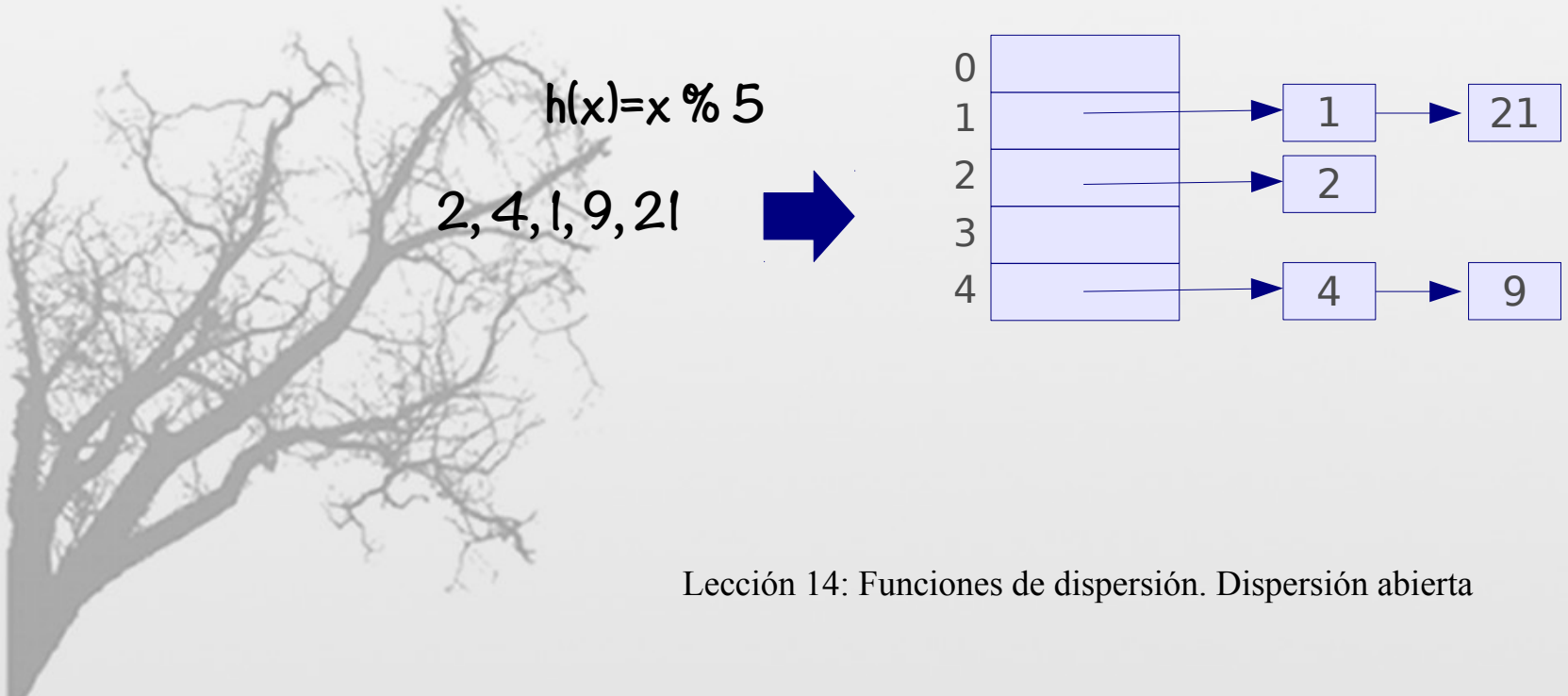
- En general con independencia de la función de dispersión escogida es inevitable que aparezcan colisiones
- También es inevitable que muchas posiciones de la tabla nunca sean utilizadas
- Existen generadores de funciones de dispersión perfectas, que no generan colisiones siempre que se conozcan los datos a priori (CMPH, gperf)

Sobre el tamaño de la tabla

- Denominamos **factor de carga** de una tabla a $\lambda=n/t$ y mide cómo de ocupada está la tabla
- Experimentalmente se concluye que cuando $0.0<=\lambda<=0.7$ el número de accesos es prácticamente constante
- Si $\lambda>0.7$ el número de colisiones y el número de accesos necesarios para acceder a un dato aumenta considerablemente
- Por tanto en general una tabla debe ser un 1/3 mayor que el número de datos a insertar

Dispersión abierta

- La **dispersión abierta o encadenamiento separado** soluciona el problema de las colisiones manteniendo una lista de entradas por cada posición de la tabla



Dispersión abierta

```
template<class T>
class Entrada {
public:
    long clave;
    T dato;

    Entrada(long aClave, T aDato): clave(aClave), dato(aDato) {}
};

template<class T>
class DispAbierta {
    typedef list<Entrada<T> > ListaEntradas;

    vector<ListaEntradas> tabla;

    // Función de dispersión
    long fDisp(int clave) {
        return clave % tabla.size();
    }

public:
    DispAbierta(int tam): tabla(tam, ListaEntradas()) {}

...
```

Dispersión abierta

```
void insertar(int clave, T &dato) {
    tabla[fDisp(clave)].push_back(Entrada<T>(clave, dato));
}

bool borrar(int clave) {
    typename vector<ListaEntradas>::iterator iv =
        tabla.begin() + fDisp(clave);

    typename ListaEntradas::iterator il = iv->begin();

    while (il != iv->end()) {
        if (il->clave == clave) {
            iv->erase(il);
            return true;
        }
        ++il;
    }

    return false;
}
```

Dispersión abierta

```
    bool buscar(int clave, T &dato) {
        typename vector<ListaEntradas>::iterator iv =
            tabla.begin() + fDisp(clave);

        typename ListaEntradas::iterator il = iv->begin();

        while (il != iv->end()) {
            if (il->clave == clave) {
                dato = il->dato;
                return true;
            }
            ++il;
        }

        return false;
    }
};
```

Redispersión

- En general la dispersión requiere conocer a priori el número máximo de datos que se va a manejar
- No obstante, también es posible optar por **redispersar** los datos en una tabla mayor cuando λ supera 0.7 (o incluso 1 en el caso de la dispersión abierta)
- Una buena estrategia para minimizar el número de redispersiones es elegir un tamaño de tabla igual al doble del anterior

Conclusiones

- La dispersión es una técnica muy sencilla que proporciona la funcionalidad de un contenedor asociativo con un rendimiento óptimo O(1) en todas las operaciones
- Sin embargo es necesario:
  - conocer bien la naturaleza de las claves,
  - encontrar una buena función de dispersión
  - dimensionar correctamente el problema para evitar redispersiones

Conclusiones

- La dispersión es la técnica más utilizada para el mantenimiento de las tablas de símbolos en compiladores/intérpretes
- La dispersión abierta es la implementación más directa para una tabla de dispersión
- La dispersión abierta requiere el uso de una estructura de datos secundaria que puede no ser factible en ciertas aplicaciones (p. ej. en un fichero en disco)
- En la próxima lección estudiaremos estrategias para guardar los datos exclusivamente en la tabla principal