Tablas de dispersión

Hash tables

Concepto de dispersión

- Si las llaves fueran números 0..M-1, un simple arreglo serviría como una tabla de símbolos no ordenada.
- Si las llaves no son 0..M-1? Se pueden definir funciones que conviertan valores en un dominio a números 0..M-1 :

$$h : Dominio \rightarrow \{0..N-1\}$$

estas funciones se llaman funciones de dispersión (hash functions).

Colisiones

- Normalmente es imposible que la funcion *h()* sea inyectiva, *i.e.* se pueden presentar casos en que dos o más elementos del dominio se mapeen en el mismo índice.
- Cuando esto ocurre, se le denomina una colisión. Las implementaciones de la tabla de dispersión deben manejar estos casos para evitar pérdida de información.

Funciones hash

- Si los índices son numéricos, se pueden convertir fácilmente el rango 0..M-1 mediante la función

$$h(x) = x \bmod M$$

- La selección de M es muy importante. La estrategia general es tomar un número primo para que todos los bits de *x* influyan en el valor del resultado.

Hash para Strings y Objects

- Idea general: Tomar el valor numérico asociado a los caracteres como un número base R:

$$h(x) = x_{n-1}R^{n-1} + \ldots + x_0 R^0 = (x_{n-1}R + x_{n-2})R + \ldots + x_0$$

- En principio cualquier tipo de dato se puede llevar a una representación numérica (binaria), *e.g.* Date, Arreglos, en general cualquier ADT.

Java hashCode

- La clase Object provee una implementación de función hash que heredan todas las demás clases

```
public int hashCode()
```

- Debe ser consistente con `equals`:
  - Si a.equals(b) entonces a.hashCode()==b.hashCode()
- Comúnmente las subclases sobre-escriben el método `hashCode()`.

hashCode() to hash value

- El entero de 32 bits no está en el rango 0..M-1 deseado.
- Se resuelve usando el operador módulo así:

```
private int hash(Key key) {
  return (key.hashCode()&0x7fffffff) % m;
}
```

- La máscara de bits tiene por fin eliminar el bit de signo.

Propiedades de la función hash

- Debe ser consistente: El mismo objeto siempre debe retornar el mismo hash.
- Debe ser eficiente de calcular: Idealmente ~1.
- Debe distribuir uniformemente las llaves en el intervalo 0..M-1.

Hashing con encadenamiento

- La tabla de símbolos se estructura como un arreglo de M tablas de símbolos secuenciales.
- El valor hash de la llave indica la entrada del arreglo al que pertenece la llave.
- Llaves con el mismo hash se encadenan en una misma lista.

Implementación

- Variable de instancia y constructores:

```
private SequentialSearchST<Key, Value>[] st;


public SeparateChainingHashST() {
    this(INIT_CAPACITY);
}


public SeparateChainingHashST(int m) {
    this.m = m;
    st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[m];
    for (int i = 0; i < m; i++)
        st[i] = new SequentialSearchST<Key, Value>();
}
```

Implementación

- get / put

```
public Value get(Key key) {
    if (key == null) throw new NullPointerException("argument to get() is null");
    int i = hash(key);
    return st[i].get(key);
}


public void put(Key key, Value val) {
    if (key == null) throw new NullPointerException("argument is null");
    if (val == null) {
        delete(key);
        return;
    }
    // double table size if average length of list >= 10
    if (n >= 10*m) resize(2*m);
    int i = hash(key);
    if (!st[i].contains(key)) n++;
    st[i].put(key, val);
}
```

Implementación

- delete

```
public void delete(Key key) {
       if (key == null) throw new IllegalArgumentException("argument is null");

       int i = hash(key);
       if (st[i].contains(key)) n--;
       st[i].delete(key);

       // halve table size if average length of list <= 2
       if (m > INIT_CAPACITY && n <= 2*m) resize(m/2);
   }
```

Implementación

- Iterador sobre las llaves

```
public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        for (int i = 0; i < m; i++) {
            for (Key key : st[i].keys())
                queue.enqueue(key);
        }
        return queue;
    }
```

Ver la implementación completa

Análisis de desempeño

- Peor caso: Puede ocurrir una colisión entre N objetos. En este caso, todos coinciden en una sola posición del arreglo y las búsquedas se reducen a búsquedas secuenciales, *i.e.* el número de comparaciones es ~N.
- Caso promedio: Asumiendo una distribución uniforme de los hashes, entonces los N objetos se distribuyen entre las M posiciones dando listas de aproximadamente N/M elementos. Por tanto el número de comparaciones seria ~N/M.

Valor de M

- Es definido por el cliente al momento de crear la tabla de dispersión.
- Se puede seleccionar M>N, con esto se logra que en promedio la longitud de las listas sea N/M<1.
- En estas condiciones el tiempo de acceso es constante.
- N no se conoce de antemano. Se puede aplicar la técnica de hacer `resize()` del arreglo para mantener el invariante N/M<1.

Ejemplo: Intersección ~N

```
public static <T> Iterable<T> interseccion(T[] a, T[] b) {
    SeparateChainingHashST<T,Boolean> aKeys = new SeparateChainingHashST<>();
    Queue<T> resultado = new Queue<>();
    for(T x: a)
        aKeys.put(x, true);
    for(T x: b)
        if (aKeys.contains(x)) resultado.enqueue(x);
    return resultado;
}
```