Estructuras de Datos
Clase 8 – Mapeos, Diccionarios y Conjuntos

Dr. Sergio A. Gómez
http://cs.uns.edu.ar/~sag

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Claves y prioridades

- Clave: Atributo de un individuo que sirve para identificarlo en un conjunto de individuos
- Ejemplo: DNI, Número de libreta para alumnos.
- Ejemplo: Número de afiliado para afiliados a una obra social.
- Ejemplo: Número de cuenta para una cuenta en un banco.

ADT Comparador

Problema: ¿Cómo comparar claves de tipo genérico K?

compare(a,b) = Retorna un entero i tal que:

- i<0, si a<b
- i=0, si a=b
- i>0, si a>b

Ocurre un error si a y b no pueden ser comparados.

Está especificado por la interfaz java.util.Comparator.

Ejemplo de Comparador

```
public class Persona {   // Archivo: Persona.java
        protected String nombre;
        … otros atributos…
        public Persona(String nombre ) { this.nombre = nombre; }
        public String getNombre() { return nombre; }
        … otras operaciones…
}
public class ComparadorPersona<E extends Persona>   // Genericidad restringida
        implements java.util.Comparator<E> {  // Archivo: ComparadorPersona.java
    public int compare( E a, E b ) {   // Comparo las personas por su nombre
        return  a.getNombre().compareTo(b.getNombre());
    }
}
public class Test {    // Archivo: Test.java
   public static void main( String [] args ) {
        Persona p1 = new Persona( “Sergio” );
        Persona p2 = new Persona( “Tomás” );
        ComparadorPersona<Persona> comp = new ComparadorPersona<Persona>();
        if( comp.compare( p1, p2 ) < 0 ) System.out.println( “p1 es menor que p2” );
        else if comp.compare( p1, p2 ) == 0 ) System.out.println( “p1 es igual a p2” );
        else System.out.println( “p1 es mayor que p2” );
   }
}
```

Comparador por defecto

El comparador por defecto delega su comportamiento en el comportamiento de la operación compareTo del tipo básico E:

```
public class DefaultComparator<E extends Comparable<E>>
   implements java.util.Comparator<E> {
            public int compare( E a, E b ) {
                        return a.compareTo( b );
            }
}
public class Test2 {
   public static void main( String [] args ) {
            Integer x1 = 5, x2 = 1;
            DefaultComparator<Integer> comp = new DefaultComparator<Integer>();
            System.out.println( "comp(x1,x2): " + comp.compare(x1, x2)); //  retorna 1

            String s1 = "pedro", s2 = "pablo";
            DefaultComparator<String> comp2 = new DefaultComparator<String>();
            System.out.println( "comp(s1,s2): " + comp2.compare(s1, s2)); // retorna 4

            Float f1 = 5.3f, f2 = 21.8f;
            DefaultComparator<Float> comp3 = new DefaultComparator<Float>();
            System.out.println( "comp(f1,f2): " + comp3.compare(f1, f2)); // retorna -1
   }
}
```

Entradas

Problema: ¿Cómo asociar claves con valores?

```
public interface Entry<K,V> {
        public K getKey();          // Retorna la clave de la entrada
        public V getValue();        // Retorna el valor de la entrada
}
public class Entrada<K,V> implements Entry<K,V> {
        private K clave;
        private V valor;

        public Entrada(K k, V v) { clave = k; valor = v; }
        public K getKey() { return clave; }
        public V getValue() { return value; }
        public void setKey( K k ) { clave = k; }
        public void setValue(V v) { value = v; }
        public String toString( ) {
                return "(" + getKey() + "," + getValue() + ")" ;
        }
}
```

Mapeo

- Un mapeo permite almacenar elementos de tal manera que puedan ser localizados rápidamente usando una clave.
- Motivación: Un elemento generalmente tiene información útil además de su clave.
- Ejemplo: Un alumno tiene clave Libreta pero almacena nombre, fecha de nacimiento, dirección, teléfono, correo electrónico, etc.
- Un mapeo puede ser interpretado como una función (o correspondencia) de dominio K y codominio V.
- Un mapeo m puede ser interpretado como un conjunto de entradas (k,v) donde k tiene tipo K y v tiene tipo V:

$$m = \{ (k_1,v_1), (k_2,v_2), \ldots, (k_n,v_n)\}$$

ADT Mapeo (Map)

Dado un mapeo M:

- **size():** Retorna el número de entradas de M
- **isEmpty():** Testea si M es vacío
- **get(k):**
  - Si M contiene una entrada e con clave igual a k, retorna el valor de v;
  - sino retorna null
- **put(k,v):**
  - Si M no tiene una entrada con clave k, entonces agrega una entrada (k,v) a M y retorna null
  - Si M ya tiene una entrada e con clave k, reemplaza el valor con v en e y retorna el valor viejo de e.

- **remove(k):**
  - Remueve de M la entrada con clave k y retorna su valor
  - Si M no tiene entrada con clave k, retorna null.
- **keys():**
  - Retorna una colección iterable de las claves en M.
  - keys().iterator(): retorna un iterador de claves.
- **values():**
  - Retorna una colección iterable con los valores de las claves almacenadas en M
  - values().iterator(): retorna un iterador de valores
- **entries():**
  - Retorna una colección iterable con las entradas de M.
  - entries().iterator() retorna un iterador de entradas.

Ejemplo

Operación	Salida	Mapeo M de enteros en caracteres
isEmpty()	true	∅
put(5, A)	null	{(5,A)}
put(7,B)	null	{(5,A), (7,B)}
put(2,C)	null	{(5,A), (7,B), (2,C)}
put(8,D)	null	{(5,A), (7,B), (2,C), (8,D)}
put(2,E)	C	{(5,A), (7,B), (2,E), (8,D)}
get(7)	B	{(5,A), (7,B), (2,E), (8,D)}
get(4)	null	{(5,A), (7,B), (2,E), (8,D)}
get(2)	E	{(5,A), (7,B), (2,E), (8,D)}
size()	4	{(5,A), (7,B), (2,E), (8,D)}
remove(5)	A	{(7,B), (2,E), (8,D)}
remove(2)	E	{(7,B), (8,D)}
get(2)	null	{(7,B), (8,D)}
isEmpty()	false	{(7,B), (8,D)}

Interfaz Map en Java

```
public interface Map<K,V> {
        public int size();
        public boolean isEmpty();
        public V get(K k);
        public V put(K k, V v);
        public V remove(K k);
        public Iterable<K> keys();
        public Iterable<V> values();
        public Iterable<Entry<K,V>> entries();
}
```

Ejemplo

```
Map<Integer, Character> mapeo =
                    new MapeoConLista<Integer, Character>();
System.out.println( mapeo.isEmpty() );
mapeo.put( 5, ‘A’ );
mapeo.put( 7, ‘B’ );
mapeo.put( 2, ‘C’ );
mapeo.put( 8, ‘D’ );
mapeo.put( 2, ‘E’ );
System.out.println( mapeo.get(2) );

for( Entry<Integer,Character> e : mapeo.entries())
{
        System.out.println("Entrada: " + e);
}
```

Implementación de Mapeo con Lista

```
public class MapeoConLista<K, V> implements Map<K, V> {
    protected ListaDoblementeEnlazada<Entrada<K,V>> S;

    public MapeoConLista() { S = new ListaDoblementeEnlazada<Entrada<K,V>>(); }

    // put(k,v): Si M no tiene una entrada con clave k, entonces
    // agrega una entrada (k,v) a M y retorna null
    // Si M ya tiene una entrada e con clave k,
    //reemplaza el valor con v en e y retorna el valor viejo de e.
    public V put( K k, V v ) {
        for( Position<Entrada<K,V>> p : S.positions() )
            if( p.element().getKey().equals( k ) {
                V aux = p.element().getValue();
                p.element().setValue( v );
                return aux;
            }
        S.addLast(new Entrada<K,V>(k, v) );
        return null;
    }
}
```

Complejidad:

Si el mapeo tiene n entradas:

$T_{put}(n) = O(n)$

Implementación de Mapeo con Lista (cont)

```
// get(k): Retorna el valor para la clave k en M o null si no hay entrada con clave k.
public V get(K k) {
        for( Position<Entrada<K,V>> p : S.positions() )
                if( p.element().getKey().equals( k ) )
                        return p.element().getValue();
        return null;
}
```

Complejidad:
Si el mapeo tiene n entradas:
$T_{get}(n) = O(n)$

```
// remove(k): Retorna el valor removido v para (k,v) o null si k no está en M
public V remove(K k ) {
        for( Position<Entrada<K,V>> p : S.positions() )
                if( p.element().getKey().equals( k ) ) {
                        V v = p.element().getValue();
                        S.remove( p );
                        return v;
                }
        return null;
}
```

Complejidad:
Si el mapeo tiene n entradas:
$T_{remove}(n) = O(n)$

Aplicación

- Problema: Dada una cola Q de caracteres, determinar la frecuencia de apariciones de cada carácter.
- Nota: La frecuencia de apariciones de un carácter C se define como la cantidad de apariciones de C dividido la cantidad total de caracteres.
- Ejemplo: Si ‘R’ apareció 10 veces y hay 100 caracteres en la secuencia, entonces la frecuencia de apariciones de ‘R’ es 10/100 = 0,1.

```
public static void calcularFrecuenciaApariciones( Queue<Character> q ) {
    Map<Character, Integer> m = new MapeoConLista<Character, Integer>();
    int cantidadTotalCaracteres = q.size();
    while( !q.isEmpty() ) {
        char c = q.dequeue();
        if( m.get(c) == null )
                m.put( c, 1 );
        else
                m.put( c, m.get(c) + 1 );
     }

     for( Entry<Character, Integer> e : m.entries() )
        System.out.println( "La frecuencia de " + e.getKey() + " es " +
                ((float) e.getValue()) / cantidadTotalCaracteres );
}
```

Complejidad temporal:

Sea n = cantidad de elementos de la cola q

$T(n) = c_1 + n(c_2 + 2T_{get}(n) + T_{put}(n)) + c_3 n$

$= c_1 + n(c_2 + 3n) + c_3 n = O(n^2)$

Alternativa: Si a= tamaño del alfabeto => T(n,a) = c1 + n(c2 + 3a) + a = O(na)

Diccionarios

- Un diccionario almacena pares clave-valor (k,v) (como los mapeos)
- Las claves son de tipo K y los valores de tipo V genéricos (como los mapeos)
- (A diferencia de los mapeos) un diccionario puede almacenar claves repetidas.
- Hay dos tipos de diccionarios:
  - Diccionarios no ordenados: Las claves sólo se comparan por igual.
  - Diccionarios ordenados: Hay una relación de orden total definida para las claves.

ADT Diccionario

Dado un diccionario no ordenado D:

- **size():** Retorna el número de entradas de D
- **isEmpty():** Testea si D está vacío
- **find(k):** Si D contiene una entrada e con clave igual a k, entonces retorna e, sino retorna null
- **findAll(k):** Retorna una colección iterable conteniendo todas las entradas con clave igual a k
- **insert(k,v):** Inserta en D una entrada e con clave k y valor v y retorna la entrada e
- **remove(e):** Remueve de D una entrada e, retornando la entrada removida; ocurre un error si e no está en D
- **entries():** Retorna una colección iterable con las entradas clave-valor de D.

Ejemplo

Operación	Salida	Diccionario de enteros en caracteres
insert(5,A)	(5,A)	{(5,A)}
insert(7,B)	(7,B)	{(5,A), (7,B)}
insert(2,C)	(2,C)	{(5,A), (7,B), (2,C)}
insert(8,D)	(8,D)	{(5,A), (7,B), (2,C), (8,D)}
insert(2,E)	(2,E)	{(5,A), (7,B), (2,C), (8,D), (2,E)}
find(7)	(7,B)	{(5,A), (7,B), (2,C), (8,D), (2,E)}
find(4)	null	{(5,A), (7,B), (2,C), (8,D), (2,E)}
find(2)	(2,C)	{(5,A), (7,B), (2,C), (8,D), (2,E)}
findAll(2)	{(2,C), (2,E)}	{(5,A), (7,B), (2,C), (8,D), (2,E)}
size()	5	{(5,A), (7,B), (2,C), (8,D), (2,E)}
remove(find(5))	(5,A)	{(7,B), (2,C), (8,D), (2,E)}
find(5)	null	{(7,B), (2,C), (8,D), (2,E)}

Interfaz Java Dictionary

```
public interface Dictionary<K,V> {
    // size(): Retorna el número de entradas de D
    public int size();

    // isEmpty(): Testea si D está vacío
    public boolean isEmpty();

    // find(k): Si D contiene una entrada e con clave igual a k,
    //      entonces retorna e, sino retorna null
    public Entry<K,V> find(K k);

    // findAll(k): Retorna una colección iterable conteniendo
    // todas las entradas con clave igual a k
    public Iterable<Entry<K,V>> findAll(K k);

    // insert(k,v): Inserta en D una entrada e con clave k y valor v y retorna la entrada e
    public Entry<K,V> insert(K k, V v);

    // remove(e): Remueve de D una entrada e, retornando la entrada removida;
    //  ocurre un error si e no está en D
    public Entry<K,V> remove( Entry<K,V> e ) throws NonExistentEntryException;

    // entries(): Retorna una colección iterable con las entradas clave-valor de D.
    public Iterable<Entry<K,V>> entries();
}
```

Implementación de Diccionarios no ordenados con Lista

```
public class DiccionarioConLista<K,V> implements Dictionary<K,V> {
   protected ListaDoblementeEnlazada<Entry<K,V>> D;

   public DiccionarioConLista() { D = new ListaDoblementeEnlazada<Entry<K,V>>(); }
   public Iterable<Entry<K,V>> entries() { return D; }

   // insert( k,v): Agrega una entrada (k,v) a D y la devuelve
   public Entry<K,V> insert(K k, V v ) {
            Entry<K,V> e = new Entrada<K,V>(k,v);
            D.addLast( e );   // Asumo que funciona en orden 1.
            return e;
   }        T_insert(n) = O(1) si n = cantidad de entradas del diccionario

    // remove(e): Remueve la entrada e del diccionario y retorna la entrada removida.
    public Entry<K,V> remove( Entry<K,V> e ) throws NonExistentEntryException {
            for( Position<Entry<K,V>> p : D.positions() ) {
                     if( p.element() == e )  { D.remove(p);     return e; }
            }
            throw new NonExistentEntryException( "Diccionario::remove: …" );
}       T_remove(n) = O(n) si n = cantidad de entradas del diccionario
```

Implementación de Diccionarios no ordenados con Lista (cont)

```
public Iterable<Entry<K,V>> findAll( K k ) {
   ListaDoblementeEnlazada<Entry<K,V>> l =
            new ListaDoblementeEnlazada<Entry<K,V>>();
   for( Position<Entry<K,V>> p : D.positions() )
            if( p.element().getKey().equals(k) )
                     l.addLast( p.element() );
  return l;
}
```

$T_{findAll}(n) = O(n)$ cuando:

n = cantidad de entradas del diccionario

la lista tiene referencia al último nodo (trailer)

Conjuntos

Un conjunto, caracterizado con el tipo Set<E>, tiene definidas las operaciones:

- insert(x): Inserta un elemento x el conjunto
- delete(x): Elimina el elemento x del conjunto
- member(x): Retorna true si x pertenece al conjunto
- intersection(S): Interseca al conjunto con otro conjunto S y retorna un nuevo conjunto intersección
- union(S): Une al conjunto con el conjunto S y retorna un nuevo conjunto union
- complement(): Calcula y retorna el complemento del conjunto (solo se puede implementar si el dominio es finito)
- iterator(): Retorna un iterador con los elementos del conjunto

Conjuntos: Implementación

Usando una lista doblemente enlazada:

- insert(x): insertar x al principio de la lista en O(1)
- delete(x): Buscar x en la lista y eliminarlo en O(n)
- member(x): Buscar x en la lista en O(n)
- intersection(S): por cada elemento x de this buscarlo en S y si está en S insertarlo en el resultado. Si this tiene n elementos y S tiene m elementos, entonces T(n,m) = O(n*m)
- union(S): cada elemento de this se inserta en el resultado y cada elemento de S si no está en el resultado se inserta en el resultado. T(n,m) = O(n + m*n)
- complement(): para cada elemento del dominio si no está en this se agrega al resultado. T(n) = O(tamaño del dominio * n)
- iterator(): Retornar un iterador para la lista subyacente en O(1)

Tarea: Implementar la interfaz Set<E> usando como representación interna:

(1) una lista enlazada de E's y (2) un mapeo de E en Boolean.

Ejercicio: Tipo Bag (bolsa)

Un bag es una colección que soporta elementos repetidos:

Interfaz Bag<E> extiende a Iterable<E>:

void add(E item): *inserta un ítem*

boolean isEmpty(): *¿está la bolsa vacía?*

int size(): *número de elementos en la bolsa*

void remove(E item)*: elimina una aparición de ítem en la bolsa*

int getCount(E ítem): *cuenta cuántas veces está ítem en la bolsa*

Implementar la interfaz Bag usando dos representaciones posibles: (1) una lista doblemente enlazada y (2) un diccionario.

Bibliografía

- Capítulo 9 de M. Goodrich & R. Tamassia, Data Structures and Algorithms in Java. Fourth Edition, John Wiley & Sons, 2006.