

Estructuras de Datos

EEDD - GRADO EN ING. INFORMÁTICA - UCO

Mapas

23/04/25 fjmadrid@uco.es

ChangeLog

23/4/2025

- Corregida tabla final comparando complejidades asumiendo que el borrado no necesita localizar.
- Mejoras estéticas y corrección de erratas.

EEDD - GRADO EN ING. INFORMATICA - UCO

Contenidos

- Concepto de “mapa”.
- Especificación.
- Diseño.

EEDD - GRADO EN ING. INFORMÁTICA - UCO

Motivación

- Se está diseñando un detector de ataques DOS (“Denial Of Service”) para un servidor (por ejemplo DNS, web, mail, sshd ...).
- Para ello se rastrea, cada segundo, un log del sistema con las ip’s que han accedido al servicio durante la última hora.
- Si una ip ha accedido más de número máximo de veces, se considera que está haciendo un ataque DOS y se bloqueará (banea) en el firewall durante un tiempo predeterminado.

Motivación

```
Algorithm detectDOS(  
  log:Log, //Array of pairs <Time, IP>  
  maxAcc:Integer) //Max. num. of acc.  
Aux  
  i:Integer //First unprocessed line of log.  
  j:Integer //First line in current 1h window.  
  c:? //Save a counter by active ip.  
Begin  
  i ← 0  
  j ← 0  
  while system::sleep(1) do //sleep 1 second.  
    updateCounters(log, i, j, c, maxAcc)  
  end-while  
end.
```

```
Algorithm updateCounters(  
  log:Log,  
  Var i:Integer,  
  Var j:Integer,  
  Var c:?,  
  maxAcc:Integer)  
Begin  
  //update new accesses.  
  while log[i].time < system::now() do  
    increment(log[i].ip, c)  
    if nAcc(log[i].ip, c) >= maxAcc then  
      system::banIP(log[i].ip)  
    end-if  
    i ← i + 1  
  end-while  
  //remove old accesses.  
  while log[j].time < system::now()-3600 do  
    decrement(log[j].ip, c)  
    j ← j + 1  
  end-while  
end.
```

```
Algorithm nAcc(ip:IP, c?):Integer //number of actives accesses for ip.
```

```
Algorithm increment(ip:IP, c?) //increment the number of accesses.  
  //post: nAcc(ip, c) = nAcc(ip:IP, old(c))+1
```

```
Algorithm decrement(ip:IP, c?) //decrement the number of accesses.  
  //post: nAcc(ip, c) = nAcc(ip:IP, old(c))-1
```

Motivación

- Diseño simple: c es un array.
 - Una IP tiene la forma 150.214.110.3 → se puede convertir en un entero de 32bits: $150 \times 2^{24} + 214 \times 2^{16} + 110 \times 2^8 + 3 = 2530635267$
 - Representar c como un array de 2^{32} enteros = 1Gb (un contador para cada posible IP aunque sólo estén activas 2).
 - Tiempo: $O(1)$
 - Memoria: $O(2^{32})$
 - ¿IPv6?
 - Memoria: $!!O(2^{128})!!$

```
Algorithm ipToInt(ip:IP):Integer //O(1)
    return ip[0]*2^24+ip[1]*2^16+ip[2]*2^8+ip[3]
```

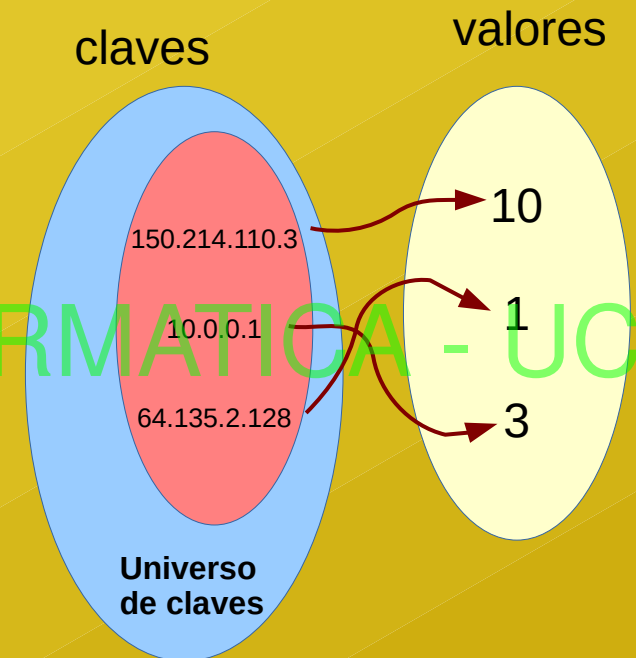
```
Algorithm nAcc(ip:IP, c: ???):Integer //O(1)
    return c[ipToInt(ip)]
```

```
Algorithm increment(ip:IP, var c:Array[Integer])//O(1)
    c[ipToInt(ip)] ← c[ipToInt(ip)] + 1
```

```
Algorithm decrement(ip:IP, var c:Array[Integer])//O(1)
    c[ipToInt(ip)] ← c[ipToInt(ip)] - 1
```

Mapa

- Definición.
 - Colección de pares <clave:valor>
 - No hay un orden definido.
 - No hay duplicación de claves.
 - Otros nombres: diccionario, arreglo asociativo, tabla de símbolos.
 - Se busca optimizar la obtención de un valor dada la clave.
 - Para nuestro problema de DOS c puede ser un mapa de Ip's a Enteros (contador).
 - Se dice **ordenado** si su recorrido genera una secuencia de pares <clave:valor> ordenada por el campo clave.



Mapa

- Especificación: TAD Par.

ADT Pair[K,V]

Makers

- **create**(K k, V v)//make a pair
 - post-c: `getFirst()==k`
 - post-c: `getSecond()==v`

Observers

- **first**():K //gets the first value.
- **second**():V //gets the second value.

Modifiers

- **setFirst**(k:K)//sets the first value.
 - post-c: `getFirst()==k`
- **setSecond**(v:V)// sets the second value.
 - post-c: `getSecond()==v`

Mapa

- Especificación: TAD Map e Iterador.

ADT Map[K,V]

Makers:

- **create()** // make an empty map.
- **post-c: isEmpty()**

Observers:

- **isEmpty():Bool** //Is the map empty?.
- **has(k:K):Bool** //Does exist a pair <k, _>?
- **get(k:K):V** //The value part of the pair <k, v>
 - **Pre-c: has(k)**
- **begin():MapIterator** // get an iterator to the begin.
- **end():MapIterator** // get an iterator to the end.
- **find(k:K):MapIterator** //find the pair <k, _>
 - **Post-c: retV==end() or retV.getKey()==k**

Modifiers:

- **insert(k:K,v:V)** //insert or update the value of the pair <k,v>
 - **Post-c: find(k).getValue()==v**
- **remove(i:MapIterator)** // remove the pair pointed by i.
 - **Pre-c: i.isValid()**
 - **Post-c: not has(i.getKey())**

ADT MapIterator[K,V]:

Observers:

- **isValid():Bool** //Is the iterator in [begin, end) sequence?
- **getKey():K** //get the key part.
 - **Pre-c: isValid()**
- **getValue():V** //get the value part.
 - **Pre-c: isValid()**

Modifiers:

- **setValue(v:V)** // set the value part.
 - **Pre-c: isValid()**
 - **Post-c: getValue()==v**
- **gotoNext()** //move iterator to next pair.
 - **Pre-c: isValid()**

Mapa

- Diseño del detector de DOS usando un mapa.

```
Algorithm nAcc(ip:IP, c:Map[IP, Integer]):Integer
```

```
  it <- c.find(ip)
```

```
  If it <> c.end() Then
```

```
    Return p.getValue()
```

```
  Else
```

```
    Return 0
```

```
Algorithm increment(ip:IP, var c:Map[IP, Integer])
```

```
  it <- c.find(ip)
```

```
  If it <> c.end() Then
```

```
    it.setValue(it.GetValue() + 1)
```

```
  Else
```

```
    c.insert(ip, 1)
```

```
Algorithm decrement(ip:IP, var c:Map[IP, Integer])
```

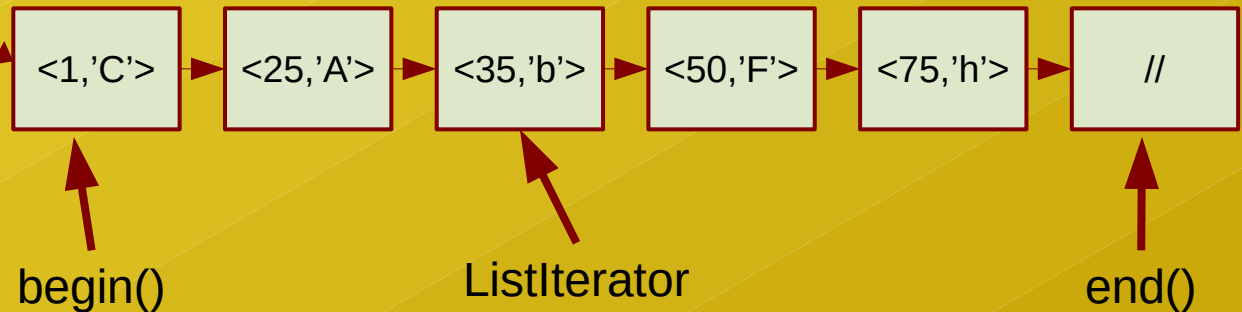
```
  //pre: c.has(ip)
```

```
  c.insert(ip, c.get(ip) - 1)
```

Mapa

- Diseño usando una lista de pares $\langle K:V \rangle$ (lista asociativa).
 - Útil cuando se estiman pocas entradas.
 - $\text{find}(k:K)$ $O(N)$
 - Podemos usar una lista ordenada por “key”.
 - Mapa ordenado pero “insert” con $O(N)$.

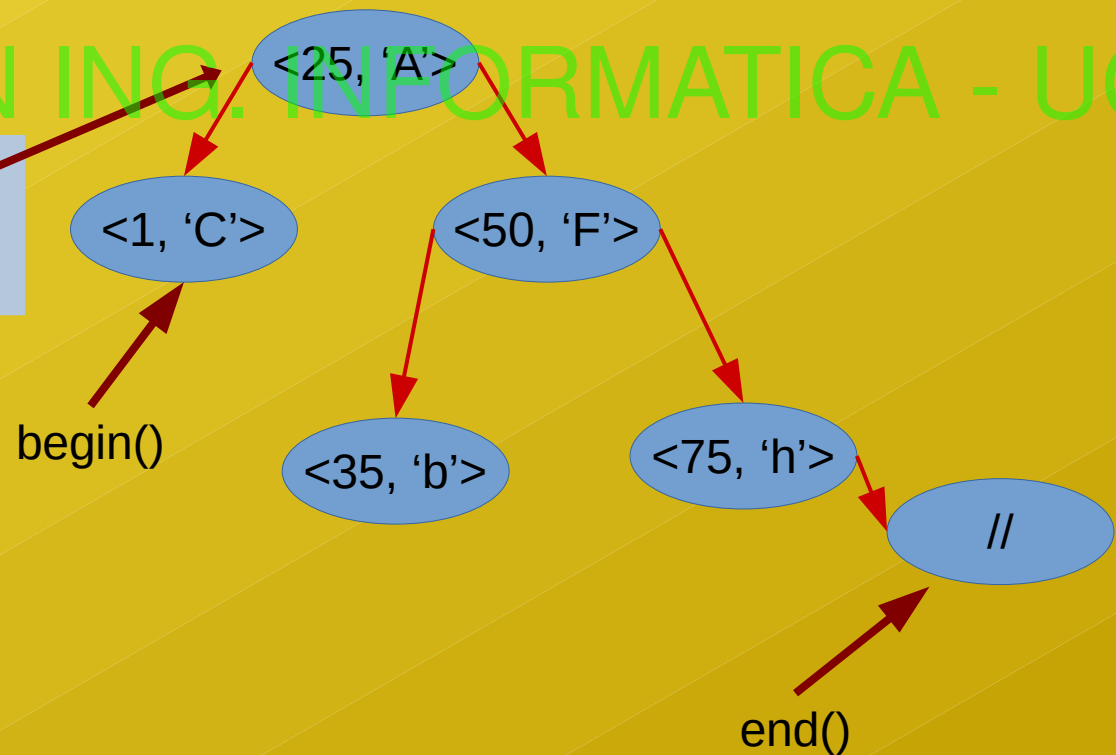
Map[K,V]
`list_:List[Pair[K,V]]`



Mapa

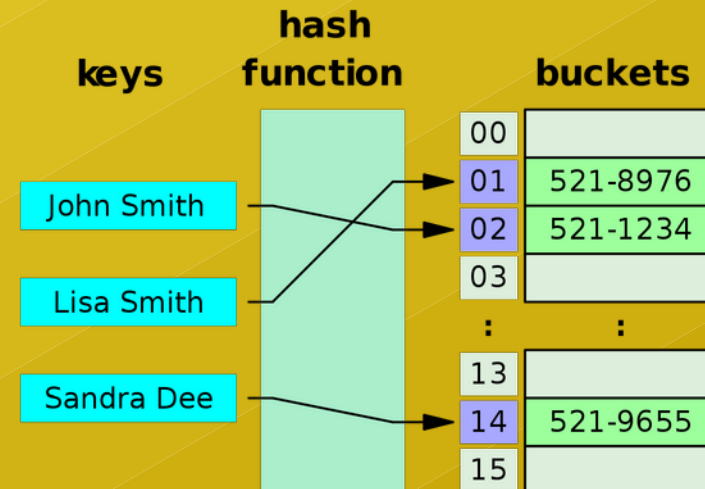
- Diseño con un BSTree.
 - Mejor si está equilibrado en altura: AVL/Red-Black.
 - $\text{find}(K) \rightarrow O(\log N)$.
 - Será un mapa ordenado con insert $O(\log N)$.

Map[K,V]
`data_:BSTree[Pair[K,V]]`



Mapa

- Implementación con una tabla HASH.
 - Combinación de array (tabla) con **función hash**:
 - h : Keys \rightarrow Integers.
 - La función hash **mapea** las claves a un índice del array con insert/remove con coste amortizado $O(1)$.
 - Se pueden producir **colisiones**.
 - El mapa no es ordenado.



Mapas

- Comparación de implementaciones.

Operación	Buscar		Insertar		Borrar (sin buscar)		¿Ordenado?
Diseño	Promedio	Peor caso	Promedio	Peor caso	Promedio	Peor caso	
Lista	$O(N)$	$O(N)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	No
Lista ordenada	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$O(1)$	Sí
ABO H = altura del árbol	$O(H)$	$O(N)$	$O(H)$	$O(N)$	$O(H)$	$O(H)$	Sí
AVL H = $\log(N)$	$O(H)$	$O(H)$	$O(H)$	$O(H)$	$O(H)$	$O(H)$	Sí
Hashing	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(1)$	No

Resumiendo

- Una mapa permite representar una función que mapea un conjunto de valores (las claves) en otro conjunto (el valor asociado a una clave).
- Tiene distintos nombres: mapa, diccionario, array asociativo, tabla de símbolos.
- Se puede implementar usando otros TAD como lista asociativa, ABO o una Tabla Hash.

Referencias

- Lecturas recomendadas:
 - Wikipedia: en.wikipedia.org/wiki/Associative_array

EEDD - GRADO EN ING. INFORMÁTICA - UCO