Estructuras de Datos

Tablas Hash

fjmadrid@uco.es

ChangeLog

24/4/2025
- Adaptar a usar iteradores.
- Mejoras estéticas.

25/4/2025
- Añadido functor Key2Int al crear una tabla hash.
- Corregir diseño del algoritmos usando iteradores.

29/4/2025
- Añadidos diseños de algoritmos para encadenamiento.
- Añadidos diseños para iterador.

Contenidos

- Definición de Tabla Hash.
- Diseño de funciones hash.
- Técnicas para la resolución de colisiones.

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 en el firewall durante un tiempo predeterminado.

Motivación

```
Algorithm DOS_detector(
  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)– 1
    j ← j + 1
  end-while
end.
```

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

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

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

Motivación

- Implementación simple.
  - 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 (un contador para cada posible IP).
  - 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:IO, var c:Array[Integer])//O(1)
  c[ip2Int(c)] ← c[ip2Int(c)] + 1

Algorithm decrement(ip:IP, var c:Array[Integer])//O(1)
  c[ip2Int(c)] ← c[ip2Int(c)] - 1
```

Tabla Hash

- Definición.
  - Combinación de Array de tamaño m y una **función hash**:
    - h: Keys → Integers {0...m-1}.
    - m es la cardinalidad de la función h.
  - La función hash **mapea** las claves a un índice del array con O(1).
  - Se producirán **colisiones** si |K| >m.

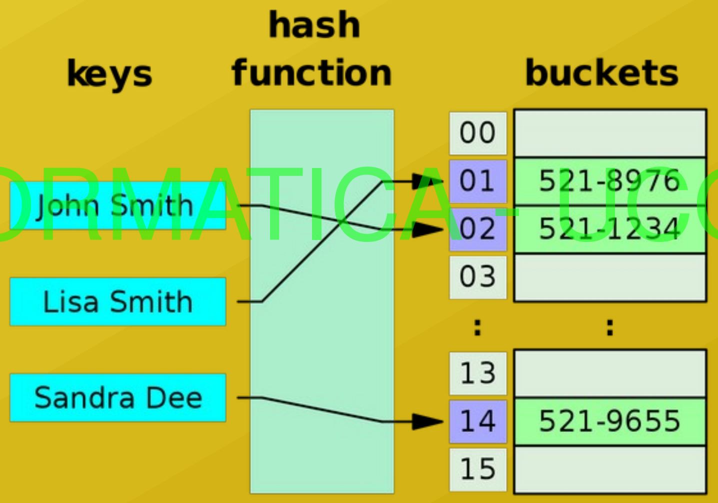


Tabla Hash

- Especificación.

ADT HashTable[K, V] implement Map[K,V] interface.

Makers:

- create(m:Integer, h:HashFunction, k2Int:Key2Int) //make an empty table.
  - post-c: loadFactor()==0

Observers:

- loadFactor():Float //The load factor.

Diseño

- Objetivos.
  - Se debe calcular de forma rápida.
  - Debe ser determinista.
  - Debe generar una distribución uniforme sobre el conjunto de enteros [0, M-1].
  - Debe reducir las colisiones de claves.

Ejemplos de malos diseños para una cardinalidad de 1000.

h1(num_telf):
   return <primeros 3 dídigitos> //957 211 035 → 957 (no uniforme).

h2(num_telf):
   return <últimos 3 dígitos> //957 211 035 → 035 (más uniforme, muchas colisiones)

h2(num_telf):
   return rand() % 1000 //957 211 035 → 01 (no es determinista)

Diseño

- Definición de Familia Universal.

Sea U el universo de las claves (todas las posibles claves).
Un conjunto de funciones hash:

$$H=\{h:U\rightarrow\{0,1,2,\ldots,m-1\}\}$$

es llamado una "**familia universal**" si para toda función h∈H la probabilidad de colisión es

$$Prob[h(x)=h(y)]\leqslant\frac{1}{m},\ \ con\ \ x,y\in U,x\neq y,$$

siendo 'm' es la **cardinalidad de la función hash**.

"m" también será el número de posiciones de la tabla.

fjmadrid@uco.es

Diseño

- Uso de aleatoriedad.
  - Usar h(x) = rand()%m da una prob. de colisión de 1/m y es uniforme en [0, m).
  - Pero esto no es determinista.
  - Sin embargo, si podemos seleccionar una función ‘hash’ de la familia H de forma aleatoria y usarla en el resto de algoritmos.

Diseño

- Factor de carga.
  - “*Si h se escoge de forma aleatoria dentro de una familia universal, el número de colisiones en una celda de la tabla será O(1+α).*”
  - El valor α es el factor de carga de la tabla:

$$\alpha = \frac{\text{número de claves almacenadas}}{\text{número de posiciones de la tabla}} = \frac{n}{m}$$

  - Si mantenemos α<1 conseguimos un número de colisiones por entrada **amortizado** Θ(1).

Diseño

- Determinación de la cardinalidad: conocido **N**.
  - Tablas estáticas con tamaño **m** = n/α.
  - Por ejemplo:
    - Estimamos que a lo sumo tendremos n=1000 claves y queremos tener un factor de carga a lo sumo de α = 0.5, entonces necesitamos una tabla con tamaño

$$m = 1000/0.5 = 2000.$$

Diseño

- Determinación de la cardinalidad: Tablas dinámicas.
  - No se puede estimar **n** o tiene gran varianza.
  - Copiar la idea de los arrays dinámicos. Empezar con un tamaño pequeño y aumentarlo para mantener α<1.
  - Llamar a **rehash()** después de cada operación inserción.


```
Algorithm HashTable::rehash():Integer
Var oldT:array[Entry[K,V]]; loadFactor:Float
begin
  loadFactor ← numOfUsedEntries_ / t.size()
  if loadFactor > 0.5 then
    oldT ← t_
    t_ ← Array[Entry[K,V]]::crete(t_.size()*2)
    h_ ← pickup at random from H with
          cardinalitiy t_.size()
    numOfUsedEtries ← 0
    For-Each entry in oldT: Do
       if (entry.isValid())
         insert(e.key(), e.value())
  end-if
  Return t_.size()
end.
```

```
HashTable[K,V]
t_:Array[Entry[K,V]]
h_:HashFunction
k2int_:Key2unt
numOfUsedEntires_:Integer
```

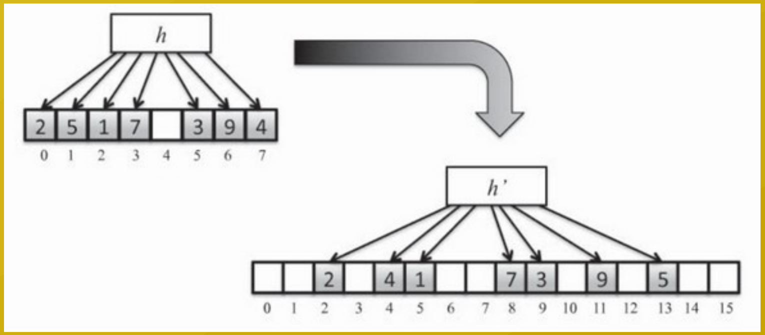


imagen tomada de [2].

Diseño de la función Hash

- Función hash para enteros.

Lema:

$$H_p = \{ h_p^{a,b}(X) = ((aX+b) \% p) \% m \}$$

$$para\ todo\ a, b : 1 \leqslant a \leqslant p-1, 0 \leqslant b \leqslant p-1$$

$$es\ una\ familia\ universal.$$

donde:

p es un número primo > |*U*|
m es la cardinalidad de la función hash (y el tamaño de la tabla).

Por lo tanto:

$$Prob[h(x)=h(y)] \leqslant \frac{1}{m}.$$

Ejemplo:
Para almacenar Ip's v4 usaremos el número primo p=4294967311 > 2^32.
Aleatoriamente han sido seleccionamos por ejemplo los valores a=32 y b=2.

Se estima que en condiciones normales el número de ip's diferentes durante una hora es alrededor de 1000. Queremos un factor de carga de α=0.5 luego m = n/α = 1000/0.5 = 2000.

Calculamos h(150.214.110.3)
ipToInt(150.214.110.3)=2530635267
h(2530635267) = ((32*2530635267+2) % p) % 2000
3670916948 % 2000 = 948

Diseño de la función Hash

- Función hash para strings: La familia polinomial.

Familia Polinomial

se define como:

$$P_p = \left\{ h_p^x(S) = \sum_{i=0}^{|S|-1} \left(x^i S[i] \% p \right) \right\}$$

con p un número primo fijo y $1 \leq x \leq p-1$

```
Algorithm PolyHash(S,p,x):Integer //O(|S|)
begin
  hash ← 0
  for i ← S.size()-1 to 0 inc -1 do
    hash ← (hash * x + S[i]) % p
  end-for
  return hash
end.
```


Supongamos que usamos x=2 y p=4294967311

Ejemplo “HOLA” → códigos ascii → 72,79,76,65

h = (72)%p + (79*2^1)%p + (76*2^2)%p + (65*2^3)%p
h = 72 + 158 + 304 + 520 = 1054

!OJO! la aritmética entera debe permitir operar con números muy grandes, al menos del orden del valor de p.

Diseño de la función Hash

- Función hash para strings: Rendimiento.

Lema:
Dados dos strings s_1 y s_2 con tamaño a lo sumo L+1, si escogemos al azar una función h_p de P_p (elegimos al azar x en [1, p-1]) la probabilidad de colisión será:

$$Prob\left[h_p(s_1)=h_p(s_2)\right]\leqslant\frac{L}{p}.$$

¿Cual será el **menor entero válido** para el parámetro **p** si queremos asegurar que el hash de dos claves distintas, de hasta 10 caracteres, no colisione con probabilidad ≤ 1/100?

fjmadrid@uco.es

Diseño de la función Hash

- Función hash para strings: acotando PolyHash.
  - El PolyHash h_p puede devolver valores [0, *p*-1] pero nuestra tabla tendrá un tamaño *m* < *p* en general.
  - Solución: usar una segunda función hash h_m para enteros elegida al azar de una familia universal H con cardinalidad m y P > p.
  - La probabilidad de colisión de dos cadenas s_1, s_2 con longitud ≤ L+1 y p>mL será:

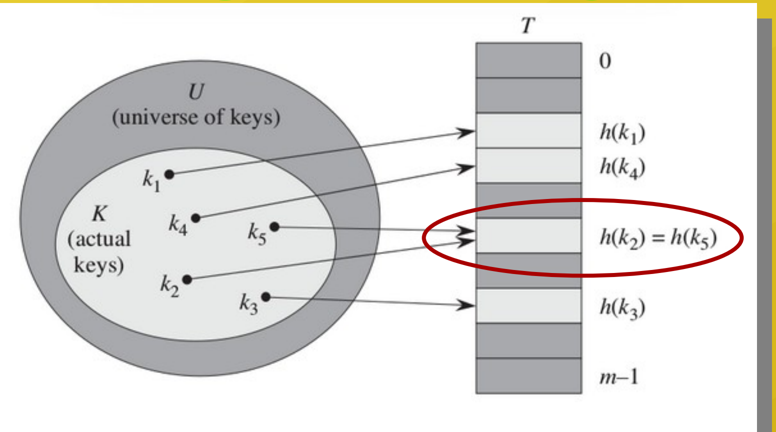
$$Prob\left[h_m(h_p(s_1))=h_m(h_p(s_2))\right]\leqslant\frac{1}{m}+\frac{L}{p}.$$

Ajuste del valor PolyHash a un tamaño de la tabla m

```
Algorithm hm(s:String):Integer
Begin
  hp ← PolyHash(S, p, x)
  hm ← hash_int(hp, a, b, P, m)
  return hm
End.
```

Resolución de colisiones

- Alternativas.
  - Direccionamiento cerrado (chaining”).
  - Direccionamiento abierto (“open addressing”).

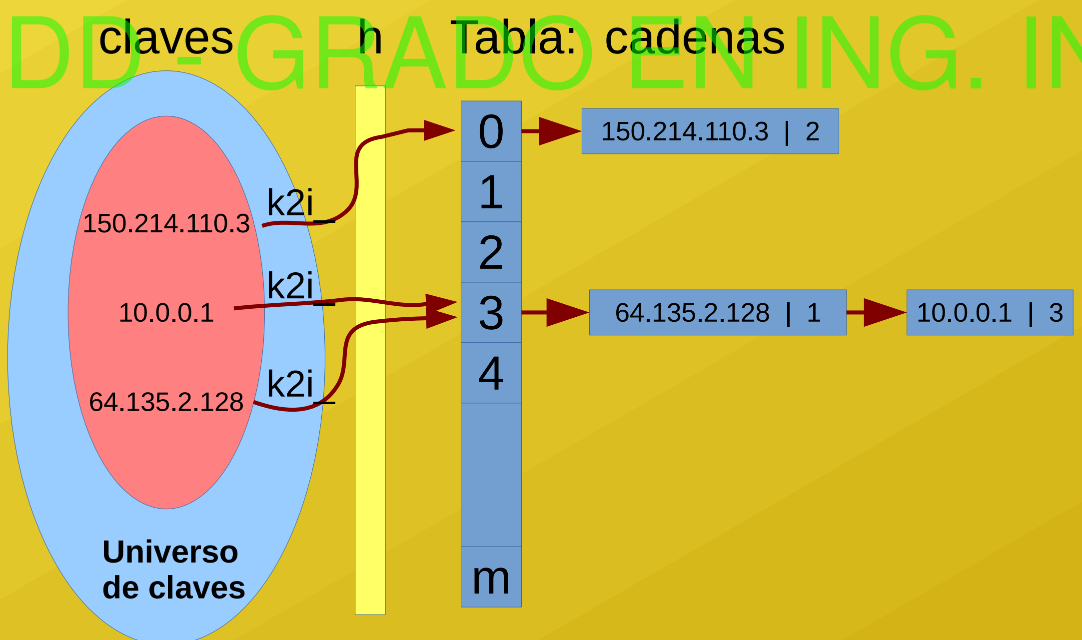


From "Introduction to Algorithms", Cormen, TH. MIT Press,

colisión

Resolución de colisiones

- Encadenamiento: diseño.
  - Cada entrada en la tabla almacena una lista (cadena) con las claves que han colisionado.



HashTable[K,V]

```
t_  :Array[List[Pair[K,V]]
h_  :HashFunction
k2i_: key2Int
num_keys_:Integer
```

HashTableIterator[K,V]

```
t_  :Array[List[Pair[K,V]]
idx_:Integer
it_ :List[Pair[K,V]::Iterator
```

Resolución de colisiones

- Encadenamiento: diseño.

```
HashTable::find(k:K):HashTableIterator
Var
  idx:Integer
  it :List[Pair[K,V]]::iterator
Begin
   idx := h_(k2i_(k))
   it := t_[idx].find(k)
   If it<>t_[idx].end() Then
     Return HashTableIterator(t_, idx,
                              it)
   Else
     Return end()
End.
```

```
HashTable::has(k:K):Bool
Begin
   Return find(k)<>end()
End.
```

```
HashTable::get(k:K):V
Pre-c: has(k)
Begin
   Return find(k).getValue()
End.
```

```
HashTable::Insert(k:K,v:V)
Var
  idx:Integer,
  it :List[Pair[K,V]]::iterator
Begin
  idx := h_(k2i_(k))
  it := t_[idx].find(k)
  If it=t_[idx].end() Then
    t_[idx].pushFront(Pair(k,v))
    num_keys_++
    rehash()
  Else
    it.setValue(v)
End.
```

```
HashTable::remove(it:HashTableIterator)
Pre-c: it.isValid()
Begin
  it.t_[it.idx_].remove(it.it_)
  num_keys_--
End.
```

Resolución de colisiones

- Encadenamiento: diseño.

```
HashTableIterator::create(
   t  :Array[List[Pair[K,V]],
   idx:Integer,
   it :List[Pair[K,V]]::Iterator)
Begin
   t_ := t
   idx_ := idx
   it_ := it
End.
```

```
HashTableIterator::getValue():V
Pre-c: isValid()
Begin
   Return it_.get().second()
End.
```

```
HashTableIterator::getKey():K
Pre-c: isValid()
Begin
   Return it_.get().first()
End.
```

```
HashTableIterator::gotoNext()
Pre-c: isValid()
Begin
  it_.gotoNext()
  If it_ = t_[idx_].end() Then
    Repeat
      idx_ := idx_+1
    Until idx_=t_.size()-1 Or
            t_[idx_].size()>0
    it_ := t_[idx_].begin()
  End-If
End.
```

```
HashTableIterator::setValue(v:V)
Pre-c: isValid()
Begin
   it_.get().setSecond(v)
End.
```

Resolución de colisiones

- Encadenamiento: diseño.

```
HashTable::begin():HashTableIterator
Var
  idx: Integer
  it_: List[Pair[K,V]]::iterator
Begin
  idx := 0
  While idx<t_.size()-1 And t_[idx].size()=0 Do
    idx := idx + 1
  End-While
  Return HashTableIterator(t_, idx, t_[idx].begin())
End.
```

```
HashTable::End():HashTableIterator
Begin
  Return HashTableIterator(t_, t_.size()-1,
                      t_[t_.size()-1].begin())
End.
```

Resolución de colisiones

- Encadenamiento. Rendimiento.

El gasto en memoria será **O(m+n)**

La complejidad promedio de las operaciones es **$\Theta(1+c)$** donde c es la longitud de la cadena más larga.

Si **h** se elige aleatoriamente de una familia universal entonces:

$$\mathbf{c \approx \alpha}$$

Asumiendo que $\alpha \ll 1$ las operaciones de inserción/borrado (usando lista doble) será

$$\mathbf{\Theta(1)}$$

COLORARIO:

Si se utiliza un función *h* de una *Familia Universal* y el esquema de encadenamiento para la resolución de colisiones en una tabla hash de tamaño *m* vacía,

realizar *n* operaciones de {inserción, borrado, búsqueda}, de las cuales hay O(m) operaciones de inserción, tendrá un complejidad conjunta promedio $\Theta(n)$,

De esta forma las operaciones se ejecutarán con un tiempo amortizado

$$\mathbf{\Theta(1)}$$

* ver [2] para su demostración.

Resolución de colisiones

- Direccionamiento abierto: funcionamiento general.
  - Una entrada de una tabla podrá tener tres estados:

    {vacía, ocupada, borrada}.
  - Al buscar un dato visitaremos ("probe") una secuencia de posiciones de la tabla hasta encontrarlo o determinar que no está almacenado.
  - Se redefine la función hash **h** de la forma:

    h(k:K, probe:Integer):Integer
  - La función h idealmente debe asegurar que:
    - la secuencia de pruebas para una clave k: {h(k, 0), h(k, 1), h(k, 2), …, h(k, m-1)} debe ser una permutación de las posiciones de la tabla {0, 1, 2, …, m-1}.
    - Lo ideal sería que todas las permutaciones posibles son equi probables.

Resolución de colisiones

- Direccionamiento abierto. Rendimiento.

El gasto en memoria será **O(n)**

Eligir **h** aleatoriamente de una familia universal H asegura secuencias de pruebas uniformes:

Teorema 1:
Suponiendo que el factor de carga $\alpha=n/m < 1$, el número esperado de pruebas en una **búsqueda fallida** es a lo sumo $1/(1-\alpha)$.

Teorema 2:
Suponiendo que el factor de carga $\alpha=n/m < 1$, el número esperado de pruebas en una **búsqueda satisfactoria** es a lo sumo $1/\alpha * Ln[1/(1-\alpha)]$.

*Ver en [2] su demostración.

COLORARIO:

Si $\alpha < 1$, el número de pruebas necesarias para insertar una clave serán a lo sumo $1/(1-\alpha)$ (requiere una búsqueda fallida para encontrar un hueco).

Si $\alpha<1$ es constante y (rehash) la inserción tendrá un coste amortizado **$\Theta(1)$**

Resolución de colisiones

- Direccionamiento abierto: diseño.
  - Una entrada de una tabla podrá tener tres estados: {vacía, válida, borrada}.

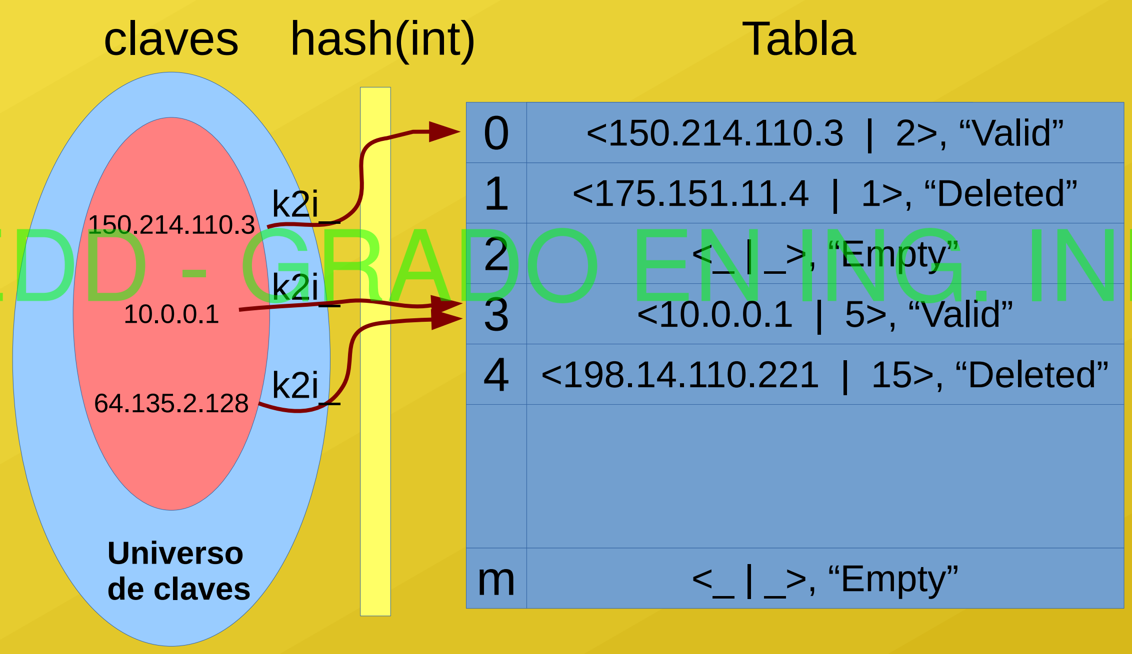
ADT TableEntry[K,V]:
Makers:
- create() //create an empty table entry.
  - post-c: isEmpty()

Observers:
- isEmpty():Bool //Is the entry empty?
- isDeleted():Bool //Is the entry deleted?
- isValid():Bool //Has the entry valid data?
- getKey():K //Get the key value.
  - pre-c: not isEmpty()
- getValue():V //Get the data value.
  - pre-c: not is Empty()

ADT TableEntry[K,V]:
Modifiers:
- set(k, v):
  - post-c: isValid()
  - post-c: getKey()==k
  - post-c: getValue()==v
- setValue(new_v:V)
  - pre-c: isValid()
  - post-c: isValid()
  - post-c: getValue()==new_v
- setEmpty()//Set the entry as empty.
  - post-c: isEmpty()
- setDeleted()//Set the entry as deleted.
  - post-c: isDeleted()

Resolución de colisiones

- Direccionamiento abierto: diseño.



HashTable[K,V]

```
t_  :Array[TableEntry[K,V]]
h_  :HashFunction
k2i_: key2Int
num_keys_:Integer
```

HashTableIterator[K,V]

```
t_  :Array[TableEntry[K,V]]
idx_:Integer
```

Resolución de colisiones

- Direccionamiento abierto: diseño.

```
Algorithm HashTable::findPosition(k:K):Integer
//pre-c: loadFactor()<1.0
Var
  keyAsInt:Integer
  probe:Integer
  goOut:Boolean;
  idx:Integer
Begin
  keyAsInt := k2Int_(k)
  probe :=  0
  Repeat
     idx := h(keyAsInt, probe)
     probe := probe+1
  Until t_[idx].isEmpty() Or t_[idx].key()=k
  Return idx
End.
```

Cuando se borra una clave, hay que dejar la entrada marcada como “borrada” en vez de “empty” para no romper las cadenas de colisiones de otras claves que pasan por esta entrada a borrar.

```
Algorithm HashTable::remove(iter:HashTableIterator)
pre-c: iter.isValid()
Begin
  t_[iter.idx_].setDeleted()
End.
```

Resolución de colisiones

- Direccionamiento abierto: diseño.

```
Algorithm HashTable::find(k:Key):HashTableIterator
Var idx:Integer
Begin
  idx := findPosition(k)
  if t_[idx].isValid() And t_[idx].key()=k Then
    Return HashTableIterator(t_, idx)
  Else
    Return end()
End.
```

```
Algorithm HashTable::has(k:Key):Bool
Begin
  Return find(k) <> end()
End.
```

```
Algorithm HashTable::get(k:Key):V
pre-c: has(k)
Begin
  Return find(k).getValue()
End.
```

Resolución de colisiones

- Direccionamiento abierto: diseño.

```
Algorithm HashTable::insert(k:K, v:V):HashTableIterator
pre-c: loadFactor()<1.0
Var
  idx:Integer
  iter:HashTableIterator
Begin
  idx := findPosition(k)
  if t_[idx].isEmpty() then
    num_used_entries_++
  t_[idx].set(k, v)
  iter := HashTableIterator(This, idx)
  If t_.size() <> rehash() Then
     iter := find(k) //what is the new position?
  Return iter
End.
```

Resolución de colisiones

- Direccionamiento abierto: diseño.

```
Algorithm HashTableIterator::gotoNext()
pre-c: isValid()
Begin
  Repeat
    idx_ := idx_+1
  Until idx_=t_.size() Or t_[idx_].isValid()
End.
```

```
Algorithm HashTableIterator::getValue()
pre-c: isValid()
Begin
  Return t_[idx_].getValue()
End.
```

```
Algorithm HashTableIterator::getKey()
pre-c: isValid()
Begin
  Return t_[idx_].getKey()
End.
```

Resolución de colisiones

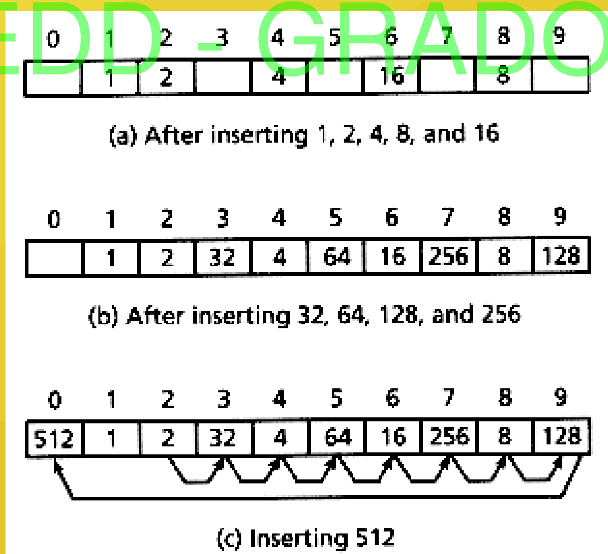
- Direccionamiento abierto: diseño.

```
Algorithm HashTable::Begin():HashTableIterator
Var idx:Integer
Begin
  idx := 0
  While idx<t_.size() And Not t_[idx].isValid()
    idx := idx+1
  Return HashTableIterator(t_, idx)
End.
```

```
Algorithm HashTableIterator::end()
Begin
  Return HashTableIterator(t_, t_.size())
End.
```

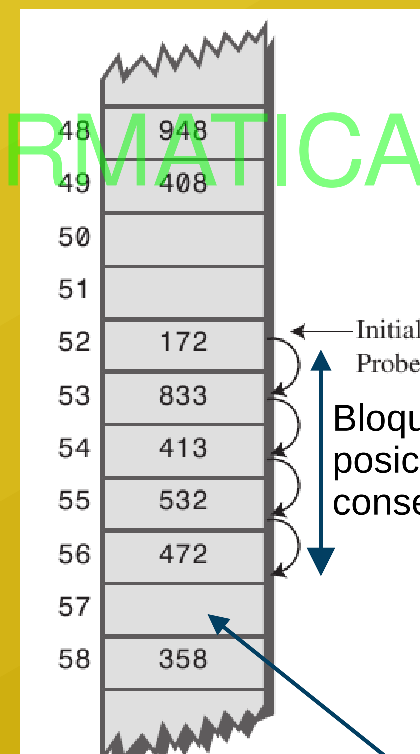
Resolución de colisiones

- Direccionamiento abierto: Linear Probing.
  - h(k, i)=(h(k,0)+i)%m
  - Genera clustering primario.



Tras el paso b)
¿Está la clave 55 ?

Tras el paso c)
Borramos la clave 4
¿por qué es necesario marcar como borrada?¿Está la clave 512?



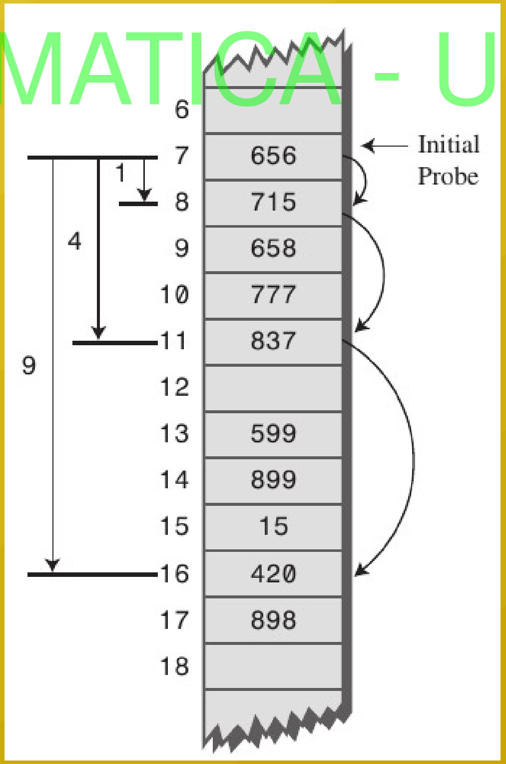
Bloque de l posiciones consecutivas.

$$Prob[crecer\ en\ 1\ posicion]=\frac{(i+1)}{m}$$

fjmadrid@uco.es

Resolución de colisiones

- Direccionamiento abierto: Quadratic Probing.
  - $h(k,i)=(h(k,0) + c_1*i + c_2*i*i)\%m$
  - Si m es potencia de dos, una buena elección es $c_1=c_2=1/2$.
  - Ver más posibilidades en [3.4].
  - Tiene clustering secundario.



Resolución de colisiones

- Direccionamiento abierto: Random probing.
  - h(k, i) : { (i==0) ? h(k,0) : (h(k, i-1)+c)%m }
  - Generar una secuencia pseudo-aleatoria con semilla h(k,0).
  - Los parámetros *c* y *m* deben ser primos relativos.
  - Reduce clustering primario, pero aún sufre de clustering secundario.

fjmadrid@uco.es

Resolución de colisiones

- Direccionamiento abierto: rehashing.
  - Tener varias funciones hash h_1, h_2, h_3, …, y si no usar “linear/quadratic/random probing” a partir de la última.
    - $h(k, i)$: { (i==0 ? $h_1(k)$: (i==1 ? $h_2(k)$: (i==2 ? $h_3(k)$: “X probing from $h_3(k)$”))) }
  - Reduce clustering primario y secundario.

Tablas Hash

- Resumiendo:
  - Son una combinación de un array con una función hash.
  - Es muy importante la elección de una buena función función hash.
  - El rendimiento y el espacio requerido dependen mucho del factor de carga.
  - Con una buena función hash y un factor de carga < 1 tendremos complejidades en tiempo amortizado de O(1).
  - Son una buena alternativa para implementar un mapa.

Referencias

- Lecturas recomendadas:

  - [1] Cap. 13 de “Data structures and software develpment in an object oriented domain”,Tremblay J.P. y Cheston, G.A. Prentice-Hall, 2001.
  - [2] Cap. 11 de “Introduction ot algorithm, Third Edition”, Thomas H. Cormen et al. MIT Press, 2009.
  - [3] Wikipedia:
    - [3.1] Tabla hash: en.wikipedia.org/wiki/Hash_table
    - [3.2] Función hash: en.wikipedia.org/wiki/Hash_function
    - Resolución de colisiones:
      - [3.3] Linear y random probing: en.wikipedia.org/wiki/Linear_probing
      - [3.4] Quadratic probing: en.wikipedia.org/wiki/Quadratic_probing
      - [3.5] Rehashing: en.wikipedia.org/wiki/Double_hashing
