UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

Welcome to Algorithms and Data Structures! - CS2100

Diccionarios

Key	Value
Apple	5
Orange	50
Peach	44
Banana	47
Plum	48
Pear	22
Strawberry	11

	A	B
1	Germany	18
2	Netherlands	15
3	Colombia	12
4	Brazil	11
5	France	10
6	Argentina	8
7	Switzerland	7
8	Algeria	7
9	Croatia	6
10	Chile	6

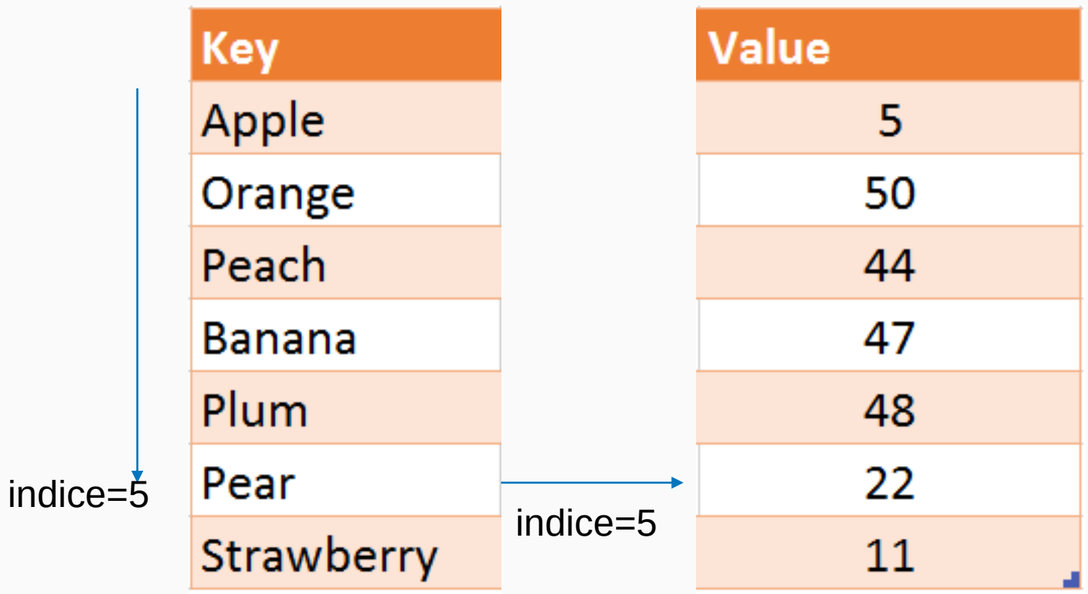
Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

MAC table

Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

Diccionario:
¿podemos utilizar dos array/listas?



¿Cómo se realiza la búsqueda?

- **Buscar el valor de “Pear”**

O(n)

Tablas hash

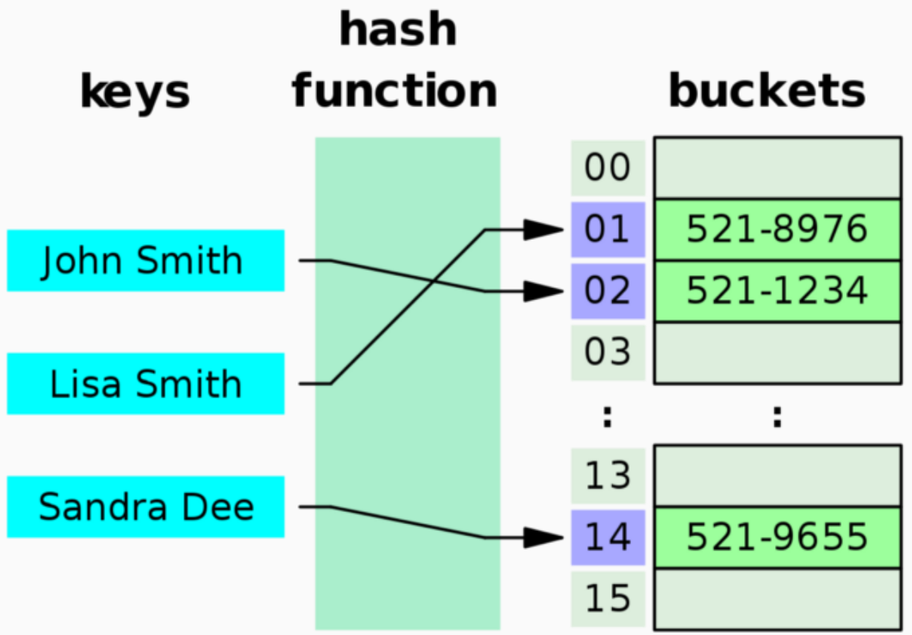
Qué son las tablas hash?
Son estructuras de datos similares a los arrays, con la diferencia que el “índice” puede ser cualquier tipo de dato comparable y no solo enteros

Restricciones:

- Claves únicas
- No sabemos en qué posición se almacenan los valores

La clave es mapeada a un índice:

```
int index = getIndex(key)
array[index] = value
```

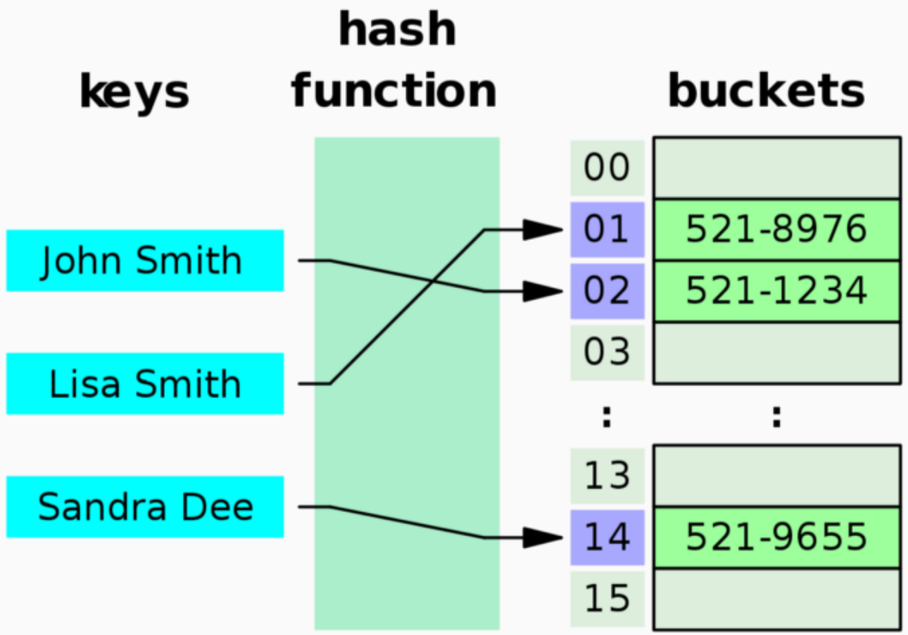


Hashing

Saben lo que es una función hash?
Es un método que nos permite obtener un índice en un array desde una key

Qué problemas podríamos tener con esa función?

- ➔ Procesar la key puede ser difícil para ciertos tipos de datos
- ➔ Manejar colisiones cuando dos keys tienen el mismo índice
- ➔ Espacio-tiempo tradeoff

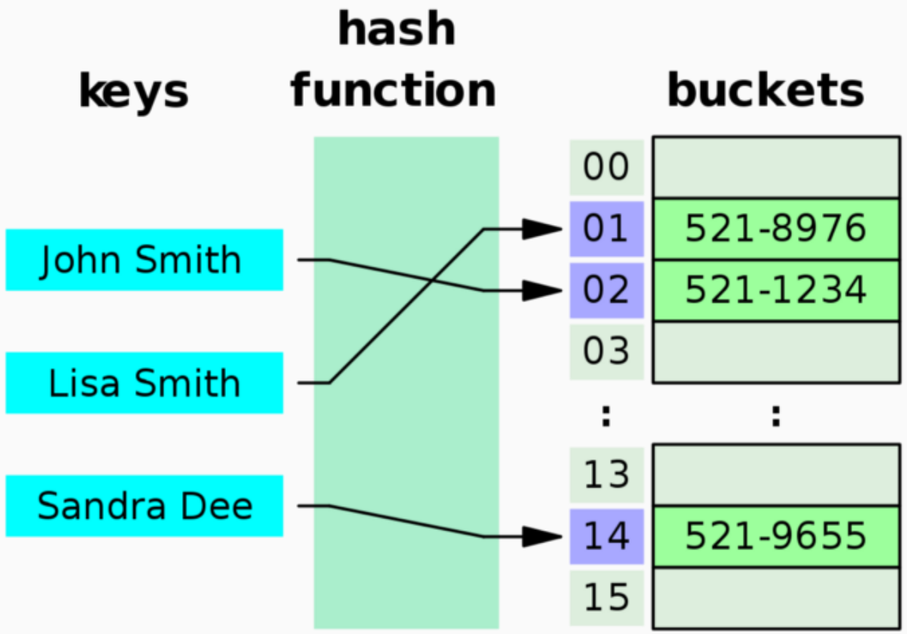


Hashing

Existen varios algoritmos para hashing, algunos retornan un output más grande que otros

Propiedades:

- Estable: El output siempre debería ser el mismo para el mismo input (invariante)
- Uniforme: Los valores hash deben ser distribuidos de manera uniforme (reducir colisiones)
- Eficiencia: Debe ser balanceado de acuerdo a las necesidades en espacio y tiempo
- Seguridad: Dado un valor hash, obtener un valor que me pueda generar ese valor hash no debería ser posible

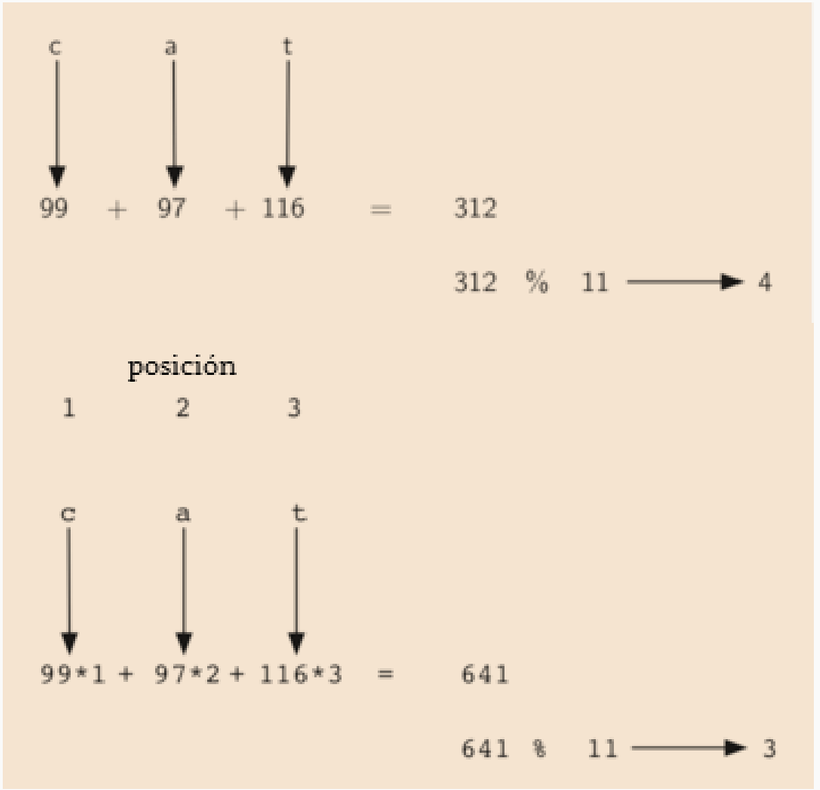


Hashing (strings)

- Implementación intuitiva (suma):

```
int additiveHash(string key) {
   int total = 0;
   for (char& character : key) {
      total += (int) character;
   }
   return total;
}
```

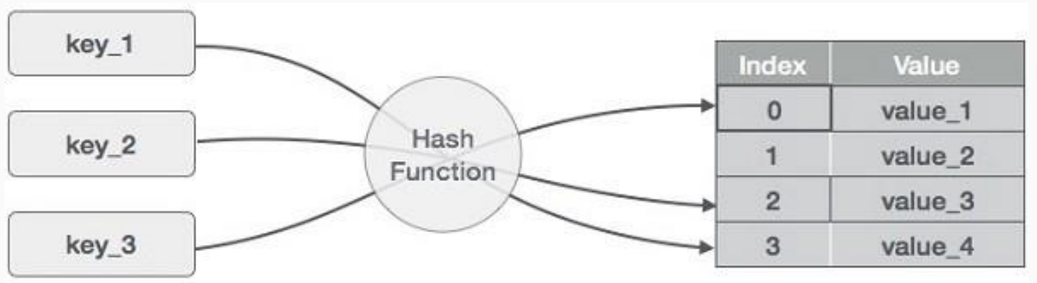
Cuál es el problema de esta función hash?



Hashing (strings)

- Algoritmo plegable:

```
const int getNextBytes(int startIndex, string str) {
  int currentFourBytes = 0;
  currentFourBytes += getByte(str, startIndex);
  currentFourBytes += getByte(str, startIndex + 1) << 8;
  currentFourBytes += getByte(str, startIndex + 2) << 16;
  currentFourBytes += getByte(str, startIndex + 3) << 24;
  return currentFourBytes;
}
```



Cuál es el problema de esta función hash?

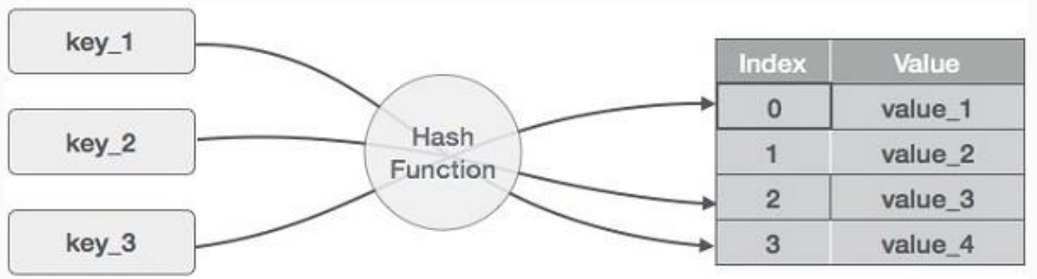
Hashing

No escriban sus propios algoritmos de hash! Utilicen las librerías ya hechas para esto

Escojan el algoritmo hash que encaje mejor con sus necesidades

MD5 solía ser un algoritmo hash bastante usado, tiene un buen balance entre estabilidad y uniformidad pero ya se han detectado colisiones.

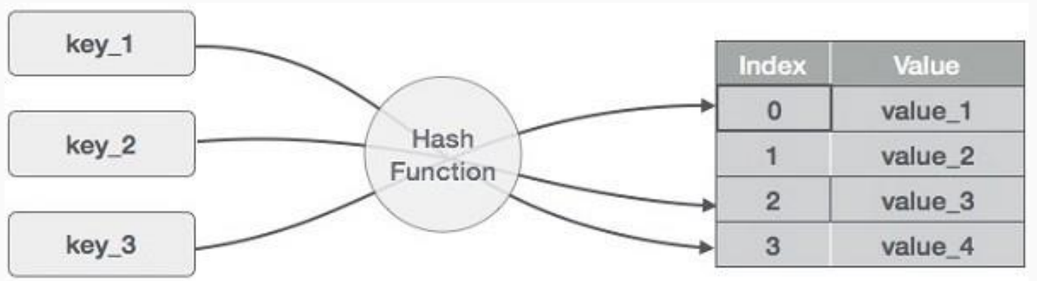
SHA-3 es un algoritmo bastante seguro pero no es muy eficiente



Agregando datos

Se inicia con un tamaño fijo del array, y se genera el hash code

int capacity = 10;
size_t **hashCode** = getHashCode(**key**);
int **index** = hashCode % capacity;
array[index] = **value**



Qué problemas podrían aparecer?

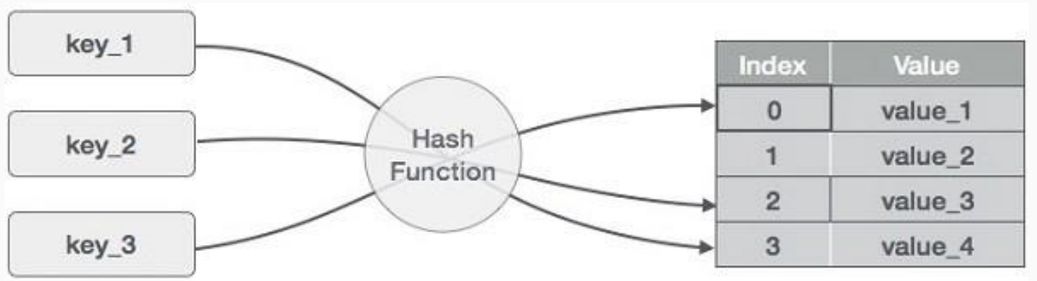
Dos elementos diferentes con el mismo hash code, significa que se les asigna el mismo índice dentro del array

Manejando colisiones

Cómo podrían manejar colisiones?

Recuerden que el usuario de una tabla hash no debe saber que hubo una colisión

- Open addressing: Moverse al siguiente índice disponible
- Chaining: Almacenar los elementos en una lista enlazada

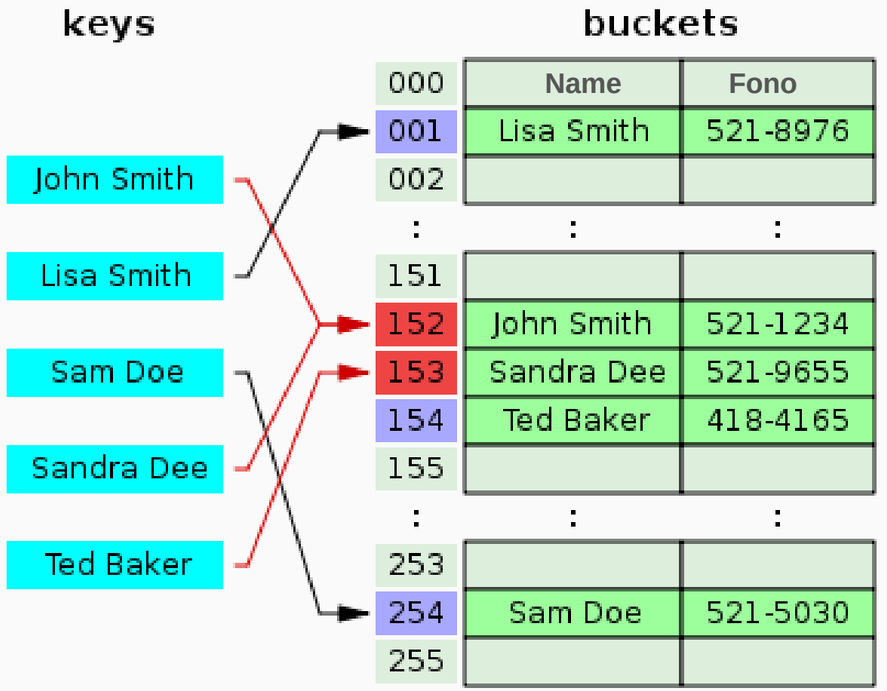


Manejando colisiones (open addressing)

Si el espacio ya está ocupado, entonces se va a buscar el siguiente espacio disponible

```
int hashCode = getHashCode(key);
int index = hashCode % capacity
while (array[index] != null){ // int/float/double is null??
    index++;
}
array[index] = value
```

Al momento de buscar el elemento se empieza a comparar el index hasta encontrar el elemento o estar seguros que no está en la tabla hash



Cómo impacta los agrupamientos (muchos index iguales) ?

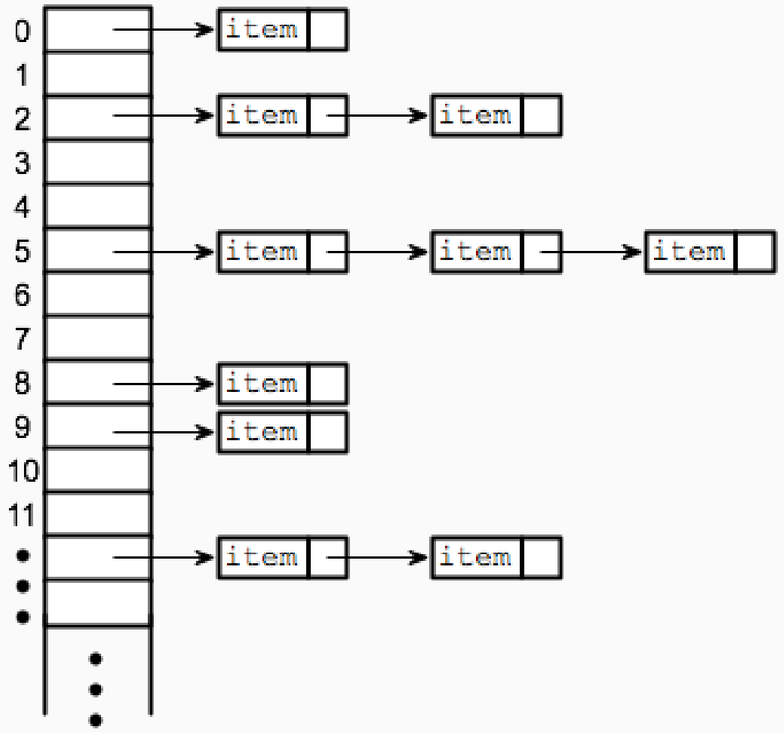
```
index = (hashCode  + salto) % arrayLength
```

Manejando colisiones (chaining)

Si el espacio ya está ocupado, entonces se agrega el elemento a la lista enlazada

```
int hashCode = getHashCode(key);
int index = hashCode % capacity
array[index].push_front(value)
```

Para encontrar el elemento, solo se debe buscar dentro de la lista enlazada



Cómo impacta los agrupamientos (muchos hashcode iguales) ?

Haciendo crecer la estructura

La probabilidad de que hayan colisiones, va a ser proporcional a la cantidad de espacios disponibles del array

Para controlar esto, se debe tener un factor de llenado, que nos indicará cuán lleno está nuestro hash table.

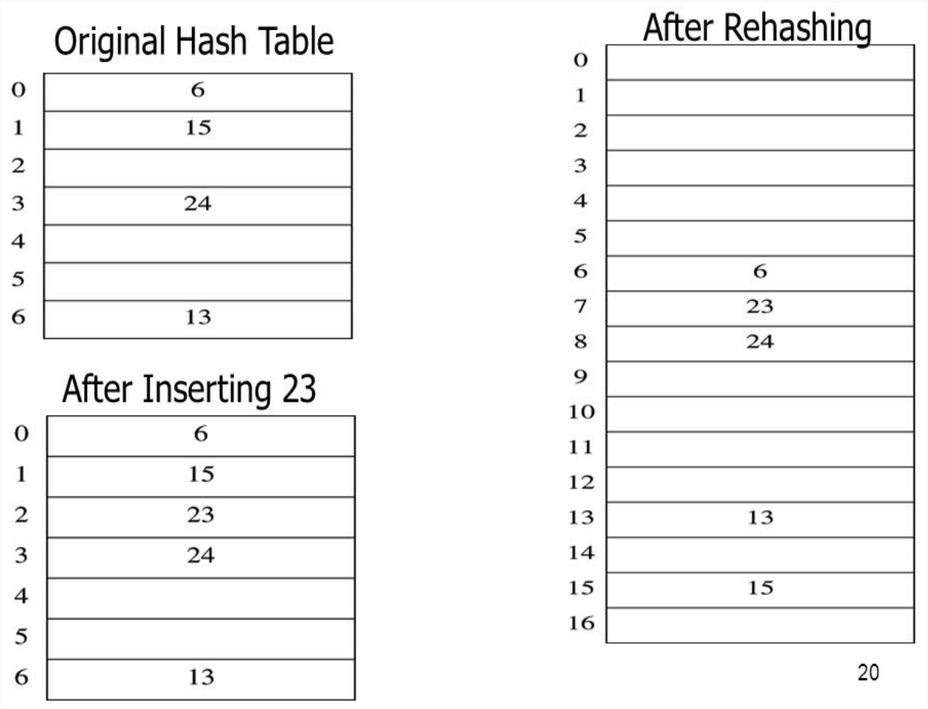
fillFactor = # of elements / hash table length

Al agregar un elemento verificar:

```
if (fillFactor >= maxFillFactor)
    rehashing();
```

¿Cuánto cuesta el rehashing?

¿Rehashing con chaining?



Manejando colisiones (open addressing)

Ejemplo:

Insertar los siguientes keys:

2, 3, 6, 8, 10, 11, 15, 19, 20, 22, 25, 30

Tamaño de la tabla hash: 5

maxFillFactor = 0,8

HashCode = Key

Resize: 2x

```
if (fillFactor >= maxFillFactor)
    rehashing();
```

Manejando colisiones (open addressing)

Ejemplo:

Insertar los siguientes keys:

2, 3, 6, 8, 10, 11, 15, 19, 20, 22, 25, 30

Tamaño de la tabla hash: 5

maxFillFactor = 0,8

HashCode = Key

Resize: 2x

```
if (fillFactor >= maxFillFactor)
    rehashing();
```

0	
1	6
2	2
3	3
4	8

0	10
1	11
2	2
3	3
4	
5	15
6	6
7	
8	8
9	19

1	
2	2
3	3
4	22
5	25
6	6
7	
8	8
9	
10	10
11	11
12	30
13	
14	
15	15
16	
17	
18	

Manejando colisiones (open addressing)

Ejemplo:

Insertar los siguientes keys:

2, 3, 6, 8, 10, 11, 15, 19, 20, 22, 25, 30

Tamaño de la tabla hash: 5

maxFillFactor = 0,8

HashCode = Key

Resize: 2x

```
if (fillFactor >= maxFillFactor)
    rehashing();
```

0	
1	**6**
2	**2**
3	**3**
4	**8**

0	**10**
1	**11**
2	**2**
3	**3**
4	
5	**15**
6	**6**
7	
8	**8**
9	**19**

0	**20**
1	
2	**2**
3	**3**
4	**22**
5	**25**
6	**6**
7	
8	**8**
9	
10	**10**
11	**11**
12	**30**
13	
14	
15	**15**
16	
17	
18	
19	**19**

Manejando colisiones (chaining)

Ejemplo:

Insertar los siguientes keys:

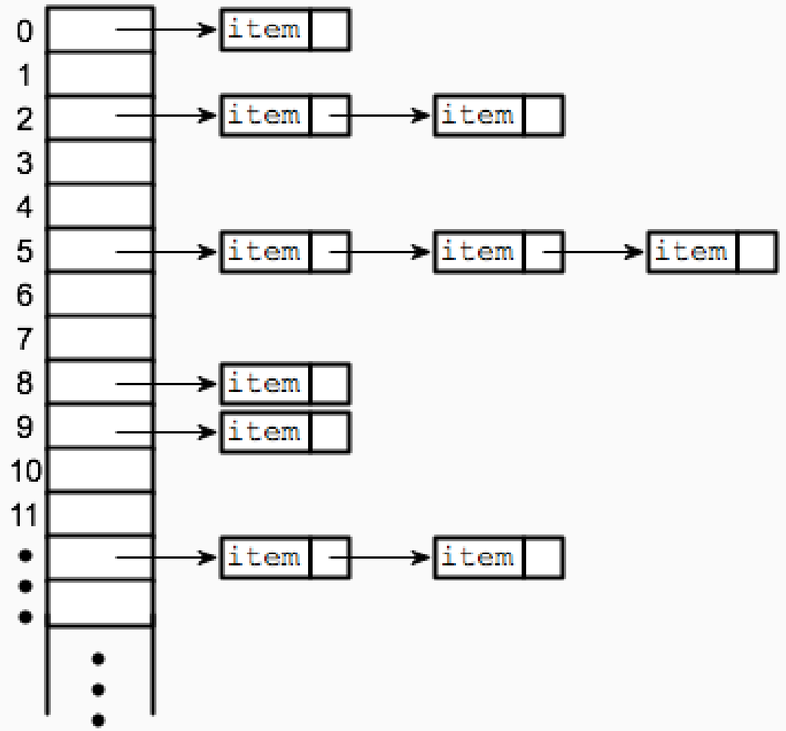
2, 3, 6, 8, 10, 11, 15, 19, 20, 22, 25, 30

Tamaño de la tabla hash: 5

maxFillFactor = 0,8

HashCode = Key

FillFactor = ¿ ***# of elements / hash table length ?***



Manejando colisiones (chaining)

Ejemplo:

Insertar los siguientes keys:

2, 3, 6, 8, 10, 11, 15, 19, 20, 22, 25, 30

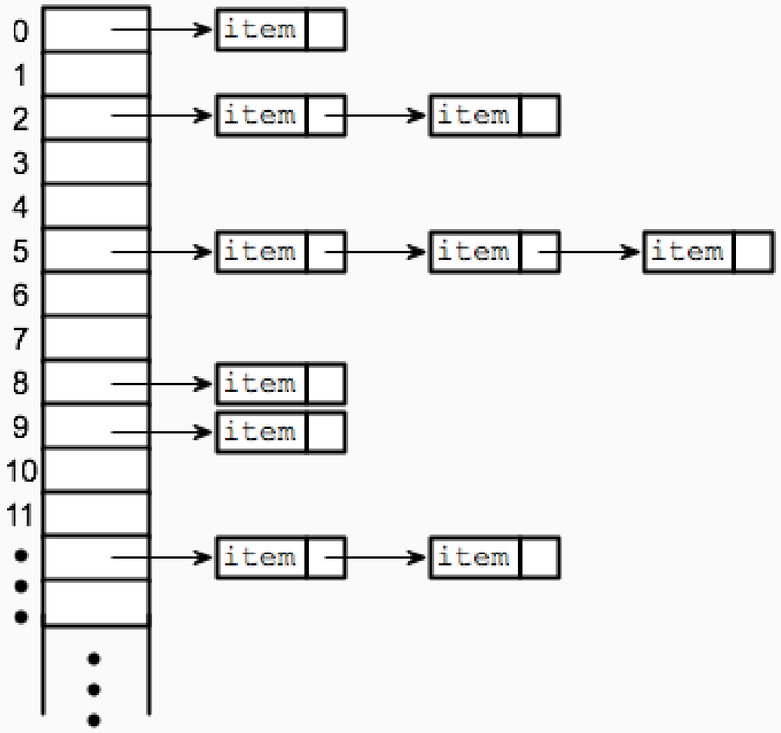
Tamaño de la tabla hash: 5

maxFillFactor = 0,5

HashCode = Key

maxColision (k) = 3

FillFactor = ***# of elements / (capacity * k)***



Manejando colisiones (chaining)

Ejemplo:

Insertar los siguientes keys:

2, 3, 6, 8, 10, 11, 15, 19, 20, 22, 25, 30

Tamaño de la tabla hash: 5

maxFillFactor = 0,5

HashCode = Key

maxColision (k) = 3

FillFactor = ***# of elements / (capacity * k)***

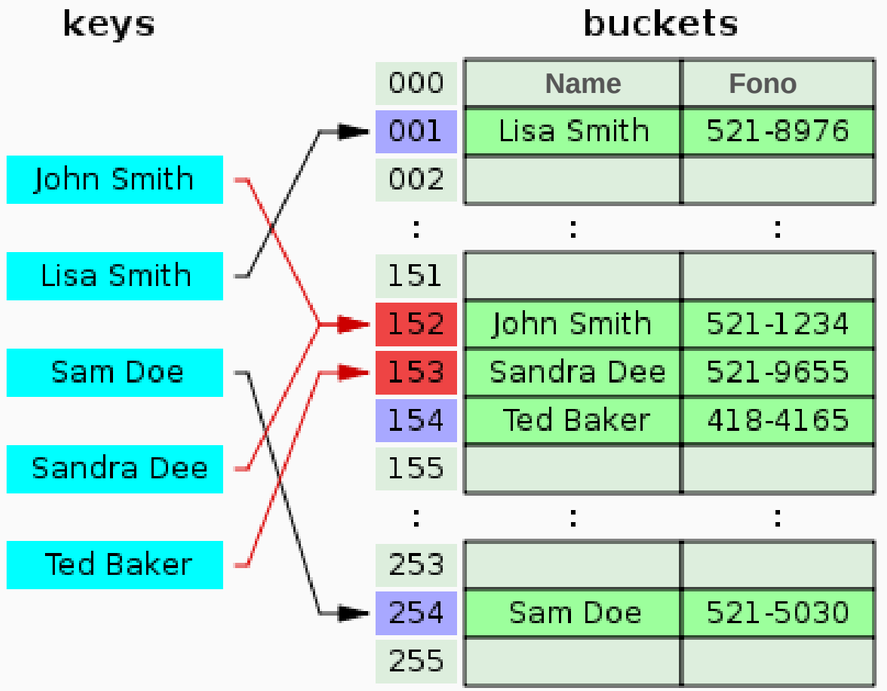
0	15,10
1	11, 6
2	2
3	8, 3
4	19

0	30,20,10
1	11
2	22,2
3	3
4	
5	25,15
6	6
7	
8	8
9	19

Encontrar elementos (open addressing)

Al buscar un elemento va a depender de como estemos manejando las colisiones:

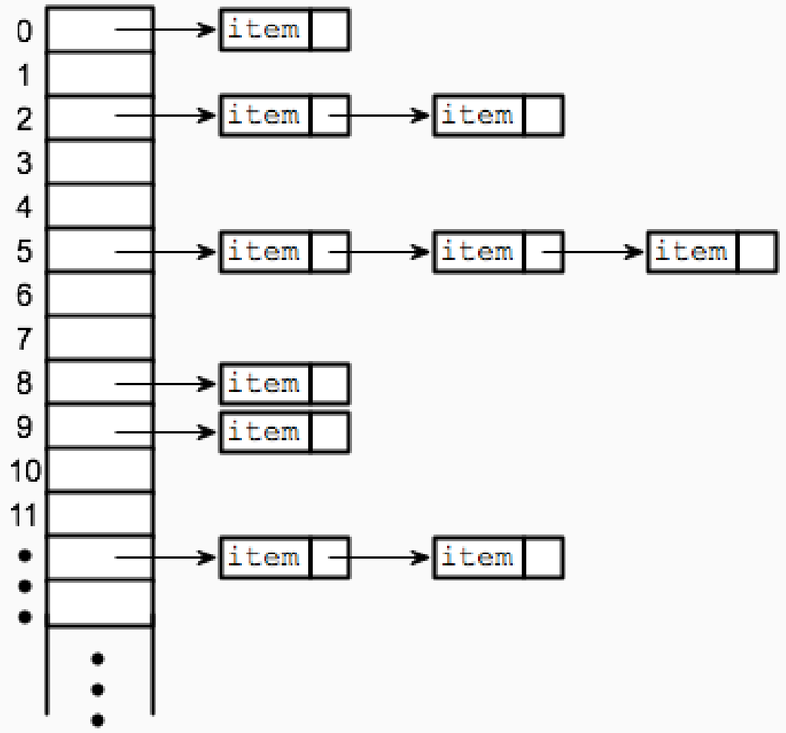
- Open addressing:
  a. Se obtiene el índice de la key
  b. Se verifica que no sea nulo
  c. Si hay colisión, se sigue avanzando hasta encontrar el elemento



Encontrar elementos (chaining)

Al buscar un elemento va a depender de como estemos manejando las colisiones:

- Chaining:
  a. Se obtiene el índice de la key
  b. Se busca en la lista de la posición



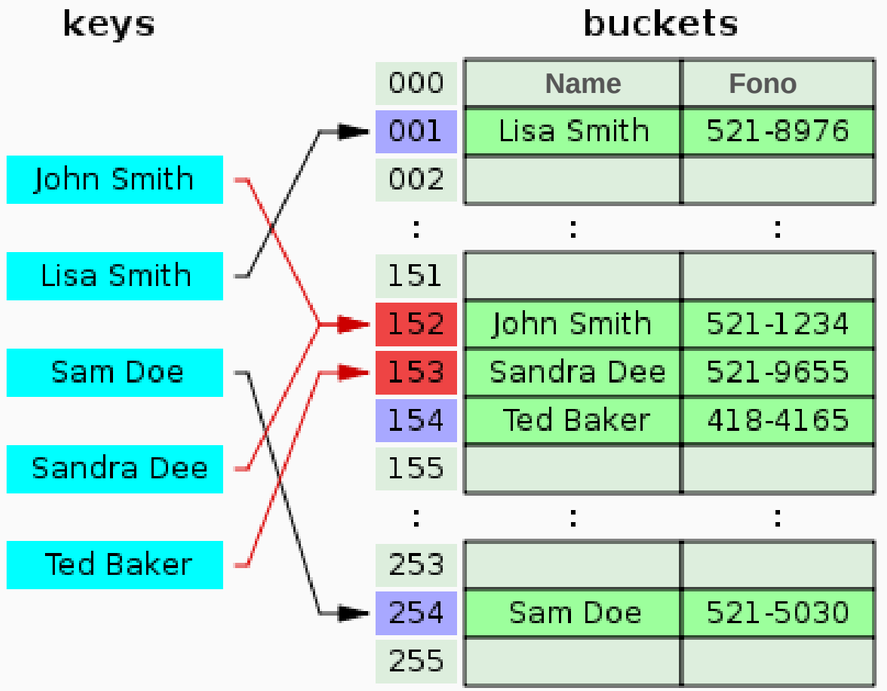
Remover elementos (open addressing)

El remover elementos va a depender de como estemos manejando las colisiones:

Open addressing:

1. Se obtiene el índice del elemento
2. Se verifica que no sea nulo en el array
3. Si las keys coinciden, remover el elemento
4. Si las keys no coinciden (probablemente haya una colisión), revisar el siguiente elemento

Es bastante difícil de mantener

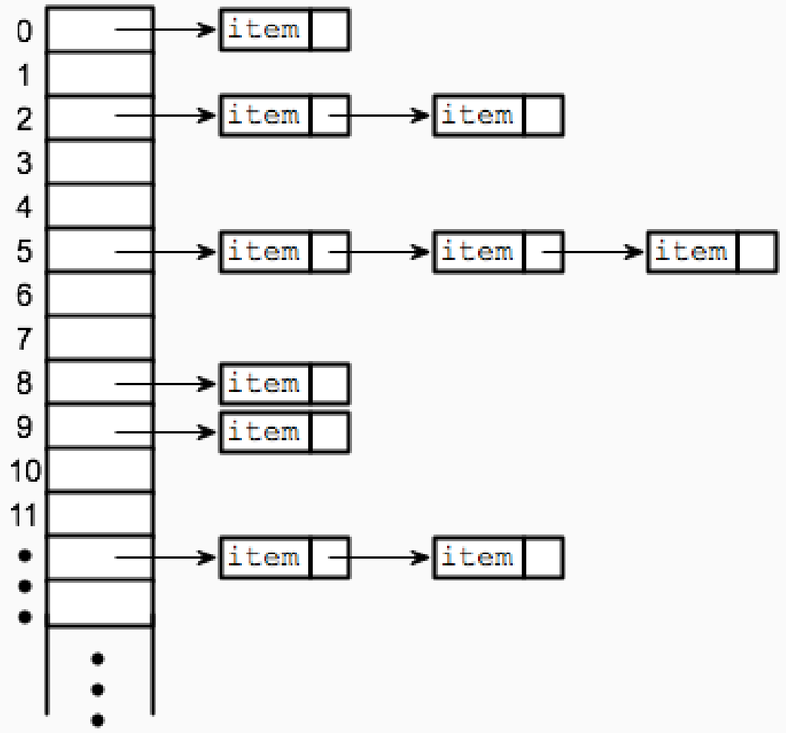


Remover elementos (chaining)

Chaining:

1. Se obtiene el índice del elemento
2. Se elimina el elemento de la lista simplemente enlazada

Es fácil de mantener e implementar



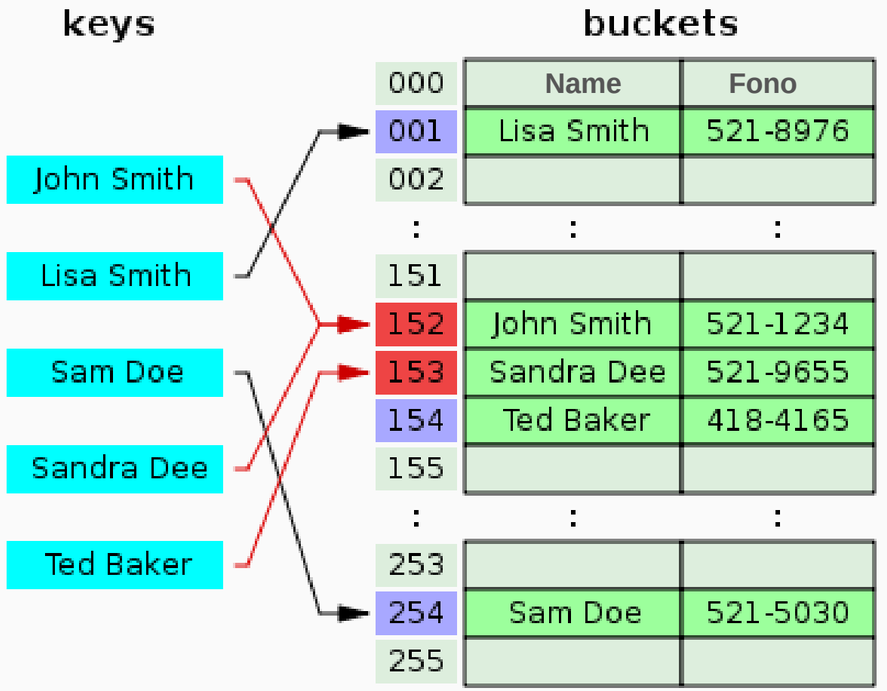
Enumerar elementos (open addressing)

Cuántas maneras de enumerar se deben tener?

Dos, una para las llaves y otra para los valores

- Open addressing:

```
foreach (item in array) {
        if (item != null ){
                ....
        }
}
```



Enumerar elementos (chaining)

Cuántas maneras de enumerar se deben tener?

Dos, una para las llaves y otra para los valores

- Chaining:

```
foreach(list in array) {
        foreach(item in list) {
                .....
        }
}
```

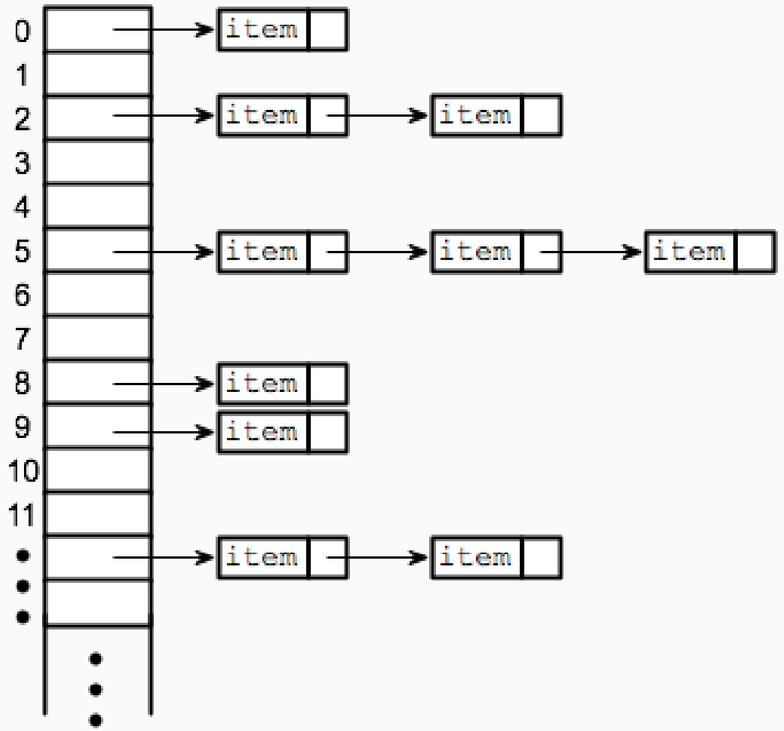
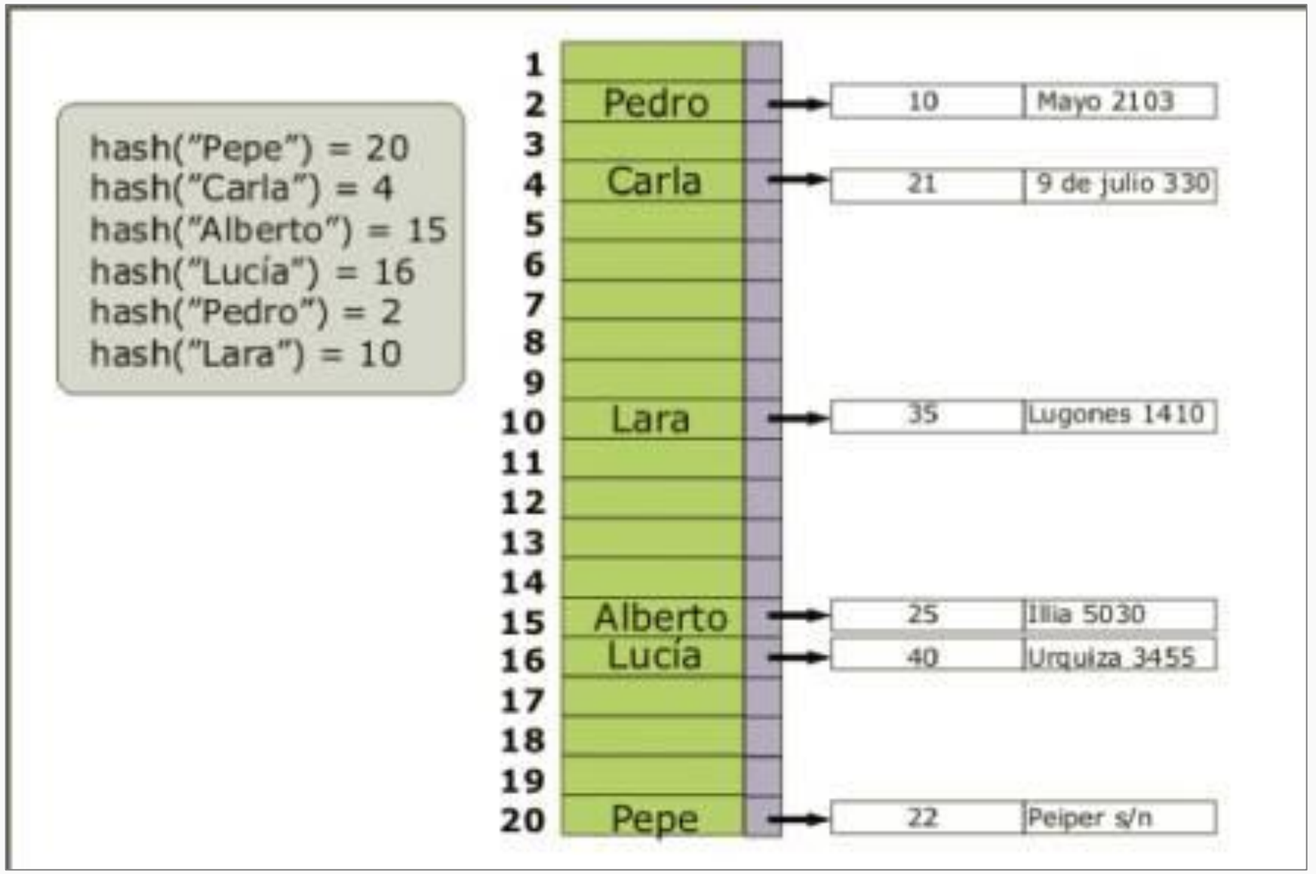


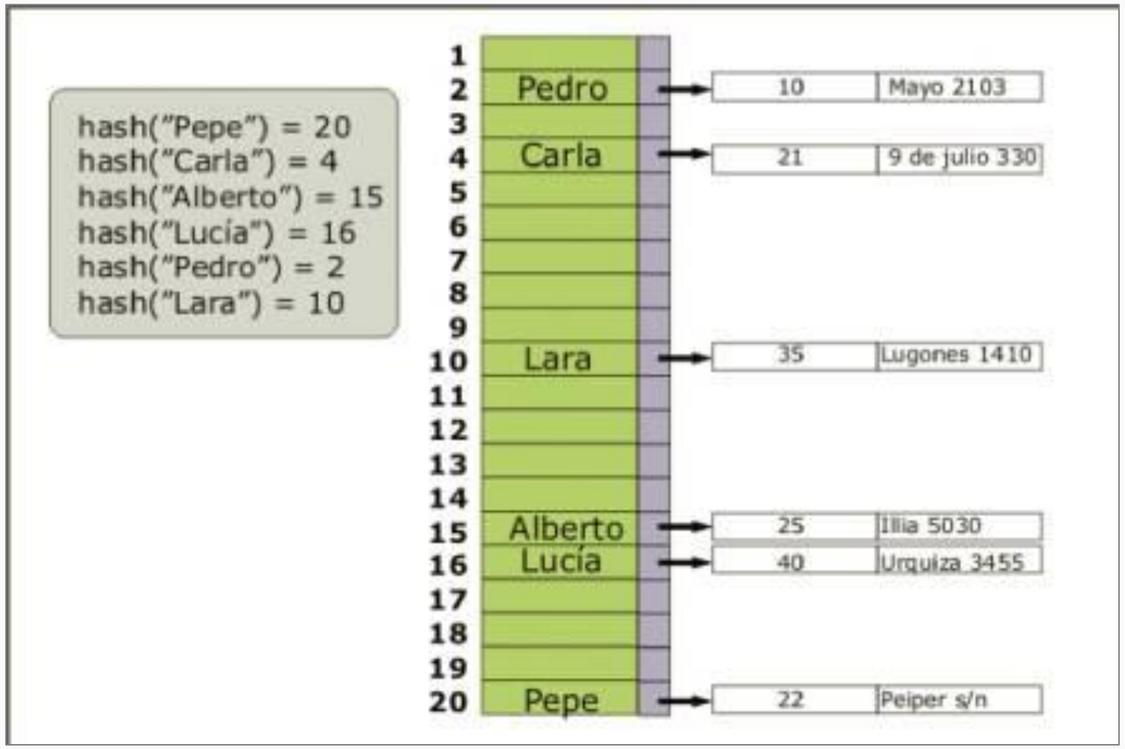
Tabla Hash para base de datos

<key : registro>



Hashing: problemas

¿Cómo reportar los elementos ordenados de manera eficiente?



Hashing (Complejidad Open Addressing)

Operación	Caso Mejor/promedio	Peor Caso
Inserción	Ω(1)	O(N)
Búsqueda	Ω(1)	O(N)
Eliminación	Ω(1)	O(N)
Rehashing	Ω(N)	O(N)
Traer Ordenado	Ω(N * log(N))	O(N * log(N))

N: total de elementos
K: promedio de elementos en la colision K<=N

Hashing (Complejidad Chaining)

Operación	Caso Mejor/promedio	Peor Caso
Inserción	Ω(1)	O(K)
Búsqueda	Ω(1)	O(K)
Eliminación	Ω(1)	O(K)
Rehashing	Ω(N)	O(N) * O(k)
Ordenar	Ω(N * log(N))	O(N * log(N))

N: total de elementos
K: promedio de elementos en la colision K<=N

Hashing (Complejidad)

¿Y si usamos un árbol binario de búsqueda balanceado?

```
Class NodeBTS{
        TK key;
        TV value;
        NodeBTS *left;
        NodeBTS* right;
};
```

Operación	Caso Mejor/promedio	Peor Caso
Inserción	Ω(log(N))	O(log(N))
Búsqueda	Ω(log(N))	O(log(N))
Eliminación	Ω(log(N))	O(log(N))
Rehashing	--	--
Ordenar	Ω(N)	O(N)

UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

Welcome to Algorithms and Data Structures! - CS2100