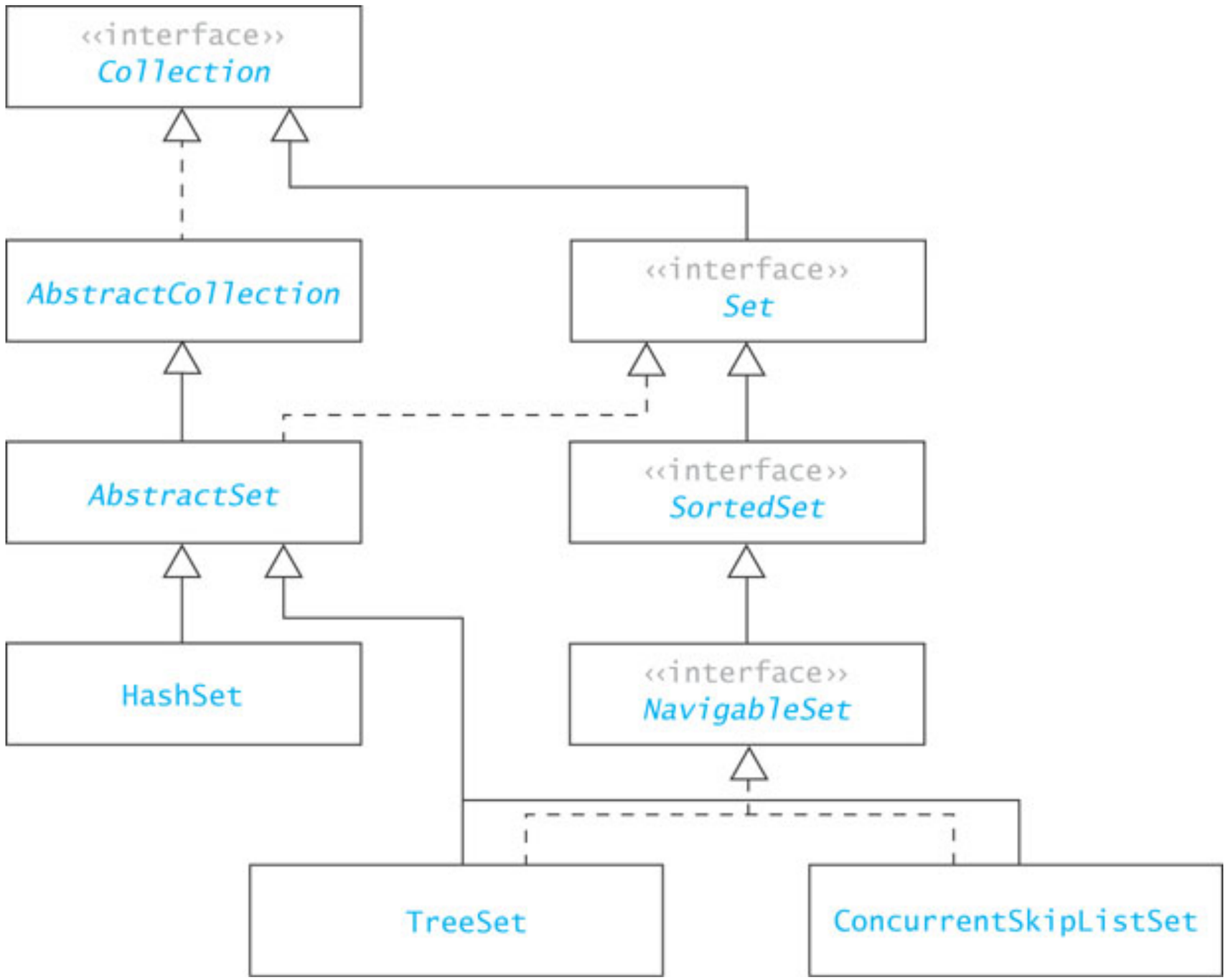
Avbildningar och hashtabeller

Koffman & Wolfgang

- kapitel 7, mestadels avsnitt 2–4

Mängder i Java



Mängd-gränssnittet

Method	Behavior
`boolean add(E obj)`	Adds item `obj` to this set if it is not already present (optional operation) and returns **`true`**. Returns false if `obj` is already in the set.
`boolean addAll(Collection<E> coll)`	Adds all of the elements in collection `coll` to this set if they're not already present (optional operation). Returns **`true`** if the set is changed. Implements *set union* if `coll` is a `Set`.
`boolean contains(Object obj)`	Returns **`true`** if this set contains an element that is equal to `obj`. Implements a test for *set membership*.
`boolean containsAll(Collection<E> coll)`	Returns **`true`** if this set contains all of the elements of collection `coll`. If `coll` is a set, returns **`true`** if this set is a subset of `coll`.
`boolean isEmpty()`	Returns **`true`** if this set contains no elements.
`Iterator<E> iterator()`	Returns an iterator over the elements in this set.
`boolean remove(Object obj)`	Removes the set element equal to `obj` if it is present (optional operation). Returns **`true`** if the object was removed.
`boolean removeAll(Collection<E> coll)`	Removes from this set all of its elements that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set difference* operation.
`boolean retainAll(Collection<E> coll)`	Retains only the elements in this set that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set intersection* operation.
`int size()`	Returns the number of elements in this set (its cardinality).

Set<E> vs. List<E>

Mängder får endast innehålla unika element:

- Metoden .add(E) returnerar false om elementet redan finns

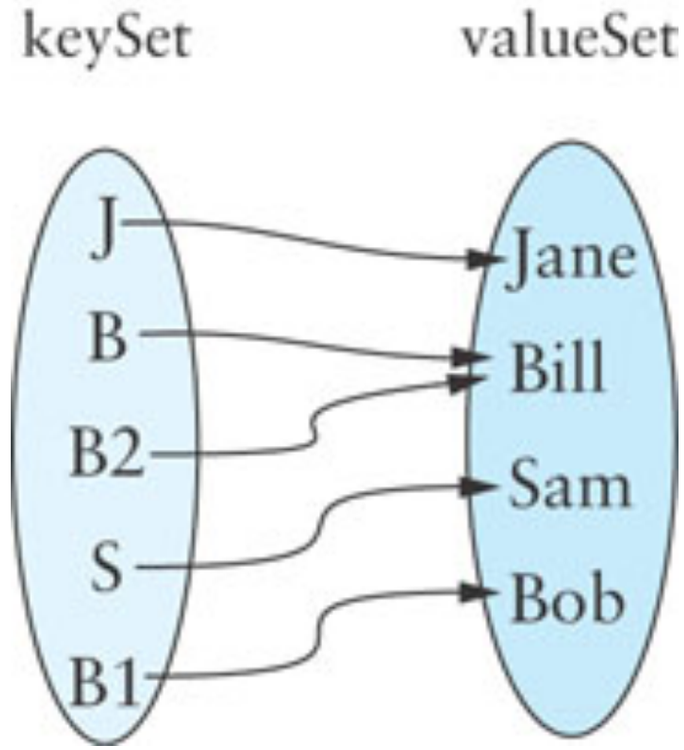
Mängder är oordnade:

- Mängder har ingen metod .get(int)
- Du kan iterera genom elementen i en mängd, men ordningen är godtycklig

Avbildningar, Map<K,V>

En avbildning är ekvivalent
med en mängd nyckel-värde-par:

- nycklar måste vara unika
- värden kan förekomma flera gånger



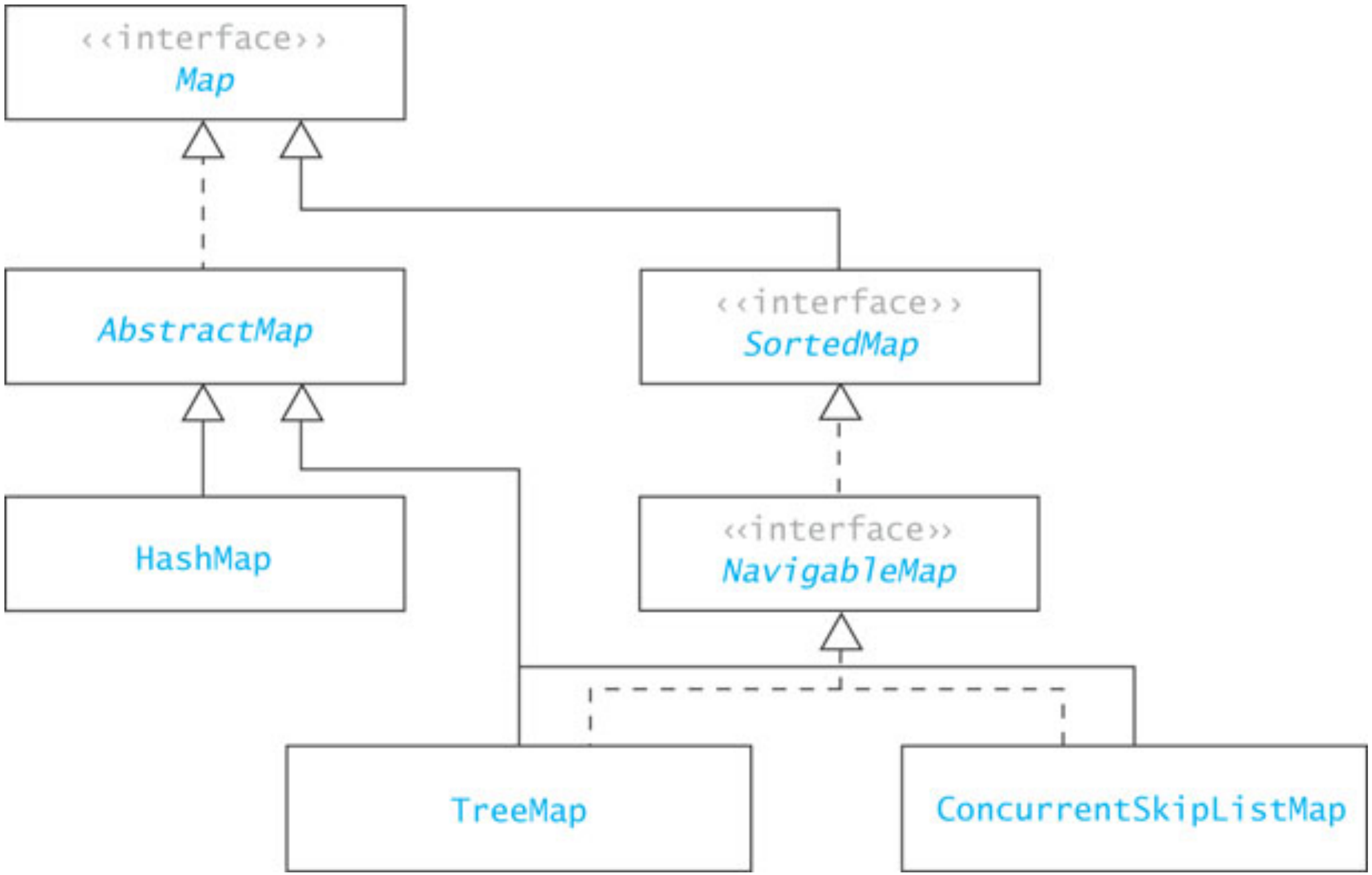
Map-gränssnittets viktigaste metoder:

```
interface Map<K,V> {
      public V get(Object key)
      public V put(K key, V value)
}
```

En mängd kan definieras som en avbildning:

- Set<E> = Map<E, –>
- där – är något ointressant, eftersom mängder bara bryr sig om nycklarna

Avbildningar i Java



Gränssnittet Map<K,V>

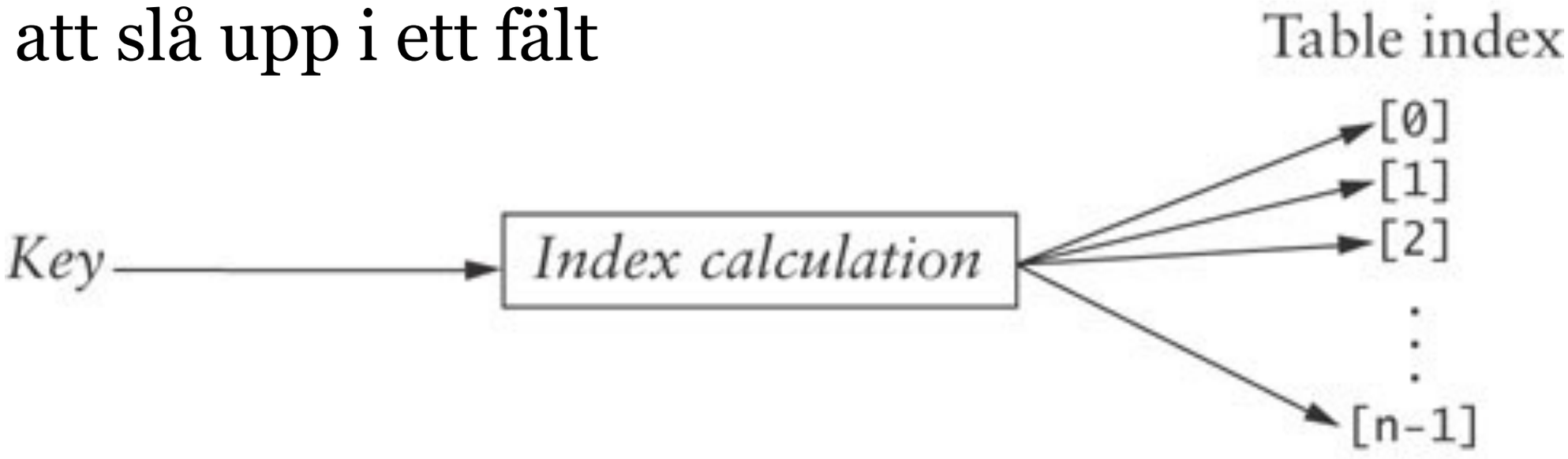
Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this map contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`int size()`	Returns the number of `key-value` mappings in this map.

Hashtabeller

En hashtabell låter oss slå upp ett värde efter en nyckel istället för en position

Om vi definierar hashtabellen korrekt så blir uppslagning, insättning och borttagning konstanta, O(1)

Grundidén är att transformera nyckeln till ett heltal (dess hashkod), som sedan används för att slå upp i ett fält



Hashkoder och tabellindex

Antag att vi ska lagra Huffmankoderna från kapitel 6.

Om texten bara innehåller ASCII-värden (de först 128 Unicode-tecknen), så kan vi använda ett fält med storlek 128 och låta ASCII-koden vara positionen.

- men, om vi vill kunna använda alla 1 114 112 Unicode-tecken, hur gör vi då?
- vi kanske kan använda en tabell med 200 tecken och beräkna index såhär:

```
int index = unicode % 200
```

Hashkoder och tabellindex

Antag att vi ska lagra Huffmankoderna från kapitel 6.

Om texten bara innehåller ASCII-värden (de först 128 Unicode-tecknen), så kan vi använda ett fält med storlek 128 och låta ASCII-koden vara positionen.

- men, om vi vill kunna använda alla 1 114 112 Unicode-tecken, hur gör vi då?
- vi kanske kan använda en tabell med 200 tecken och beräkna index såhär:

```
int index = unicode % 200
```

...	...
65	A, 8
66	B, 2
67	C, 3
68	D, 4
69	E, 12
70	F, 2
71	G, 2
72	H, 6
73	I, 7
74	J, 1
75	K, 2
...	...

Hashkoder och kollisioner

Säg att vi använder en tabell med 200 tecken och beräknar index:

```
int index = unicode % 200
```

Hur kommer då följande text att kodas?

- . . . *mañana (tomorrow), I'll finish my program*. . .

Givet följande Unicode-värden:

Hexadecimal	Decimal	Character	Name
0x0029	41	)	right parenthesis
0x00F1	241	ñ	small letter n with tilde

så blir indexen för bokstäverna ’ñ’ och ’)’ bägge 41

- då får vi en kollision
- för att kunna implementera hashtabeller måste vi hantera kollisioner

Hur man genererar hashkoder

In de flesta tillämpningar kommer en nyckel att vara en sträng, eller något ännu mer avancerat

- antalet möjliga nycklar är mycket mycket fler än tabellstorleken
- vi behöver kunna generera hashkoder som ger så få kollisioner som möjligt
- målet är att få en jämn fördelning mellan alla tabellceller

Naiva algoritmer kan ge många kollisioner:

- att summera Unicode-värdena för varje tecken i en sträng ger samma hashkod för ”stram” och ”smart”

Java-metoden .hashCode()

String.hashCode() använder följande formel:

- $s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-1}$
- där s_k är Unicode-värdet för tecken nr k i strängen, och n är strängens längd
- hashkoden är ett 32-bitars heltal (int), så den trunkeras automatiskt
- 31 är ett primtal, och primtal genererar relativt få kollisioner

För att få index i tabellen så tar vi bara resten:

```
index = key.hashCode() % table.length
```

Öppen adressering vs. kedjning

Det finns två huvudsätt att organisera hashtabeller.

I en hashtabell med öppen adressering används ”linear probing” för att hämta element:

- om tabellcellen för en viss nyckel redan är upptagen med en annan nyckel, så flyttar vi oss till nästa tabellcell
- vi fortsätter flytta oss vidare ända tills vi hittar rätt nyckel eller en tom cell
- detta förutsätter att tabellen aldrig blir full

I en hashtabell med ”kedjning” (chaining) pekar varje tabellcell på en länkad lista med nyckel-värde-par:

- om tre nycklar ger samma hashkod, så har den motsvarande länkade listan tre element
- för att söka efter en viss nyckel så får man gå igenom den länkade listan ett element i taget

Öppen adressering

Algorithm for Accessing an Item in a Hash Table

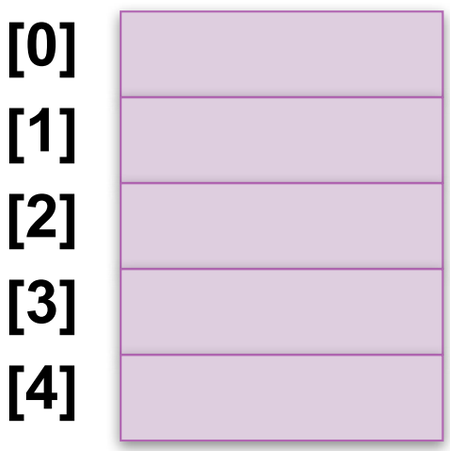
1. Compute the index by taking the item's `hashCode() % table.length`.
2. `if table[index]` is `null`
3. The item is not in the table.
4. `else if table[index]` is equal to the item
5. The item is in the table.

   `else`
6. Continue to search the table by incrementing the index until either the item is found or a `null` entry is found.

- när tabellindex blir för stort så måste det börja om från noll, som i ett cirkulärt fält
- det kan leda till en oändlig loop
- vi måste alltså se till att tabellen aldrig blir helt full, genom att öka storleken när *fyllnadsgraden* (load factor) överstiger ett gränsvärde (t.ex. 75%)

Exempel: Insättning

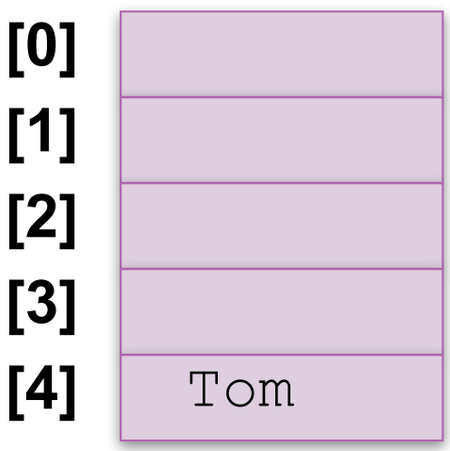
```
Tom  Dick  Harry  Sam  Pete
```



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

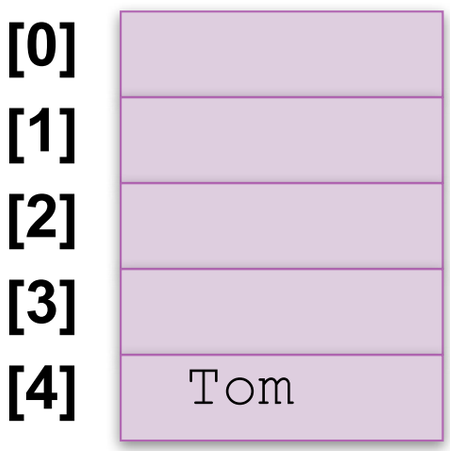
Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

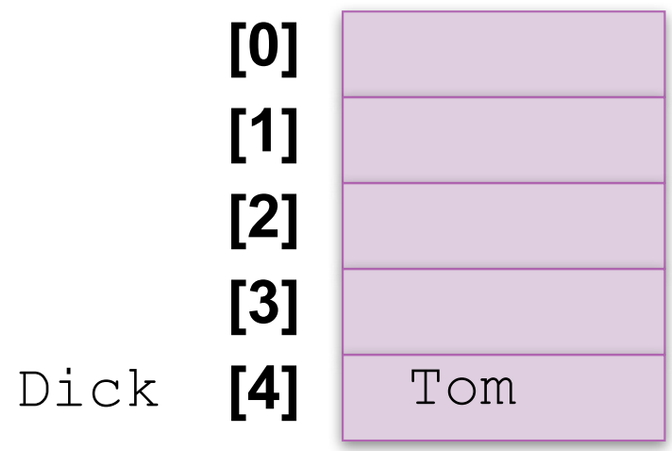
Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

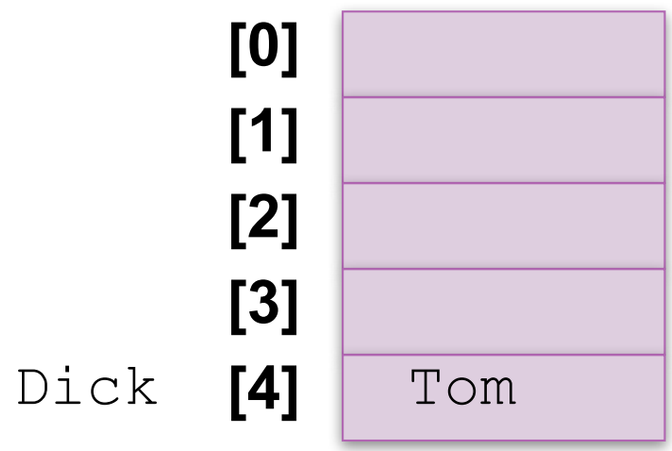
Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

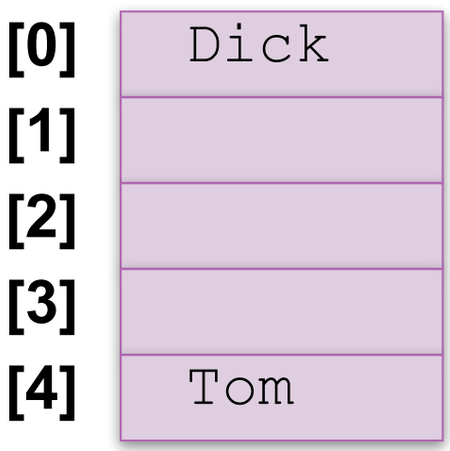
Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

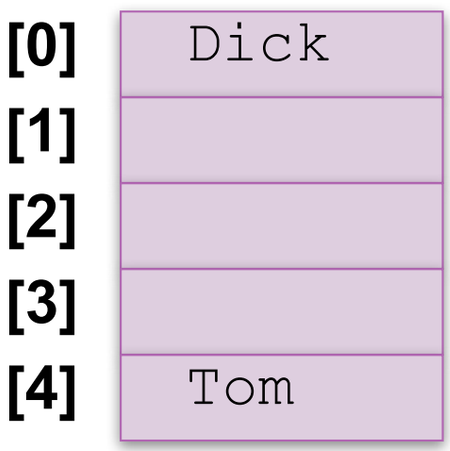
Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Sam Pete

[0]	Dick
[1]	
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Sam Pete

[0]	Dick
[1]	
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Pete

	[0]	Dick
	[1]	
	[2]	
	[3]	Harry
Sam	**[4]**	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Pete

	[0]	Dick
	[1]	
	[2]	
	[3]	Harry
Sam	[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Pete

Sam

[0]	Dick
[1]	
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Pete

Sam

[0]	Dick
[1]	
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Pete

[0]	Dick
[1]	Sam
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

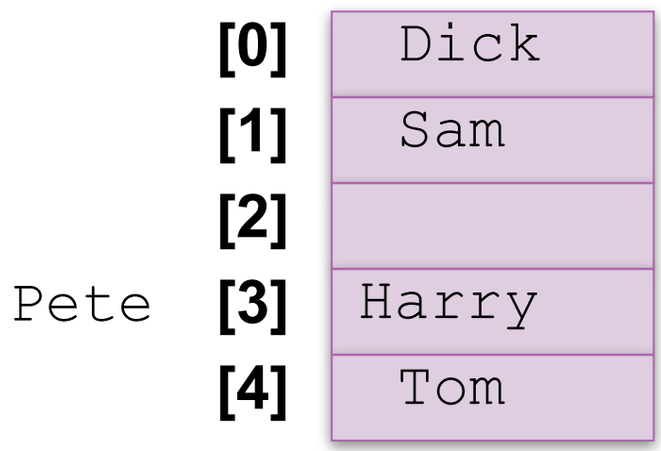
Pete

[0]	Dick
[1]	Sam
[2]	
[3]	Harry
[4]	Tom

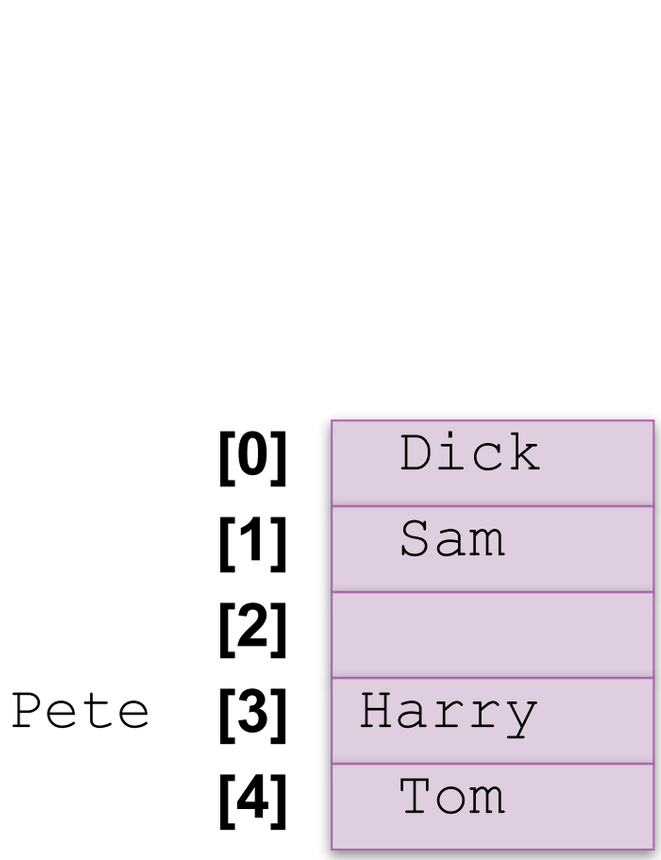
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



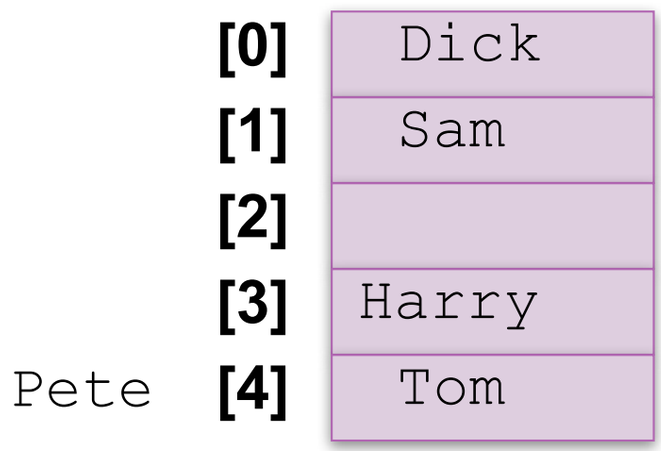
Exempel: Insättning



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

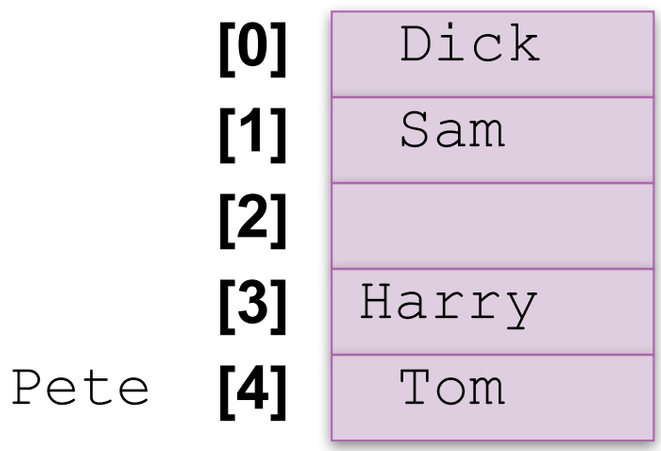
Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

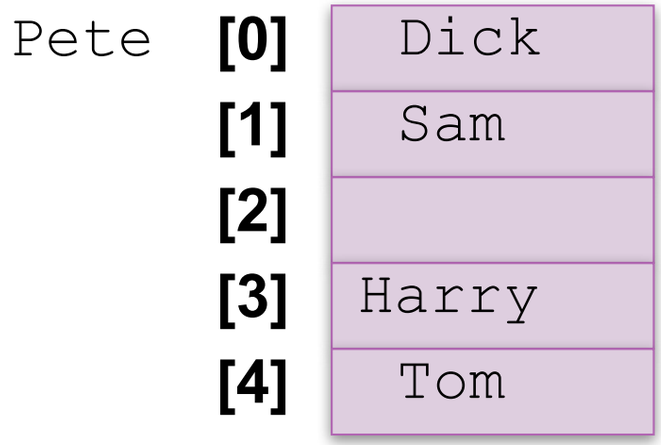


Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

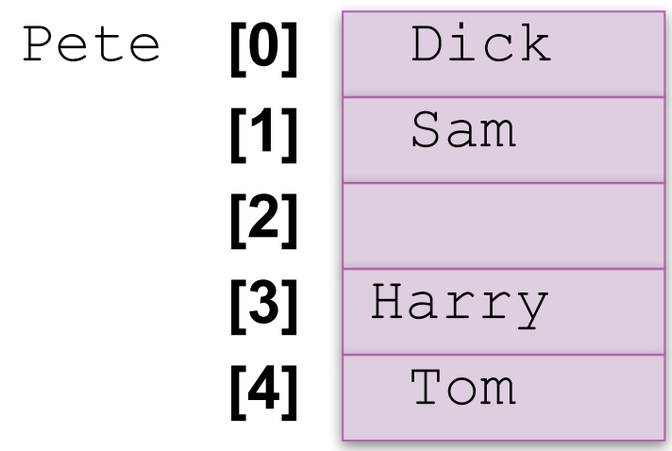


Exempel: Insättning



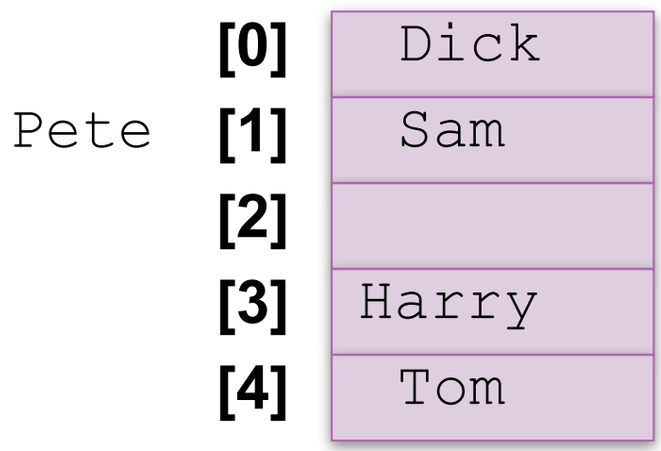
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



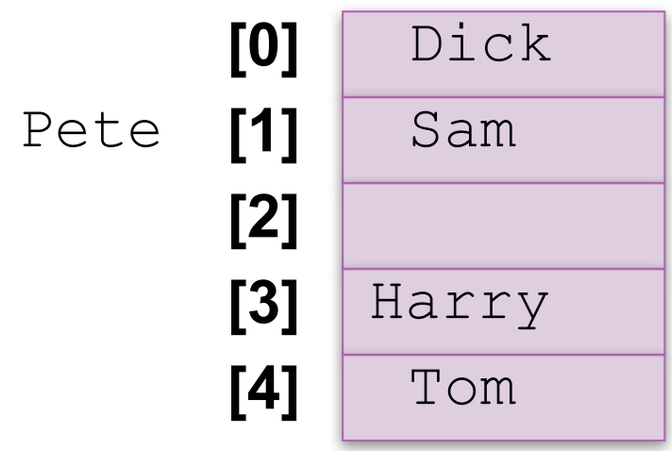
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Uppslagning av ”Tom” och ”Harry” går direkt; för de övriga krävs en linjär sökning

Exempel: Utöka storleken

Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7

Exempel: Utöka storleken

Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7

Det bästa sättet att minimera antalet kollisioner (och reducera söktiden) är att öka tabellstorleken

Traversera en hashtabell

Du kan inte traversera en hashtabell på ett meningsfullt sätt, eftersom hashkoderna är godtyckliga

[0]	Dick
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

Dick, Sam, Pete, Harry, Tom

[0]	
[1]	
[2]	
[3]	Tom
[4]	
[5]	Dick
[6]	Sam
[7]	Pete
[8]	
[9]	
[10]	Harry

Tom, Dick, Sam, Pete, Harry

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)

[0]	Dick
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)

[0]	
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

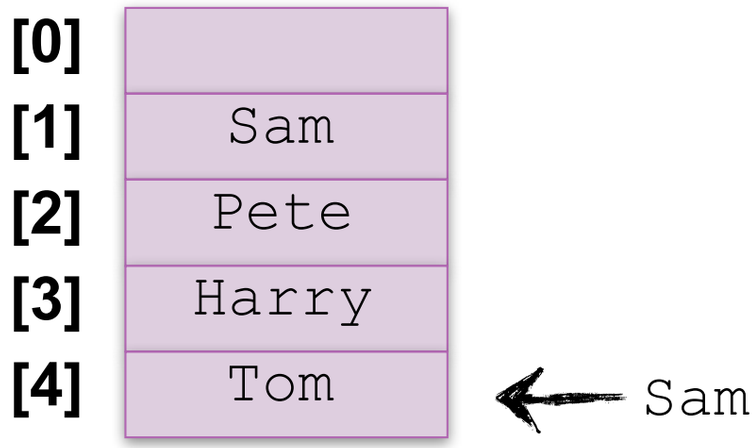
- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

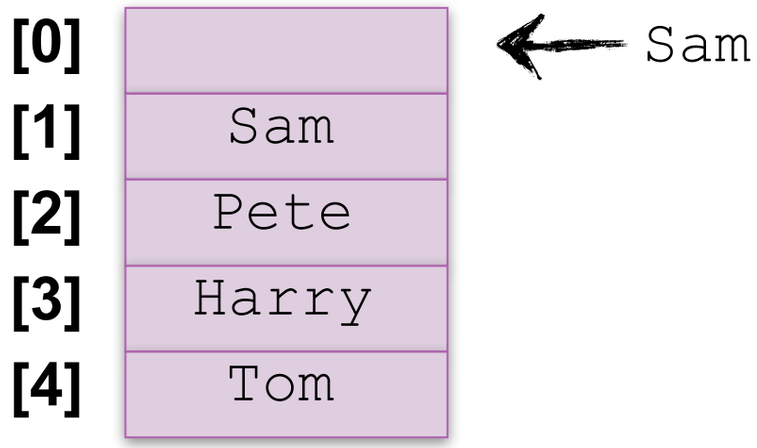
- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

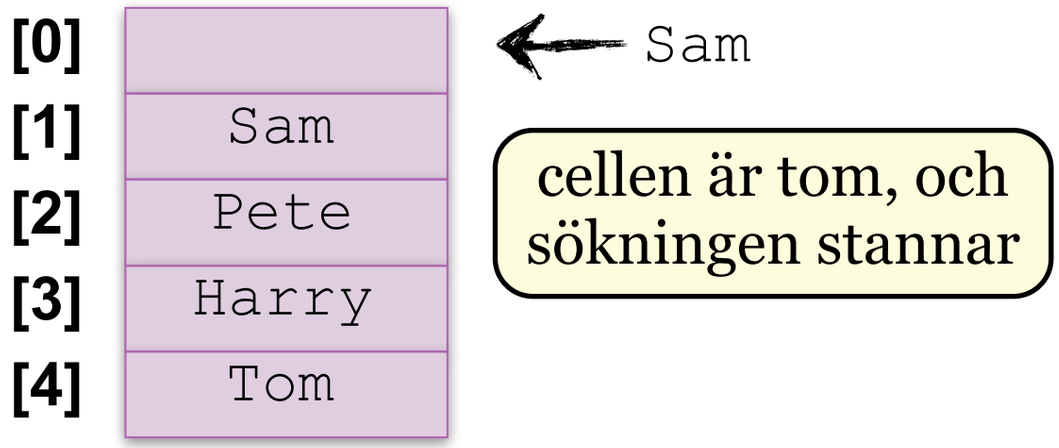
- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)

[0]	Dick
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)

[0]	**DELETED**
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

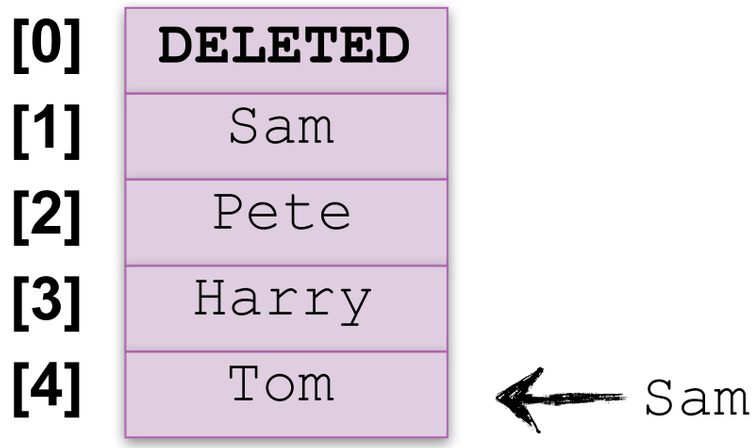
- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

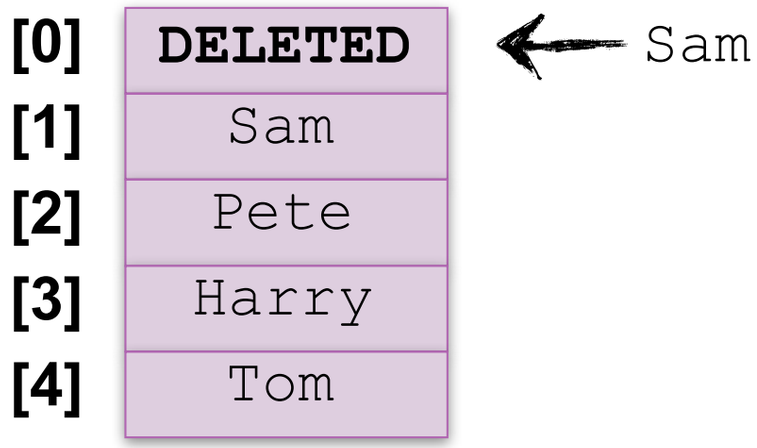
- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Att ta bort ett element

När man tar bort ett element, så kan man *inte* sätta dess tabellcell till null:

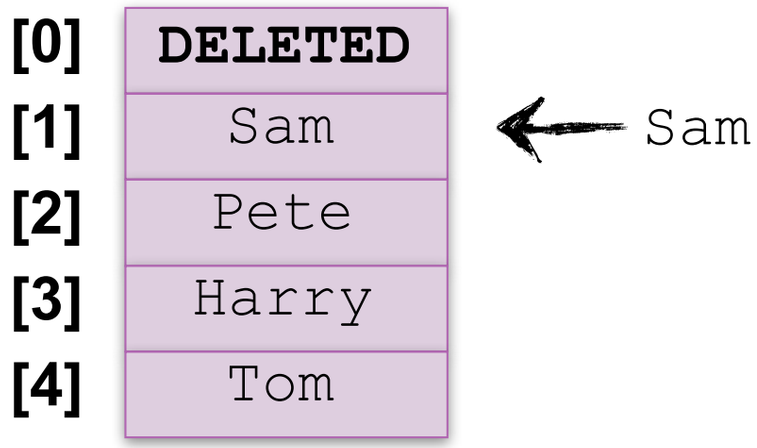
- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar *inte* fyllnadsgraden (load factor)

Exempel:

- remove(”Dick”), följt av get(”Sam”)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Förstora tabellen

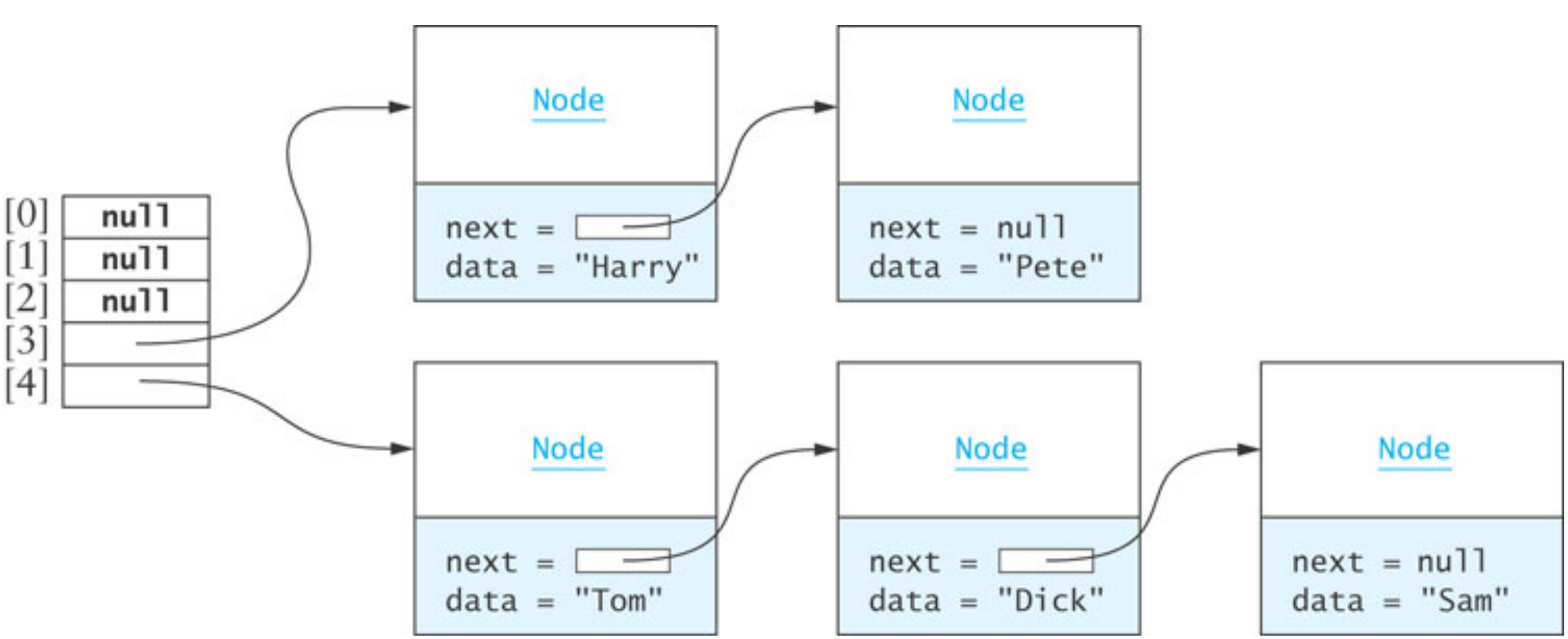
Ju fullare tabellen är, desto fler kollisioner.
När tabellen är tillräckligt full så skapar vi en ny, dubbelt så stor tabell.

- ungefär som vi gjorde för ArrayList
- det går inte att direktkopiera värdena från den gamla tabellen, eftersom hashkoderna kan ändras radikalt
- vi måste gå igenom alla värden i tabellen, ett efter ett, och beräkna en ny hashkod
- eftersom de borttagna inte stoppas in på nytt så sparar vi lite minne

Kedjning (chaining)

Kedjning är ett alternativ till öppen adressering.

Varje tabellcell pekar på en länkad lista som innehåller alla element med samma hashkod.



Fördelar/nackdelar

Fördelar med kedjning:

- man kan lagra fler element i hashtabellen
- man behöver bara leta igenom de nycklar som har samma hashkod
  - vid öppen adressering kan nycklar med olika hashkod ligga omlott i tabellen
- det är enklare att lägga till och ta bort element

Nackdelar med kedjning:

- tar mer plats, p.g.a. den länkade listan
- vi måste fortfarande omallokera tabellen då och då
  - detta för att bibehålla konstant komplexitet för uppslagning, insättning och borttagning

Öppen adressering vs. kedjning

fyllningsgrad (load factor)	antal jämförelser (öppen adressering)	antal jämförelser (kedjning)
0 %	1,00	1,00
25 %	1,17	1,13
50 %	1,50	1,25
75 %	2,50	1,38
85 %	3,83	1,43
90 %	5,50	1,45
95 %	10,50	1,48
100 %	—	1,50
200 %	—	2,00
300 %	—	2,50

Öppen adressering vs. kedjning

fyllningsgrad (load factor)	antal jämförelser (öppen adressering)	antal jämförelser (kedjning)
0 %	1,00	1,00
25 %	1,17	1,13
50 %	1,50	1,25
75 %	2,50	1,38
85 %	3,83	1,43
90 %	5,50	1,45
95 %	10,50	1,48
100 %	—	1,50
200 %	—	2,00
300 %	—	2,50

å andra sidan så kräver kedjning med 300% fyllningsgrad totalt sett mer minne än en större öppen adressering-tabell med 75% fyllnadsgrad

Effektivitet

Insättning, uppslagning och borttagning

- är konstanta, O(1), i *medeltal*
- under förutsättning att hashfunktionen är bra (dvs, ger en jämn fördelning av hashkoder)
- dessutom måste vi utöka själva tabellen när fyllnadsgraden blir för hög, även vid kedjning

Komplexiteten kan bli linjär, O(*n*), om

- vi har en dålig hashfunktion
- vi aldrig utökar den underliggande tabellen

Implementation, gränssnitt

interface KWHashMap<K,V> (en delmängd av Map<K,V>)

Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this table contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key`. Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this table if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping.
`int size()`	Returns the size of the table.

class Entry<K,V> (implementerar gränssnittet Map.Entry<K,V>)

Data Field	Attribute
`private K key`	The key.
`private V value`	The value.
Constructor	**Behavior**
`public Entry(K key, V value)`	Constructs an `Entry` with the given values.

Method	Behavior
`public K getKey()`	Retrieves the key.
`public V getValue()`	Retrieves the value.
`public V setValue(V val)`	Sets the value.

Implementation, öppen adr.

class HashtableOpen<K,V>
(implementerar gränssnittet KWHashMap<K,V>)

Data Field	Attribute
`private Entry<K, V>[] table`	The hash table array.
`private static final int START_CAPACITY`	The initial capacity.
`private double LOAD_THRESHOLD`	The maximum load factor.
`private int numKeys`	The number of keys in the table excluding keys that were deleted.
`private int numDeletes`	The number of deleted keys.
`private final Entry<K, V> DELETED`	A special object to indicate that an entry has been deleted.

Implementation, öppen adr.

class HashtableOpen<K,V>
(implementerar gränssnittet KWHashMap<K,V>)

Data Field	Attribute
`private Entry<K, V>[] table`	The hash table array.
`private static final int START_CAPACITY`	The initial capacity.
`private double LOAD_THRESHOLD`	The maximum load factor.
`private int numKeys`	The number of keys in the table excluding keys that were deleted.
`private int numDeletes`	The number of deleted keys.
`private final Entry<K, V> DELETED`	A special object to indicate that an entry has been deleted.

t.ex. 0.75

t.ex. 101 eller vad-som-helst

.find()

En privat hjälpmetod som letar upp positionen för en nyckel

Method	Behavior
`private int find(Object key)`	Returns the index of the specified `key` if present in the table; otherwise, returns the index of the first available slot.
`private void rehash()`	Doubles the capacity of the table and permanently removes deleted items.

private int find(Object key):

1. **set** index **to** key.hashCode() % table.length
2. **if** index < 0**, add** table.length
3. **while** table[index] **is not** null **and** key != **the key of** table[index]**:**
4. **increment** index
5. **if** index ≥ table.length**:**
6. **set** index **to** 0
7. **return** index

.get() och .remove()

public V get(Object key):

1. **set** index **to** find(key)
2. **if** key == **the key of** table[index]**:**
3. **return the value of** table[index]
4. **else:**
5. **return** null

public V remove(Object key)

1. **set** index **to** find(key)
2. **if** table[index] **is** null**:**
3. **return** null
4. **otherwise** key **was found:**
5. **set** table[index] **to the special entry DELETED**
6. **increment** numDeletes**, and decrement** numKeys
7. **return the old value of** table[index]

.put() och .rehash()

public V put(K key, V value):

1. **set** index **to** find(key)
2. **if** table[index] **is** null:
3. **set** table[index] **to a new** Entry(key, value)
4. **increment** numKeys
5. **if the load factor** > **the load threshold,** rehash()
6. **return** null
7. **otherwise** key **was found:**
8. **set the value of** table[index] **to** value
9. **return the old value of** table[index]

private void rehash()

1. **allocate a** new table **that is at least double the size of** table
2. **reset** numKeys **and** numDeletes **to** 0
3. **for each** entry **in** table **that is not DELETED:**
4. **reinsert** entry **into the** new table

Implementation, kedjning

class HashtableChain<K,V>
(implementerar gränssnittet KWHashMap<K,V>)

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.

Implementation, kedjning

class HashtableChain<K,V>
(implementerar gränssnittet KWHashMap<K,V>)

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.

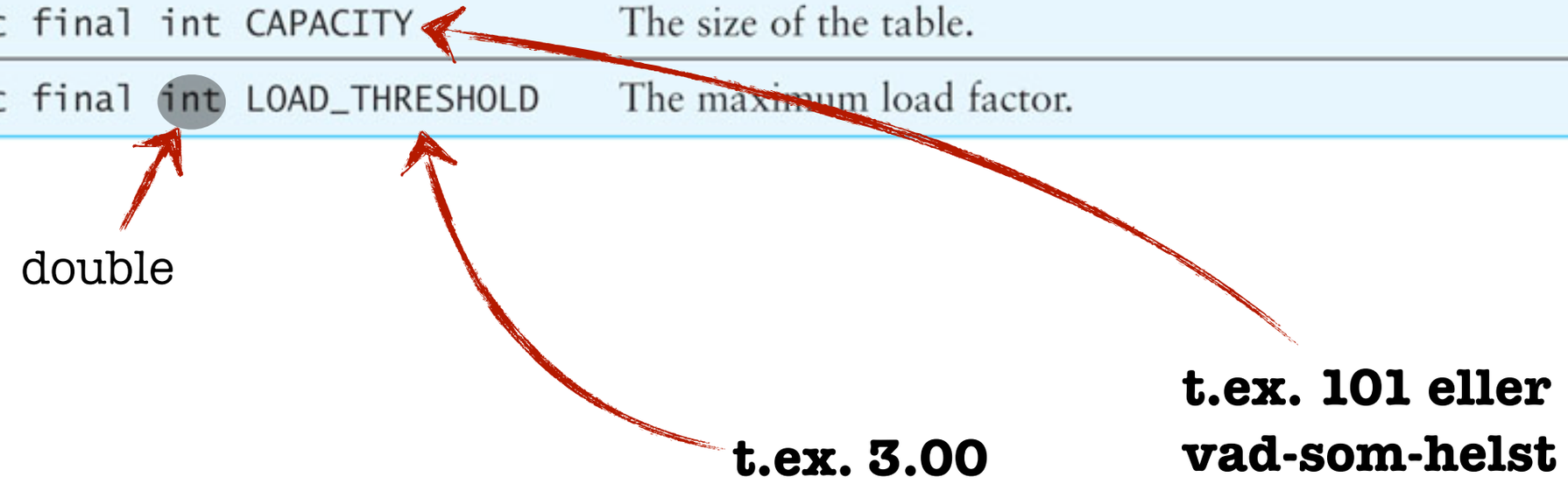
t.ex. 101 eller vad-som-helst

t.ex. 3.00

Implementation, kedjning

class HashtableChain<K,V>
(implementerar gränssnittet KWHashMap<K,V>)

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.



.get()

public V get(Object key)

1. **set** index **to** key.hashCode() % table.length
2. **if** index < 0**, add** table.length
3. **if** table[index] **is** null**:**
4. **return** null
5. **for each** entry **in the list at** table[index]**:**
6. **if** key == **the key of** entry**:**
7. **return the value of** entry
8. **return** null

.put()

public V put(K key, V value)

1. **set** index **to** key.hashCode() % table.length
2. **if** index < 0**, add** table.length
3. **if** table[index] **is** null**:**
4. **set** table[index] **to a new linked list**
5. **for each** entry **in the list at** table[index]**:**
6. **if** key == **the key of** entry**:**
7. **set the value of** entry **to** value
8. **return the old value of** entry
9. **otherwise** key **was not found:**
10. **insert a new** Entry(key, value) **into the list at** table[index]
11. **increment** numKeys
12. **if the load factor** > **the load threshold,** rehash()
13. **return** null

.remove()

public V remove(Object key)

1. **set** index **to** key.hashCode() % table.length
2. **if** index < 0**, add** table.length
3. **if** table[index] **is** null**:**
4. **return** null
5. **for each** entry **in the list at** table[index]**:**
6. **if** key == **the key of** entry**:**
7. **remove** entry **from the list**
8. **if the list is empty:**
9. **set** table[index] **to** null
10. **decrement** numKeys
11. **return the value of** entry
12. **otherwise** key **is not in the table:**
13. **return** null