Föreläsning 9

- Hashtabell

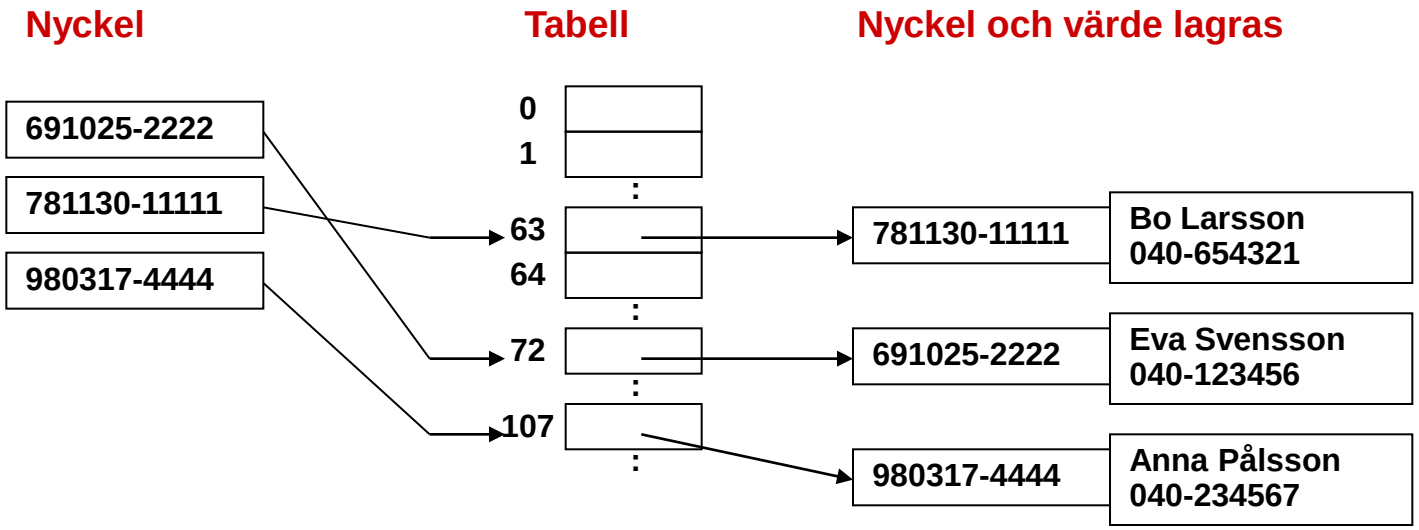
Skansholm: s 664-666, 670-673

Wikipedia: http://en.wikipedia.org/wiki/Hash_table

In computer science, a hash table or hash map is a data structure that uses a hash function to efficiently map certain identifiers or keys (e.g., person names) to associated values (e.g., their telephone numbers). The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought.

Hashtabell

- En hashtabell bygger på en **array**.
- Oftast lagrar man **par av objekt (nyckel, värde)**. Med hjälp av nyckeln kan man effektivt söka efter värdet i tabellen.
  I exemplet nedan används paret **(personnummer , namn + hemtelefon)**.
- Varje objekt som ska lagras i tabellen får en position i tabellen genom en hash-funktion. Hash-funktionen räknar ut positionen med hjälp av nyckeln.



HashEx.java

Hash-funktionen

Hash-funktionen ska uppfylla följande egenskaper:

- Den ska exekveras snabbt
- Den ska sprida elementen (positionerna) i hela tabellen

I java används metoderna **hashCode** respektive **equals** vid användning av Hash-tabeller.

- hashCode för att beräkna en position i en tabell
- equals för att jämföra nycklar

Element för vilka equals-metoden returnerar true ska ge samma returvärde vid anrop till metoden hashCode.

I klassen **Object** returnerar equals-metoden true om två objektreferenser är identiska, dvs. om ett objekt jämförs med sig själv. Därför beräknas hashCode-funktionens returvärde med hjälp av objektreferensen (objektets adress i minnet).

En följd av detta är att objekt kan ha olika hashCode vid olika programkörningar.

En annan följd är att två objekt med samma värde i instansvariablerna inte ger true vid anrop till equals och inte heller samma hashCode.

Hash-funktionen

Ofta vill man att equals-metoden ska jämföra en eller flera instansvariabler i objekten som jämförs. Detta är av speciellt intresse för klassen **String** eftersom en nyckel ofta är av typen String..

Klassen **String** överskuggar metoderna **hashCode** och **equals**.

- **equals-metoden** returnerar true om två strängar innehåller samma sekvens med tecken
- **hashCode-metoden** returnerar samma värde för strängar som innehåller samma sekvens med tecken. Hashmetoden ser ungefär ut så här:

```
private char[] value;
:
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < value.length; i++) {
        hash = 31*hash + value[i];
    }
    return hash;
}
```

Vissa andra klasser, t.ex. **Integer**, **Long** och **Double** överskuggar också metoderna **equals** och **hashCode** för att fungera väl i hash-tabeller.

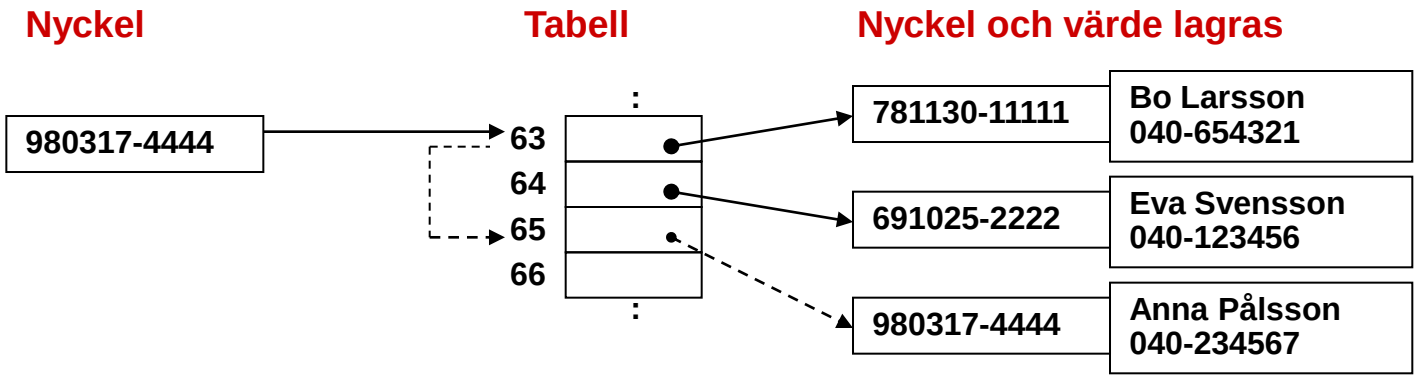
Chars.java

Kollisioner i tabellen

Det inträffar då och då att två olika nycklar ger samma position i tabellen. Det finns flera strategier för att lösa detta problem:

- Söka en annan position som är ledig. Denna strategi kallas för **sluten hashing**.

  I exemplet nedan ger nyckeln ”980317-4444” positionen 63. Men position 63 är upptagen, där är redan information lagrad. Nästa lediga position är 65. En tänkbar lösning är att lagra det nya objektet i position 65.

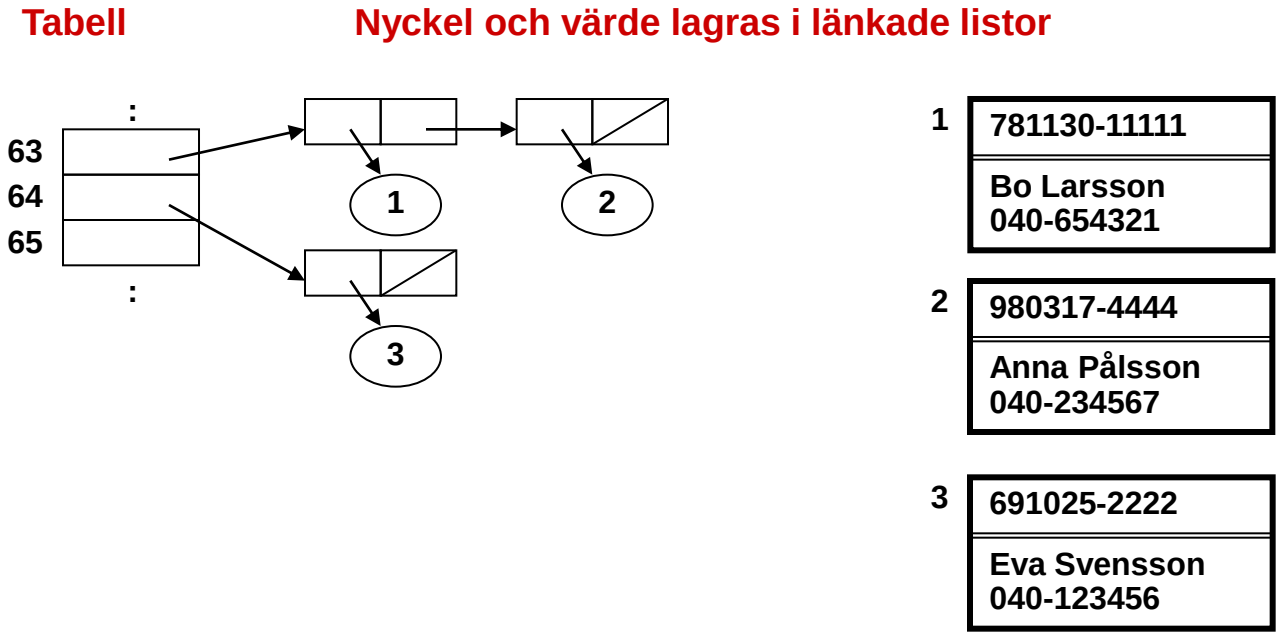


Kollisioner i tabellen

Det inträffar då och då att två olika nycklar ger samma position i tabellen. Det finns flera strategier för att lösa detta problem:

- Använda en struktur i varje position i arrayen som medger lagring av fler än ett element, t.ex. en länkad lista. Denna strategi kallas för **öppen hashing**.

  I exemplet nedan ger nycklarna ”781130-1111” och ”980317-4444” samma position. De lagras i en länkad lista. Även fler element kan lagras i position 63.



Ett exempel på sluten hashing

Algoritm för att sätta in ett element, put(Object key, Object value):

Om element med samma nyckel inte finns i tabellen
    Beräkna position i tabellen med hjälp av anrop till key.hashCode()
    Om positionen är upptagen så
        sök efter nästa lediga position i arrayen. Om du når slutet i arrayen så fortsätt sökningen i position 0.
    Placera det nya paret <key, value> i den tomma positionen

Algoritm för att söka ett element, get(Object key):

Beräkna position i tabellen med hjälp av anrop till key.hashCode()
Om positionen är upptagen och nyckeln ej är samma / positionen removed så
    sök vidare i arrayen tills du finner en tom position / sökt nyckel
Om den funna positionen har samma nyckel så
    returnera värdet
Annars
    returnera null

Algoritm för att ta bort ett element, remove(Object key):

Beräkna position i tabellen med hjälp av anrop till key.hashCode()
Sök med start i postionen ett element med samma nyckell eller som är tomt
Om den funna positionen inte är tom / listan slut så
    ta bort paret (key, value)

HashtableCH.java

Ett exempel på sluten hashing

Att medge att element tas ut hash-tabellen innebär vissa bekymmer. Vid tidigare anrop till put-metoden är det ju fullt möjligt att det borttagna elementet är ett som passerats vid sökandet efter en tom position.

För att put / remove / get ska fungera på avsett sätt måste varje position i arrayen ha ett av tre **tillstånd**, nämligen **EMPTY**, **OCCUPIED** eller **REMOVED**. **REMOVED** behövs för att metoderna **put, remove** och **get** inte ska sluta sin sökning efter en nyckel för tidigt. De avbryts endast då de kommer till en position med korrekt nyckel / vars tillstånd är **EMPTY**.

Varje position i tabellen måste alltså kunna lagra:

- Tillstånd
- Nyckel
- Värde

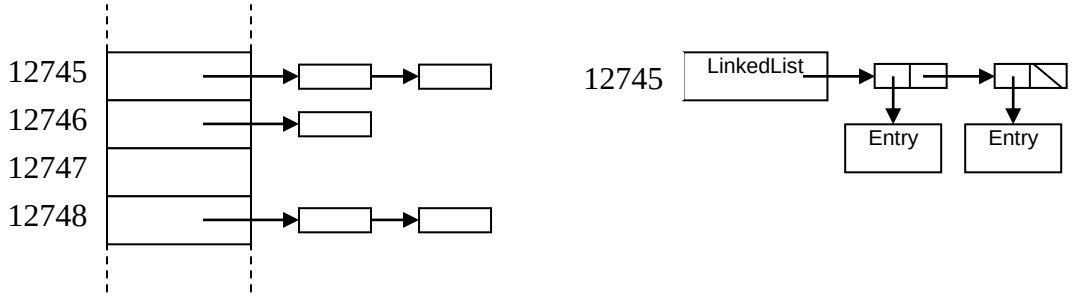
```
class Bucket {
   static final int EMPTY = 0, OCCUPIED = 1, REMOVED = 2;
   int state = EMPTY;
   Object key;
   Object value;
}
```

Bucket.java

Ett exempel på öppen hashing

Vid öppen hashing låter man varje tabellposition vara en datastruktur som kan lagra ett godtyckligt antal element. Om flera nycklar leder till samma tabellposition så kommer flera <key,value>-par att lagras i positionen.

Om varje position är en LinkedList kan en bit av tabellen se ut så här:



Ett <key,value>-par lagras i ett objekt av typen Entry:

```
class Entry {
   Object key;
   Object value;

   public Entry( Object key, Object value ) {
      this.key = key;
      this.value = value;
   }

   // jämför två nycklar, returnerar true om lika
   public boolean equals( Object obj ) {
      Entry keyValue = ( Entry )obj;
      return key.equals( keyValue.key );
   }
}
```

Entry.java

Ett exempel på öppen hashing

Algoritm för att sätta in ett element, put(Object key, Object value):

Beräkna position i tabellen med hjälp av anrop till key.hashCode()
Skapa ett Entry-objekt med paret <key,value>
Sök efter Entry-objektet i den länkade listan
Om objektet inte finns så
    Placera Entry-objektet i listan

Algoritm för att söka ett element, get(Object key):

Beräkna position i tabellen med hjälp av anrop till key.hashCode()
Skapa ett Entry-objekt med paret <key,null>
Sök efter Entry-objektet i den länkade listan
Om objekt med samma key finns så
    Returnera det funna objektets value
Returnera null

Algoritm för att ta bort ett element, remove(Object key):

Beräkna position i tabellen med hjälp av anrop till key.hashCode()
Skapa ett Entry-objekt med paret <key,null>
Sök efter Entry-objektet i den länkade listan
Om objekt med samma key finns så
    Ta bort det funna objektet ur listan

HashtableOHjava