sid 1052 - 1076

Föreläsning 9

Hashtabell

MALMÖ HÖGSKOLA

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

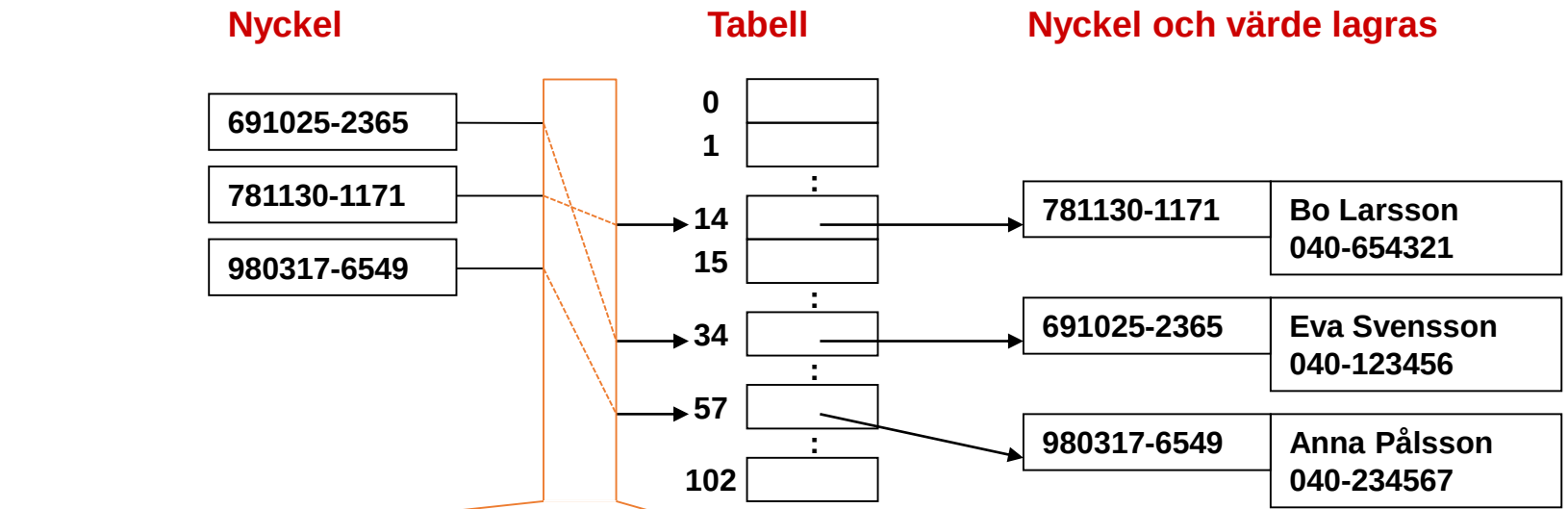
In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

Wikipedia: http://sv.wikipedia.org/wiki/Hashtabell

Inom datavetenskap är hashtabell en datastruktur där data sparas tillsammans med en nyckel. Positionen i strukturen beräknas med en hashfunktion.

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.




```
private int hashIndex(String key) {
    int nbr1 = Integer.parseInt(key.substring(1,5));
    int nbr2 = Integer.parseInt(key.substring(7));
    return (nbr1+nbr2) % 103;
}
```

Speed.java

HashCode.java

MALMÖ HÖGSKOLA

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.
- två objekt med samma värde i instansvariablerna inte ger true vid anrop till equals och förmodligen inte samma hashCode.

<u>Några olika hashing-metoder</u>

The Extraction Method

Ta ut t.ex. J från Jonas

The Division Method

Dela nyckel med heltal

The Folding Method

The Mid-Square Method

The Radix Transformation Method

The Digit Analysis Method

The Length-Dependent Method

MALMÖ HÖGSKOLA

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	TestChars.java

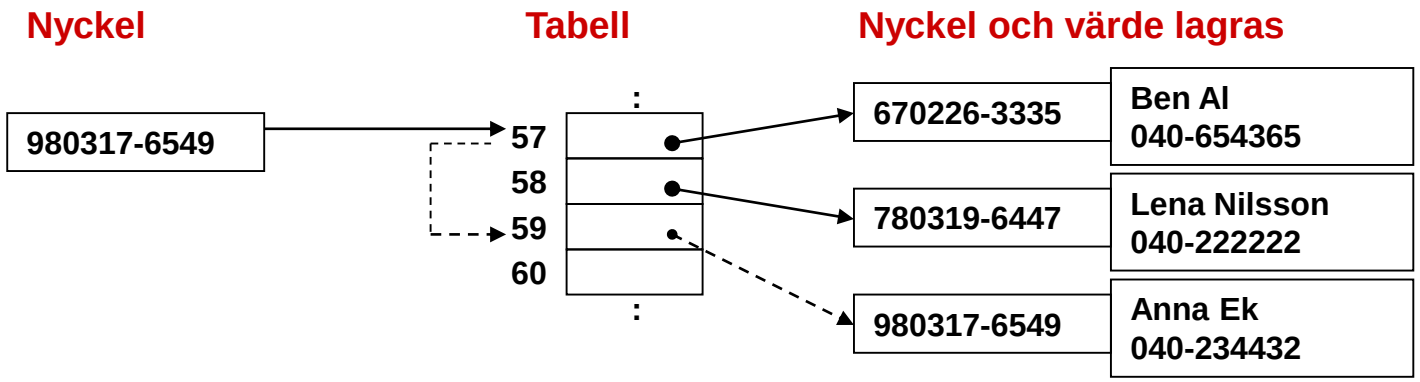
MALMÖ HÖGSKOLA

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** (Open Adressing).

  I exemplet nedan ger nyckeln ”980317-6549” positionen 57. Men position 57 är upptagen, där är redan information lagrad. Nästa lediga position är 59. En tänkbar lösning är att lagra det nya objektet i position 59.



Linear Probing

Testa nästa plats i tabellen

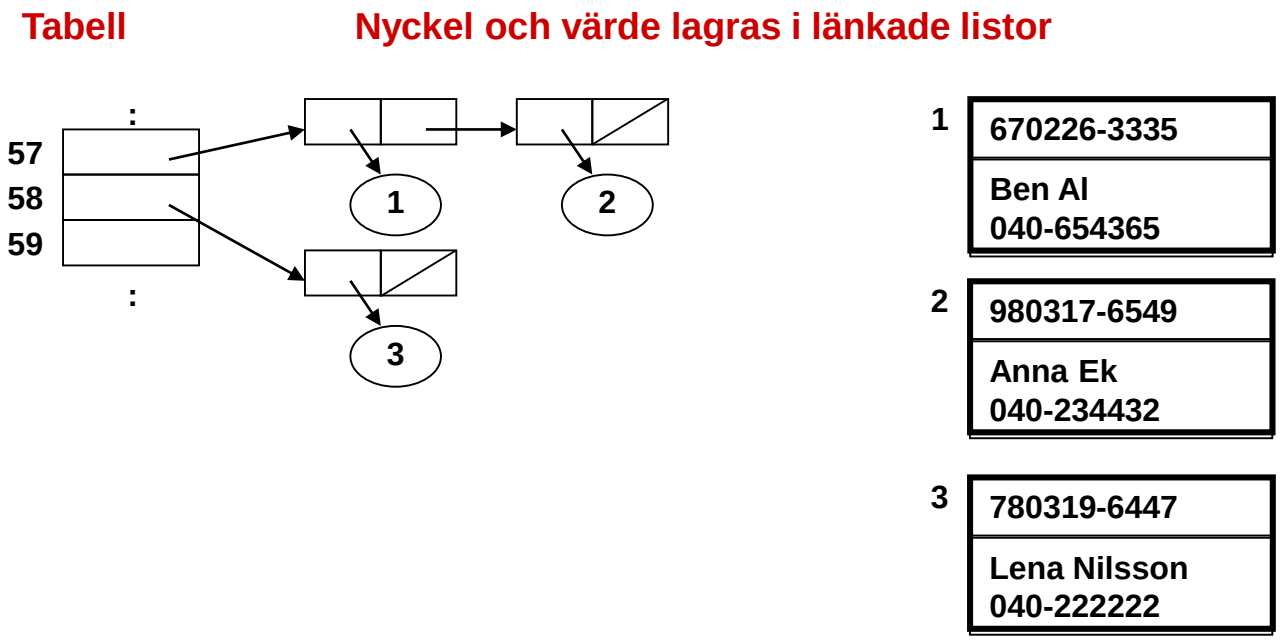
MALMÖ HÖGSKOLA

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**. (Chaining)

  I exemplet nedan ger nycklarna ”670226-3335” och ”980317-6549” samma position. De lagras i en länkad lista. Även fler element kan lagras i position 57.



MALMÖ HÖGSKOLA

Interfacet Map<K,V>

f14

Interface Map<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

`public interface` **`Map<K,V>`**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`clear()` Removes all of the mappings from this map.

```
public interface Map<K,V> {
    void clear();
    int size();
    boolean isEmpty();
    boolean containsKey(K key);
    V get(K key);
    V put(K key, V value);
    V remove(K key);
    Iterator<K> keys();
    Iterator<V> values();
}
```

MALMÖ HÖGSKOLA

Ett exempel på sluten hashing - BUCKET

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<K,V> {
   static final int EMPTY = 0, OCCUPIED = 1, REMOVED = 2;
   int state = EMPTY;
   K key;
   V value;
}
```

```
class Hashtable<K,V> implements Map<K,V> {
   private Bucket<K,V>[] table;
   :
}
```

Bucket.java

MALMÖ HÖGSKOLA

Ett exempel på sluten hashing

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

Om tabellen är full så öka kapacitet
Beräkna *position* i tabellen med hashfunktionen
Medan tabell[*position*] ej är EMPTY och *key* ej är samma som i tabell[*position*]
    Lagra första förekomsten som är REMOVED (om sådan påträffas)
    Ändra *position* till nästa tabellposition
Om tabell[*position*] är OCCUPIED
    Ändra value i tabellpos och returnera det gamla värdet
Annars om REMOVED påträffad
    Ändra key, value och state (OCCUPIED) i REMOVED-positionen, öka size och returnera *null*
Annars
    Ändra key, value och state (OCCUPIED) i tabell[*position*], returnera *null*

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

Beräkna *position* i tabellen med hashfunktionen
Medan tabell[*position*] ej är EMPTY och *key* ej är samma som i tabell[*position*]
    Ändra *position* till nästa tabellposition
Om tabell[*position*] är OCCUPIED
    Returnera värdet
Annars
    Returnera *null*

HashtableCH.java

MALMÖ HÖGSKOLA

Ett exempel på sluten hashing

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

Beräkna *position* i tabellen med hashfunktionen

Medan tabell[*position*] ej är EMPTY och *key* ej är samma som i tabell[*position*]

    Ändra *position* till nästa tabellposition

Om tabell[*position*] är OCCUPIED

    Tilldela *key* och *value* värdet *null* och *state* värdet REMOVED, minska size och returnera det borttagna värdet

Annars

    Returnera *null*

Algoritm för att returnera Iterator<K>, keys():

Skapa ett ArrayList<K>-objekt

Iterera genom tabellen

    Om en tabellpostion är OCCUPIED

        Lägg till tabellpositionens key i ArrayListen

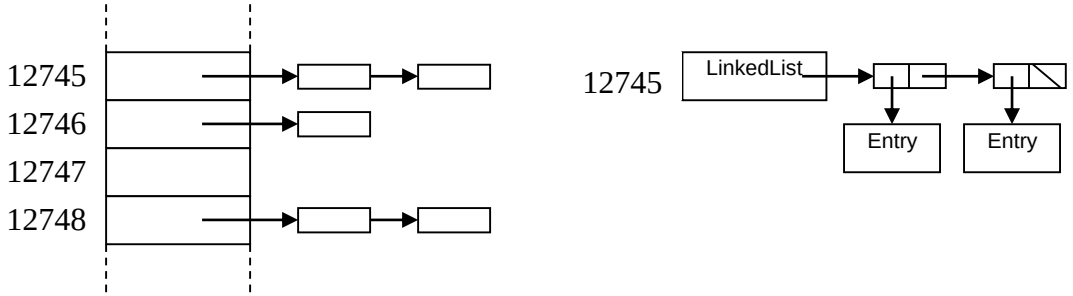
Returnera Iterator-implementering för ArrayListen

HashtableCH.java

MALMÖ HÖGSKOLA

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<Entry> kan en bit av tabellen se ut så här:



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

```
class Entry<K,V> {
   K key;
   V value;

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

   // jämför två nycklar, returnerar true om lika
   public boolean equals( Object obj ) {
      if( !(obj instanceof Entry) )
         return false;
      Entry<K,V> keyValue = ( Entry<K,V> )obj;
      return key.equals( keyValue.key );
   }
}
```

Entry.java

MALMÖ HÖGSKOLA

Ett exempel på öppen hashing

Algoritm för att sätta in ett element, put(K key, V value):
Beräkna *position* i tabellen med hashfunktionen
Skapa ett Entry-objekt med paret <key,value>
Sök efter Entry-objektet i den länkade listan i tabell[*position*]
Om Entry-objekt med samma key inte finns i listan så
    Placera Entry-objektet först i listan, öka *size* och returnera *null*
Annars
    Byt ut Entry-objektet i listan, returnera det gamla värdet

Algoritm för att söka ett element, get(K key):
Beräkna *position* i tabellen med hashfunktionen
Skapa ett Entry-objekt med paret <key,null>
Sök efter Entry-objektet i den länkade listan i tabell[*position*]
Om Entry-objekt med samma *key* finns i listan så
    Returnera det funna objektets *value*
Returnera *null*

Algoritm för att ta bort ett element, remove(K key):
Beräkna *position* i tabellen med hashfunktionen
Skapa ett Entry-objekt med paret <key,null>
Sök efter Entry-objektet i den länkade listan i tabell[*position]*
Om Entry-objekt med samma *key* finns i listan så
    Ta bort det funna objektet ur listan, minska size och returnera *value*
Annars
    Returnera null

HashtableOHjava

MALMÖ HÖGSKOLA

Föreläsning 10

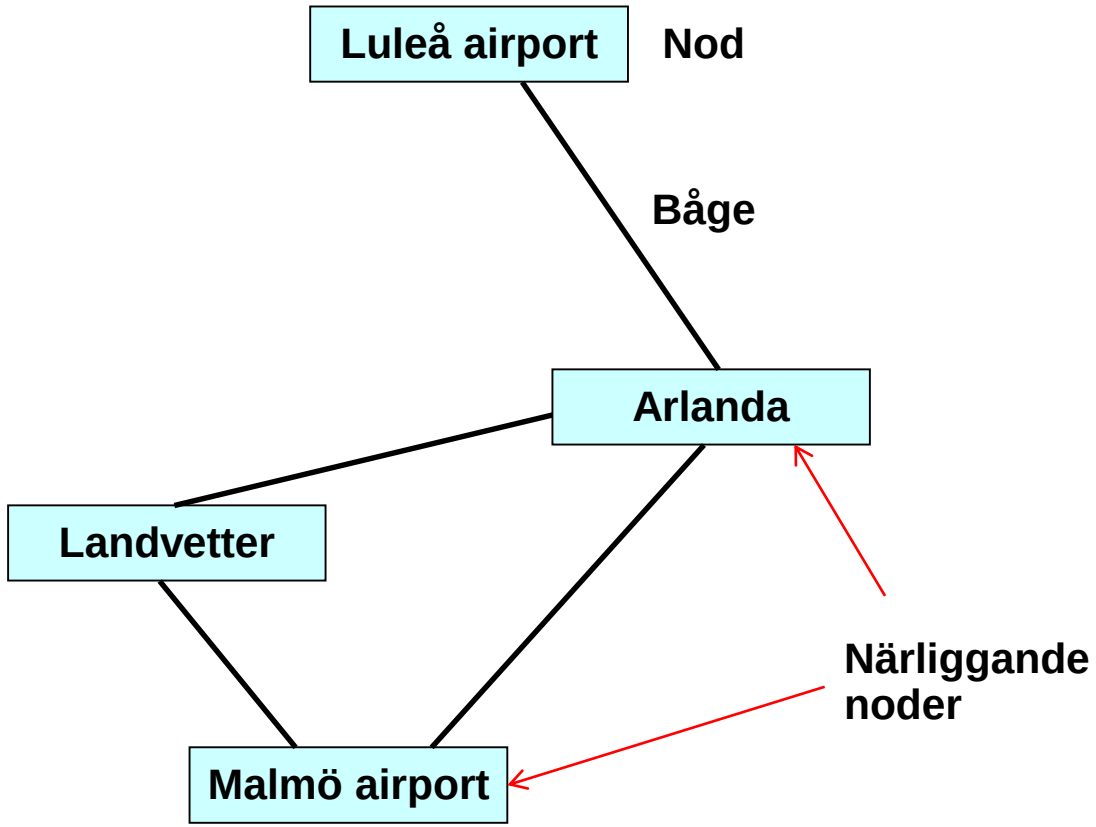
Graf

JF 19

Graf

I en graf kan en **nod** (**vertex**) referera till ett godtyckligt antal andra noder. En länk till en annan nod kallas för en **båge** (**edge**).
Två noder som sammanbinds med en båge är **närliggande** (**adjacent**).

Ett exempel på en graf är flygplatser sammanbundna med flyglinjer.

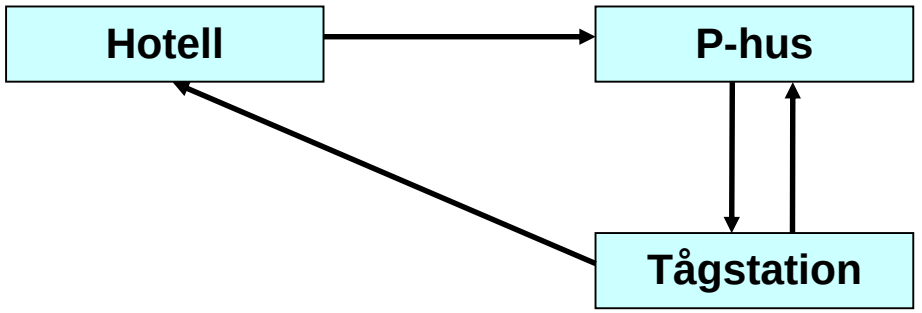


MALMÖ HÖGSKOLA

Graf

I en **riktad graf** (**directed graph**) har varje båge en riktning. Bågen startar i en nod och slutar i en nod.

Exempel: Om man ska ta sig från P-hus till Hotell med bil måste man färdas via Tågstation.



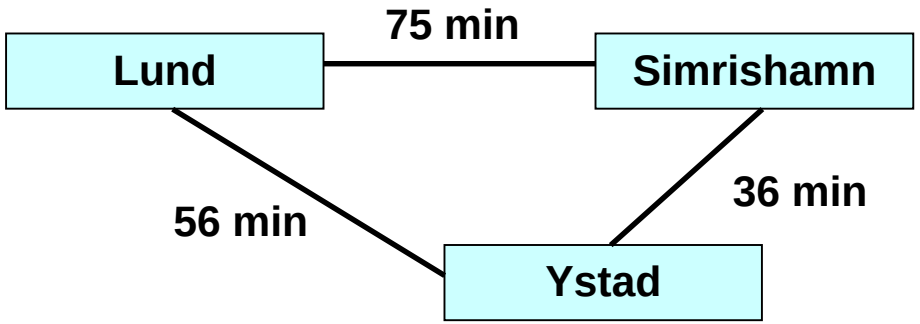
En graf där bågarna ej har riktning är en **oriktad graf** (**undirected graph**).

MALMÖ HÖGSKOLA

Graf

I en **viktad graf (weighted graph)** har varje båge en vikt. Vikten kan t.ex. beskriva avståndet mellan noderna (i tid eller sträcka eller …)

Exempel: Om man kör bil från Lund till Simrishamn tar det 75 minuter. Men om man tar vägen via Ystad tar det 92 minuter (56 + 36 minuter).



En graf där bågarna ej har någon vikt är en **oviktad graf**.

MALMÖ HÖGSKOLA

Graf - implementering

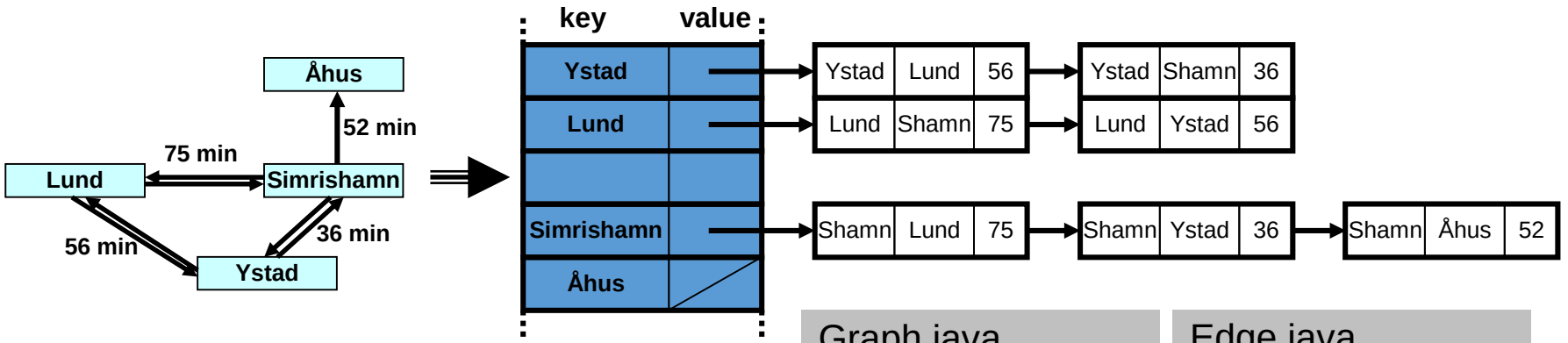
En graf består av en mängd av noder och en mängd av bågar.

Vi kommer implementera en graf som är ***riktad*** och ***viktad***:

Klassen **Edge(from, to, weight)** representerar en båge i grafen.

Klassen **Graph(graph)** representerar en graf med noder och bågar. Noder och bågar lagras i en ***HashMap*** där varje nyckel är en nod och det tillhörande värdet är en lista med bågar som startar i noden:

```
public class Graph<T> {
   private HashMap< T, ArrayList<Edge<T>> > graph = new …
   :
}
```



Graph.java | Edge.java

MALMÖ HÖGSKOLA

Graf - implementering

Operationer i klassen **Graph<T>**:

- **addVertex(T vertex) : void**
- **addVertex(T vertex, ArrayList< Edge<T>> adjacentList) : void**
- **addEdge(T from, T to, int weight) : boolean**
- **removeVertex(T vertex) : ArrayList<Edge<T>>**
- **removeEdge(T from, T to) : boolean**
- **getAdjacentList(T vertex) : ArrayList<Edge<T>>**
- **containsVertex(T vertex) : boolean**
- **getEdge(T from, T to) : Edge**

Graf – traversera på djupet

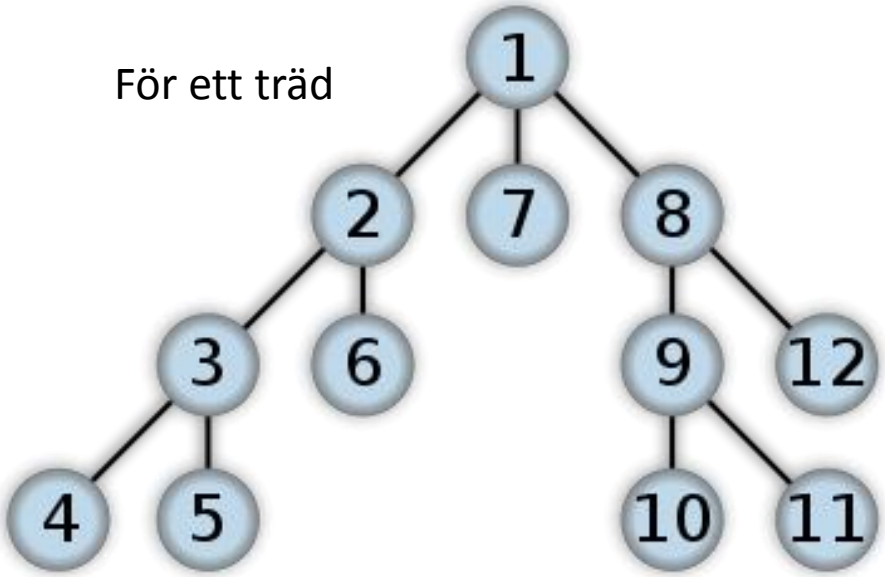
Strategi vid traversering på djupet i en graf är ungefär samma som man använder för att söka genom en labyrint. Testa olika vägar tills hela labyrinten är besökt.

1 Flytta från nod till nod utan att besöka en nod två gånger.

2 Om man besökt alla (nåbara) noder så är traverseringen klar.

3 Om man inte kan flytta vidare till en icke besökt nod så gå tillbaka, nod för nod, tills det finns ett alternativt vägval som leder till en icke besökt nod. Och fortsätt sedan enligt punkt 1.

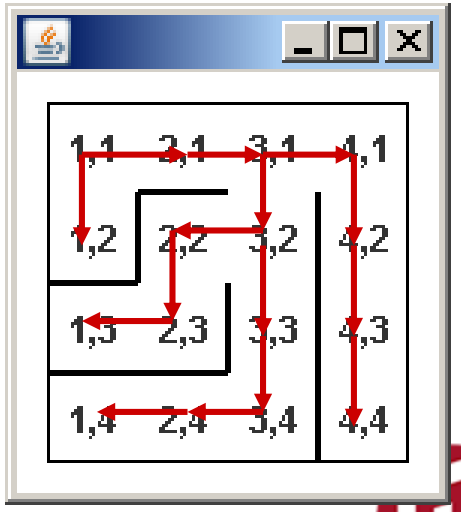
För ett träd



Exempel:

Traversering med start i 1,1

```
1,1-> 1,2
1,1-> 2,1-> 3,1-> 3,2-> 2,2-> 2,3-> 1,3
                  3,2-> 3,3-> 3,4-> 2,4-> 1,4
            3,1-> 4,1-> 4,2-> 4,3-> 4,4
```



MALMÖ HÖGSKOLA

Graf – traversera på djupet

```
public static <T> Iterator<T> traversalDF(Graph<T> graph, T start) {
    LinkedList<Edge<T>> stack = new LinkedList<Edge<T>>();
    LinkedHashSet<T> visited = new LinkedHashSet<T>();
    Edge<T> edge;

    visited.add(start);
    stack.addAll(graph.getAdjacentList(start));
    while( !stack.isEmpty() ) {
        edge = stack.removeLast();
        if(!visited.contains(edge.getTo())) {
            visited.add(edge.getTo());
            stack.addAll(graph.getAdjacentList(edge.getTo()));
        }
    }
    return visited.iterator();
}
```

GraphExample.java GraphSearch.java

MALMÖ HÖGSKOLA

Graf – sökning på djupet

Strategi vid sökning på djupet i en graf är ungefär samma som man använder för att hitta i en labyrint. Testa olika vägar tills man kommit till målet.

1 Flytta från nod till nod utan att besöka en nod två gånger.

2 Om man kommer till noden man söker så är sökningen klar.

3 Om man inte kan flytta vidare till en icke besökt nod så gå tillbaka nod för nod tills det finns ett alternativt vägval som leder till en icke besökt nod. Och fortsätt sedan enligt punkt 1.

Exempel:

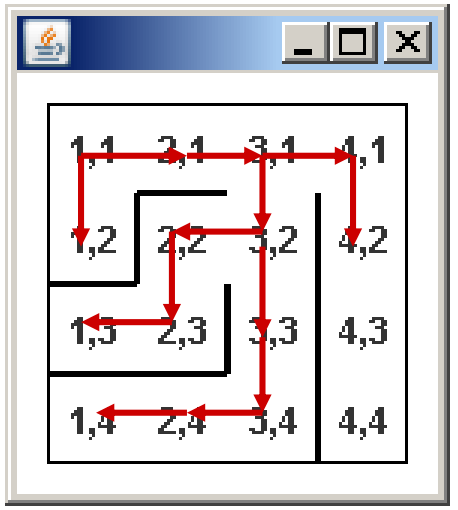
Sökning från 1,1 till 4,2 kan t.ex. bli så här

```
1,1-> 1,2
1,1-> 2,1-> 3,1-> 3,2-> 2,2-> 2,3-> 1,3
                  3,2-> 3,3-> 3,4-> 2,4-> 1,4
            3,1-> 4,1-> 4,2
```

Sökning från 1,1 till 4,2 kan t.ex. bli så här

```
1,1-> 2,1-> 3,1-> 4,1-> 4,2
```

Sökresultatet beror på i vilken ordning bågarna lagras



MALMÖ HÖGSKOLA

Graf – sökning på djupet

```
public static <T> ArrayList<Edge<T>> depthFirstSearch(Graph<T> graph, T from, T to) {
    LinkedList<Edge<T>> stack = new LinkedList<Edge<T>>();
    HashMap<T, Edge<T>> visited = new HashMap<T, Edge<T>>();
    boolean arrived = from.equals(to);
    Edge<T> edge;

    visited.put(from,null);
    stack.addAll(graph.getAdjacentList(from));
    while( !stack.isEmpty() && !arrived ) {
        edge = stack.removeLast();
        if(!visited.containsKey(edge.getTo())) {
            visited.put(edge.getTo(),edge);
            stack.addAll(graph.getAdjacentList(edge.getTo()));
            arrived = to.equals(edge.getTo());
        }
    }
    return getPath(from, to, visited);
}
```

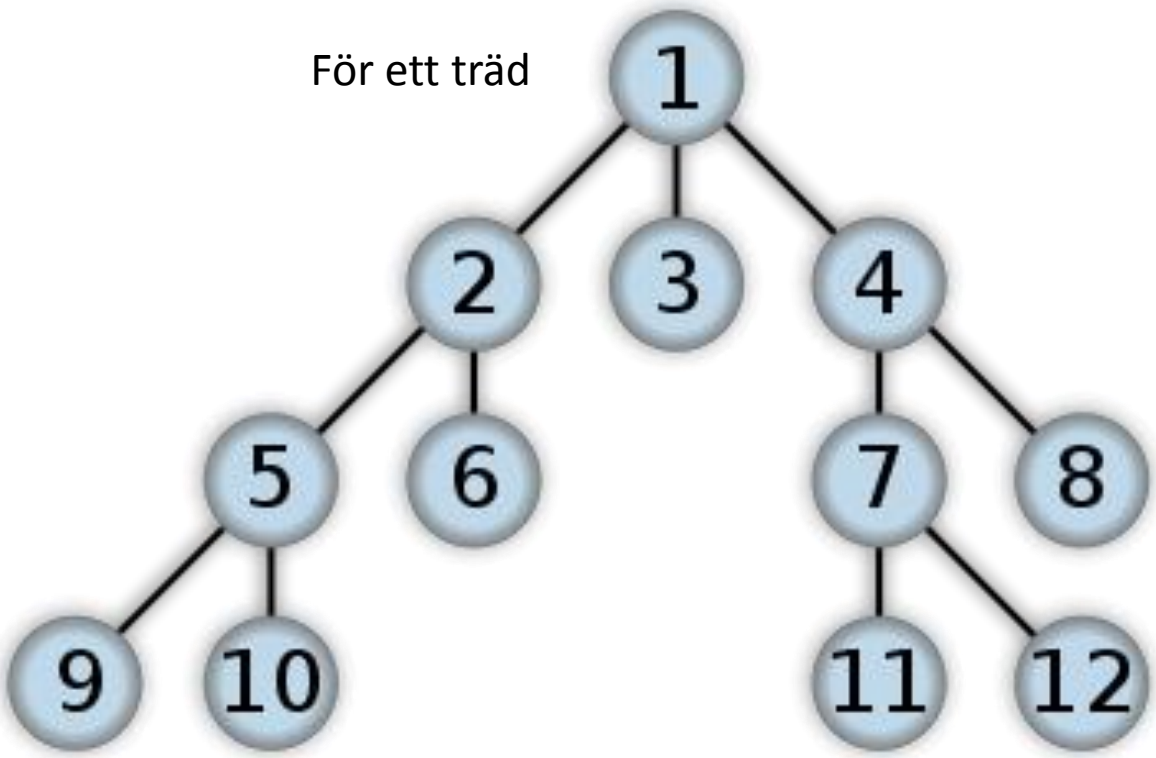
GraphExample.java GraphSearch.java

MALMÖ HÖGSKOLA

Graf – traversera på bredden

Strategi vid traversering på bredden är att besöka alla närliggande noder. Och därefter besöka deras närliggande noder osv. På det viset breder sökningen ut sig från startnoden.

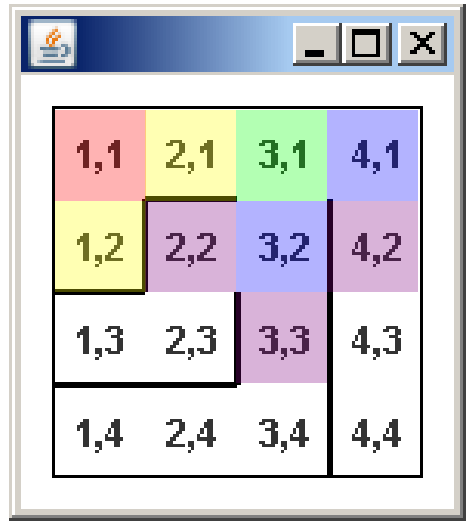
För ett träd



Exempel:

Traversera med start i 1,1:

1,1-> 1,2 och 2,1	Startnod
1,2 stopp 2,1-> 3,1	1 nod bort
3,1-> 3,2 och 4,1	2 noder bort
3,2-> 2,2 och 3,3 4,1-> 4,2	3 noder bort
2,2-> 2,3 3,3-> 3,4 4,2-> 4,3 osv	4 noder bort



MALMÖ HÖGSKOLA

Graf – traversera på bredden

```
public static <T> Iterator<T> traversalBF(Graph<T> graph, T start) {
    LinkedList<Edge<T>> queue = new LinkedList<Edge<T>>();
    LinkedHashSet<T> visited = new LinkedHashSet<T>();
    Edge<T> edge;

    visited.add(start);
    queue.addAll(graph.getAdjacentList(start));
    while( !queue.isEmpty() ) {
        edge = queue.removeFirst();
        if(!visited.contains(edge.getTo())) {
            visited.add(edge.getTo());
            queue.addAll(graph.getAdjacentList(edge.getTo()));
        }
    }
    return visited.iterator();
}
```

GraphExample.java GraphSearch.java

MALMÖ HÖGSKOLA

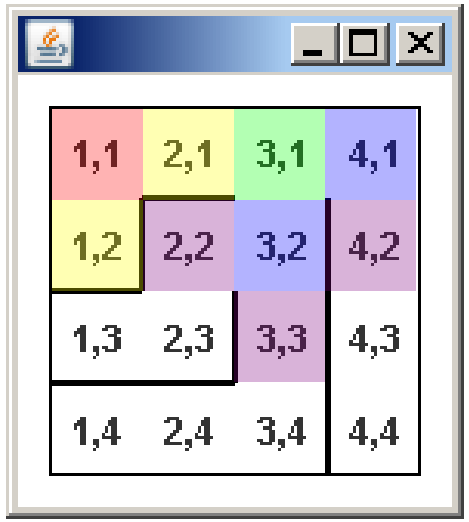
Graf – sökning på bredden

Strategi vid sökning på bredden är att utforska alla närliggande noder. Och därefter utforska deras närligande noder osv. På det viset breder sökningen ut sig från startnoden.

Exempel:

Sökning från 1,1 till 4,2, Utforskade noder:

1,1-> 1,2 och 2,1	Startnod
1,2 stopp 2,1-> 3,1	1 nod bort
3,1-> 3,2 och 4,1	2 noder bort
3,2-> 2,2 och 3,3 4,1-> 4,2	3 noder bort
2,2-> 2,3 3,3-> 3,4 **4,2**-> 4,3, **framme**	4 noder bort



MALMÖ HÖGSKOLA

Graf – sökning på bredden

```
public static <T> ArrayList<Edge<T>> breadthFirstSearch(Graph<T> graph, T from, T to) {
    LinkedList<Edge<T>> queue = new LinkedList<Edge<T>>();
    HashMap<T, Edge<T>> visited = new HashMap<T, Edge<T>>();
    boolean arrived = from.equals(to);
    Edge<T> edge;

    visited.put(from,null);
    queue.addAll(graph.getAdjacentList(from));
    while( !queue.isEmpty() && !arrived ) {
        edge = queue.removeFirst();
        if(!visited.containsKey(edge.getTo())) {
            visited.put(edge.getTo(),edge);
            queue.addAll(graph.getAdjacentList(edge.getTo()));
            arrived = to.equals(edge.getTo());
        }
    }
    return getPath(from, to, visited);
}
```

GraphExample.java GraphSearch.java

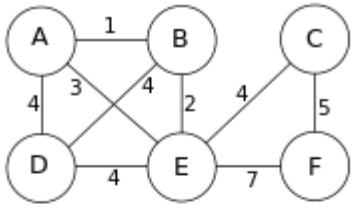
MALMÖ HÖGSKOLA

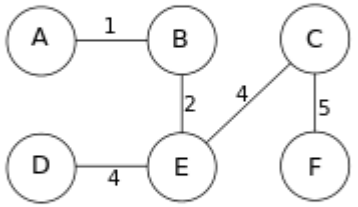
Minimum Spanning Tree

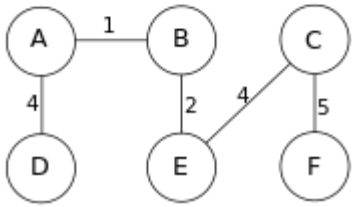
Den lägsta kostnaden att sammanbinda noderna i en graf beskrivs av ett ***minimalt uppspännande träd*** (***Minimal Spanning Tree***). En graf kan innehålla flera likvärdiga lösningar.

Algoritm (Input: Grafen *graph* och en *node* i grafen)

Skapa en graf, *mst*
Skapa en minheap, *heap*
Lägg till *node* i *mst*
Alla bågar som går från *node* placeras i *heap*
Så länge det finns fler bågar i *heap*
    Ta båge ur *heap* (det är bågen med lägst vikt) och lagra i *edge*
    Lagra noden som *edge* leder till i *node*
    Om *node* ej finns i *mst*
        Lägg till *node* i *mst*
        Lägg till *edge* i *mst*
        Alla bågar som går från *node* placeras i *heap*
returnera *mst*







MALMÖ HÖGSKOLA

Minimum Spanning Tree

Den lägsta kostnaden att sammanbinda noderna i en graf beskrivs av ett ***minimalt uppspännande träd*** (***Minimal Spanning Tree***). En graf kan innehålla flera likvärdiga lösningar.

```
public static <T> Graph<T> minimumSpanningTree(Graph<T> graph, T from) {
   Graph<T> minmumSpanningTree = new Graph<T>();
   ArrayHeap<Edge<T>> heap = new arrayHeap<Edge<T>>(10);
   ArrayList<Edge<T>> adjacentList = graph.getAdjacentList(from);
   Edge<T> edge;

   minmumSpanningTree.addVertex(from);
   for(Edge<T> e : adjacentList)
      heap.insert(e);
   while( heap.size()>0 ) {
      edge = heap.delete();
      if(!minmumSpanningTree.containsVertex(edge.getTo())) {
         minmumSpanningTree.addVertex(edge.getTo());
         minmumSpanningTree.addEdge(edge.getFrom(), edge.getTo(), edge.getWeight());
         adjacentList = graph.getAdjacentList(edge.getTo());
         for(Edge<T> e : adjacentList)
            heap.insert(e);
      }
   }
   return minmumSpanningTree;
}
```

GraphSearch.java

MALMÖ HÖGSKOLA

Dijkstra – optimrad sökning på bredden

```
public static <T> ArrayList<Edge<T>> dijkstraSearch(Graph<T> graph, T from, T to) {
    Candidate<T> candidate;
    PriorityQueue<Candidate<T>> queue = new PriorityQueue<Candidate<T>>();
    HashMap<T, Edge<T>> visited = new HashMap<T, Edge<T>>();
    boolean arrived = from.equals(to);
    ArrayList<Edge<T>> adjacentList = graph.getAdjacentList(from);

    visited.put(from,null);
    for(Edge<T> edge : adjacentList)
        queue.add(new Candidate<T>(edge,edge.getWeight()));
    while( !queue.isEmpty() && !arrived ) {
        candidate = queue.remove();
        if(!visited.containsKey(candidate.edge.getTo())) {
            visited.put(candidate.edge.getTo(),candidate.edge);
            adjacentList = graph.getAdjacentList(candidate.edge.getTo());
            for(Edge<T> edge : adjacentList)
                queue.add(new Candidate<T>(edge,candidate.getTotalWeight() + edge.getWeight()));
            arrived = to.equals(candidate.edge.getTo());
        }
    }
    return getPath(from, to, visited);
}
```

Candidate<T> implements Comparable<Candidate<T>>
- edge : Edge<T> - totalWeight : int
+Candidate(Edge<T>,int) +compareTo(Candidate<T>)

GraphSearch.java

MALMÖ HÖGSKOLA

Frågor?