F11 - Rekursion

ID1004 Objektorienterad programmering

Fredrik Kilander fki@kth.se

Rekursion

- Rekursion är en programmeringsteknik
- En metod anropar sig själv

```
public String reverse (String s) {

   if (s.length() == 1) {
     return s;
   }
   else {
     return reverse(s.substring(1)) + s.substring(0, 1);
   }

 }
```

”abc”.substring(1) → ”bc” ”abc”.substring(0,1) → ”a”

Rekursion

- Rekursion är en programmeringsteknik
- En metod anropar sig själv

- En rekursiv metod gör tre saker:
  - kontrollerar om det finns ett enkelt svar (stoppvillkor)
  - gör en del av arbetet
  - anropar sig själv för att göra resten

Rekursion

- Rekursion är en programmeringsteknik
- En metod anropar sig själv

```
public String reverse (String s) {

   if (s.length() == 1) {
     return s;
   }
   else {
     return reverse(s.substring(1)) + s.substring(0, 1);
   }

}
```

Stoppvillkor

En del av arbetet

Resten av arbetet

”abc”.substring(1) → ”bc”

”abc”.substring(0,1) → ”a”

Rekursion är iteration (loop)

- Istället för for, while eller do så används upprepade anrop till den egna metoden
- Varje anrop *förminskar* problemet lite grand
- Till slut finns bara enkla svar
- Ett enkelt svar kan lösas utan rekursion
- Rekursionen når då sitt stoppvillkor

Rekursiva definitioner

- **dekoration**: *s*. ett ornament eller utsmyckning som används för att dekorera med

- A *List* is a:
  - number
  - or a number comma *List*

3
12, 15
22, 1, -45, 3, 3, 91

Oändlig rekursion == oändlig loop

- **rekursion**: *s*. se rekursion

- Stoppvillkor är nödvändigt
- Stoppvillkoren måste vara heltäckande

ändlig rekursion == ändlig loop

- **rekursion**: *s*. antingen finns ett enkelt svar, eller så kan ett delsvar hittas och resten av svaret fås med rekursion över det resterande problemet.

- Stoppvillkor är nödvändigt
- Stoppvillkoren måste vara heltäckande

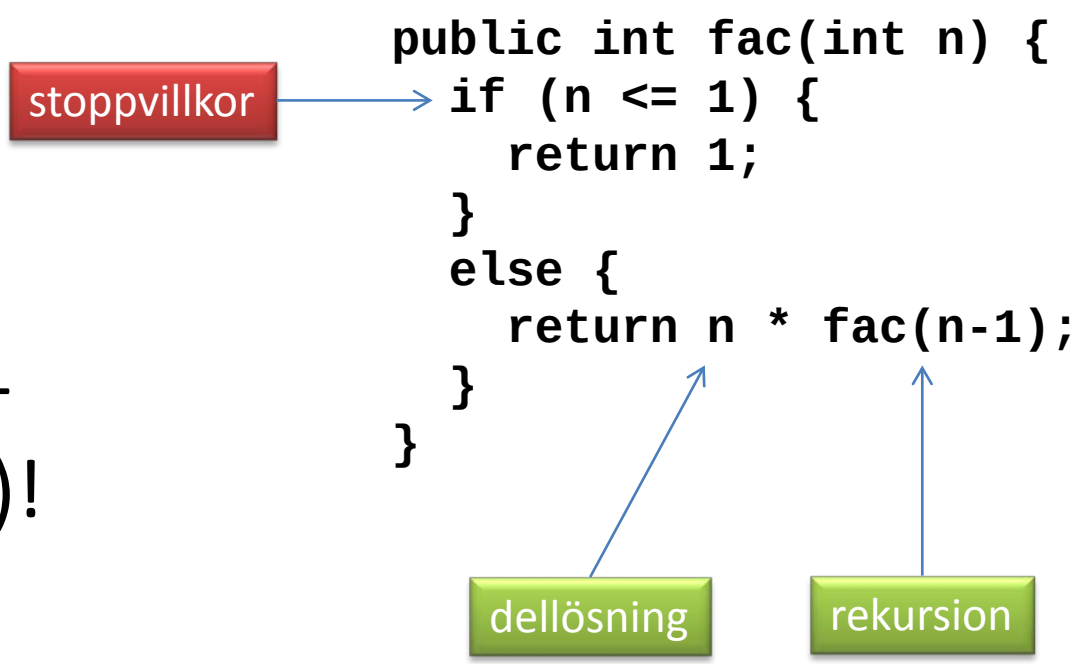
Fakultetsfunktionen

- 0! = 1
- 1! = 1
- 2! = 2 * 1
- 3! = 3 * 2 * 1
- n! = n * (n-1)!

```
public int fac(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fac(n-1);
  }
}
```

Fakultetsfunktionen !

- 0! = 1
- 1! = 1
- 2! = 2 * 1
- 3! = 3 * 2 * 1
- n! = n * (n-1)!



Summafunktionen

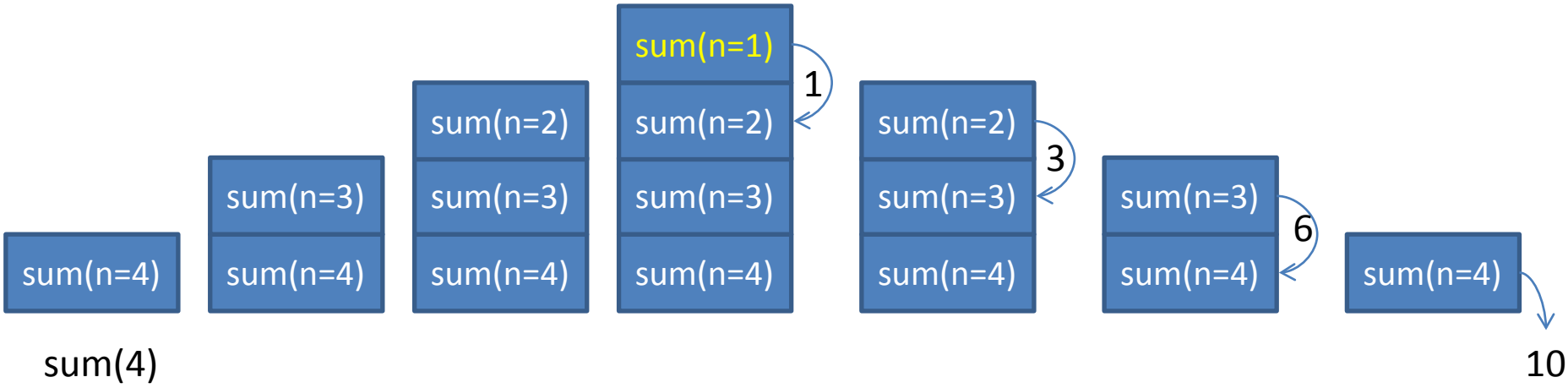
- sum(1) = 1
- sum(2) = 2 + 1
- sum(3) = 3 + 2 + 1
- sum(n) = n+sum(n-1)

```
public int sum(int n) {
  if (n == 1) {
    return 1;
  }
  else {
    return n + sum(n-1);
  }
}
```

Varför rekursion fungerar

- Parametrar och metodvariabler är lokala
- Varje metodanrop skapar nya lokala variabler
- Dessa finns så länge metoden är aktiv

```
int sum(int n) { return (n==1) ? n : n + sum(n-1);}
```



Rekursion vs explicit iteration

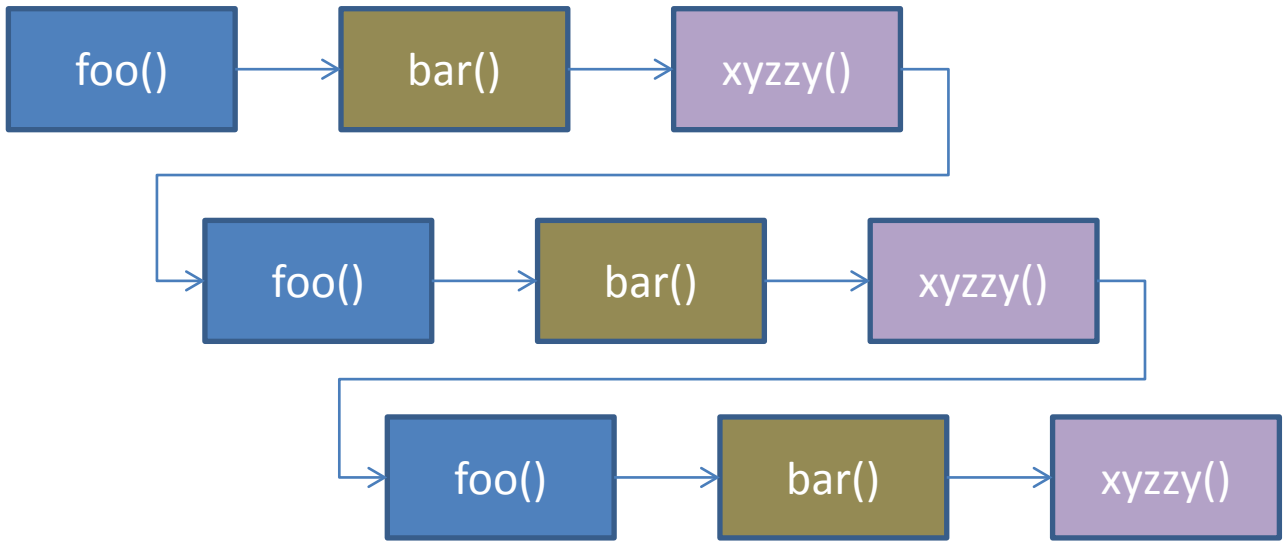
- Rekursion ger *ibland* elegantare lösningar

```
public int sum(int n) {
  int summa = 0;
  for (int i = 1; i <= n; i++) {
    summa = summa + i;
  }
  return summa;
}
```

```
public int sum(int n) {
  return (n==1) ? 1 : n + sum(n-1);
}
```

Indirekt rekursion

- Metod foo(..) anropar bar(..)
- bar(..) anropar xyzzy(..)
- xyzzy(..) anropar foo(..)



Sökning i en labyrint

- ’Labyrinten’ är en matris av int

```
private int[][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                         {1,0,1,1,1,0,1,1,1,1,0,0,1},
                         {0,0,0,0,1,0,1,0,1,0,1,0,0},
                         {1,1,1,0,1,1,1,0,1,0,1,1,1},
                         {1,0,1,0,0,0,0,1,1,1,0,0,1},
                         {1,0,1,1,1,1,1,1,0,1,1,1,1},
                         {1,0,0,0,0,0,0,0,0,0,0,0,0},
                         {1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

- 1 är öppen, 0 är stängd (vägg, kloss)
- Hitta en väg från 0,0 till 7,12

Sökning i en labyrint

- ’Labyrinten’ är en matris av int

start

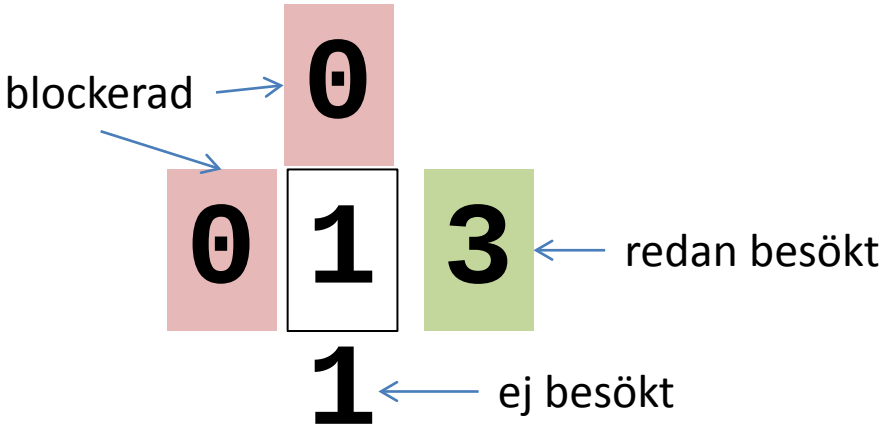
```
private int[][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                         {1,0,1,1,1,0,1,1,1,1,0,0,1},
                         {0,0,0,0,1,0,1,0,1,0,1,0,0},
                         {1,1,1,0,1,1,1,0,1,0,1,1,1},
                         {1,0,1,0,0,0,0,1,1,1,0,0,1},
                         {1,0,1,1,1,1,1,1,0,1,1,1,1},
                         {1,0,0,0,0,0,0,0,0,0,0,0,0},
                         {1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

mål

- 1 är öppen, 0 är stängd (vägg, kloss)
- Hitta en väg från 0,0 till 7,12

Labyrintalgoritmen

- Om det finns en väg till utgången från denna cell (rad och kolumn)
- så måste det också finnas en ej besökt väg till utgången från någon av de närmaste grannarna

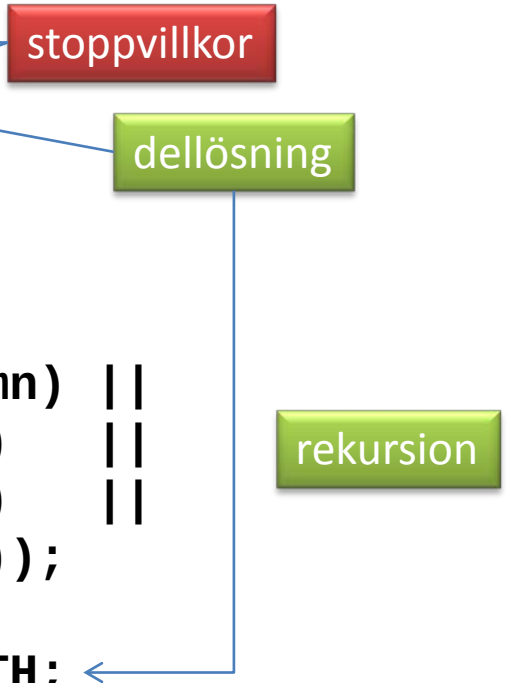


Labyrintalgoritmen

```
public boolean traverse(int row, int column){
  boolean done = false;
  if (valid(row, column)){
    grid[row][column] = TRIED;
    if (atGoal(row,column)) {
      done = true;
    }
    else {
      done = traverse(row + 1, column); // try down
      if (!done)
        done = traverse(row, column+1); // try right
      if (!done)
        done = traverse(row-1, column); // try up
      if (!done)
        done = traverse(row, column-1); // try left
    }
    if (done) grid[row][column] = PATH;
  }
  return done;
}
```

Labyrintalgoritmen v2

```
public boolean traverse(int row, int column){
  boolean done = false;
  if (valid(row, column)){
    grid[row][column] = TRIED;
    if (atGoal(row,column)) {
      done = true;
    }
    else {
      done = (traverse(row + 1, column) ||
              traverse(row, column+1)   ||
              traverse(row-1, column)   ||
              traverse(row, column-1));
    }
    if (done) grid[row][column] = PATH;
  }
  return done;
}
```



Att tänka rekursivt

- Kan problemet göras mindre i identiska steg?
- Vilka är stoppvillkoren?

- Blir koden enklare?
- Blir koden mer eller mindre förståelig?

Binärsökning

```
private int search (int [] ar, int left, int key, int right) {

    int index = (left + right) / 2;
    int value = ar[index];

    if (right < left) {
      return -1;          // Not found
    }
    else if (value == key) {
      return index;               // Found
    }
    else if (key < value) {
      return search (ar, left, key, index - 1);
    }
    else {
      return search (ar, index + 1, key, right);
    }
  }


  public int search (int [] ar, int key) {
    return search(ar, 0, key, ar.length - 1);
  }
```

Binärsökning

```
private int search (int [] ar, int left, int key, int right) {

    int index = (left + right) / 2;
    int value = ar[index];

    if (right < left) {
      return -1;          // Not found
    }
    else if (value == key) {
      return index;               // Found
    }
    else if (key < value) {
      return search (ar, left, key, index - 1);
    }
    else {
      return search (ar, index + 1, key, right);
    }
  }


  public int search (int [] ar, int key) {
    return search(ar, 0, key, ar.length - 1);
  }
```

Stoppvillkor

Stoppvillkor

Förminska och gör resten av arbetet

Towers of Hanoi

- Tre pinnar, start, mål och hjälp
- Uppträdda på varje pinne ligger skivor av olika storlek
- En mindre skiva måste alltid ligga ovanför en större skiva

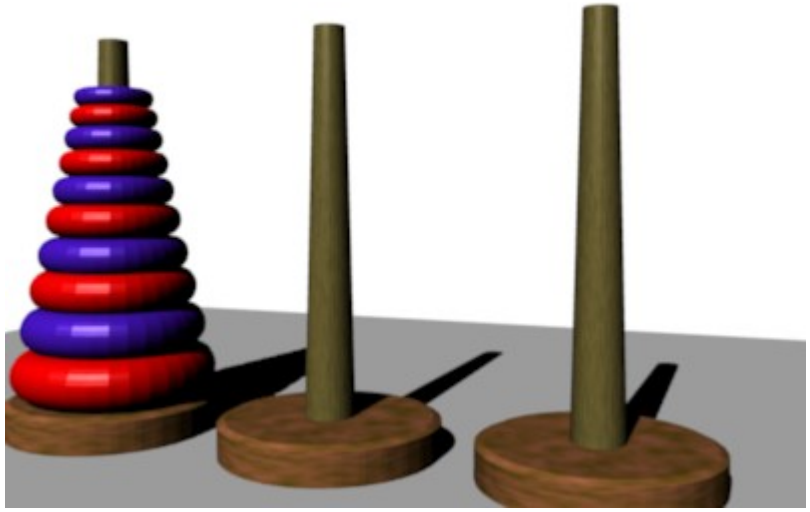


Bild: http://mathforum.org/mathimages/index.php/Image:10_Ring_Hanoi.jpg

Towers of Hanoi

- För att flytta ett torn om n skivor givet startpinne, målpinne och hjälppinne
- Om n=1, flytta direkt till målpinnen
- annars:
  - flytta tornet n-1 från start till hjälp, mha mål
  - flytta n till målet
  - flytta n-1 från hjälp till mål, mha start

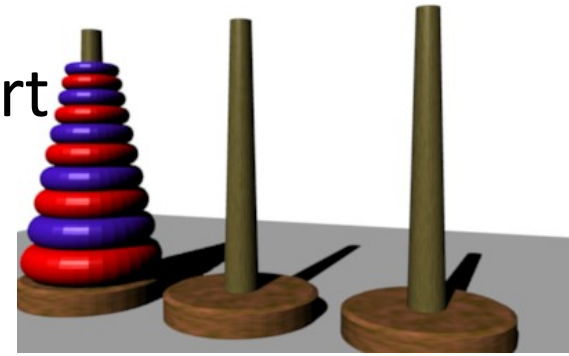
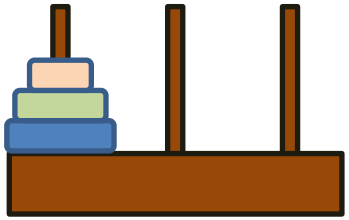


Bild: http://mathforum.org/mathimages/index.php/Image:10_Ring_Hanoi.jpg

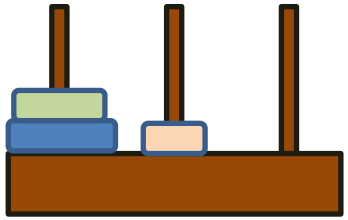
2^n-1 drag

start mål hjälp

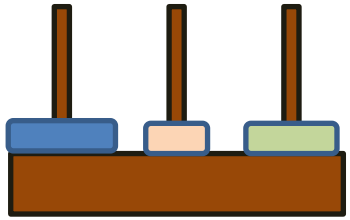


1

hjälp mål

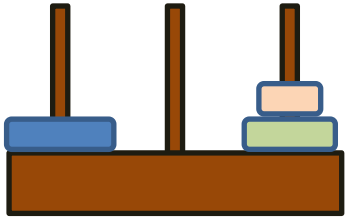


2



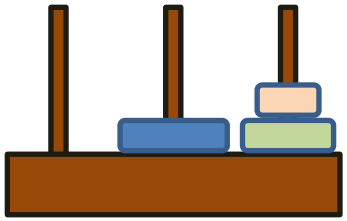
3

n-1 flyttat till hjälp

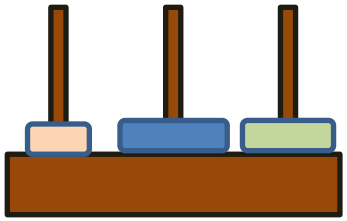


4

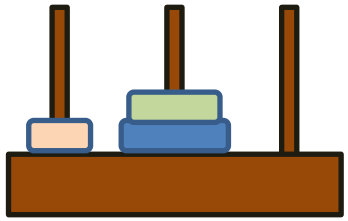
Flytta skiva 3 direkt till mål



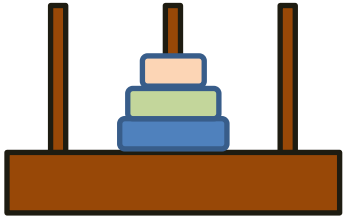
5



6



7



8

Towers of Hanoi

```
private void moveTower(int numDisks, int start, int end, int temp){

  if (numDisks == 1) {
    moveOneDisk(start, end);
  }
  else {
    moveTower(numDisks-1, start, temp, end);
    moveOneDisk(start, end);
    moveTower(numDisks-1, temp, end, start);
  }
}

private void moveOneDisk(int start, int end){
  System.out.println(”Move one disk from ” + start + ” to ” + end);
}
```

Rekursion – sammanfattning

- Vissa problem lämpar sig för rekursiv lösning
- Problemet har en iterativ reduktion
  - Talserie
  - Lista
- Rekursionens styrka utnyttjas bäst vid navigering i flera dimensioner
  - Labyrint
  - Träd- eller graf

Rekursion – sammanfattning

- Rekursiva lösningar har ofta även en traditionell iterativ lösning
- Rekursionen tar plats på anropsstacken
- En rekursiv algoritm kan vara svår att förstå

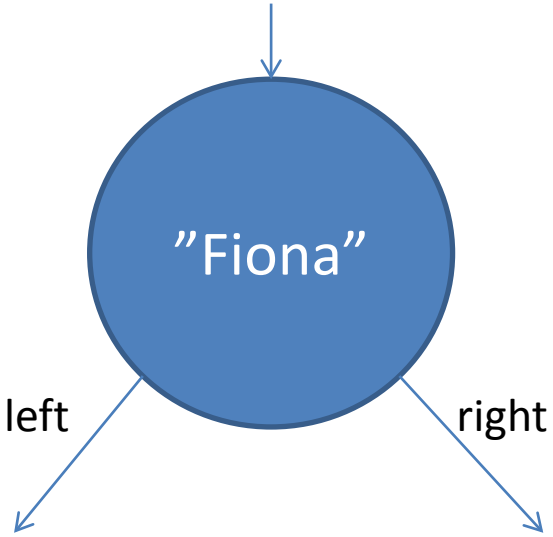
- Även om den är elegant

GNU: GNU’s Not Unix

SLUT PÅ BILDER

Sökning i ett binärt träd

- Trädet är en riktad, acyklisk graf av noder
- Varje nod innehåller en sträng och högst två referenser till andra träd



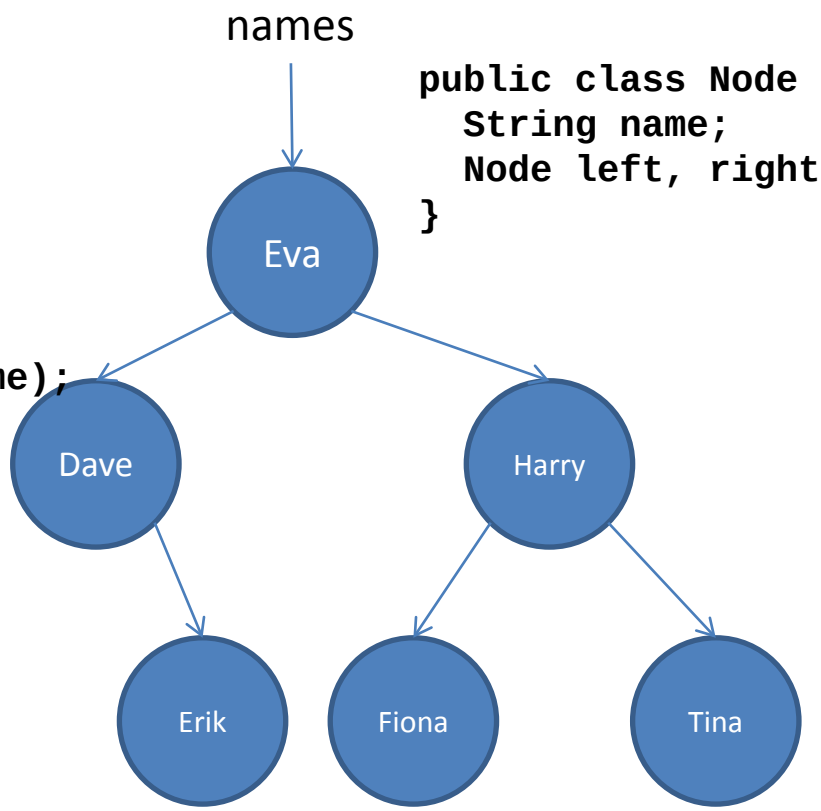

```
public class Node {
  String name;
  Node left, right;
}
```

Sökning i binärt träd

```
Node names = ...;

search(names, ”Fiona”);
```

```
Node search(Node node, String key){
  if (node != null) {
    int comparison = key.compareTo(node.name);
    if (comparison == 0) {
      return node;
    }
    else if (comparison < 0) {
      return search(node.left, key);
    }
    else {
      return search(node.right, key);
    }
  }
  return null;
}
```




```
public class Node {
  String name;
  Node left, right;
}
```

Sökning i binärt träd - OOP

```
Node names = ...;

names.search(”Fiona”);
```

```
public class Node {
  String name;
  Node left, right;

  Node search(String key){
    int comparison = key.compareTo(name);
    if ((comparison < 0) && (left != null)) {
      return left.search(key);
    }
    else if ((0 < comparison) && (right != null)) {
      return right.search(key);
    }
    else if (comparison == 0) {
      return this;
    }
    else {
      return null;
    }
  }
}
```