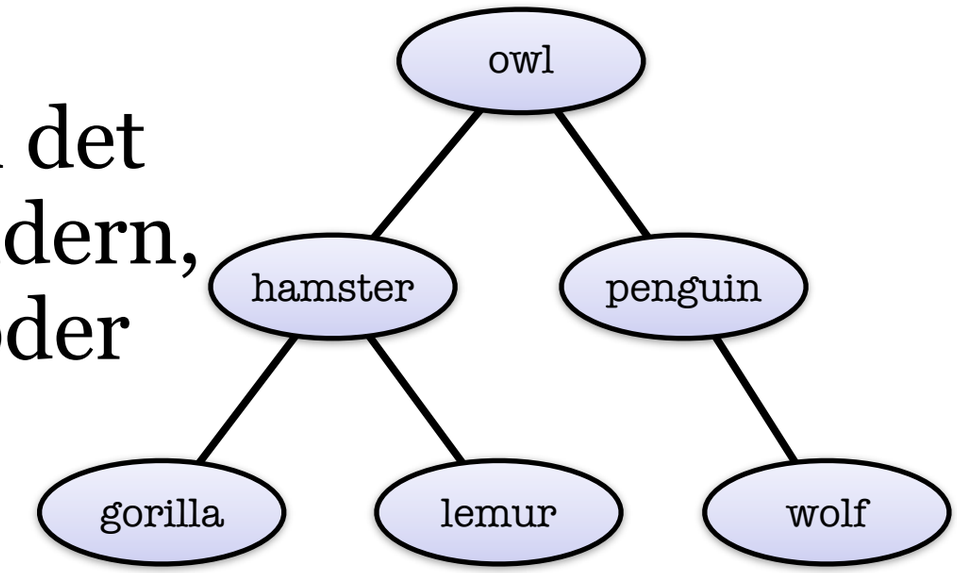
Binära sökträd
Weiss, avsnitt 4.3

Peter Ljunglöf

DAT036, Datastrukturer

30 nov 2012

Binära sökträd



I ett binärt sökträd är alla noder i det vänstra delträdet mindre än föräldern, som i sin tur är mindre än alla noder i det högra delträdet

Eller, mer formellt:

T är ett sökträd omm T är tomt, eller:

- T har två barn, T_L och T_R, som är binära sökträd, och
- värdet i T:s rotnod är större än alla värden i T_L, och
- värdet i T:s rotnod är mindre än alla värden i T_R

Detta är en *invariant*, som man kan använda för att kontrollera att ens kod är korrekt

Sökning i binära sökträd

Att söka efter *target* i ett binärt sökträd:

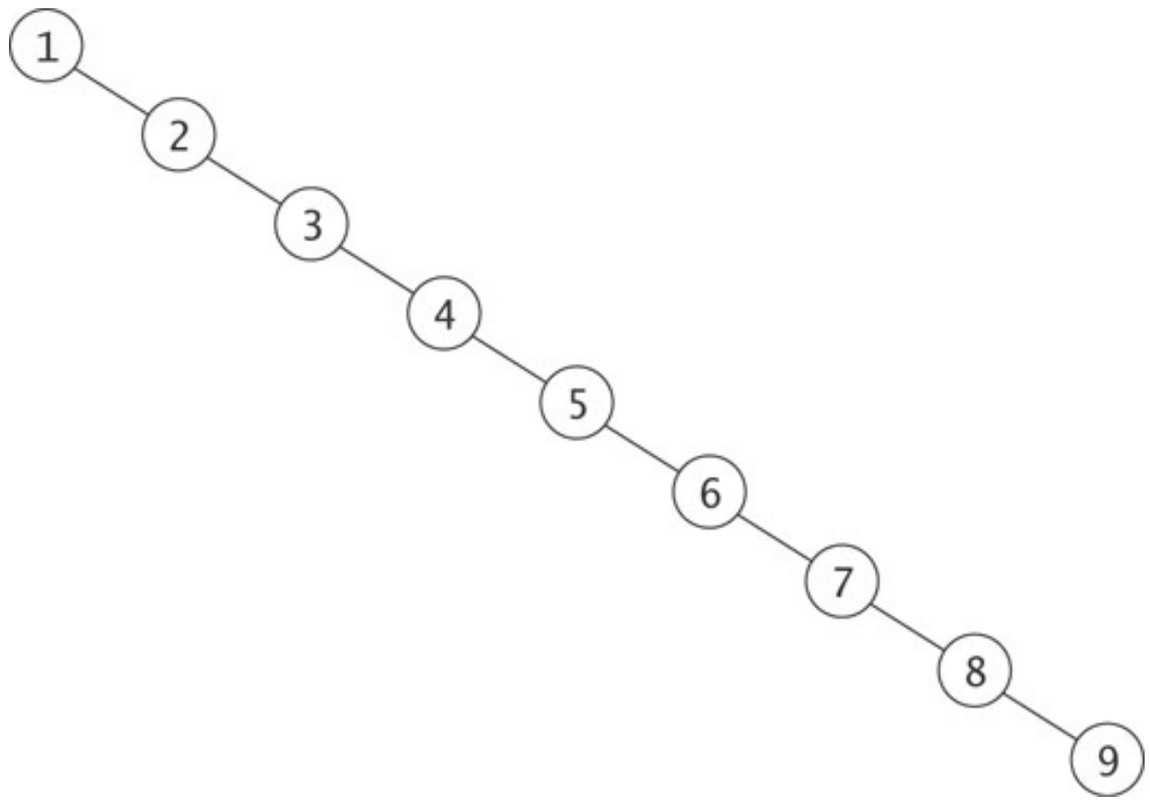
- om trädet är tomt:
  - return null
- annars, om *target* matchar rotnodens data:
  - return rotnodens data
- annars, om *target* är mindre än rotnodens data:
  - sök efter *target* i vänstra delträdet
- annars är *target* större än rotnodens data:
  - sök efter *target* i högra delträdet

Sökning i binära sökträd

BST-sökning har *potential* att få logaritmisk komplexitet, O(log *n*)

Men i värsta fall är sökningen linjär, O(*n*)

- om trädet är väldigt skevt:



Trädtraversering

Preorder:

- besök rotnoden, sedan T_L, sedan T_R

Inorder:

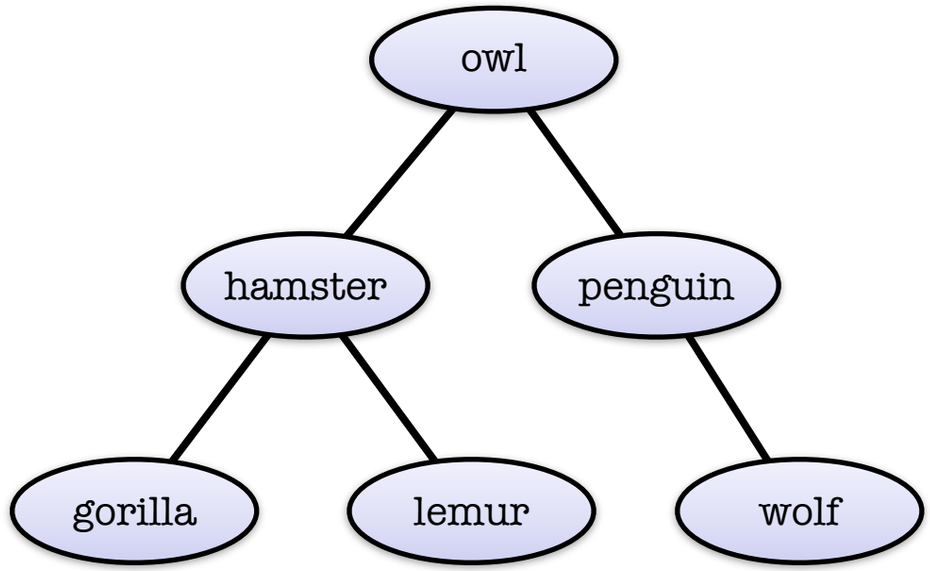
- besök T_L, sedan rotnoden, sedan T_R

Postorder:

- besök T_L, sedan T_R, sedan rotnoden

Inordertraversering av sökträd

Om vi besöker noderna
i ett binärt sökträd
inorder så får vi ut
noderna sorterade.



I detta fall:

gorilla, hamster, lemur, owl, penguin, wolf

Implementation [fig. 4.16–17]

```
public class BinarySearchTree<T> extends ... {
    private BinaryNode<T> root;

    private static class BinaryNode<T> {
        T element;
        BinaryNode<T> left;
        BinaryNode<T> right;

        BinaryNode(T elem, BinaryNode<T> lt,
                           BinaryNode<T> rt) {
            element = elem; left = lt; right = rt;
        }
    }
}
```

Rekursiv sökning

Metod *contains*(T *x*):

- return *contains*(*x, root*)

Rekursiv privat metod *contains*(T *x*, BinaryNode<T> *t*):

- om *x* == *null*: return *false*
- om *x* == *t.element*: return *true*
- om *x* < *t.element*: return *contains*(*x, t.left)*
- om *x* > *t.element*: return *contains*(*x, t.right)*

Iterativ sökning

Sökningen går också att implementera iterativt

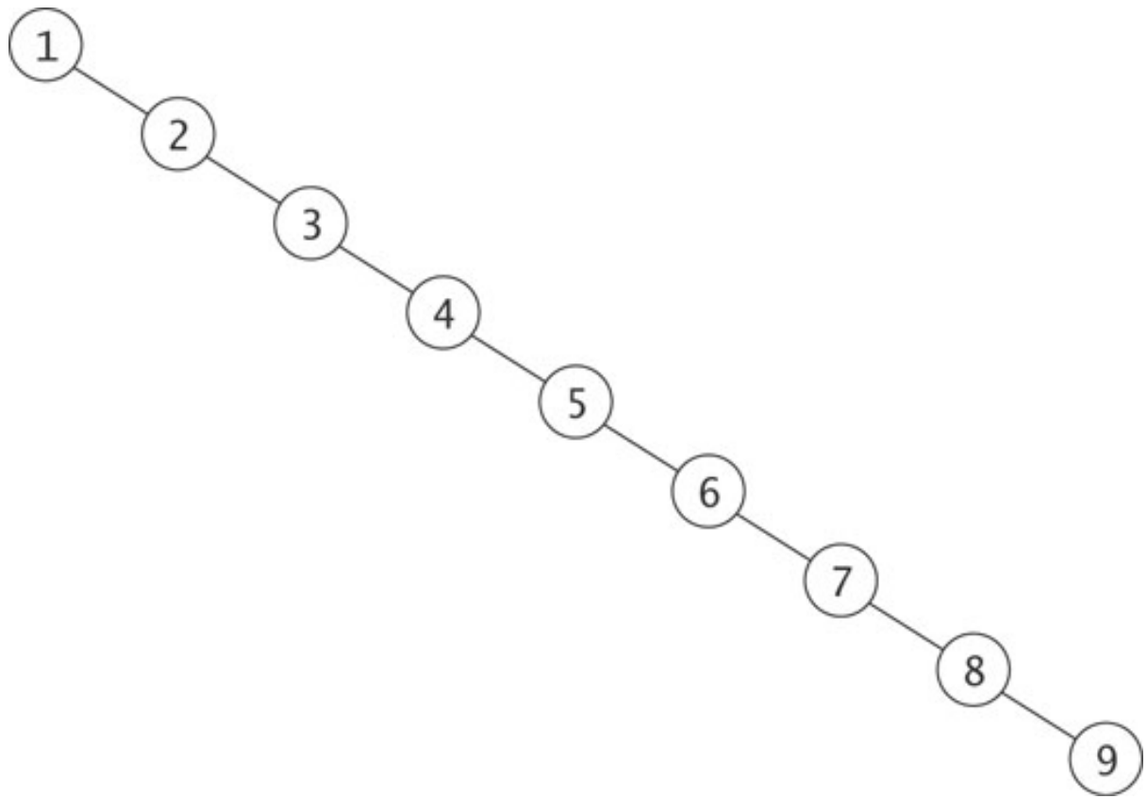
Iterativ metod *contains*(E *x*):

- BinaryNode<T> *t* = *root*
- repetera ända tills *t* == *null*:
  - om *x* == *t.element*: return *true*
  - om *x* < *t.element*: *t* = *t.left*
  - om *x* > *t.element*: *t* = *t.right*
- return *null*

Effektivitet

Om trädet är balanserat (eller nästan balanserat) blir sökningen logaritmisk, O(log n)

I värsta fall har trädet bara högerdelträd (eller vänster-), och då blir sökningen linjär, O(n)



Att lägga till element

Metod *insert*(T *x*):

- *root* = *insert*(*x*, *root*)

Rekursiv privat metod *insert*(T *x*, BinaryNode<T> *t*):

- om *t* == *null*: *t* = new BinaryNode<T>(*x*)
- om *x* < *t.element*: *t.left* = *insert*(*x*, *t.left*)
- om *x* > *t.element*: *t.right* = *insert*(*x*, *t.right*)
- return *t*

Hemläxa:

- formulera och implementera en iterativ variant

Att ta bort element

Först letar vi förstås upp noden som ska tas bort

- men vi ser till att komma ihåg dess förälder

Om noden inte har några barn:

- ta bort förälderns referens till noden

Om noden bara har ett barn:

- peka om förälderns referens till nodens enda barn

Om noden har två barn, så ersätter vi nodens data med det största elementet i det vänstra delträdet

- det är alltså inorder-föregångaren
- det går också att ersätta med det minsta elementet i det högra delträdet

Att ta bort element

Metod *remove*(T *x*):

- *root* = *remove*(*x*, *root*)

Rekursiv privat metod *remove*(T *x*, BinaryNode<T> *t*):

- om *t* == *null*: return *t*
- om *x* < *t.element*: *t.left* = *remove*(*x*, *t.left*)
- om *x* > *t.element*: *t.right* = *remove*(*x*, *t.right*)
- om *x* == *t.element*:
  - om *t.left* == *null*: *t* = *t.right*
  - om *t.right* == *null*: *t* = *t.left*
  - annars: *t.element* = *findMin*(*t.right*).*element*
t.right = *remove*(*t.element*, *t.right*)
- return *t*