Träd, binära träd och sökträd

Koffman & Wolfgang

- kapitel 6, avsnitt 1–4

Träd

Träd är *ickelinjära* och *hierarkiska*:

- i motsats till listor och fält
- en trädnod kan ha flera efterföljare (”barn”)
- men bara en föregångare (”förälder”)

Träd är en *rekursiv* datastruktur

Exempel på strukturer som kan lagras i träd:

- Javas klasshierarki
- en katalog på hårddisken, och dess underkataloger
- ett familjeträd

Binära träd

Vi fokuserar på *binära träd*:

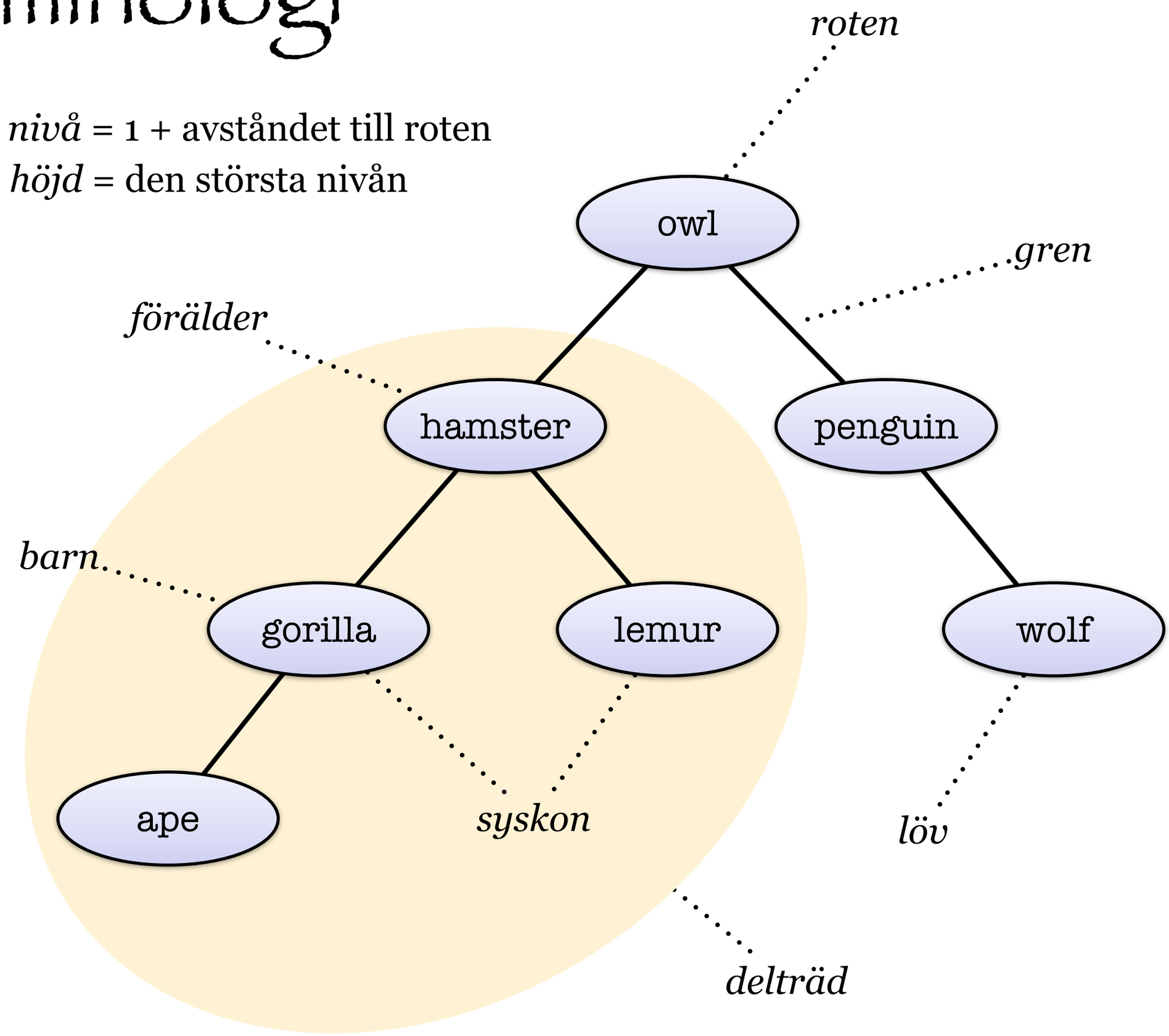
- varje nod har maximalt två barn

Binära träd kan implementeras som:

- en länkad struktur
  - binära sökträd, avsnitt 6.4
- ett fält
  - heap, avsnitt 6.5

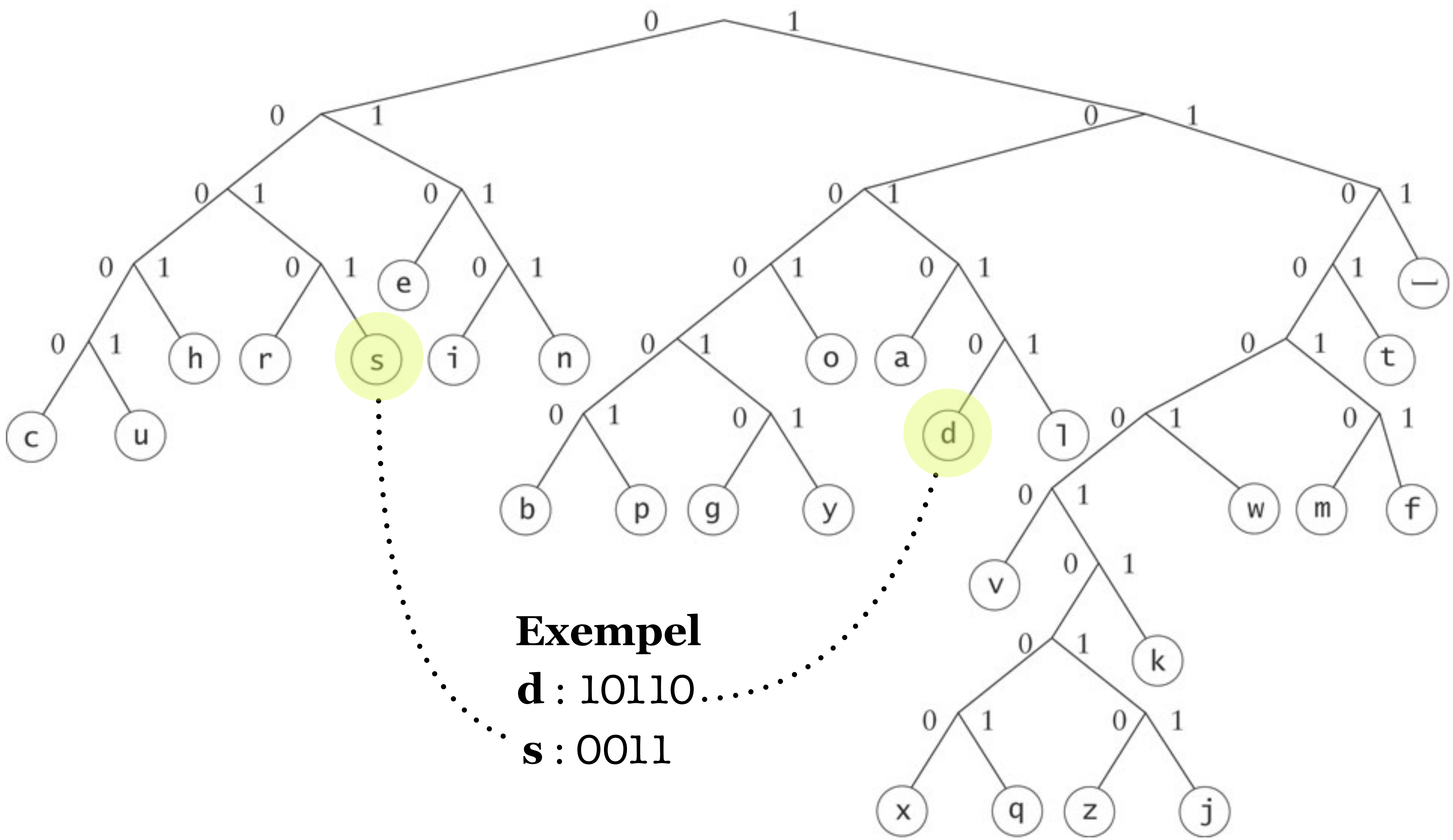
Terminologi

en nods *nivå* = 1 + avståndet till roten
en träds *höjd* = den största nivån

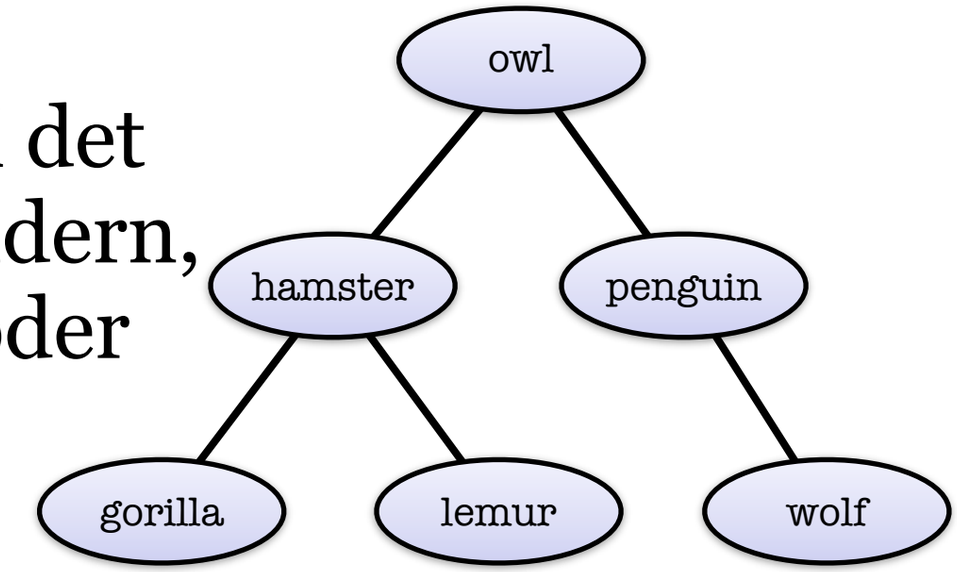


Exempel: Huffmanträd

Ett Huffmanträd används för att representera en Huffmankod



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:

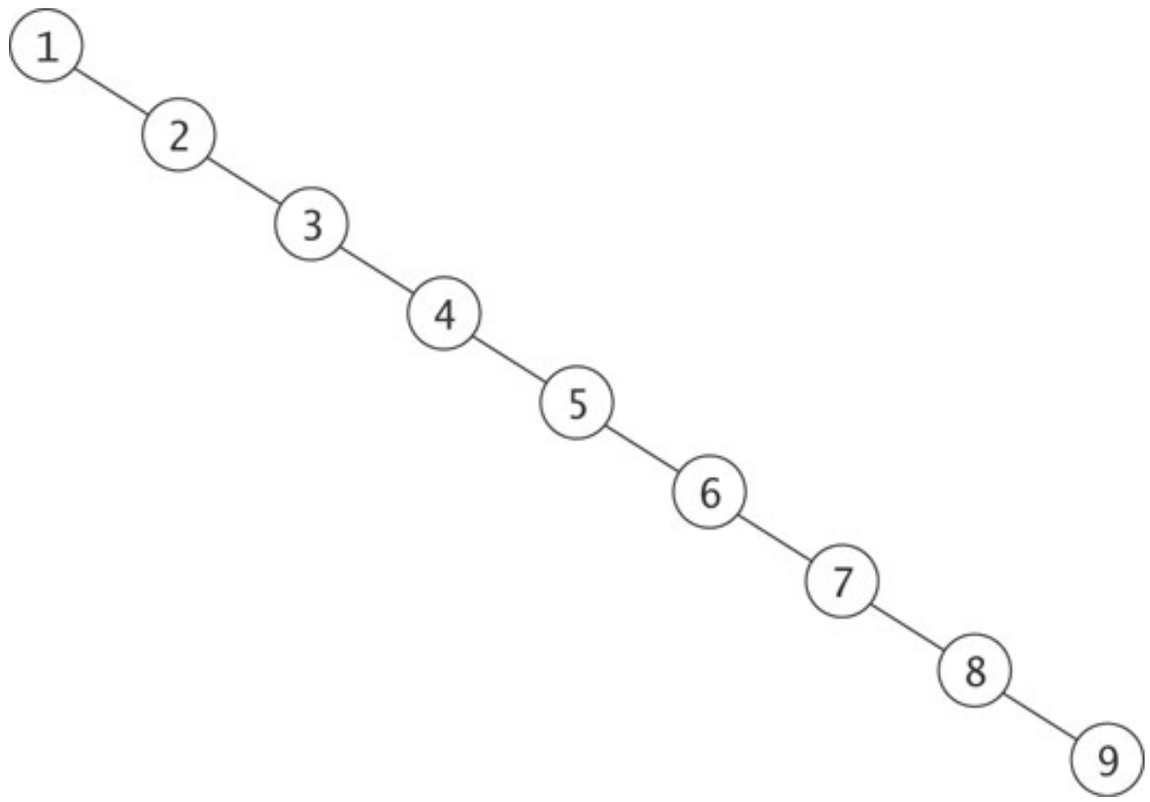
- om trädet är tomt:
  - return null
- annars, om *target* matchar rotnodens data:
  - return rotnodens data
- annars, om *target* är mindre än rotnodens data:
  - sök efter *target* i vänstra delträdet
- annars är *target* större än rotnodens data:
  - sök efter *target* i högra delträdet

Sökning i binära sökträd

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

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

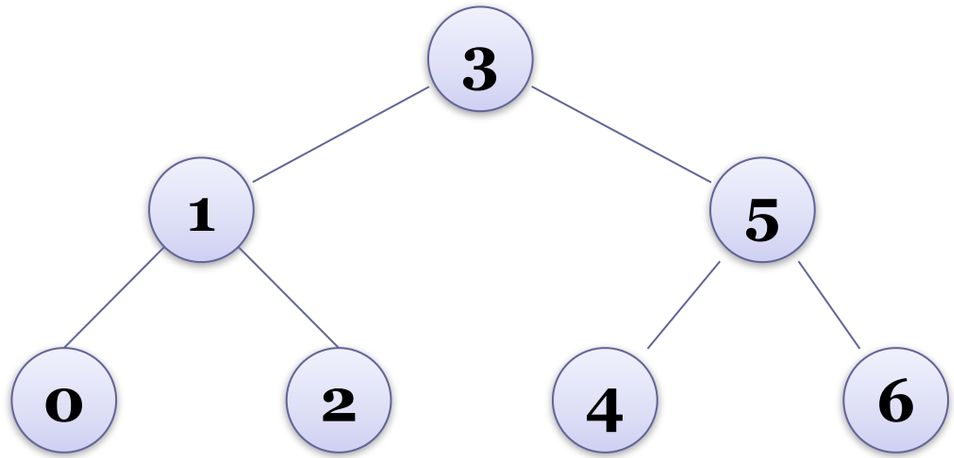
- om trädet är väldigt skevt:



Perfekta binära träd

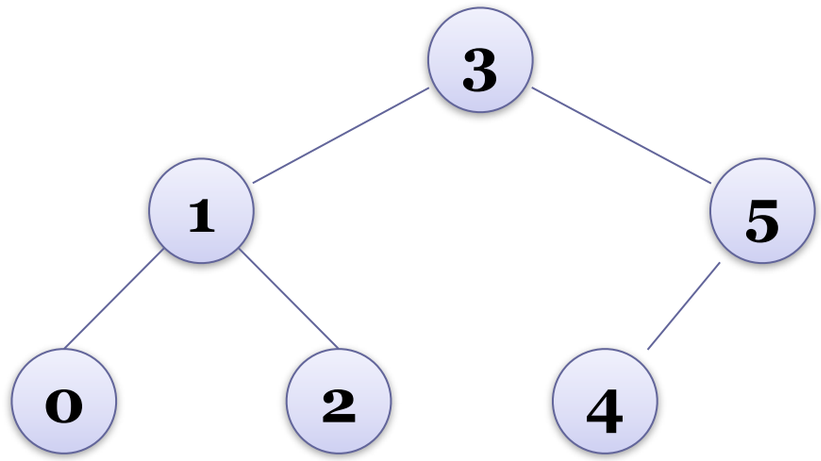
Ett *perfekt* binärt träd är ett binärt träd med höjd n som har exakt $2^n - 1$ noder

I detta fall är $n = 3$ och $2^n - 1 = 7$



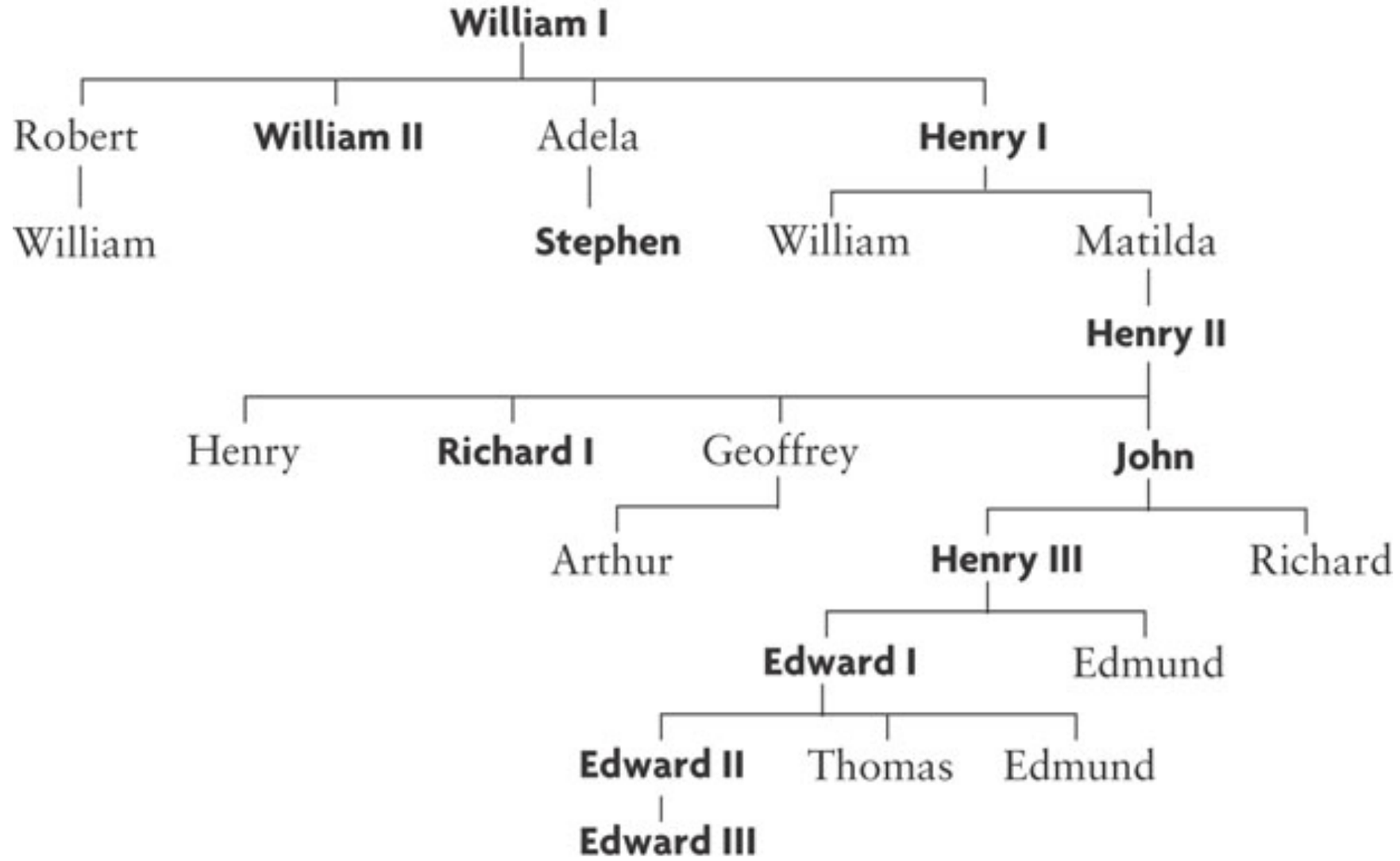
Nivåbalanserade binära träd

Ett *nivåbalanserat* (fullständigt/complete) binärt träd är ett perfekt binärt träd (ner till nivå $n-1$), med några extra löv på nivå n, där alla ligger till vänster



Generella träd

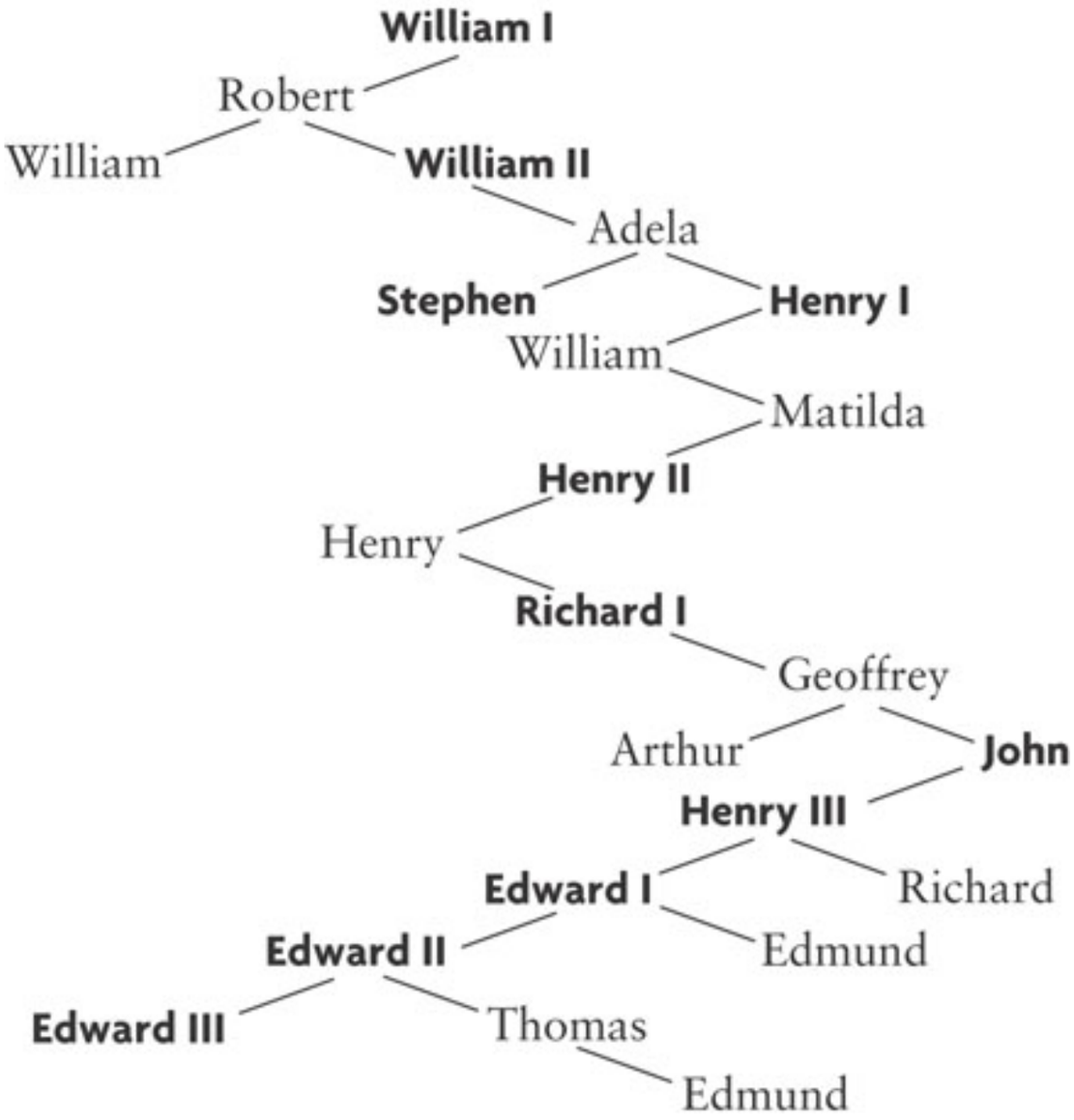
Noderna i generella träd kan ha hur många barn som helst



Binärisering av generella träd

Ett generellt träd kan representeras av ett binärt träd:

- första barnet blir ett vänsterbarn
- dess högersyskon blir ett högerbarn
- ...vars högersyskon blir högerbarn



Trädtraversering

Preorder:

- besök rotnoden, sedan T_L, sedan T_R

Inorder:

- besök T_L, sedan rotnoden, sedan T_R

Postorder:

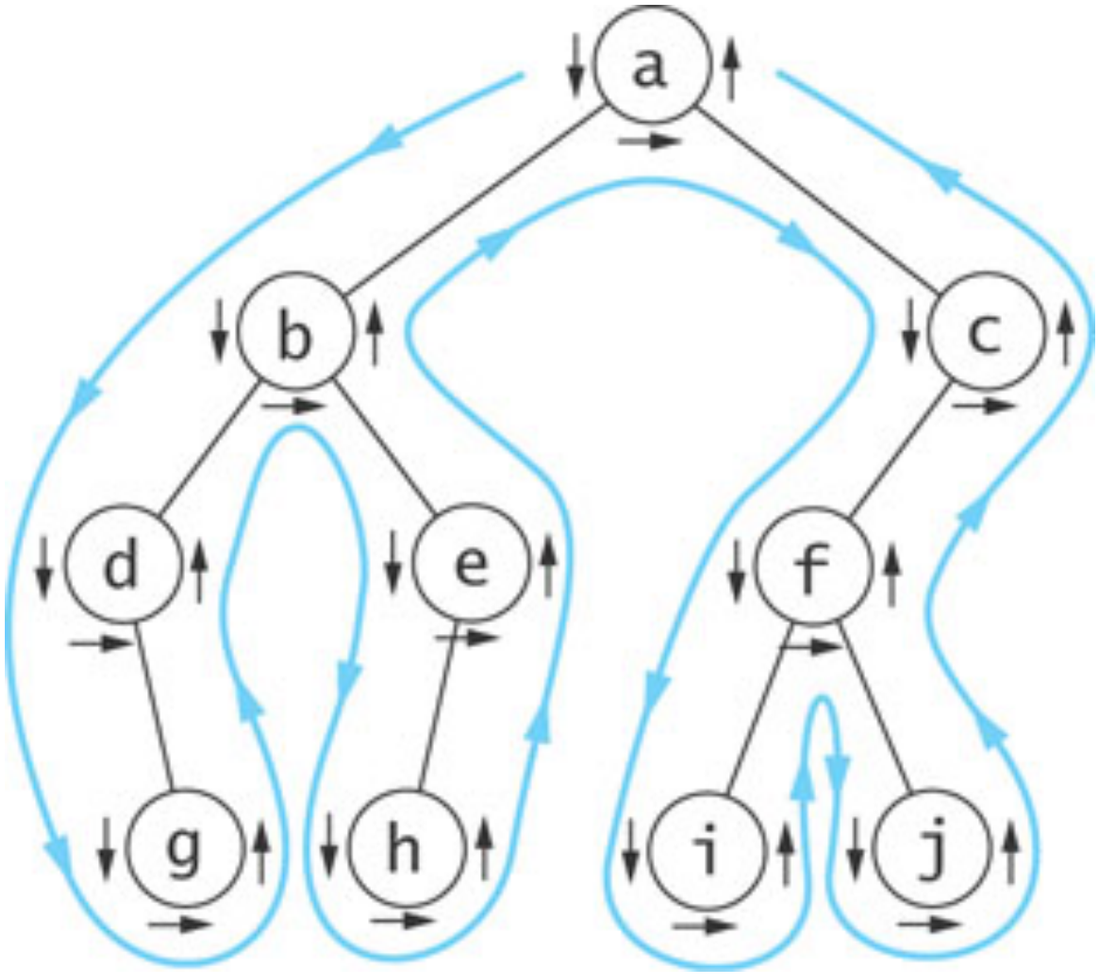
- besök T_L, sedan T_R, sedan rotnoden

Trädtraversering

”You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree

If the mouse always keeps the tree to the left, it will trace a route known as the *Euler tour*

The Euler tour is the path traced in blue in the figure on the right”



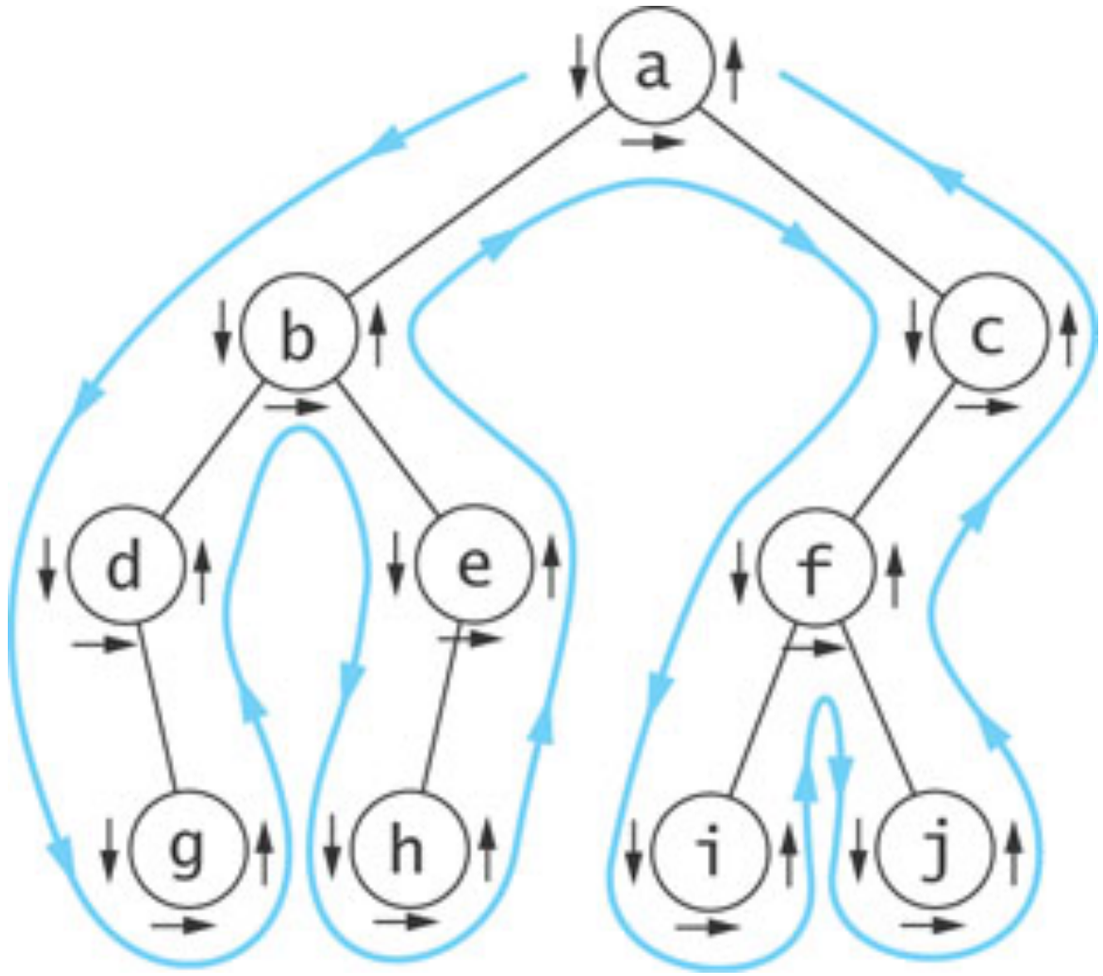
Preorder

Musen besöker varje nod *innan* den besöker delträden.

(De nedåtpekande pilarna).

I dethär fallet blir det:

- `a b d g e h c f i j`



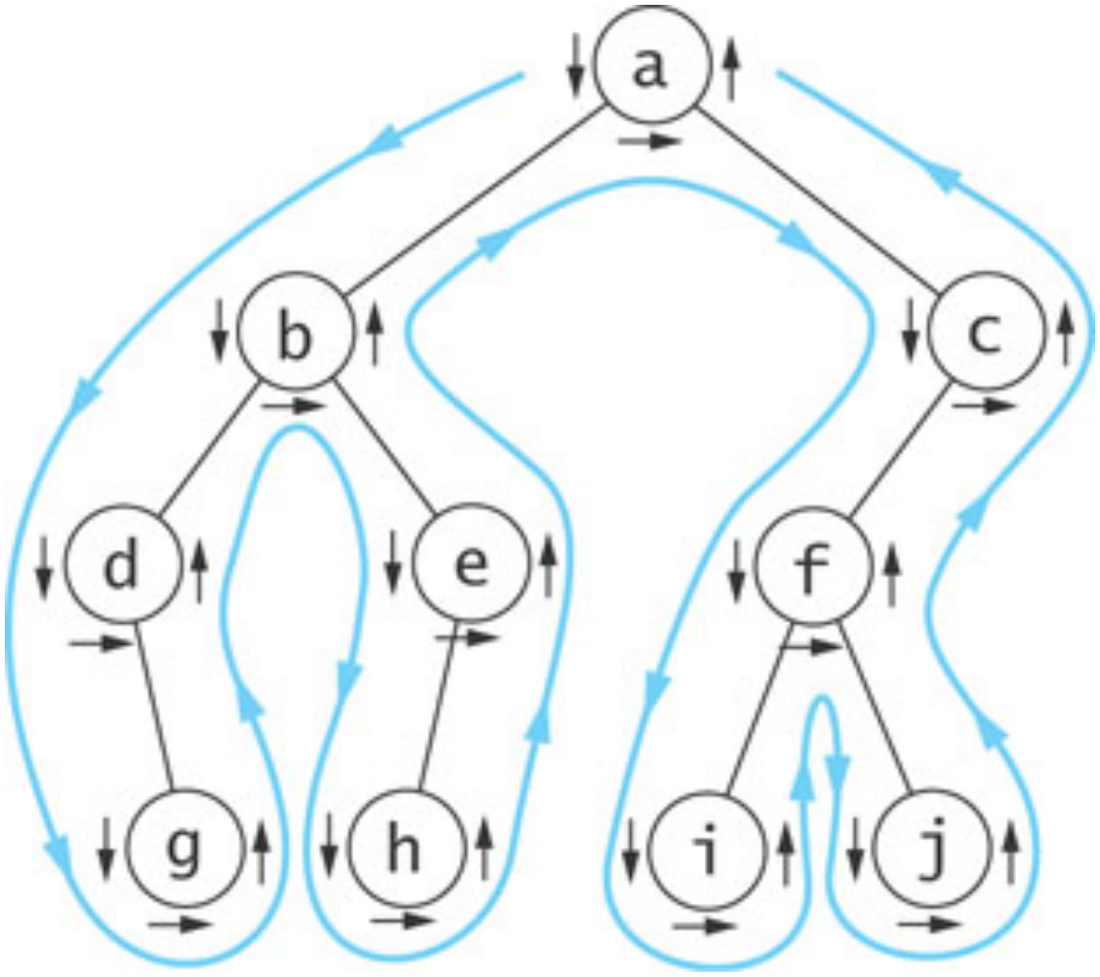
Inorder

Musen besöker först det vänstra delträdet, sedan noden, och sist det högra delträdet.

(Högerpilarna).

I dethär fallet blir det:

- `d g b h e a i f j c`



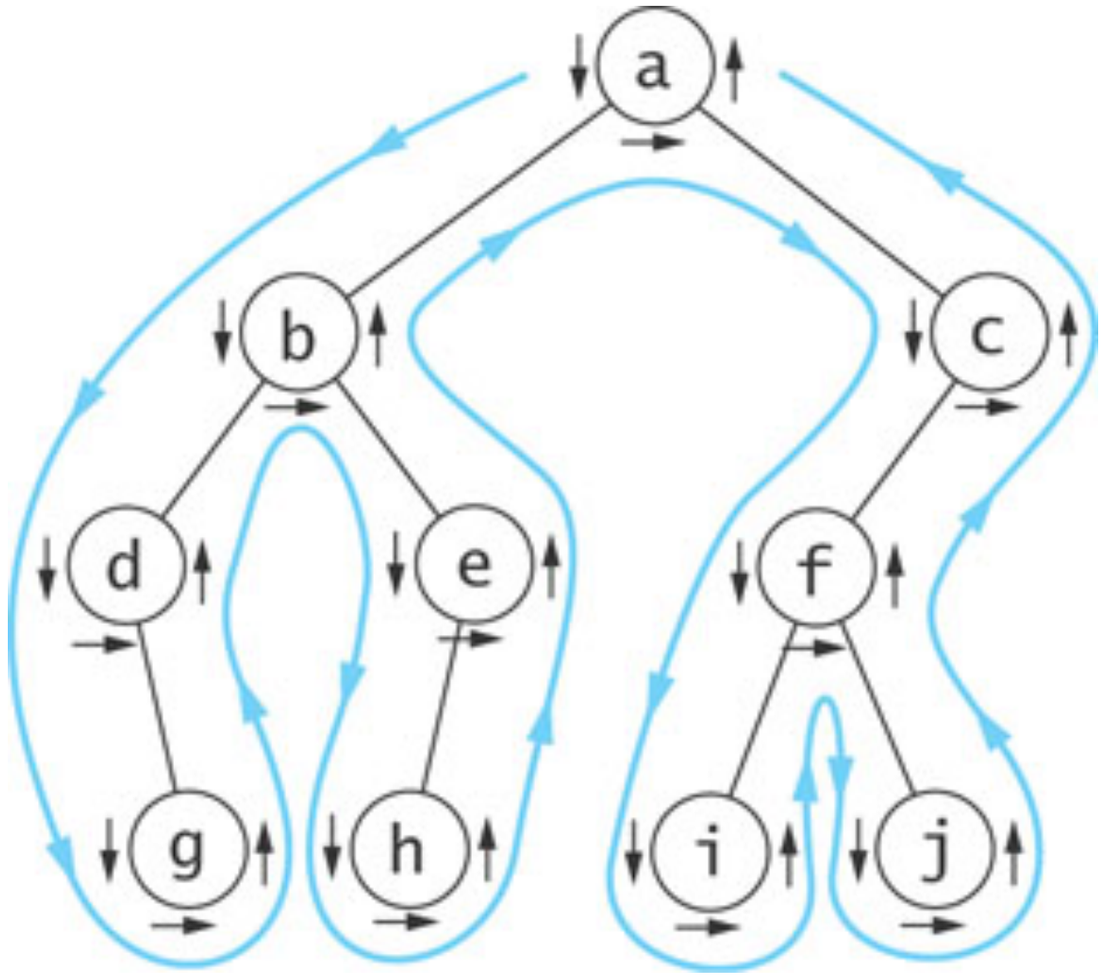
Postorder

Musen besöker noden *efter* att den har besökt delträden.

(De nedåtpekande pilarna).

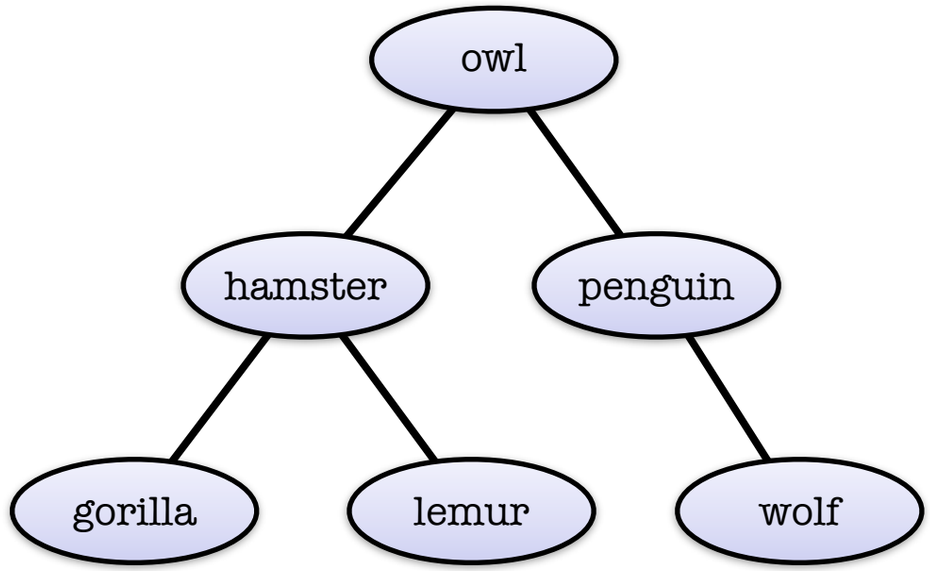
I dethär fallet blir det:

- `g d h e b i j f c a`



Inordertraversering av sökträd

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



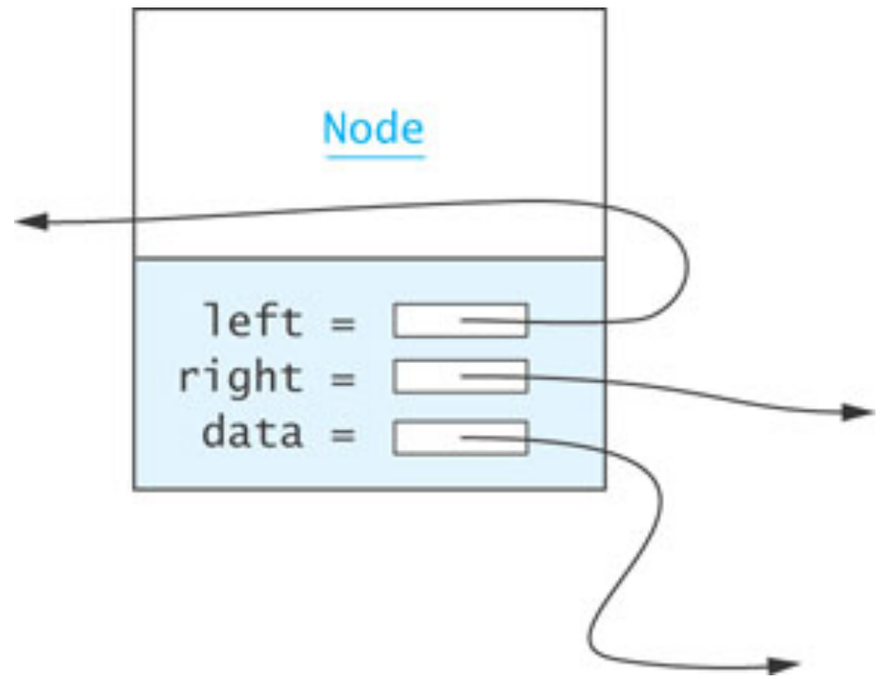
I detta fall:

gorilla, hamster, lemur, owl, penguin, wolf

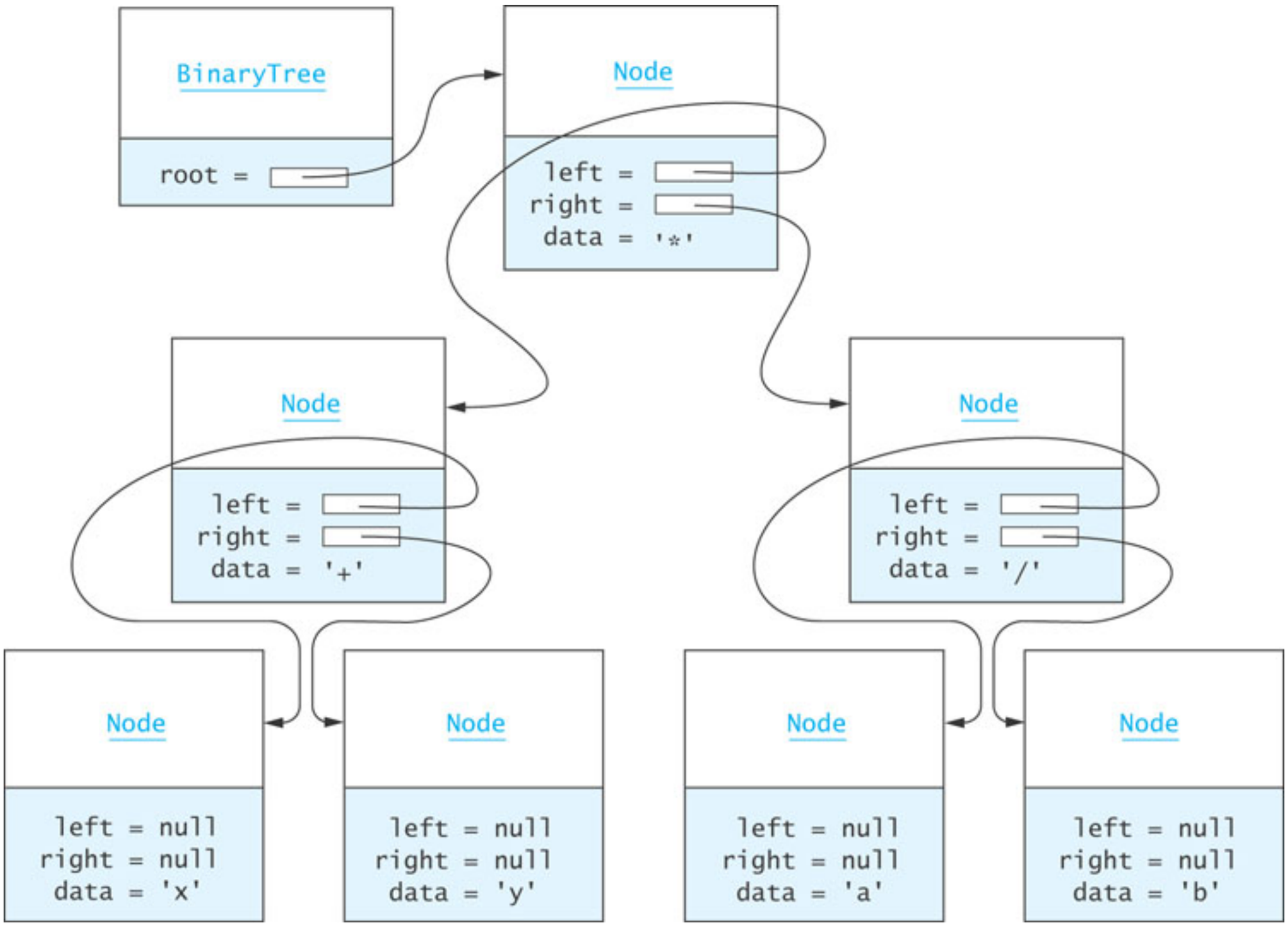
Att implementera binära träd

Klassen `Node<E>`:

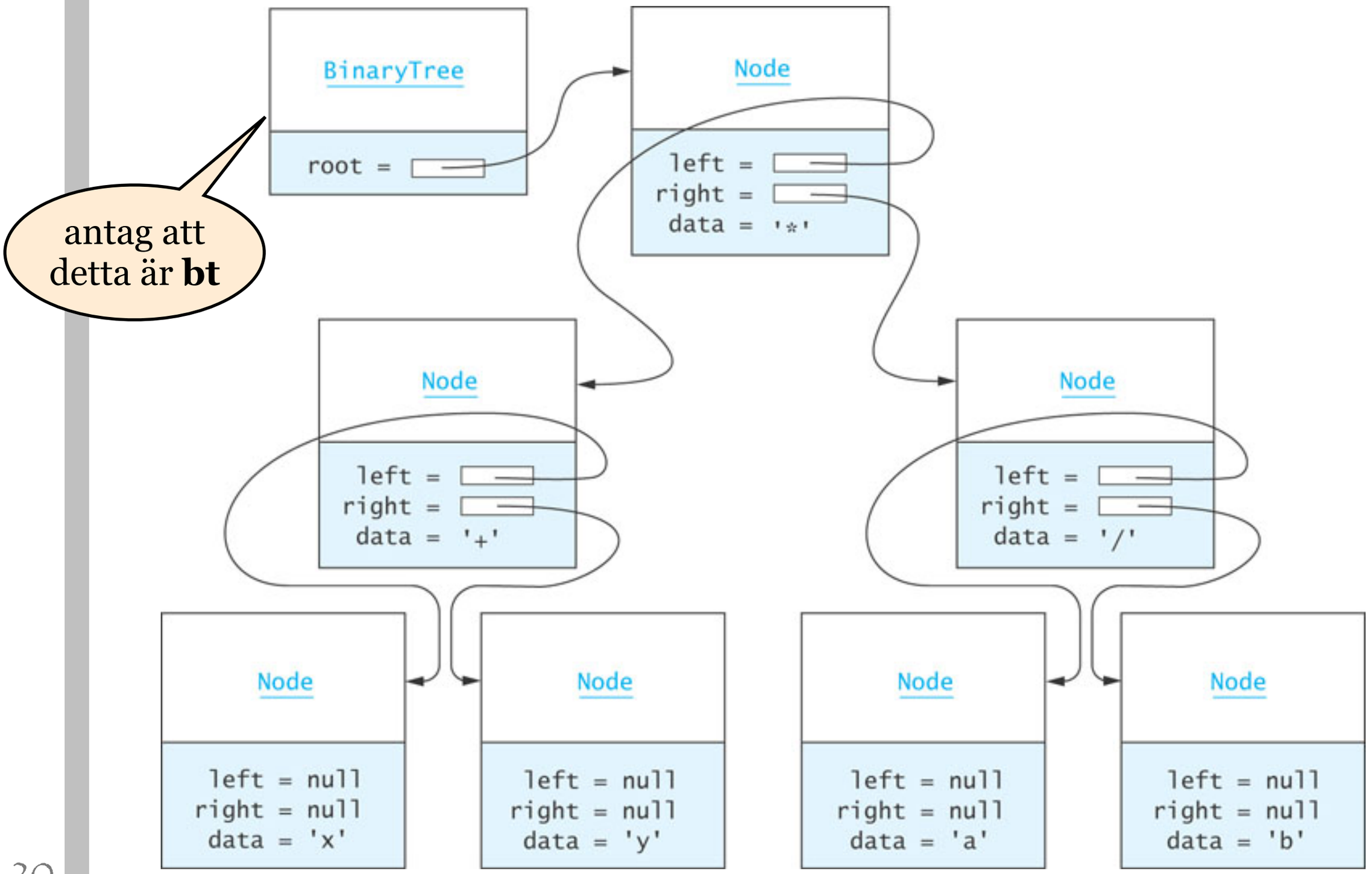
- en nod består av länkar till sina efterföljare
- precis som länkade listor
- fast nu är det *två* länkar istället för en



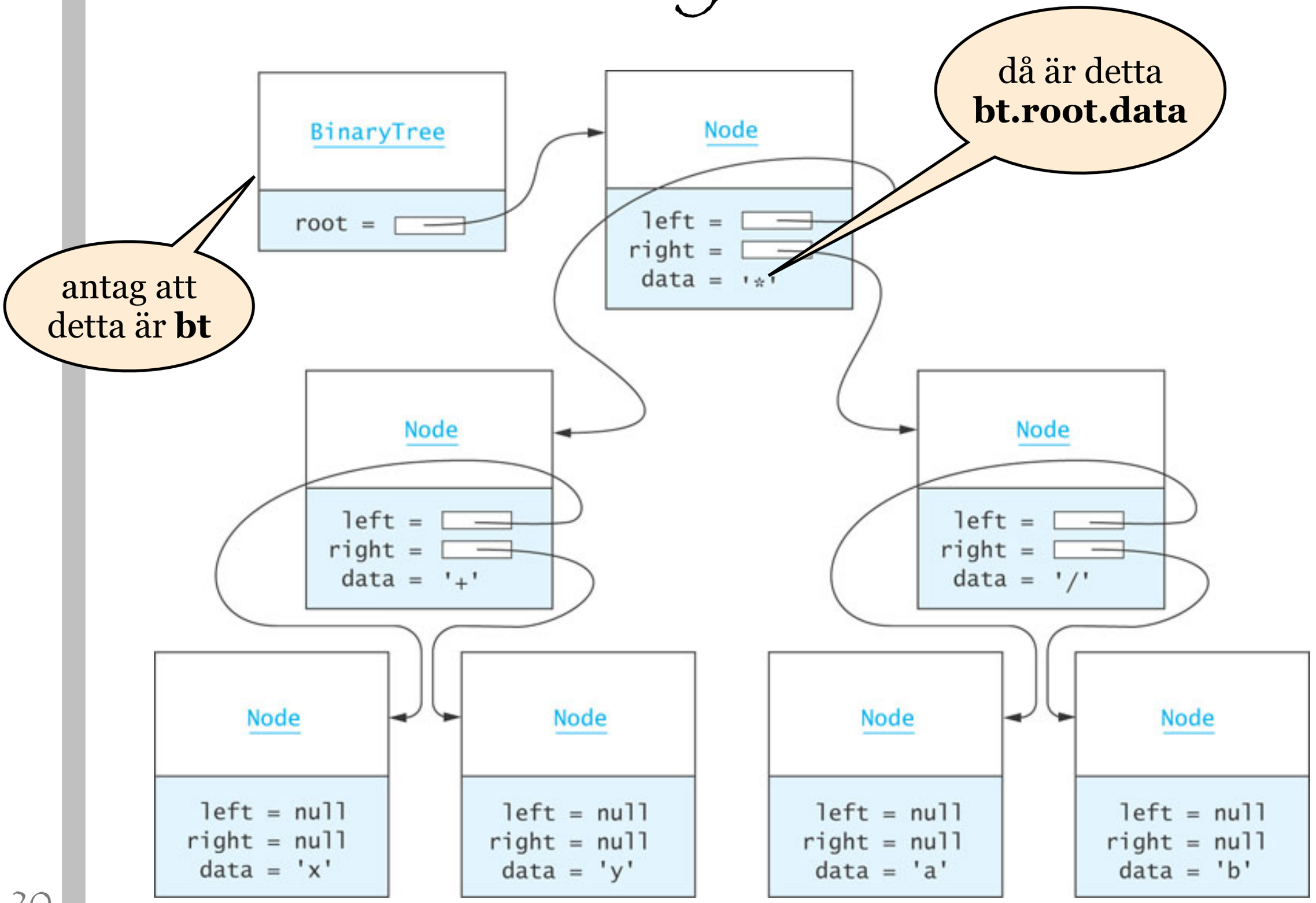
Klassen BinaryTree<E>



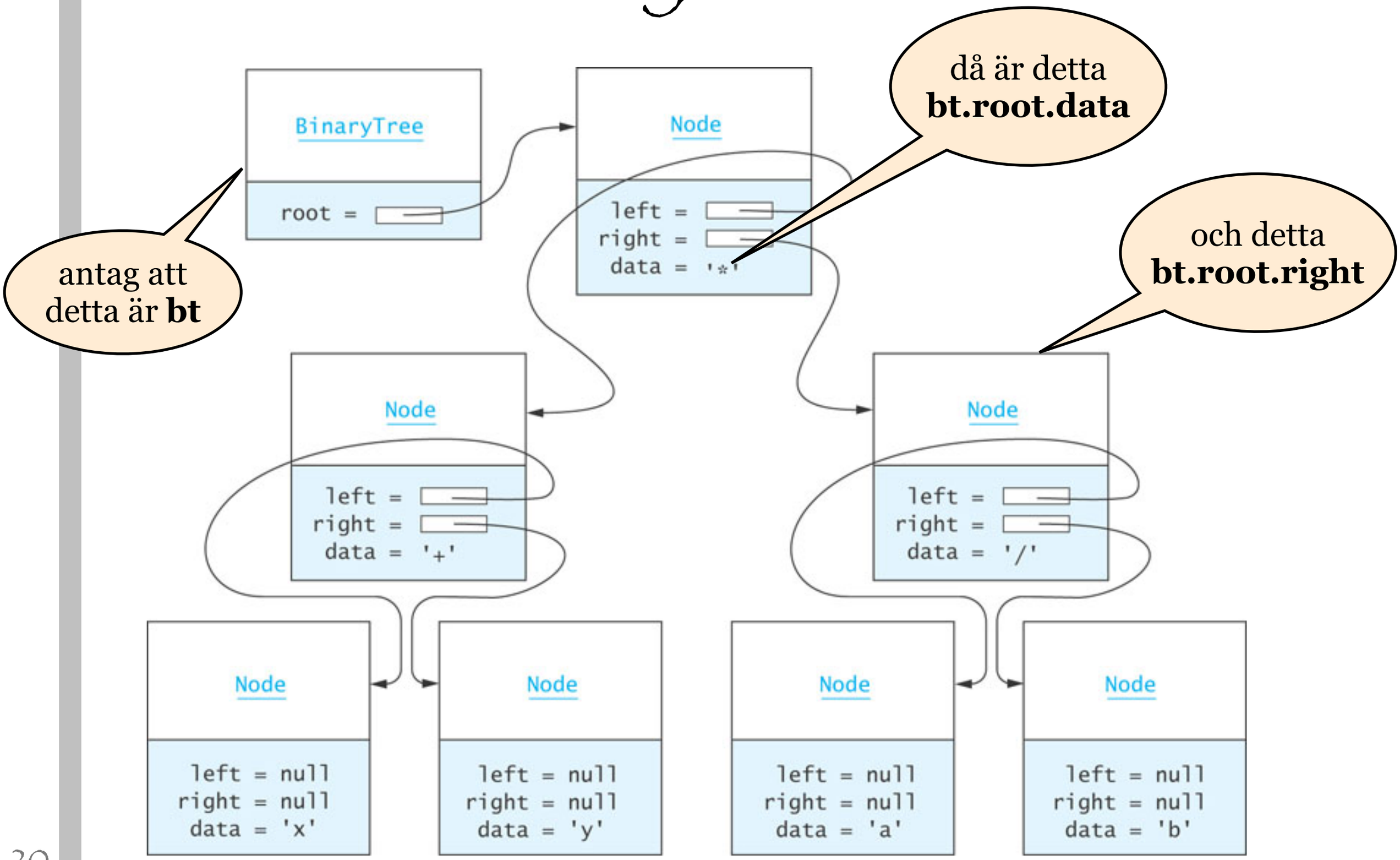
Klassen BinaryTree<E>



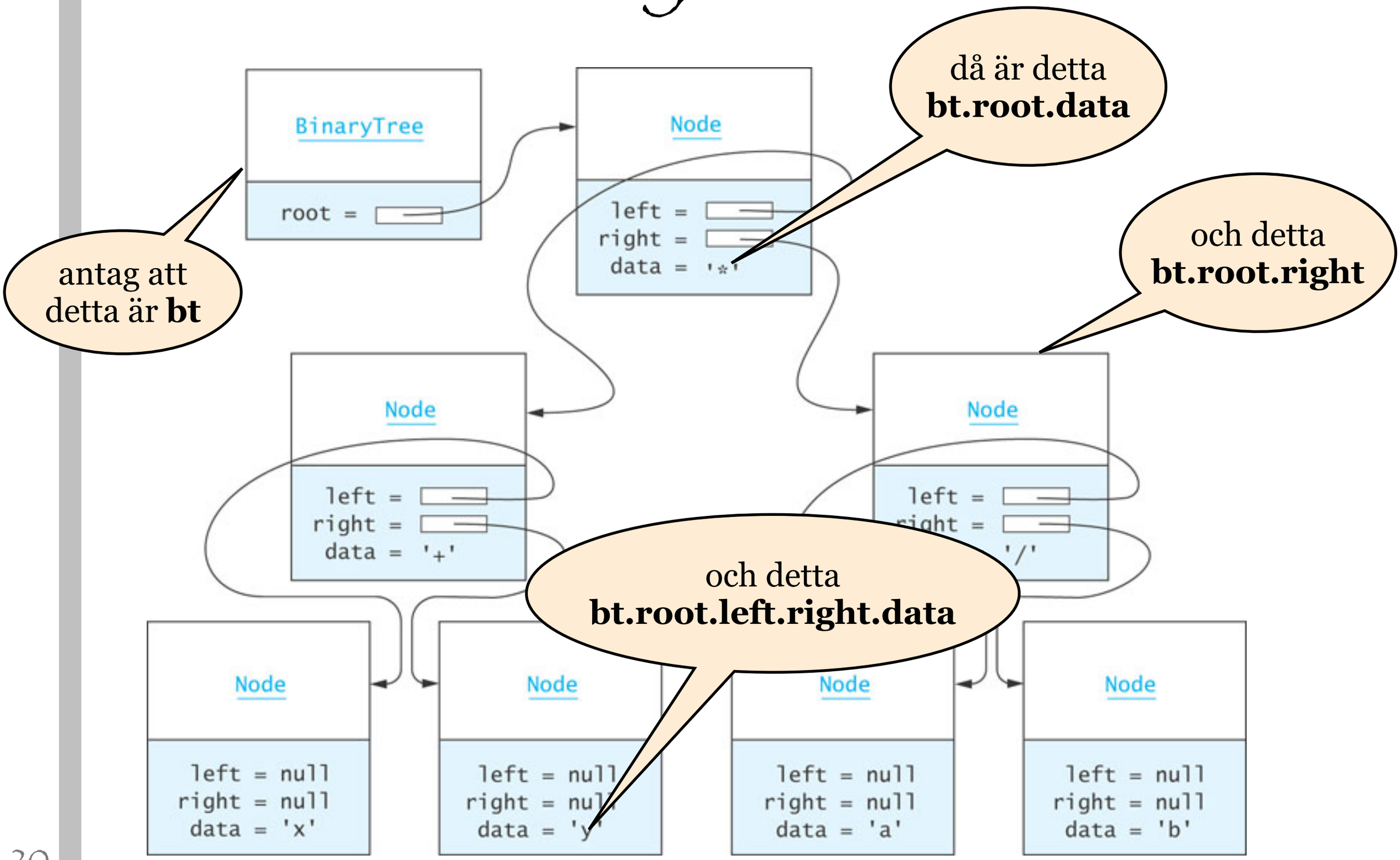
Klassen BinaryTree<E>



Klassen BinaryTree<E>



Klassen BinaryTree<E>



Klassen BinaryTree<E>

Data Field	Attribute
`protected Node<E> root`	Reference to the root of the tree.
Constructor	**Behavior**
`public BinaryTree()`	Constructs an empty binary tree.
`protected BinaryTree(Node<E> root)`	Constructs a binary tree with the given node as the root.
`public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E> rightTree)`	Constructs a binary tree with the given data at the root and the two given subtrees.

Method	Behavior
`public BinaryTree<E> getLeftSubtree()`	Returns the left subtree.
`public BinaryTree<E> getRightSubtree()`	Returns the right subtree.
`public E getData()`	Returns the data in the root.
`public boolean isLeaf()`	Returns **`true`** if this tree is a leaf, **`false`** otherwise.
`public String toString()`	Returns a `String` representation of the tree.
`private void preOrderTraverse(Node<E> node, int depth, StringBuilder sb)`	Performs a preorder traversal of the subtree whose root is `node`. Appends the representation to the `StringBuilder`. Increments the value of `depth` (the current tree level).
`public static BinaryTree<E> readBinaryTree(Scanner scan)`	Constructs a binary tree by reading its data using `Scanner scan`.

Konstruerare

```
public BinaryTree( E data,
                   BinaryTree<E> leftTree,
                   BinaryTree<E> rightTree)
{
    root = new Node<E>(data);

    if (leftTree != null)
        root.left = leftTree.root;
    else
        root.left = null;

    if (rightTree != null)
        root.right = rightTree.root;
    else
        root.right = null;
}
```

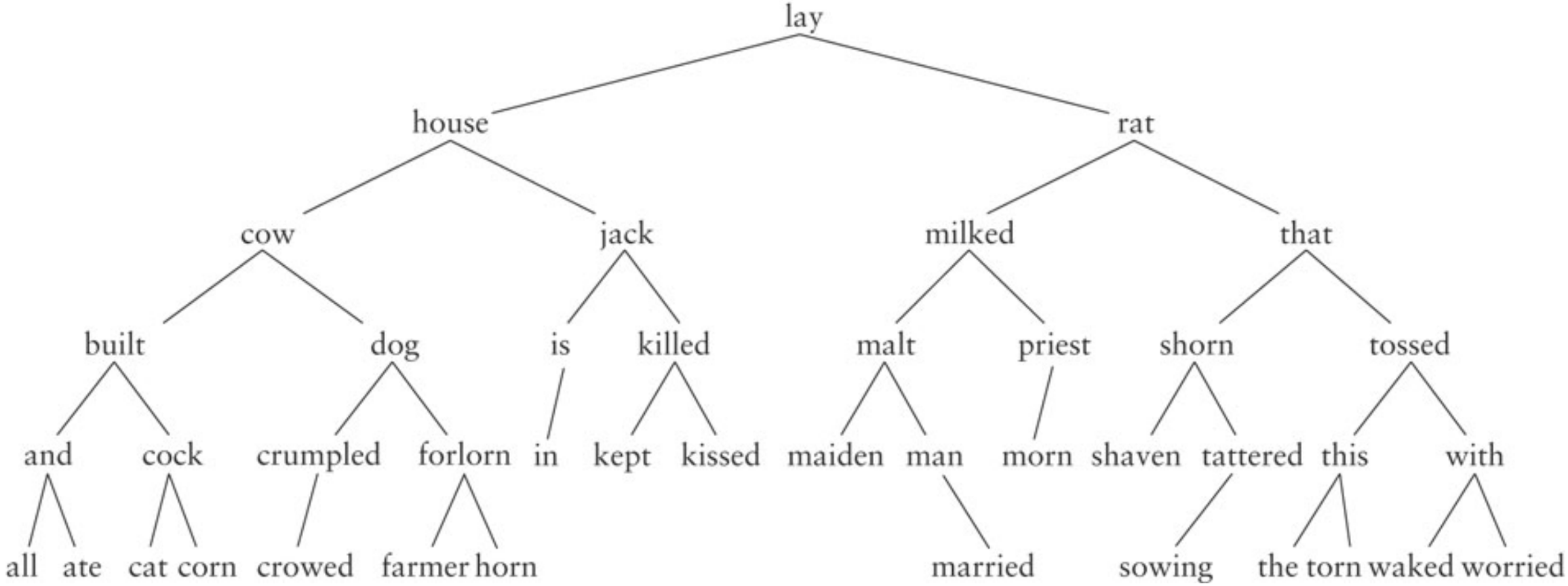
Konstruerare

```
public BinaryTree( E data,
                   BinaryTree<E> leftTree,
                   BinaryTree<E> rightTree)
{
    root = new Node<E>(data);

    if (leftTree != null)
        root.left = leftTree.root;
    else
        root.left = null;

    if (rightTree != null)
        root.right = rightTree.root;
    else
        root.right = null;
}
```

i de flesta trädmetoder behöver vi kontrollera om någon pekare är **null**

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



Sökning i ett binärt sökträd

Algoritm *search*(BinaryTree<E> *tree*, E *target*):

- return *searchNode*(*tree.root*)

Rekursiv privat algoritm *searchNode*(Node<E> *node*):

- om *node* == *null*: return *null*
- om *target* == *node.data*: return *node.data*
- om *target* < *node.data*: return *searchNode*(*node.left)*
- om *target* > *node.data*: return *searchNode*(*node.right)*

Jämför likheten med binärsökning i ett fält.
(föreläsningen om rekursion).

Iterativ sökning

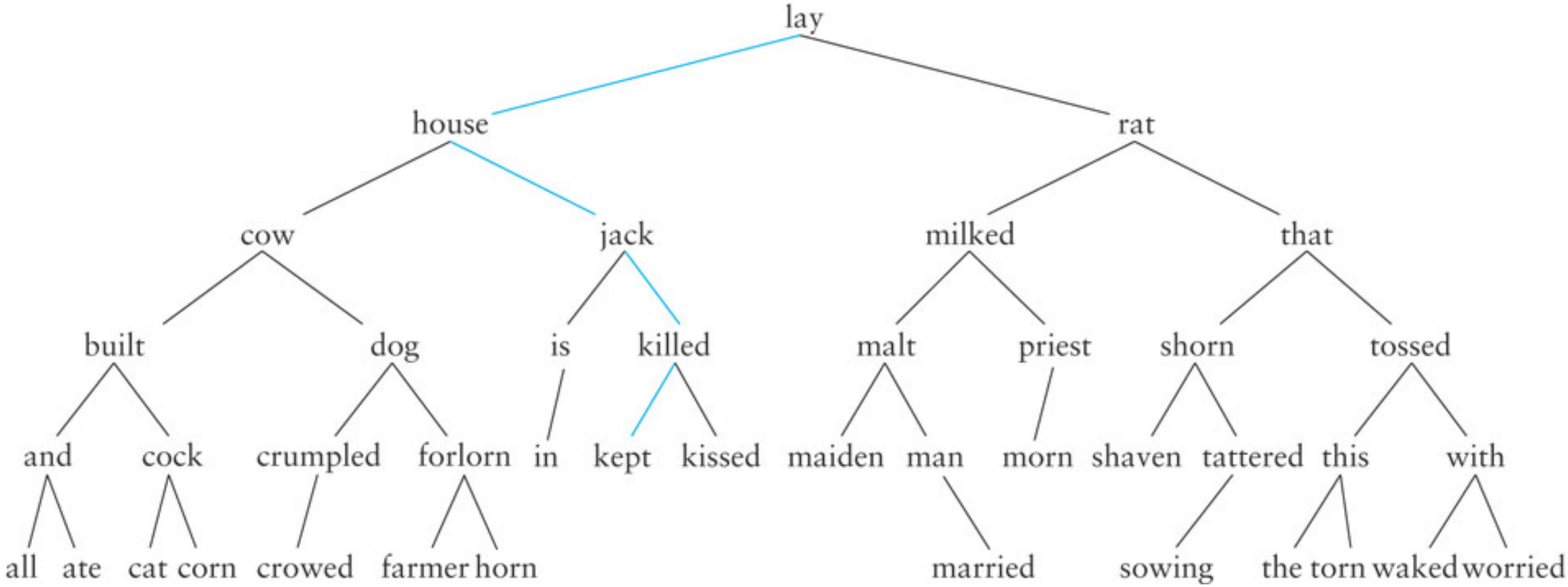
Precis som med binärsökning i fält så går det att implementera iterativt

Iterativ algoritm *search*(BinaryTree<E> *tree*, E *target*):

- *node* = *tree.root*
- repetera ända tills *node* == *null*:
  - om *target* == *node.data*: return *node.data*
  - om *target* < *node.data*: *node* = *node.left*
  - om *target* > *node.data*: *node* = *node.right*
- return *null*

Sökning, exempel

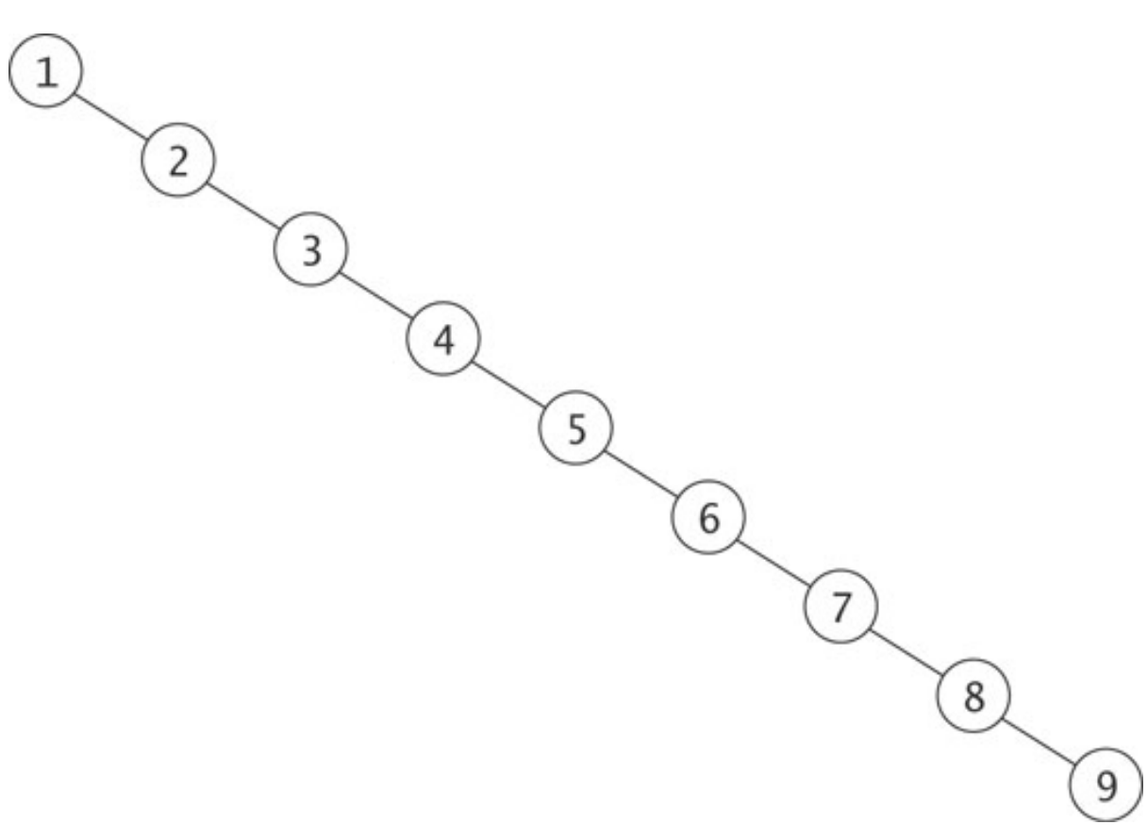
Att söka efter ”kept” i sökträdet kräver bara 4 jämförelser.



Effektivitet

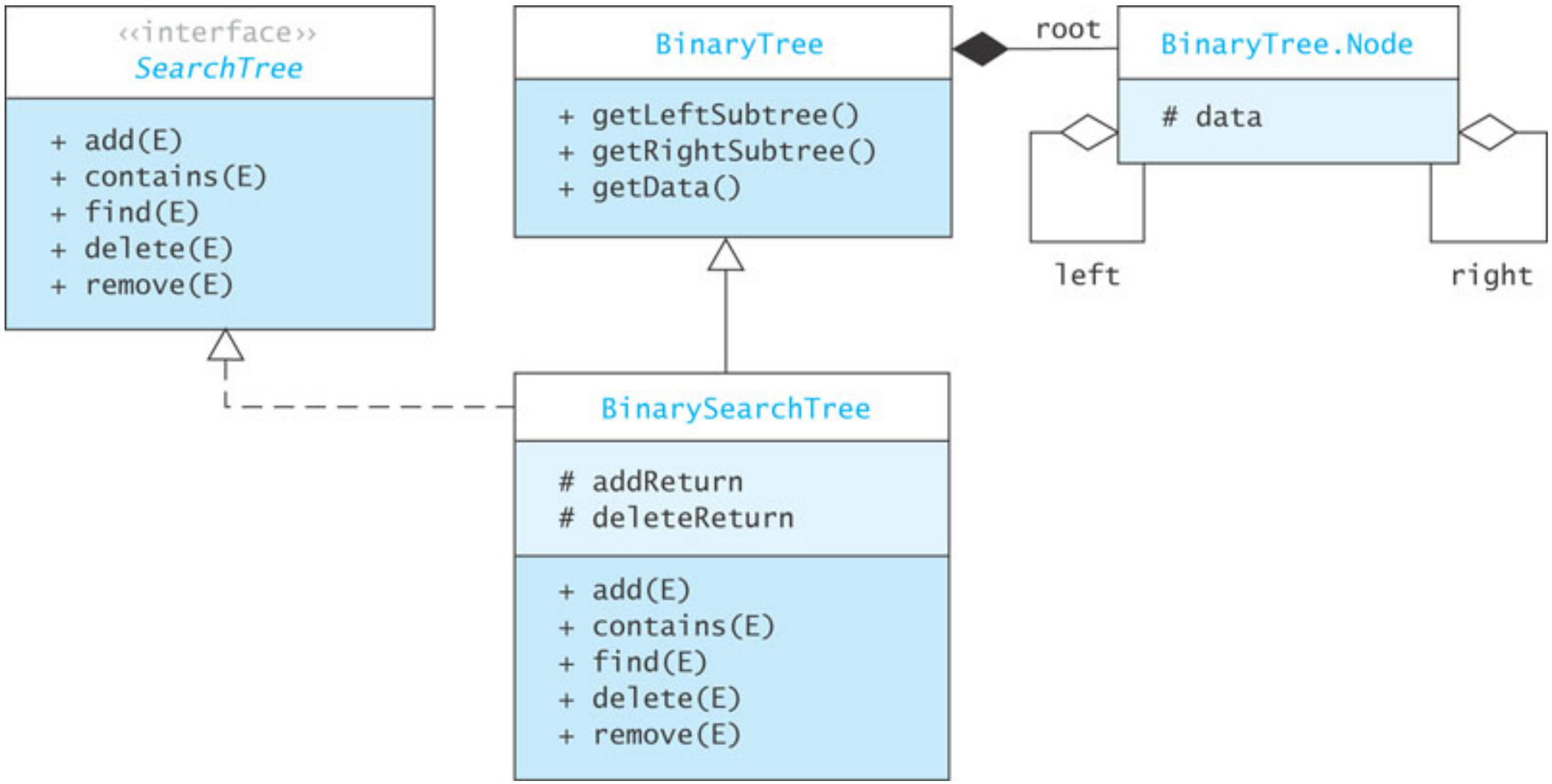
Om trädet är balanserat (eller nästan balanserat) blir sökningen logaritmisk, O(log n)

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



Klassen BinarySearchTree<E>

Figur 6.15, sid 318

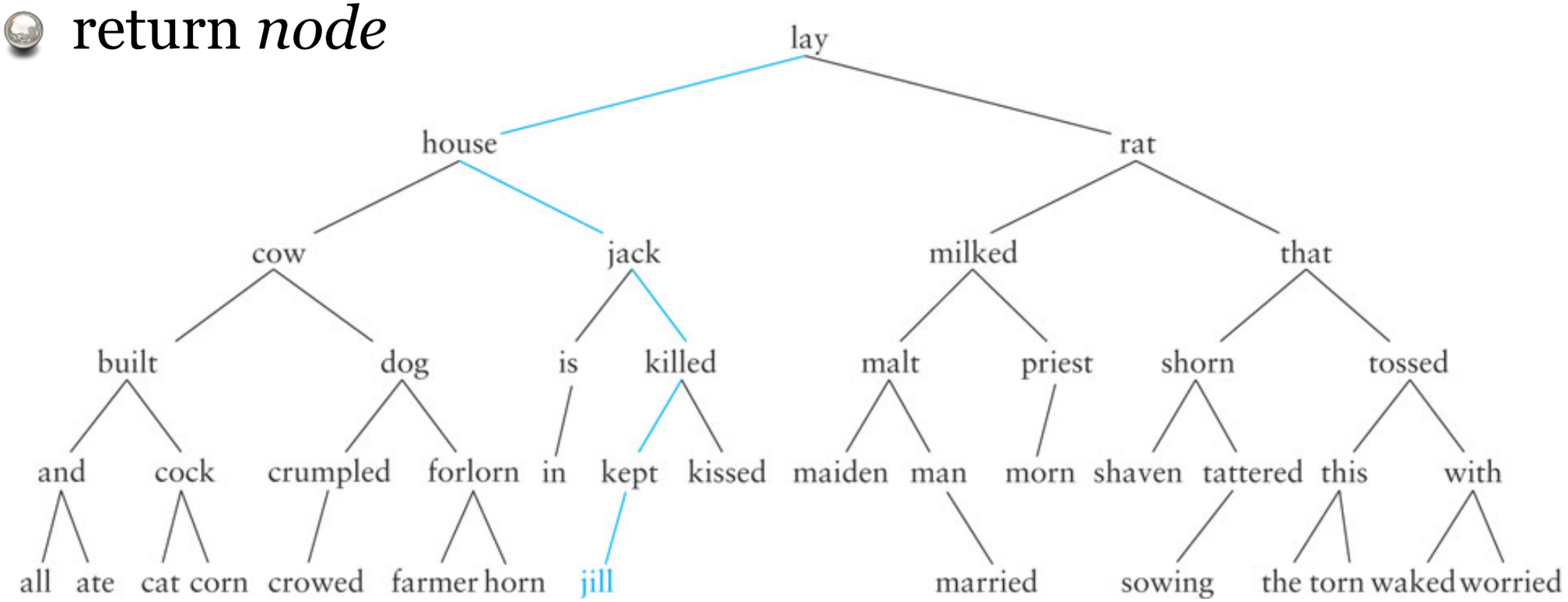


Att lägga till element

Metoden *add*(E *item*) anropar metoden *add*(*root*, *item*).

Rekursiv algoritm *add*(Node<E> *node*, E *item*):

- om *node* == *null*: *node* = new Node<E>(*item*)
- om *item* < *node.data*: *node.left* = *add*(*node.left*, *item*)
- om *item* > *node.data*: *node.right* = *add*(*node.right*, *item*)
- return *node*



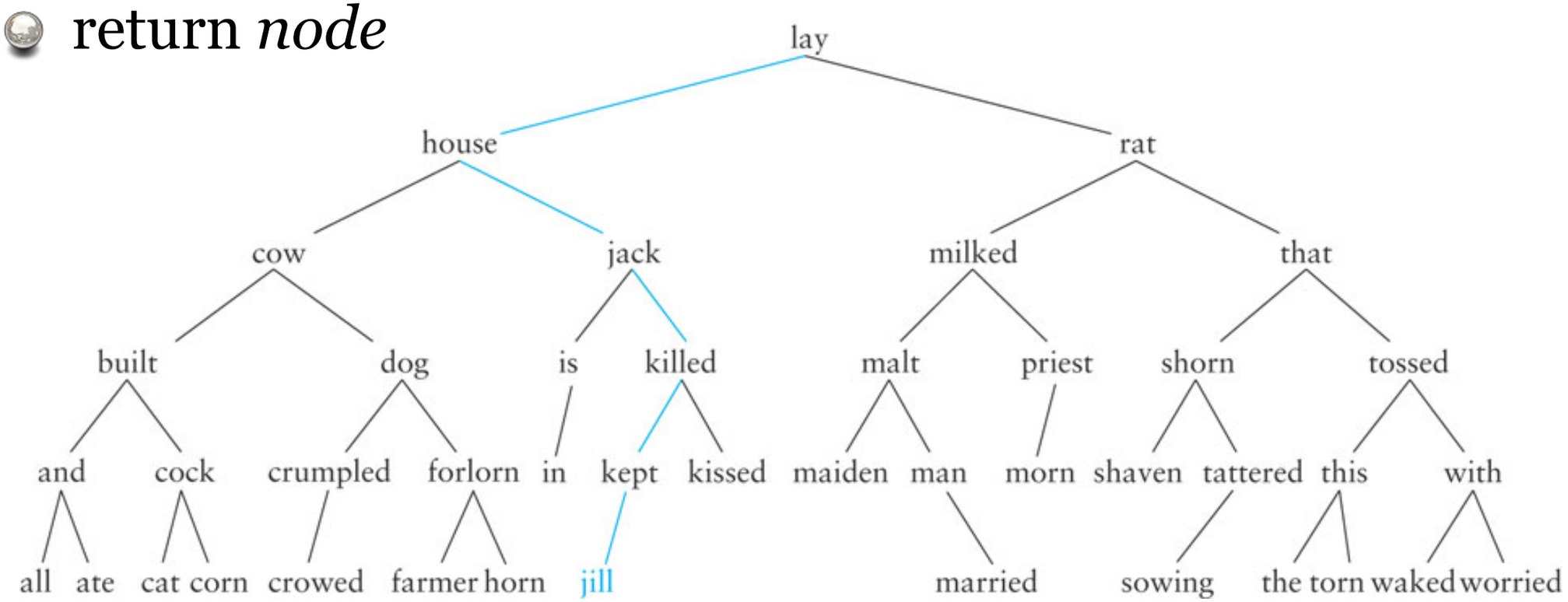
Att lägga till element

hemläxa:
formulera och implementera en iterativ varint

Metoden *add*(E *item*) anropar metoden *add*(*root*, *item*).

Rekursiv algoritm *add*(Node<E> *node*, E *item*):

- om *node* == *null*: *node* = new Node<E>(*item*)
- om *item* < *node.data*: *node.left* = *add*(*node.left*, *item*)
- om *item* > *node.data*: *node.right* = *add*(*node.right*, *item*)
- return *node*



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:

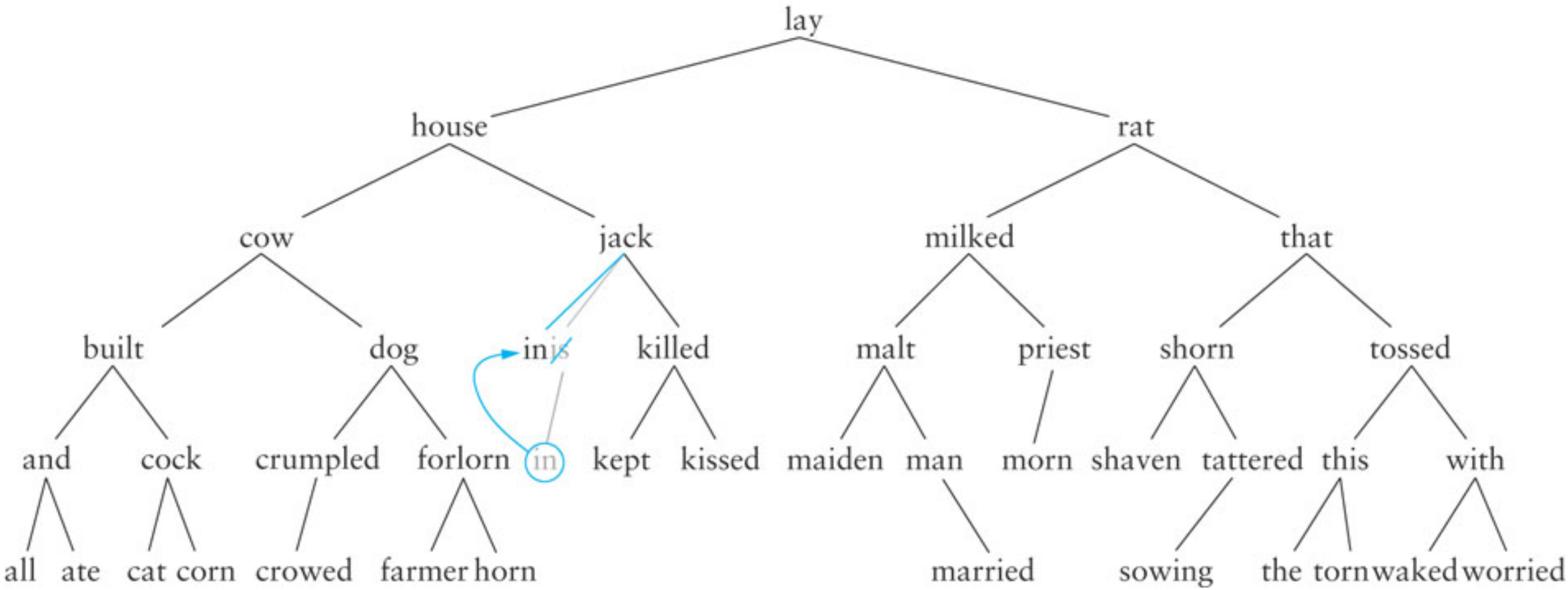
- ta bort förälderns referens till noden

Om noden bara har ett barn:

- peka om förälderns referens till nodens enda barn

Ta bort en nod med ett barn

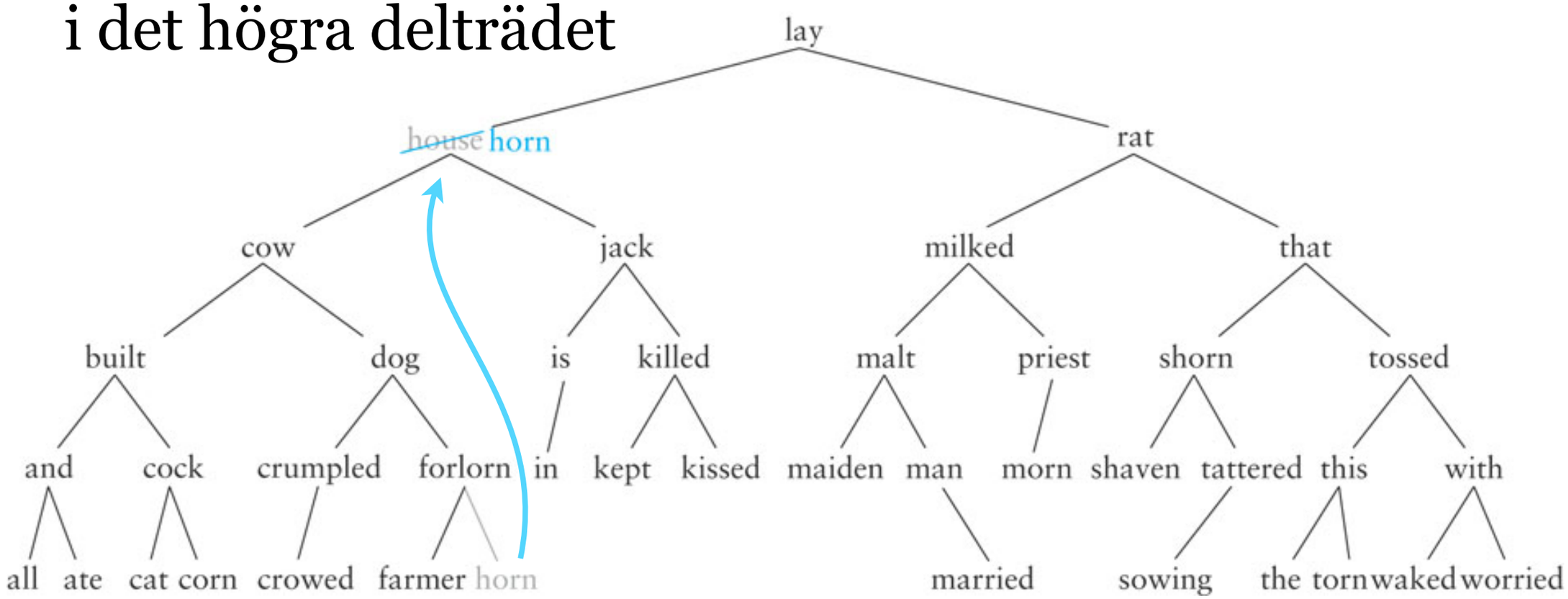
Noden ”*is*” har ett enda barn



Ta bort en nod med två barn

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

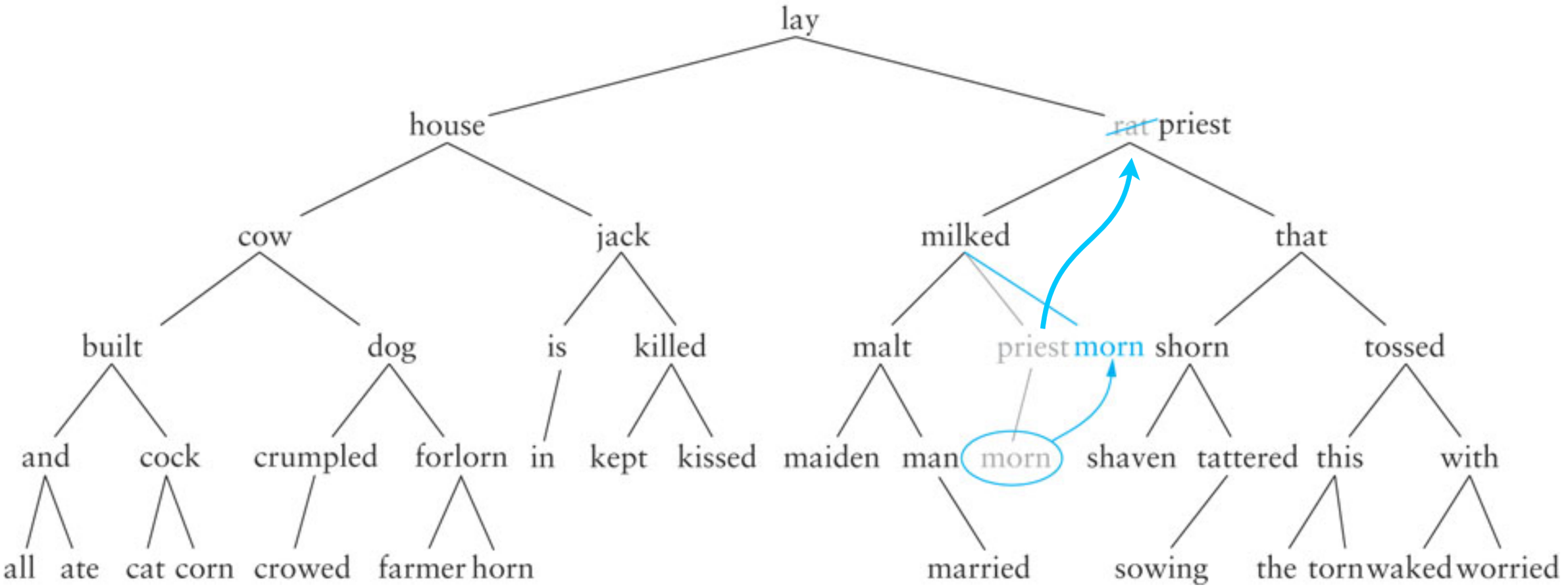
- det är alltså inorder-föregångaren
- det går också att ersätta med det minsta elementet i det högra delträdet



Ännu ett exempel

”*rat*” har två barn och ersätts av ”*priest*”

- men ”*priest*” har ett vänsterbarn, ”*morn*”, som då måste flyttas upp



Algoritm för att ta bort element

Recursive Algorithm for Removal from a Binary Search Tree

1. `if` the root is null
2. The item is not in tree – return `null`.
3. Compare the item to the data at the local root.
4. `if` the item is less than the data at the local root
5. Return the result of deleting from the left subtree.
6. `else if` the item is greater than the local root
7. Return the result of deleting from the right subtree.
8. `else` *// The item is in the local root*
9. Store the data in the local root in `deletedReturn`.
10. `if` the local root has no children
11. Set the parent of the local root to reference `null`.
12. `else if` the local root has one child
13. Set the parent of the local root to reference that child.
14. `else` *// Find the inorder predecessor*
15. `if` the left child has no right child it is the inorder predecessor
16. Set the parent of the local root to reference the left child.
17. `else`
18. Find the rightmost node in the right subtree of the left child.
19. Copy its data into the local root's data and remove it by setting its parent to reference its left child.

Algoritm för att ta bort element

Recursive Algorithm for Removal from a Binary Search Tree

```
1.  if the root is null
2.      The item is not in tree – return null.
3.  Compare the item to the data at the local root.
4.  if the item is less than the data at the local root
5.      Return the result of deleting from the left subtree.
6.  else if the item is greater than the local root
7.      Return the result of deleting from the right subtree.
8.  else // The item is in the local root
9.      Store the data in the local root in deletedReturn.
10.     if the local root has no children
11.         Set the parent of the local root to reference null.
12.     else if the local root has one child
13.         Set the parent of the local root to reference that child.
14.     else // Find the inorder predecessor
15.         if the left child has no right child it is the inorder predecessor
16.             Set the parent of the local root to reference the left child.
17.         else
18.             Find the rightmost node in the right subtree of the left
                child.
19.             Copy its data into the local root's data and remove it by
                setting its parent to reference its left child.
```

avancerad hemläxa: formulera och implementera en iterativ variant

Att kontrollera ett sökträd

Implementera en kontroll av datastrukturens invariant:

```
public boolean isSearchTree() {
        return checkNode(root);
}
private boolean checkNode(Node<E> node) {
        if (node == null) return true;
        if (!checkNode(node.left)) return false;
        for (E x : allDataValues(node.left))
                if (x ≥ node.data) return false;
        if (!checkNode(node.right)) return false;
        for (E x : allDataValues(node.right))
                if (x ≤ node.data) return false;
        return true;
}
```

Denna metod kan sedan stoppas in i en **assert** när du tillverkar dina egna metoder, som en kontroll. Se mer på Oracles Javasidor:

http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html

Att kontrollera ett sökträd

Implementera en kontroll av datastrukturens invariant:

```
public boolean isSearchTree() {
        return checkNode(root);
}
private boolean checkNode(Node<E> node) {
        if (node == null) return true;
        if (!checkNode(node.left)) return false;
        for (E x : allDataValues(node.left))
                if (x ≥ node.data) return false;
        if (!checkNode(node.right)) return false;
        for (E x : allDataValues(node.right))
                if (x ≤ node.data) return false;
        return true;
}
```

du måste implementera denhär metoden också, och den måste fungera även om argumentet är **null**

Denna metod kan sedan stoppas in i en **assert** när du tillverkar dina egna metoder, som en kontroll. Se mer på Oracles Javasidor:

http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html