Introduction to
Data Structures and Algorithms

Chapter: Binary Search Trees

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Search Trees

- Search trees can be used to **support dynamic sets**, i.e. data structures that change during lifetime, where an ordering relation among the keys is defined.
- They support many operations, such as
  - SEARCH,
  - MINIMUM, MAXIMUM,
  - PREDECESSOR, SUCCESSOR,
  - INSERT, DELETE.

- The time that an operation takes depends on the height h of the tree.

Binary Search Trees

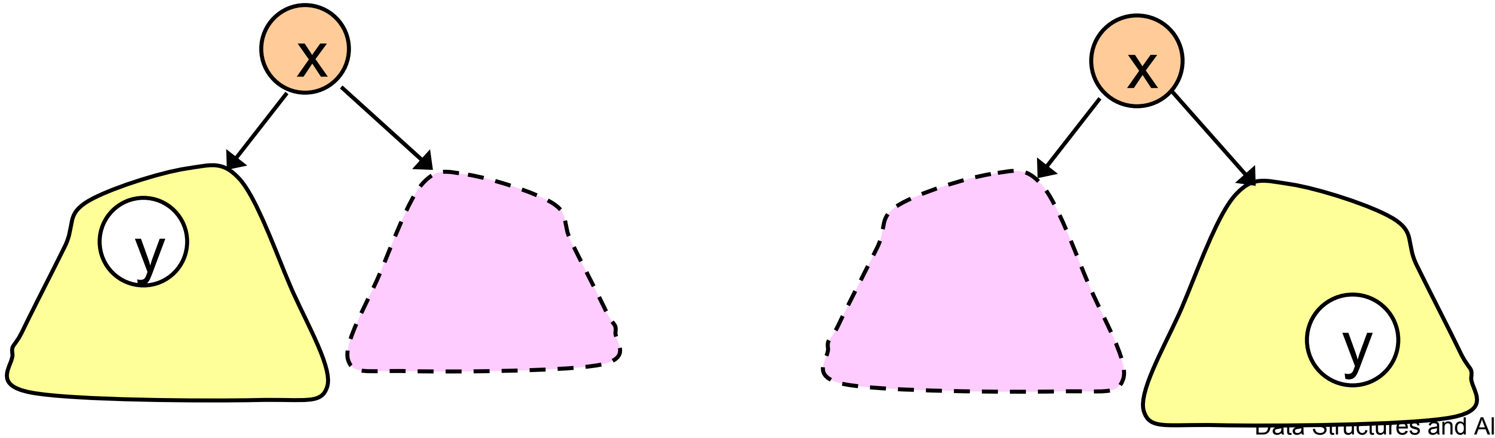
Binary Search Trees

- In the following we look at binary search trees
  – a special kind of binary trees –
  that support the mentioned operations on dynamic sets

Binary Search Trees

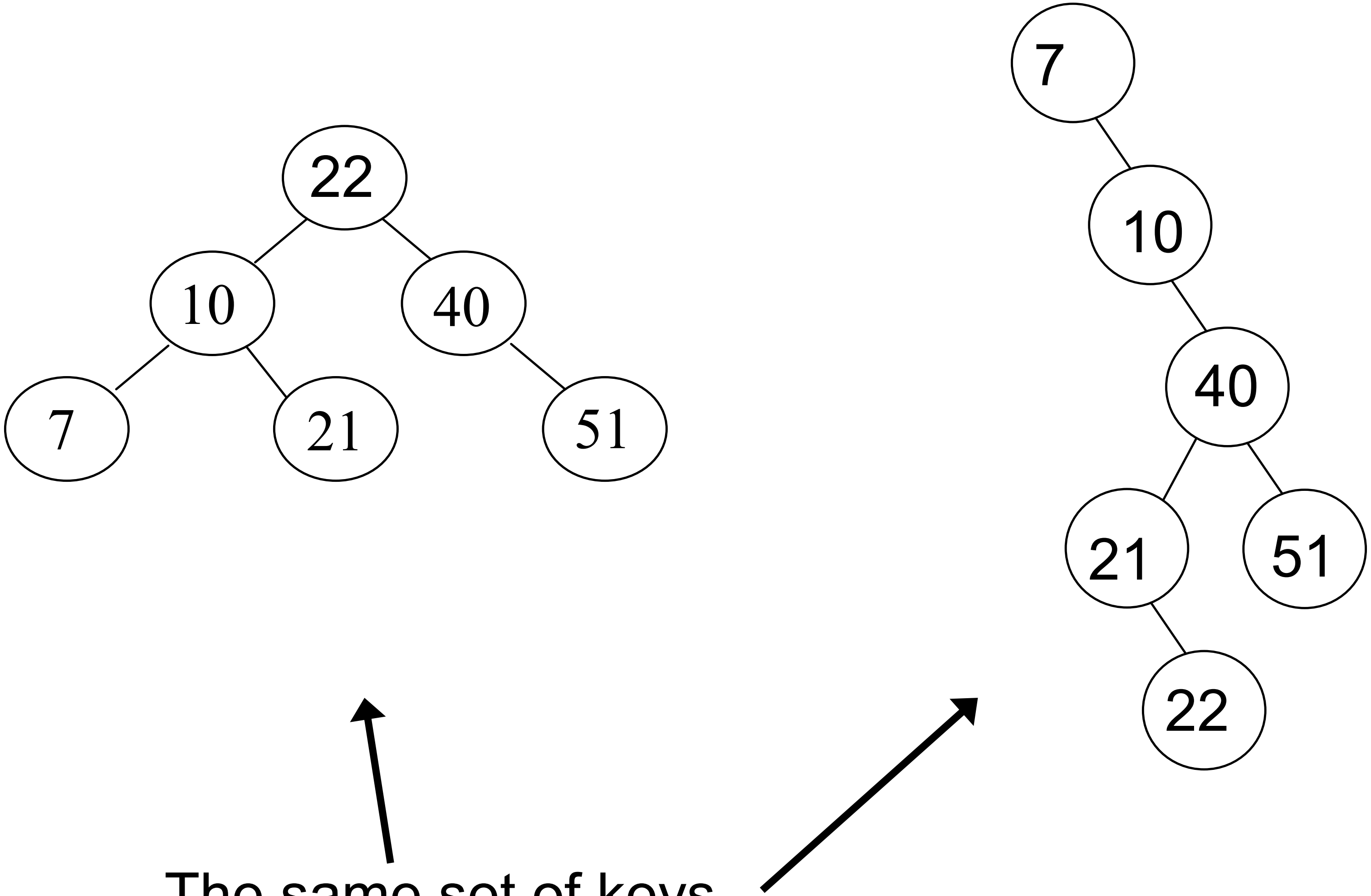
Binary Search Tree

- A binary tree is a **binary search tree**, if the following **binary-search-tree property** is satisfied:
- For any node x of the tree:
  - If y is a node in the left subtree of x then key[y] ≤ key[x]
  - If y is a node in the right subtree of x then key[y] ≥ key[x]



Binary Search Trees

Examples of Binary Search Trees

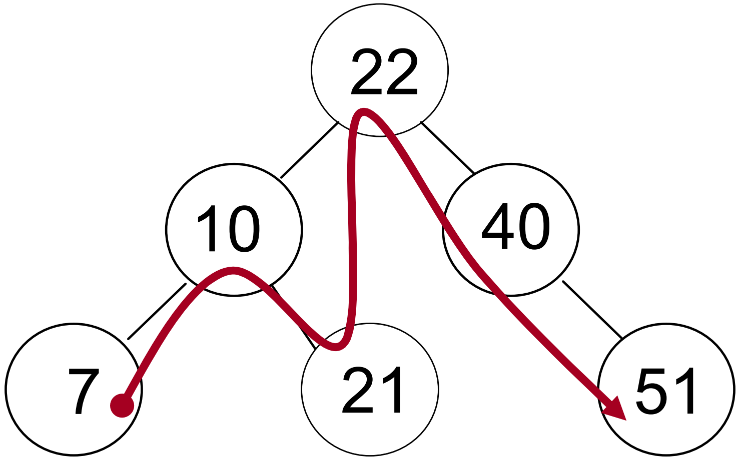


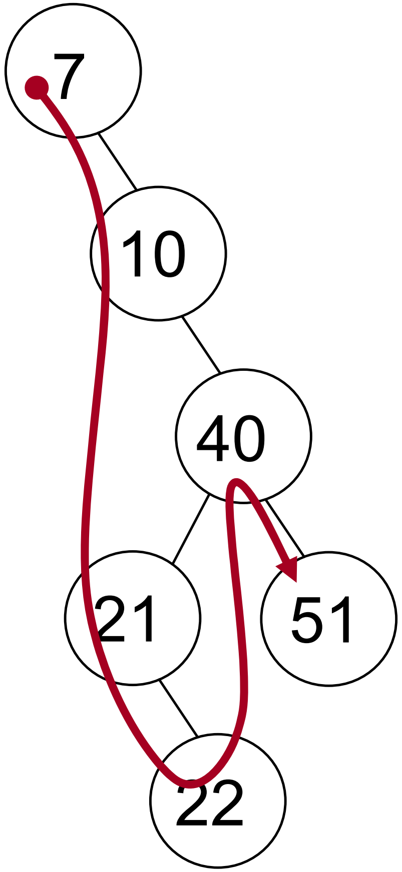
The same set of keys

Binary Search Trees

Example:

Printing the keys of a Binary Search Tree in ascending order





Binary Search Trees

Inorder tree walk

- **`Inorder_Tree_Walk`** prints the elements of a binary search tree in ascending order

```
Inorder_Tree_Walk(x)
   if x!=NIL then
      Inorder_Tree_Walk(left[x])
      print(key[x])
      Inorder_Tree_Walk(right[x])
```

- For a tree T the initial call is

```
Inorder_Tree_Walk(root[T])
```

Binary Search Trees

Inorder tree walk

- If x is the root of a binary (search) tree,
  the runtime of **`Inorder_Tree_Walk`** is Θ(n)

- Intuitive explanation:
  after the initial call of **`Inorder_Tree_Walk`** the following is true:

  for each of the (n -1) “not-NIL” nodes of the tree
  there are exactly two calls of **`Inorder_Tree_Walk`** **–**
  one for its left child and one for its right child
  (for details see [Corman])

Binary Search Trees

Searching for a node with given key k

- Recursive algorithm

```
Tree_Search(x,k)
   if x=NIL or k=key[x] then
      return x
   if k<key[x]
      then return Tree_Search(left[x],k)
      else return Tree_Search(right[x],k)
```

- For a tree T the initial call for searching for key k is

  `Inorder_Tree_Walk(root[T],k)`

- Runtime of **`Tree_Search(x,k)`** is O(h) where h is the height of the tree

Binary Search Trees

Searching for a node with given key k

- Non-recursive algorithm

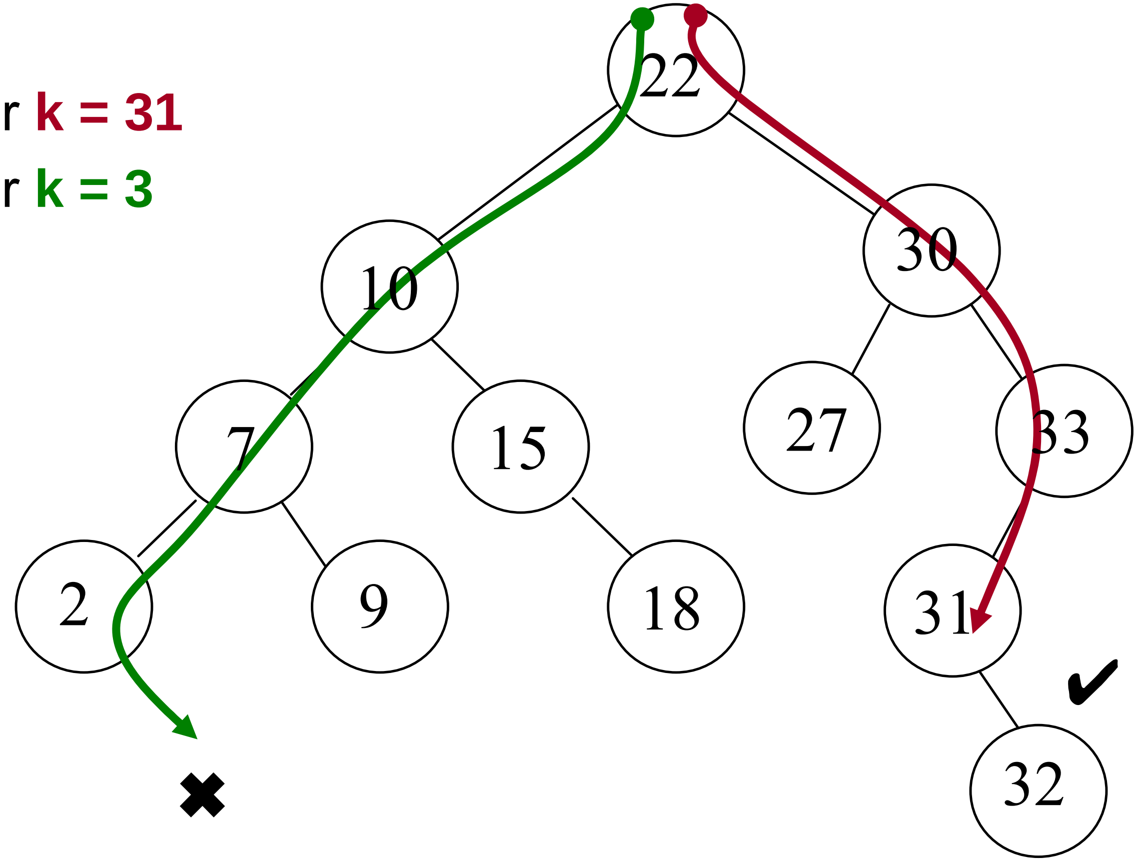
```
Iterative_Tree_Search(x,k)
   while x!= NIL and k!= key[x] do
      if k<key[x]
      then x:= left[x]
      else x:= right[x]
   return x
```

- This non-recursive (**iterative**) version of tree-search is usually more efficient in practice since it avoids the runtime system overhead for recursion

Binary Search Trees

Searching for a node with given key

- Example
  - Search for **k = 31**
  - Search for **k = 3**



Binary Search Trees

Minimum and maximum

- The element with minimum key value is always in the "leftmost" position of the binary search tree (not necessarily a leaf!)

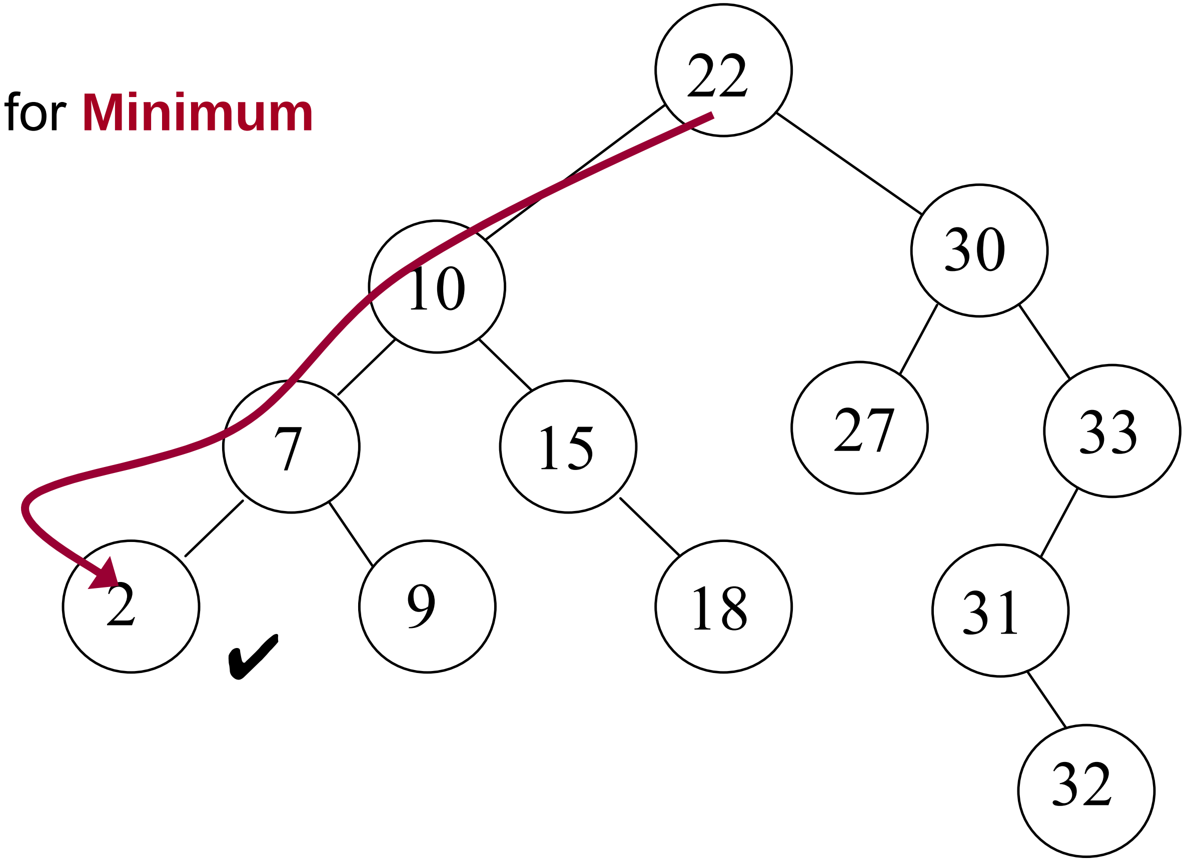
```
Tree_Minimum(x)
   while left[x]!=NIL do // left[x]=NIL ⇨ no smaller element
      x := left[x]
   return x
```

- Runtime of TreeMinimum and TreeMaximum ist O(h)
- Remark:
  The pseudo code TreeMaximum for the maximum is analogous

Binary Search Trees

Minimum and maximum

- Example
  - Search for **Minimum**



Binary Search Trees

Successor and predecessor

- If all keys are distinct, the **successor** (**predecessor**) of node x in sorted order is the node with the smallest key larger (smaller) than key[x]

```
Tree_Successor(x)
   if right[x]!=NIL then
      return Tree_Minimum(right[x])
   y := p[x]
   while y!=NIL and x=right[y] do
      x := y
      y := p[y]
   return[y]
```

Binary Search Trees

Successor and predecessor

- Remark: The pseudo code for Tree_Predescessor is analogous
- Runtime of Tree_Successor or Tree_Predecessor ist O(h)

- Because of the binary search property, for finding the successor or predecessor it is not necessary to compare the keys: We find the successor or predecessor because of the structure of the tree.
- So even if keys are not distinct, we define the **successor** (**predecessor**) of node x as the nodes returned by Tree_Successor(x) or Tree_Predeccessor(x).

Binary Search Trees

First summary

- On a binary search tree of heigth h,
  the dynamic-set operations
  SEARCH, MINIMUM, MAXIMUM,
  SUCCESSOR and PREDECESSOR
  can be implemented in time O(h)

Binary Search Trees

Insertion and deletion

- The operations of insertion and deletion **change** the dynamic set represented by a binary search tree
- If a new node is inserted into a binary search tree, or if a node is deleted, the structure of the tree has to be modified in such a way that the binary-search-tree property will still hold:
- A new node z to be inserted has fields
  - left[z] = right[z] = NIL
  - key[z] = v,   v - any value
- The new node is always inserted as a leaf

Binary Search Trees

Insertion (pseudo code)

Runtime of Tree_Insert: O(h)

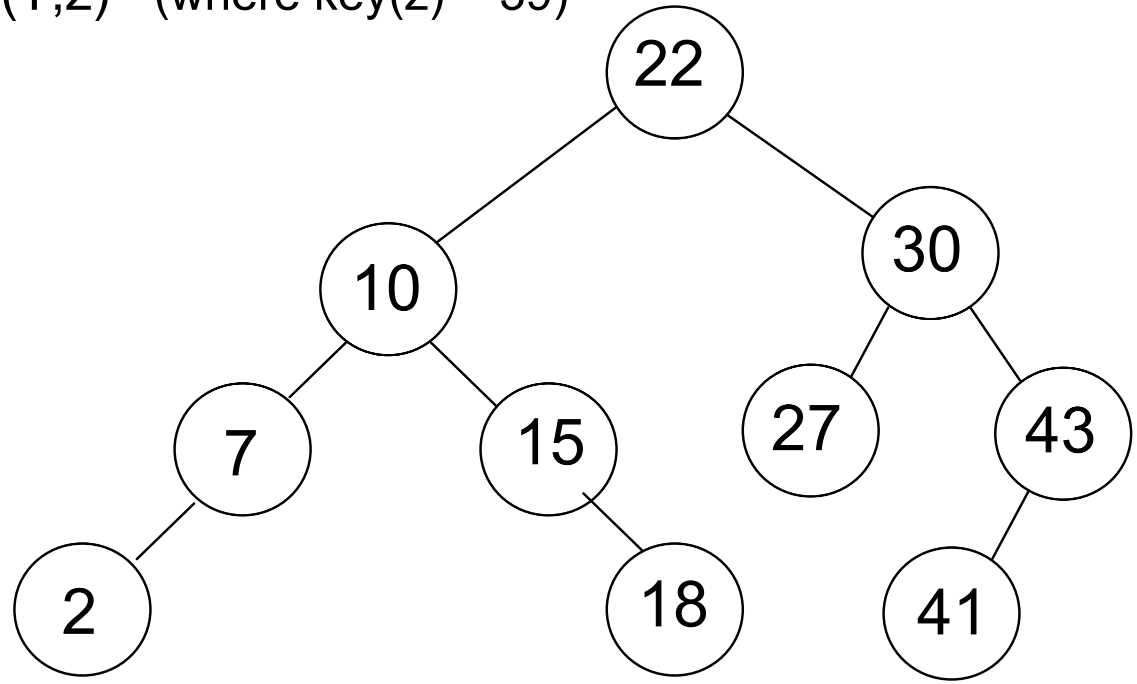
```
Tree_Insert(T,z)
   y := NIL
   x := root[T]
   while x!=NIL do
      y := x
      if key[z]<key[x]
         then x := left[x]
         else x := right[x]
   p[z] := y
   if y=NIL
      then root[T] := z        // Tree t was empty
      else if key[z]<key[y]
         then left[y] := z
         else right[y] := z
```

Binary Search Trees

Insertion (example)

a) Tree_Insert(T,z)   (where key(z) = 14)

b) Tree_Insert(T,z)   (where key(z) = 39)

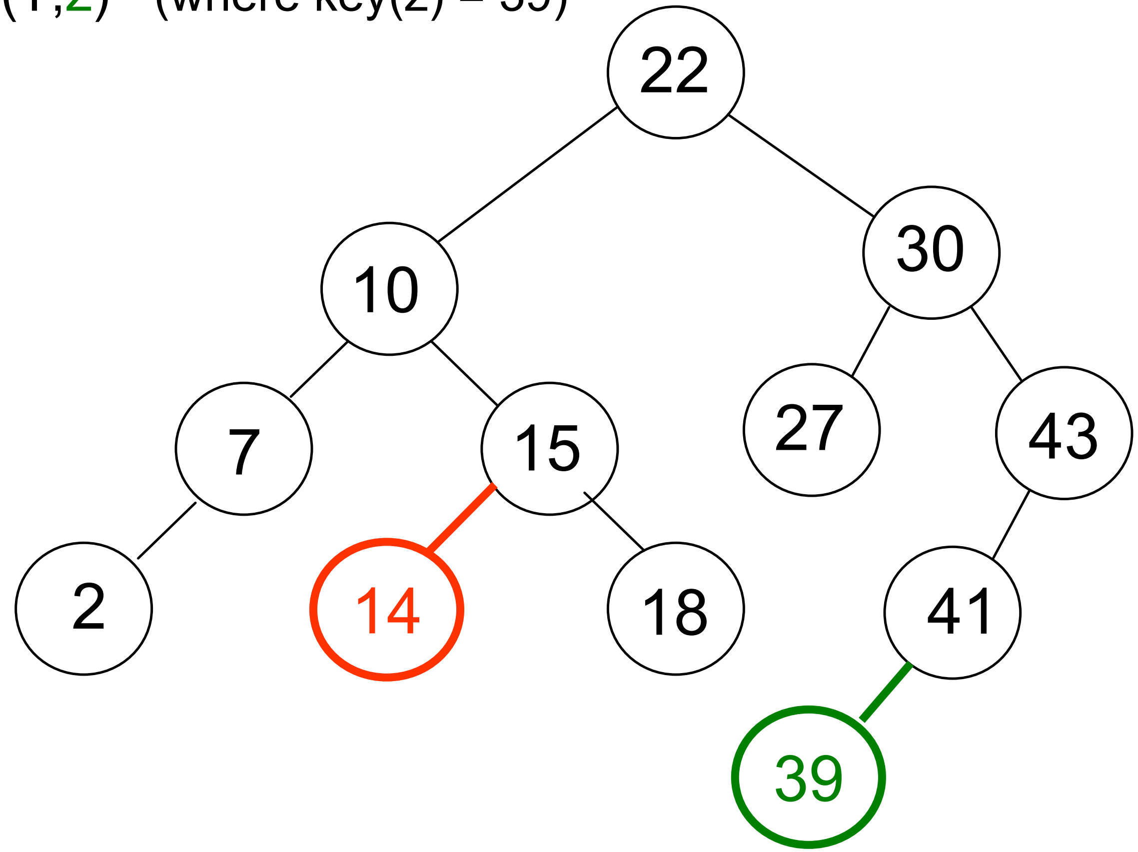


Binary Search Trees

Insertion (example)

a) Tree_Insert(T,z) (where key(z) = 14)

b) Tree_Insert(T,z) (where key(z) = 39)



Binary Search Trees

Deletion

- For deleting a node z from a binary search tree we can distinguish three cases
  - z is a leaf
  - z has only one child
  - z has two children
- If z is a leaf, the leaf is simply deleted
- If z has only one child, then z is “spliced out” by making a new link from p[z] to the child of z
- If z has two children, then we first find its successor y (which has no left child!), then splice out y, and finally replace the contents of z (key and satellite data) by the contents of y.

Binary Search Trees

Deletion (pseudo code)

```
Tree_Delete(T,z)

   if left[z]= NIL or right[z]= NIL
      then y := z
      else y := Tree_Successor(z)

   if left[y]!= NIL
      then x := left[y]
      else x := right[y]
   if x != NIL
      then p[x] := p[y]
   if p[y] = NIL

      then root[T] := x
```

(go on next page)

Binary Search Trees

Deletion (pseudo code)

```
   else if y = left[p[y]]
           then left[p[y]]:= x
           else right[p[y]]:= x
if y != z
   then key[z] := key[y]
        (copy y`s satellite data into z)
return y
```

- The runtime of Tree_Delete is O(h)

Binary Search Trees

Analysis (results)

- Height of randomly built binary search trees with n nodes
    - Best case: h = lg n
    - Worst case: h = n – 1

    - Average case behaviour:
        - Expected height of a randomly built search tree with n modes = O(lg n)

Binary Search Trees

Analysis (results)

- If the tree is a complete binary tree with n nodes,
  then the worst-case time is Θ(lg n).

- If the tree is very unbalanced (i.e. the tree is a linear chain),
  the worst-case time is Θ(n).

- Luckily, the expected height of a **randomly built** binary search tree is O(lg n)
  ⇨ basic operations take time Θ(lg n) on average.