Tutorial 9

Example 1

- pre: $s = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$

  res := s.retrieve(k)

  post: res $= (k_i, v_i)$ if $k = k_i$ for some $i \in \{1, \ldots, n\}$ and res $= nil$ otherwise; and $s' = s$

- In the book.

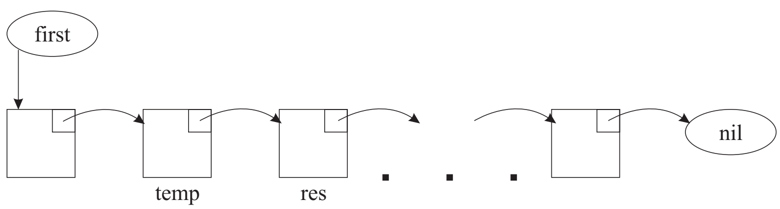

Figure 1: Overview of the representation in the memory.

- **retrieve(key:KEY_TYPE):ENTRY_TYPE** =
  $temp := first$
  $res := first$
  while NOT $res = nil$ and NOT $res.key = key$ **do**
  $temp := res$
  $res := res.next\_entry$
  end while
  if NOT $res = nil$ and NOT $temp = res$ **then**
  $temp.next\_entry := res.next\_entry$
  $res.next\_entry := first$
  $first := res$
  end if
  return res

The implementation is still correct with regard to the pre- and post- conditions above, as the set contains the same entries. It is only the order of the elements that has been changed.

Example 2

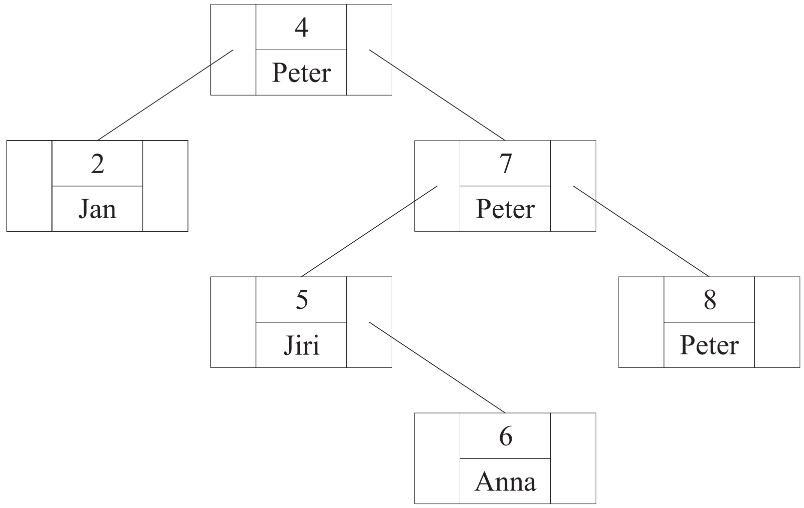

Figure 2: The resulting symbol table.

Example 3

- The procedure print can be implemented using inorder traversal.

  print(x:ENTRY_TYPE, a:KEY_TYPE, b:KEY_TYPE) =
  inorder_traversal(x);
  end

  Inorder traversal calls the procedure visit exactly once for each node, therefore we only need to define the visit procedure.

  visit(x) =
  if $x.key \geq a$ and $x.key \leq b$ **then**
  display(x.value)
  end if

- The worst-case time complexity of **print(t.root,a,b)** is $O(|t|)$ since each node is visited exactly once and every visit of a node takes $O(1)$. This complexity does not depend on the number of printed values.

- The following modification of inorder_traversal reduces the number of visited nodes, provided that the interval $[a, b]$ contains only some of the keys stored in the nodes.

inorder_traversal(x:ENTRY_TYPE) =
if NOT $t.nil\_entry(x)$ **then**
  if $a \leq x.key$ **then**
    inorder_traversal($t.left\_child(x)$)
  end if
  visit(x)
  if $x.key \leq b$ **then**
    inorder_traversal($t.right\_child(x)$)
  end if
end if

Example 4

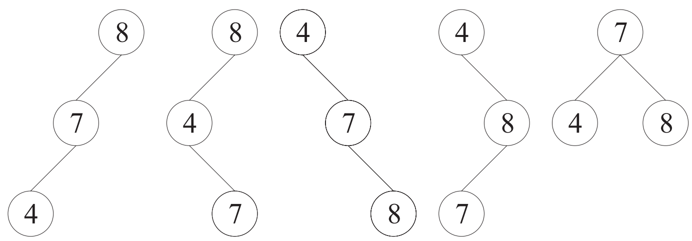

Figure 3: All the possible binary search trees.