```
BSTNode:
    info: TElem
    left: ↑ BSTNode
    right: ↑ BSTNode
```

```
BinarySearchTree:
    root: ↑ BSTNode
```

```
function search_rec (node, elem) is:
//pre: node is a BSTNode and elem is the TElem we are searching for
    if node = NIL then
        search_rec ← false
    else
        if [node].info = elem then
            search_rec ← true
        else if [node].info < elem then
            search_rec ← search_rec([node].right, elem)
        else
            search_rec ← search_rec([node].left, elem)
    end-if
end-function
```

```
function search (tree, e) is:
//pre: tree is a BinarySearchTree, e is the elem we are looking for
    search ← search_rec(tree.root, e)
end-function
```

```
function search (tree, elem) is:
//pre: tree is a BinarySearchTree and elem is the TElem we are searching for
    currentNode ← tree.root
    found ← false
    while currentNode ≠ NIL and not found execute
        if [currentNode].info = elem then
            found ← true
        else if [currentNode].info < elem then
            currentNode ← [currentNode].right
        else
            currentNode ← [currentNode].left
        end-if
    end-while
    search ← found
end-function
```

- Regarding the search algorithm, best case complexity is $\Theta(1)$, average case is $\Theta(log_2 n)$ and worst case is $\Theta(n)$.

```
function initNode(e) is:
//pre: e is a TComp
//post: initNode ← a node with e as information
   allocate(node)
   [node].info ← e
   [node].left ← NIL
   [node].right ← NIL
   initNode ← node
end-function
```

```
function insert_rec(node, e) is:
//pre: node is a BSTNode, e is TComp
//post: a node containing e was added in the tree starting from node
   if node = NIL then
      node ← initNode(e)
   else if [node].info ≥ e then
      [node].left ← insert_rec([node].left, e)
   else
      [node].right ← insert_rec([node].right, e)
   end-if
   insert_rec ← node
end-function
```

- Complexity: O(n) ($\Theta(n)$ in worst case, but $\Theta(log_2 n)$ on average)
- Like in case of the *search* operation, we need a wrapper function to call *insert_rec* with the root of the tree.

```
function minimum(tree) is:
//pre: tree is a BinarySearchTree
//post: minimum ← the minimum value from the tree
   currentNode ← tree.root
   if currentNode = NIL then
      @empty tree, no minimum
   else
      while [currentNode].left ≠ NIL execute
         currentNode ← [currentNode].left
      end-while
      minimum ← [currentNode].info
   end-if
end-function
```

- Complexity of the minimum operation: $O(n)$

```
function parent(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL
//post: returns the parent of node, or NIL if node is the root
   c ← tree.root
   if c = node then //node is the root
      parent ← NIL
   else
      while c ≠ NIL and [c].left ≠ node and [c].right ≠ node execute
         if [c].info ≥ [node].info then
            c ← [c].left
         else
            c ← [c].right
         end-if
      end-while
      parent ← c
   end-if
end-function
```

- Complexity: $O(n)$

```
function successor(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL
//post: returns the node with the next value after the value from node
//or NIL if node is the maximum
   if [node].right ≠ NIL then
      c ← [node].right
      while [c].left ≠ NIL execute
         c ← [c].left
      end-while
      successor ← c
   else
      c ← node
      p ← parent(tree, c)
      while p ≠ NIL and [p].left ≠ c execute
          c ← p
          p ← parent(tree, p)
      end-while
      successor ← p
   end-if
end-function
```

- If *parent* is $\Theta(1)$, complexity of successor is $O(n)$
- If *parent* is $O(n)$, complexity of successor is $O(n^2)$

- $O(n^2)$ is given by the potentially repeated calls for the *parent* function. But we only need the *last* parent, where we went left. We can do one single traversal from the root to our node, and every time we continue left (i.e. current node is greater than the one we are looking for) we memorize that node in a variable (and change the variable when we find a new such node). When the current node is at the one we are looking for, this variable contains its successor.

BST - Remove a node

- When we want to remove a value (a node containing the value) from a binary search tree we have three cases:
  - The node to be removed has no descendant
    - Set the corresponding child of the parent to NIL
  - The node to be removed has one descendant
    - Set the corresponding child of the parent to the descendant
  - The node to be removed has two descendants
    - Find the maximum of the left subtree, move it to the node to be deleted, and delete the maximum
      OR
    - Find the minimum of the right subtree, move it to the node to be deleted, and delete the minimum