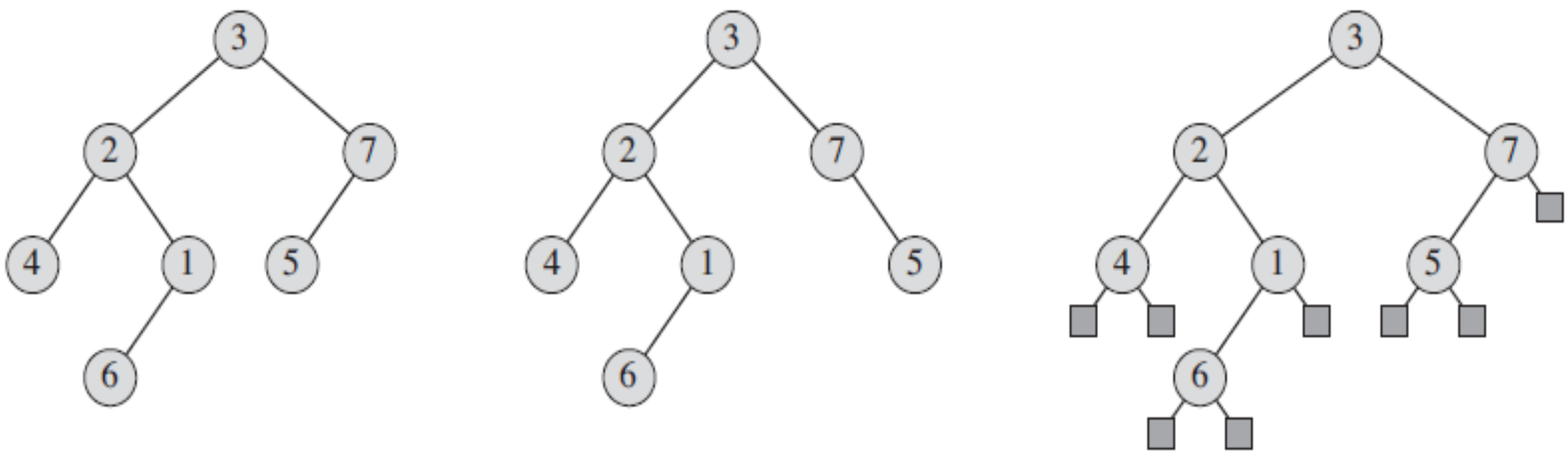
Binary Search Trees

Recap….

- **Binary tree:** A binary tree is defined recursively.
- **A binary tree T is a structure defined on finite set of nodes that either**
  - Contains no nodes (**the *empty tree*** or ***null tree)*** denoted NIL or
  - Composed of three disjoint set of nodes:
    - a **root node**
    - a binary tree called its **left subtree**
    - a binary tree called its **right subtree**

Reading Exercise

- Example of a Binary Tree:



(a) (b) (c)

Recap: CLRS Appendix B.5: Trees, Binary trees

Binary Search Tree (BST)

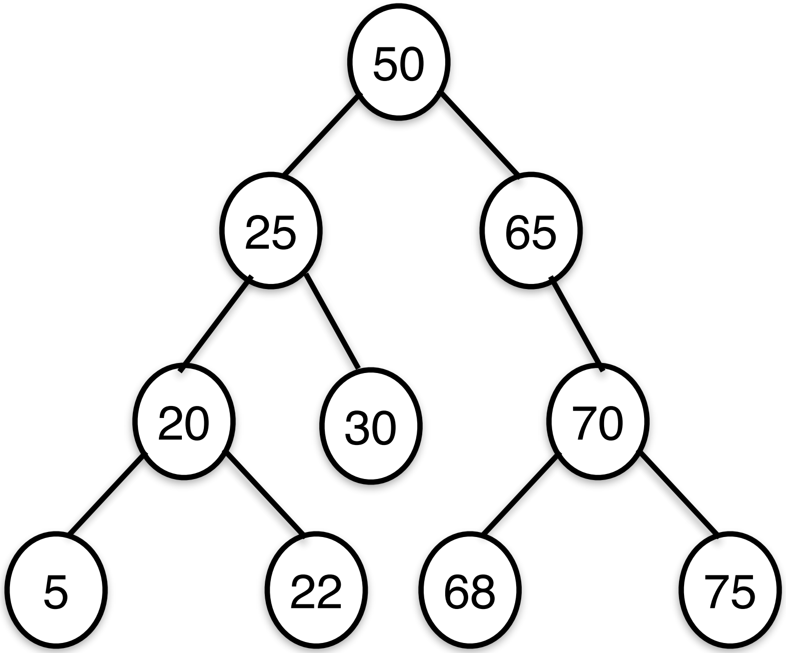
- BST is organized as a binary tree.
- *key* and satellite data, pointers *left*, *right*, and p
- value NIL, if a child or the parent is missing
- Root node - only node in the tree whose parent is NIL.
- Represent BST by a linked data structure (linked list) in which each node is an object.

BST property

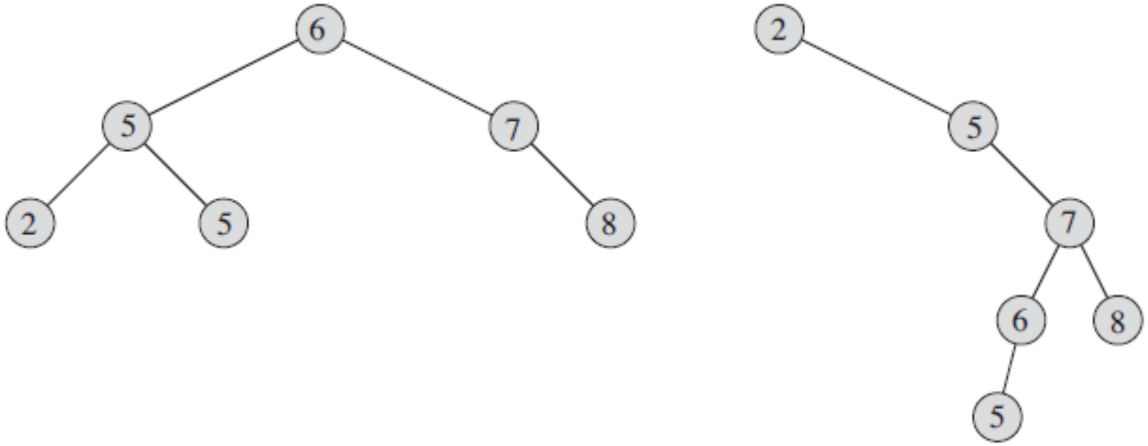
- Keys in a BST satisfy the ***binary-search-tree property***

Let x be a node in a binary search tree.

- If y is a node in the left subtree of x, then $y.key <= x.key$
- If y is a node in the right subtree of x, then $y.key >= x.key$



Binary Search Tree (BST) - Example



(a) (b)

- **Different binary search trees can represent the same set of values**
- Running time for various operations - **Proportional to height of the tree**
- Fig (a) - BST on 6 nodes with height 2, Fig (b) - BST on 6 nodes with height 4
- **Ex:** Draw two more different BSTs with the same set of keys.

How to print the elements in BST?

Traversals:

- In order (In order tree walk)
  - Prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree
  - **LeftSubTree—Root—RightSubTree**
- Pre order (Pre order tree walk)
  - Prints the root before the values in either subtree
  - **Root—LeftSubTree—RightSubTree**
- Post order (Post order tree walk)
  - Prints the root after the values in its subtrees.
  - **LeftSubTree—RightSubTree—Root**

Inorder-tree traversal

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

Call the function: INORDER-TREE-WALK(T.*root)*

Correctness of the algorithm follows by induction using the binary-search-tree property.

Running time – Inorder traversal

- $\Theta(n)$-time to walk an n-node BST
- After the initial call, the procedure calls itself recursively exactly twice for each node in the tree,
  - once for its left child and
  - once for its right child.

Running Time of INORDER-TREE-WALK

- Let $T(n)$ denote the time taken by INORDER-TREE-WALK, when it is called on the root of an n-node subtree.

- Since INORDER-TREE-WALK visits all n nodes of the subtree - $T(n)=\Omega(n)$ ——————————————(1)

- Now, show that $T(n)=O(n)$.

- INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq$ NIL), $T(0)=c$ for some constant $c > 0$.

- For $n > 0$, INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has n-k-1 nodes.

- Time to perform INORDER-TREE-WALK(x) is bounded by $T(n) <= T(k)+T(n-k-1)+d$, for some constant $d > 0$

- It reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

- Substitution method to show that $T(n)=O(n)$ by proving that $T(n) <= (c+d)n + c$.

- For $n = 0$, we have $(c+d).0 + c = c = T(0)$.

- For $n > 0$, we have $T(n) <= T(k)+T(n-k-1)+d$

Substitute $T(k) = (c+d)k + c$ and
$T(n\text{-}k\text{-}1) = (c + d)(n - k - 1) + c$

$$
\begin{aligned}
T(n) &\leq T(k) + T(n-k-1) + d \\
&= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\
&= (c+d)n + c - (c+d) + c + d \\
&= (c+d)n + c\,,
\end{aligned}
$$

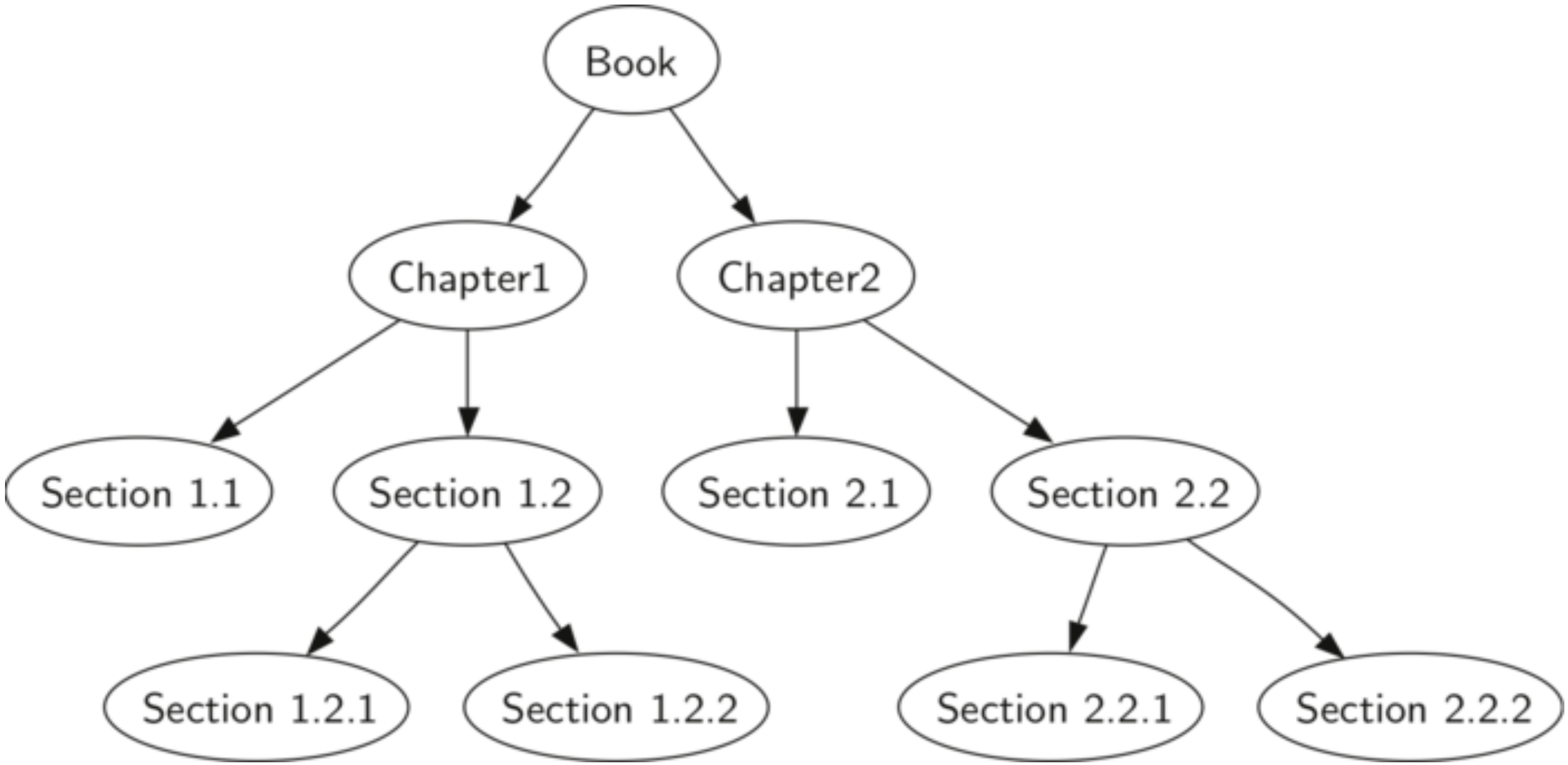
Thus, $T(n) = O(n)$ ---------------------------- (2)

From (1) and (2), it is clear that $T(n) = \Theta(n)$

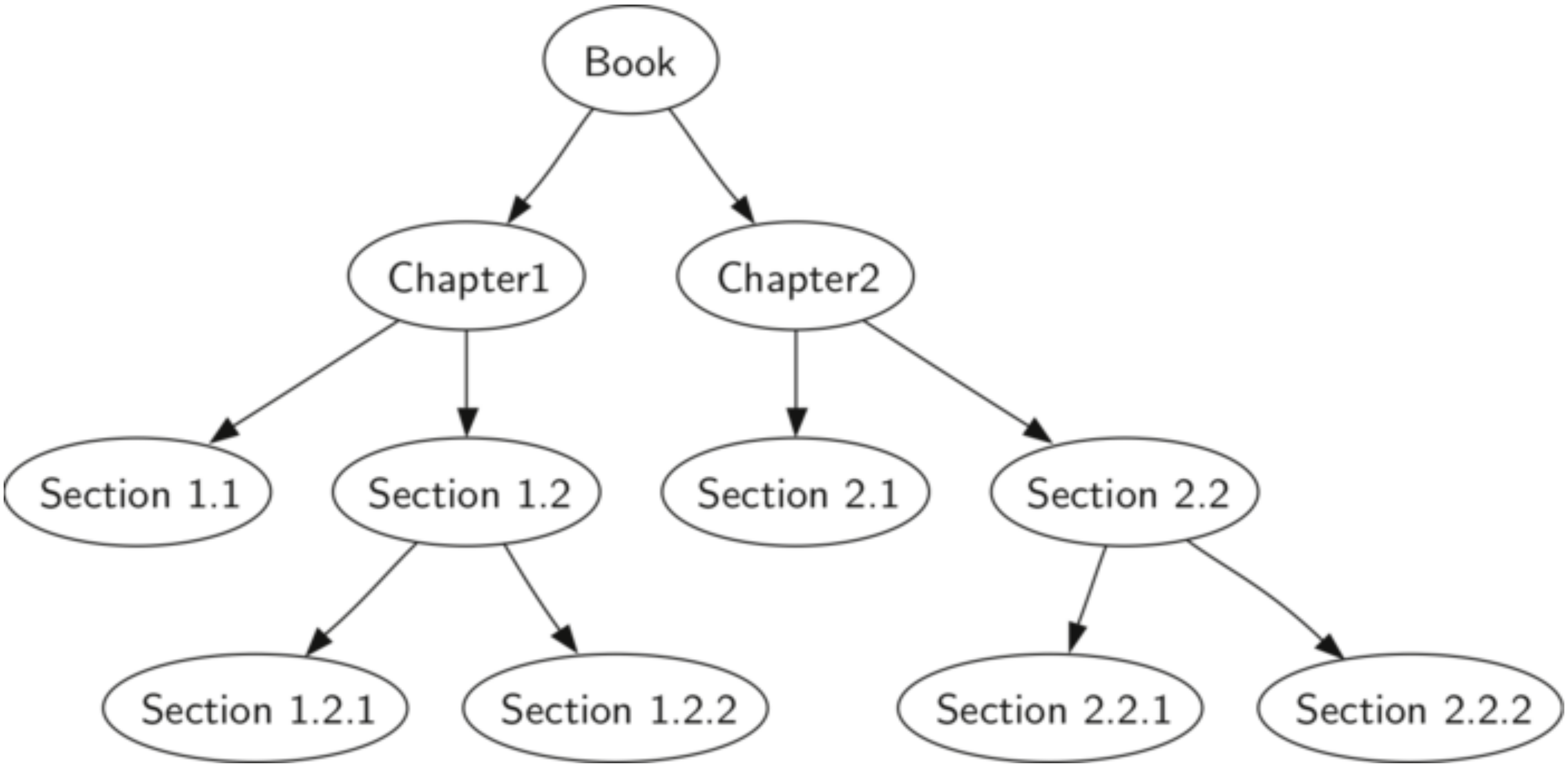
Application of traversals

Question: How do you print Table of contents in a book?

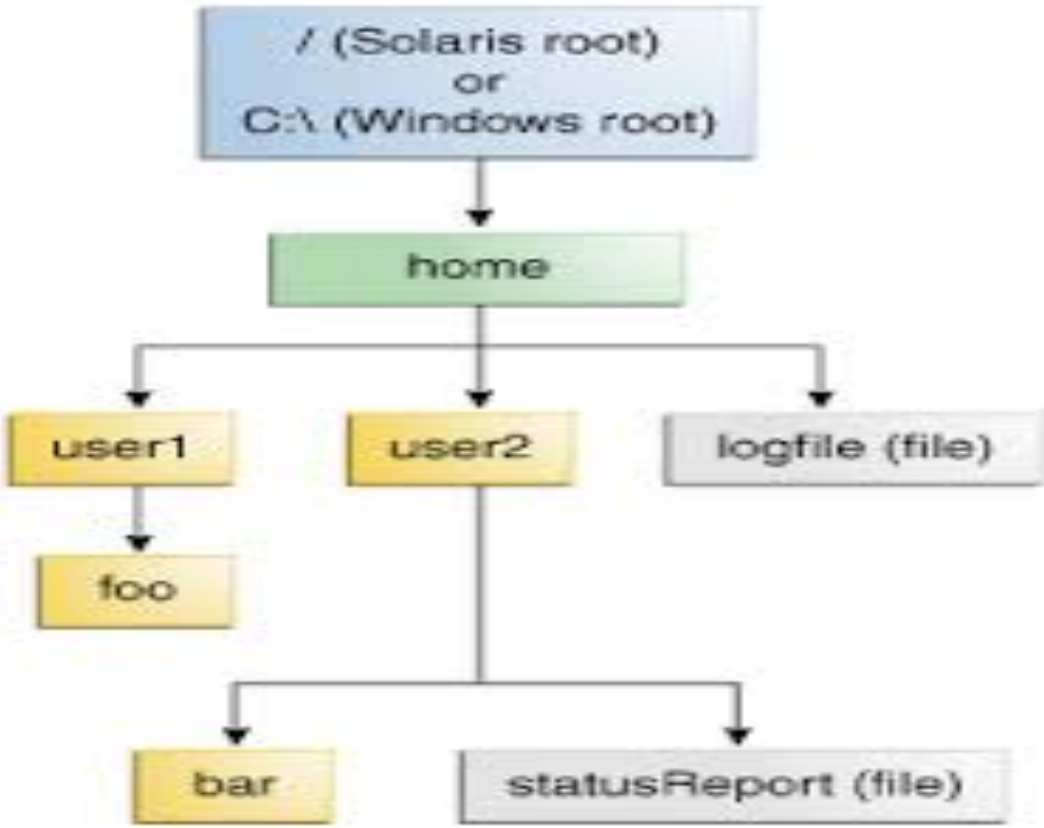
A book is represented as a binary tree as follows:



Print the Table of Contents



Directory structure of Operating Systems



Directory structure – Unix/Linux

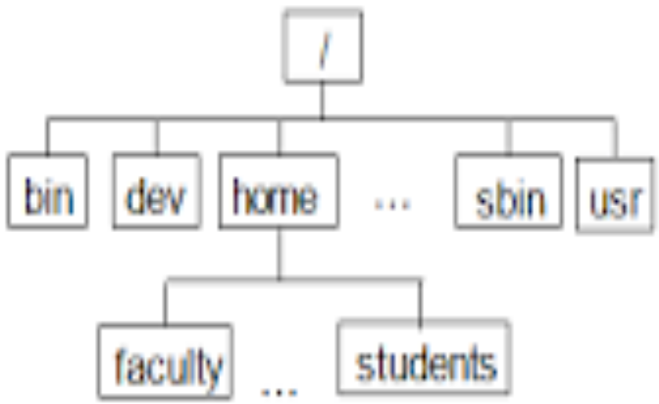


Figure of UNIX/Linux directory hierarchy

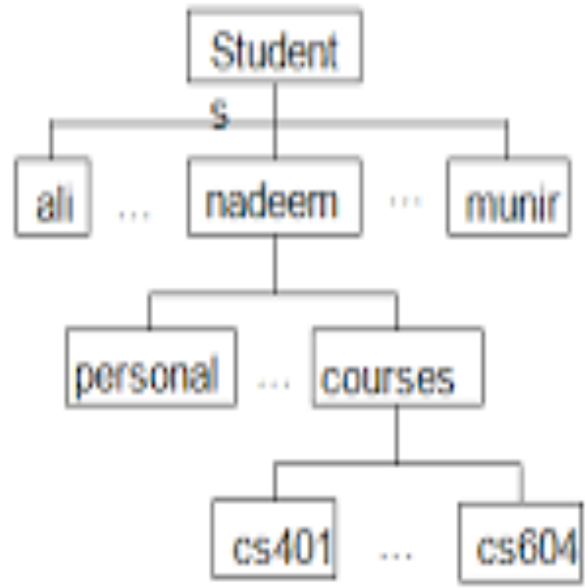


Figure of Home directories of students

How do you find the size of “Student” directory, given the size of files in each of the subfolders ?

Exercises

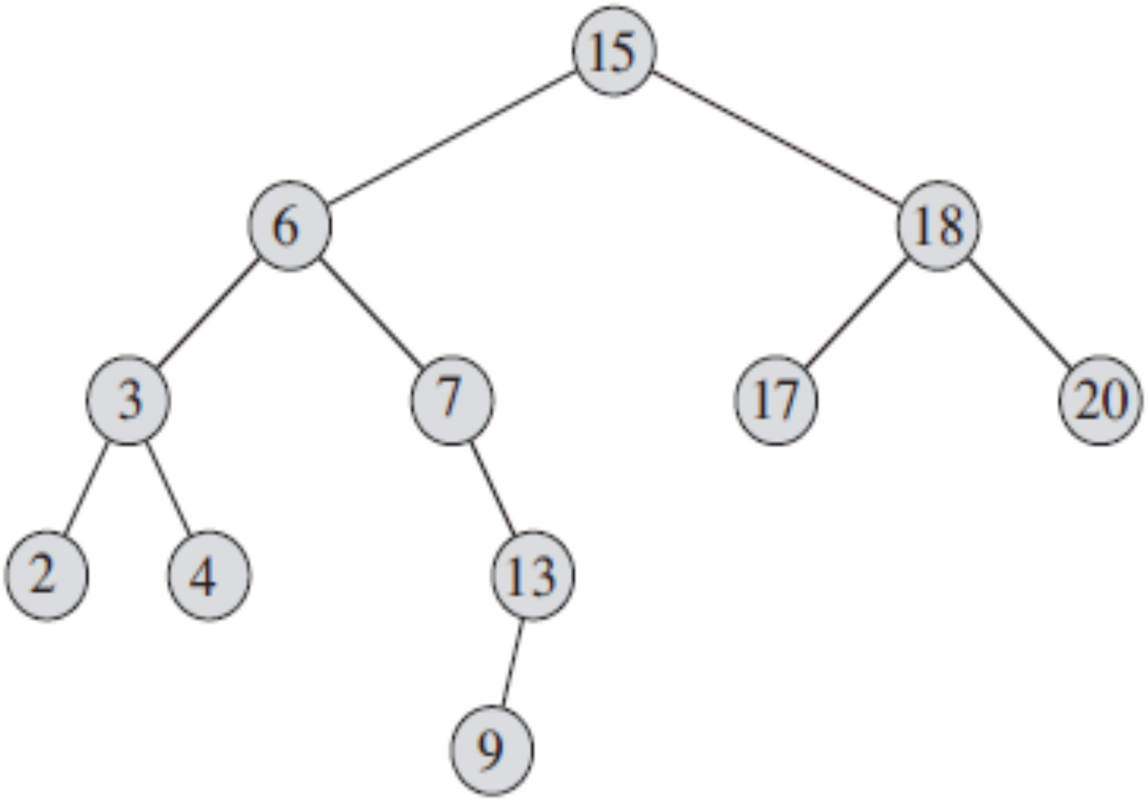
- Write the pseudocode for Postorder and Preorder traversal algorithms (recursive)
- Prove the correctness of these traversal algorithms
- Derive the running time of Postorder and Preorder traversal algorithms

Querying a binary search tree

- **Query operations** - Search, Minimum, Maximum, Successor and Predecessor
- BST support these operations each one in time **O(h)** on any binary search tree of **height h.**

Exercise

Search for the key 13



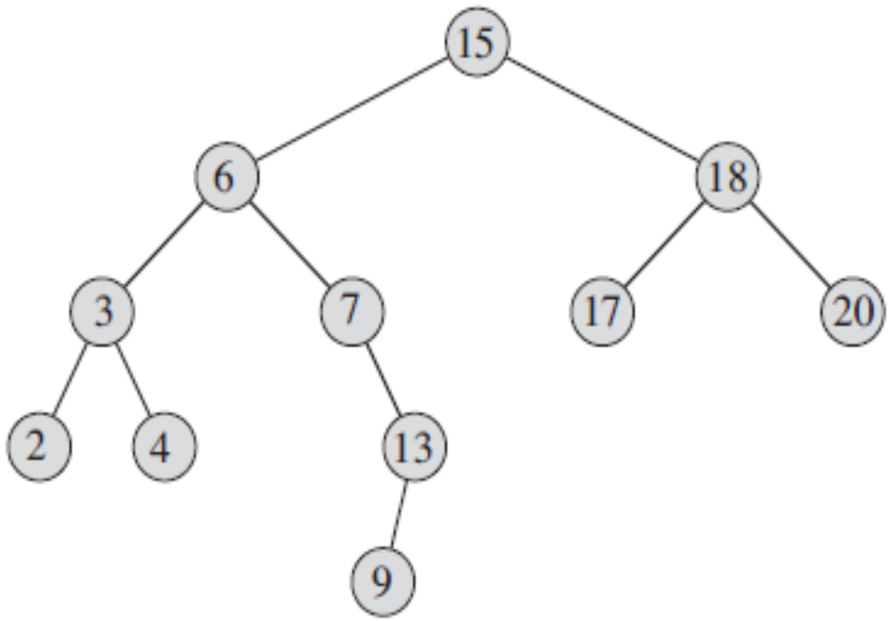
TREE-SEARCH(x, k)

- Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

Exercise

Search for the key 15 and 14



TREE-SEARCH(x, k)

- The procedure begins its search at the root and traces a simple path downward in the tree
- For each node *x* it encounters, it compares the key k with x.*key*.
- If the two keys are equal, the search terminates.
- If k is smaller than x.*key*, the search continues in the left subtree of x, since the binary-search tree property implies that k could not be stored in the right subtree.

TREE-SEARCH(x, k)

- Symmetrically, if k is larger than x.*key*, the search continues in the right subtree.
- The nodes encountered during the recursion form a simple path downward from the root of the tree.
- Running time of TREE-SEARCH is O(h), where h is the height of the tree.

Iterative version - Search

- Write the iterative version of Tree-Search(x,k)

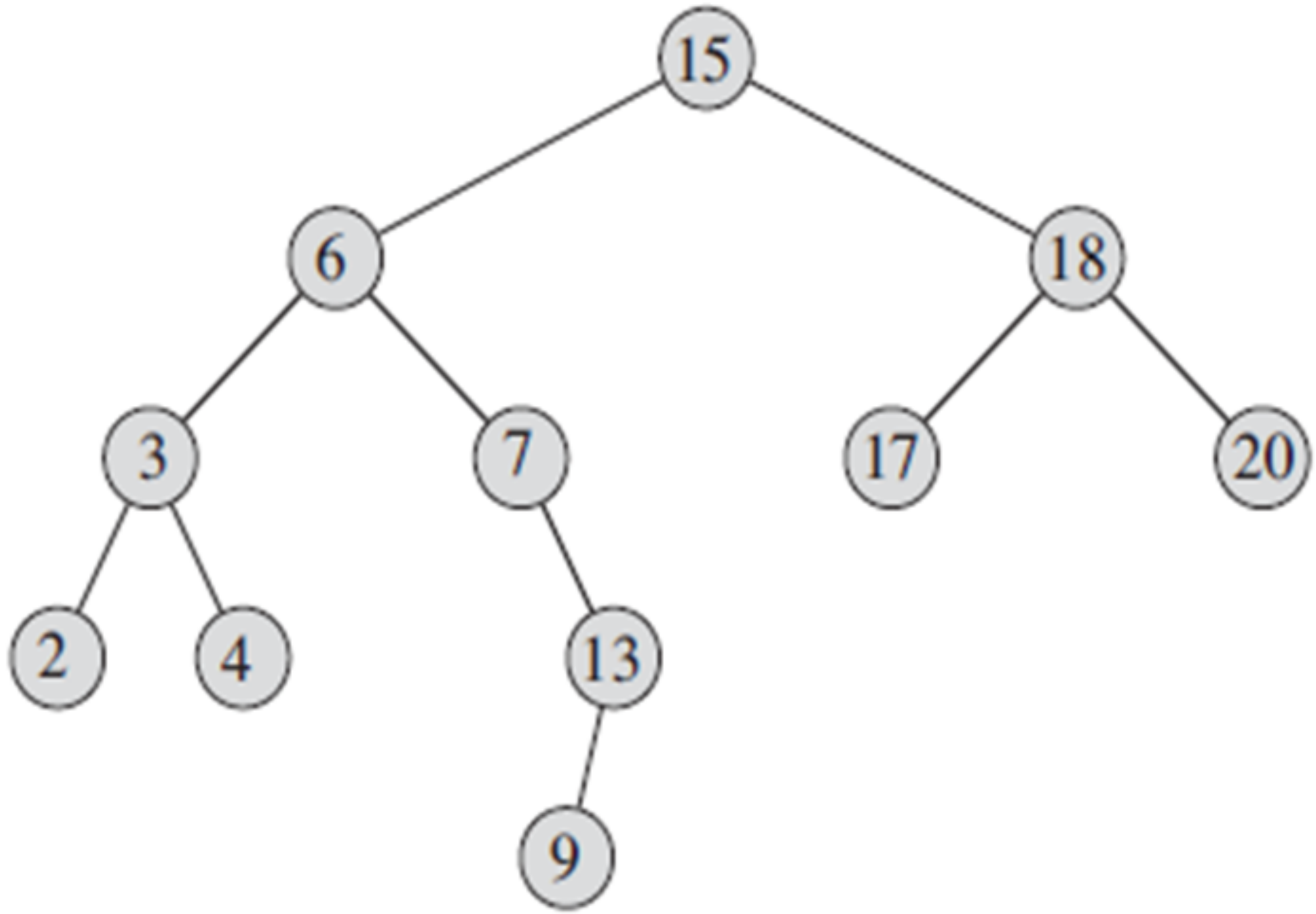
- Generally, iterative version is more efficient than recursive version.

- Why….?

Tree-Search(x,k)

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

HOW DO WE FIND THE MINIMUM AND MAXIMUM ELEMENT IN BST?


15
6
18
3
7
17
20
2
4
13
9

Minimum element

- The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be non-NIL:

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

TREE-MINIMUM(x)

- Correctness:

The binary-search-tree property guarantees that TREE-MINIMUM is correct.

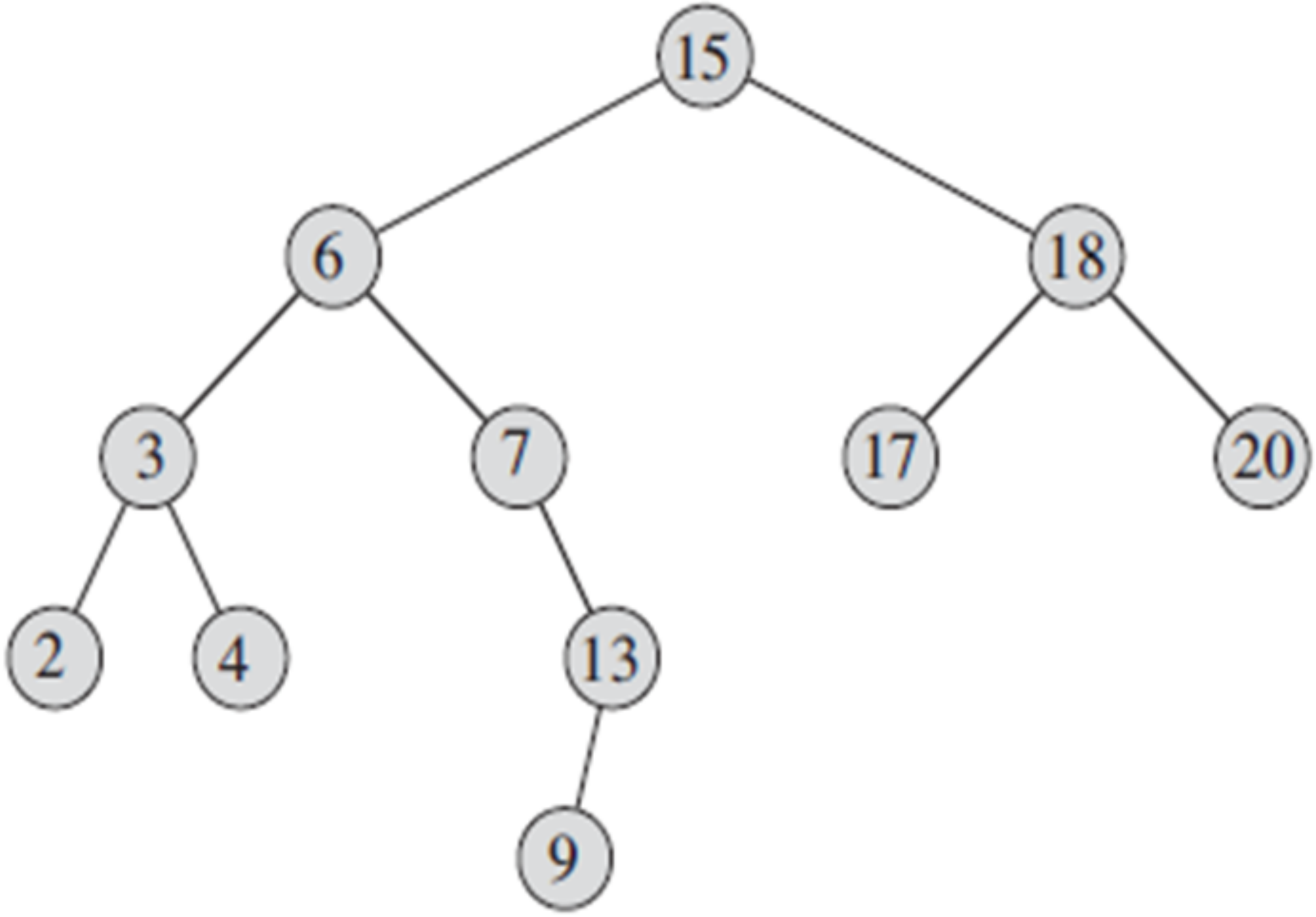
- Node x has no left subtree

since every key in the right subtree of x is at least as large as x.*key*, the minimum key in the subtree rooted at x is x.*key*.

- Node x has a left subtree

No key in the right subtree is smaller than x.*key* and every key in the left subtree is not larger than x.*key*, the minimum key in the subtree rooted at x resides in the subtree rooted at x.*left*.

TREE -MAXIMUM



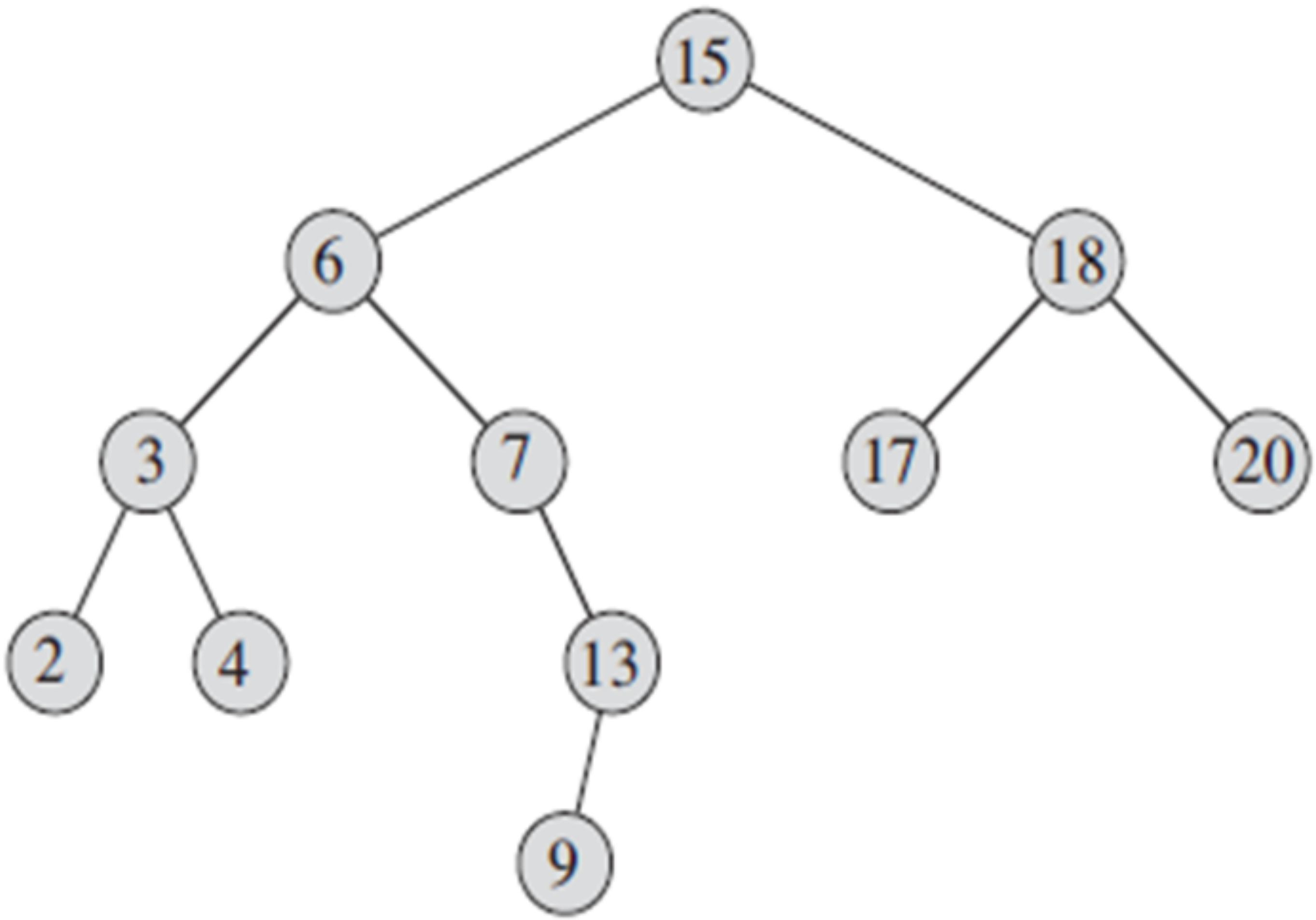
TREE-MAXIMUM(x)

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

- Running time of Tree-Minimum and Tree-Maximum
- In this case also, the sequence of nodes encountered forms a simple path downward from the root.

How do we find the Predecessor and Successor of a node in BST?

SUCCESSOR & PREDECESSOR OF A NODE



Successor and predecessor

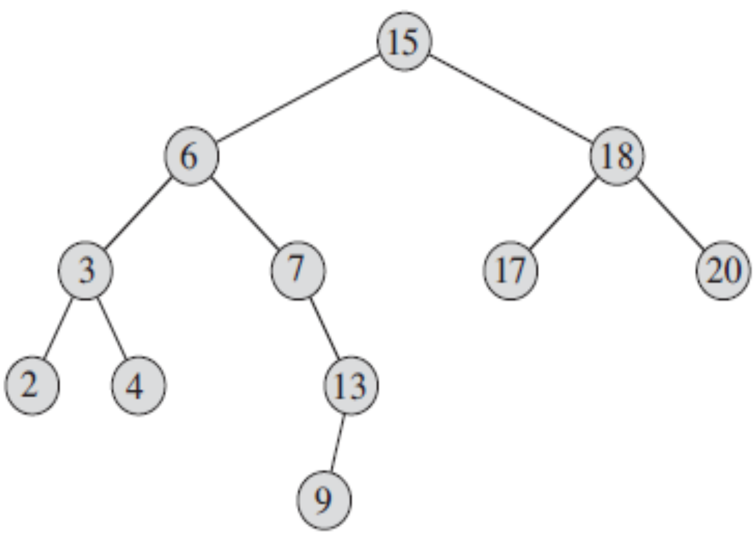
- Given a node in a BST, sometimes we need to find its successor in the sorted order determined by an in order tree walk.
- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.*key*.
- The structure of a BST allows us to determine the successor of a node without ever comparing keys.
- The following procedure returns the successor of a node x in a BST if it exists, and NIL if x is the largest key in the tree

TREE-SUCCESSOR(x)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL                          // Case 1
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y                                 // Case 2
    y = y.p
return y
```

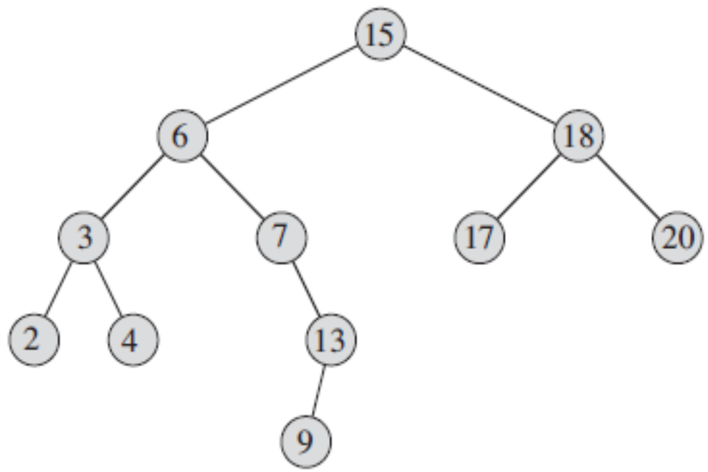
TREE-SUCCESSOR(x): 2 cases

- Case 1: If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree, which we find in line 2 by calling TREE-MINIMUM(x.*right*)
- Eg: Successor of the node with key 15 ?



- Case 2: If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Eg: Successor of the node with key 13?

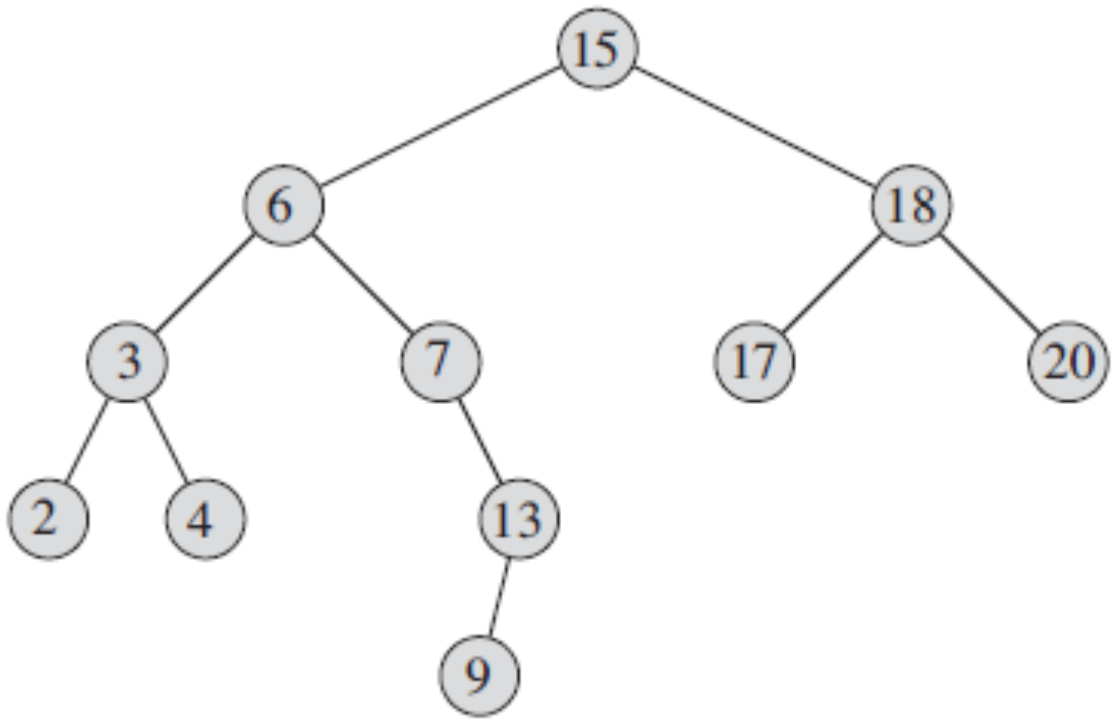


- To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3–7 of TREE-SUCCESSOR handle this case.

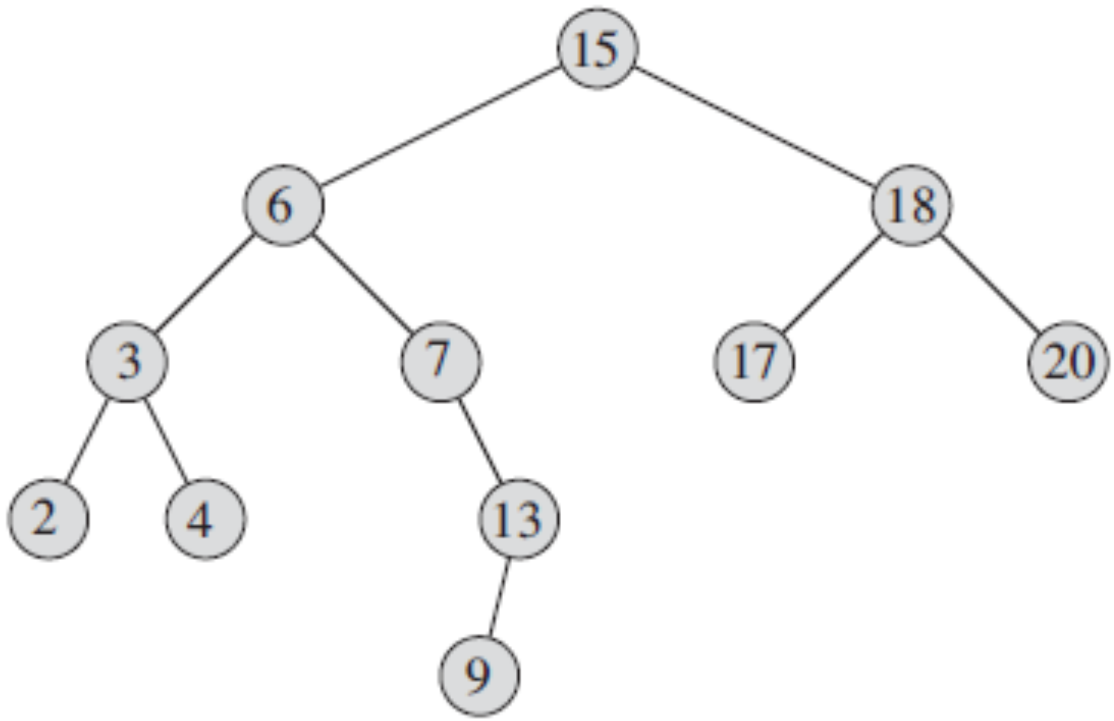
TREE-SUCCESSOR(x)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)        // Case 1
y = x.p
while y ≠ NIL and x == y.right
    x = y                               // Case 2
    y = y.p
return y
```

Trace the pseudo code to find the successor of 4



Trace the pseudo code to find the successor of 9



Running time

- The running time of TREE-SUCCESSOR on a tree of height h is O(h), since we either follow a simple path up the tree or follow a simple path down the tree.
- The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time O(h)

Exercise

- Write the pseudo-code for TREE-PREDECESSOR(x)
- Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x) respectively

Running-time

We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in O(h) time on a binary search tree of height h.

References

- CLRS Book