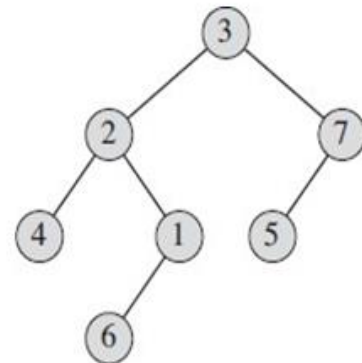


Binary Search Trees

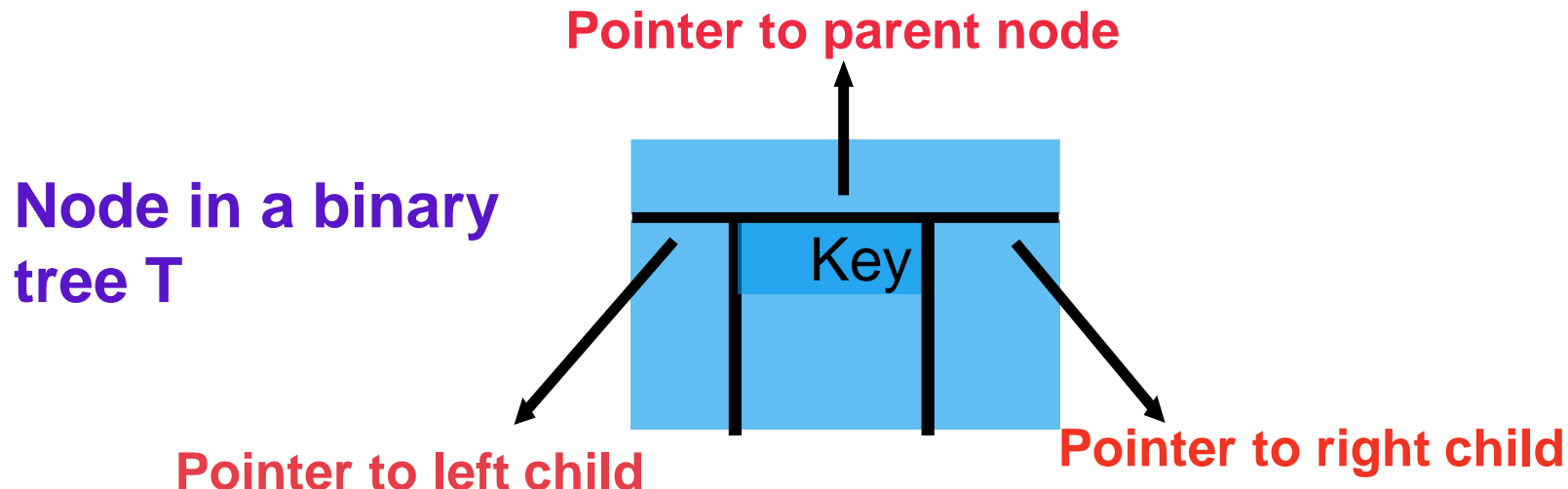
Recap

- **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**



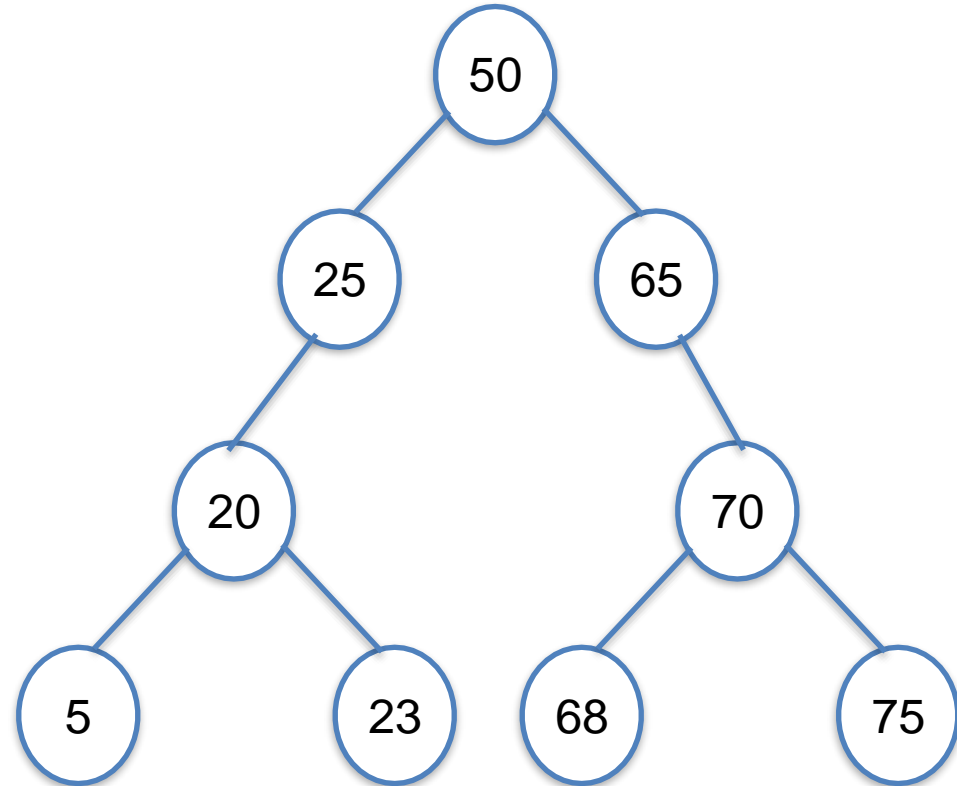
Binary Search Tree (BST)

- BST is organized as a **binary tree**.
- **key** and satellite data, pointers **left**, **right**, and **parent**
- **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

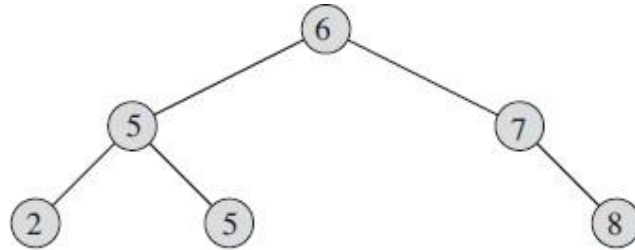


BST property

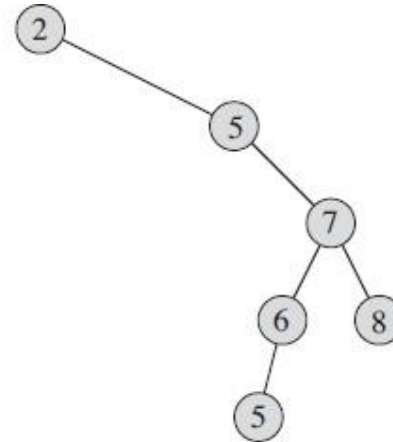
- Keys in a BST satisfy the *binary-search-tree property*
- Let x be a node in a binary search tree.
- If y is a node in the left subtree of x , then $y.key \leq x.key$
- If y is a node in the right subtree of x , then $y.key \geq x.key$



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?

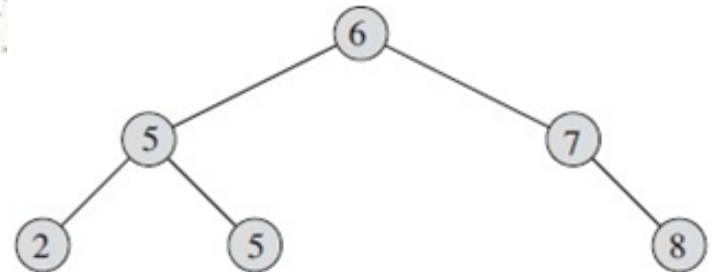
Traversals:

- 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)

```
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )
```

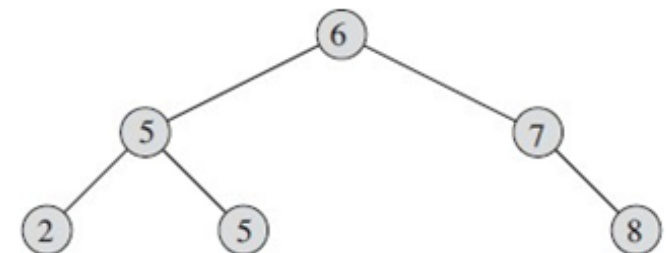


Call the function: INORDER-TREE-WALK($T.\text{root}$)

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.



INORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )
```


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)

INORDER-TREE-WALK(x)

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

```
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )
```
- INORDER-TREE-WALK takes a small, **constant amount of time on an empty subtree** (for the test $x \neq \text{NIL}$), $T(0)=c$ 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 $T(n) \leq T(k) + T(n-k-1) + d$, 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 $T(n) \leq (c+d)n + c$.
- For $n = 0$, we have $(c+d).0 + c = c = T(0)$.
- For $n > 0$, we have $T(n) \leq T(k) + T(n-k-1) + d$

Substitute $T(k) = (c+d)k + c$ and

$$T(n-k-1) = (c + d)(n - k - 1) + c$$

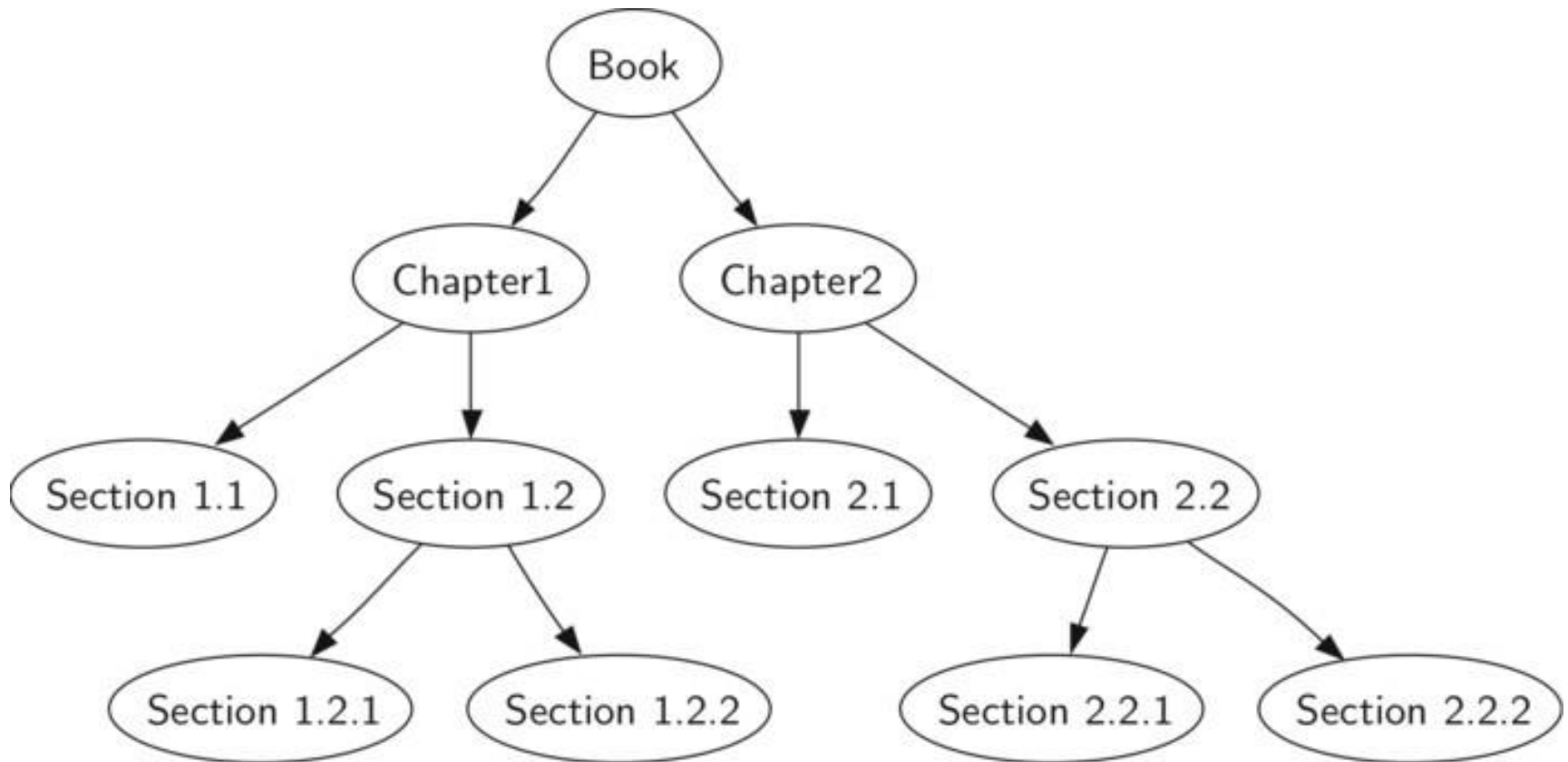
$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c, \end{aligned}$$

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

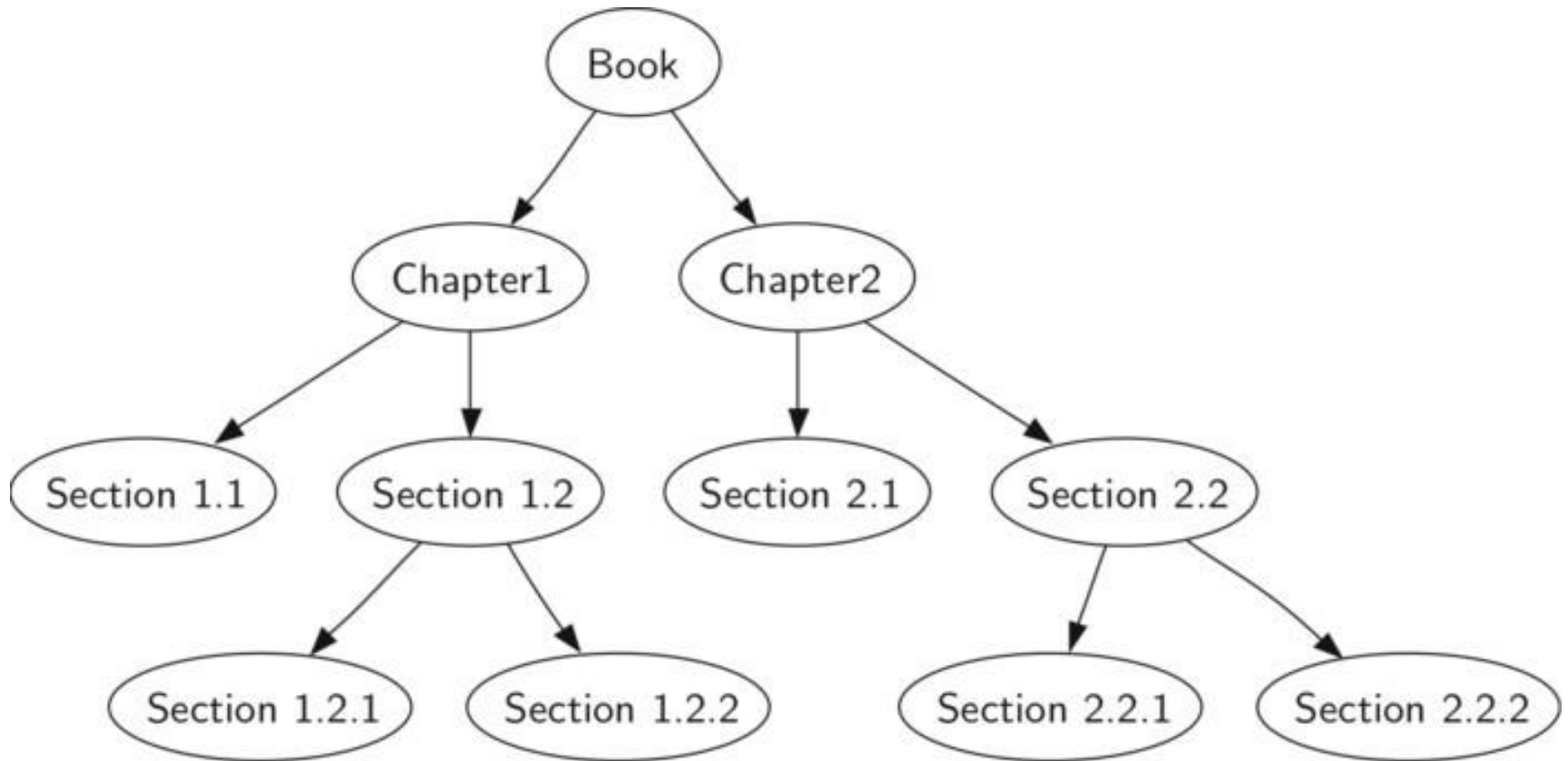
From (1) and (2), it is clear that $T(n) = \Theta(n)$

Application of traversals

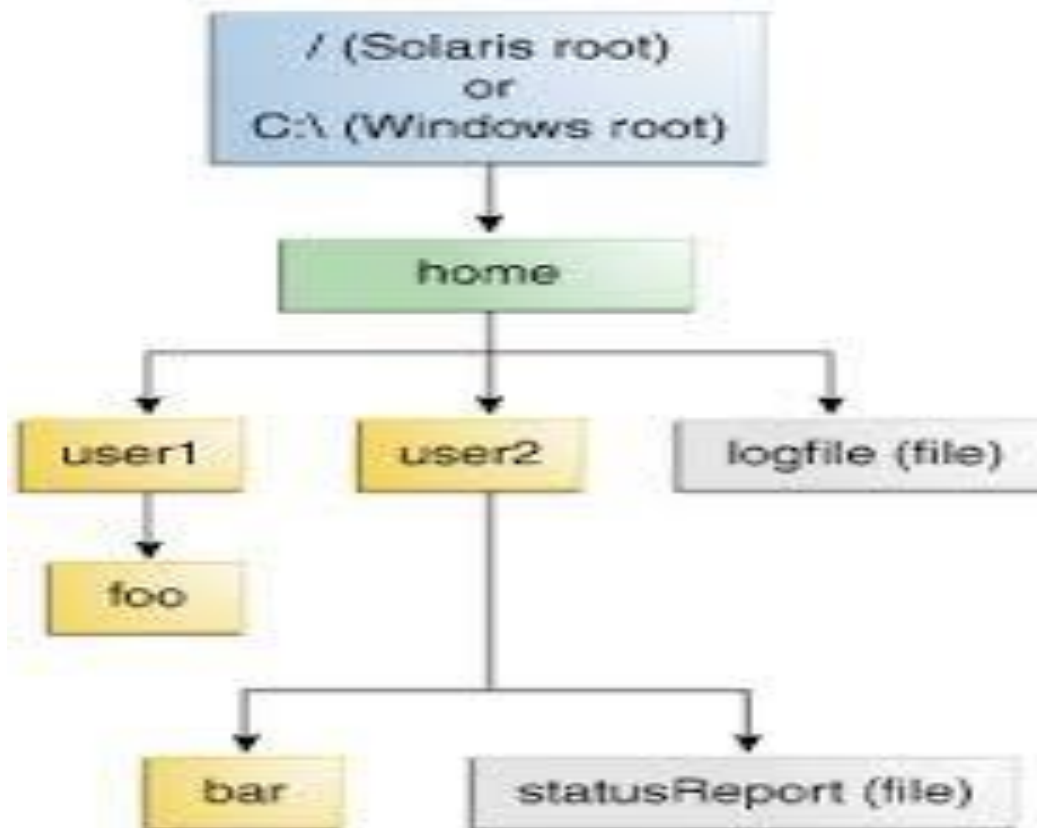
A book is represented as a **binary tree** as follows:



Print the Table of Contents



Directory structure of Operating Systems



Directory structure – Unix/Linux

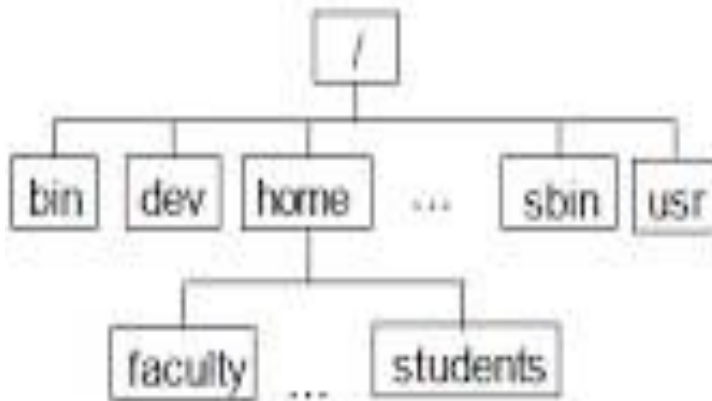


Figure of UNIX/Linux directory hierarchy

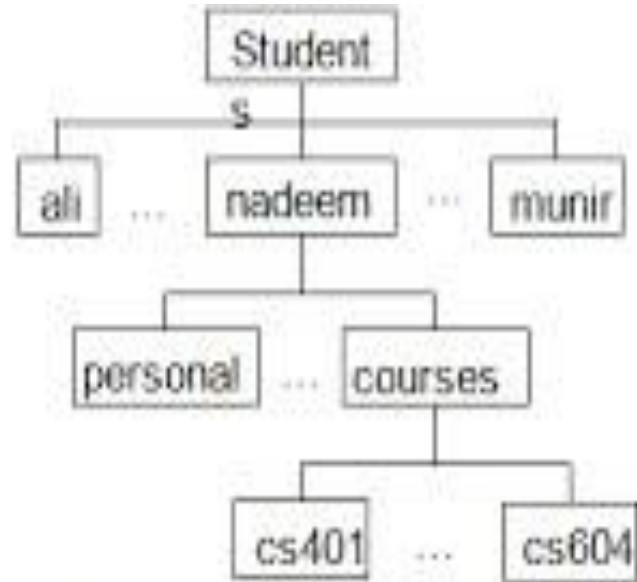


Figure of Home directories of students

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

Exercises

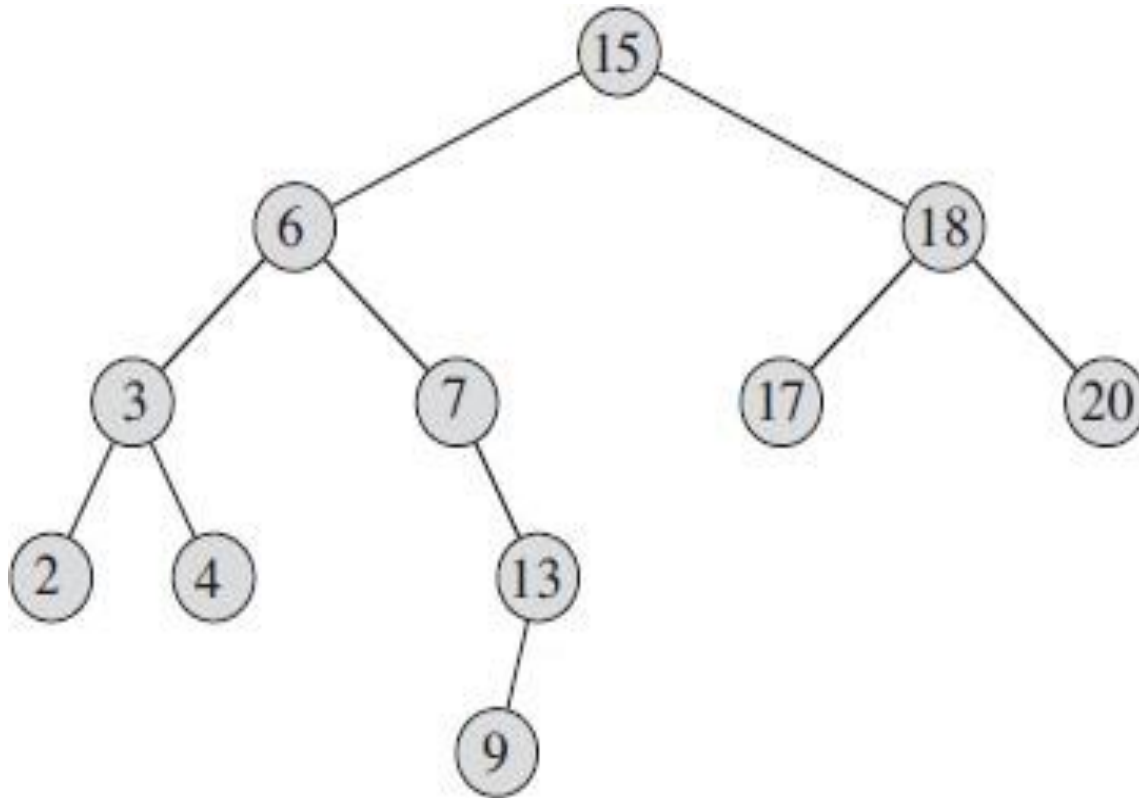
- Write the pseudocode for Postorder and Preorder traversal algorithms (recursive)
- Prove the correctness of these traversal algorithms
- Derive the running time of Postorder and Preorder traversal algorithms

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)

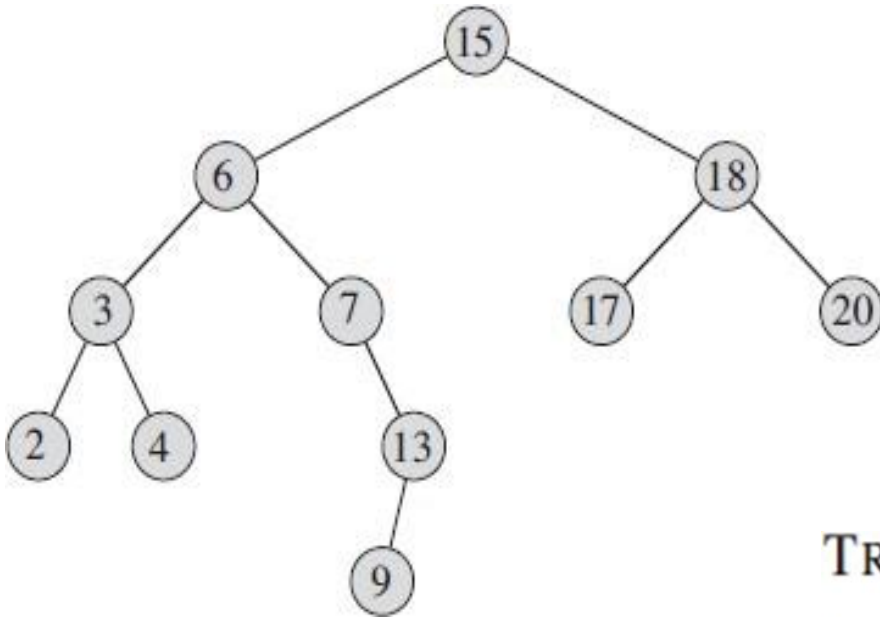
- Given a pointer to the root of the tree and a key k , **TREE-SEARCH** returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

TREE-SEARCH(x, k)

```
1  if  $x == \text{NIL}$  or  $k == x.\text{key}$ 
2      return  $x$ 
3  if  $k < x.\text{key}$ 
4      return TREE-SEARCH( $x.\text{left}, k$ )
5  else return TREE-SEARCH( $x.\text{right}, k$ )
```

Exercise

Search for the key 15 and 14



TREE-SEARCH(x, k)

- 1 if $x == \text{NIL}$ or $k == x.\text{key}$
- 2 return x
- 3 if $k < x.\text{key}$
- 4 return TREE-SEARCH($x.\text{left}, k$)
- 5 else return TREE-SEARCH($x.\text{right}, k$)

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

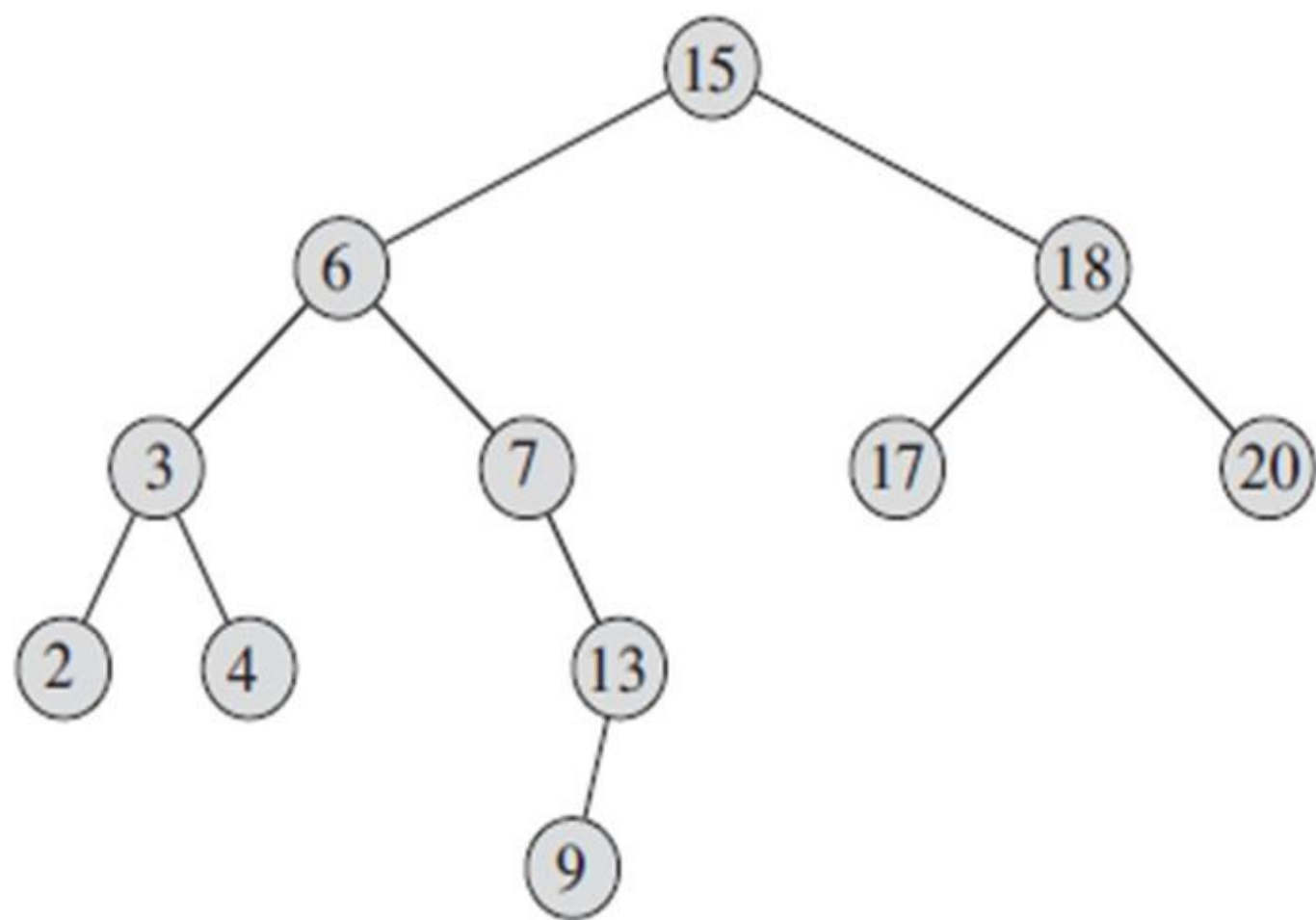
- Write the iterative version of Tree-Search(x,k)
- Generally, iterative version is more efficient than recursive version.
- Why....?

Tree-Search(x, k)

ITERATIVE-TREE-SEARCH(x, k)

```
1  while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2      if  $k < x.\text{key}$ 
3           $x = x.\text{left}$ 
4      else  $x = x.\text{right}$ 
5  return  $x$ 
```

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



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)

1 **while** $x.left \neq \text{NIL}$

2 $x = x.left$

3 **return** x

TREE-MINIMUM(x)

- Correctness:

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

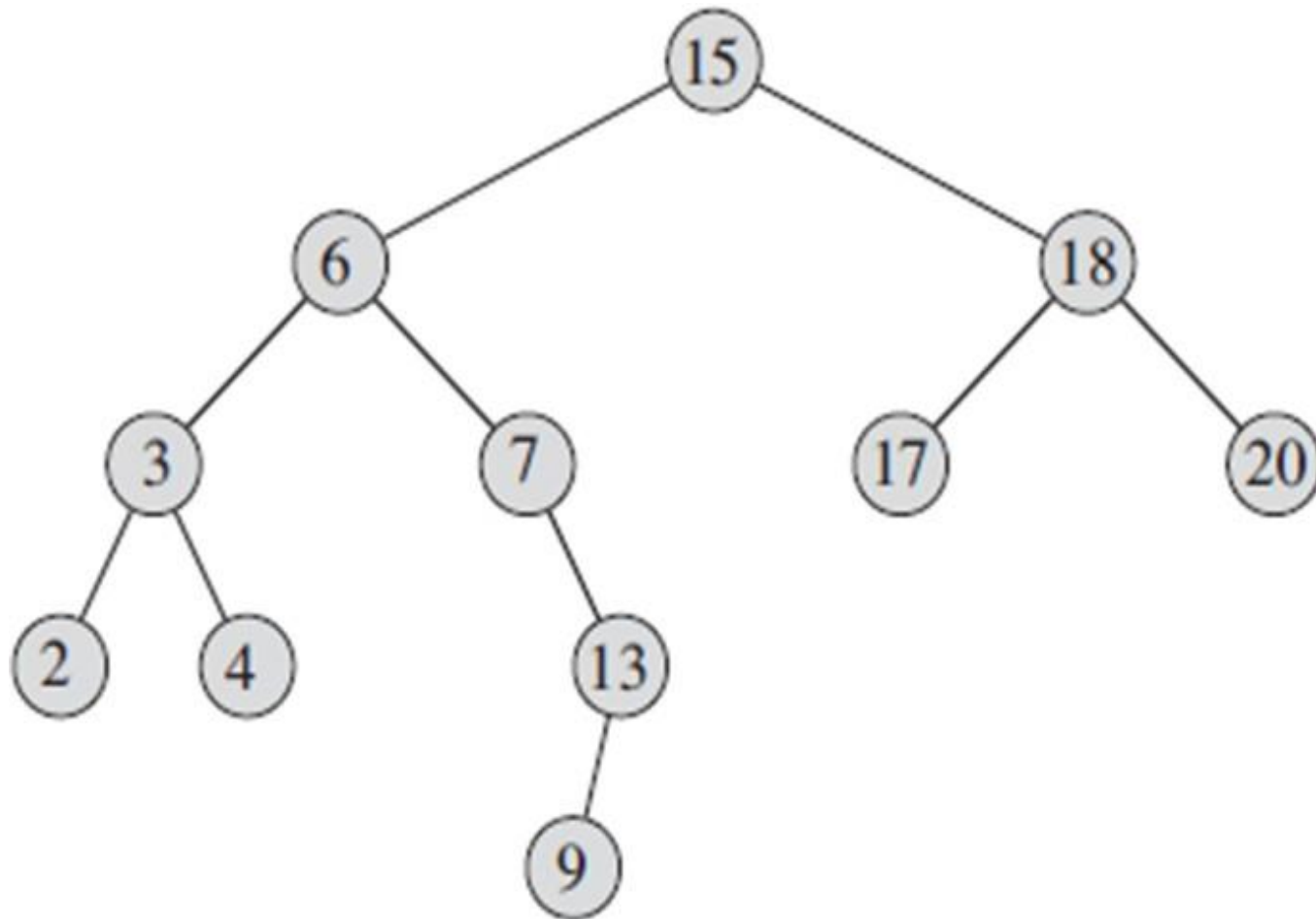
- Node x has no left subtree

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*)

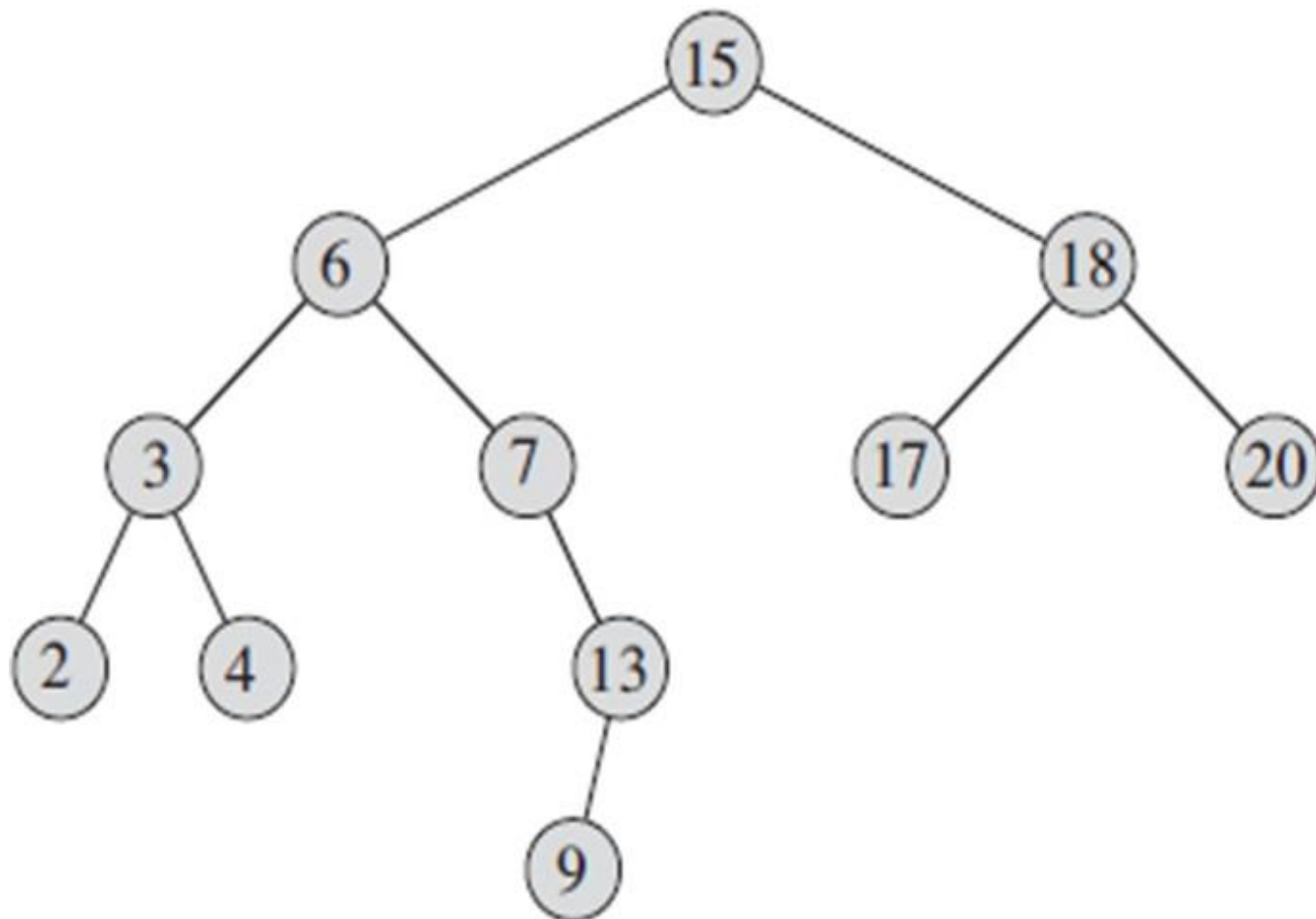
TREE-MAXIMUM(*x*)

```
1  while x.right ≠ NIL
2      x = x.right
3  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



References

- CLRS Book