Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Data Structures and Algorithms
★ Binary Tree ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

Mid-Term

7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

Final-Exam

10 LABS

Objectives

- What is a Trees?
- Tree Terminology
- Tree examples
- Ordered Trees
- Tree Traversals
- Binary Trees
- Implementing Binary Trees
- Binary Search Tree
- Insertion
- Deletion

Introduction

- Trees are one of the fundamental data structures
- Many real-world phenomena cannot be represented in the data structures we've had so far

Array & Linked List

- Array
  - Easy to search
    - $O(\log_2 n)$ – Binary search
  - Slow Insertion & Deletion
- Linked List
  - Easy to insert, delete
  - But slow in searching, deleting the given item, …

Trees

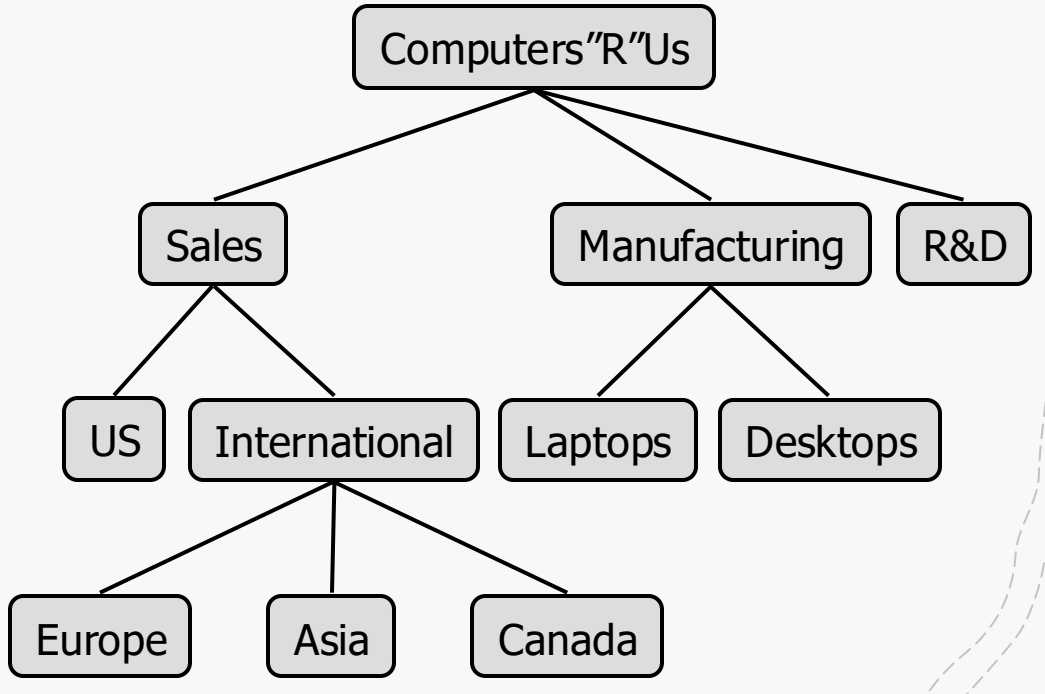
Is a data structure with

- Quick Insertion
- Quick Deletion
- Quick Searching

→ Interesting data structures

What is a Tree?

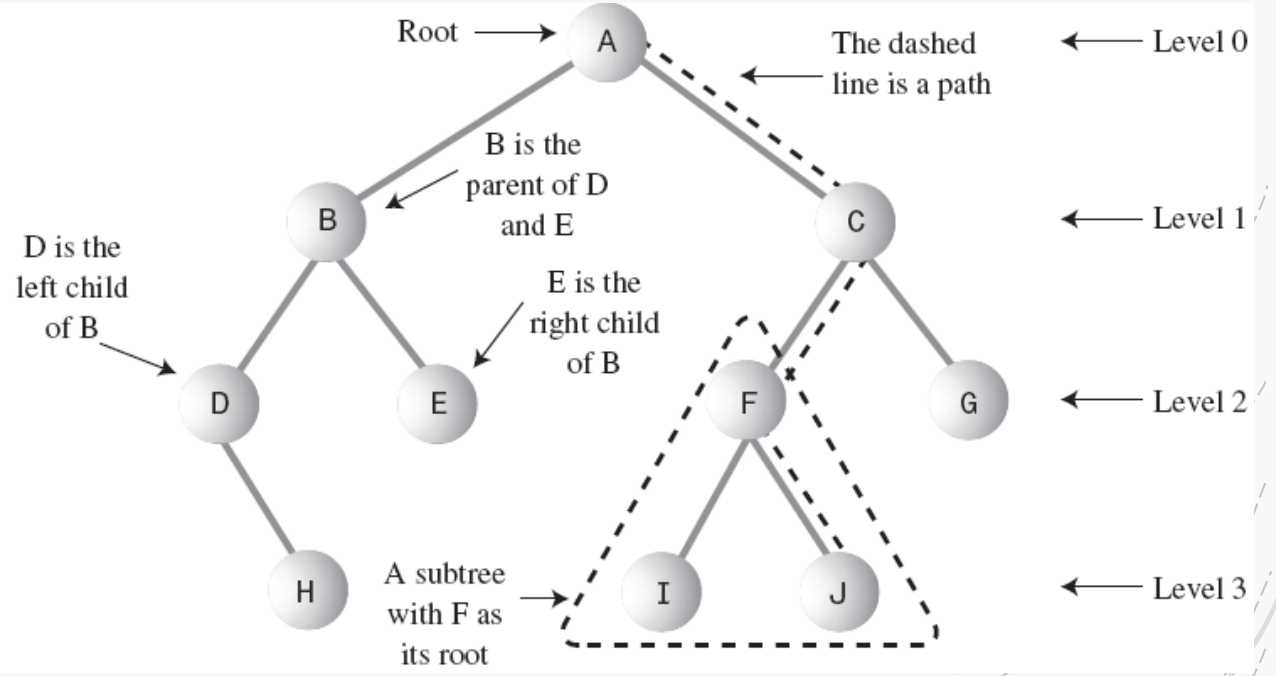
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Inspiration: family trees
- Applications:
  - Organization charts
  - File systems
  - Programming environments
- Every node except one (a root) has a unique parent.



- Empty structure is an empty tree.
- Non-empty tree consists of a root and its children, where these children are also trees.

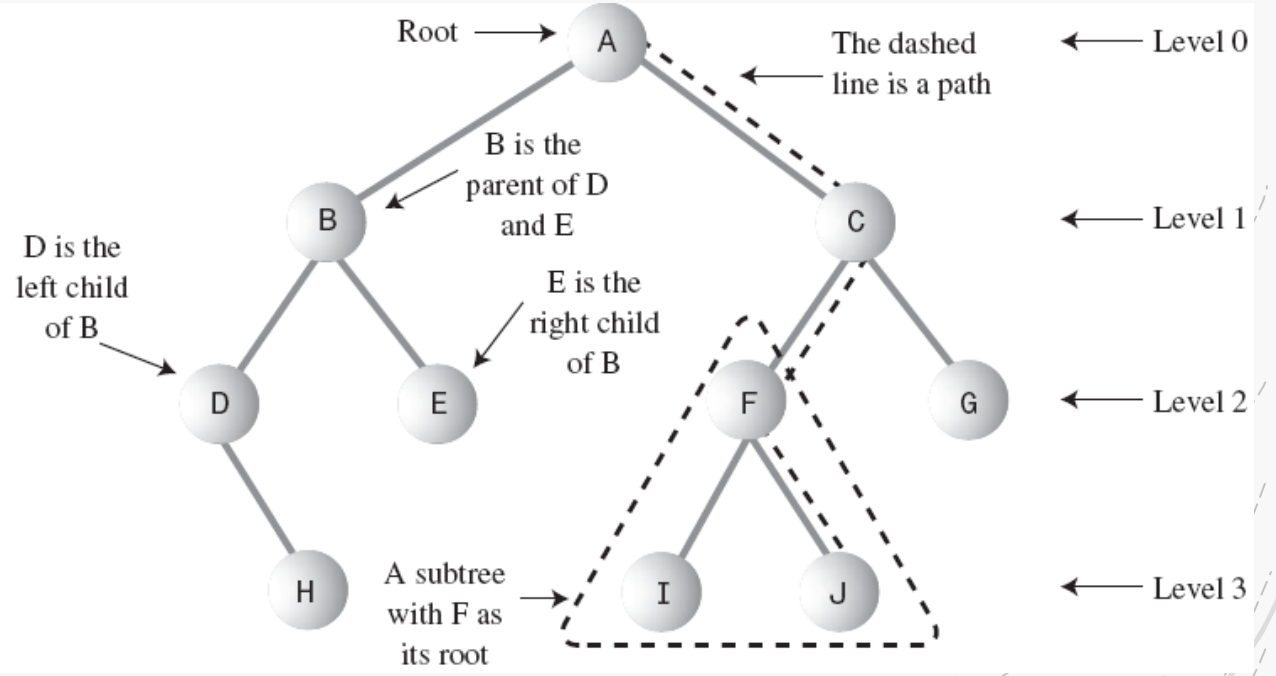
Trees terminology

- **Root**: unique node without a parent
- **Internal** node: node with at least one child (A, B, C, D, F)
- **External (leaf)** node: node without children (E, G, H, I, J)
- **Ancestors** of a node: parent, grandparent, great-grandparent, …
- **Descendants** of a node: child, grandchild, great-grandchild, etc.



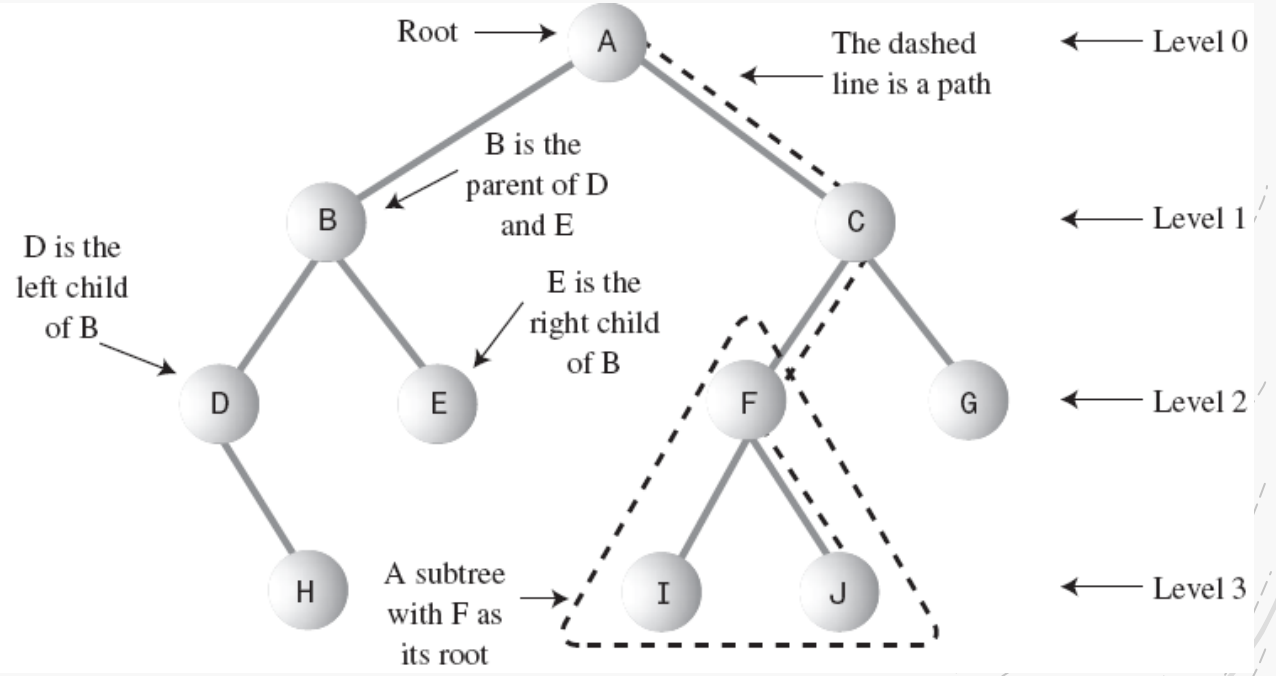
Trees terminology

- **Level** of a node: The level of the root is 1. If a father has level i then its children has level i+1.
  - The level is known with other name: depth.
  - In some documents a level of the root is defined as 0
- **Height** of a tree: maximum level in a tree, thus a single node is a tree of height 1. The height of an empty tree is 0.
- **Height** of a node p is the height of the sub-tree with root p.



Trees terminology

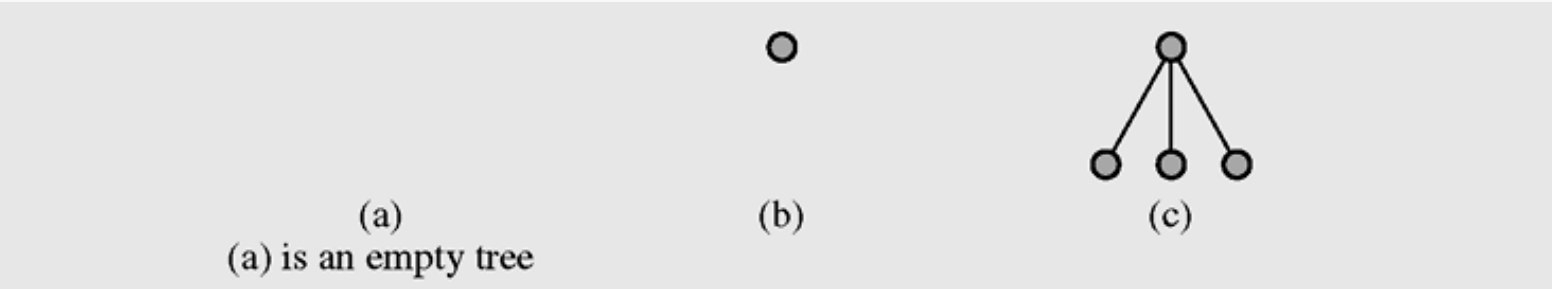
- **Degree (order)** of a node: number of its non-empty children.
- Each node must be reachable from the root through a unique sequence of arcs (edges), called a ***path***
- The number of arcs in a path is called the length of the path

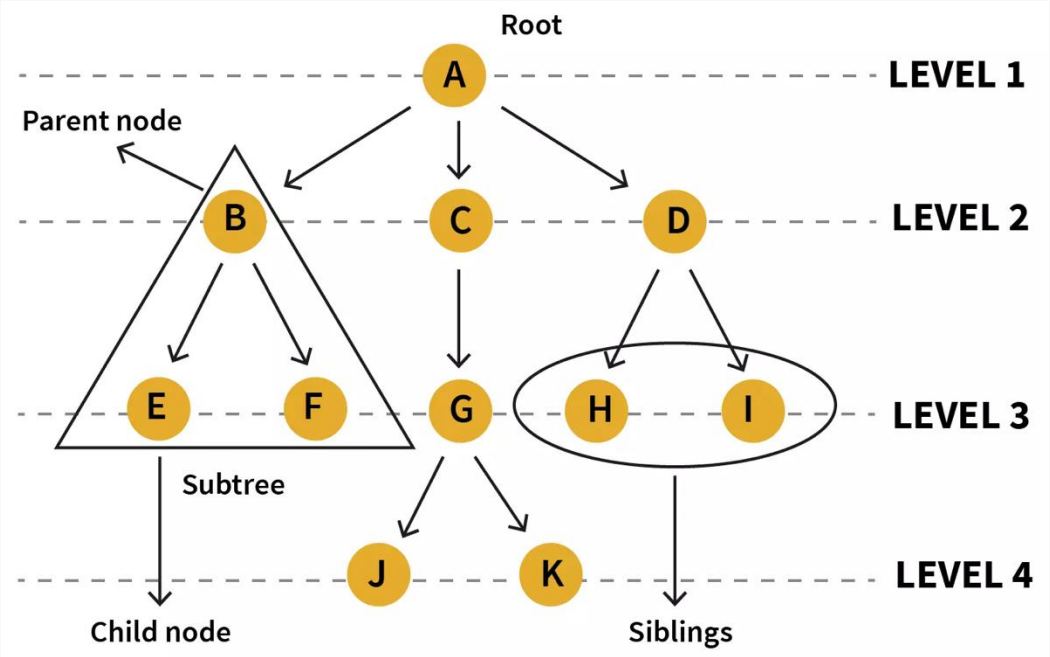


Properties of Trees

- Have one and only one ***Root***
- One and only one ***Path*** from root to any other node
  - → Only one parent
  - → No circles
- Example
  - File hierarchy

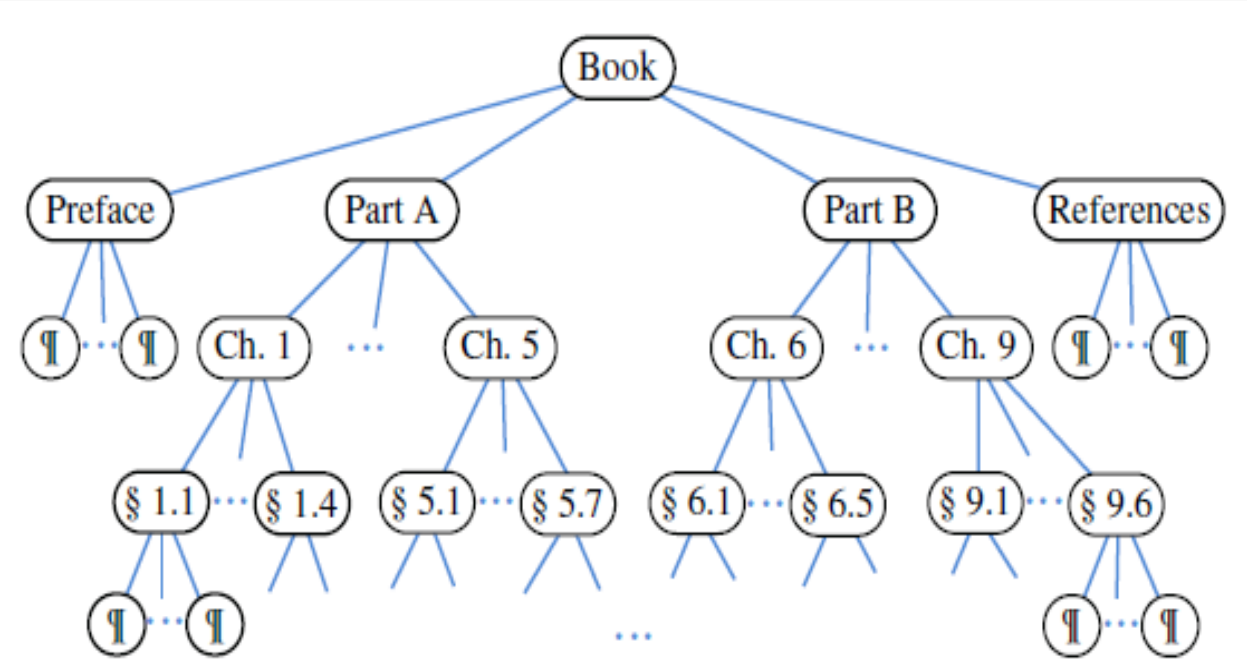
Tree examples





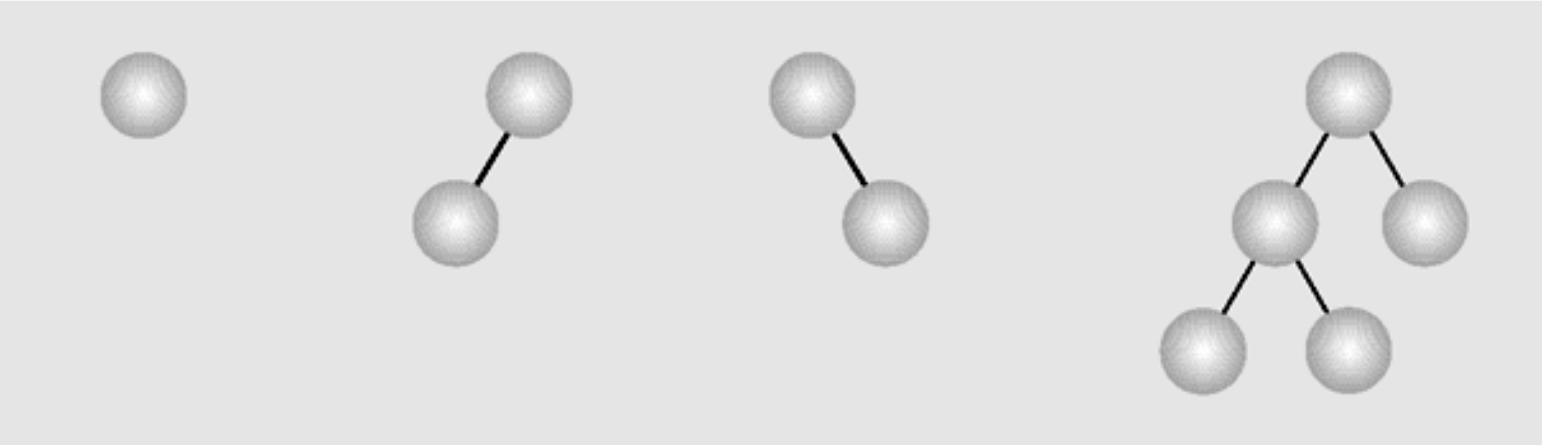
Ordered Trees

- A tree is ordered if there is a meaningful linear order among the children of each node
- Purposefully identify the children of a node as being the first, second, third, and so on.
- Such an order is usually visualized by arranging siblings left to right, according to their order.



Binary Trees

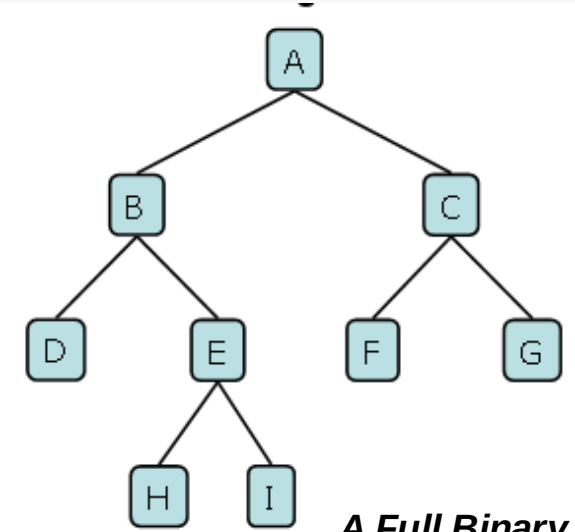
- A binary tree is a tree in which each node has at most two children.
  - An empty tree is also a binary tree.
- Each child may be empty and designated as either a left child or a right child



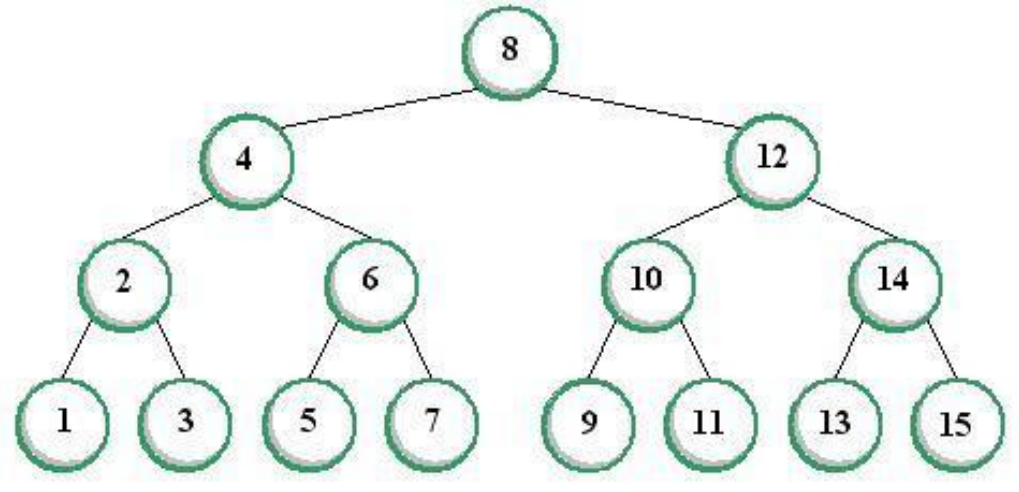
Examples of binary trees

Types of Binary Trees

- In a **proper binary tree** (sometimes **full binary tree** or **2-tree**), every node other than the leaves has two children.
- In a **complete binary tree**, all non-terminal nodes have both their children, and all leaves are at the same level.



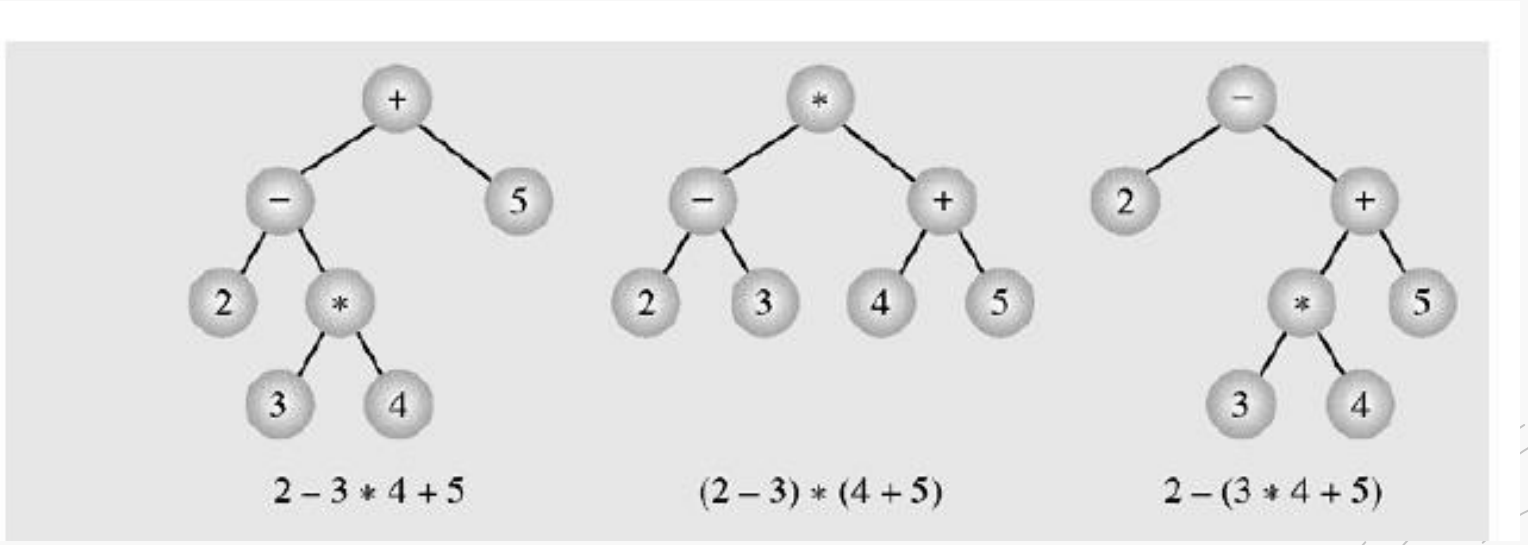
A Full Binary Tree



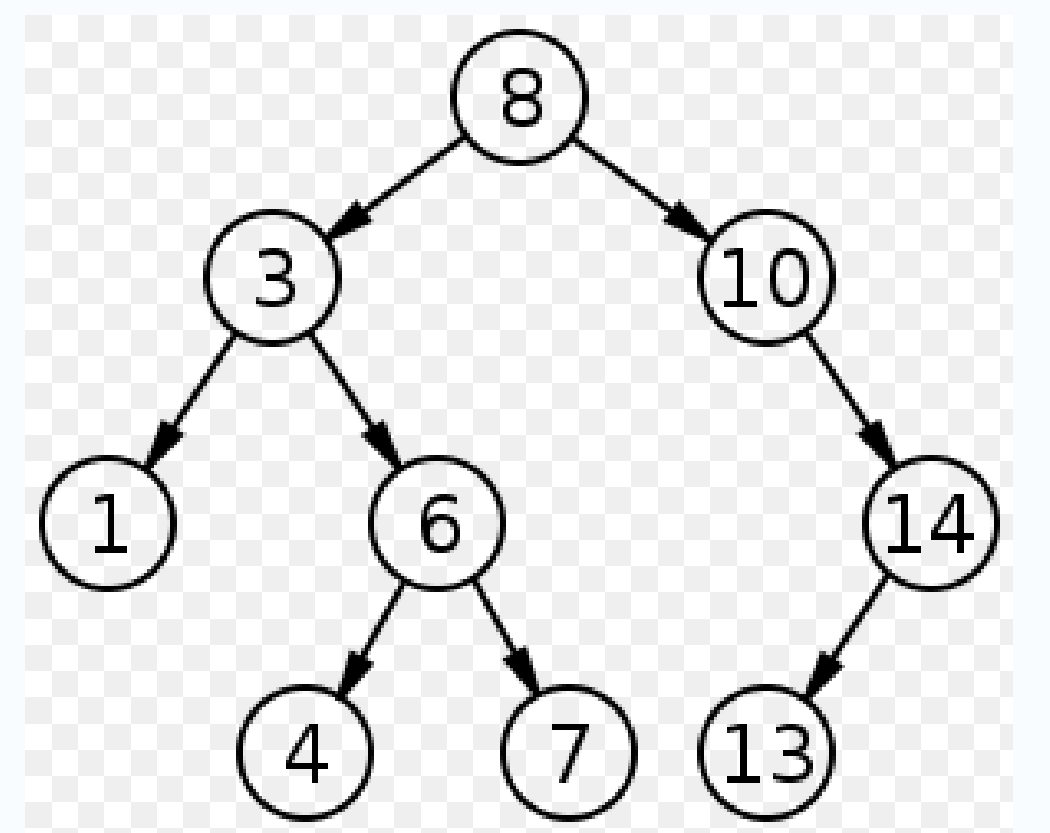
A Complete Binary Tree

Binary Tree example – Expression Tree

- Typical arithmetic expression tree is a proper binary tree, since each operator +, −, ∗, and / takes exactly two operands. Of course, if we were to allow unary operators, like negation (−), as in "−*x*," *then we could have an improper binary* tree.



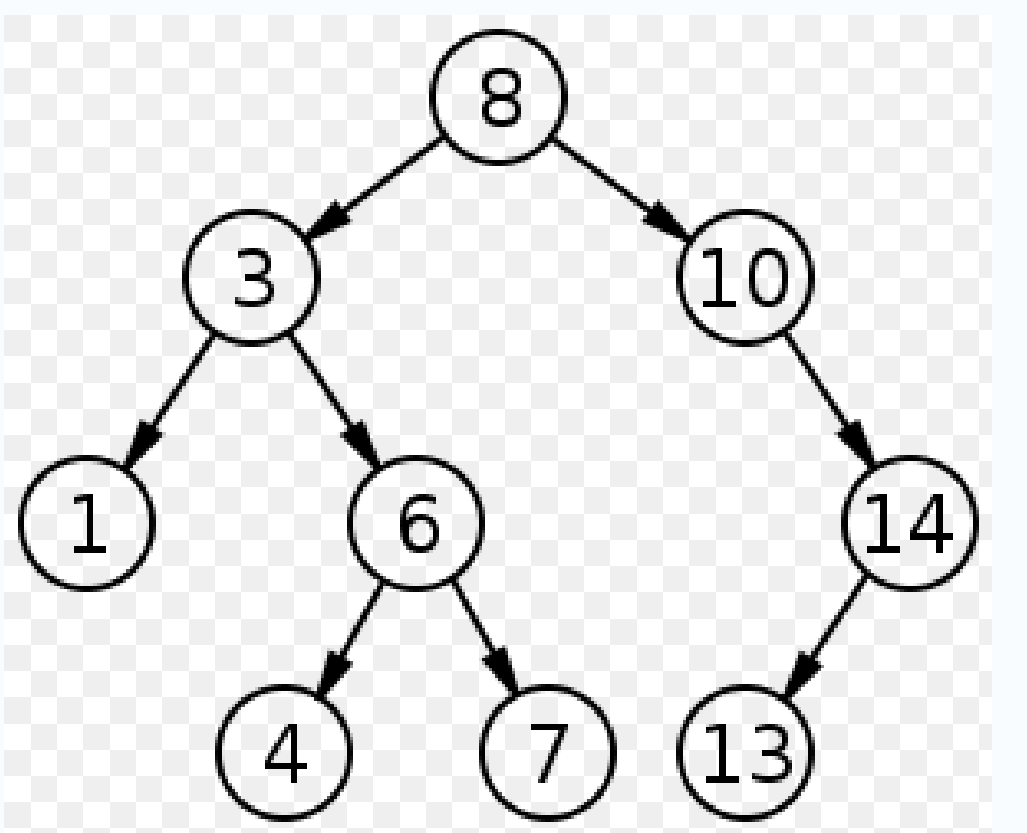
Binary search tree (BST)



- **Properties**
- Node's left child:
  - Key < Key of Parent
- Node's right child:
  - Key > Key of Parent

Binary search tree (BST)

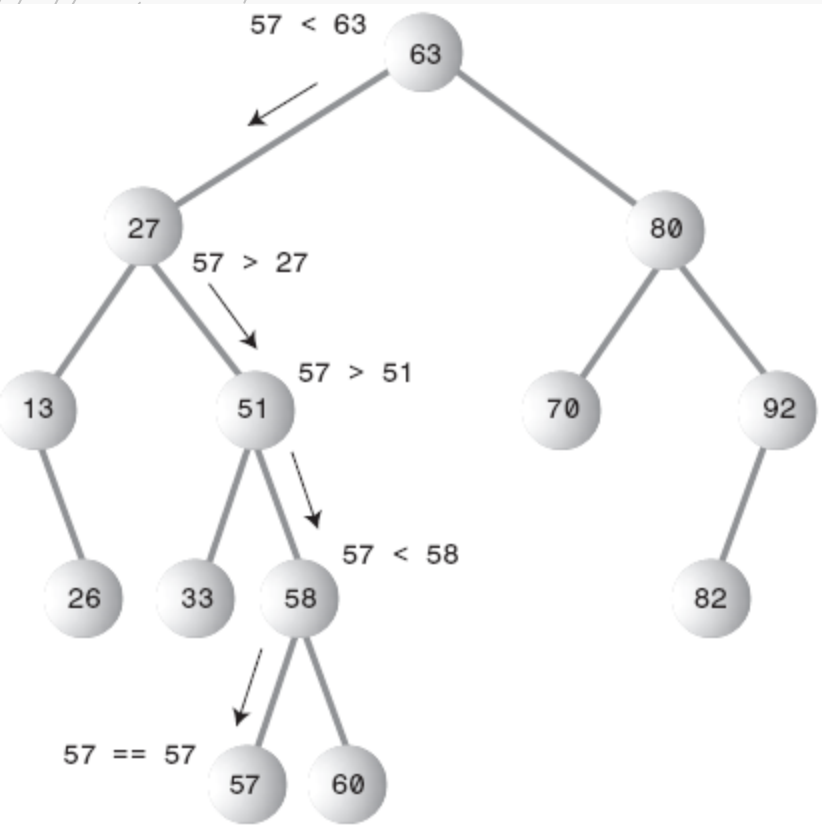
- In computer science, a **binary search tree** (**BST**) is a node based binary tree data structure which has the following properties:
  - The ***left subtree*** of a node contains only nodes with keys ***less than the node's key***.
  - The ***right subtree*** of a node contains only nodes with keys ***greater than the node's key***.
  - Both the left and right subtrees must also be binary search trees.
- From the above properties it naturally follows that:
  - Each node (item in the tree) has a distinct key.



Operations

- Need to carry out basic tree operations such as
  - finding a node,
  - traversing a tree,
  - adding a node,
  - deleting a node, etc.

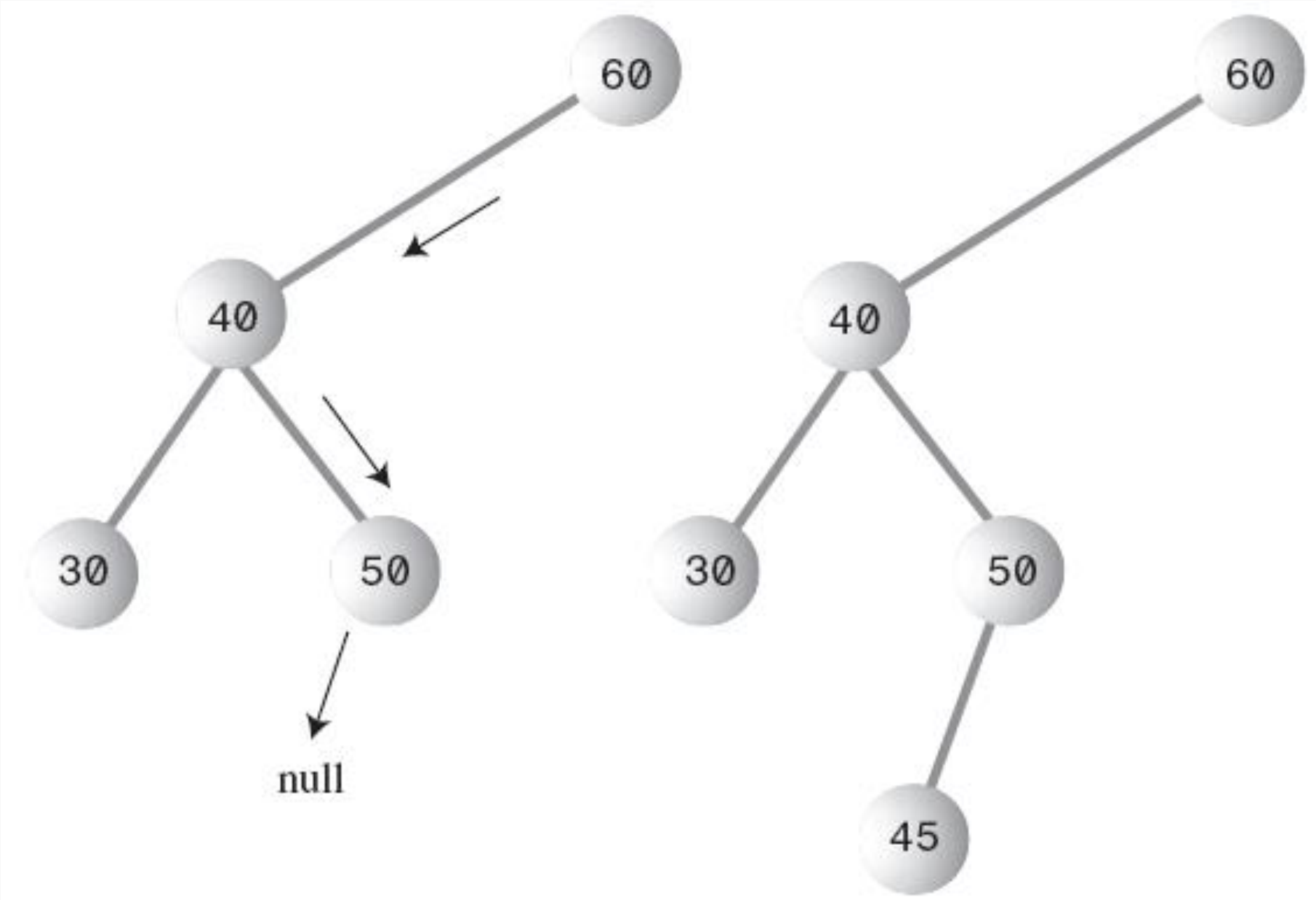
Finding




```
Node current = root;                    // start at root

while(current.iData != key)             // while no match,
   {
   if(key < current.iData)              // go left?
      current = current.leftChild;
   else
      current = current.rightChild;     // or go right?
   if(current == null)                  // if no child,
      return null;                      // didn't find it
   }
return current;                         // found it
```

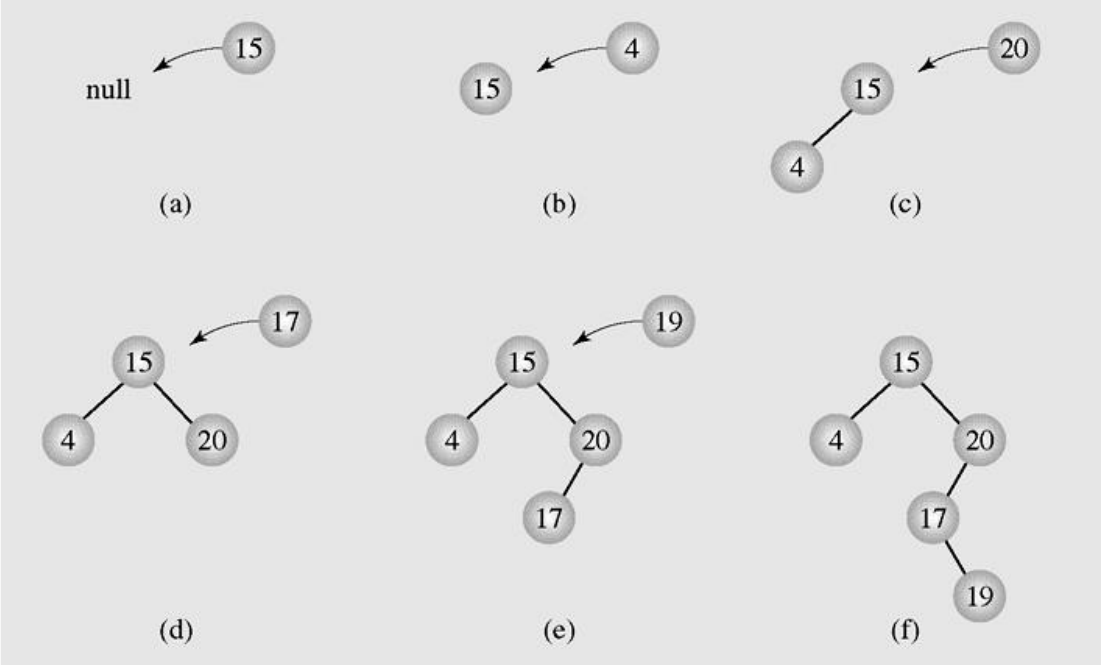
Insert

- Insert 45 into the tree



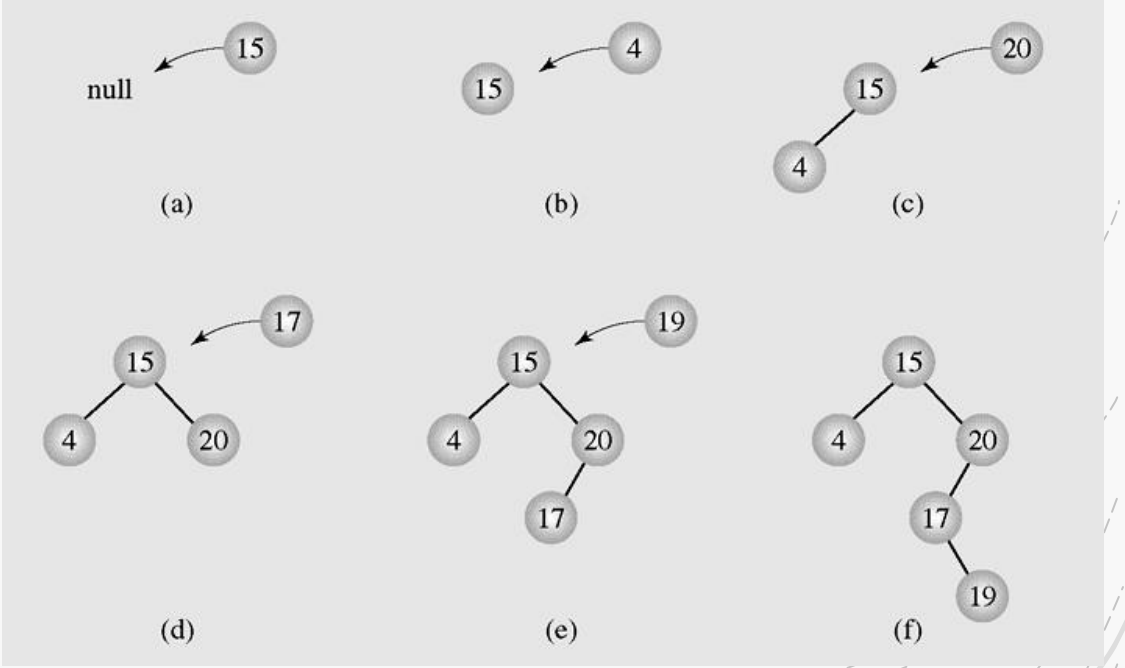
Insertion

- Unless we run out of memory, we always found the place to insert
- Can handle duplication by modifying the search condition.



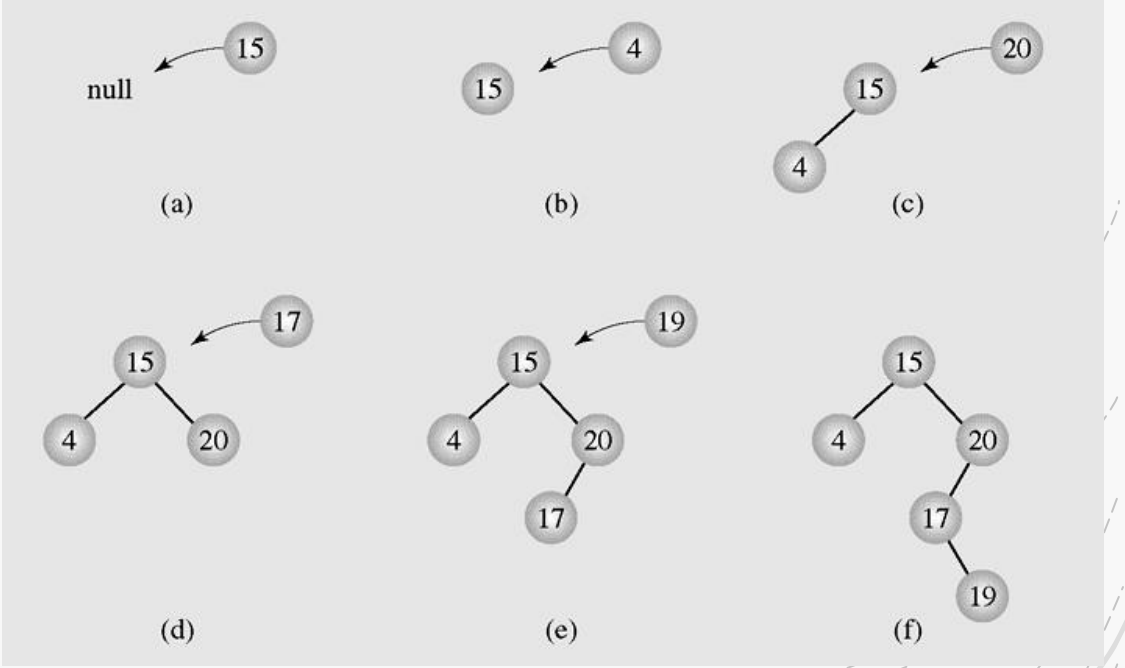
Insert a node

```
Node newNode = new Node();      // make new node
newNode.iData = id;             // insert data
newNode.dData = dd;
if(root==null)                  // no node in root
   root = newNode;
else                            // root occupied
   {
   Node current = root;         // start at root
   Node parent;
```



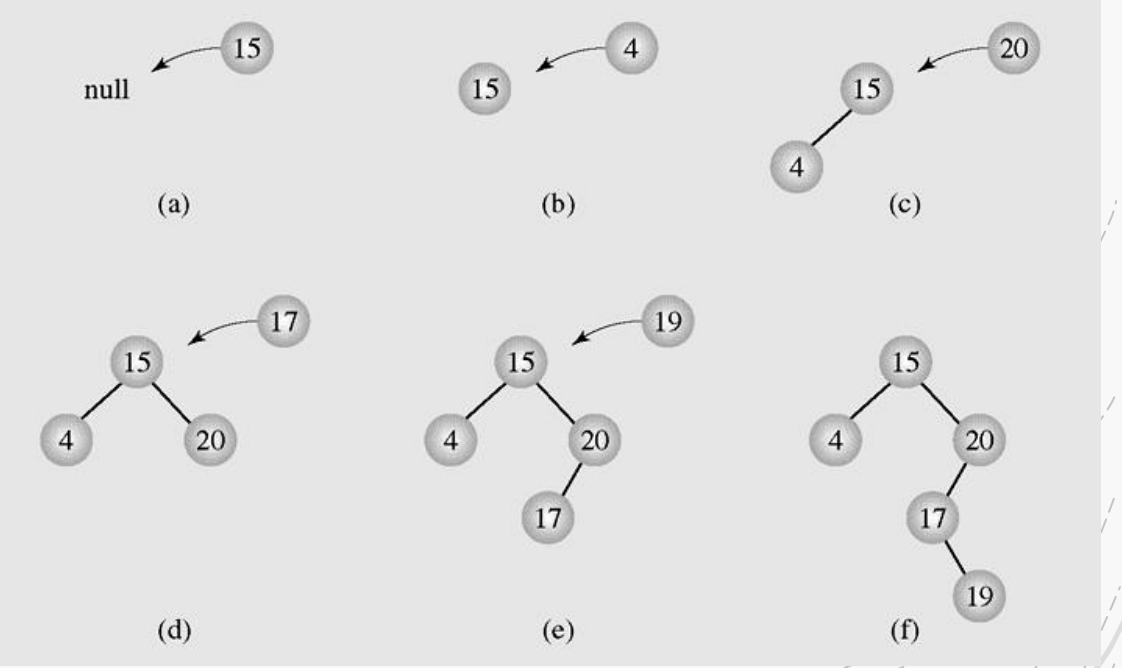
Insert a node (cont.)

```
while(true)                    // (exits internally)
   {
   parent = current;
   if(id < current.iData)      // go left?
      {
      current = current.leftChild;
      if(current == null)      // if end of the line,
         {                     // insert on left
         parent.leftChild = newNode;
         return;
         }
      }  // end if go left
   else                        // or go right?
```



Insert a node (cont.)

```
current = current.rightChild;
if(current == null)  // if end of the line
   {                 // insert on right
   parent.rightChild = newNode;
   return;
   }
```



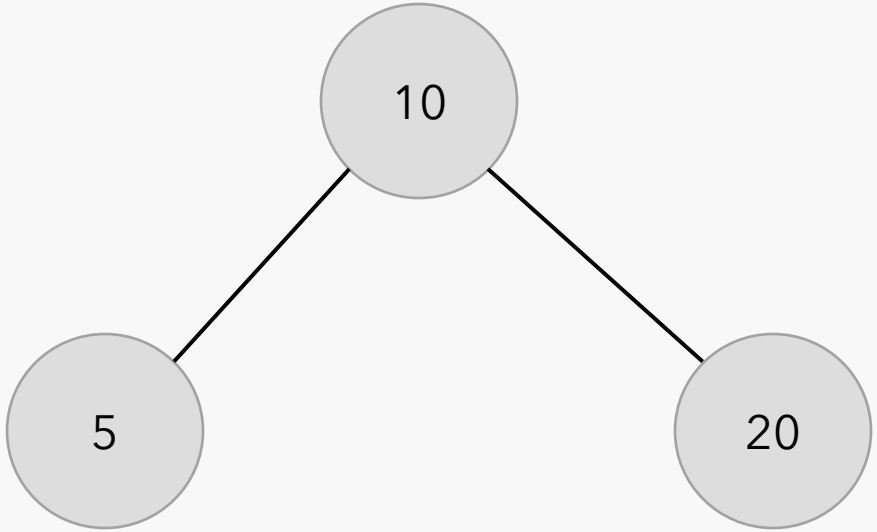
Traversing

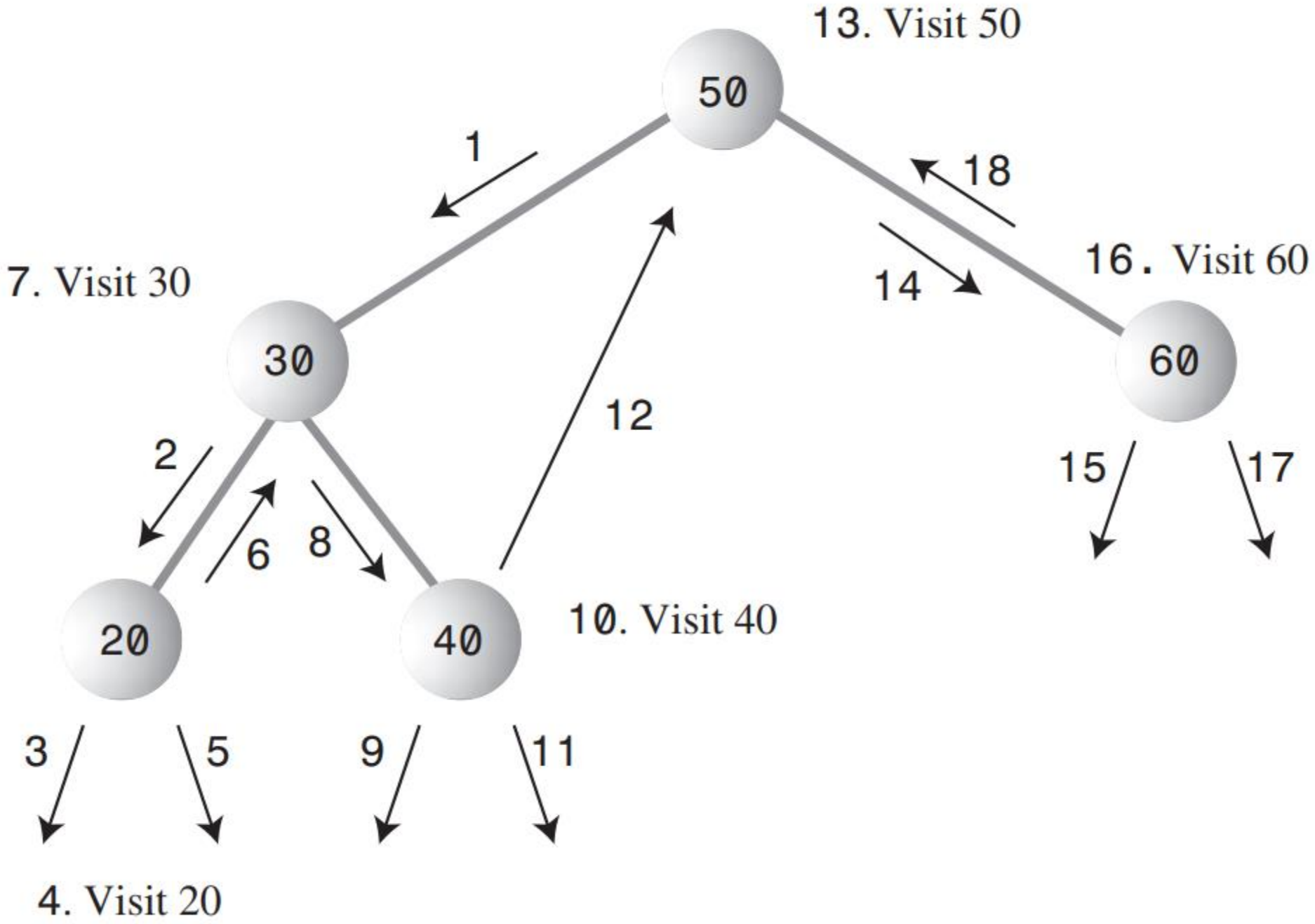
- All the objects stored in an array or linked list can be accessed sequentially
- Visit all nodes of the tree exactly one time in a predictable and efficient manner
- 3 ways to traverse a tree
  - Pre-order
  - In-order
  - Post-order

In-order traversing

- Traverse the node’s left subtree
- Visit the node
- Traverse the node’s right subtree

➔ 5, 10, 20




13. Visit 50
50
1
18
7. Visit 30
16. Visit 60
14
30
60
12
2
15
17
6
8
20
40
10. Visit 40
3
5
9
11
4. Visit 20

Traverse

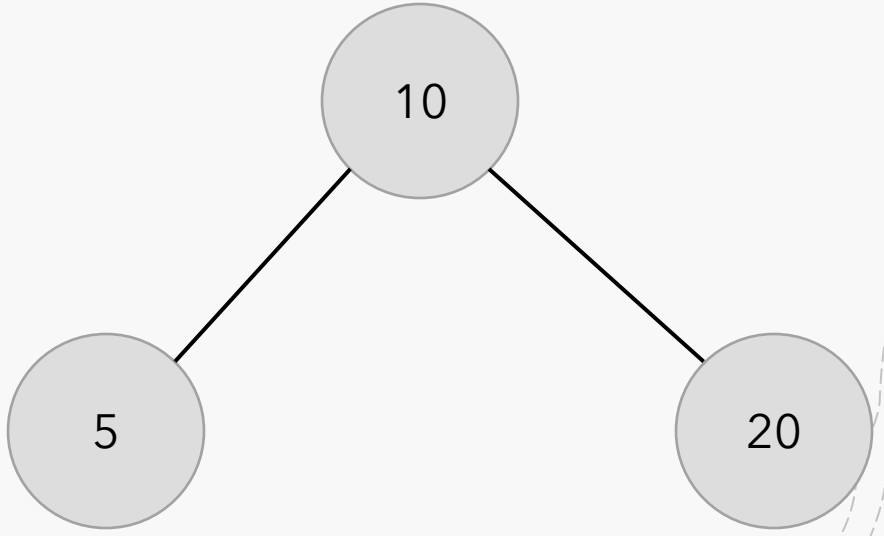
```
private void inOrder(node localRoot)
    {
    if(localRoot != null)
        {
        inOrder(localRoot.leftChild);

        System.out.print(localRoot.iData + " ");
        inOrder(localRoot.rightChild);
        }
    }
```

Pre-order & Post-order

- Pre-order:
  - 1. Visit the node.
  - 2. Call itself to traverse the node's left subtree.
  - 3. Call itself to traverse the node's right subtree.
  - *Application*: print a structured document

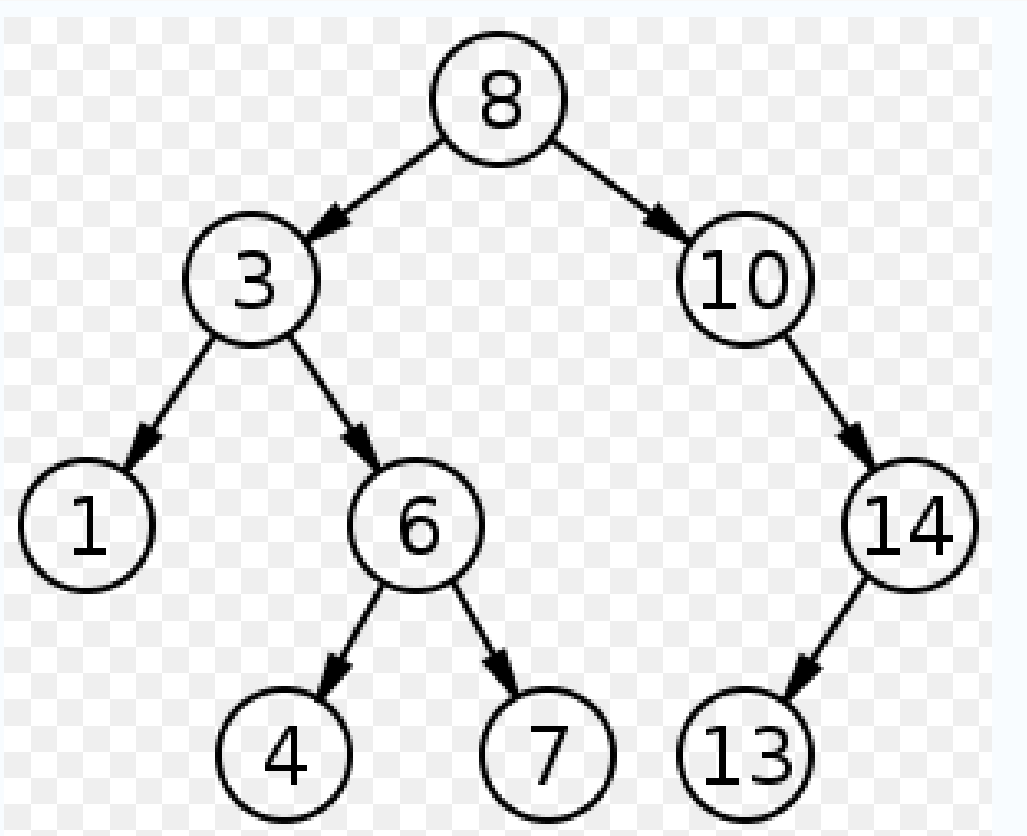
  ➔10, 5, 20
- Post-order:
  - 1. Call itself to traverse the node's left subtree.
  - 2. Call itself to traverse the node's right subtree.
  - 3. Visit the node.
  - *Application*: compute space used by files in a directory and its subdirectories

  ➔ 5, 20, 10

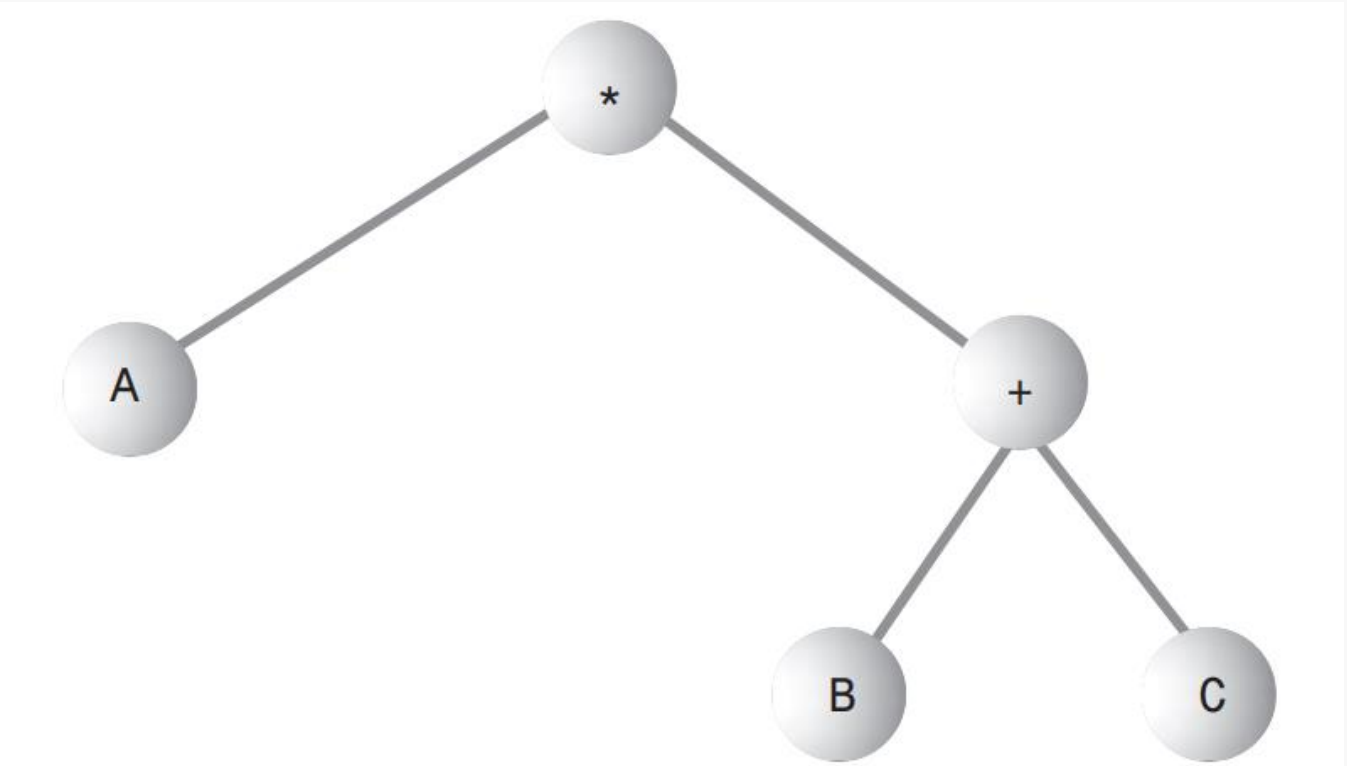


Traversal practice

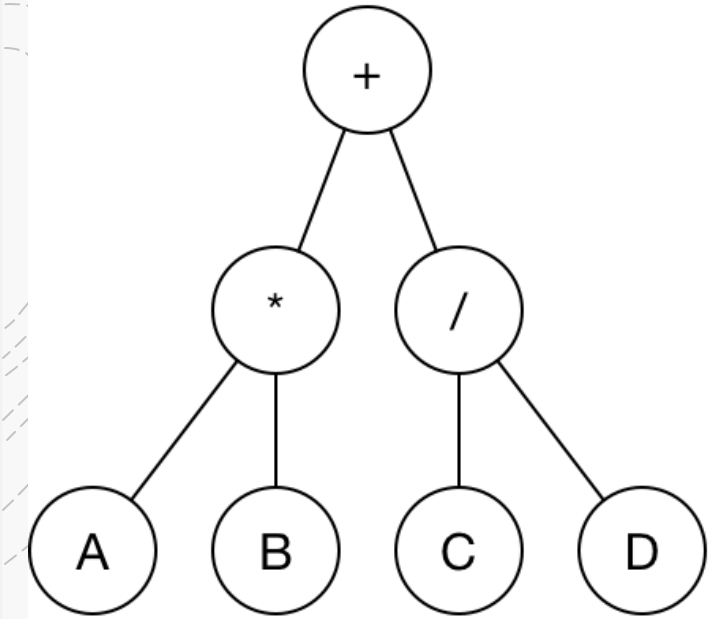
- In-order
- Pre-order
- Post-order

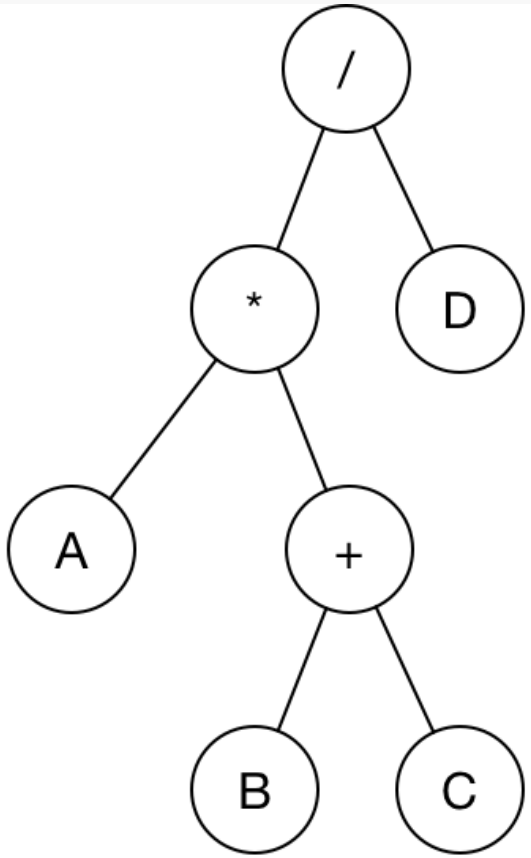


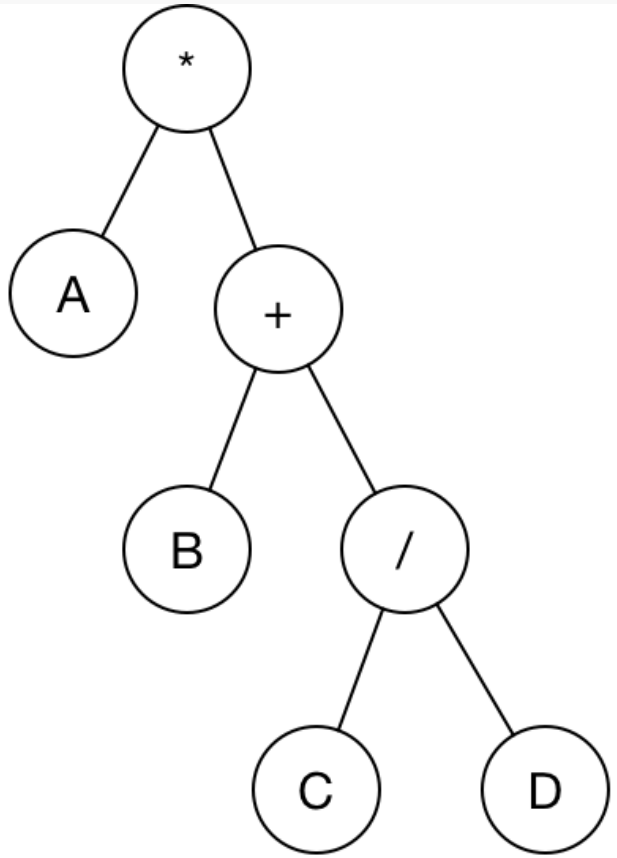
Algebraic expression




```
Infix: A*(B+C)
Prefix: *A+BC
Postfix: ABC+*
```







((A * B) + (C / D))

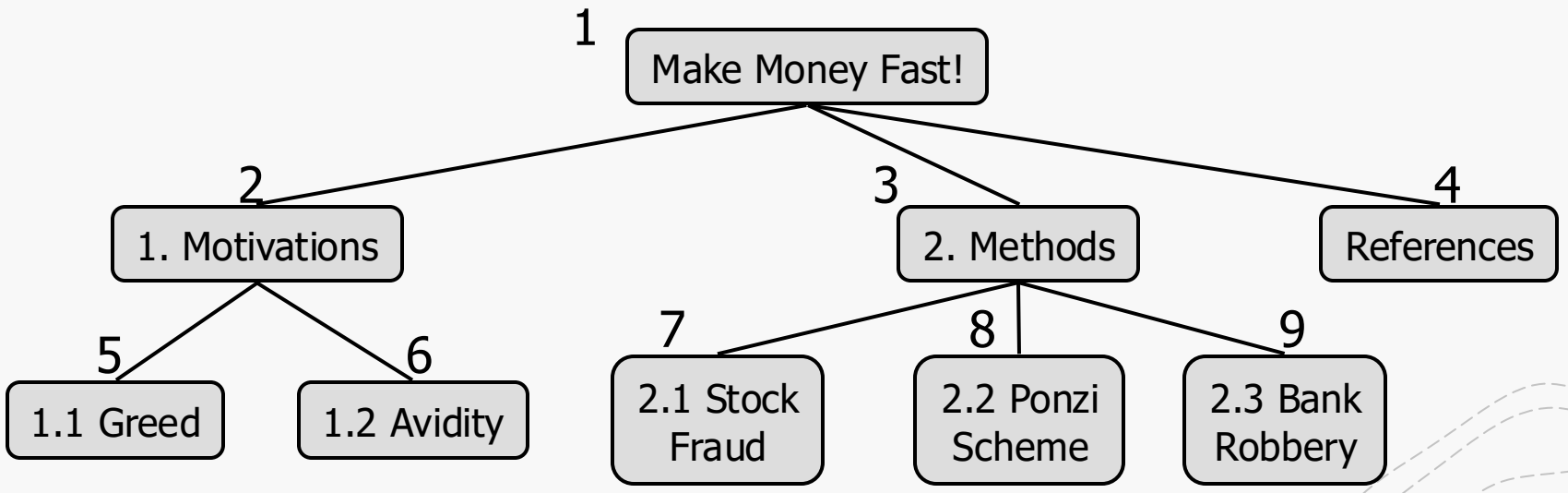
((A * (B + C)) / D)

(A * (B + (C / D)))

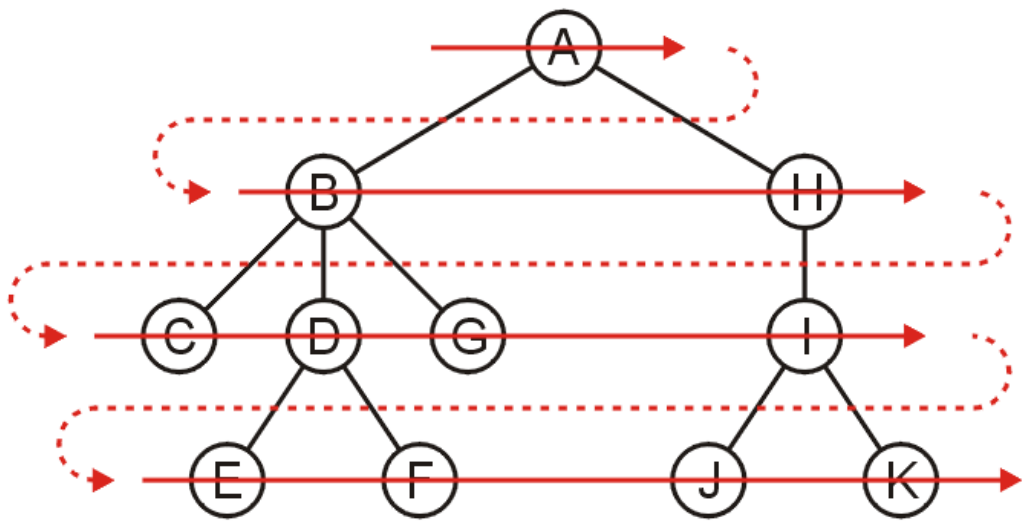
Infix	Postfix	Prefix
A * B + C / D	A B * C D / +	+ * A B / C D
A * (B + C) / D	A B C + * D /	/ * A + B C D
A * (B + C / D)	A B C D / + *	* A + B / C D

Breadth-First Search (BFS) Traversal of a tree

- The breadth-first traversal visits all nodes at depth ***k*** before proceeding onto depth ***k + 1***
- Easy to implement using a queue
- Also called “Level-order traversal”
- Application: visit family tree by generations.



Breadth-First Search (BFS) Traversal of a tree



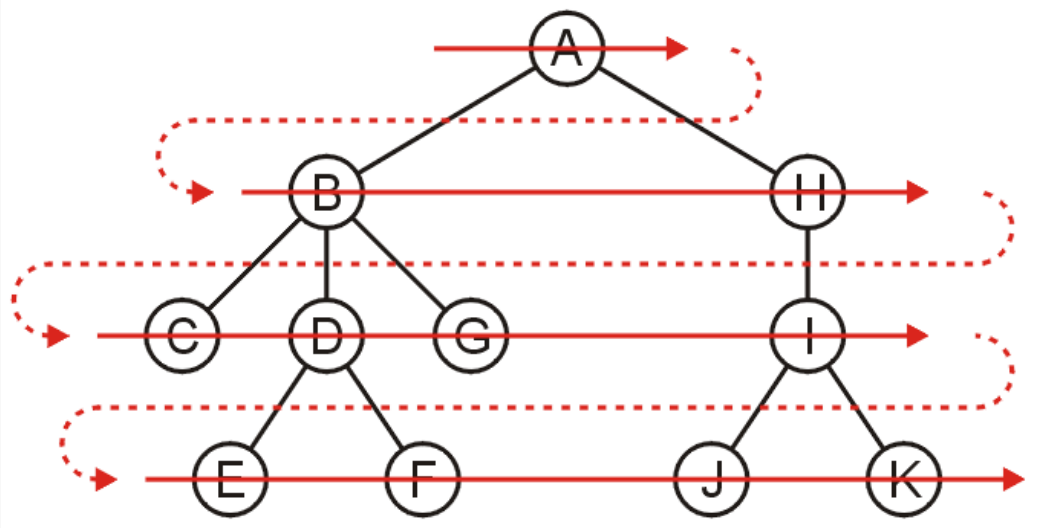
Algorithm *breadthOrder(v)*

- *visit(v)*
- visit all children v1,v2,.. of v
- visit all children of v1, then all children of v2,..

Breadth-First Search (BFS) Traversal of a tree

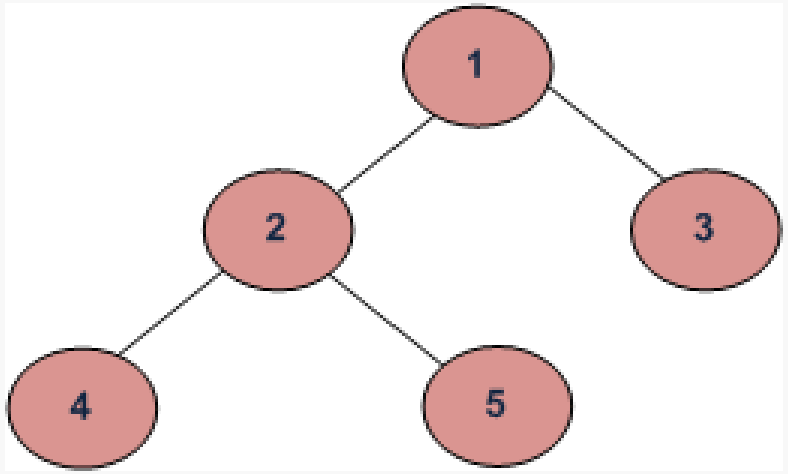
The implementation was already discussed:

- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Push all of its children of the front node onto the queue
  - Pop the front node
- Run time is O(n)
- Order: A B H C D G I E F J K



Breadth-First Search (BFS) Traversal of a tree

- Output?
- BFS Traversal :
  - 1 2 3 4 5
  - or 1 2 3 5 4
  - or 1 3 2 4 5
  - or 1 3 2 5 4

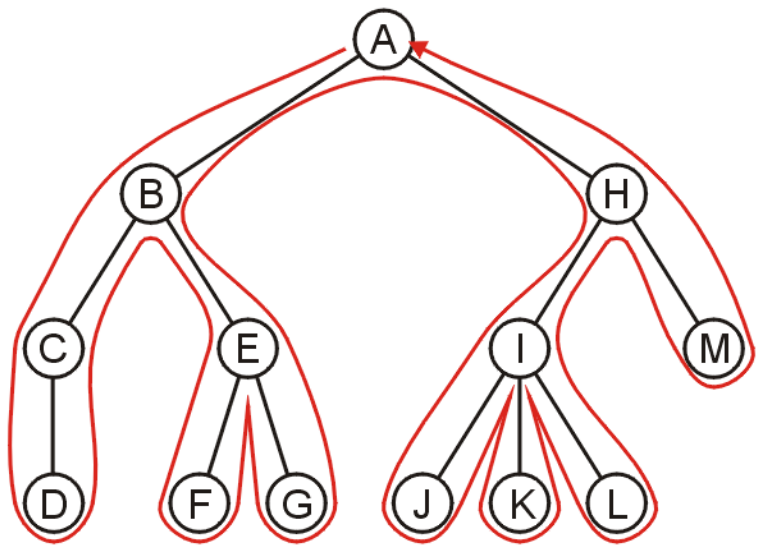


Algorithm *breadthOrder(v)*

- *visit*(*v*)
- visit all children v1,v2,.. of v
- visit all children of v1, then all children of v2,..

Depth-First Search (DFS) Traversal of a tree

- Go as deep as possible before visiting other siblings: ***depth-first traversals***
- The key idea behind DFS is backtracking
  - At any node, we proceed to the first child that has not yet been visited
  - Or, if we have visited all the children (of which a leaf node is a special case), we backtrack to the parent and repeat this decision-making process
- We end once all the children of the root are visited



Depth-First Search (DFS) Traversal of a tree

- The key idea behind DFS is backtracking

N = node
L = left
R = right

Pre-order (NLR)

Algorithm *preOrder*(*v*)
    visit(*v*)
    preOrder (left child of v)
    preOrder (right child of v)

In-order (LNR)

Algorithm *inOrder*(*v*)
    inOrder (left child of v)
    visit(*v*)
    inOrder (right child of v)

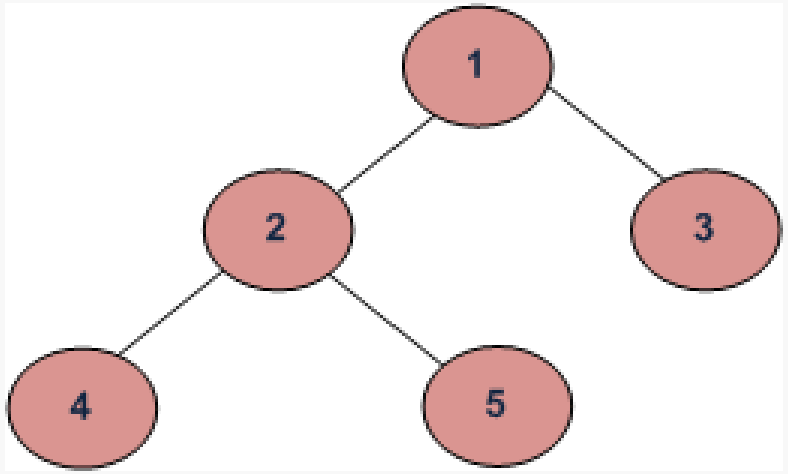
Post-order (LRN)

Algorithm *postOrder*(*v*)
    postOrder (left child of v)
    postOrder (right child of v)
    visit(*v*)

Depth-First Search (DFS) Traversal of a tree

N = node, L = left, R = right

- Output?



- DFS Traversal :
  - Pre-order Traversal : 1 2 4 5 3
  - In-order Traversal : 4 2 5 1 3
  - Post-order Traversal : 4 5 2 3 1

Pre-order (NLR)

Algorithm *preOrder*(*v*)
  visit(*v*)
  preOrder (left child of v)
  preOrder (right child of v)

In-order (LNR)

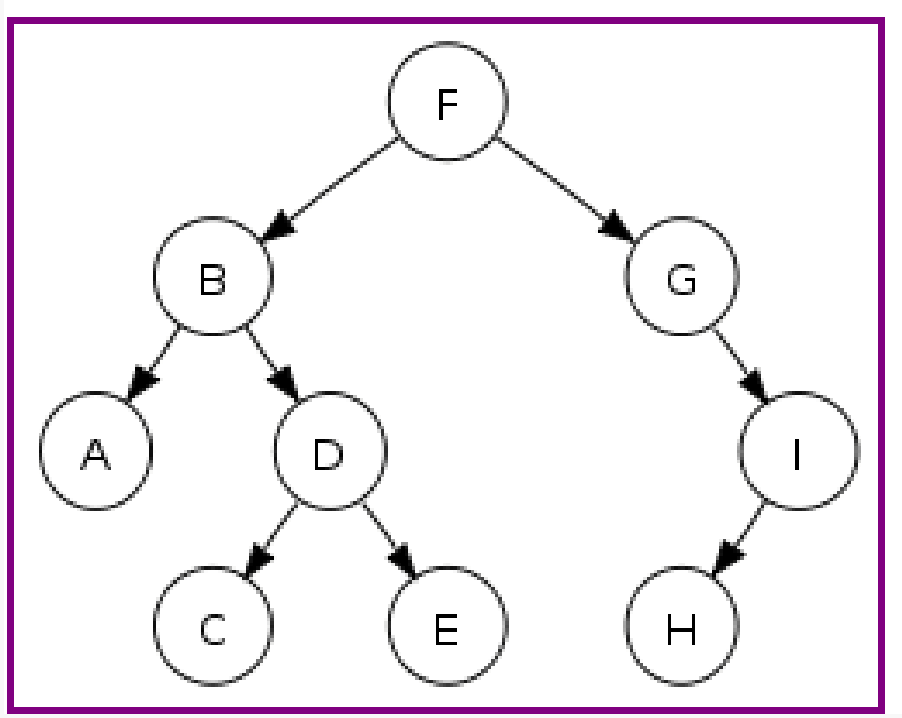
Algorithm *inOrder*(*v*)
  inOrder (left child of v)
  visit(*v*)
  inOrder (right child of v)

Post-order (LRN)

Algorithm *postOrder*(*v*)
  postOrder (left child of v)
  postOrder (right child of v)
  visit(*v*)

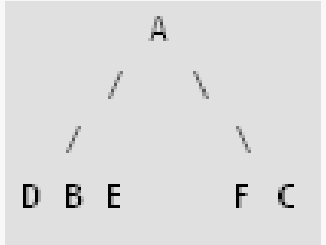
Binary Tree Traversal example

- Pre-order traversal:
  - F, B, A, D, C, E, G, I, H (root, left, right)
- In-order traversal:
  - A, B, C, D, E, F, G, H, I (left, root, right)
- Post-order traversal:
  - A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal (breadth first):
  - F, B, G, A, D, I, C, E, H

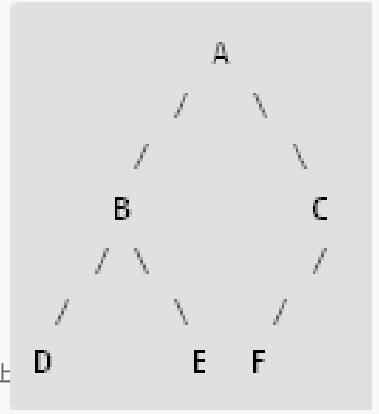


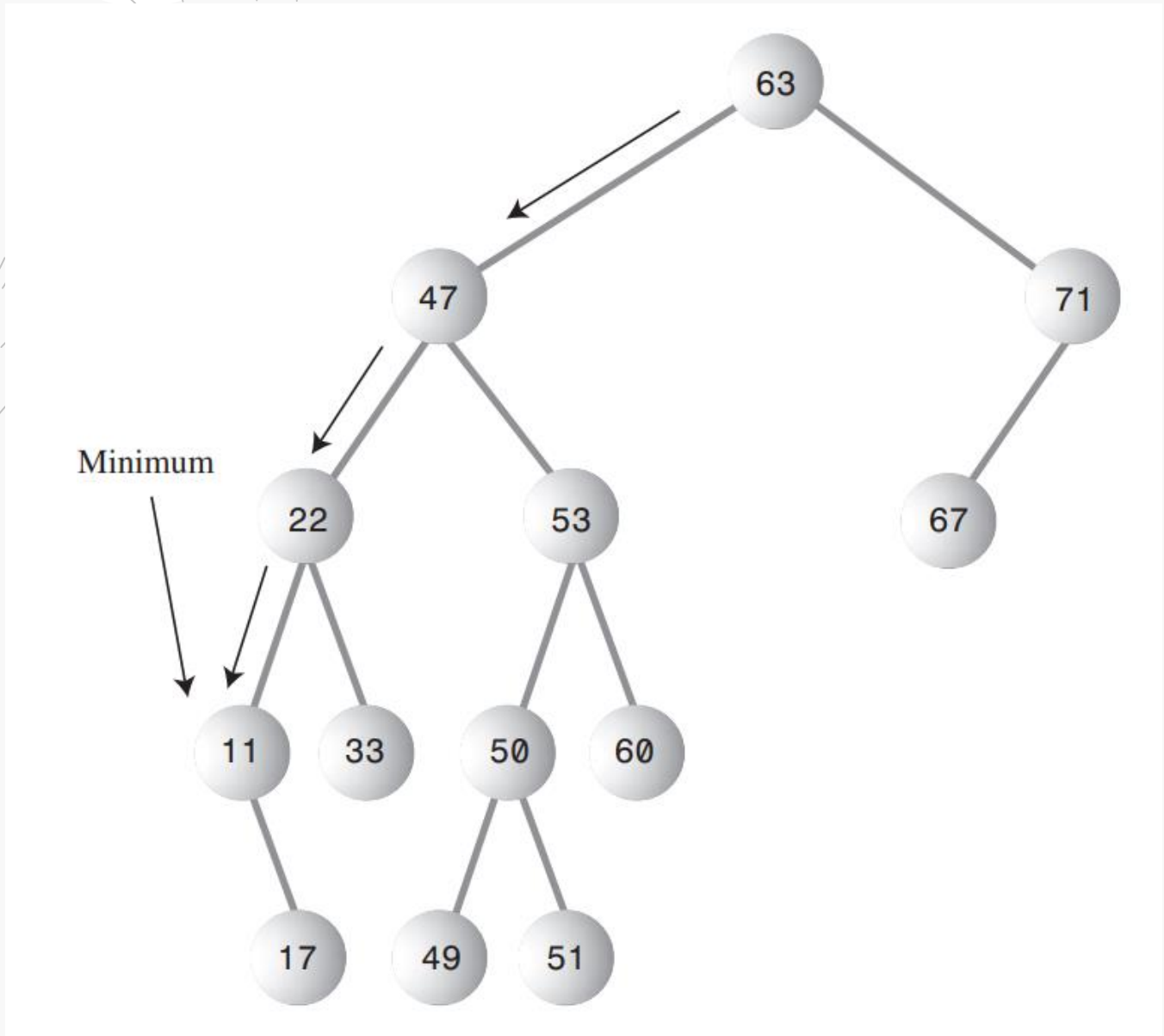
Construct Binary Tree from given traversals

- From given traversals we can construct the tree. More exactly speaking, we can construct a tree from in-order & pre-order or in-order & post-order traversals.
- Let us consider the below traversals:
  - In-order sequence: D B E A F C
  - Pre-order sequence: A B D E C F
- In a Preorder sequence, leftmost element is the root of the tree. So we know 'A' is root for given sequences.
- By searching 'A' in In-order sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So we know below structure now:



- We recursively follow above steps and get the following tree:





Find max & min

Find min

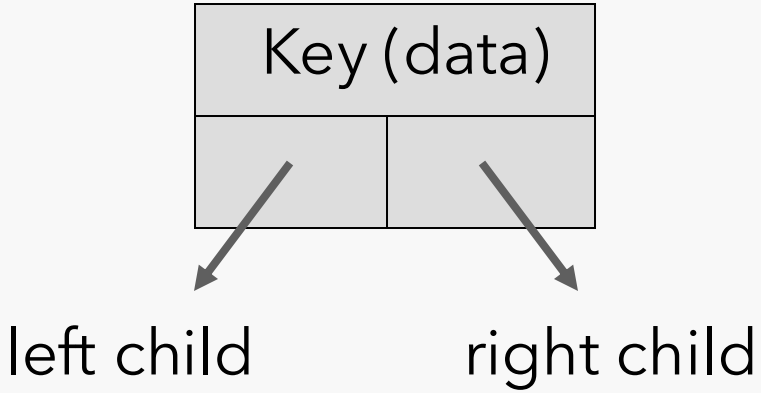
```
public Node minimum()      // returns node with minimum key value
   {
   Node current, last;
   current = root;                    // start at root
   while(current != null)             // until the bottom,
     {
     last = current;                    // remember node
     current = current.leftChild;       // go to left child
     }
  return last;
  }
```

Implementing Binary Trees

- Binary trees can be implemented in at least two ways:
  - As arrays
  - As linked structures
- To implement a tree as an array, a node is declared as an object with an information field and two “reference” fields. These reference fields contain the indexes of the array cells in which the left and right children are stored, if there are any.
- However, it is hard to predict how many nodes will be created during a program execution. (Therefore, how many spaces should be reserved for the array?)

Implementing Binary Trees

- To implement a tree as a linked structure, a node is declared as an object with an information field and two “reference” fields.

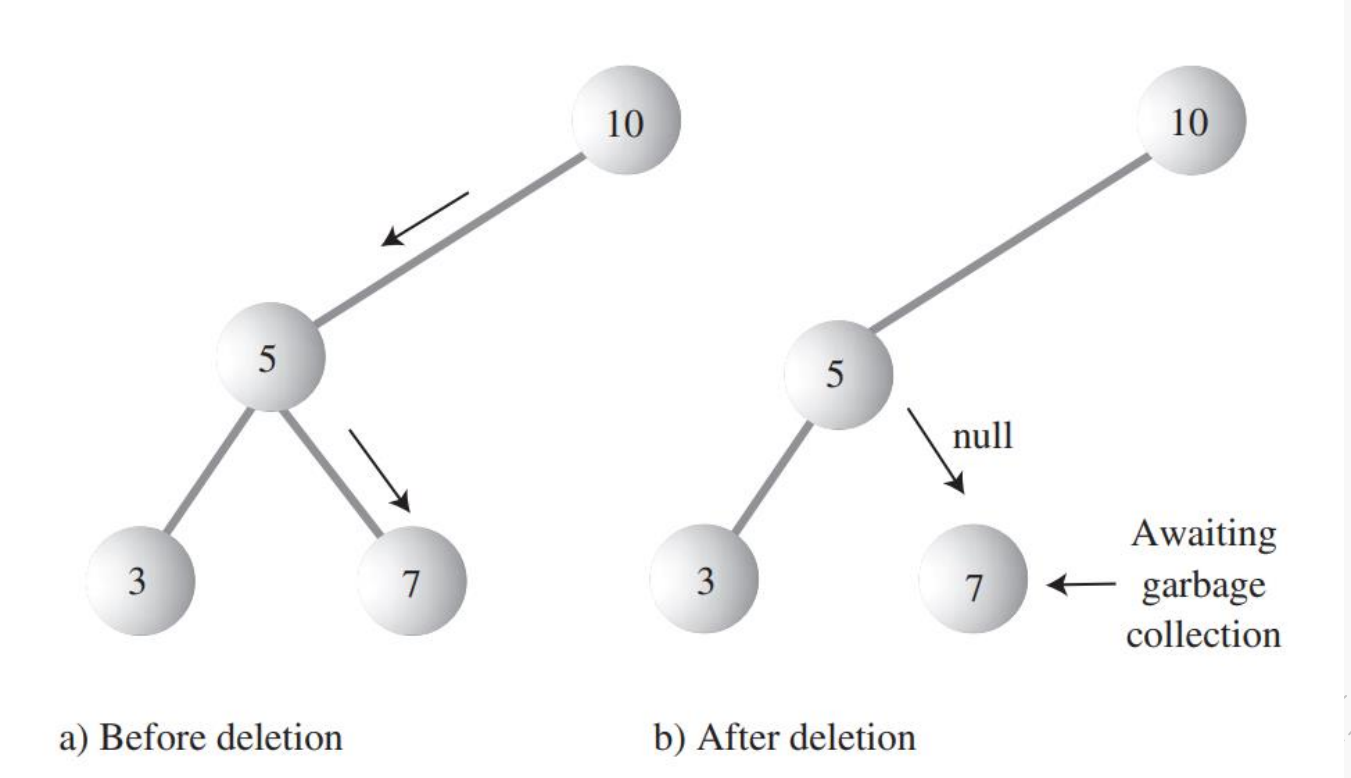



```
class Node
{
    int value;
    Node leftChild;
    Node rightChild;
}
```

Deleting a Node

- Most complicated operation
- 3 main cases
  - Node to be deleted is a leaf
  - Node to be deleted has one child
  - Node to be deleted has two children

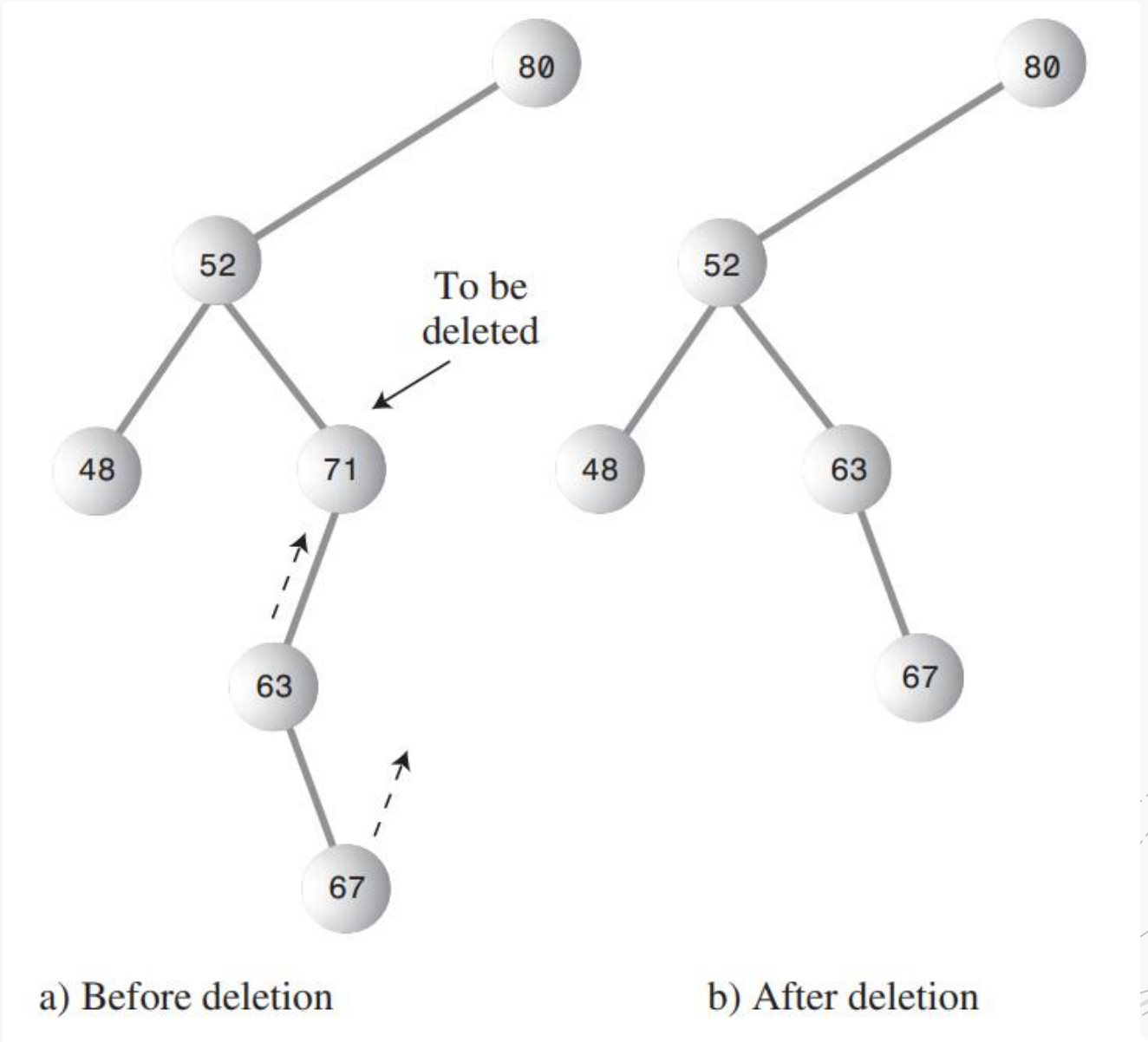
Case 1: Node has no children



Case 1: Node has no children

```
// delete() continued...
// if no children, simply delete it
if(current.leftChild==null &&
                              current.rightChild==null)
   {
   if(current == root)              // if root,
      root = null;                  // tree is empty
   else if(isLeftChild)
      parent.leftChild = null;      // disconnect
   else                             // from parent
      parent.rightChild = null;
   }
```

Case 2: Node has one child



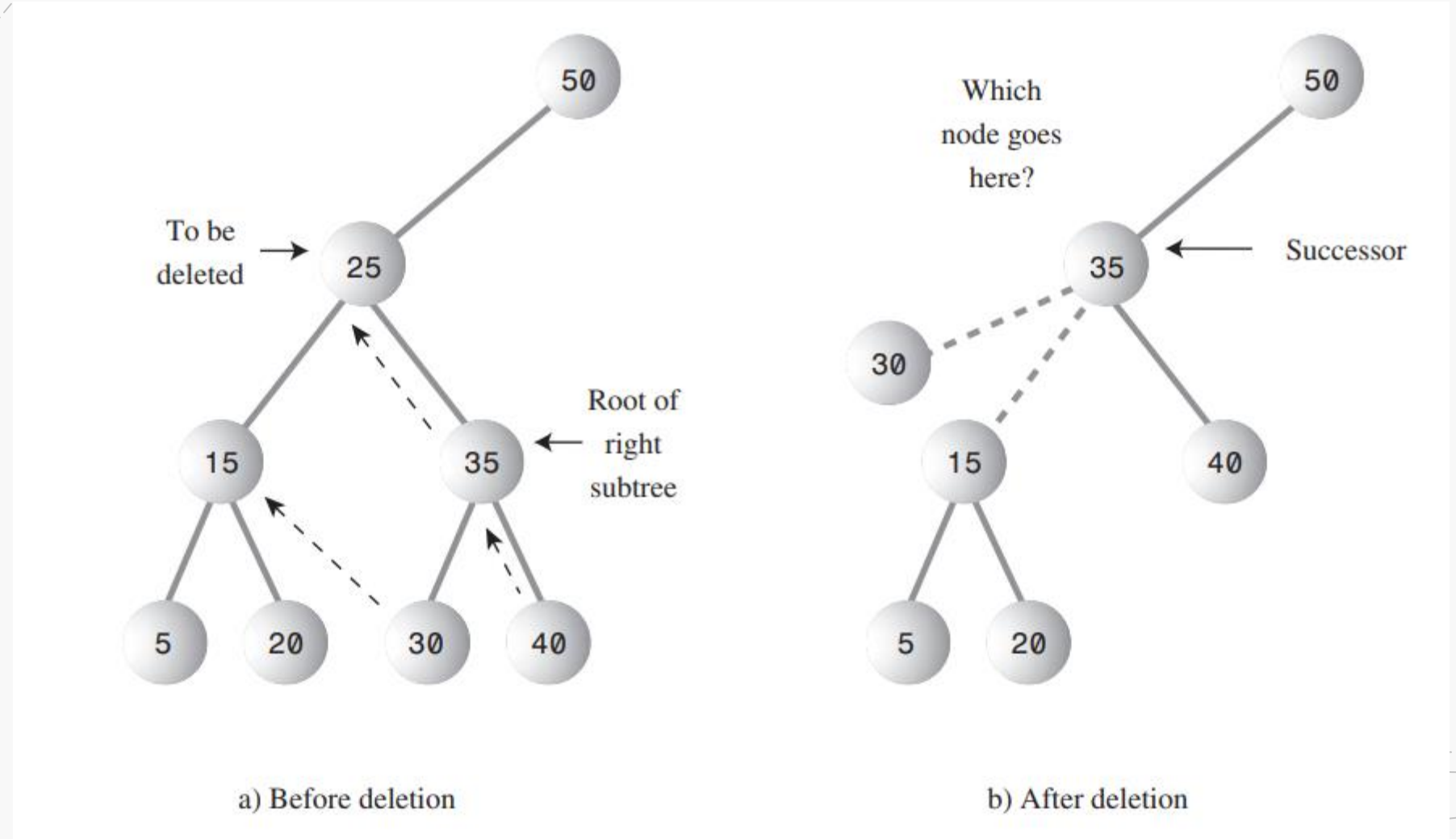
Case 2: Node has one child

- There are four variations:

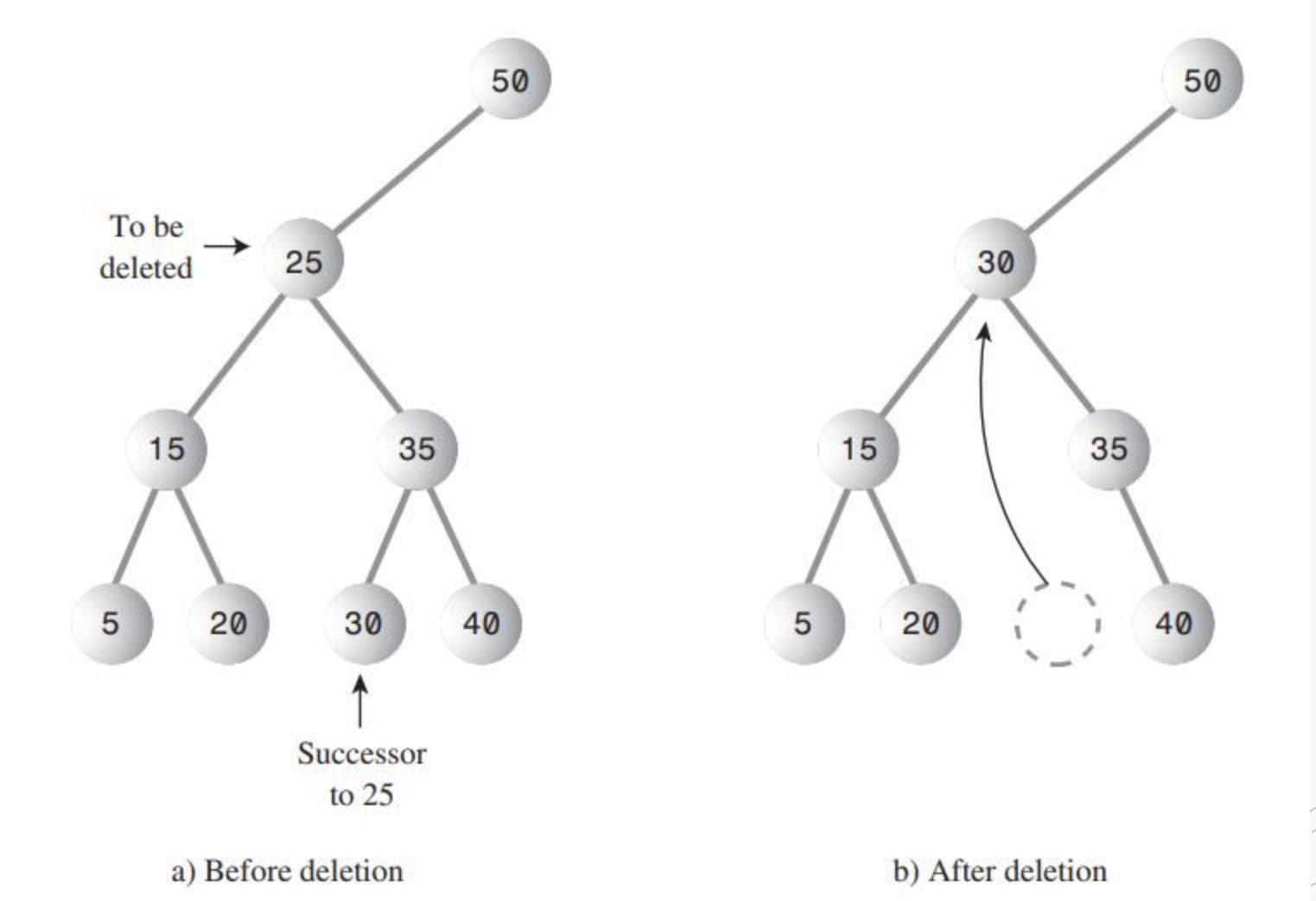
1. The child of the node to be deleted is a left child
    1. The node to be deleted is the left child of its parent
    2. The node to be deleted is the right child of its parent
2. The child of the node to be deleted is a right child
    1. The node to be deleted is the left child of its parent
    2. The node to be deleted is the right child of its parent
3. A specialized situation: the node to be deleted may be the root

```
// delete() continued...
// if no right child, replace with left subtree
else if(current.rightChild==null)
   if(current == root)
      root = current.leftChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.leftChild;
   else                           // right child of parent
      parent.rightChild = current.leftChild;

// if no left child, replace with right subtree
else if(current.leftChild==null)
   if(current == root)
      root = current.rightChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.rightChild;
   else                           // right child of parent
      parent.rightChild = current.rightChild;
```

Case 3: Node has two children



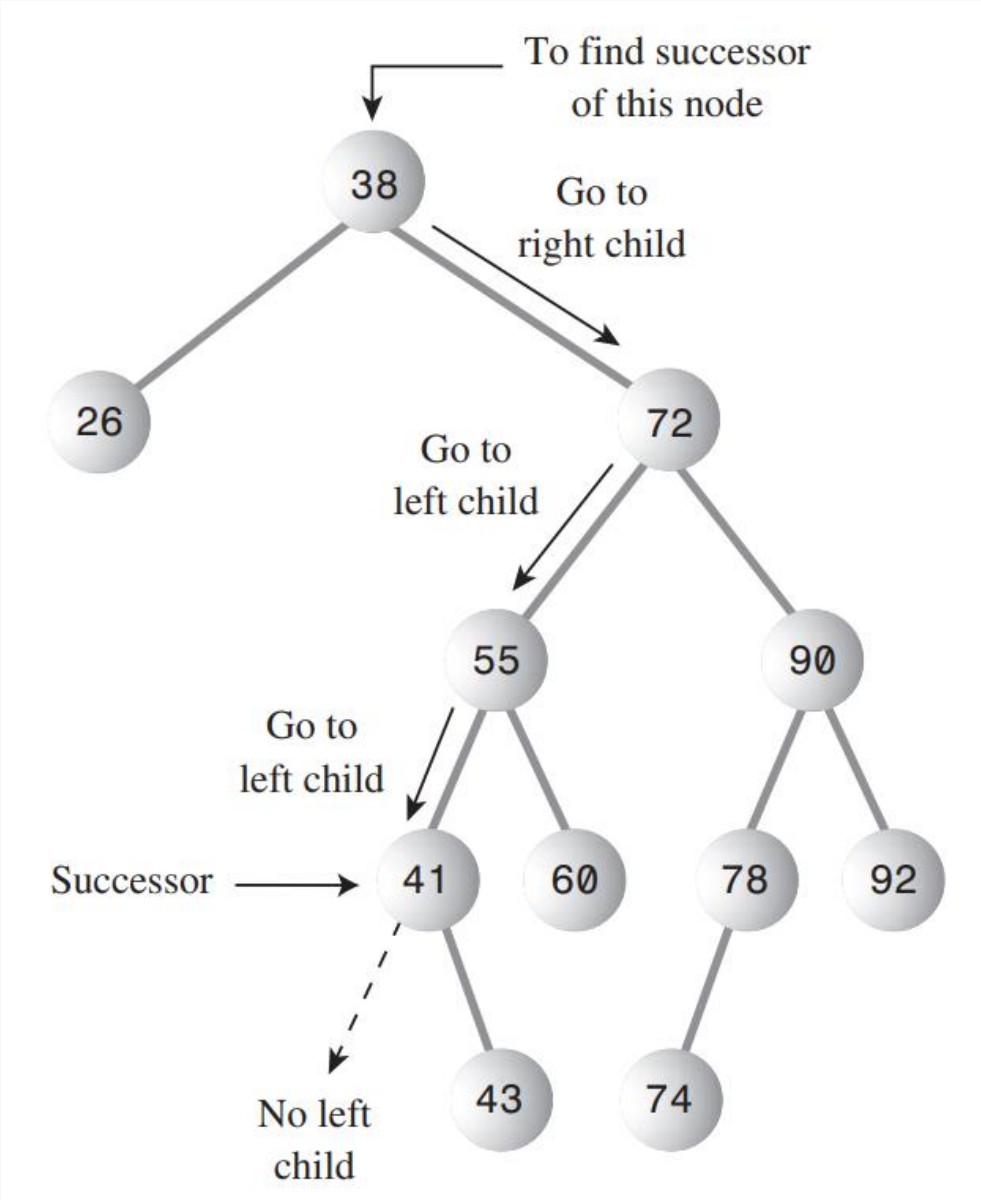
Case 3: Node has two children



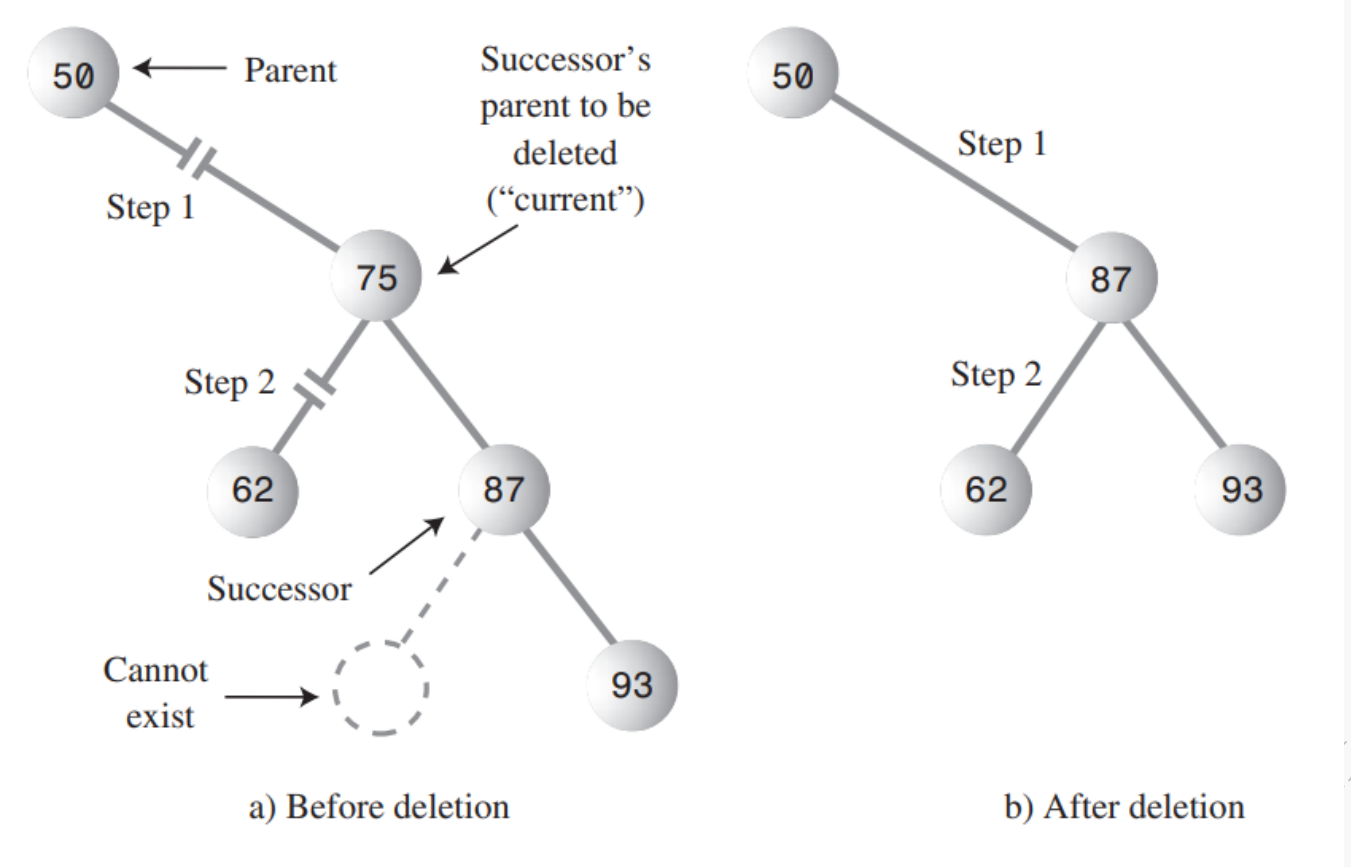
Find the successor

- **Successor is:**
- The smallest of the set of nodes that are larger than the given node

Find successor

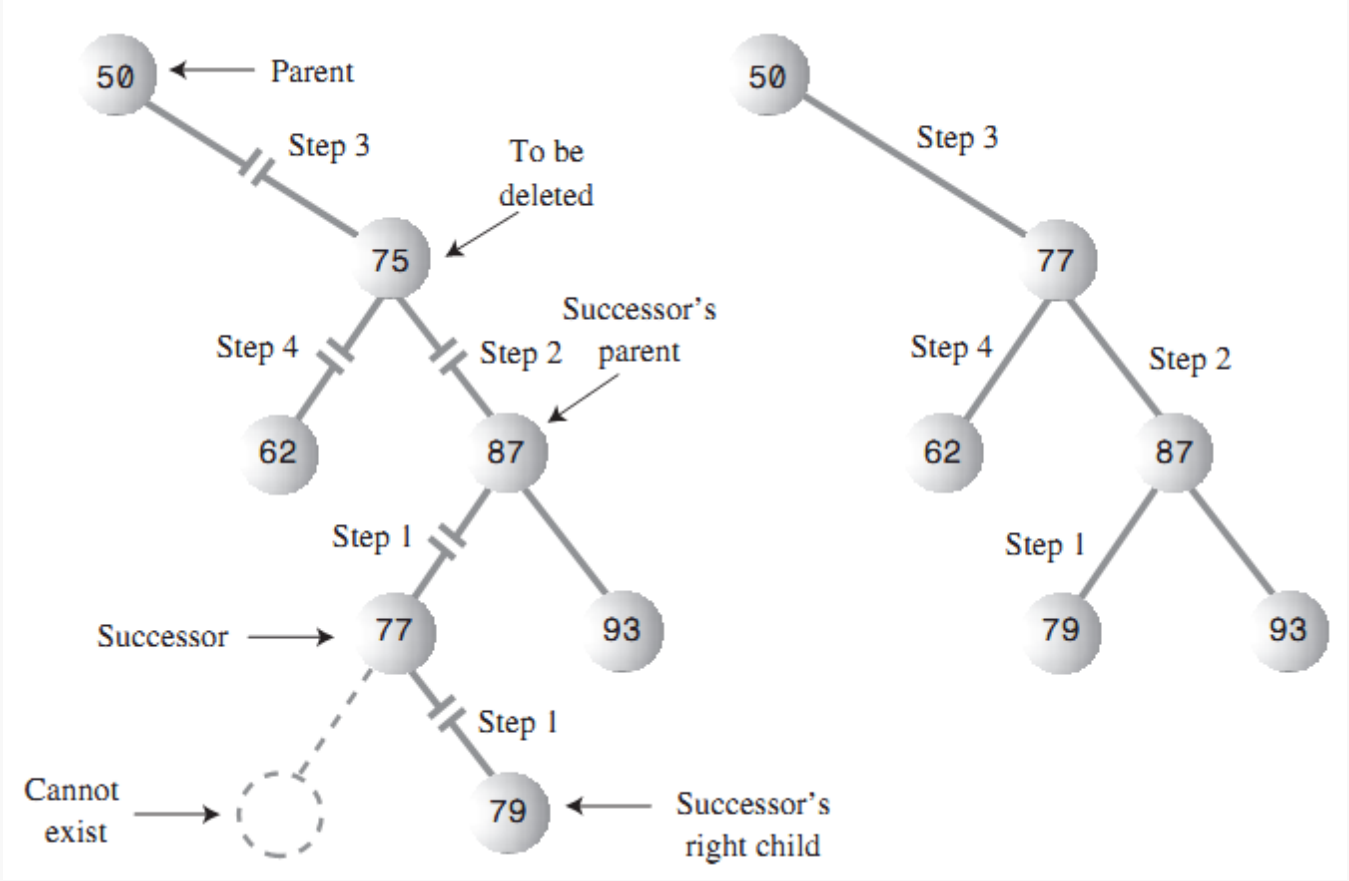


Case 3.1: Successor is the right child of ***delNode***



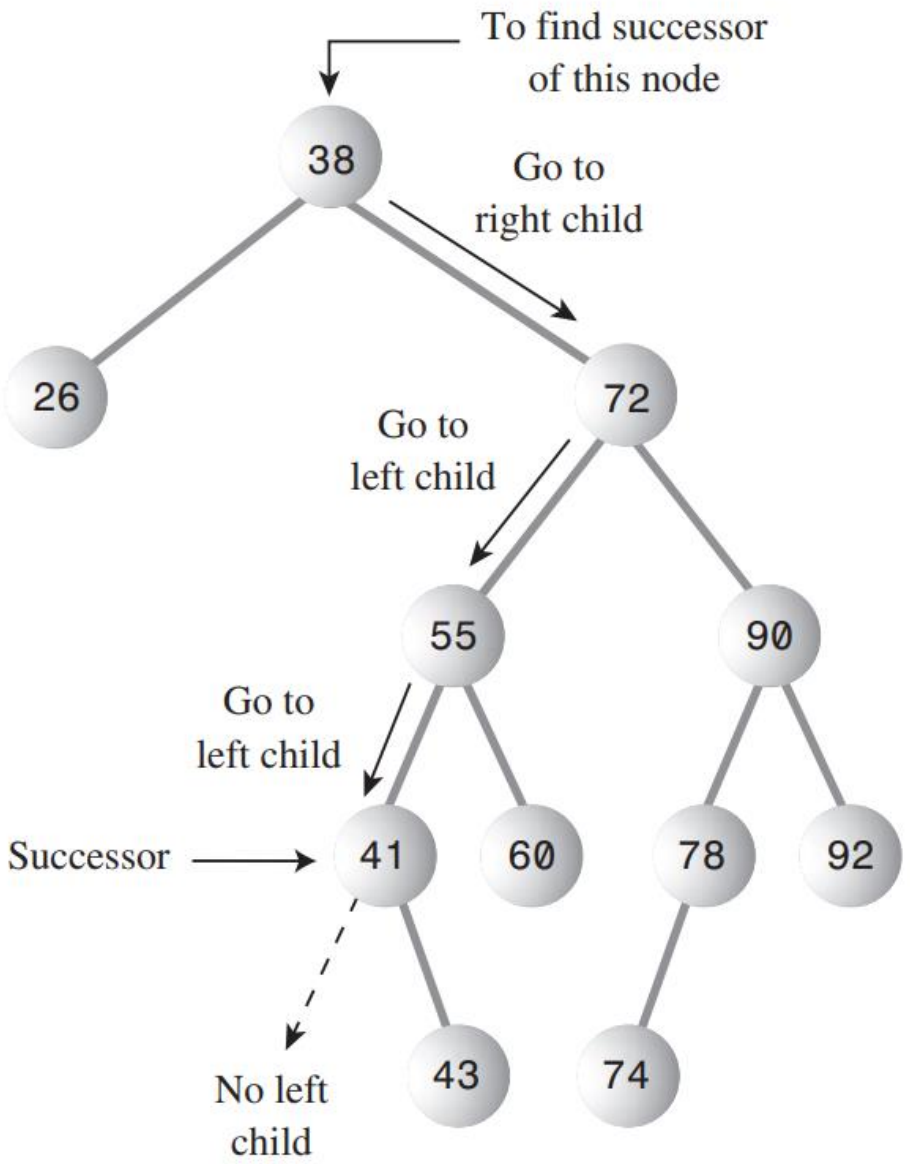

```
1. parent.rightChild = successor;
2. successor.leftChild = current.leftChild;
```

Case 3.2: Successor is left descendant of right child




```
1. successorParent.leftChild = successor.rightChild;
2. successor.rightChild = current.rightChild;
3. parent.rightChild = successor;
4. successor.leftChild = current.leftChild;
```

Find successor

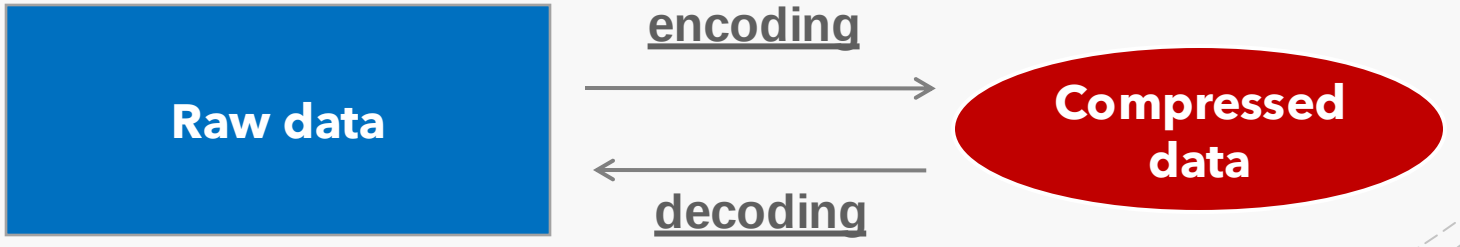



```
// returns node with next-highest value after delNode
// goes to right child, then right child's left descendants

private node getSuccessor(node delNode)
   {
   Node successorParent = delNode;
   Node successor = delNode;
   Node current = delNode.rightChild;   // go to right child
   while(current != null)               // until no more
      {                                 // left children,
      successorParent = successor;
      successor = current;
      current = current.leftChild;      // go to left child
      }
                                        // if successor not
   if(successor != delNode.rightChild)  // right child,
      {                                 // make connections
      successorParent.leftChild = successor.rightChild;
      successor.rightChild = delNode.rightChild;
      }
   return successor;
   }
```

Huffman Encoding

Data Compression

- "In computer science and information theory, **data compression** or **source coding** is the process of encoding information using fewer bits (or other information-bearing units) than an unencoded representation would use through use of specific encoding schemes." (Wikipedia)
- Compression reduces the consumption of storage (disks) or bandwidth.
- However, it needs processing time to restore or view the compressed code.



Data Compression

- Types of compression
  - *Lossy*: MP3, JPG
  - *Lossless*: ZIP, GZ
- Compression Algorithm:
  - Huffman Encoding
  - LZW (Lempel-Ziv-Welch) Compression technique
  - RLE: Run Length Encoding
- Performance of compression depends on file types.

Uniquely Decodable Codes

- A variable length code assigns a bit string (codeword) of variable length to every message value
  - e.g., `a = 1, b = 01, c = 101, d = 011`
- What if you get the sequence of bits `1011` ?
  - Is it `aba, ca,` or `ad`?
- A **uniquely decodable code** is a variable length code in which bit strings can always be uniquely decomposed into its codewords.

Text Compression

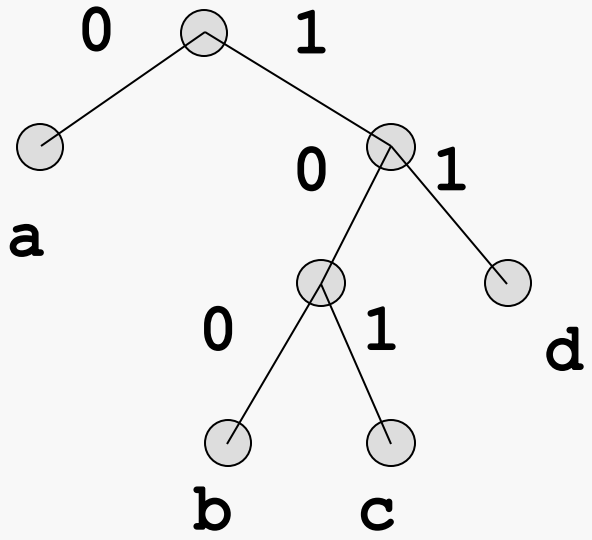
- Problem: Efficiently encode a given string X by a smaller string Y
- Applications:
  - Save memory and/or bandwidth
- Huffman's algorithm
  - computes frequency f(c) for each character c
  - encodes high-frequency characters with short code
  - no code word is a prefix of another code word
  - uses optimal encoding tree to determine the code words

Text Compression

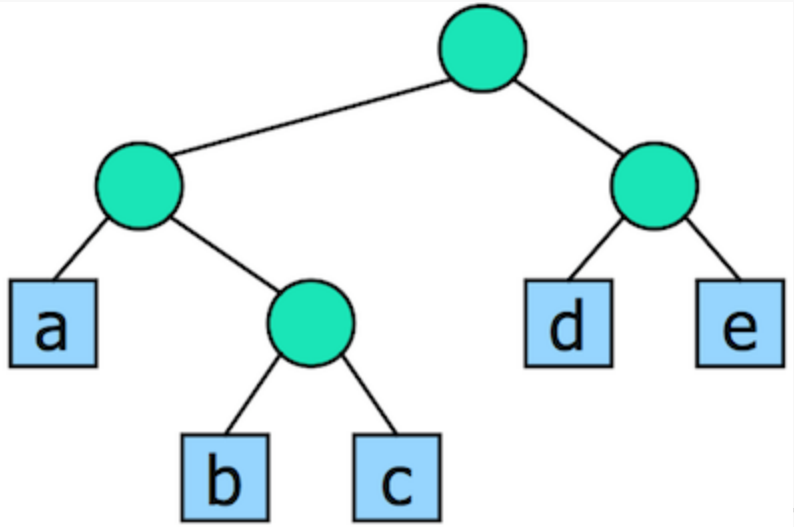
- **Code** … mapping of each character to a binary code word
- **Prefix code** … binary code such that no code word is prefix of another code word
- **Encoding** tree …
  - represents a prefix code
  - each leaf stores a character
  - code word given by the path from the root to the leaf (0 for left child, 1 for right child)

Prefix Codes

- A prefix code is a variable length code in which no codeword is a prefix of another codeword
  - e.g., `a = 0, b = 100, c = 101, d = 11`
- Can be viewed as a binary tree with message values at the leaves and 0 or 1s on the edges.



00	010	011	10	11
a	b	c	d	e

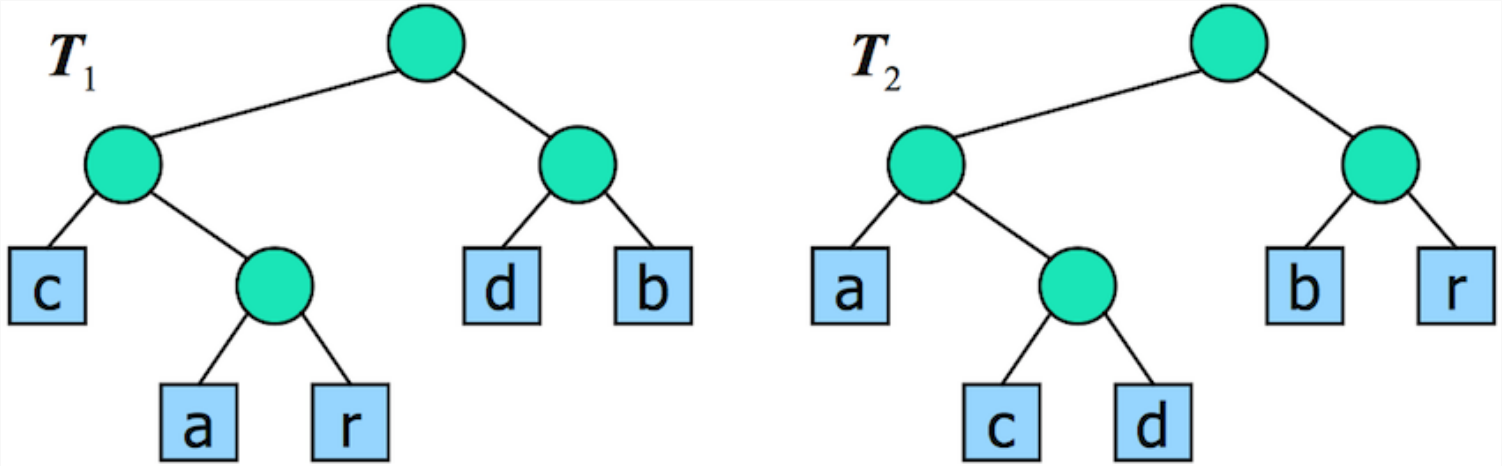


Text Compression

- **Text compression problem**
- Given a text T, find a prefix code that yields the shortest encoding of T
  - short codewords for frequent characters
  - long code words for rare characters

Text Compression

- Example: T = **abracadabra**

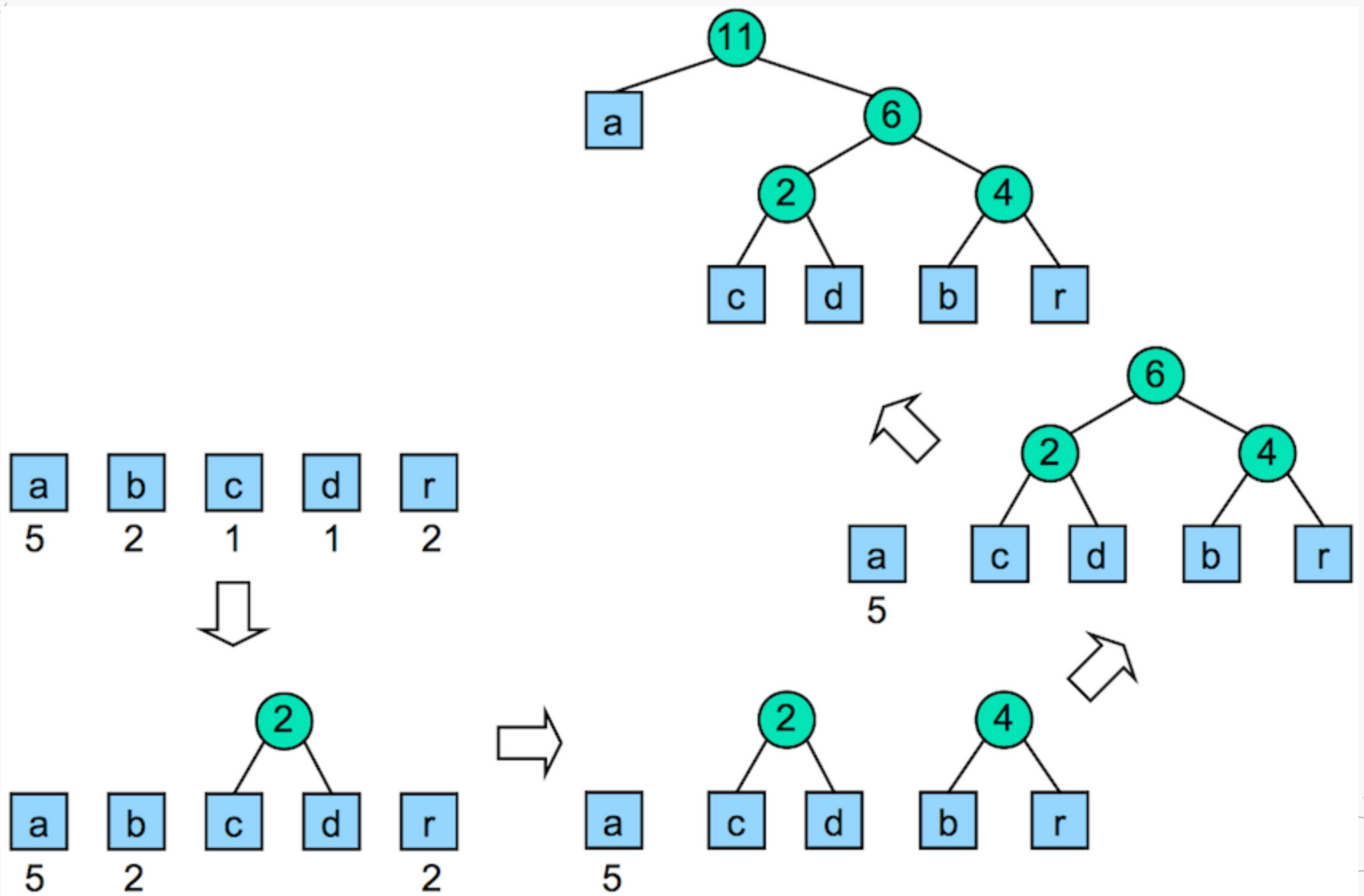


- *T1* requires 29 bits to encode text *T*,
- *T2* requires 24 bits

Text Compression

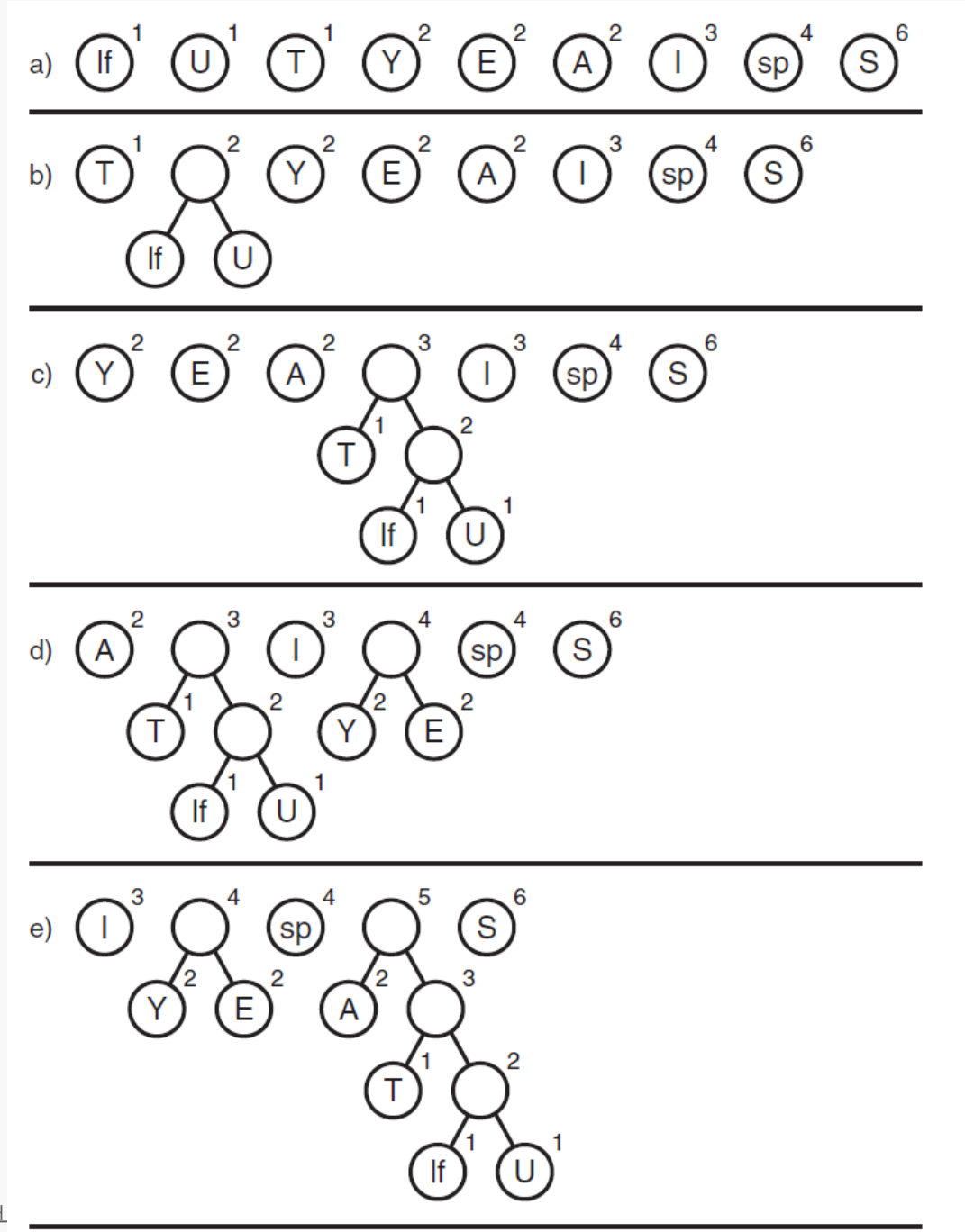
- Huffman's algorithm
  - computes frequency f(c) for each character
  - successively combines pairs of lowest-frequency characters to build encoding tree "bottom-up"
- Example: **abracadabra**

abracadabra

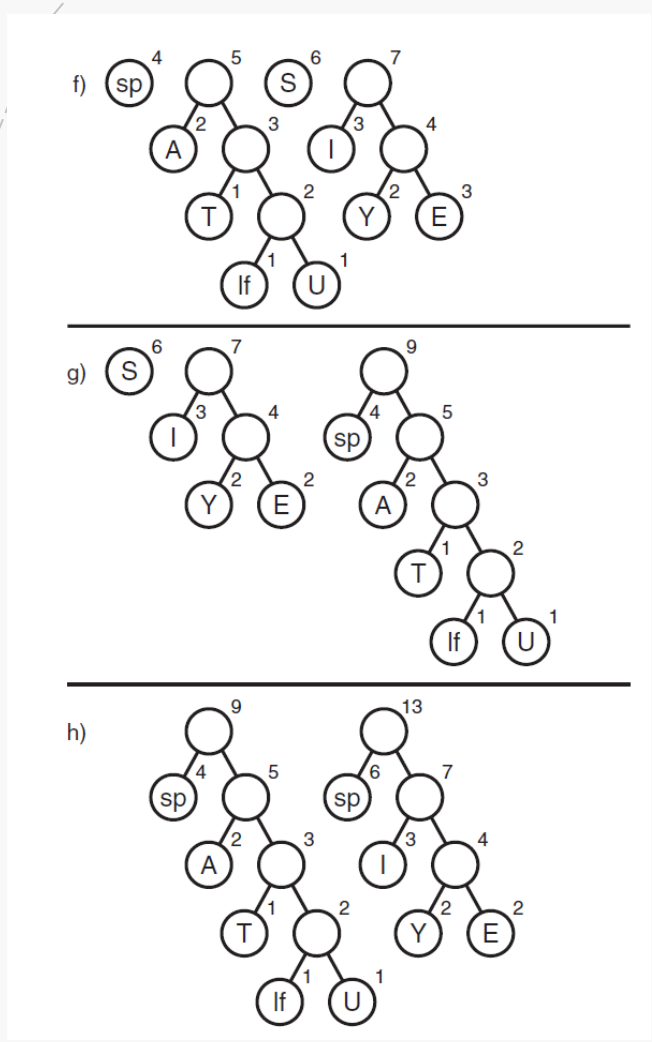


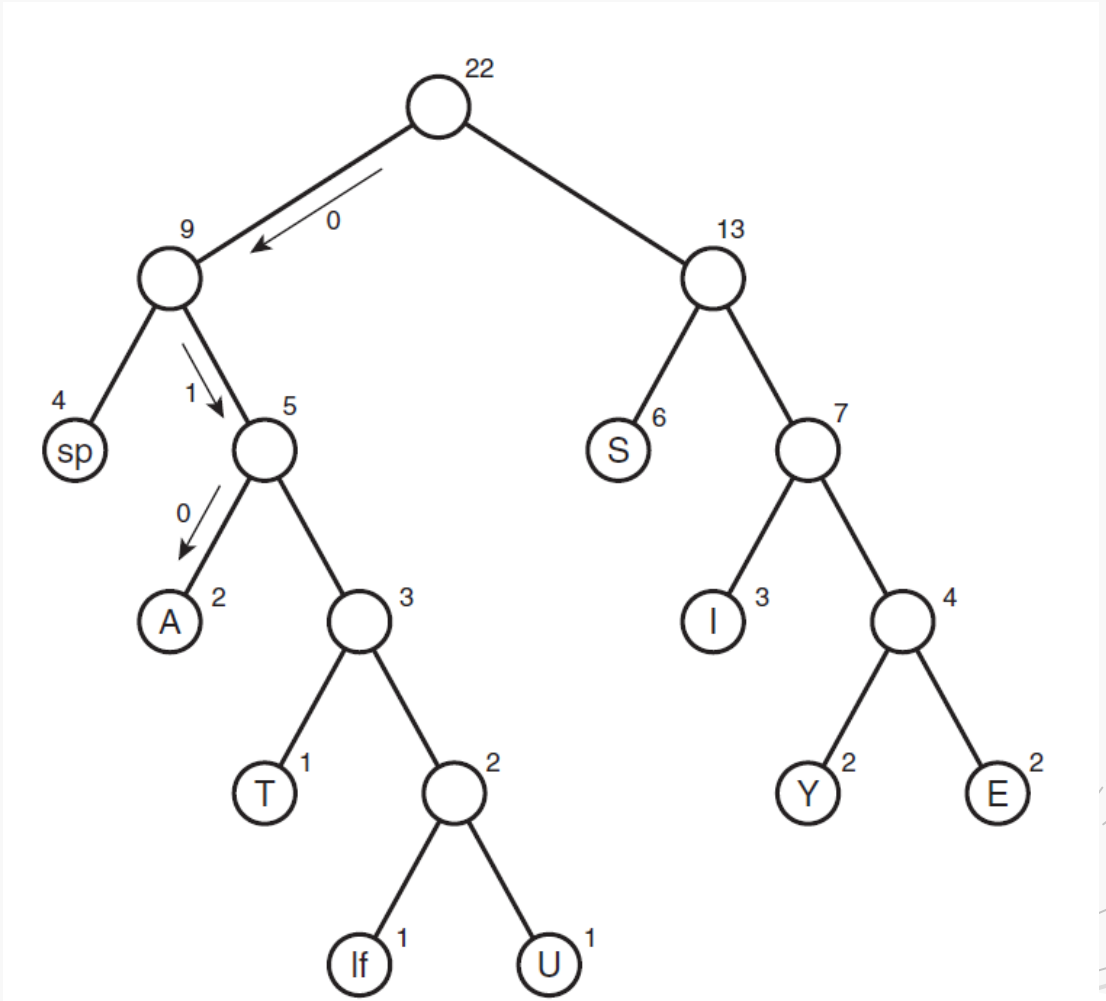
SUSIE SAYS IT IS EASY

Character	Count
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Space	4
Linefeed	1



SUSIE SAYS IT IS EASY





Red-Black tree, 2-3-4 tree

Home reading (chapter 9 & 10)

Vietnam National University of HCMC
International University
School of Computer Science and Engineering

THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn

https://vichithanh.github.io

HTTPS://VICHITHANH.GITHUB.IO

SCAN ME