# Binary Trees

Tran Thanh Tung

# Content

- Introduction
- General trees & binary trees
- Binary search trees (BST)
- Operations on BST (Conceptual)
  - Find
  - Traverse
  - Insert
  - Delete

# 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(log2n) – 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

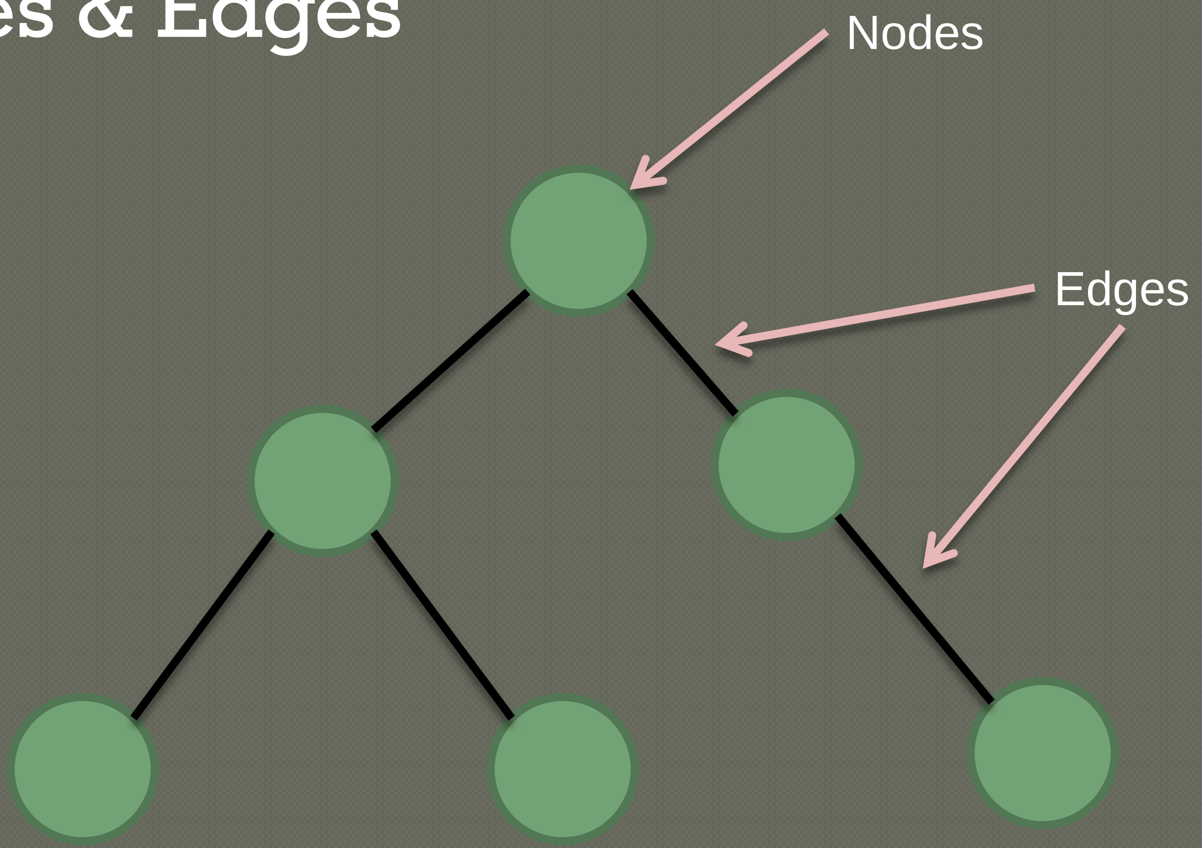
- Quick Searching

→ Interesting data structures

# What is a Tree?

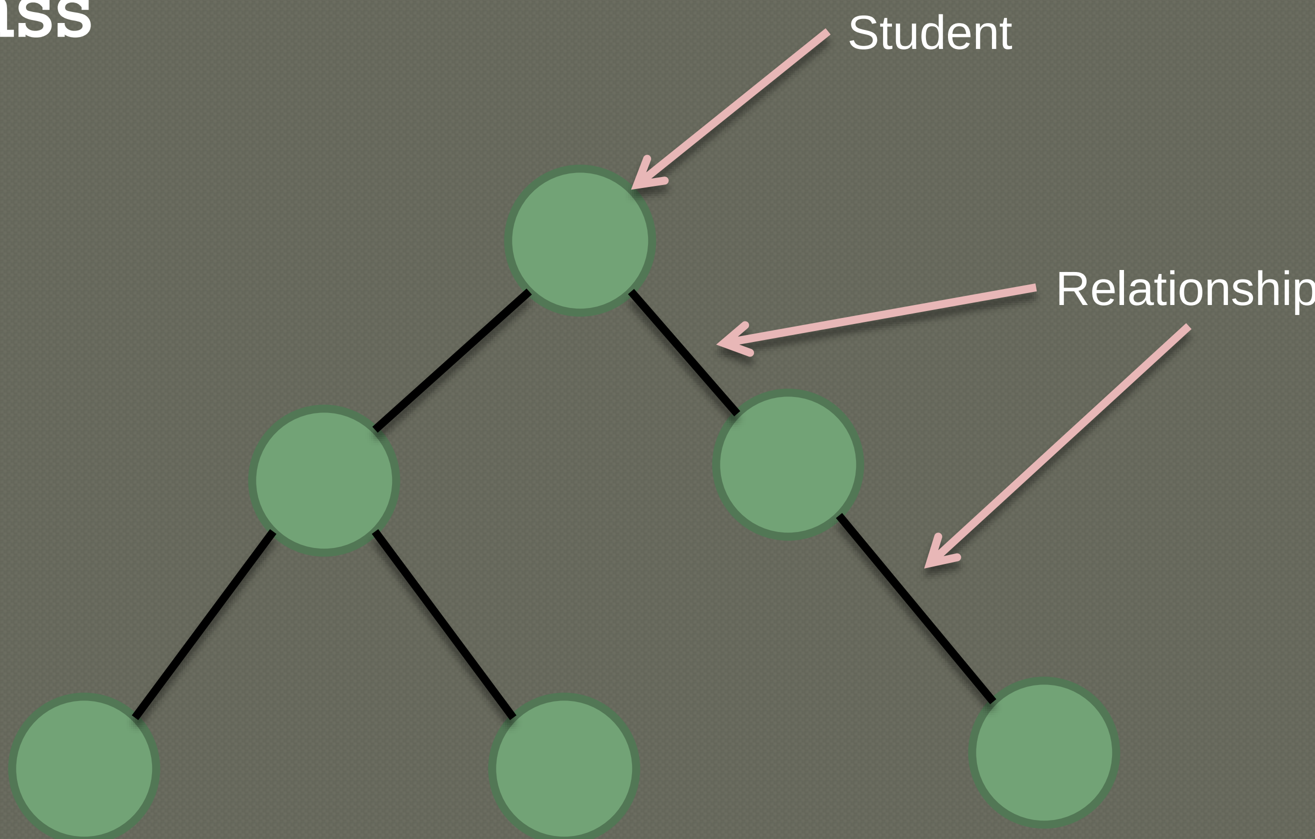
A tree consists of

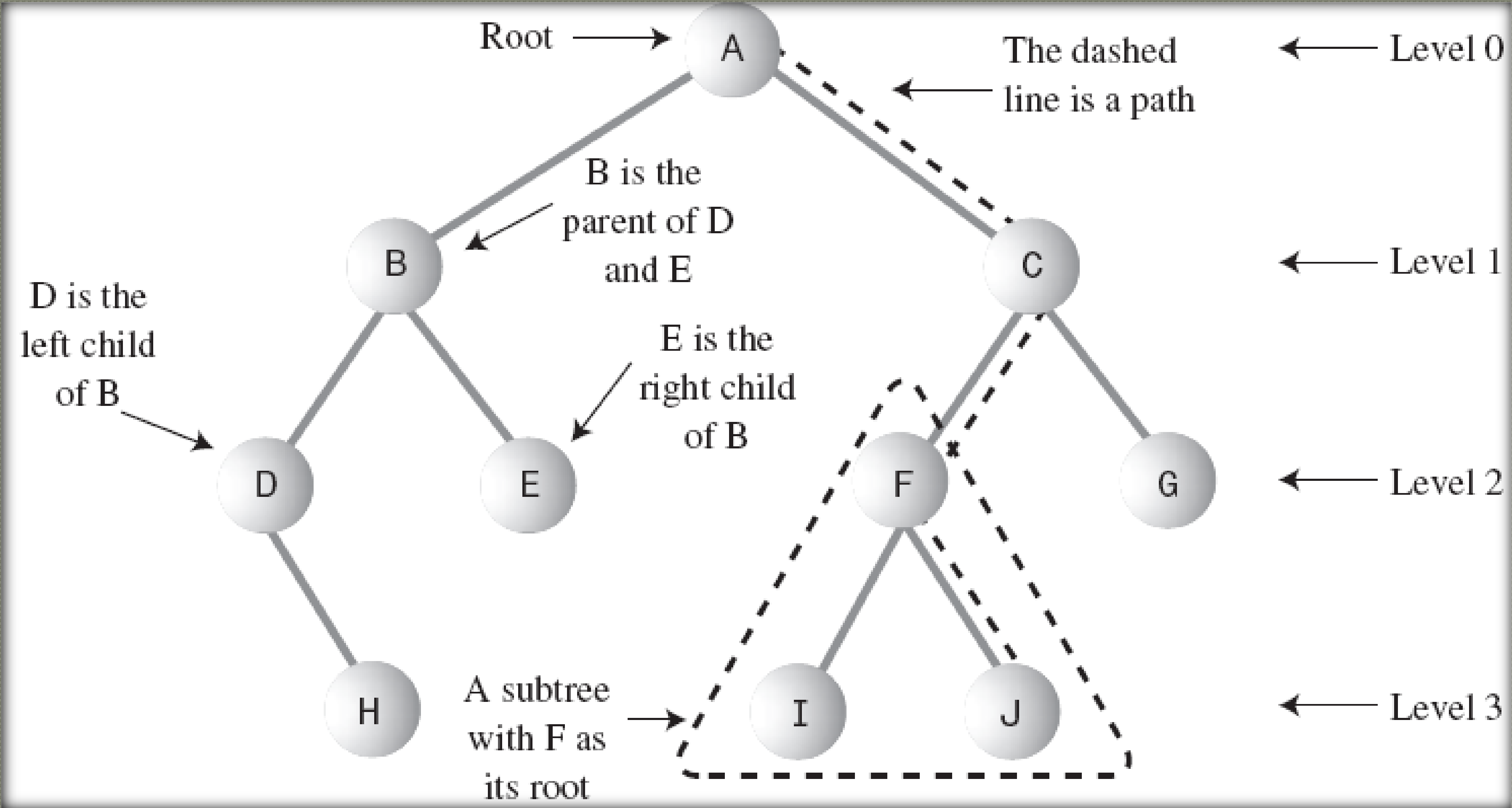
- Nodes & Edges



# Trees as tool

- Describe relationships between students in class






# Trees terminology

Path, Root, Parent, Child, Leaf, Subtree, Visiting, Traversing, Levels, Keys

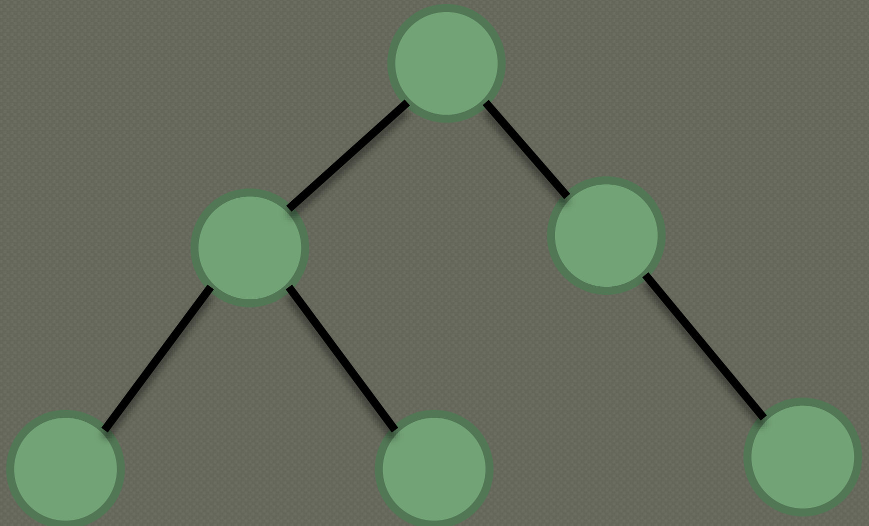
# Properties of Trees

- 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

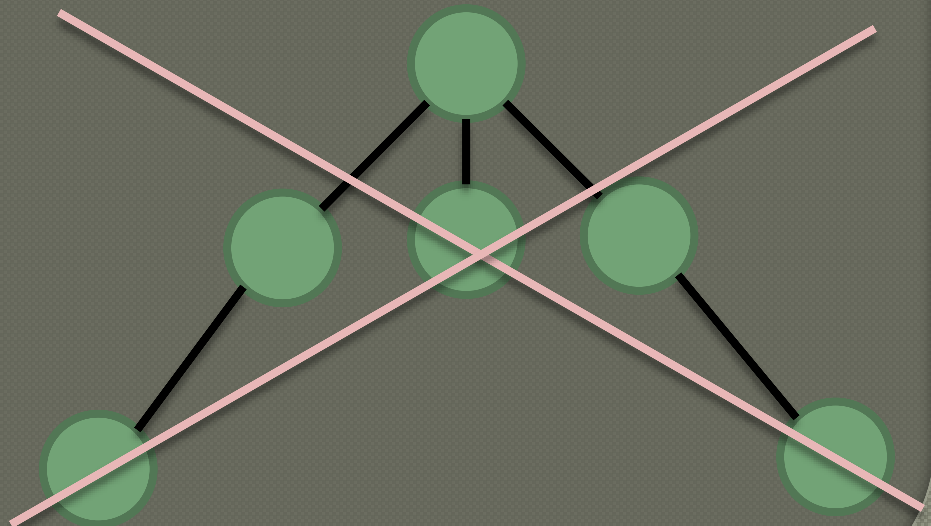
# Binary tree

- Is a tree and
- every node have at most two children

Binary tree



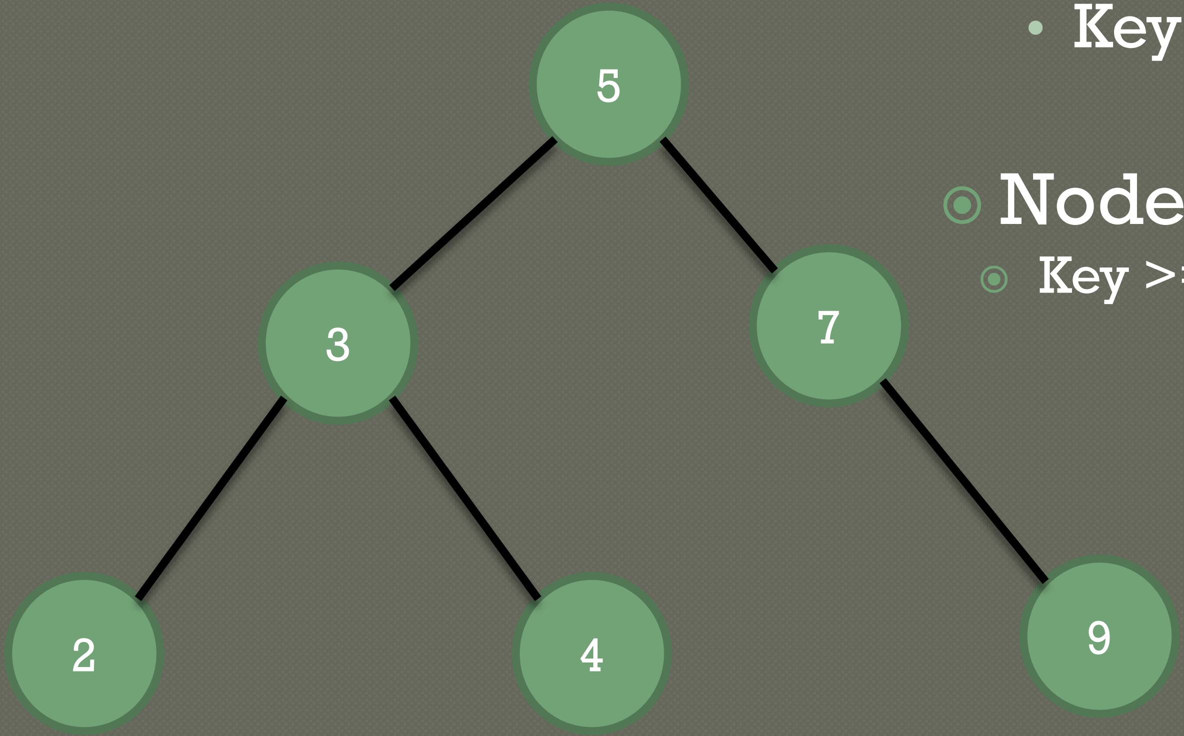
Not a binary tree



# Binary search tree (BST)

Properties

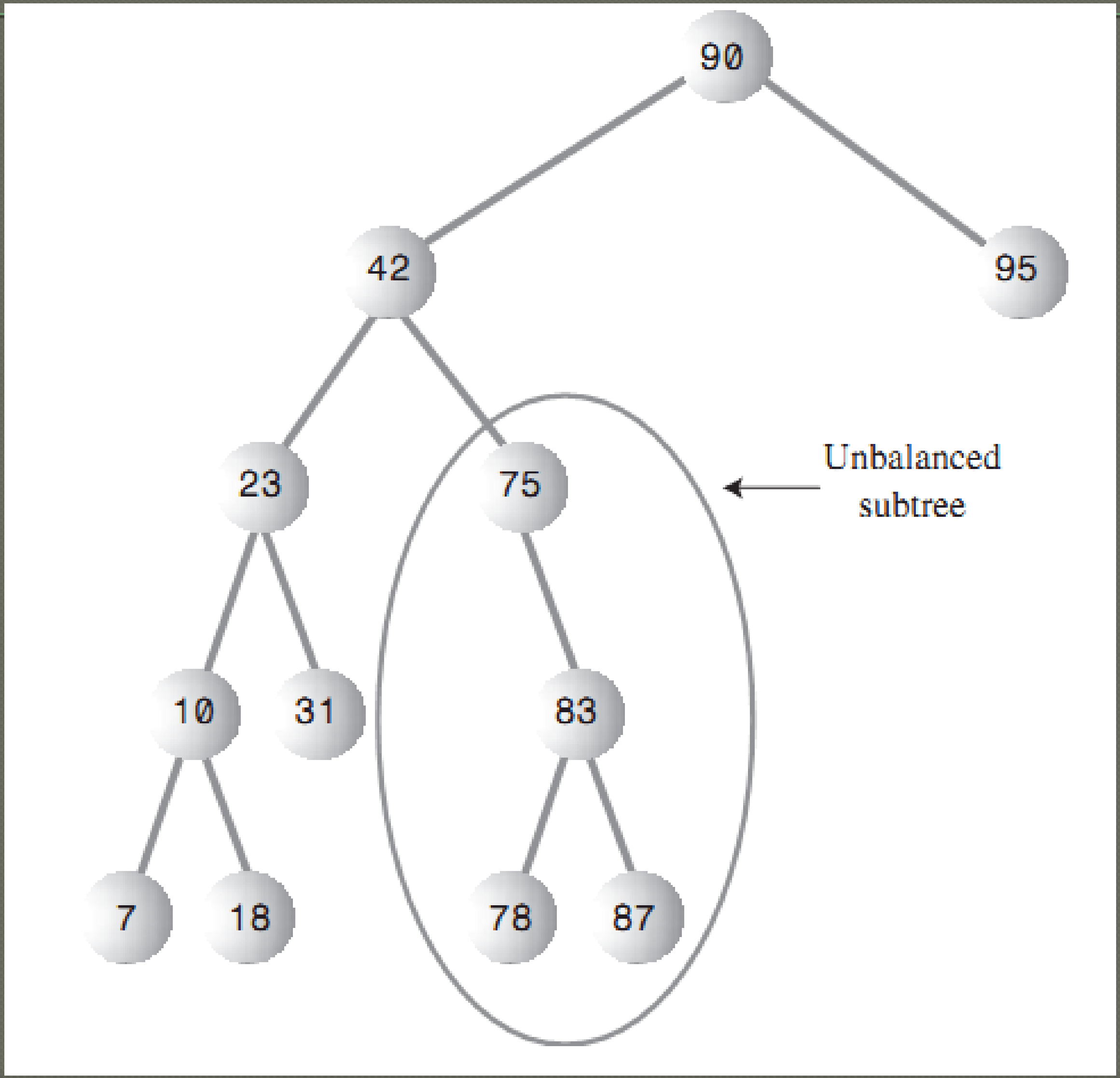
- Node’s left child:
  - Key < Key of Parent
- Node’s right child:
  - Key >= Key of Parent



# How Do BST Work?

- Need to carry out basic tree operations such as
  - finding a node,
  - traversing a tree,
  - adding a node,
  - deleting a node, etc.
- This is what this chapter is all about

# Unbalanced tree

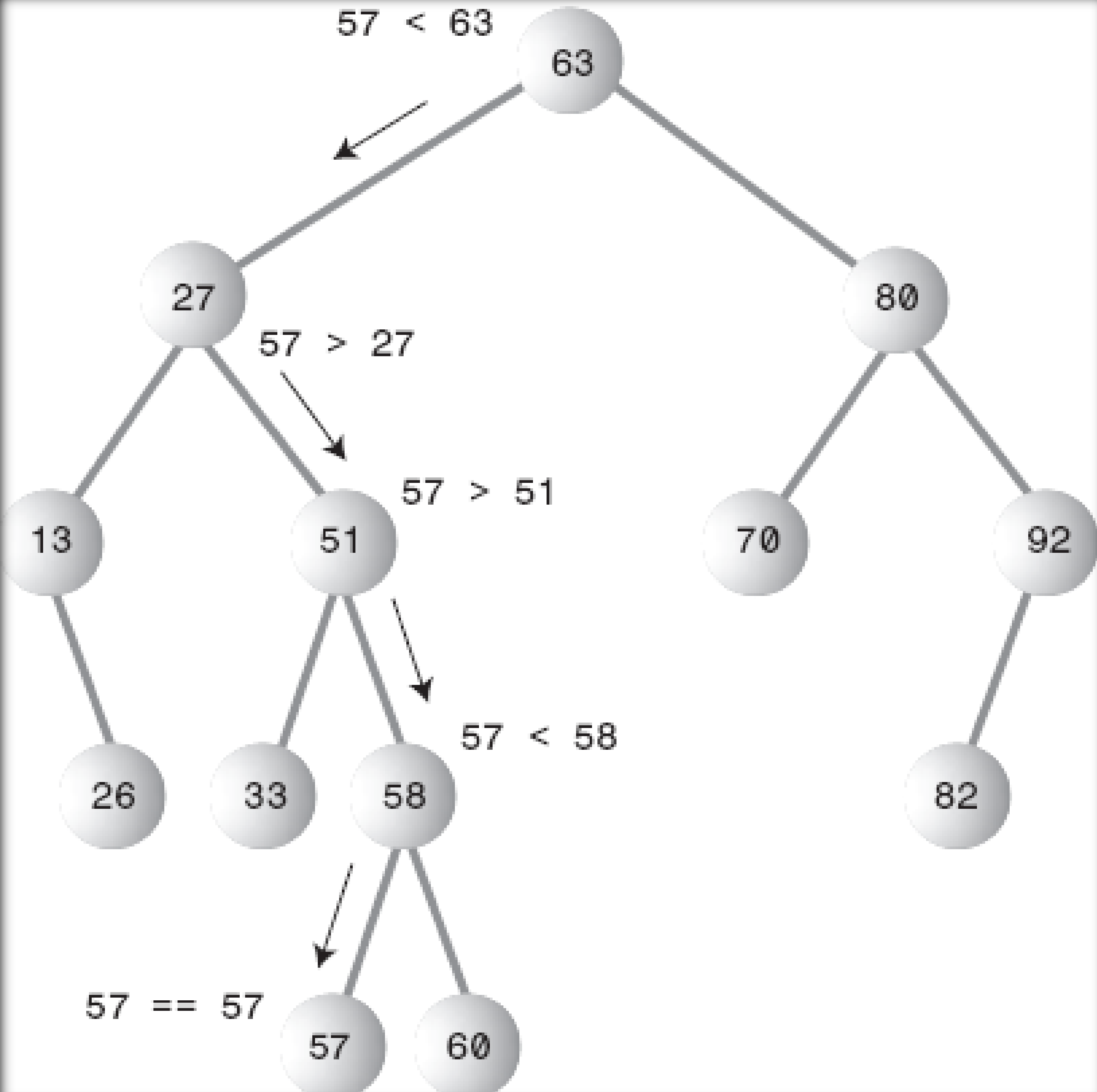


# Unbalanced tree

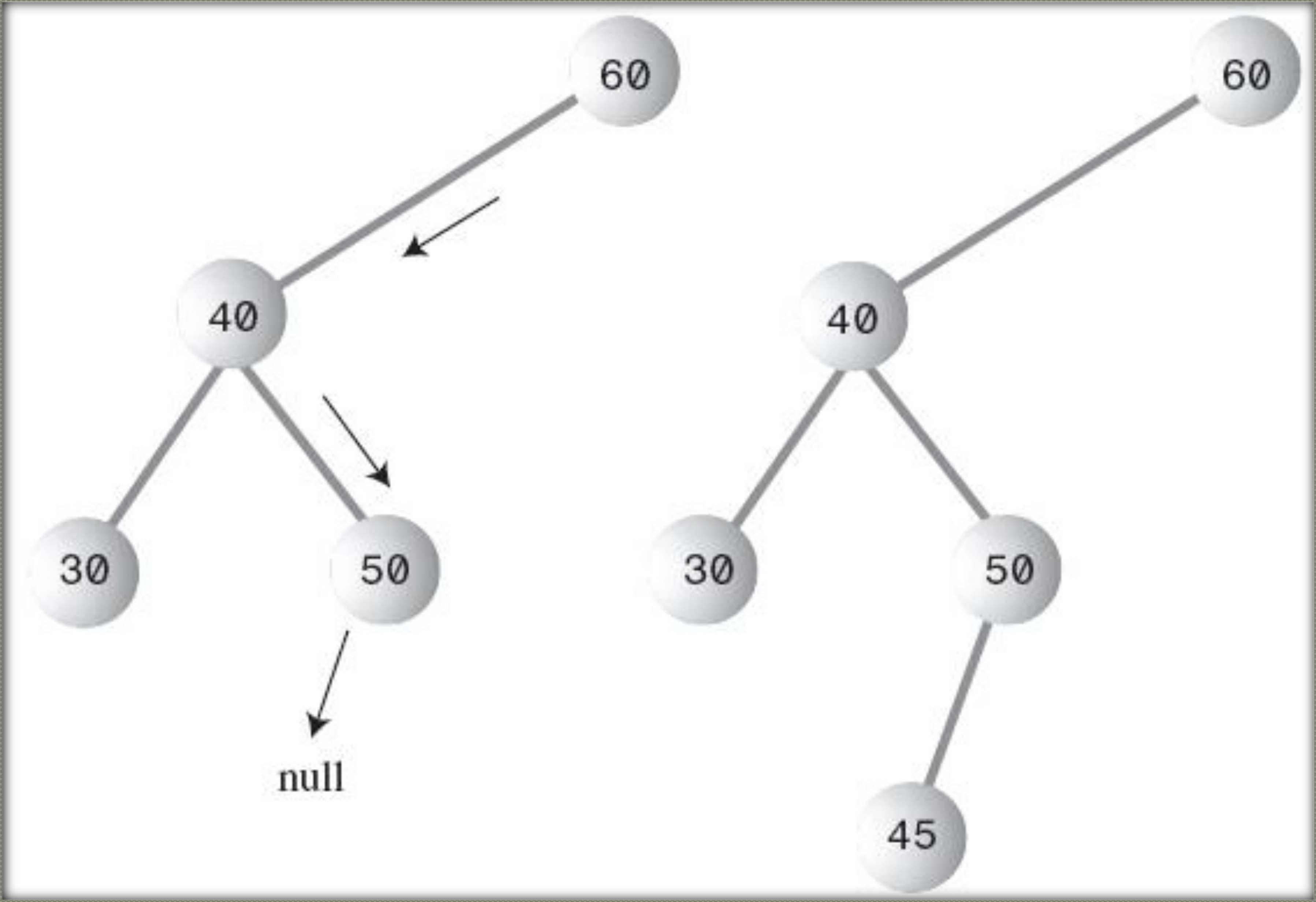
- Is when nodes are mostly in one side
- Is the result of the way they are created
- Prefer balanced tree

# Operations on BST

Conceptual algorithms




## Finding




Insert 45 into the tree

# Insertion

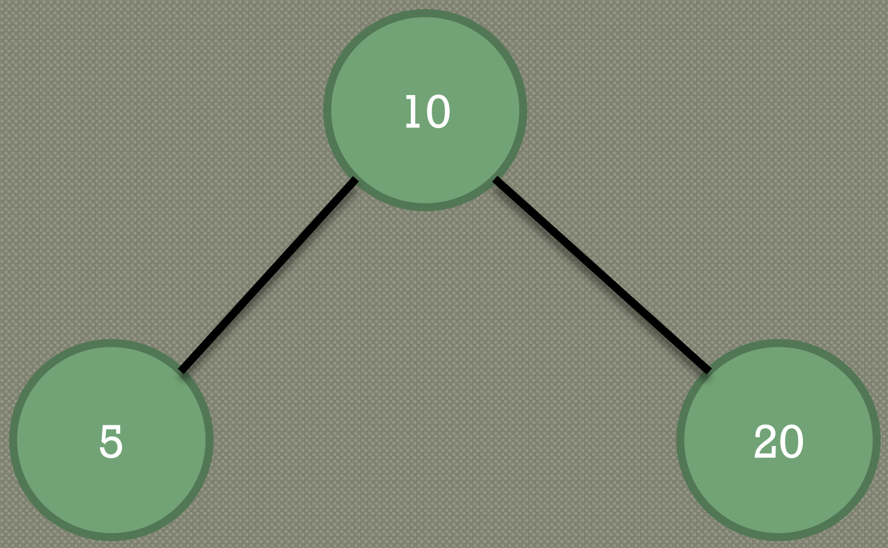
- Unless we run out of memory, we always found the place to insert

- Can handle duplication by modifying the search condition.

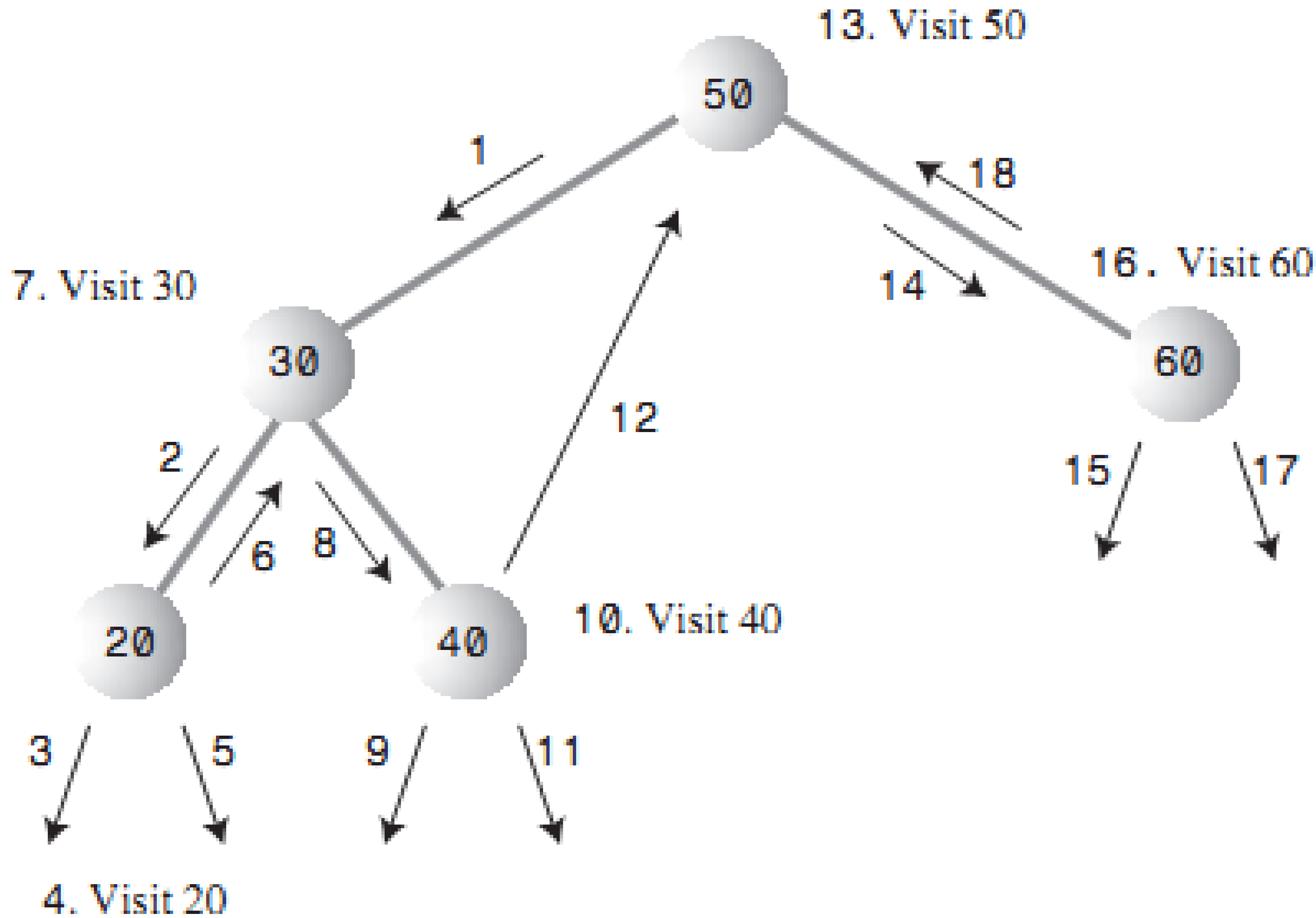
# Traversing

- Visit all nodes of the tree
- 3 ways to traverse a tree
  - Preorder
  - Inorder
  - Postorder

- Inorder traversing
  - Traverse the node's left subtree
  - Visit the node
  - Traverse the node's right subtree

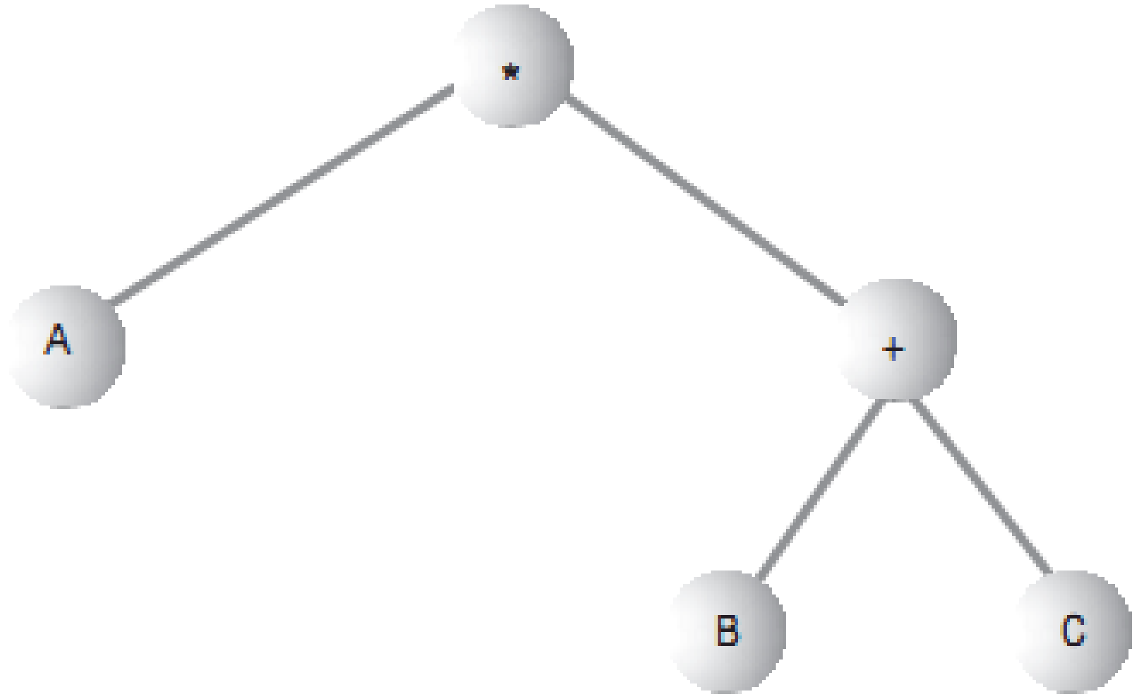



Traverse the tree



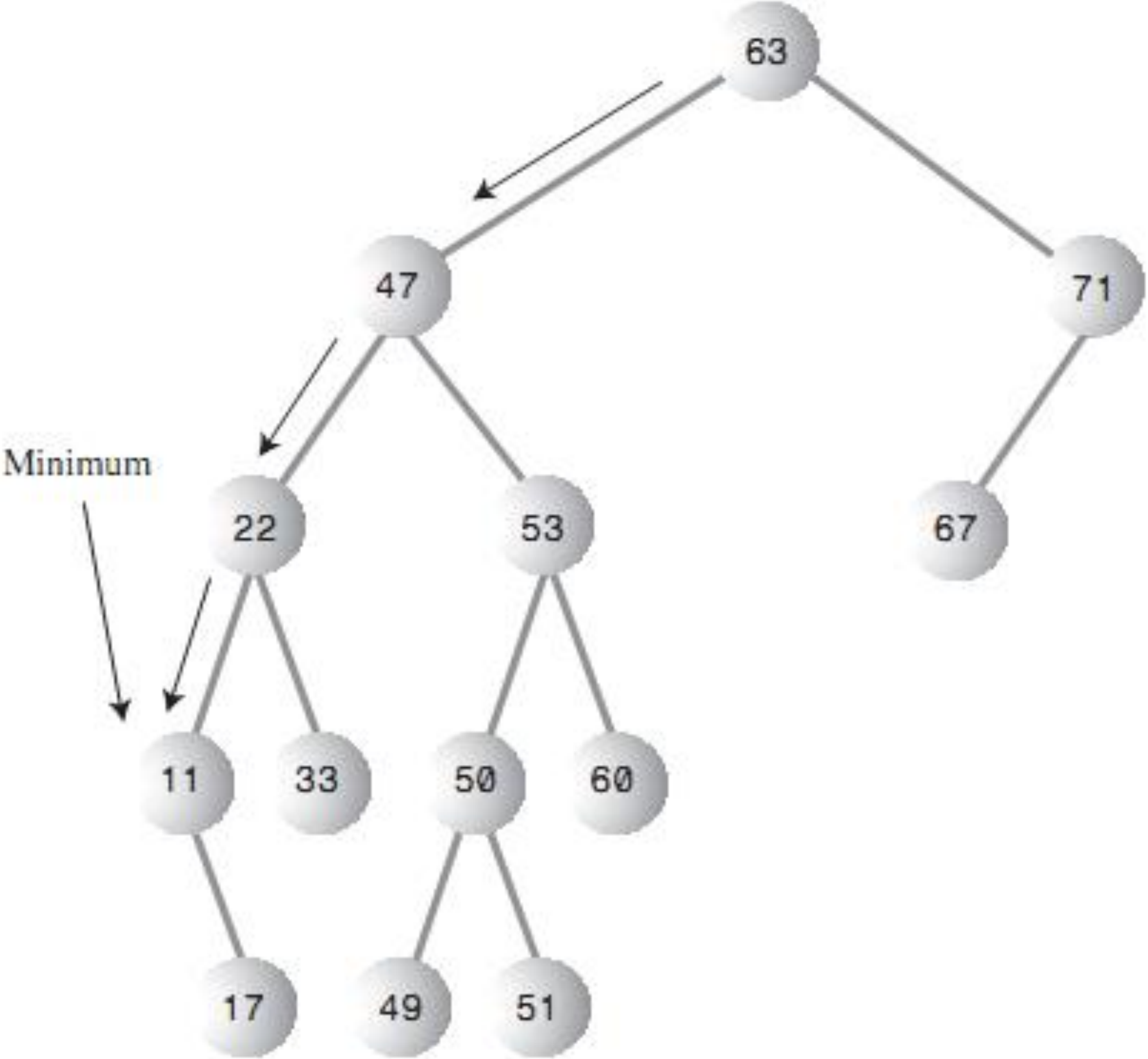

Traverse the tree

# Algebraic expression




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

# Find max & min

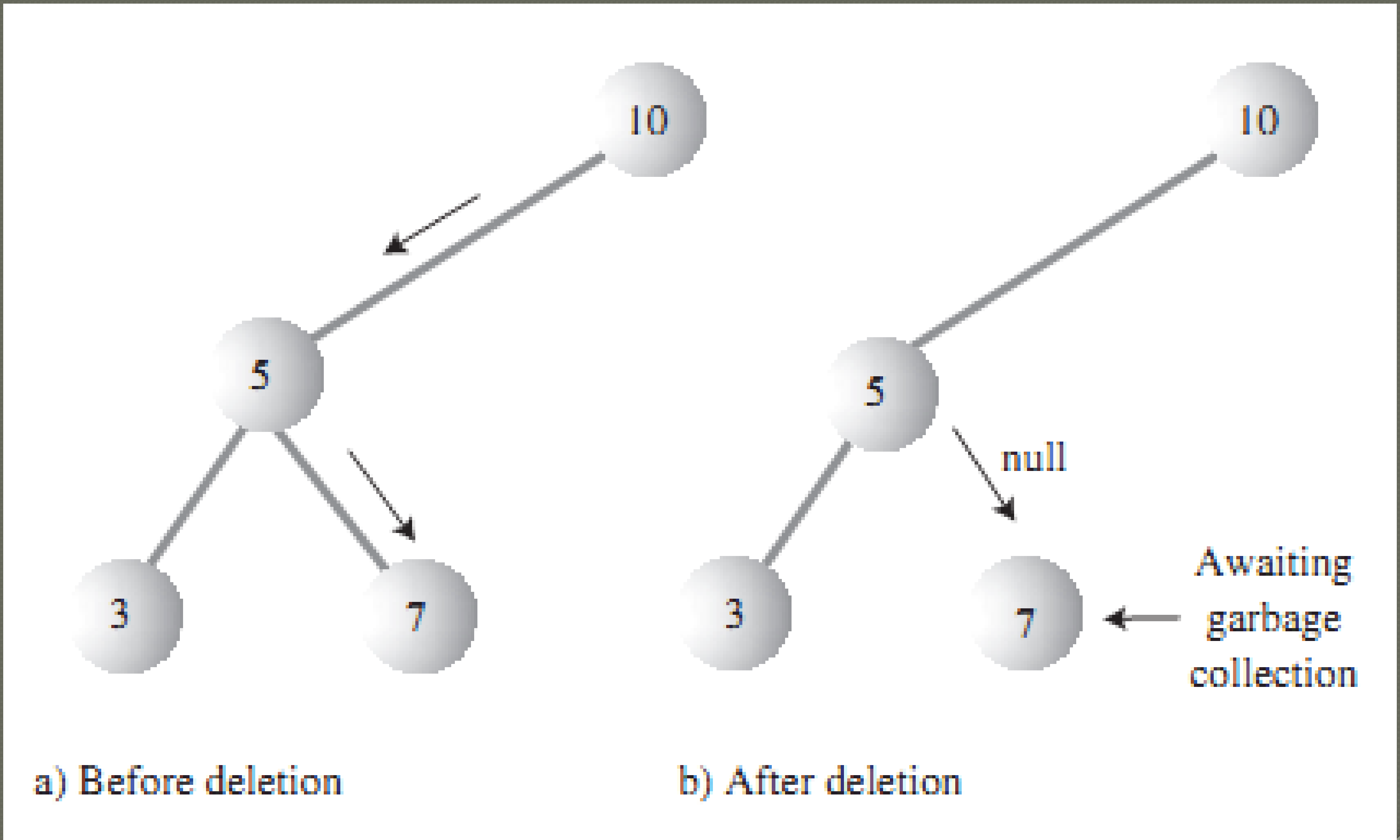


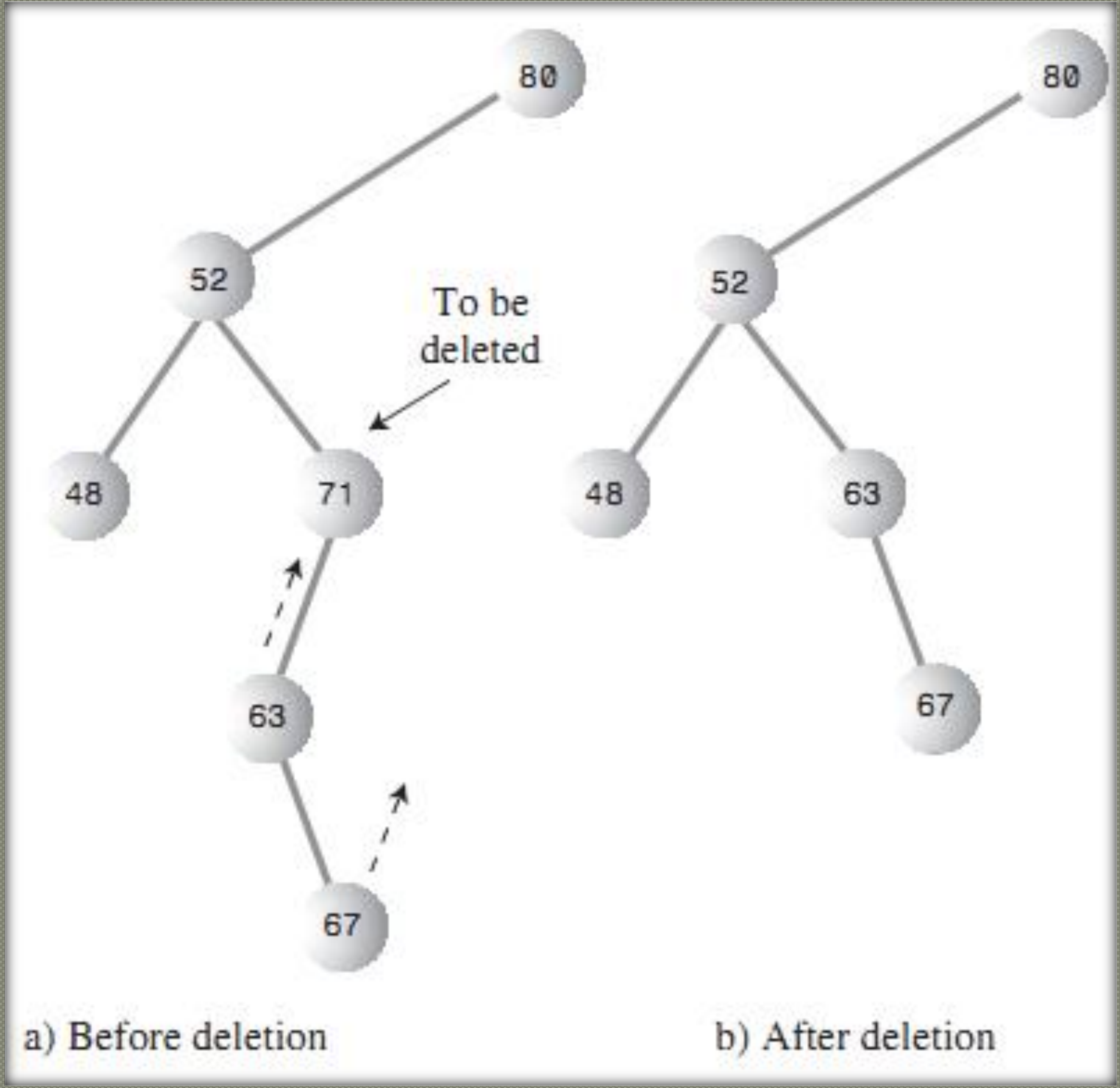
# Deleting a Node

- 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

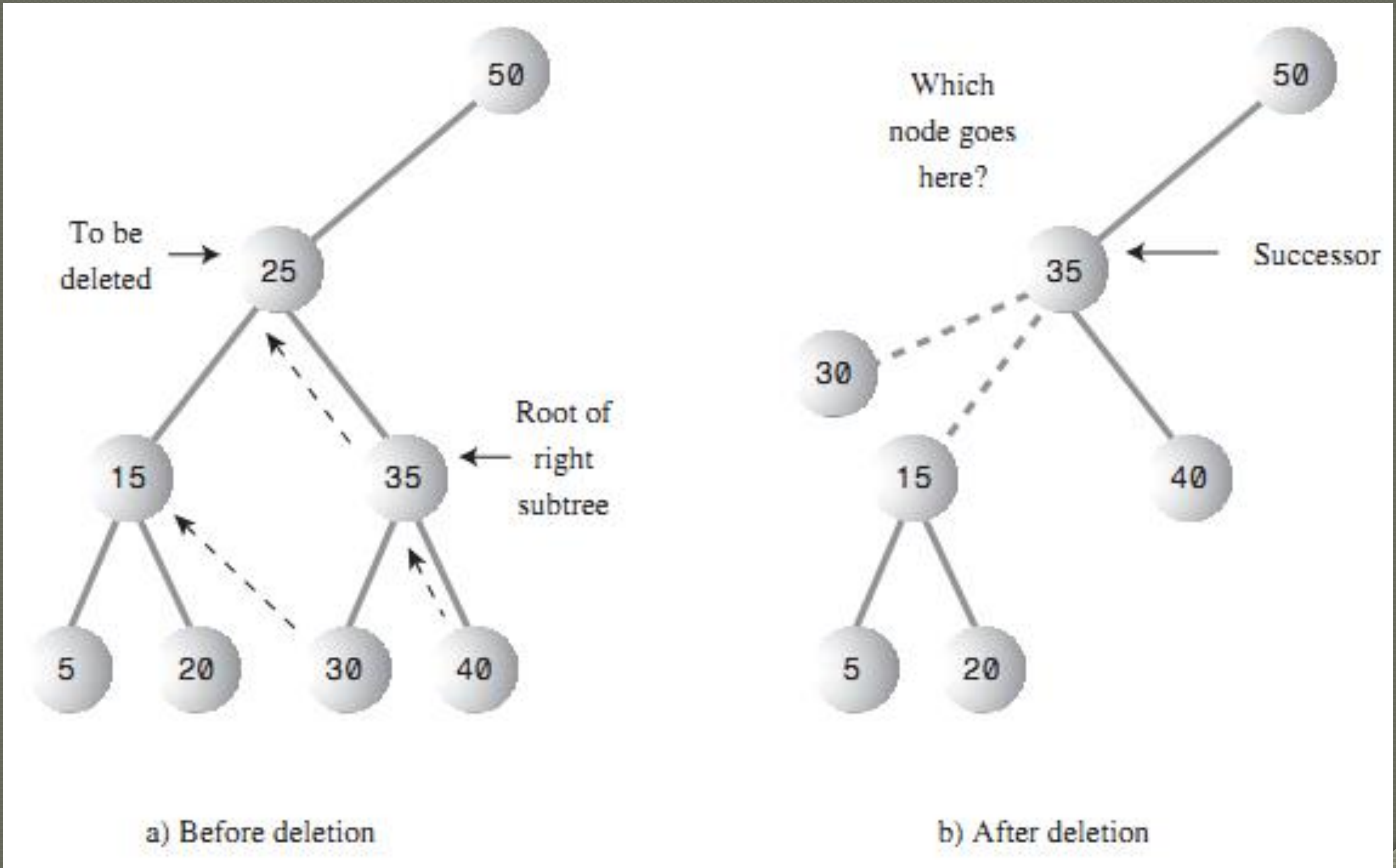
# Case 1: Node has no children



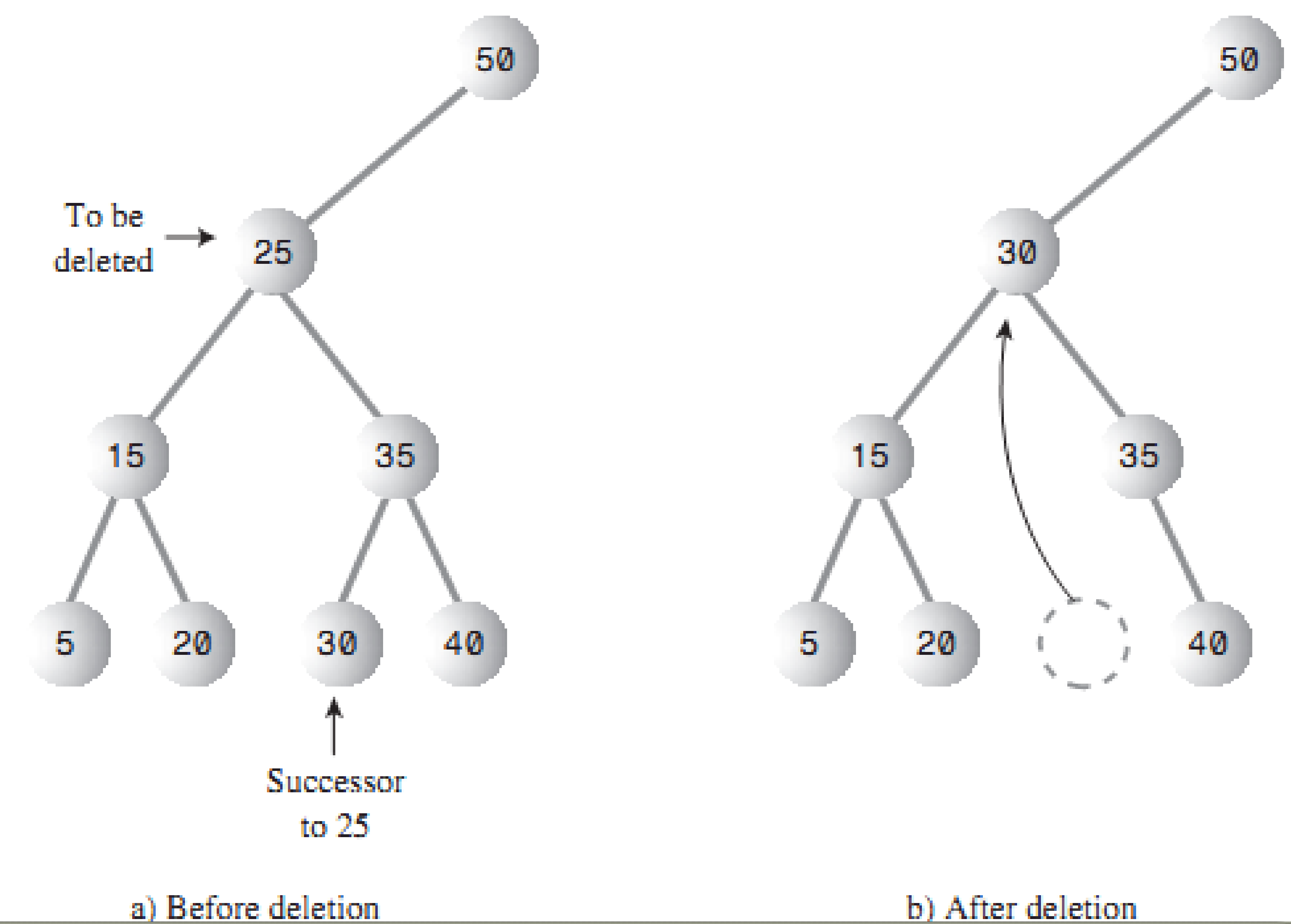



Case2: Node has one child

# Case 3: Node has two children



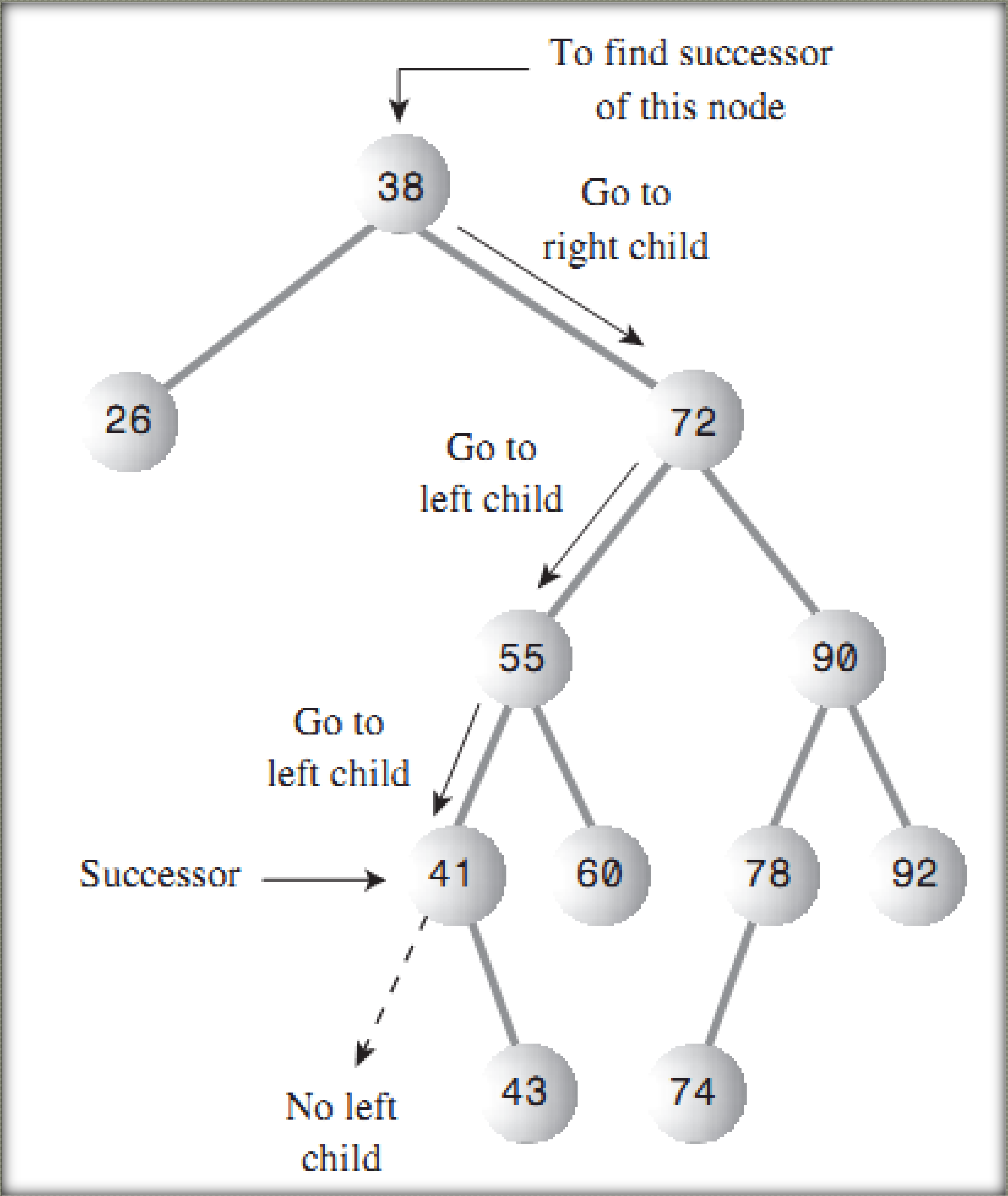
# Case 3: Node has two children



# Find the successor

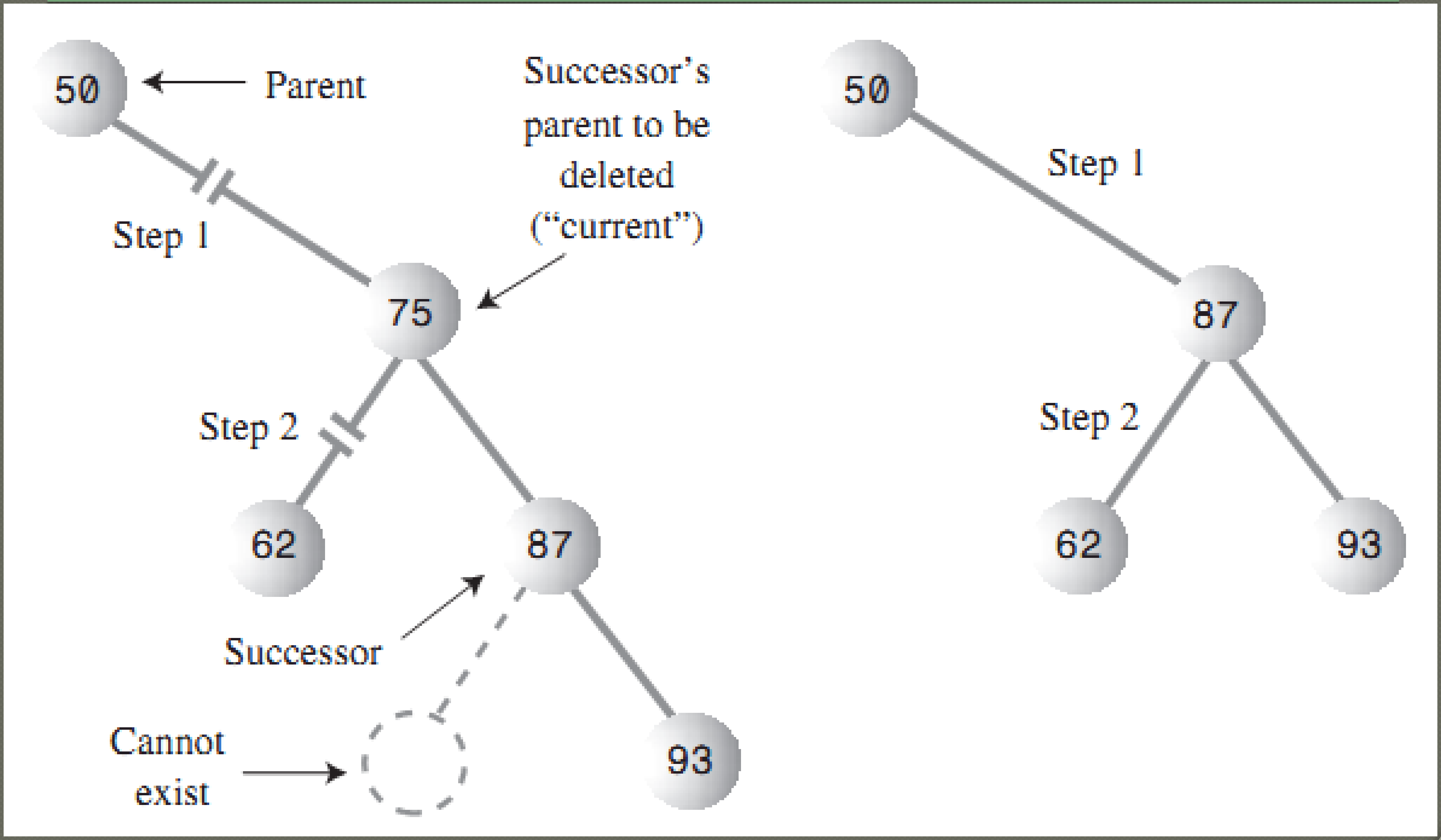
- Successor:

  The smallest of the set of nodes that are larger than the given node

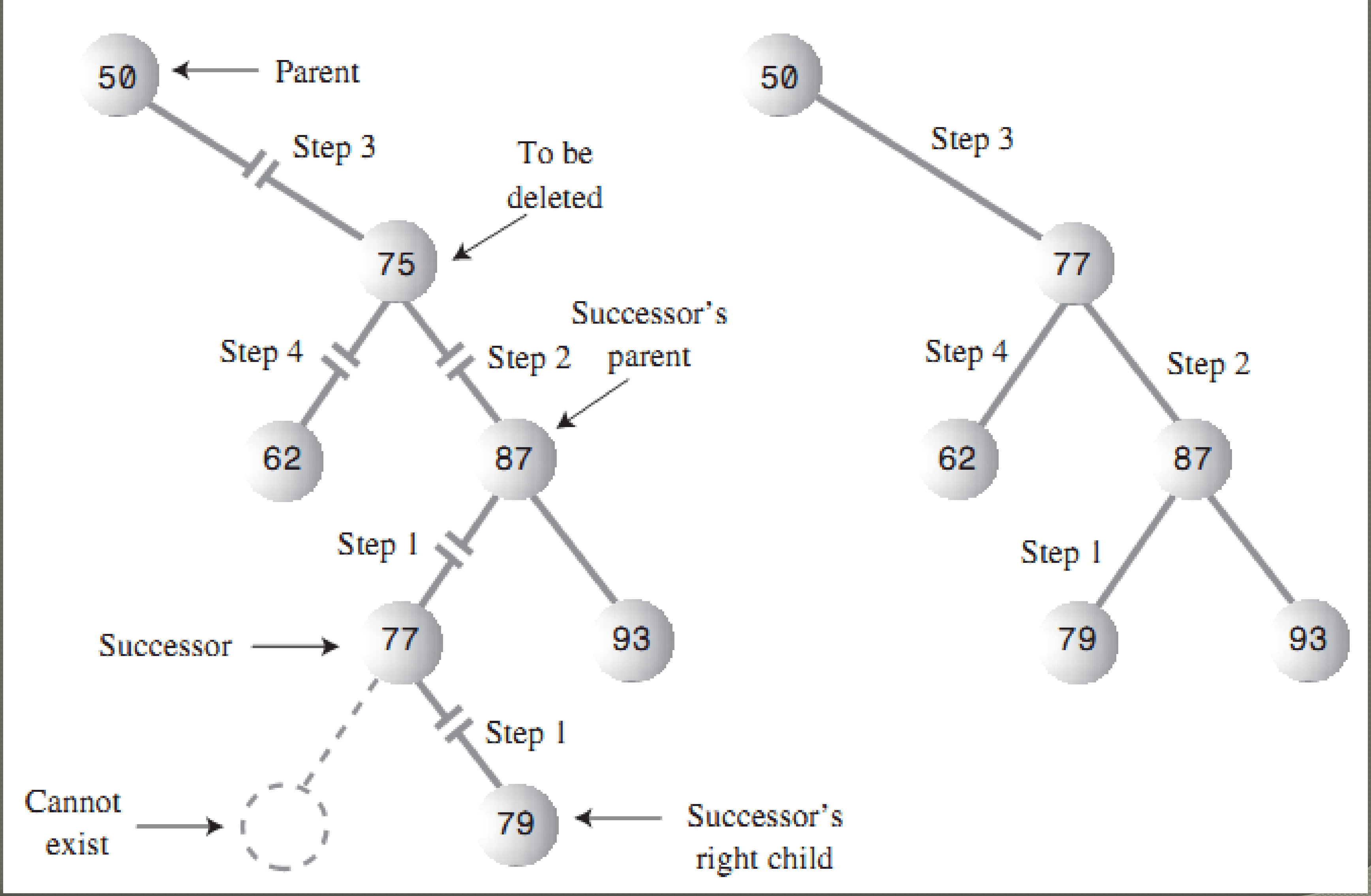



# Find successor

# Case 3.1: Successor is the right child of delNode



# Case 3.2: Successor is left descendant of right child



# Implementation of BST

# Find a node

```
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 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;
   }
```

Homework:

Handle the duplication when insert a new node

See page 380-381 for full code of insertion.

# Traverse

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

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

# 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;
  }
```

Homework: Write the find max function

# Deletion

- Homework:

Read code from 406-415

# Huffman Code

Homework:
Read &
Implement Huffman Code (if possible)

# Red-Black tree, 2-3-4 tree

Read yourself