ZAMAN UNIVERSITY

Data Structures and Algorithms

Chapter 5

Tree

Outline

- Binary Trees
- Traversing Binary Tree
- Red-Black Trees
- Red-Black Tree Insertions
- 2-3-4 Trees

Outline

- Binary Trees
- Traversing Binary Tree
- Red-Black Trees
- Red-Black Tree Insertions
- 2-3-4 Trees

Binary Trees

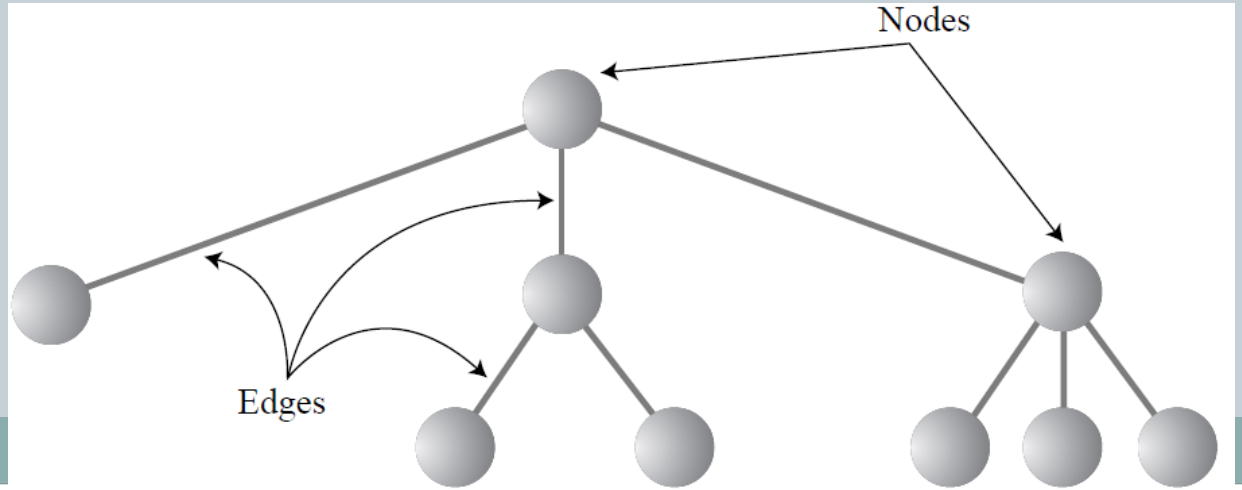
- Binary Trees are one of the fundamental data structures used in programming
- Binary Trees provide advantages that the data structures such as arrays and lists cannot

Why Use Binary Trees?

- Usually because it combines the advantages of two other structures: an ordered array and a linked list
- Slow insertion and deletion in an **ordered array**:
  - during insertion a new item into an ordered array, you first must find where the item will go, and then move all the items with greater keys up one space
  - deletion involves the same multimove operation
- Slow Searching in a **Linked List**:
  - searching will be start checking from head until find the looking item
- Trees is data structure with the quick insertion and deletion

What Is a Tree?

- A tree consists of nodes connected by edges
- The nodes are represented as circles, and the edges as lines connecting the circles
- Typically there is one node in the top row of a tree, with lines connecting to more nodes on the second row, even more on the third, and so on
- Thus trees are small on the top and large on the bottom
- There are different kinds of trees: binary tree and multiway tree



Tree Terminology[1]

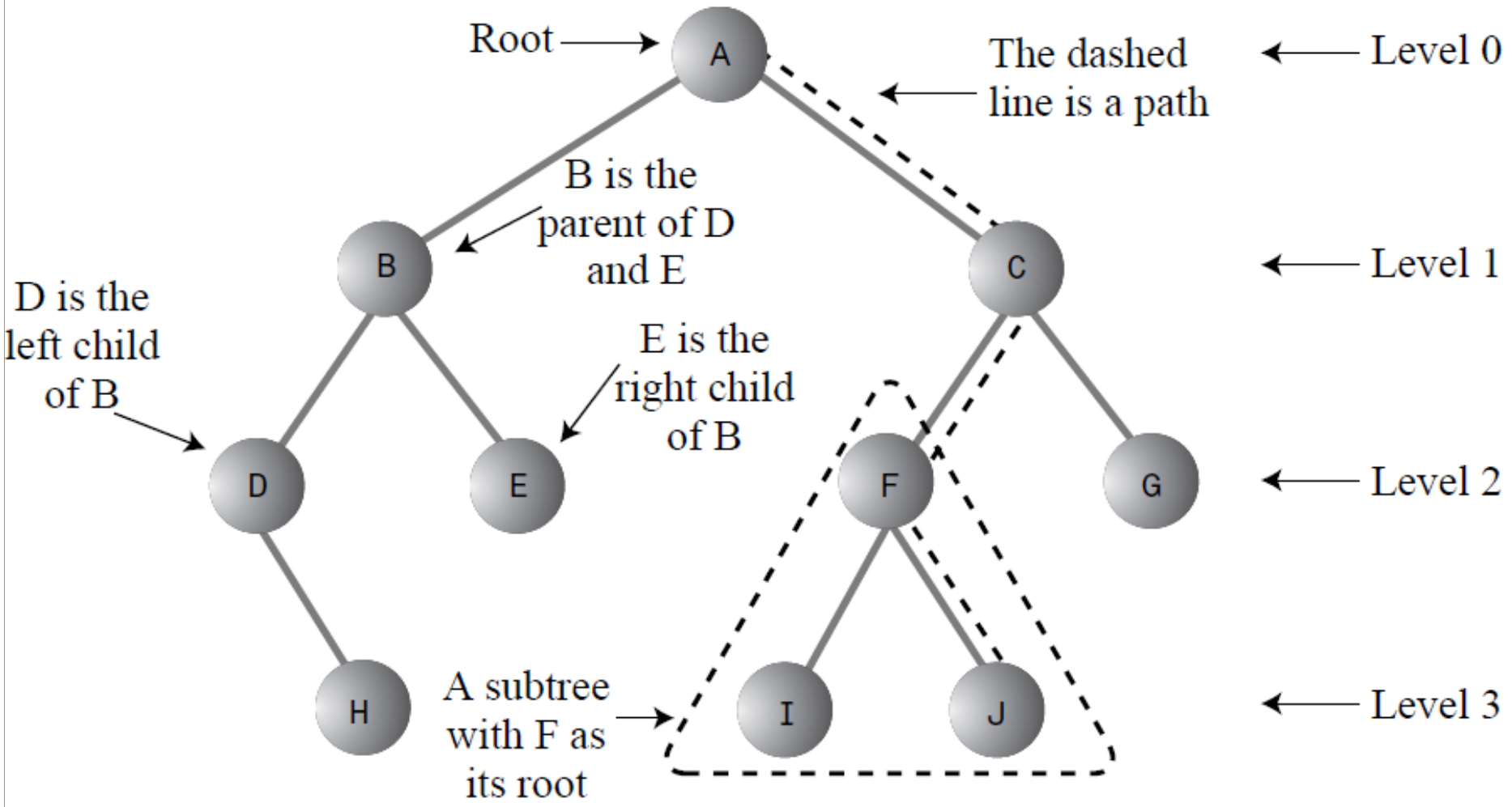
- **Path** - Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a *path*.
- **Root** - The node at the top of the tree is called the *root*. There is only one root in a tree.
- **Parent** - Any node (except the root) has exactly one edge running upward to another node. The node above it is called the *parent* of the node.
- **Child** - Any node can have one or more lines running downward to other nodes. These nodes below a given node are called its *children*.
- **Leaf** - A node that has no children is called a *leaf node* or simply a *leaf*. There can be only one root in a tree, but there can be many leaves.

Tree Terminology[2]

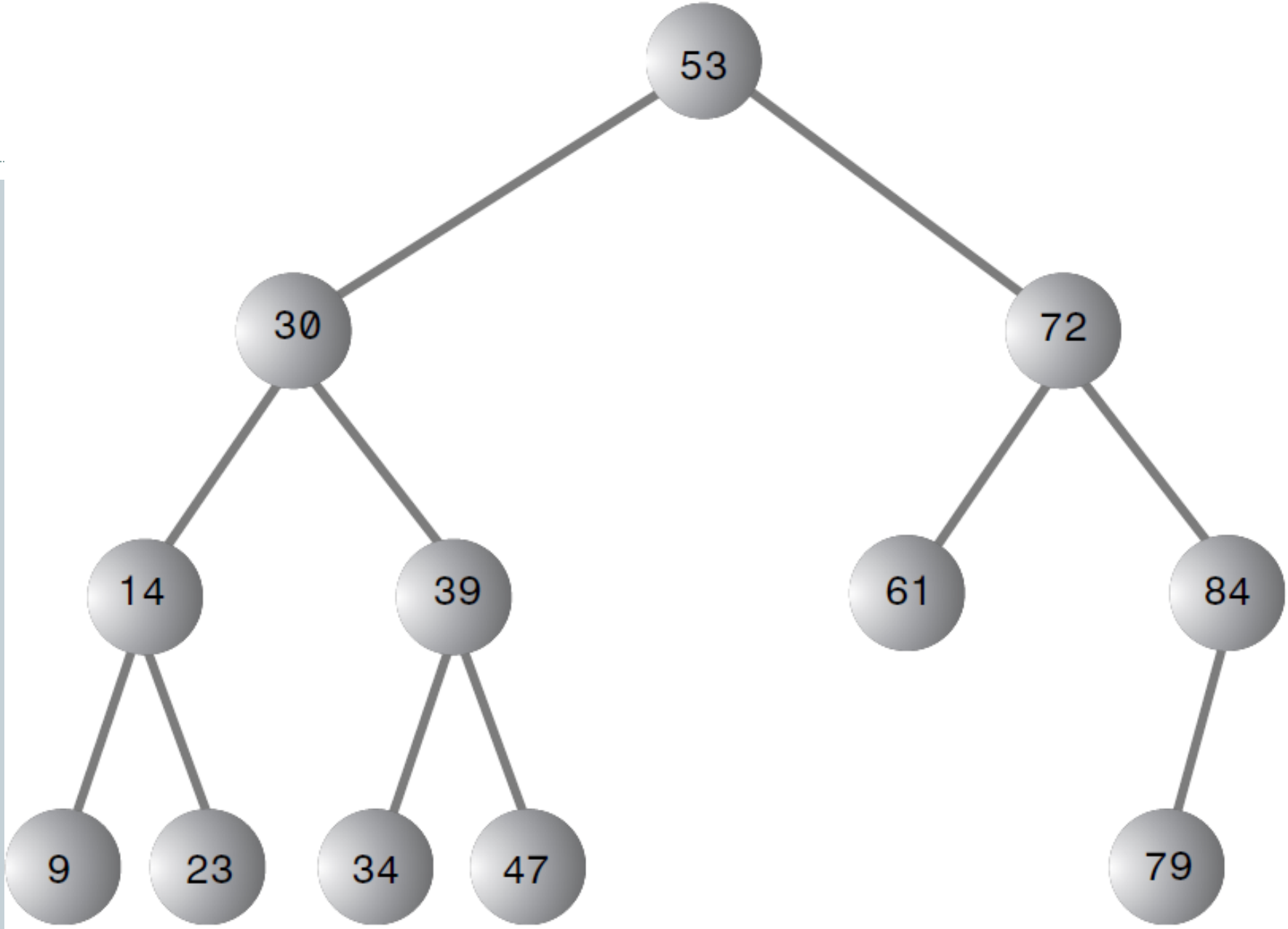
- **Subtree** - Any node can be considered to be the root of a *subtree*, which consists of its children, and its children's children, and so on. If you think in terms of families, a node's subtree contains all its descendants.
- **Visiting** - A node is *visited* when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data members, or displaying it.
- **Traversing** - To *traverse* a tree means to visit all the nodes in some specified order.
- **Levels** - The *level* of a particular node refers to how many generations the node is from the root. If we assume the root is Level 0, its children will be Level 1, its grandchildren will be Level 2, and so on.

Tree Terminology[3]

- **Keys** - In tree diagrams, when a circle represents a node holding a data item, the key value of the item is typically shown in the circle.
- **Binary Trees** - If every node in a tree can have at most two children, the tree is called a *binary tree*. The two children of each node in a binary tree are called the *left child* and the *right child*.
- **Binary Search Tree** is - a node's left child must have a key less than its parent, and a node's right child must have a key greater than or equal to its parent.



H, E, I, J, and G are leaf nodes



A Binary Search Tree

Search in BST

1. Start at the root
2. Compare the value of the item you are searching for the value stored at the root
3. If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*
4. If it is less than the value stored at the root, then search the *left subtree*
5. If it is greater than the value stored at the root, then search the *right subtree*
6. Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

Insert New Node in BST


```
Insert(Node root, Key k)
1. if (root == null) // insertion position found
2. return new Node(k);
3. if (k <= root.key) // proceed to the left branch
4.       root.left = Insert(root.left, k);
5. else // k > root.key, i.e. proceed to the right branch
6.       root.right = Insert(root.right, k);
```

https://www.cs.usfca.edu/~galles/visualization/BST.html

Delete Node in Binary Search Tree

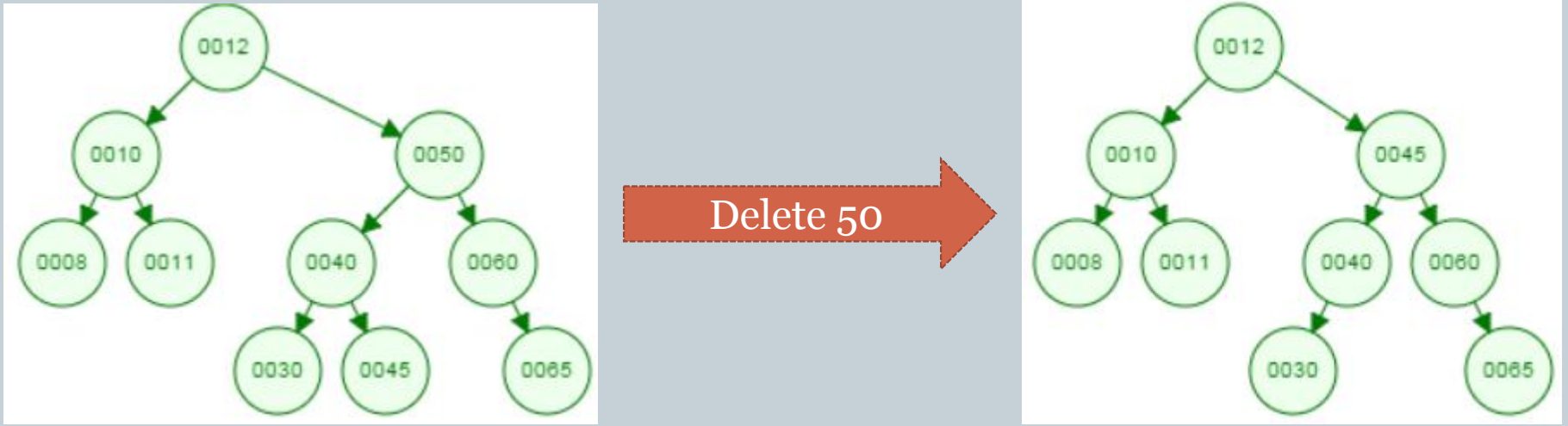
- During deletion node, there possibilities arise:

  1. Deleted node *is* *leaf*

  2. Deleted node has only *one child*

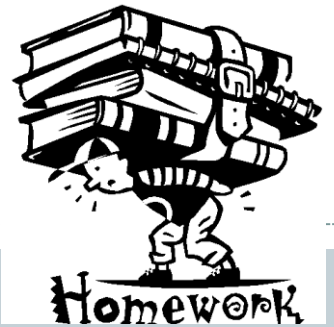
Delete Node in Binary Search Tree (cont.)

3. Deleted node has *two children* - replace node with largest value in left subtree



Pseudo Code of Deletion a Node from BST

1. Find a node with value likes Key
2. Study the found (deleted) node for the following condition:
   - 2.1. node is leaf: Just remove it from BST;
   - 2.2. node has one child: replace node with the child node
     - 2.2.1. if the node has right child then replace with right child;
     - 2.2.2. if the node has left child then replace with left child;
   - 2.3. node has two children: replace node with largest value in left sub-tree
     - 2.3.1. Go to Left child (LeftSubChild) of deleted node
     - 2.3.2. if LeftSubChild has no children (left and right) then replace deleted node with LeftSubChild
     - 2.3.3. if LeftSubChild has right child then
       - 2.3.4. go to the end of right child (node with largest value)
       - 2.3.5. replace node with largest value with deleted node
     - 2.3.4. if LeftSubChild has NO right child
       then replace LeftSubChild with deleted node



1. Create a class of Binary Search Tree
2. Create a search function in BST
3. Create an insertion function in BST
4. Create deletion function in BST

To be continued…