CSCE 3110
Data Structures & Algorithms

- Trees
- Reading: Weiss, chap. 4

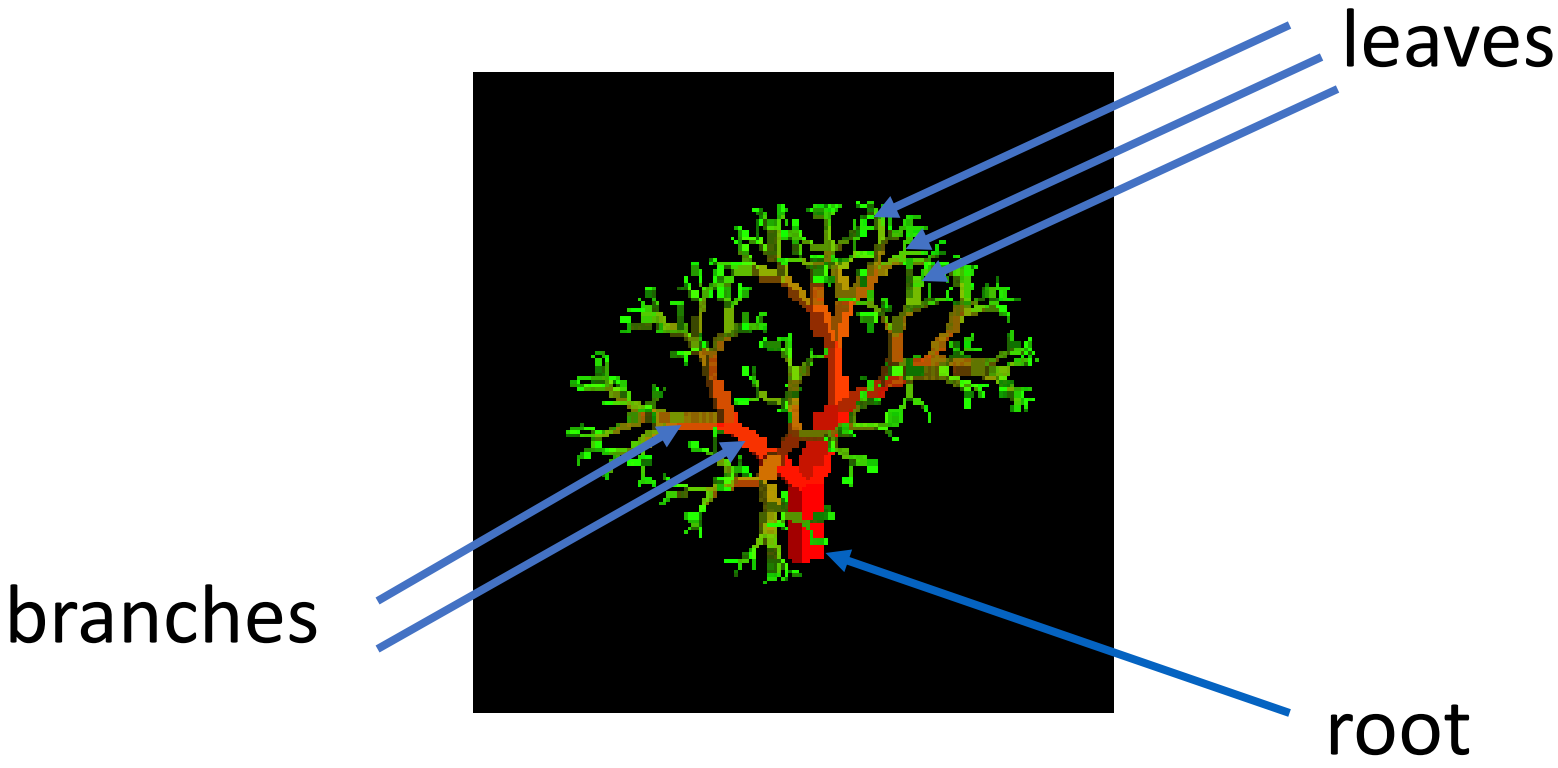
Trees

- Linked lists require O(N) time to traverse the list.
- For large lists this is prohibitive.
- A simple data structure called a “tree” can store data with most operations running in O(log N) time.

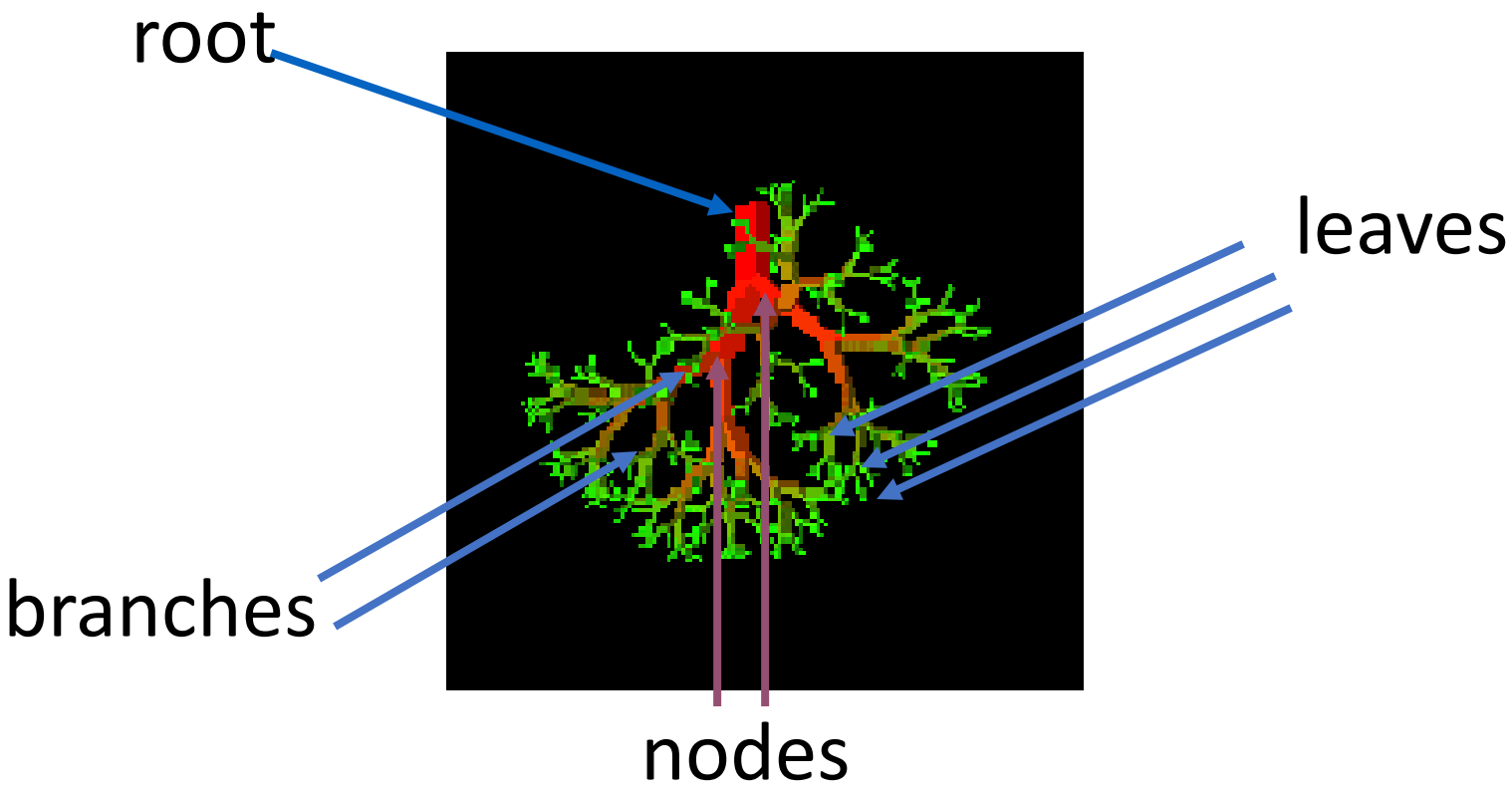
Content

- Trees
  - tree traversal
- Binary Trees
- Binary Search Trees
- Balanced BST
  - AVL trees
- Splay trees

Nature View of a Tree

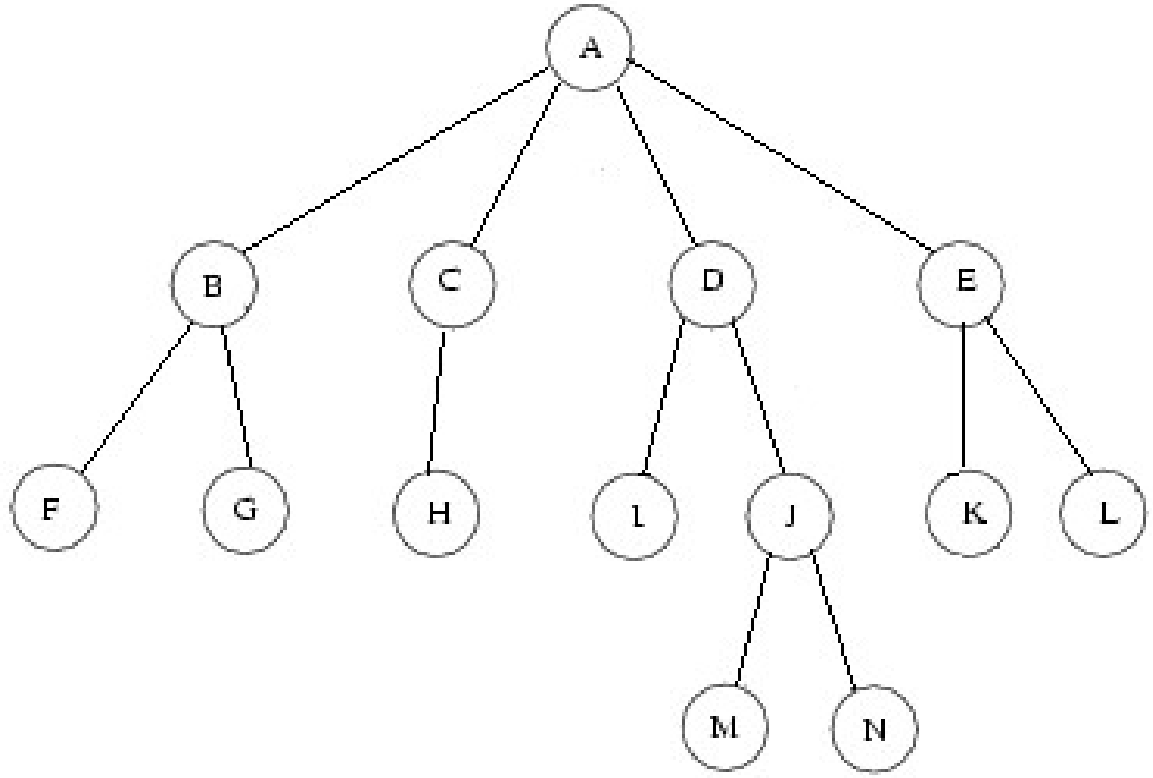


Computer Scientist's View



What is a Tree

- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems



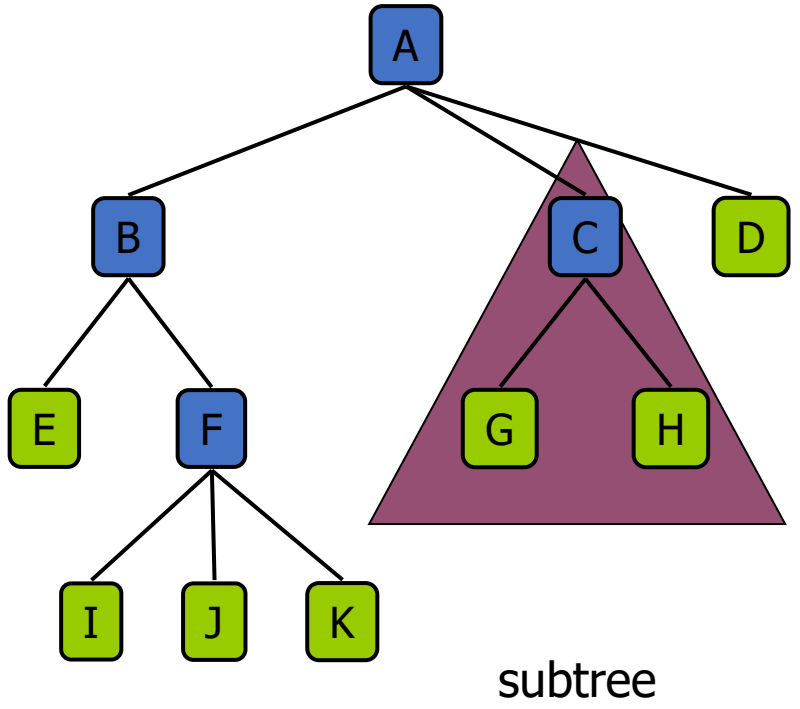
n nodes, n-1 edges (edge to every node except root)

Definition and Tree Trivia

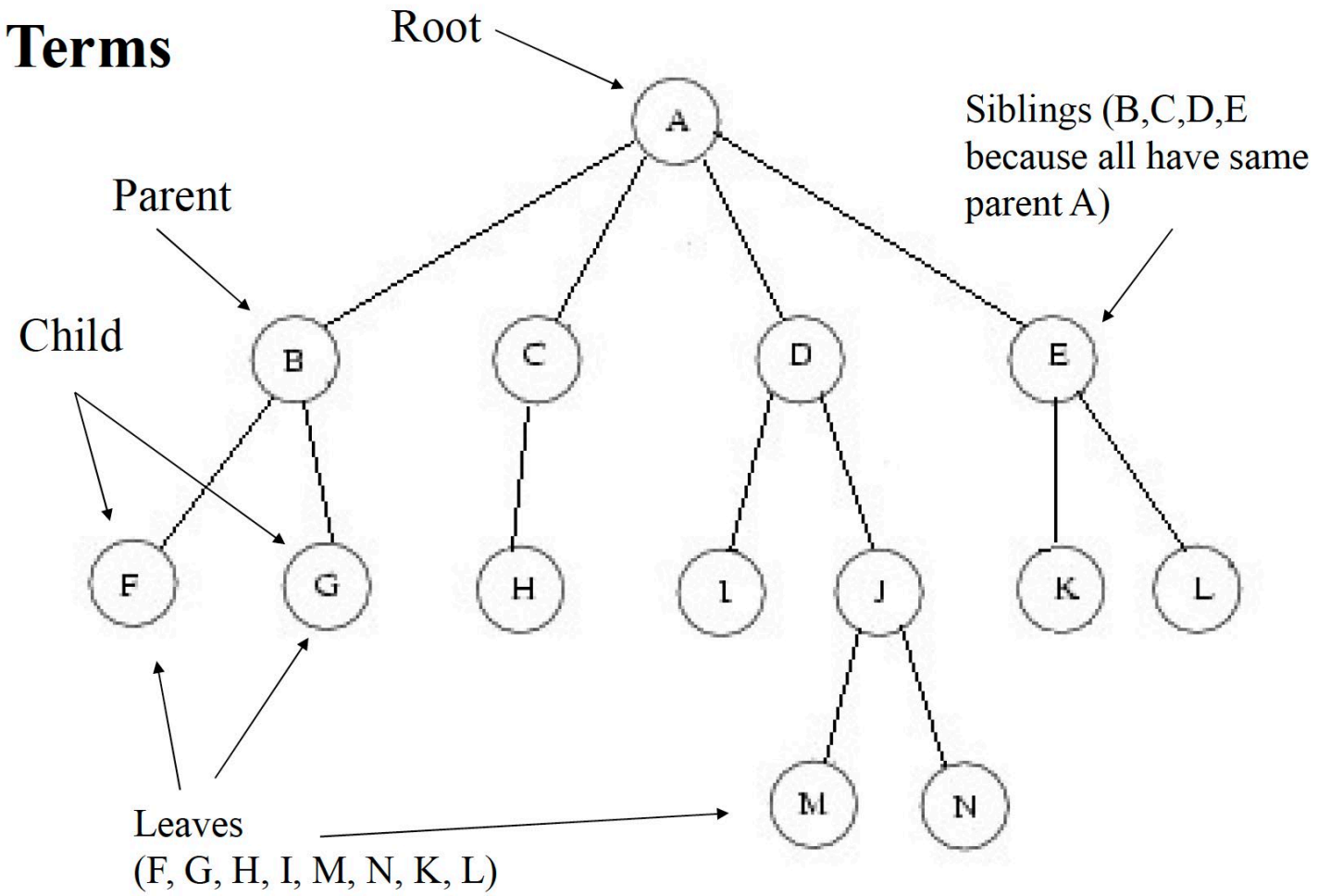
- Recursive Definition of a Tree:
- A tree is a set of nodes that is
  - a. an empty set of nodes, or
  - b. has one node called the root from which zero or more trees (subtrees) descend.
- A tree with N nodes always has ___ edges
- Two nodes in a tree have at most how many paths between them?

Tree Terminology

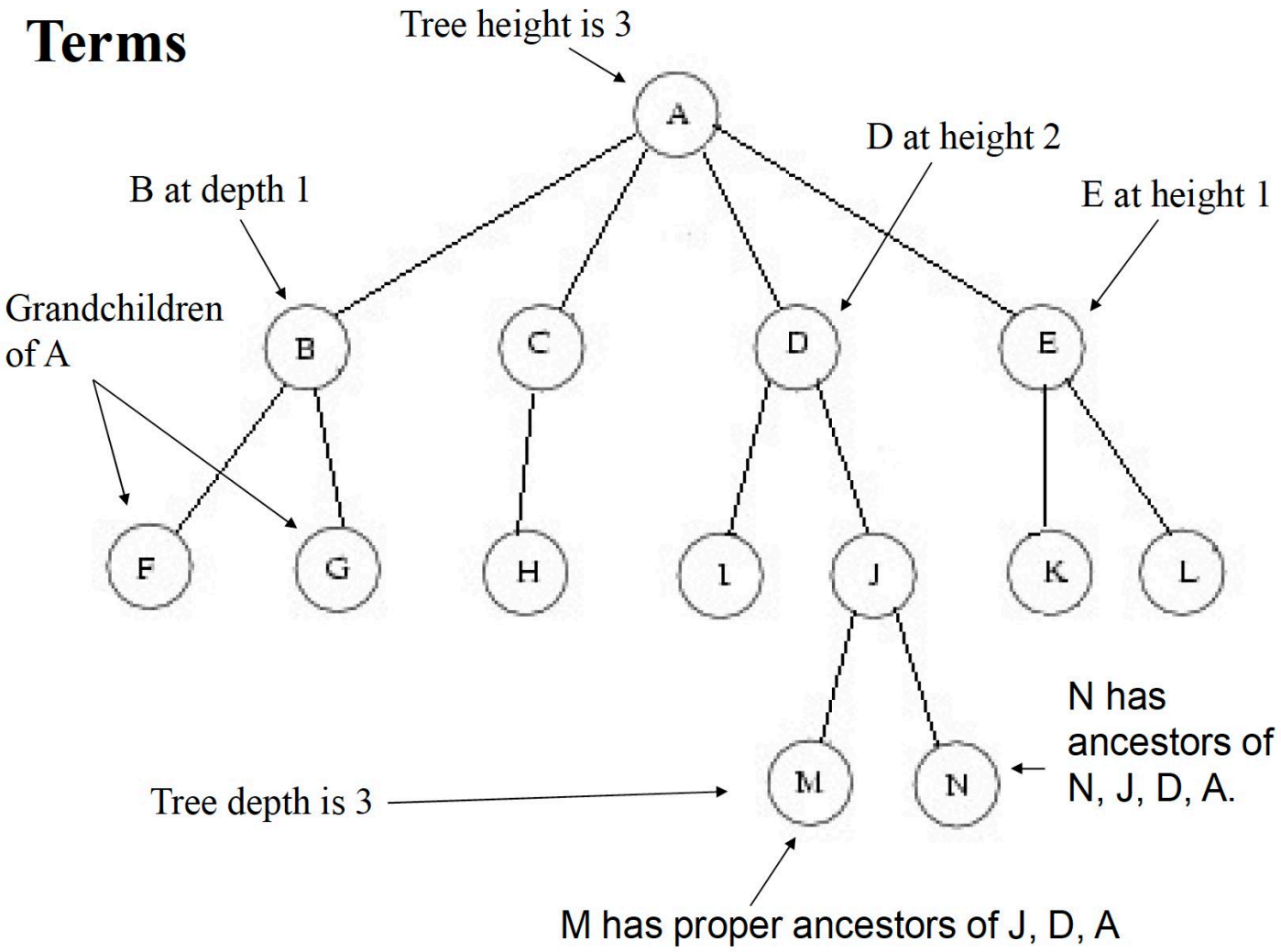
- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors / *length from root (root at 0)*
- **Height** of a node: number of edges on the longest path from the node to a leaf*(leaf at 0)*
- **Height/Depth** of a tree: maximum depth of any node / the length of the longest path from the root node to a leaf .
- Degree of a node: the number of its children
- Degree of a tree: the maximum degree of its node.
- **Subtree**: tree consisting of a node and its descendants



Tree


Terms
Root
Parent
Child
Siblings (B,C,D,E because all have same parent A)
A
B
C
D
E
F
G
H
I
J
K
L
M
N
Leaves
(F, G, H, I, M, N, K, L)

Tree


Terms
Tree height is 3
A
D at height 2
B at depth 1
E at height 1
Grandchildren of A
B
C
D
E
F
G
H
I
J
K
L
N has ancestors of N, J, D, A.
Tree depth is 3
M
N
M has proper ancestors of J, D, A

YMTT (Yet More Tree Terminology)

- Binary: each node has at most two children
- n-ary: each child has at most n children
- Complete: Each row of the tree is filled in left to right before the next one is started
- How deep can a complete binary tree be with n nodes? Or how large d can be?

  •[log_2 n]: complete binary tree is as balanced as possible, and its height grows logarithmically with the number of nodes.

Expression 1: $\sum(2 + 2^2 + 2^3 + ... + 2^n)$

This is the sum of a geometric series. The series is $2 + 2^2 + 2^3 + \cdots + 2^n$.

The sum of a geometric series with the first term a, common ratio r, and n terms is given by:

$$S = a \times \frac{r^n - 1}{r - 1}$$

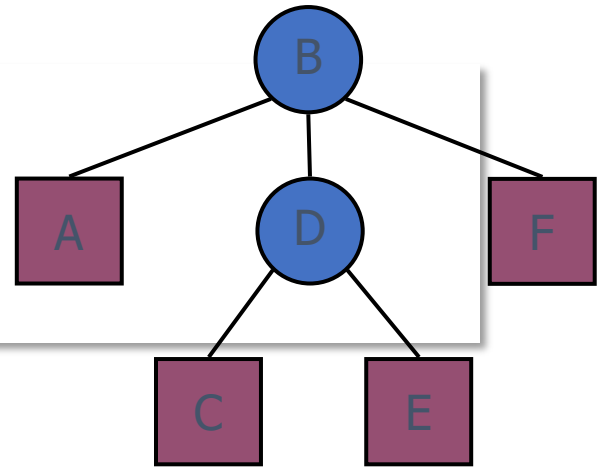
For this particular series:

- The first term $a = 2$,
- The common ratio $r = 2$, and
- The number of terms is $n - 1$ (since it starts from 2^1).

The sum is:

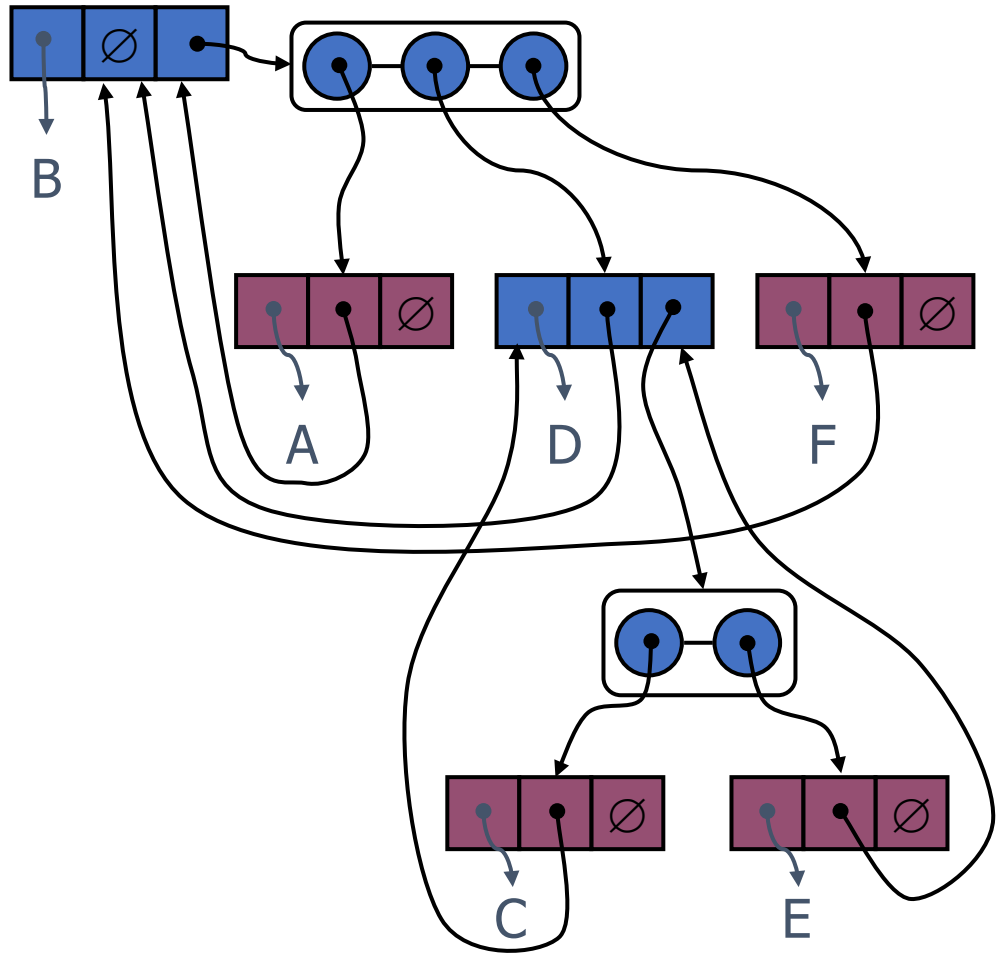
$$S = 2 \times \frac{2^n - 1}{2 - 1} = 2 \times (2^n - 1) = 2^{n+1} - 2$$

A Tree Representation



- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes

```
// A node of N-ary tree
struct node {
        char element;
        node * parent;
        node * child[N];
};
```



Tree ADT

- We use positions to abstract nodes
- Query methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Update methods:
  - insert(p)
  - delete(p)
  - swapElements(p, q)
  - object replaceElement(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Tree Traversals

- Many algorithms involve walking through a tree, and performing some computation at each node

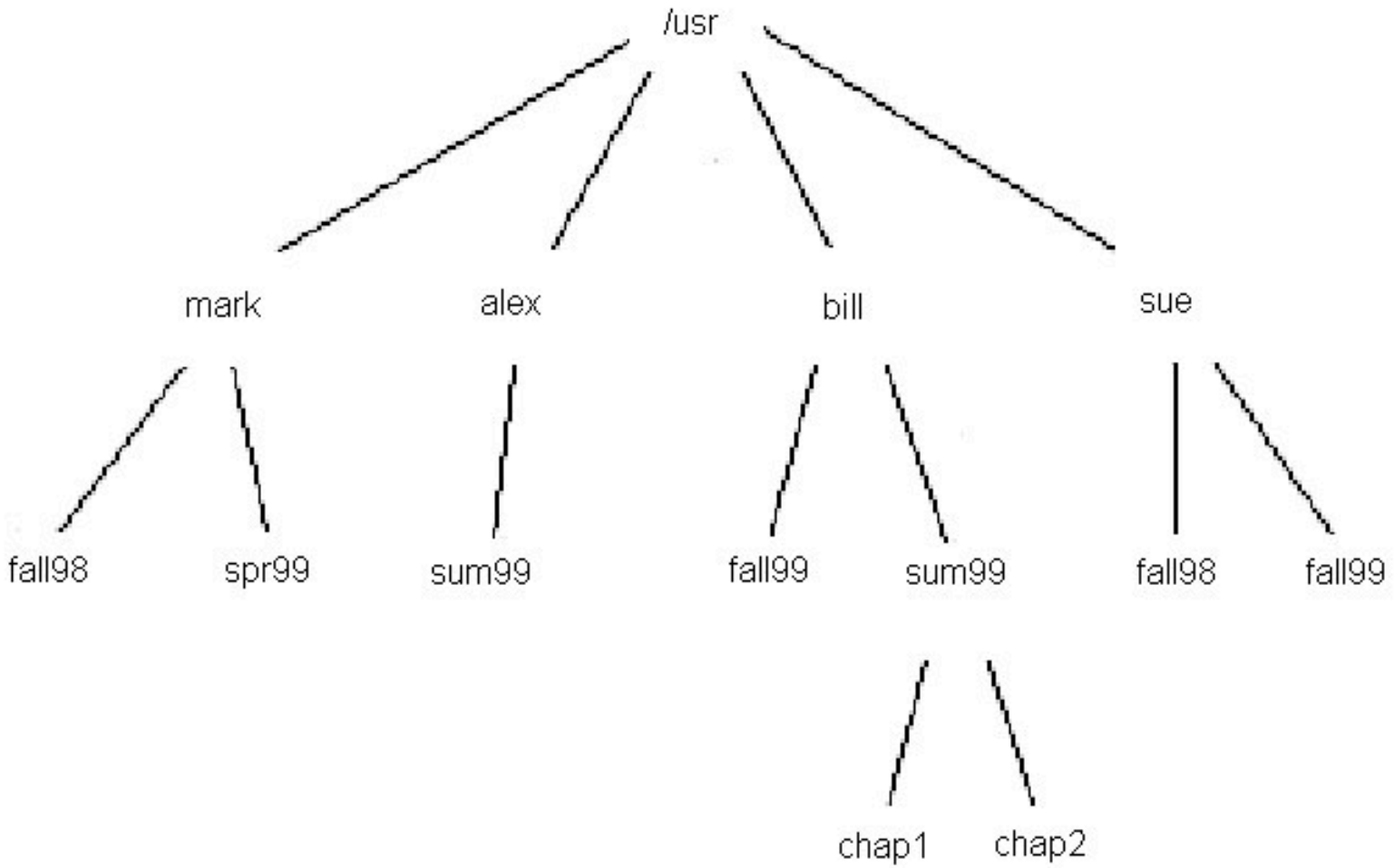
- Walking through a tree is called a **traversal**

- Common kinds of traversal
  - Pre-order: node, then children
  - Post-order: children, then node
  - Level-order: nodes at depth d, nodes at depth d+1, …
  - In-order: left, then node, then right (specific to binary trees)

Tree Traversals

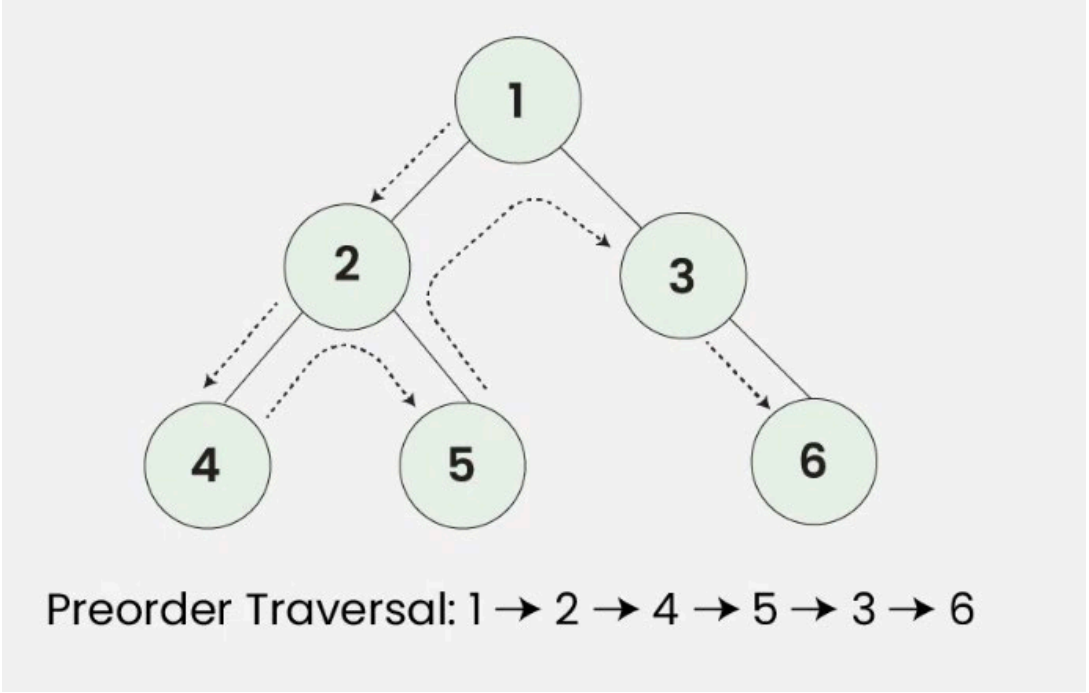
- Directory structures can be implemented as trees.
- It makes sense to print a directory in preorder.
- Preorder prints the parent before printing its children, thus printing the directory name before printing the files it contains.
- We might traverse a directory in postorder if we wished to sum file sizes and roll these up to the parent levels.

Tree for Directory Structure

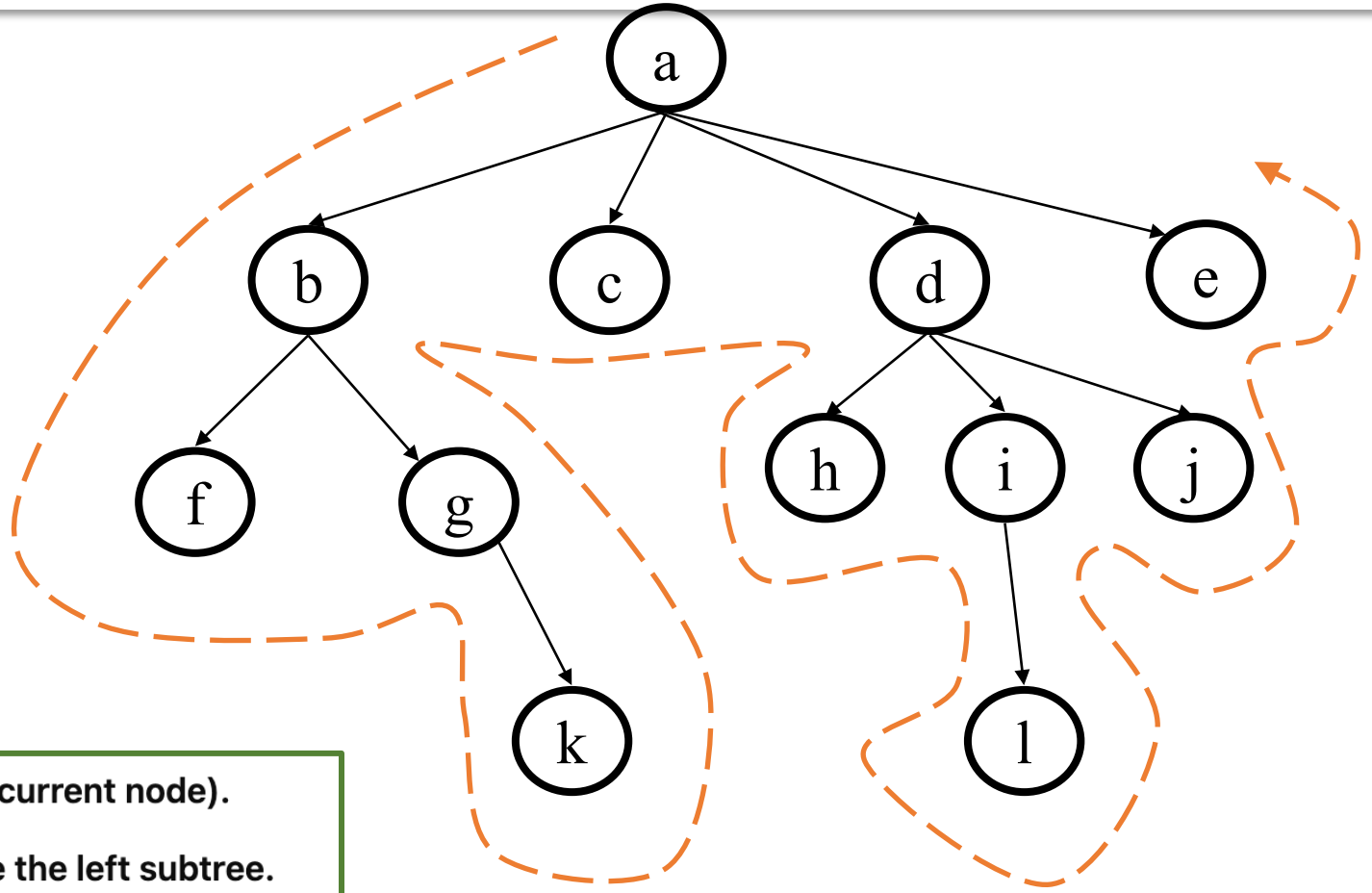


Pre-Order Traversal

1. **Visit the root node (current node).**
2. **Recursively traverse the left subtree.**
3. **Recursively traverse the right subtree.**

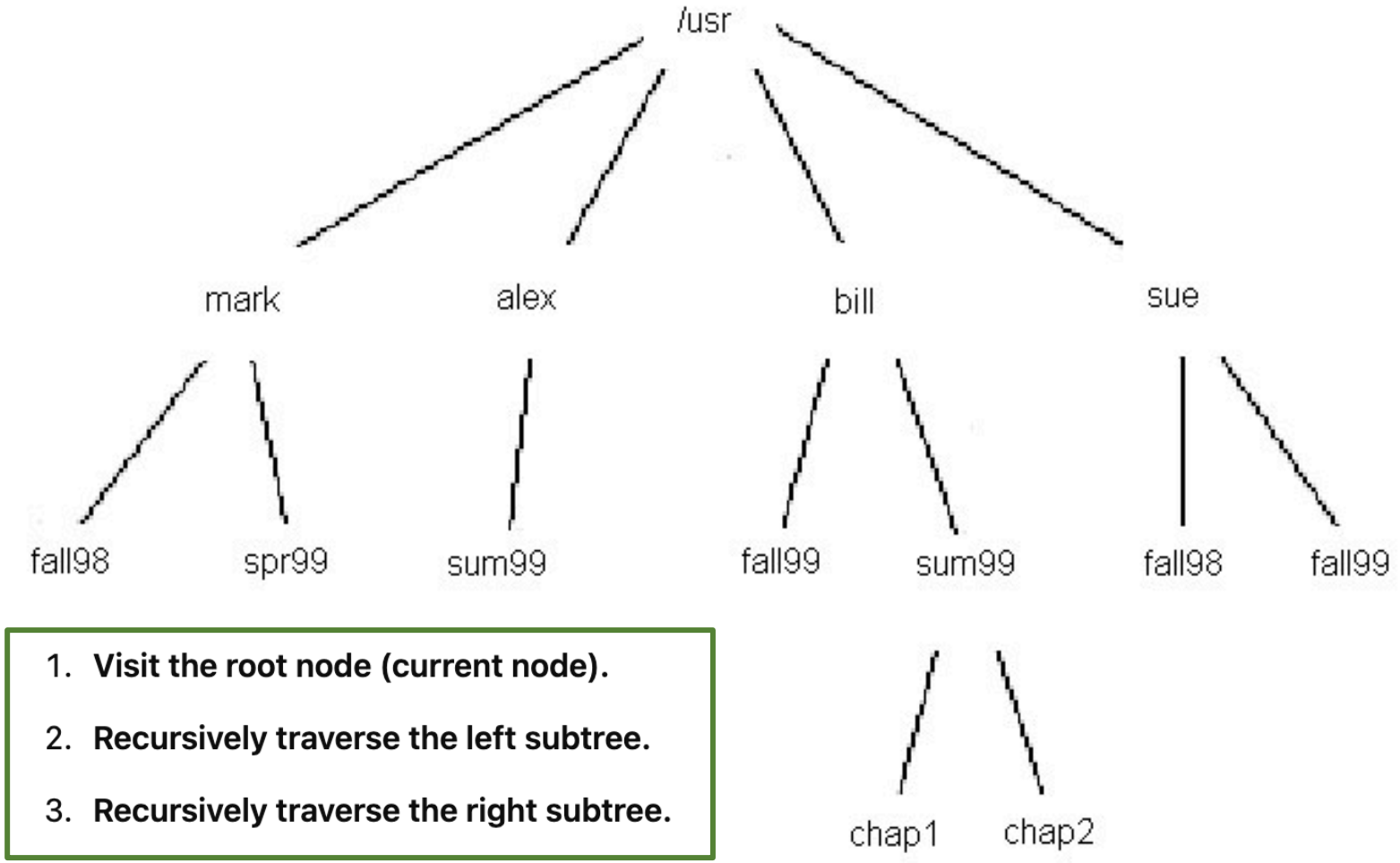


Pre-Order Traversal Example



1. Visit the root node (current node).
2. Recursively traverse the left subtree.
3. Recursively traverse the right subtree.

Tree Preorder



1. **Visit the root node (current node).**
2. **Recursively traverse the left subtree.**
3. **Recursively traverse the right subtree.**

/usr
- **mark**
  - **fall98**
  - **spr99**
- **alex**
  - **sum99**
- **bill**
  - **fall99**
  - **sum99**
    - **chap1**
    - **chap2**
- **sue**
  - **fall98**
  - **fall99**

Pre-Order Traversal

- Perform computation at the node, then recursively perform computation on each child

```
preorder(node * n) {
        node * c;
        if (n != NULL) {

                // DO SOMETHING;

                c = n->first_child();
                while (c != NULL) {
                        preorder(c);
                        c = c->next_sibling();
                }
        }
}
```

1. Visit the root node (current node).
2. Recursively traverse the left subtree.
3. Recursively traverse the right subtree.

Pre-Order Applications

- Use when computation at node depends upon values calculated higher in the tree (closer to root)

- Example: computing depth
  - The depth of a node is the number of edges from the node to the tree's root node
  - depth(node) = 1 + depth(parent of node)

Pre-Order Example: Computing Depth of All Nodes

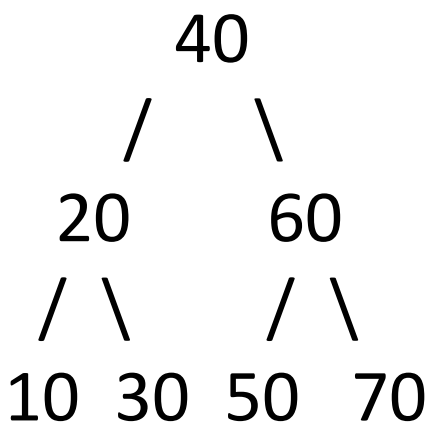
- Add a field `depth` to all nodes
- Call Depth(root,0) to set `depth` field

```
Depth(node * n, int d) {
  node * c;

  if (n != NULL) {
     n->depth = d;

     c = n->first_child();
     while (c != NULL) {
           Depth(c, d+1);
           c = c->next_sibling();
     }
  }
}
```

Practice: Pre-Order Traversal



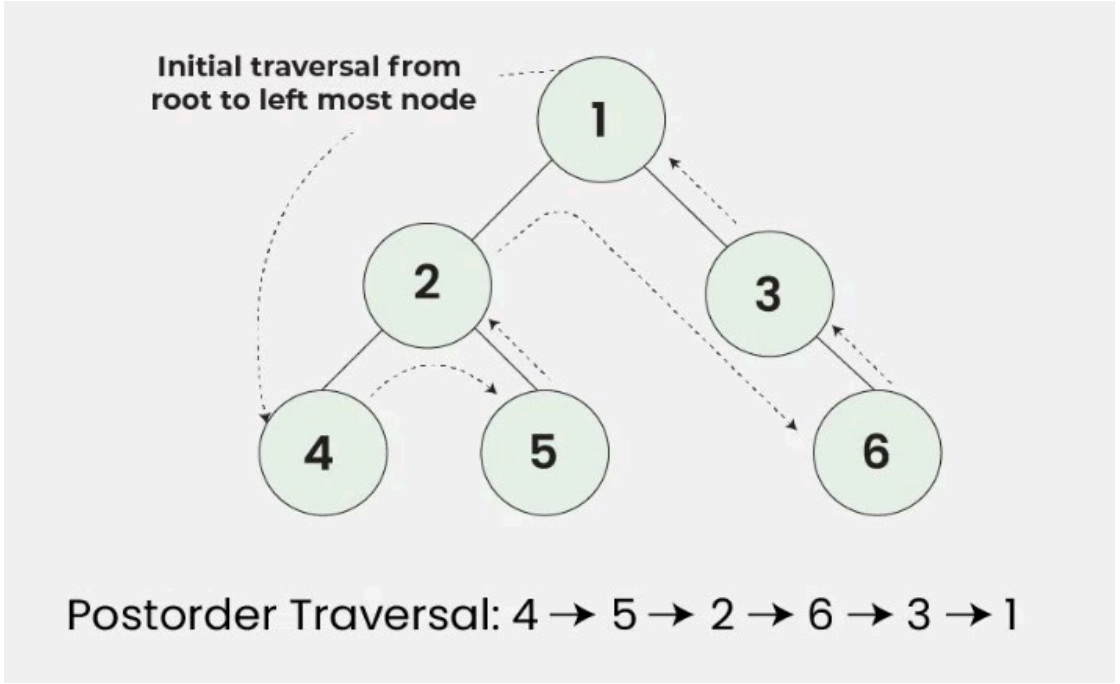
Pre-order traversal will be:
40 – 20 – 10 – 30 – 60 – 50 - 70

Post-Order Traversal

1. **Recursively traverse the left subtree** (or first child in the case of a general tree).
2. **Recursively traverse the right subtree** (or the next sibling).
3. **Visit the root node** (or the current node).

In post-order traversal, you first visit all the children of a node before processing the node itself.

This traversal is useful in situations where you need to process all dependencies of a node before processing the node itself (e.g., in expression trees or for deleting a tree).



Post-Order Traversal

- Recursively perform computation on each child, then perform computation at node

```
postorder(node * n) {
  node * c;
  if (n != NULL) {
      c = n->first_child();
      while (c != NULL) {
             postorder(c);
             c = c->next_sibling();
      }

  // DO SOMETHING;

}
```

Post-Order Applications

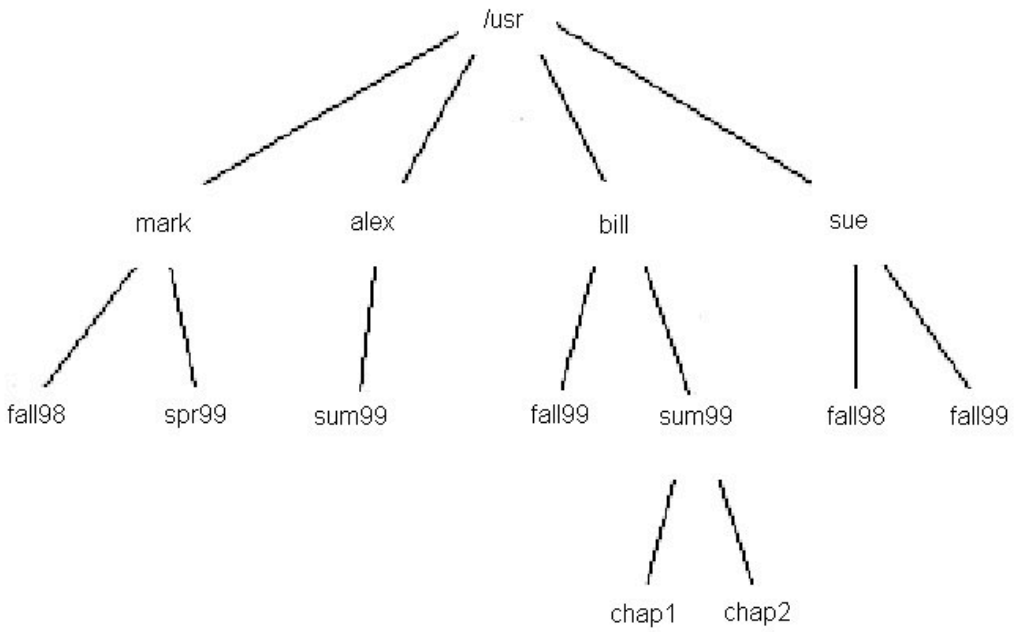
- Use when computation at node depends on values calculated **lower** in tree (closer to leaves)

- Example: computing height
  - The height of a node is the number of edges on the longest path from the node to a leaf.
  - A leaf node will have a height of 0.
  - height(node) = 1 + MAX(height($child_1$), … height($child_k$))

- Example: size of tree rooted at node

  size(node) = 1 + size($child_1$) + … + size($child_k$)

Tree Postorder



Assume each is a directory Except chap1 and chap2.

Each directory is 1 block.
chap 1 is 5 blocks.
chap 2 is 6 blocks.

fall98	1
spr99	1
mark	3
sum99	1
alex	2
fall99	1
chap1	5
chap2	6
sum99	12
bill	14
fall98	1
fall99	1
sue	3
/usr	23

mark is 3 because it counts for 1 and has two children

Post-Order Example: Computing Size of Tree

- Call Size(root) to compute number of nodes in tree

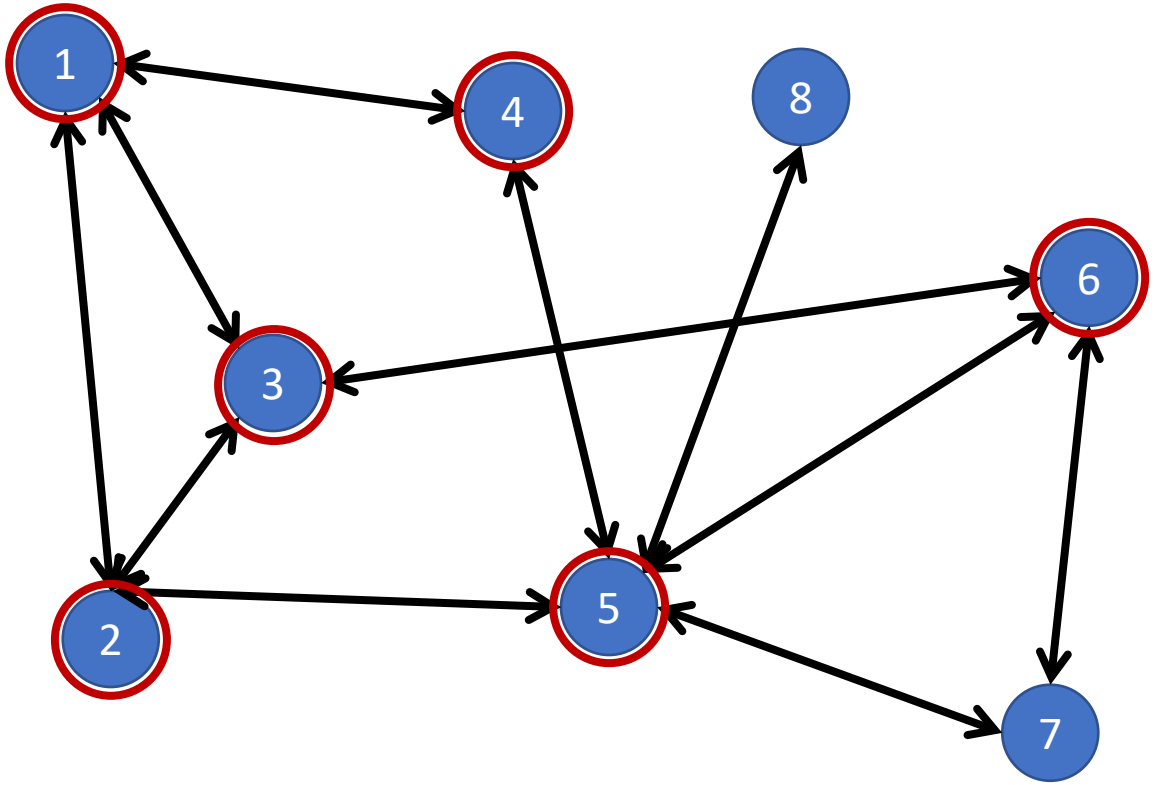
```
int Size(node * n) {
  node * c;
  if (n == NULL) return 0;
  else {
      int m = 1;
      c = n->first_child();

      while (c != NULL) {
             m += Size(c);
             c = c->next_sibling();
      }
   }
   return m;
}
```

Depth-First Search

- Pre-Order and Post-Order traversals are examples of depth-first search:
  - Nodes are visited *deeply* on left-most branches *before* any nodes are visited on right-most branches
  - NOTE: visiting right deeply before left would still be depth-first - crucial idea is “go deep first”
- In DFS the nodes “being worked on” are kept on a stack (where?)

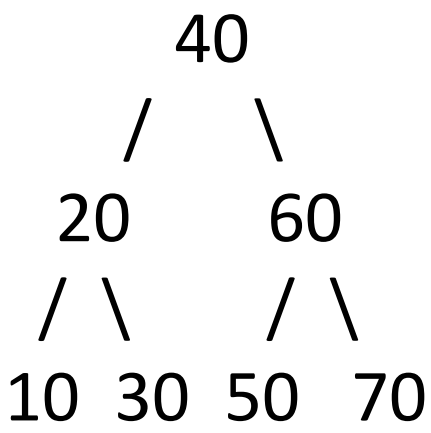
Practice: Depth-First Search



- Start at 2, search for 4, smaller value preference

DFS Path: 2 → 1 → 3 → 6 → 5 → 4

Practice: Post-Order Traversal

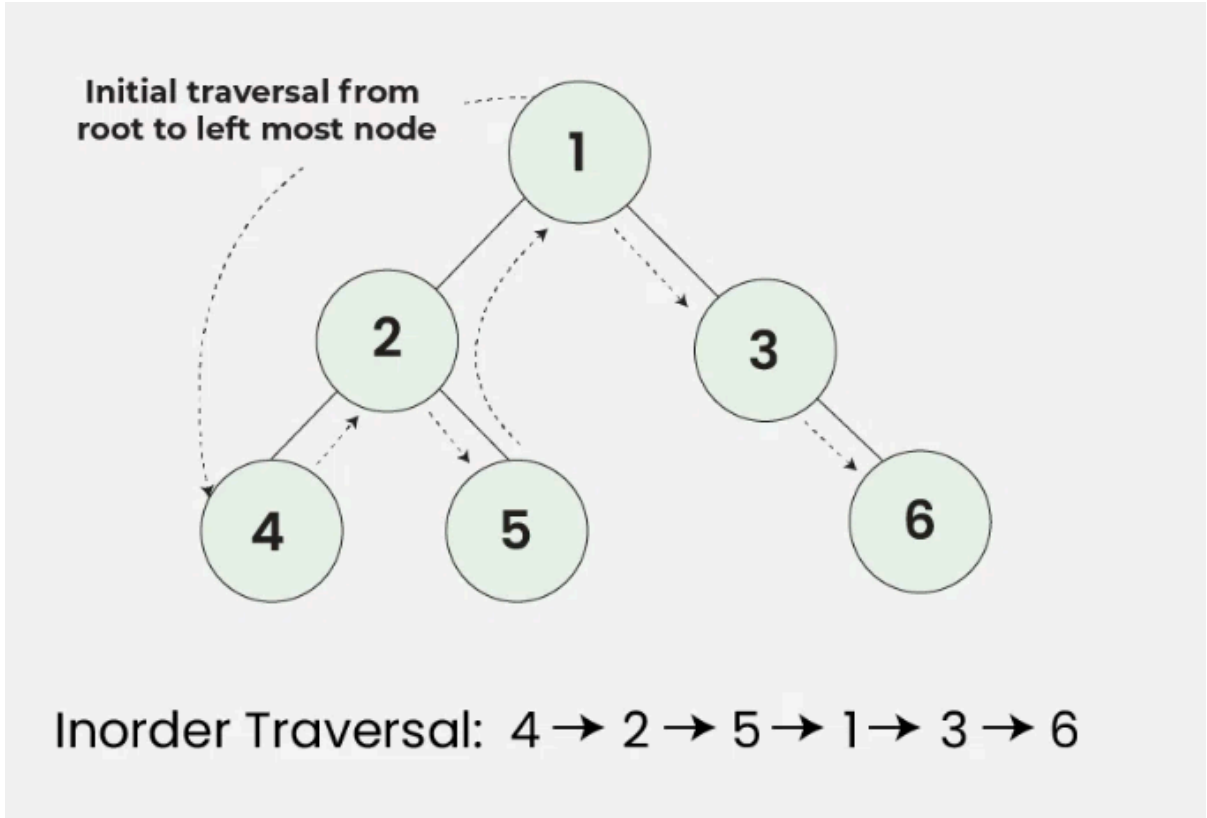


Post-order traversal will be:
10 → 30 → 20 → 50 → 70 → 60 → 40

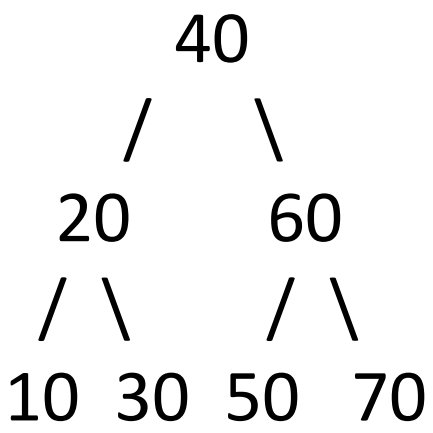
In-Order Traversal

1. **Recursively traverse the left subtree.**
2. **Visit the current node.**
3. **Recursively traverse the right subtree.**

The left subtree is always fully processed before the current node is visited, and the right subtree is processed afterward. In-order traversal is especially useful in **binary search trees (BSTs)** because it visits the nodes in **sorted order** (ascending).

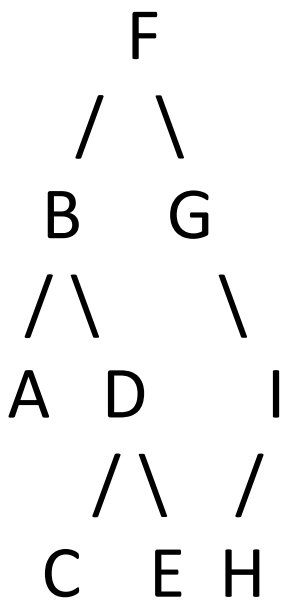


Practice: In-Order Traversal



In-order traversal will be:
10 → 20 → 30 → 40 → 50 → 60 → 70

Practice: In-Order Traversal



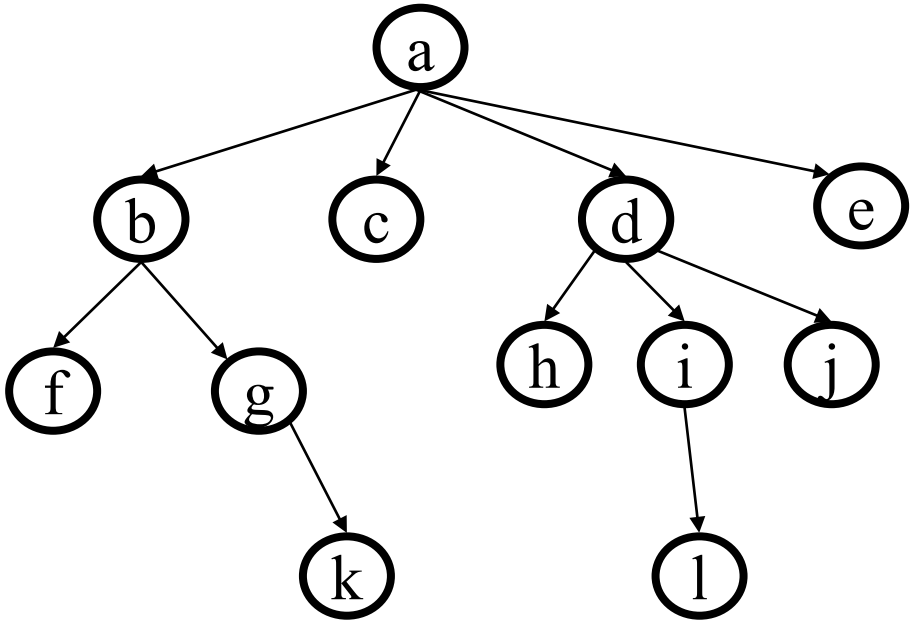
In-order traversal will be:

A → B → C → D → E → F → G → H → I

Level-Order (Breadth-First) Traversal

- Consider task of traversing tree level-by-level from top to bottom (alphabetic order, in example below)

- Which data structure can best keep track of nodes?



Level-Order (Breadth-First) Algorithm

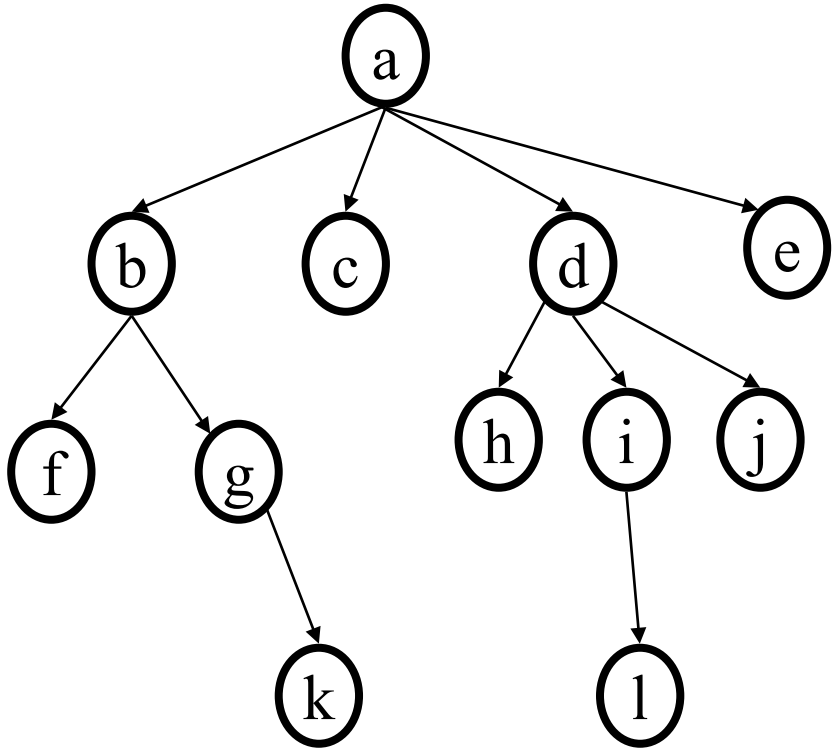
Put root in a Queue

Repeat until Queue is empty:

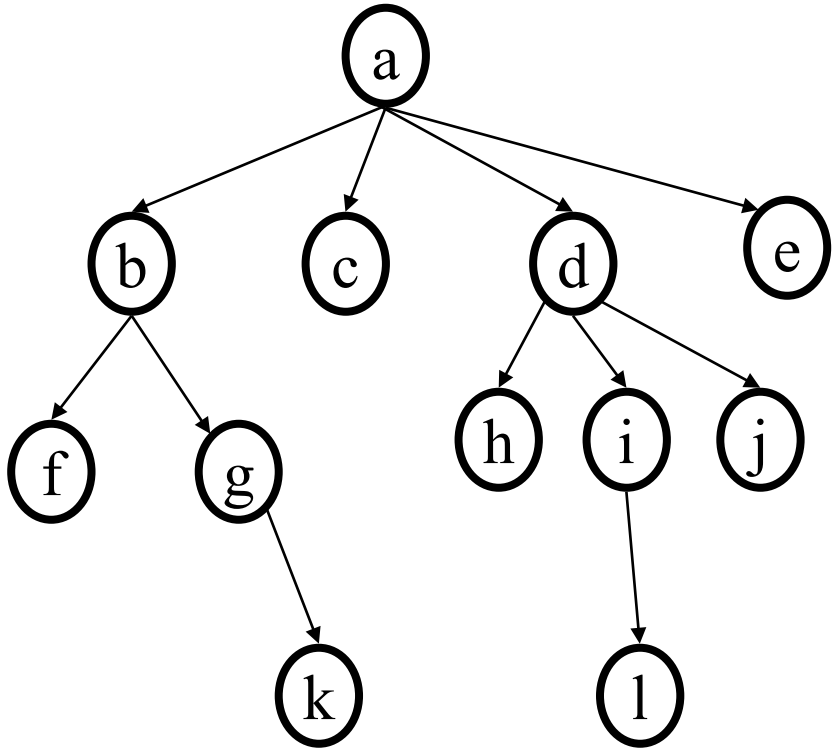
- Dequeue a node
- Process it
- Add its children to queue

Example: Level-Order Queue

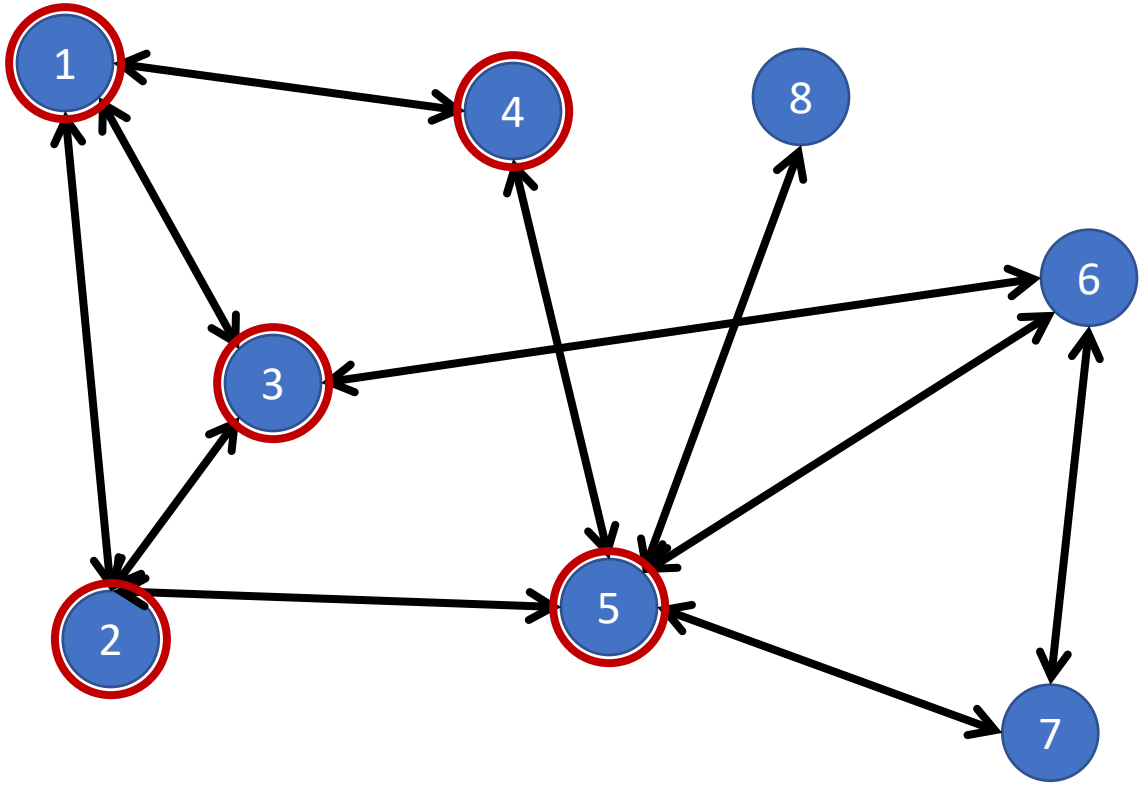
Process	Enqueue	Q
	a	**a**
a	b,c,d,e	**bcde**
b	f,g	**cdefg**
c		**defg**
d	h,i,j	**efghij**
e		**fghij**
f		**ghij**
g	k	**hijk**
h		**ijk**
i	l	**jkl**
j		**kl**
k		**l**
l		



Example: Level-Order Queue



Practice: Breadth-First Search



- Start at 2, search for 4, smaller value preference

Discovery Queue

2	1	3	5	4	6	7	8

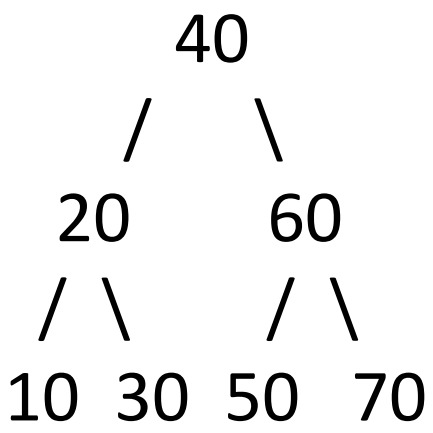
Applications of Breadth-First Search

- Find shortest path from root to a given node N
  - if node N is at depth k, BFS will never visit a node at depth > k
  - important for really deep trees

- Generalizes to finding shortest paths in graphs

- Spidering the world wide web
  - **Web Crawling**: BFS is used in web crawling (spidering), where from a root URL, web pages are fetched in layers, getting pages farther and farther away.

Practice: Level-Order Traversal



In-order traversal will be:
40 → 20 → 60 → 10 → 30 → 50 → 70

Level-Order Example: Printing the Tree

- Call Print(root) to print tree contents

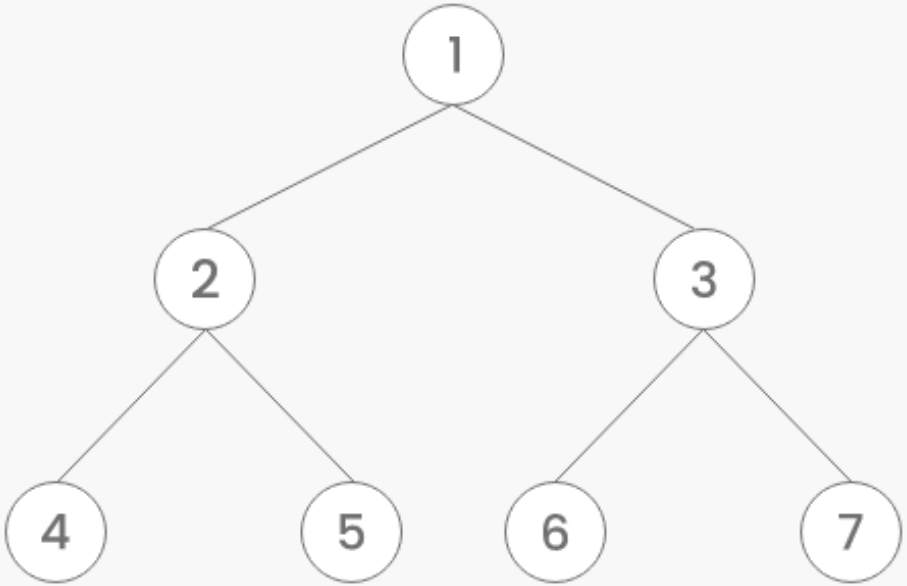
```
print(node * root) {
  node * n, c;  queue Q;

  Q.enqueue(root);
  while (! Q.empty()) {
      n = Q.dequeue();
      print n->data;

      c = n->first_child();
      while (c != NULL) {
             Q.enqueue(c);
             c = c->next_sibling();
      }
  }
}
```

Tree Traversal

Tree Traversal Techniques



Inorder Traversal

4	2	5	1	6	3	7

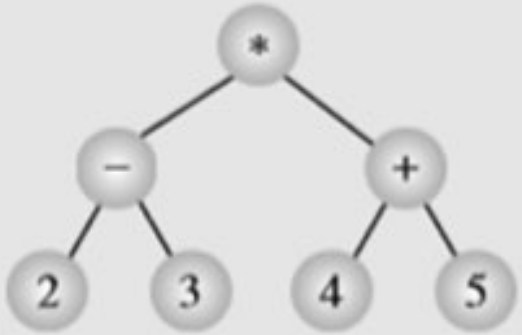
Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

Tree Traversal

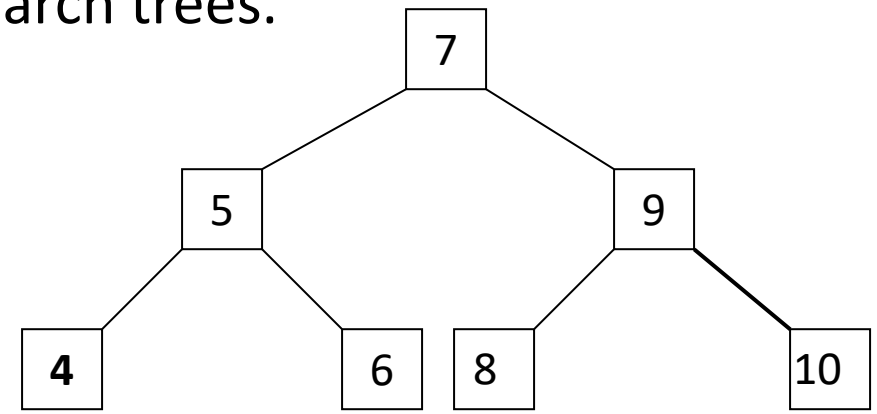


	$2-3*4+5$	$(2-3)*(4+5)$	$2-(3*4+5)$
Preorder	$+-2*345$	$*-23+45$	$-2+*345$
Inorder	$2-3*4+5$	$2-3*4+5$	$2-3*4+5$
Postorder	$234*-5+$	$23-45+*$	$234*5+-$
	(a)	(b)	(c)

Binary Search Tree

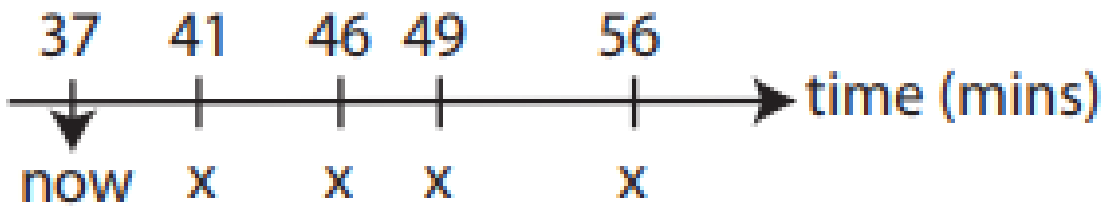
- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
  - Every element has a unique key/value.
  - The keys in the nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree (BST property)
  - The left and right subtrees are also binary search trees.

- For any given node with value N:
  - All nodes in the **left subtree** have values **less than** N.
  - All nodes in the **right subtree** have values **greater than** N.



Runway reservation system

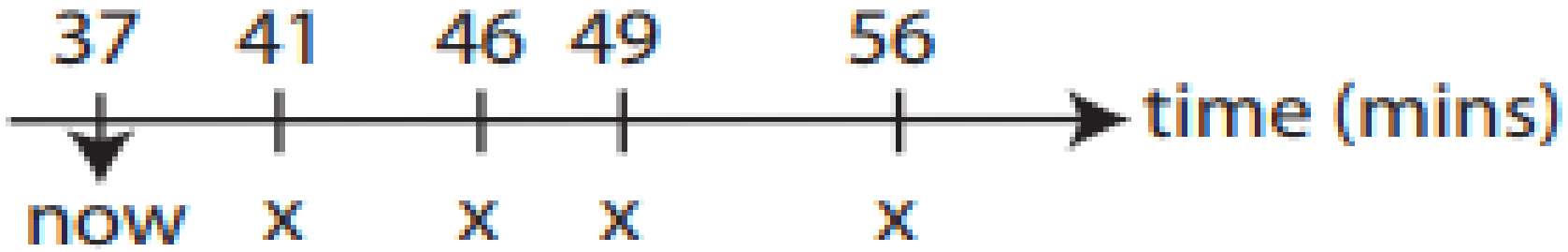
- Airport with single (very busy) runway
- “Reservations” for future landings
- When plane lands, it is removed from the system
- Reserve request specify “requested landing time” t
- Add t to the set if no other landings are scheduled within k minutes either way. Assume that k can vary.
  - – else error, don’t schedule



Reserved time

Runway Reservation System

- Let R denote the reserved landing times: $R = (41, 46, 49, 56)$ and $k = 3$
- Request for time: 44, 53, 20



- The request for **44 minutes** is rejected due to a conflict with 46 minutes.
- The requests for **53 minutes** and **20 minutes** can be scheduled without conflicts.

Runway reservation system

- Keep `R` as a sorted list. Algorithm:

```
init:  R = [ ]
req(t):
       if t < now: return "error"
       for i in range (len(R)):    ## Conflict Checking
              if abs(t-R[i]) < k: return "error"
       R.append(t)
       R = sorted(R)
land:  t = R[0]
       if (t != now)
              return error
       R = R[1: ] (drop R[0] from R)
```

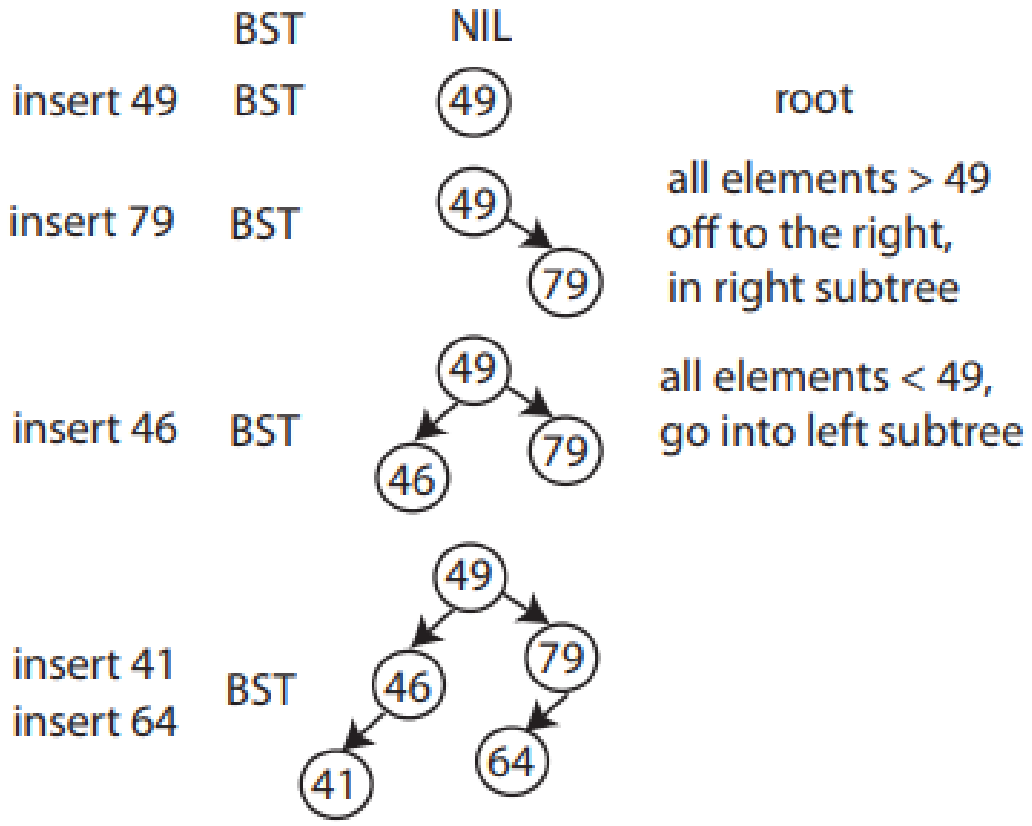
Can we do better?

- Sorted list: Appending and sorting takes $\Theta(n \log n)$ time. However, it is possible to insert new time/plane rather than append and sort, but insertion takes $\Theta(n)$ time. A k minute check can be done in $O(1)$ once the insertion point is found.
- Sorted array: It is possible to do binary search to find place to insert in $O(\lg n)$ time. Using binary search, we find the smallest i such that $R[i] \geq t$, i.e., the next larger element. We then compare $R[i]$ and $R[i-1]$ against t. Actual insertion however requires shifting elements which requires $\Theta(n)$ time.
- Unsorted list/array: k minute check takes $O(n)$ time.

Binary Search Trees (BST)

- Insertion: `insert(val)`

```
typedef struct BinaryNode {
        int  key;
        BinaryNode *right;
        BinaryNode *left;
} ptnode;
```



Definition: A Binary Search Tree (BST) is a binary tree where:

- The left child of any node contains values smaller than the node's key.
- The right child of any node contains values larger than the node's key.

Insertion into a BST

```
void insert (ptnode * & node, int key){
  if (!node){
        node = (ptnode) malloc(sizeof(ptnode));
        node->key = key;
        node->left = node->right = NULL;

  } else if (key < node->key) {
        insert(node->left, key);

  } else if (key > node->key) {
        insert(node->right, key);
  }
}
```

The process continues until it finds an empty position (NULL) where the new node can be inserted.

time complexity? given the height of the tree h

Time Complexity: $O(h)$, where h is the height of the tree.

- In the worst case, $h = n$, so $O(n)$.
- In the best case (balanced tree), $h = \log n$, so $O(\log n)$.

Insertion into BST Time Complexity Analysis

Time Complexity Analysis:

The time complexity of inserting a node into a **Binary Search Tree (BST)** depends on the **height** of the tree, denoted as h.

- In the **best case** (for a balanced BST), the height of the tree is **logarithmic** with respect to the number of nodes, i.e., $h = O(\log n)$, where n is the number of nodes in the tree.
- In the **worst case** (for an unbalanced or skewed BST), the tree can degenerate into a **linked list**, and the height becomes equal to n, the number of nodes, i.e., $h = O(n)$.

Insertion Time Complexity:

- **Best Case (Balanced Tree)**: $O(\log n)$
- **Worst Case (Unbalanced Tree)**: $O(n)$

Finding a value in the BST if it exists: find(val)

- Follow left and right pointers until you find it or hit NULL.

Search in BST - Pseudocode

```
if the tree is empty
        return NULL     # The value does not exist in the tree.

else if  the item in the node equals the target
        return the node value     # The value is found, return the node.

else if  the item in the node is greater than the target
        return the result of searching the left subtree   # Search in the left subtree.

else if  the item in the node is smaller than the target
        return the result of searching the right subtree   # Search in the right subtree.
```

Search in a BST

```
Ptnode search(ptnode * root,  int key){
  /* return a pointer to the node that
    contains key. If there is no such
   node, return NULL */

  if (!root) return NULL;

  if (key == root->key) return root;

  if (key < root->key)
       return search(root->left, key);

  return search(root->right, key);
}
```

Finding the minimum element in a BST: findmin()

- Key is to just go left till you cannot go left anymore.

```
function findmin(node):
    if node is NULL:
        return NULL

    while node.left is not NULL:
        node = node.left

    return node
```

time complexity? given the height of the tree h

BST Operations: Removal

- uses a binary search to locate the target item:
  - **starting at the root** it probes down the tree till it finds the target or reaches NULL
- removal of a node must not leave a ‘gap’ in the tree,

Removal in BST - Pseudocode

if the tree is empty return false

Attempt to locate the node containing the target using the binary search
  if the target is not found return false
  else the target is found, so remove its node:

Case 1: if the node has 2 empty subtrees
        replace the link in the parent with null

Case 2: if the node has a left and a right subtree
- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

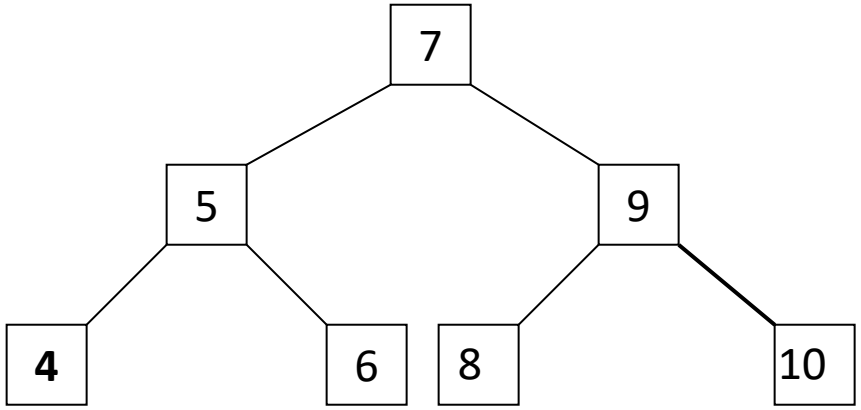
Case 3: if the node has no left child

- link the parent of the node to the right (non-empty) subtree

Case 4: if the node has no right child

- link the parent of the target to the left (non-empty) subtree

Removal in BST: Example



Remove Node 4

- **Node 4** is a **leaf node** (no children).
- To remove a leaf node, we simply set the parent's pointer to NULL.

Remove Node 9

- **Node 9** has **one child** (right child 10).
- To remove a node with one child, we replace it with its child.

Analysis of BST Operations

- The complexity of operations `search`, `insert` and `remove` in BST is $O(h)$, where h is the height.
- $O(\log n)$ when the tree is balanced, i.e., $h = O(\log n)$. The updating operations may cause the tree to become unbalanced.
- The tree can degenerate to a linear shape and the operations will become $O(n)$

Worst Case

```
BST tree = new BST();
for (int i = 8; i >= 1; i--)
   tree.insert (i);
```

Output:

```
>>>> Items in worst order:
        8
       7
      6
     5
    4
   3
  2
 1
```

Balanced BSTs

Prevent the degeneration of the BST :

- A BST can be set up to maintain balance during updating operations (insertions and removals)
- To achieve a worst-case runtime of $O(\log n)$ for searching, inserting and deleting, $\mathrm{h} = O(\log n)$

- Two types we'll look at :
  - AVL trees
  - splay trees
- There are many other types of balanced BSTs: 2-4 trees, Red-Black trees, B-trees

AVL Trees

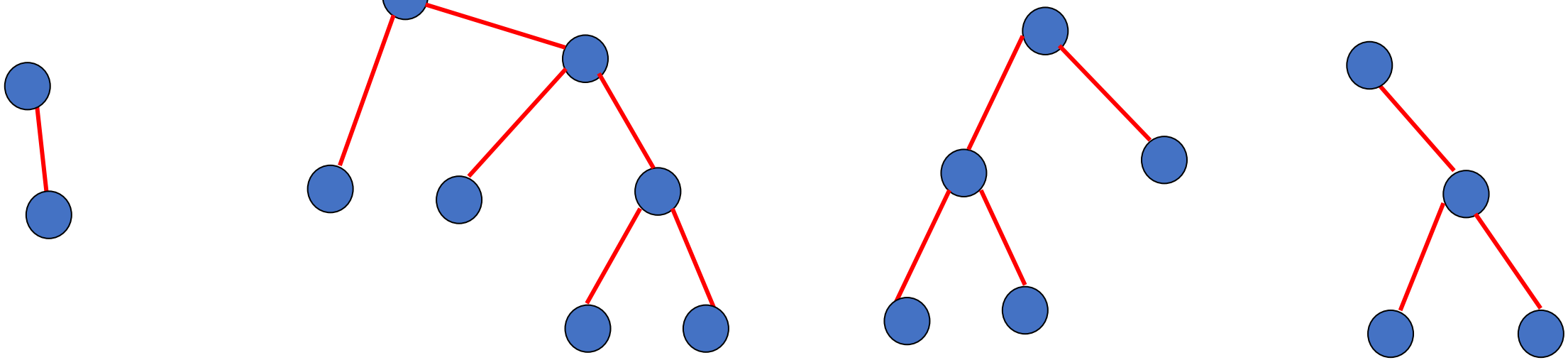
- Invented in 1962 by Russian mathematicians Adelson-Velsky and Landis

An AVL tree is a **self-balancing Binary Search Tree (BST)**.

AVL property:

- The height of the left and right sub-trees of the root differ by at most 1
- The left and right sub-trees are AVL trees
- treat nil tree (empty subtree) as height -1

Which of these are AVL trees, assuming that they are BSTs?



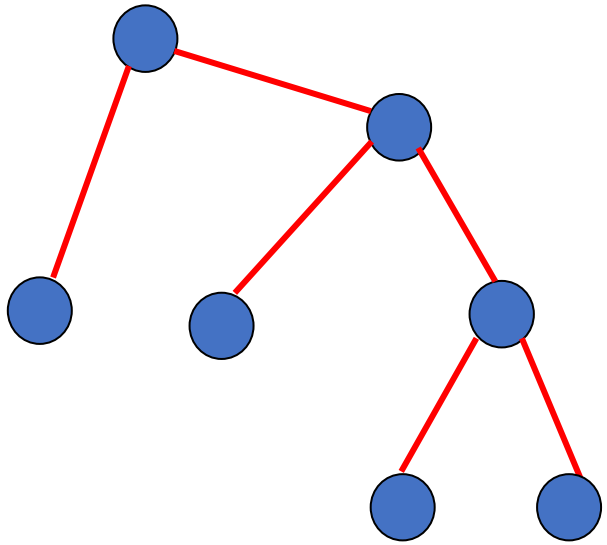
AVL Trees

Which of these are AVL trees, assuming that they are BSTs?

- The **balance factor** of the root node is:
  - One subtree has a height of **0** (the single child node).
  - The other subtree is **nil**, which is treated as having a height of **-1**.
  - So, the balance factor is $0 - (-1) = 1$, which is within the acceptable range of -1 to 1 for an AVL tree.

AVL Trees

Which of these are AVL trees, assuming that they are BSTs?



- The left child of this node has height **0** (a leaf node), and the right child has height **2** (from the right branch).

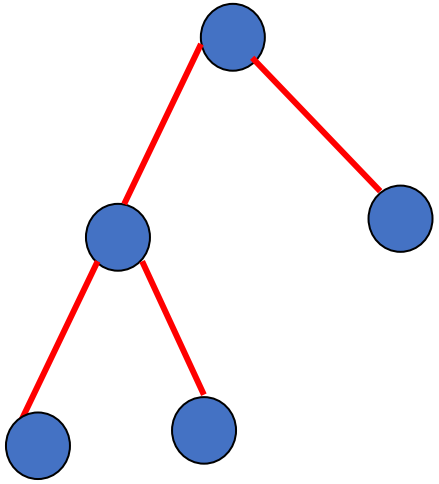
The balance factor for this node is:

$$0-2=-2$$

This balance factor exceeds the allowed range (-1 to 1), so the tree is **unbalanced** and thus **not an AVL tree**.

AVL Trees

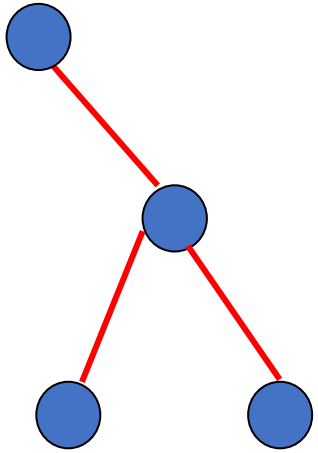
Which of these are AVL trees, assuming that they are BSTs?



The left child of the root has a left subtree with a height of **1** and a right subtree with a height of **0**, resulting in a balance factor of **1 - 0 = 1**, which is also valid.
Thus, this tree does meet the AVL property

AVL Trees

Which of these are AVL trees, assuming that they are BSTs?

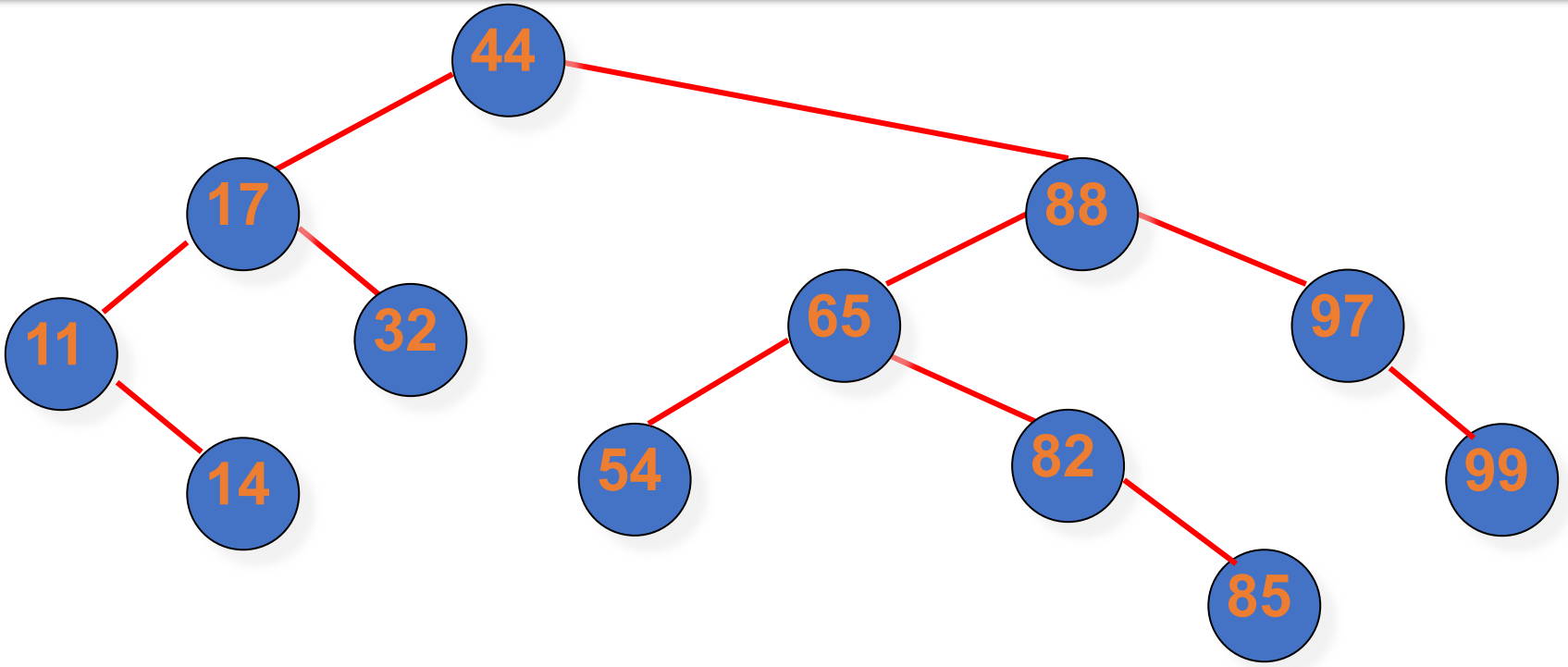


- The **root node** has:
  - A **left subtree** that is **empty** (nil), which is treated as having a height of **-1**.
  - A **right subtree** with height **1** (since it has two levels).
- The balance factor for the root is:

$$(-1) - 1 = -2$$

which is **outside the valid AVL range** of -1 to 1.

Valid AVL Tree



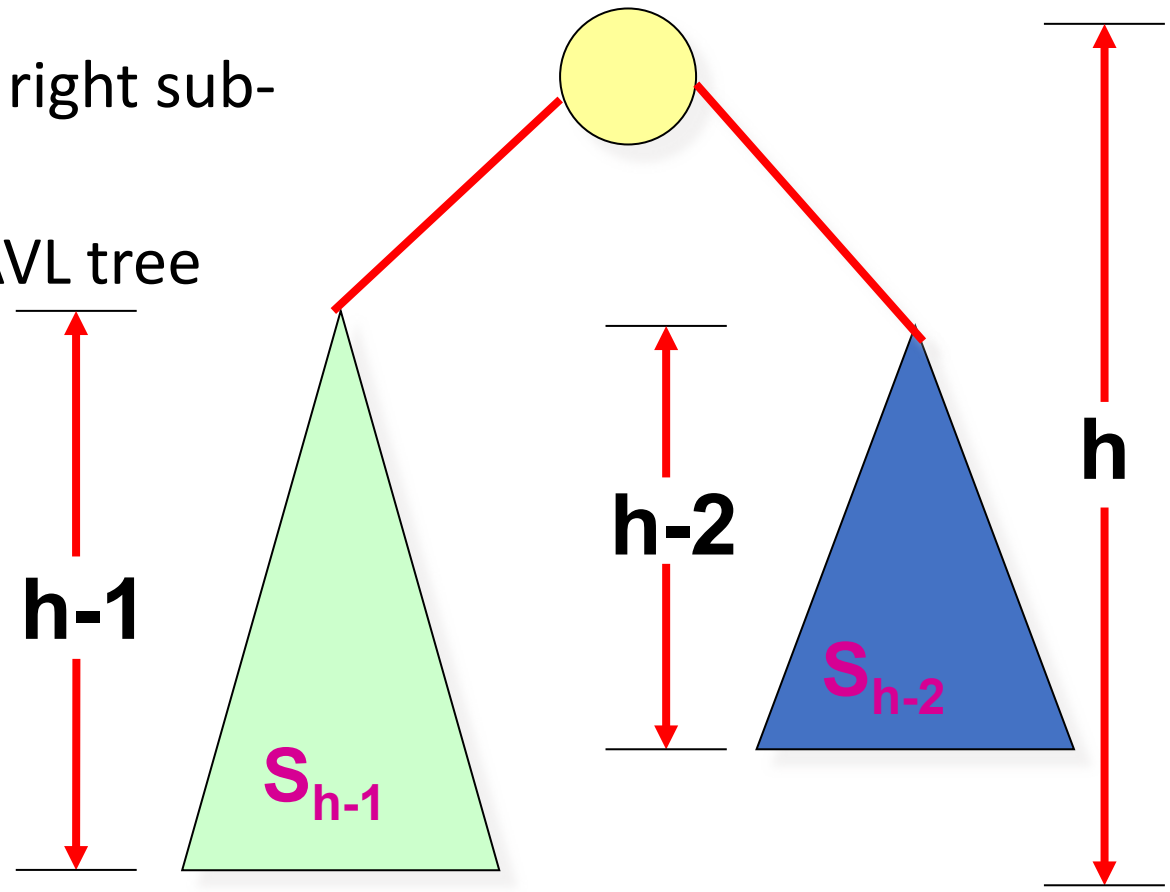
Note: it is not a requirement that all leaves be on the same or adjacent level

AVL Trees: Balanced

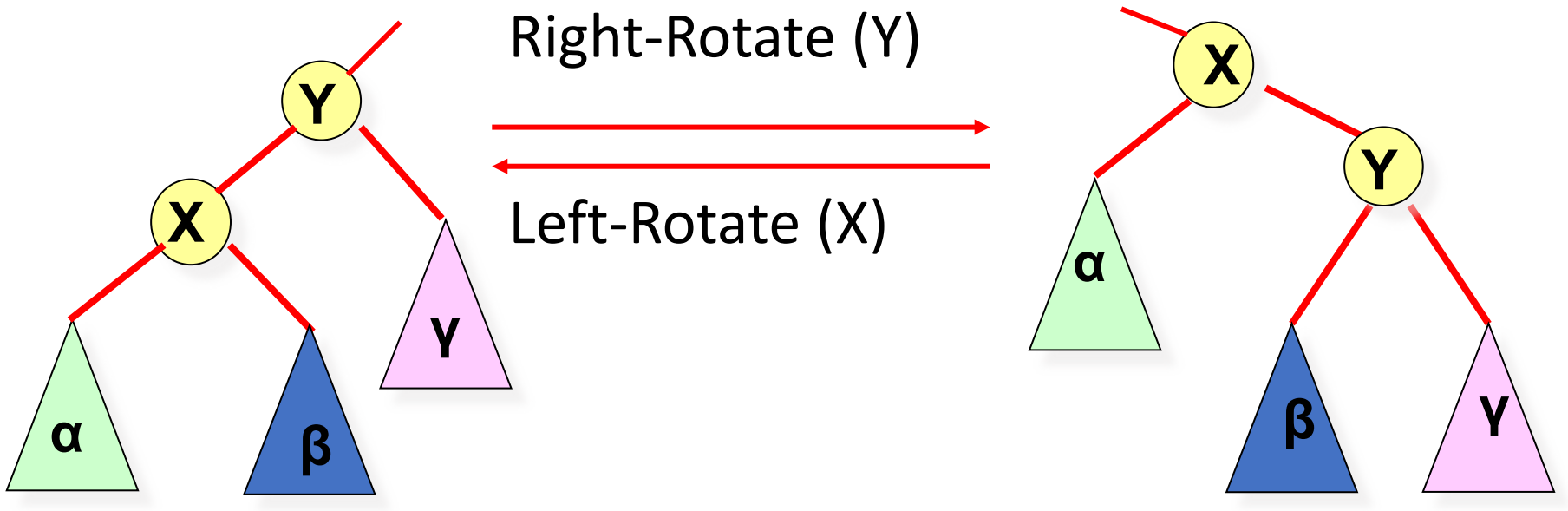
- Aim to get a tight upper bound for h
- Worst when the height of the left and right sub-trees of every node differs by 1
- let $N_h = (min.)\#$ nodes in height-h AVL tree

- $N_h = N_{h-1} + N_{h-2} + 1$
- $> 2N_{h-2}$
- $N_h > 2^{h/2}$
- $h < 2 \log N_h$

the number of nodes grows rapidly as the height increases.



Rotations



Rotations maintain the ordering property of BSTs.

A rotation is an O(1) operation

A **rotation** is a tree restructuring operation that maintains the ordering property of a BST while adjusting the tree's shape to maintain balance, especially in self-balancing trees like AVL trees.

Rotations

Restoring the Balance Factor:

- In an AVL tree, the **balance factor** of any node is the difference in height between its left and right subtrees. The balance factor should be **-1, 0, or 1** for every node.
- After an insertion or deletion, if the balance factor of a node becomes less than -1 or greater than 1, the tree is **unbalanced**.
- **Rotations** (left or right) are performed to bring the tree back into balance by adjusting the structure of the tree while keeping the order of elements intact.

A **rotation** is a tree restructuring operation that maintains the ordering property of a BST while adjusting the tree's shape to maintain balance, especially in self-balancing trees like AVL trees.

AVL Insert

1. insert as in simple BST

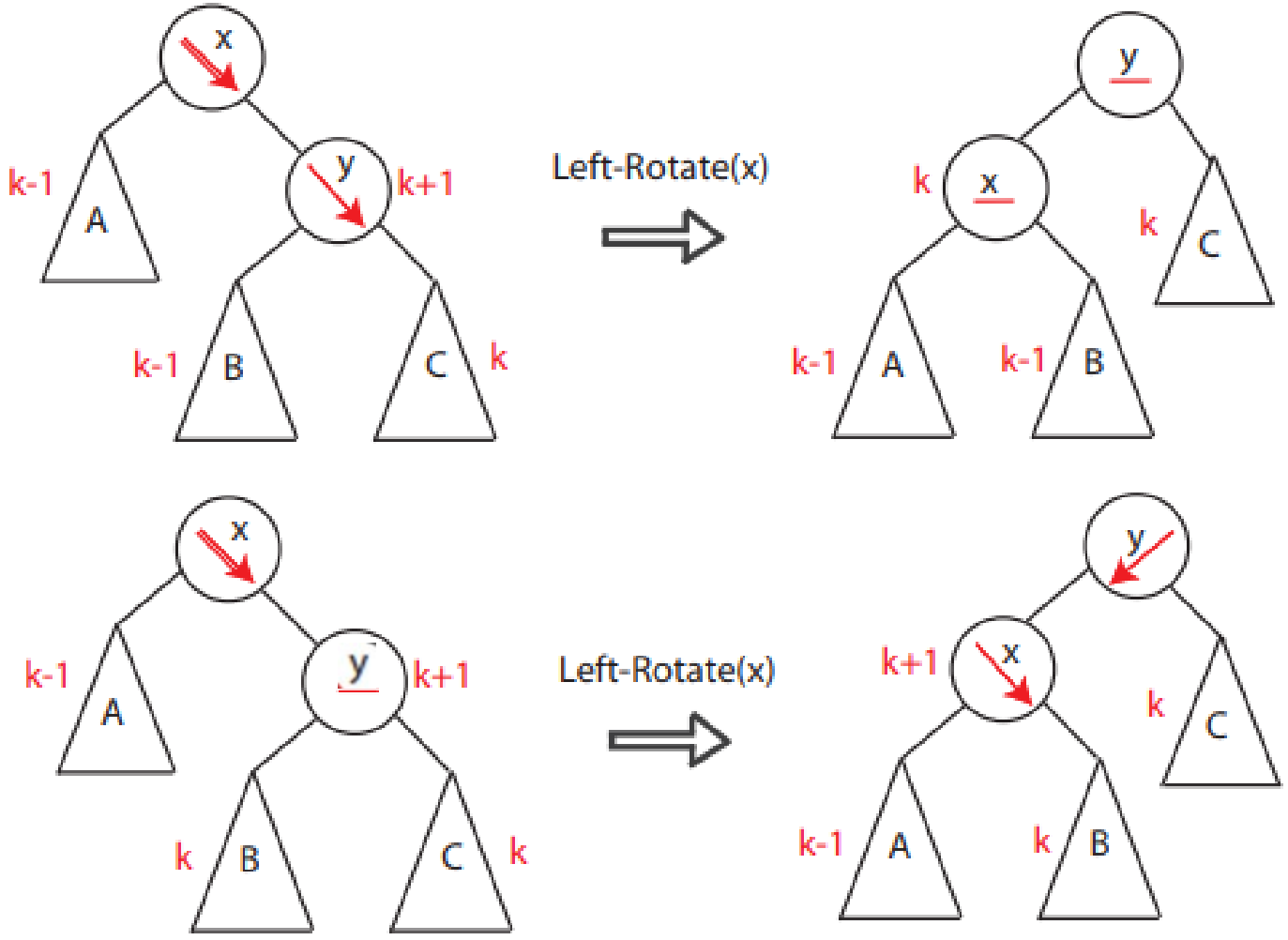
2. work your way up tree from the node inserted, restoring AVL property (and updating heights as you go).

- suppose x is the lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- two cases:
  - Case 1: x's right child is right-heavy or balanced → single rotation
  - Case 2: else (x's right child is left heavy)→ double rotations
- then continue up to x's parent, grandparent, great-grandparent . . .

Case 1

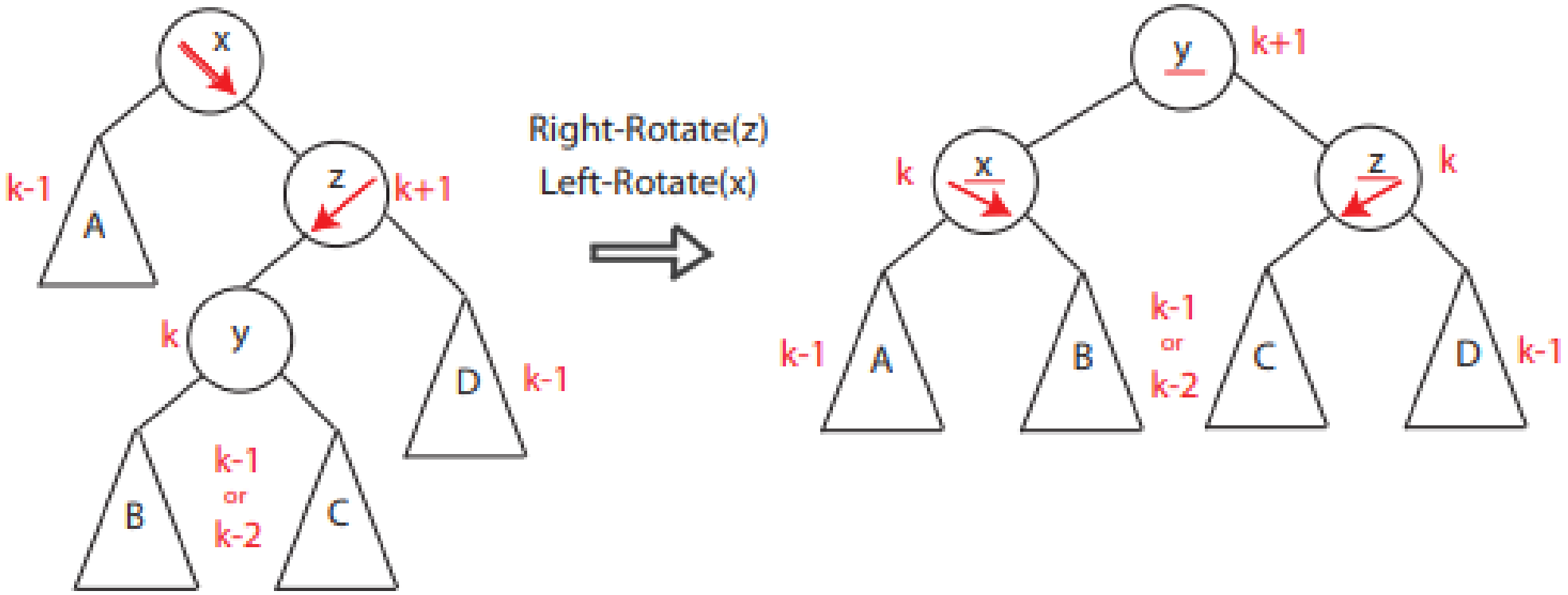
Left Rotation (Single Rotation):

A **left rotation** is applied to **x** to restore the balance. This operation shifts **y** up to become the new root of the subtree, while **x** becomes the left child of **y**.

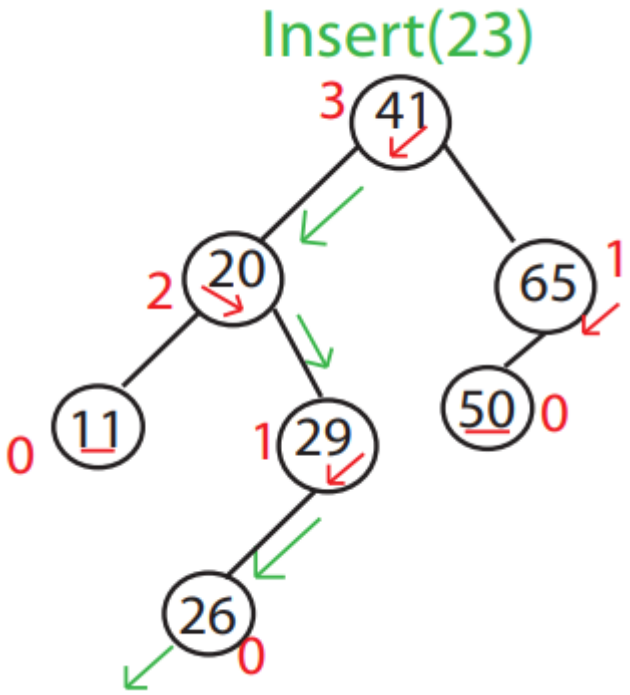


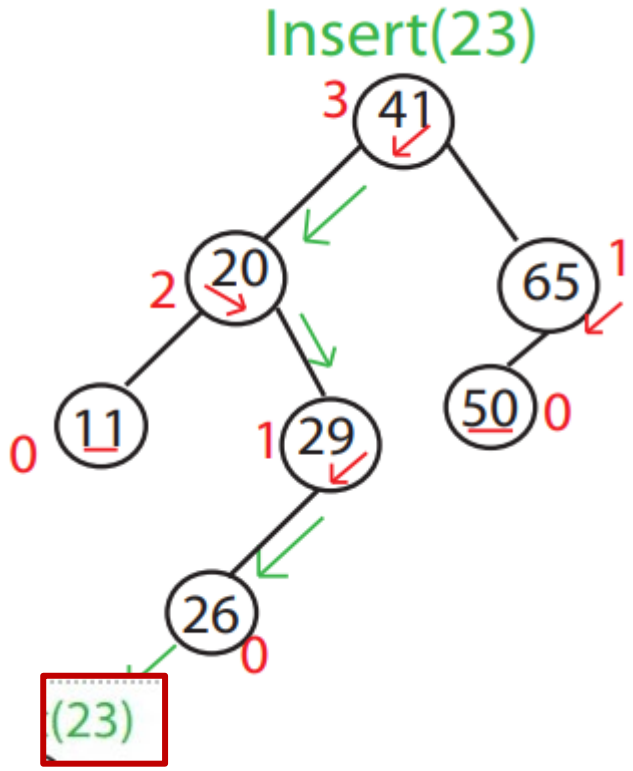
Case 2

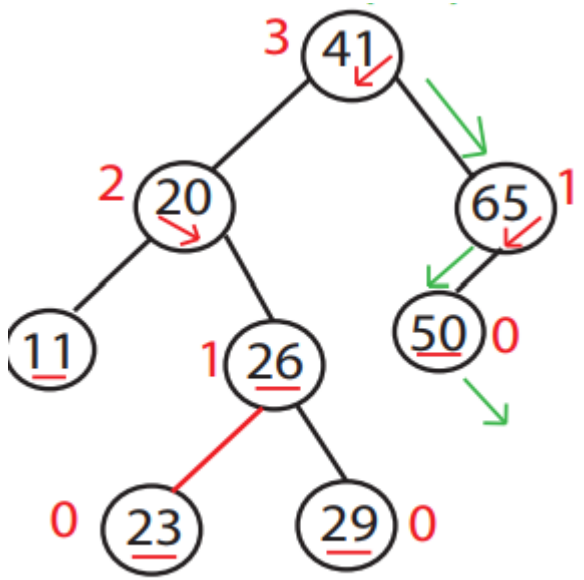
ouble rotation (a combination of a right and left rotation) to restore balance



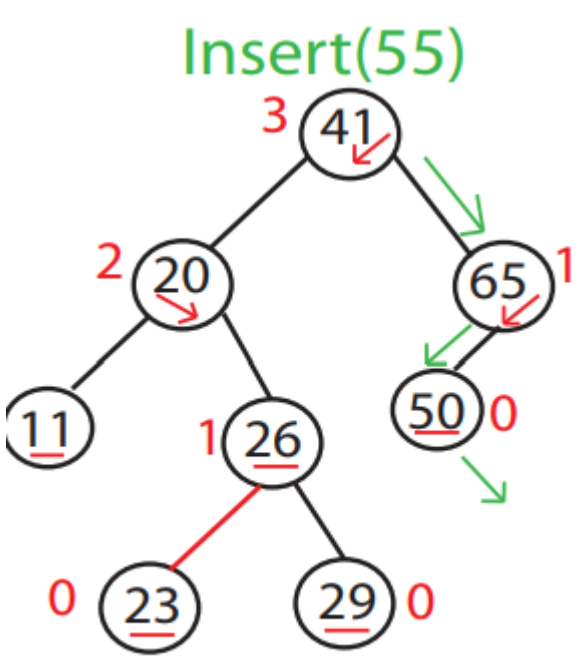
Example

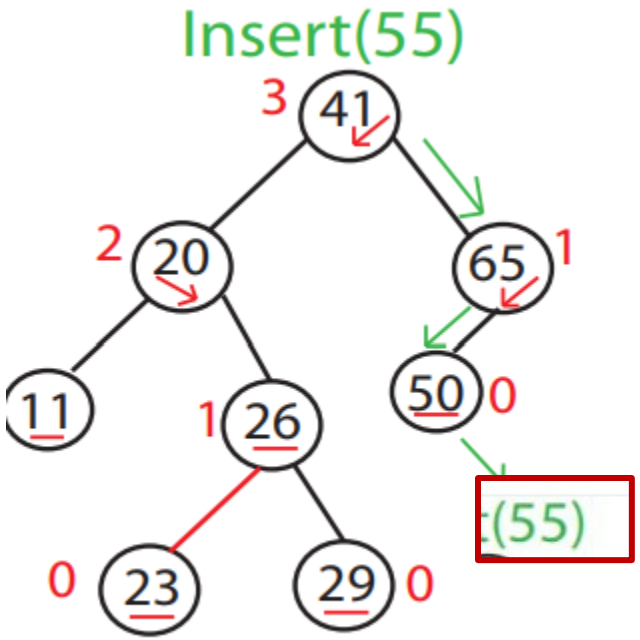

Insert(23)
3
41
2
20
65
1
0
11
1
29
50
0
26
0

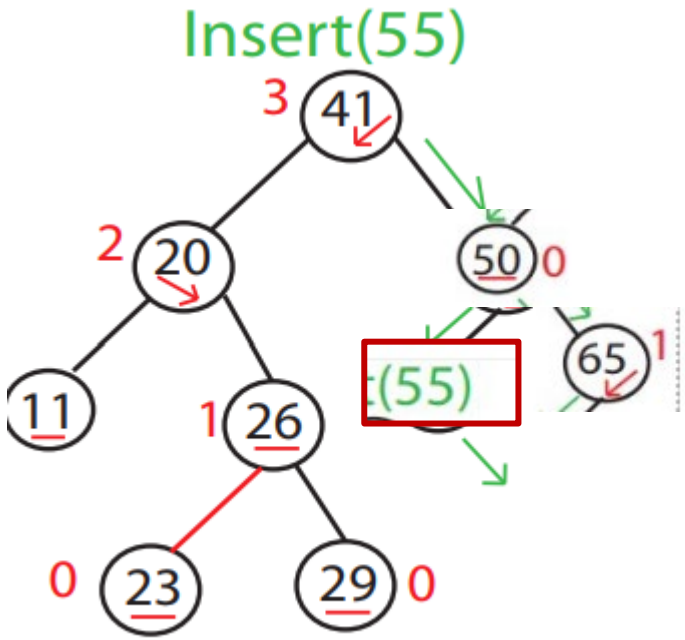

Insert(23)
3
41
2
20
65
1
0
11
1
29
50
0
26
0
(23)


3
41
2
20
65
1
11
1
26
50
0
0
23
29
0

Example



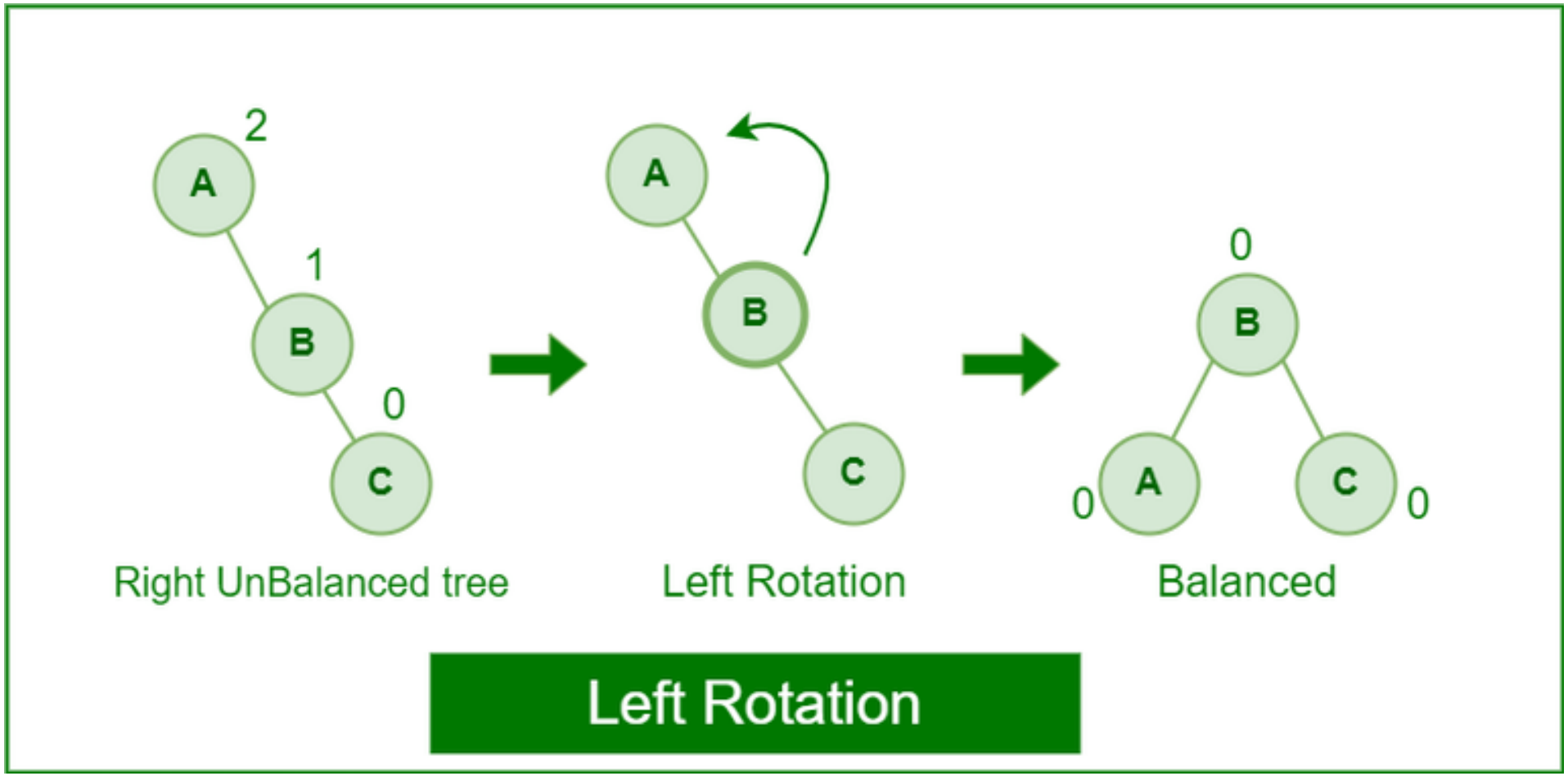




https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

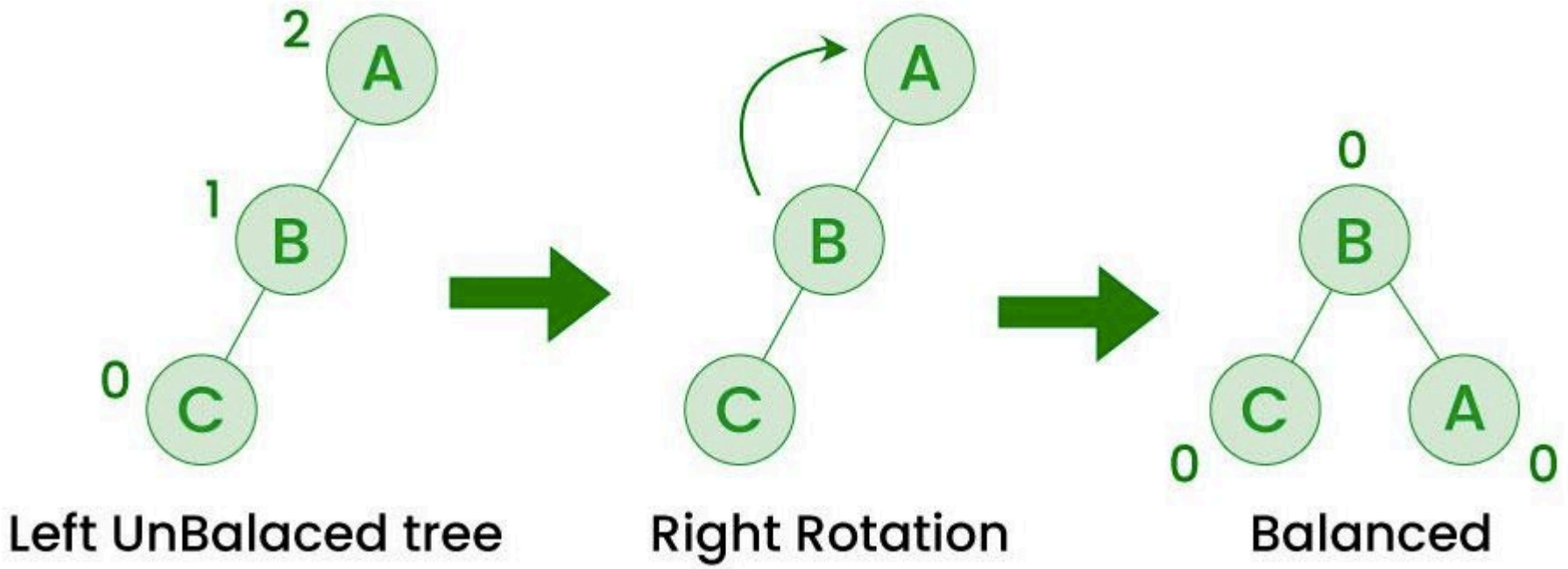
Left Rotation:

When a node is
he right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



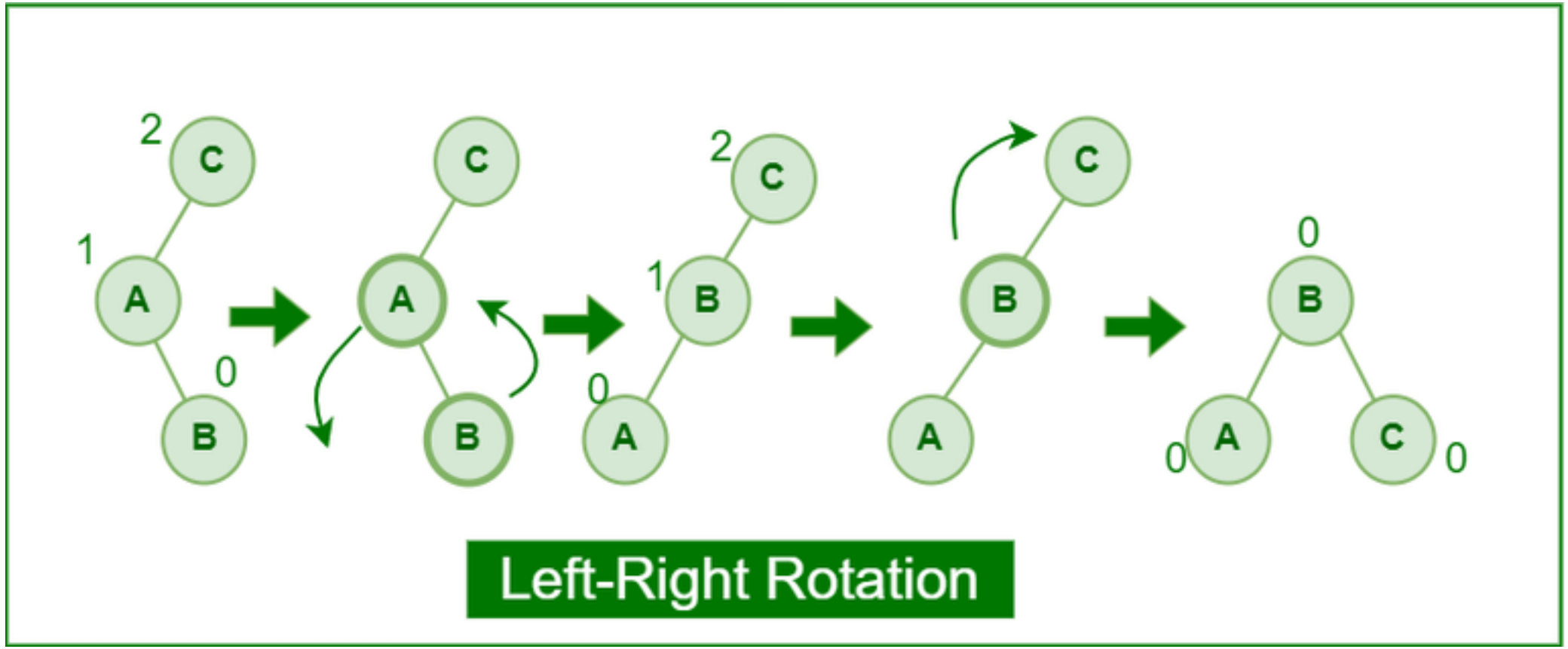
Right Rotation:

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



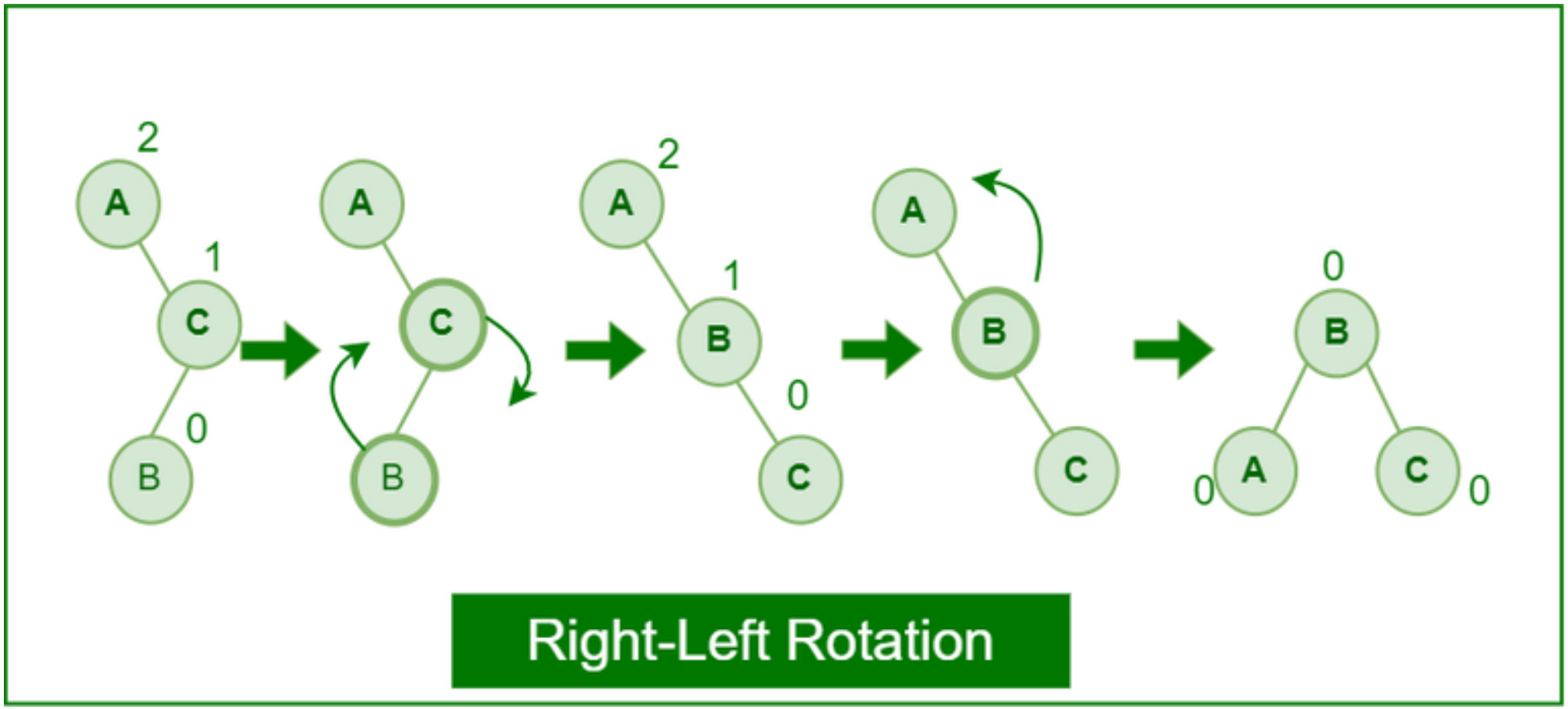
Left-Right Rotation:

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



Balancing AVL tree

```
void balance( AvlNode * & t ) {
        if( t == NIL ) return;
        if( height( t->left ) - height( t->right ) > 1)
                if( height( t->left->left ) >= height( t->left->right ) )
                        rightRotate( t );
                else
                        leftRightRotate( t );
        else if( height( t->right ) - height( t->left ) > 1)
                if( height( t->right->right ) >= height( t->right->left ) )
                        leftRotate( t );
                else
                         rightLeftRotate( t );
        t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

Balancing AVL tree

```
void rightRotate( AvlNode * & k2 ) {
        AvlNode *k1 = k2->left;
        k2->left = k1->right;
        k1->right = k2;
        k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
        k1->height = max( height( k1->left ), k2->height ) + 1;
        k2 = k1;
}
```

Comments

- In general, process may need several rotations before done with an Insert. $O(\log n)$ rotations may be required
- Running time for inserting each item into AVL tree?

AVL Delete

Deletion is similar — harder but possible.

1. Perform standard BST delete

2. Work your way up tree from the node deleted, restoring AVL property (and updating heights as you go).

- suppose x is the lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- two cases:
  - Case 1: x's right child is right-heavy or balanced → single rotation
  - Case 2: else (x's right child is left heavy)→ double rotations
- then continue up to x's parent, grandparent, great-grandparent . . .

Advantages/Disadvantage of AVL Trees

- Advantages
  - $O(\log n)$ worst-case searches, insertions and deletions
- Disadvantages
  - Complicated Implementation
    - Must keep balancing info in each node
    - To find node to balance, must go back up in the tree: easy if pointer to parent, otherwise difficult
    - Deletion complicated by numerous potential rotations