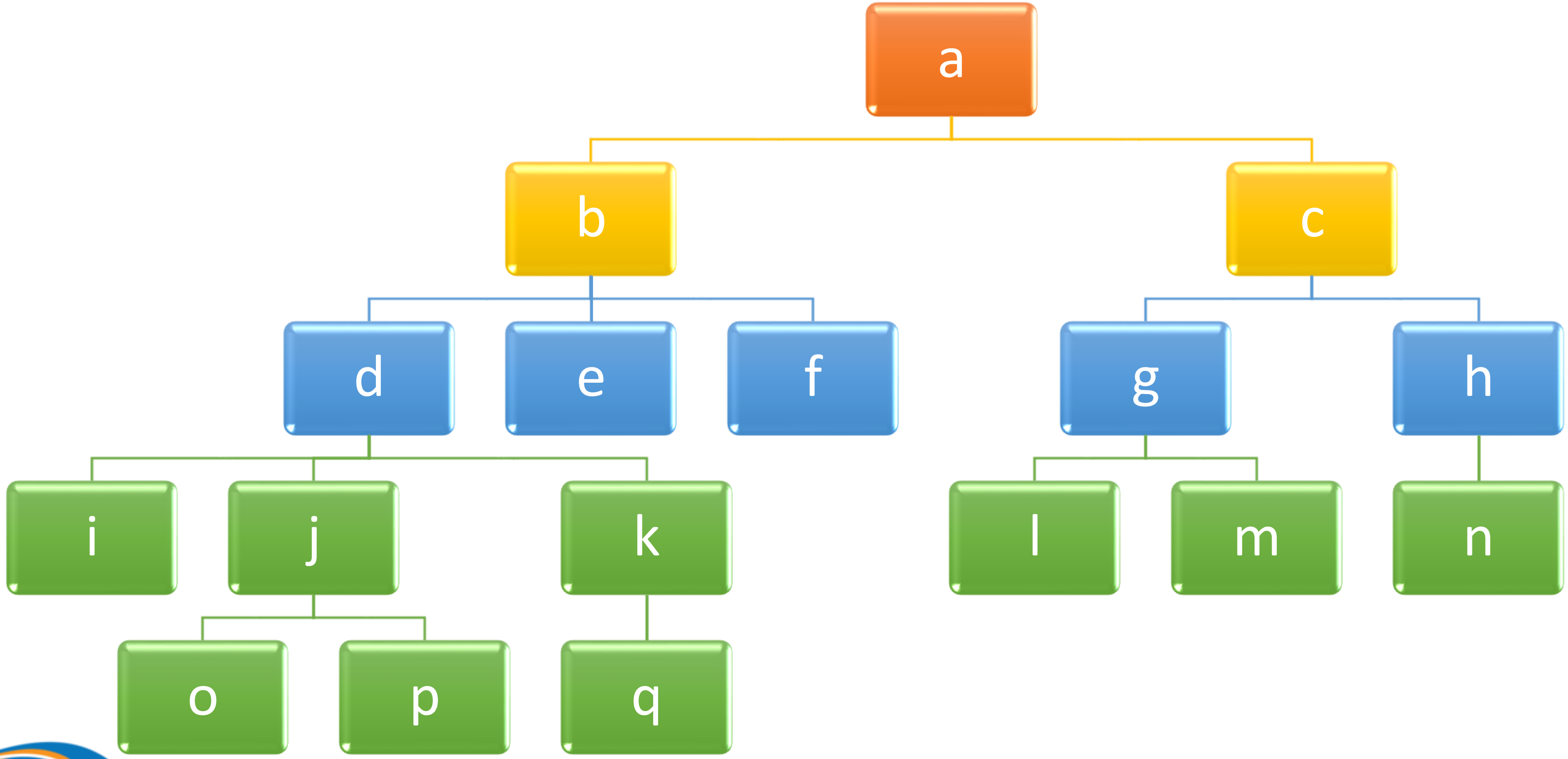
Tree Structures

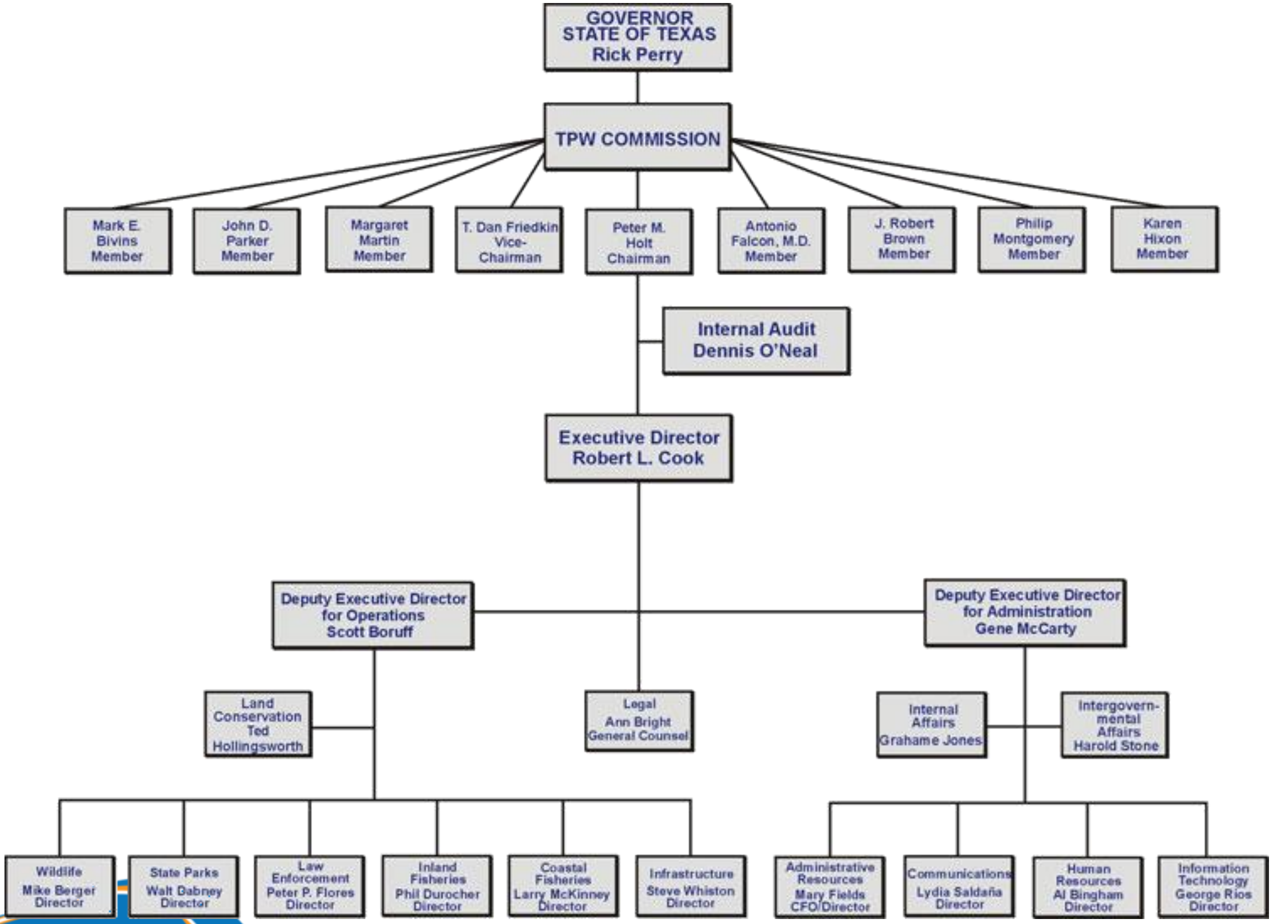
Contents

- Terminologies
- Tree traversals
- Tree representation
- Binary tree
- Binary search tree
- AVL tree

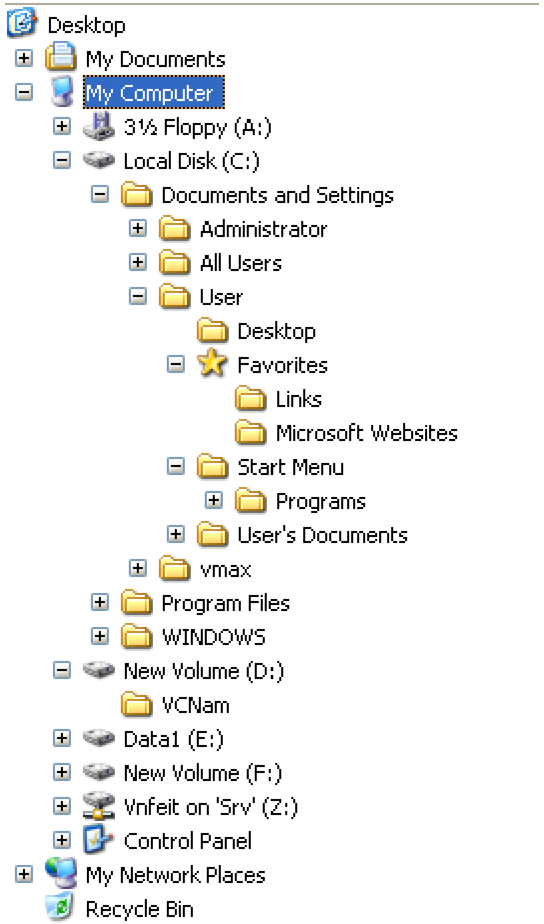
Some Examples



Some Examples



Organizational chart

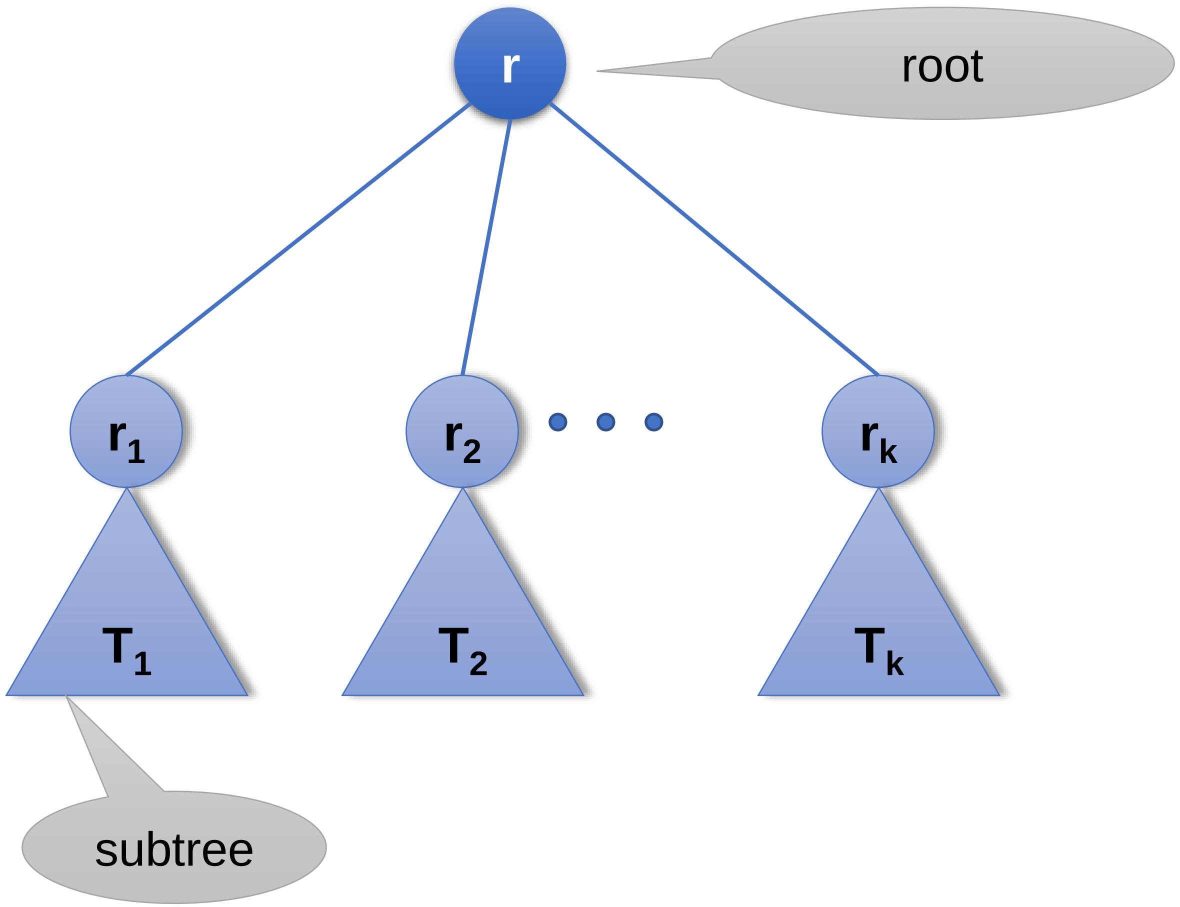


Directory tree

Trees

- Used to represent **relationships**
  - Parent-child between nodes in tree
  - Between ancestor and descendant
  - Links between **nodes** are called **edges**

Trees



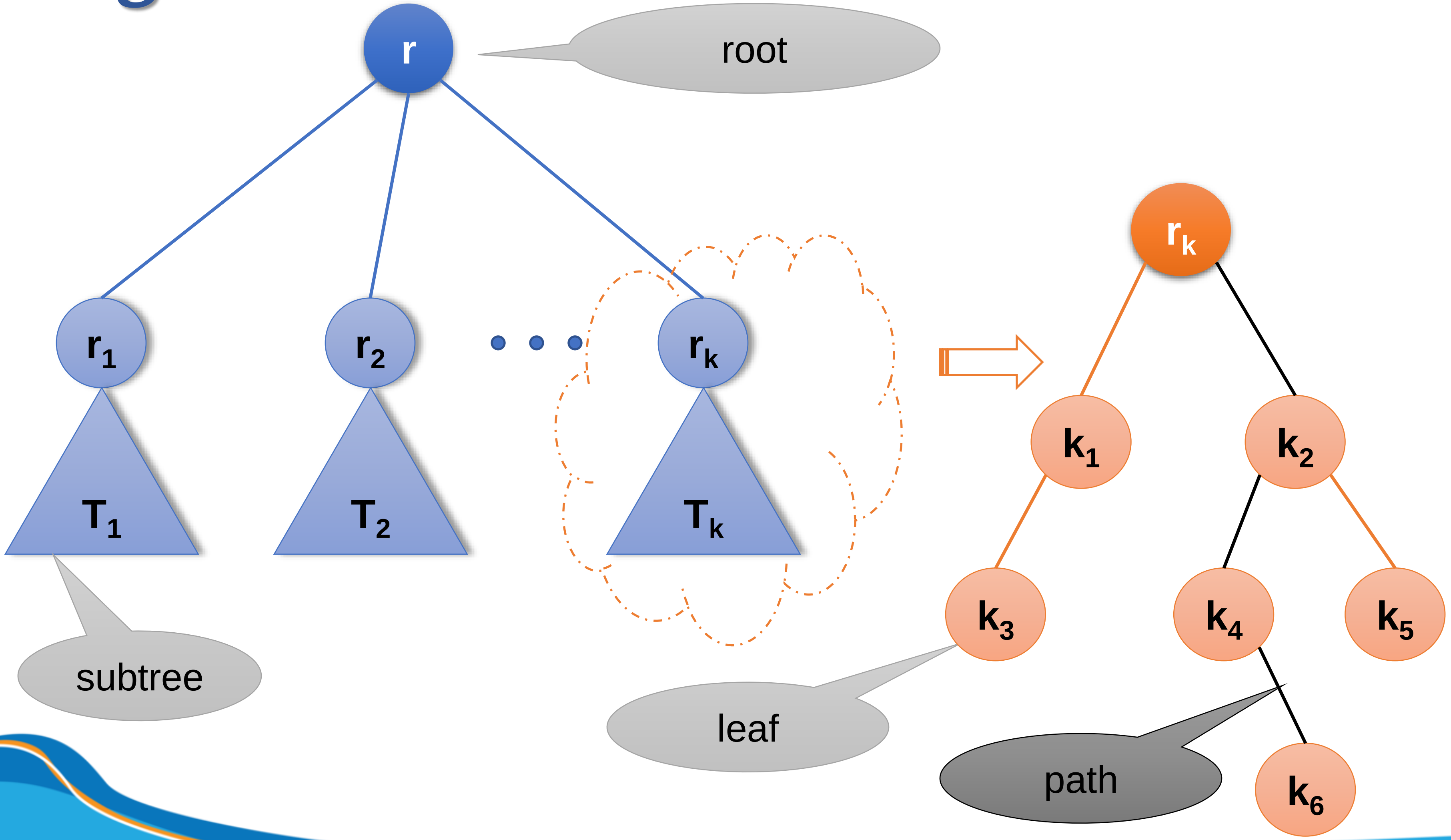
Terminologies

- node: an item/element in a tree.
- parent (of node *n*): The node **directly above** node *n* in the tree.
- child (of node *n*): The node **directly below** node *n* in the tree.
- root: The only node in the tree with no parent.
- leaf: A node with no children.
- path: A sequence of nodes and edges connecting a node with the nodes below it.

Terminologies

- siblings: nodes with the same parent.
- ancestor (of node *n*): nodes on the path from the root to node *n*.
- descendant (of node *n*): nodes on the path from node *n* to a leaf.
- subtree (of node *n*): A tree that consists of a child node and the child's descendants.

Terminologies



Terminologies

- degree/order
  - Order of node *n*: number of children of node *n*.
  - Order of a tree: the maximum order of nodes in that tree.
- depth/level (of node *n*)
  - Level of root is 1.
  - Level of node *n* is level of its parent plus 1.

```
if node n is root
    level(n) = 1
else
    level(n) = 1 + level(parent(n))
```

Terminologies

- Height of tree: number of nodes in the longest path from the root to a leaf.

- Height of a tree T in terms of level of its nodes
  - If T is empty, its height is 0.
  - If T is not empty, its height is equal to the maximum level of its nodes.

Terminologies

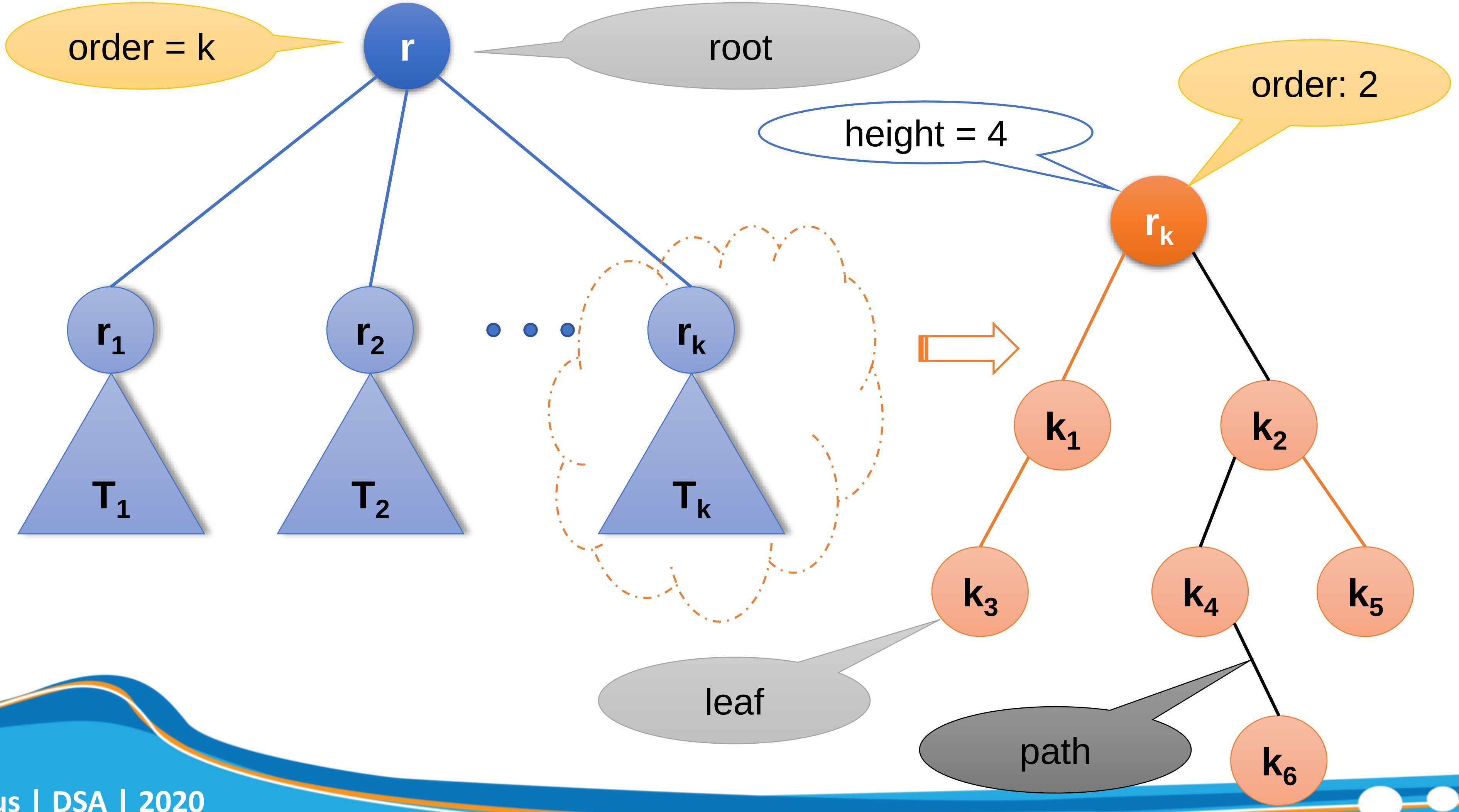
- Height of tree *T*:

```
if T is empty
    height(T) = 0
else
    height(T) = 1 + max{height(Ti)}, Ti is a subtree of T
```

- Height of tree *T*:

```
if T is empty
    height(T) = 0
else
    height (T) = max{level(Ni)}, Ni∈T
```

Terminologies



Type of trees

- Binary tree
  - A node has maximum of two children
  - Children of a node are left and right subtrees that are also binary trees
- n-ary tree
  - A node has maximum of n children
  - Children of a node are n-ary trees

Traversals

Traversal

- Visit each node in a tree **exactly once**.

- Many operations need using tree traversals.

- The basic tree traversals:
  - Pre-order
  - In-order
  - Post-order

Pre-order Traversal

```
PreOrder(root)
{
   if root is empty
      Do_nothing;
   Visit root; //Print, Add, ...
   //Traverse every Child_i.
   PreOrder(Child_0);
   PreOrder(Child_1);
   …
   PreOrder(Child_k-1);
}
```

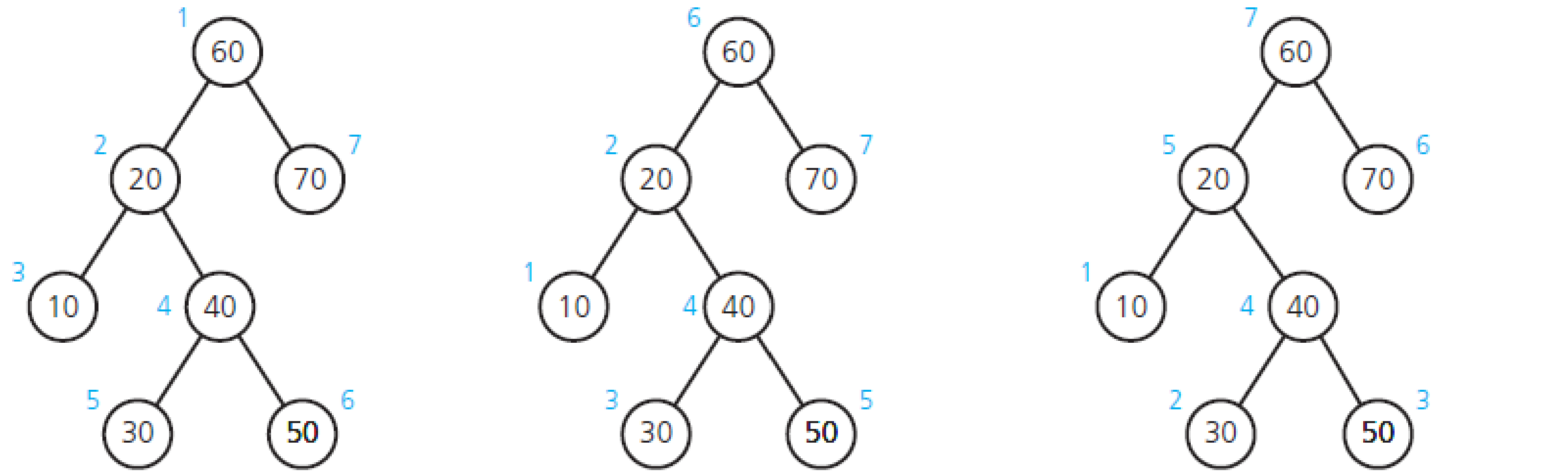
Post-order Traversal

```
PostOrder(root)
{
  if root is empty
     Do_nothing;
  //Traverse every Child_i
  PostOrder(Child_0);
  PostOrder(Child_1);
  …
  PostOrder(Child_k-1);
  Visit at root; //Print, Add, ...
}
```

In-order Traversal

```
InOrder(root)
{
   if root is empty
       Do_nothing;
   //Traverse the child at the first position
   InOrder(Child0);
   Visit at root;
   //Traverse other children
   InOrder(Child1);
   InOrder(Child2);
   …
   InOrder(Childk-1);
}
```

Traversals



(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

Examples

Pre-order

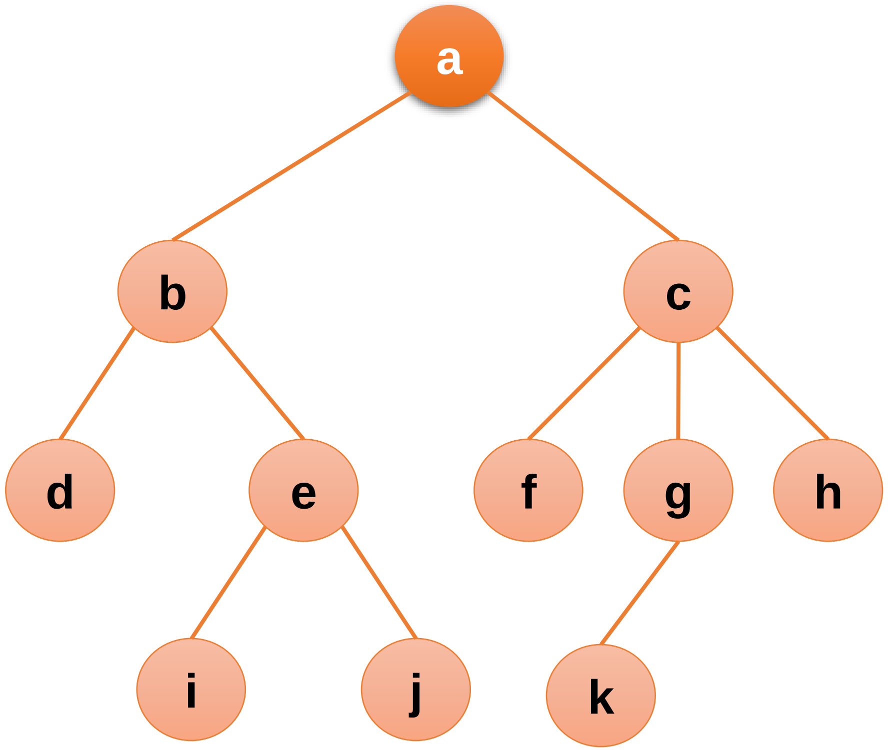
- *a b d e i j c f g k h*

In-order

- d b i e j a f c k g h

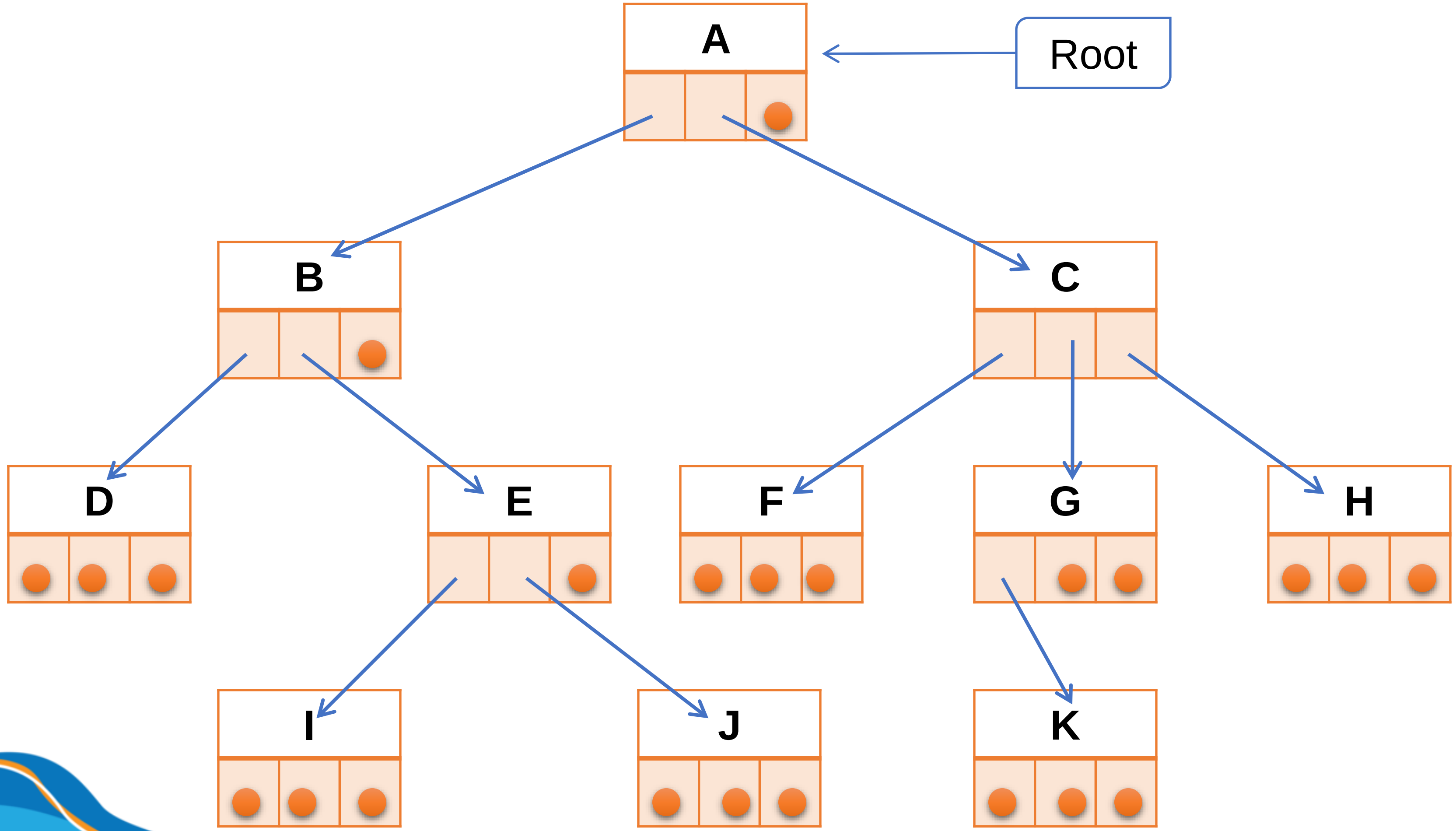
Post-order

- *d i j e b f k g h c a*

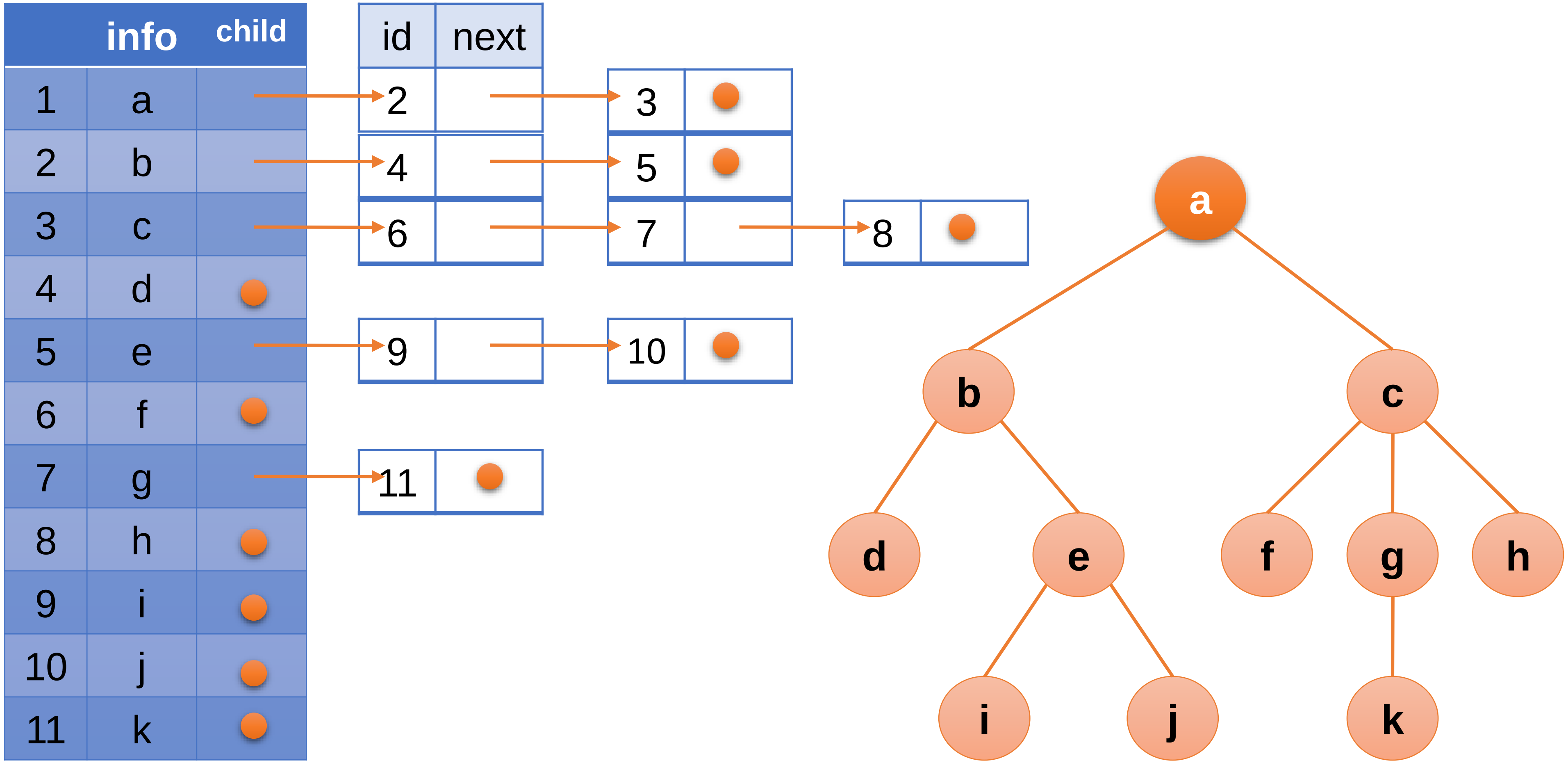


Tree Representation

Tree Representation

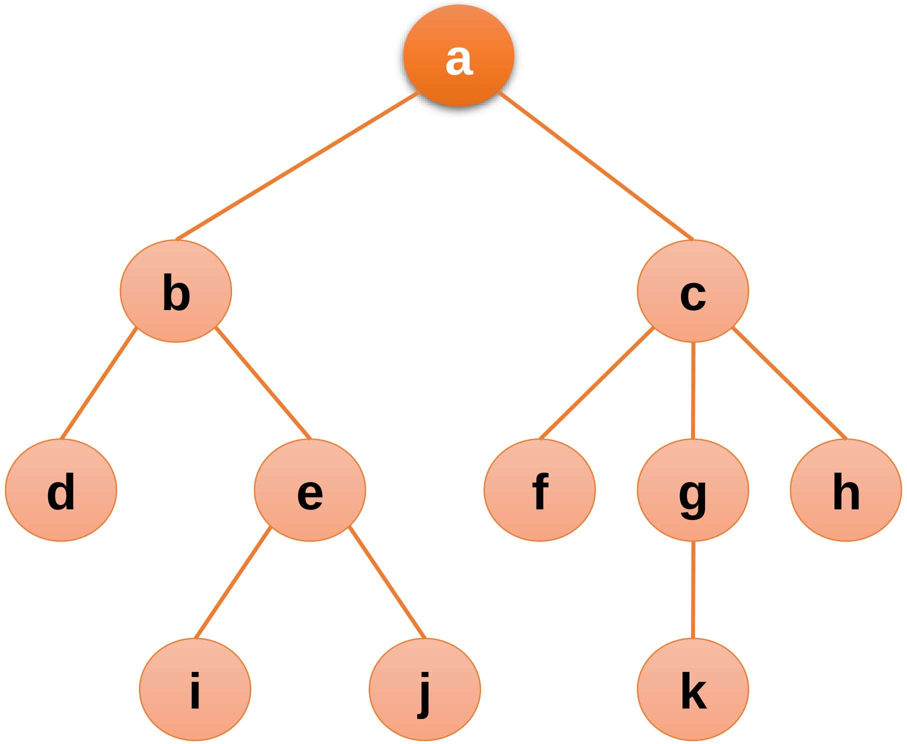


Tree Representation

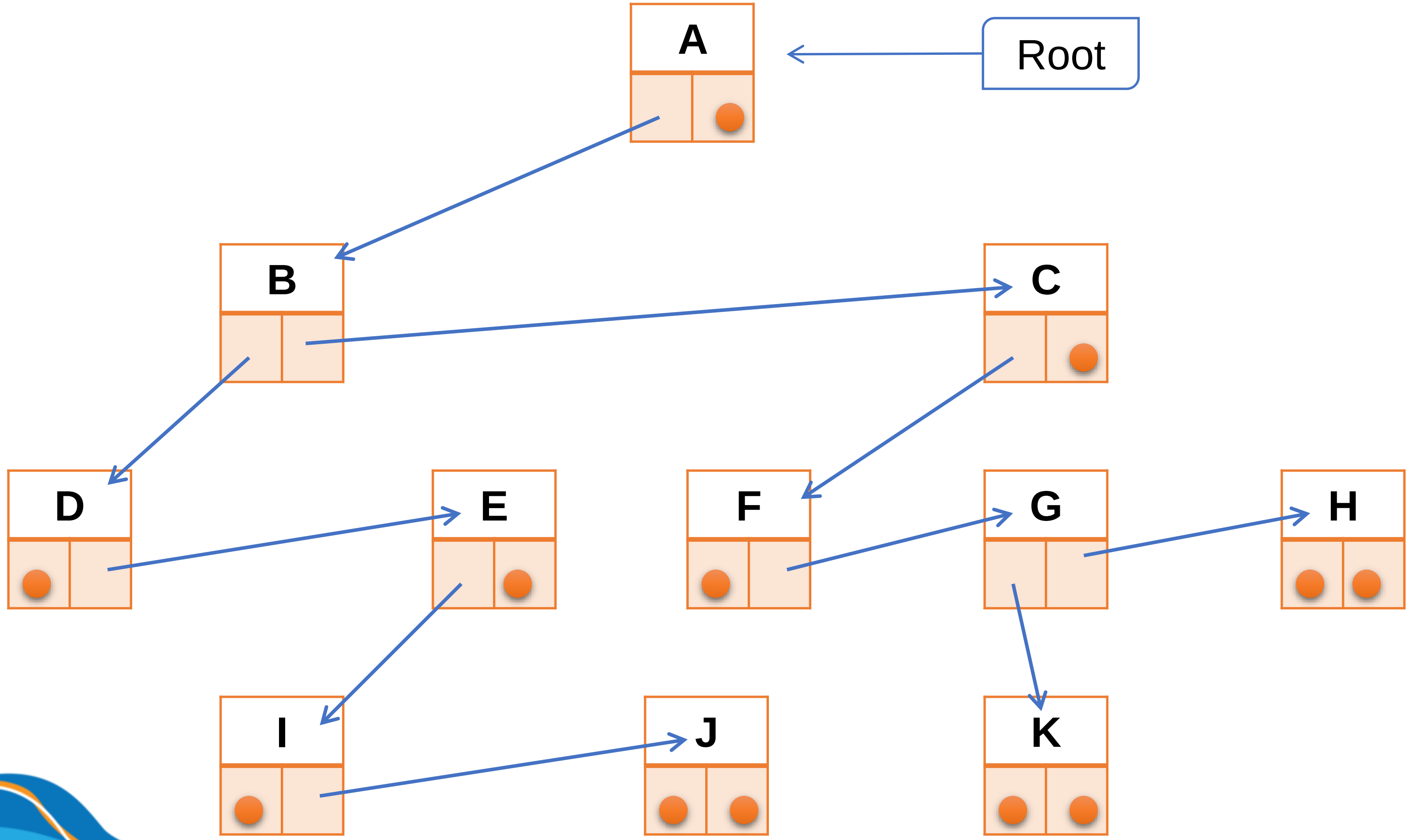


Tree Representation

	Info	Eldest Child	Next Sibling
1	a	2	0
2	b	4	3
3	c	6	0
4	d	0	5
5	e	9	0
6	f	0	7
7	g	11	8
8	h	0	0
9	i	0	10
10	j	0	0
11	k	0	0

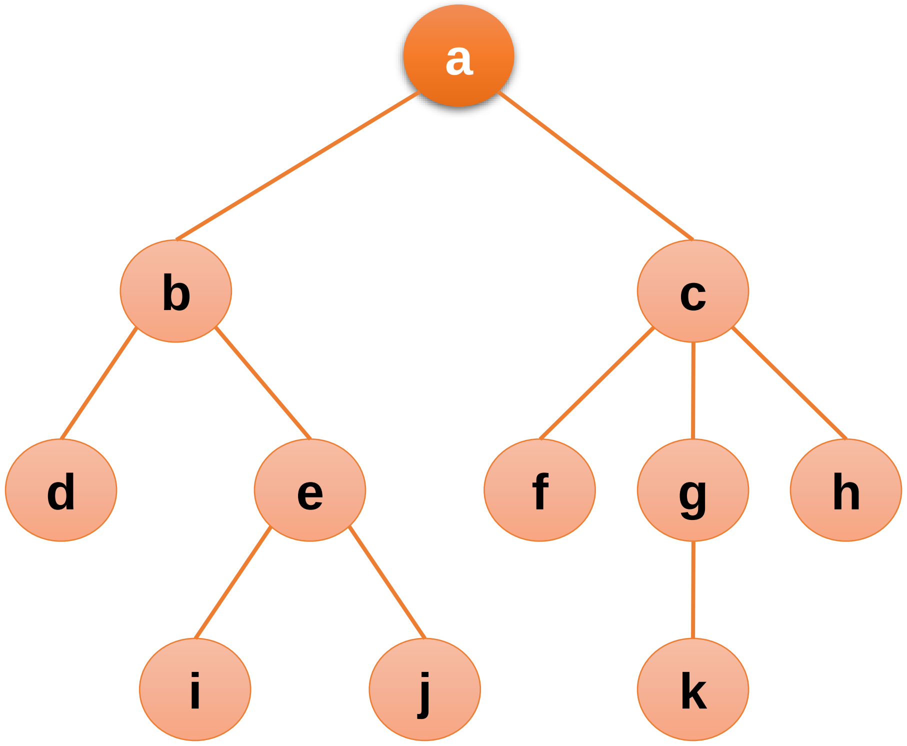


Tree Representation



Tree Representation

	Info	Parent
1	a	0
2	b	1
3	c	1
4	d	2
5	e	2
6	f	3
7	g	3
8	h	3
9	i	5
10	j	5
11	k	7

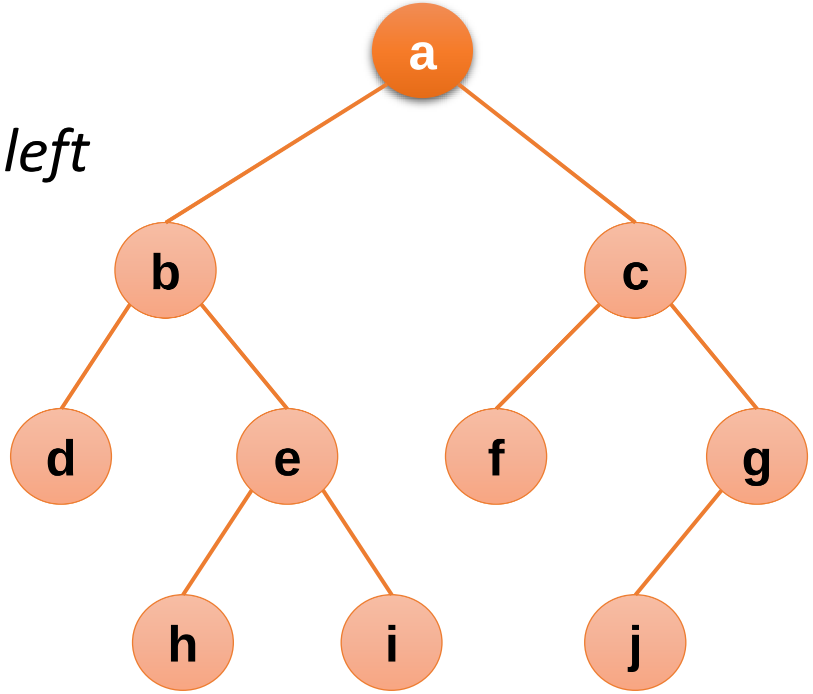


Binary Tree

Binary Tree

- Set T of nodes that is either empty or partitioned into disjoint subsets.
  - Single node *r*, the root
  - Two possibly empty sets that are binary trees, called *left* and *right* subtrees of *r*.

- Other definition: A rooted binary tree has a root node and every node has at most two children.



Type of Binary Trees

- Complete binary tree
- Full binary tree
- Perfect binary tree
- Heap

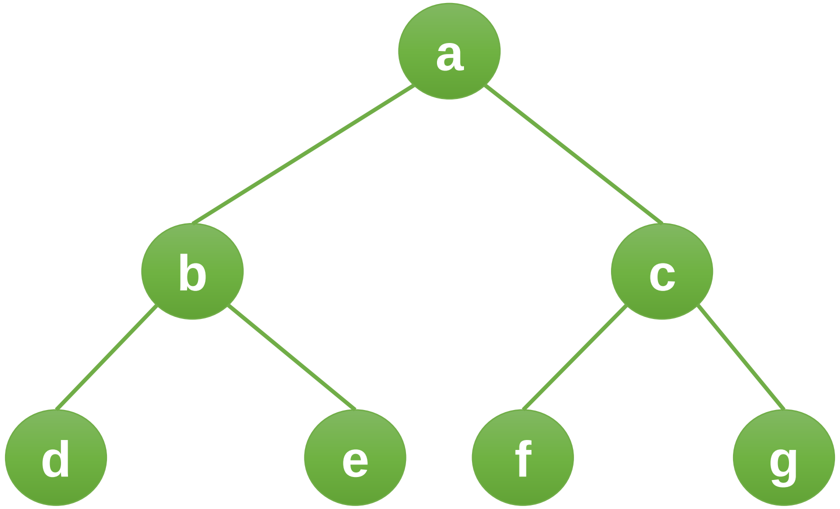
Perfect Binary Tree

- A **perfect binary tree** is a binary tree in which
  - all interior nodes have two children
  - and all leaves have the same depth or same level.

- In a perfect binary tree of height *h*, all nodes that are at a level less than *h* have two children each.

Perfect Binary Tree

- If *T* is empty, *T* is a perfect binary tree of height 0.
- If *T* is not empty and has height $h > 0$, T is a perfect binary tree if its root's subtrees are both perfect binary trees of height $h - 1$.



Complete Binary Tree

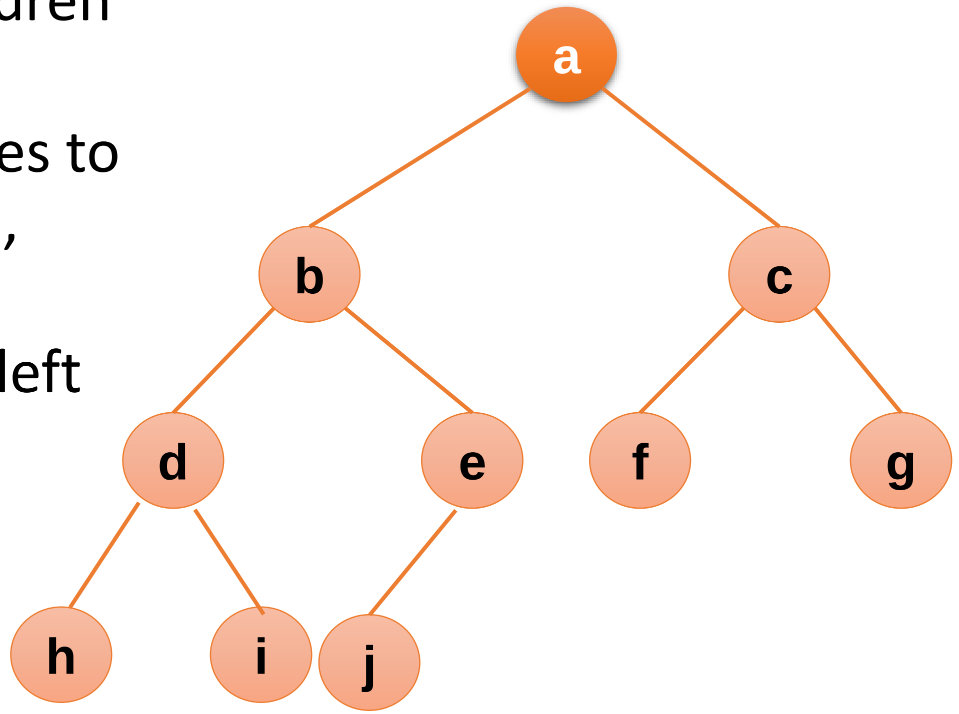
- A **complete binary tree** of height *h* is a binary tree that is **perfect** down to level *h* – 1, with level *h* filled in from left to right.

- In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

- Other definition: A **complete binary tree** is a perfect binary tree whose rightmost leaves (perhaps all) have been removed.

Complete Binary Tree

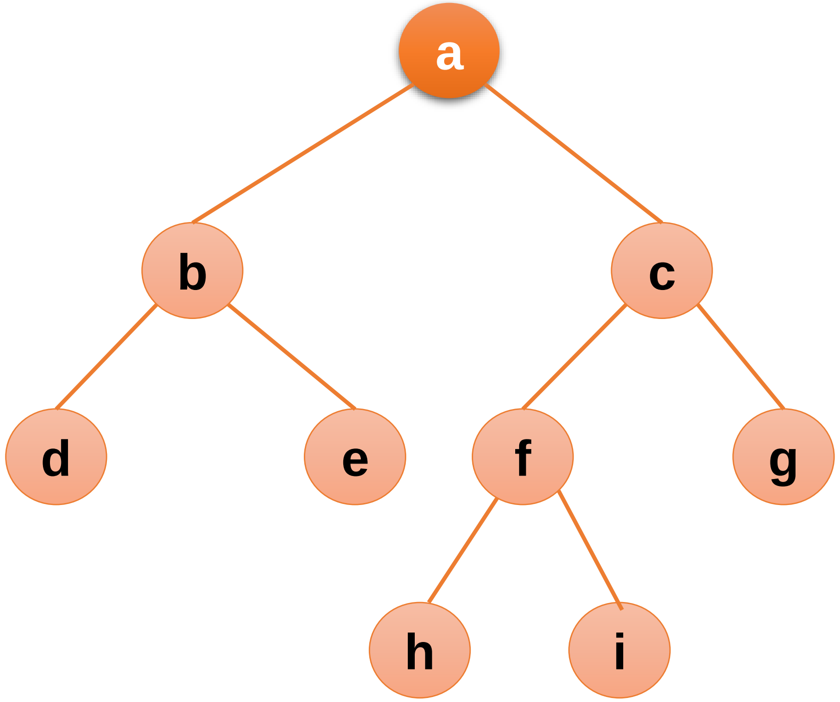
- A binary tree is complete if
  - All nodes at level h – 2 and above have two children each, and
  - When a node at level h – 1 has children, all nodes to its left at the same level have two children each, and
  - When a node at level h – 1 has one child, it is a left child



Full Binary Tree

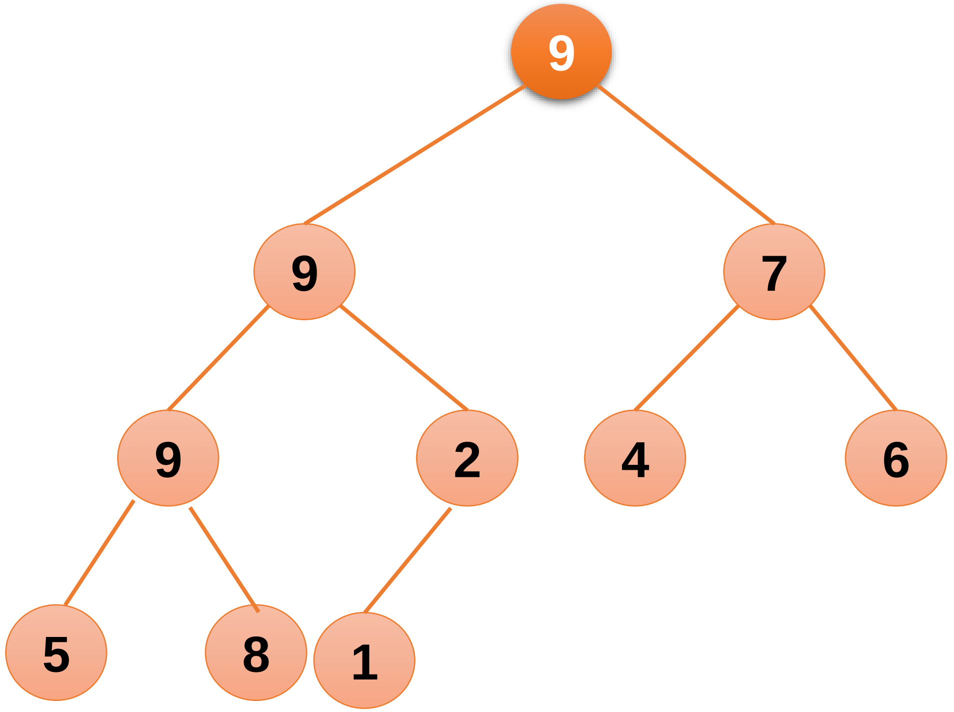
- A full binary tree (sometimes referred to as a **proper binary tree** or **a plane binary tree**) is a binary tree in which ***every node has either 0 or 2 children***.

- A full binary tree is either:
    - A single vertex.
    - A tree whose root node has two subtrees, both of which are full binary trees.



Heap

- A heap is a ***complete binary tree*** that either is empty or
  - Its root
    - (Max-heap): Contains a value greater than or equal to the value in each of its children, and
    - (Min-heap): Contains a value less than or equal to the value in each of its children, and
    - Has heaps as its subtrees



Number of Nodes

- Given a binary tree *T* height of *h*.
- What is the maximum number of nodes?
- What is the minimum number of nodes?

Height of Tree

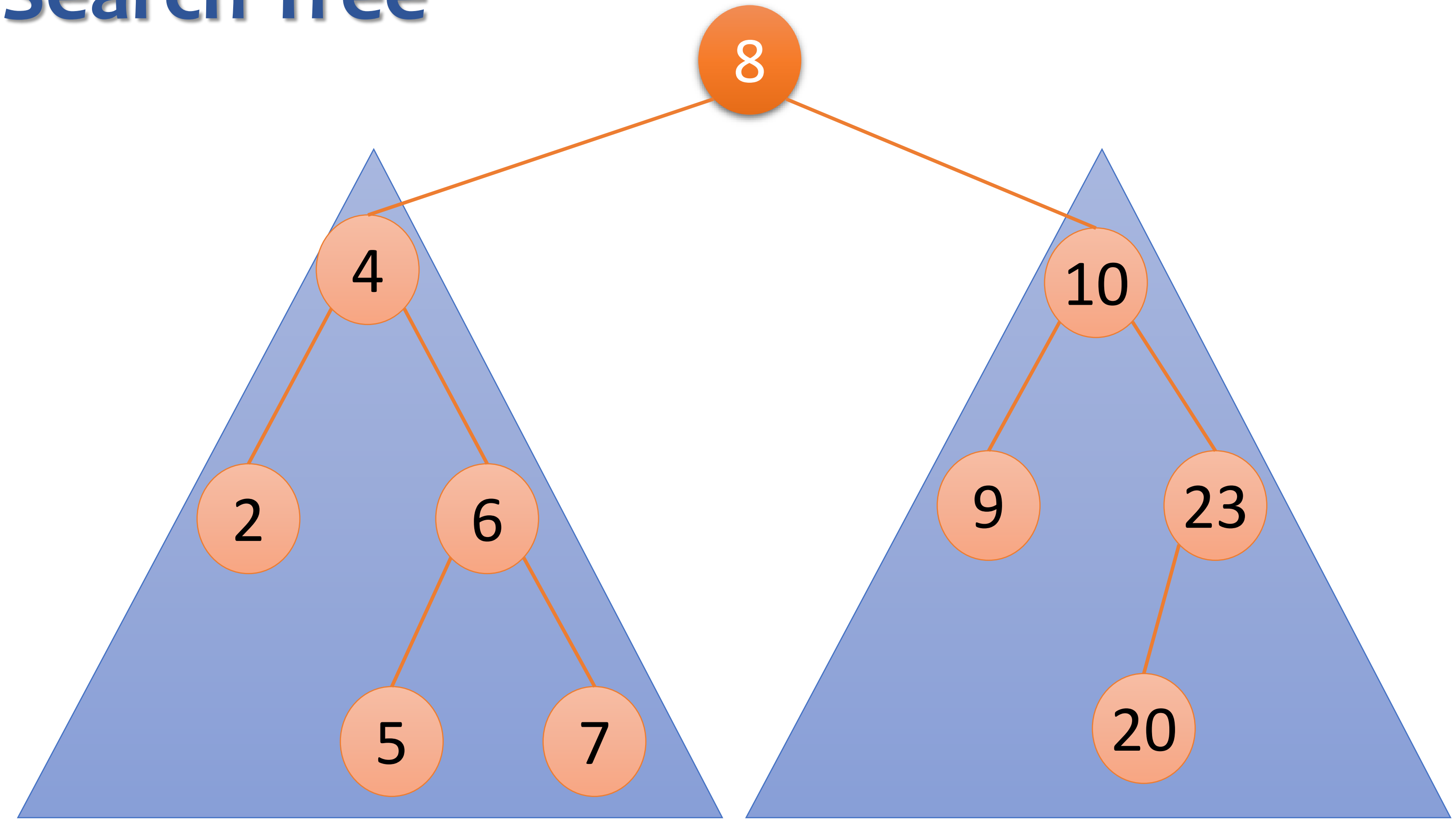
- Given a binary tree T with n nodes.
- What is the maximum height of that tree?
- What is the minimum height of that tree?

Binary Search Tree

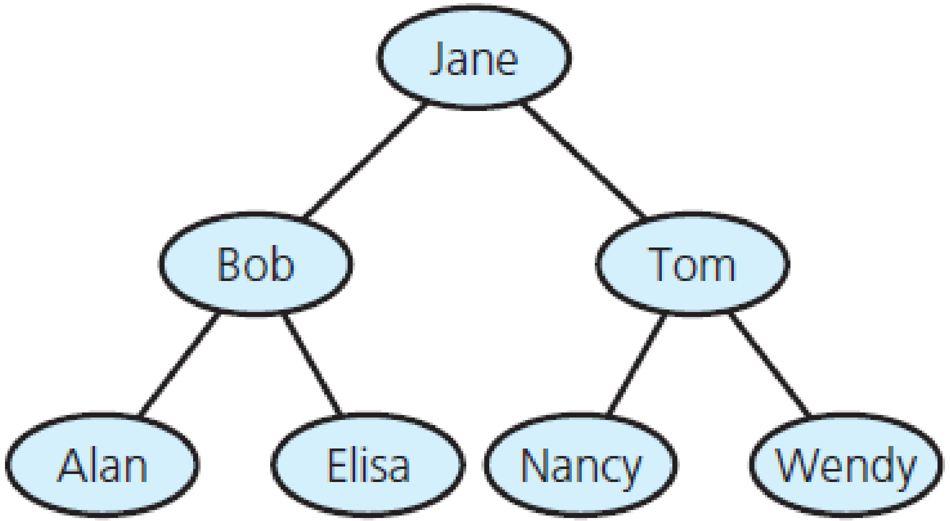
Binary Search Tree

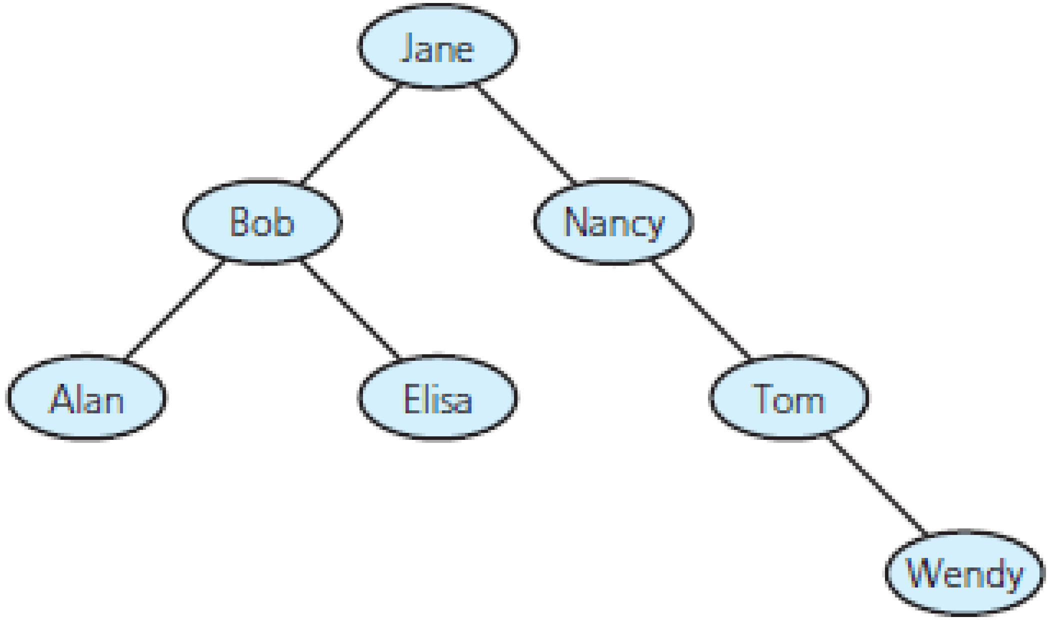
- A binary search tree is a binary tree in which each node *n* has properties:
  - *n* 's value greater than all values in left subtree T_L
  - *n* 's value less than all values in right subtree T_R
  - Both T_R and T_L are binary search trees.

Binary Search Tree



Binary Search Tree





Operations

- Insert (a key)
- Search (a key)
- Remove (a key)
- Traverse
- Sort (based on key value)
- Rotate (Left rotation, Right rotation)

Insertion

```
Insert (root, Data)
{
   if (root is NULL) {
         Create a new_Node containing Data
         This new_Node becomes root of the tree
   }
   //Compare root's key with Key
   if root's Key is less than Data's Key
         Insert Key to the root's RIGHT subtree
   else if root's Key is greater than Data's Key
         Insert Key to the root's LEFT subtree
   else
         Do nothing //Explain why?
}
```

Insertion

- Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?

15, 5, 12, 8, 23, 1, 17, 21

Insertion

- Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?
  - W, T, N, J, E, B, A
  - W, T, N, A, B, E, J
  - A, B, W, J, N, T, E
  - B, T, E, A, N, W, J

Search

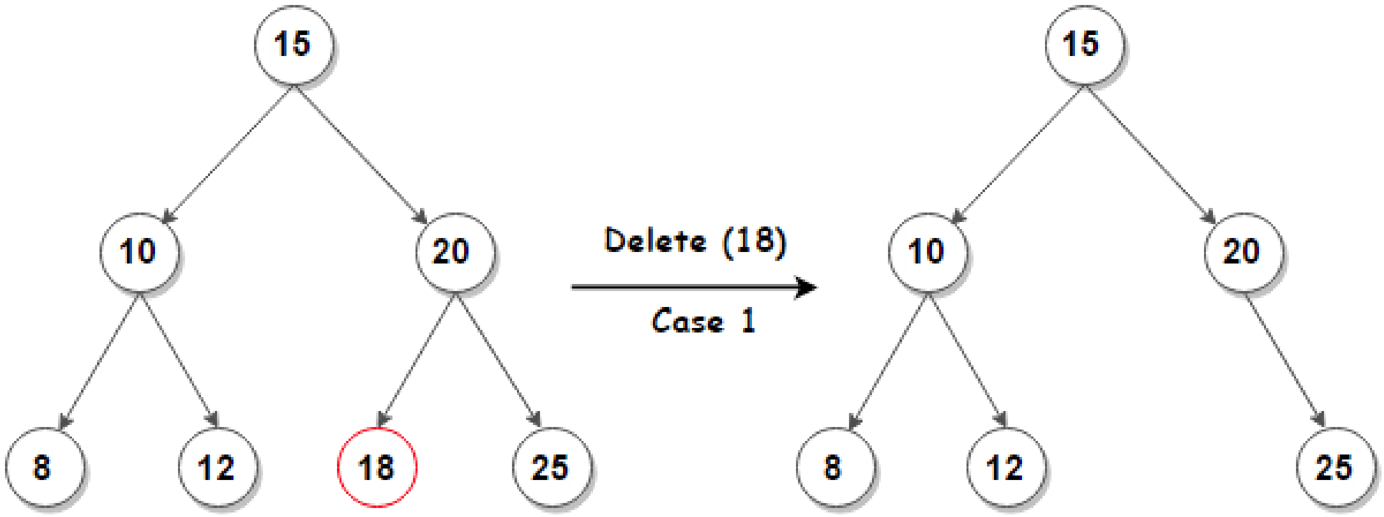
```
Search (root, Data)
{
    if (root is NULL) {
            return NOT_FOUND;
    }
    //Compare root's key with Key
    if root's Key is less than Data's Key
            Search Data in the root's RIGHT subtree
    else if root's Key is greater than Data's Key
            Search Data in the root's LEFT subtree
    else
            return FOUND //Explain why?
}
```

Deletion

- When we delete a node, three possibilities arise.
- Node to be deleted:
  - **is leaf:**
    - Simply remove from the tree.
  - has **one child:**
    - Remove the node and replace it with its child.
  - has **two children:**
    - Find in-order successor (predecessor) ***S_Node*** of the node.
    - Copy contents of ***S_Node*** to the node and *recursively* delete the ***S_Node***.

Deletion

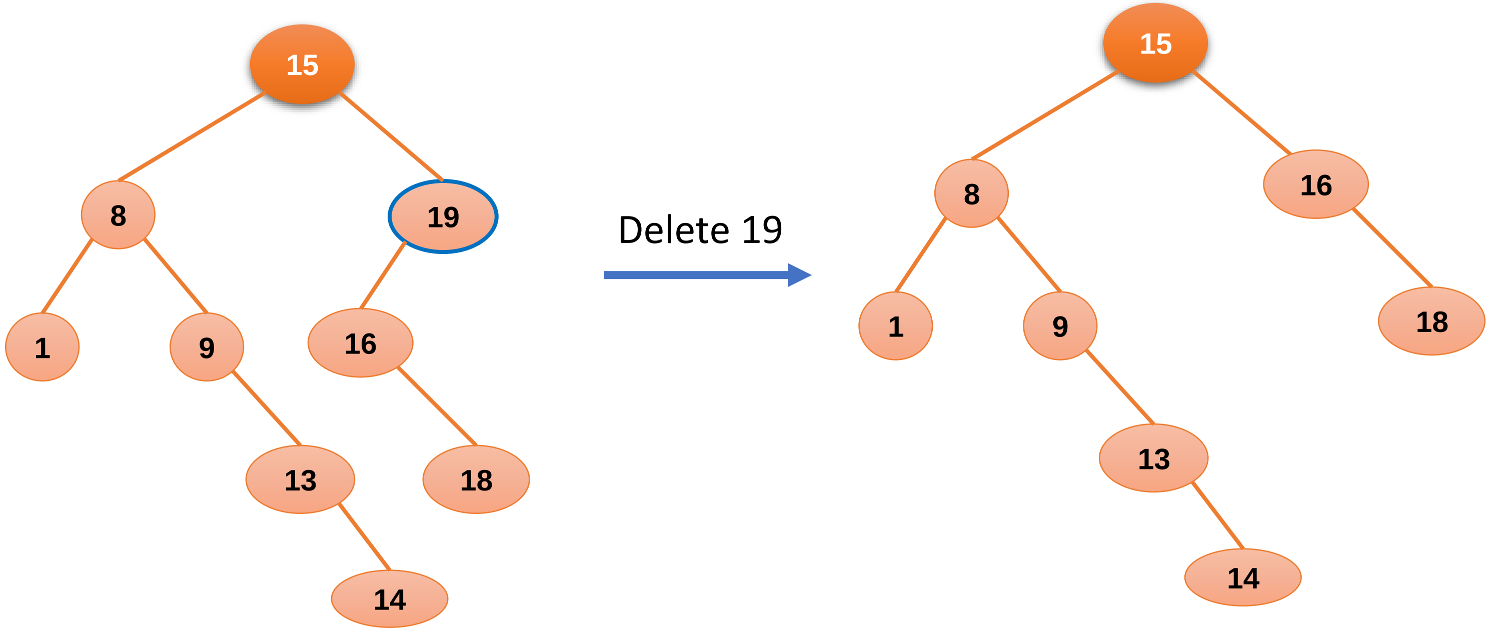
- Node is leaf



Source: https://www.techiedelight.com/deletion-from-bst/

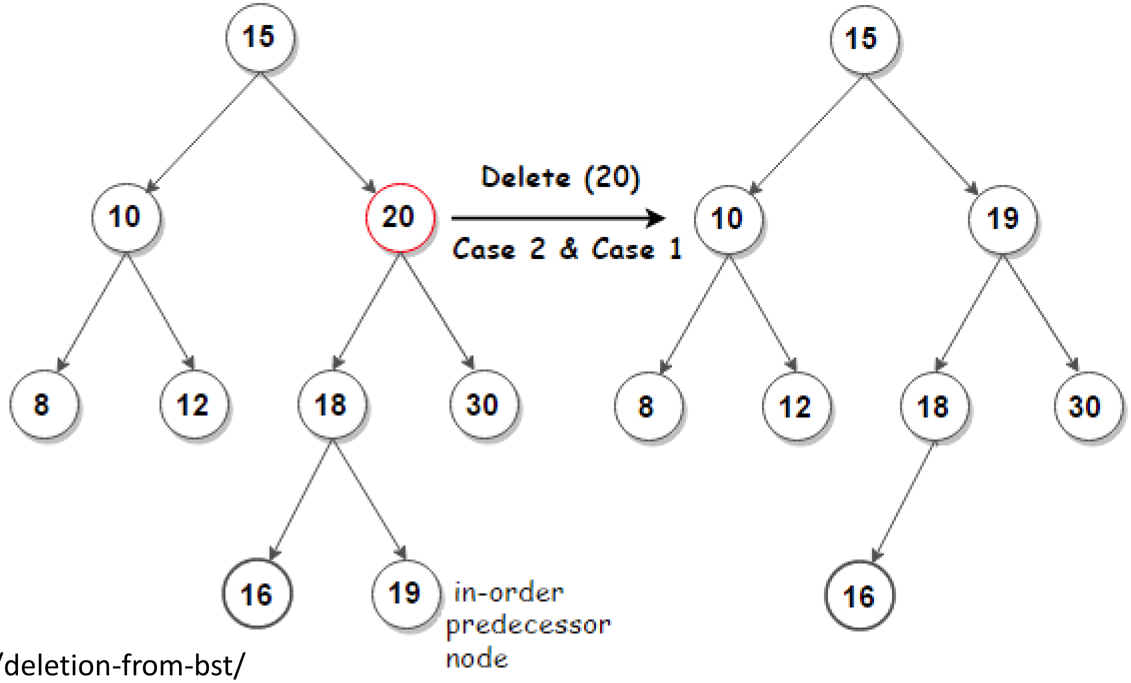
Deletion

- Node has one child



Deletion

- Node has two children



Source: https://www.techiedelight.com/deletion-from-bst/

Let's draw tree after deleting 19!

Traversals

- Pre-order:

  Node - Left - Right

- In-order:

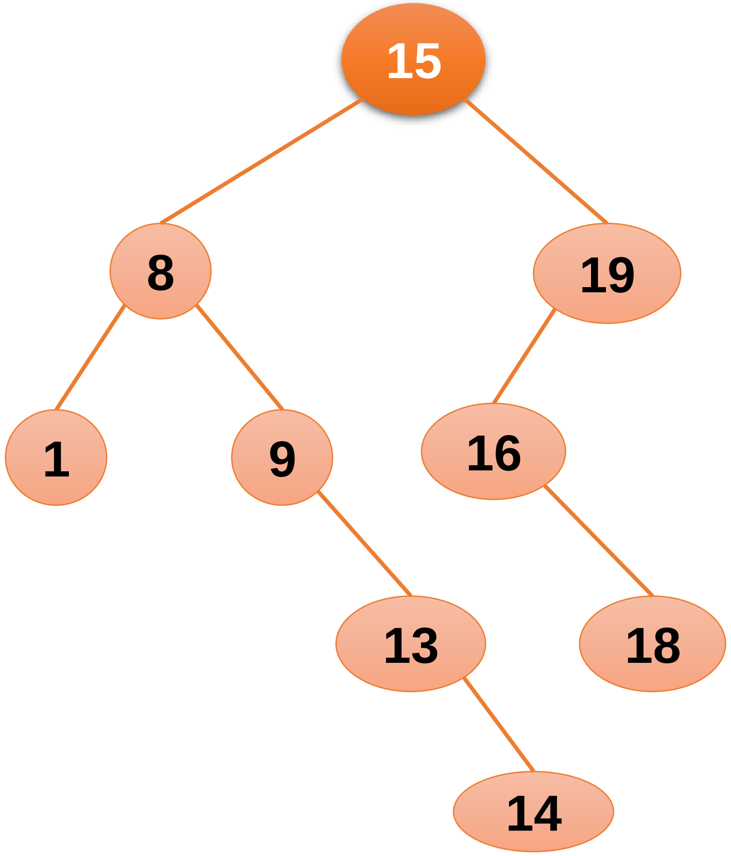
  Left - Node - Right

- Post-order:

  Left - Right - Node

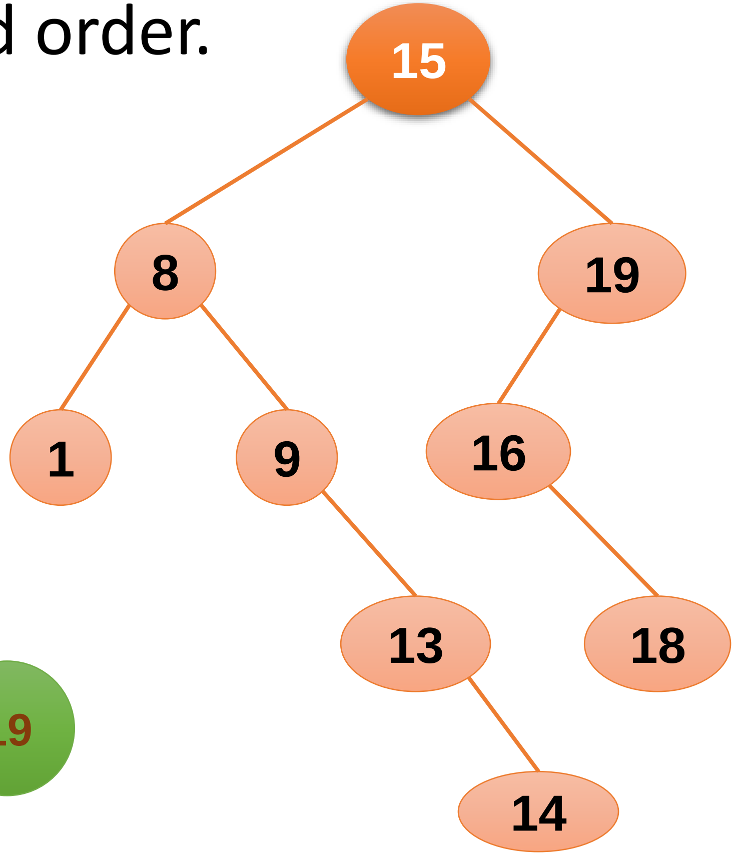
Traversals

- What are the pre-order, in-order and post-order traversals of this binary search tree?



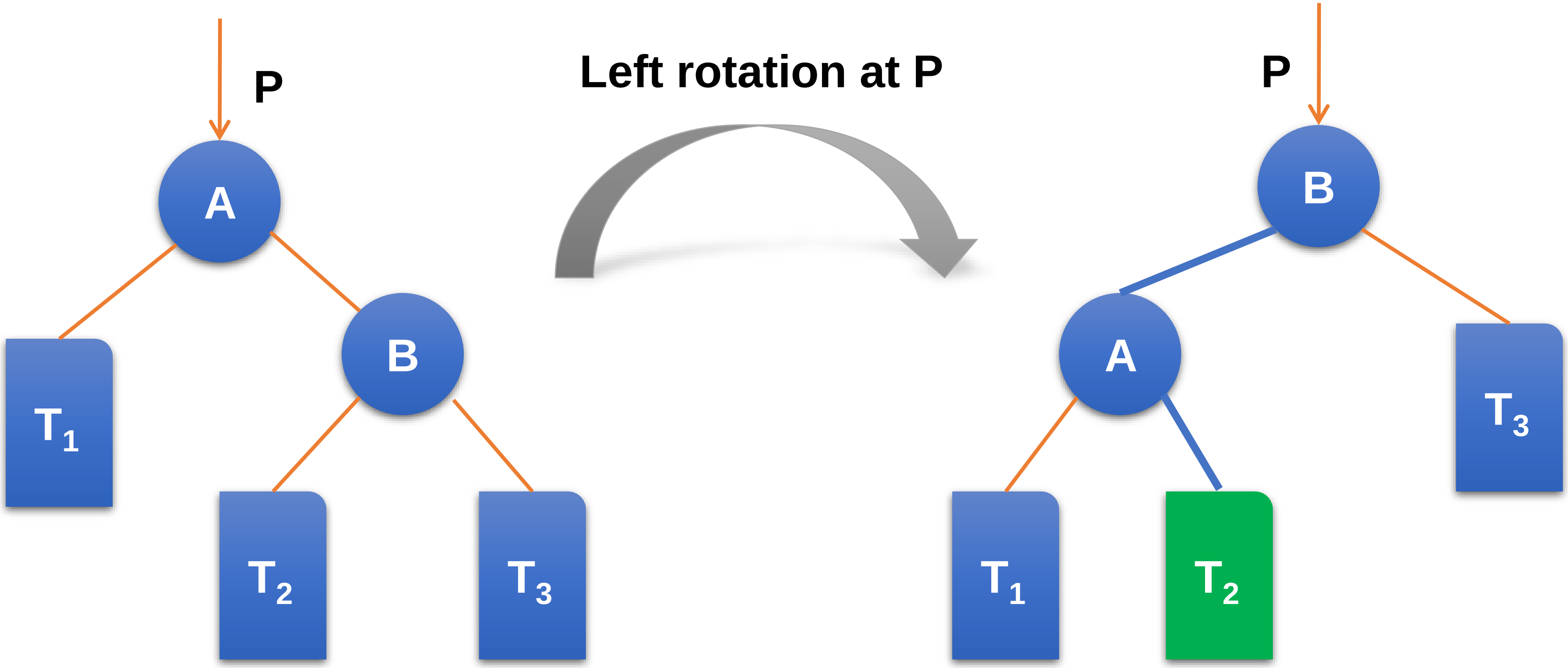
Sort

- In-order traversal gives the key values in sorted order.

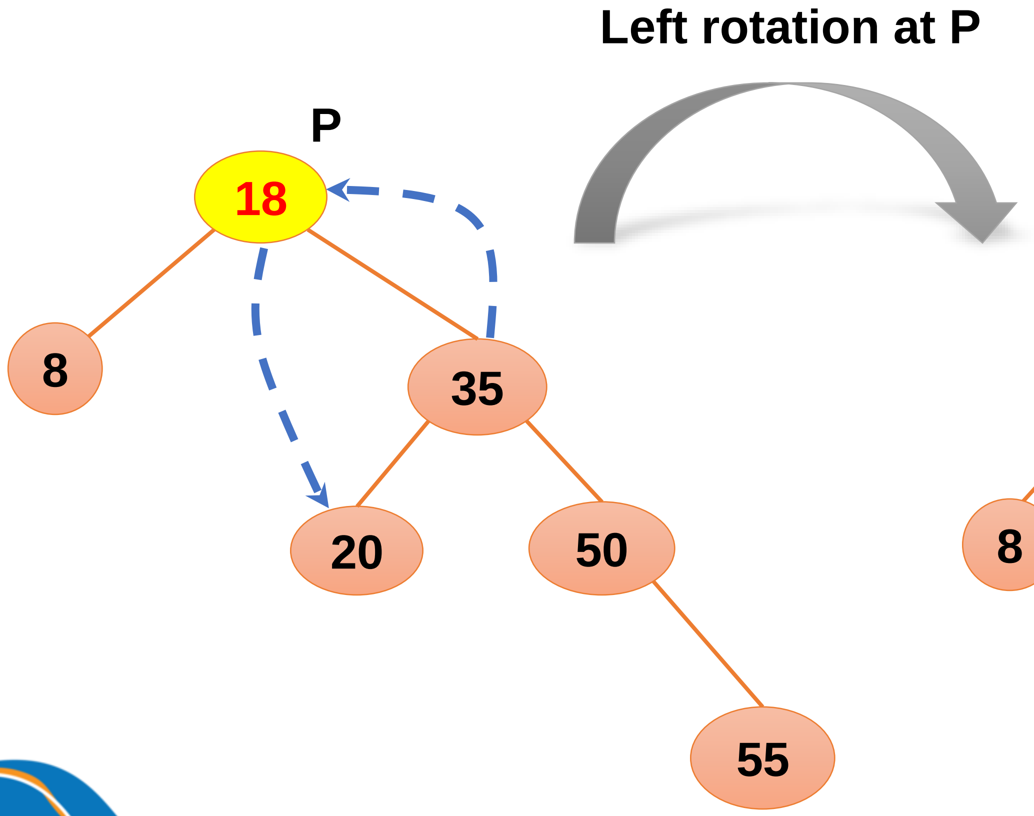


1 8 9 13 14 15 16 18 19

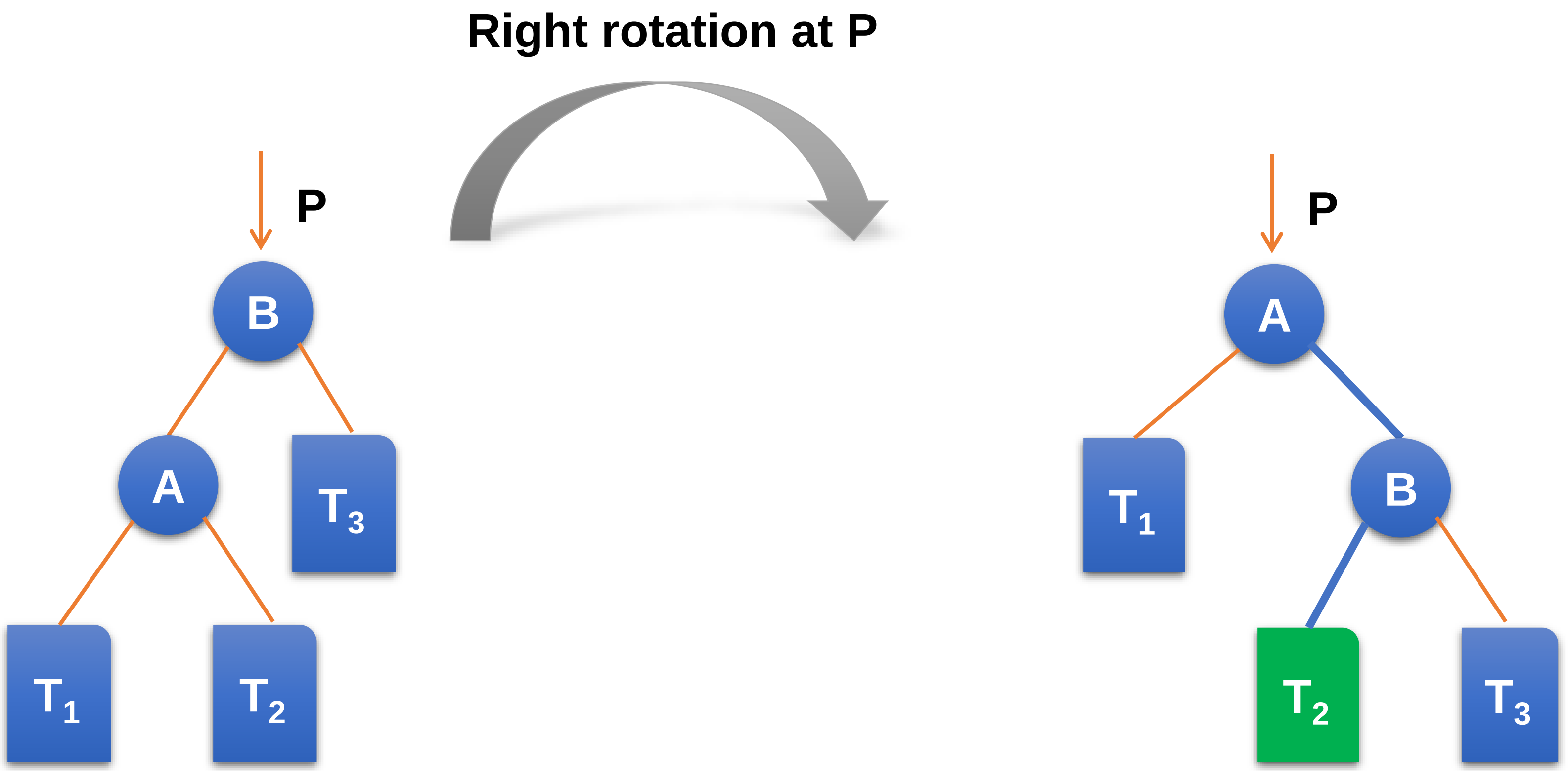
Left Rotation



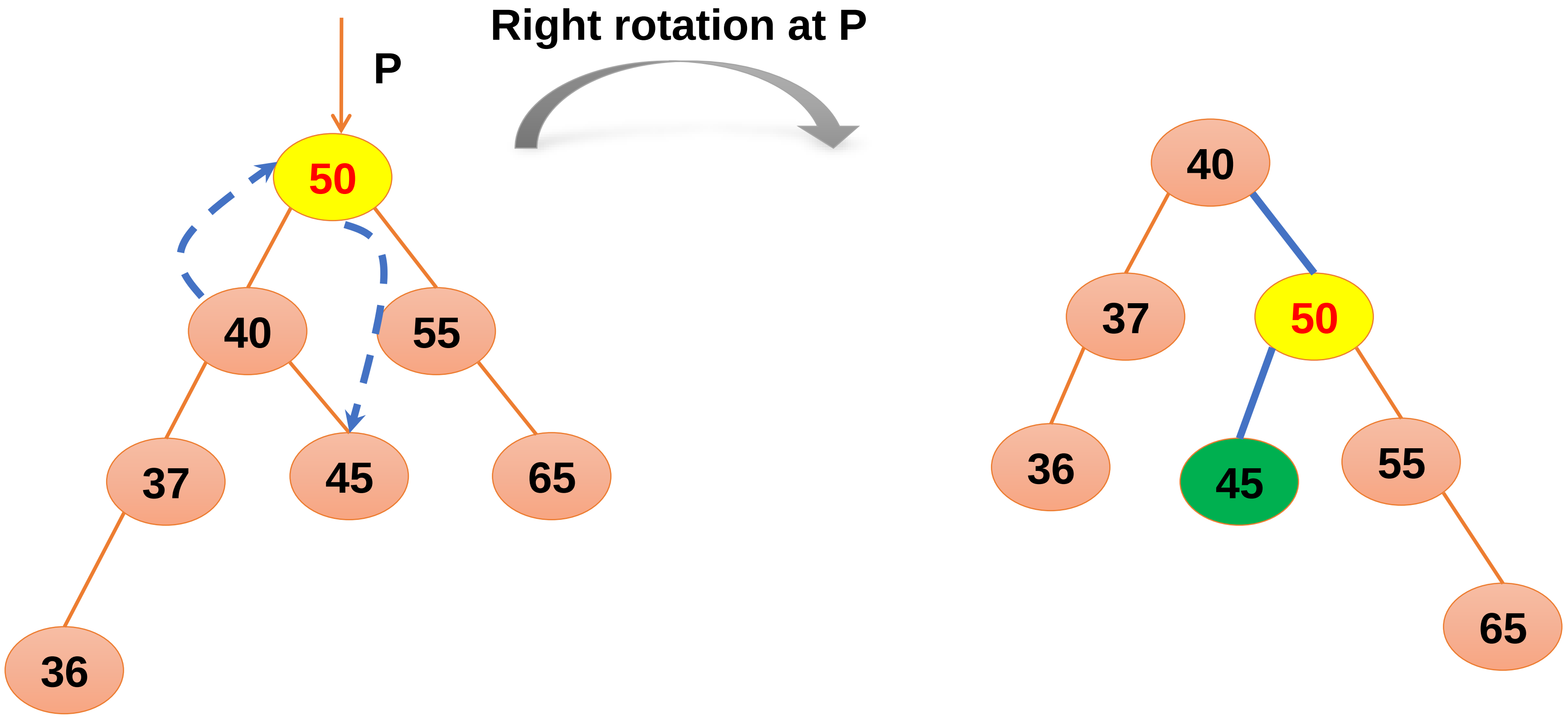
Left Rotation



Right Rotation



Right Rotation



Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Removal	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$

Very Bad Binary Search Tree

- Beginning with an empty binary search tree, what binary search tree is formed when inserting the following values in the order given?

2, 4, 6, 8, 10, 12, 14, 18, 20

AVL Tree

AVL Tree

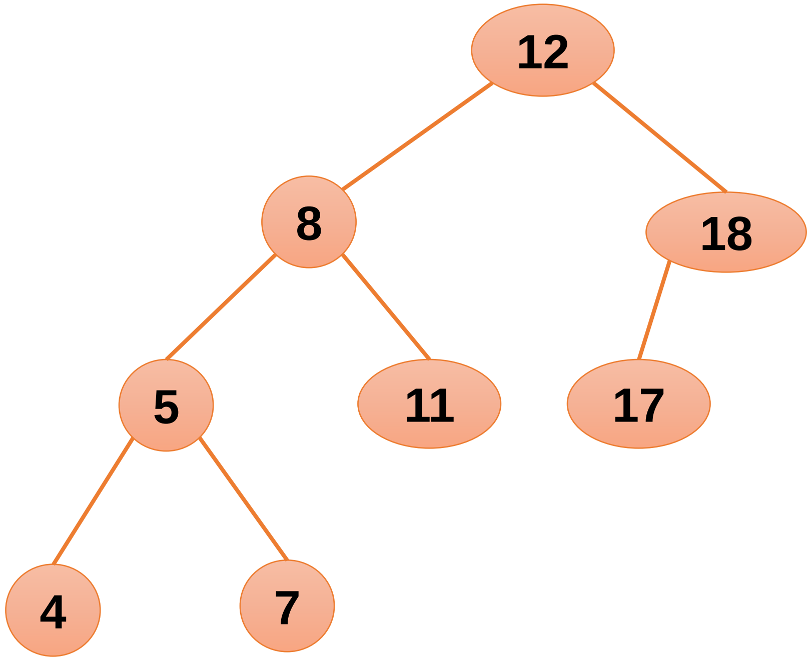
- Named for inventors, (Georgii) **A**delson-**V**elsky and (Evgenii) **L**andis

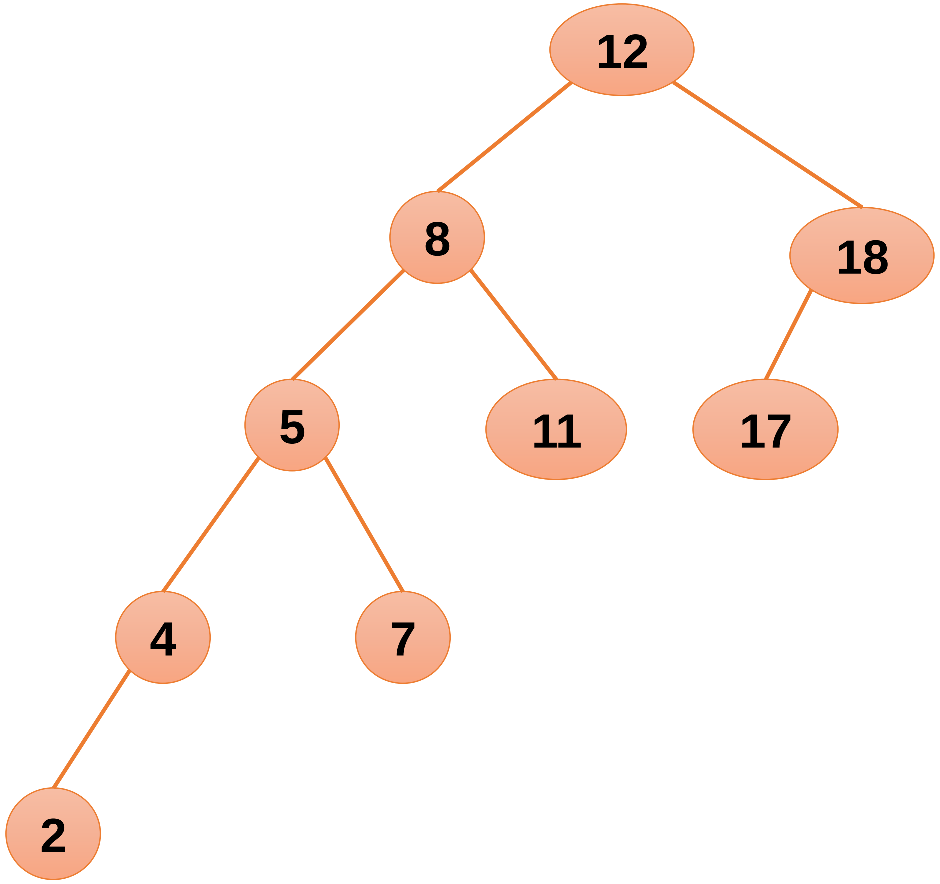
- Invented in 1962 (paper "*An algorithm for organization of information*").

- AVL Tree is a **self-balancing** binary search tree where
  - for ALL nodes, the difference between height of the left subtrees and the right subtrees **cannot be more than one**.

AVL Tree

- Which is the AVL Tree?





AVL Tree

- A balanced binary search tree
  - Maintains height close to the minimum
  - After insertion or deletion, check the tree is still AVL tree – determine whether any node in tree has left and right subtrees whose heights differ by more than 1

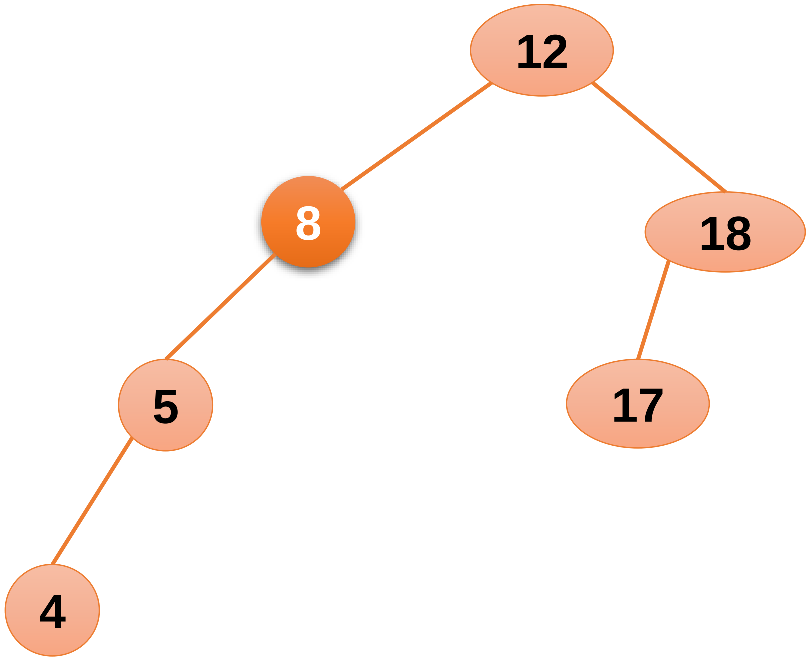
- Can search AVL tree almost as efficiently as minimum-height binary search tree.

Cases of Un-balanced Nodes

- Left-Left case
- Left-Right case
- Right-Right case
- Right-Left case

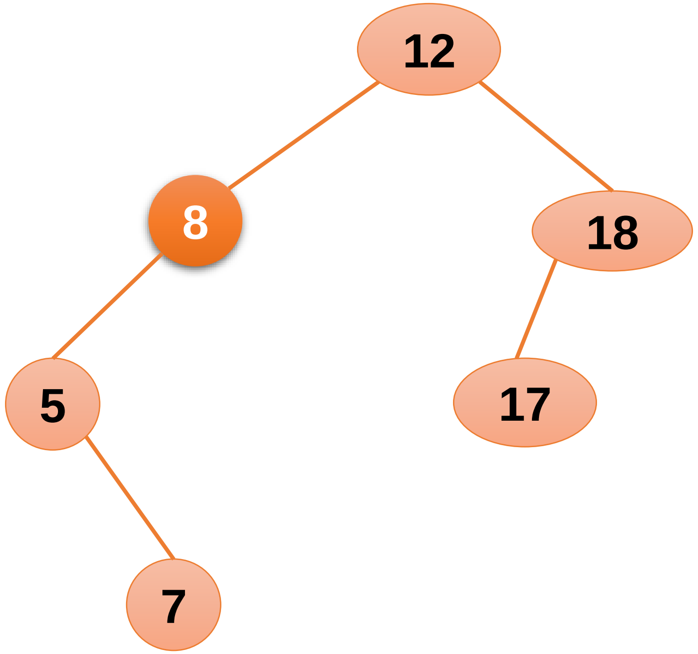
Cases of Un-balanced Nodes

- Left-Left case



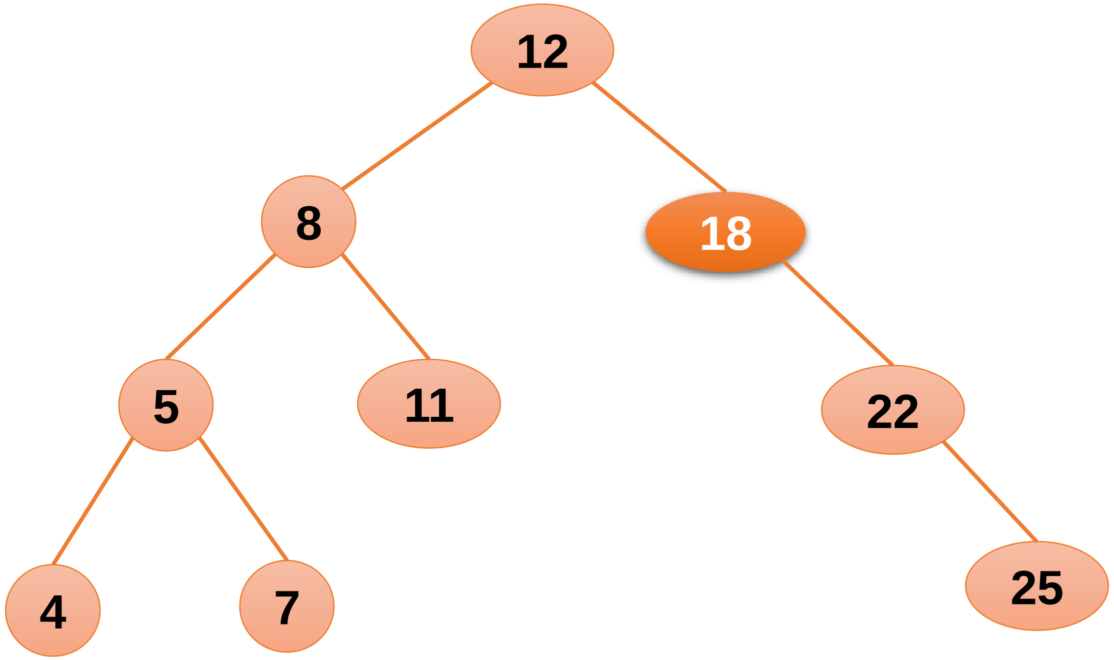
Cases of Un-balanced Nodes

- Left-Right case



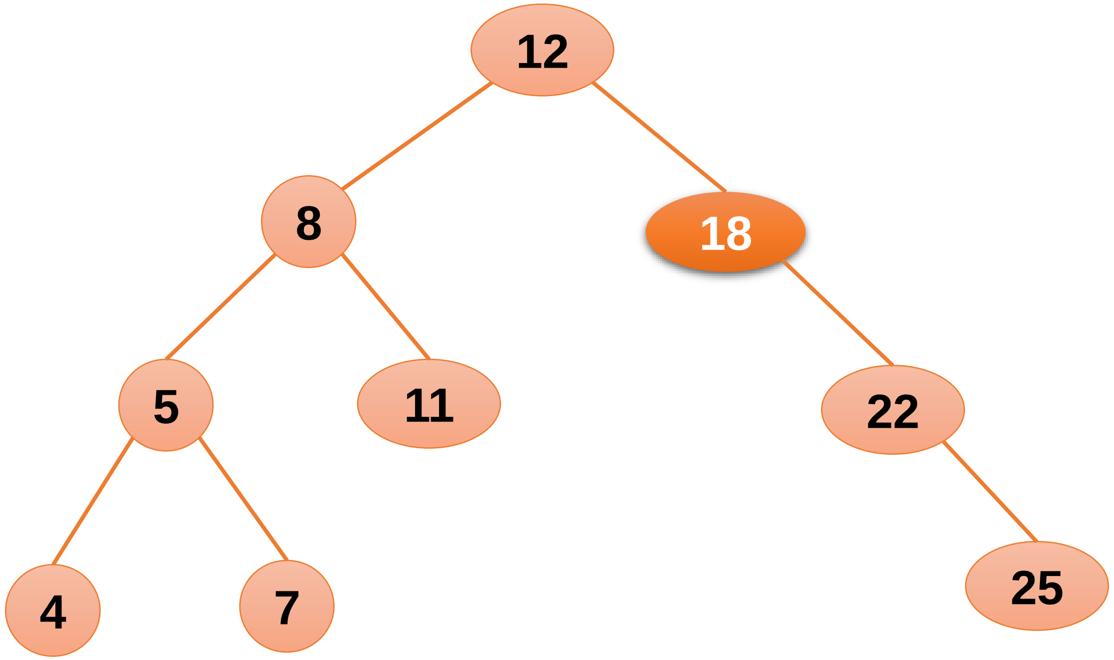
Cases of Un-balanced Nodes

- Right-Right case



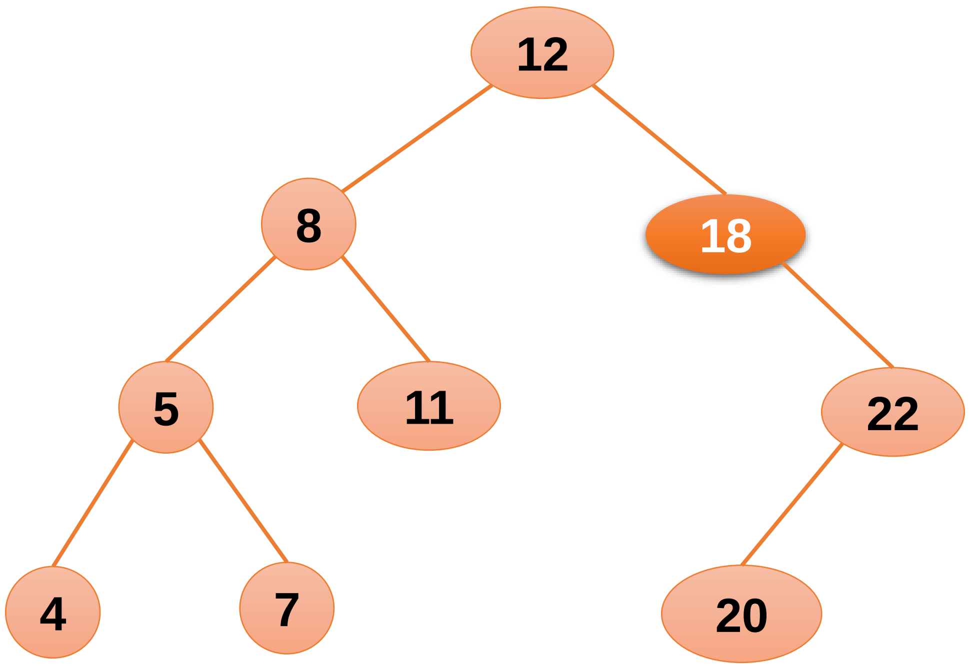
Cases of Un-balanced Nodes

- Right-Right case



Cases of Un-Balanced Nodes

- Right-Left case



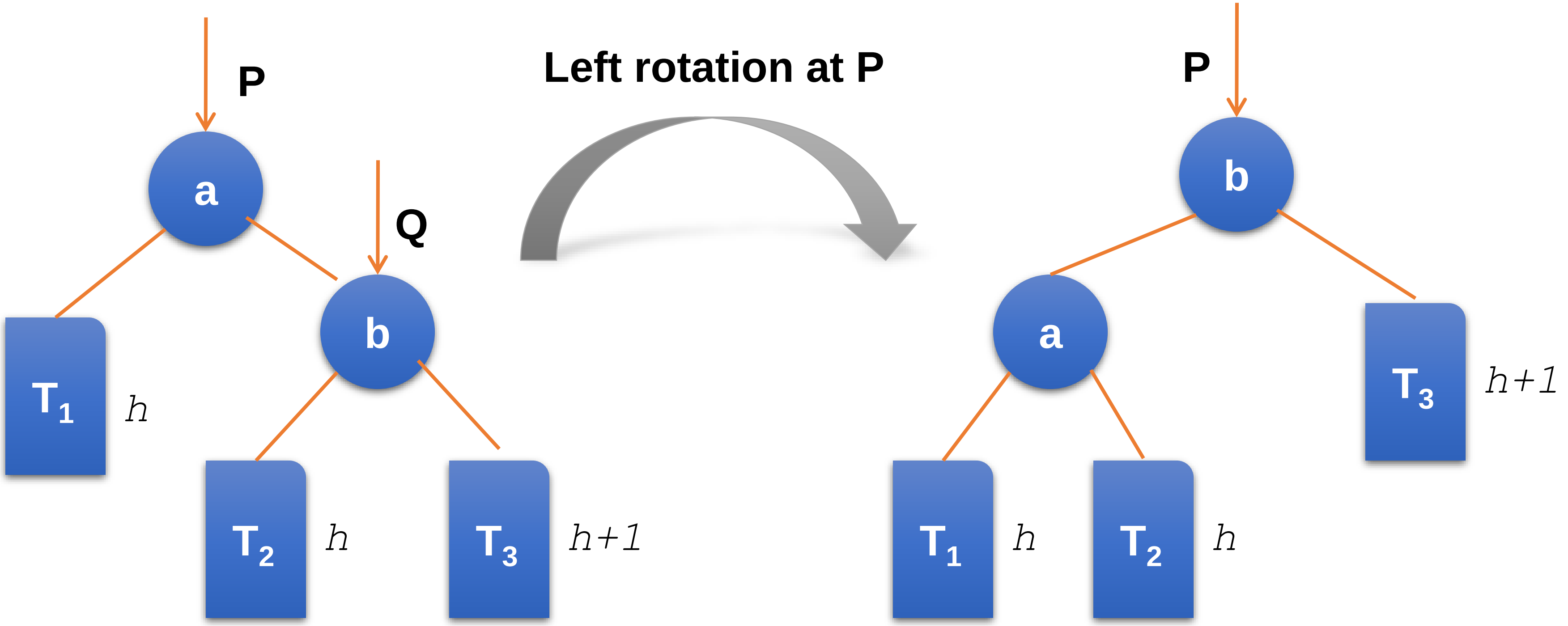
Un-balanced Resolving

- Right-Right case:
  - Left rotation at un-balanced node.

- Right-Left case:
  - Right rotation at un-balanced node's right child.
  - Left rotation at un-balanced node.

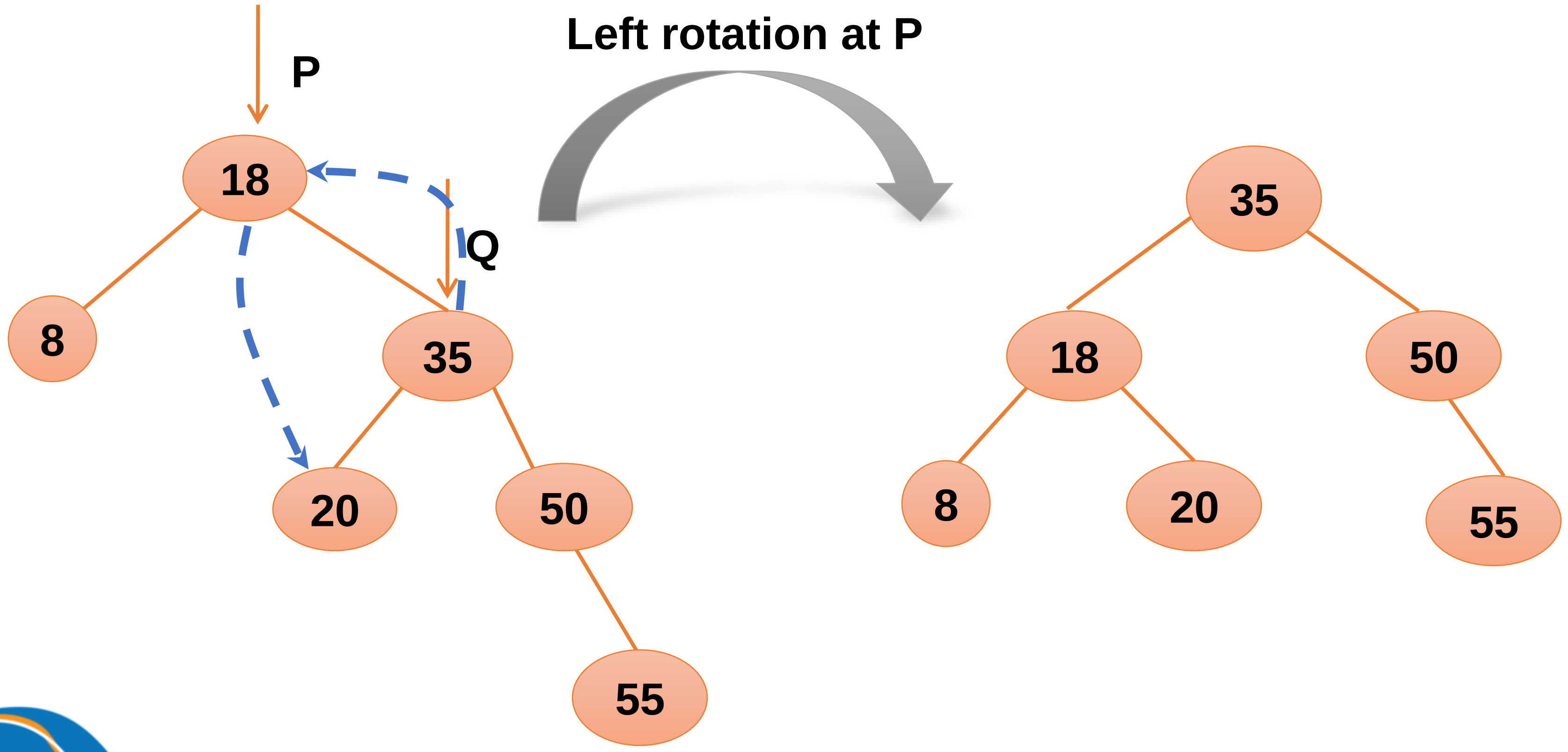
Un-balanced Resolving

- Right-Right case:



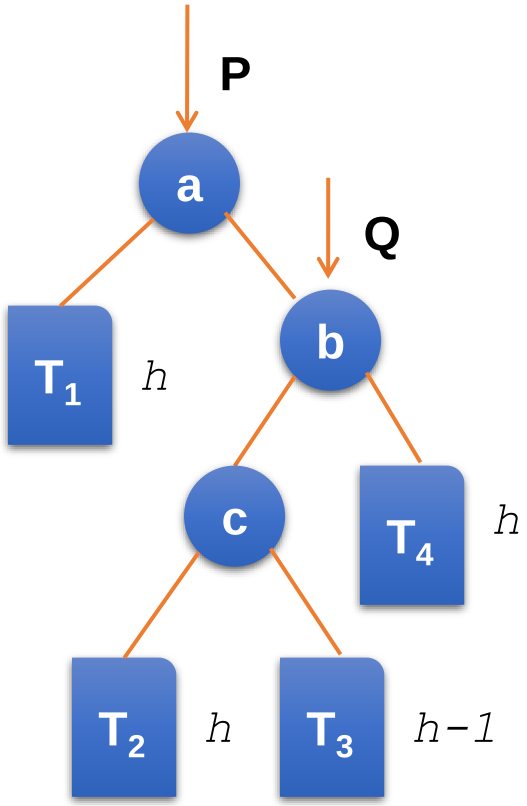
Un-balanced Resolving

- Right-Right case:



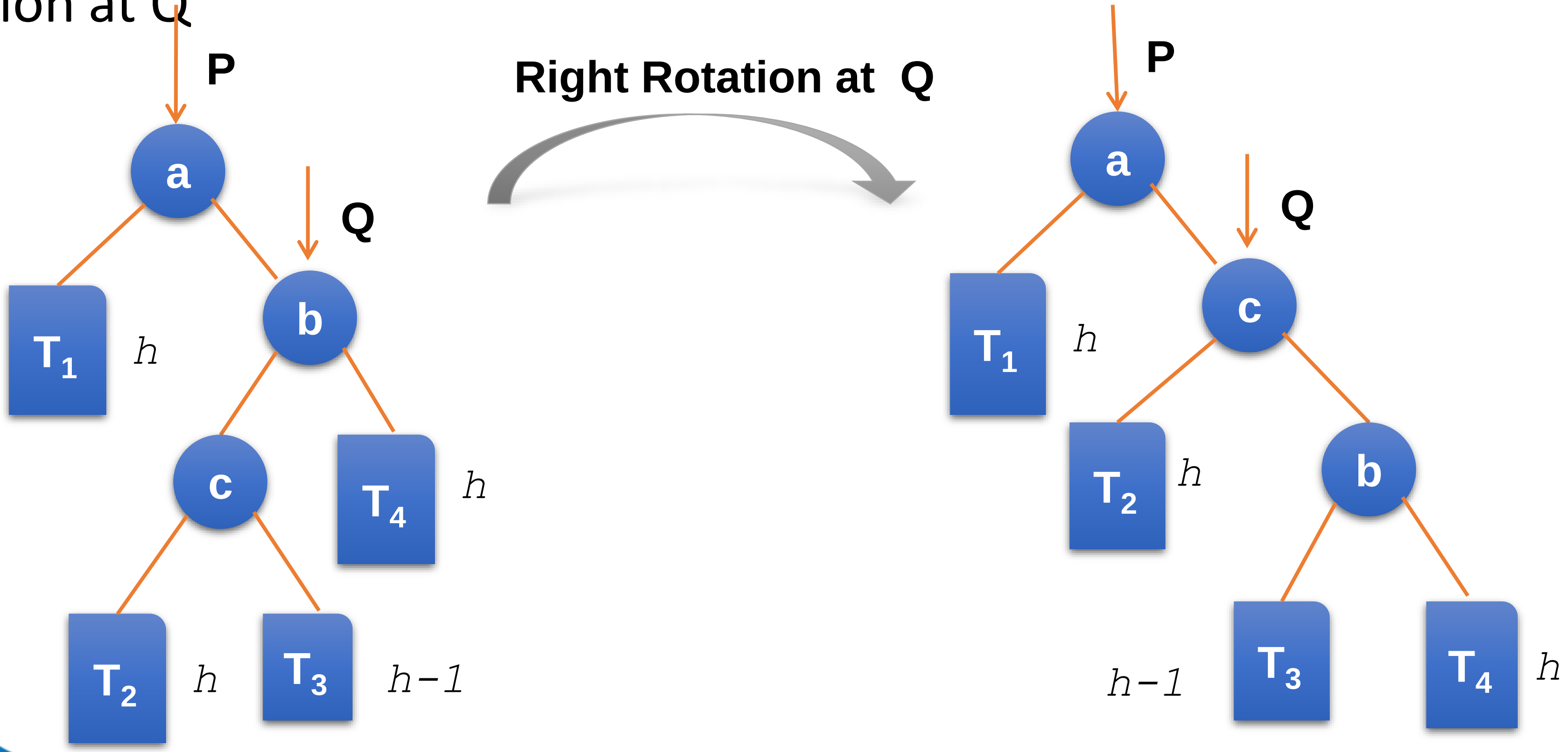
Un-balanced Resolving

- Right-Left case:
  - Right rotation at Q
  - Left rotation at P



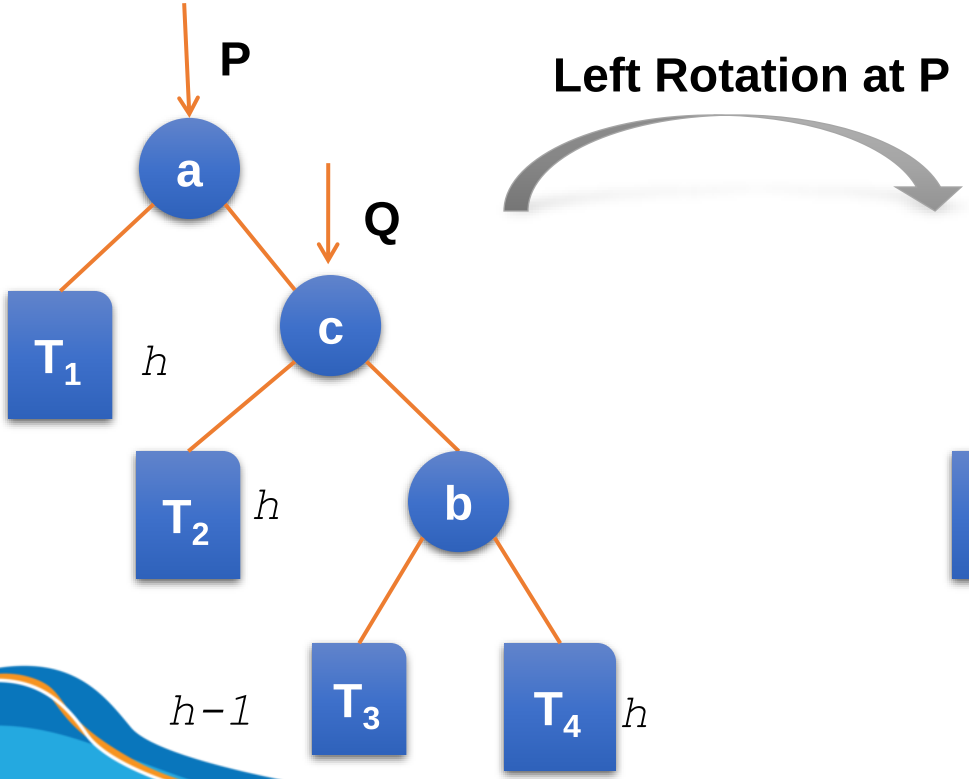
Un-balanced Resolving

- Right-Left case:
  - Right rotation at Q

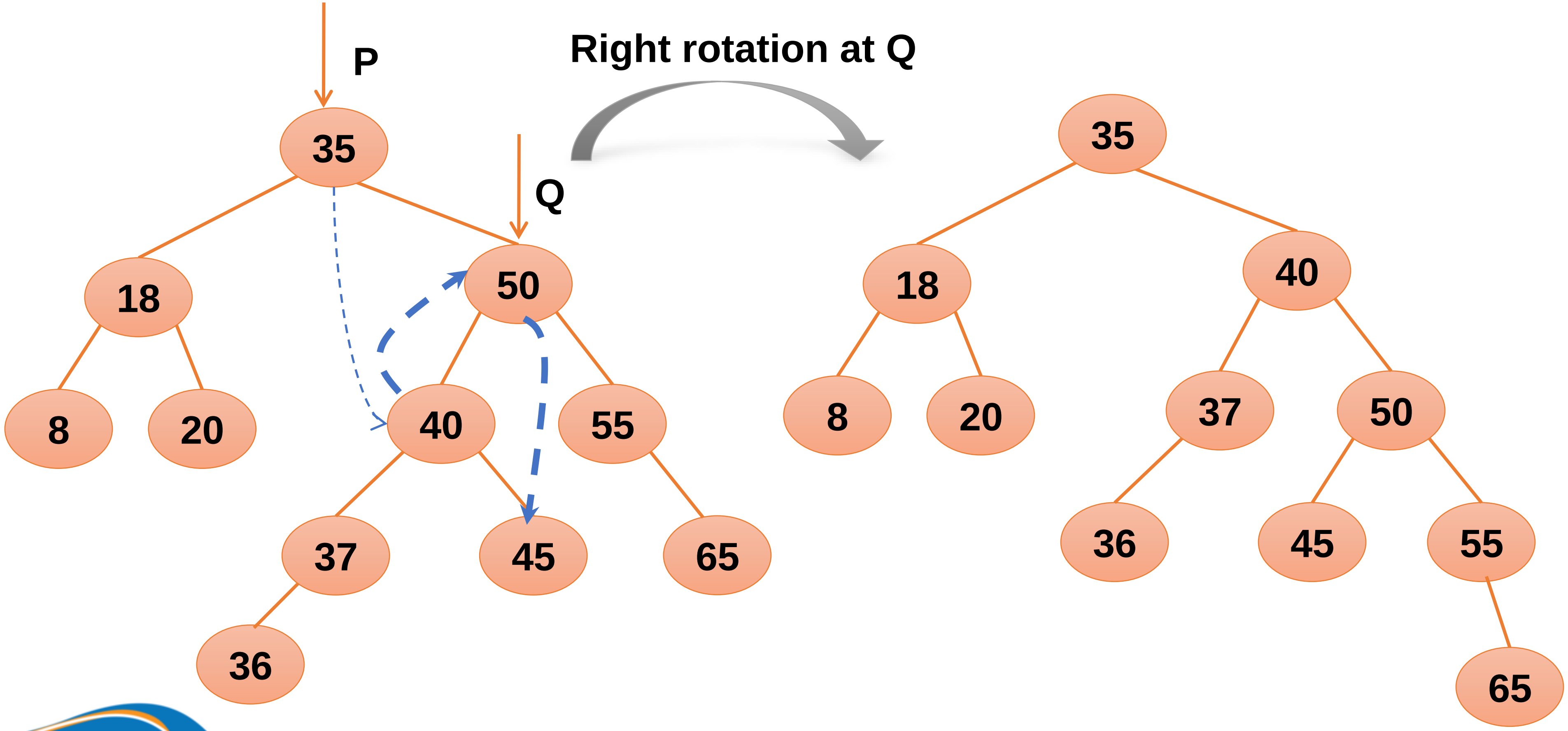


Un-balanced Resolving

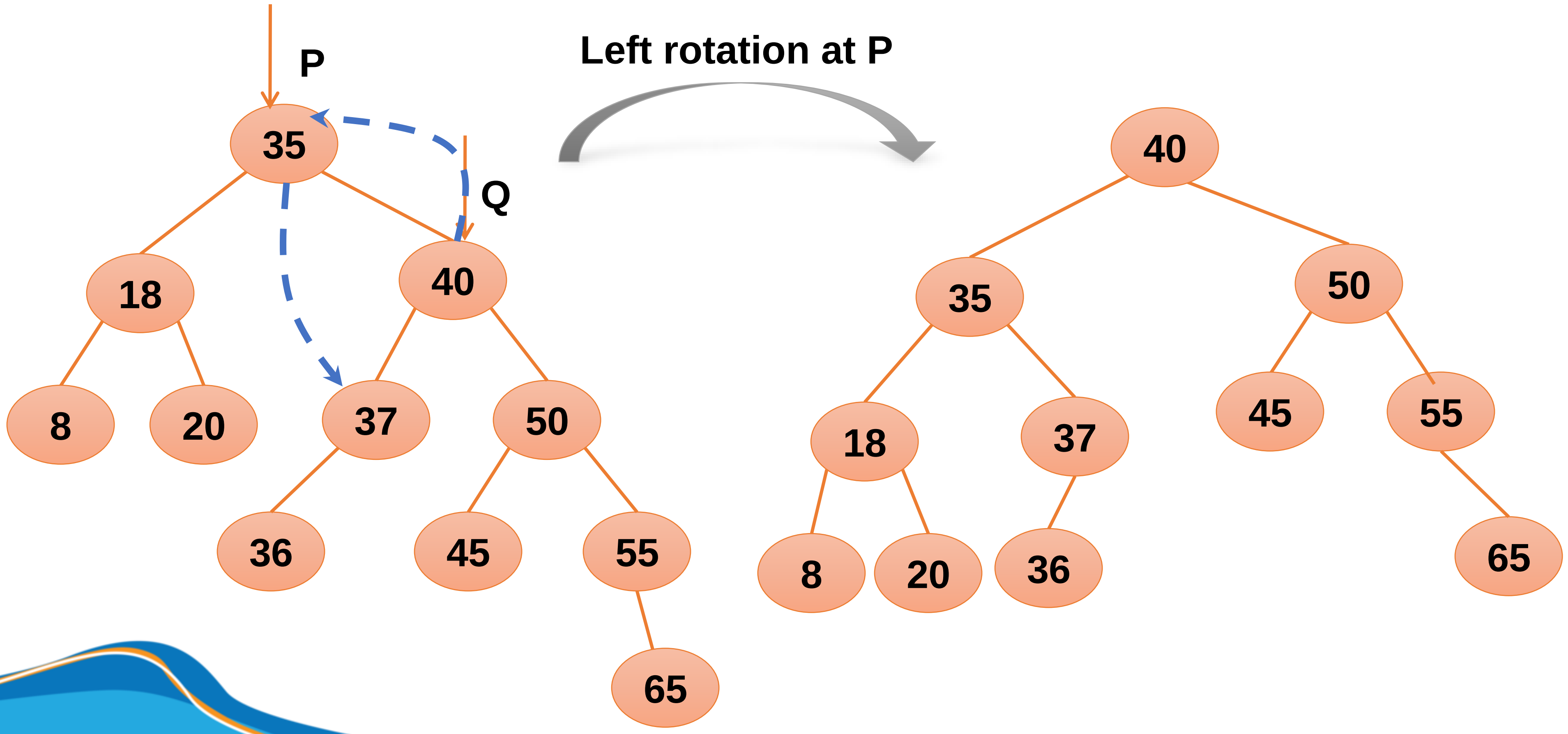
- Right-Left case:
  - Right rotation at P



Un-balanced Resolving



Un-balanced Resolving



Un-balanced Resolving

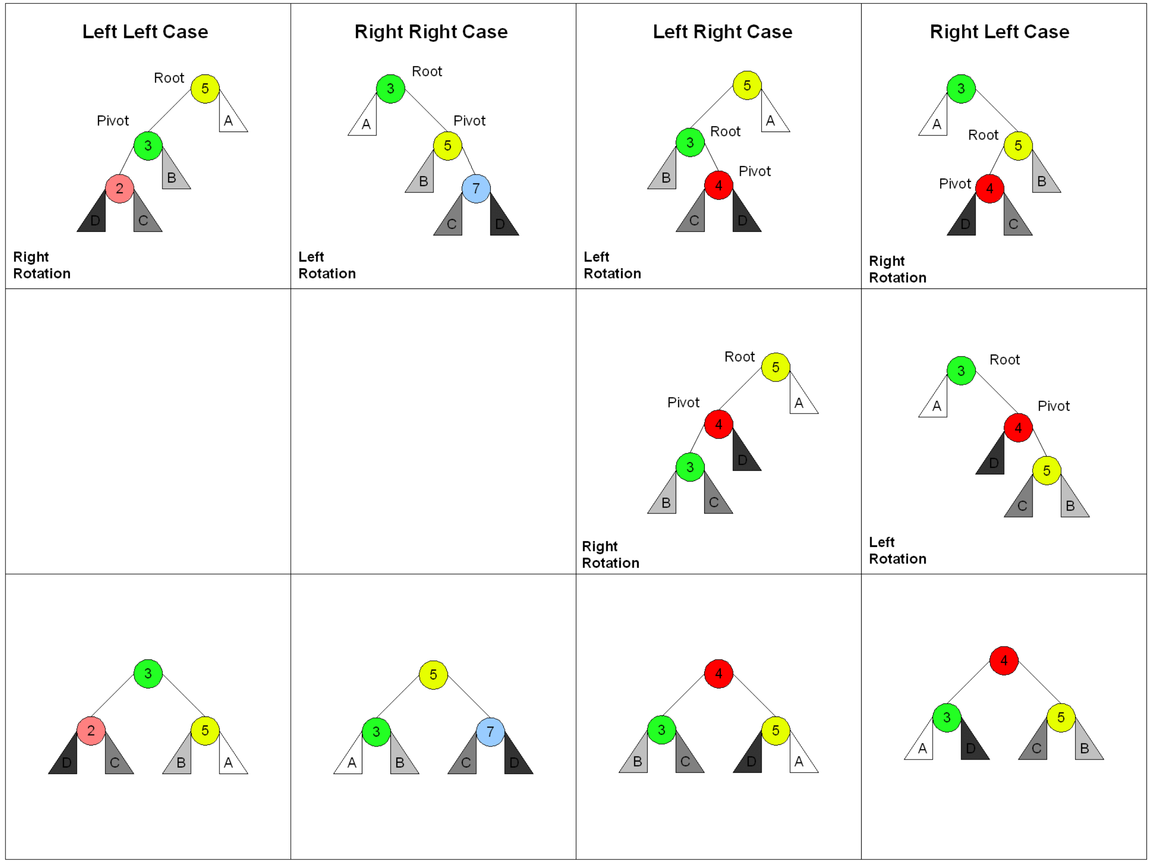
- Left-Left case:
  - Right rotation at un-balanced node.

- Left-Right case:
  - Left rotation at un-balanced node's left child.
  - Right rotation at un-balanced node.

Un-balanced Resolving

There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.



Source: Wikipedia

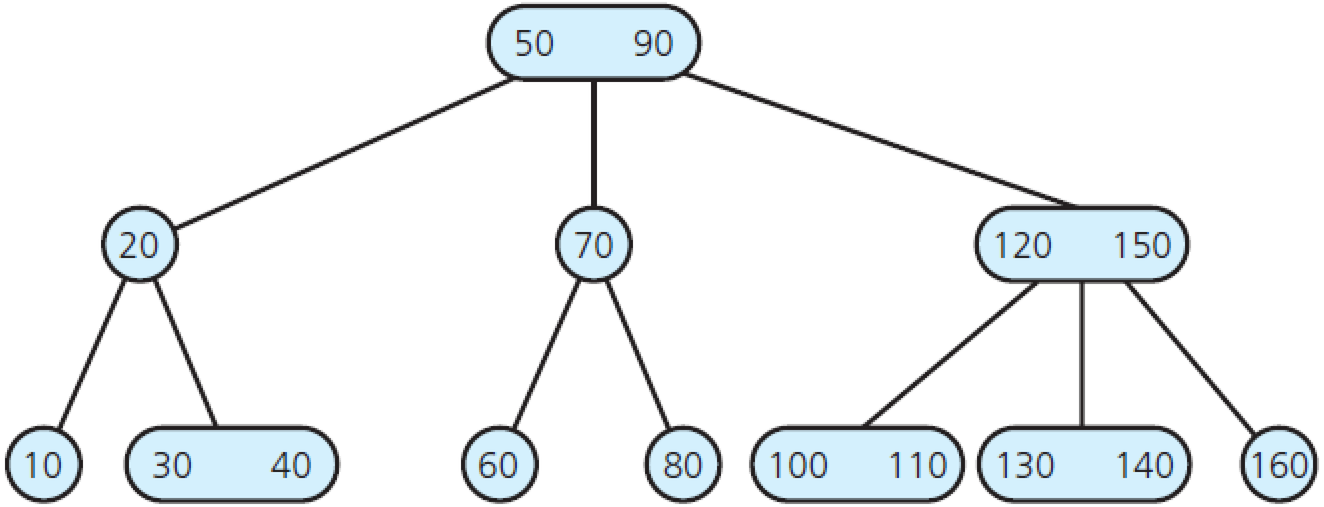
2-3 Trees, 2-3-4 Trees

2-3 Trees, 2-3-4 Trees

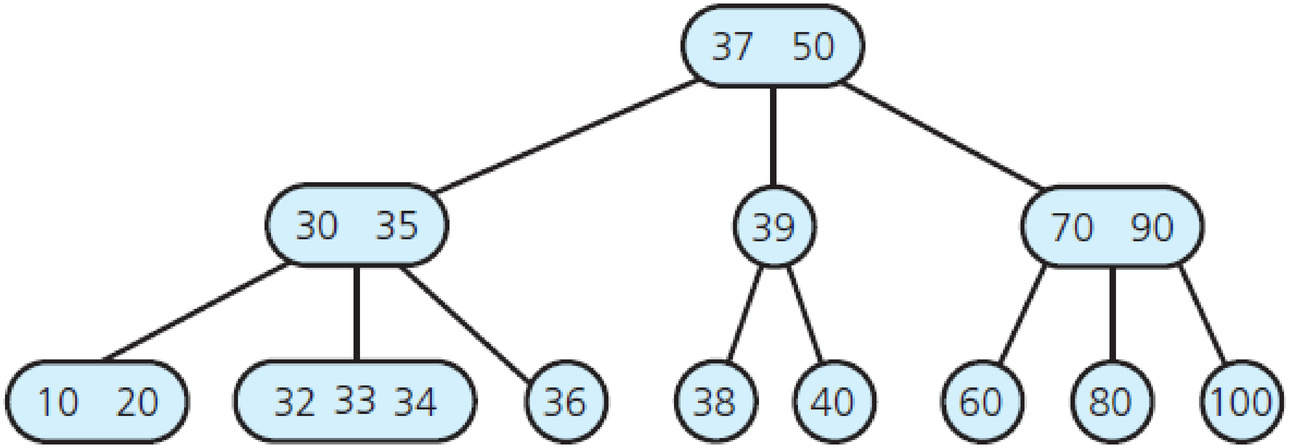
- A 2-3 tree is not a binary tree. Neither is a 2-3-4 tree.

- A 2-3 tree, a 2-3-4 tree are never taller than a minimum-height binary tree.

2-3 Trees



2-3-4 Trees



2-node, 3-node, 4-node

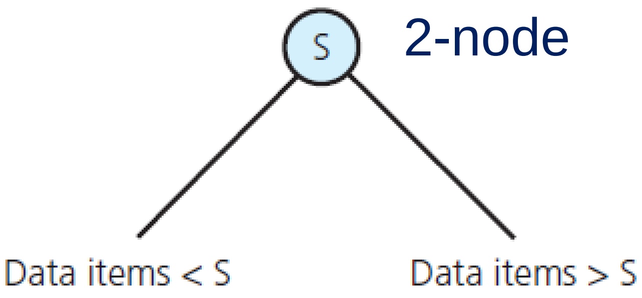
- A `2-node` (has two children) must contain single data item greater than left child's item(s) and less than right child's item(s).

- A `3-node` (has three children) must contain two data items, `S` and `L`, such that
  - S is greater than left child's item(s) and less than middle child's item(s);
  - L is greater than middle child's item(s) and less than right child's item(s).

- A `4-node` (has our children) must contain three data items `S`, `M`, and `L` that satisfy:
  - S is greater than left child's item(s) and less than middle-left child's item(s)
  - M is greater than middle-left child's item(s) and less than middle-right child's item(s);
  - L is greater than middle-right child's item(s) and less than right child's item(s).

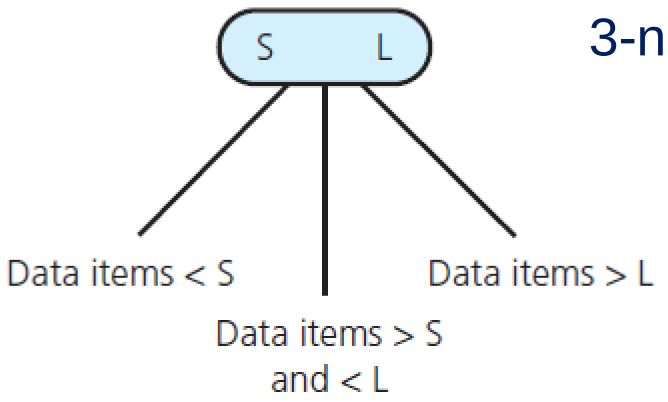
2-node, 3-node, 4-node

(a)



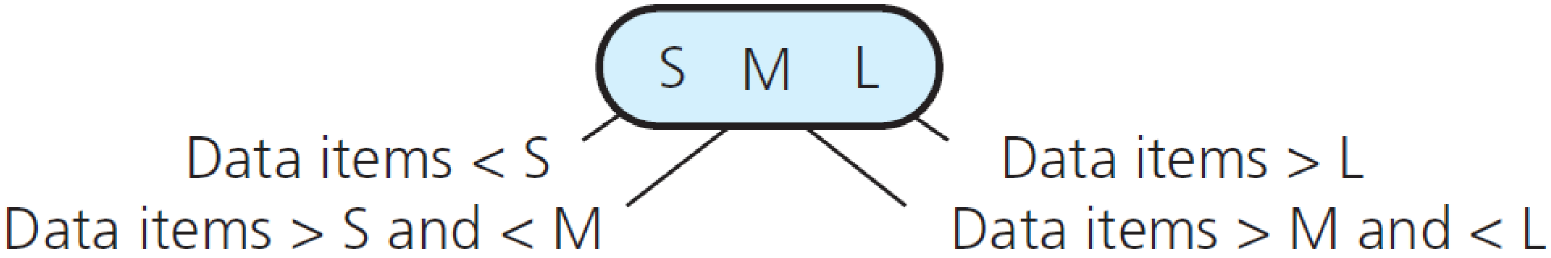
2-node

(b)



3-node

4-node



2-3 Trees

- 2–3 trees were invented by John Hopcroft in 1970.

- 2-3 tree is a tree in which
  - Every internal node is either a 2-node or a 3-node.

  - Leaves have no children and may contain either one or two data items.

2-3-4 Trees

- 2-3 tree is a tree in which
  - Every internal node is a 2-node, a 3-node or a 4-node.
  - Leaves have no children and may contain either one, two or three data items.

Insertion Operation

Inserting Data into a 2-3 Tree

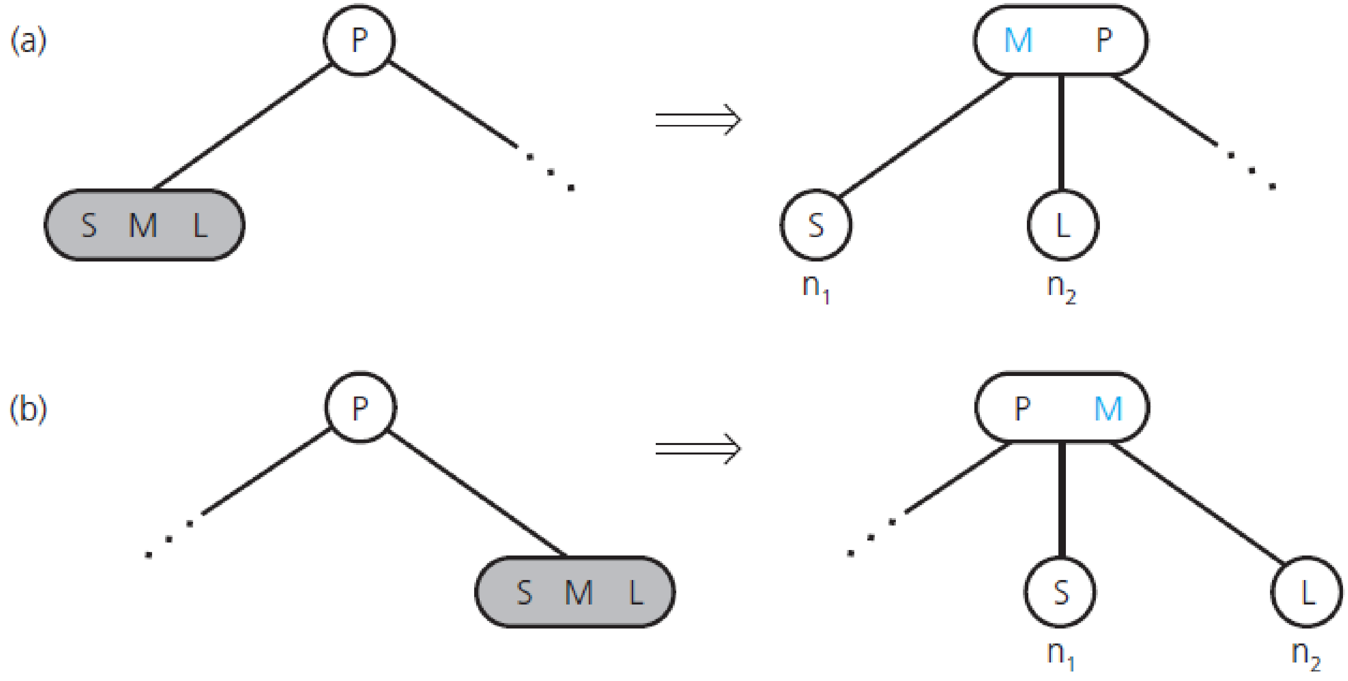
```
// Inserts a new item into a 2-3 tree whose items are distinct and differ from the
// new item.
insertItem(23Tree: TwoThreeTree, newItem: ItemType)

   Locate the leaf, leafNode, in which newItem belongs
   Add newItem to leafNode

   if (leafNode has three items)
       split(leafNode)


// Splits node n, which contains two items. Note: If n is
// not a leaf, it has four children.
split(n: TwoThreeNode)

   if (n is the root)
       Create a new node p
   else
       Let p be the parent of n

   Replace node n with two nodes, n1 and n2, so that p is their parent
   Give n1 the item in n with the smallest value
   Give n2 the item in n with the largest value
```

Inserting Data into a 2-3 Tree

```
split(n: TwoThreeNode)

   if (n is the root)
       Create a new node p
   else
       Let p be the parent of n

   Replace node n with two nodes, n1 and n2, so that p is their parent
   Give n1 the item in n with the smallest value
   Give n2 the item in n with the largest value

   if (n is not a leaf)
   {
      n1 becomes the parent of n's two leftmost children
      n2 becomes the parent of n's two rightmost children
   }
   Move the item in n that has the middle value up to p
   if (p now has three items)
      split(p)
```

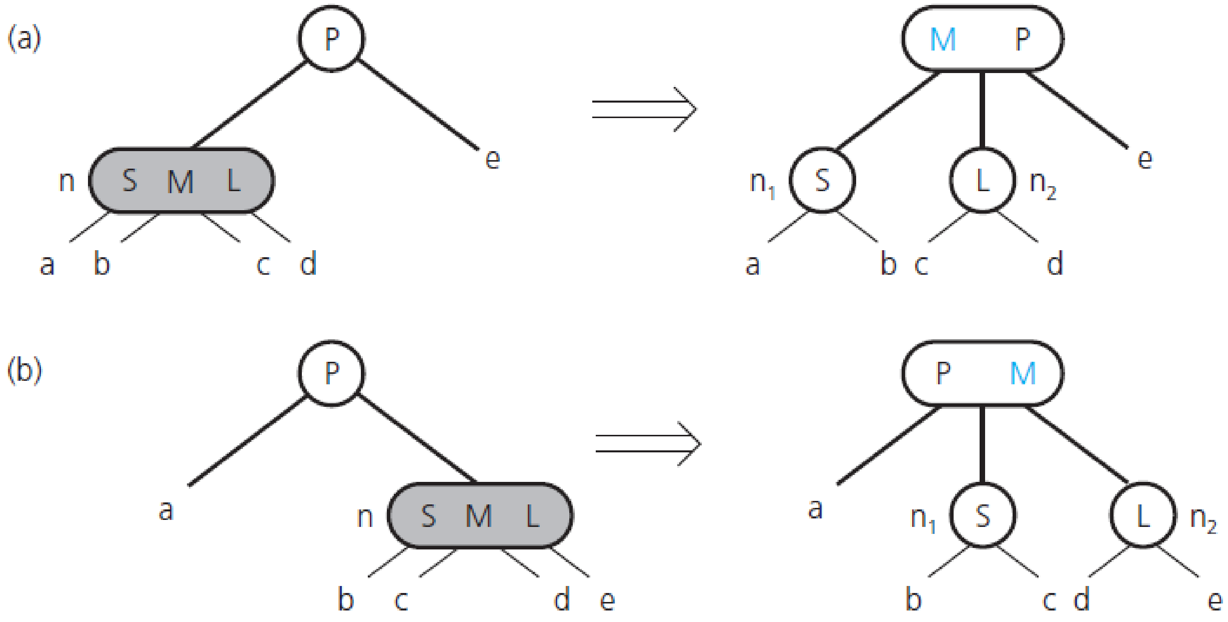
Inserting Data into a 2-3 Tree

- Splitting a leaf in a 2-3 tree when the leaf is a
  - (a) left child; (b) right child



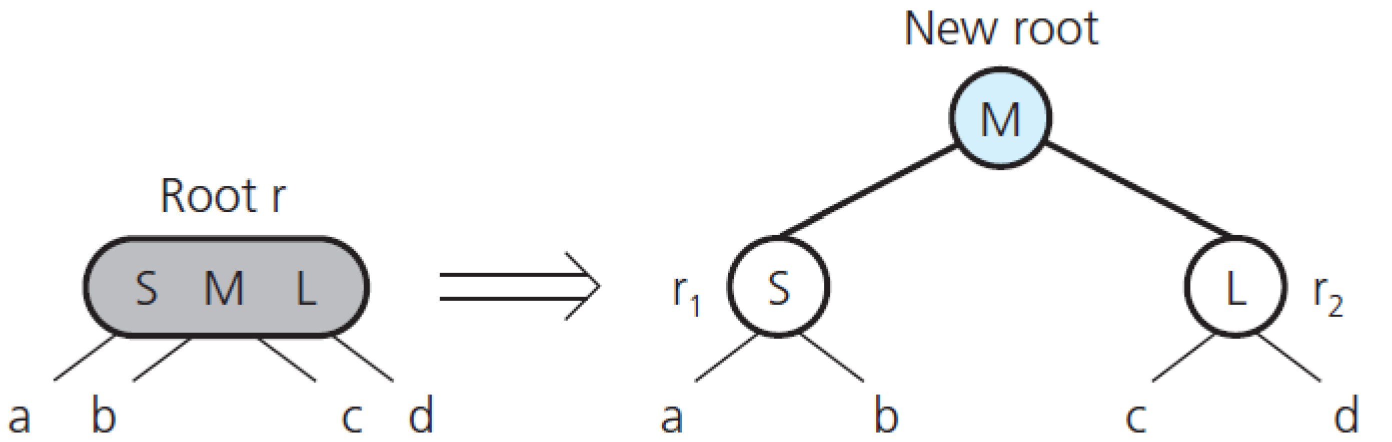
Inserting Data into a 2-3 Tree

- Splitting an internal node in a 2-3 tree when the node is a (a) left child; (b) right child



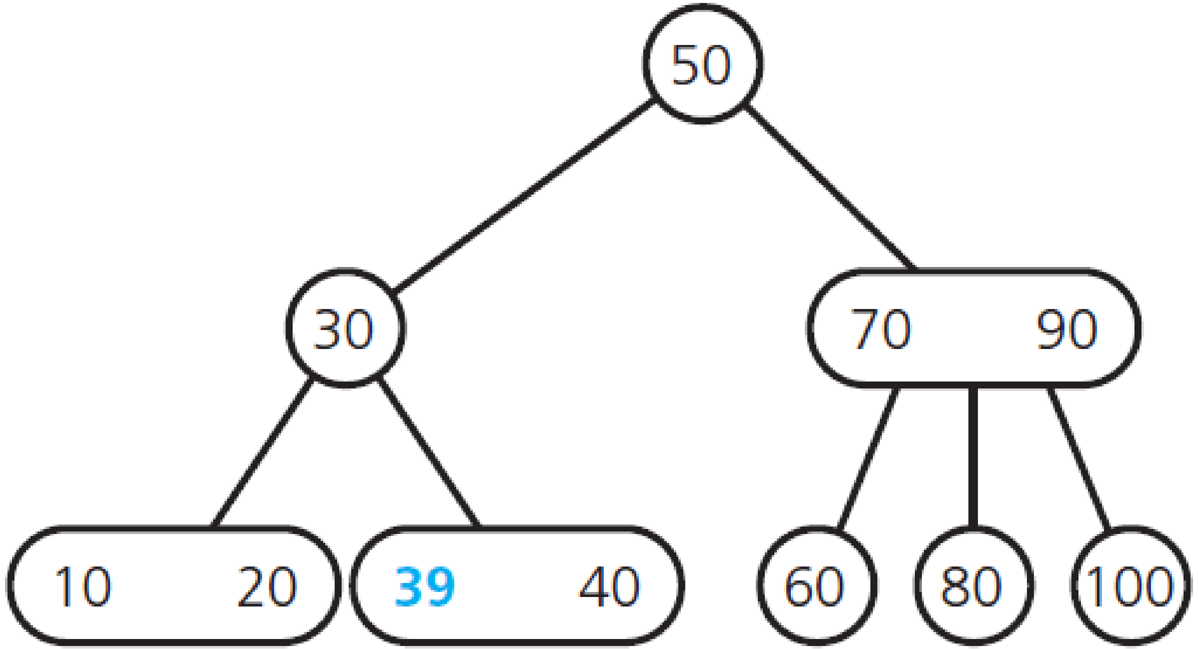
Inserting Data into a 2-3 Tree

- Splitting the root of a 2-3 tree



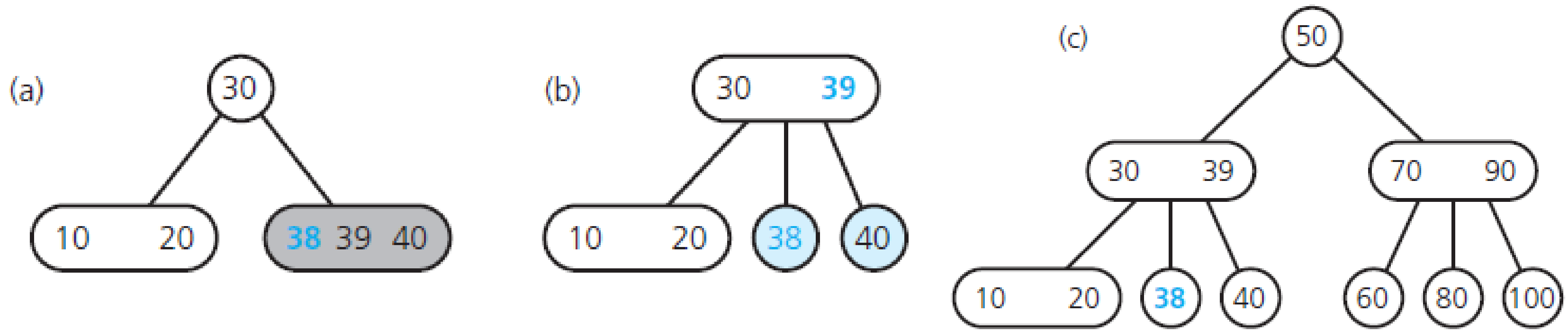
Examples

- After inserting 39 into the tree



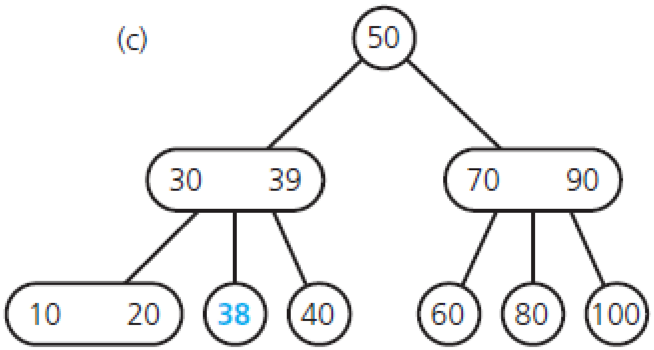
Examples

- The steps for inserting 38 into the tree
  - (a) The located node has no room;
  - (b) the node splits;
  - (c) the resulting tree

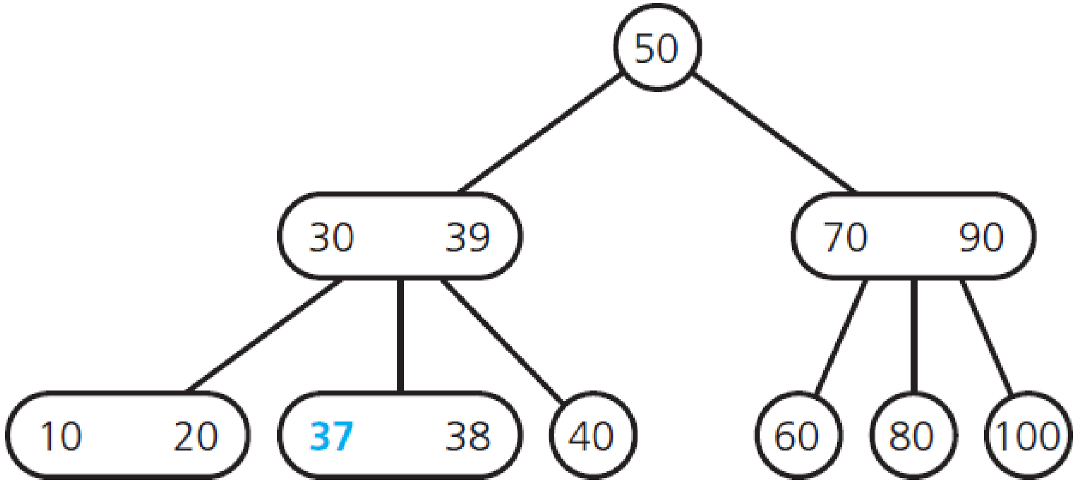


Examples

- After inserting 37 into the tree



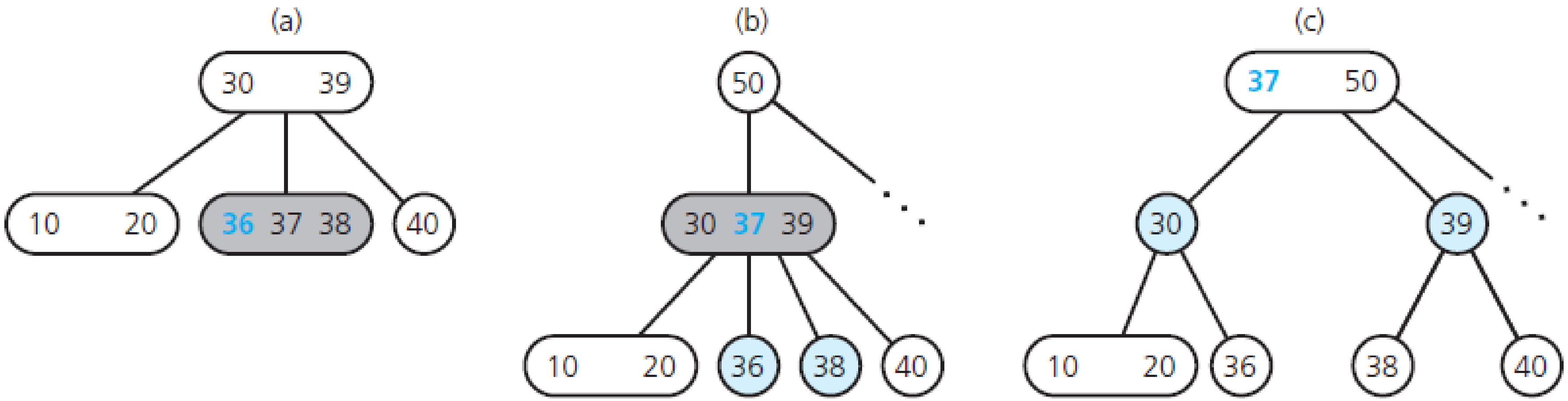
Original tree



Resulting tree

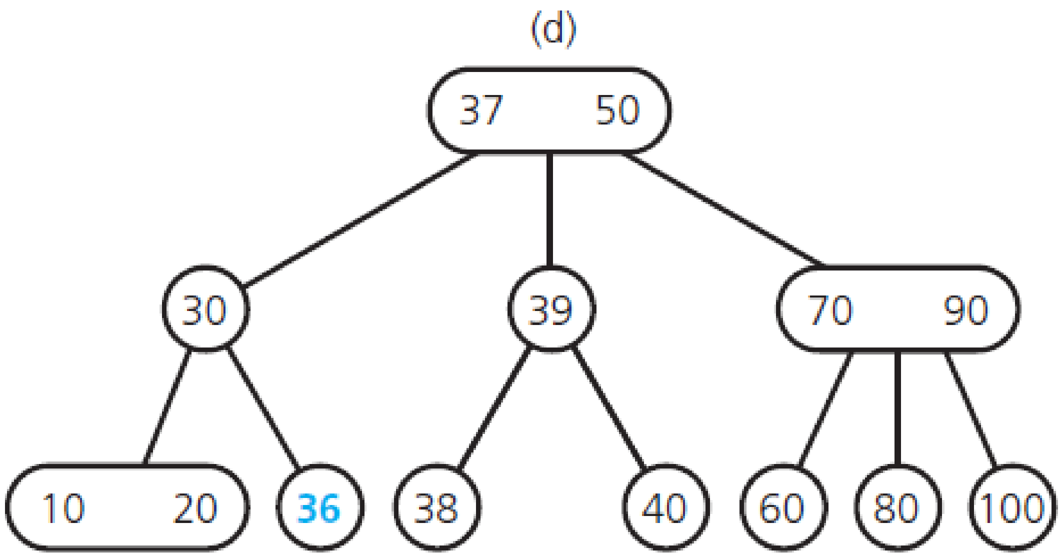
Examples

- The steps for inserting 36 into the tree



Examples

- the resulting tree

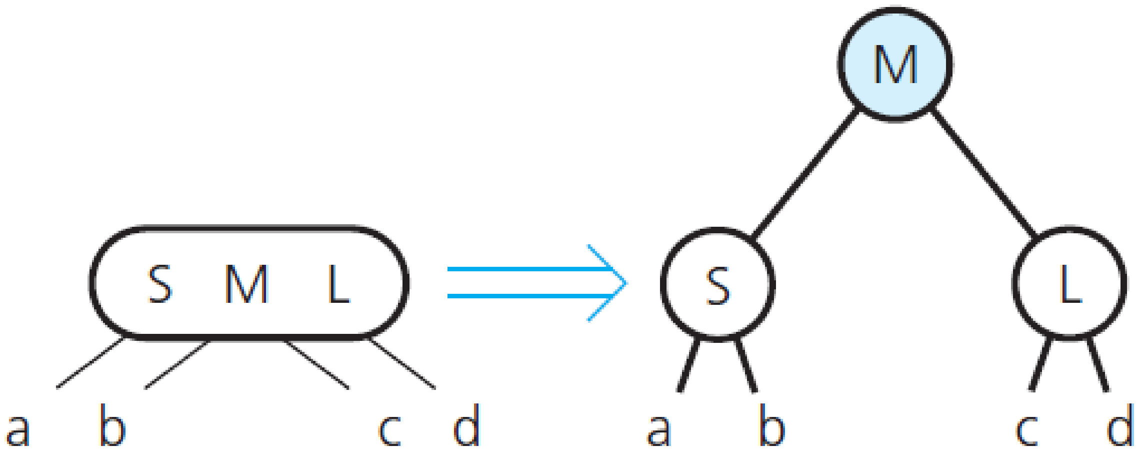


Inserting Data into a 2-3-4 Tree

- Insertion algorithm splits a node by moving one of its items up to its parent node
- Splits 4-nodes as soon as it encounters them on the way down the tree from the root to a leaf

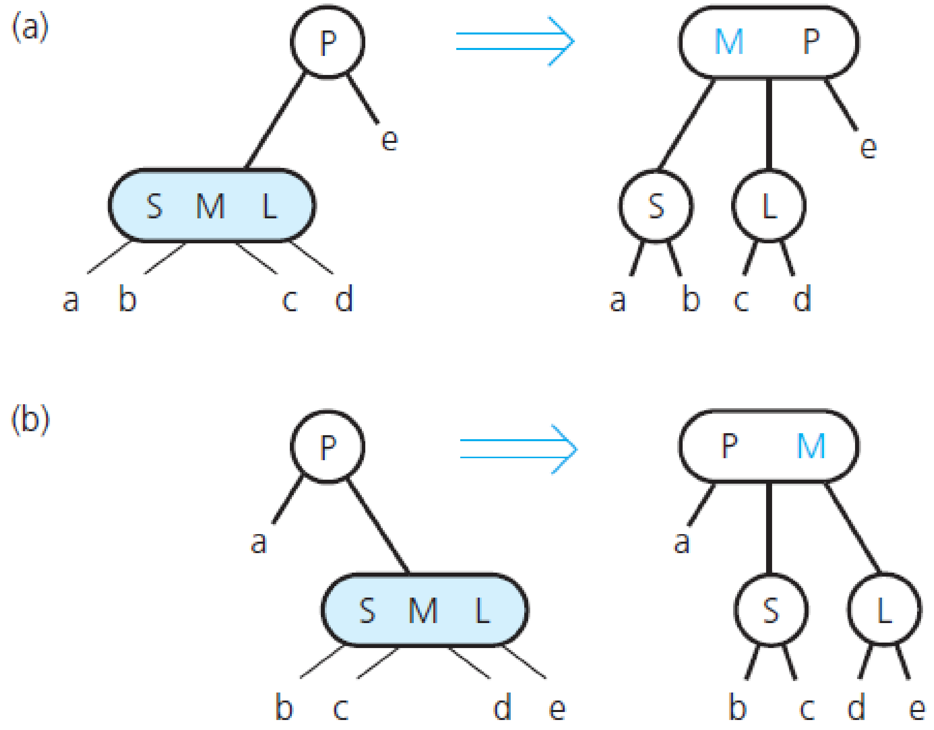
Inserting Data into a 2-3-4 Tree

- Splitting a 4-node root during insertion into a 2-3-4 tree



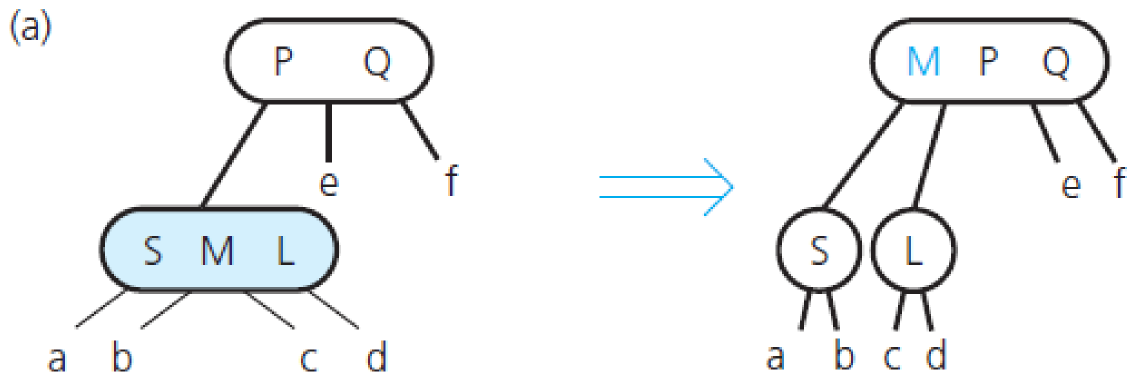
Inserting Data into a 2-3-4 Tree

- Splitting a 4-node whose parent is a 2-node during insertion into a 2-3-4 tree, when the 4-node is a (a) left child; (b) right child



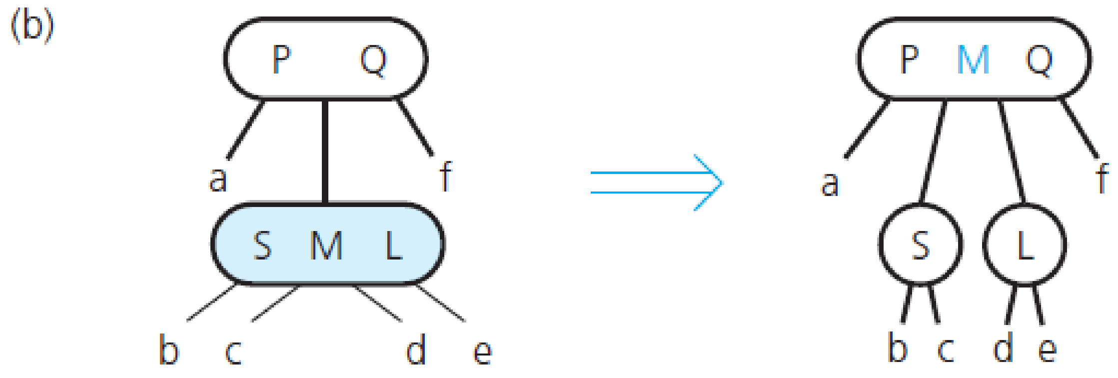
Inserting Data into a 2-3-4 Tree

- Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (a) left child



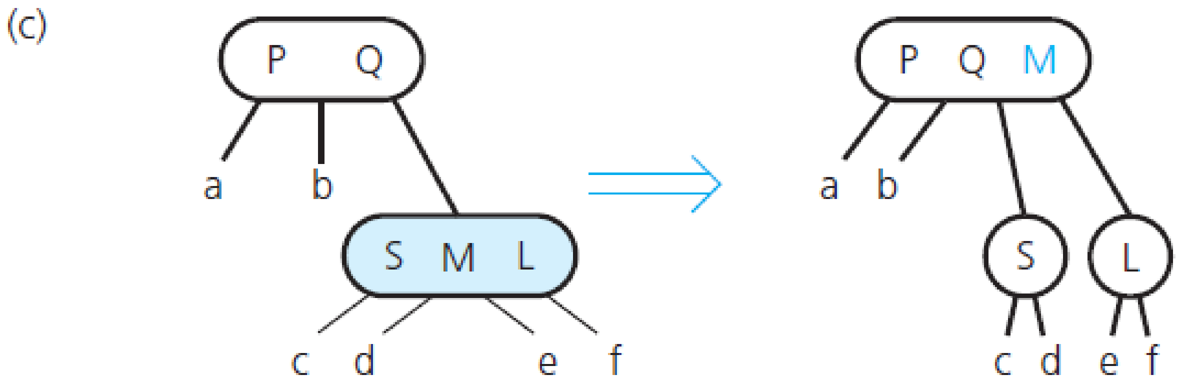
Inserting Data into a 2-3-4 Tree

- Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (b) middle child



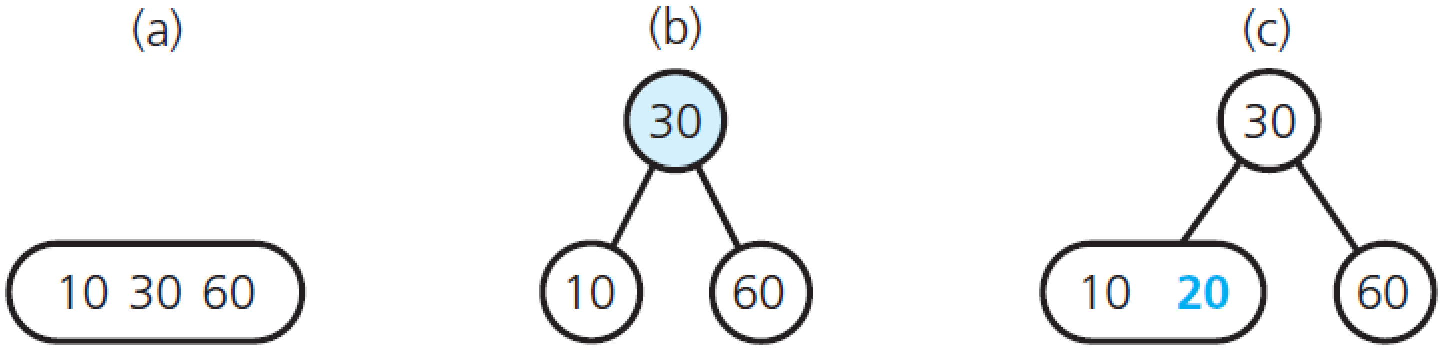
Inserting Data into a 2-3-4 Tree

- Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (c) right child



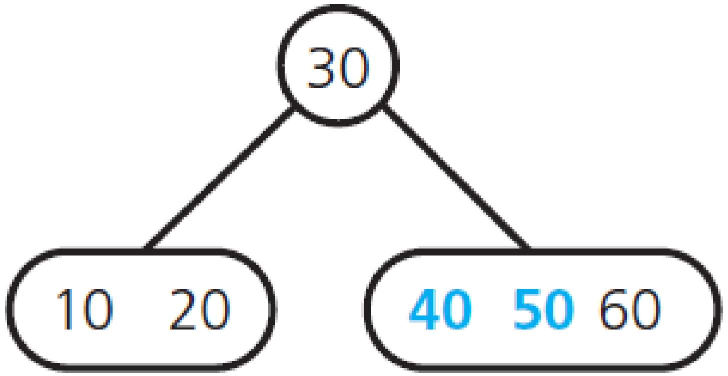
Examples

- Inserting 20 into a 2-3-4 tree
  - (a) the original tree;
  - (b) after splitting the node;
  - (c) after inserting 20



Examples

- After inserting 50 and 40 into the tree

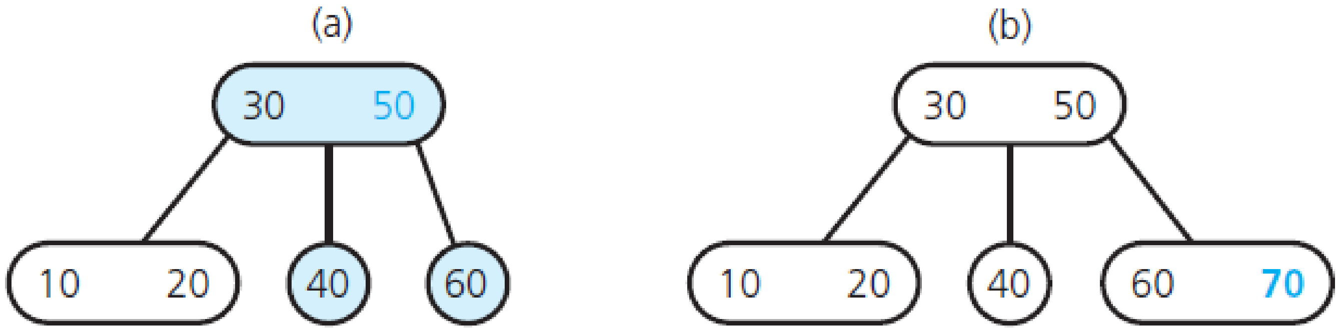


Examples

- The steps for inserting 70 into the tree

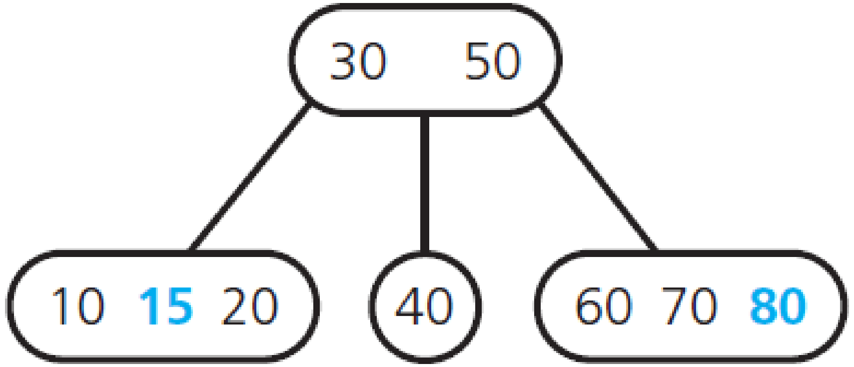
  (a) after splitting the 4-node;

  (b) after inserting 70



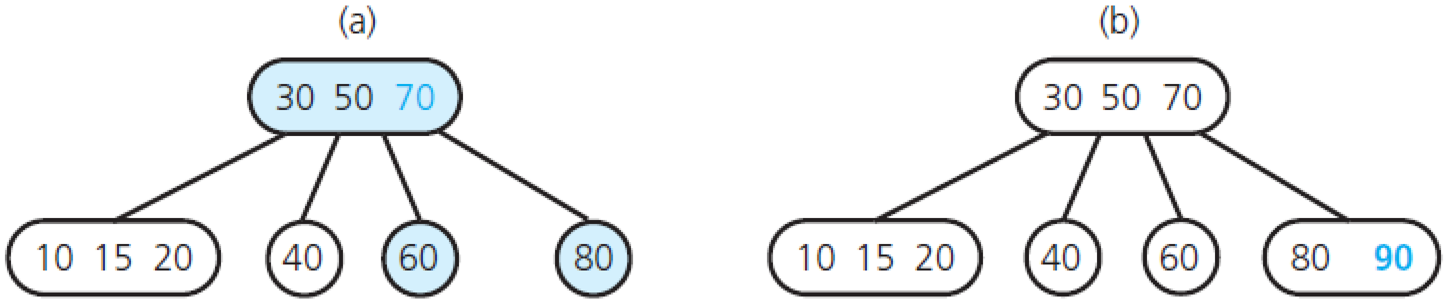
Examples

- After inserting 80 and 15 into the tree



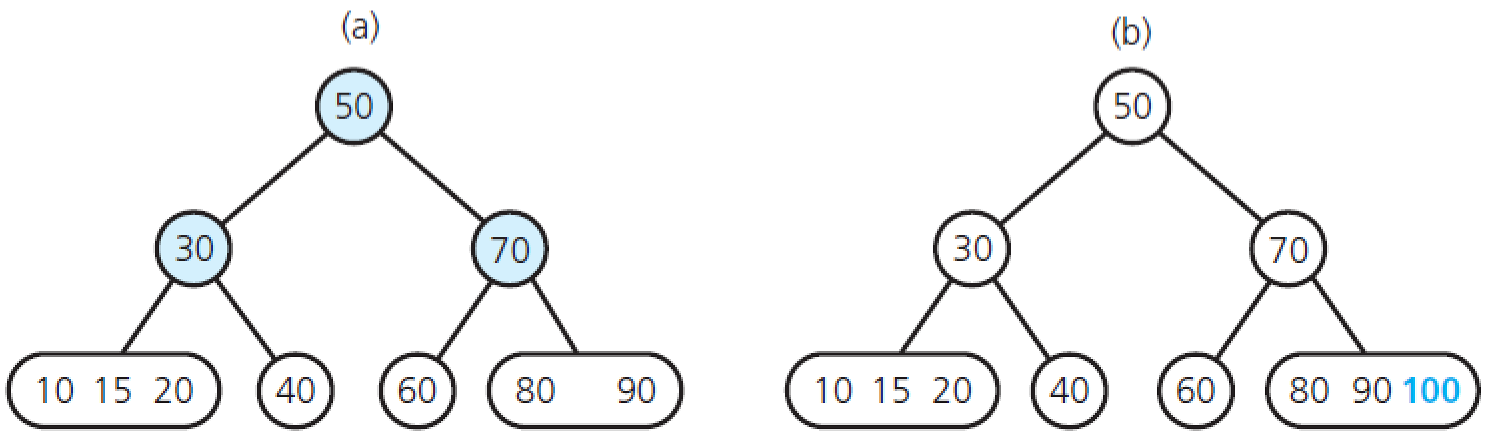
Examples

- The steps for inserting 90 into the tree



Examples

- The steps for inserting 100 into the tree



Removal Operation

Removing Data from a 2-3 Tree

```
// Removes the given data item from a 2-3 tree. Returns true if successful
// or false if no such item exists.
removeItem(23Tree: TwoThreeTree, dataItem: ItemType): boolean

   Attempt to locate dataItem
   if (dataItem is found)
   {
      if (dataItem is not in a leaf)
         Swap dataItem with its inorder successor, which will be in a leaf leafNode

      // The removal always begins at a leaf
      Remove dataItem from leaf leafNode
      if (leafNode now has no items)
         fixTree(leafNode)
      return true
   }
   else
      return false
```

Removing Data from a 2-3 Tree

```
    else
        return false

// Completes the removal when node n is empty by either deleting the root,
// redistributing values, or merging nodes. Note: If n is internal, it has one child.
fixTree(n: TwoThreeNode)

    if (n is the root)
        Delete the root
    else
    {
        Let p be the parent of n
        if (some sibling of n has two items)
        {
            Distribute items appropriately among n, the sibling, and p
```

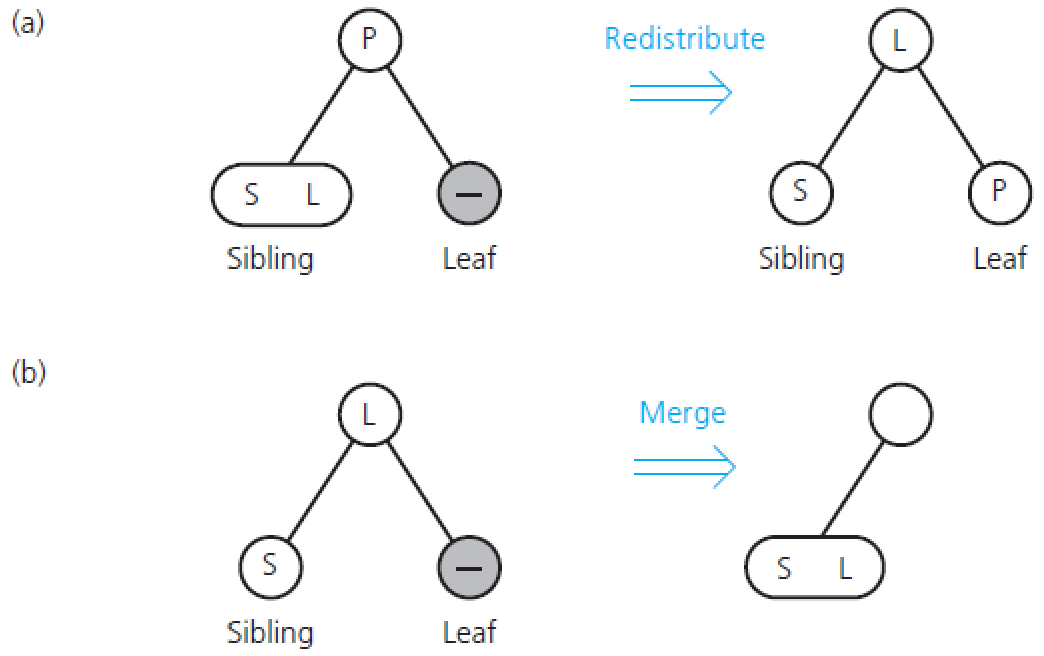
Removing Data from a 2-3 Tree

```
        if (n is internal)
            Move the appropriate child from sibling to n
    }
    else // Merge the node
    {
        Choose an adjacent sibling s of n
        Bring the appropriate item down from p into s
        if (n is internal)
            Move n's child to s
        Remove node n
        if (p is now empty)
            fixTree(p)
    }
}
```

Removing Data from a 2-3 Tree

(a) Redistributing values;

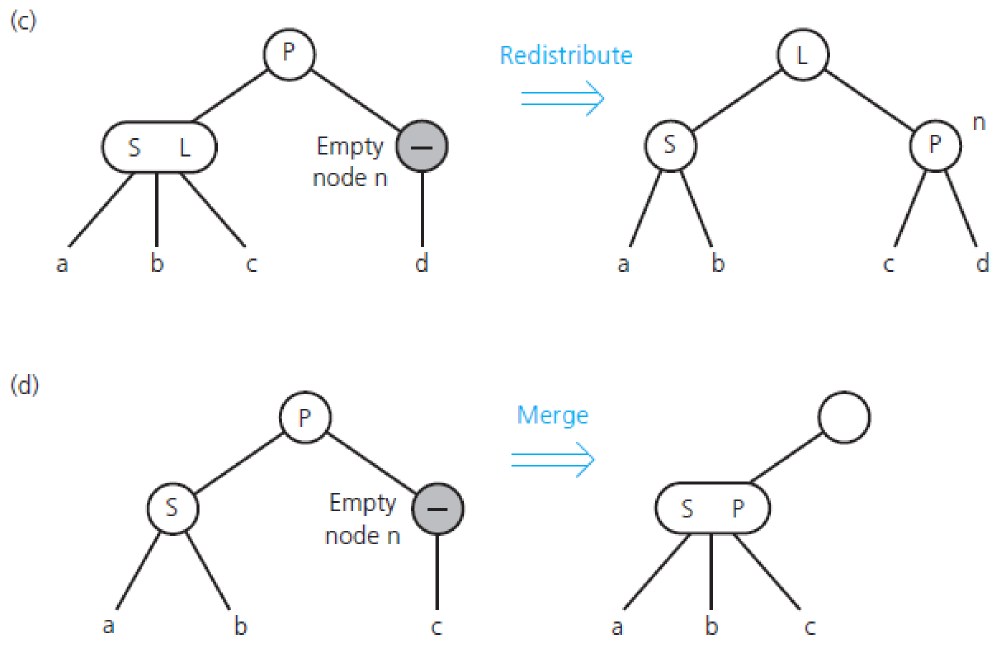
(b) merging a leaf:



Removing Data from a 2-3 Tree

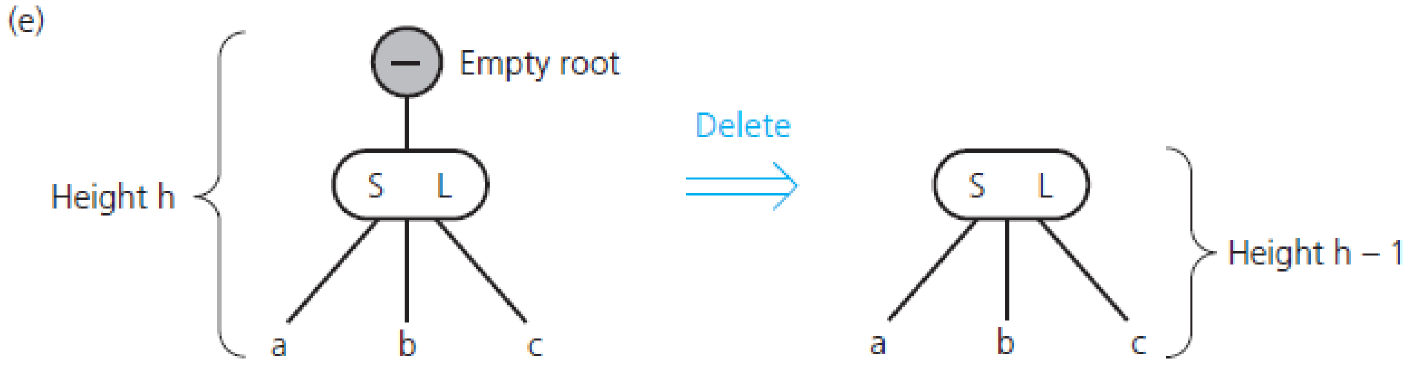
(c) redistributing values and children

(d) merging internal nodes



Removing Data from a 2-3 Tree

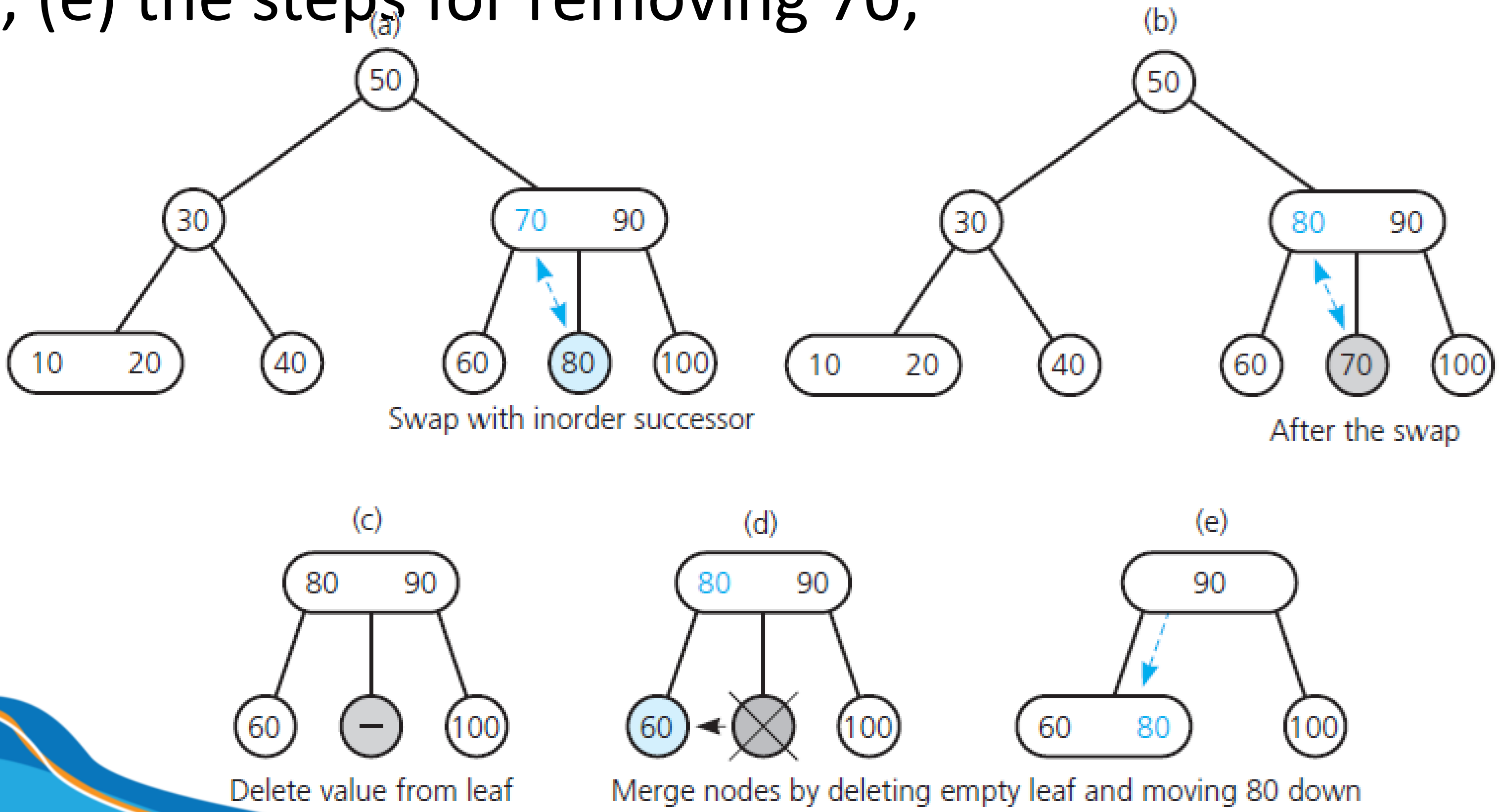
(e) deleting the root



Examples

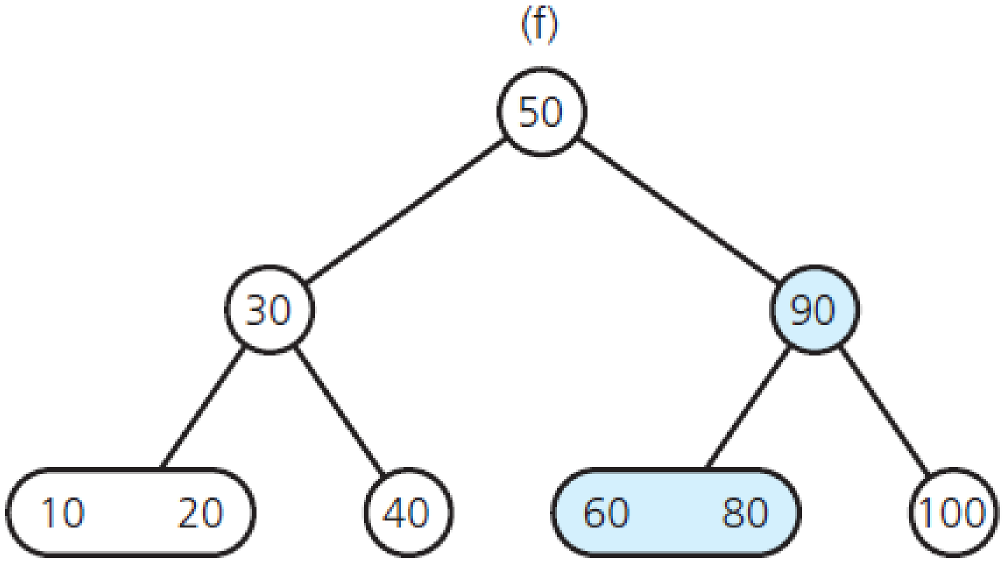
(a) A 2-3 tree;

(b), (c), (d), (e) the steps for removing 70;



Examples

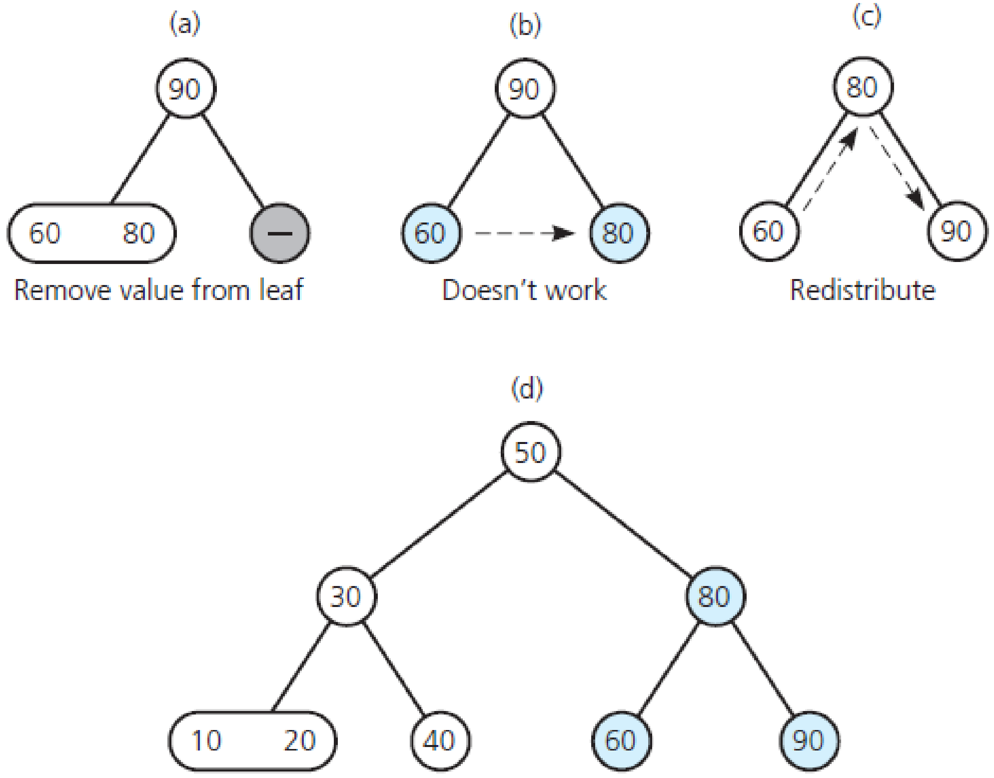
- the resulting tree



Examples

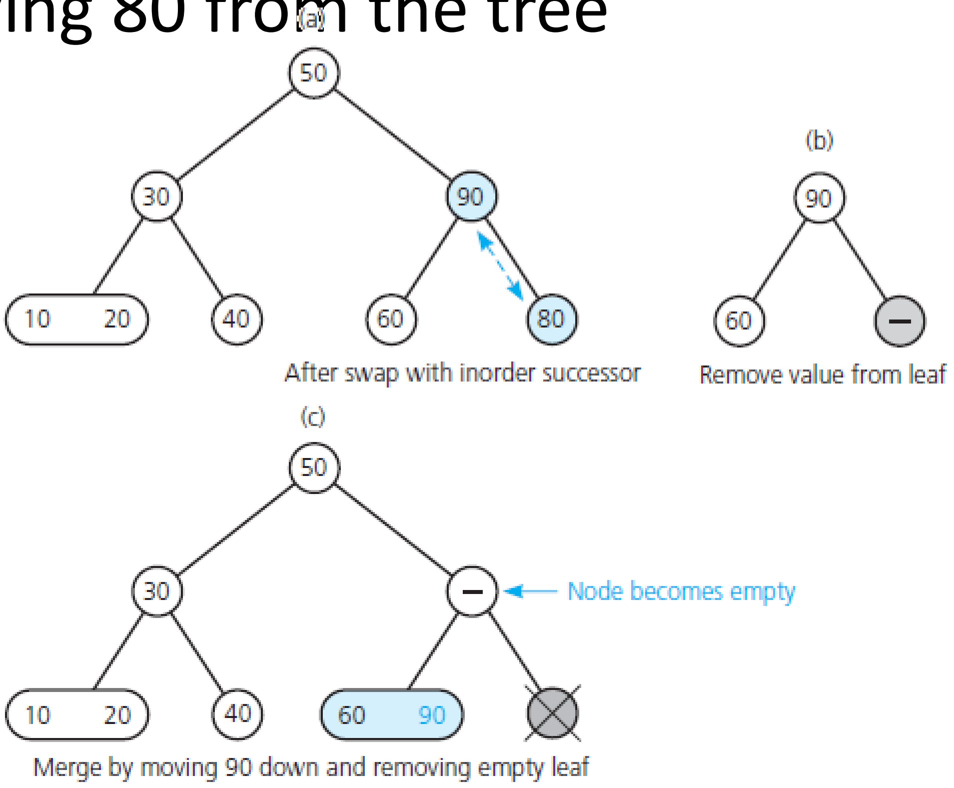
(a), (b), (c) The steps for removing 100 from the tree;

(d) the resulting tree



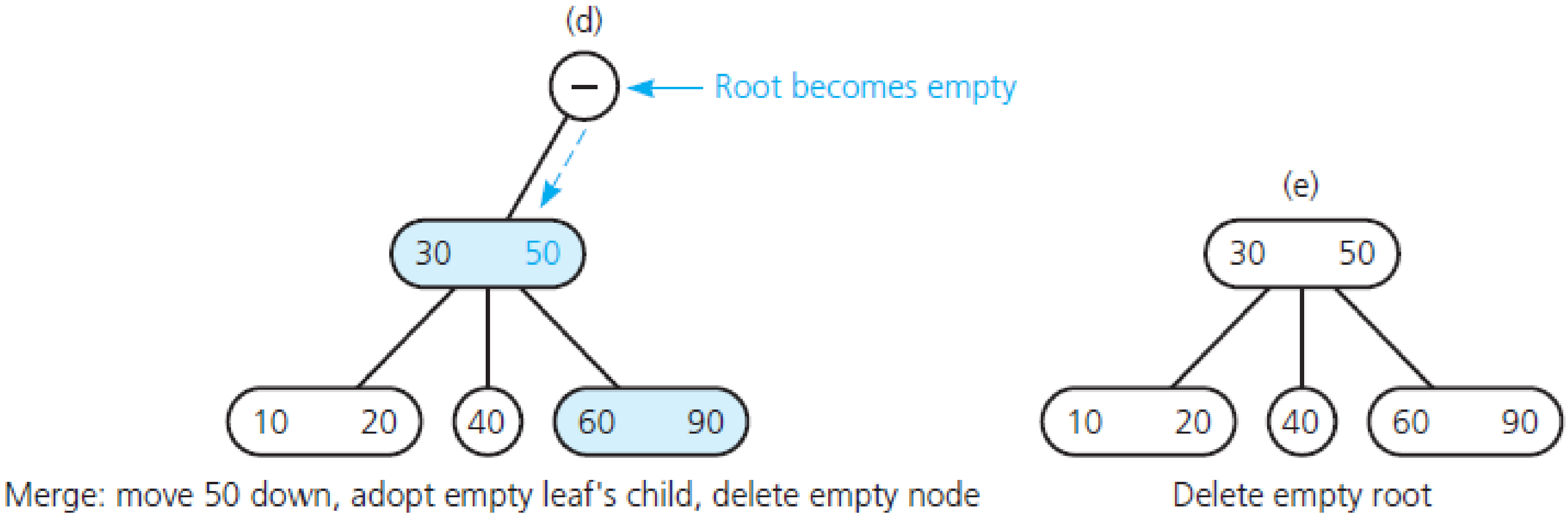
Examples

The steps for removing 80 from the tree



Examples

The steps for removing 80 from the tree



Removing Data from a 2-3-4 Tree

- Removal algorithm has same beginning as removal algorithm for a 2-3 tree
- Locate the node `n` that contains the item `I` you want to remove.
- Find `I` 's in-order successor and swap it with `I` so that the removal will always be at a leaf.
- If leaf is either a 3-node or a 4-node, remove `I` .

Questions and Answers