Binary Trees

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

Karachi Campus

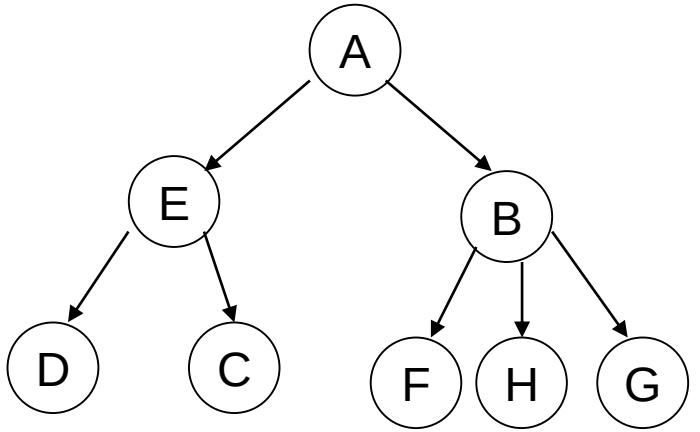
Trees

- What is a Tree?
- Tree terminology
- Why trees?
- What is a N-ary tree?
- Implementing trees
- Binary trees
- Binary tree implementation
- Application of Binary trees
- Binary Search Tree

What is a Tree?

- A non-linear data structure for data with some hierarchy in it.
- A tree, is a finite set of nodes together with a finite set of directed edges that define parent-child relationships. Each directed edge connects a parent to its child. Example:

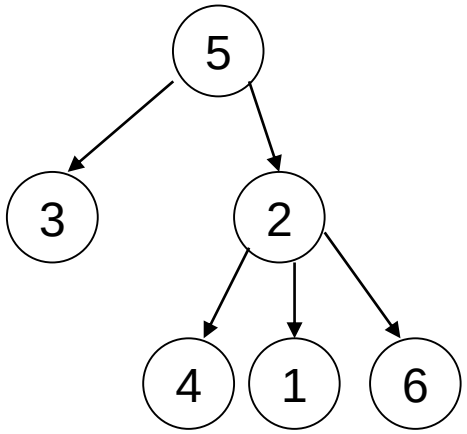
Nodes={A,B,C,D,E,f,G,H}
Edges={(A,B),(A,E),(B,F),(B,G),(B,H),(E,C),(E,D)}



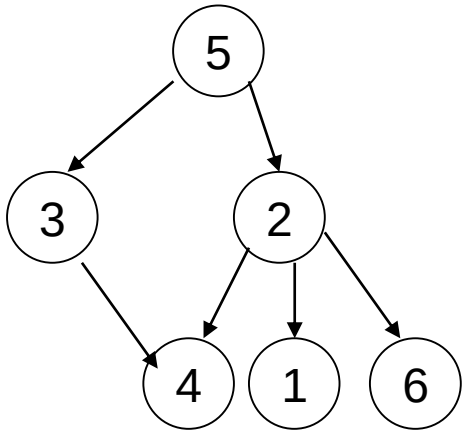
- A directed **path** from node m1 to node m_k is a list of nodes $m_1, m_2, \ldots, m_k$ such that each is the parent of the next node in the list. The length of such a path is k - 1.
- Example: A, E, C is a directed path of length 2.

What is a Tree?

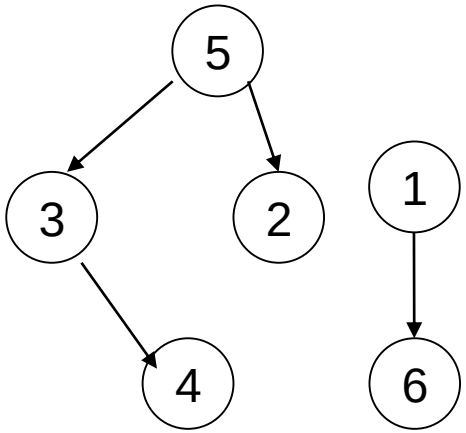
- A tree satisfies the following properties:
  1. It has one designated node, called the root, that has no parent.
  2. Every node, except the root, has exactly one parent.
  3. A node may have zero or more children.
  4. There is a unique directed path from the root to each node.



tree



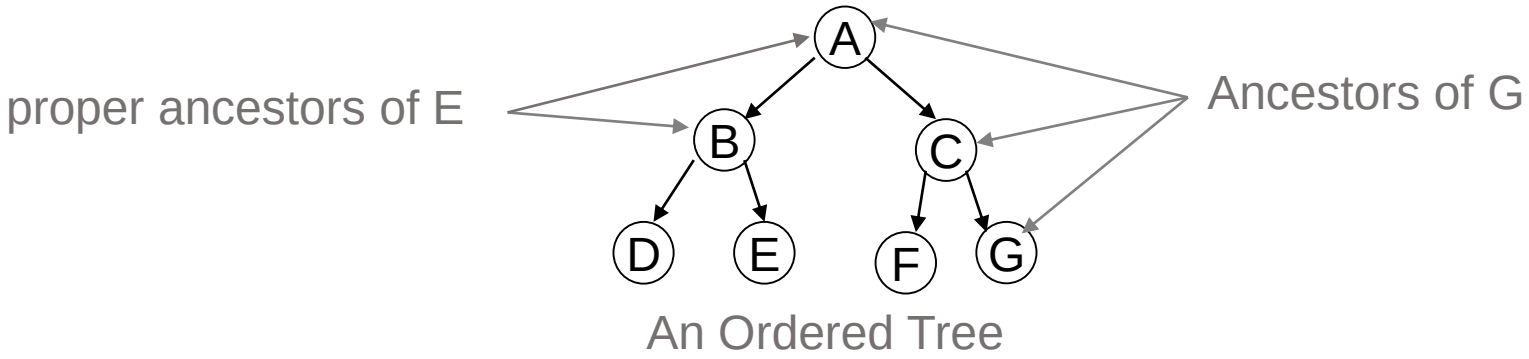
Not a tree



Not a tree

Tree Terminology

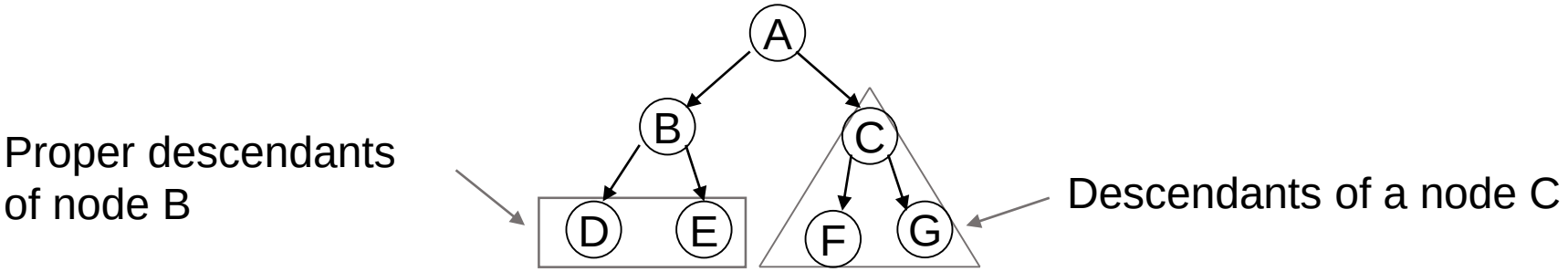
- Ordered tree: A tree in which the children of each node are linearly ordered (usually from left to right).



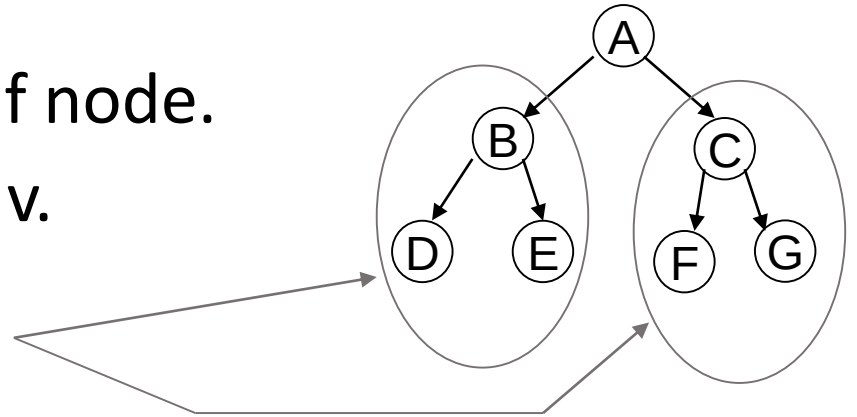
- **Ancestor** of a node v: Any node, including v itself, on the path from the root to the node.
- **Proper ancestor** of a node v: Any node, excluding v, on the path from the root to the node.

Tree Terminology (Contd.)

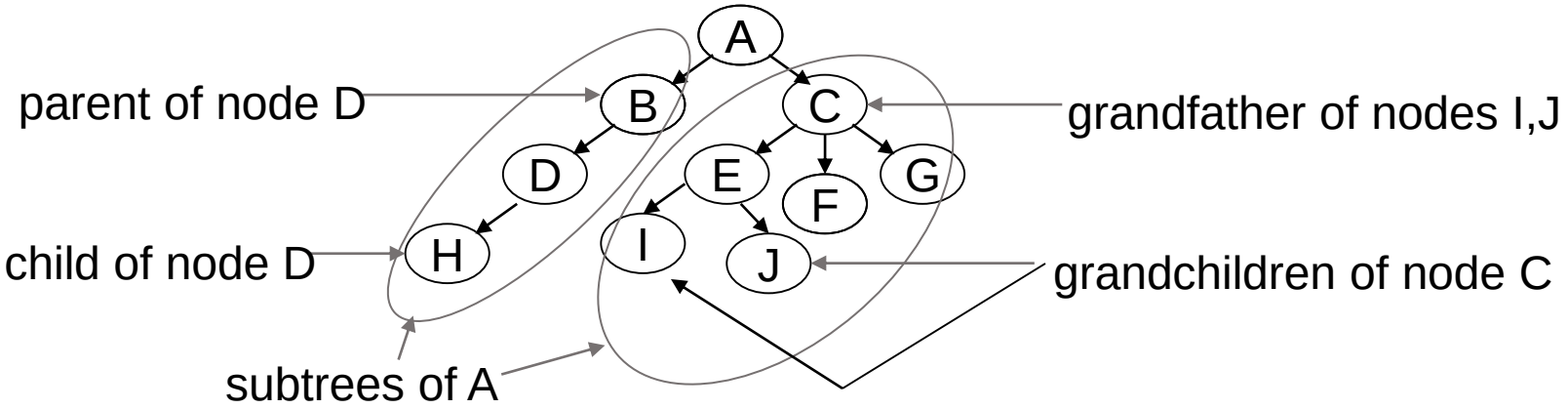
- **Descendant** of a node v: Any node, including v itself, on any path from the node to a leaf node (i.e., a node with no children).

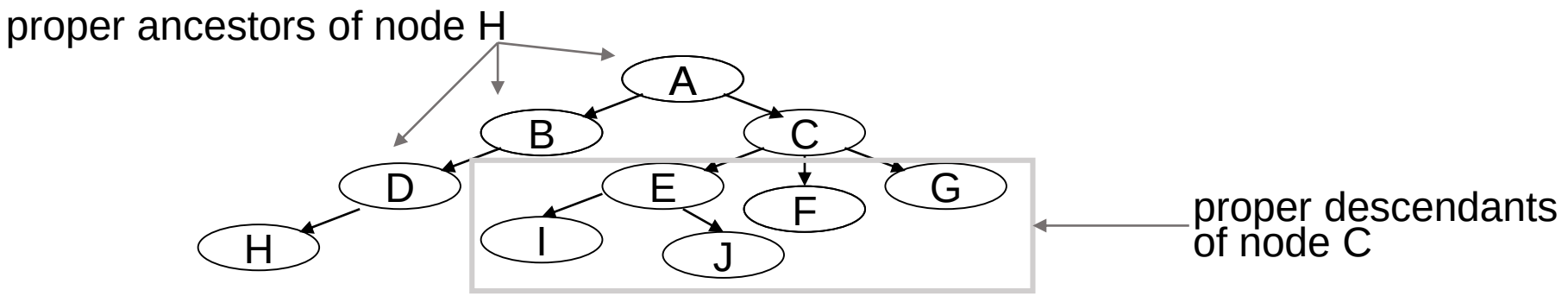


- **Proper descendant** of a node v: Any node, excluding v, on any path from the node to a leaf node.
- **Subtree** of a node v: A tree rooted at a child of v.



Tree Terminology (Contd.)

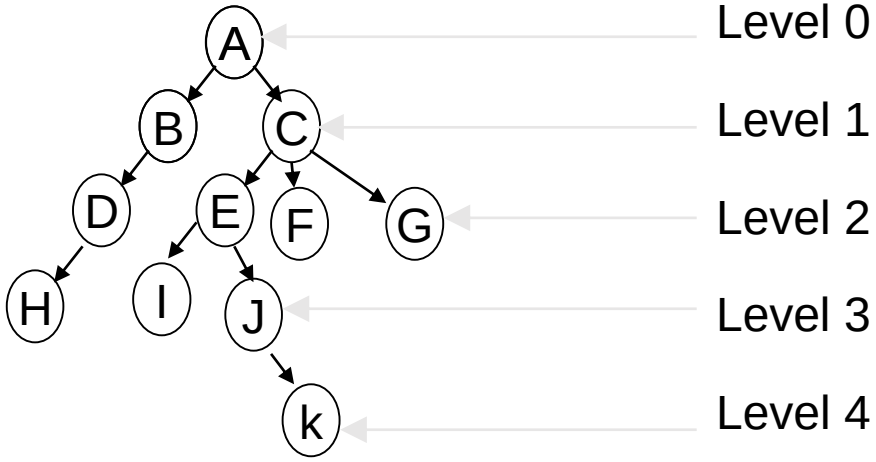




Tree Terminology (Contd.)

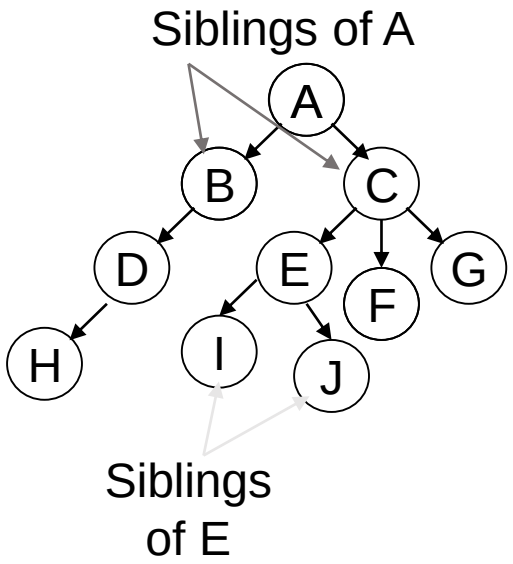
- **Level** (or depth) of a node v: the length of the path from the root to v.
- **Height** of a node v: The length of the longest path from v to a leaf node.
  - The **height of a tree** is the height of its root node.
  - By definition the height of an empty tree is -1.

- The height of the tree is 4.
- The height of node C is 3.



Tree Terminology (Contd.)

An Ordered Tree with size of 10



- **Degree**: The number of subtrees of a node
  - Each of node D and B has degree 1.
  - Each of node A and E has degree 2.
  - Node C has degree 3.
  - Each of node F,G,H,I,J has degree 0.
- **Leaf**: A node with degree 0.
- **Internal** or interior node: a node with degree greater than 0.

  A node with at least one child.
- **External node** – A node with no children.
- **Edge** – Connection between one node to another.
- **Siblings**: Nodes that have the same parent.
- **Size**: The number of nodes in a tree.
- **Depth** – The depth of a node is the number of edges from the node to the root node.

Why Trees?

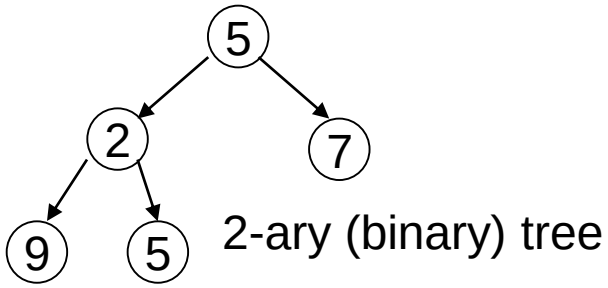
- Trees are very important data structures in computing.
- They are suitable for:
  - Hierarchical structure representation, e.g.,
    - File directory.
    - Organizational structure of an institution.
    - Class inheritance tree.
  - Problem representation, e.g.,
    - Expression tree.
    - Decision tree.
  - Efficient algorithmic solutions, e.g.,
    - Search trees.
    - Efficient priority queues via heaps.

Why Trees?

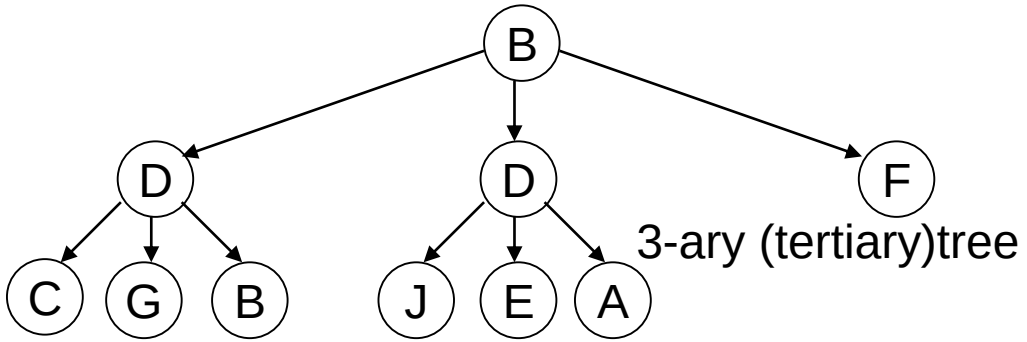
- Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
- Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
- Like Linked Lists and unlike Arrays, Trees don’t have an upper limit on number of nodes as nodes are linked using pointers.

N-ary Trees

- An N-ary tree is an ordered tree that is either:
  1. Empty, or
  2. It consists of a root node and at most N non-empty N-ary subtrees.
- It follows that the degree of each node in an N-ary tree is at most N.
- Example of N-ary trees:



2-ary (binary) tree



3-ary (tertiary)tree

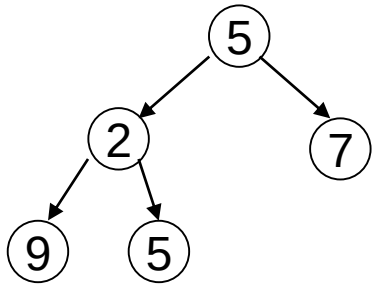
N-ary Trees Implementation

```
public static class TreeNode{
        int val;
        List<TreeNode> children = new LinkedList<>();

        TreeNode(int data){
            val = data;
        }

        TreeNode(int data,List<TreeNode> child){
            val = data;
            children = child;
        }
}
```

Binary Trees

- A binary tree is an N-ary tree for which N = 2.
- Thus, a binary tree is either:
  1. An empty tree, or
  2. A tree consisting of a root node and at most two non-empty binary subtrees.
- The Maximum number of nodes in a binary tree of height ‘h’ is $2^{h+1} - 1$.
- The maximum number of nodes at level ‘l’ of a binary tree is 2^{l}.

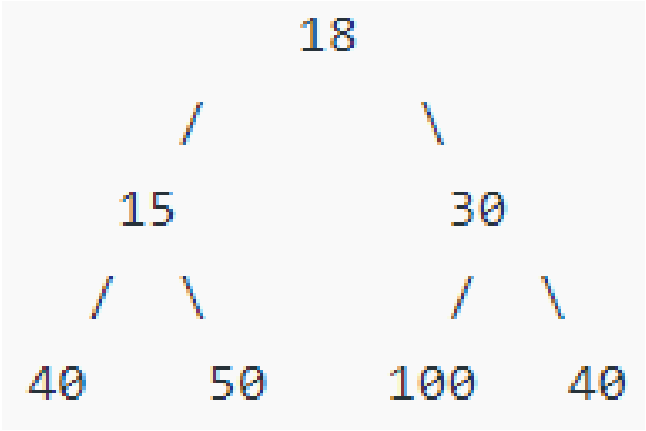
Example:

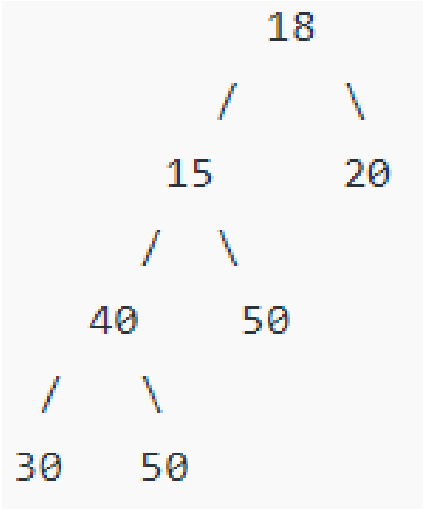


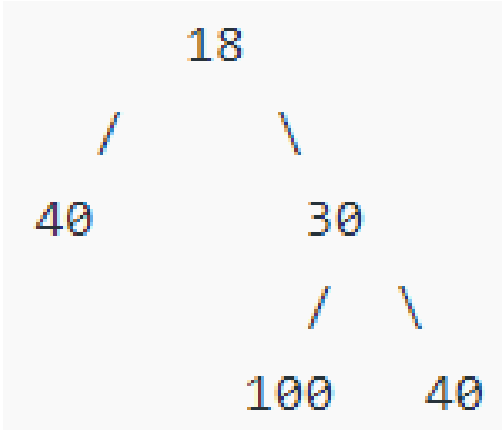
Binary Trees (Contd.)

- A full binary tree is a tree in which every node has either 0 or 2 children.
- A complete binary tree is either an empty binary tree or a binary tree in which:
  1. Each level k, $k \geq 0$, other than the last level contains the maximum number of nodes for that level, that is 2^k.
  2. The last level may or may not contain the maximum number of nodes.
  3. If a slot with a missing node is encountered when scanning the last level in a left to right direction, then all remaining slots in the level must be empty (i.e., all nodes in the last level are as far left as possible).
- (Height-) Balanced binary tree: a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

- Full binary tree

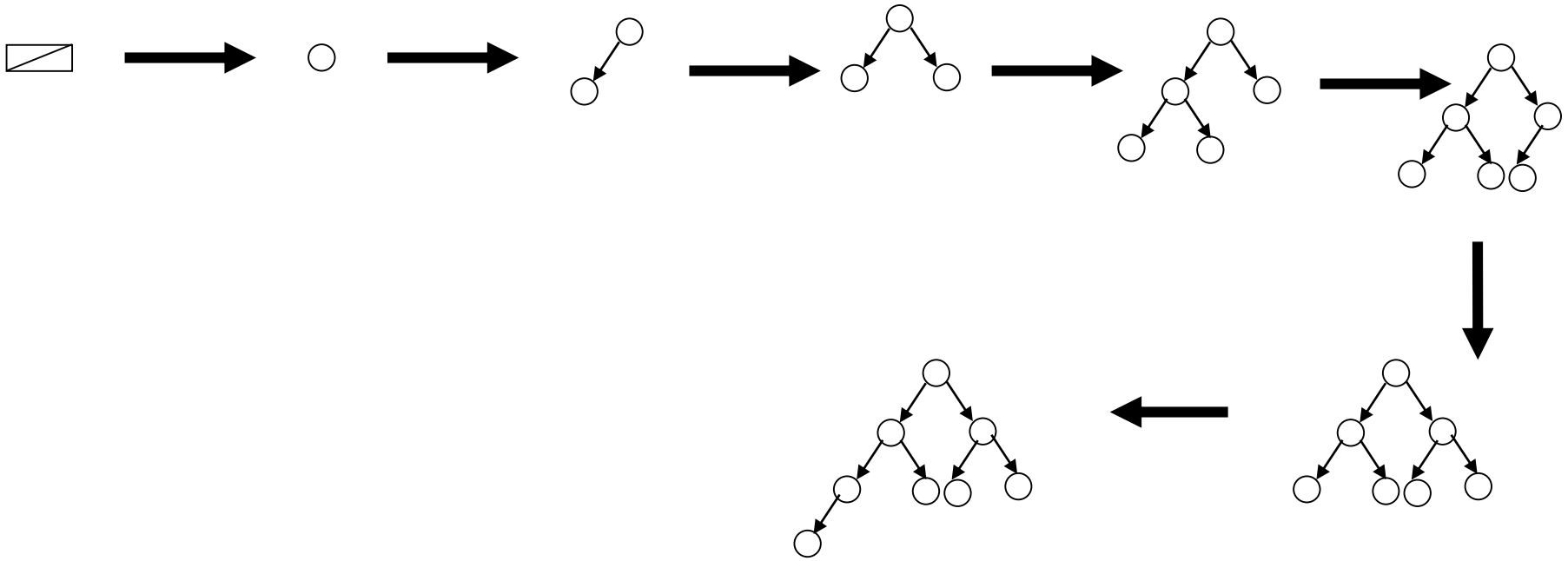






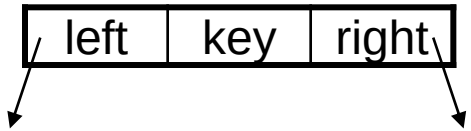
Binary Trees (Contd.)

- Example showing the growth of a complete binary tree:

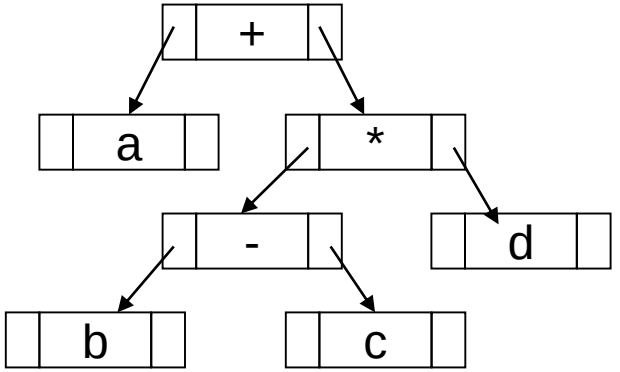


BinaryTree Node Implementation

```
class Node
{
 public:
    int key;
    Node* left;
    Node* right;

    Node(int item)
    {
        key = item;
        left = right = NULL;
    }
};
```

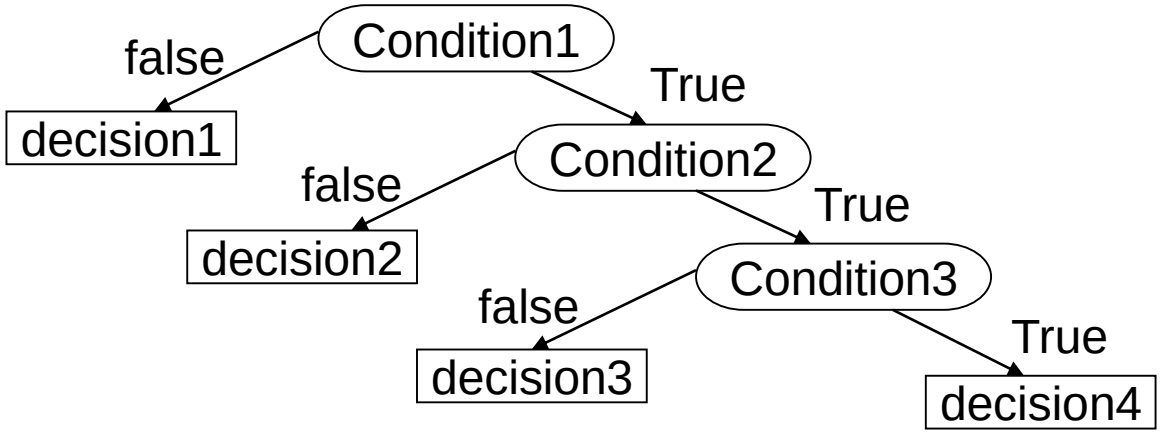


Example: A binary tree representing a + (b - c) * d

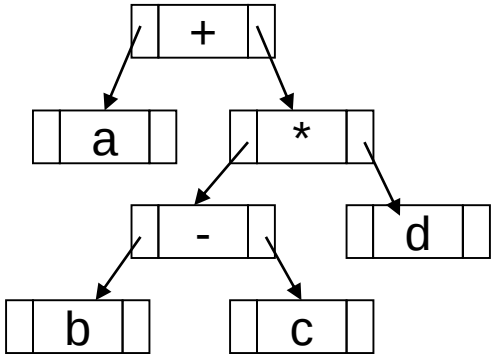


Application of Binary Trees

- Binary trees have many important uses. Two examples are:
  1. Binary decision trees.
     - Internal nodes are conditions.
     - Leaf nodes denote decisions.

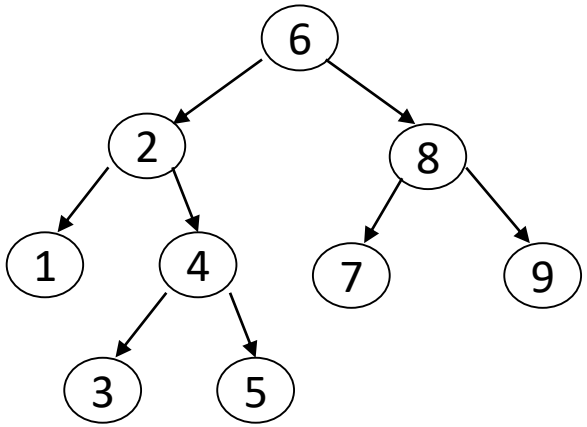


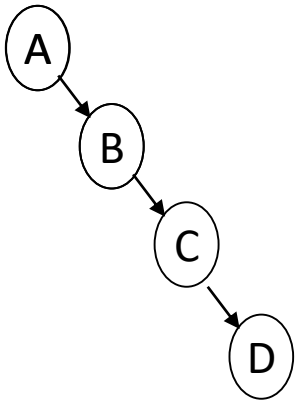
2. Expression Trees



Binary Search Trees (Definition)

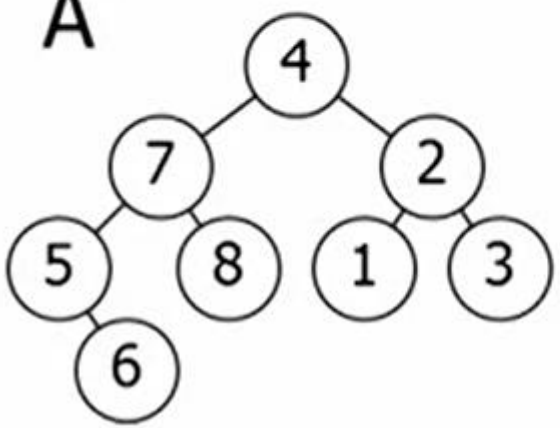
- A binary search tree (BST) is a binary tree that is empty or that satisfies the BST ordering property:
  1. The key of each node is greater than each key in the left subtree, if any, of the node.
  2. The key of each node is less than each key in the right subtree, if any, of the node.
- Thus, each key in a BST is unique.
- Examples:



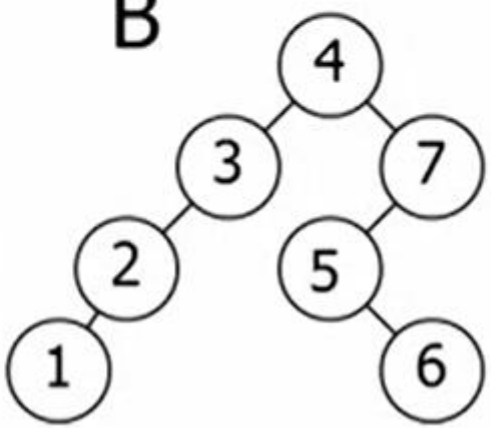


Which of the following Trees satisfies the Search Tree Property?

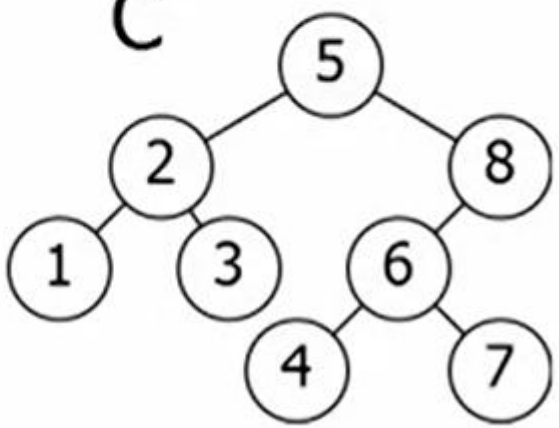
A



B



C



Why BST?

- BSTs provide good logarithmic time performance in the best and average cases.

- Average case complexities of using linear data structures compared to BSTs:

Data Structure	Retrieval	Insertion	Deletion
BST	O(log n) FAST	O(log n) FAST	O(log n) FAST
Sorted Array	O(log n) FAST*	O(n) SLOW	O(n) SLOW
Sorted Linked List	O(n) SLOW	O(n) SLOW	O(n) SLOW

*using binary search

Tree Traversal Algorithms

- *Tree traversal* is the process of visiting each node in the tree exactly one time.
- Traversal may be interpreted as putting all nodes on one line or linearizing a tree.

- **Breadth-First Search (BFS) Algorithm:** It starts from the root node and visits all nodes (from left to right) of current level before moving to the next level in the tree.

- **Depth-First Search (DFS) Algorithm:** It starts from the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this.

BST Traversal

Breadth First Traversal (Or Level Order Traversal)

- It uses a queue. After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited.
- Considering that for a node on level *n,* its children are on level *n* + 1, by placing these children at the end of the queue, they are visited after all nodes from level *n* are visited.

BST Traversal

- Depth First Traversals
  - Inorder Traversal (Left-Root-Right)
    - Traverse the left subtree, i.e., call Inorder(left-subtree)
    - Visit the root.
    - Traverse the right subtree, i.e., call Inorder(right-subtree)
  - Preorder Traversal (Root-Left-Right)
    - Visit the root.
    - Traverse the left subtree, i.e., call Preorder(left-subtree)
    - Traverse the right subtree, i.e., call Preorder(right-subtree)

BST Traversal

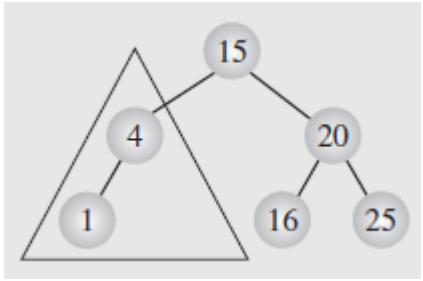
- Postorder Traversal (Left-Right-Root)
  - Traverse the left subtree, i.e., call Postorder(left-subtree)
  - Traverse the right subtree, i.e., call Postorder(right-subtree)
  - Visit the root.

BST Traversal

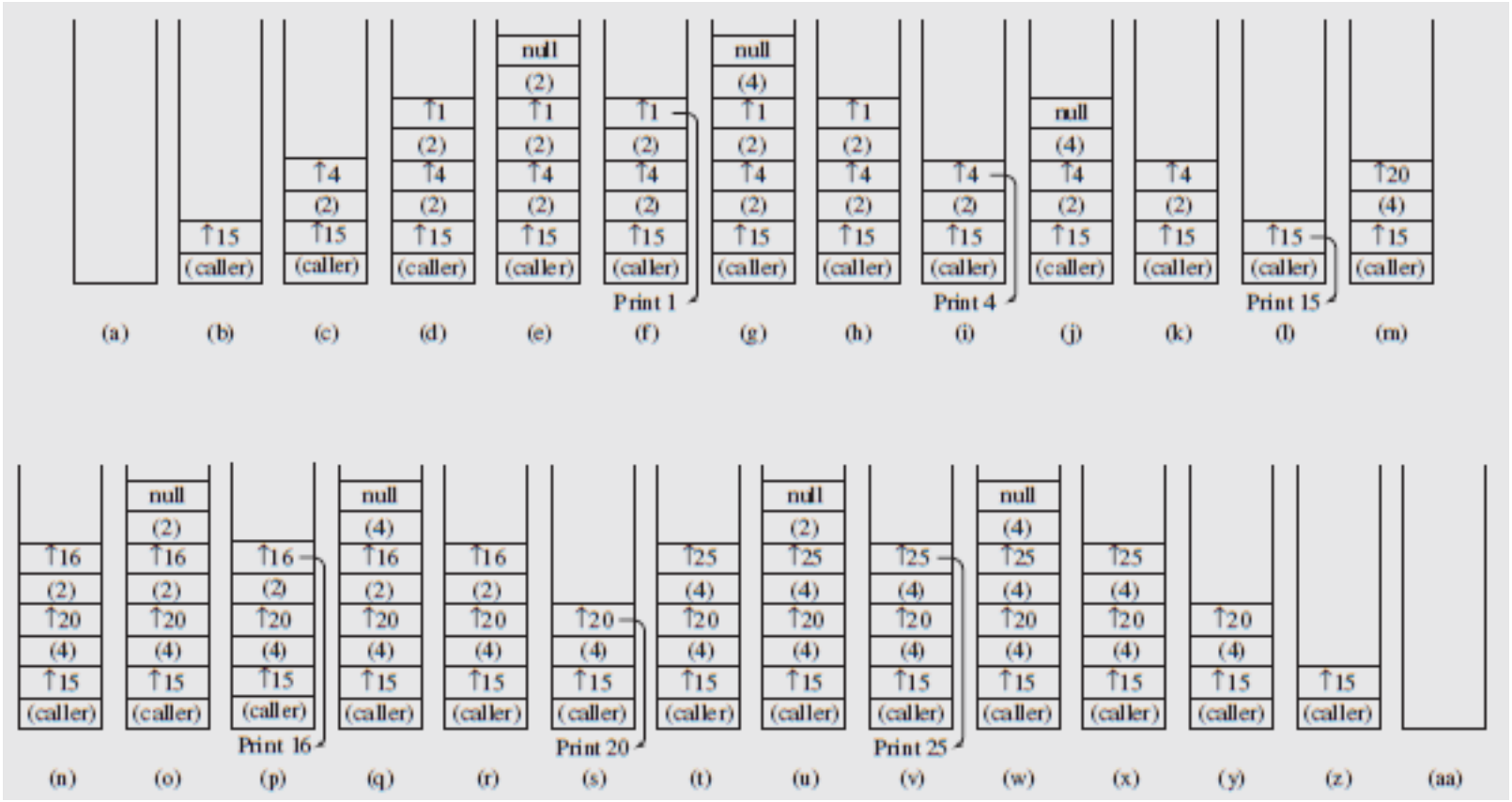
- Each traversal require O(n) time as they visit every node exactly once.
- Order level traversal requires extra space as we use a queue.
- The size of this queue in the worst case will be the number of nodes in the last level which is 2^l.
- In depth first traversal, the maximum space needed is equal to the height of the tree, which in the worst case (skewed tree) can be O(n). Note, in the worst case every node except the root node will be the left child (or right child).

A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child. There are left skewed or right skewed trees.

BST Traversal – Runtime Stack Changes




```
template<class T>
void BST<T>::inorder(BSTnode<T> *node) {
     if (node != 0) {
/* 1 */      inorder(node->left);
/* 2 */      visit(node);
/* 3 */      inorder(node->right);
/* 4 */ }
}
```



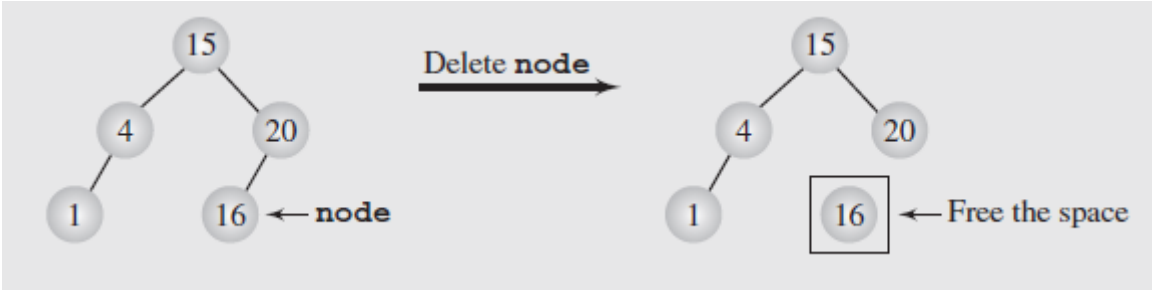
Non-recursive Implementation of Inorder Traversal

- We can clearly see the power of recursion: iterativeInorder() is almost unreadable, and without thorough explanation, it is not easy to determine the purpose of this function.

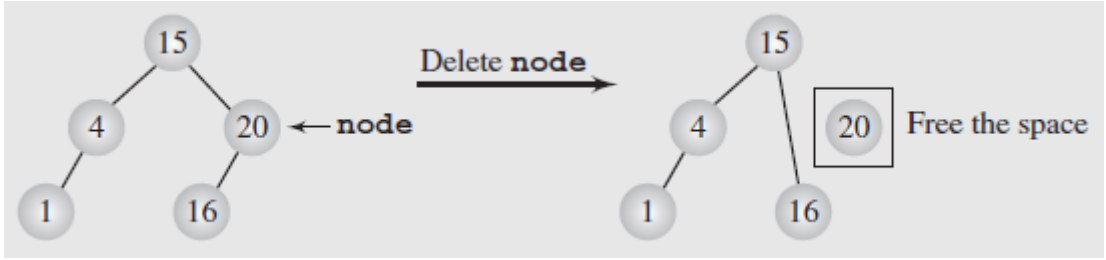
```
template<class T>
void BST<T>::iterativeInorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    while (p != 0) {
        while (p != 0) {          // stack the right child (if any)
            if (p->right)         // and the node itself when going
               travStack.push(p->right); // to the left;
            travStack.push(p);
            p = p->left;
        }
        p = travStack.pop();      // pop a node with no left child
        while (!travStack.empty() && p->right == 0) { // visit it
            visit(p);             // and all nodes with no right
            p = travStack.pop();  // child;
        }
        visit(p);                 // visit also the first node with
        if (!travStack.empty())   // a right child (if any);
             p = travStack.pop();
        else p = 0;
    }
}
```

BST - Delete

1. The node is a leaf; it has no children. The pointer of its parent is set to null and the node is deleted.



2. The node has one child. The parent's pointer to the node is reset to point to the node's child. In this way, the node's children are lifted up by one level. great-great-. . . grandchildren lose one "great".



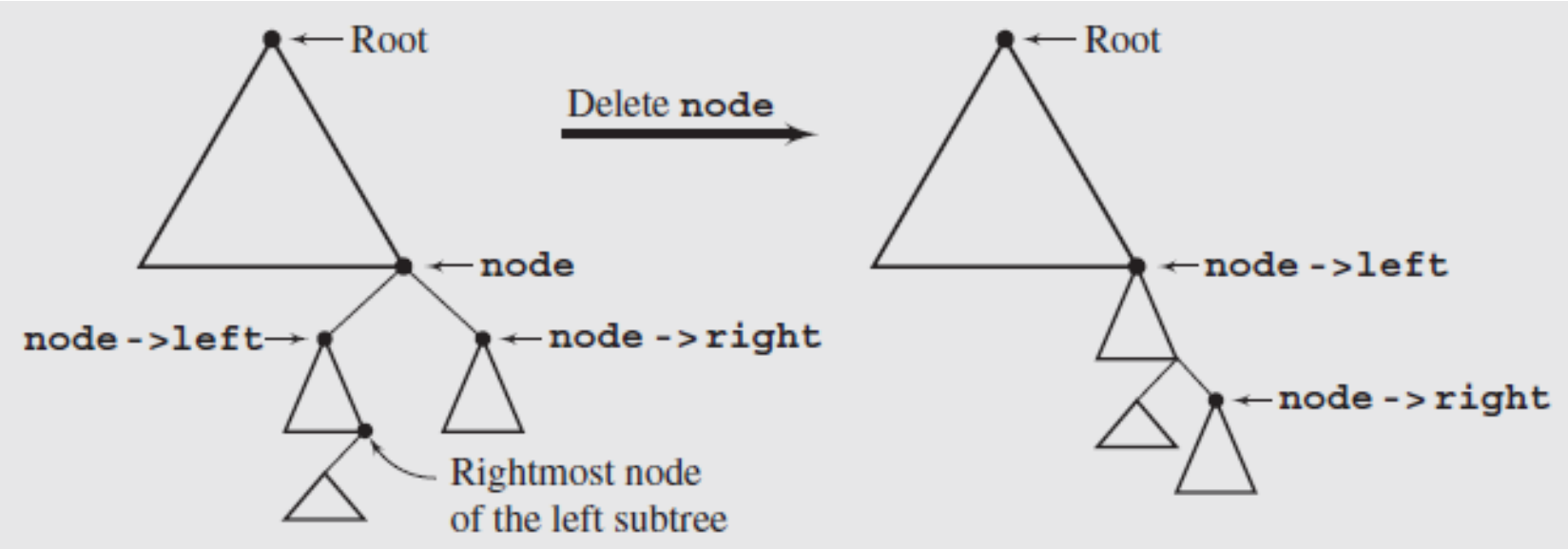
BST - Delete

- The node has two children. No one-step operation can be performed because the parent’s right or left pointer cannot point to both the node’s children at the same time. Use Delete by Merging or Delete by Copying.

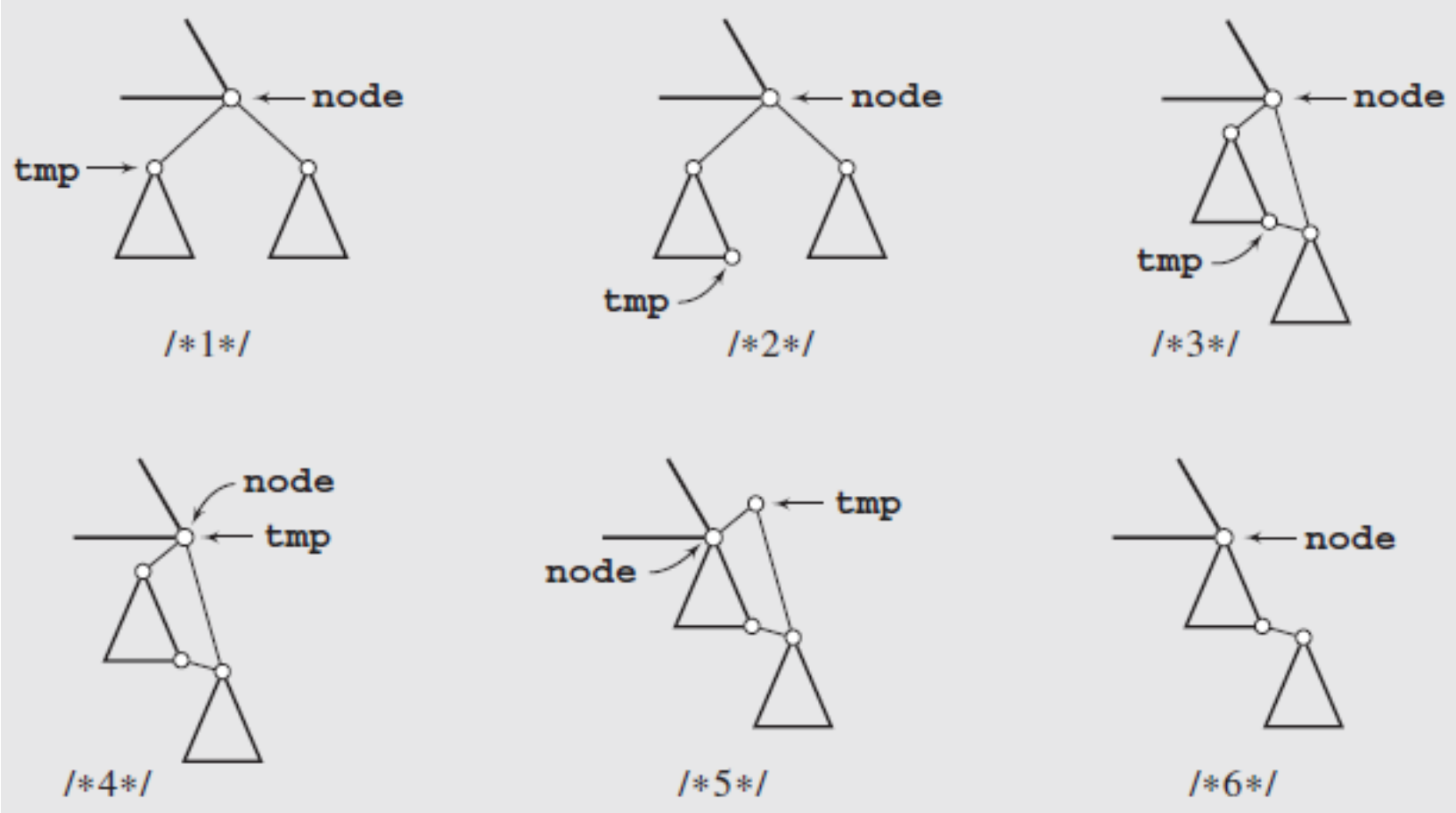
Delete by Merging

- Make one tree out of the two subtrees of the node and then attach it to the node's parent.
- Knowing that in BST every key of the right subtree is greater than every key of the left subtree, find in the left subtree the node with the greatest key and make it a parent of the right subtree.
- The desired node is the rightmost node of the left subtree.
- Locate it by moving along this subtree and taking right pointers until null is encountered.
- This last right most node in the left subtree of the node to be deleted has no right child so it is safe to attach to it the right subtree of our node.

Delete by Merging



Delete by Merging



Delete by Merging

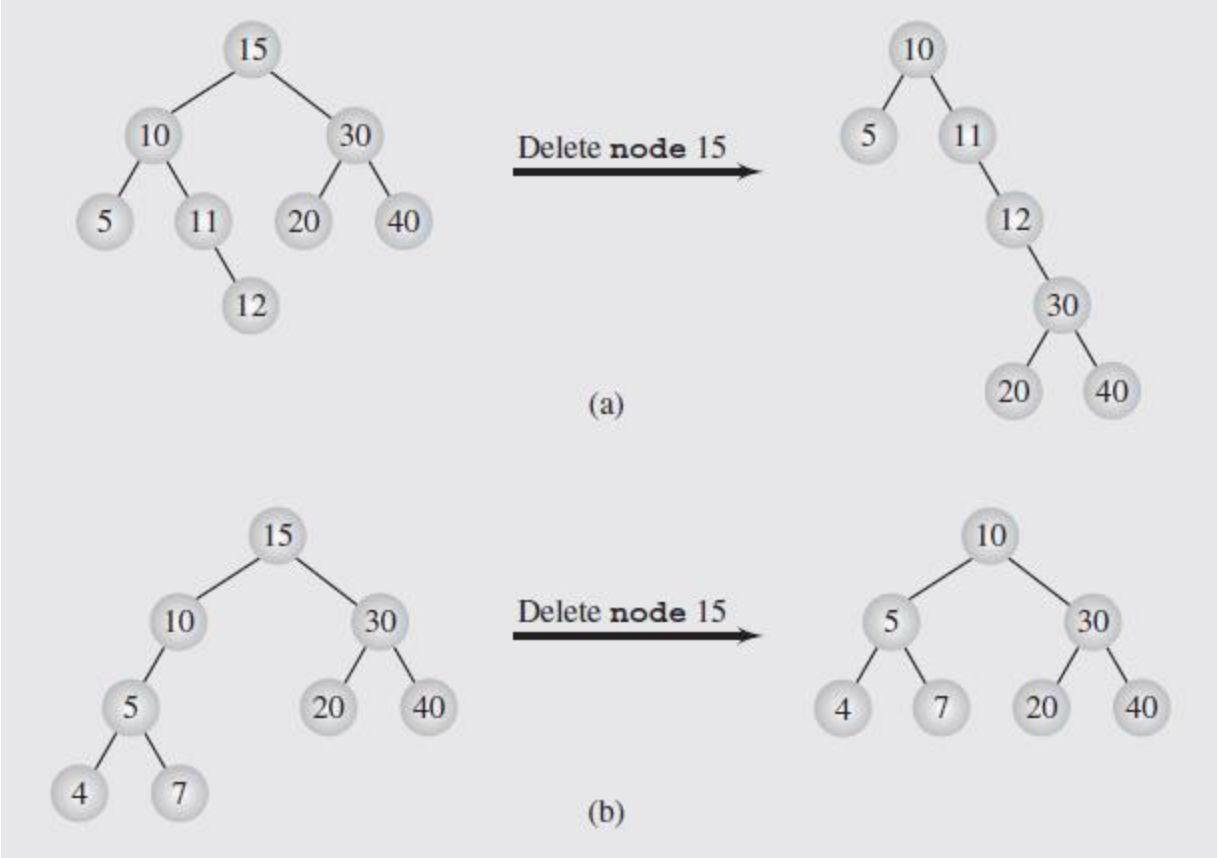
```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T> *tmp = node;
    if (node != 0) {
        if (!node->right)              // node has no right child: its left
            node = node->left;         // child (if any) is attached to its
                                       // parent;
        else if (node->left == 0)      // node has no left child: its right
            node = node->right;        // child is attached to its parent;
        else {                         // be ready for merging subtrees;
            tmp = node->left;          // 1. move left
            while (tmp->right != 0)    // 2. and then right as far as
                                       // possible;
               tmp = tmp->right;
            tmp->right =               // 3. establish the link between
               node->right;            //    the rightmost node of the left
                                       //    subtree and the right subtree;
            tmp = node;                // 4.
            node = node->left;         // 5.
        }
        delete tmp;                    // 6.
    }
}
```

Delete by Merging

```
template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->el == el)
            break;
        prev = node;
        if (el < node->el)
            node = node->left;
        else node = node->right;
    }
    if (node != 0 && node->el == el)
        if (node == root)
            deleteByMerging(root);
        else if (prev->left == node)
            deleteByMerging(prev->left);
        else deleteByMerging(prev->right);
    else if (root != 0)
        cout << "element" << el << "is not in the tree\n";
    else cout << "the tree is empty\n";
}
```

Delete by Merging

- The algorithm for deletion by merging may result in increasing the height of the tree. In some cases, the new tree may be highly unbalanced.
- Sometimes the height may be reduced.



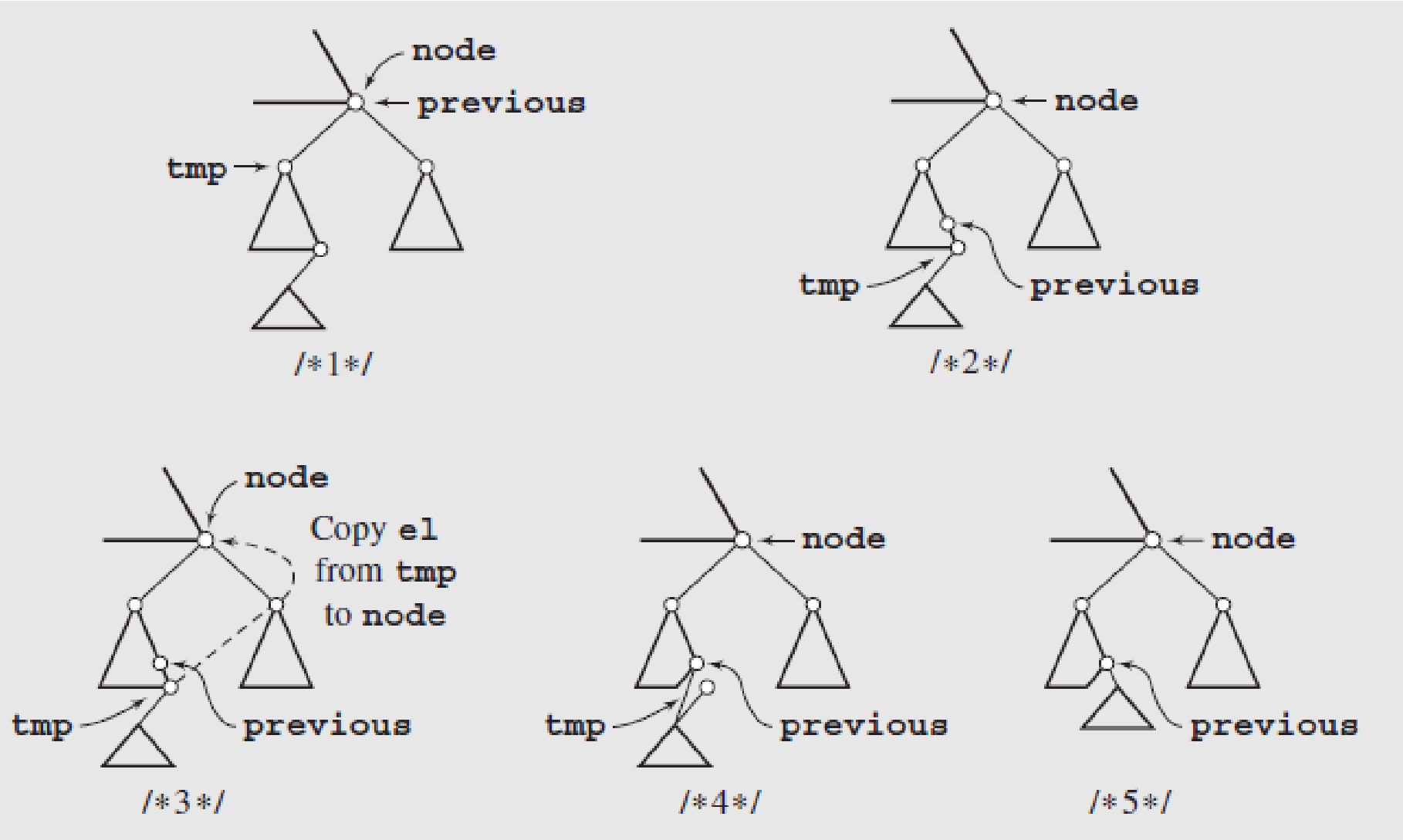
Delete by Copying

- Replace the key being deleted with its immediate predecessor (or successor).
- A key's predecessor is the key in the rightmost node in the left subtree (and analogically, its immediate successor is the key in the leftmost node in the right subtree).
- Predecessor is located by moving one step to the left by first reaching the root of the node's left subtree and then moving as far to the right as possible.
- Next, the key of the located node replaces the key to be deleted.
- Here, one of two simple cases comes into play 1) If the rightmost node is a leaf 2) If it has one child.

Delete by Copying

```
template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *previous, *tmp = node;
     if (node->right == 0)                    // node has no right child;
          node = node->left;
     else if (node->left == 0)                // node has no left child;
          node = node->right;
     else {
          tmp = node->left;                   // node has both children;
          previous = node;                    // 1.
          while (tmp->right != 0) {           // 2.
              previous = tmp;
              tmp = tmp->right;
          }
          node->el = tmp->el;                 // 3.
          if (previous == node)
               previous ->left  = tmp->left;
          else previous ->right = tmp->left;  // 4.
     }
     delete tmp;                              // 5.
}
```

Delete by Copying



Delete by Copying

- findAndDeleteByCopying(), is just like findAndDeleteByMerging(), but it calls deleteByCopying() instead of deleteByMerging().
- It always deletes the node of the immediate predecessor of the key in node, possibly reducing the height of the left subtree and leaving the right subtree unaffected.
- Eventually, the entire tree may become right unbalanced, with the right subtree bushier and larger than the left subtree.
- A simple solution: alternately delete the predecessor of the key in node from the left subtree and delete its successor from the right subtree.

Check If BT is BST?

- Trick is to verify if a node's value is in a valid range.
- Start the range with the smallest possible and the largest possible values.
- Narrow the range in the recursive calls for its children, according to node's value.

```
bool isBST(BSTNode* root, int min, int max){
        // base condition
        if (root == NULL)
                return true;

        // check if node is out-of-range
        if (root->key < min || root->key > max)
                return false;

        return isBST(root->left, min, root->key-1) &&
                isBST(root->right, root->key+1, max);

}
```

Left Heavy Or Right Heavy Tree?

```
int isLeftHeavy(BSTNode* root){
        int leftNodes = countNodes(root->left);
        int rightNodes = countNodes(root->right);

        if (leftNodes > rightNodes)
                return 0;
        if (leftNodes < rightNodes)
                return 1;
        else
                return 2;
```

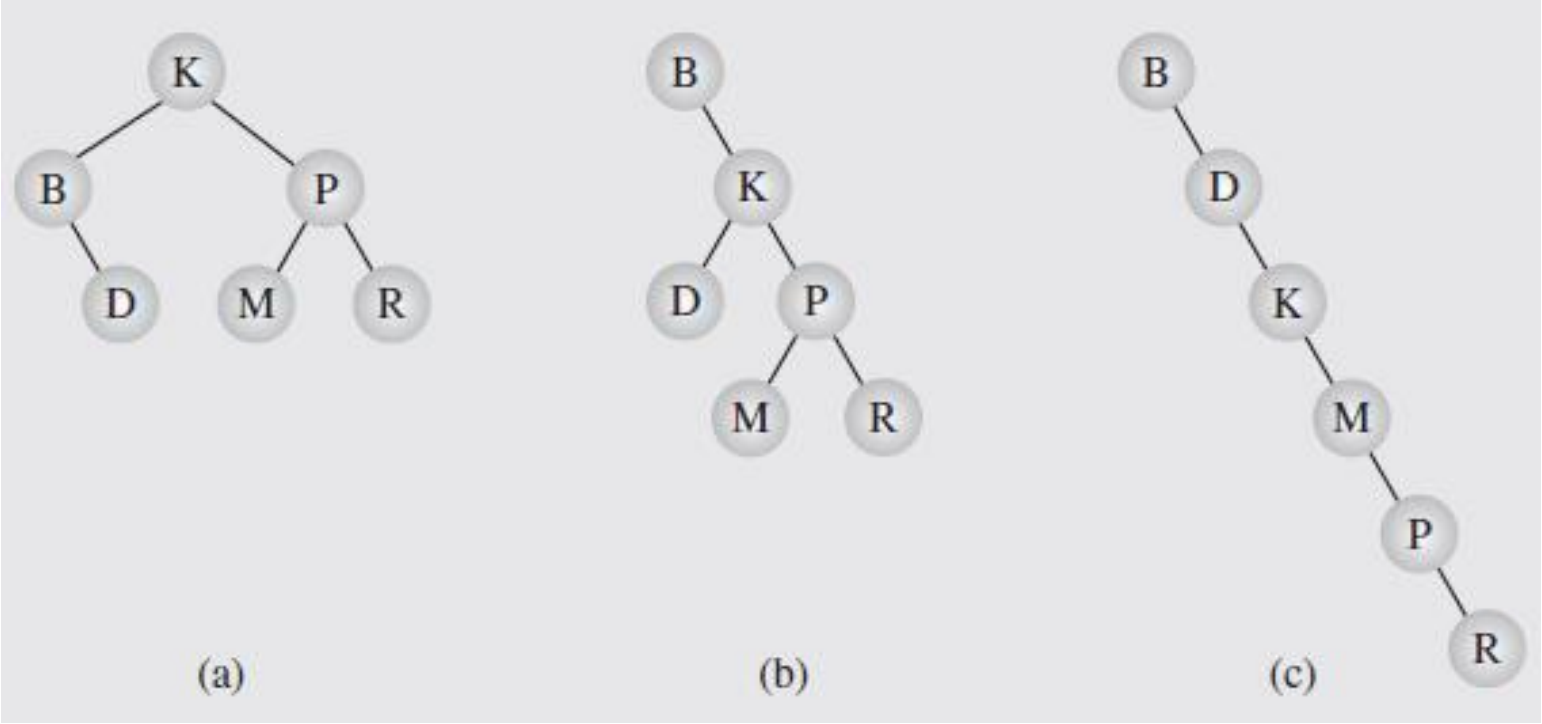
```
int countNodes(BSTNode* root){
        if (root == NULL)
                return 0;

        queue<BSTNode*> q;
        q.push(root);
        int count = 1;
        while (!q.empty()){
                BSTNode* temp = q.front();
                q.pop();
                if (temp->left != NULL){
                    q.push(temp->left);
                    count++;
                }
                if (temp->right != NULL){
                    q.push(temp->right);
                    count++;
                }
        }
        return count;
}
```

Balancing a Tree

- It is worth the effort to build a balanced tree or modify an existing tree so that it is balanced. Why?
- If 10,000 elements are stored in a perfectly balanced tree, then the tree is of height $\lceil \log(10{,}000) \rceil = \lceil 13.289 \rceil = 14$.
- It means if there are 10,000 nodes in a balanced tree, only 14 nodes to be checked to locate an element. On contrary, if tree is not balanced or a linked list is used then 10,000 nodes can be checked in the worst case.

Balancing a Tree

- Different binary search trees with the same information.



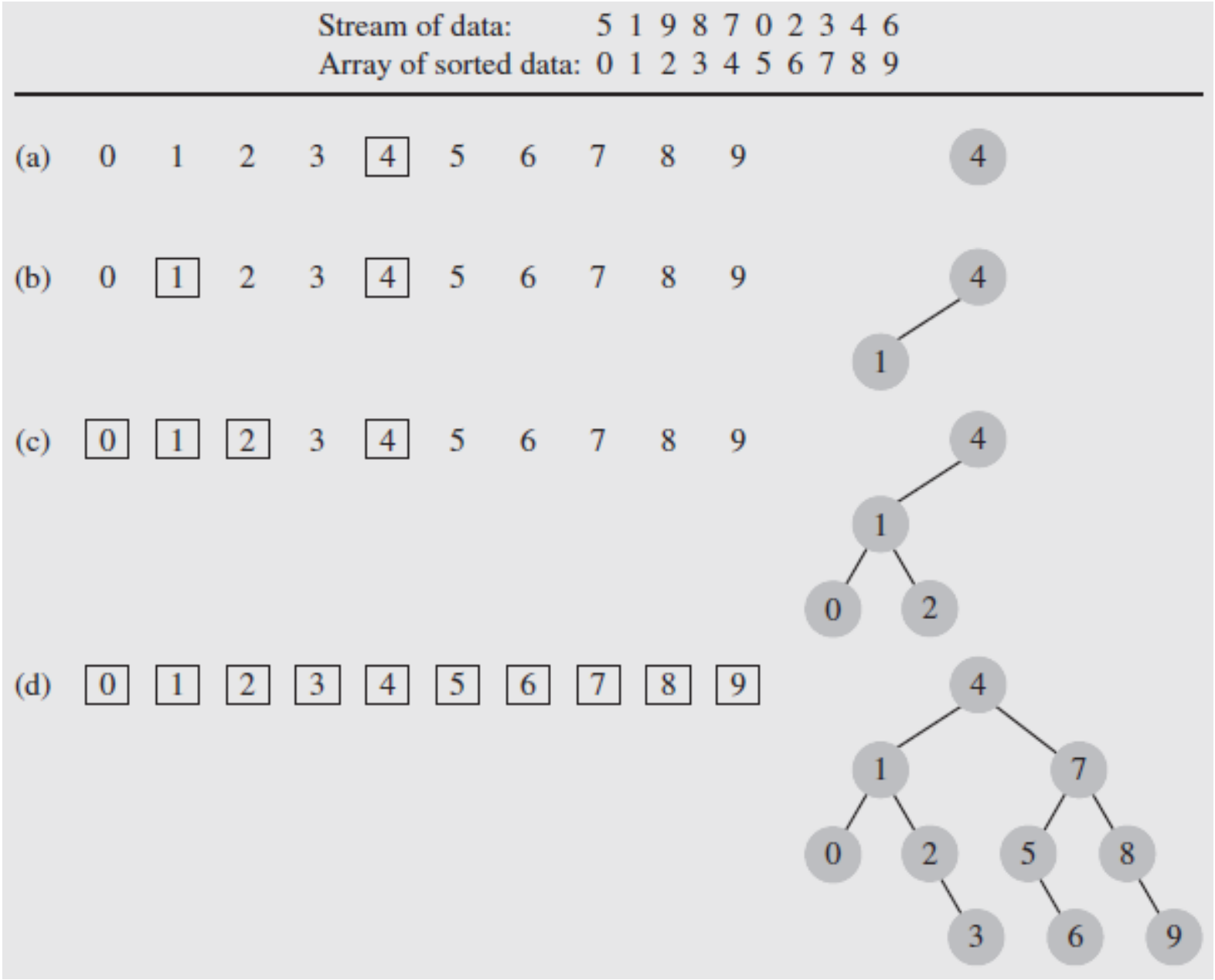
Balancing a Tree

- What's the issue? Answer: Data streaming in a particular order.
- For (c), data streamed in ascending order.
- For (b), node B became root, then its left subtree can only have one node, A, and its right subtree will have rest of the nodes (i.e, C to Z).
- (a) is nicely balanced, because its root (K=11th letter) is closer to the middle letter (M). Also, root's right child (P) is more or less in middle of K and Z.
- Using the observations from (a) we can write an algorithm to create a balanced binary tree which will be similar to binary search algorithm.

Balancing a Tree

- When data arrive, store all of them in an array.
- Sort the array (using an efficient sorting algorithm).
- Make the middle element of the sorted array the root of tree.
- Now, there are two subarrays: one from the start to middle and another from the middle (root) to the end.
- For the left subarray, select its middle element and make it left child of root. Similarly, for the right subarray, select the middle element and make it right child of root.
- Now, there are four subarrays. Select middle element from each and assign as left or right child.

Balancing a Tree



Balancing a Tree

- A simple recursion based implementation can be used if we first insert the root, then its left child, then the left child of this left child, and so on.

```
template<class T>
void BST<T>::balance(T data[], int first, int last) {
  if (first <= last) {
     int middle = (first + last)/2;
     insert(data[middle]);
     balance (data,first,middle-1);
     balance (data,middle+1,last);
  }
}
```

Balancing a Tree

- Drawback?
- All data must be put in an array before the tree can be created.
- It can be problematic when the tree has to be used while the data to be included in the tree are still coming.
- An improvement: transfer the data from an unbalanced tree to the array using an inorder traversal. Delete the tree and recreate by using balance().
- It eliminates the requirement for a sorting algorithm because the inorder traversal will put the data in the array in sorted order.

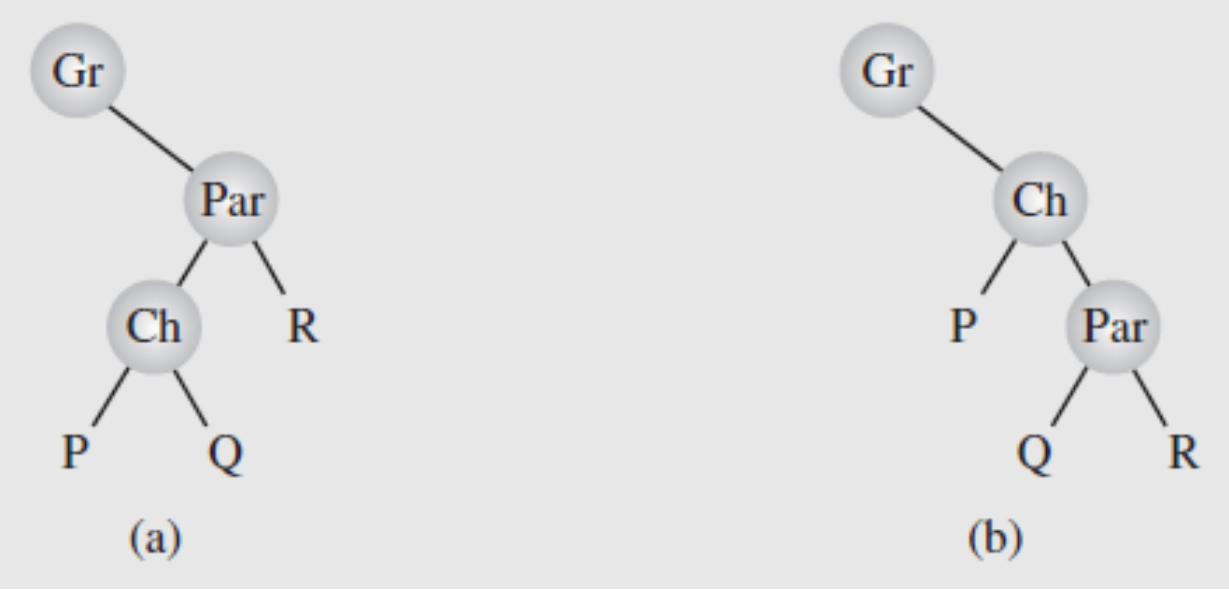
Tree Balancing – DSW Algorithm

- An algorithm that require little additional storage for intermediate variables and use no sorting procedure.
- The key idea in this algorithm is the *rotation.* There are two types of rotation, left and right, which are symmetrical to one another.
- The right rotation of the node *Ch* about its parent *Par* is performed as,

```
rotateRight(Gr, Par, Ch)
  if Par is not the root of the tree // i.e., if Gr is not null
    grandparent Gr of child Ch becomes Ch's parent;
  right subtree of Ch becomes left subtree of Ch's parent Par;
  node Ch acquires Par as its right child;
```

Tree Balancing – DSW Algorithm

- Left rotation



Tree Balancing – DSW Algorithm

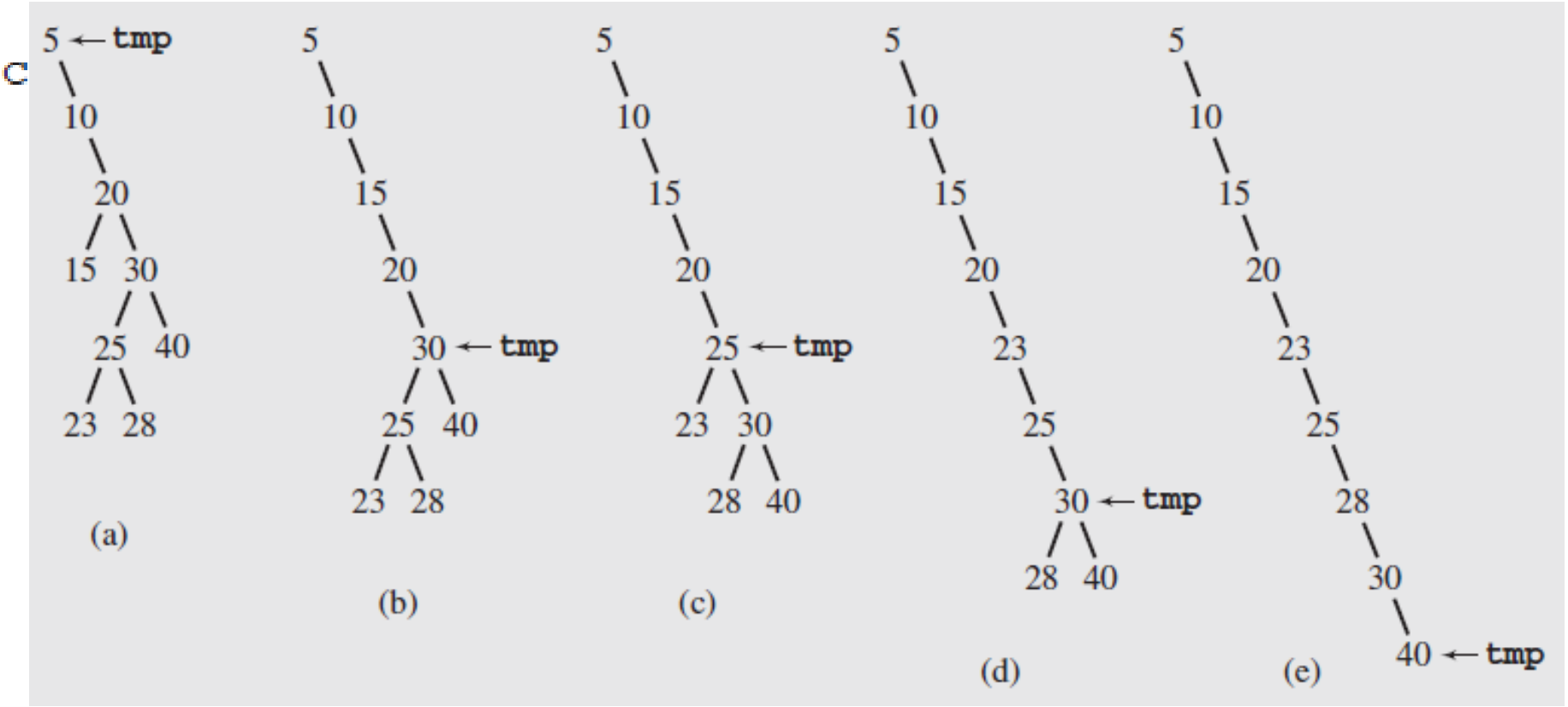
- The DSW algorithm transfigures an arbitrary binary search tree into a linked list-like tree called a *backbone* or *vine.*
- Then this elongated tree is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.
- In the first phase, a backbone is created using the following routine:

Tree Balancing – DSW Algorithm

```
createBackbone(root)
  tmp = root;
  while (tmp != 0)
    if tmp has a left child
        rotate this child about tmp; // hence the left child
                                     // becomes parent of tmp;
        set tmp to the child that just became parent;
    else set tmp to its right child;
```

Note that a rotation requires knowledge about the parent of tmp, so another pointer has to be maintained when implementing the algorithm.

Tree Balancing – DSW Algorithm



Tree Balancing – DSW Algorithm

- In the best case, when the tree is already a backbone, the while loop is executed *n* times and no rotation is performed.
- In the worst case, when the root does not have a right child, the while loop executes $2n - 1$ times with $n - 1$ rotations performed, where *n* is the number of nodes in the tree.
- It means the run time of the first phase is $O(n)$.
- In the second case (worst case), for each node except the one with the smallest value, the left child of *tmp* is rotated about *tmp*. After all rotations are finished, *tmp* points to the root, and after *n* iterations, it descends down the backbone to become null.

Tree Balancing – DSW Algorithm

- In the second phase, the backbone is transformed into a balanced tree by using createPefectTree() fucntion.
- In each pass down the backbone, every second node down to a certain point is rotated about its parent.

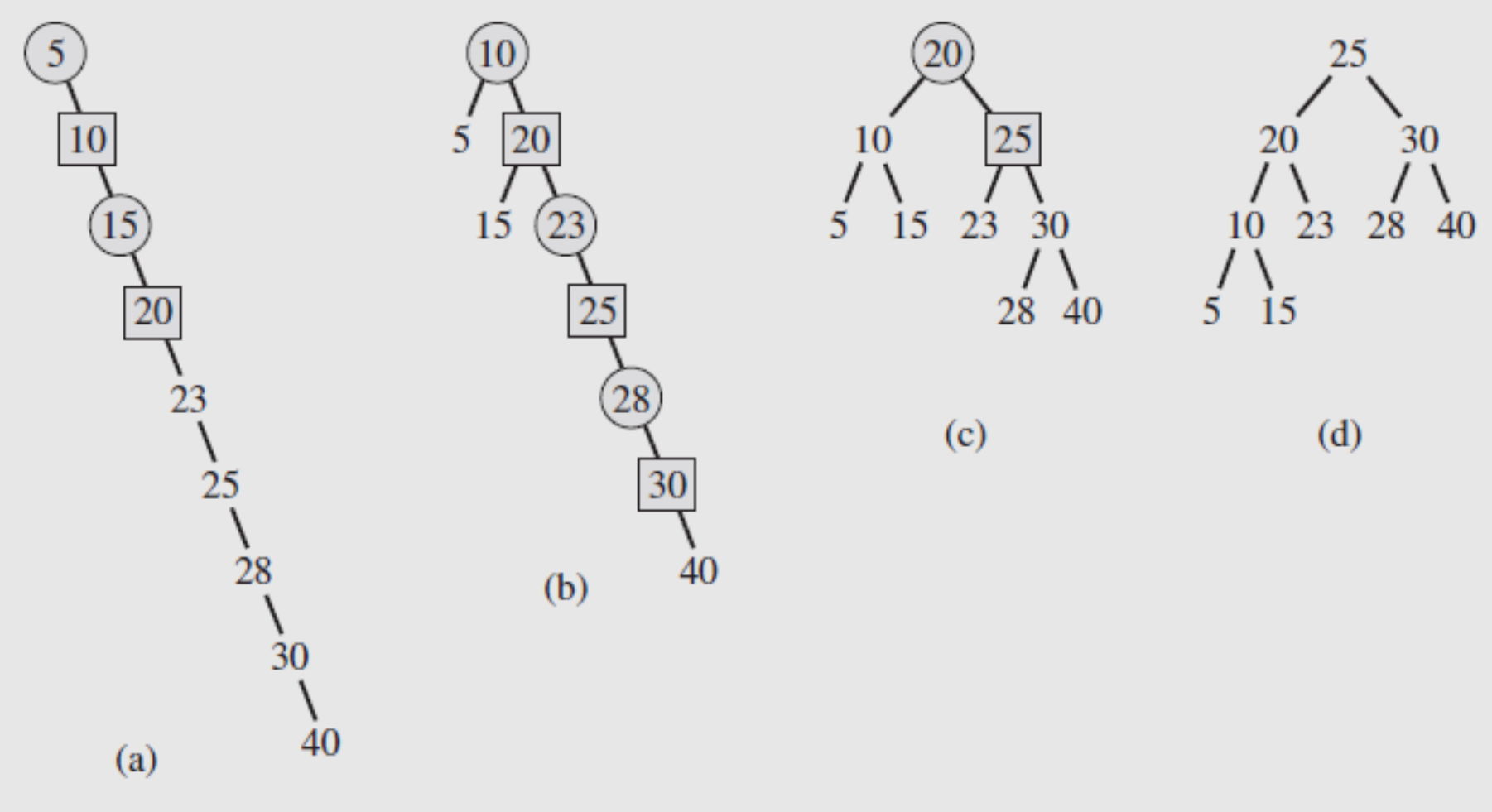
```
createPerfectTree()
  n = number of nodes;
  m = 2^⌊lg(n+1)⌋ - 1;
  make n-m rotations starting from the top of backbone;
  while (m > 1)
    m = m/2;
    make m rotations starting from the top of backbone;
```

Tree Balancing – DSW Algorithm

- In the first three lines, calculate how many nodes we expect on the bottommost level, which is n-m, and perform that many left rotations starting from root.
- *m* initially is the number of nodes in the closest complete binary tree, given by $2^{\lfloor \lg(n+1) \rfloor} - 1$. Here, log is base 2.
- If there are 9 nodes, m=7, we expect 2 children on the bottom level (i.e., overflowing nodes).
- It means the first two odd nodes will be getting left rotated.

Tree Balancing – DSW Algorithm

(a) Shows the first pass.
(b) to (d) show changes for m=7/2 and m=3/2 passes



In each backbone, the nodes to be promoted by one level by left rotations are shown as squares; their parents, about which they are rotated, are circles.

Tree Balancing – DSW Algorithm

- The number of rotations in createPerfectTree() turns out to be $n - \lfloor \lg(n + 1) \rfloor$, which means the number of rotations is $O(n)$.
- Remember, creating a backbone also required at most $O(n)$ rotations.
- The cost of global rebalancing with the DSW algorithm is optimal in terms of time because it grows linearly with n and requires a very small and fixed amount of additional storage.

Tree Balancing – AVL Tree

- DSW rebalanced the tree globally; each and every node could have been involved in rebalancing either by moving data from nodes or by reassigning new values to pointers.
- However, if a tree becomes unbalanced after an insertion or deletion, it can be balanced locally (i.e., by rotating few nodes).
- AVL Tree can perform tree balancing by doing local rotations.
- AVL tree is a tree in which the height of the left and right subtrees of every node differ by at most one.
- In AVL tree, a balance factor is computed for every node.

Tree Balancing – AVL Tree

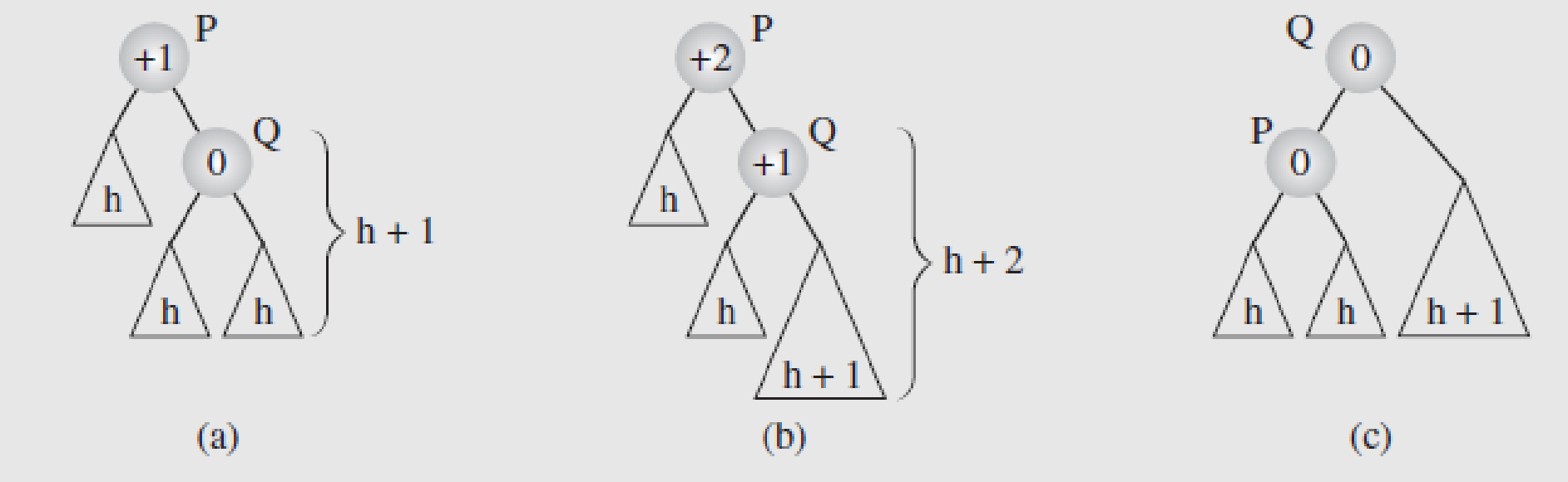
- A balance factor is the height of the right subtree minus the height of the left subtree.
- For every node, the balance factor should be -1, 0, or 1.
- Note, ALV tree is similar to balanced tree. But the one which also a binary search tree.
- The height of tree in AVL tree is less than or equal to O(log n). Therefore, the worst case search requires $O(\log n)$ comparisons.

Tree Balancing – AVL Tree

- If the balance factor of any node in an AVL tree becomes less than –1 or greater than 1, the tree has to be balanced.
- There are four situations in which AVL tree can become unbalanced.

Tree Balancing – AVL Tree

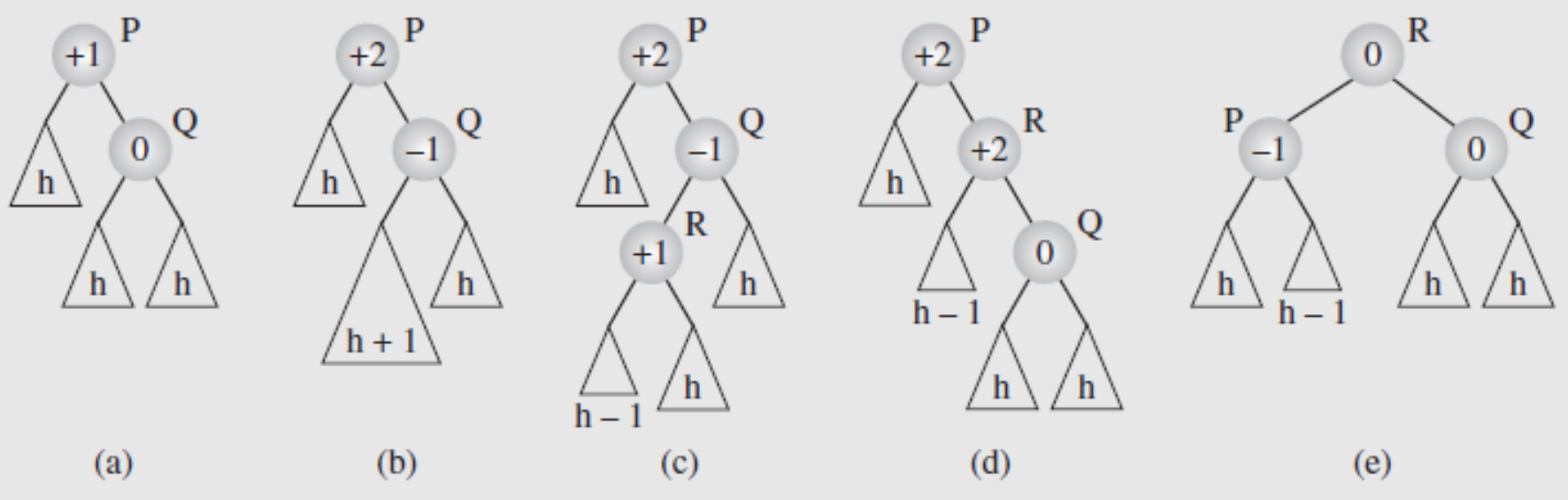
- The first case, the result of inserting a node in the right subtree of the right child.
- In the AVL tree in (a), a node is inserted somewhere in the right subtree of *Q* (shown in b).
- As a result, the balance factor of P becomes 2.
- By performing one left rotation of node Q about its parent (P) we can re-balance every node.



The heights of the participating subtrees are indicated within these subtrees.

Tree Balancing – AVL Tree

- The second case is when inserting a node in the left subtree of the right child.
- The resulting tree is shown in (b) and (c) in which node P has become unbalanced. Note, R's balance factor can also be -1.
- Now, we need double rotation. First right rotate R about node Q (d) and then left rotate R about node P (e).



Tree Balancing – AVL Tree

- Note, in both the cases, *P* can be part of a larger AVL tree; it can be a child of some other node in the tree.
- The other two cases are symmetrical.
- The problem is in finding a node *P* for which the balance factor becomes unacceptable after a node has been inserted into the tree.
- This node can be detected by moving up toward the root of the tree from the position in which the new node has been inserted and by updating the balance factors of the nodes encountered.

Tree Balancing – AVL Tree

- First, Insert the node using a insert function like we saw in BST (assume it returns the node inserted).
- Then, call the upateBalanceFactor() with Q being the next inserted node.
- If updated Q's balance factor is zero, it means no node with balance factor +-1 will be above whose balance factor will change to +-2, so stop.

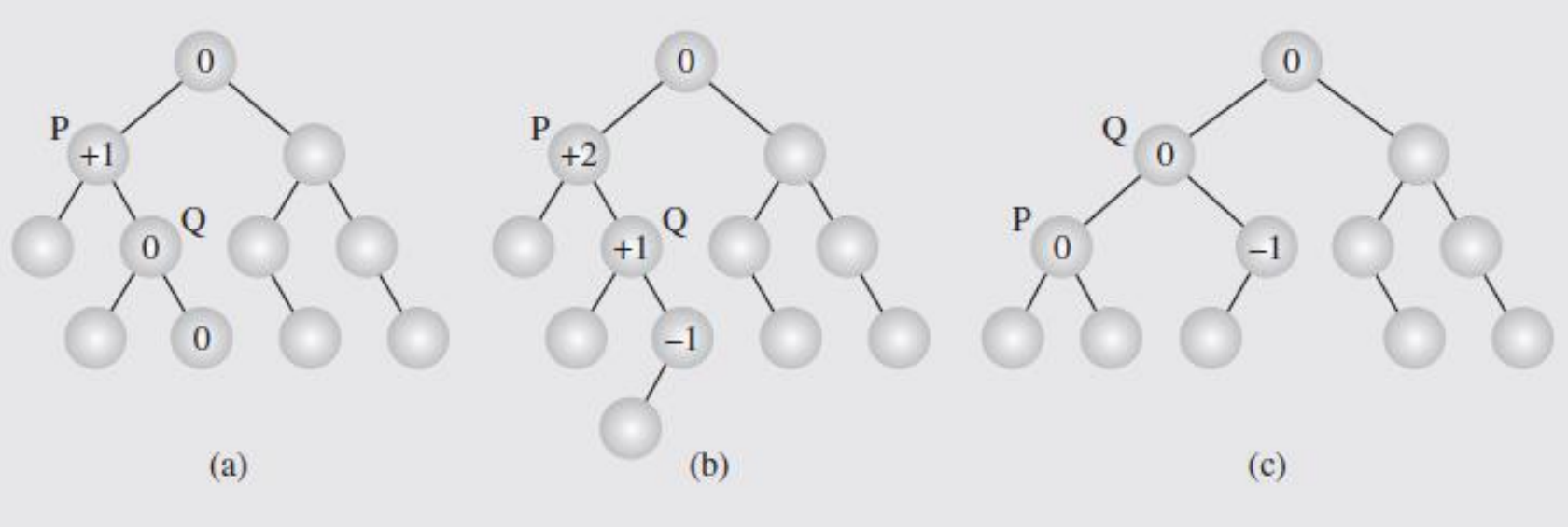
```
updateBalanceFactors()
   Q = the node just inserted;
   P = parent of Q;
   if Q is the left child of P
         P->balanceFactor--;
   else P->balanceFactor++;
   while P is not the root and P->balanceFactor ≠ ±2
         Q = P;
         P = P's parent;
         if Q->balanceFactor is 0
               return;

         if Q is the left child of P
               P->balanceFactor--;
         else P->balanceFactor++;
 if P->balanceFactor is ±2
         rebalance the subtree rooted at P;
```

Tree Balancing – AVL Tree

- If a node with ±1 balance was present in this path then after insertion its balance factor would become ±2.
- If a node with a ±1 balance factor is encountered, the balance factor may be changed to ±2, and the first node whose balance factor is changed in this way becomes the root *P* of a subtree for which the balance has to be restored.
- Note that the balance factors do not have to be updated above this node because they remain the same.

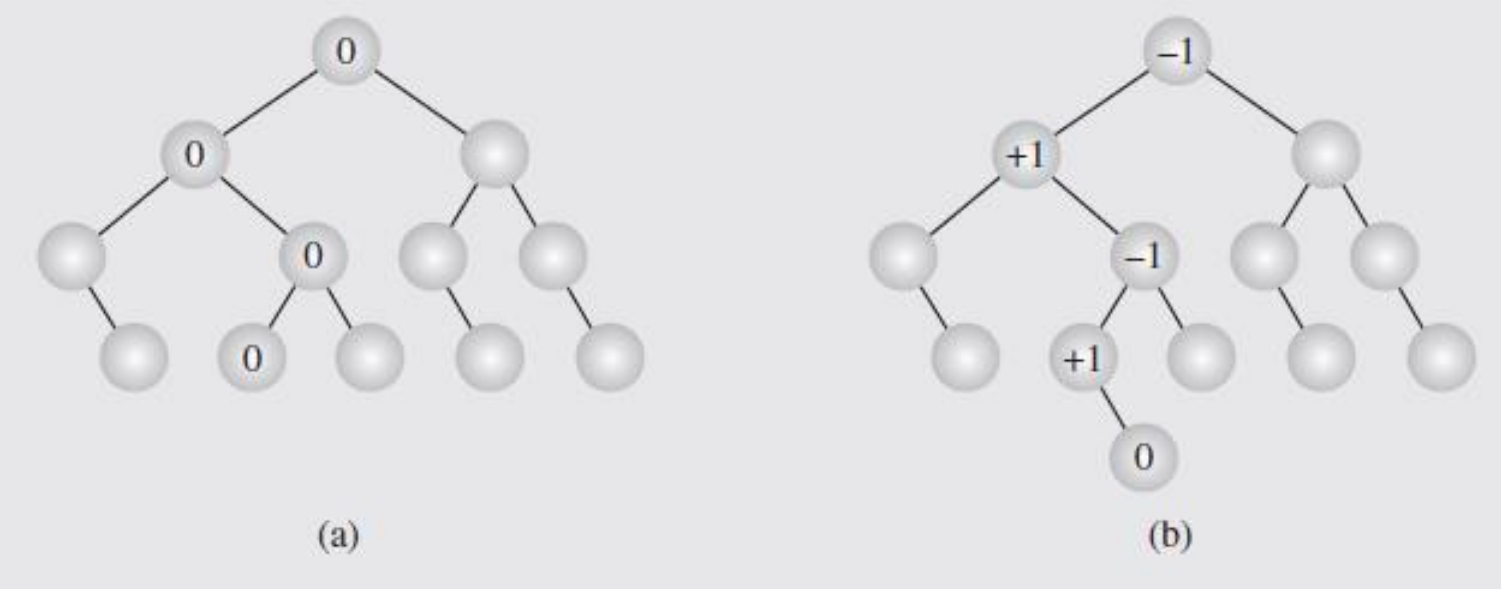
Tree Balancing – AVL Tree

- In (a), a path is marked with one balance factor equal to +1. Insertion of a new node at the end of this path results in an unbalanced tree (b), and the balance is restored by one left rotation (c).



Tree Balancing – AVL Tree

- However, if the balance factors on the path from the newly inserted node to the root of the tree are all zero, all of them have to be updated, but no rotation is needed for any of the encountered nodes.



Tree Balancing – AVL Tree

- Deletion may be more time-consuming than insertion. First, we apply deleteByCopying() to delete a node.
- After a node has been deleted from the tree, balance factors are updated from the parent of the deleted node up to the root.
- For each node in this path whose balance factor becomes ±2, a single or double rotation has to be performed to restore the balance of the tree.
- Unlike insertion, the rebalancing does not stop after the first node *P* is found for which the balance factor would become ±2.

Tree Balancing – AVL Tree

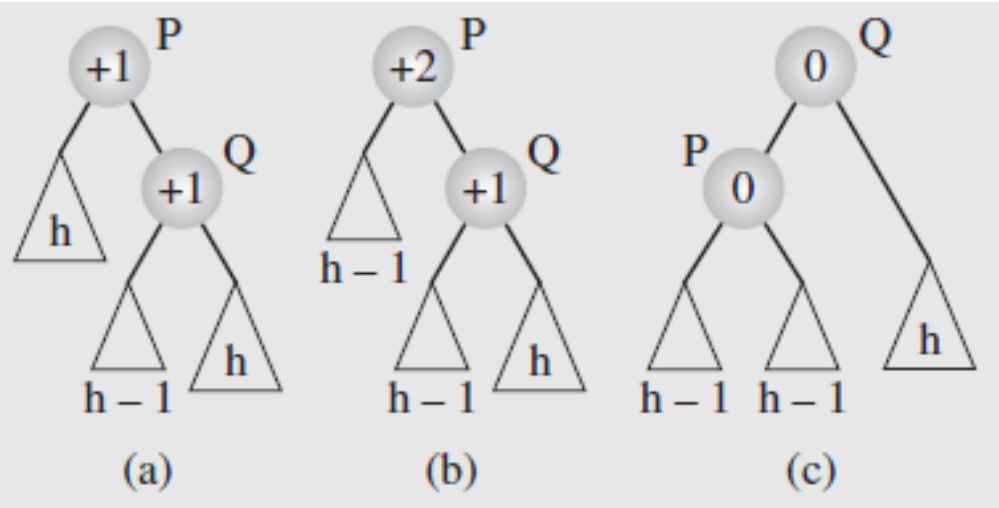
- It means deletion leads to at most $O(\lg n)$ rotations, because in the worst case, every node on the path from the deleted node to the root may require rebalancing.
- Remember, in deleteByCopying() only one subtree's height reduces by one (left subtree in our deleteByCopying() implemenation).
- It means if the node being deleted has balance factor of zero then after deletion it will become +1 so no need of rotation for this node.
- We have to discuss cases where the node being deleted has balance factor of +1 because after deletion it will become +2.

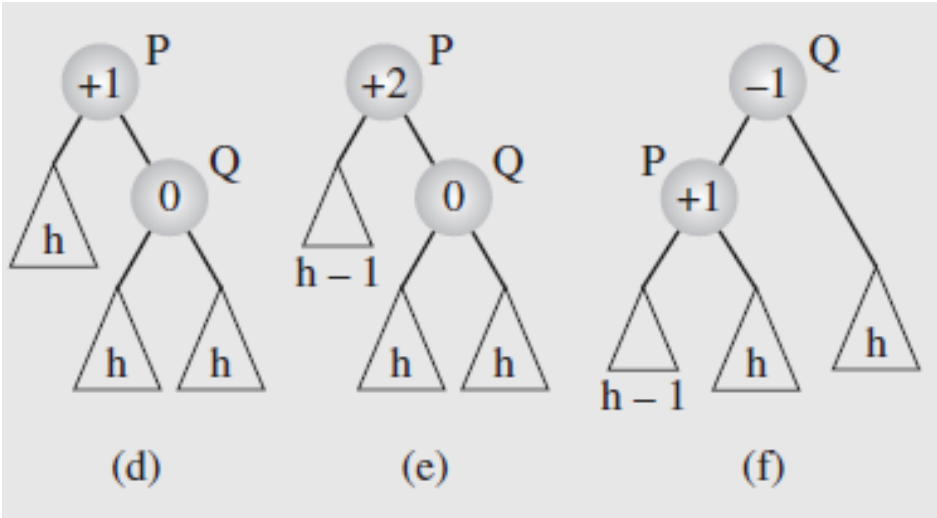
Tree Balancing – AVL Tree

- Deletion of a node does not have to necessitate an immediate rotation because it may improve the balance factor of its parent (by changing it from ±1 to 0), but it may also worsen the balance factor for the grandparent (by changing it from ±1 to ±2).
- We illustrate only those cases that require immediate rotation. There are four such cases (plus four symmetric cases).
- In each of these cases, we assume that the left child of node *P* is deleted.

Tree Balancing – AVL Tree

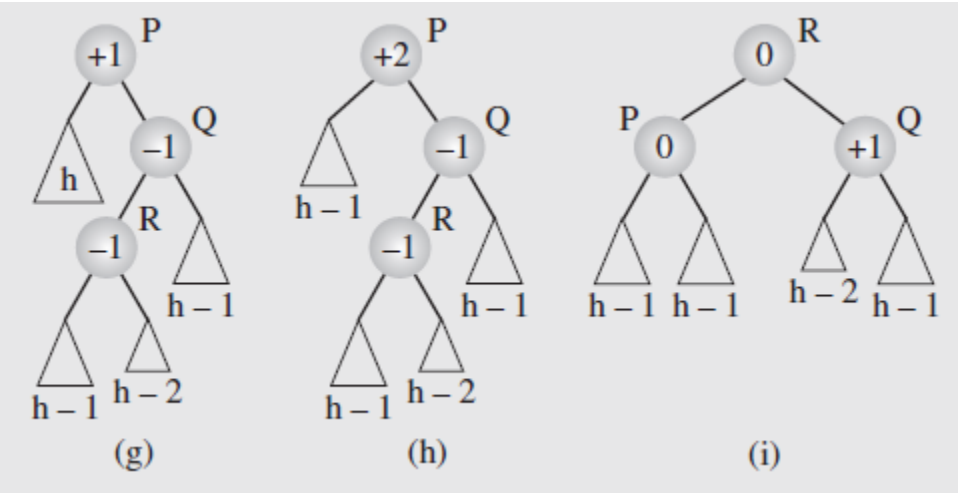
- In the first case, the tree in (a) turns, after deleting a node, into the tree in (b).
- In the second case, *P* has a balance factor equal to +1, and its right subtree *Q* has a balance factor equal to 0 (d). After deleting a node in the left subtree of *P* (e), the tree is rebalanced by the same rotation as in the first case (f).
- Both the cases can be processed together in an implementation after checking that the balance factor of Q is +1 or 0.

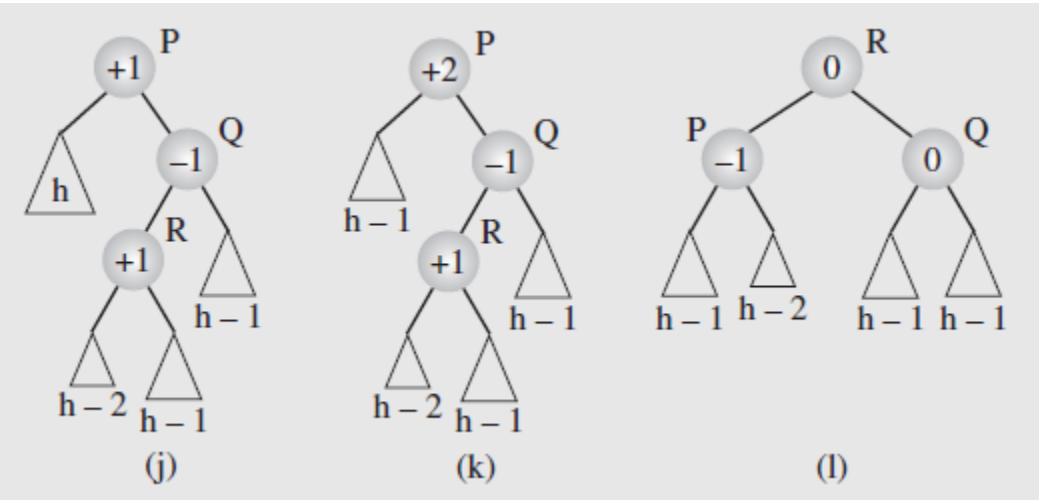




Tree Balancing – AVL Tree

- If *Q* is –1, we have two other cases.
- In the third case, the left subtree *R* of *Q* has a balance factor equal to –1 (g).
- The fourth case differs from the third in that *R*'s balance factor equals +1 (j).





Expression Tree

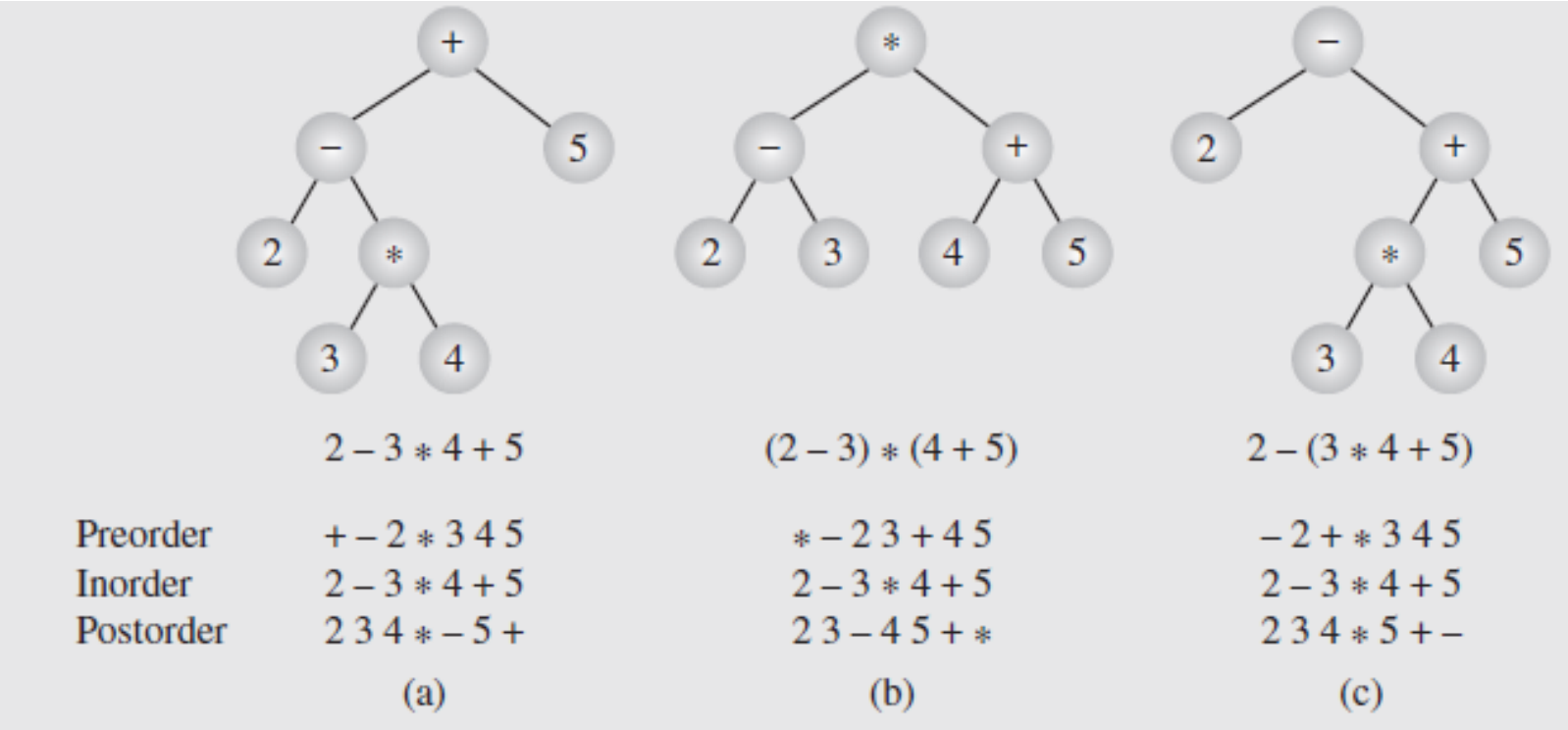
- An application of binary trees for unambiguous representation of arithmetical, relational, or logical expressions.
- A special notation that allows us to eliminate all parentheses from formulas, also called *Polish notation*.
- Compilers need to generate assembly code in which one operation is executed at a time and the result is retained for other operations.
- Therefore, all expressions have to be broken down unambiguously into separate operations and put into their proper order.
- That is where Polish notation is useful.

Expression Tree

- Special form of binary tree in which leaves are operands and internal nodes are operators.
- The result of this algebraic expression, $2 – 3 \cdot 4 + 5$, depends on the order in which the operations are performed.
- It can also be evaluated like $(2 – 3) \cdot (4 + 5)$ or $2 – (3 \cdot 4 + 5)$.
- In the first expression, we know in what order to evaluate it.
- But the computer does not know that, in such a case, multiplication has precedence over addition and subtraction.
- If we want to override the precedence, then parentheses are needed.

Expression Tree

- Alternatively, use expression trees to distinguish between the three notations (seen before).
- No need to use parentheses and yet no ambiguity arises.



	(a)	(b)	(c)
	$2 - 3 * 4 + 5$	$(2 - 3) * (4 + 5)$	$2 - (3 * 4 + 5)$
Preorder	$+ - 2 * 3\ 4\ 5$	$* - 2\ 3 + 4\ 5$	$- 2 + * 3\ 4\ 5$
Inorder	$2 - 3 * 4 + 5$	$2 - 3 * 4 + 5$	$2 - 3 * 4 + 5$
Postorder	$2\ 3\ 4 * - 5 +$	$2\ 3 - 4\ 5 + *$	$2\ 3\ 4 * 5 + -$

Expression Tree

- We can traverse these trees to get infix notation (via inorder traversal), prefix notation (via preorder traversal), and postfix notation (via postorder traversal).
- Interestingly, inorder traversal of all three trees results in the same output, which is the initial expression. It means that inorder tree traversal is not suitable for generating unambiguous output.
- But the other two traversals are.
- Some programming languages are using Polish notation. For example, Forth and PostScript use postfix notation. LISP and, to a large degree, LOGO use prefix notation.

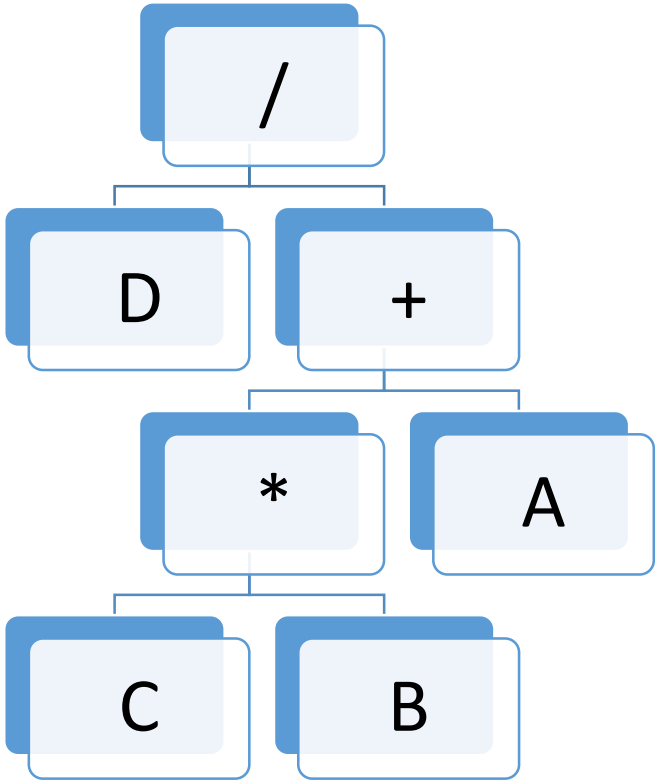
Expression Tree – Construction

- One simple approach is to use postfix expression and a stack.
- Therefore, it requires to convert infix expression into a postfix.

- Loop through the input postfix expression character by character. Check for every character,
  - If it is an operand, create a node for the operand, then push it to the stack
  - If it is a operator then,
    - create a node with the operator,
    - pop two elements from the stack and make them as left and right child of the operator's node,
    - Push this node back to the stack
- Finally, the stack will have only one element, the root of an expression tree.

Expression Tree – Construction

- For infix expression ((A+(B*C))/D), the postfix is ABC*+D/



Expression Tree – Construction

- Another approach is to use precedence of operators.
- We know * and / have higher precedence then + and –
- In case of multiple operators with the same precedence, we check associativity determines the direction in which they execute.
- For both the pairs (*/ and +-), the associativity if from left to right.
- The higher precedence operators are processed first and should be close to the bottom of the tree (where the operands are).
- The least precedence operator will be at the top (root node).

Expression Tree – Construction

1. Scan the expression left to right and find the least precedence operator from the infix expression.
   - For example, for given infix a* b/c + d/e*f + g - x*y, the least precedence is of the minus just before x*y (because it will be evaluated after the first two plus). Make minus the root.
2. Next step, expression's part on the right of the root operator becomes right subtree of the root. Similarly, left part becomes left subtree.
   - For example, sub-expression x*y will be the right child of minus node and sub-expression a* b/c + d/e*f + g will be the lest child.

- Repeat from step 1 for both the child until reach the leave nodes.