Balanced Trees

After studying this chapter, you should be able to

- Define a self-balancing binary search tree
- Define an AVL tree and its properties
- Define the four AVL rotation operations: rotate left, rotate right, rotate left-right, and rotate right-left
- Apply the correct AVL rotation operations to an unbalanced tree to regain balance
- Implement the AVL rotation algorithms in C++
- Define a Red-Black tree and its properties
- Understand the Red-Black tree recoloring operations used to maintain its properties
- Implement the recoloring and restructuring Red-Black tree algorithms in C++
- Define a B-tree and its properties
- Understand the basics of B-tree insertion
- Explain why we might us different tree ADTs in different application domains and how they can impact performance

Trees Plus

- Chapter 8 introduced the binary tree, a useful structure for implementing ADTs
- However, its efficiency is only guaranteed if the tree is balanced (near $\log_2 N$ height)
- This chapter explores binary trees that *self-balance* when nodes are added or removed

Balanced Binary Trees

- **Balance Factor:** The difference in height between the two subtrees of a binary tree
- A tree is **balanced** if its balance factor is 0 or 1
- By rebalancing when the balance factor grows too large, tree operations will remain efficient
- This should be checked after insertion and deletion operations, since these can change the height of the tree

AVL Trees

- **AVL Tree:** A binary search tree that balances itself when the balance factor is greater than 1
- It uses *tree rotations* to rebalance the tree if necessary after insertion and deletion
- AVL Trees are $O(\log_2 N)$ in the average and worst cases

Note: AVL created by two mathematicians:
Georgii Adelson-Velskii and Avgenii Landis

Tree Rotations

- **Rotation:** Switching the positions of a child node and its parent to regain balance.
- The direction to rotate depends on the structure of the tree and the location of the inserted or deleted node
- Every imbalance can be solved with *only one or two rotations*

The Four Rotation Cases

- The rotation cases are based on the location of the inserted node
  - They’re the same for deleting a node; for simplicity, this discussion focuses on inserting nodes
- The following examples discuss the rotations in terms of an unbalanced node *T* with two children, *L* (on the left) and *R* (on the right)
- Remember, rotations are only used if the tree becomes unbalanced

The Four Rotation Cases (cont.)

- **Right rotation:** Used when inserting into the left subtree of T’s left child
- **Left rotation:** Used when inserting into the right subtree of T’s right child
- **Right-Left rotation:** Used when inserting into the left subtree of T’s right child
- **Left-Right rotation:** Used when inserting into the right subtree of T’s left child

Right Rotation

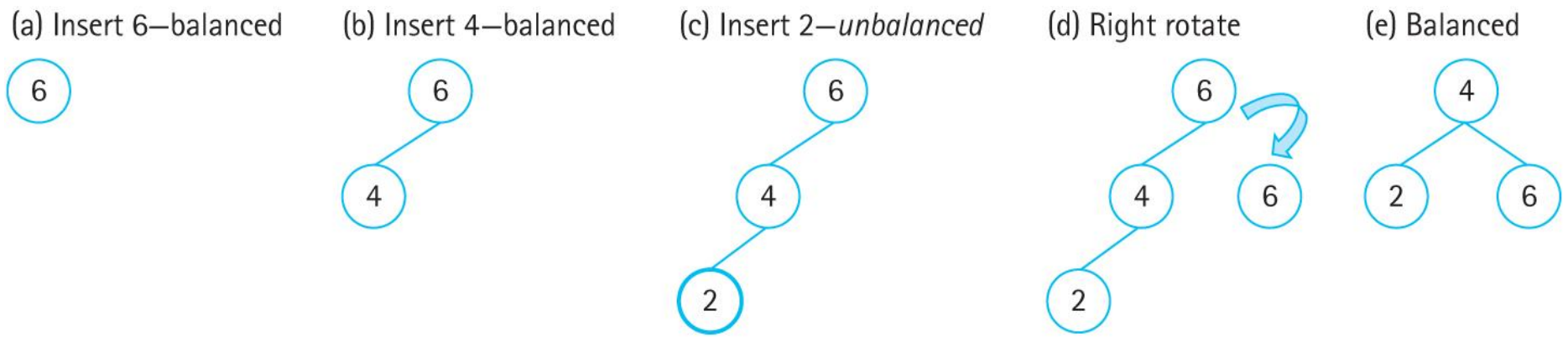


Figure 10.2 Right rotation on an unbalanced tree (a) Insert 6- balanced (b) Insert 4- balanced (c) Insert 2- unbalanced (d) Right rotate (e) balanced

Left Rotation

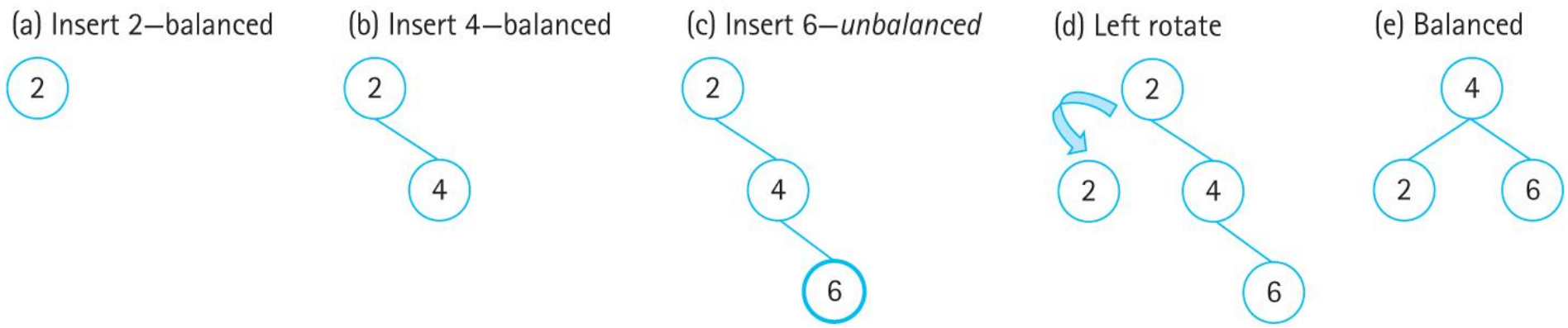


Figure 10.3 Left rotation on an unbalanced tree (a) Insert 2- balanced
(b) Insert 4-balanced (c) Insert 6- unbalanced (d) Left rotate (e) Balanced

Single Rotations

- A single left or right rotation reduces the height of the tree by 1
- A right rotation is used if the height of the left subtree is greater than the right subtree
- A left rotation is used if the height of the right subtree is greater than the left subtree

Right Rotation

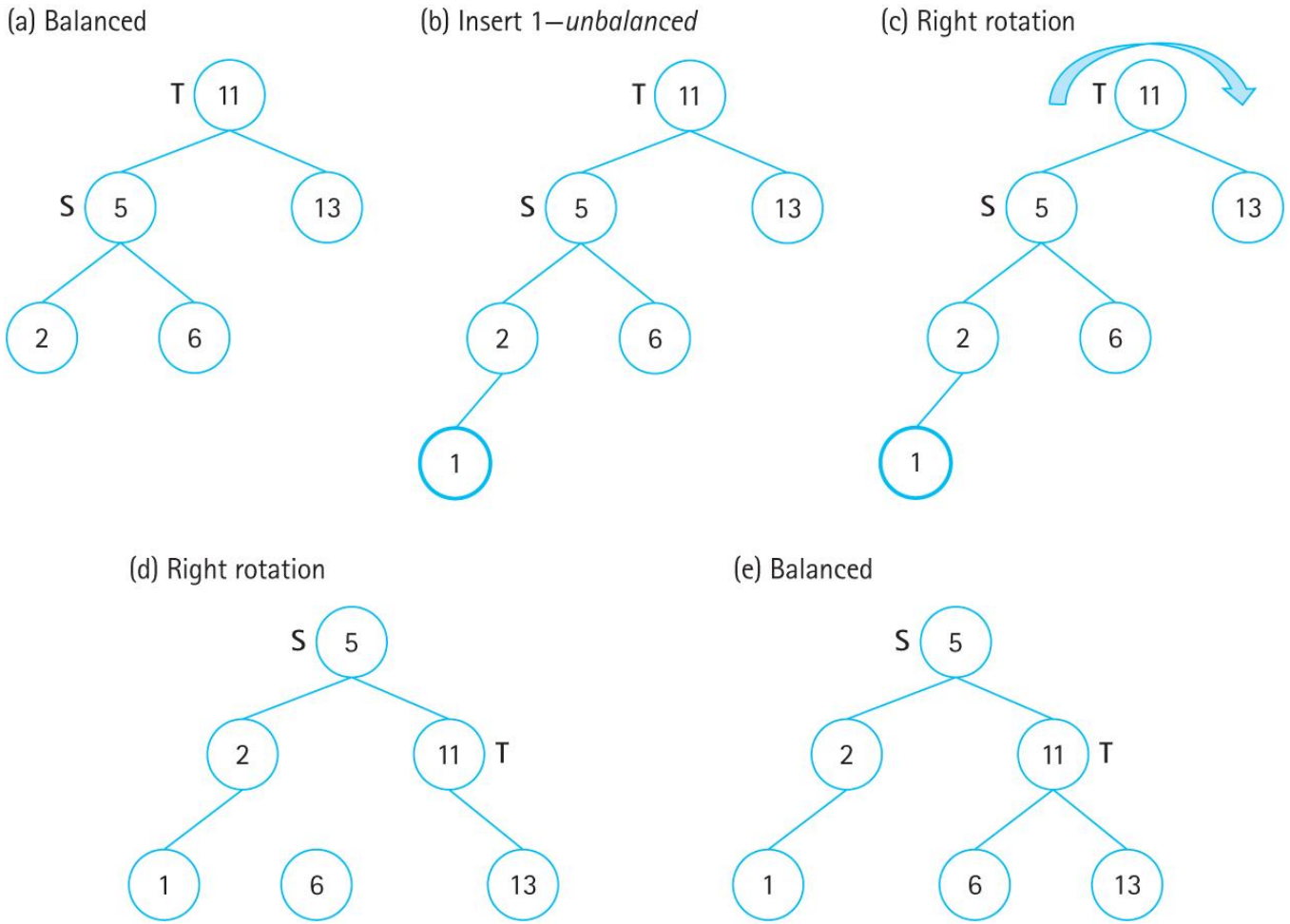


Figure 10.6 Right rotation on AVL tree after insert (a) Balanced (b) Insert 1- unbalanced (c) Right rotation (d) Right rotation (e) Balanced

Double Rotations

- Sometimes a single rotation is not enough
- Instead, a double rotation is used
- The first rotation in the double sets up the tree for the second rotation

Right-Left Rotation

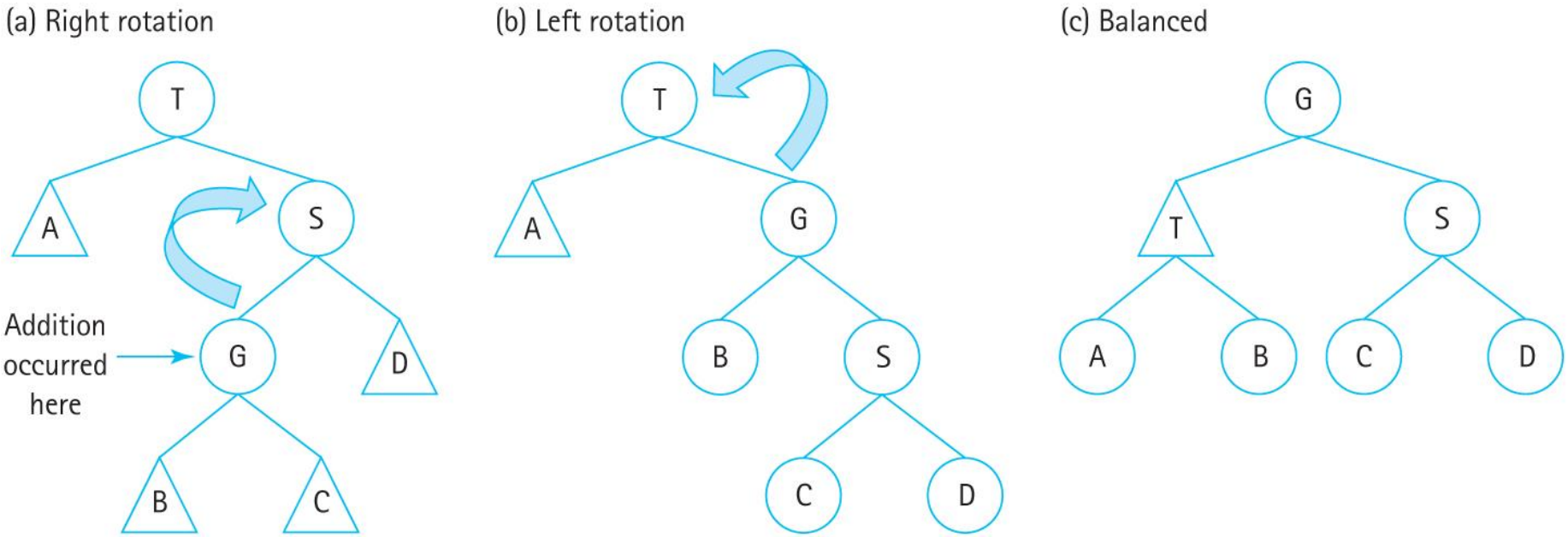


Figure 10.9 General right-left rotation on AVL tree (a) right-rotation (b) left-rotation (c) balanced

Left-Right Rotation

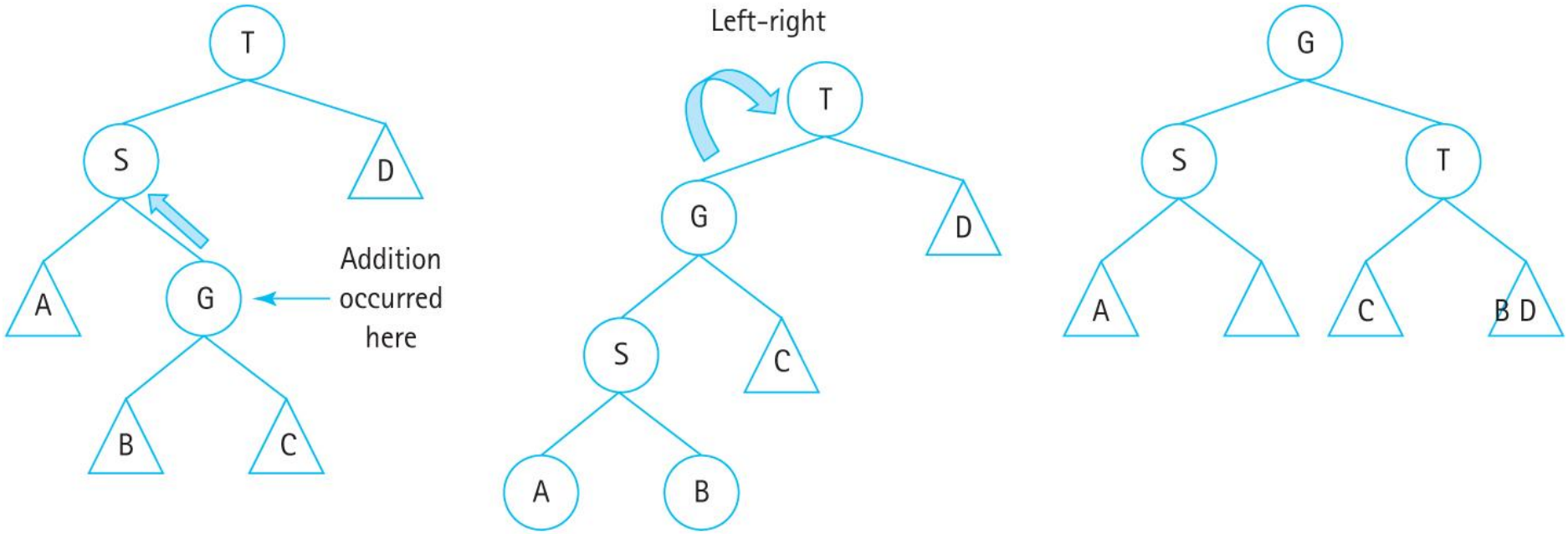
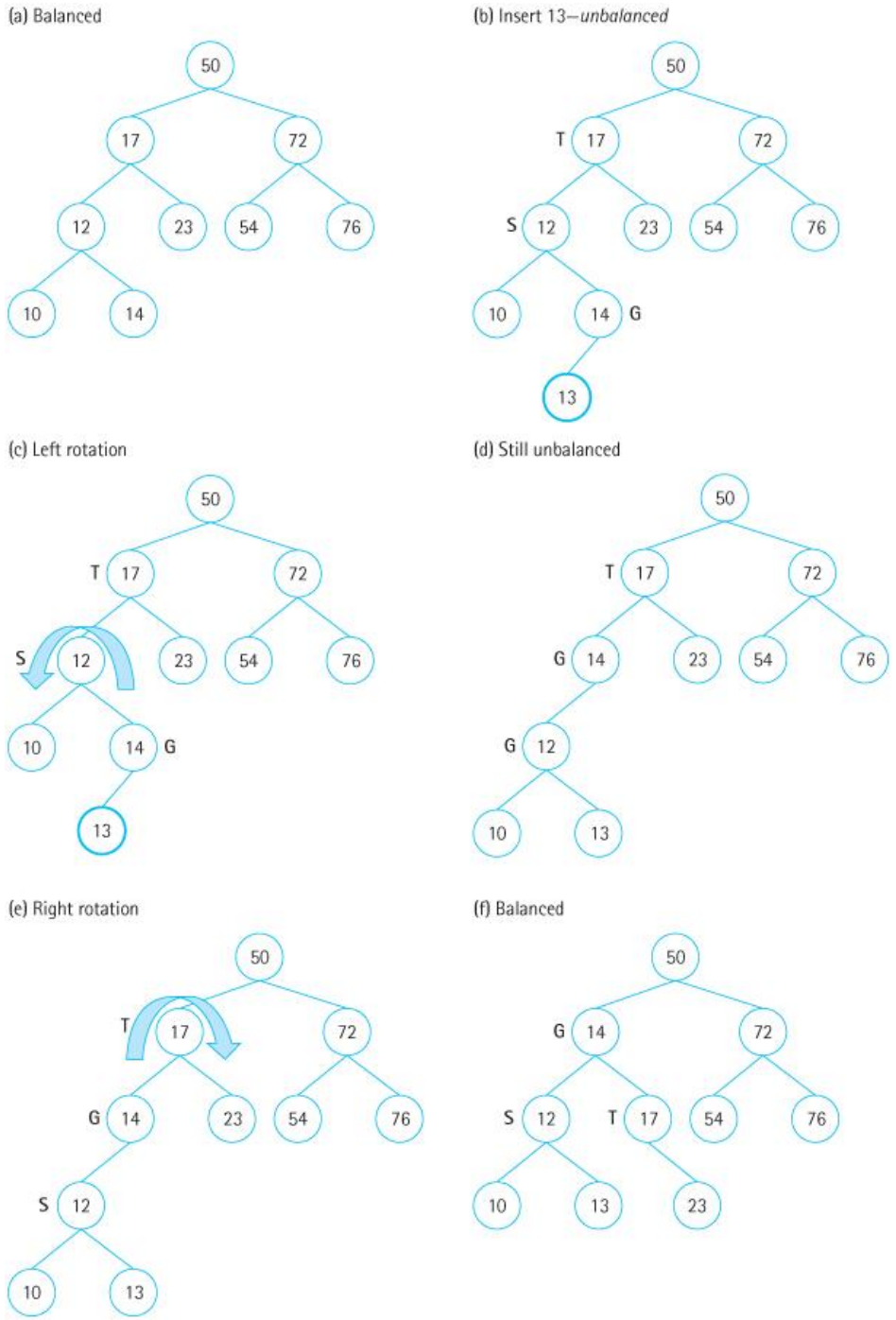


Figure 10.10 General right-left rotation on AVL tree (a) right-rotation (b) left-rotation (c) balanced

Figure 10.8 Left-right rotation on AVL tree after insert (a) Balanced (b) Insert 13-unbalanced (c) Left rotation (d) Still unbalanced (e) Right rotation (f) Balanced



AVL Trees: Logical Level

- AVL trees and binary search trees share the same operations
- AVL trees add additional conditions to PutItem and DeleteItem: The tree will be balanced before and after these operations are performed

AVL Trees: Application Level

- Binary search trees are great for random, unchanging data; the tree can be balanced once and left alone
- AVL trees are better when insertion and deletion operations are performed over the lifetime of the tree

AVL Trees: Implementation Level

- The AVL Tree implementation has a few more helper functions:
  - The four rotation functions
  - Difference, for finding the difference in height between two subtrees
  - Balance, for deciding which rotation is needed
- PutItem and DeleteItem call Balance after performing the insertion/deletion

Rotation Functions

- The rotations themselves are straightforward
- Each one takes the unbalanced node as a parameter and returns the updated root node
- Compare the code to the rotations diagrams

RotateRight

```
TreeNode* RotateRight(TreeNode* T)
// Returns the tree node resulting from
// a right rotation.
{
  TreeNode* S = T->left;
  TreeNode* B = S->right;
  S->right = T;
  T->left = B;
  return S;
}
```

RotateRight (cont.)

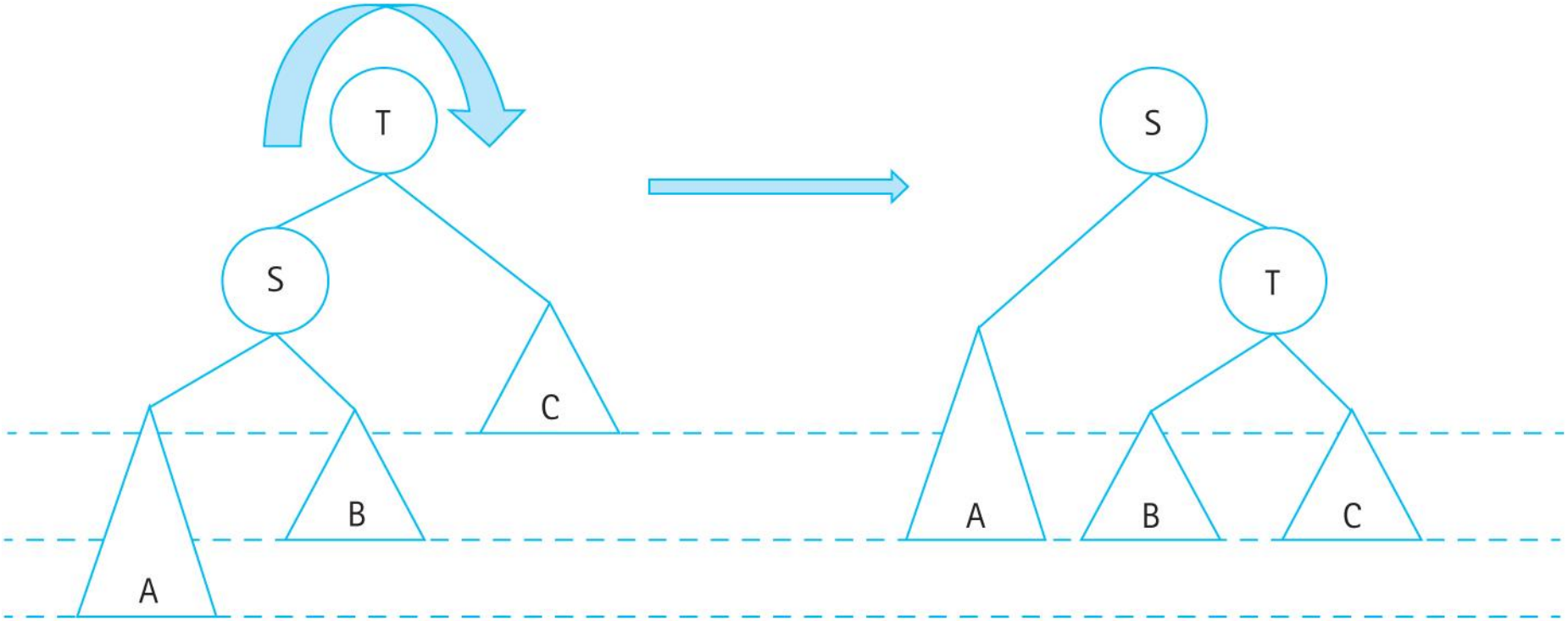


Figure 10.4 Right rotation on an AVL tree to maintain balance

RotateRight (cont.)

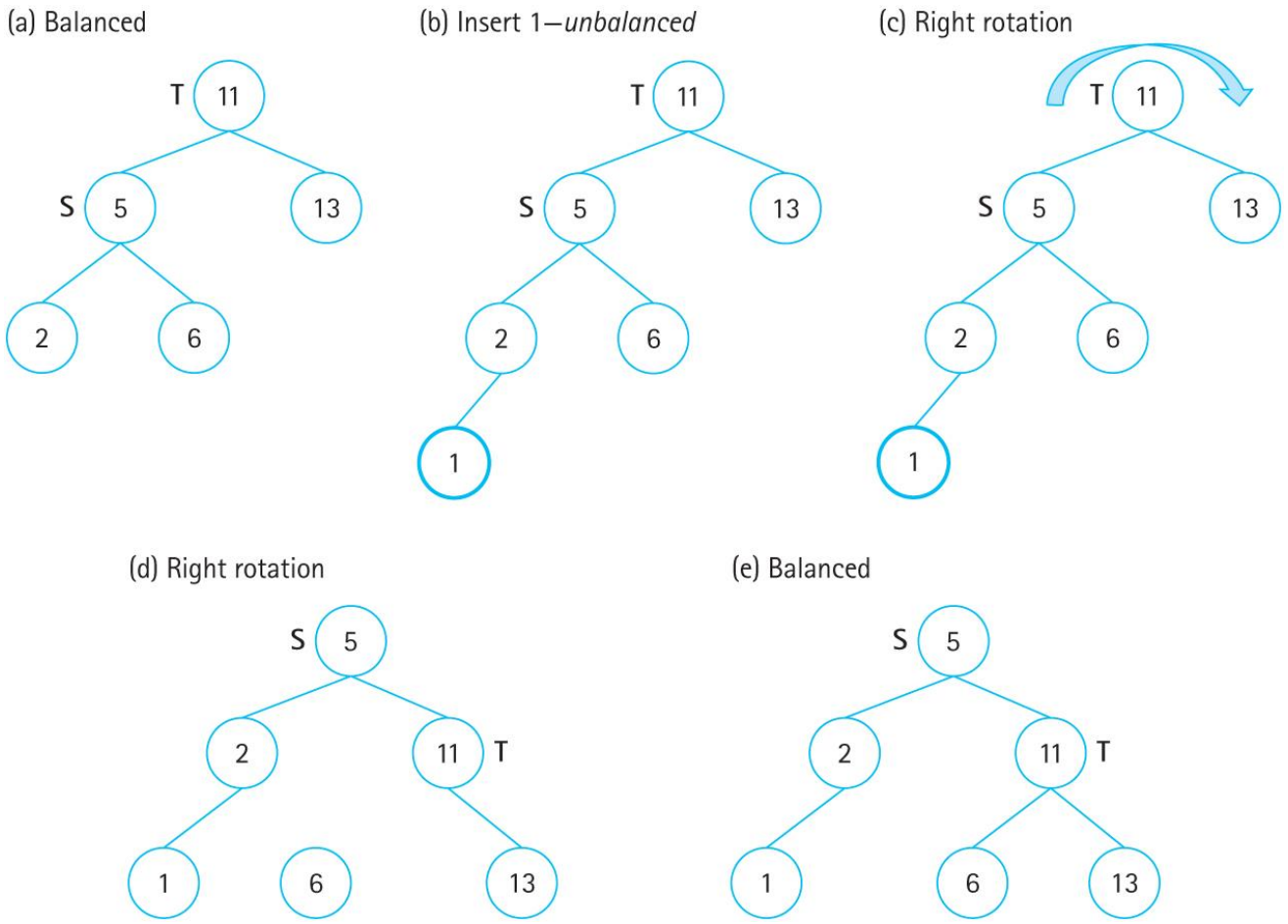


Figure 10.6 Right rotation on AVL tree after insert (a) Balanced (b) Insert 1- unbalanced (c) Right rotation (d) Right rotation (e) Balanced

RotateLeft

```
TreeNode* RotateLeft(TreeNode* T)
// Returns the tree node resulting from
// a left rotation.
{
  TreeNode* S = T->right;
  TreeNode* B = S->left;
  S->left = T;
  T->right = B;
  return S;
}
```

RotateLeft (cont.)

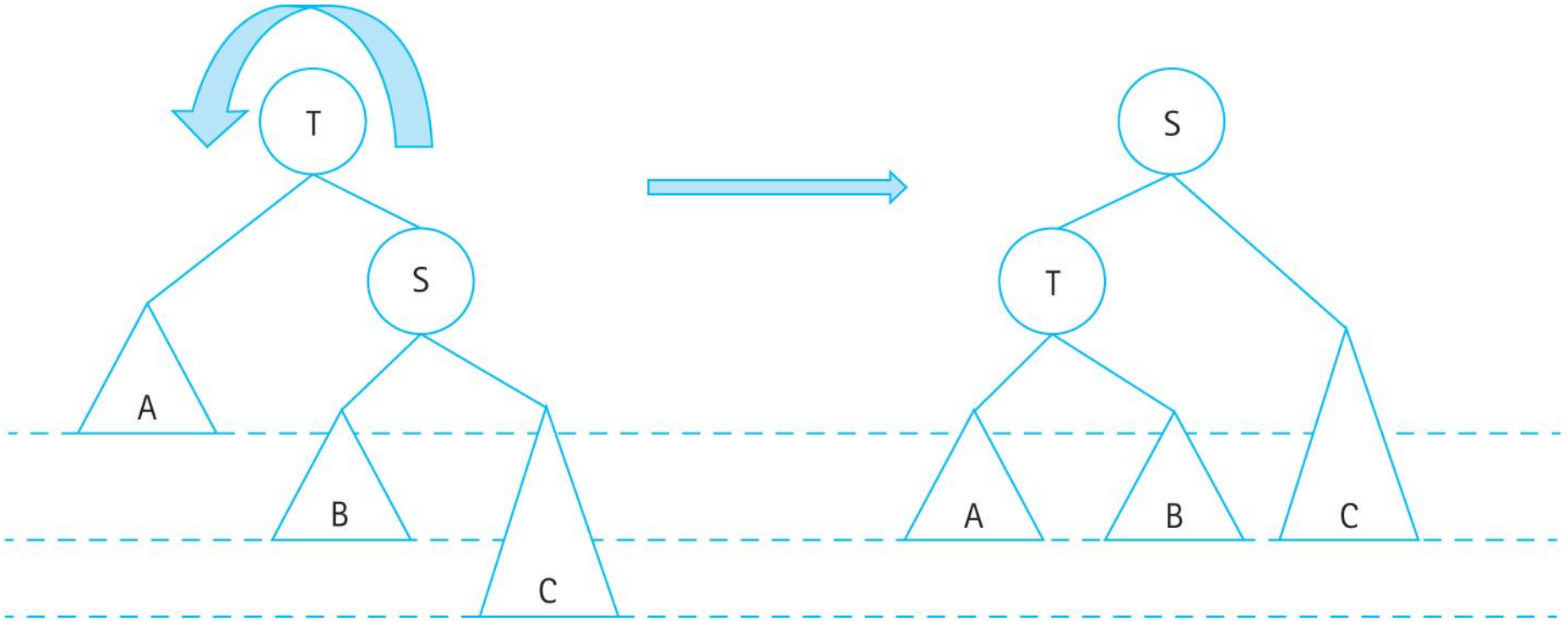


Figure 10.5 Left rotation on an AVL tree to maintain balance

RotateRightLeft

```
TreeNode* RotateRightLeft(TreeNode* T)
// Returns the tree node resulting from a
// right-left rotation.
{
  TreeNode* S = T->right;
  T->right = RotateRight(S);
  return RotateLeft(T);
}
```

RotateRightLeft (cont.)

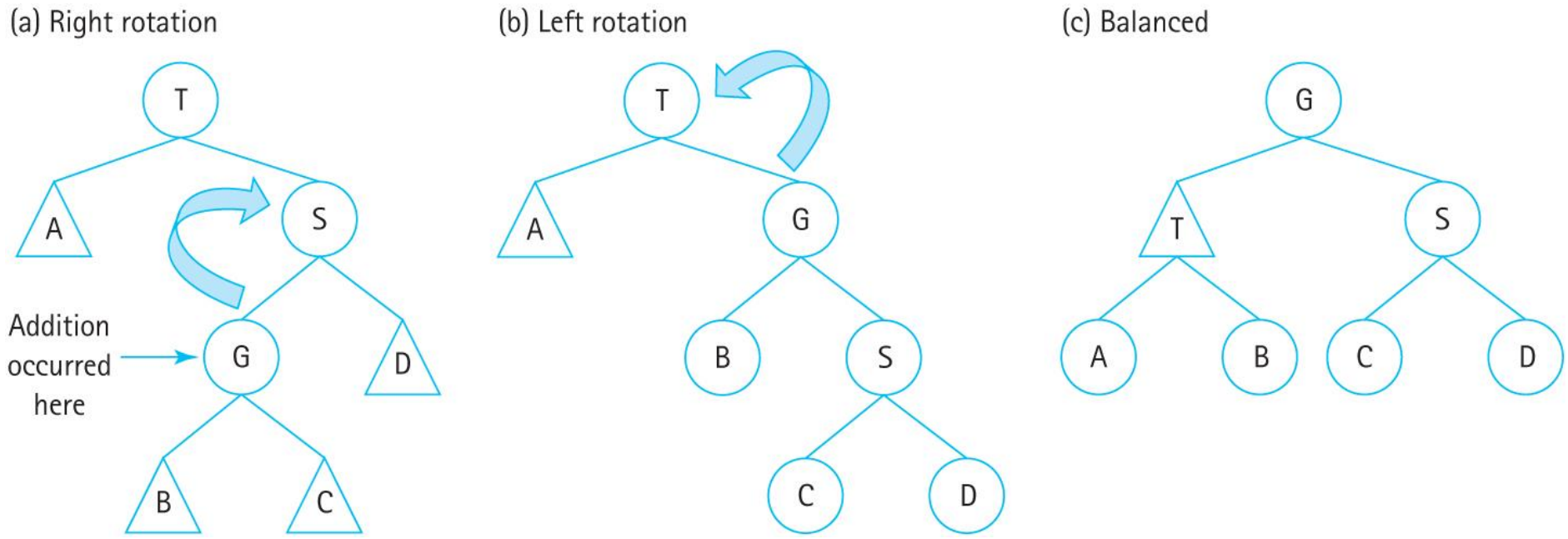


Figure 10.9 General right-left rotation on AVL tree (a) right-rotation (b) left-rotation (c) balanced

RotateLeftRight (cont.)

```
TreeNode* RotateLeftRight(TreeNode* T)
// Returns the tree node resulting from a
// left-right rotation.
{
  TreeNode* S = T->left;
  T->left = RotateLeft(S);
  return RotateRight(T);
}
```

RotateLeftRight (cont.)

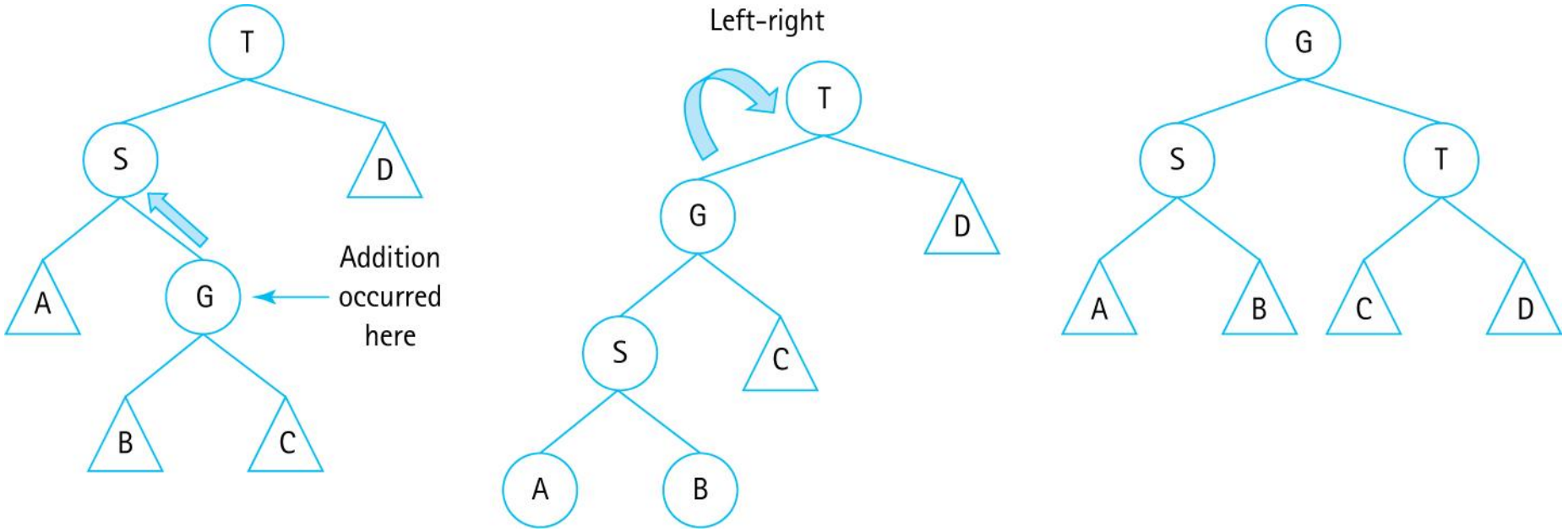


Figure 10.10 General left-right rotation on AVL tree

Difference

- Difference returns the difference in height between the two subtrees of the given node
- If the difference is positive, the left tree is taller
- If the difference is negative, the right tree is taller
- This is used by Balance to determine which rotation to use

Difference (cont.)

```
int Difference(TreeNode* T) const
// Returns the difference between the heights
// of the left and right subtrees of T.
// Assumes the given TreeNode* T is not null.
{
  return Height(T->left) - Height(T->right);
}
```

Height

- A recursive helper function that calculates the height of the tree rooted at the given node
- The height of a tree is the height of its tallest subtree + 1
- If the node is NULL, the height is 0 (base case)

Height (cont.)

```
int Height(TreeNode* T) const
// Returns the height of a tree T.
{
  if (T == null)
    return 0;
  else {
    int heightLeft = Height(T->left);
    int heightRight = Height(T->right);
    if (heightLeft > heightRight)
      return heightLeft + 1;
    else
      return heightRight + 1;
  }
}
```

Balance

- Balance is called by Insert to check if a rotation is needed to balance the tree
- If a node’s height difference (the balance factor) is greater than 1, it is unbalanced
- The Difference function is used to determine which rotation is needed

Balance (cont.)

```
TreeNode* Balance(TreeNode* T)
// Checks and balances the subtree T.
{
  int balanceFactor = Difference(T);
  if (balanceFactor > 1) { // Left subtree is taller
    if (Difference(T->left) > 1)
      return RotateRight(T);
    else
      return RotateLeftRight(T);
  }
  else if (balanceFactor < -1) { // Right subtree is taller
    if (Difference(T->right) < 0)
      return RotateLeft(T);
    else
      return RotateRightLeft(T);
  }
  else
    return T;
}
```

Insert and PutItem

- These methods are almost identical to the BST versions, except for calls to Balance
- Insert: After inserting into the left or right subtree, Balance is called on that subtree
- PutItem: After the call to Insert, the root is balanced

AVL Tree Limitations

- AVL Trees have increased overhead due to the rotation operations
- The cost is somewhat amortized across searches, but many updates will overwhelm the savings from balancing the tree
- Is there another structure that promises balanced trees and $O(\log_2 N)$ efficiency?

Red-Black Trees

- Red-Black trees reduce the number of rotations at the cost of a perfectly balanced tree
- This reduces overhead while still maintaining efficient $O(\log_2 N)$ search time
- They are widely used in many applications, such as the C++ standard library

Red-Black Tree Properties

1) A node in a Red-Black tree is labeled as either *red* or *black*
2) The root is labeled *black*
3) All leaves that are NULL are *black*
4) If a node is labeled *red*, both of its children must be labeled *black*
5) Every path from a given node to a NULL node must have the same number of black nodes

Red-Black Tree Properties

- Coloring NULL nodes black makes the implementation simpler
- Black nodes can have children of either color
- The property that all paths must have the same number of black nodes is called **black-height**

Red-Black Tree Examples

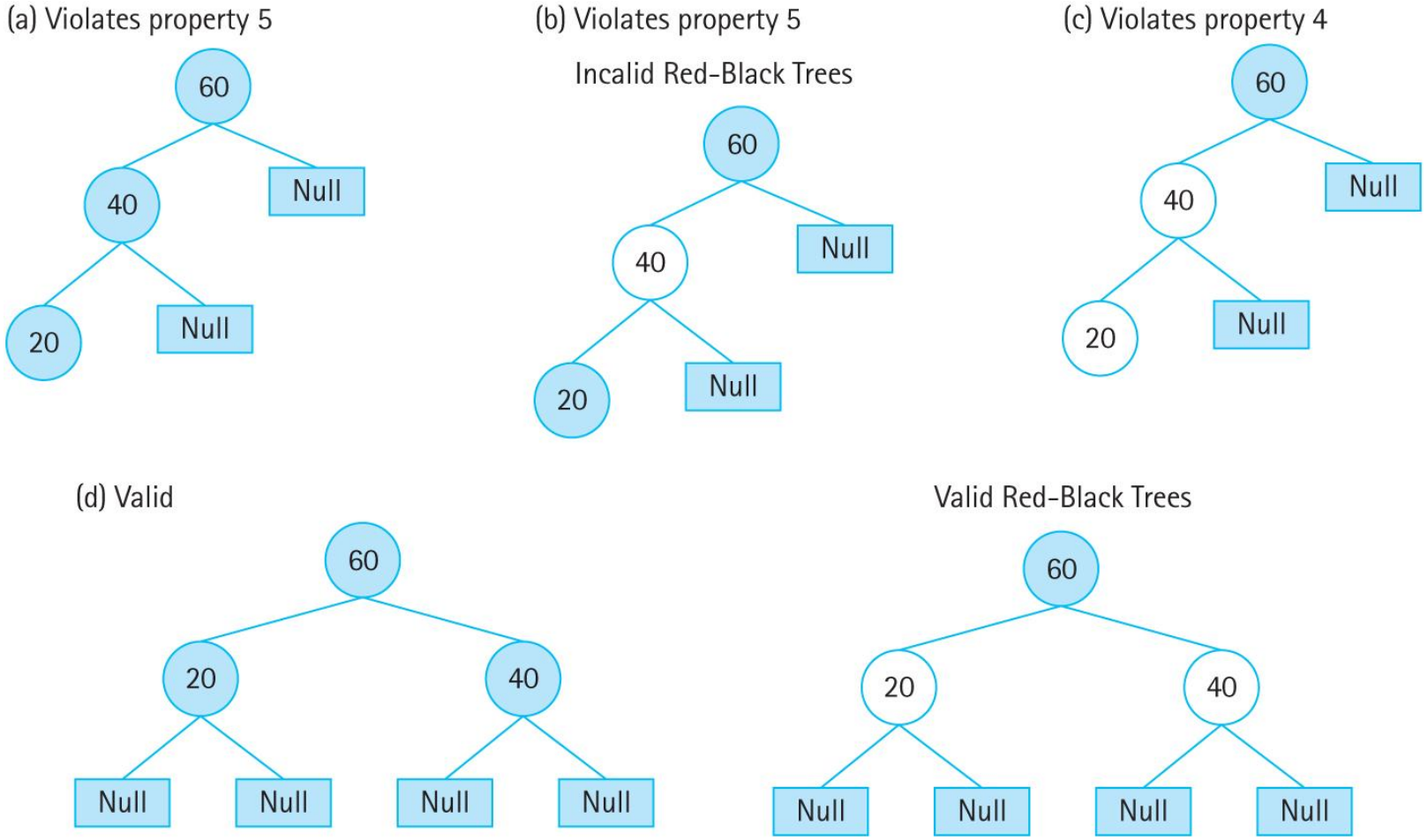


Figure 10.11 Valid/Invalid Red-Black Trees (a) violates property 5 (b) violates property 5 (c) violates property 4 (d) valid (e) valid red-black trees

Red-Black Tree Changes

- Red-Black trees require a few small changes
- Each node has a color, Red or Black
- Nodes also have a pointer to their parent node

Inserting into Red-Black Trees

- Red-Black trees use **recoloring** to determine if a tree is balanced
- If recoloring fails, the red-black properties are violated and rotations are needed
- New nodes are automatically labeled red
- Recoloring is best described in relation to a node’s “family”

Node Family Tree

- **Parent:** The node whose child is N
- **Uncle:** N's parent's sibling
- **Grandparent:** The parent of N's parent

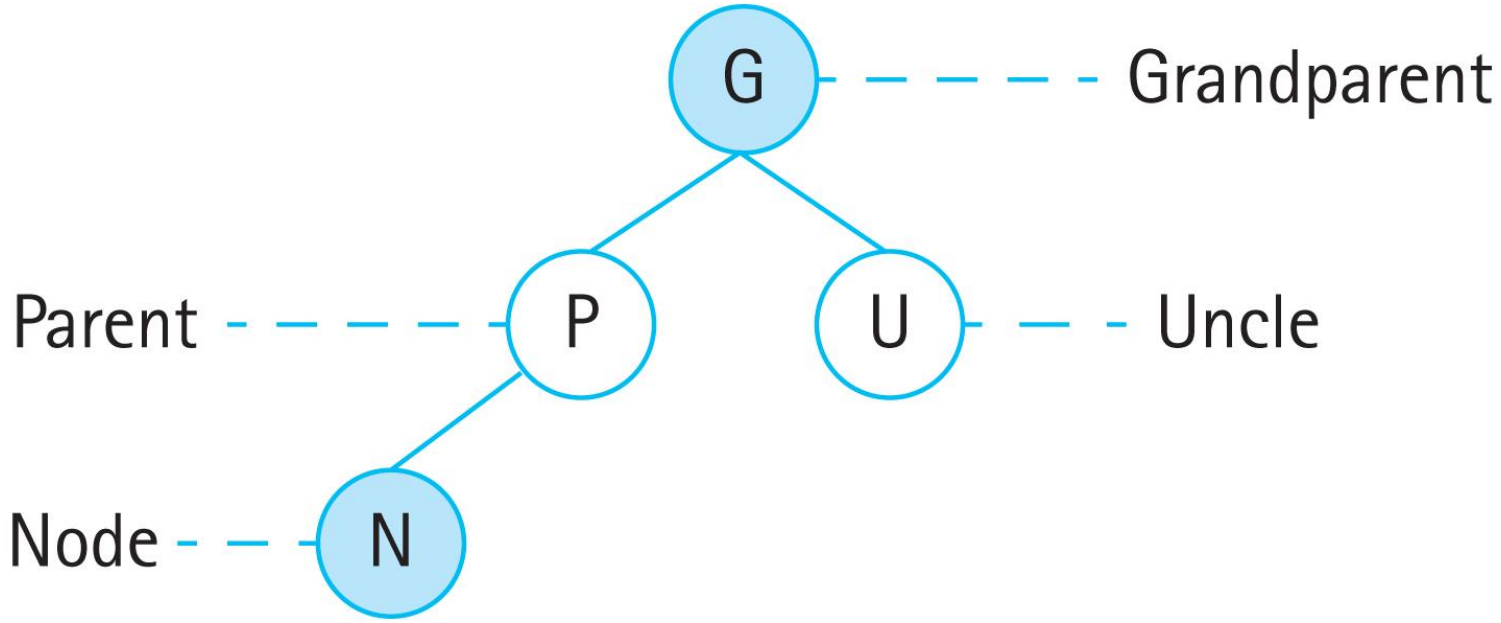


Figure 10.12 The Family Tree

Inserting into Red-Black Trees

- If N (the new node) is the root node, change its color to black
- If N’s parent is black, nothing needs to be done
- If N’s parent and uncle are both red, change both to black, change N’s grandparent to red and recursively recolor N’s grandparent
- If N’s parent is red and the uncle is black, the tree must be restructured

Recoloring

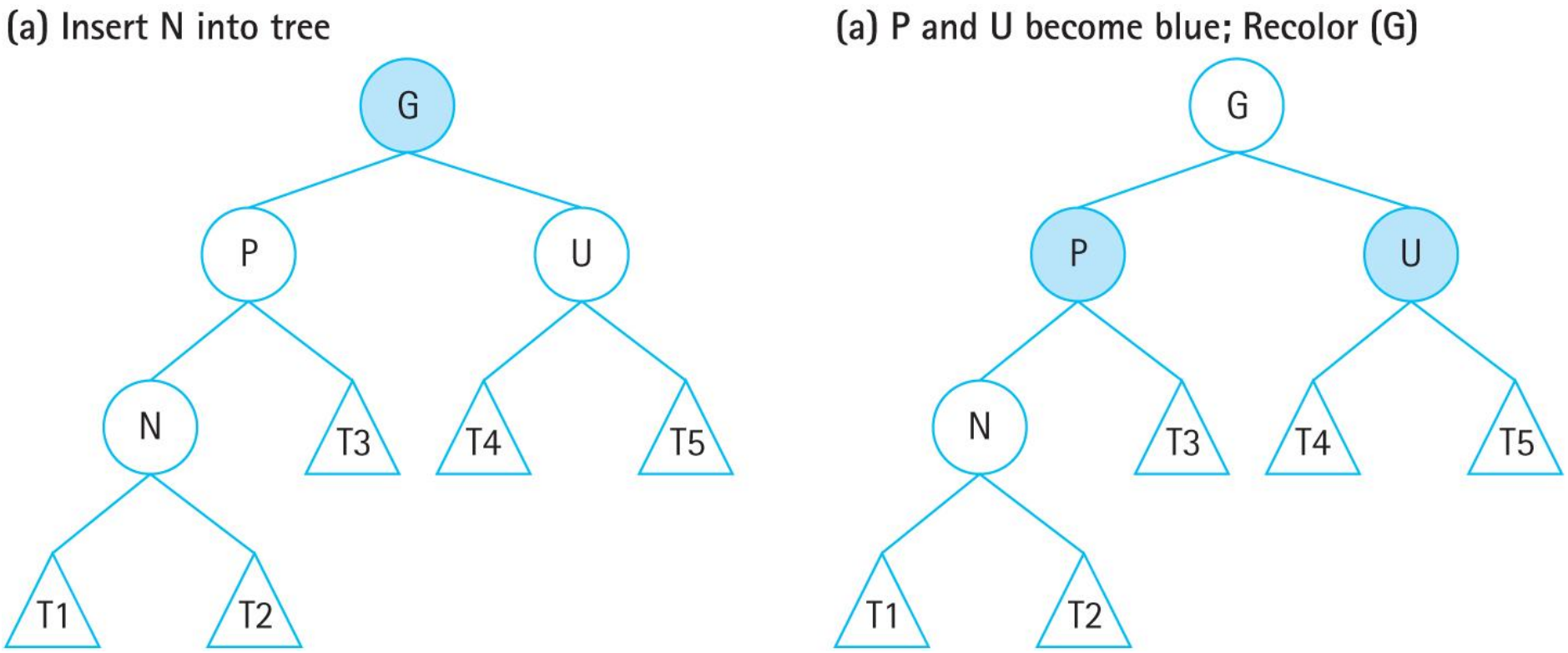


Figure 10.13 ReColor after Inserting N (a) Insert N into tree (b) P and U become black; ReColor(G)

Restructuring

- Like AVL trees, there are four cases for using rotations to balance the tree
- However, the cases are based on the colors of the nodes, not on the balance factor
- Restructuring also involves swapping colors

The Four Restructuring Cases

- N is the left child of P, P is the left child of G:
  - Right rotation and swap the colors of G and P
- N is the right child of P, P is the left child of G:
  - Left-Right rotation and swap G’s and N’s colors
- N is the right child of P, P is the right child of G:
  - Left rotation and swap the colors of G and P
- N is the left child of P, P is the right child of G:
  - Right-Left rotation and swap G’s and N’s colors

The Four Restructuring Cases

- These should look familiar: They’re the same cases as with AVL trees
- Instead of rotating based on which subtree of which child was changed, it looks at the colors of the parent and grandparent
- Note the parallels between cases
  - Single rotations swap the colors of P and G
  - Double rotations swap the colors of G and N

Left-Left Restructuring

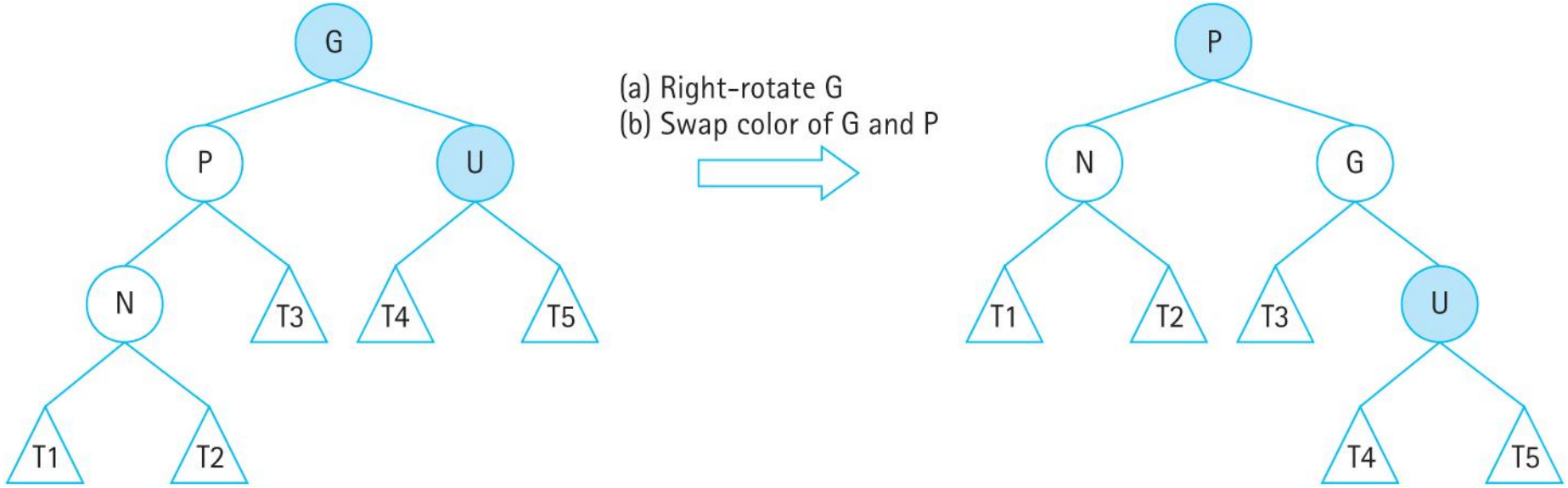


Figure 10.14 Left-Left Restructuring

Left-Right Restructuring

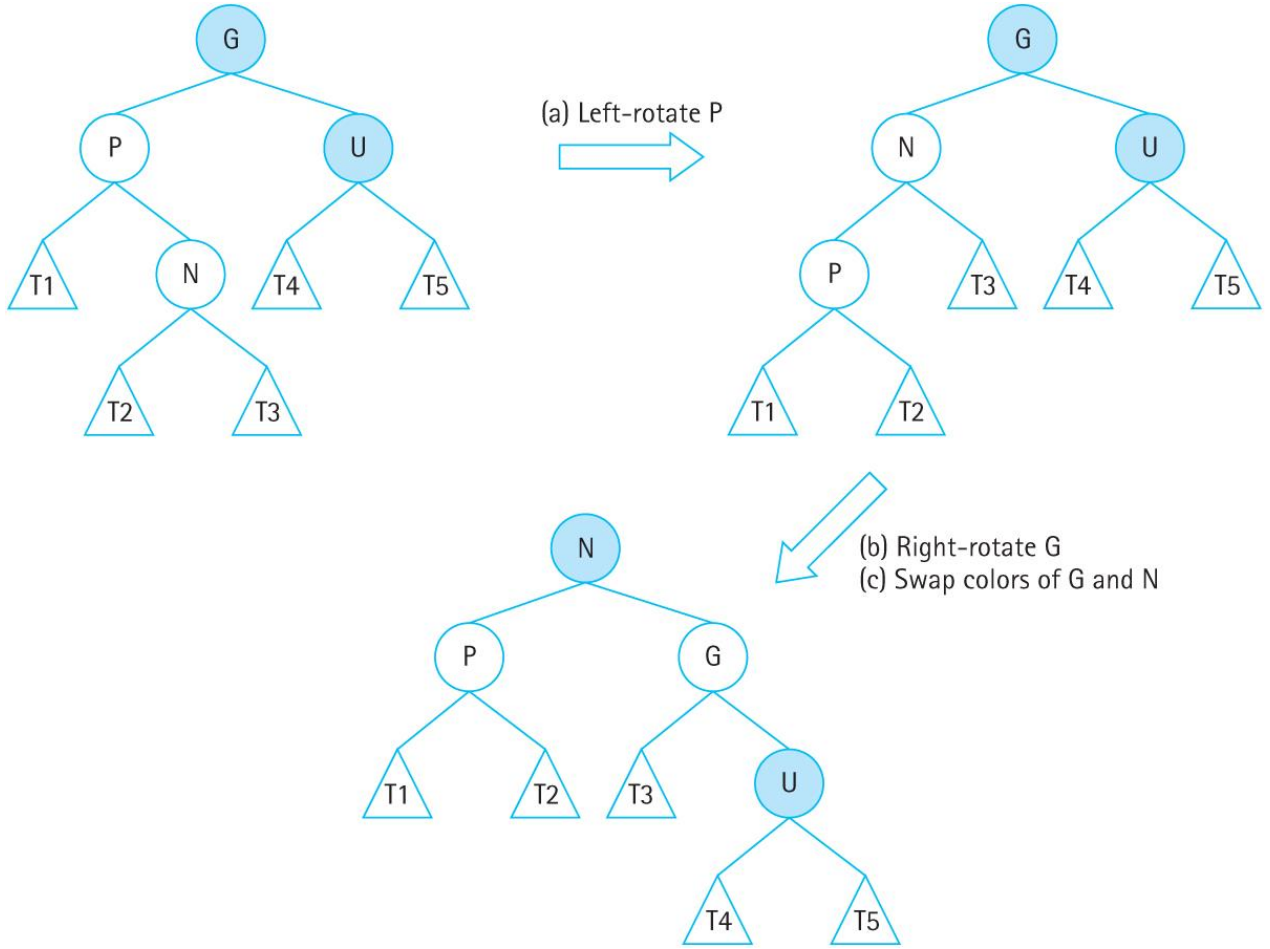


Figure 10.15 Left-Right Restructuring

Right-Right Restructuring

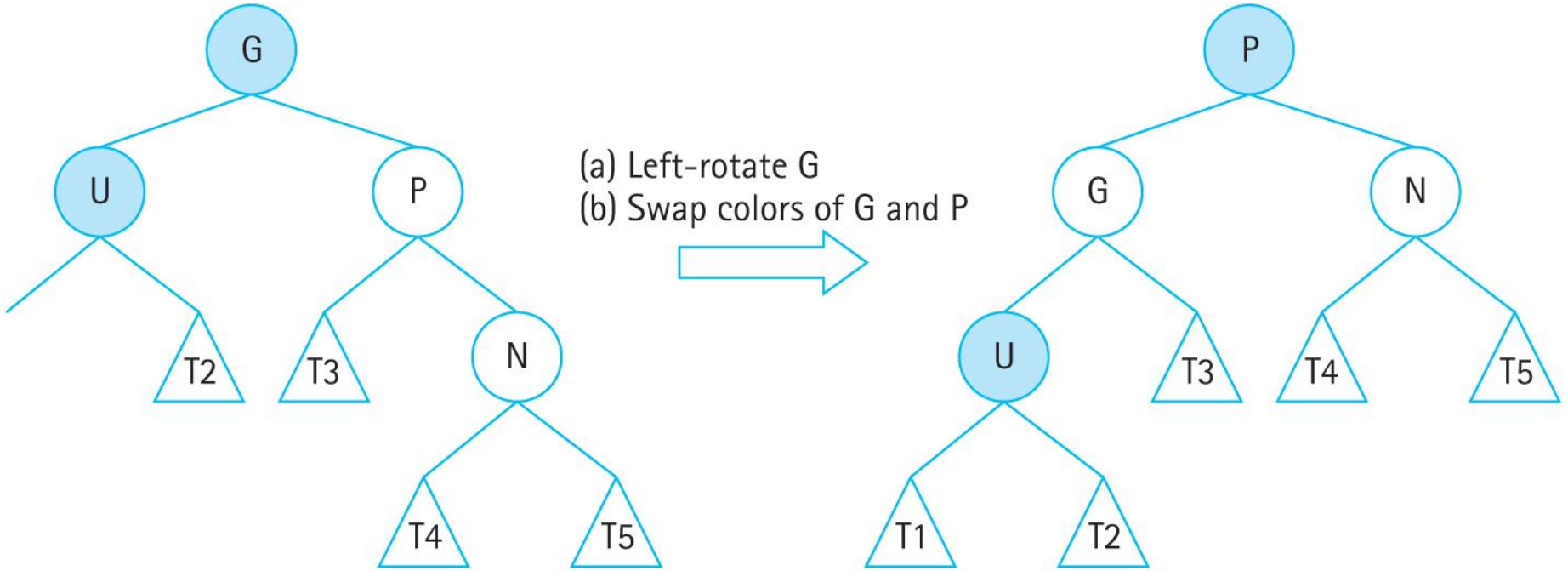


Figure 10.16 Right-Right Restructuring

Right-Left Restructuring

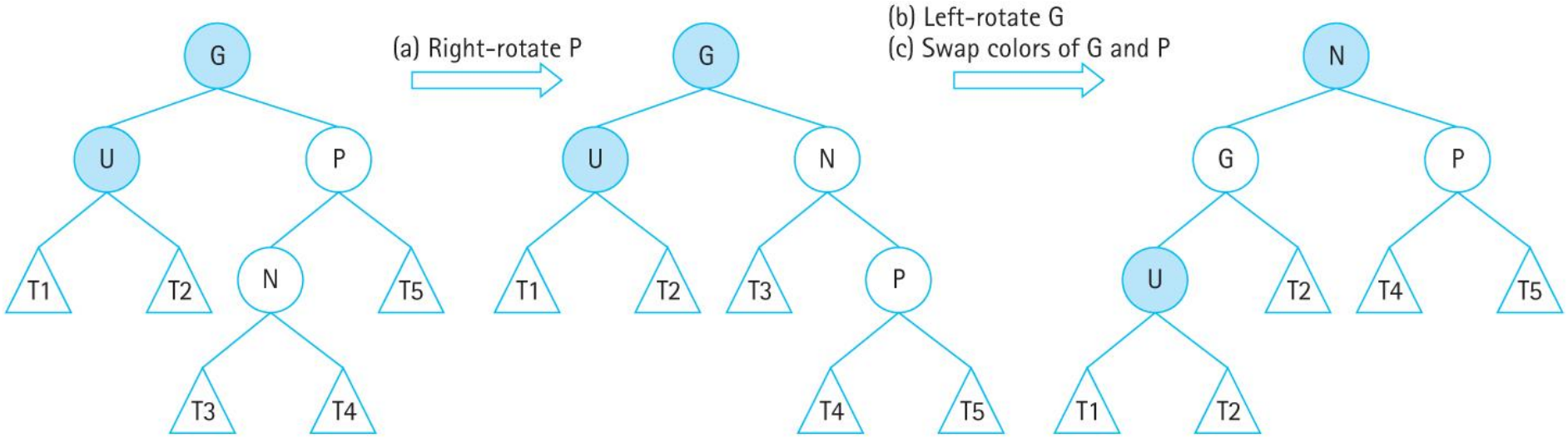


Figure 10.17 Right-Left Restructuring

Implementing Recoloring

- The red/black color is coded as an enum
- Each node has a color and a pointer to its parent in addition to the usual BST fields
- Helper functions are used for getting the parent, grandparent, and uncle of a node to improve the clarity of the code
- Two more helpers are used to get and set the color of a node

Recolor

```
void ReColor(TreeNode* N)
{
  if (N == NULL) return; // do nothing if N is NULL
  TreeNode* P = Parent(N);
  TreeNode* G = GrandParent(N);
  TreeNode* U = Uncle(N);
  if (P == NULL) N->color = BLACK; // N is the root node
  else if (P != NULL || GetColor(P) != BLACK) {
    if (GetColor(U) == RED) {
      SetColor(P, BLACK);
      SetColor(U, BLACK);
      SetColor(G, RED);
      ReColor(G);
    }
    else
      ReStructure(N); // Left as an exercise for the reader
  }
}
```

PutItem

- The algorithm is generally the same, except for coloring the node
- All nodes are red when inserted, and a call to Recolor fixes the tree
- The PutItem implementation walks the tree to find the correct place to insert the new node, then calls Recolor on the new node

Insert

```
void Insert(TreeNode*& tree, TreeNode* parent, ItemType item)
// Inserts item into tree.
// Post: item is in tree; search property is maintained.
{
  if (tree == NULL)
  {// Insertion place found.
    tree = new TreeNode;
    tree->right = NULL;
    tree->left = NULL;
    tree->parent = parent;
    tree->info = item;
    ReColor(tree);
  }
  else if (item < tree->info)
    Insert(tree->left, tree, item); // Insert in left subtree.
  else
    Insert(tree->right, tree, item); // Insert in right subtree.
}
```

B-Trees

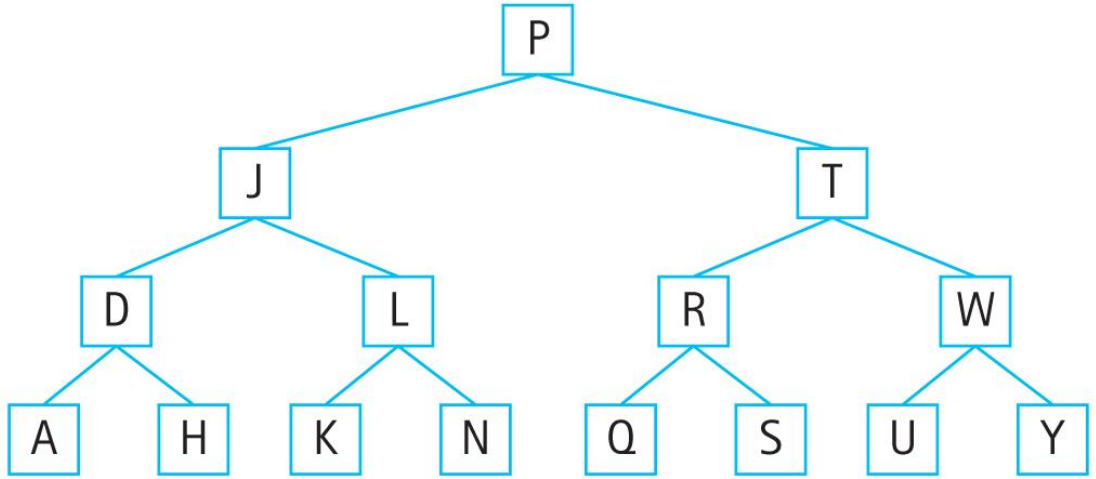
- **B-Trees** are a generalization of binary trees
- Each node in a B-Tree contains multiple data values and links to multiple children
- Nodes in a B-Tree have a maximum number of children, *m*; this is the **order** of the B-Tree
- B-Trees are self-balancing with $O(\log_2 N)$ search

B-Tree Properties

1) Each node must have at most *m* children
2) Each internal node has at least *m*/2 children
3) The root node must have at least 2 children, if it is not a leaf
4) A nonleaf node with *k* children must have k-1 data items
5) All leaves must appear at the same level of the tree

B-Tree Examples

(a) B-Tree of order 2



(b) B-Tree of order 5

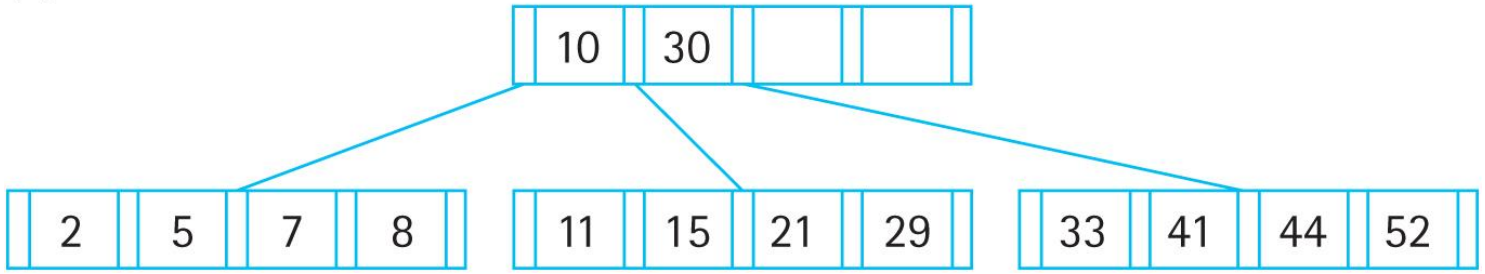


Figure 10.18 B-tree Examples (a) B-Tree of order 2 (b) B-Tree of order 5

Searching a B-Tree

- Searching a B-Tree requires comparing the item to multiple keys in each node
- If the item is less than the first key, the first child is searched
- If the item is greater than the last key, the last child is searched
- If the item is between two keys, the child that is between those two keys is searched

Searching Walkthrough

Using the B-Tree of order 5, search for the key 21:

- Is 21 <= 10? No. Check the next key.
- Is 21 <= 30? Yes. It’s not equal, so proceed to the second child.
- Is 21 <= 11? No.
- Is 21 <= 15? No.
- Is 21 <= 21? Yes. The value has been found.

Inserting into B-Trees

- Unsurprisingly, insertion (and deletion) into a B-Tree is somewhat complex
- Insertion attempts to fill nodes that don’t have a full complement of keys
- If a node is full, the algorithm splits nodes to make more room

Splitting Nodes

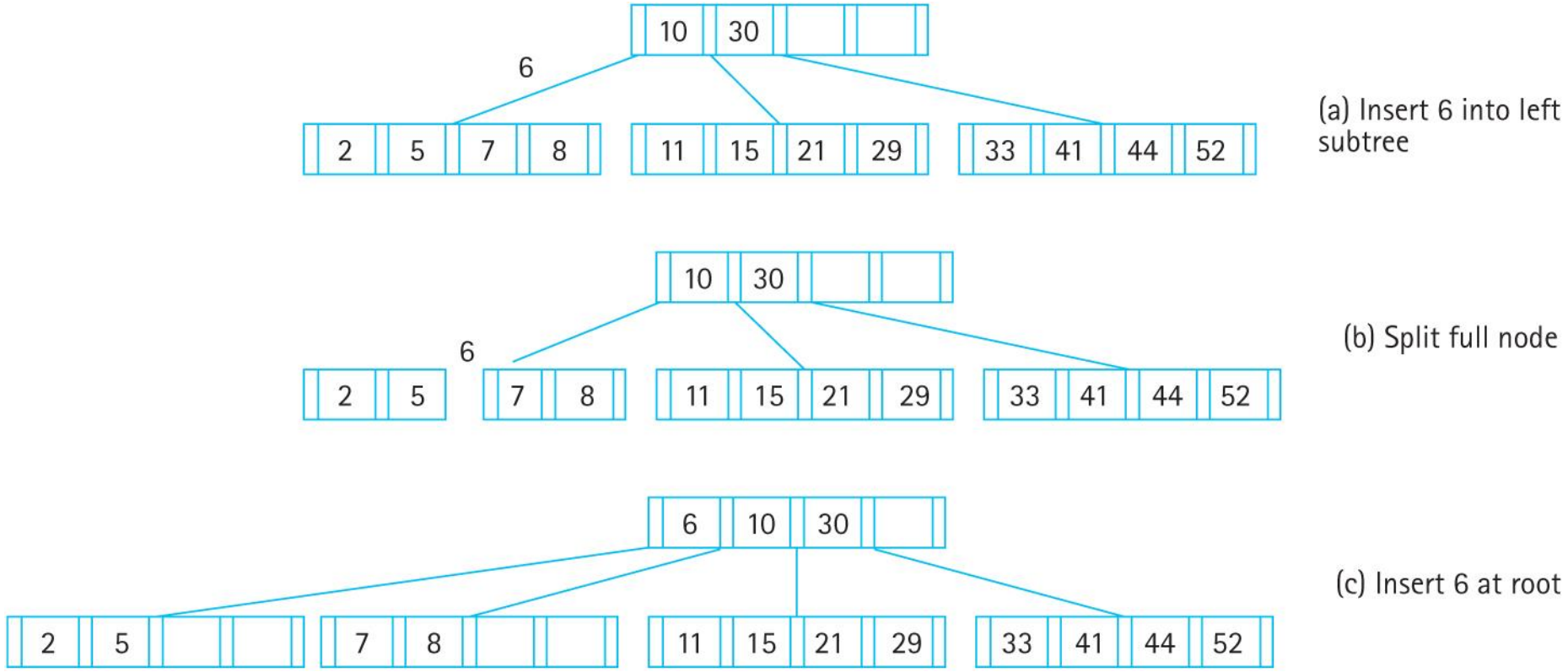


Figure 10.19 B-tree Insertion (a) Insert 6 into left subtree (b) Split full node (c) Insert 6 at root

InsertBTree (Node N)

- Search the B-Tree for the leaf N should be inserted into
- If there are fewer than *m-1* keys in the leaf, add N into the appropriate place, shifting other keys if necessary
- If there is no room, the node must be split

InsertBTree (Node N) (cont.)

- Split the node into two equal parts and one middle element
- Add the middle element to the parent and add the two parts as children
- If the parent is full, continue splitting recursively up the tree
- Split the root and add a new root node if necessary

Splitting Nodes

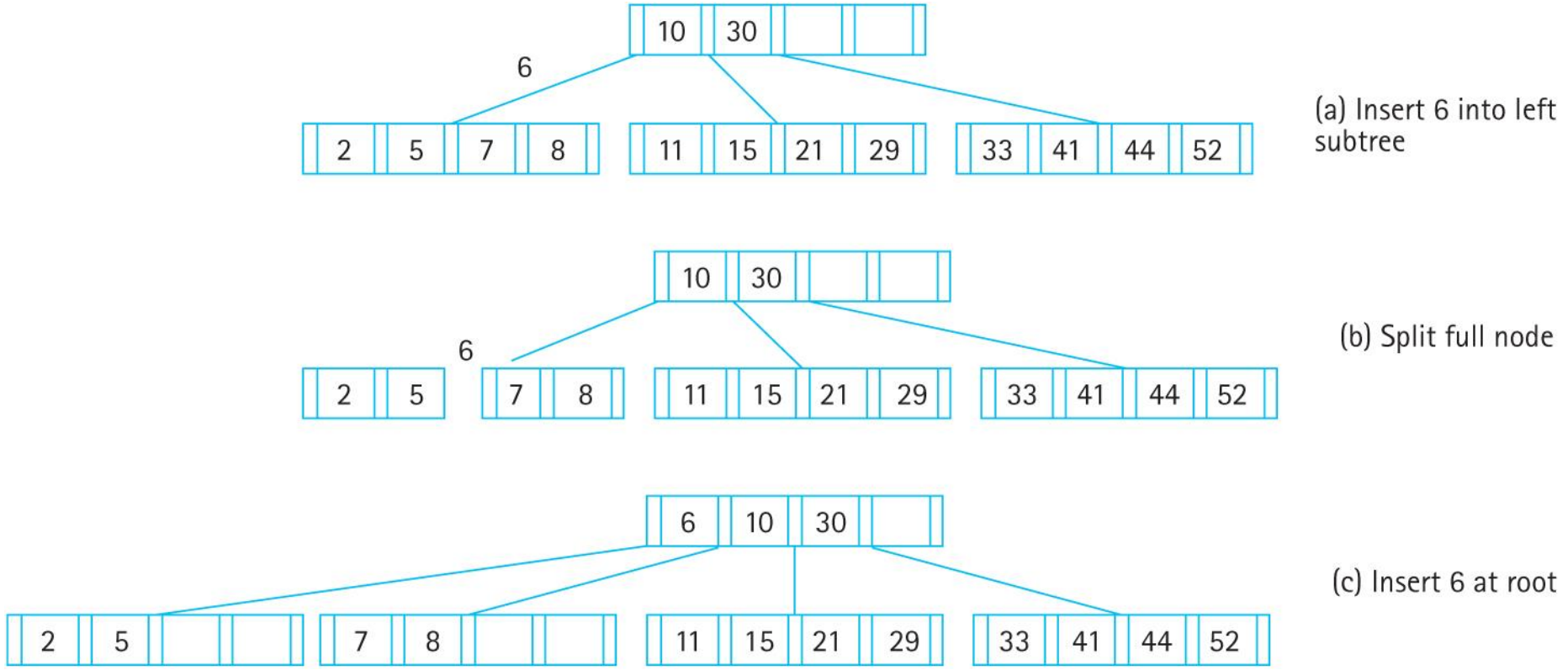


Figure 10.19 B-tree Insertion (a) Insert 6 into left subtree (b) Split full node (c) Insert 6 at root

B-Tree Applications

- B-Trees are well suited to being written to and read from disks
- Each node can the size of an I/O block, which can be read/written quickly, by setting the order of the B-Tree to an appropriate size
- Applications like databases and file systems, which can store terabytes of data, are often implemented using B-Trees

The End!