Binary Trees

CSC 249

General Properties

Generally, trees:

- Consist of nodes and edges (which connect nodes)
- Begin with a root node, and the tree structure is usually defined as a simple wrapper of the root node
- Grow downwards, from root, to branch (internal) nodes, to leaf nodes at the bottom.
- For a *balanced* tree, the height is log 2 (number of nodes)

A **binary tree** is made up of a finite set of elements called **nodes**. This set either is empty or consists of a node called the **root** together with two binary trees, called the left and right **subtrees**, which are disjoint from each other and from the root. (Disjoint means that they have no nodes in common.) The roots of these subtrees are **children** of the root. There is an **edge** from a node to each of its children, and a node is said to be the **parent** of its children.

If n_1, n_2, ..., n_k is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a **path** from n_1 to n_k. The **length** of the path is $k - 1$. If there is a path from node *R* to node *M*, then *R* is an **ancestor** of *M*, and *M* is a **descendant** of *R*. Thus, all nodes in the tree are descendants of the

root of the tree, while the root is the ancestor of all nodes. The **depth** of a node *M* in the tree is the length of the path from the root of the tree to *M*. The **height** of a tree is one more than the depth of the deepest node in the tree. All nodes of depth d are at **level** d in the tree. The root is the only node at level 0, and its depth is 0. A **leaf** node is any node that has two empty children. An **internal** node is any node that has at least one non-empty child.

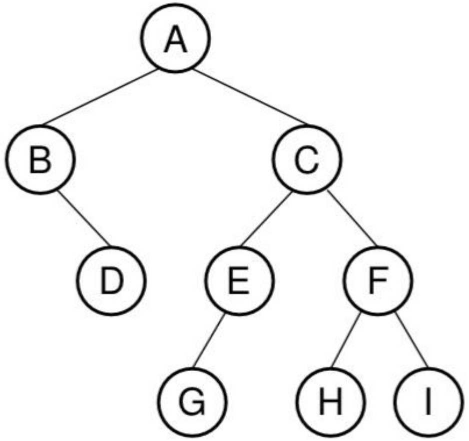


Figure 5.1 A binary tree. Node *A* is the root. Nodes *B* and *C* are *A*'s children. Nodes *B* and *D* together form a subtree. Node *B* has two children: Its left child is the empty tree and its right child is *D*. Nodes *A*, *C*, and *E* are ancestors of *G*. Nodes *D*, *E*, and *F* make up level 2 of the tree; node *A* is at level 0. The edges from *A* to *C* to *E* to *G* form a path of length 3. Nodes *D*, *G*, *H*, and *I* are leaves. Nodes *A*, *B*, *C*, *E*, and *F* are internal nodes. The depth of *I* is 3. The height of this tree is 4.

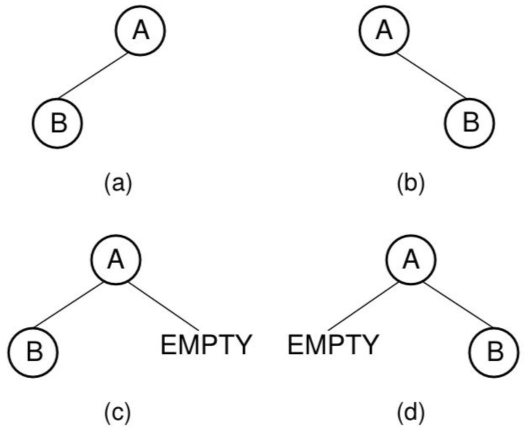


Figure 5.2 Two different binary trees. (a) A binary tree whose root has a non-empty left child. (b) A binary tree whose root has a non-empty right child. (c) The binary tree of (a) with the missing right child made explicit. (d) The binary tree of (b) with the missing left child made explicit.

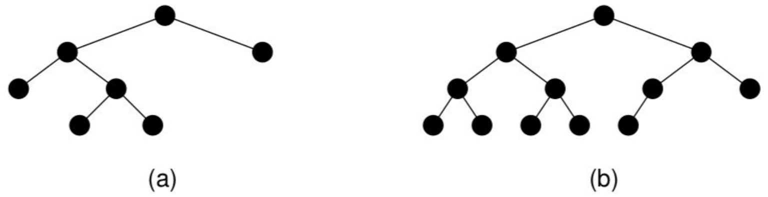


Figure 5.3 Examples of full and complete binary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

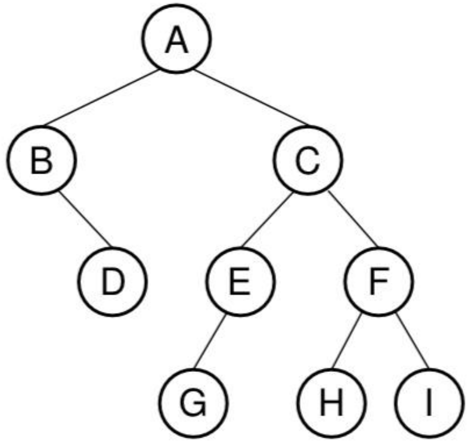
```
// Binary tree node abstract class
template <typename E> class BinNode {
public:
  virtual ~BinNode() {} // Base destructor

  // Return the node's value
  virtual E& element() = 0;

  // Set the node's value
  virtual void setElement(const E&) = 0;

  // Return the node's left child
  virtual BinNode* left() const = 0;

  // Set the node's left child
  virtual void setLeft(BinNode*) = 0;

  // Return the node's right child
  virtual BinNode* right() const = 0;

  // Set the node's right child
  virtual void setRight(BinNode*) = 0;

  // Return true if the node is a leaf, false otherwise
  virtual bool isLeaf() = 0;
};
```

Sample Interface

Figure 5.5 A binary tree node ADT.

Traversal

- Preorder:
  - Visit node
  - Visit left subtree
  - Visit right subtree
- Postorder:
  - Visit left subtree
  - Visit right subtree
  - Visit node
- Inorder:
  - Visit left subtree
  - Visit node
  - Visit right subtree



Example 5.1 The preorder enumeration for the tree of Figure 5.1 is

ABDCEGFHI.

The first node printed is the root. Then all nodes of the left subtree are printed (in preorder) before any node of the right subtree.

Alternatively, we might wish to visit each node only *after* we visit its children (and their subtrees). For example, this would be necessary if we wish to return all nodes in the tree to free store. We would like to delete the children of a node before deleting the node itself. But to do that requires that the children's children be deleted first, and so on. This is called a **postorder traversal**.

Example 5.2 The postorder enumeration for the tree of Figure 5.1 is

DBGEHIFCA.

An **inorder traversal** first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree). The

Sample Implementation

```
// Simple binary tree node implementation
template <typename Key, typename E>
class BSTNode : public BinNode<E> {
private:
  Key k;                      // The node's key
  E it;                       // The node's value
  BSTNode* lc;                // Pointer to left child
  BSTNode* rc;                // Pointer to right child

public:
  // Two constructors -- with and without initial values
  BSTNode() { lc = rc = NULL; }
  BSTNode(Key K, E e, BSTNode* l =NULL, BSTNode* r =NULL)
    { k = K; it = e; lc = l; rc = r; }
  ~BSTNode() {}                 // Destructor

  // Functions to set and return the value and key
  E& element() { return it; }
  void setElement(const E& e) { it = e; }
  Key& key() { return k; }
  void setKey(const Key& K) { k = K; }

  // Functions to set and return the children
  inline BSTNode* left() const { return lc; }
  void setLeft(BinNode<E>* b) { lc = (BSTNode*)b; }
  inline BSTNode* right() const { return rc; }
  void setRight(BinNode<E>* b) { rc = (BSTNode*)b; }

  // Return true if it is a leaf, false otherwise
  bool isLeaf() { return (lc == NULL) && (rc == NULL); }
};
```

Figure 5.7 A binary tree node class implementation.

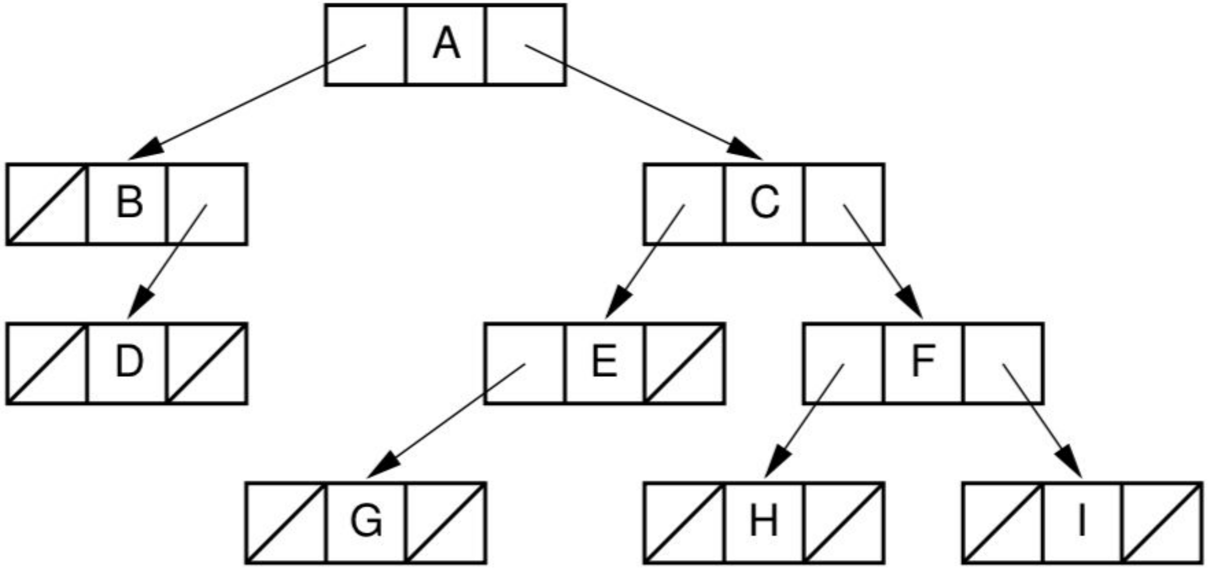


Figure 5.8 Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value.

BST Operations

Search

Insert

Remove

Traverse (Print the tree)

Search

For BST:

- Is a Tree
- is Binary (0-2 children per node)
- has the **Search Property:**

```
left child <= node <= right child
```

(The Invariant property that defines a particular Binary Tree as a Binary Search Tree)

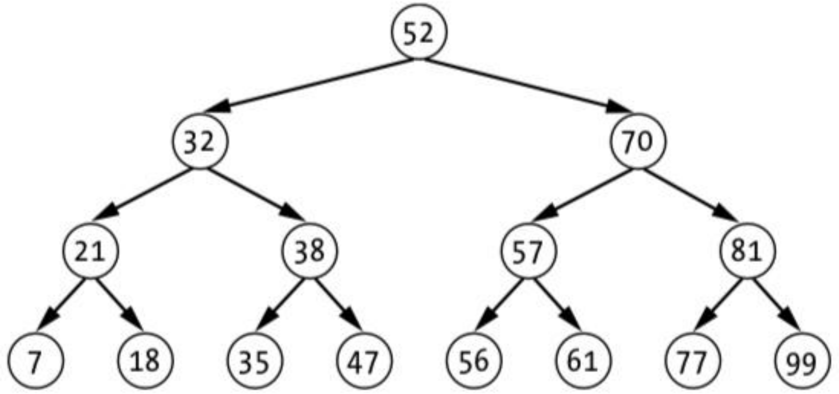


Figure 5-4: The values of the nodes in a binary search tree are ordered by the binary search tree property.

The power of the binary search tree stems from how values are organized within the tree. The *binary search tree property* states:

> For any node *N*, the value of any node in *N*'s left subtree is less than *N*'s value, and the value of any node in *N*'s right subtree is greater than *N*'s value.

In other words, the tree is organized by the values at each node, as shown in Figure 5-4. The values of the data in the left node and all nodes below it are less than the value of the current node. Similarly, the values of the data in the right node and all nodes below it are greater than the value of the current node. The values thus serve two roles. First, and most obviously, they indicate the value stored at that node. Second, they define the tree's structure below that node by partitioning the subtree into two subsets.

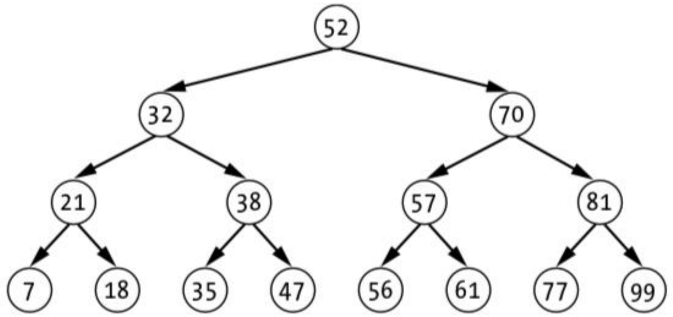


Figure 5-4: The values of the nodes in a binary search tree are ordered by the binary search tree property.

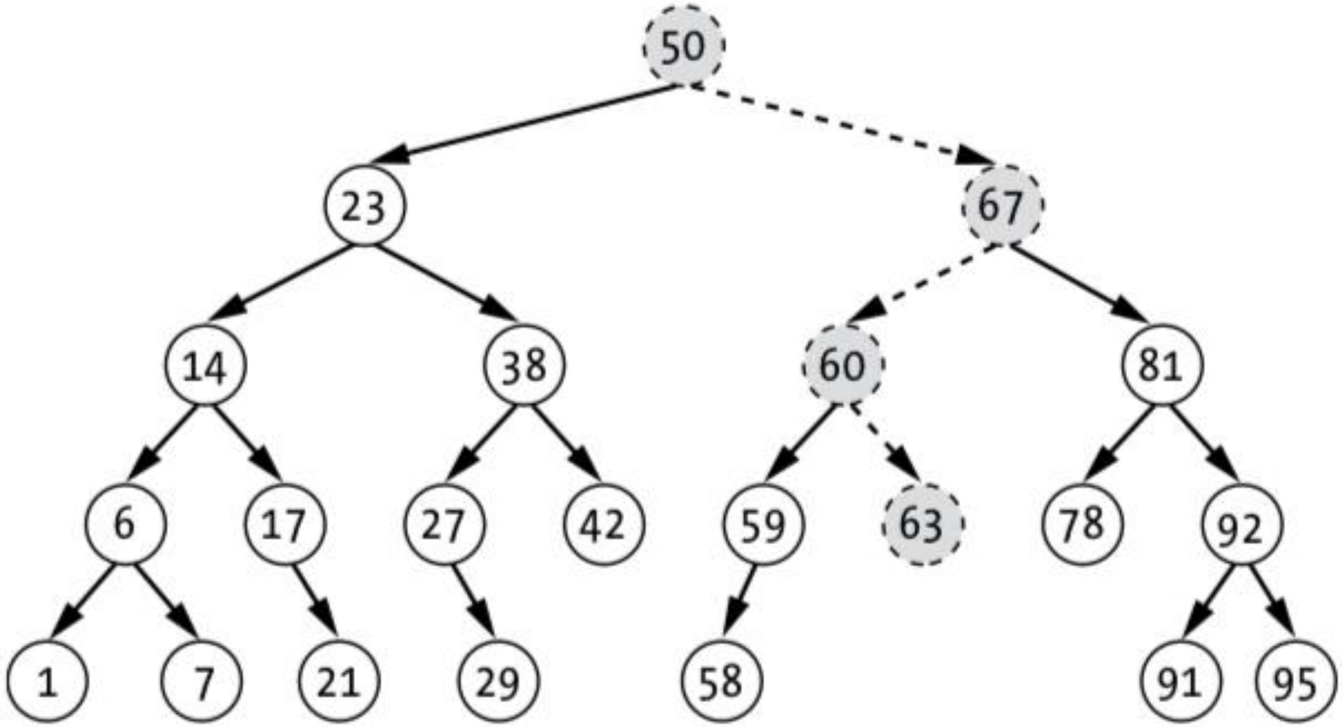


Figure 5-7: The complete path of a search of a binary search tree for value 63

Implementing Search

We can simplify the logic for using binary search trees by wrapping the entire tree in a thin data structure that contains the root node:

```
BinarySearchTree {
    TreeNode: root
}
```

While this might seem like a waste (more complexity and an extra data structure), it provides an easy-to-use interface for the tree and greatly simplifies our handling of the root node. When using a wrapper data structure (or class) for our binary search tree, we also need to provide top-level functions to add or find nodes. These are relatively thin wrappers with a special case for handling a tree without any nodes.

To search a tree, the code again starts by checking whether the tree is empty (`tree.root == null`):

```
FindTreeNode(BinarySearchTree: tree, Type: target):
    IF tree.root == null:
        return null
    return FindValue(tree.root, target)
```

Iterative and Recursive Searches

We implement this search with either an iterative or recursive approach. The following code uses a recursive approach, where the search function calls itself using the next node in the tree, initially called on the root node of the tree. The code returns a pointer to the node containing the value, allowing us to retrieve any auxiliary information from the node.

```
FindValue(TreeNode: current, Type: target):
  ❶ IF current == null:
         return null
  ❷ IF current.value == target:
         return current
  ❸ IF target < current.value AND current.left != null:
         return FindValue(current.left, target)
  ❹ IF target > current.value AND current.right != null:
         return FindValue(current.right, target)
  ❺ return null
```

This algorithm performs only a few tests at each node; if any of the tests pass, we end the function by returning a value. First, the code checks that the current node is not null, which can happen when searching an empty tree. If it is null, the tree is empty and, by definition, does not contain the value of interest ❶. Second, if the current node's value equals our target value, the code has found the value of interest and returns the node ❷. Third, the code checks whether it should explore the left subtree and, if so, returns whatever it finds from that exploration ❸. Fourth, the code checks whether it should explore the right subtree and, if so, return whatever it finds from that exploration ❹. Note that in both the left and right cases, the code also checks that the corresponding child exists. If none of the tests trigger, the code has made it to a node that doesn't match our target value and does not have a child in the correct direction. It has reached a dead end and is forced to admit defeat by returning a failure value such as null ❺. A dead end occurs whenever there is no child in the correct direction, so it is possible for an internal node with a single child to still be a dead end for a search.

```
FindValue(TreeNode: current, Type: target):
 ❶ IF current == null:
        return null
 ❷ IF current.value == target:
        return current
 ❸ IF target < current.value AND current.left != null:
        return FindValue(current.left, target)
 ❹ IF target > current.value AND current.right != null:
        return FindValue(current.right, target)
 ❺ return null
```

The iterative approach to searching a binary search tree replaces the recursion with a WHILE loop that iterates down the tree. The search again starts at the tree's root.

```
FindValueItr(TreeNode: root, Type: target):
 ❶ TreeNode: current = root
 ❷ WHILE current != null AND current.value != target:
     ❸ IF target < current.value:
            current = current.left
        ELSE:
            current = current.right
 ❹ return current
```

The code starts by creating a local variable current to point to the current node in the search ❶. Initially, this will be the root node, which may be null in an empty tree. Then a WHILE loop keeps iterating down the tree until it either hits a dead end (current == null) or finds the correct value (current.value == target) ❷. Within the loop, the code checks whether the next child should be to the left or right ❸ and reassigns current to point to the corresponding child. The function concludes by returning current, which is either the found node or, if the tree is empty or the value is not found, null ❹.

Insertion

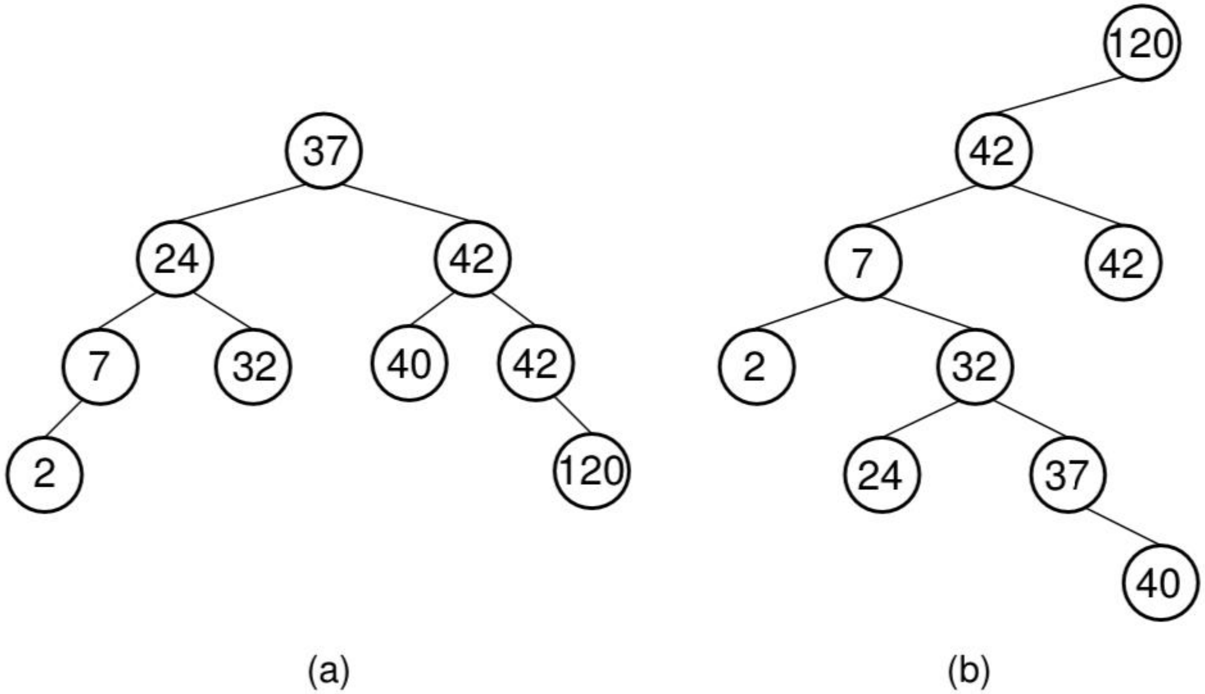


Figure 5.13 Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

with insertion, ORDER MATTERS!

As with our search function, we start with a wrapper function for
tion that handles the case of empty trees:

```
InsertTreeNode(BinarySearchTree: tree, Type: new_value):
    IF tree.root == null:
        tree.root = TreeNode(new_value)
    ELSE:
        InsertNode(tree.root, new_value)
```

First, the code checks whether the tree is empty (`tree.root == null` so, it creates a new root node with that value. Otherwise, it calls `Insert` on the root node, kicking off the recursive process from below. Thus, can ensure that `InsertNode` is called with a valid (non-null) node.

```
InsertNode(TreeNode: current, Type: new_value):
 ❶ IF new_value == current.value:
        Update node as needed
        return
 ❷ IF new_value < current.value:
     ❸ IF current.left != null:
            InsertNode(current.left, new_value)
        ELSE:
            current.left = TreeNode(new_value)
            current.left.parent = current
    ELSE:
     ❹ IF current.right != null:
            InsertNode(current.right, new_value)
        ELSE:
            current.right = TreeNode(new_value)
            current.right.parent = current
```

For example, if we want to add the number 77 to the binary search tree in Figure 5-9, we progress down through nodes 50, 67, 81, and 78 until we hit a dead end at the node with value of 78. At this point, we find ourselves without a valid child in the correct direction. Our search is at a dead end. We create a new node with the value of 77 and make it the node 78's left child.

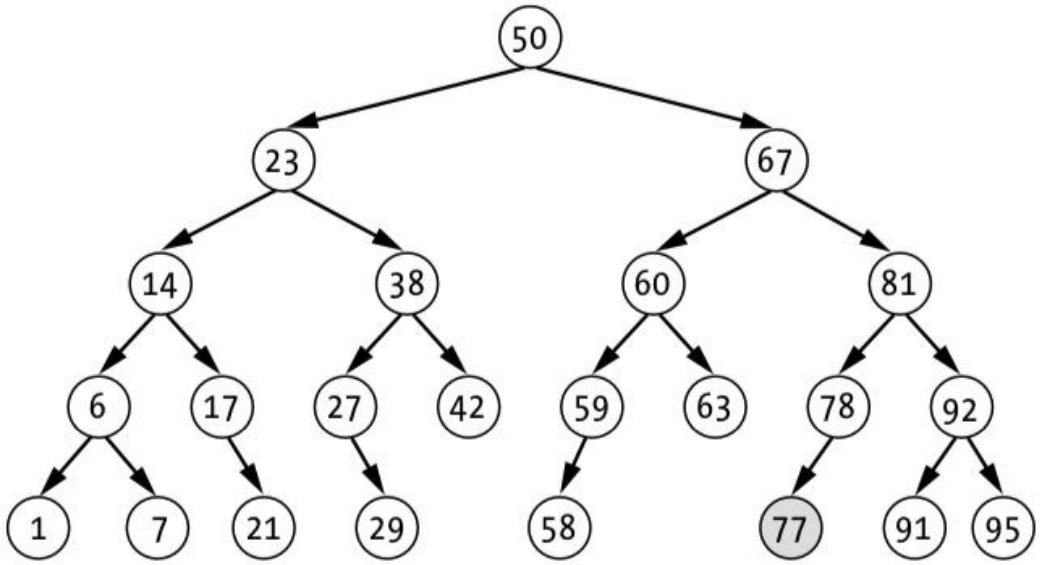


Figure 5-9: Inserting the value 77 into our binary search tree

Delete

This operation is a bit too complex to fully get into at this time, but we'll look at the cases involved for:

- if the node has no children
- or one child
- or two children

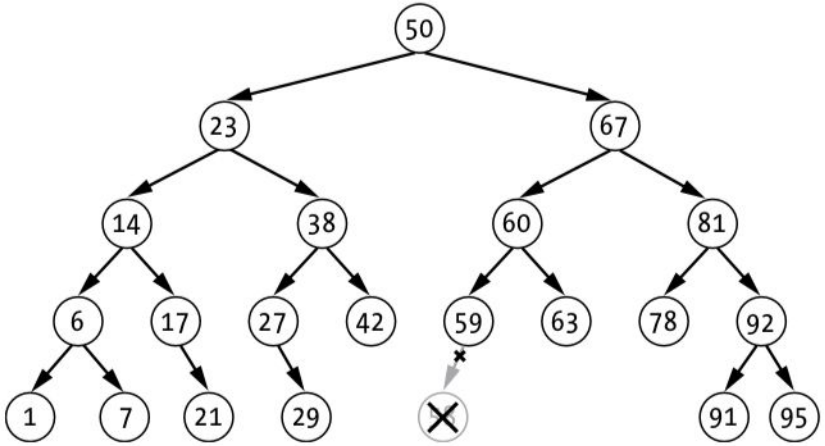


Figure 5-10: Remove a leaf node from a binary search tree by deleting it and updating the pointer from its parent node.

Removing leaf nodes shows the value of storing a pointer to the parent node: it allows us to search for the node to delete, follow the parent pointer back to that node's parent, and set the corresponding child pointer to `null`. Storing this single piece of additional data makes the deletion operation much simpler.

If the target node has a single child, we remove it by promoting that single child to be the child of the deleted node's parent. This is like removing a manager from our reporting hierarchy without shuffling anyone else around. When the manager leaves, their boss assumes management of the former employee's single direct report. For example, if we wanted to remove node 17 from our example tree, we could simply shift node 21 up to take its place as shown in Figure 5-11. Node 14 now links directly to node 21.

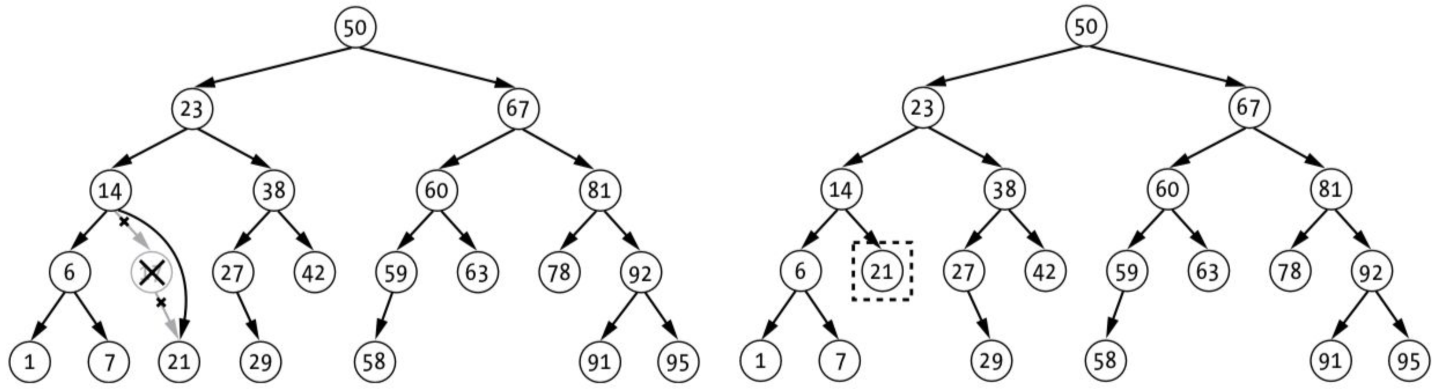
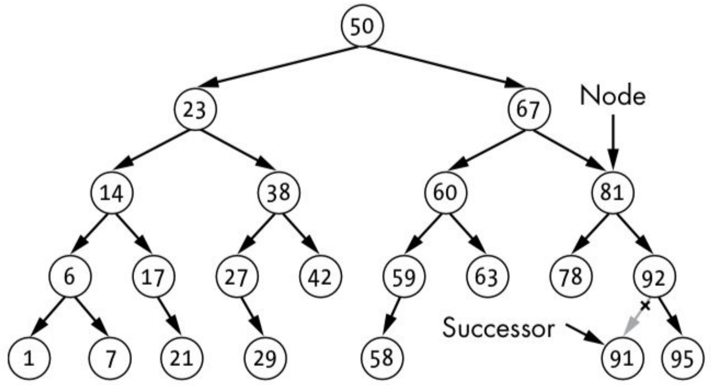
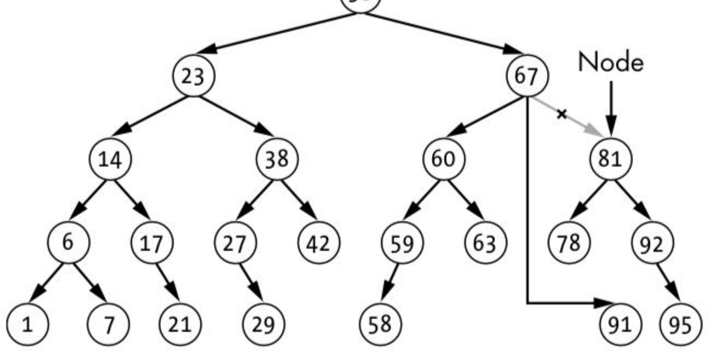


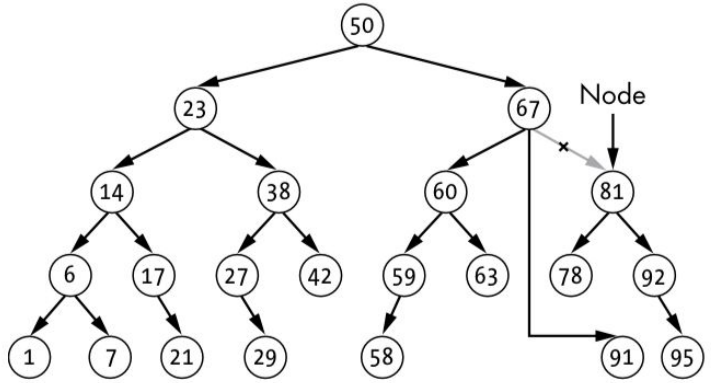
Figure 5-11: Remove an internal node with a single child by changing the pointers (left) and shifting that child up (right).



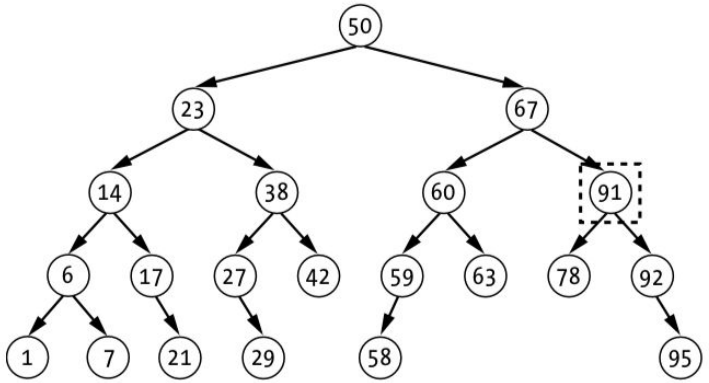
(1)



(2)



(2)



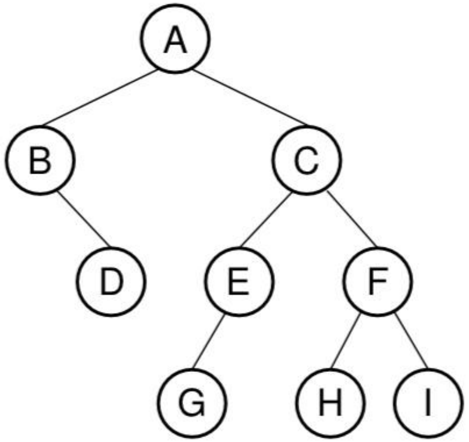
(3)

Figure 5-12: To remove an internal node with two children, we first swap the node's successor into that position.

Traversal (Print)

Remember, Traversal:

- Preorder:
  - Visit node
  - Visit left subtree
  - Visit right subtree
- Postorder:
  - Visit left subtree
  - Visit right subtree
  - Visit node
- Inorder:
  - Visit left subtree
  - Visit node
  - Visit right subtree



Printing a BST uses Inorder

Inorder traversal prints the farthest "left" node first, and the farthest "right" node last.

This will print a BST in numerical order.