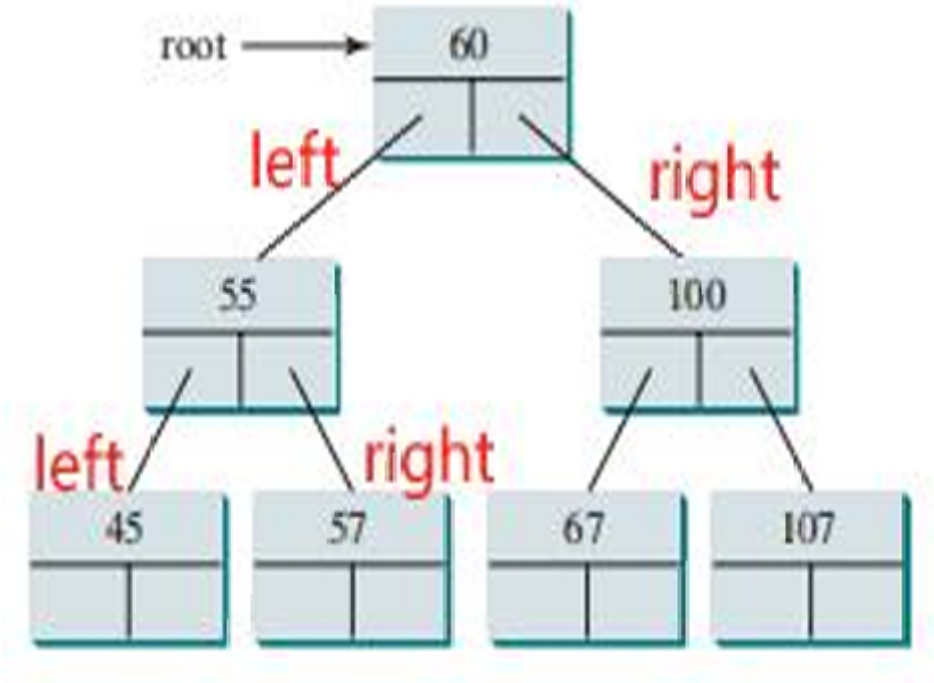
1.二叉树（Binary Trees）

二叉树是一种分层结构，它要么是空的，要么由一个元素（称为根节点）和两个不同的二叉树（称为左子树和右子树）组成。左子树和右子树本身也是二叉树，可以继续按照这种结构分解。
一个节点的左（右）子树的根节点被称为该节点的左（右）子节点。
没有孩子的节点称为叶子节点。

我们在学习数据结构的时候就知道如何用链式节点去表示二叉树，每个节点包含一个元素值，以及两个名为“left”和“right”的链接，分别指向左孩子和右孩子。

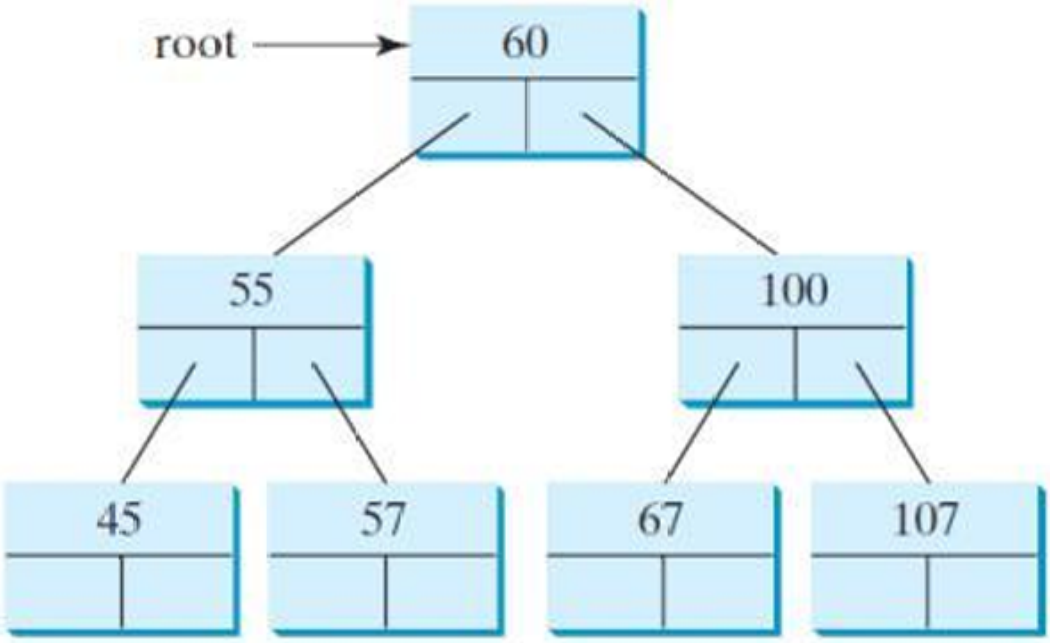
```
class TreeNode<E> {
    // The value stored in this node (of generic type E)
    E element;

    // Reference to the left and right child nodes
    TreeNode<E> left;
    TreeNode<E> right;

    // Constructor
    // Initialize the node with a given value
    public TreeNode(E o) {
        element = o;
    }
}
```



1.1 二叉搜索树（Binary Search Tree，简称BST）

二叉搜索树是一种特殊的二叉树。
这种二叉树有以下属性：

1. 默认情况下，二叉搜索树中没有重复的元素。
2. 对于树中的每个节点，其左子树中所有节点的值都小于该节点的值。对于树中的每个节点，其右子树中所有节点的值都大于该节点的值。



下面我们将对BST的各种操作进行简单讲解。

1.1.1 插入操作

插入操作的步骤如下：

1. 如果树为空（root == null），直接创建一个新节点作为根节点。
2. 如果树不为空，从根节点开始，通过比较待插入元素与当前节点的值，沿着树向下查找插入位置：如果待插入元素小于当前节点的值，向左子树移动。

如果待插入元素大于当前节点的值，向右子树移动。
如果待插入元素与当前节点的值相等，说明元素已存在，返回 false。

3. 找到合适的插入位置后，创建一个新节点，并将其链接到父节点的左孩子或右孩子上。

代码如下：

```
// 插入元素的方法
    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element); // 如果树为空，直接创建根节点
        } else {
            // Locate the parent node
            TreeNode<E> current = root, parent = null;
            while (current != null) {
                if (element.compareTo(current.element) < 0) { // 如果待插入元素小于
当前节点的值
                    parent = current;
                    current = current.left; // 向左子树移动
                } else if (element.compareTo(current.element) > 0) { // 如果待插入
元素大于当前节点的值
                    parent = current;
                    current = current.right; // 向右子树移动
                } else {
                    return false; // 如果元素已存在，返回false
                }
            }
            // 插入新节点
            if (element.compareTo(parent.element) < 0) {
                parent.left = new TreeNode<>(element); // 插入到左子树
            } else {
                parent.right = new TreeNode<>(element); // 插入到右子树
            }
        }
        return true; // 元素插入成功
    }
```

1.1.2 搜索操作

搜索的步骤更为简单，因为插入的操作其实包含了搜索的逻辑。
步骤如下，

1. 从根节点开始，根据目标元素与当前节点值的比较结果，决定向左子树还是右子树移动。
2. 如果找到目标元素，返回 true。
3. 如果遍历到叶子节点仍未找到目标元素，返回 false。
   代码如下。

```
// 查找元素的方法
    public boolean search(E element) {
        // Start from the root
        TreeNode<E> current = root;
        while (current != null) {
            if (element.compareTo(current.element) < 0) {
                current = current.left; // Go left
            } else if (element.compareTo(current.element) > 0) {
```

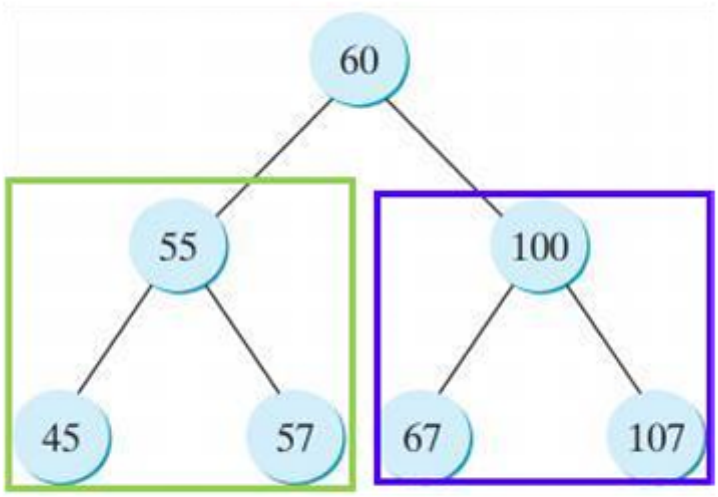
```
                current = current.right; // Go right
            } else {
                // Element matches current.element
                return true; // Element is found
            }
        }
        return false; // Element is not in the tree
    }
```

1.1.3 树的遍历（Tree Traversal）

树的遍历是访问树中每个节点恰好一次的过程。有几种不同的遍历方式：前序遍历、中序遍历、后序遍历、深度优先遍历和广度优先遍历。

1.1.3.1 前序遍历（Preorder Traversal）

首先访问当前节点，然后递归地访问当前节点的左子树，最后递归地访问当前节点的右子树。
下图展示了一个树。



对于这个树，它的前序遍历的结果是：60、55、45、57、100、67、107。

1.1.3.2 中序遍历（Inorder Traversal）

先递归地访问当前节点的左子树，然后访问当前节点本身，最后递归地访问当前节点的右子树。
对于前面的树，它的中序遍历的结果是：45、55、57、60、67、100、107。

1.1.3.3 后序遍历（Postorder Traversal）

先访问当前节点的左子树，然后访问当前节点的右子树，最后访问当前节点本身。
对于前面的树，它的后序遍历的结果是：45、57、55、67、107、100、60。

1.1.3.4 广度优先遍历（Breadth-First Traversal）

从根节点开始，逐层访问所有节点，先访问根节点，然后从左到右访问根节点的所有子节点，接着是子节点的所有子节点，依此类推。
对于前面的树，它的广度优先遍历的结果是：60、55、100、45、57、67、107。

1.1.3.5 深度优先遍历（Depth-First Traversal）

从根节点开始，沿着每个分支尽可能深地访问节点，直到到达叶子节点，然后再回溯到上一层节点，继续访问其他分支。
对于前面的树，它的深度优先遍历的结果是：60、55、45、57、100、67、107。

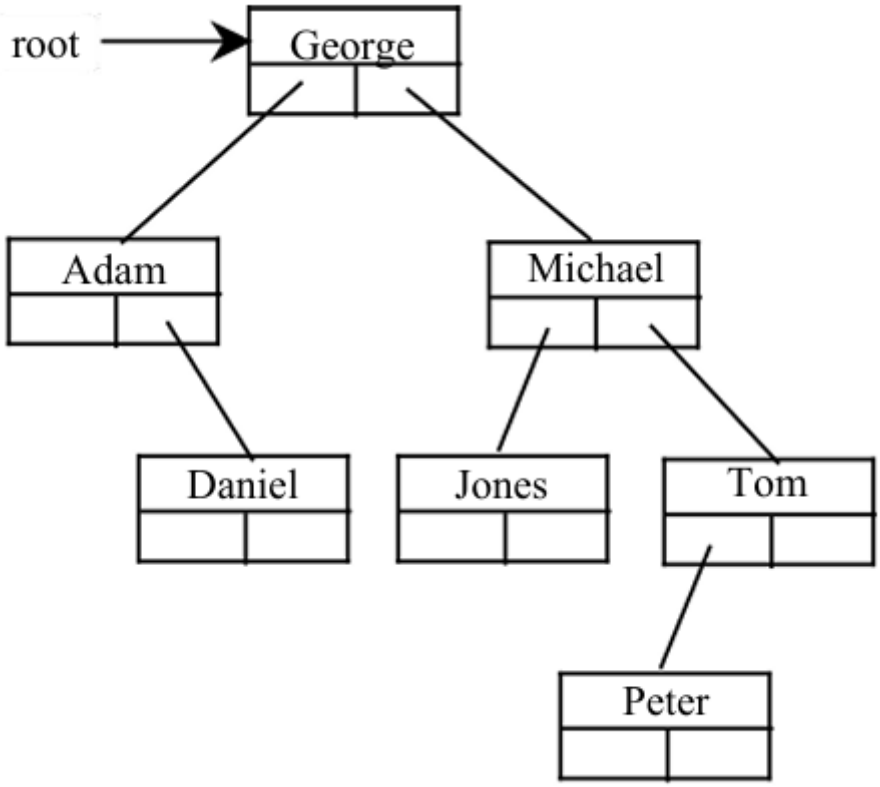
中序遍历和后序遍历的代码如下。

```
@Override
/** Inorder traversal from the root */
public void inorder() {
    inorder(root);
}

/** Inorder traversal from a subtree */
protected void inorder(TreeNode<E> root) {
    if (root == null) return;
    inorder(root.left); // First visit the left subtree
    System.out.print(root.element + " "); // Then visit the root
    inorder(root.right); // Finally visit the right subtree
}

@Override
/** Postorder traversal from the root */
public void postorder() {
    postorder(root);
}

/** Postorder traversal from a subtree */
protected void postorder(TreeNode<E> root) {
    if (root == null) return;
    postorder(root.left); // First visit the left subtree
    postorder(root.right); // Then visit the right subtree
    System.out.print(root.element + " "); // Finally visit the root
}
```

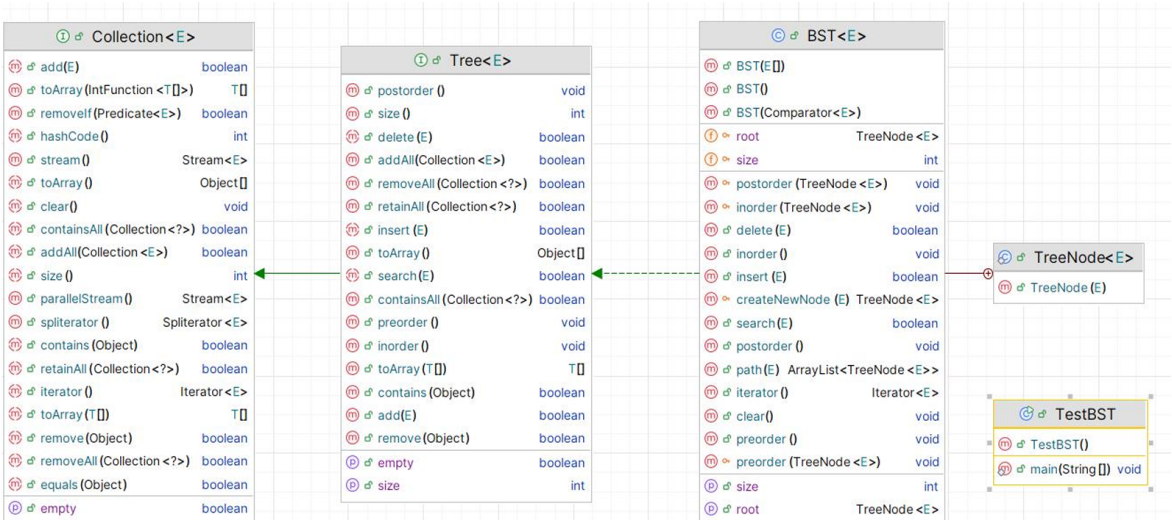
测试代码如下。

```
public class TestBST {
  public static void main(String[] args) {
    // Create a BST
    BST<String> tree = new BST<>();
    tree.insert("George");
    tree.insert("Michael");
    tree.insert("Tom");
    tree.insert("Adam");
    tree.insert("Jones");
    tree.insert("Peter");
    tree.insert("Daniel");

    // Traverse tree
    System.out.print("Inorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
```

```
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());

    // Search for an element
    System.out.print("\nIs Peter in the tree? " +
      tree.search("Peter"));

    // Get a path from the root to Peter
    System.out.print("\nA path from the root to Peter is: ");
    java.util.ArrayList<BST.TreeNode<String>> path
      = tree.path("Peter");
    for (int i = 0; path != null && i < path.size(); i++)
      System.out.print(path.get(i).element + " ");

    Integer[] numbers = {2, 4, 3, 1, 8, 5, 6, 7};
    BST<Integer> intTree = new BST<>(numbers);
    System.out.print("\nInorder (sorted): ");
    intTree.inorder();
  }
}
```



下面展示了关于二叉搜索树的UML图。



Tree 接口继承自 Collection 接口，扩展了树相关的操作方法，如后序遍历（postorder()）、获取树的大小（size()）、搜索（search(E)）等。
BST 类实现了 Tree 接口，提供了二叉搜索树的具体实现。
TreeNode 类是 BST 类的一个内部类，用于表示二叉搜索树中的节点。
TestBST 类包含了 main 方法，用于测试二叉搜索树的各种操作。

1.1.3.6 使用迭代器进行遍历

前面的方法通常只用于显示树中的元素。如果需要对二叉树中的元素进行处理（而不仅仅是显示它们），这些方法就不够用了。
为了解决这个问题我们可以使用迭代器（Iterator），因为它允许灵活和可定制的处理树元素，而不仅仅是显示它们。
Tree 接口扩展了 java.util.Collection 接口。由于 Collection 接口扩展了 java.lang.Iterable，因此 BST（二叉搜索树）也是 Iterable 的子类。
作为 Iterable 的子类，BST 可以直接定义一个迭代器类来实现 java.util.Iterator 接口。
代码如下。

```
@Override /** Obtain an iterator. Use inorder. */
  public java.util.Iterator<E> iterator() {
    return new InorderIterator();
  }

  // Inner class InorderIterator
  private class InorderIterator implements java.util.Iterator<E> {
    // Store the elements in a list
    private java.util.ArrayList<E> list =
      new java.util.ArrayList<>();
    private int current = 0; // Point to the current element in list

    public InorderIterator() {
      inorder(); // Traverse binary tree and store elements in list
    }

    /** Inorder traversal from the root*/
    private void inorder() {
      inorder(root);
    }

    /** Inorder traversal from a subtree */
    private void inorder(TreeNode<E> root) {
      if (root == null) return;
      inorder(root.left);
      list.add(root.element);
      inorder(root.right);
    }

    @Override /** More elements for traversing? */
    public boolean hasNext() {
      if (current < list.size())
        return true;

      return false;
    }

    @Override /** Get the current element and move to the next */
    public E next() {
      return list.get(current++);
    }

    @Override // Remove the element returned by the last next()
    public void remove() {
      if (current == 0) // next() has not been called yet
        throw new IllegalStateException();

      delete(list.get(--current));
      list.clear(); // Clear the list
      inorder(); // Rebuild the list
    }
  }

  @Override /** Remove all elements from the tree */
  public void clear() {
```

```
    root = null;
    size = 0;
  }
```

测试代码如下。

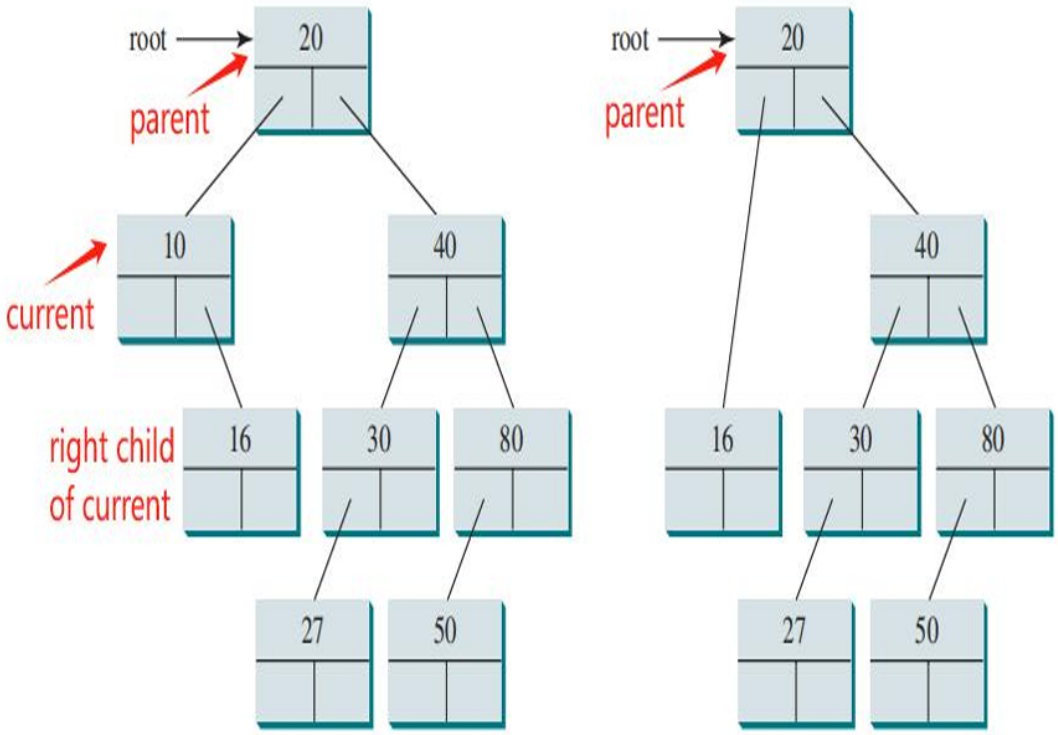
```
public class TestBSTWithIterator {
  public static void main(String[] args) {
    BST<String> tree = new BST<>();
    tree.insert("George");
    tree.insert("Michael");
    tree.insert("Tom");
    tree.insert("Adam");
    tree.insert("Jones");
    tree.insert("Peter");
    tree.insert("Daniel");

    for (String s: tree)
      System.out.print(s.toUpperCase() + " ");
  }
}
```

1.1.4 删除操作

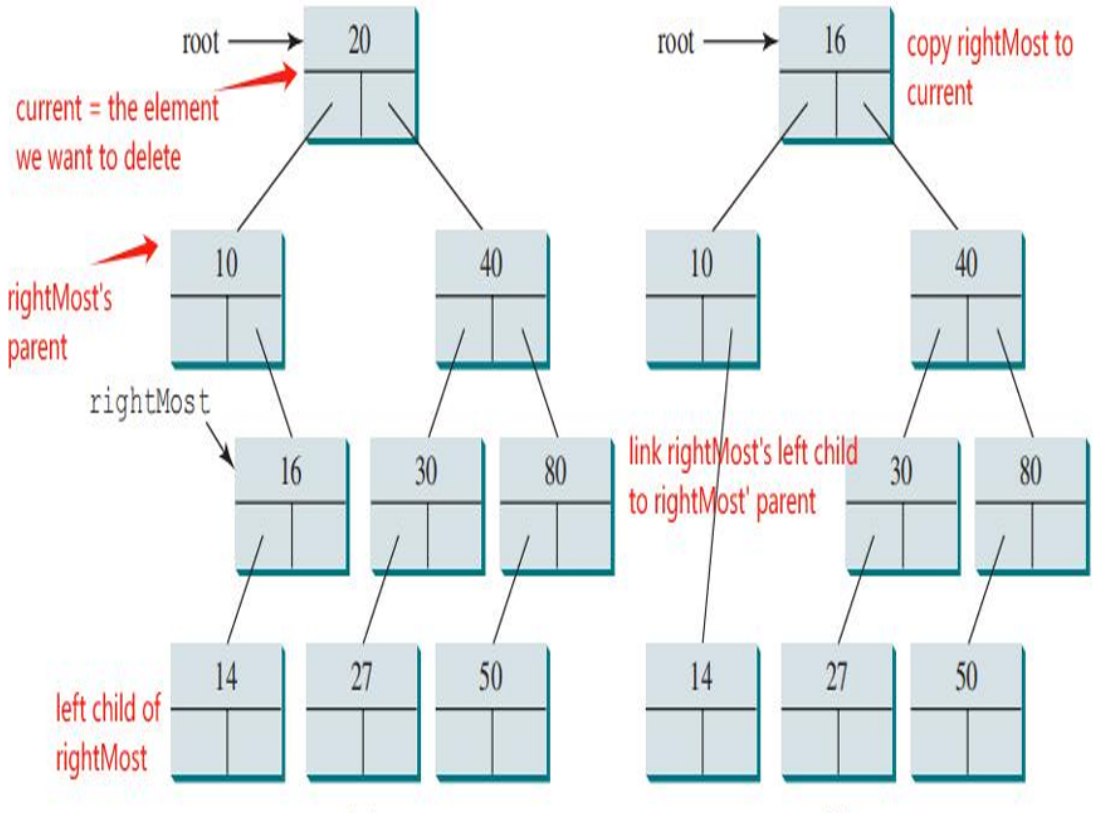
删除操作也是与前面的插入操作类似，需要先找到要删除的节点.
步骤如下。

1. 从根节点开始，沿着树向下遍历，根据BST的性质来找到目标节点。
   此时 current 指向二叉树中包含要删除元素的节点，parent 指向 current 节点的父节点。 current 节点可能是 parent 节点的左孩子或右孩子。
2. 情况1：如果 current 节点没有左孩子，那么删除操作相对简单。此时，有两种子情况：
   如果 current 节点也没有右孩子（即 current 是一个叶子节点），那么可以直接从树中移除该节点，将其父节点的相应指针（左或右）设置为 null。
   如果 current 节点有一个右孩子，那么可以用其右孩子替换 current 节点，然后更新 parent 节点

的指针。



3. 情况2：如果 current 节点有左孩子，删除操作会更复杂。此时，通常的做法是：
   找到 current 节点的中序后继（即 current 节点右子树中值最小的节点，也就是右子树的最左节点）。
   用中序后继的值替换 current 节点的值，然后删除中序后继节点。由于中序后继节点没有左孩子，这将简化为情况 1 的操作。



代码如下。

```
public boolean delete(E e) {
    // Locate the node to be deleted and also locate its parent node
```

```
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null) {
      if (c.compare(e, current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (c.compare(e, current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        break; // Element is in the tree pointed at by current
    }

    if (current == null)
      return false; // Element is not in the tree

    // Case 1: current has no left child
    if (current.left == null) {
      // Connect the parent with the right child of the current node
      if (parent == null) {
        root = current.right;
      }
      else {
        if (c.compare(e, parent.element) < 0)
          parent.left = current.right;
        else
          parent.right = current.right;
      }
    }
    else {
      // Case 2: The current node has a left child
      // Locate the rightmost node in the left subtree of
      // the current node and also its parent
      TreeNode<E> parentOfRightMost = current;
      TreeNode<E> rightMost = current.left;

      while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right; // Keep going to the right
      }

      // Replace the element in current by the element in rightMost
      current.element = rightMost.element;

      // Eliminate rightmost node
      if (parentOfRightMost.right == rightMost)
        parentOfRightMost.right = rightMost.left;
      else
        // Special case: parentOfRightMost == current
        parentOfRightMost.left = rightMost.left;
    }

    size--; // Reduce the size of the tree
```

```
    return true; // Element deleted successfully
  }
```

测试代码如下。

```
package chapter25;

public class TestBSTDelete {
  public static void main(String[] args) {
    BST<String> tree = new BST<>();
    tree.insert("George");
    tree.insert("Michael");
    tree.insert("Tom");
    tree.insert("Adam");
    tree.insert("Jones");
    tree.insert("Peter");
    tree.insert("Daniel");
    printTree(tree);

    System.out.println("\nAfter delete George:");
    tree.delete("George");
    printTree(tree);

    System.out.println("\nAfter delete Adam:");
    tree.delete("Adam");
    printTree(tree);

    System.out.println("\nAfter delete Michael:");
    tree.delete("Michael");
    printTree(tree);
  }

  public static void printTree(BST tree) {
    // Traverse tree
    System.out.print("Inorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

1.1.5 二叉树的时间复杂度

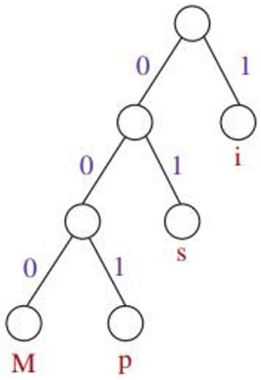
1. 遍历操作的时间复杂度
   中序遍历（Inorder Traversal）、前序遍历（Preorder Traversal） 和 后序遍历（Postorder Traversal）：时间复杂度为 O(n)，其中 n 是树中节点的数量。
   这是因为在这些遍历中，每个节点恰好被访问一次。无论树的形状如何（完全平衡或完全不平衡），只要节点总数为 n，就需要进行 n 次访问。

2. 搜索（Search）、插入（Insertion）和删除（Deletion）：
   时间复杂度是树的高度。
   树的高度是从根节点到最远叶子节点的最长路径上的节点数。
   最坏情况下，树的高度是$O(n)$。
   这种情况发生在树完全不平衡时，即树退化成链表形式，其中每个节点只有一个子节点（要么是左孩子，要么是右孩子）。
   在这种情况下，搜索、插入和删除操作需要访问从根节点到目标节点的路径上的所有节点，导致时间复杂度为$O(n)$。
   平衡二叉搜索树（如 AVL 树或红黑树）：通过保持树的平衡，可以确保树的高度尽可能小。
   在平衡二叉搜索树中，树的高度通常是$O(logn)$，这显著提高了搜索、插入和删除操作的效率。

1.2 霍夫曼编码

在ASCII（美国信息交换标准代码）中，每个字符都用8位（1字节）来编码。例如，字符"M"被编码为二进制的01001101。
霍夫曼编码通过为更频繁出现的字符使用更少的位来压缩数据。
例如，单词"Mississippi"在ASCII编码中需要11字节（88位），但在霍夫曼编码中，如果根据字符出现的频率构建编码，可能只需要22位（约3字节）。

霍夫曼编码的编码是基于文本中字符的出现频率，使用一种特殊的二叉树——霍夫曼编码树来构建的。
在霍夫曼树中，每个内部节点（非叶子节点）都有两个子节点，分别代表向左和向右的边。
这些边被赋予二进制值，通常是0和1。左边缘通常被赋予值0，右边缘被赋予值1，但这个规则可以根据具体实现而变化。
在霍夫曼树中，每个字符都对应于树的一个叶子节点（树的最末端节点，没有子节点）。
叶子节点包含了要编码的字符和该字符在原始数据中的频率。
每个字符的编码是由从树的根节点到该字符对应叶子节点的路径上的边值（0或1）组成的。
路径上的每个边值按顺序连接起来，形成该字符的霍夫曼编码。
下图展示了一个例子。



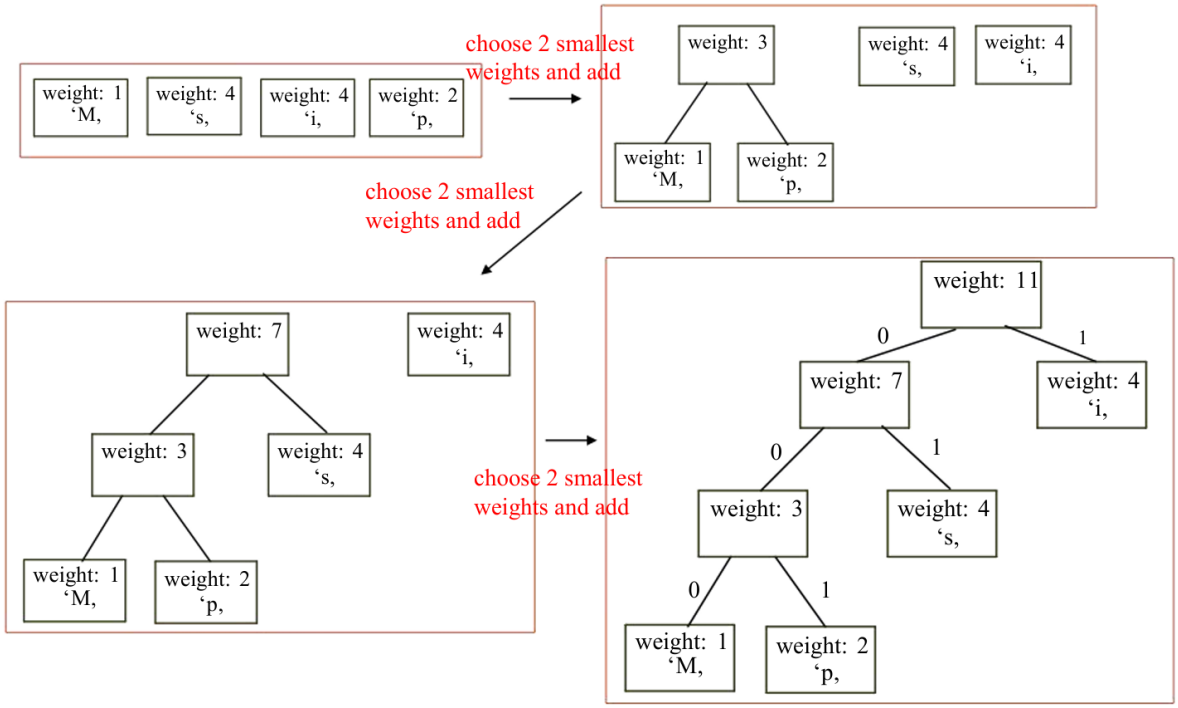
Character	Code	Frequency
M	000	1
p	001	2
s	01	4
i	1	4

Mississippi ========>00010101101011001 0011========>Mississippi
is encoded to is decoded to

构件霍夫曼树的过程使用贪心策略，详细步骤如下：

1. 从一片森林（forest）开始，每棵树（tree）只包含一个节点（node），该节点代表一个字符。
   节点的权重（weight）是字符在文本中的出现频率。
2. 重复以下步骤，直到只剩下一棵树（即霍夫曼树）：
   从森林中选择两棵权重最小的树。这通常通过优先队列（priority queue）实现，优先队列可以用堆（heap）数据结构来实现，以便于快速找到最小元素。
   创建一个新节点作为这两棵树的父节点。
   新树的权重是这两棵子树权重的和。

下图展示了这个过程。



相关的代码如下。

```
import java.util.Scanner;

public class HuffmanCode {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a text: ");
    String text = input.nextLine();

    int[] counts = getCharacterFrequency(text); // Count frequency

    System.out.printf("%-15s%-15s%-15s%-15s\n",
      "ASCII Code", "Character", "Frequency", "Code");

    Tree tree = getHuffmanTree(counts); // Create a Huffman tree
    String[] codes = getCode(tree.root); // Get codes

    for (int i = 0; i < codes.length; i++)
      if (counts[i] != 0) // (char)i is not in text if counts[i] is 0
        System.out.printf("%-15d%-15s%-15d%-15s\n",
          i, (char)i + "", counts[i], codes[i]);
  }

  /** Get Huffman codes for the characters
   * This method is called once after a Huffman tree is built
   */
  public static String[] getCode(Tree.Node root) {
    if (root == null) return null;
    String[] codes = new String[2 * 128];
    assignCode(root, codes);
    return codes;
  }
```

```
  /* Recursively get codes to the leaf node */
  private static void assignCode(Tree.Node root, String[] codes) {
    if (root.left != null) {
      root.left.code = root.code + "0";
      assignCode(root.left, codes);

      root.right.code = root.code + "1";
      assignCode(root.right, codes);
    }
    else {
      codes[(int)root.element] = root.code;
    }
  }

  /** Get a Huffman tree from the codes */
  public static Tree getHuffmanTree(int[] counts) {
    // Create a heap to hold trees
    Heap<Tree> heap = new Heap<>(); // Defined in Listing 24.10
    for (int i = 0; i < counts.length; i++) {
      if (counts[i] > 0)
        heap.add(new Tree(counts[i], (char)i)); // A leaf node tree
    }

    while (heap.getSize() > 1) {
      Tree t1 = heap.remove(); // Remove the smallest weight tree
      Tree t2 = heap.remove(); // Remove the next smallest weight
      heap.add(new Tree(t1, t2)); // Combine two trees
    }

    return heap.remove(); // The final tree
  }

  /** Get the frequency of the characters */
  public static int[] getCharacterFrequency(String text) {
    int[] counts = new int[256]; // 256 ASCII characters

    for (int i = 0; i < text.length(); i++)
      counts[(int)text.charAt(i)]++; // Count the character in text

    return counts;
  }

  /** Define a Huffman coding tree */
  public static class Tree implements Comparable<Tree> {
    Node root; // The root of the tree

    /** Create a tree with two subtrees */
    public Tree(Tree t1, Tree t2) {
      root = new Node();
      root.left = t1.root;
      root.right = t2.root;
      root.weight = t1.root.weight + t2.root.weight;
    }

    /** Create a tree containing a leaf node */
```

```
    public Tree(int weight, char element) {
      root = new Node(weight, element);
    }

    @Override /** Compare trees based on their weights */
    public int compareTo(Tree t) {
      if (root.weight < t.root.weight) // Purposely reverse the order
        return 1;
      else if (root.weight == t.root.weight)
        return 0;
      else
        return -1;
    }

    public class Node {
      char element; // Stores the character for a leaf node
      int weight; // weight of the subtree rooted at this node
      Node left; // Reference to the left subtree
      Node right; // Reference to the right subtree
      String code = ""; // The code of this node from the root

      /** Create an empty node */
      public Node() {
      }

      /** Create a node with the specified weight and character */
      public Node(int weight, char element) {
        this.weight = weight;
        this.element = element;
      }
    }
  }
}
```

2. 练习

2.1 非递归的中序遍历方法

实现一个非递归的中序遍历方法，并且编写一个测试程序来演示这个方法。
示例代码如下。

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;

class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
    }
```

```
}

class BST<E extends Comparable<E>> {
    private TreeNode<E> root;
    private int size;

    public BST() {
        root = null;
        size = 0;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
            size++;
            return true;
        } else {
            TreeNode<E> current = root;
            TreeNode<E> parent = null;
            while (current != null) {
                parent = current;
                if (element.compareTo(current.element) < 0) {
                    current = current.left;
                } else if (element.compareTo(current.element) > 0) {
                    current = current.right;
                } else {
                    return false; // Duplicate element
                }
            }
            if (element.compareTo(parent.element) < 0) {
                parent.left = new TreeNode<>(element);
            } else {
                parent.right = new TreeNode<>(element);
            }
            size++;
            return true;
        }
    }

    public void nonRecursiveInorder() {
        Stack<TreeNode<E>> stack = new Stack<>();
        TreeNode<E> current = root;
        while (current != null || !stack.isEmpty()) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }
            current = stack.pop();
            System.out.print(current.element + " ");
            current = current.right;
        }
    }
}

public class TestBST {
```

```
    public static void main(String[] args) {
        BST<Integer> bst = new BST<>();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter 10 integers:");
        for (int i = 0; i < 10; i++) {
            System.out.print("Enter integer " + (i + 1) + ": ");
            int number = scanner.nextInt();
            bst.insert(number);
        }

        System.out.println("\nInorder traversal (non-recursive):");
        bst.nonRecursiveInorder();

        scanner.close();
    }
}
```

2.2 非递归的前序遍历方法

实现一个非递归的前序遍历方法，并且编写一个测试程序来演示这个方法。
示例代码如下。

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;

class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
    }
}

class BST<E extends Comparable<E>> {
    private TreeNode<E> root;

    public BST() {
        root = null;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
            return true;
        } else {
            TreeNode<E> current = root;
            while (true) {
                if (element.compareTo(current.element) < 0) {
                    if (current.left == null) {
                        current.left = new TreeNode<>(element);
```

```
                    return true;
                }
                current = current.left;
            } else if (element.compareTo(current.element) > 0) {
                if (current.right == null) {
                    current.right = new TreeNode<>(element);
                    return true;
                }
                current = current.right;
            } else {
                return false; // Duplicate element
            }
        }
    }

    public void nonRecursivePreorder() {
        Stack<TreeNode<E>> stack = new Stack<>();
        TreeNode<E> current = root;

        while (current != null || !stack.isEmpty()) {
            while (current != null) {
                System.out.print(current.element + " ");
                stack.push(current);
                current = current.left;
            }
            current = stack.pop();
            current = current.right;
        }
    }
}

public class TestBST {
    public static void main(String[] args) {
        BST<Integer> bst = new BST<>();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter 10 integers:");
        for (int i = 0; i < 10; i++) {
            System.out.print("Enter integer " + (i + 1) + ": ");
            int number = scanner.nextInt();
            bst.insert(number);
        }

        System.out.println("\nPreorder traversal (non-recursive):");
        bst.nonRecursivePreorder();

        scanner.close();
    }
}
```

2.3 返回二叉树中叶子节点的数量

定义一个新的类 BSTWithNumberOfLeaves，它继承自 BST 类，并添加一个新的方法 getNumberOfLeaves() 来返回二叉树中叶子节点的数量。
示例代码如下。

```
class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
        this.left = null;
        this.right = null;
    }
}

class BST<E extends Comparable<E>> {
    protected TreeNode<E> root;

    public BST() {
        root = null;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
            return true;
        } else {
            TreeNode<E> current = root;
            TreeNode<E> parent = null;
            while (current != null) {
                parent = current;
                if (element.compareTo(current.element) < 0) {
                    current = current.left;
                } else if (element.compareTo(current.element) > 0) {
                    current = current.right;
                } else {
                    return false; // Duplicate element
                }
            }
            if (element.compareTo(parent.element) < 0) {
                parent.left = new TreeNode<>(element);
            } else {
                parent.right = new TreeNode<>(element);
            }
            return true;
        }
    }

    // 其他必要的BST方法...
}

class BSTWithNumberOfLeaves<E extends Comparable<E>> extends BST<E> {
    public int getNumberOfLeaves() {
        return countLeaves(root);
```

```
    }

    private int countLeaves(TreeNode<E> node) {
        if (node == null) {
            return 0;
        }
        if (node.left == null && node.right == null) {
            return 1;
        }
        return countLeaves(node.left) + countLeaves(node.right);
    }
}

public class TestBST {
    public static void main(String[] args) {
        BSTWithNumberOfLeaves<Integer> bst = new BSTWithNumberOfLeaves<>();
        // 假设这里插入了一些元素
        // ...

        System.out.println("Number of leaves: " + bst.getNumberOfLeaves());
    }
}
```

2.4 前序遍历的迭代器

添加一个名为 preorderIterator 的方法，该方法返回一个迭代器，用于以前序遍历的方式遍历二叉搜索树（BST）中的元素。
示例代码如下。

```
import java.util.Iterator;
import java.util.Stack;

class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
    }
}

class BST<E extends Comparable<E>> {
    private TreeNode<E> root;

    public BST() {
        root = null;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
            return true;
        } else {
```

```
            TreeNode<E> current = root;
            while (true) {
                if (element.compareTo(current.element) < 0) {
                    if (current.left == null) {
                        current.left = new TreeNode<>(element);
                        return true;
                    }
                    current = current.left;
                } else if (element.compareTo(current.element) > 0) {
                    if (current.right == null) {
                        current.right = new TreeNode<>(element);
                        return true;
                    }
                    current = current.right;
                } else {
                    return false; // Duplicate element
                }
            }
        }
    }

    // 非递归前序遍历的辅助方法
    private void preorderHelper(TreeNode<E> node, Stack<TreeNode<E>> stack) {
        if (node != null) {
            stack.push(node);
        }
    }

    // 返回前序遍历的迭代器
    public java.util.Iterator<E> preorderIterator() {
        Stack<TreeNode<E>> stack = new Stack<>();
        preorderHelper(root, stack);
        return new PreorderIterator(stack);
    }

    // 实现前序遍历的迭代器
    private class PreorderIterator implements Iterator<E> {
        private Stack<TreeNode<E>> stack;

        public PreorderIterator(Stack<TreeNode<E>> stack) {
            this.stack = stack;
        }

        @Override
        public boolean hasNext() {
            return !stack.isEmpty();
        }

        @Override
        public E next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException();
            }
            TreeNode<E> node = stack.pop();
            preorderHelper(node.right, stack); // 先右后左，保证前序
```

```
            preorderHelper(node.left, stack);
            return node.element;
        }
    }
}
```