实验10

练习1

题目

在忽略字符串中的非英语字母字符并把大小写英语字母字符看成等价后，正读和反读完全一样的字符串称为英语回文。如： Rise to vote, sir. No lemons no melon. Was it a car or a cat I saw?
判断输入的字符串是否为英语回文。要求使用栈和队列。

解析

仅仅是栈和队列的简单应用，不做过多解释。

代码

Exercise_1.java

```java
import java.util.Scanner;

public class Exercise_1 {
    public static void main(String[] args) throws Exception {
        // read data
        Scanner in = new Scanner(System.in);
        String s = in.nextLine();
        in.close();
        // initialize
        Queue<Character> queue = new Queue<Character>();
        Stack<Character> stack = new Stack<Character>();
        // add elements of string to queue / stack
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isLetter(c)) {
                queue.enqueue(Character.toLowerCase(c));
                stack.push(Character.toLowerCase(c));
            }
        }
        // take out elements of queue/stack one by one, and compare their
differences
        boolean result = true;
        while (stack.peek() != null) {
            if (queue.dequeue() != stack.pop()) {
                result = false;
                break;
            }
        }
        // print result
        System.out.println(result);
    }
}
```

输入

```
Rise to vote, sir.
```

输出

```
true
```

练习2

题目

要求用二叉链表实现二叉树。根据与一棵二叉树对应的扩展二叉树的层序序列构造该二叉树。输出该二叉树的先序、中序、后序和层序序列。交换该二叉树的所有结点的左、右子树。输出新二叉树的先序、中序、后序和层序序列。

解析

先序、中序、后序遍历原理很简单，不做解释。

层序遍历时创建一个队列q。初始时根节点入队，然后不断出队并使出队结点的左右孩子重新入队（如果孩子不为null），直到队列为空，结点的出队顺序即为层序遍历的结果。

交换左右子树和层序遍历的原理类似，只是将记录结点的出队顺序改成交换左右孩子。

代码

BiTree.java

```java
// 使用Java泛型实现的二叉树，利用扩展二叉树的层序序列构建。
public class BiTree<Item> {
    private Node root;

    // Node类型
    private class Node {
        Item data;
        Node left;
        Node right;

        public Node(Item data, Node left, Node right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    // 初始化
    public BiTree() {
        this.root = null;
    }

    public BiTree(Item data) {
        this.root = new Node(data, null, null);
    }
```

```
// 使用扩展二叉树构造二叉树
public BiTree(Item[] arr) {
    if (arr.length == 0) {
        this.root = null;
    } else {
        this.root = new Node(arr[0], null, null);
        Queue<Node> queue = new Queue<Node>();
        queue.enqueue(this.root);
        int i = 1;
        while (i < arr.length) {
            Node p = queue.dequeue();
            if (arr[i] != null) {
                p.left = new Node(arr[i], null, null);
                queue.enqueue(p.left);
            }
            if (arr[i+1] != null) {
                p.right = new Node(arr[i+1], null, null);
                queue.enqueue(p.right);
            }
            i += 2;
        }
    }
}

// 交换左右子树
public void exchange() {
    Queue<Node> queue = new Queue<Node>();
    queue.enqueue(this.root);
    while (!queue.isEmpty()) {
        Node p = queue.dequeue();
        Node tmp = p.left;
        p.left = p.right;
        p.right = tmp;
        if (p.left != null) {
            queue.enqueue(p.left);
        }
        if (p.right != null) {
            queue.enqueue(p.right);
        }
    }
}

// 先序遍历
public Item[] preOrder() throws Exception {
    return preOrder(this.root);
}

public Item[] preOrder(Node T) throws Exception {
    List<Item> result = new List<Item>();
    if (T != null) {
        result.append(T.data);
        result.extend(preOrder(T.left));
        result.extend(preOrder(T.right));
    }
    return result.toArray();
}
```

```
    // 中序遍历
    public Item[] inOrder() throws Exception {
        return inOrder(this.root);
    }

    public Item[] inOrder(Node T) throws Exception {
        List<Item> result = new List<Item>();
        if (T != null) {
            result.extend(inOrder(T.left));
            result.append(T.data);
            result.extend(inOrder(T.right));
        }
        return result.toArray();
    }

    // 后序遍历
    public Item[] postOrder() throws Exception {
        return postOrder(this.root);
    }

    public Item[] postOrder(Node T) throws Exception {
        List<Item> result = new List<Item>();
        if (T != null) {
            result.extend(postOrder(T.left));
            result.extend(postOrder(T.right));
            result.append(T.data);
        }
        return result.toArray();
    }

    // 层序遍历
    public Item[] levelOrder() {
        Queue<Node> queue = new Queue<Node>();
        List<Item> result = new List<Item>();
        queue.enqueue(this.root);
        while (!queue.isEmpty()) {
            Node p = queue.dequeue();
            result.append(p.data);
            if (p.left != null) {
                queue.enqueue(p.left);
            }
            if (p.right != null) {
                queue.enqueue(p.right);
            }
        }
        return result.toArray();
    }
}
```

Exercise_2.java

```
import java.util.Arrays;

public class Exercise_2 {
    public static void main(String[] args) throws Exception {
        // read data and create BiTree
        String[] in = { ... }; // in的具体值将在“输入”中给出
        BiTree<String> a = new BiTree<String>(in);
        // print old tree preOrder, inOrder, postOrder
        System.out.println("Old Tree");
        System.out.println("preOrder:   " + Arrays.toString(a.preOrder()));
        System.out.println("inOrder:    " + Arrays.toString(a.inOrder()));
        System.out.println("postOrder:  " + Arrays.toString(a.postOrder()));
        System.out.println("levelOrder: " + Arrays.toString(a.levelOrder()));
        System.out.println();
        // swap left and right
        a.exchange();
        // print new tree preOrder, inOrder, postOrder
        System.out.println("New Tree");
        System.out.println("preOrder:   " + Arrays.toString(a.preOrder()));
        System.out.println("inOrder:    " + Arrays.toString(a.inOrder()));
        System.out.println("postOrder:  " + Arrays.toString(a.postOrder()));
        System.out.println("levelOrder: " + Arrays.toString(a.levelOrder()));
    }
}
```

输入

```
{ "A",
  "B",  "C",
  "D", null,  "E",  "F",
 null,  "G", null, null,  "H", null,
 null, null, null, null}
```

输出

```
Old Tree
preOrder:   [A, B, D, G, C, E, F, H]
inOrder:    [D, G, B, A, E, C, H, F]
postOrder:  [G, D, B, E, H, F, C, A]
levelOrder: [A, B, C, D, E, F, G, H]

New Tree
preOrder:   [A, C, F, H, E, B, D, G]
inOrder:    [F, H, C, E, A, B, G, D]
postOrder:  [H, F, E, C, G, D, B, A]
levelOrder: [A, C, B, F, E, D, H, G]
```

附

代码中所使用的Queue和List已经在实验6和实验9中出现过。但为了严谨起见，在此附上Queue.java和List.java的代码。

Queue.java

```
import java.util.Iterator;

// 使用Java泛型实现的链式队列，支持迭代和toString
class Queue<Item> implements Iterable<Item> {
    public Node front;
    public Node rear;
    private int N;

    private class Node {
        Item data;
        Node next;

        public Node(Item data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    public boolean isEmpty() {
        return this.front == null;
    }

    public int length() {
        return this.N;
    }

    public Item getHead() {
        if (this.front != null) {
            return this.front.data;
        } else {
            return null;
        }
    }

    public Item getTail() {
        if (this.rear != null) {
            return this.rear.data;
        } else {
            return null;
        }
    }

    public void enqueue(Item data) {
        if (this.front == null) {
            this.front = new Node(data, null);
            this.rear = this.front;
        } else {
            this.rear.next = new Node(data, null);
            this.rear = this.rear.next;
        }
        this.N++;
    }

    public Item dequeue() {
        if (this.front == null) {
            return null;
        } else {
            Item data = this.front.data;
            this.front = this.front.next;
            if (this.front == null) {
                this.rear = null;
            }
            this.N--;
            return data;
        }
    }

    public void reverse() {
        if (this.front == null) {
            return;
        }
        Node newFront = new Node(this.front.data, null);
        Node p = this.front.next;
        while (p != null) {
            newFront = new Node(p.data, newFront);
            p = p.next;
        }
        this.front = newFront;
    }

    public Queue<Item> reversed() {
        if (this.front == null) {
            return new Queue<Item>();
        }
        Queue<Item> result = new Queue<Item>();
        result.front = new Node(this.front.data, null);
        Node p = this.front.next;
        while (p != null) {
            result.front = new Node(p.data, result.front);
            p = p.next;
        }
        return result;
    }

    public Item[] toArray() {
        Item[] result = (Item[]) new Object[this.N];
        Node p = this.front;
        for (int i = 0; i < this.N; i++) {
            result[i] = p.data;
            p = p.next;
        }
        return result;
    }

    public int[] toArrayInt() {
        int[] result = new int[this.N];
        Node p = this.front;
        for (int i = 0; i < this.N; i++) {
            result[i] = (int) p.data;
            p = p.next;
        }
        return result;
    }

    public String toString() {
        Node p = this.front;
        StringBuilder builder = new StringBuilder();
        builder.append("[");
        while (p != null) {
            builder.append(p.data + ", ");
            p = p.next;
        }
        builder.delete(builder.length()-2, builder.length());
        builder.append("]");
        return builder.toString();
    }

    public Iterator<Item> iterator() {
        return new QueueIterator();
    }

    private class QueueIterator implements Iterator<Item> {
        private Node current = front;

        public boolean hasNext() {
            return current != null;
        }

        public void remove() {}

        public Item next() {
            Item data = current.data;
            current = current.next;
            return data;
        }
    }
}
```

List.java

```
import java.util.Iterator;

// 使用Java泛型实现的链表，支持迭代、输出数组、排序、逆序等方法
public class List<Item> implements Iterable<Item> {
    private Node head;
    private int curLen;

    // Node类型
    private class Node {
        Item data;
        Node next;

        public Node(Item data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    // 构造函数
    public List() {
        this.head = new Node(null, null);
        this.curLen = 0;
    }

    public List(Item[] arr) {
        this();
        Node p = this.head;
        for (Item i : arr) {
            p.next = new Node(i, null);
            p = p.next;
        }
    }

    public List(List<Item> lst) {
        this();
        Node p = this.head;
        for (Item i : lst) {
            p.next = new Node(i, null);
            p = p.next;
        }
    }

    // 获取头节点
    public Node head() {
        return this.head;
    }

    // 是否为空
    public boolean isEmpty() {
        return this.curLen == 0;
    }

    // 获取长度
    public int length() {
        return this.curLen;
    }

    // 获取对应下标的元素
    public Item get(int index) throws Exception {
        if (index < 0 || index >= this.curLen) {
            throw new Exception("Invalid index " + index);
        }
        Node p = this.head.next;
        for (int i = 0; i < index; i++) {
            p = p.next;
        }
        return p.data;
    }

    // 增加元素
    public void append(Item x) {
        Node p = this.head;
        for (int i = 0; i < this.curLen; i++) {
            p = p.next;
        }
        p.next = new Node(x, null);
        this.curLen++;
    }

    // 扩展列表
    public void extend(Item[] arr) {
        Node p = this.head;
        for (int i = 0; i < this.curLen; i++) {
            p = p.next;
        }
        for (int i = 0; i < arr.length; i++) {
            p.next = new Node(arr[i], null);
            p = p.next;
        }
        this.curLen += arr.length;
    }

    public void extend(List<Item> lst) {
        Node p = this.head;
        for (int i = 0; i < this.curLen; i++) {
            p = p.next;
        }
        for (Item i : lst) {
            p.next = new Node(i, null);
            p = p.next;
        }
        this.curLen += lst.length();
    }

    // 插入元素
    public void insert(int index, Item x) throws Exception {
        if (index < 0 || index > this.curLen) {
            throw new Exception("Invalid index " + index);
        }
        Node p = this.head;
        for (int i = 0; i < index; i++) {
            p = p.next;
        }
        p.next = new Node(x, p.next);
        this.curLen++;
    }

    // 删除指定元素
    public void remove(Item x) throws Exception {
        Node p = this.head;
        while (p.next != null && !p.next.data.equals(x)) {
            p = p.next;
        }
        if (p.next == null) {
            throw new Exception(x + "is not in list");
        } else {
            p.next = p.next.next;
            this.curLen--;
        }
    }

    // 按下标删除元素
    public void delete(int index) throws Exception {
        if (index < 0 || index >= this.curLen) {
            throw new Exception("Invalid index " + index);
        }
        Node p = this.head;
        for (int i = 0; i < index; i++) {
            p = p.next;
        }
        p.next = p.next.next;
        this.curLen--;
    }
```

```java
    // 查找元素对应下标
    public int index(Item x) throws Exception {
        Node p = this.head.next;
        int i;
        for (i = 0; p != null && !p.data.equals(x); i++) {
            p = p.next;
        }
        if (p != null) {
            return i;
        } else {
            throw new Exception(i + "is not in list");
        }
    }

    // 返回复制后的新列表
    public List<Item> copy() {
        List<Item> result = new List<Item>();
        Node p = this.head.next;
        Node q = result.head;
        while (p != null) {
            q.next = new Node(p.data, null);
            q = q.next;
            result.curLen++;
            p = p.next;
        }
        return result;
    }

    // 统计列表中指定元素的个数
    public int count(Item x) {
        int result = 0;
        Node p = this.head.next;
        while (p != null) {
            if (p.data.equals(x)) {
                result++;
            }
            p = p.next;
        }
        return result;
    }

    // 就地逆序
    public void reverse() {
        Node newHead = new Node(null, null);
        Node p = this.head.next;
        while (p != null) {
            newHead.next = new Node(p.data, newHead.next);
            p = p.next;
        }
        this.head = newHead;
    }

    // 返回逆序链表
    public List<Item> reversed() {
        List<Item> result = new List<Item>();
        result.head = new Node(null, null);
        Node p = this.head.next;
        while (p != null) {
            result.head.next = new Node(p.data, result.head.next);
            result.curLen++;
            p = p.next;
        }
        return result;
    }

    // 返回数组形式的列表
    public Item[] toArray() {
        Item[] result = (Item[]) new Object[this.curLen];
        // 由于类型擦除的问题，Java不支持泛型数组的创建，只能强制转换
        Node p = this.head.next;
        for (int i = 0; i < this.curLen; i++) {
            result[i] = p.data;
            p = p.next;
        }
        return result;
    }

    // 返回数组形式的列表并将全部元素强制转换为int
    public int[] toArrayInt() {
        int[] result = new int[this.curLen];
        Node p = this.head.next;
        for (int i = 0; i < this.curLen; i++) {
            result[i] = (int) p.data;
            p = p.next;
        }
        return result;
    }

    // 重写toString方法
    public String toString() {
        Node p = this.head.next;
        StringBuilder builder = new StringBuilder();
        builder.append("[");
        while (p != null) {
            builder.append(p.data + ", ");
            p = p.next;
        }
        if (this.curLen > 0) {
            builder.delete(builder.length() - 2, builder.length());
        }
        builder.append("]");
        return builder.toString();
    }

    // 返回迭代器，以支持迭代
    public Iterator<Item> iterator() {
        return new LinkListIterator();
    }

    // 迭代器
    private class LinkListIterator implements Iterator<Item> {
        private Node current = head.next;

        public boolean hasNext() {
            return current != null;
        }
        public void remove() {}

        public Item next() {
            Item data = current.data;
            current = current.next;
            return data;
        }
    }
}
```

心得体会

1. 二叉树是一种高效的数据结构，使用二叉树可以达到很多算法的理论极限。
2. 二叉树可以使用其扩展二叉树的任意遍历方式构造。