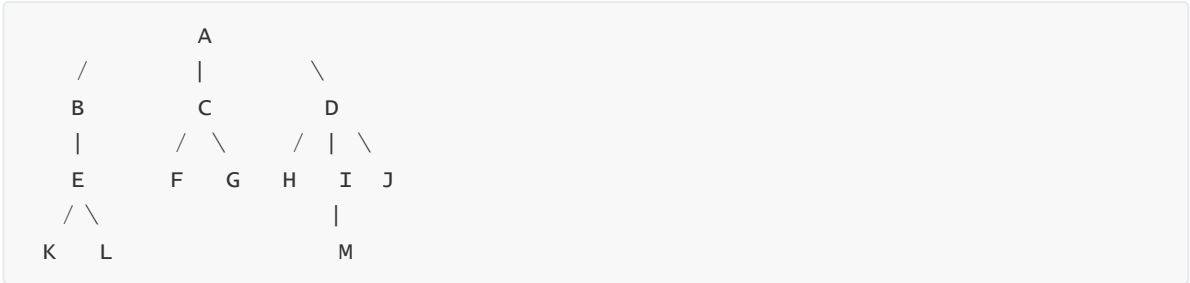
实验11

练习1

题目

要求用孩子兄弟链表实现（有序）树。根据一棵规模大于 1 的树的边序列构造该树。边用<x, y>表示，其中 x 是 y 的双亲，若 y 没有双亲，则 x 用一个标志表示。在边序列中，所有的 y 按层序排列。例如，下图所示树的边序列为 <#, A>, <A, B>, <A, C>, <A, D>, <B, E>, <C, F>, <C, G>, <D, H>, <D, I>, <D, J>, <E, K>, <E, L>, <I, M>。输出该树的螺旋遍历序列。螺旋遍历树是，若树为空，则什么也不做，否则，访问根，从右往左访问第 1 层的所有结点（设根的层数为 0），从左往右访问第2 层的所有结点，从右往左访问第 3 层的所有结点，从左往右访问第 4 层的所有结点……例如，下图所示树的螺旋遍历序列为ADCBEFGHIJMLK。



解析

以二维数组x的形式输入边序列，构造孩子兄弟链表，例如{{null, "A"}, { "A", "B"}, {"A", "C"}, {"A", "D"}}。创建辅助队列queue，开始时根节点的值初始化为x[0][1]，然后根节点入队。遍历数组x，若当前边的起始结点与上一条边的起始结点不同（即x[i][0] != x[i-1][0]），则不断出队，直到出队结点的值为当前边的起始结点，将该结点的孩子设为当前边的终止结点，并使当前边的终止结点入队；若当前边的起始结点与上一条边的起始结点相同，则设置队列中最后一个结点的兄弟为当前边的终止结点（不出队），并使当前边的终止结点入队。遍历结束时，孩子兄弟链表即构造完成。

需要注意的是，边序列必须是层序的，否则构造会出现问题。

螺旋遍历需要建立一个辅助栈stack和一个辅助队列queue。开始时根节点入队，然后不断重复以下步骤直至辅助栈和辅助队列都为空：

1. 队列中的结点依次出队，记录出队顺序，对每一个结点，将其孩子及孩子的所有兄弟都压入栈。
2. 栈中的结点依次出栈，记录出栈顺序，对每一个结点，将其孩子及孩子的所有兄弟加入缓存队列（不是之前的辅助队列），并将这些队列依次压入缓存栈（不是之前的辅助栈）。
3. 依次弹出缓存栈中的缓存队列，对每一个缓存队列，将其结点依次加入初始时定义的辅助队列之中。

当辅助栈和辅助队列都为空时，得到的结点顺序即为螺旋遍历序列。

代码

CST.java

```
// 利用Java泛型实现的孩子兄弟链表，利用边序列构造
public class CST<Item> {
    private Node root;

    private class Node {
        Item data;
        Node child;
        Node sibling;

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

    public CST() {
        this.root = null;
    }

    public CST(Item[][] x) {
        if (x.length == 0) {
            this.root = null;
        } else {
            this.root = new Node(x[0][1], null, null);
            Queue<Node> queue = new Queue<Node>();
            queue.enqueue(this.root);
            for (int i = 1; i < x.length; i++) {
                if (x[i][0] != x[i-1][0]) {
                    Node p = queue.dequeue();
                    while (p.data != x[i][0]) {
                        p = queue.dequeue();
                    }
                    p.child = new Node(x[i][1], null, null);
                    queue.enqueue(p.child);
                } else {
                    Node p = queue.getTail();
                    p.sibling = new Node(x[i][1], null, null);
                    queue.enqueue(p.sibling);
                }
            }
        }
    }

    public Item[] spiralOrder() {
        Stack<Node> stack = new Stack<Node>();
        Queue<Node> queue = new Queue<Node>();
        List<Item> result = new List<Item>();
        Node p;
        queue.enqueue(this.root);
        while (!queue.isEmpty() || !stack.isEmpty()) {
            while (!queue.isEmpty()) {
                p = queue.dequeue();
                result.append(p.data);
                p = p.child;
```

```
                while (p != null) {
                    stack.push(p);
                    p = p.sibling;
                }
            }
            Stack<Queue<Node>> queueStack = new Stack<Queue<Node>>();
            while (!stack.isEmpty()) {
                p = stack.pop();
                result.append(p.data);
                p = p.child;
                while (p != null) {
                    queue.enqueue(p);
                    p = p.sibling;
                }
                queueStack.push(queue);
                queue = new Queue<Node>();
            }
            while (!queueStack.isEmpty()) {
                Queue<Node> tmpQueue = queueStack.pop();
                while (!tmpQueue.isEmpty()) {
                    queue.enqueue(tmpQueue.dequeue());
                }
            }
        }
        return result.toArray();
    }
}
```

Exercise_1.java

```
import java.util.Arrays;

public class Exercise_1 {
    public static void main(String[] args) throws Exception {
        // read data and build tree
        String[][] in = { ... }; // in的具体值将在“输入”中给出
        CST<String> a = new CST<String>(in);
        // output
        System.out.println(Arrays.toString(a.spiralOrder()));
    }
}
```

输入

```
{{null, "A"},
 { "A", "B"}, {"A", "C"}, {"A", "D"},
 { "B", "E"}, {"C", "F"}, {"C", "G"}, {"D", "H"}, { "D", "I"}, {"D", "J"},
 { "E", "K"}, {"E", "L"}, {"I", "M"}}
```

输出

```
import java.util.Arrays;

public class Exercise_1 {
    public static void main(String[] args) throws Exception {
```

```
        // read data and build tree
        String[][] in = {{null, "A"},
                         { "A", "B"}, {"A", "C"}, {"A", "D"},
                         { "B", "E"}, {"C", "F"}, {"C", "G"}, {"D", "H"}, { "D",
"I"}, {"D", "J"},
                         { "E", "K"}, {"E", "L"}, {"I", "M"}};
        CST<String> a = new CST<String>(in);
        // output
        System.out.println(Arrays.toString(a.spiralOrder()));
    }
}
```

附

代码中所使用的List、Stack和Queue已经分别在实验6、实验8和实验9中定义过。但为了严谨起见，在此附上List.java、Stack.java和Queue.java的代码。

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--;
    }

    // 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;
        }
    }
```

```
}
```

Stack.java

```
import java.util.Iterator;

// 使用Java泛型实现的链栈，支持迭代和toString
class Stack<Item> implements Iterable<Item> {
    private Node top;
    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.top == null;
    }

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

    public Item peek() {
        if (this.top == null) {
            return null;
        } else {
            return this.top.data;
        }
    }

    public void push(Item data) {
        this.top = new Node(data, this.top);
        this.N++;
    }

    public Item pop() {
        if (this.top == null) {
            return null;
        } else {
            Item data = this.top.data;
            this.top = this.top.next;
            this.N--;
            return data;
        }
    }

    public String toString() {
        Node p = this.top;
        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 StackIterator();
    }

    private class StackIterator implements Iterator<Item> {
        private Node current = top;

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

        public void remove() {
        }

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

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 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;
        }
    }
}
```

心得体会

1. 栈和队列的使用能够高效地实现许多算法。
2. 孩子兄弟链表可以用来高效地表示树结构。