

# Java 数据结构和算法

## 第 0 讲 综述

参考教材：Java 数据结构和算法（第二版），[美] Robert lafore

### 1. 数据结构的特性

数据结构	优点	缺点
数组	插入快；如果知道下标，可以非常快地存取	查找慢，删除慢，大小固定
有序数组	比无序的数组查找快	删除和插入慢，大小固定
栈	提供后进先出方式的存取	存取其他项很慢
队列	提供先进先出方式的存取	存取其他项很慢
链表	插入快，删除快	查找慢
二叉树	查找、插入、删除都快（如果树保持平衡）	删除算法复杂
红-黑树	查找、插入、删除都快；树总是平衡的	算法复杂
2-3-4 树	查找、插入、删除都快；树总是平衡的；类似的树对磁盘存储有用	算法复杂
哈希表	如果关键字已知，则存储极快；插入快	删除慢，如果不知道关键字则存储很慢，对存储空间使用不充分
堆	插入、删除快；对大数据项的存取很快	对其他数据项存取慢
图	对现实世界建模	有些算法慢且复杂

### 2. 经典算法总结

查找算法：线性查找和二分查找

排序算法：

**用表展示**

## 第一讲 数组

### 1. Java 中数组的基础知识

#### 1) 创建数组

在 Java 中把数组当作对象来对待，因此在创建数组时必须使用 new 操作符：

```
int[] intArr = new int[10];
```

一旦创建数组，数组大小便不可改变。

#### 2) 访问数组数据项

数组数据项通过方括号中的下标来访问，其中第一个数据项的下标是 0：

```
intArr[0] = 123;
```

### 3) 数组的初始化

当创建数组之后，除非将特定的值赋给数组的数据项，否则它们一直是特殊的 null 对象。

```
int[] intArr = {1, 2, 3, 4, 5};
```

等效于下面使用 new 来创建数组并初始化：

```
int[] intArr = new int[5];
intArr[0] = 1;
intArr[1] = 2;
intArr[2] = 3;
intArr[3] = 4;
intArr[4] = 5;
```

## 2. 面向对象编程方式

### 1) 使用自定义的类封装数组

MyArray 类：

```
public class MyArray {
    private long[] arr;
    private int size; //记录数组的有效长度

    public MyArray() {
        arr = new long[10];
    }

    public MyArray(int maxSize) {
        arr = new long[maxSize];
    }

    //向数组中插入数据
    public void insert(long element) {
        arr[size] = element;
        size++;
    }

    //显示数组中的数据
    public void show() {
        for(int i=0; i<size; i++) {
            if(i==0) {
                System.out.print("[" + arr[i] + ", ");
            } else if(i==size-1) {
                System.out.println(arr[i] + "]");
            } else {
                System.out.print(arr[i] + ", ");
            }
        }
    }
}
```



```
        }
    }

//根据值查找索引（出现该值的第一个位置）：线性查找
public int queryByValue(long element) {
    int i;
    for(i=0; i<size; i++) { // linear search
        if(arr[i] == element) break;
    }
    if(i == size) {
        return -1;
    } else {
        return i;
    }
}

//根据索引查找值
public long queryByIndex(int index) {
    if(index >= size || index < 0) {
        throw new ArrayIndexOutOfBoundsException();
    } else {
        return arr[index];
    }
}

//删除数据
public void delete(int index) {
    if(index >= size || index < 0) {
        throw new ArrayIndexOutOfBoundsException();
    } else {
        //当 size=maxSize，删除最后一个数时，不会执行 for
        for(int i=index; i<size-1; i++) {
            arr[index] = arr[index + 1];
            System.out.println("for");
        }
        size--;
    }
}

//更新数据
public void update(int index, long value) {
    if(index >= size || index < 0) {
        throw new ArrayIndexOutOfBoundsException();
    }
}
```

```

        } else {
            arr[index] = value;
        }
    }
}

```

## 2) 添加类方法实现数据操作

测试 MyArray 类方法:

```

public void testMyArray() throws Exception {
    MyArray myArray = new MyArray();
    myArray.insert(123);
    myArray.insert(456);
    myArray.insert(789);
    myArray.show(); // [123, 456, 789]
    System.out.println(myArray.queryByValue(111)); // -1
    System.out.println(myArray.queryByIndex(2)); // 789
    myArray.delete(2);
    myArray.show(); // [123, 456]
    myArray.update(0, 0);
    myArray.show(); // [0, 456]
}

```

## 3. 有序数组

### 1) 有序数组简介以及其优点

有序数组是一种数组元素按一定的顺序排列的数组，从而方便使用二分查找来查找数组中特定的元素。有序数组提高了查询的效率，但并没有提高删除和插入元素的效率。

### 2) 构建有序数组

将 2.1 中自定义的类封装数组 MyArray 的 insert 方法改为如下:

```

// 向有序数组中插入数据，按大小从前往后排列
public void insert(long element) {
    int i;
    for(i=0; i<size; i++) { // find where it goes
        if(element<arr[i]) break;
    }
    for(int j=size; j>i; j--) { // move bigger ones up
        arr[j] = arr[j-1];
    }
    arr[i] = element;
    size++;
}

```

得到有序数组的类封装 MyOrderedArray 类，测试该类中的 insert 方法:



```
public void testMyOrderedArray() throws Exception {
    MyOrderedArray myOrderedArray = new MyOrderedArray();
    myOrderedArray.insert(999);
    myOrderedArray.insert(555);
    myOrderedArray.insert(777);
    myOrderedArray.show(); // [555, 777, 999]
}
```

## 4. 查找算法

### 1) 线性查找

在查找过程中，将要查找的数一个一个地与数组中的数据项比较，直到找到要找的数。在 2.1 中自定义的类封装数组 MyArray 的 queryByValue 方法，使用的就是线性查找。

### 2) 二分查找

二分查找（又称折半查找），即不断将有序数组进行对半分割，每次拿中间位置的数和要查找的数进行比较：如果要查找的数<中间数，则表明要查的数在数组的前半段；如果要查的数>中间数，则表明该数在数组的后半段；如果要查的数=中间数，则返回中间数。

在有序数组的类封装类 MyOrderedArray 中添加 binarySearch 方法

```
//根据值二分查找索引（前提：有序）
public int binarySearch(long value) {
    int middle = 0;
    int left = 0;
    int right = size - 1;

    while(true) {
        middle = (left + right) / 2;
        if(arr[middle] == value) {
            return middle; // found it
        } else if(left > right) {
            return -1; // can't found it
        } else { // divide range
            if(arr[middle] > value) {
                right = middle - 1; // in lower half
            } else {
                left = middle + 1; // in upper half
            }
        }
    }
}
```

测试该二分查找方法：



```
public void testMyOrderedArray() throws Exception {
    MyOrderedArray myOrderedArray = new MyOrderedArray();
    myOrderedArray.insert(999);
    myOrderedArray.insert(555);
    myOrderedArray.insert(777);
    myOrderedArray.insert(333);
    System.out.println(myOrderedArray.binarySearch(333)); //0
}
```

## 第二讲 简单排序

本讲提到的排序算法都假定了数组作为数据存储结构，本讲所有算法的时间复杂度都是。在大多数情况下，假设当数据量比较小或基本上有序时，插入排序算法是三种简单排序算法中最好的选择，是应用最多的。对于更大数据量的排序来说，后面讲到的快速排序通常是最快的方法。

### 1. 冒泡排序

#### 1) 基本思想

在要排序的一组数中，对当前还未排好序的范围内的全部数，自下而上对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。

#### 2) 算法实现

冒泡排序的 Java 代码：

```
// bubble sort
public static void bubbleSort(long[] arr) {
    long temp;
    for(int i=0; i<arr.length-1; i++) { //outer loop (forward)
        for(int j=arr.length-1; j>i; j--) { //inner loop (backward)
            if(arr[j] < arr[j-1]) { // swap them
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
        }
    }
}
```

测试冒泡排序及输出结果：

```
public void testBubbleSort() throws Exception {
    long[] arr = {79, 91, 13, 52, 34};
    Sort.bubbleSort(arr);
    System.out.println(Arrays.toString(arr)); // [13, 34, 52, 79, 91]
}
```

## 2. 选择排序

### 1) 基本思想

在要排序的一组数中，选出最小的一个数与第一个位置的数交换；然后在剩下的数当中再找最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。

与冒泡排序相比，选择排序将必要的交换次数从  $O(N^2)$  减少到  $O(N)$ ，但比较次数仍然保持为  $O(N^2)$ 。

### 2) 算法实现

选择排序的 Java 代码：

```
// select sort
public static void selectSort(long[] arr) {
    long temp;
    for(int i=0; i<arr.length-1; i++) { //outer loop
        int k = i; // location of minimum
        for(int j=i+1; j<arr.length; j++) { //inner loop
            if(arr[j] < arr[k]) {
                k = j; // a new minimum location
            }
        }
        temp = arr[i];
        arr[i] = arr[k];
        arr[k] = temp;
    }
}
```

测试选择排序及输出结果：

```
public void testSelectSort() throws Exception {
    long[] arr = {79, 91, 13, 52, 34};
    Sort.selectSort(arr);
    System.out.println(Arrays.toString(arr)); // [13, 34, 52, 79, 91]
}
```

## 3. 插入排序

### 1) 基本思想

在要排序的一组数中，假设前面( $n-1$ ) $[n \geq 2]$ 个数已经是排好顺序的（局部有序），现在要把第  $n$  个数插到前面的有序数中，使得这  $n$  个数也是排好顺序的。如此反复循环，直到全部排好顺序。

在插入排序中，一组数据仅仅是局部有序的；而冒泡排序和选择排序，一组数据项在某个时刻是完全有序的。

## 2) 算法实现

插入排序的 Java 代码:

```
// insert sort
public static void insertSort(long[] arr) {
    long temp;
    for(int i=1; i<arr.length; i++) { // i is marked location
        temp = arr[i]; // remove marked item
        int j = i; // start shifts at i
        while(j>=1 && arr[j-1]>temp) { // until one is smaller
            arr[j] = arr[j-1]; // shift item right
            j--; // go left one position
        }
        arr[j] = temp; // insert marked item
    }
}
```

测试插入排序以及输出结果:

```
public void testInsertSort() throws Exception {
    long[] arr = {79, 91, 13, 52, 34, 34};
    Sort.insertSort(arr);
    System.out.println(Arrays.toString(arr));
    // [13, 34, 34, 52, 79, 91]
}
```

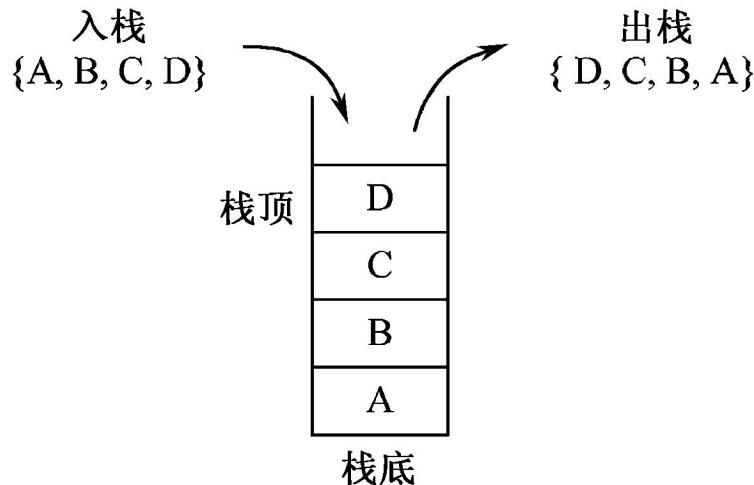
# 第三讲 栈和队列

栈和队列都是抽象数据类型 (abstract data type, ADT)，它们既可以用数组实现，又可以用链表实现。

## 1. 栈

### 1) 栈模型

栈 (Stack, 又 LIFO: 后进先出) 是一种只能在固定的一端进行插入和删除的数据结构。栈只允许访问一个数据项: 即最后插入的数据项, 移除这个数据项后才能访问倒数第二个插入的数据项, 以此类推。栈可以用数组来实现, 也可以用链表来实现。



## 2) 栈的数组实现

栈的 Java 代码:

```

public class MyStack {
    private long[] arr; //底层使用数组实现
    private int top;

    public MyStack() {
        arr = new long[10];
        top = -1;
    }

    public MyStack(int maxSize) {
        arr = new long[maxSize];
        top = -1;
    }

    // put item on top of stack
    public void push(long value) {
        arr[++top] = value;
    }

    // take item from top of stack
    public long pop() {
        return arr[top--];
    }

    // peek at top of stack
    public long peek() {
    }
}

```



```
    return arr[top];  
}  
  
// true if stack is empty  
public boolean isEmpty() {  
    return (top == -1);  
}  
  
// true if stack is full  
public boolean isFull() {  
    return (top == arr.length-1);  
}  
}
```

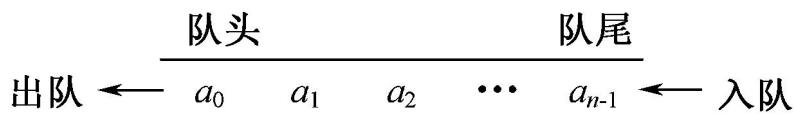
测试栈的特性：

```
public void testMyStack() throws Exception {  
    MyStack myStack = new MyStack(4);  
    myStack.push(12);  
    myStack.push(34);  
    myStack.push(56);  
    myStack.push(78);  
    System.out.println(myStack.isFull()); // true  
    while (!myStack.isEmpty()) {  
        System.out.print(myStack.pop()); //78, 56, 34, 12  
        if (!myStack.isEmpty()) {  
            System.out.print(", ");  
        }  
    }  
    System.out.println();  
    System.out.println(myStack.isFull()); // false  
}
```

## 2. 队列

### 1) 队列模型

队列 (Queue, 又 FIFO: 先进先出) 是一种插入时在一端进行而删除时在另一端进行的



数据结构。

为解决顺序队列假溢出问题，而采用循环队列：即让队头、队尾指针在达到尾端时，又绕回到开头。

## 2) 队列的数组实现

队列的 Java 代码:

```
public class MyQueue {  
    private long[] arr;  
    private int size;  
    private int front;  
    private int rear;  
  
    public MyQueue() {  
        arr = new long[10];  
        size = 0;  
        front = 0;  
        rear = -1;  
    }  
    public MyQueue(int maxSize) {  
        arr = new long[maxSize];  
        size = 0;  
        front = 0;  
        rear = -1;  
    }  
  
    // put item at rear of queue  
    public void insert(long value) {  
        if(isEmpty()) { // throw exception if queue is full  
            throw new ArrayIndexOutOfBoundsException();  
        }  
        if(rear == arr.length-1) { //deal with wraparound (环绕式处理)  
            rear = -1;  
        }  
        arr[++rear] = value; // increment rear and insert  
        size++; // increment size  
    }  
  
    // take item from front of queue  
    public long remove() {  
        long value = arr[front++]; // get value and increment front  
        if(front == arr.length) { // deal with wraparound  
            front = 0;  
        }  
        size--; // one less item  
        return value;  
    }  
}
```



```
// peek at front of queue
public long peek() {
    return arr[front];
}

// true if queue is empty
public boolean isEmpty() {
    return (size == 0);
}

// true if queue is full
public boolean isFull() {
    return (size == arr.length);
}
}
```

测试队列的特性：

```
public void testMyQueue() throws Exception {
    MyQueue myQueue = new MyQueue(4);
    myQueue.insert(12);
    myQueue.insert(34);
    myQueue.insert(56);
    myQueue.insert(78);
    System.out.println(myQueue.isFull()); // true
    while (!myQueue.isEmpty()) {
        System.out.print(myQueue.remove()); //12, 34, 56, 78
        if (!myQueue.isEmpty()) {
            System.out.print(", ");
        }
    }
    System.out.println();
    System.out.println(myQueue.isEmpty()); // true
    myQueue.insert(99);
    System.out.println(myQueue.peek()); // 99
}
```

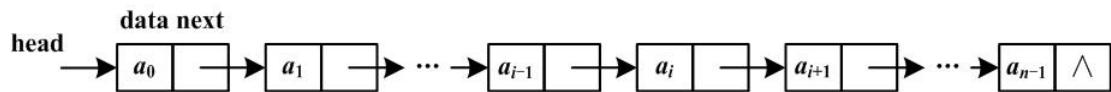
## 第四讲 链表

### 1. 链结点

一个链结点（Link，或称结点）是某个类的对象，该对象中包含对下一个链结点引用的



字段（通常叫 next）。而链表本身的对象中有一个字段指向对第一个链结点的引用。



Link 类的定义：

```
public class Link {  
    public long data; // data item  
    public Link next; // next link in list  
  
    public Link(long data) {  
        this.data = data;  
    }  
  
    public void displayLink() { // display ourself  
        System.out.print(data + "--> ");  
    }  
}
```

## 2. 单链表（LinkedList）

LinkedList 类显示了一个单链表，该链表可以进行的操作有：

- 在链表头插入一个数据（insertFirst）：设置新添加的结点的 next 设为原来的头结点，再将新添加的结点设为头结点。
- 在链表头删除一个数据（deleteFirst）：将第二个结点设为头结点。
- 遍历链表显示所有数据（displayList）：使用临时指针 current，从头结点开始，沿着链表先前移动，依次遍历每个节点。

LinkedList 类：

```
public class LinkedList {  
    private Link first; // reference to first link on list  
  
    public LinkedList() {  
        this.first = null;  
    }  
  
    // insert at start of list  
    public void insertFirst(long value) {  
        Link newLink = new Link(value);  
        newLink.next = first; // newLink --> old first  
        first = newLink; // first --> newLink  
    }  
  
    // delete first item  
    public Link deleteFirst() {
```



```
if(first == null) {  
    return null;  
}  
Link temp = first;  
first = first.next; // delete it: first --> old next  
return temp; // return deleted link  
}  
  
public void displayList() {  
    Link current = first; // start at beginning of list  
    while(current != null) { // until end of list  
        current.displayLink();  
        current = current.next; // move list to next link  
    }  
    System.out.println();  
}  
}
```

测试 LinkList 类的方法:

```
public void testLinkList() throws Exception {  
    LinkList linkList = new LinkList();  
    linkList.insertFirst(88);  
    linkList.insertFirst(66);  
    linkList.insertFirst(44);  
    linkList.insertFirst(22);  
    linkList.displayList(); //22--> 44--> 66--> 88-->  
  
    linkList.deleteFirst();  
    linkList.displayList(); //44--> 66--> 88-->  
}
```

### 3. 查找和删除指定链结点

此外，还能对指定的结点值进行查找和删除:

- 查找指定结点 (find) : 类似于之前的 displayList 方法。
- 删除指定结点 (delete) : 在删除指定结点时，必须把前一个结点和后一个结点连接在一起，所以需要一个临时指针 (previous) 来保存对前一个结点的引用。

向 LinkList 类中添加查找 (find) 和删除 (delete) 方法:

```
// find link with given key  
public Link find(long key) {  
    Link current = first; // start at 'first'  
    while(current.data != key) {  
        if(current.next == null) { // didn't find it  
            return null;
```



```
        } else { // go to next link
            current = current.next;
        }
    }
    return current;
}

// delete link with given key
public Link delete(long key) {
    Link current = first; // search for expected link
    Link previous = first;
    while(current.data != key) {
        if(current.next == null) {
            return null; // didn't find it
        } else {
            previous = current;
            current = current.next; // go to next link
        }
    } // find it
    if(current == first) { // if first link, change first
        first = first.next;
    } else { // otherwise, bypass it
        previous.next = current.next;
    }
    return current;
}
```

测试添加的 find 和 delete 方法：

```
public void testLinkList() throws Exception {
    LinkList linkList = new LinkList();
    linkList.insertFirst(88);
    linkList.insertFirst(66);
    linkList.insertFirst(44);
    linkList.insertFirst(22);
    linkList.displayList(); // 22--> 44--> 66--> 88-->

    linkList.find(44).displayLink(); // 44-->
    System.out.println();
    linkList.delete(44).displayLink(); // 44-->
    System.out.println();
    linkList.displayList(); // 22--> 66--> 88-->
}
```

## 第五讲 双端链表和双向链表

### 1. 双端链表

链表中保存着对最后一个链结点的引用

- 从头部进行插入（insertFirst）：要对链表进行判断，如果链表为空，则设置尾结点为新添加的结点。
- 从尾部进行插入（insertLast）：如果链表为空，则直接设置头结点为新添加的结点，否则设置尾结点的后一个节点为新添加的结点。
- 从头部进行删除（deleteFirst）：判断头结点是否有下一个结点，如果没有则设置尾结点为 null。

双端链表 FirstLastList 类：

```
public class FirstLastList {  
    private Link first; // reference to first link  
    private Link last; // reference to last link  
  
    public FirstLastList() {  
        this.first = null;  
        this.last = null;  
    }  
  
    public boolean isEmpty() { // true if no links  
        return first == null;  
    }  
  
    // insert at front of list  
    public void insertFirst(long value) {  
        Link newLink = new Link(value);  
        if(isEmpty()) { // if empty list  
            last = newLink; // newLink <-- last  
        }  
        newLink.next = first; // newLink --> old first  
        first = newLink; // first --> newLink  
    }  
  
    // insert at end of list  
    public void insertLast(long value) {  
        Link newLink = new Link(value);  
        if(isEmpty()) { // if empty list  
            first = newLink; // first --> newLink  
        } else {  
        }  
    }  
}
```



```
        last.next = newLink; // old last --> newLink
    }
    last = newLink; // newLink <-- last
}

// delete first link
public Link deleteFirst() {
    Link temp = first;
    if(first.next == null) { // if only one item
        last = null; // null <-- last
    }
    first = first.next; // first --> old next
    return temp;
}

public void displayList() {
    Link current = first; // start at beginning of list
    while(current != null) { // until end of list
        current.displayLink();
        current = current.next; // move list to next link
    }
    System.out.println();
}
}
```

测试 FirstLastList 类及输出结果:

```
public void testFirstLastList() throws Exception {
    FirstLastList list = new FirstLastList();
    list.insertLast(24);
    list.insertFirst(13);
    list.insertFirst(5);
    list.insertLast(46);
    list.displayList(); //5--> 13--> 24--> 46-->

    list.deleteFirst();
    list.displayList(); //13--> 24--> 46-->
}
```

## 2. 双向链表

双向链表既允许向后遍历，也允许向前遍历整个链表。即每个节点除了保存了对下一个节点的引用，同时后保存了对前一个节点的引用。

- 从头部进行插入（insertFirst）：要对链表进行判断，如果为空，则设置尾结点为新添

加的结点；如果不为空，还需要设置头结点的前一个结点为新添加的结点。

- 从尾部进行插入（insertLast）：如果链表为空，则直接设置头结点为新添加的结点，否则设置尾节点的后一个结点为新添加的结点。同时设置新添加的结点的前一个结点为尾结点。
- 从头部进行删除（deleteFirst）：判断头结点是否有下一个结点，如果没有则设置尾结点为 null；否则设置头结点的下一个结点的 previous 为 null。
- 从尾部进行删除（deleteLast）：如果头结点后没有其他结点，则设置头结点为 null。否则设置尾结点的前一个结点的 next 为 null。设置尾结点为其前一个结点。
- 在指定结点后插入（insertAfter）：
- 删除指定结点（deleteKey）：不再需要在使用一个临时的指针域指向下一个结点。

双向链表的链接点 Link 类的定义：

```
public class DoubleLink {
    public long data; // data item
    public DoubleLink next; // next link in list
    public DoubleLink previous; // previous link in list

    public DoubleLink(long data) {
        this.data = data;
    }

    public void displayLink() { // display ourself
        System.out.print(data + "<==> ");
    }
}
```

双向链表 DoublyLinkedList 类：

```
public class DoublyLinkedList {
    private DoubleLink first; // reference to first link
    private DoubleLink last; // reference to last link

    public DoublyLinkedList() {
        this.first = null;
        this.last = null;
    }

    public boolean isEmpty() { // true if no links
        return first == null;
    }

    // insert at front of list
    public void insertFirst(long value) {
        DoubleLink newLink = new DoubleLink(value);
        if(isEmpty()) { // if empty list
            first = newLink;
            last = newLink;
        } else {
            newLink.next = first;
            first.previous = newLink;
            first = newLink;
        }
    }

    // delete from front of list
    public void deleteFirst() {
        if(first == null)
            return;
        if(first == last)
            last = null;
        first.next.previous = null;
        first = first.next;
    }
}
```



```
        last = newLink; // newLink <-- last
    } else {
        first.previous = newLink; // newLink <-- old first
    }
    newLink.next = first; // newLink --> old first
    first = newLink; // first --> newLink
}

// insert at end of list
public void insertLast(long value) {
    DoubleLink newLink = new DoubleLink(value);
    if(isEmpty()) { // if empty list
        first = newLink; // first --> newLink
    } else {
        last.next = newLink; // old last --> newLink
        newLink.previous = last; //old last <-- newLink
    }
    last = newLink; // newLink <-- last
}

// delete first link
public DoubleLink deleteFirst() {
    DoubleLink temp = first;
    if(first.next == null) { // if only one item
        last = null; // null <-- last
    } else {
        first.next.previous = null; // null <-- old next
    }
    first = first.next; // first --> old next
    return temp;
}

//delete last link
public DoubleLink deleteLast() {
    DoubleLink temp = last;
    if(first.next == null) { // if only one item
        first = null; // first --> null
    } else {
        last.previous.next = null; //old previous --> null
    }
    last = last.previous; // old previous <-- last
    return temp;
}
```



```
// insert data just after key
public boolean insertAfter(long key, long data) {
    DoubleLink current = first; //start at beginning
    while(current.data != key) { //until match is found
        if(current.next == null) {
            return false;
        } else {
            current = current.next;
        }
    } //find the position of key
    DoubleLink newLink = new DoubleLink(data); // make new link
    if(current == last) { //if last link,
        newLink.next = null; // newLink --> null
        last = newLink; // newLink <-- last
    } else { // not last link.
        newLink.next = current.next; // newLink --> old next
        current.next.previous = newLink; //newLink <-- old next
    }
    current.next = newLink; // old current --> newLink
    newLink.previous = current; // old current <-- newLink
    return true;
}

// delete link with given key
public DoubleLink deleteKey(long key) {
    DoubleLink current = first; // search for expected link
    while(current.data != key) {
        if(current.next == null) {
            return null; // didn't find it
        } else {
            current = current.next; // go to next link
        }
    } // find it
    if(current == first) { // if first link, change first
        first = first.next; // first --> old next
    } else { // otherwise, old previous --> old next
        current.previous.next = current.next;
    }
    if(current == last) { // last item?
        last = last.previous; // old previous <-- last
    } else { //not last: old previous <-- old next
        current.next.previous = current.previous;
    }
    return current;
}
```



```
public void displayForward() {
    System.out.print("first-->last: ");
    DoubleLink current = first; // start at beginning of list
    while(current != null) { // until end of list
        current.displayLink();
        current = current.next; // move list to next link
    }
    System.out.println();
}

public void displayBackward() {
    System.out.print("last-->first: ");
    DoubleLink current = last; // start at end of list
    while(current != null) { // until start of list
        current.displayLink();
        current = current.previous; // move list to previous link
    }
    System.out.println();
}
}
```

测试该类的主要方法:

```
public void testDoublyLinkedList() throws Exception {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insertLast(24);
    list.insertFirst(13);
    list.insertFirst(5);
    list.insertLast(46);
    list.displayForward();
    //first-->last: 5<==> 13<==> 24<==> 46<==>
    list.displayBackward();
    //last-->first: 46<==> 24<==> 13<==> 5<==>

    list.deleteFirst();
    list.displayForward(); //first-->last: 13<==> 24<==> 46<==>
    list.deleteLast();
    list.displayForward(); //first-->last: 13<==> 24<==>

    list.insertAfter(13, 17);
    list.displayForward(); //first-->last: 13<==> 17<==> 24<==>
    list.deleteKey(24);
    list.displayForward(); //first-->last: 13<==> 17<==>
}
```

## 第六讲 递归的应用

递归（Recursion）是一种在方法（函数）的定义中调用方法自身的编程技术。

- 递归算法解决问题的特点：

(1) 递归就是在过程或函数里调用自身。

(2) 在使用递归策略时，必须有一个明确的递归结束条件，称为递归出口。

(3) 递归算法解题通常显得很简洁，但递归算法解题的运行效率较低。所以一般不提倡用递归算法设计程序。

(4) 在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储。递归次数过多容易造成栈溢出等。所以一般不提倡用递归算法设计程序。在实际编程中尤其要注意栈溢出问题。

- 构成递归需具备的条件：

a. 子问题须与原始问题为同样的事，且更为简单；

b. 不能无限制地调用本身，须有个出口，化简为非递归状况处理。

### 1. 三角数字

该数列中的首项为 1，第 n 项是由第 n-1 项加 n 后得到的。

#### 1) 使用循环查找第 n 项

```
public static int triangleByWhile(int n) {
    int total = 0;
    while(n > 0) {
        total = total + n;
        n--;
    }
    return total;
}
System.out.println(Triangle.triangleByWhile(5)); //15
```

#### 2) 使用递归查找第 n 项

```
public static int triangleByRecursion(int n) {
    if(n == 1) {
        return 1;
    } else {
        return n + triangleByRecursion(n-1);
    }
}
```

```
System.out.println(Triangle.triangleByRecursion(5)); //15
```

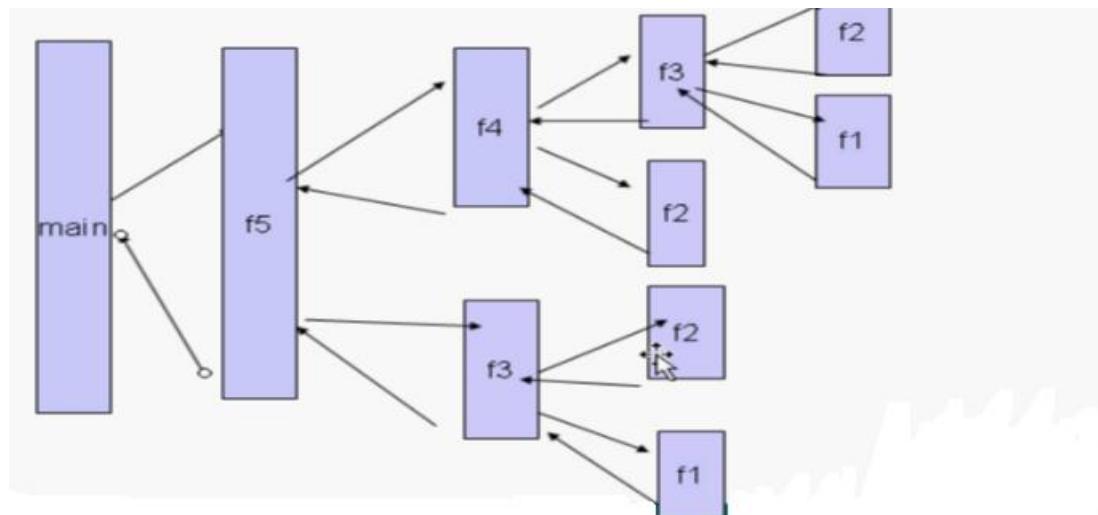
## 2. Fibonacci 数列

Fibonacci 数列的第 0 项为 0，第二项为 1，第 n 项为第 n-1 项加上 n-2 项后得到。该数列的递归算法实现如下：

```
public static int fibo(int n) {
    if (n==1 || n==2) {
        return 1;
    } else {
        return fibo(n-1) + fibo(n-2);
    }
}
```

```
System.out.println(FibonacciSequence.fibo(8)); //21
```

程序的运行流程如下：



# 第七讲 递归的高级应用

## 1. 汉诺塔问题

汉诺塔问题描述如下：

有三根杆子 A, B, C。A 杆上有 N 个(N>1)穿孔圆盘，盘的尺寸由下到上依次变小。要求按下列规则将所有圆盘移至 C 杆：

每次只能移动一个圆盘；

大盘不能叠在小盘上面。

提示：可将圆盘临时置于 B 杆，也可将从 A 杆移出的圆盘重新移回 A 杆，但都必须遵循上述两条规则。

问：如何移？最少要移动多少次？

汉诺塔问题的递归求解方法（移动子树）：把除了最低端盘子外的所有盘子形成的子树移动到一个中介塔上，然后把最低端盘子移到目标塔上，如此循环最终把那个子树移动到目标塔上。

Java 代码实现如下：

```

public static void doTowers(int topN, char from, char inter, char to)
{
    if (topN == 1) {
        System.out.println("Disk 1 from " + from + " to " + to);
    } else {
        doTowers(topN-1, from, to, inter); // from --> inter
        System.out.println("Disk " + topN + " from " + from
                           + " to " + to );
        doTowers(topN-1, inter, from, to); // inter --> to
    }
}

HanoiTower.doTowers(3, 'A', 'B', 'C');
// Disk 1 from A to C
// Disk 2 from A to B
// Disk 1 from C to B
// Disk 3 from A to C
// Disk 1 from B to A
// Disk 2 from B to C
// Disk 1 from A to C

```

## 2. 递归的二分查找

## 3. 归并排序

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

## 4. 消除递归

递归算法实际上是一种分而治之的方法，它把复杂问题分解为简单问题来求解。对于某



些复杂问题(例如 hanio 塔问题)，递归算法是一种自然且合乎逻辑的解决问题的方式，但是递归算法的执行效率通常比较差。因此，在求解某些问题时，常采用递归算法来分析问题，用非递归算法来求解问题；另外，有些程序设计语言不支持递归，这就需要把递归算法转换为非递归算法。将递归算法转换为非递归算法有两种方法，一种是直接求值，不需要回溯；另一种是不能直接求值，需要回溯。前者使用一些变量保存中间结果，称为直接转换法；后者使用栈保存中间结果，称为间接转换法，下面分别讨论这两种方法。

### 1. 直接转换法

直接转换法通常用来消除尾递归和单向递归，将递归结构用循环结构来替代。尾递归是指在递归算法中，递归调用语句只有一个，而且是处在算法的最后。例如求阶乘的递归算法：

```
public long fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

当递归调用返回时，是返回到上一层递归调用的下一条语句，而这个返回位置正好是算法的结束处，所以

，不必利用栈来保存返回信息。对于尾递归形式的递归算法，可以利用循环结构来替代。例如求阶乘的递归算法

可以写成如下循环结构的非递归算法：

```
public long fact(int n)
{
    int s=0;
    for (int i=1; i
        s=s*i; //用 s 保存中间结果
    return s;
}
```

单向递归是指递归算法中虽然有多处递归调用语句，但各递归调用语句的参数之间没有关系，并且这些递归

调用语句都处在递归算法的最后。显然，尾递归是单向递归的特例。例如求斐波那契数列的递归算法如下：

```
public int f(int n)
{
    if (n==1 || n==0) return 1;
    else return f(n-1)+f(n-2);
}
```

对于单向递归，可以设置一些变量保存中间结构，将递归结构用循环结构来替代。例如求斐波那契数列的算

法中用 s1 和 s2 保存中间的计算结果，非递归函数如下：

```
public int f(int n)
{
    int i, s;
    int s1=1, s2=1;
    for (i=3; i {
```

```

s=s1+s2;
s2=s1; // 保存 f(n-2)的值
s1=s; //保存 f(n-1)的值
}
return s;
}

```

## 2. 间接转换法

该方法使用栈保存中间结果，一般需根据递归函数在执行过程中栈的变化得到。其一般过程如下：

```

将初始状态 s0 进栈
while (栈不为空)
{
    退栈, 将栈顶元素赋给 s;
    if(s 是要找的结果) 返回;
    else {
        寻找到 s 的相关状态 s1;
        将 s1 进栈
    }
}

```

间接转换法在数据结构中有较多实例，如二叉树遍历算法的非递归实现、图的深度优先遍历算法的非递归实现等等，请读者参考主教材中相关内容

# 第八讲 希尔排序

希尔排序是由 Donald L.Shell 提出来的，希尔排序基于插入排序，并且添加了一些新的特性，从而大大提高了插入排序的执行效率。

插入排序的缺陷：多次移动。假如一个很小的数据在靠右端位置上，那么要将该数据排序到正确的位置上，则所有的中间数据都要向右移动一位。

希尔排序的优点：希尔排序通过加大插入排序中元素元素之间的间隔，并对这些间隔的元素进行插入排序，从而使得数据可以大幅度的移动。当完成该间隔的排序后，希尔排序会减少数据间的间隔再进行排序。依次进行下去。

## 1. 基本思想

希尔排序（最小增量排序）：算法先将要排序的一组数按某个增量  $d$  ( $n/2, n$  为要排序数的个数) 分成若干组，每组中记录的下标相差  $d$ 。对每组中全部元素进行直接插入排序，然后再用一个较小的增量 ( $d/2$ ) 对它进行分组，在每组中再进行直接插入排序。当增量减到 1 时，进行直接插入排序后，排序完成。

间隔的计算：间隔  $h$  的初始值为 1，通过  $h = 3*h + 1$  来循环计算，知道该间隔大于数组的大小时停止。最大间隔为不大于数组大小的最大值。

间隔的减少：通过公式  $h = (h - 1)/3$  来计算。

## 2. 算法实现

希尔排序的 Java 代码:

```
// Shell sort
public static void shellSort(long[] arr) {
    int h = 1;
    while(h < arr.length/3) { //find initial value of h
        h = h * 3 + 1;
    } // h = {1, 4, 13, 40, 121, ...}
    while(h>0) { // decreasing h, until h=1
        // insert sort
        long temp;
        for(int i=h; i<arr.length; i++) { // i is marked location
            temp = arr[i]; // remove marked item
            int j = i; // start shifts at i
            while(j > h-1 && arr[j-h]>temp) { // until one is smaller
                arr[j] = arr[j-h]; // shift item right
                j -= h; // go left h position
            }
            arr[j] = temp; // insert marked item
        }
        // decrease h
        h = (h-1)/3;
    }
}
```

测试希尔排序及输出结果:

```
public void testShellSort() throws Exception {
    long[] arr = {10, 5, 8, 7, 1, 6, 4, 9, 2, 3};
    Sort.shellSort(arr);
    System.out.println(Arrays.toString(arr));
    // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
```

## 第九讲 快速排序

### 1. 快速排序的基本思想

快速排序: 选择一个关键字, 通常选择第一个元素或者最后一个元素, 通过一趟扫描,

将待排序列分成两部分，一部分比关键字小，另一部分大于等于关键字，此时关键字在其排好序后的正确位置，然后再用同样的方法递归地排序划分出来的两部分。

如何进行划分：设定关键字，将比关键字小的数据放在一组，比关键字大的放在另一组。

如何自动设定关键字：设置数组最右端的数据为关键字。

## 2. 快速排序的算法实现

快速排序的 Java 代码：

```
public class Sort {  
    public static void quickSort(long[] arr, int left, int right) {  
        if(right-left <= 0) { // if size <= 1,  
            return; // already sorted  
        } else { // otherwise  
            long key = arr[right]; // rightmost item  
            // partition range  
            int point = partition(arr, left, right, key);  
            quickSort(arr, left, point-1); // sort left side  
            quickSort(arr, point+1, right); // sort right side  
        }  
    }  
  
    public static int partition(long[] arr, int left, int right, long key)  
    {  
        int leftP = left - 1; //left (after ++)  
        int rightP = right; //right-1 (after --)  
        while(true) {  
            while(leftP<right && arr[++leftP]<key)  
                ; // find bigger item  
            while(rightP>left && arr[--rightP]>key)  
                ; // find smaller item  
//            while(arr[++leftP]<key);  
//            while(rightP>0 && arr[--rightP]>key);  
            if(leftP >= rightP) { // if pointers cross,  
                break; // partition done  
            } else { // not crossed, so swap elements  
                long temp = arr[leftP];  
                arr[leftP] = arr[rightP];  
                arr[rightP] = temp;  
            }  
        }  
        // restore key  
        long temp = arr[leftP];  
        arr[leftP] = arr[right];  
    }  
}
```

```
    arr[right] = temp;
    return leftP; // return key location
}
}
```

测试快速排序及输出结果：

```
public void testQuickSort() throws Exception {
    // also can fill arr with random numbers
    long[] arr = {10, 5, 8, 7, 1, 6, 4, 9, 2, 3};
    Sort.quickSort(arr, 0, arr.length-1);
    System.out.println(Arrays.toString(arr));
    // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
```

## 补充 基数排序

基数排序：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

# 第十讲 二叉树的基本概念

## 1. 树

### 1) 为什么要使用树

有序数组插入数据项和删除数据项太慢

链表查找数据太慢

在树中能快速地查找、插入和删除数据项

### 2) 树的基本概念

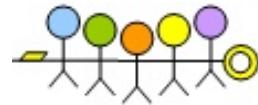
路径：顺着连接节点的边从一个节点到另一个节点，所经过的节点顺序排列称为路径。

根：树最上面的节点称为根节点。一棵树只有一个根，而且从根到任何一个节点有且只有一条路径。

父节点：每个节点都有一条边向上连接到另一个节点，这个节点就称为是下面这个节点的父节点。

子节点：每个节点都有一条边向下连接到另一个节点，下面的节点就是该节点的子节点。

叶子节点：没有子节点的节点称为叶子节点。



子树：每个节点都可以作为一个子树的根，它和它所有的子节点以及子节点的子节点组合在一起就是一个子树。

访问：访问一个节点是为了在这个节点上执行一些操作，如查看节点的数据项。但是如果仅仅是经过了一个节点，不认为是访问了这个节点。

层：一个节点的层数是指从根节点开始到这个节点有多少代。

## 2. 二叉树

树的每个节点最多只能有两个子节点的树，称为二叉树。

二叉树的代码实现：

节点类：

```
public class Node {  
    long data;  
    Node leftChild;  
    Node rightChild;  
  
    public Node(long data) {  
        this.data = data;  
    }  
}
```

二叉树：

```
public class BinaryTree {  
    Node root;  
    public void insert(long value) {}  
    public void delete(long value) {}  
    public void find(long value) {}  
}
```

# 第十一讲 二叉树的基本操作

## 1. 插入节点

从根节点开始查找一个相应的节点，这个节点将成为新插入节点的父节点。当父节点找到后，通过判断新节点的值比父节点的值的大小来决定是连接到左子节点还是右子节点。

插入节点的 Java 代码实现：

```
public void insert(long value) {  
    Node newNode = new Node(value);  
  
    if (root == null) { //no node in root  
        root = newNode;
```



```
    } else { // root occupied
        Node current = root; //start at root
        Node parent; //point to parent
        while(true) {
            parent = current;
            if(value < current.data) { //go left?
                current = current.leftChild;
                if(current == null) { //if end of the line
                    parent.leftChild = newNode; //insert on left
                    return;
                }
            } else { //or go right?
                current = current.rightChild;
                if(current == null) { //if end of the line
                    parent.rightChild = newNode; //insert on right
                    return;
                }
            }
        }
    }
}
```

测试该方法及输出：

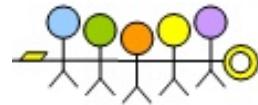
```
public void testBinaryTree() throws Exception {
    BinaryTree tree = new BinaryTree();
    tree.insert(34);
    tree.insert(21);
    tree.insert(67);
    tree.insert(56);
    System.out.println(tree.root.data); //34
    System.out.println(tree.root.leftChild.data); //21
    System.out.println(tree.root.rightChild.data); //67
    System.out.println(tree.root.rightChild.leftChild.data); //56
}
```

## 2. 查找节点

从根节点开始查找，如果查找的值比当前节点的值小，则继续查找器做左子树，否则查找其右子树。

查找节点的 Java 代码实现：

```
//find node with given key
public Node find(long value) {
    Node current = root; //start at root
    while(current.data != value) { //while no match,
```



```
    if(value < current.data) { //go left?
        current = current.leftChild;
    } else { //or go right?
        current = current.rightChild;
    }
    if(current == null) { // if no child,
        return null; //didn't find
    }
}
return current;
}
```

测试查找节点的方法及输出：

```
System.out.println(tree.find(21).data);
```

## 第十二讲 遍历二叉树

遍历树：遍历树是根据一个特定的顺序访问树的每一个节点，根据顺序的不同分为前序、中序和后序遍历三种。

### 1. 前序遍历

前序遍历：先访问根节点，再前序遍历左子树，最后前序遍历右子树；

前序遍历的 Java 代码实现：

```
public void frontOrder(Node node) {
    if(node != null) {
        System.out.println(node.data);
        frontOrder(node.leftChild);
        frontOrder(node.rightChild);
    }
}
tree.frontOrder(tree.root); //34 21 67 56
```

### 2. 中序遍历

中序遍历：先中序遍历左子树，再访问根节点，最后中序遍历右子树；

中序遍历按关键值的升序节点，从而形成一组有序数据。

中序遍历的 Java 代码实现：

```
public void inOrder(Node node) {
    if(node != null) {
```

```

        inOrder(node.leftChild);
        System.out.println(node.data);
        inOrder(node.rightChild);
    }
}

tree.inOrder(tree.root); //21 34 56 67

```

### 3. 后序遍历

后序遍历：先后序遍历左子树，再后序遍历右子树，最后访问根节点。

后序遍历的 Java 代码实现：

```

public void afterOrder(Node node) {
    if (node != null) {
        afterOrder(node.leftChild);
        afterOrder(node.rightChild);
        System.out.println(node.data);
    }
}

tree.afterOrder(tree.root); //21 56 67 34

```

## 第十三讲 删 除二叉树节点

### 1. 删 除节点的三种情况

删除节点是二叉树操作中最复杂的。在删除之前首先要查找要删的节点。找到节点后，这个要删的节点可能会有三种情况需要考虑：

- 该节点是叶子节点，没有子节点

要删除叶子节点，只需要改变该节点的父节点的引用值，将指向该节点的引用设置为 null 就可以了。

- 该节点有一个子节点。

改变父节点的引用，将其指向要删除节点的子节点。

- 该节点有两个子节点。

要删除节点有两个子节点，就需要使用它的中序后继来替代该节点。

### 2. 删 除节点的代码实现

删除节点的 Java 代码实现：

```
// delete node with given key
```



```
public boolean delete(long value) {  
    Node current = root;  
    Node parent = current;  
    boolean isLeftChild = true;  
    while(current.data != value) { //while no match,  
        parent = current;  
        if(value < current.data) { //go left?  
            current = current.leftChild;  
            isLeftChild = true;  
        } else { //or go right?  
            current = current.rightChild;  
            isLeftChild = false;  
        }  
        if(current == null) { // if no child,  
            return false; //didn't find  
        }  
    } //find the node to delete  
    // if no child,  
    if(current.leftChild==null && current.rightChild==null) {  
        if(current == root) { // if root,  
            root = null; // tree is empty  
        } else if(isLeftChild) {  
            parent.leftChild = null;  
        } else {  
            parent.rightChild = null;  
        }  
    } else if(current.leftChild == null) {  
        // if no left child, replace with right subtree  
        if(current == root) {  
            root = current.rightChild;  
        } else if(isLeftChild) { // left child of parent  
            parent.leftChild = current.rightChild;  
        } else { // right child of parent  
            parent.rightChild = current.rightChild;  
        }  
    } else if(current.rightChild == null) {  
        // if no right child, replace with left subtree  
        if(current == root) {  
            root = current.leftChild;  
        } else if(isLeftChild) { // left child of parent  
            parent.leftChild = current.leftChild;  
        } else { // right child of parent  
            parent.rightChild = current.leftChild;  
        }  
    }  
}
```



```
    } else { //two children, so replace with inorder successor
        //get successor of node to delete (current)
        Node successor = getSuccessor(current);
        // connect parent of current to successor instead
        if(current == root) {
            root = successor;
        } else if(isLeftChild) {
            parent.leftChild = successor;
        } else {
            parent.rightChild = successor;
        }
        //connect successor to current's left child
        successor.leftChild = current.leftChild;
    }
    return true;
}

// get successor to replace deleted node
// returns node with next-highest value after delNode
// goes to right child, then right child's left descendants
private Node getSuccessor(Node delNode) {
    Node current = delNode.rightChild; //go to right child
    Node successorParent = delNode;
    Node successor = delNode;

    while(current != null) { //until no more
        successorParent = successor;
        successor = current;
        current = current.leftChild; //go to left child
    }
    // if successor not right child, make connections
    if(successor != delNode.rightChild) {
        successorParent.leftChild = successor.rightChild;
        successor.rightChild = delNode.rightChild;
    }
    return successor;
}
```

测试删除各种节点的情况：

tree.delete(56); // no child
tree.inOrder(tree.root); //21 34 67 70
tree.delete(67); // only left child
tree.inOrder(tree.root); //21 34 56
tree.delete(21); // two children
tree.inOrder(tree.root); //34 56 67

## 第十四讲 红黑树

### 1. 平衡树与非平衡树

#### 1) 二叉树的问题

前面介绍了二叉树，普通的二叉树作为数据存储的工具有很大的优势，可以快速插入、删除和查找数据项。遗憾的是，这仅仅是相对于插入随机数据，如果插入的数据是有序的，速度就变得特别的慢。

#### 2) 平衡树和非平衡树

平衡树：插入随机的数据。

非平衡树：插入有序的数据。

完全平衡树：

### 2. 红黑规则

#### 1) 红黑规则

- (1) 每个节点不是红色的就是黑色的
- (2) 根总是黑色的
- (3) 如果节点是红色的，则它的子节点必须是黑色的
- (4) 从根节点到叶节点的每条路径，必须包含相同数目的黑色节点

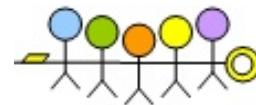
#### 2) 纠正违规

将不符合红黑规则的数据纠正为红黑树。

- (1) 改变节点的颜色
- (2) 执行旋转操作

## 第十五讲 哈希表

### 1. 什么是哈希表



哈希表是一种数据结构，它可以提供快速的插入和查找操作。哈希表是基于数组来实现的。

## 2. 哈希化

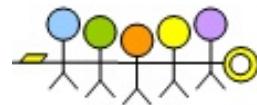
### 1) 直接将关键字作为索引

Info 类：

```
public class Info {  
    private int id;  
    private String name;  
  
    public Info(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

关键字作为索引的 Java 代码实现：

```
public class HashTable {  
    Info[] arr;  
    public HashTable() {  
        arr = new Info[10000];  
    }  
    public HashTable(int maxSize) {  
        arr = new Info[maxSize];  
    }  
    // insert element  
    public void insert(Info info) {  
        arr[info.getId()] = info;  
    }  
    // find element  
    public Info find(int id) {  
        return arr[id];  
    }
```



```
}
```

测试插入和查找方法:

```
public void testHashTable() throws Exception {
    HashTable table = new HashTable();
    table.insert(new Info(1, "zhangSan"));
    table.insert(new Info(2, "tianQi"));
    System.out.println(table.find(1).getName());
}
```

## 2) 将单词转换成索引

①将字母转换成 ASCII 码，然后进行相加

转码索引的 Java 代码实现:

```
public int hashCode(String key) {
    int code = 0;
    for(int i=key.length()-1; i>=0; i--) {
        int value = key.charAt(i) - 96;
        code += value;
    }
    return code;
}

// insert element
public void insert(Info info) {
    arr[hashCode(info.getId())] = info;
}

// find element
public Info find(String key) {
    return arr[hashCode(key)];
}
```

测试及输出:

```
public void testHashTable() throws Exception {
    HashTable table = new HashTable();
    table.insert(new Info("abc", "zhangSan"));
    table.insert(new Info("bbb", "tianQi"));
    System.out.println(table.find("abc").getName()); //tianQi
    System.out.println(table.find("bbb").getName()); //tianQi
}
```

从上可见上面的编码方式存在很高的重复率。

## ②幂的连乘

幂的连乘编码 Java 代码实现:

```
public int hashCode(String key) {
```

```

int code = 0;
int power27 = 1;
for(int i=key.length()-1; i>=0; i--) {
    int value = key.charAt(i) - 96;
    code += value * power27;
    power27 *= 27;
}
return code;
}

```

测试及输出：

```

System.out.println(table.find("abc").getName()); //zhangSan
System.out.println(table.find("bbb").getName()); //tianQi

```

### ③压缩可选值

为了解决编码后的数据过大，对其进行取模运算

```

public int hashCode(String key) {
    BigInteger hashVal = new BigInteger("0");
    BigInteger pow27 = new BigInteger("1");
    for(int i = key.length() - 1; i >= 0; i--) {
        int letter = key.charAt(i) - 96;
        BigInteger letterB = new BigInteger(String.valueOf(letter));
        hashVal = hashVal.add(letterB.multiply(pow27));
        pow27 = pow27.multiply(new BigInteger(String.valueOf(27)));
    }
    return hashVal.mod(
        new BigInteger(String.valueOf(arr.length))).intValue();
}

```

## 3. 压缩后仍可能出现的问题

冲突，不能保证每个单词都映射到数组的空白单元。

解决办法：

- ①开放地址法
- ②链地址法

## 第十六讲 开放地址法

### 1. 什么是开放地址法

当冲突发生时，通过查找数组的一个空位，并将数据填入，而不再用哈希函数得到的数据组下标，即开放地址法。

## 2. 数据的插入

数据插入的 Java 代码实现：

```
// insert element
public void insert(Info info) {
    String key = info.getId();
    int code = hashCode(key); // hash the key
    while(arr[code] != null && arr[code].getName() != null) {
        ++code; // go to next cell
        code %= arr.length; //wrap around if necessary
    }
    arr[code] = info;
}

HashTable table = new HashTable();
table.insert(new Info("a", "zhangSan"));
table.insert(new Info("ct", "tianQi"));
```

## 3. 数据的查找

数据查找的 Java 代码实现：

```
// find element with key
public Info find(String key) {
    int code = hashCode(key); //hash the key
    while(arr[code] != null) { //until empty cell.
        if(arr[code].getId() == key) { //found the key?
            return arr[code]; //yes, return element
        }
        ++code; // go to next cell
        code %= arr.length; //wrap around if necessary
    }
    return null;
}

System.out.println(table.find("a").getName()); //zhangSan
System.out.println(table.find("ct").getName()); //tianQi
```

## 4. 数据的删除

数据删除的 Java 代码实现：



```
// delete element
public Info delete(String key) {
    int code = hashCode(key); //hash the key
    while(arr[code] != null) { //until empty cell,
        if(arr[code].getId() == key) {
            Info temp = arr[code]; //save item
            temp.setName("caonima");
            arr[code].setName(null); //delete item
            return temp; //return item
        }
        ++code; // go to next cell
        code %= arr.length; //wrap around if necessary
    }
    return null;
}
System.out.println(table.delete("a").getName()); //null
```

## 第十七讲 链地址法

### 1. 什么是链地址法

在哈希表每个单元中设置链表。某个数据项的关键字还是像通常映射到哈希表的单元中，而数据项本身插入到单元的链表中。

### 2. 数据的插入

LinkedList 相关类：

```
public class Link {
    public Info info; // data item
    public Link next; // next link in list

    public Link(Info info) {
        this.info = info;
    }
}

public class LinkedList {
    private Link first; // reference to first link on list

    public LinkedList() {
```



```
this.first = null;
}

// insert at start of list
public void insertFirst(Info info) {
    Link newLink = new Link(info);
    newLink.next = first; // newLink --> old first
    first = newLink; // first --> newLink
}

// delete first item
public Link deleteFirst() {
    if(first == null) {
        return null;
    }
    Link temp = first;
    first = first.next; // delete it: first --> old next
    return temp; // return deleted link
}

// find link with given key
public Link find(String key) {
    Link current = first; // start at 'first'
    while(!key.equals(current.info.getId())) {
        if(current.next == null) { // didn't find it
            return null;
        } else { // go to next link
            current = current.next;
        }
    }
    return current;
}

// delete link with given key
public Link delete(String key) {
    Link current = first; // search for expected link
    Link previous = first;
    while(!key.equals(current.info.getId())) {
        if(current.next == null) {
            return null; // didn't find it
        } else {
            previous = current;
            current = current.next; // go to next link
        }
    }
}
```



```
    } // find it
    if(current == first) { // if first link, change first
        first = first.next;
    } else { // otherwise, bypass it
        previous.next = current.next;
    }
    return current;
}
}
```

数据插入的 Java 代码实现:

```
// insert element
public void insert(Info info) {
    String key = info.getId();
    int code = hashCode(key); // hash the key
    if(arr[code] == null) {
        arr[code] = new LinkList();
    }
    arr[code].insertFirst(info);
}

HashTable table = new HashTable();
table.insert(new Info("a", "zhangSan"));
table.insert(new Info("ct", "wangWu"));
```

### 3. 数据的查找

数据查找的 Java 代码实现:

```
// find element with key
public Info find(String key) {
    int code = hashCode(key); //hash the key
    return arr[code].find(key).info;
}

System.out.println(table.find("a").getName()); //zhangSan
System.out.println(table.find("ct").getName()); //tianQi
```

### 4. 数据的删除

数据删除的 Java 代码实现:

```
// delete element
public Info delete(String key) {
    int code = hashCode(key); //hash the key
```

```
    return arr[code].delete(key).info;
}

System.out.println(table.delete("a").getName()); //zhangSan
System.out.println(table.find("a").getName());
//java.lang.NullPointerException
```

## 补充 堆排序

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

堆的定义如下：具有  $n$  个元素的序列  $(h_1, h_2, \dots, h_n)$ ，当且仅当满足  $(h_i \geq h_{2i}, h_i \geq h_{2i+1})$  或  $(h_i \leq h_{2i}, h_i \leq h_{2i+1})$  ( $i=1, 2, \dots, n/2$ ) 时称之为堆。在这里只讨论满足前者条件的堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项（大顶堆）。完全二叉树可以很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。初始时把要排序的数的序列看作是一棵顺序存储的二叉树，调整它们的存储序，使之成为一个堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点交换。然后对前面  $(n-1)$  个数重新调整使之成为堆。依此类推，直到只有两个节点的堆，并对它们作交换，最后得到有  $n$  个节点的有序序列。从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

## 第十八讲 图简介

### 1. 图的基本概念

#### 1) 什么是图

图是一种和树相像的数据结构，通常有一个固定的形状，这是由物理或抽象的问题来决定的。

#### 2) 邻接

如果两个顶点被同一条边连接，就称这两个顶点是邻接的。



### 3) 路径

路径是从一个顶点到另一个顶点经过的边的序列。

### 4) 连通图和非连通图

至少有一条路径可以连接所有的顶点，那么这个图就是连通的，否则是非连通的。

### 5) 有向图和无向图

有向图的边是有方向的，如果只能从 A 到 B，不能从 B 到 A。

无向图的边是没有方向的，可以从 A 到 B，也可以从 B 到 A。

### 6) 带权图

有些图中，边被赋予了一个权值，权值是一个数字，可以代表如两个顶点的物理距离，或者是一个顶点到另一个顶点的时间等等。这样的图叫做带权图。

## 2. 图的 Java 代码实现

Vertex 顶点类：

```
public class Vertex {  
    char label;  
    boolean wasVisited;  
  
    public Vertex(char label) {  
        this.label = label;  
        wasVisited = false;  
    }  
}
```

Graph 图类：

```
public class Graph {  
    private int maxSize = 20;  
    private Vertex[] vertexList; //array of vertices  
    private int[][] adjmat; //adjacency matrix  
    private int nVertex; //current number of vertices  
    private MyStack theStack;  
  
    public Graph() {  
        vertexList = new Vertex[maxSize];  
        adjmat = new int[maxSize][maxSize];  
        nVertex = 0;  
        theStack = new MyStack();  
        for(int i=0; i<maxSize; i++) {  
            for(int j=0; j<maxSize; j++) {  
                adjmat[i][j] = 0;  
            }  
        }  
    }  
}
```

```

public void addVertex(char label) {
    vertexList[nVertex++] = new Vertex(label);
}

public void addEdge(int start, int end) {
    adjmat[start][end] = 1;
    adjmat[end][start] = 1;
}

public void display() {
    for(int i=0; i<nVertex; i++) {
        System.out.println(vertexList[i].label);
    }
}
}

```

测试 Graph 的相关方法：

```

public void testGraph() throws Exception {
    Graph graph = new Graph();
    graph.addVertex('A');
    graph.addVertex('B');
    graph.addVertex('C');
    graph.addVertex('D');
    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 0);
    graph.display(); //A B C D
}

```

## 第十九讲 图的搜索

### 1. 图的搜索

图的搜索是指从一个指定的顶点到达哪些顶点。

有两种常用的方法可以用来搜索图：深度优先搜索（DFS）和广度优先搜索（BFS）。  
深度优先搜索通过栈来实现，而广度优先搜索通过队列来实现。

### 2. 深度优先搜索

#### 1) 深度优先搜索的原则

- ① 如果可能，访问一个邻接的未访问的顶点，标记它，并把它放入栈中。
- ② 当不能执行规则 1 时，如果栈不能空，就从栈中弹出一个顶点。
- ③ 当不能执行规则 1 和规则 2 时，就完成了整个搜搜过程。

## 2) 深度优先搜索的 Java 代码实现

```
//depth-first search
public void dfs() { //begin at vertex 0
    vertexList[0].wasVisited = true; //mark it
    System.out.println(vertexList[0].label);
    theStack.push(0); //push it
    while(!theStack.isEmpty()) { //until stack empty,
        //get an unvisited vertex adjacent to stack top
        int v = getAdjUnvisitedVertex((int) theStack.peek());
        if(v == -1) { //if no such vertex,
            theStack.pop();
        } else { //if it exists,
            vertexList[v].wasVisited = true; //mark it
            System.out.println(vertexList[v].label);
            theStack.push(v); //push it
        }
    }
    //stack is empty, so we're done
    for(int i=0; i<nVertex; i++) { //reset flags
        vertexList[i].wasVisited = false;
    }
}

//returns an unvisited vertex adj to v
public int getAdjUnvisitedVertex(int v) {
    for(int i=0; i<nVertex; i++) {
        if(adjmat[v][i]==1 && vertexList[i].wasVisited == false) {
            return i;
        }
    }
    return -1;
}

graph.dfs(); //A B C D
```

### 3. 广度优先搜索

#### 1) 广度优先搜索的原则

- ① 访问下一个邻接的未访问过的顶点，这个顶点必须是当前节点的邻接点，标记它，并把它插入到队列中。
- ② 如果无法执行规则 1，那么就从队列头取出一个顶点，并使其作为当前顶点。
- ③ 当队列为空不能执行规则 2 时，就完成了整个搜索过程。

#### 2) 广度优先搜索的 Java 代码实现


## 第二十讲 图的最小生成树

### 1. 最小生成树

连接每个顶点最少的连线。最小生成树边的数量总是比顶点的数量少 1。

### 2. 最小生成树的 Java 代码实现

```
//minimum spanning tree (depth first)
public void mst() { //start at 0
    vertexList[0].wasVisited = true; //mark it
    theStack.push(0); //push it
    while (!theStack.isEmpty()) { //until stack empty,
        int currentVertex = (int) theStack.peek();
        //get an unvisited vertex adjacent to stack top
        int v = getAdjUnvisitedVertex(currentVertex);
        if (v == -1) { //if no more neighbors,
            theStack.pop(); //pop it away
        } else { //got a neighbor,
            vertexList[v].wasVisited = true; //mark it
            //display edge from currentVertex to v
            System.out.print(vertexList[currentVertex].label + "-");
            System.out.println(vertexList[v].label);
        }
    }
}
```



```
    theStack.push(v); //push it
}
}
//stack is empty, so we're done
for(int i=0; i<nVertex; i++) { //reset flags
    vertexList[i].wasVisited = false;
}
}
graph.mst(); //A-B    B-C C-D
```