1.1 线性表的基本概念

1、线性表的定义

线性表是具有**相同特征**数据元素的一个**有限序列**。该序列中的所含元素的**个数**即为线性表的长度（n，n>=0）。可以是一个空表

线性表是一种简单的数据结构，以一队学生为例，人数相当于线性表的长度；人数是有限的，对应线性表的有限性；组中的人都是学生，体现了线性表元素的相同特征；线性表可以是有序的，也可以是无序的。

2、线性表的逻辑特征

线性表只有一个**表头**元素，只有一个表尾元素，表头元素没有**前驱**，表尾元素没有**后继**，除了表头表尾之外，其他元素只有一个直接前驱和一个直接后继。

以学生为例，表头和表尾相当于站在队头和队尾的学生，分别只有一位；站在队头的学生前面没有学生，也就是没有前驱；站在队尾的学生后面没有人，也就是没有后继；中间的学生，紧挨着他的前面一个学生和后面一个学生都只有一位，也就是只有一个直接前驱和一个直接后继。

3、线性表的存储结构

线性表的存储结构有**顺序存储结构**和**链式存储结构**两种，前者为**顺序表**，后者为**链表**。

(1) 顺序表（array-based list)

顺序表就是将线性表中的所有元素按照逻辑顺序，依次存储到从指定的存储位置开始的一块**连续**的存储空间中。

顺序表的实现是用**数组**来存储表中的元素，一开始要分配固定长度的数组。而且数组的位置与线性表元素的位置相对应，表中的第i个元素存储在数组的第i个单元中。

顺序表对表中任意一个元素的随机访问相当容易，给出一个位置即可直接获取该位置的元素值，只需要花费**O(1)**的时间。而插入和删除则需要花费**O(n)**的时间。

(2) 链表（linked list)

链表是利用指针实现线性表，它能够按照需要为表中新的元素分配存储空间，是动态的，且不支持随机访问。

链表由一系列叫作表的结点（**node**）的对象组成，每个结点不仅包含所存元素的信息还**包含元素之间逻辑关系的信息**，如单链表中前驱结点包含后继结点的地址信息。链表的结点可以散落在内存的任意位置。建立结点类还有一个好处就是它能够被栈和队列的链接实现方式**重用**。

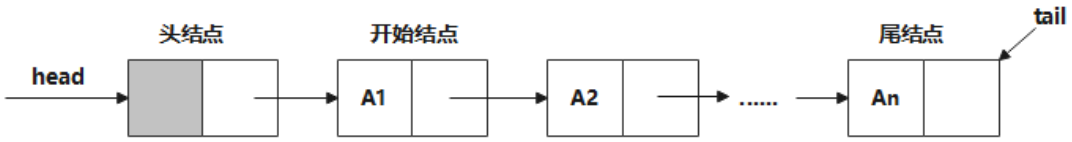
在链表中插入元素无需移动元素。

链表有以下五种形式：

1）单链表

在每个结点中除了包含数据域外还包含一个指针域，用以指向其后继结点。单链表还分为带头结点的单链表和不带头结点的单链表。

1. **带头结点的单链表**中，头指针head指向头结点，头结点的值域不包含任何信息，从头结点的后继结点开始存储信息。头指针时钟不为NULL，`head.next`等于NULL的时候，链表则为空。如下图所示：



2. **不带头结点的单链表**中的头指针head直接指向开始结点，即上图的结点A1，当head等于NULL时，链表为空。

头结点的引入方便我们进行插入和删除操作。

ps:注意区分头指针与头结点

> 头指针：指向链表中的第一个结点，即head指针
>
> 尾指针：指向链表中的最后一个结点，即tail指针
>
> 头结点：带头结点的链表中的第一个结点，只作为链表存在的标志

2）双链表

双链表即在单链表的基础上增加了一个指针域，指向当前结点的前驱，这样可以方便地由其后继来找到其前驱，从而实现输出终端结点到开始结点的数据序列。

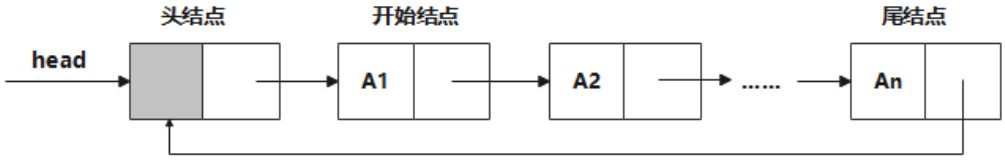
双链表也区分带头结点与不带头结点，上图为带头结点的双链表

3）循环链表

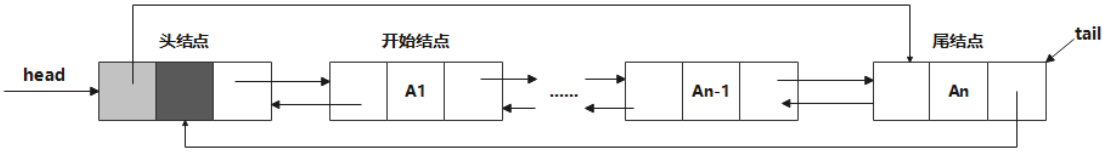
有些应用不需要线性表中有明显的头尾元素，这时候就可以利用循环链表。将链表中最后一个元素的next域中存储为指向线性表中第一个元素的指针，就构成了一个循环链表。在这种实现方式下就不需要尾指针tail了。

没有尾指针可能会使链表的操作陷入死循环，但是也可以利用head指针标记表的处理操作是否周游了整个表。

带头结点的循环单链表如下：

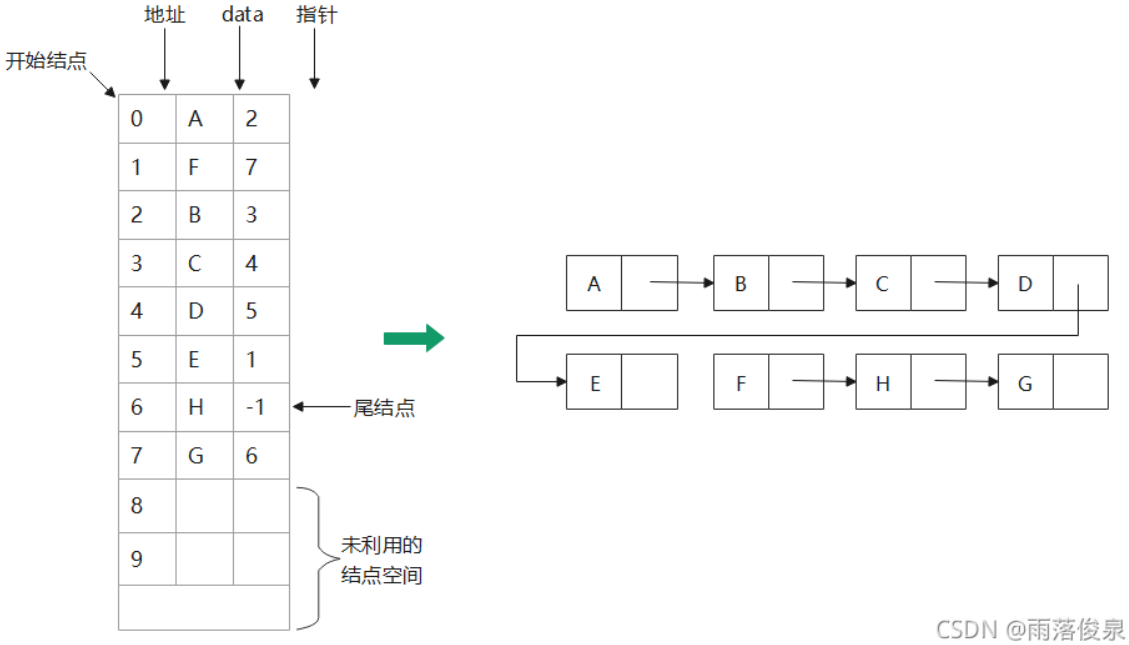


带头结点的循环双链表如下：



4）静态链表

静态链表借助一维数组来表示，如下图所示



4、顺序表与链表的比较

(1) 空间比较

1）存储方式的比较

顺序表的存储空间时一次性分配的，链表的存储空间是多次分配的

2）存储密度

存储密度=结点值域所占的存储量/结点结构所占的存储总量

顺序表=1；链表<1(存在指针域)

(2) 时间比较

1）存取方式

顺序表可以随机存取也可以顺序存取；链表只能顺序存取。所谓顺序存取，以读取为例，要读取某个元素必须遍历其之前的所有元素才能找到它）

2）插入删除时移动的元素个数

顺序表平均需要移动近一半元素；链表不需要移动元素，只需要修改指针

对于具有n个元素的顺序表，插入一个元素所需要的平均移动个数为多少？

> 1、有n+1个插入位置，概率都为p=1/(n+1)
>
> 2、假设将新元素插入在表中第i个元素之后，则需要将第i个元素之后的所有元素都往后移动一个位置，总移动个数为n-i
>
> 3、结合12，可知移动元素个数的数学期望为E=n/2

同理，删除一个元素所需要的平均移动个数为(n-1)/2

所以对于顺序表，插入和删除算法的平均时间复杂度为O(n)

1.2 顺序表的实现

首先依据线性表的一组操作来定义该对象的抽象数据类型（ADT），我利用泛型和接口来进行定义，代码如下：

```
public interface ListADT<E> {
    public void clear();//清空表

    public void insert(E it);//插入元素

    public void append(E it);//从表尾插入元素

    public Object remove();//删除当前位置的值并返回该位置的元素

    public void setFirst(); //将当前位置设置到初始位置

    public void prev();//位置前移

    public void next();//位置后移

    public void setPosition(int position); //设置当前位置

    public void setValue(E it);//设置当前位置的元素值

    public Object currValue(); //获取当前位置的元素值

    public int length();  //获取表实际大小

    public boolean isInList();//判断当前位置是否合规

    public boolean isFull();//判断表是否已经满了

    public boolean isEmpty(); //判断表是否为空
```

```
    public void print();  //打印表
}
```

自定义顺序表，实现代码如下：

```
//自定义一个顺序表
public class SequentialList<E> implements ListADT<E> {
    private static final int DEFAULT_SIZE = 10;//默认大小

    private int maxSize;//表的最大大小
    private int numInList;//表中的实际元素数
    private int curr;//当前元素的位置
    private E[] listArray; //包含所有元素的数组

    private void setUp(int sz) {//初始化方法
        maxSize = sz;
        numInList = curr = 0;
        listArray = (E[]) new Object[sz];
    }


    public SequentialList() {//默认构造
        setUp(DEFAULT_SIZE);
    }

    public SequentialList(int maxSize) {//限制大小的构造
        setUp(maxSize);
    }

    public void clear() {
        numInList = curr = 0;//元素清空
    }

    /*在当前位置插入一个元素，从curr开始的元素全部向后移动一位
       curr上的元素变为插入的元素
     */
    public void insert(E it) {
        if (isFull()) {
            System.out.println("list is full");
            return;//表满
        } else if (curr<0||curr>numInList) {
            System.out.println("bad value for curr");
            return;//当前位置不合规
        } else {
            for (int i = numInList; i > curr; i--) {
                listArray[i] = listArray[i - 1];
            }
            listArray[curr] = it;
            numInList++;
        }
    }

    //在表尾插入一个元素
    public void append(E it) {
        if (isFull()) {
            System.out.println("list is full");
```

```
            return;//表满
        } else {
            listArray[numInList] = it;
            numInList++;
        }
    }

    //删除当前位置的值并返回该位置的元素
    public E remove() {
        if (isEmpty()) {
            System.out.println("list is empty");
            return null;
        } else if (!isInList()) {
            System.out.println("no current element");
            return null;
        } else {
            E it = listArray[curr];
            for (int i = curr; i < numInList - 1; i++) {
                listArray[i] = listArray[i + 1];//元素前移
            }
            numInList--;
            return  it;
        }
    }

    //将当前位置设置到初始位置
    public void setFirst() {
        curr = 0;
    }

    //位置前移
    public void prev() {
        curr--;
    }

    //位置后移
    public void next() {
        curr++;
    }

    //设置当前位置
    public void setPosition(int position) {
        curr = position;
    }

    //设置当前位置的元素值
    public void setValue(E it) {
        if (!isInList()) {
            System.out.println("no current element");
        } else {
            listArray[curr] = it;
        }
    }

    //获取当前位置的元素值
    public E currValue() {
        if (!isInList()) {
            System.out.println("no current element");
```

```
            return null;
        } else {
            return listArray[curr];
        }
    }

    //获取顺序表实际大小
    public int length() {
        return numInList;
    }

    //判断当前位置是否合规
    public boolean isInList() {
        return (curr >= 0 && curr < numInList);
    }

    //判断顺序表是否已经满了
    public boolean isFull() {
        return numInList >= maxSize;
    }

    //判断顺序表是否为空
    public boolean isEmpty() {
        return numInList == 0;
    }

    //打印顺序表
    public void print() {
        if (isEmpty()) {
            System.out.println("empty");
        } else {
            System.out.print("(");
            for (setFirst(); isInList(); next()) {
                System.out.print(currValue() + " ");
            }
            System.out.println(")");
        }
    }
}
```

测试代码如下：

```
public class SequentialListTest {
    public static void main(String[] args) {
        //测试顺序表
        SequentialList<Integer> list = new SequentialList<>();//默认构造最多10个元素
        list.print();//打印空表
        list.insert(1);
        list.insert(4);
        list.insert(2);
        list.insert(5);
        list.insert(3);
        list.insert(0);
        list.insert(2);//插入元素
        System.out.println("after insert: ");
        list.print();
        list.setPosition(2);
```

```
        list.remove();
        System.out.println("delete the third element:");
        list.print();
        list.setFirst();
        System.out.println("set first");
        list.setValue(7);
        System.out.println("change the current element from "+list.currValue()+"
to 7:");
        list.print();
    }
}
```

运行结果如下：

```
empty
after insert:
(2 0 3 5 2 4 1 )
delete the third element:
(2 0 5 2 4 1 )
set first
change the current element from 7 to 7:
(7 0 5 2 4 1 )
```

1.3 单链表的实现

链表是由一系列叫做表的结点（node）的对象组成的，因为结点是一个独立的对象（这个与数组的一个元素相反），所以它能够很好地实现独立的结点类。

下面是结点的完整定义，叫做Link类，Link类中包含一个存储元素值的element域和一个存储表中下一个结点指针的next域。由于在由这种结点建立的链表中每个结点只有一个指向表中下一个结点的指针，所以叫做**单链表(singly linked list)**。

```
public class Link <E> {
    //单链表结点类
    private E element;
    private Link next; //指向的下一个结点
    //构造方法
    public Link(E element, Link next) {
        this.element = element;
        this.next = next;
    }

    public Link(Link next) {
        this.next = next;
    }
    //getter和setter
    public E getElement() {
        return element;
    }

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

```
    public Link getNext() {
        return next;
    }

    public void setNext(Link next) {
        this.next = next;
    }
}
```

单链表的定义如下，该单链表包括头结点：

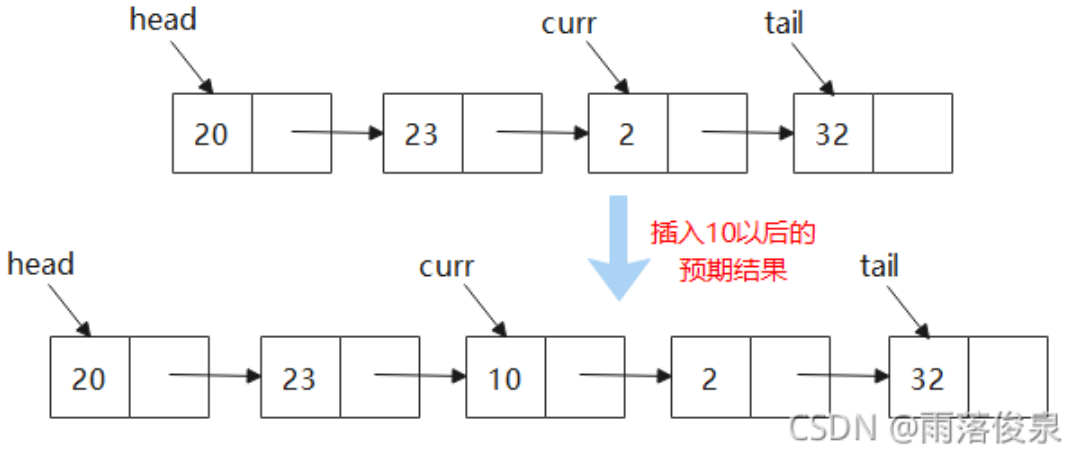
```
public class LinkedList<E> implements ListADT<E> {
    //单链表的定义
    //带有表头结点header node
    private Link<E> head;//头指针
    private Link<E> tail;//尾指针
    protected Link<E> curr;//指向当前元素前驱结点的指针


    private void setUp() {
        tail = head = curr = new Link<>(null);
    }//初始化

    //构造器
    public LinkedList() {
        setUp();
    }

    public LinkedList(int sz) {
        setUp();
    }//忽略size

    @Override
    public void clear() {  //清空所有结点
        head.setNext(null);
        curr = tail = head;
    }

    @Override
    public void insert(E it) {//在当前位置插入元素
        if (curr == null) {
            System.out.println("no current element");
            return;
        } else {
            curr.setNext(new Link(it, curr.getNext()));
            if (tail == curr) {
                tail = curr.getNext();//如果curr是尾部，则tail需要后移
            }

        }

    }

    @Override
    public void append(E it) {//在表的尾部插入元素
        tail.setNext(new Link(it, null));
        tail = tail.getNext();
```

```
    }

    @Override
    public E remove() {//删除并返回当前位置的元素值
        if (!isInList()) return null;
        E it = (E) curr.getNext().getElement();
        if (tail == curr.getNext()) tail = curr;//如果是最后一个元素，则要将尾指针前移
        curr.setNext(curr.getNext().getNext());//移除当前元素
        return it;
    }

    @Override
    public void setFirst() {//将当前位置移到开头
        curr = head;
    }

    @Override
    public void prev() {//将当前位置向前移
        if ((curr == null) || (curr == head)) {
            curr = null;
            return;
        }
        Link<E> temp = head;
        while ((temp != null) && temp.getNext() != curr) {
            temp = temp.getNext();//从头开始找，直到指针域指向curr
        }
        curr = temp;
    }

    @Override
    public void next() {//将当前位置往后移
        if (curr != null) curr = curr.getNext();
    }

    @Override
    public void setPosition(int position) {
        curr = head;
        for (int i = 0; (curr != null) && (i < position); i++) {
            curr = curr.getNext();//从头开始找
        }
    }

    @Override
    public void setValue(E it) {//设置当前位置的元素值
        if (!isInList()) {
            System.out.println("no current element");
            return;
        }
        curr.getNext().setElement(it);

    }

    @Override
    public E currValue() {//获取当前位置的元素值
       if(!isInList())return null;
       return (E) curr.getNext().getElement();
    }
```

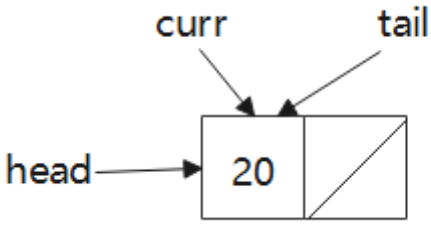
```
    @Override
    public int length() {//获取表的实际大小
        int count=0;
        for (Link temp = head.getNext(); temp!=null ; temp=temp.getNext()) {
            count++;
        }
        return count;
    }

    @Override
    public boolean isInList() {
       return (curr!=null)&&(curr.getNext()!=null);
    }

    @Override
    public boolean isFull() {
        return false;
    }

    @Override
    public boolean isEmpty() {//判断表是否满了
        return head.getNext()==null;
    }

    @Override
    //打印表
    public void print() {
        if (isEmpty()) {
            System.out.println("empty");
        } else {
            System.out.print("(");
            for (setFirst(); isInList(); next()) {
                System.out.print(currValue() + " ");
            }
            System.out.println(")");
        }
    }
}
```

1、curr指针与头结点的说明

如果curr指的是当前结点，当我们想要在这个位置插入一个新的结点，一种代价较大的方法就是从表头开始找直到找到当前结点的前驱结点，另一种更好的办法就是将新元素的值复制到当前结点中，在其后面插入一个新的结点，并将原来结点的值赋给新结点。但是这种curr指向办法不适用于删除表中的最后一个元素，因为不可避免的要改变尾结点的前一个结点（如下图，如果要删除32就要改变2结点）。

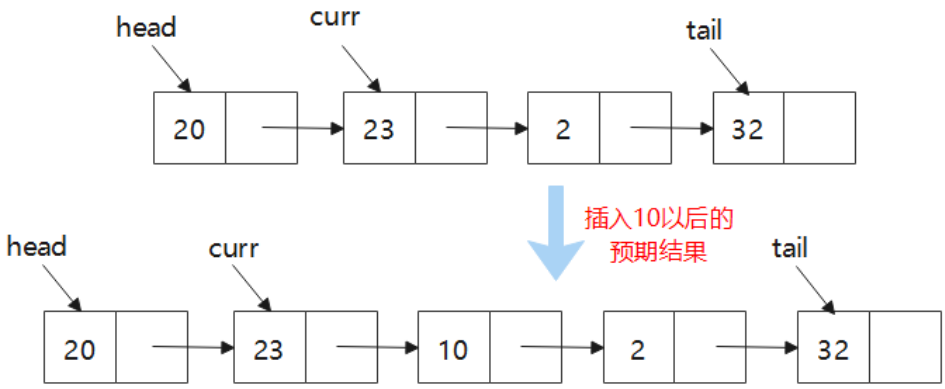


当curr指针指向当前元素的前驱结点，便可以简化插入和删除操作，但是这样会引入一个新问题，如果表中没有结点或者只有一个结点则需要特殊对待。所以我们引入**头结点（header node）**，它是表中的第一个结点，和表中的其他元素一样，只是它的值被忽略，不被看作表中的实际元素。使用头结点初始化单链表的情况如下：



将当前元素定义为所指结点的后继结点也可以帮助我们在表尾插入元素。

使用带有头结点和转义curr指针的插入图示如下，当前结点是值含有2的结点



可以看到，curr指针并没有移动。

2、插入和删除操作的说明

插入

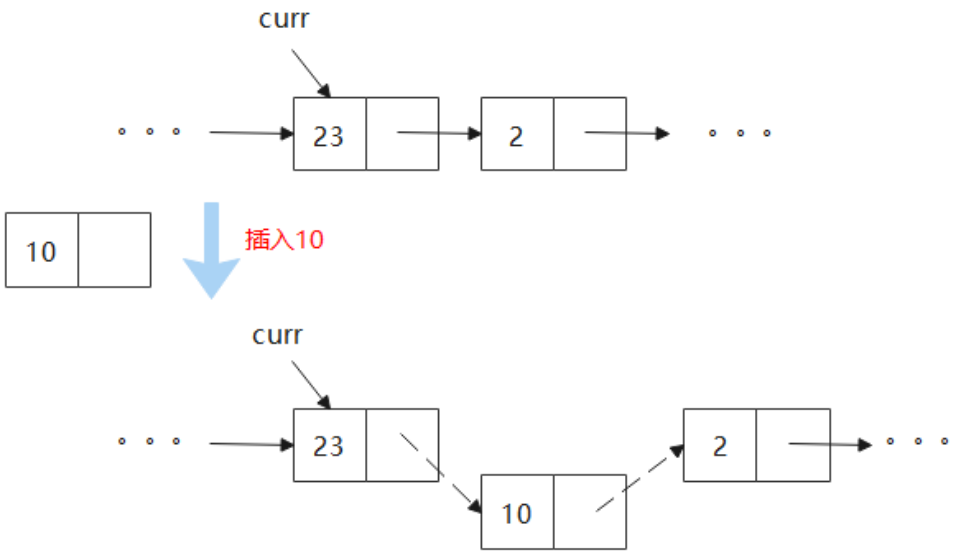
在链表中插入一个新的元素包括三步：

1. 创建新结点并赋予新值
2. 新结点的next域要赋值给当前结点的元素
3. 当前结点元素的前驱的next域要指向新插入的结点

一行代码即可搞定，需要花费O(1)的时间：

```
curr.setNext(new Link(it, curr.getNext()));
```

图解如下：

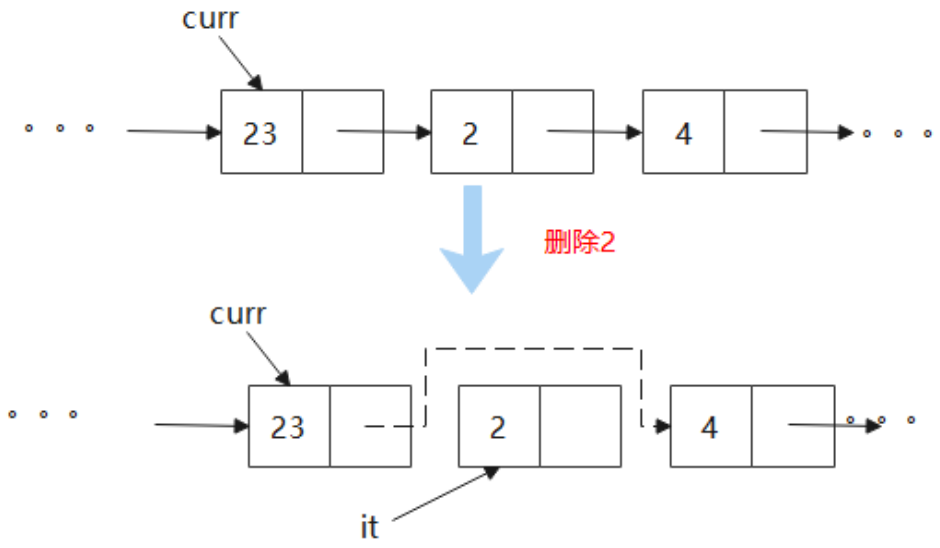


删除

从链表中删除一个结点只需要能够在被删除的结点放一个恰当的指针，改变被删除结点前驱结点的next域即可。同时我们也需要将被删除的结点赋给临时指针it，实现代码如下，需要花费O(1)的时间：

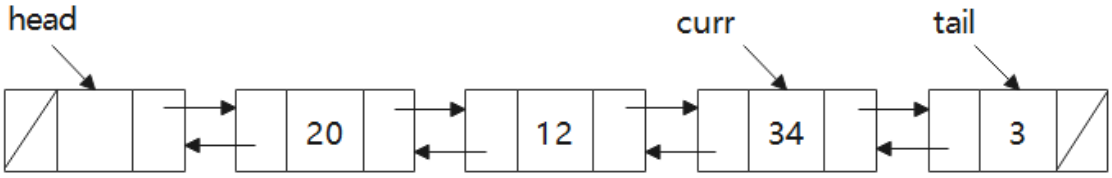
```
curr.setNext(curr.getNext().getNext());//移除当前元素
```

删除图解如下：



1.4 双链表的实现（Java）

双链表（double linked list）可以从一个结点出发，方便地在线性表中访问它的前驱结点和后继结点。双链表存储了两个指针：一个指向它的后继结点，另一个指向了它的前驱结点，结构图如下：



习惯上让curr指向包含当前元素的结点的前驱结点，这样做可以：避免在空表中插入元素的特殊情况；可以将结点插入表中的任意位置。

双链表结点的实现代码如下：

```
public class DLink<E> {
    //双链表结点类
    private E element;
    private DLink prev;//指向的前一个结点
    private DLink next; //指向的下一个结点

    public DLink(E element, DLink prev, DLink next) {
        this.element = element;
        this.prev = prev;
        this.next = next;
    }

    public DLink(DLink prev, DLink next) {
        this.prev = prev;
        this.next = next;
    }

    //getter和setter
    public E getElement() {
        return element;
    }

    public void setElement(E element) {
        this.element = element;
    }

    public DLink getPrev() {
        return prev;
    }

    public void setPrev(DLink prev) {
        this.prev = prev;
    }

    public DLink getNext() {
        return next;
    }

    public void setNext(DLink next) {
        this.next = next;
    }
}
```

双链表的实现代码如下：

```java
public class DoubleLinkedList<E> implements ListADT<E> {
    //双链表的定义
    //带有表头结点header node
    private DLink<E> head;//头指针
    private DLink<E> tail;//尾指针
    protected DLink<E> curr;//指向当前元素前驱结点的指针

    //初始化与构造器
    private void setUp() {
        curr = head = tail = new DLink<E>(null, null);
    }

    public DoubleLinkedList() {
        setUp();
    }

    public DoubleLinkedList(int sz) {
        setUp();
    }

    @Override
    public void clear() {
        head.setNext(null);
        head.setPrev(null);
        curr = tail = head;
    }

    @Override
    public void insert(E it) {//插入元素
        if (curr == null) {
            System.out.println("no current element");
            return;
        }
        curr.setNext(new DLink(it, curr, curr.getNext()));
        //插完以后
        if (curr.getNext().getNext() != null) {
            curr.getNext().getNext().setPrev(curr.getNext());
        }
        if (tail == curr) {
            tail = tail.getNext();
        }//尾部插入

    }

    @Override
    public void append(E it) {//往尾部插入元素
        tail.setNext(new DLink(it, tail, null));
        tail = tail.getNext();
    }

    @Override
    public E remove() {//删除当前位置的元素
        if (!isInList()) {
            System.out.println("no current element");
            return null;
        }
        E it = (E) curr.getNext().getElement();
        if (curr.getNext().getNext() != null) {
```

```
            curr.getNext().getNext().setPrev(curr);
        } else {
            tail = curr;//删除最后一个元素，尾指针前移
        }
        curr.setNext(curr.getNext().getNext());
        return it;
    }

    @Override
    public void setFirst() {
        curr = head;
    }

    @Override
    public void prev() {
        if (curr != null) curr = curr.getPrev();//前移
    }

    @Override
    public void next() {
        if (curr != null) curr = curr.getNext();//后移
    }

    @Override
    public void setPosition(int position) {
        curr = head;
        for (int i = 0; (curr != null) && (i < position); i++) {
            curr = curr.getNext();//从头开始找
        }
    }

    @Override
    public void setValue(E it) {
        if (!isInList()) {
            System.out.println("no current element");
            return;
        }
        curr.getNext().setElement(it);
    }

    @Override
    public E currValue() {
        if (!isInList()) return null;
        return (E) curr.getNext().getElement();
    }

    @Override
    public int length() {
        int count = 0;
        for (DLink temp = head.getNext(); temp != null; temp = temp.getNext()) {
            count++;
        }
        return count;
    }

    @Override
    public boolean isInList() {
        return (curr != null) && (curr.getNext() != null);
```

```
    }

    @Override
    public boolean isFull() {
        return false;
    }

    @Override
    public boolean isEmpty() {
        return head.getNext() == null;
    }

    @Override
    public void print() {
        if (isEmpty()) {
            System.out.println("empty");
        } else {
            System.out.print("(");
            for (setFirst(); isInList(); next()) {
                System.out.print(currValue() + " ");
            }
            System.out.println(")");
        }
    }
}
```

插入与删除的说明

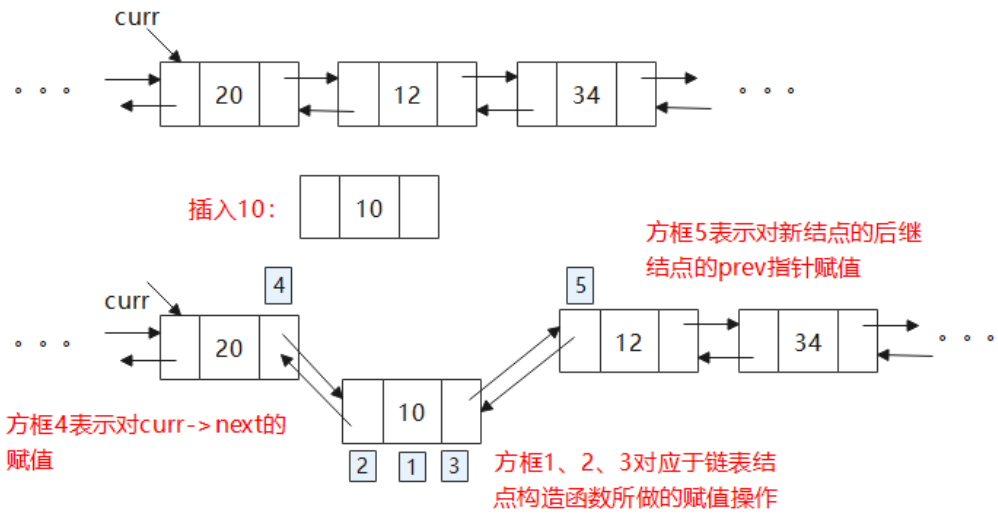
插入（头插法）

在双链表中，插入一个元素的核心代码如下：

```
curr.setNext(new DLink(it, curr, curr.getNext()));
if (curr.getNext().getNext() != null) {
    curr.getNext().getNext().setPrev(curr.getNext());
}
if(tail==curr){
    tail=tail.next();
}
```

- 新建一个结点，新结点的前驱指向curr，后继指向curr的下一个结点
- 然后将curr的下一个结点设置为新结点
- 如果curr的下下个结点不为空，则将下下个结点的前驱指向新结点
- 如果curr指向了tail指向的结点，则要将尾指针后移

图解如下：



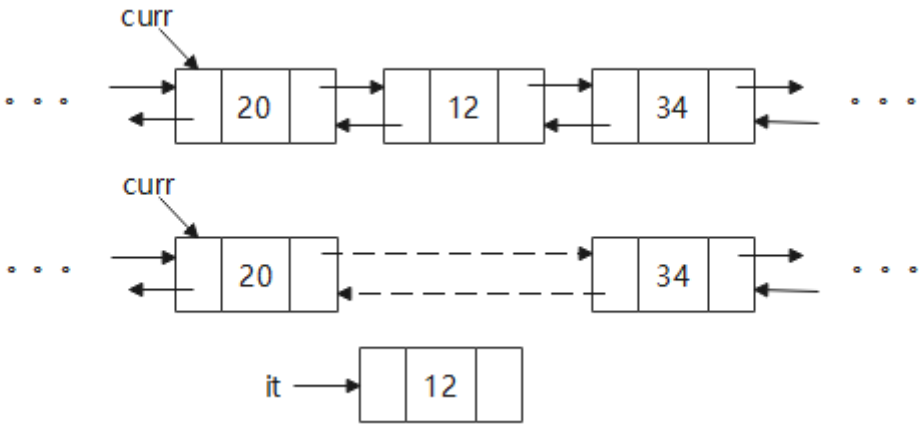
删除

在双链表中，删除当前位置的元素的核心代码如下：

```
if (curr.getNext().getNext() != null) {
    curr.getNext().getNext().setPrev(curr);
} else {
        tail = curr;//删除最后一个元素，尾指针前移
}
curr.setNext(curr.getNext().getNext());
```

- 如果被删除结点存在后继结点，则将其后继结点的prev域设置为curr
- 如果不存在后继结点，则调整tail，使其指向被删除结点的前驱结点
- 最后修改curr的next域

图解如下：



1.5 逆置问题

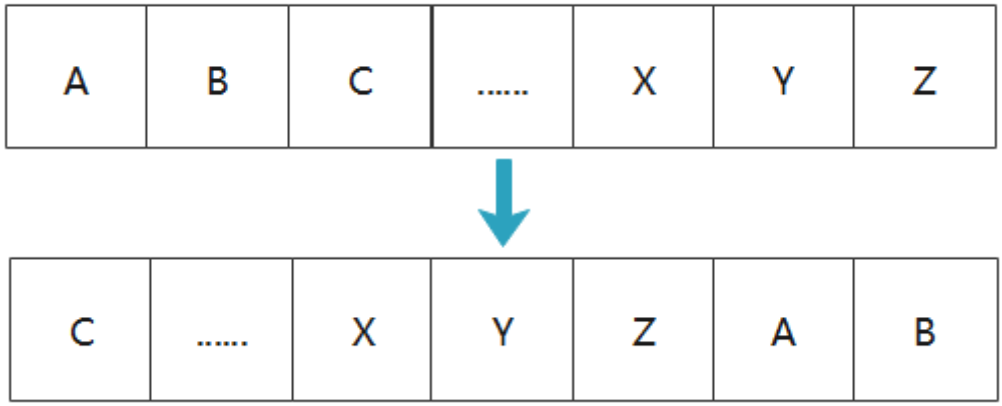
给定一个线性表，如何将其中的元素逆置？

可以设置两个整型变量i和j，i指向第一个元素，j指向最后一个元素，边交换i和j所指元素，边让i和j相向而行，直到二者相遇。假设元素存在数组a[]中，left和right是数组两端元素的下标。代码如下：

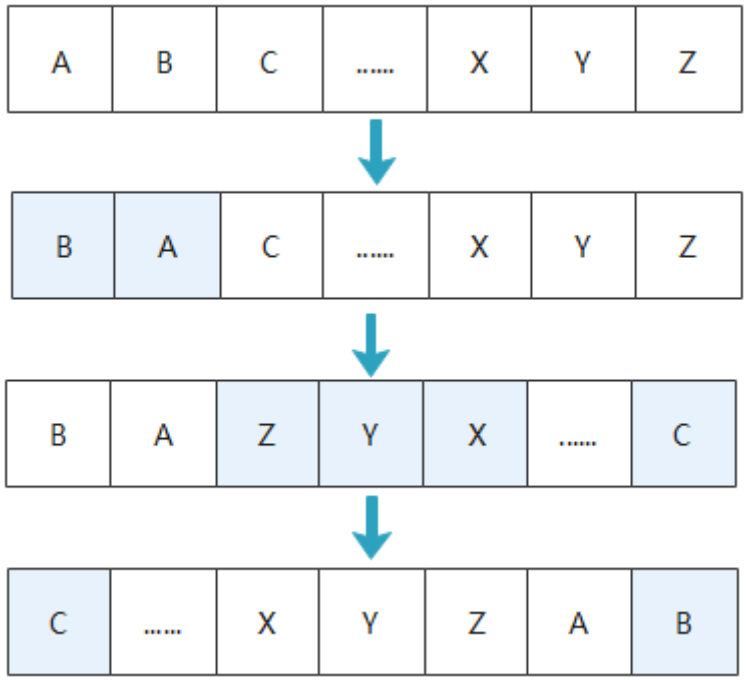
```
for(int i = left,j=right;i<j;++i,--j){
    temp=a[i];
    a[i]=a[j];
    a[j]=temp;
}
```

举例说明：

将一长度为n的数组循环左移p个位置，如下图所示



思路：只需要先将0~p-1位置的元素逆置，再将p~n-1位置的元素逆置，然后再将整个数组逆置即可




```
public class ReverseTest {
    //将一长度为n的数组循环左移p个位置
    void Reverse(int[] a, int left, int right, int k) {
        int temp;
        for (int i = left, j = right; i < left + k && i < j; i++, j--) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
```

```
    public static void main(String[] args) {
        int[] a = new int[]{1,2,3,4,5,6,7,8,9};
        System.out.print("before reverse: ");
        for (int i = 0; i <9 ; i++) {
            System.out.print(a[i]+" ");
        }
        System.out.println();
        ReverseTest  test = new ReverseTest();
        //循环左移三位
        test.Reverse(a,0,2,3);
        test.Reverse(a,3,8,6);
        test.Reverse(a,0,8,9);
        System.out.print("after reverse: ");
        for (int i = 0; i <9 ; i++) {
            System.out.print(a[i]+" ");
        }
    }
}
```

运行结果如下：

```
before reverse: 1 2 3 4 5 6 7 8 9
after reverse: 4 5 6 7 8 9 1 2 3
```

2.1 栈和队列的基本概念

2.1.1 栈的基本概念

1、栈的定义

栈是一种只能在一端进行插入或者删除操作的线性表。其中允许进行插入和删除的一段被称为**栈顶（Top）**。表的另一端称为栈底。栈底是固定不变的，元素插入栈称为**压栈（push）**，删除元素称为**出栈（pop）**。

2、栈的特点

栈的主要特点是先入后出（FILO)(FRIST IN LAST OUT)，就好像将东西压进一个箱子里，只有当上面的东西取出以后才能将最里面的东西取出。

3、栈的存储结构

栈的本质仍是线性表，所以也分为**顺序栈和链式栈。**

4、栈的数学性质

当n个元素以某种顺序进栈，并且可以在任意时刻出栈（满足先入先出的前提下）时，所获得的元素排列的数目N恰好为Catalan数，即

$$N = \frac{1}{n+1}C_{2n}^{n}$$

2.1.2 队列的基本概念

1、队列的定义

队列简称队，它也是一种操作受限的**线性表**，其限制为仅允许在表的一端进行插入，在表的另一端进行删除。可进行插入的一端称为**队尾（rear）**，可进行删除的一端称为**队头（front）**。向队列中插入新元素称为**进队**，新元素进队后就成为新的队尾元素；从队列中删除元素称为**出队**，元素出队以后，其后继元素就成为新的队头元素。

2、队的特点

队列的特点就是**先进先出（FIFO)**(FIRST IN FIRST OUT)。就像现实中的排队，先来排队的人就能先去办理业务。

3、队列的存储结构

队列按存储结构可以分为**顺序队与链队**

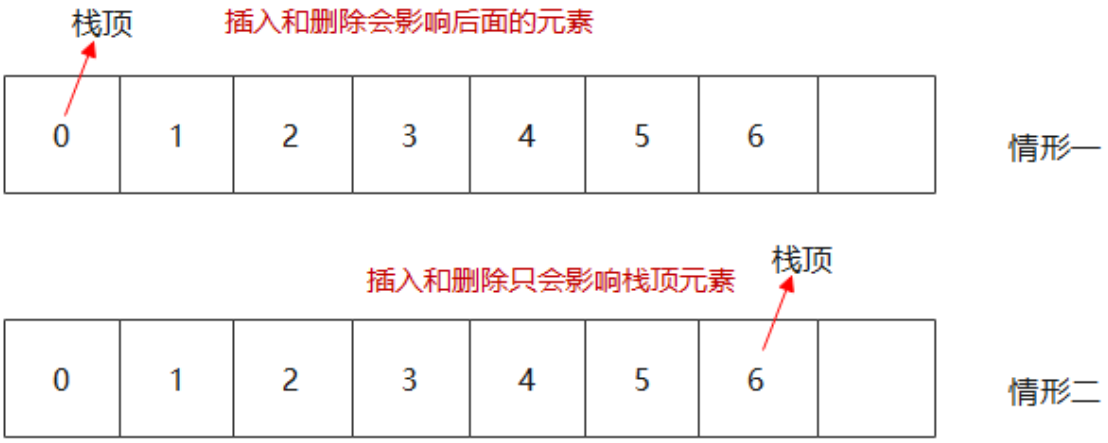
2.2 顺序栈和链栈的实现（Java）

首先定义栈的ADT（抽象数据类型），定义如下：

```
public interface StackADT<E> {
    public void clear();//清空栈中所有元素
    public void push(E it);//压栈
    public E pop();//出栈
    public E topValue();//返回栈顶元素
    public boolean isEmpty();//判断栈是否空
    public void print();//打印栈中的所有元素
}
```

2.2.1 顺序栈的实现

顺序栈的实现有两种，一种是将数组的第0个位置作为栈顶，此时的插入和删除操作的时间代价都为O（n）；另一种是将有n个元素的数组的n-1个位置作为栈顶，插入和删除都是在表尾进行操作，此时的时间复杂度为O(1)。图示如下，显然第二种方式较好。



具体的实现代码如下：

```
public class AStack<E> implements StackADT<E> {
    //构造顺序栈
    private final int DEFAULT_SIZE = 10;
    private int size;//栈最多能容纳的元素个数
    private int top;//可插入位置的下标
    private E[] listArray;//存储栈中元素的数组

    //初始化
```

```
    private void setUp(int size) {
        this.size = size;
        top = 0;
        listArray = (E[]) new Object[size];
    }

    //构造器
    public AStack() {
        setUp(DEFAULT_SIZE);
    }

    public AStack(int size) {
        setUp(size);
    }

    @Override
    public void clear() {
        top = 0;
    }

    @Override
    public void push(E it) {
        if (top >= size) {
            System.out.println("stack overflow");
            return;
        }
        listArray[top++] = it;
    }

    @Override
    public E pop() {
        if (isEmpty()){return null;}
        return listArray[--top];//top-1才是存储最顶元素的位置
    }

    @Override
    public E topValue() {
        if (isEmpty()){return null;}
        return listArray[top-1];
    }

    @Override
    public boolean isEmpty() {
        return top==0;
    }

    @Override
    public void print() {
        if (isEmpty()){
            System.out.println("empty");
        }
        for (int i=top-1;i>=0;i--){
            System.out.println(listArray[i]);
        }
    }
}
```

在以上代码中，top定义为栈中的第一个空闲位置，push和pop都是在top指示的位置进行插入和删除。push操作先将元素插入到top所在位置，然后对top加一；pop首先将top减一，top-1下标所在的元素才是真正的栈顶元素，然后删除栈顶元素。

top同样可以定义为栈中最上面元素的值，这样的话top需要初始化为-1.

测试代码如下

```
public class AStackTest {
    public static void main(String[] args) {
        AStack<Integer> myAStack=new AStack<>();
        myAStack.push(1);
        myAStack.push(2);
        myAStack.push(3);
        myAStack.push(4);
        myAStack.push(5);
        System.out.println("after pushing, the stack is: ");
        myAStack.print();
        System.out.print("top value: ");
        System.out.println(myAStack.topValue());
        System.out.println("after popping two elements: ");
        myAStack.pop();
        myAStack.pop();
        myAStack.print();
        System.out.print("top value: ");
        System.out.println(myAStack.topValue());
    }
}
```

测试结果如下

```
after pushing, the stack is:
5
4
3
2
1
top value: 5
after popping two elements:
3
2
1
top value: 3
```

2.2.2 链栈的实现

链栈的实现比链表的实现简单得多，唯一一个数据成员就是top，它是一个指向链式栈第一个结点（栈顶）的指针

为方便阅读，Link类的定义如下：

```
public class Link <E> {
    //结点类
    private E element;
    private Link next; //指向的下一个结点
    //构造方法
```

```
    public Link(E element, Link next) {
        this.element = element;
        this.next = next;
    }

    public Link(Link next) {
        this.next = next;
    }
    //getter和setter
    public E getElement() {
        return element;
    }

    public void setElement(E element) {
        this.element = element;
    }

    public Link getNext() {
        return next;
    }

    public void setNext(Link next) {
        this.next = next;
    }
}
```

LStack的实现代码如下：

```
public class LStack<E> implements StackADT<E> {
    private Link<E> top;//栈顶元素

    //初始化与构造器
    private void setUp() {
        top = null;
    }

    public LStack() {
        setUp();
    }

    public LStack(int size) {
        setUp();
    }

    @Override
    public void clear() {
        top = null;
    }

    @Override
    public void push(E it) {
        top = new Link<E>(it, top);
    }

    @Override
    public E pop() {
        if (isEmpty()) {
```

```
            System.out.println("stack is empty");
            return null;
        }
        E it = top.getElement();
        top = top.getNext();
        return it;
    }

    @Override
    public E topValue() {
        if (isEmpty()) {
            System.out.println("no top value");
            return null;
        }
        return top.getElement();
    }

    @Override
    public boolean isEmpty() {
        return top == null;
    }

    @Override
    public void print() {
        if (isEmpty()) {
            System.out.println("empty");
            return;
        }
        Link<E> temp = top;
        while (temp != null) {
            System.out.println(temp.getElement());
            temp = temp.getNext();
        }
    }
}
```

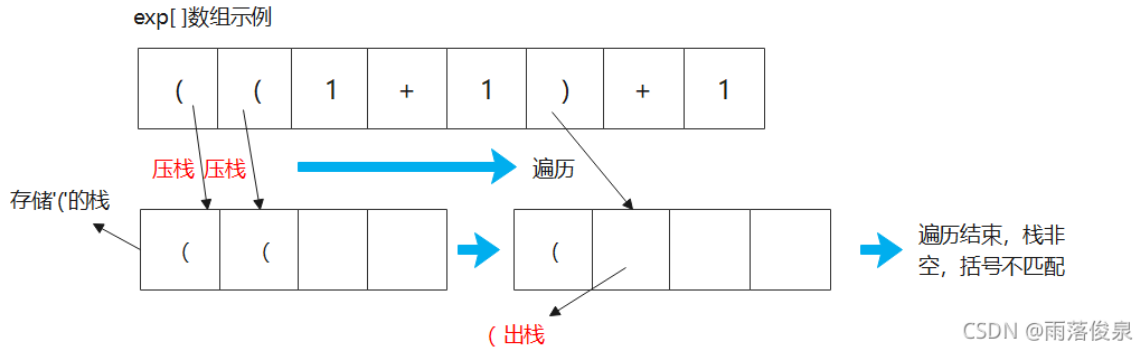
2.3 栈的应用——括号匹配与后缀表达式

在解决问题中出现了一个子问题，但是仅凭现有条件无法进行解决，可以将其记下，等待之后出现可以解决它的条件以后再返回来解决。这种问题就可以利用栈的压栈操作来进行记忆，这是栈的FILO特性所衍生出来的。

1、括号匹配问题

> 题目：编写一个算法，判断一个表达式中的括号是否正确配对，假设表达式中只有小括号，表达式已经存入ecp[]中，表达式中的字符个数为n。

思路：从头遍历表达式数组，当遇见'('时，将其压入准备的栈中等候处理，如果遇到')'，若此时栈空，则不匹配，否则将栈中的'('中弹出一个；当遍历完成以后，如果存储'('的栈为空，说明所有的括号都匹配成功了，否则不匹配。图解如下：



代码实现如下

```
public class BracketMatching {
    //利用栈完成括号匹配
    boolean match(char[]exp){
        char stack[] = new char[exp.length];
        int top=-1;
        for (int i=0;i<exp.length;i++){
            if (exp[i]=='('){
                stack[++top]='(';//遇到'('则入栈
            }
            if (exp[i]==')'){
                if (top==-1){
                    return false;//栈空说明')'比'('多，不匹配
                }else {
                    --top;//栈不空则将栈中的一个'('弹出
                }
            }
        }
        if (top==-1){
            return true;//栈空则说明所有括号都被处理掉
        }
        return false;//否则括号不匹配
    }

    public static void main(String[] args) {
        BracketMatching test = new BracketMatching();
        String string = "((1+1)+(2+1)))";
        char[] exp=string.toCharArray();
        System.out.println(string+"括号匹配: "+test.match(exp));
    }
}
```

运行结果

```
((1+1)+(2+1)))括号匹配: false
```

2、前缀、中缀、后缀表达式

算术表达式分为前缀式、中缀式、后缀式。

(1) 前缀表达式（波兰表达式）

1)前缀表达式又称波兰式，前缀表达式的运算符位于操作数之前

2)举例说明： (3+4)×5-6 对应的前缀表达式就是 - × + 3 4 5 6

前缀表达式的计算机求值

从右至左扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（栈顶元素 和 次顶元素），并将结果入栈；重复上述过程直到表达式最左端，最后运算得出的值即为表达式的结果

[1] 例如: (3+4)×5-6 对应的前缀表达式就是 **- × + 3 4 5 6 ,** 针对前缀表达式求值步骤如下

> 1. 从右至左扫描，将6、5、4、3压入堆栈
> 2. 遇到+运算符，因此弹出3和4（3为栈顶元素，4为次顶元素），计算出3+4的值，得7，再将7入栈
> 3. 接下来是×运算符，因此弹出7和5，计算出7×5=35，将35入栈
> 4. 最后是-运算符，计算出35-6的值，即29，由此得出最终结果

(2) 中缀表达式

1)中缀表达式就是**常见的运算表达式**，如(3+4)×5-6

2)中缀表达式的求值是我们人最熟悉的，但是对计算机来说却不好操作，因此，在计算结果时，往往会将中缀表达式转成其它表达式来操作(一般转成后缀表达式.)

(3) 后缀表达式

1)后缀表达式又称**逆波兰表达式**,与前缀表达式相似，只是运算符位于操作数之后

2)中举例说明： (3+4)×5-6 对应的后缀表达式就是3 4 + 5 × 6 –

后缀表达式的计算机求值

从左至右扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（次顶元素 和 栈顶元素），并将结果入栈；重复上述过程直到表达式最右端，最后运算得出的值即为表达式的结果

[2] 例如: (3+4)×5-6 对应的后缀表达式就是 **3 4 + 5 × 6 - ,** 针对后缀表达式求值步骤如下

> 1. 从左至右扫描，将3和4压入堆栈；
> 2. 遇到+运算符，因此弹出4和3（4为栈顶元素，3为次顶元素），计算出3+4的值，得7，再将7入栈；
> 3. 将5入栈；
> 4. 接下来是×运算符，因此弹出5和7，计算出7×5=35，将35入栈；
> 5. 将6入栈；
> 6. 最后是-运算符，计算出35-6的值，即29，由此得出最终结果

示例代码如下

```
public class Postfix_Expression_Calculation {
    //利用栈对后缀表达式求值
    //简化起见，每个数字只有一位,且假设表达式都能正常计算
    static int calculate(String expS) {
        Stack<Integer> stack = new Stack<>();//存储数字的栈
        char[] exp = expS.toCharArray();
        for (int i = 0; i < exp.length; i++) {
            if (exp[i] >= '0' && exp[i] <= '9') {//遇到数字则压栈
```

```
                stack.push(exp[i] - '0');
            } else if (exp[i] != ' ') {
                int num2 = stack.pop();
                int num1 = stack.pop();//弹出栈顶的两个元素

                int temp = -1;//存储中间结果
                switch (exp[i]) {
                    case '+':
                        temp = num1 + num2;
                        break;
                    case '-':
                        temp = num1 - num2;
                        break;
                    case '×':
                        temp = num1 * num2;
                        break;
                    case '/':
                            temp = num1 / num2;
                        break;
                }
                stack.push(temp);//将中间结果压栈
            }
        }
        return stack.peek();//返回栈顶元素
    }

    public static void main(String[] args) {
        String exp = "1234×++2/";
        int result = calculate(exp);
        System.out.println("the result of Postfix_Expression:" + exp + "is " +
result);
    }
}
```

运算结果如下：

```
the result of Postfix_Expression:1234×++2/is 7
```

(4) 中缀表达式转化为后缀表达式

在开发中，我们需要将中缀表达式转化为后缀表达式，算法如下

[1] 初始化两个栈：运算符栈s1和储存中间结果的栈s2；

[2] 从左至右扫描中缀表达式；

[3] 遇到操作数时，将其压s2；

[4] 遇到运算符时，比较其与s1栈顶运算符的优先级：

1. 如果s1为空，或栈顶运算符为左括号“(”，则直接将此运算符入栈；
2. 否则，若优先级比栈顶运算符的高，也将运算符压入s1；
3. 否则，将s1栈顶的运算符弹出并压入到s2中，再次转到(4-1)与s1中新的栈顶运算符相比较；

[5] 遇到括号时：

1. 如果是左括号“(”，则直接压入s1

2. 如果是右括号“)”，则依次弹出s1栈顶的运算符，并压入s2，直到遇到左括号为止，此时将这一对括号丢弃

6 重复步骤2至5，直到表达式的最右边

7 将s1中剩余的运算符依次弹出并压入s2

8 依次弹出s2中的元素并输出，**结果的逆序即为中缀表达式对应的后缀表达式**

举例说明 将中缀表达式1+((2+3)×4)-5转换为后缀表达式

扫描到的元素	s2(栈底->栈顶)	s1 (栈底->栈顶)	说明
1	1	空	数字，直接入栈
+	1	+	s1为空，运算符直接入栈
(	1	+ (	左括号，直接入栈
(	1	+ ((	同上
2	1 2	+ ((	数字
+	1 2	+ ((+	s1栈顶为左括号，运算符直接入栈
3	1 2 3	+ ((+	数字
)	1 2 3 +	+ (	右括号，弹出运算符直至遇到左括号
×	1 2 3 +	+ (×	s1栈顶为左括号，运算符直接入栈
4	1 2 3 + 4	+ (×	数字
)	1 2 3 + 4 ×	+	右括号，弹出运算符直至遇到左括号
-	1 2 3 + 4 × +	-	-与+优先级相同，因此弹出+，再压入-
5	1 2 3 + 4 × + 5	-	数字
到达最右端	**1 2 3 + 4 × + 5 -**	空	s1中剩余的运算符

代码实现:

```
public class Nifix_to_Postfix {
    //输入一个中缀表达式，将其转换为后缀表达式
    //简化起见，每个数字只有一位,且假设表达式都能正常计算
    static String change(String nifix) {
        Stack<Character> s1 = new Stack<>();//存储运算符的栈
        Stack<Character> s2 = new Stack<>();//存储中间结果的栈
        char[] nifixExp = nifix.toCharArray();//将字符串转化为字符数组
        for (int i = 0; i < nifixExp.length; i++) {//从左到右扫描
            if (nifixExp[i] >= '0' && nifixExp[i] <= '9') {
                s2.push(nifixExp[i]);//遇到数字则压入s2
            } else if (nifixExp[i] == '(') {
                s1.push(nifixExp[i]);//左括号直接压入s1
            } else if (nifixExp[i] == ')') {
                while (s1.peek()!='('){
                    s2.push(s1.pop());
                }
                s1.pop();//把左括号也弹出
```

```
            } else {
                /**
                 * 遇到运算符时，比较其与s1栈顶运算符的优先级：
                 * 1) 如果s1为空，或栈顶运算符为左括号“(”，则直接将此运算符入栈；
                 * 2) 否则，若优先级比栈顶运算符的高，也将运算符压入s1；
                 * 3) 否则，将s1栈顶的运算符弹出并压入到s2中，再次转到(1)与s1中新的栈顶运算
符相比较；
                 */
                while (true) {
                    if (s1.empty() || s1.peek() == '('
                            || comparePriority(nifixExp[i], s1.peek())) {
                        s1.push(nifixExp[i]);
                        break;
                    } else {
                        s2.push(s1.pop());
                    }
                }
            }
        }
        //把s1中剩余的运算符一次弹出并压入s2
        while(!s1.empty()){
            s2.push(s1.pop());
        }
        //依次弹出s2中的元素并输出，
        // 结果的逆序即为中缀表达式对应的后缀表达式
        char [] temp=new char[s2.size()];
        for (int i = 0; !s2.empty() ; i++) {
            temp[i]=s2.pop();
        }
        String string = "";
        for (int i = temp.length-1; i >=0 ; i--) {
            string+=temp[i];
        }
        return string;
    }

    private static boolean comparePriority(char c1, char c2) {//比较运算符优先级
        int priority1 = (c1 == '+' || c1 == '-' ? 1 : 2);
        int priority2 = (c2 == '+' || c1 == '-' ? 1 : 2);
        if (priority1 - priority2 > 0) {
            return true;//c1优先级较高
        }
        return false;
    }

    public static void main(String[] args) {
        String test = "(1+2+3x4)/3";
        System.out.println("\"" +test+"\" after change: \""+change(test)+"\"");
        System.out.println("the result of Postfix_Expression:
\""+change(test)+"\" is "
                +Postfix_Expression_Calculation.calculate(change(test)));
    }
}
```

运行结果如下：

```
"(1+2+3×4)/3" after change: "12+34×+3/"
the result of Postfix_Expression: "12+34×+3/" is 5
```

3、综合应用——求布尔表达式的值

西交的软件朋友们，此题仅供思路参考，千万不要复制粘贴哦

题目：

> 本题是要计算类似如下的布尔表达式：(T |T)& F &(F|T)，其中 T 表示 True，F 表示 False。表 达式可以包含如下运算符：！表示 not（非），&表示 and（与），|表示 or（或），^表示 xor（异或），并允许使用括号。 为了执行表达式的运算，要考虑运算符的优先级，以上逻辑运算符的优先级由高到低为： not、and、xor、or。括号运算符的优先级高于逻辑运算符。当运算符相同时，遵循左结合。表达 式的计算最终结果只能是 T 或 F。 对输入的表达式的要求如下：
>
> 1）一个表达式不超过 100 个字符，字符间可以用任意个空格分开，或者根本没有空格，所以 表达式总的长度也就是字符的个数，它是未知的。
>
> 2）要能处理表达式中出现括号不匹配、运算符缺少运算操作数等常见的输入错误。

```
public class BoolCalculate {
    //计算布尔表达式的值
    //利用自定义的链栈
    static char boolCalculate(String expS) throws Exception {//主函数
        LStack<Character> stack = new LStack<>();//存储数字的栈
        char[] exp = expS.toCharArray();
        for (int i = 0; i < exp.length; i++) {
            if (exp[i] == 'T' || exp[i] == 'F') {//遇到T\F则压栈
                stack.push(exp[i]);
            } else if (exp[i] != ' ') {
                char char1 = stack.pop();
                char temp;
                if (exp[i] == '!') {//取非运算符只需要一个元素
                    temp = logicCalculate(exp[i], char1, ' ');//存储中间运算结果
                } else {
                    char char2 = stack.pop();//弹出栈顶的第二个元素
                    temp = logicCalculate(exp[i], char1, char2);//根据运算符进行逻辑运算，存储中间运算结果
                }

                stack.push(temp);//将中间结果压栈
            }
        }

        return stack.topValue();
    }

    static char logicCalculate(char operator, char char1, char char2) {//逻辑运算
        switch (operator) {
            case '!':  //非运算
                if (char1 == 'T') return 'F';
                else return 'T';
            case '&': //与运算
                if (char1 == 'T' && char2 == 'T') return 'T';
                else return 'F';
            case '|': //或运算
                if (char1 == 'F' && char2 == 'F') return 'F';
```

```
                else return 'T';
            case '^': //异或运算
                if ((char1 == 'T' && char2 == 'T') || (char1 == 'F' && char2 ==
'F')) return 'F';
                else return 'T';
            default:
                return ' ';
        }
    }

    //中缀转后缀
    static String toPostfix(String nifix) throws Exception {
        LStack<Character> s1 = new LStack<>();//存储运算符的栈
        LStack<Character> s2 = new LStack<>();//存储中间结果的栈
        char[] nifixExp = nifix.toCharArray();//将字符串转化为字符数组
        for (int i = 0; i < nifix.length(); i++) {
            if (nifixExp[i] != 'F' && nifixExp[i] != 'T' && nifixExp[i] != '!'
                    && nifixExp[i] != '&' && nifixExp[i] != '|' && nifixExp[i]
!= '^'
                    && nifixExp[i] != ' ' && nifixExp[i] != '(' && nifixExp[i]
!= ')') {
                throw new Exception("输入符号有问题");
            }
        }
        if (!BracketMatching.match(nifixExp)) {
            throw new Exception("括号不匹配");
        } else {
            for (int i = 0; i < nifixExp.length; i++) {//从左到右扫描
                if (nifixExp[i] == 'T' || nifixExp[i] == 'F') {
                    s2.push(nifixExp[i]);//遇到T/F则压入s2
                } else if (nifixExp[i] == '(') {
                    s1.push(nifixExp[i]);//左括号直接压入s1
                } else if (nifixExp[i] == ')') {
                    while (s1.topValue() != '(') {
                        s2.push(s1.pop());
                    }
                    s1.pop();//把左括号也弹出
                } else if (nifixExp[i] != ' ') {
                    /**
                     * 遇到运算符时，比较其与s1栈顶运算符的优先级：
                     * 1) 如果s1为空，或栈顶运算符为左括号“(”，则直接将此运算符入栈；
                     * 2) 否则，若优先级比栈顶运算符的高，也将运算符压入s1；
                     * 3) 否则，将s1栈顶的运算符弹出并压入到s2中，再次转到(1)与s1中新的栈
顶运算符相比较；
                     */
                    while (true) {
                        if (s1.isEmpty() || s1.topValue() == '('
                                || comparePriority(nifixExp[i], s1.topValue()))
{
                            s1.push(nifixExp[i]);
                            break;
                        } else {
                            s2.push(s1.pop());
                        }
                    }
                }
            }
        }
```

```
        //把s1中剩余的运算符一次弹出并压入s2
        while (!s1.isEmpty()) {
            s2.push(s1.pop());
        }
        //依次弹出s2中的元素并输出，
        // 结果的逆序即为中缀表达式对应的后缀表达式
        char[] temp = new char[s2.size()];
        for (int i = 0; !s2.isEmpty(); i++) {
            temp[i] = s2.pop();
        }
        String string = "";
        for (int i = temp.length - 1; i >= 0; i--) {
            string += temp[i];
        }
        return string;

    }

    private static boolean comparePriority(char c1, char c2) {//比较运算符优先级
        int priority1 = 0, priority2 = 0;
        switch (c1) {
            case '!':
                priority1 = 4;
                break;
            case '&':
                priority1 = 3;
                break;
            case '^':
                priority1 = 2;
                break;
            case '|':
                priority1 = 1;
                break;
        }
        switch (c2) {
            case '!':
                priority2 = 4;
                break;
            case '&':
                priority2 = 3;
                break;
            case '^':
                priority2 = 2;
                break;
            case '|':
                priority2 = 1;
                break;
        }
        return priority1 - priority2 > 0;//c1优先级较高
    }

    public static void main(String[] args) throws Exception {
        System.out.print("please input your nifix expression: ");
        Scanner in = new Scanner(System.in);
        String input = in.nextLine();//next()方法不能带空格
        char result = boolCalculate(toPostfix(input));
        System.out.println("the result of: " + input + " is " + result);
    }
```

```
}
```

运行结果如下：

```
please input your nifix expression: (T |T)& F &(F|T)
the result of: (T |T)& F &(F|T) is F
```

2.4 顺序队和链队的实现(Java)

同栈一样，队列也是一种受限的线性表。队列元素只能从队尾插入，称为**入队（enqueue）**，只能从队首删除，称为**出队（dequeue）**。

定义队列的ADT如下：

```
public interface QueueADT <E>{
    //定义队列的抽象数据类型
    public void clear();//清空队列
    public void enqueue(E it);//入队
    public E dequeue();//出队
    public E firstValue();//：获得队首元素
    public boolean isEmpty();//判断队列是否空
    public boolean isFull();//判断队列是否为满
    public void print();//打印队列
    public int size();//获得队列中实际的元素个数
}
```

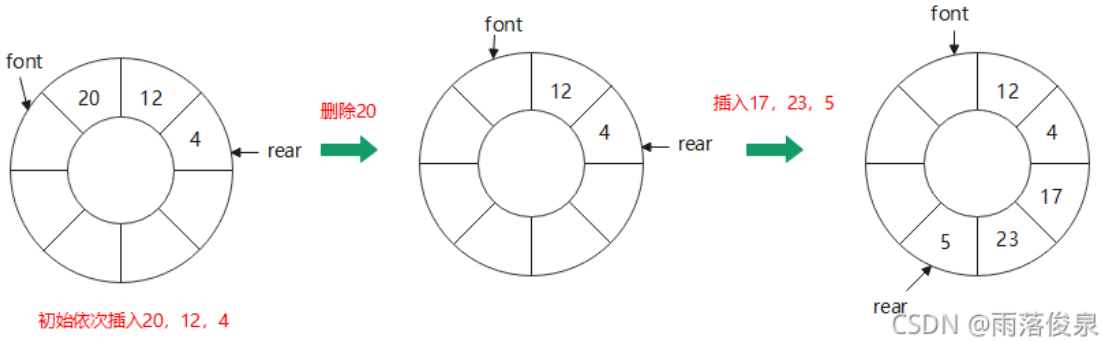
2.4.1 顺序队

在顺序队时，通常让队尾指针rear指向刚进队的元素位置（队尾），队首指针front指向刚出队的元素位置（队首的前驱）。因此，元素进队的时候，rear向后移动；元素出队的时候，front也要向后移动。这样一系列操作以后，rear和front都会到达数组的末端，导致元素无法插入（"假溢出"）。

要解决以上问题，需要将数组设置成一个环（可以通过取模操作实现），让rear和front沿着环走，这样就产生了循环队列。在这种方法中，数组位置的编号是0~size-1，size-1被定义为0的前驱。

为了便于判断队列空还是满，我们将存储队列元素的数组的大小定义为队列允许存储的**最大长度+1**，成员size用来控制队列的循环。**当font=rear，队列为空；当font=(rear+1)%size，队列为满；当font=(rear-1)%size，队列只有一个元素**

图示如下：



实现代码如下

```
```

```java
public class AQueue<E> implements QueueADT<E> {
    //定义顺序队
    private static final int DEFAULT_SIZE=10;
    private int max_size;//定义队列的实际最大容纳量+1
    private int size;//队列中实际的元素个数
    private int front;//队首元素的前驱元素下标
    private int rear;//队尾元素下标
    private E[] listArray;//存储元素的下标

    private void setUp(int size){
        this.max_size=size+1;
        front=rear=0;
        listArray=(E[])new Object[size+1];
    }

    public AQueue() {
        setUp(DEFAULT_SIZE);
    }
    public AQueue(int size){
        setUp(size);
    }

    @Override
    public void clear() {//清空队列元素
        front=rear=0;
    }

    @Override
    public void enqueue(E it) {//进队
        if (isFull()){
            System.out.println("queue is full!");
            return;//队列已经满了
        }
        rear=(rear+1)%max_size;
        listArray[rear]=it;
        this.size++;
    }

    @Override
    public E dequeue() {//出队
        if (isEmpty()){
            System.out.println("queue is empty");
            return null;
        }
        front=(front+1)%max_size;
        size--;
        return listArray[front];

    }

    @Override
    public E firstValue() {
        if (isEmpty()){
            System.out.println("queue is empty");
            return null;
        }
        return listArray[(front+1)%max_size];
    }
```

```
    @Override
    public boolean isEmpty() {
        return front==rear;
    }

    @Override
    public boolean isFull() {
        return front==(rear+1)%max_size;
    }

    @Override
    public void print() {
        if (isEmpty()){
            System.out.println("queue is empty");
            return;
        }
        int temp= front+1;
        while((temp%max_size)!=rear){
            System.out.print(listArray[temp]+" ");
            temp++;
        }
        System.out.println(listArray[temp]);
    }

    @Override
    public int size() {
        return this.size;
    }
}
```

测试代码如下

```
public class AQueueTest {
    public static void main(String[] args) {
        AQueue<Integer> test = new AQueue<>();
        test.print();
        test.enqueue(1);
        test.enqueue(21);
        test.enqueue(3);
        test.enqueue(43);
        test.enqueue(3);
        test.enqueue(7);
        System.out.print("enqueue: ");
        test.print();
        System.out.println("the number of queue elements: " + test.size());
        System.out.println("get the front element: "+test.firstValue());
        test.dequeue();
        test.dequeue();
        System.out.print("dequeue two elements: ");
        test.print();
    }
}
```

运行结果如下

```
queue is empty
enqueue: 1 21 3 43 3 7
the number of queue elements: 6
get the front element: 1
dequeue two elements: 3 43 3 7
```

2.4.2 链队

链队的实现比顺序队简单的多，成员front和rear分别指向队首元素和队尾元素。不需要表头结点。

实现代码如下

```
public class LQueue<E> implements QueueADT<E> {
    private Link<E> front;//指向队首元素的指针
    private Link<E> rear;//指向队尾元素的指针
    private int size=0;//队列中实际的元素个数

    private void setUp() {
        front = rear = null;
    }

    public LQueue() {
        setUp();
    }

    public LQueue(int size) {
        setUp();
    }

    @Override
    public void clear() {
        front = rear = null;
    }

    @Override
    public void enqueue(E it) {//进队
        if (isEmpty()) {
            front = rear = new Link<E>(it, null);//队列为空
        } else {
            rear.setNext(new Link<E>(it, null));
            rear = rear.getNext();//尾指针后移
        }
        size++;

    }

    @Override
    public E dequeue() {//出队
        if (isEmpty()) {
            System.out.println("queue is empty");
            return null;
        }
        E it = front.getElement();
        front = front.getNext();
        if (front == null) rear = null;//如果队空了，也要将尾指针设为null
        size--;
```

```
        return it;
    }

    @Override
    public E firstValue() {
        if (isEmpty()) {
            System.out.println("queue is empty");
            return null;
        }
        return front.getElement();
    }

    @Override
    public boolean isEmpty() {
        return front == null;
    }

    @Override
    public boolean isFull() {
        return false;
    }

    @Override
    public void print() {
        if (isEmpty()) {
            System.out.println("queue is empty");
            return;
        }
        Link<E> temp = front;
        if (temp == rear) {
            System.out.println(temp.getElement());
            return;
        }
        while (temp != rear) {
            System.out.print(temp.getElement() + " ");
            temp = temp.getNext();
        }
        System.out.println(temp.getElement());
    }

    @Override
    public int size() {
        return size;
    }
}
```

2.5 队列的应用——基数排序

1、算法介绍

基数排序的关键思想是“多关键字排序”，而其具有两种基本的实现方式

1 最高位优先(MSD)

先按最高位排成若干子序列，再对每个子序列依次按高位排序，以扑克牌为例，就是先按照花色排成四个子序列，再对每个花色的13张牌进行排序，最后使整个牌有序。

[2] 最低位优先(LSD)

这种方式不用先分成子序列，每次排序全体关键字都参加。最低位可以优先这样进行，不通过比较，而是通过“分配”和“收集”。同样以扑克牌为例，可以先按照数字将牌分配到1~13的13个桶中，然后从第一个桶开始依次收集；再将收集好的牌按照花色分配到4个桶中，也是从第一个桶开始收集。经过两次“分配”和“收集”操作，最终使牌有序

2、算法流程

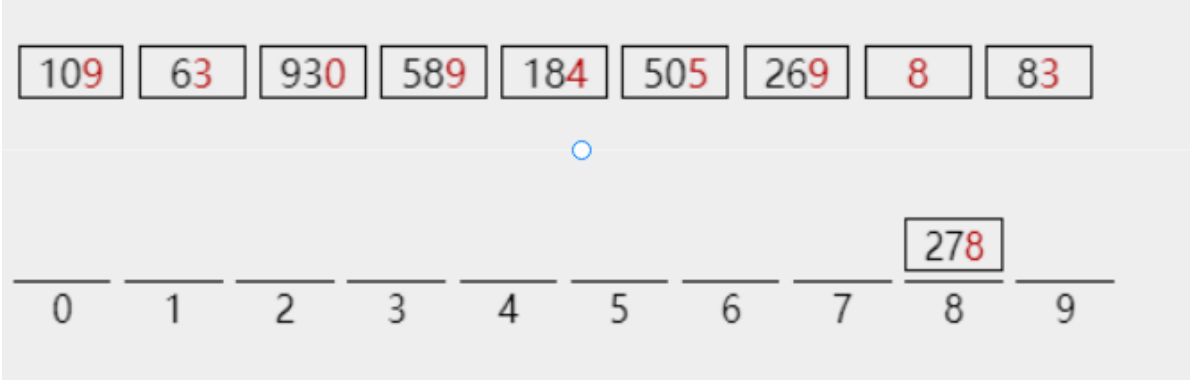
以LSD为例，说明基数排序的过程，原始序列为：278 109 63 930 589 184 505 269 8 83

每个关键字的每一位都是由数字组成的，数字的范围是0~9，所以准备10个桶来放关键字，需要注意组成关键字的每一位不一定是数字，也有可能是扑克牌的花色（要准备4个桶），或者是英文字母（不区分大小的话要准备26个桶）。这里的桶，就是一个**先进先出的队列。**

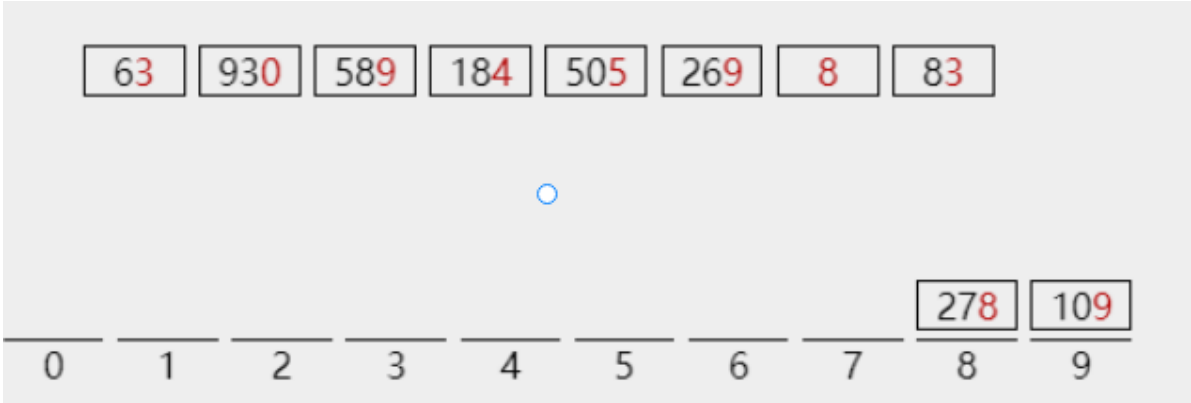
[1] 进行第一趟分配和收集，按照最后一位

1）分配过程如下（关键字从桶的上面进入）

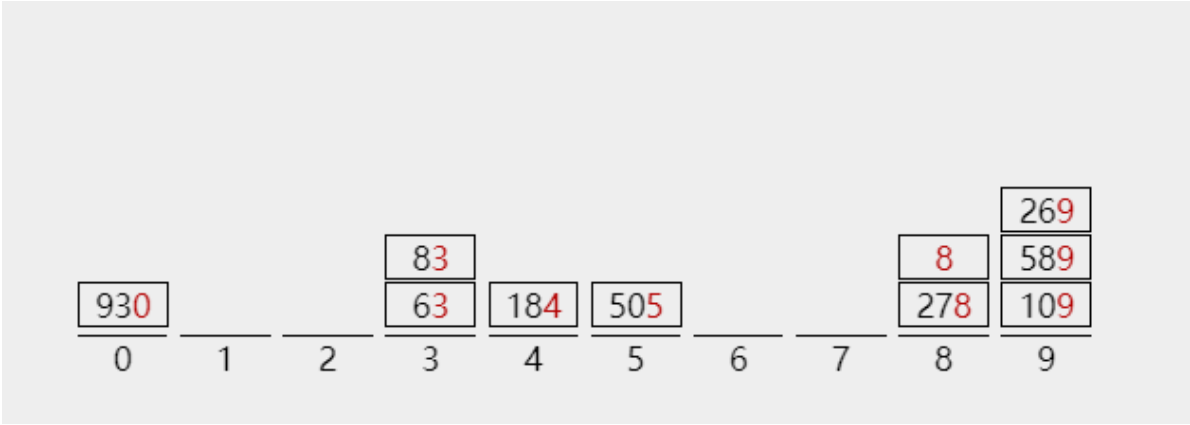
278的最低位是8，放入桶8中



109的最低位是9，放进桶9



按照以上方法，依次将数字放入桶中，完成第一趟分配



2）收集过程如下，按照0~9的顺序收集，注意关键字从桶的下面收集

> 桶0：930
>
> 桶1：不收集
>
> 桶2：不收集
>
> 桶3：63、83
>
> .....
>
> 桶8：278，8
>
> 桶9：109，589，269

将每桶收集的关键字依次排开，第一趟收集后的结果为

> 930 63 83 184 505 278 8 109 589 269

[2] 在第一趟排序结果的基础上，进行第二次分配与收集，按照中间位

1）第二趟分配的结果如下

2）第二趟的收集过程如下

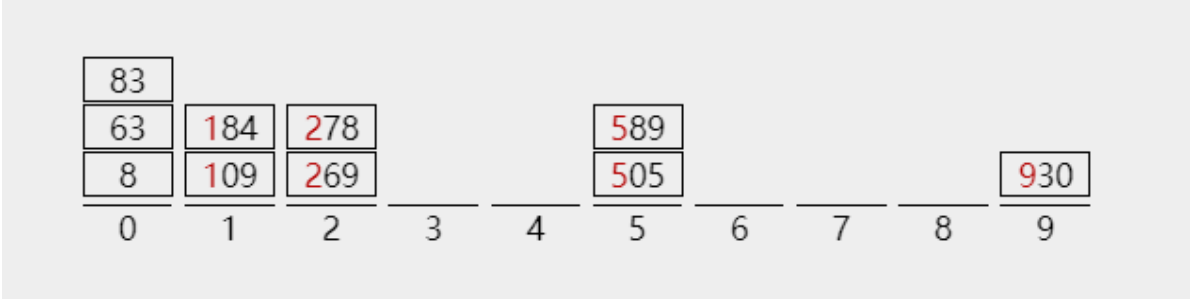
> 桶0：505，8，109
>
> 桶1：不收集
>
> 桶2：不收集
>
> 桶3：930
>
> .....
>
> 桶8：83，184，589
>
> 桶9：不收集

第二趟的收集结果如下：

> 505 8 109 930 63 269 278 83 184 589

[3] 在第二趟排序结果的基础上，进行第三趟分配与收集，按照最高位

1）第三趟的分配结果如下



2）进行第三趟收集

> 桶0：8，63，83
>
> 桶1：109，184
>
> 桶2：269，178
>
> 桶3：不收集
>
> .....
>
> 桶8：不收集
>
> 桶9：930

第三趟的收集结果如下：

> 8 63 83 109 184 269 278 505 589 930

此时，最高位有序，最高位相同的关键字按照中间位有序，中间位相同的关键字按最低位有序，于是整个序列有序，基数排序结束

LSD的基数排序适用于位数小的数列，如果位数多的话，使用MSD的效率会比较好。MSD的方式与LSD相反，是由高位数为基底开始进行分配，但在分配之后并不马上合并回一个数组中，而是在每个“桶子”中建立“子桶”，将每个桶子中的数值按照下一数位的值分配到“子桶”中。在进行完最低位数的分配后再合并回单一的数组中。

3、算法性能分析

时间复杂度：平均和最坏情况下都是 $O(d(n+r_d))$

空间复杂度：$O(r_d)$

其中，n为序列中的关键字数，d为关键字的关键字位数，如930，由3位组成，d=3；r_d 为关键字基的个数，这里的基指的是构成关键字的符号，如关键字为数值时，构成关键字的符号就是0~9这些数字，一共十个，所以 $r_d = 10$

> ☆:时间复杂度分析
>
> 基数排序每一趟进行分配和收集。分配需要依次对序列中的每个关键字进行，即需要顺序扫描整个序列，所以有n这一项；收集需要依次对每个桶进行，而桶的数量取决于关键字的取值范围，如放数字的桶有10个，放字母的桶有26个，刚好是r_d ,所以有r_d 这一项，于是进行一趟分配和收集则需要花费 $n+r_d$ 。整个排序需要进行的趟数即为关键字的关键字位数，即为d。所以基数排序的时间复杂度即为 $O(d(n+r_d))$

4、举例说明

利用队列实现对某一个数据序列的排序（采用基数排序），其中对数据序列的数据（第 1 和第 2 条进行说明）和队列的存储方式（第 3 条进行说明）有如下的要求：

1）当数据序列是整数类型的数据的时候，数据序列中每个数据的位数不要求等宽，比如： 1、21、12、322、44、123、2312、765、56

2）当数据序列是字符串类型的数据的时候，数据序列中每个字符串都是等宽的，比如："abc","bde","fad","abd","bef","fdd","abe"

3）要求重新构建队列的存储表示方法：使其能够将 n 个队列顺序映射到一个数组 listArray 中， 每个队列都表示成内存中的一个循环队列【这一项是可选项】

思路：对于数字和等长字符串，可采用最低位优先法排序；对于要求3，即是要用一个存储数组存多个队列的值，这时候可以利用指针数组front[n]和rear[n]进行求解，注意数组下标的变化（这一问我参考了大佬的答案,就不放出来了）

```
public class RadixSort {
    public static void LSD(int[] num) {
        //对数字采用最低位优先法排序
        MyQueue queue = new MyQueue(10, num.length);//分配10个队列
        int digits = getNumDigits(num);
        int mode = 1;
        while (digits != 0) {
            for (int i = 0; i < num.length; i++) {
                queue.enqueue((num[i] / mode) % 10, num[i]);
                //按桶分配,(num[i]/mode)%10表示取的位数
            }
            int k = 0;
            for (int j = 0; j < 10; j++) {
                while (!queue.isEmpty(j)) {
                    num[k] = (int) queue.dequeue(j);
                    k++;
                }
            }//出队
            digits--;//位上移
            mode = mode * 10;
        }
    }

    public static void LSD(String[] str) {
        //对字符串采取最低位优先法
        //对数字采用最低位优先法排序
        MyQueue queue = new MyQueue(27, str.length);//分配27个队列
        //27个桶，其中第27个桶用来存储除字母以外的其他字符，不区分大小写
        int digits = str[0].length();//等长字符的长度
        int mode = 1;
        while (digits != 0) {
            for (int i = 0; i < str.length; i++) {
                int index;//桶的下标
                if (str[i].charAt(digits - 1) >= 'A' && str[i].charAt(digits -
1) <= 'Z') {
                    index = str[i].charAt(digits - 1) - 'A';
                } else if (str[i].charAt(digits - 1) >= 'a' &&
str[i].charAt(digits - 1) <= 'z') {
                    index = str[i].charAt(digits - 1) - 'a';
                } else {
                    index = 26;
                }//不区分大小写
```

```
                queue.enqueue(index, str[i]);
                //按桶分配
            }
            int k = 0;
            for (int j = 0; j < 27; j++) {
                while (!queue.isEmpty(j)) {
                    str[k] = (String) queue.dequeue(j);
                    k++;
                }
            }//出队
            digits--;//位上移
        }
    }

    static int getNumDigits(int[] num) {//获得最大的数的位数
        int max = num[0];//最大的数
        int digits = 0;//位数
        for (int i = 0; i < num.length; i++) {
            if (num[i] > max) max = num[i];
        }
        while (max / 10 != 0) {
            digits++;
            max = max / 10;
        }
        if (max % 10 != 0) {
            digits++;
        }
        return digits;
    }


    public static void main(String[] args) {
        int[] num = {12, 32, 2, 231, 14, 23};
        System.out.println("before sorting: " + Arrays.toString(num));
        LSD(num);
        System.out.println("after sorting: " + Arrays.toString(num));
        String[] strings = {"abc", "bde", "fad", "abd", "bef", "fdd ", "abe" };
        System.out.println("before sorting: " + Arrays.toString(strings));
        LSD(strings);
        System.out.println("after sorting: " + Arrays.toString(strings));
    }
}
```

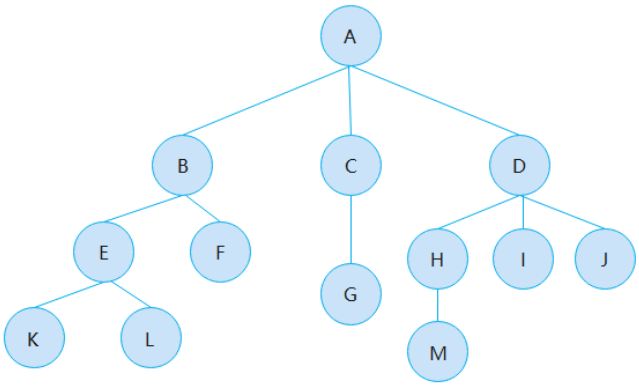
运行结果如下

```
before sorting: [12, 32, 2, 231, 14, 23]
after sorting: [2, 12, 14, 23, 32, 231]
before sorting: [abc, bde, fad, abd, bef, fdd , abe]
after sorting: [abc, abd, abe, bde, bef, fad, fdd ]
```

3.1 树和二叉树的基本概念与性质

1、树的定义

树是一种非线性的数据结构，下图就是一个树，它是由若干个结点（如A,B,C）组成的结合，是由唯一的**根**（结点A）和若干棵互不相交的子树（如B,E,F,K,L这五个结点组成的树就是一棵**子树**。其中，每一棵子树也是一棵树，也是由唯一的根结点和若干课棵互不相干的子树组成的，由此可见，树的定义是递归的。需注意，树的结点数目可以为零，当结点数为0时，这是一棵**空树**。



2、跟树相关的概念

结点：结点不仅包含数据元素，而且包含指向子树的指针信息，如A,B,C都是结点，A中不仅有数据元素A，也包含了指向结点B,C,D所在子树的指针。

结点的度：指的是结点拥有的子树个数或者分支的个数，比如A中有三个子树，所以A结点的度为3.

树的度：各个结点中度的最大值

叶结点：又叫做终端结点，指的是度为0的结点，如图中K,L,F,G,M,I,J

分支结点：也叫做非终端结点，指的是度不为零的结点。

子结点：结点的子树的根结点，如A结点的子结点就是B,C,D

父结点：与子结点的定义相对应，B,C,D结点的父结点就是A结点

兄弟：同一个父结点的子结点之间互为兄弟，如B,C,D互为兄弟

堂兄弟：双亲在同一层的结点互为堂兄弟，如E和H就互为堂兄弟

祖先：从根到某结点的路径上的所有结点，都是这个结点的祖先，如K结点的祖先就有A,B,E

子孙：以某结点为根的子树中的所有结点，如B结点的子孙就有E,F,K,L

层次：以根开始，**根为第0层**（不同的教材定义不同），根的孩子为第1层，根的孩子的孩子为第2层，以此类推

结点的深度和高度：

1）**结点的深度**是从根结点到该结点路径上的结点数**（路径的长度）**

2）从某结点往下走可能到达多个叶子节点，对应了多条通往这些叶子节点的路径，其中最长的路径上结点的个数即为该结点在树中的高度，如图中B结点的高度为3，就是从B到K的结点数为3

3）根结点的高度即为树的高度

树的高度：树中结点的最大深度+1，如图中所示树的高度就为4

无序树：树中结点的子树没有顺序，可以任意交换

有序树：树中结点的子树是有顺序的，不能交换

森林：若干课互不相交的树组成的集合

3、二叉树的定义

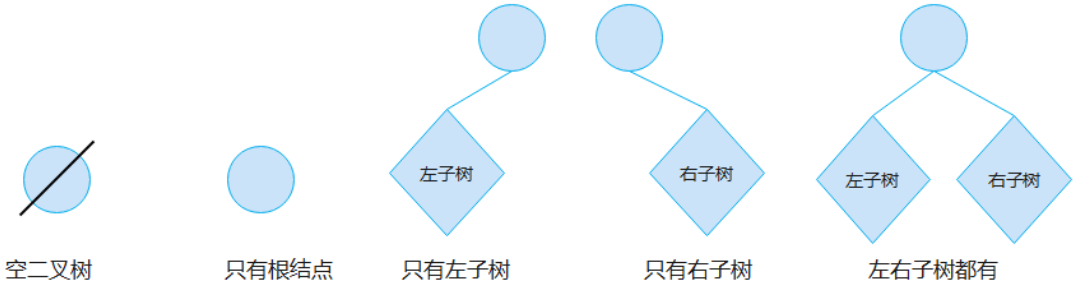
在树的基础上，加上以下两个条件就构成了二叉树：

[1] 每个结点最多只有两棵子树，即二叉树中结点的度只有0、1、2

[2] 子树有左右之分，不能颠倒

由以上的定义可知，二叉树由五个基本的形态：

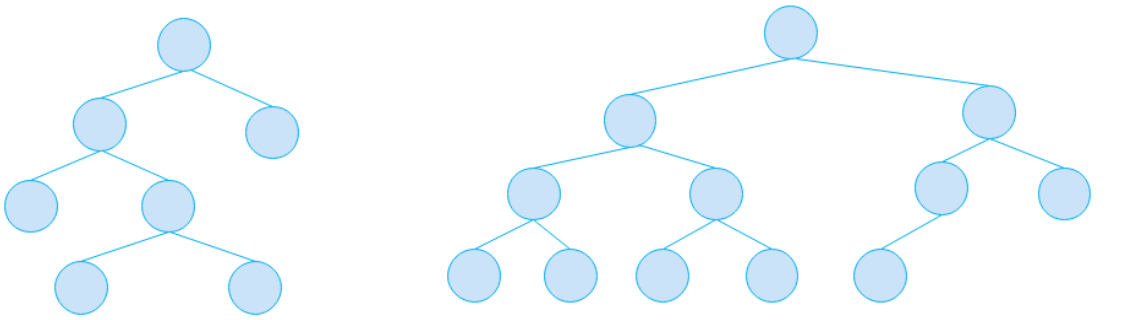
空二叉树、只有根结点、只有左子树、只有右子树、既有左子树也有右子树，如下图所示；



满二叉树与完全二叉树

满二叉树：full binary tree 满二叉树的每一个结点，要么是一个恰有两个非空结点的分支结点要么是一个叶子结点。

完全二叉树：complete binary tree 完全二叉树有严格的要求：从根结点起，每一层**从左往右填充**（不能跳着填入）。一棵高度为d的完全二叉树除了d-1层（最后一层）以外，每一层都是满的



满二叉树（不是完全二叉树）　　完全二叉树（不是满二叉树）

4、二叉树的性质定理

[1] **满二叉树定理**：非空满二叉树的叶结点数等于其分支结点数+1

[2] **定理：一棵非空二叉树空子树的数目等于其结点数目+1**，证明如下：

证明1：设二叉树T，将其中所有的空子树都换成叶结点，更换以后的树记作T'，所有原来树T的结点都变为树T'的分支结点，由满二叉树的定义可知，T'为满二叉树。根据满二叉树定理，新添加的叶结点数等于树T的结点树+1，由于新添加的叶结点为原来书中的空子树，因此树T中的空子树数目等于其结点数+1

证明2：有定义可知，树T中每个结点都有2个子结点，因此一棵由n个结点的树一共有2n个子结点，除了根结点以外，每个结点都有一个父结点，于是共有n-1个父结点，即有n-1个非空子结点，既然一共有2n个子结点，那么其中必然有n+1个空的子结点，即有n+1个空子树

[3] **定理**：对任何一棵二叉树T，如果其叶结点树为n_0，则具有两棵子树的分支结点数为n_2，则有$n_0 = n_2 + 1$

证明：二叉树中，只有三种结点：叶结点n_0，有一棵子树的结点n_1，有两棵子树的结点n_2，设该树的节点总数为n，则有

$n = n_0 + n_1 + n_2$，也有$n = \underbrace{0 \times n_0 + 1 \times n_1 + 2 \times n_2}_{\text{分支数}} + 1$，结合这两个式子即可证明

[4] **定理**：高度为k的二叉树最多有$2^k - 1 (k \geq 1)$个结点

[5] **定理：** 具有$n(n \geq 1)$个结点的完全二叉树的高度为$\lfloor \log_2 n \rfloor + 1$，也可写作$h = \lceil \log_2 (n+1) \rceil$

证明：由定理四可知：$2^{h-1} - 1 < n \leq 2^h - 1$，其中h为这棵二叉树的高度，也可以写作$2^{h-1} \leq n < 2^h$，对其取对数则有$h - 1 \leq \log_2 n < h$，由于h为整数，所以对$\log_2 n$向下取整即可得到一个等式：$h - 1 = \lfloor \log_2 n \rfloor$,故有$h = \lfloor \log_2 n \rfloor + 1$

[6] 有n个结点的完全二叉树，对其结点从上到下、从左到右依次编号（编号范围为1~n），则结点之间有如下关系：

若i为某结点a的编号，则：

如果$i \neq 1$,则a的父结点编号为$\lfloor i/2 \rfloor$

如果$2i \leq n$，则a左孩子的编号为$2i$；如果$2i > n$则a无左孩子

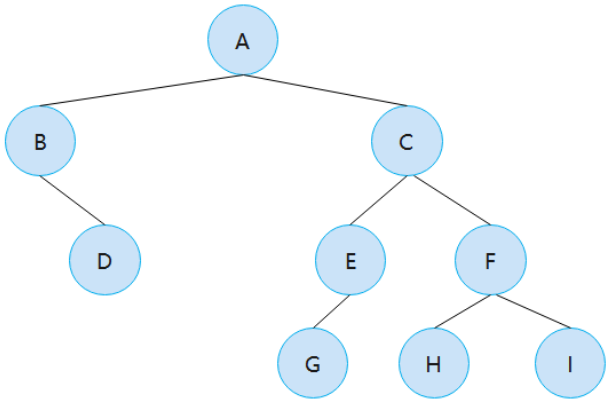
如果$2i + 1 \leq n$，则a右孩子的编号为$2i + 1$；如果$2i + 1 > n$则a无右孩子

如果从0开始编号，则其左孩子编号为$2i + 1$，右孩子编号为$2i + 2$，父结点编号为$\lfloor i/2 \rfloor + 1$

3.2 二叉树的周游与实现

3.2.1 二叉树的周游

按照一定顺序访问二叉树的结点，称为一次周游或遍历。对每个结点都进行一次访问并列出，称为二叉树的枚举。通常二叉树的周游分为前序周游、中序周游和后序周游，以下图二叉树来进行举例说明。



前序周游：先访问结点，后访问其子结点。上图前序周游的结果为A,B,D,C,E,G,H,I

后序周游：先访问结点的子结点（包括他们的子树），然后再访问该结点。上图二叉树经过后序周游枚举出来的结果为：D,B,G,E,H,I,F,C,A

中序周游：先访问左子结点（包括整个子树），然后是该结点，最后访问右子结点（包括整个子树）。上图二叉树经过后序周游枚举出来的结果为：B,D,A,G,E,C,H,F,I

周游路线可以很容易通过递归实现，初始调用时传入根结点指针，然后按照既定的顺序周游结点及其子结点。以前序周游为例：

```
void preOrder(BinNode root){
   if(root==null) return;//空树
   visit(root);
   preOrder(root.getLeft());
   preOrder(root.getRight());
}
```

对于中序周游和后序周游，只需要修改后面三个方法的顺序即可。

3.2.2 二叉树的实现

二叉树的实现有两种方式，顺序存储，即利用数组进行存储，以及链式存储，利用指针进行存储。

数组存储方式的分析

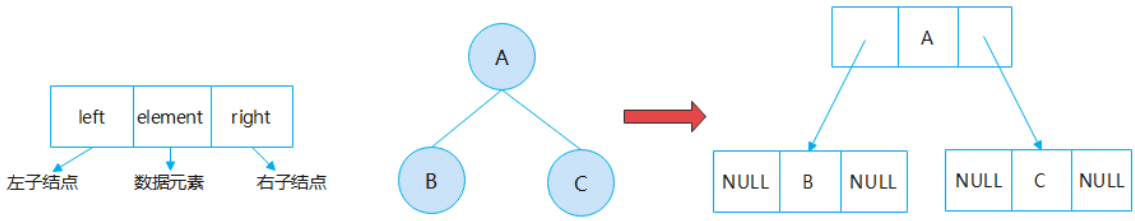
> 优点：通过下标方式访问元素，速度快。**对于有序数组**，还可使用二分查找提高检索速度。
>
> 缺点：如果要检索具体某个值，或者插入值(按一定顺序)**会整体移动**，效率较低

链式存储方式的分析

> 优点：在一定程度上对数组存储方式有优化(比如：插入一个数值节点，只需要将插入节点，链接到链表中即可， 删除效率也很好)。
>
> 缺点：在进行检索时，效率仍然较低，比如(检索某个值，需要从头节点开始遍历)

1、链式二叉树的实现

对于链式存储结构，我们需要定义含有一个数据域和两个指针域的链式结点结构，其中left为左子结点，element为数据元素，right为右子结点。结点定义以及二叉树表示如下：



对于链式结点，其定义如下：

```
public class BinNodePtr {
    //链式二叉树的结点类
    private Object element;//存储数据的元素
    private BinNodePtr left;//左子结点
    private BinNodePtr right;//右子结点

    //constructor部分
    public BinNodePtr(){}
    public BinNodePtr(Object element){
        this.element=element;
    }

    //getter与setter部分
    public Object getElement() {
        return element;
```

```
    }

    public void setElement(Object element) {
        this.element = element;
    }

    public BinNodePtr getLeft() {
        return left;
    }

    public void setLeft(BinNodePtr left) {
        this.left = left;
    }

    public BinNodePtr getRight() {
        return right;
    }

    public void setRight(BinNodePtr right) {
        this.right = right;
    }

    @Override
    public String toString() {
        return "BinNodePtr{" +
                "element=" + element +
                '}';
    }
}
```

而链式二叉树中只有一个私有属性root，表示的是这棵树的根结点，对于简单二叉树，我们这里就只定义其遍历方法、查找方法和删除方法。

对于其遍历方法，这里我用了三种，前面介绍周游的时候就已经说明了

```
    //前序遍历
    public void preOrder(BinNodePtr node) {
        if (node != null) {
            System.out.println(node);
            preOrder(node.getLeft());
            preOrder(node.getRight());
        } else return;
    }

    //中序遍历
    public void infixOrder(BinNodePtr node) {
        if (node != null) {
            preOrder(node.getLeft());
            System.out.println(node);
            preOrder(node.getRight());
        } else return;
    }

    //后序遍历
    public void postOrder(BinNodePtr node) {
        if (node != null) {
            preOrder(node.getLeft());
```

```
            preOrder(node.getRight());
            System.out.println(node);
        } else return;
    }
```

然后是查找，查找同样可以使用前序、中序以及后序：

```
    /*
     * @Description: 利用前序、中序、后序分别查找element
     * @Author yjq
     * @Date 2021/9/21 21:05
     * @Param node, element
     * @Return 查找到返回该node，查找不到则返回null
     * @Exception
     */
    //前序查找
    public BinNodePtr preOrderSearch(BinNodePtr node, Object element) {
        if (node != null) {
            if (node.getElement().equals(element)) {
                return node;//先比较当前结点
            }
            BinNodePtr temp = null;
            if (node.getLeft() != null) {
                temp = preOrderSearch(node.getLeft(), element);
                //左子结点不为空则接着前序递归查找
            }
            if (temp != null) {
                return temp;//返回值不为null说明找到了
            }
            if (node.getRight() != null) {
                temp = preOrderSearch(node.getRight(), element);
            }
            return temp;
        } else {
            return null;
        }
    }

    //中序查找
    public BinNodePtr infixOrderSearch(BinNodePtr node, Object element) {
        if (node != null) {
            BinNodePtr temp = null;
            if (node.getLeft() != null) {
                temp = infixOrderSearch(node.getLeft(), element);
                //左子结点不为空则接着前序递归查找
            }
            if (temp != null) {
                return temp;//返回值不为null说明找到了
            }
            if (node.getElement().equals(element)) {
                return node;//比较当前结点
            }
            if (node.getRight() != null) {
                temp = infixOrderSearch(node.getRight(), element);
            }
            return temp;
        } else {
```

```
            return null;
        }
    }

    //后序查找
    public BinNodePtr postOrderSearch(BinNodePtr node, Object element) {
        if (node != null) {
            BinNodePtr temp = null;
            if (node.getLeft() != null) {
                temp = postOrderSearch(node.getLeft(), element);
                //左子结点不为空则接着前序递归查找
            }
            if (temp != null) {
                return temp;//返回值不为null说明找到了
            }
            if (node.getRight() != null) {
                temp = postOrderSearch(node.getRight(), element);
            }
            if (temp != null) {
                return temp;//返回值不为null说明找到了
            }
            if (node.getElement().equals(element)) {
                return node;//比较当前结点
            }
            return temp;
        } else {
            return null;
        }
    }
```

最后是删除结点，这里会比较难一些，注意我们要比较的是当前结点的子结点，而不是当前结点。

```
    /*
     * 递归删除结点
     * 分为两种情况：如果删除的是叶结点，则删除该结点4
     * 如果删除的是非叶结点，则删除该子树
     */
    public void delNode(BinNodePtr node, Object element) {
        //需要注意，我们判断的是当前结点的子结点是否需要删除，
        //而不是判断当前结点是否要删除
        if (node.equals(root)&&node==null){
            System.out.println("the tree is empty!");
        }
        if (node.getLeft() != null) {
            if (node.getLeft().getElement().equals(element)){
                node.setLeft(null);
                return;
            }else {
                delNode(node.getLeft(),element);//向左子树递归删除
            }
        }
        if (node.getRight()!=null){
            if (node.getRight().getElement().equals(element)){
                node.setRight(null);
                return;
            }else {
                delNode(node.getRight(),element);//向右子树递归删除
```

```
            }
        }
    }
```

最后测试一下：

```
public class BinaryTreePtrTest {
    public static void main(String[] args) {
        BinaryTreePtr test = new BinaryTreePtr();
        //创建结点
        BinNodePtr root = new BinNodePtr("A");
        BinNodePtr node1 = new BinNodePtr("B");
        BinNodePtr node2 = new BinNodePtr("C");
        BinNodePtr node3 = new BinNodePtr("D");
        BinNodePtr node4 = new BinNodePtr("E");
        BinNodePtr node5 = new BinNodePtr("F");
        BinNodePtr node6 = new BinNodePtr("G");
        BinNodePtr node7 = new BinNodePtr("H");
        BinNodePtr node8 = new BinNodePtr("I");

        //暂时用手动创建二叉树
        test.setRoot(root);
        root.setLeft(node1);
        root.setRight(node2);
        node1.setRight(node3);
        node2.setLeft(node4);
        node2.setRight(node5);
        node4.setLeft(node6);
        node5.setLeft(node7);
        node5.setRight(node8);

        //遍历测试
        System.out.println("前序周游");
        test.preOrder(test.getRoot());
        System.out.println("中序周游");
        test.infixOrder(test.getRoot());
        System.out.println("后序周游");
        test.postOrder(test.getRoot());

        //查找测试
        System.out.println("查找B");
        System.out.println(test.infixOrderSearch(test.getRoot(),"B"));

        //删除测试
        System.out.println("删除C");
        test.delNode(test.getRoot(),"C");
        test.infixOrder(test.getRoot());
    }
}
```

测试结果如下：

```
前序周游
BinNodePtr{element=A}
BinNodePtr{element=B}
BinNodePtr{element=D}
```

```
BinNodePtr{element=C}
BinNodePtr{element=E}
BinNodePtr{element=G}
BinNodePtr{element=F}
BinNodePtr{element=H}
BinNodePtr{element=I}
中序周游
BinNodePtr{element=B}
BinNodePtr{element=D}
BinNodePtr{element=A}
BinNodePtr{element=C}
BinNodePtr{element=E}
BinNodePtr{element=G}
BinNodePtr{element=F}
BinNodePtr{element=H}
BinNodePtr{element=I}
后序周游
BinNodePtr{element=B}
BinNodePtr{element=D}
BinNodePtr{element=C}
BinNodePtr{element=E}
BinNodePtr{element=G}
BinNodePtr{element=F}
BinNodePtr{element=H}
BinNodePtr{element=I}
BinNodePtr{element=A}
查找B
BinNodePtr{element=B}
删除C
BinNodePtr{element=B}
BinNodePtr{element=D}
BinNodePtr{element=A}

Process finished with exit code 0
```

2、顺序二叉树

顺序二叉树一般都是完全二叉树，之后的堆排序中将会用到，到时候再具体的说明。顺序二叉树用数组来实现，n个结点的二叉树可以使用大小为n的数组来实现，因此不存在结构性开销，这一点比链式存储要好。而且每个结点其父结点与子结点下标之间存在规律如下：

公式中r表示结点的下标，n表示二叉树结点的总数：

1. $Parent(r) = (r-1)/2 \qquad 0 < r < n$
2. $Leftchild(r) = 2r+1 \qquad 2r+1 < n$
3. $Rightchild(r) = 2r+2 \qquad 2r+2 < n$
4. $Leftsibling(r) = r-1 \qquad$ 当r为偶数而且$0 < r < n$时
5. $Rightsibling(r) = r+1 \qquad$ 当r为奇数而且$r+1 < n$时

3.3 Huffman树

基本概念

路径长度：两个结点之间路径上的分支数

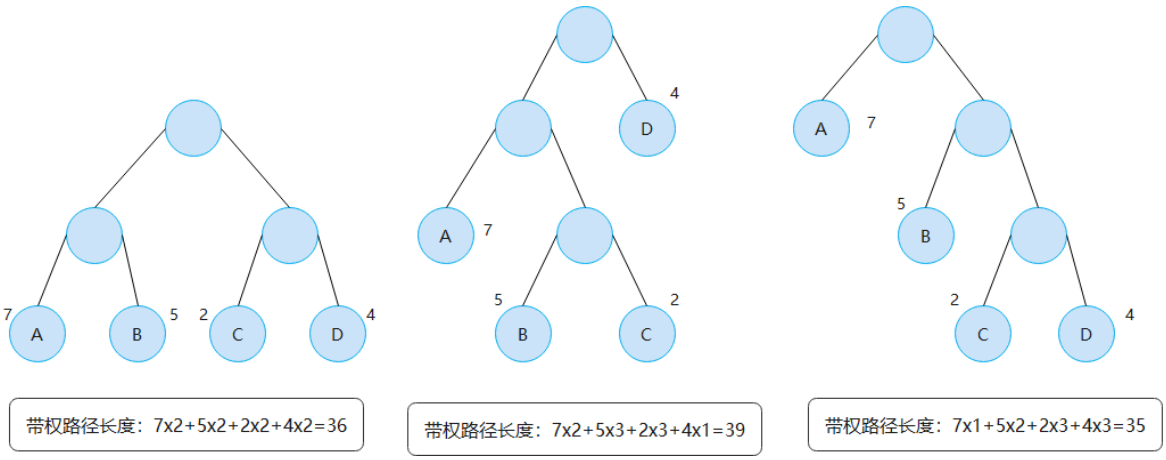
树的外部路径长度：各叶结点到根结点的路径长度之和

树的内部路径长度：各非叶结点到根结点的路径长度之和

树的带权路径长度：树中所有叶子结点的带权路径长度之和

Huffman树定义:是一类带权（外部）路径（Weighted PathLength）长度最短的树

举例：求下面二叉树的带权路径长度



☑ "权"大的叶结点深度小，它相对于总路径长度的花费最小，因此，其他叶结点如果"权"小，就会被推到树的较深处

构造算法

如何构造Huffman树?

1 根据给定的n个权值$\{w_1, w_2 \ldots, w_n\}$，构造n棵二叉树的集合$F = \{T_1, T_2, \ldots, T_n\}$，其中每棵二叉树中均**只含一个带权值为$w_i$的根结点**,其左、右子树为空树;

2 在F中选取其根结点的权值为最小的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和;

3 从F中删去这两棵树，同时加入刚生成的新树;

4 (4)重复(2)和(3)两步，直至F中只含一棵树为止

Huffman编码

前缀码

使用Huffman树编制的代码具有前缀特性prefix：一组代码中的任何一个代码都不是另一个代码的前缀

这种特性保证了代码串被反编码时不会有多种可能

字符编码

利用Huffman树的特性为使用频率不同的字符编写不等长的编码，从而缩短整个文件的长度

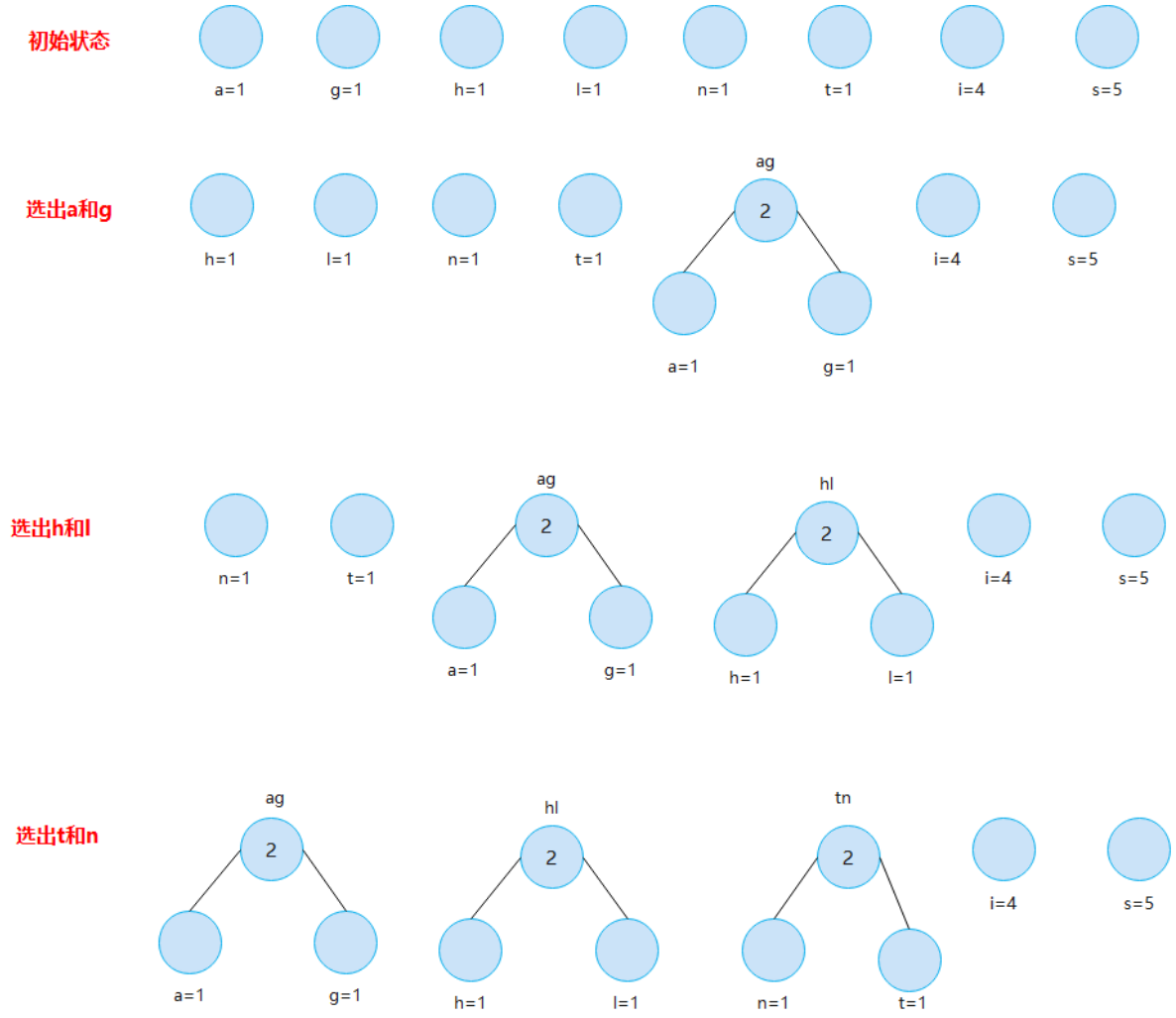
```
This is isinglass
```

> ■ t的频度是1 h的频度是1 i的频度是4 s的频度是5
> ■ n的频度是1 g的频度是1 a的频度是1 l的频度是1

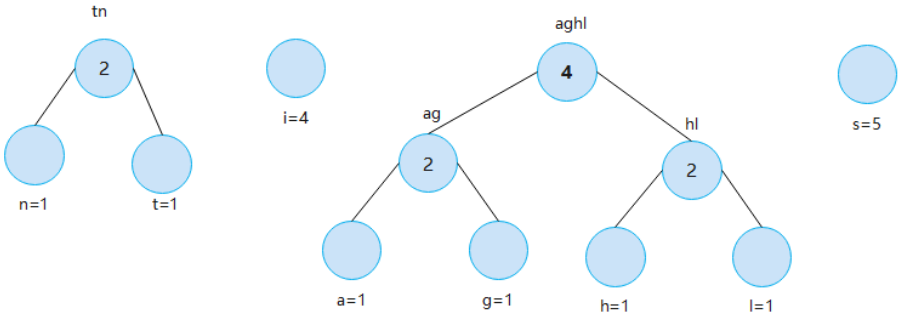
如果采用等长的编码形式，上面的八个字母则需要三位二进制编码
长度=15*3=45

按照上面字母出现的频度创建一个Huffman树

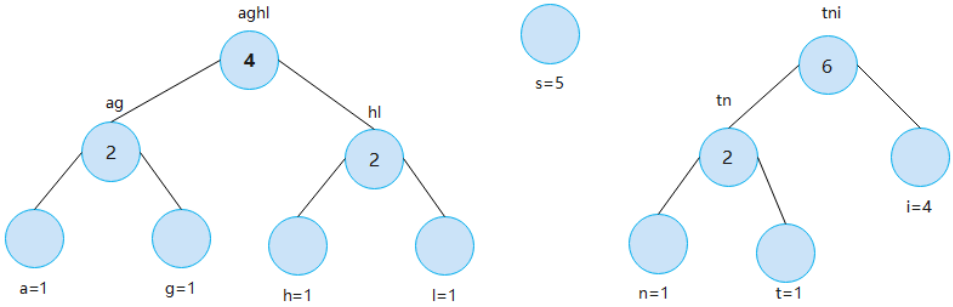
图解



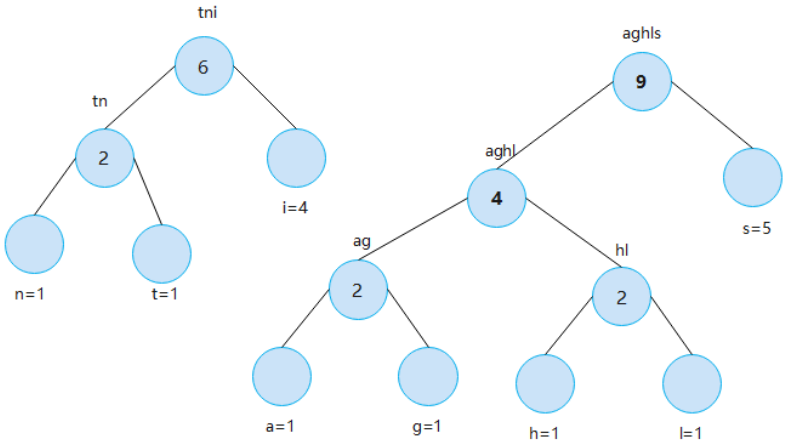
选出ag和hl



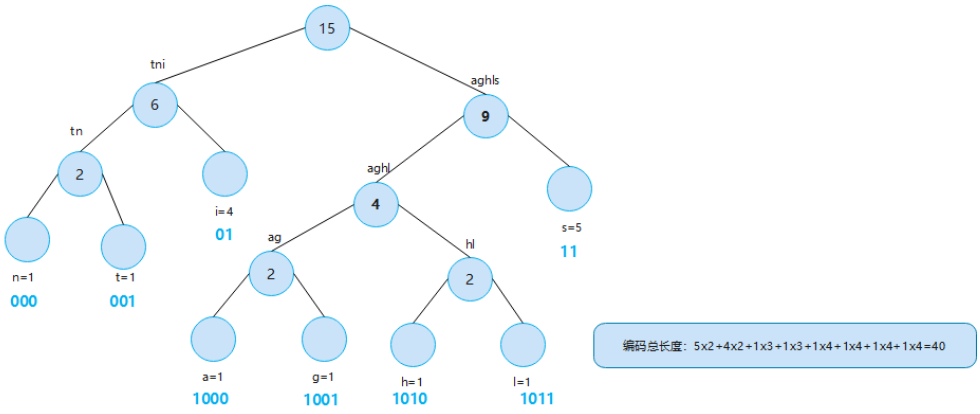
选出tn和i



选出aghl和s



最终Huffman树



代码实现（java）

```
class Letter {
    char element;//字母
    double weight;//字母出现的频率

    public Letter(char element, double weight) {
        this.element = element;
        this.weight = weight;
    }

    public char getElement() {
        return element;
    }

    public void setElement(char element) {
        this.element = element;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }
}

class HuffTreeNode {
    Letter letter;
    HuffTreeNode left;//左子结点
    HuffTreeNode right;//右子结点

    public Letter getLetter() {
        return letter;
    }

    public void setLetter(Letter letter) {
        this.letter = letter;
    }

    public HuffTreeNode getLeft() {
        return left;
    }

    public void setLeft(HuffTreeNode left) {
        this.left = left;
    }

    public HuffTreeNode getRight() {
        return right;
    }

    public void setRight(HuffTreeNode right) {
        this.right = right;
    }
}
```

```
public class HuffmanTree {
    //简单使用冒泡排序
    private void sort(HuffTreeNode[] nodes) {
        int flags = 0;
        for (int i = 0; i < nodes.length-1; i++) {
            for (int j = 0; j < nodes.length-1-i; j++) {
                if (nodes[j].letter.weight > nodes[j + 1].letter.weight) {
                    HuffTreeNode temp = nodes[j];
                    nodes[j] = nodes[j + 1];
                    nodes[j + 1] = temp;
                    flags = 1;//不是有序的，flags设置为1；

                }
            }
            if (flags == 0)
                return;
        }
    }

    /**
     * 根据字母及其频数生成Huffman树
     * @param letters
     * @return
     */
    public HuffTreeNode generateHuffTree(Letter[] letters) {
        HuffTreeNode[] nodes = new HuffTreeNode[letters.length];
        for (int i = 0; i < letters.length; i++) {
            nodes[i] = new HuffTreeNode();
            nodes[i].letter = letters[i];
        }
        while (nodes.length > 1) {
            sort(nodes);
            HuffTreeNode node1 = nodes[0];
            HuffTreeNode node2 = nodes[1];
            HuffTreeNode newTree = new HuffTreeNode();
            Letter temp = new
Letter('0',node1.getLetter().getWeight()+node2.getLetter().getWeight());
            newTree.setLetter(temp);
            newTree.setLeft(node1);
            newTree.setRight(node2);
            HuffTreeNode[] nodes2 = new HuffTreeNode[nodes.length - 1];//新的结点
数组，长度减一
            for (int i = 2; i < nodes.length; i++) {
                nodes2[i - 2] = nodes[i];
            }
            nodes2[nodes2.length - 1] = newTree;
            nodes = nodes2;
        }
        return nodes[0];
    }

    /**
     * 后序遍历
     * @param root 根结点
     * @param code 编码
     */
    public void print(HuffTreeNode root,String code){
```

```
        if(root != null) {
            print(root.getLeft(),code+"0");
            print(root.getRight(),code+"1");
            if(root.getLeft() == null && root.getRight() == null) {
                String m=root.getLetter().getElement()+"频
数:"+root.getLetter().getWeight()+" 哈夫曼编码: "+code;
                System.out.println(m);
            }
        }
    }
    public static void main(String[] args) {
        Letter a = new Letter('a', 1);
        Letter g = new Letter('g', 1);
        Letter h = new Letter('h', 1);
        Letter l = new Letter('l', 1);
        Letter n = new Letter('n', 1);
        Letter t = new Letter('t', 1);
        Letter i = new Letter('i', 4);
        Letter s = new Letter('s', 5);
        Letter[] test = {a, g, h, l, n, t, i, s};
        HuffmanTree huffmanTree = new HuffmanTree();
        huffmanTree.print(huffmanTree.generateHuffTree(test),"");
    }
}
```

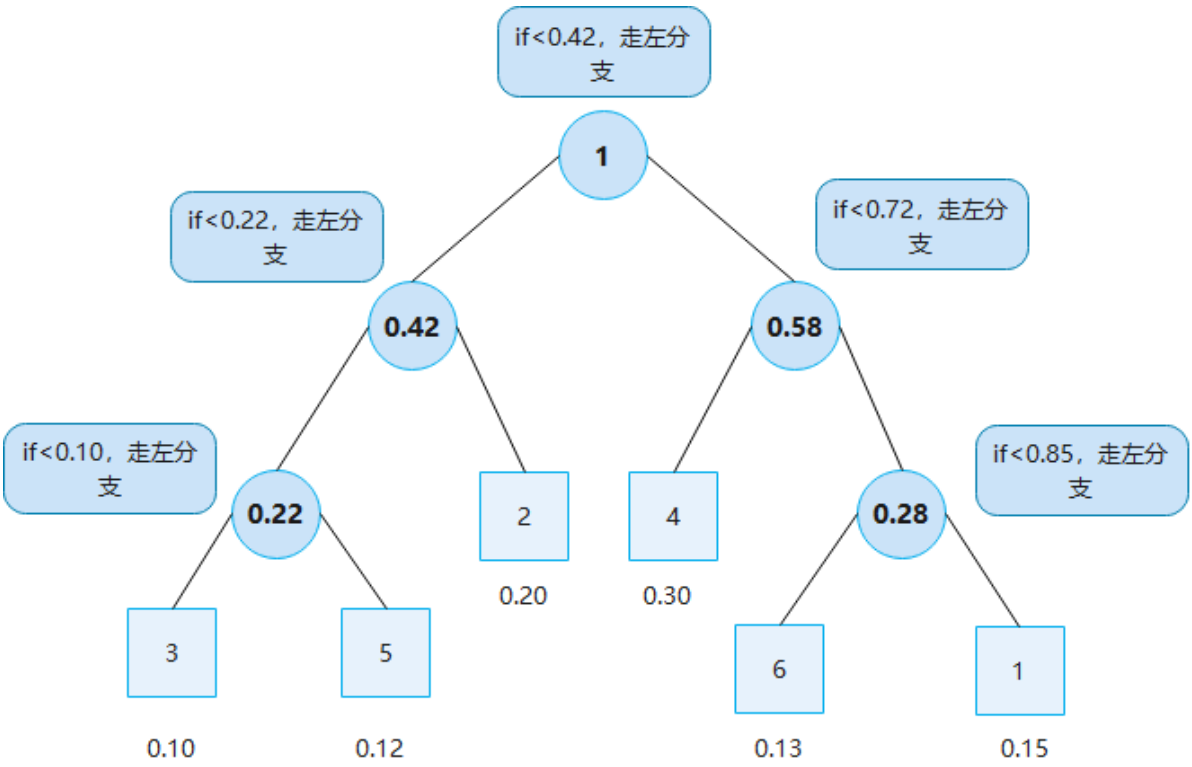
```
n频数:1.0 哈夫曼编码：000
t频数:1.0 哈夫曼编码：001
i频数:4.0 哈夫曼编码：01
a频数:1.0 哈夫曼编码：1000
g频数:1.0 哈夫曼编码：1001
h频数:1.0 哈夫曼编码：1010
l频数:1.0 哈夫曼编码：1011
s频数:5.0 哈夫曼编码：11
```

非等概率随机数

按照给定的概率生成相应的随机数
比如有1、2、3、4、5、6这6个数，编写一个随机发生器，使其能够按照如下概率（0.15、0.20、0.10、0.30、0.12和0.13）生成相应的6个数

[1] 解决方法一：可以使用JavaAPI中的随机数生成函数，生成[0，1）之间的数，按照区间生成数字

[2] 解决方法二：使用Huffman树减少比较次数




```
    public static int randomGenerate() {
        double temp = Math.random();
        int result=0;
        if (temp < 0.42) {
            if (temp < 0.22) {
                if (temp < 0.10) {
                    result = 3;
                } else {
                    result = 5;
                }
            } else {
                result = 2;
            }
        } else {
            if (temp < 0.72) {
                result = 4;
            } else {
                if (temp < 0.85) {
                    result = 6;
                } else {
                    result = 1;
                }
            }
        }
        return result;
    }
```

3.4 二叉检索树

二叉检索树的作用

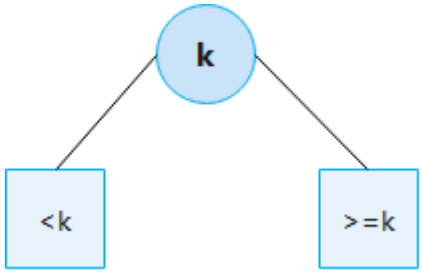
[1] 提供了查找元素花费logn的时间能力

[2] 提供了插入和删除元素花费logn的时间能力

对于10000个数据

> ■使用线性表查找元素平均需要比较5000次
>
> ■使用二叉检索树查找元素平均则只需要14次

定义：二叉检索树的任何一个结点，设其值为K，则该结点左子树中任意一个结点的值都小于K；该结点右子树中任意一个结点的值都大于或等于K。



接口与结点定义

对于BST，其接口定义如下

```
public interface BSTADT <K extends Comparable<K>, V> {
    public void insert(K key, V value);//插入结点
    public V remove(K key);//根据key值删除结点
    public boolean update(BinNode<K, V> rt, K key, V value);//更新结点值
    public V search(K key);//搜索key所对应结点的value
    public int getHeight(BinNode<K, V> rt);//获得树高
    public boolean isEmpty();//判断是否为空
    public void clear();//清空树
}
```

树的结点定义如下

```
public class BinNode<K extends Comparable<K>, V> {
    private K key;
    private V value;

    private BinNode<K, V> left;
    private BinNode<K, V> right;

    public BinNode(K key, V value){
        left = right = null;
        this.key=key;
        this.value=value;
    }

    public BinNode() {
    }

    public boolean isLeaf() {
        return left == null && right == null;
    }

    public K getKey() {
        return key;
    }
```

```
    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }

    public BinNode<K, V> getLeft() {
        return left;
    }

    public void setLeft(BinNode<K, V> left) {
        this.left = left;
    }

    public BinNode<K, V> getRight() {
        return right;
    }

    public void setRight(BinNode<K, V> right) {
        this.right = right;
    }
}
```

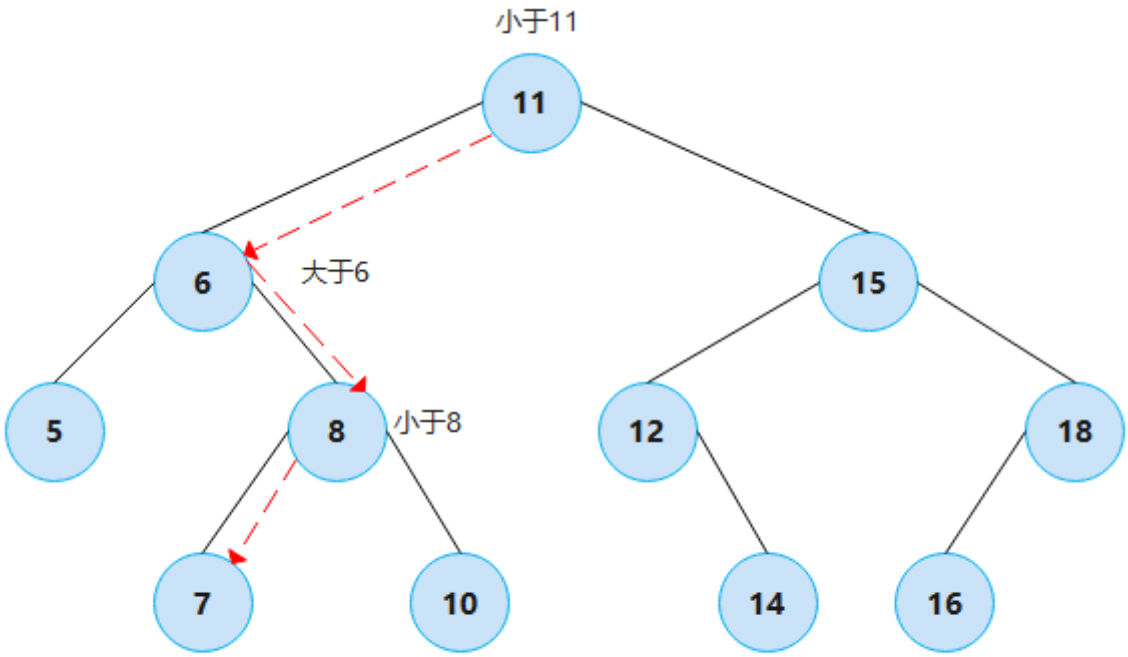
查找一个元素

在二叉检索树中查找一个元素的算法

[1] 设置当前结点指向根结点

[2] 重复以下步骤:

> ■如果当前结点为空，则退出，没有找到要匹配的元素
>
> ■如果当前结点所包含的元素的关键字大于要查找的元素的关键字，设当前结点指向其左子结点
>
> ■如果当前结点所包含的元素的关键字小于要查找的元素的关键字，设当前结点指向其右子结点
>
> ■否则，匹配元素找到，退出



二叉检索树查找7

```
public V search(K key) {
    return search(root, key);
}

private V search(BinNode<K, V> rt, K key) {
    try {
        if (key == null)
            throw new Exception("key is null");
        if (rt == null)
            return null;
        if (key.compareTo(rt.getKey()) < 0)
            return search(rt.getLeft(), key);//小于当前key值则往左子树查找
        if (key.compareTo(rt.getKey()) > 0)
            return search(rt.getRight(), key);//大于当前key值则往右子树查找
        return rt.getValue();//找到值
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

查找与删除最小元素

查找最小元素比较简单，只需要一直往左边递归查找，直到左子结点为空

删除最小元素思想如下


```
private K getMinNode(BinNode<K, V> rt) {
    if (rt.getLeft() == null)
        return rt.getKey();
    else
```

```
        return getMinNode(rt.getLeft());
}
//返回的是更新以后的根结点
private BinNode<K, V> removeMinNode(BinNode<K, V> rt) {
    if (rt.getLeft() == null) {
        return rt.getRight();
    }//最后一个结点的右结点，为空则返回空，否则返回结点
    rt.setLeft(removeMinNode(rt.getLeft()));//不断递归更新
    //保证了二叉检索树的规范性
    return rt;
}
```

插入一个元素

先寻找该插入的叶结点或者分支结点

插入的位置应该是所属位置父结点的空子结点

```
public void insert(K key, V value) {
    try {
        if (key == null) {
            throw new Exception("Key is null, insert fault!");
        }
        if (value == null) {
            throw new Exception("Value is null, insert fault!.");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    root = insertHelp(root, key, value);
}

private BinNode<K, V> insertHelp(BinNode<K, V> rt, K key, V value) {
    if (rt == null) {
        return new BinNode<K, V>(key, value);
    }

    if (key.compareTo(rt.getKey()) < 0) {
        rt.setLeft(insertHelp(rt.getLeft(), key, value));
    }//跟删除结点有异曲同工之妙

    else if (key.compareTo(rt.getKey()) > 0) {
        rt.setRight(insertHelp(rt.getRight(), key, value));
    }//跟删除结点有异曲同工之妙

    else {
        rt.setValue(value);
    }
    return rt;
}
```

平衡性

平衡因子：左右子树的高度之差

只要平衡因子的绝对值小于等于1，就说明这棵树是平衡的

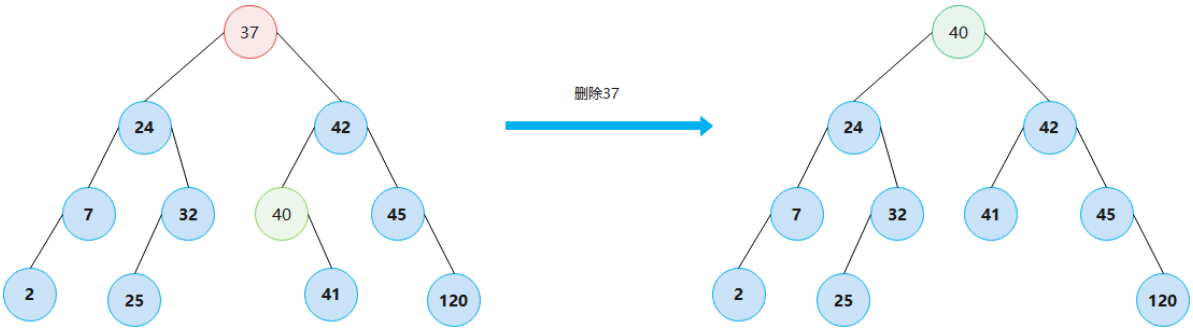
对于二叉检索树，如果给定的元素序列顺序性好，则平衡性很差、

> 可以通过旋转来解决平衡因子被破坏的情况（之后会细讲）

删除一个元素

若删除结点有两个结点，则比较特殊，要考虑二叉检索树的结构不被破坏

> 利用替换，将带有两个子结点的删除换成带有一个子结点的删除
>
> 将右子树中的最小结点替换待删除的结点




```
/**
 *
 * @param key 关键字
 * @return 删除的结点的值
 */
public V remove(K key) {
    removeValue = null;
    try {
        if (key == null)
            throw new Exception("Key is null, remove failure");
        root = removeHelp(root, key);
        return removeValue;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

private BinNode<K, V> removeHelp(BinNode<K, V> rt, K key) {
    if (rt == null)
        return null;
    if (key.compareTo(rt.getKey()) < 0) {
        rt.setLeft(removeHelp(rt.getLeft(), key));
    } else if (key.compareTo(rt.getKey()) > 0) {
        rt.setRight(removeHelp(rt.getRight(), key));
    } else {
        if (rt.getLeft() == null) {
            removeValue = rt.getValue();
            rt = rt.getRight();
            //左子结点为空，直接将右子结点作为当前根结点
        } else if (rt.getRight() == null) {
            removeValue = rt.getValue();
            rt = rt.getLeft();
            //右子结点为空，直接将左子结点作为当前根结点
        } else {
            //待删除结点有两个子结点
```

```
            rt.setKey((K) getMinNode(rt.getRight()).getKey());
            rt.setValue((V) getMinNode(rt.getRight()).getValue());
            //将当前结点的key和value更新为右子树中的最小结点的值
            rt.setRight(removeMinNode(rt.getRight()));
            //将当前结点的右子结点进行更新
        }
    }
    return rt;
}
```

各个时间代价

搜索代价

> 平衡二叉检索树的操作代价为O(logn)
>
> 非平衡的二叉检索树最差的代价为O(n)

插入、删除的代价与搜索代价类同

周游一个二叉检索树的代价为O(n)

使一个二叉检索树保持平衡才能真正发挥二叉检索树的作用

源代码

```
/**
 * @author yjq
 * @version 1.0
 * @date 2021/11/20 22:51
 */

public class BinarySearchTree<K extends Comparable<K>, V> implements BSTADT<K,
V> {
    private BinNode<K, V> root;
    private V removeValue;

    public BinarySearchTree() {
        root = null;
    }

    public BinNode<K, V> getRoot() {
        return root;
    }

    public void insert(K key, V value) {
        try {
            if (key == null) {
                throw new Exception("Key is null, insert fault!");
            }
            if (value == null) {
                throw new Exception("Value is null, insert fault!.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        root = insertHelp(root, key, value);
    }
```

```
private BinNode<K, V> insertHelp(BinNode<K, V> rt, K key, V value) {
    if (rt == null) {
        return new BinNode<K, V>(key, value);
    }

    if (key.compareTo(rt.getKey()) < 0) {
        rt.setLeft(insertHelp(rt.getLeft(), key, value));
    }//跟删除结点有异曲同工之妙

    else if (key.compareTo(rt.getKey()) > 0) {
        rt.setRight(insertHelp(rt.getRight(), key, value));
    }

    else {
        rt.setValue(value);
    }//key值相同则更新
    return rt;
}

/**
 *
 * @param key 关键字
 * @return 删除的结点的值
 */
public V remove(K key) {
    removeValue = null;
    try {
        if (key == null)
            throw new Exception("Key is null, remove failure");
        root = removeHelp(root, key);
        return removeValue;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

private BinNode<K, V> removeHelp(BinNode<K, V> rt, K key) {
    if (rt == null)
        return null;
    if (key.compareTo(rt.getKey()) < 0) {
        rt.setLeft(removeHelp(rt.getLeft(), key));
    } else if (key.compareTo(rt.getKey()) > 0) {
        rt.setRight(removeHelp(rt.getRight(), key));
    } else {
        if (rt.getLeft() == null) {
            removeValue = rt.getValue();
            rt = rt.getRight();
            //左子结点为空，直接将右子结点作为当前根结点
        } else if (rt.getRight() == null) {
            removeValue = rt.getValue();
            rt = rt.getLeft();
            //右子结点为空，直接将左子结点作为当前根结点
        } else {
            //待删除结点有两个子结点
            rt.setKey((K) getMinNode(rt.getRight()).getKey());
            rt.setValue((V) getMinNode(rt.getRight()).getValue());
```

```
                //将当前结点的key和value更新为右子树中的最小结点的值
                rt.setRight(removeMinNode(rt.getRight()));
                //将当前结点的右子结点进行更新
            }
        }
        return rt;
    }

    private BinNode getMinNode(BinNode<K, V> rt) {
        if (rt.getLeft() == null)
            return rt;
        else
            return getMinNode(rt.getLeft());
    }

    //返回的是更新以后的根结点
    private BinNode<K, V> removeMinNode(BinNode<K, V> rt) {
        if (rt.getLeft() == null) {
            return rt.getRight();
        }
        rt.setLeft(removeMinNode(rt.getLeft()));
        //保证了二叉检索树的规范性
        return rt;
    }

    public V search(K key) {
        return search(root, key);
    }

    private V search(BinNode<K, V> rt, K key) {
        try {
            if (key == null)
                throw new Exception("key is null");
            if (rt == null)
                return null;
            if (key.compareTo(rt.getKey()) < 0)
                return search(rt.getLeft(), key);//小于当前key值则往左子树查找
            if (key.compareTo(rt.getKey()) > 0)
                return search(rt.getRight(), key);//大于当前key值则往右子树查找
            return rt.getValue();//找到值
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    public boolean update(K key, V value) {
        return update(root, key, value);
    }

    public boolean update(BinNode<K, V> rt, K key, V value) {
        try {
            if (key == null)
                throw new Exception("Key is null, update failure.");
            if (value == null)
                throw new Exception("value is null, update failure");
            if (key.compareTo(rt.getKey()) == 0) {
                rt.setValue(value);
```

```
                return true;
            }
            if (key.compareTo(rt.getKey()) < 0)
                return update(rt.getLeft(), key, value);
            if (key.compareTo(rt.getKey()) > 0)
                return update(rt.getRight(), key, value);
            return false;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    public boolean isEmpty() {
        return root == null;
    }

    public void clear() {
        root = null;
    }


    public int getHeight(BinNode<K, V> rt) {
        int height = 0;
        if (rt == null)
            return 0;
        else
            height++;
        height += Math.max(getHeight(rt.getLeft()), getHeight(rt.getRight()));
        return height;
    }

}
```

3.5 优先队列

概述

定义

> 按照重要性和优先级来组织的对象称为优先队列
>
> 是一种ADT

应用

> 在多用户的环境中，操作系统调度程序必须决定在若干进程中运行哪个进程
>
> 发话到打印机中的若个作业可能在某些时候并不想按照先来先打印的方式运行

优先队列所需要的操作

> **插入**: 增加一个带有重要级别的元素，插入到队列中的位置并不在意
>
> **删除**: 队列中的重要级别最高的那个元素
>
> **获得头元素**: 队列中的重要级别最高的那个元素

一般队列

> 插入：增加一个元素，这个元素被插入到队列中队尾
>
> 删除：剩除一个队列中队头的那个元素
>
> 获得头元素：获得队列中队头的那个元素

实现方式

使用排序的线性表

插入元素

> 在线性表中按照重要级别扫描到合适的位置处，然后将该元素插入，耗时O(n)

删除元素

> 在线性表中直接删除头位置的元素即可

使用二叉查找树

插入元素

> 在二叉树平衡的情况下：O(log n)

删除元素

> 在二叉树平衡的情况下：O(log n)

优先队列的本身的特点并不需要二叉查找树的一些基本操作

使用堆实现优先队列

堆的两条性质

[1] 从结构性质看,堆是一棵**完全二叉树**,故可以用数组代替链表形式来实现之

[2] 从堆序性质看，堆能够**快速的找出重要级别最高的元素**

> 重要级别最高的元素是根元素,对根元素的访问是最快的获取速度
>
> 根据二叉树的递归定义，我们考虑任意子树也应该是堆，那么应该有下面的结论
>
> > 在堆中，对于每一个结点X。X的**父亲的重要级别高于(或等于）X的重要级别。**除了根结点之外(该结点没有父亲)

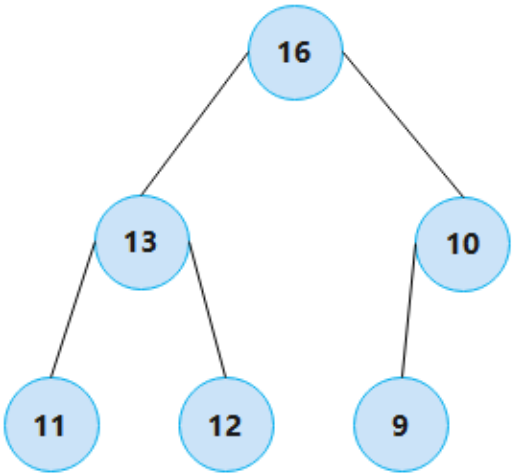
堆

最大值堆(本文中的都是大顶堆)

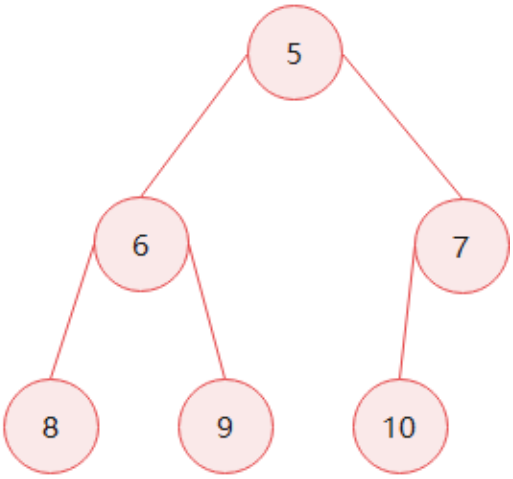
> 任意一个结点的关键字值都大于或者等于其任意一个子结点存储的值

最小值堆

> 任意一个结点的关键字值都小于或者等于其任意一个子结点存储的值



最大值堆



最小值堆

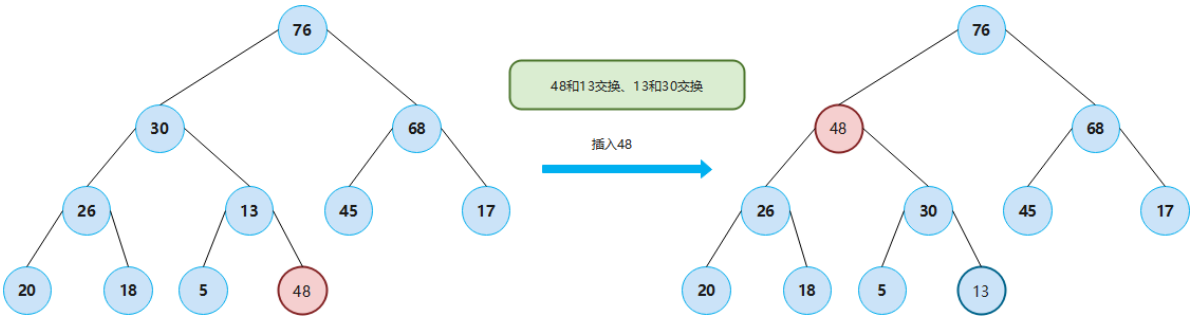
插入元素

插入过程

[1] 将要插入的元素插入到堆中的“最后一个元素”

[2] 重复做如下步骤

> 比较这个元素和其父的重要性
>
> 满足堆的性质则结束，否则将这个元素和其父元素交换
>
> 如果这个元素已经成为根元素也结束




```
/**
 * 往堆中插入元素
 *
 * @param number 元素值
 */
public void insert(int number) {
    try {
        if (isFull()) {
            throw new Exception("Array is full!");
        }
        int temp = ++currentSize;
        array[temp] = number;//将number放在数组最后
        while ((temp != 1) && (array[temp] > array[getParent(temp)])) {
            swap(array, temp, getParent(temp));
            temp = getParent(temp);//如果比父结点的值大则于其交换
        }//注意根结点的下标为1，不是0
```

```
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

删除元素

删除: 队列中的重要级别最高的那个元素

删除过程

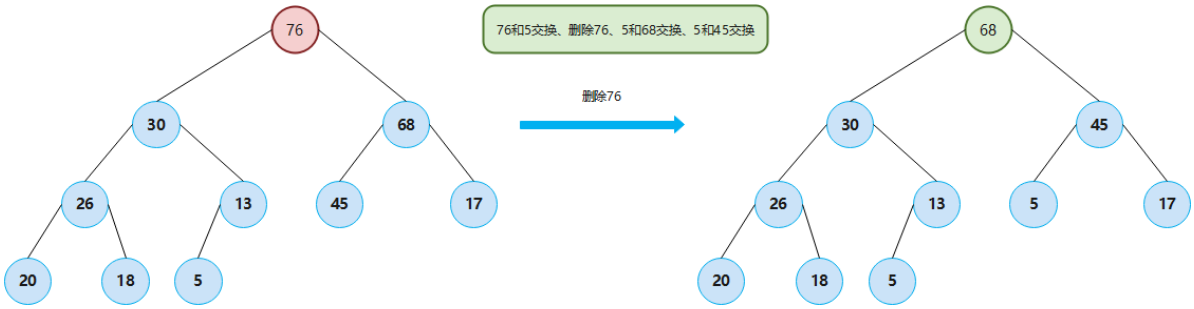
[1] 保留根结点所维护的元素

[2] 将“最后”结点的元素拷贝到根结点

[3] 删除“最后”结点

[4] 重复做如下步骤:

> 将这个结点与它的孩子们进行重要性的比较
>
> 满足堆的性质则结束，否则与重要级别高的孩子结点进行交换
>
> 当这个结点成为叶结点的时候结束




```
/**
 * 删除堆顶元素
 */
public void deleteMax() {
    try {
        if (isEmpty()) {
            throw new Exception("Array is empty!");
        } else {
            swap(array, 1, currentSize--);//将堆顶元素放到最后,同时删除
            if (currentSize != 0) siftDown(1);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * 小的值下沉
 *
 * @param pos 当前位置
 */
private void siftDown(int pos) {
    try {
```

```
        if (pos < 0 || pos > currentSize) {
            throw new Exception("Illegal position!");
        }
        while (!isLeaf(pos)) {
            int j = getLeft(pos);
            if ((j < currentSize) && (array[j] < array[j + 1])) j++;
            //跟子树中最大的值交换
            if (array[pos] > array[j]) return;
            //当前值已经比子树中的值都大，则返回
            swap(array, pos, j);//交换
            pos = j;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

创建堆

[1] 可以按照一个元素一个元素的方式插入

> 时间代价是O(nlogn)

[2] 按照堆可以被存放到数组这种特性，当所有的元素都已存入到数组时，我们可以采取更高效的策略

> 时间代价是**O(n)**

利用数组创建堆

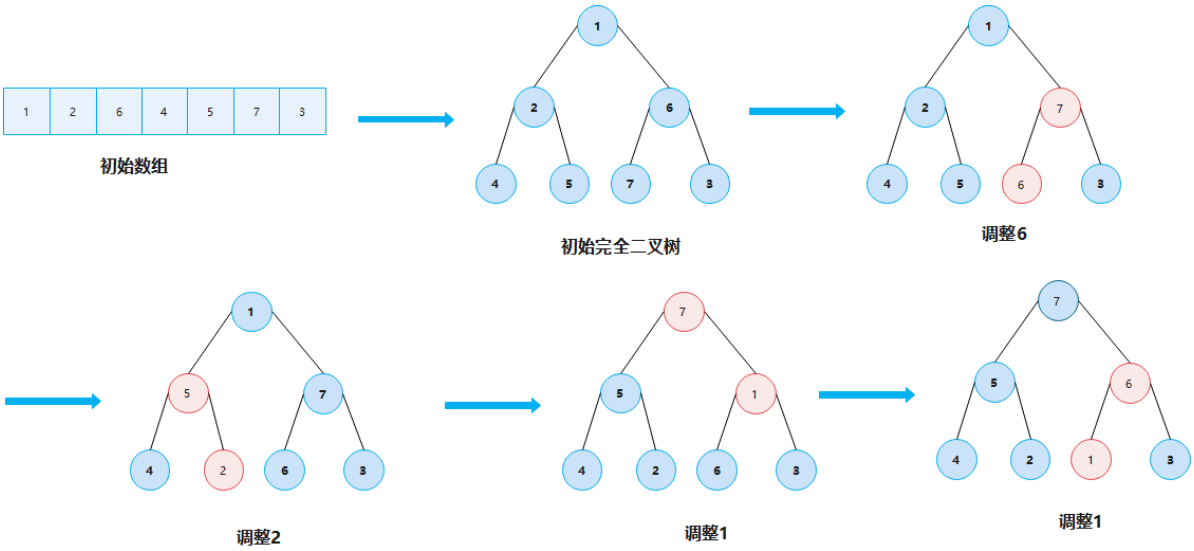
对于数组中任意位置i上的元素，其左儿子在位置2i上，右儿子在左儿子后的单元（2i+1）中，它的父亲则在i/2取下整上

```
private int getLeft(int i) {
    return 2 * i;
}

private int getRight(int i) {
    return 2 * i + 1;
}

private int getParent(int i) {
    return i / 2;
}}
```

对于完全二叉树，叶结点近乎占了一半，所以对于初始化的数组来说，其中有一半以上的元素满足堆序

调整顺序从下到上，从右到左，调整不满足堆序的结点

```
private void buildHeap() {
    for (int i = currentSize / 2; i > 0; i--) {
        siftDown(i);//对每个非叶子结点进行下沉操作
        //从右到左，从下到上
    }
}
```



堆的应用

可以用来进行排序：O(N)+O((N-1)logN)=O(NlogN)

适用频繁增加删除的情况：如Huffman树的创建

代码实现

一个完整的大顶堆实现如下

```
public class MaxHeap {
    private static final int DEFAULT_CAPACITY = 10;//默认大小
    private int currentSize;//当前堆的大小
    private int[] array;//堆数组

    public MaxHeap() {
        this.array = new int[DEFAULT_CAPACITY + 1];
        currentSize = 0;
    }

    public MaxHeap(int size) {
        this.array = new int[size + 1];
        currentSize = 0;
    }

    public MaxHeap(int[] array) {

        this.array = new int[array.length + 1];
        for (int i = 0; i < array.length; i++) {
            this.array[i + 1] = array[i];
        }//从1开始算
        currentSize = array.length;
        buildHeap();
    }

    /**
     * 往堆中插入元素
     *
     * @param number 元素值
```

```
     */
    public void insert(int number) {
        try {
            if (isFull()) {
                throw new Exception("Array is full!");
            }
            int temp = ++currentSize;
            array[temp] = number;//将number放在数组最后
            while ((temp != 1) && (array[temp] > array[getParent(temp)])) {
                swap(array, temp, getParent(temp));
                temp = getParent(temp);//如果比父结点的值大则于其交换
            }//注意根结点的下标为1，不是0
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public int findMax() {
        return array[1];
    }

    /**
     * 删除堆顶元素
     */
    public void deleteMax() {
        try {
            if (isEmpty()) {
                throw new Exception("Array is empty!");
            } else {
                swap(array, 1, currentSize--);//将堆顶元素放到最后,同时删除
                if (currentSize != 0) siftDown(1);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 小的值下沉
     *
     * @param pos 当前位置
     */
    private void siftDown(int pos) {
        try {
            if (pos < 0 || pos > currentSize) {
                throw new Exception("Illegal position!");
            }
            while (!isLeaf(pos)) {
                int j = getLeft(pos);
                if ((j < currentSize) && (array[j] < array[j + 1])) j++;
                //跟子树中最大的值交换
                if (array[pos] > array[j]) return;
                //当前值已经比子树中的值都大，则返回
                swap(array, pos, j);//交换
                pos = j;
            }
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }

    public void print() {
        for (int i = 1; i <= currentSize; i++) {
            System.out.print(array[i] + " ");
        }
    }

    public void heapSort() {
        while (currentSize != 0) {
            System.out.print(findMax() + " ");
            deleteMax();
        }
    }

    private boolean isEmpty() {
        return currentSize == 0;
    }

    private boolean isFull() {
        return currentSize == array.length - 1;
    }


    private boolean isLeaf(int i) {
        return i > currentSize / 2;
    }

    private void buildHeap() {
        for (int i = currentSize / 2; i > 0; i--) {
            siftDown(i);//对每个非叶子结点进行下沉操作
            //从右到左，从下到上
        }
    }

    private int getLeft(int i) {
        return 2 * i;
    }

    private int getRight(int i) {
        return 2 * i + 1;
    }

    private int getParent(int i) {
        return i / 2;
    }

    private void swap(int[] array, int x, int y) {
        int temp = array[y];
        array[y] = array[x];
        array[x] = temp;
        return;
    }


    public static void main(String[] args) throws Exception {
        int[] test = {1, 2, 6, 4, 5, 7, 3};
```

```
        MaxHeap maxHeap = new MaxHeap(test);
        maxHeap.print();
        maxHeap.deleteMax();
        System.out.println();
        System.out.println("delete 7:");
        maxHeap.print();
        maxHeap.insert(7);
        System.out.println();
        System.out.println("insert 7:");
        maxHeap.print();
        System.out.println();
        System.out.println("heapsort: ");
        maxHeap.heapSort();
    }
}
```

测试结果如下:

```
7 5 6 4 2 1 3
delete 7:
6 5 3 4 2 1
insert 7:
7 5 6 4 2 1 3
heapsort:
7 6 5 4 3 2 1
```

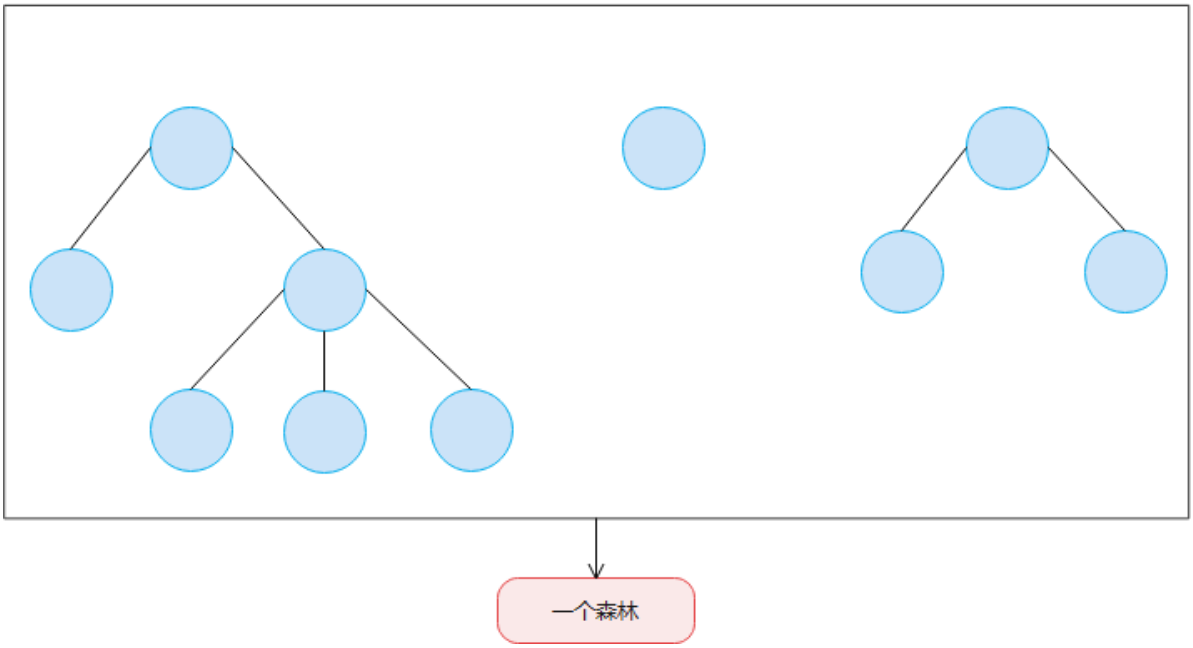
3.6 树、森林和并查集[图文详解]

树的定义与术语

树的定义：一棵树T是由一个或一个以上结点组成的有限集，其中有一个特定的结点R称为T的根结点。集合(T-{R}）中的其余结点可被划分为n>=0个不相交的子集T、T2、...、Tn，其中每个子集都是树，并且其相应的根结点R1、R2、 ... 、Rn是R的子结点。

> 子集T(1≤ i ≤n）称为树T的子树(subtree)
>
> 结点的出度(out degree）定义为该结点的子结点的数目

森林的定义：零个或多个树的一个有序集合

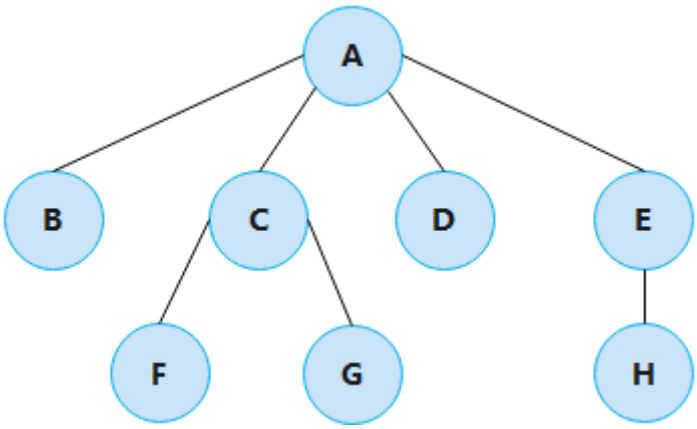


树结点ADT

```
public interface GTNode {
    //树结点ADT
    Object value();
    boolean isLeaf();
    GTNode getParent();
    GTNode getLeftMostChild();
    GTNode getRightSibling();
    void setValue(Object value);
    void setParent(GTNode parent);
    void insertFirst(GTNode first);
    void insertNext(GTNode next);
    void removeFirst();
    void removeNext();
}
```

树的遍历

跟二叉树的遍历类似



前序周游：先访问结点，后访问其子结点。上图前序周游的结果为ABCFGDEH

```java
/**
 * 从根结点开始先序遍历
 * @param rt 根结点
 */
private static void preOrder(GTNode rt){
    if (rt==null)return;
    visit(rt);
    GTNode temp = rt.getLeftMostChild();
    while (temp!=null){
        preOrder(temp);
        temp = temp.getRightSibling();
    }
}
```

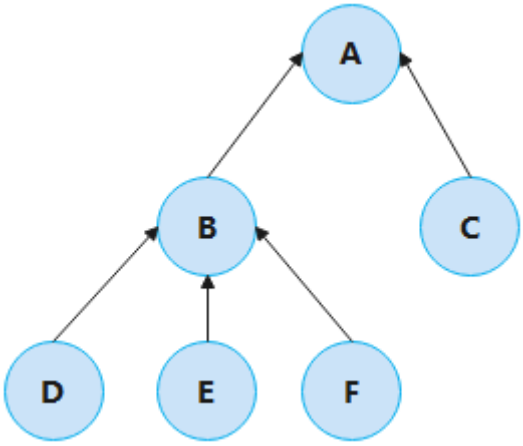
后序周游：先访问结点的子结点（包括他们的子树），然后再访问该结点。上图二叉树经过后序周游枚举出来的结果为：BFGCDHEA

```java
/**
 * 从根结点开始后序遍历
 * @param rt 根结点
 */
private static void postOrder(GTNode rt){
    if (rt==null)return;
    GTNode temp = rt.getLeftMostChild();
    while (temp!=null){
        postOrder(temp);
        temp = temp.getRightSibling();
    }
    visit(rt);
}
```

树的实现

父指针表示法

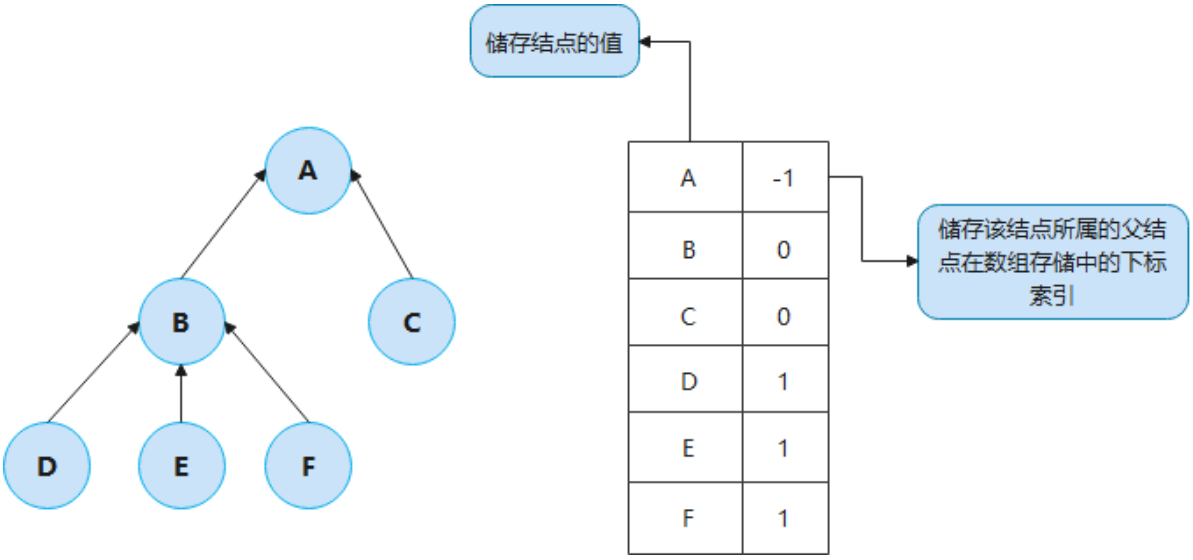
每个结点只保存一个指针域指向其父结点



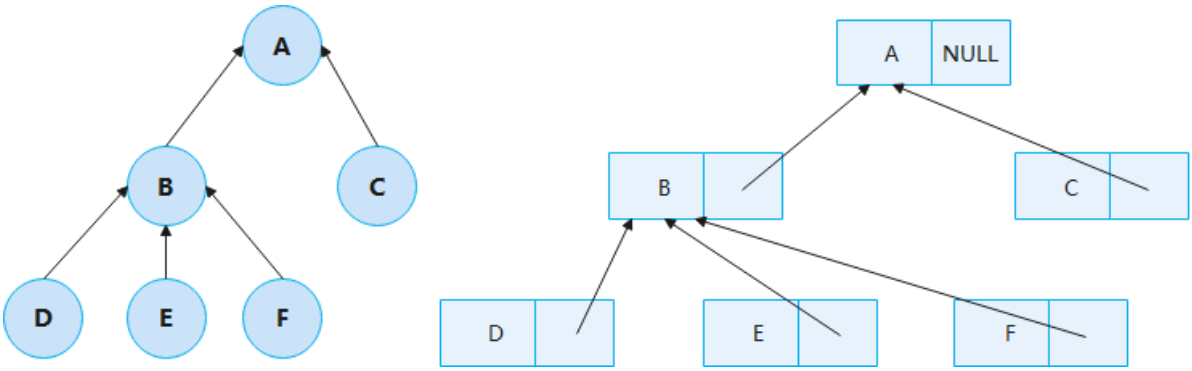
适用的应用：等价类问题的处理（并查集）

缺点：对找到一个结点的最左子结点或右侧兄弟结点这样的重要操作是不够的

数组实现形式



链表实现方式



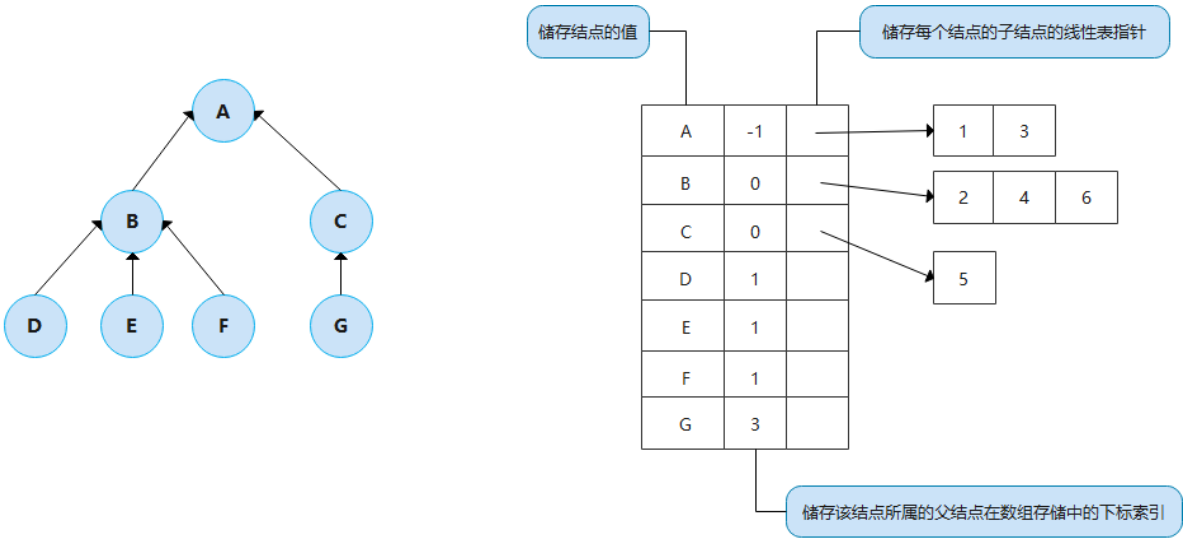
子结点表表示法

每个结点存储一个线性表的指针，该线性表用来存储该结点的所有子结点

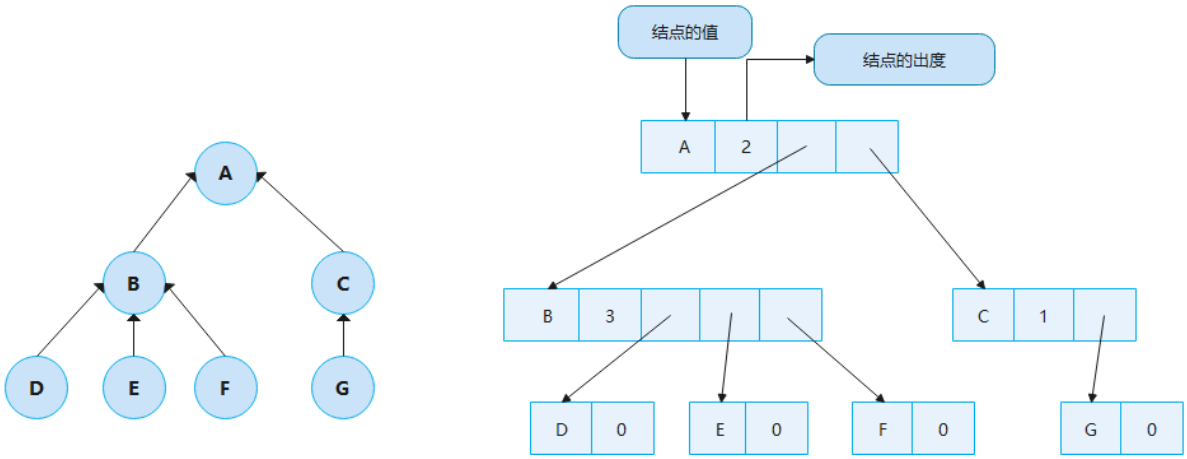
优势：寻找某个结点的子结点非常方便

缺点：寻找某个结点的兄弟结点比较困难

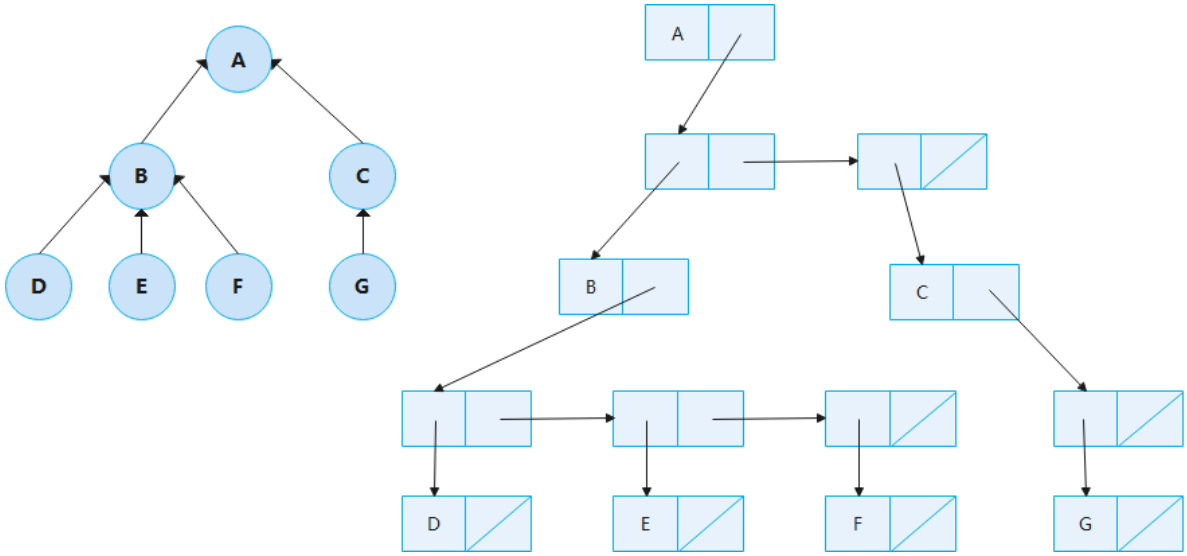
数组实现形式



链表实现方法1



链表实现方式2

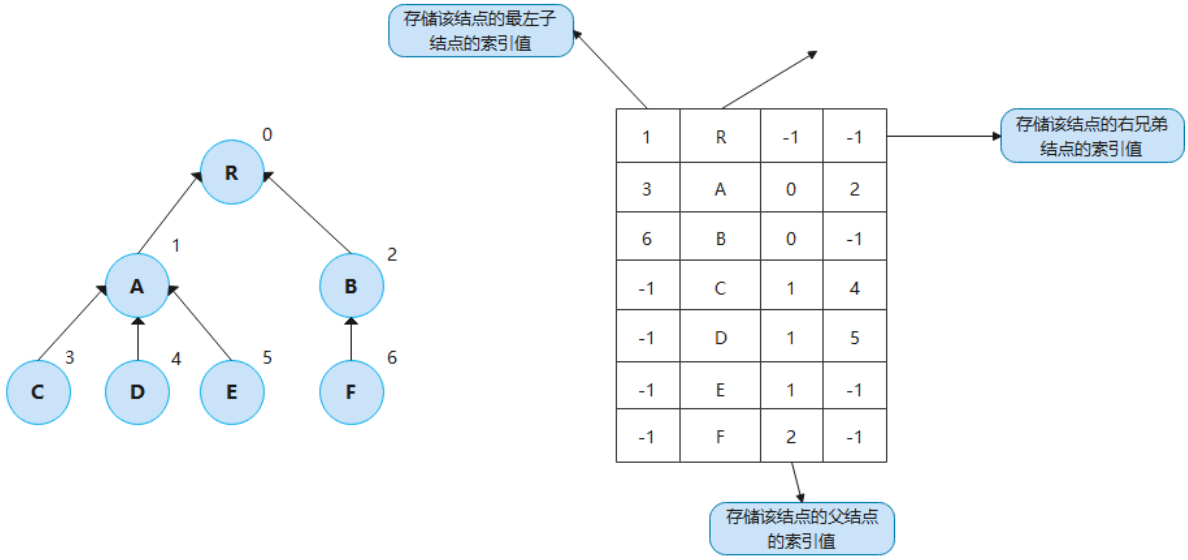


左子结点/右兄弟结点表示法

每个结点都存储**结点的值、最左子结点的位置和右侧兄弟结点的位置**

优点：ADT中规定的基本操作都可以较为容易的实现

数组实现方式

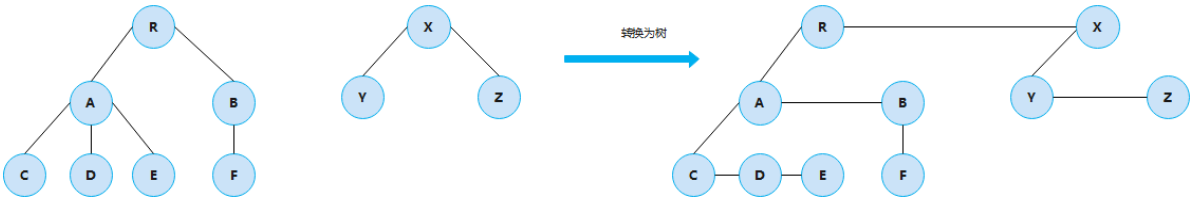


链表实现方式

这种表示方式和二叉树的链表表示方式在物理结构上是一致的

森林和二叉树的转换关系

将森林中的根节点连接起来，并且将每个结点的子结点之间连接起来，最后去掉除每个结点与最左子结点之间的连接线之外的其余连线



结论：任何二叉树都对应一个唯一的森林

> 根结点加左子树对应第一棵树，右子树对应新的森林构成的树，以次递归定义

设F=(T1，T2，T3，....Tn,)是树的一个森林，对应于F的二叉树B(F)的**严格定义**如下：

> 如果n=0，则B(F)为空
>
> 如果n#0，则B(F)的根是$root(T_1)$; B(F)的左子树是$B(T_{11}，T_{12}....T_{1m})$，其中$T_{11}，T_{12}....T_{1m}$是$T_1$树的子树:B(F)的右子树是$B(T_2,T_3.....T_n)$

森林的遍历

将树的根去掉之后，就成为了森林，所以树和森杯的遍历本质是和同的

深度优先后根遍历

[1] 若森林$F=\varnothing$，返回

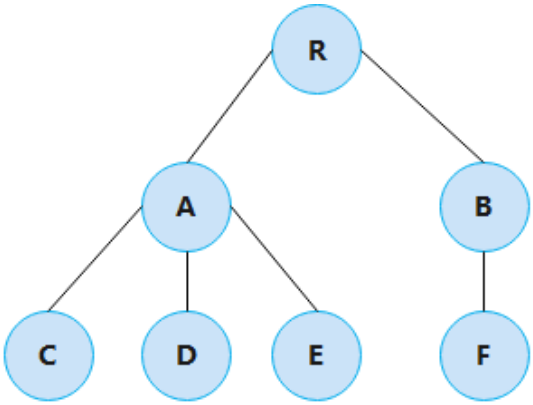
[2] 否则后根遍历森林F第一棵树的根结点的子树森林$T_{11}，T_{12}....T_{1m}$

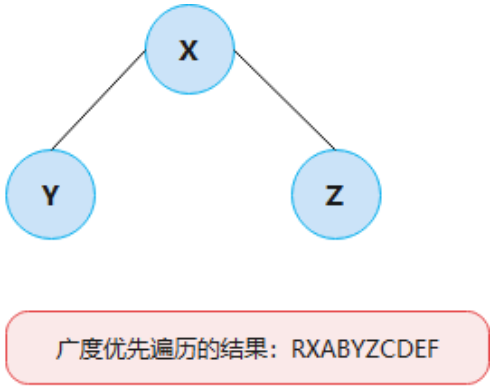
[3] 访问森林F第一棵树的根结点r1

[4] 后根遍历森林中除第一棵树外其他树组成的森林$T_2,T_3.....T_n$

广度优先遍历

[1] 若森林F为空，返回
[2] 否则依次遍历各棵树的根结点
[3] 依次遍历各棵树根结点的所有子女
[4] 依次遍历这些子女结点的子女结点





不相交集ADT（并查集）

不相交集合：**是由一组互不相交的集合组成的一个集合结构**，并在此集合上定义了运算Union和Find

每一个要处理的元素都仅仅属于一个集合

> 集合之间是不相交的
>
> 一开始，每个集合包含一个元素

每一个集合都有一个名称,这个名称可以用该集合中的任何一个元素名称

用途：主要用来解决等价问题

若对于每一对元素(a，b)，a和b之间满足如下三种关系，则称a和b之间是**等价关系**

> 1 自反性: aRa
>
> 2 对称性: aRb当且仅当bRa
>
> 3 传递性:若aRb且bRc则aRc

生活中的等价关系：电器连通性、城市之间的连通性等

需要支持的两个操作

Find(elementname)

> 返回包含给定元素的集合名字
>
> 不同于查找方式中的返回结果

Union(elementname1,elementname2)

> 生成一个新的集合，该集合是elementname1所属的集合set1和lelementname2所属的集合set2的并集

实现方式

使用数组

由一个具有n个元素组成的数组储存各个不相交的集合

初始状态:每个元素都隶属于一个集合，该集合的名字就是该元素在数组中的下标：set[i]=i

```
Union(i,j): 对每一个k，如果set[k]==下标为j的元素所属的集合名称，则设置set[k]=下标为i的元素所属的集合名称
Find(i): 返回set[i]即可
```

每个find操作的时间复杂性为O(1)，每个union操作的时间复杂性为O(N)，那么做N次union操作的时间复杂性就是O(N^2)

```
public class UnionFindArray {
    //用数组实现并查集
    public int[] set;

    /**
     * 初始化并查集，每个集合一个元素，其名称即元素值
     *
     * @param N 元素总个数
     */
    public void init(int N) {
        set = new int[N];
        for (int i = 0; i < N; i++) {
            set[i] = i;
        }
    }

    /**
     * 返回包含给定元素的集合名字
     *
     * @param i 给定元素
     * @return 集合名字
     */
    public int find(int i) {
        return set[i];
    }

    /**
     * 生成一个新的集合，该集合是i所属的集合set1和j所属的集合set2的并集
     *
     * @param i
     * @param j
     */
    public void union(int i, int j) {
        int setName1 = find(i);
        int setName2 = find(j);
        for (int k = 0; k < set.length; k++) {
            if (set[k] == setName2) {
                set[k] = setName1;
            }
        }
    }

    public void print() {
        for (int i = 0; i < set.length; i++) {
            System.out.print(set[i]+" ");
        }
    }

    public static void main(String[] args) {
        UnionFindArray test = new UnionFindArray();
        test.init(10);
        System.out.println("find(1): " + test.find(1));
```

```
        test.union(2,5);
        System.out.print("union(2,5): ");
        test.print();
        System.out.println();
        test.union(3,6);
        System.out.print("union(3,6): ");
        test.print();
        System.out.println();
        test.union(2,6);
        System.out.print("union(2,6): ");
        test.print();
    }
}
```

```
find(1): 1
union(2,5): 0 1 2 3 4 2 6 7 8 9
union(3,6): 0 1 2 3 4 2 3 7 8 9
union(2,6): 0 1 2 2 4 2 2 7 8 9
```

使用树

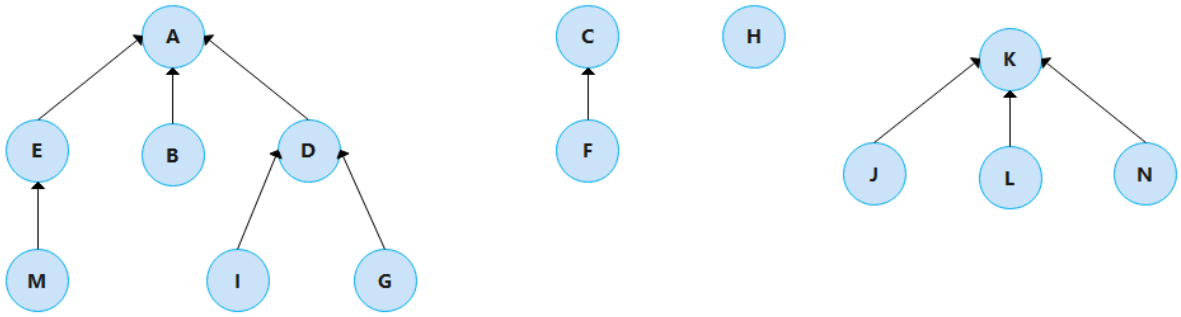
不相交集合可以表示一个森林

森林中的每棵树表示为一个集合

树中每个结点的存放顺序没有任何约束，所以可以采用树的父指针表示法来描述树

树的实现还是采用数组这种物理形式

-1	0	-1	0	0	2	3	-1	3	10	-1	10	4	10
A	B	C	D	E	F	G	H	I	J	K	L	M	N
0	1	2	3	4	5	6	7	8	9	10	11	12	13



数组中的每个元素存储树中的每个结点，结点应该包含父指针信息和所存储的元素内容

```
public class GTNodeA {
    private int parent;//父结点
    private Object element;//存储元素内容

    public GTNodeA() {
        this(null, -1);
    }

    public GTNodeA(Object element) {
        this(element, -1);
    }
```

```
    public GTNodeA(Object element, int parent) {
        this.element = element;
        this.parent = parent;
    }

    public int getParent() {
        return parent;
    }

    public int setParent(int parent) {
       return this.parent = parent;
    }

    public Object getElement() {
        return element;
    }

    public void setElement(Object element) {
        this.element = element;
    }
}
```

代码实现

```
public class UnionFindTree {
    //用树实现并查集
    public GTNodeA[] set;

    public UnionFindTree(GTNodeA[] set) {
        this.set = set;
    }

    /**
     * 返回包含给定元素的集合名字
     *
     * @param i 元素下标
     * @return 以根结点下标作为集合名字
     */
    public int find(int i) {
        GTNodeA current = set[i];
        while (current.getParent() >= 0) {
            i = current.getParent();//将下标变为父结点的下标
            current = set[i];
        }//所需时间依赖于第i个元素在树中的层次
        return i;
    }

    /**
     * 生成一个新的集合，该集合是i所属的集合set1和j所属的集合set2的并集
     *
     * @param i
     * @param j
     */
    public void union(int i, int j) {
        int root1 = find(i);
        int root2 = find(j);
```

```
        if (root1 != root2) {
            set[root2].setParent(root1);
            //将其中一棵树的根结点的父结点设置为
            //另一棵树的根结点
        }
    }

    public void print() {
        for (int i = 0; i < set.length; i++) {
            System.out.print(set[i].getElement() + " ");
        }
    }

    public static void main(String[] args) {
        GTNodeA node_A = new GTNodeA("A");
        GTNodeA node_C = new GTNodeA("C");
        GTNodeA node_H = new GTNodeA("H");
        GTNodeA node_K = new GTNodeA("K");
        GTNodeA node_E = new GTNodeA("E", 0);
        GTNodeA node_B = new GTNodeA("B", 0);
        GTNodeA node_D = new GTNodeA("D", 0);
        GTNodeA node_F = new GTNodeA("F", 2);
        GTNodeA node_J = new GTNodeA("J", 10);
        GTNodeA node_L = new GTNodeA("L", 10);
        GTNodeA node_N = new GTNodeA("N", 10);
        GTNodeA node_M = new GTNodeA("M", 4);
        GTNodeA node_I = new GTNodeA("I", 3);
        GTNodeA node_G = new GTNodeA("G", 3);
        GTNodeA[] test = {node_A, node_B, node_C, node_D,
                          node_E, node_F, node_G, node_H,
                          node_I, node_J, node_K, node_L,
                          node_M, node_N};
        UnionFindTree testTree = new UnionFindTree(test);
        System.out.print("initialized forest: ");
        testTree.print();
        System.out.println();
        System.out.println("find(4): "+testTree.find(4));
        System.out.println("find(5): "+testTree.find(5));
        testTree.union(0,2);
        System.out.print("union(0,2),find(5): "+testTree.find(5));
    }
}
```

```
initialized forest: A B C D E F G H I J K L M N
find(4): 0
find(5): 2
union(0,2),find(5): 0
```

重量平衡原则

为了使union N个元素的时间复杂性降低到O(NlogN)，可以使用**重量平衡原则**

当两个集合合并的时候(也就是两棵树合并为棵树)，可以将结点数小的那棵树合并到结点数较多的那棵树上

统计结点数的巧妙方法

> 初始化时，每个结点的父结点下标都是-1，如A,B,C都是-1，合并AB以后，设A的父结点下标为-2，其绝对值则为以A为根结点的树的结点数

```
    /**
     * 生成一个新的集合，该集合是i所属的集合set1和j所属的集合set2的并集
     *
     * @param i
     * @param j
     */
    public void union(int i, int j) {
        int root1 = find(i);
        int num1 = set[root1].getParent();
        int root2 = find(j);
        int num2 = set[root2].getParent();
        if (num1 <= num2) {
            set[root1].setParent(num1+num2);
            set[root2].setParent(root1);
            //将其中结点数少的一棵树的根结点的父结点设置为
            //结点数多的一棵树的根结点
        } else {
            set[root2].setParent(num1+num2);
            set[root1].setParent(root2);
            //将其中结点数少的一棵树的根结点的父结点设置为
            //结点数多的一棵树的根结点
        }
    }
```

```
    public static void main(String[] args) {
        GTNodeA node_A = new GTNodeA("A");
        GTNodeA node_C = new GTNodeA("C");
        GTNodeA node_H = new GTNodeA("H");
        GTNodeA node_K = new GTNodeA("K");
        GTNodeA node_E = new GTNodeA("E");
        GTNodeA node_B = new GTNodeA("B");
        GTNodeA node_D = new GTNodeA("D");
        GTNodeA node_F = new GTNodeA("F");
        GTNodeA node_J = new GTNodeA("J");
        GTNodeA node_L = new GTNodeA("L");
        GTNodeA node_N = new GTNodeA("N");
        GTNodeA node_M = new GTNodeA("M");
        GTNodeA node_I = new GTNodeA("I");
        GTNodeA node_G = new GTNodeA("G");
        GTNodeA[] test = {node_A, node_B, node_C, node_D,
                node_E, node_F, node_G, node_H,
                node_I, node_J, node_K, node_L,
                node_M, node_N};
        UnionFindTree testTree = new UnionFindTree(test);
        System.out.print("initialized forest: ");
        testTree.print();
        System.out.println();
        System.out.println("find(1): " + testTree.find(1));
        System.out.println("find(3): " + testTree.find(3));
        System.out.println("find(4): " + testTree.find(4));
        testTree.union(0, 4);
        testTree.union(0, 1);
        testTree.union(0, 3);
```

```
        System.out.println("union(0,4), union(0,1), union(0,3)");
        System.out.println("find(1): " + testTree.find(1));
        System.out.println("find(3): " + testTree.find(3));
        System.out.println("find(4): " + testTree.find(4));
    }
```

```
initialized forest: A B C D E F G H I J K L M N
find(1): 1
find(3): 3
find(4): 4
union(0,4), union(0,1), union(0,3)
find(1): 0
find(3): 0
find(4): 0
```

路径压缩

在查找某个元素是否属于某个集合时，将该结点到根结点路径上所有结点的父指针全部改为指向根结点，这种方式**可以产生极浅的树**

```
/**
 * 返回包含给定元素的集合名字
 *
 * @param i 元素下标
 * @return 以根结点下标作为集合名字
 */
public int find(int i) {
    GTNodeA current = set[i];
    if (current.getParent()<0) return i;
    return current.setParent(find(current.getParent()));
}//使用路劲压缩法：在查找某个元素是否属于某个集合时，将该结点到根结点路径上
// 所有结点的父指针全部改为指向根结点，这种方式可以产生极浅的树
```

结合重量权衡原则来归并集合的话，对n个结点进行n次find操作的路径压缩开销为O(NlogN)

4.1 图的基本概念与实现

图的定义

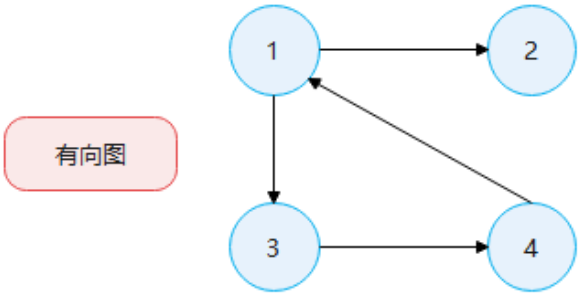
Graph= (V，E)

V={x|x是一个结点，表示一个数据元素}、

> 图中的数据元素x也称为顶点（vertex)
>
> V是顶点的有穷非空集合

E={<x，y>|x，y都属于V，并且x连接到y}

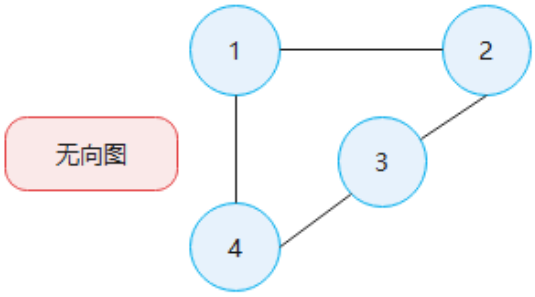
> E是两个顶点之间的关系的集合
>
> 如果<x, y>属于E，那么表示x到y的一条弧(arc)，且x称为尾(tail)，y称为头(head),具有这种性质的图称为**有向图(digraph)**
>
> 如果<x,y>属于E，必有<y,x>也属于E，则以无序对(x, y)代替这两个有序对，表示x和y之间的一条边(edge)，此时的图称为**无向图**(undigraph)



G1

V(G1)={1,2,3,4}

E(G1)={<1,2>,<1,3>,<3,4><4,1>}



G2

V(G2)={1,2,3,4}

E(G1)={<1,2>,<1,4>,<3,4><2,3>}

术语

完全图

用|V|表示G中顶点的个数，用|E|表示G中弧或者边的个数

当不考虑顶点到其自身的弧或者边时，那么|V|和|E|乏间有如下关系:

> 1 如果G是无向图，|E|的取值范围是0到|V|(|V|-1)/2
>
> 当|E|为最大值时，该无向图称为**完全图(completedgraph)**
>
> 2 如果G是有向图，|E|的取值范围是0到|V|(|V|-1)
>
> 当|E|为最大值时，该有向图称为**有向完全图**
>
> 3 如果G中有少量的边或弧(如|E|<|V| log|V|)时，称该G为**稀疏图**(sparse graph),反之称为稠密图（dense graph）

当G中的边或弧具有与它相关的数,这种与图的边或弧相关的数叫做**权(weight)**,这种带权的图通常称为**网(network)**

顶点相关概念

对于**无向图G**，如果边(v,u)属于E，则称顶点v和u互为**邻接点(adjacent)**,并且称边(v,u)与顶点和u相关联(incident)

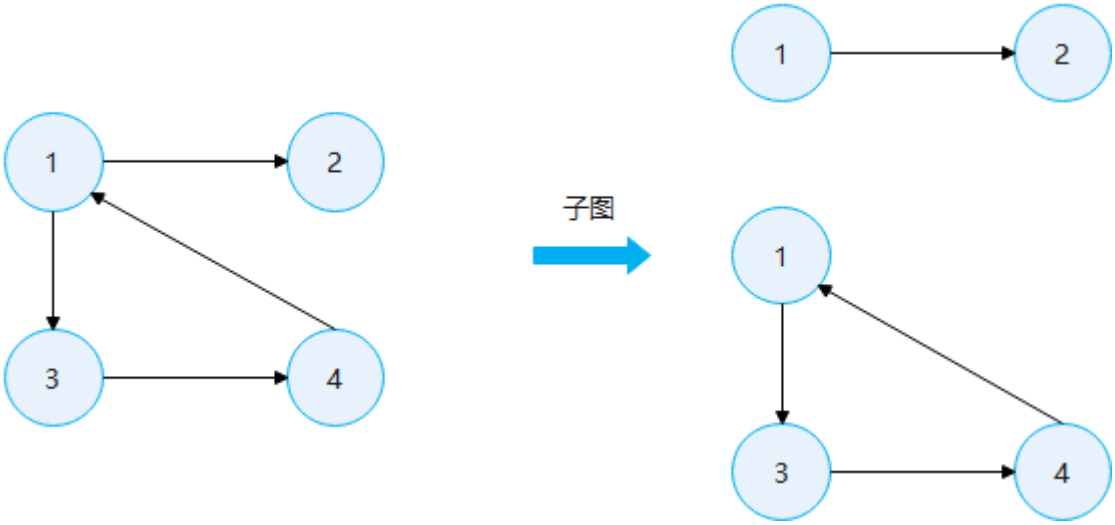
> 顶点v的度(degree)是和v相关联的边的个数

对于**有向图G**，如果弧<v,u>属于E，则称顶点v邻接到(to)顶点u，顶点u邻接自(from)顶点v，并且称弧<v,u>与顶点v、u相关联

> 以顶点v为头的弧的数目称为v的入度(indegree)
>
> 以顶点v为尾的弧的数目称为v的出度(outdegree)
>
> 顶点v的度=顶点v的入度+顶点v的出度

子图

假设有两个图G=(V,E)和G'=(V' ,E'),如果V'是V的子集，且E'是E的子集,称G'是G的子图



路径相关概念

在G=(V，E)中，当从v到u存在如下的边或弧时,我们称这是v到u的一条路径，该路径的长度则为这些边或弧的个数

> 1 当G是无向图时，边是: $(v, v_1), (v_1, v_2), \ldots, (v_n, u)$
>
> 2 当G是有向图时，弧是: $< v, v_1 >, < v_1, v_2 >, \ldots, < v_n, u >$

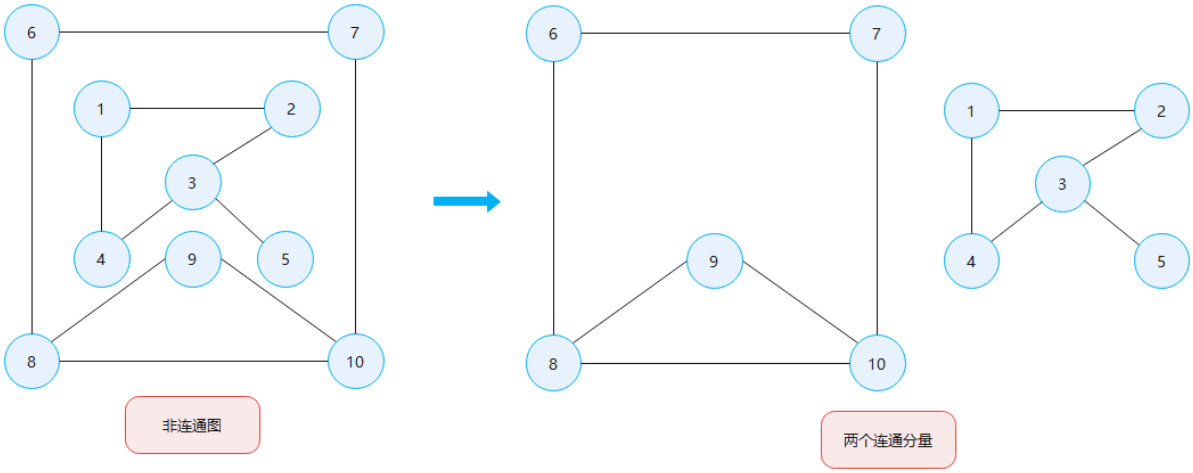
当在一条路径中出现的顶点不重复出现时则称该路径为**简单路径(simple path)**

图的连通

在一个图G中，如果从顶点v到顶点u有路径，则称v和u是连通的(connected)

如果G是**无向图**。那么当G中的任意两个顶点vi和vj，都有vi和vj是连通的，则称G是**连通图**(connected graph)

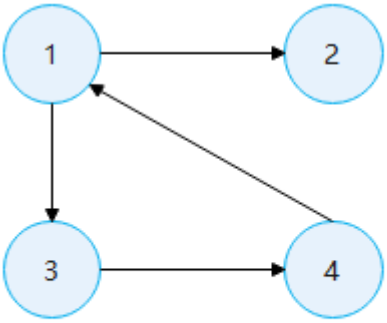
当无向图G不是一个连通图时，那么该无向图的**极大连通子图**则称为连通分量(connected component)



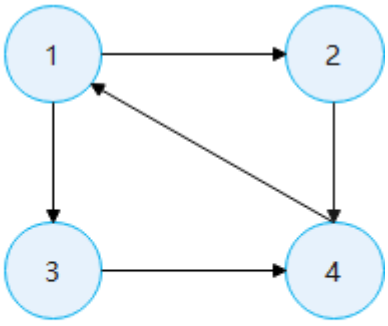
如果G是有向图，对于每一对顶点vi和vj，都有从vi到vj和vj到vi的路径，则称该G是**强连通图**

如果G不是强连通图，但是将G中的弧想象为边时能成为一个连通图时，我们称这个有向图G为**弱连通**(weakly connected)

有向图中的**极大强连通子图称为有向图的强连通分量**



弱连通图



强连通图

如果图G中不存在环，那么称G为**无环图(acycle)**

一个无环图如果既是无向图也是连通图，则该图称为**自由树(free tree)**

一个无环图如果是有向图，则简称为**DAG(directed acyclic graph)**

图的实现方式

图有两种常见的表示方法

[1] 相邻矩阵

> 是由|V|* |V|个元素组成的矩阵
>
> 矩阵中某个坐标对所对应的元素值表示该坐标对所对应的顶点之间的关系

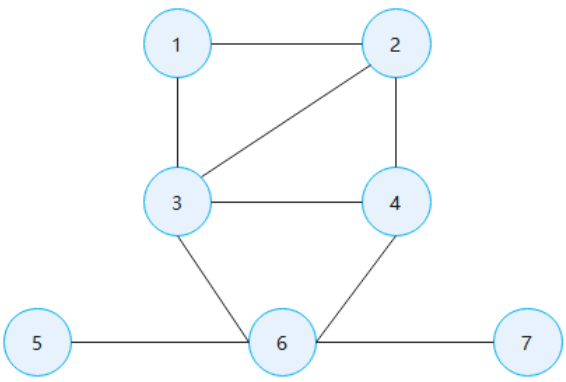
[2] 邻接表

> 是一个以链表为元素的数组
>
> 数组有|V|个元素，每个元素表示一个顶点第i个元素存储的链表中的内容为第i个顶点连接到的所有顶点信息

相邻矩阵

如果一个图有|V|个顶点，那么就定义一个具有|V|* |V|的矩阵M，M中的每个元素的取值如下（无权）

> $M_{ij} = 1$ <i,j>是一条弧
>
> $M_{ij} = 0$ <i,j>不是一条弧



相邻矩阵

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	1	1	0	0	0
3	1	1	0	1	0	1	0
4	0	1	1	0	0	1	0
5	0	0	0	0	0	1	0
6	0	0	1	1	1	0	1
7	0	0	0	0	0	1	0

有权的时候

> $M_{ij} = Weight_{ij}$ <i,j>是一条弧
>
> $M_{ij} = \infty$ <i,j>不是一条弧

时间复杂性分析

判断一个边是否存在：O(1)

寻找某个顶点所能到达的相邻顶点：O(|V|)

寻找所有的边：O(|V|^2)

增加或者删除一条边：O(1)

适用于稠密图

代码实现

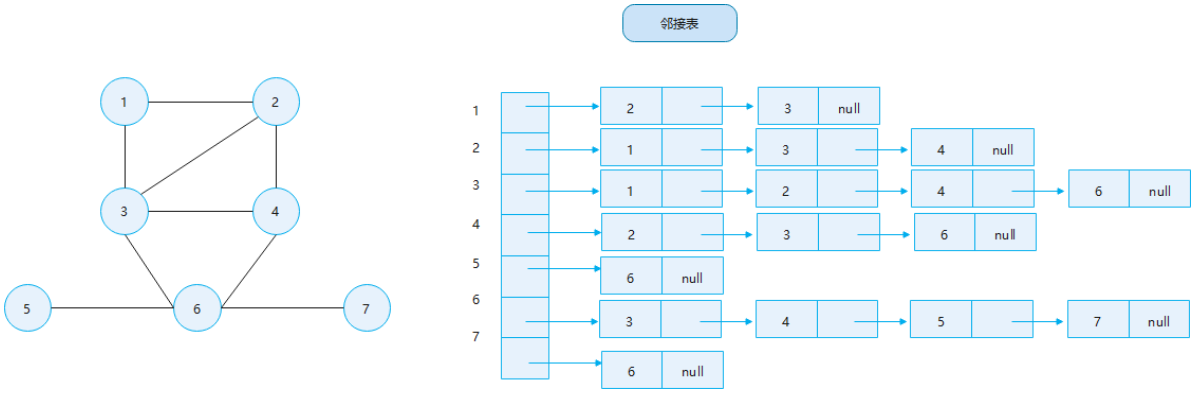
```
public class GraphM {
    //用相邻矩阵实现无向图
    private ArrayList<String> V;//顶点
    private int E;//边数
    private int[][] matrix;//相邻矩阵

    /**
     * 初始化无向图
     * @param n 结点数
     */
    public GraphM(int n) {
        matrix = new int[n][n];
        V = new ArrayList<String>(n);
    }

    /**
     * 添加结点
     * @param vertex 结点值
     */
    public void insertVertex(String vertex){
        V.add(vertex);
    }

    /**
     * 插入边
     * @param v1 起点下标
     * @param v2 终点下标
     * @param weight 权值，这里0表示不相邻，1表示相邻
     */
    public void insertEdge(int v1,int v2,int weight) {
        matrix[v1][v2] = weight;
        matrix[v2][v1] = weight;
        E++;//边数+1
    }

    /**
     * 获得v1->v2的权值
     * @param v1
     * @param v2
```

```
     * @return
     */
    public int getWeight(int v1,int v2){
        return matrix[v1][v2];
    }

    /**
     * 获得边的数量
     * @return
     */
    public int getEdgeNum(){
        return E;
    }

    /**
     * 获得结点数
     * @return
     */
    public int getVertexNum(){
        return V.size();
    }

    //打印邻接矩阵
    public void print() {
        for (int[] edge : matrix) {
            System.out.println(Arrays.toString(edge));
        }
    }

    public static void main(String[] args) {
        //定义图的所有顶点
        String[] vertexs = {"1","2","3","4"};
        //创建图
        GraphM graph = new GraphM(vertexs.length);
        //添加顶点到图中
        for (String vertex : vertexs) {
            graph.insertVertex(vertex);
        }
        //添加边到图中
        graph.insertEdge(0,1,1);
        graph.insertEdge(0,2,1);
        graph.insertEdge(0,3,1);
        graph.insertEdge(2,3,1);
        graph.print();

    }
}
```

```
[0, 1, 1, 1]
[1, 0, 0, 0]
[1, 0, 0, 1]
[1, 0, 1, 0]
```

邻接表

是一个以链表为元素的数组



时间复杂性分析

判断一个边是否存在：O(|V|)

寻找某个顶点所能到达的相邻顶点：O(|V|)

寻找所有的边：O(|E|)

增加或者删除一条边：O(|V|)

适用于稀疏图

代码实现

```
public class GraphList {
    //使用邻接表实现无向图
    private int V;//顶点数
    private int E;//边数
    private ArrayList<Integer> [] adjacencyList;//邻接表

    //对图进行初始化
    public GraphList(int v) {
        this.V = v;
        this.E = 0;
        this.adjacencyList = new ArrayList[v];
        for (int i = 0; i <v ; i++) {
            adjacencyList[i] = new ArrayList<>(v);
        }
    }

    //获取顶点数
    public int getV() {
        return V;
    }

    //获取边数
    public int getE() {
        return E;
    }

    /**
     * 插入边
     * @param v1 起点
     * @param v2 终点
```

```
     */
    public void insertEdge(int v1,int v2){
        adjacencyList[v1].add(v2);
        adjacencyList[v2].add(v1);
        E++;
    }

    public ArrayList<Integer> getAdjacentVertexs(int v){
        return adjacencyList[v];
    }

    public void print(){
        for (int i = 0; i < V; i++) {
            System.out.println("vertex:"+i+" : "+getAdjacentVertexs(i));
        }
    }

    public static void main(String[] args) {
        GraphList graphList = new GraphList(5);
        graphList.insertEdge(0,1);
        graphList.insertEdge(0,2);
        graphList.insertEdge(1,2);
        graphList.insertEdge(3,4);
        graphList.print();
    }
}
```

```
vertex:0 : [1, 2]
vertex:1 : [0, 2]
vertex:2 : [0, 1]
vertex:3 : [4]
vertex:4 : [3]
```

4.2 图的遍历

概念

🏷 定义

> 基于图的拓扑结构，以特定的顺序依次访问图中各顶点，从概念上讲与树的遍历类似

🏷 基本思想

从图中的某个顶点作为出发的起点，然后试探性的访向其余顶点

🏷 可能遇到的问题

> 从起点出发可能到达不了所有其他顶点
>
> > 非连通图的存在
>
> 可能会陷入死循环图中存在回路

🏷 解决方法

为图中的每个顶点增加标志位

通过标志位决定顶点是否被访问过，用来**解决死循环**的问题

通过标志位决定经过一次试探后，还有哪些顶点没有被访问过，用来**解决非连通图**的问题

深度优先搜索

类似于树的先序遍历，是树的先序遍历的推广

DFS是对图的很多问题处理的基础

> 给出指定两个顶点之间的路径
>
> 判断图是否有回路
>
> 判断图是否是连通图，如果不连通，则有几个连通分量

遍历过程

[1] 假设初始状态是图中所有顶点未曾被访问，则深度优先搜索可以从图中某个顶点v出发

[2] 访问这个v顶点，然后依次从v的未被访问的**邻接点出发深度优先遍历图**,直至图中所有和v有路径相连的顶点都被访问到

[3] 如果此时图中尚有顶点未被访问，则另选图中一个未曾被访向的顶点作起始点，重复上述过程,直至图中所有顶点都被访问到为止

举例说明

代码实现

相邻矩阵

```
/**
    * 获得某个结点的第一个邻接点下标
    *
    * @param v 某个顶点下标
    * @return 第一个邻接点下标
    */
   public int getFirst(int v) {
       for (int i = 0; i < v.size(); i++) {
           if (matrix[v][i] > 0) {
               return i;
           }
       }
       return -1;
   }

   /**
    * @param v1 某个顶点坐标
    * @param v2 前一个邻接结点坐标
    * @return 下一个邻接结点坐标
    */
   public int getNext(int v1, int v2) {
       for (int i = v2 + 1; i < v.size(); i++) {
           if (matrix[v1][i] > 0) {
               return i;
           }
       }
```

```
        return -1;
    }

    public boolean isEdge(int v1, int v2) {
        return isLegalIndex(v1) && isLegalIndex(v2) && matrix[v1][v2] > 0;
    }

    public boolean isLegalIndex(int v) {
        return v >= 0 && v < V.size();
    }

    public void DFS() {
        for (int i = 0; i < getVertexNum(); i++) {
            setMark(i, UNVISITED);
        }//初始化所有顶点
        for (int i = 0; i < getVertexNum(); i++) {
            if (getMark(i) == UNVISITED) {
                DFSHelp(i);
            }
        }
    }

    public void DFSHelp(int v) {
        System.out.print(V.get(v) + " ");
        setMark(v, VISITED);
        for (int edge = getFirst(v); isEdge(v, edge); edge = getNext(v, edge)) {
            if (getMark(edge) == UNVISITED) {
                DFSHelp(edge);
            }
        }
    }
    public static void main(String[] args) {
        //定义图的所有顶点
        String[] vertexs = {"A", "B", "C", "D", "E", "F"};
        //创建图
        GraphMatrix graph = new GraphMatrix(vertexs.length);
        //添加顶点到图中
        for (String vertex : vertexs) {
            graph.insertVertex(vertex);
        }
        //添加边到图中
        graph.insertEdge(0, 2, 1);
        graph.insertEdge(0, 4, 1);
        graph.insertEdge(1, 2, 1);
        graph.insertEdge(1, 5, 1);
        graph.insertEdge(2, 3, 1);
        graph.insertEdge(2, 5, 1);
        graph.insertEdge(3, 5, 1);
        graph.insertEdge(4, 5, 1);
        graph.print();
        graph.DFS();
    }
```

```
[0, 0, 1, 0, 1, 0]
[0, 0, 1, 0, 0, 1]
[1, 1, 0, 1, 0, 1]
[0, 0, 1, 0, 0, 1]
[1, 0, 0, 0, 0, 1]
[0, 1, 1, 1, 1, 0]
A C B F D E
```

邻接表

```
    public int getFirst(int v) {
        return adjacencyList[v].size() == 0 ? -1 : adjacencyList[v].get(0);
    }

    public int getNext(int v, int index) {
        return adjacencyList[v].size() <= index ? -1 :
adjacencyList[v].get(index);
    }

    public boolean isEdge(int v1, int v2) {
        if (!isLegalIndex(v1) || !isLegalIndex(v2)) return false;
        for (int i = 0; i < adjacencyList[v1].size(); i++) {
            if (adjacencyList[v1].get(i) == v2) return true;
        }
        return false;
    }

    public boolean isLegalIndex(int v) {
        return v >= 0 && v < V;
    }

    public void DFS() {
        for (int i = 0; i < V; i++) {
            setMark(i, UNVISITED);
        }//初始化所有顶点
        for (int i = 0; i < V; i++) {
            if (getMark(i) == UNVISITED) {
                DFSHelp(i);
            }
        }
    }

    public void DFSHelp(int v) {
        System.out.print(v + " ");
        setMark(v, VISITED);
        int i = 0;//邻接链表下标
        for (int index = getFirst(v); isEdge(v, index); index = getNext(v, ++i))
{
            if (getMark(index) == UNVISITED) {
                DFSHelp(index);
            }
        }
    }
```

```
vertex:0 : [2, 4]
vertex:1 : [2, 5]
vertex:2 : [0, 1, 3, 5]
vertex:3 : [2, 5]
vertex:4 : [0, 5]
vertex:5 : [1, 2, 3, 4]
0 2 1 5 3 4
```

广度优先遍历

遍历过程

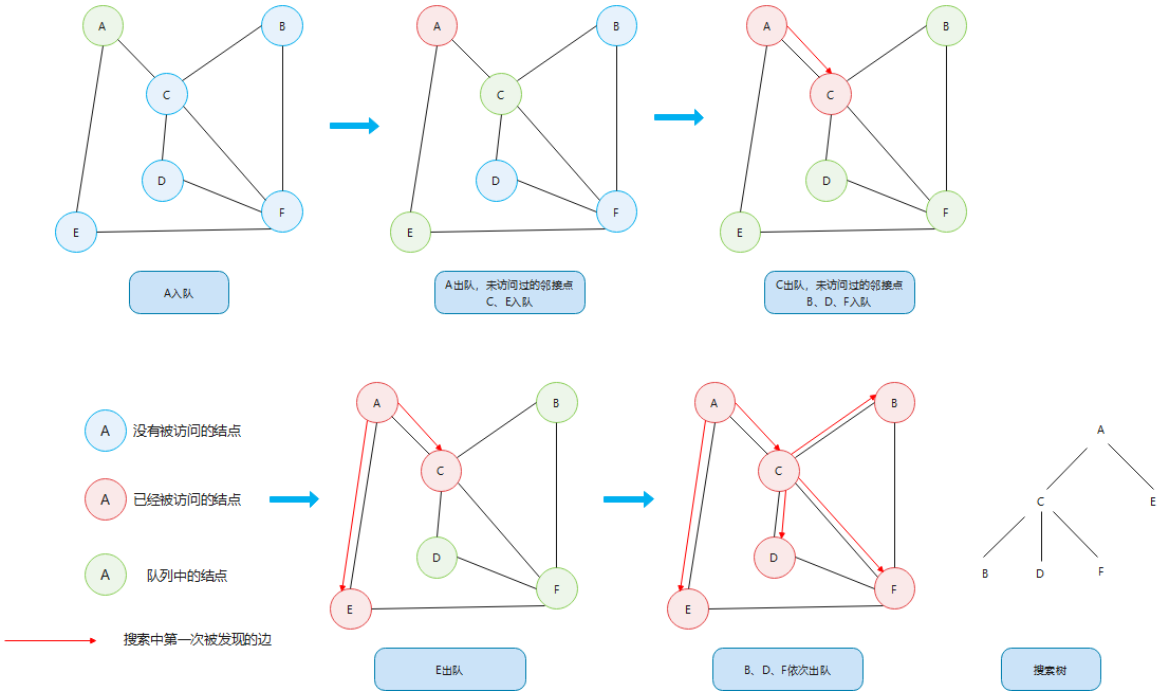
[1] 假设从图中某个顶点v出发，在访问了v之后，依次访问v的**各个未曾访问过的邻接点**，并保证先被访问的顶点的邻接点“要先于”后被访问的顶点的邻接点的访问，直至图中所有已被访问的顶点的邻接点都被访问到

[2] 若此时图中还有未被访问的顶点，则任选其中之一作为起点，重新开始上述过程，直至图中所有顶点都被访问到

访问特征

保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问，也就是先到先被访间，这正好是队列的特点，因此可以使用队列来实现

举例说明



代码实现

```
    public void BFS() {
        for (int i = 0; i < getVertexNum(); i++) {
            setMark(i, UNVISITED);
        }//初始化所有顶点
        for (int i = 0; i < getVertexNum(); i++) {
            if (getMark(i) == UNVISITED) {
                BFSHelp(i);
            }
```

```
        }
    }

    public void BFSHelp(int v){
        LQueue<Integer> queue = new LQueue<>();
        queue.enqueue(v);
        setMark(v,VISITED);
        while (!queue.isEmpty()){
            int temp = queue.dequeue();
            System.out.print(V.get(temp)+" ");
            for (int index = getFirst(temp); isEdge(temp, index); index =
getNext(temp, index)) {
                if (getMark(index)==UNVISITED){
                    queue.enqueue(index);
                    setMark(index,VISITED);
                }
            }
        }
    }
```

```
A C E B D F
```

DFS与BFS比较

在访问结点的时机方面:

> DFS可以在处理某个结点的所有邻接结点之前接受访问，也可以在处理完某个结点的所有邻接结点之后接受访问
>
> BFS则只有在结点入队时(或者出队时）接受访问

4.3 拓扑排序与最短路径问题

拓扑排序

一项工程往往可以分解为一些具有相对独立性的子工程，通常称这些子工程为“活动”

> 子工程的完成意味着整个工程的完成
>
> 子工程之间在进行的时间上有着一定的相互制约关系
>
> > 盖大楼的第一步是打地基，而房屋的内装修必须在房子盖好之后才能开始进行

可用一个**有向图**表示子工程及其相互制约的关系，其中以**顶点表示活动，弧表示活动之间的优先制约关系**，称这种有向图为活动在顶点上的网络，简称活动顶点网络，或AOV (Activity On Vertex)网

> ☆ 要学会将现实问题抽象成用图表示：顶点的物理意义与弧的物理意义

概念

AOV网的定义

> 是一个**有向图**，该图中的**顶点表示活动，图中的弧表示活动之间的优先关系**

前驱(predecessor)、后继(successor)

> 顶点i是顶点j的前驱当且仅当从顶点i有一条有向路径到达顶点j，顶点j也称为顶点i的后继

活动之间的优先关系**满足传递关系**、非自反关系

> 对任意顶点i。j,k,如果i是j的前驱并且j是k的前驱，那么i一定也是k的前驱
>
> 对任意顶点i，i是i的前驱永远为假

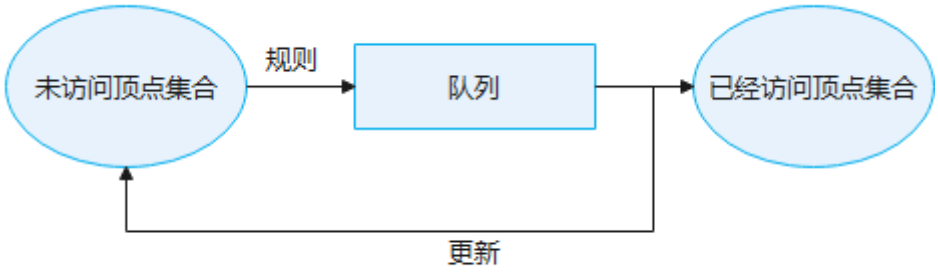
不允许有环出现

> 否则意味着某个活动的开始是以这个活动的结束为先决条件的

拓扑排序的定义

> 一个G中所有顶点的一种线性顺序
>
> 对于G中的任意顶点i和j，如果i是j的前驱，那么在这个线性顺序中i一定在j之前

关键是规则和更新条件



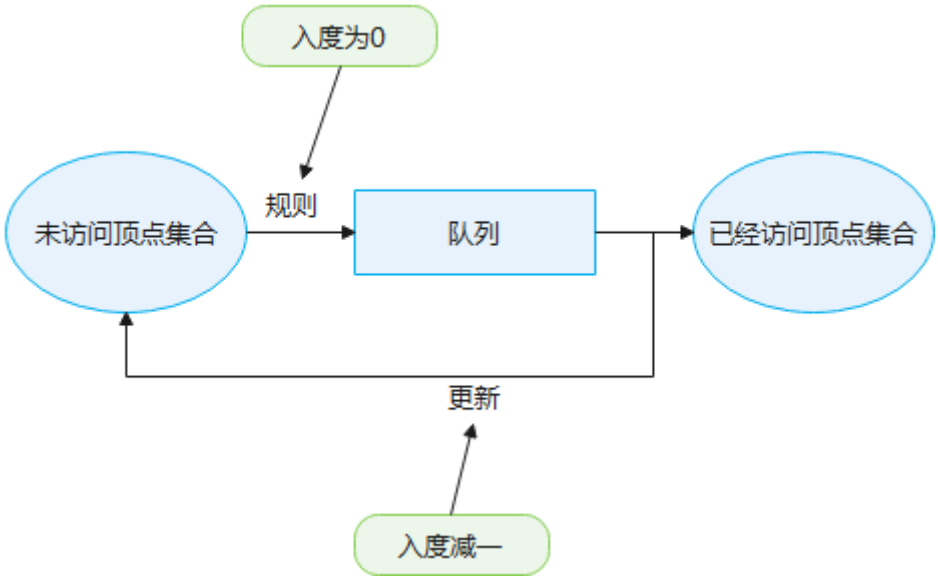
通过BFS获得一个拓扑序列

[1] 扫描整个图，计算每个顶点的入度-让入度为0的顶点进入队列

[2] 如果队列不空，从队列中删除一个顶点并输出，同时将其所有相邻顶点的入度数减一，当某个相邻的顶点的入度数为0时，则将这个顶点插入到队列中

[3] 重复上述步骤直到队列为空

[4] 如果还有顶点没有输出，那么表明这个图**有环,不符合AOV网的定义**




```
public class Digraph {
    //用邻接表实现有向图

    private int E;//边数
    private ArrayList<Integer>[] adjacencyList;//邻接表
    private ArrayList<String> vertexs;//存储顶点
```

```
    private int[] in_degree;//存储每个结点的入度数

    //对图进行初始化
    public Digraph(int v) {
        this.vertexs = new ArrayList<>(v);
        this.E = 0;
        this.adjacencyList = new ArrayList[v];
        this.in_degree = new int[v];
        for (int i = 0; i < v; i++) {
            in_degree[i] = 0;
        }
        for (int i = 0; i < v; i++) {
            adjacencyList[i] = new ArrayList<>(v);
        }
    }

    //获取顶点数
    public int getVertexNum() {
        return vertexs.size();
    }

    //获取边数
    public int getEdgeNum() {
        return E;
    }

    /**
     * 插入边
     *
     * @param v1 起点
     * @param v2 终点
     */
    public void insertEdge(int v1, int v2) {
        adjacencyList[v1].add(v2);
        in_degree[v2]++;//入度加一
        E++;
    }

    /**
     * 添加结点
     *
     * @param vertex 结点值
     */
    public void insertVertex(String vertex) {
        vertexs.add(vertex);
    }


    public int getFirst(int v) {
        return adjacencyList[v].size() == 0 ? -1 : adjacencyList[v].get(0);
    }

    public int getNext(int v, int index) {
        return adjacencyList[v].size() <= index ? -1 :
adjacencyList[v].get(index);
    }

    public boolean isEdge(int v1, int v2) {
```

```
        if (!isLegalIndex(v1) || !isLegalIndex(v2)) return false;//先判断下标是否合
法
        for (int i = 0; i < adjacencyList[v1].size(); i++) {
            if (adjacencyList[v1].get(i) == v2) return true;
        }
        return false;
    }

    public boolean isLegalIndex(int v) {
        return v >= 0 && v < getVertexNum();
    }

    public void topoSort() {
        String[] toposort = topoSortHelp();
        if (toposort == null) {
            System.out.println("存在环! 拓扑排序失败! ");
            return;
        }
        for (String vertex:toposort) {
            System.out.print(vertex + " ");
        }
    }

    public String[] topoSortHelp() {
        String[] toposort = new String[getVertexNum()];//拓扑排序后的结点序列
        LQueue<Integer> queue = new LQueue<>();
        int[] count = new int[getVertexNum()];
        for (int i = 0; i < getVertexNum(); i++) {
            count[i] = in_degree[i];
        }//初始化所有顶点
        int topo_index = 0;
        label:
        for (int i = 0; i < getVertexNum(); i++) {
            if (count[i] == 0) {
                queue.enqueue(i);
            }
            while (!queue.isEmpty()) {
                int temp = queue.dequeue();
                toposort[topo_index++] = vertexs.get(temp);
                if (topo_index==getVertexNum()) break label;//排序结束
                int j = 0;
                for (int index = getFirst(temp); isEdge(temp, index); index =
getNext(temp, ++j)) {
                    count[index]--;//入度减一
                    if (count[index] == 0) {
                        queue.enqueue(index);
                    }
                }
            }
        }
        for (int i = 0; i < getVertexNum(); i++) {
            if (count[i] != 0) {
                return null;
            }
        }//如果还有结点的入度不为0，说明存在环
        return toposort;
    }
```

```
    public static void main(String[] args) {
        String[] test = {"A", "B", "C", "D", "E", "F"};
        Digraph digraph = new Digraph(test.length);
        for (String vertex : test) {
            digraph.insertVertex(vertex);
        }
        digraph.insertEdge(0, 2);
//        digraph.insertEdge(2, 0);
        digraph.insertEdge(0, 4);
        digraph.insertEdge(1, 5);
        digraph.insertEdge(2, 1);
        digraph.insertEdge(2, 3);
        digraph.insertEdge(2, 5);
        digraph.insertEdge(3, 5);
        digraph.insertEdge(4, 5);
        digraph.topoSort();
    }
}
```

```
A C E B D F
```

最短路径问题

概念

路径的代价

> 对于无权图来说，路径的代价就是指路径的长度
>
> 对于有权图来说，路径的代价是指这个路径所经过的所有边上的权重之和

最短路径

> 给定两个顶点A和B，从A到B的一条有向简单路径而且此路径有以下属性:即不存在另外一条这样的路径且有更小的代价

三种类型

[1] 源点-汇点最短路径（Source Sink Shortest Path)

> 从图G= (V，E）中，给定一个起始顶点s和一个结束顶点t，在图中找出从s到t的一条最短路径

[2] 单源最短路径(Single Source Shortest Path)

> 从图G= (v，E）中，找出从某个给定源顶点s∈v到v中的每个顶点的最短路径

[3] 全源最短路径（all-pairs shortest-paths)

> 对于图G=(v，E)，对任意的v，u∈v，都能知道v和u之间的最短路径值

相应的算法

[1] 不带权值的图的最短路径

使用广度优先搜索就可以解决

[2] 带有权值（正值）的图的最短路径

> 单源最短路径(single-source shortest path)

> 使用Dijkstra算法

每对顶点间的最短路径(all-pairs shortest-paths)

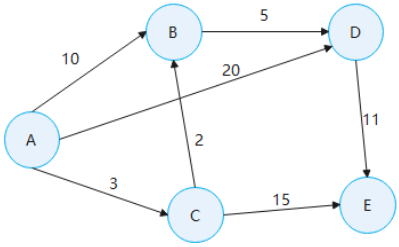
> 使用|V|次Dijkstra算法
>
> 使用FLayd算法

[3] 带有负权值（但不含有负权值环）的图的最短路径

> 单源最短路径：Bellman-ford算法
>
> 全源最短路径：Floyd算法

示例

给定一个带权图，以顶点A为源点，得到从A到图中其他顶点的最短路径



from	to	paths	lengths	shortest path
A	B	(A,B), (A,C,B)	10, 5	(A,C,B)
	C	(A,C)	3	(A,C)
	D	(A,D), (A,B,D),(A,C,B,D)	20,15,10	(A,C,B,D)
	E	(A,C,E), (A,D,E), (A,B,D,E), (A,C,B,D,E)	18,31,26,21	(A,C,E)

Dijkstra算法

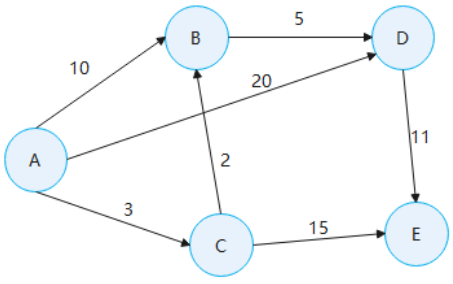
利用BFS搜索思想，只不过将顶点从一个集合拉到另一个集合的规则不同

算法思想

[1] 按路径代价递增的次序产生最短路径

[2] 设集合S存放已经求得的最短路径的终点，从V-S中选择一个顶点t，t是目前所有还没有求得最短路径顶点中与V0之间的距离最短的顶点；将t加入到S中，并且更新V0到V-S中所有能够到达顶点之间的距离:如此反复，直到V-S中没有可以从V0到达的顶点为止

图解



迭代次数	初始化	1	2	3	4	5
活动结点		A	C	B	D	E
A	0	0	0	0	0	0
B	无穷	10	5	5	5	5
C	无穷	3	3	3	3	3
D	无穷	20	20	10	10	10
E	无穷	无穷	18	18	18	18

代码（java）

```
public class Edge {
    private int v1;//起点
    private int v2;//终点
    private int weight;//权重

    public Edge(int v1, int v2) {
        this.v1 = v1;
```

```
        this.v2 = v2;
    }

    public Edge(int v1, int v2, int weight) {
        this.v1 = v1;
        this.v2 = v2;
        this.weight = weight;
    }

    public int getV1() {
        return v1;
    }

    public void setV1(int v1) {
        this.v1 = v1;
    }

    public int getV2() {
        return v2;
    }

    public void setV2(int v2) {
        this.v2 = v2;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }
}
public class WeightedDigraph {
    //带权有向图使用邻接表实现
    private static final int VISITED = 1;//已经访问
    private static final int UNVISITED = 0;//未访问
    private ArrayList<String> vertexs;//存储顶点
    private ArrayList<Edge>[] adjacencyList;//邻接表存边
    private ArrayList<Edge> edges;//存储边
    private int[] Mark;//判定一个结点是否被访问过
    private int currentIndex;//当前邻接表下标

    /**
     * 构造器
     *
     * @param n
     */
    public WeightedDigraph(int n) {
        this.vertexs = new ArrayList<>(n);
        this.adjacencyList = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            adjacencyList[i] = new ArrayList<>(n);
        }
        this.Mark = new int[n];
        this.edges = new ArrayList<>(0);

    }
```

```java
    /**
     * 添加结点
     *
     * @param vertex 结点值
     */
    public void insertVertex(String vertex) {
        vertexs.add(vertex);
    }

    /**
     * 插入边
     *
     * @param v1 起点
     * @param v2 终点
     */
    public void insertEdge(int v1, int v2) {
        adjacencyList[v1].add(new Edge(v1, v2));
    }

    public void insertEdge(int v1, int v2, int weight) {
        adjacencyList[v1].add(new Edge(v1, v2, weight));
    }

    public void insertEdge(Edge edge) {
        adjacencyList[edge.getV1()].add(edge);
    }

    public int getVertexNum() {
        return Mark.length;//点的数量
    }

    public int getEdgeNum() {
        return edges.size();//边的数量
    }


    /**
     * 获得邻接表中的下标为index的边
     *
     * @param v1    顶点下标
     * @param index 第index-1个元素
     * @return
     */
    private Edge getAdjacentElement(int v1, int index) {
        if (index >= adjacencyList[v1].size() || index < 0) return null;
        return adjacencyList[v1].get(index);
    }

    private boolean isInVertexList(int v) {
        return v >= 0 && v < getVertexNum();
    }

    /**
     * 获得从v1出发的第一条边
     *
     * @param v1
     * @return
```

```
     */
    public Edge getFirstEdge(int v1) {
        currentIndex = 0;
        return getAdjacentElement(v1, currentIndex);
    }

    /**
     * 获得从v1出发的下一条边
     *
     * @param v1
     * @return
     */
    public Edge getNextEdge(int v1) {
        return getAdjacentElement(v1, ++currentIndex);
    }

    /**
     * 判断一条边是否在图中
     *
     * @param edge
     * @return
     */
    public boolean isEdge(Edge edge) {
        if (edge == null) return false;
        if (!isInVertexList(edge.getV1()) ||
                !isInVertexList(edge.getV2())) return false;
        int v = edge.getV1();
        for (Edge i = getFirstEdge(v); i.getV2() >= 0; i = getNextEdge(v)) {
            if (i.getV2() == edge.getV2()) return true;
        }
        return false;
    }

    public boolean isEdge(int i, int j) {
        if (!isInVertexList(i) || !isInVertexList(j)) return false;
        for (Edge w = getFirstEdge(i); w.getV2() >= 0; w = getNextEdge(j)) {
            if (j == w.getV2()) return true;
        }
        return false;
    }

    public void printGraph() {
        for (int i = 0; i < getVertexNum(); i++) {
            System.out.print("vertex " + vertexs.get(i) + " : [ ");
            for (int j = 0; j < adjacencyList[i].size(); j++) {
                int index = getAdjacentElement(i, j).getV2();
                System.out.print(vertexs.get(index) + " ");
            }
            System.out.println("]");
        }
    }

    public void BFS() {
        for (int i = 0; i < getVertexNum(); i++) {
            setMark(i, UNVISITED);
        }//初始化所有顶点
        for (int i = 0; i < getVertexNum(); i++) {
            if (getMark(i) == UNVISITED) {
```

```
                BFSHelp(i);
            }
        }
    }

    public void BFSHelp(int v) {
        LQueue<Integer> queue = new LQueue<>();
        queue.enqueue(v);
        setMark(v, VISITED);
        while (!queue.isEmpty()) {
            int temp = queue.dequeue();
            System.out.print(vertexs.get(temp) + " ");
            for (Edge w = getFirstEdge(temp); isEdge(w); w =
getNextEdge(w.getV1())) {
                if (getMark(w.getV2()) == UNVISITED) {
                    queue.enqueue(w.getV2());
                    setMark(w.getV2(), VISITED);
                }
            }
        }
    }

    private int getMinVertex(int[] distance) {
        int i, min = 0, min_index = 0;
        for (i = 0; i < distance.length; i++) {
            if (getMark(i) == UNVISITED) {
                min = distance[i];
                min_index = i;
            }
        }
        for (int j = 0; j < distance.length; j++) {
            if (distance[j] < min && getMark(j) == UNVISITED) {
                min = distance[j];
                min_index = j;
            }
        }
        return min_index;
    }

    public int[] Dijkstra(int start) {
        int[] distance = new int[getVertexNum()];
        for (int i = 0; i < getVertexNum(); i++) {
            distance[i] = Integer.MAX_VALUE;//初始化
            setMark(i, UNVISITED);
        }
        distance[start] = 0;
        for (int i = 0; i < getVertexNum(); i++) {
            int v = getMinVertex(distance);//替换规则
            if (distance[v] == Integer.MAX_VALUE) return null;
            setMark(v, VISITED);
            for (Edge w = getFirstEdge(v); isEdge(w); w =
getNextEdge(w.getV1())) {
                if (distance[w.getV2()] > distance[v] + w.getWeight()) {
                    distance[w.getV2()] = distance[v] + w.getWeight();//更新
                }
            }
        }
        return distance;
```

```
    }

    public void showDijkstra() {
        int[] result = Dijkstra(0);
        System.out.println("  to   A B  C  D E");
        System.out.print("from A ");
        for (int i = 0; i < result.length; i++) {
            if (result[i] == Integer.MAX_VALUE) {
                System.out.print("max");
            } else {
                System.out.print(result[i] + " ");
            }
        }
    }

    public void setMark(int v, int val) {
        Mark[v] = val;
    }

    public int getMark(int v) {
        return Mark[v];
    }

    public static void main(String[] args) {
        //定义图的所有顶点
        String[] vertexs = {"A", "B", "C", "D", "E"};
        WeightedDigraph graph = new WeightedDigraph(5);
        for (String vertex : vertexs) {
            graph.insertVertex(vertex);
        }
        graph.insertEdge(0, 1, 10);
        graph.insertEdge(0, 2, 3);
        graph.insertEdge(0, 3, 20);
        graph.insertEdge(1, 3, 5);
        graph.insertEdge(2, 1, 2);
        graph.insertEdge(2, 4, 15);
        graph.insertEdge(3, 4, 11);
        graph.printGraph();
        System.out.println("showDijkstra");
        graph.showDijkstra();
        System.out.println();
        graph.BFS();
    }
}
```

```
vertex A : [ B C D ]
vertex B : [ D ]
vertex C : [ B E ]
vertex D : [ E ]
vertex E : [ ]
showDijkstra
  to   A B  C  D E
from A 0 5 3 10 18
A B C D E
```

算法分析

需要扫描|V|次

> 每次扫描都需要扫描|V|个顶点以求得最短路径值的顶点

每扫描到一条边就需要更新一次distance值，由于有|E|条边，所以需更新|E|次

总的时间消耗为O(|V|^2+|E|)

算法改进

利用**优先队列**寻找最小值

总的时间消耗为O((|V|+|E|) log|E|)

4.4 最小支撑树(Minimum-cost Spanning Tree)

概念

定义

给定一个**连通无向图G**，且它的每条边均有相应的长度或权值，则MST是一个包括G中的**所有顶点及其边子集的图**，边的子集满足下列条件:

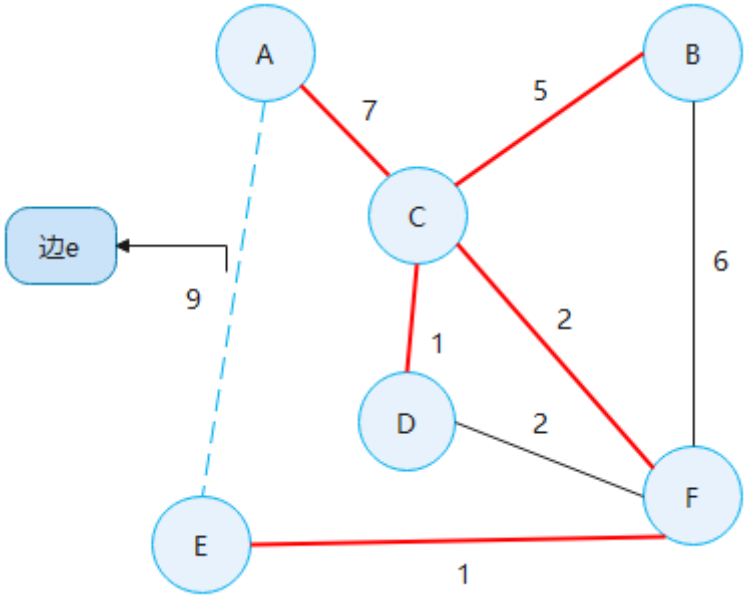
> 这个子集中所有边的权之和为所有子集中最小的
>
> 子集中的边能够保证图是连通的

环性质

环性质

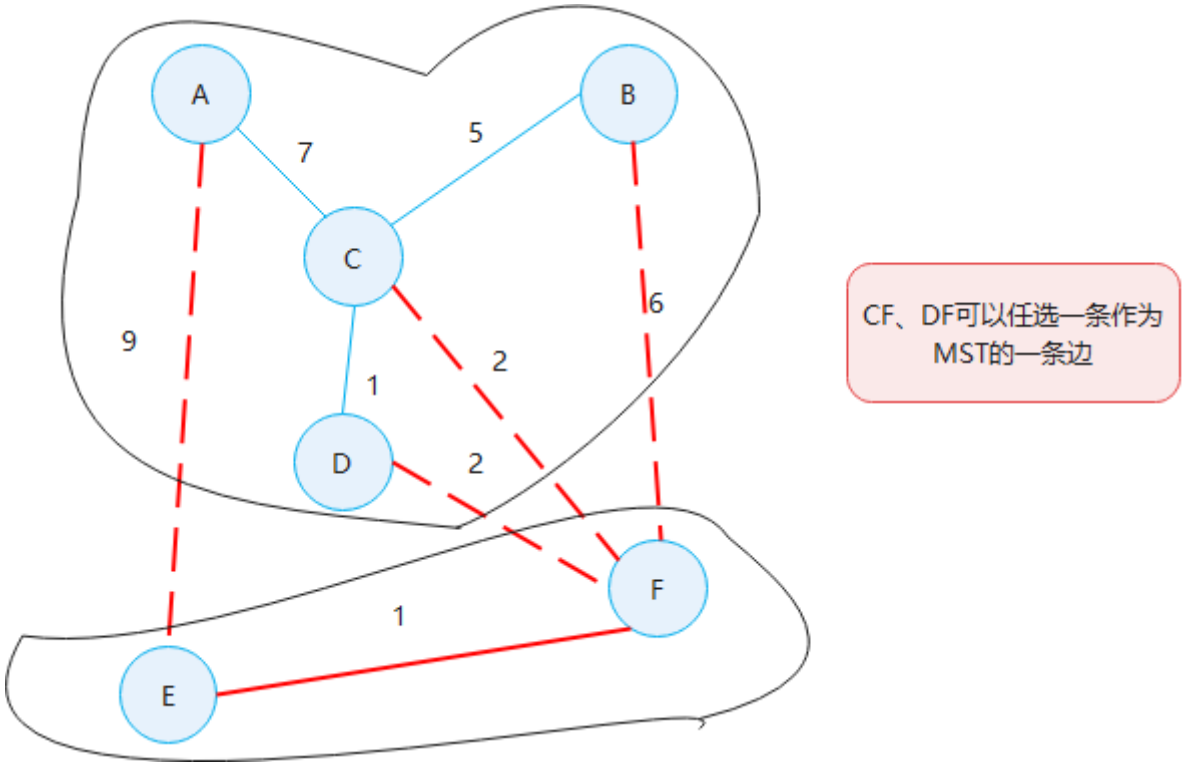
假设T是一个**有权无向图**G=(V,E)的MST，如果选择一条属于E，但不属于T的边e加入到MST,从而使T形成一个环时，那么这个环中的任意一条边f都洞足如下关系

> weight(f) <= weight(e)



分割性质

设集合U和W是图G=(V,E)的顶点集合的两个子集,这两个顶点子集将图分成了两部分，其中e是所有能够连接两个部分中权最小的边，那么e将是MST的一条边



Prim算法——从点出发

算法步骤

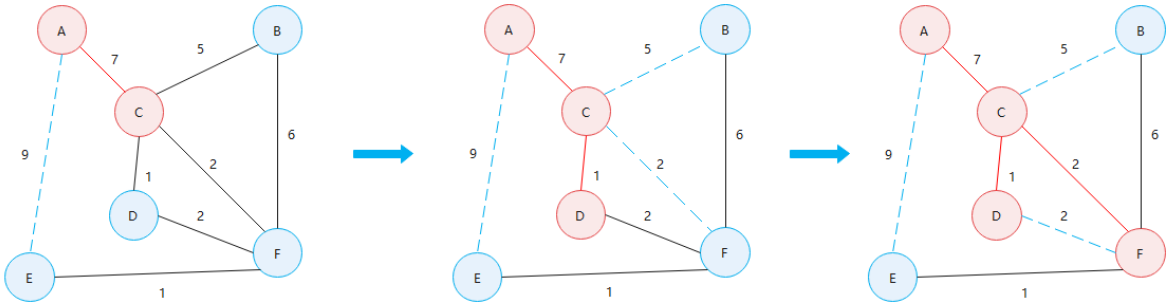
[1] 选择图中的任意一个顶点N开始，初始化MST为N

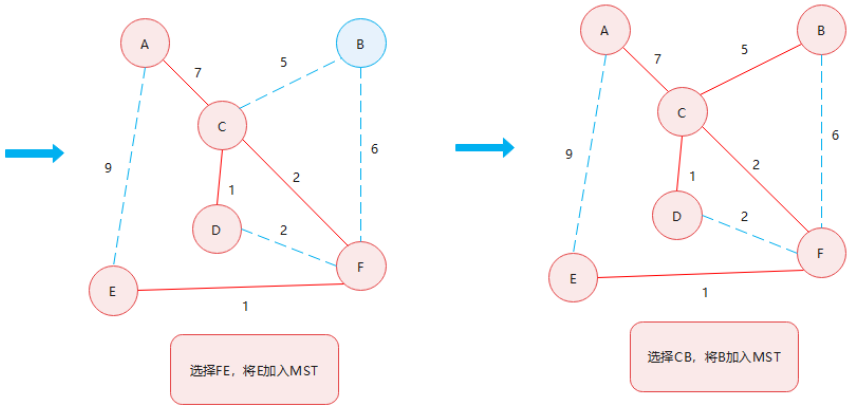
[2] 计算MST中每个顶点到不在MST中的每个顶点之间的距离

[3] 选择这些距离中最小的那条边，并将这条边中的不在MST中的顶点加入到MST中

[4] 重复步骤2和3，直到没有可以加入到MST中的顶点为止

示例





相邻矩阵实现带权无向图

```
public class GraphMatrix {
    //用相邻矩阵实现无向图
    private ArrayList<String> V;//顶点
    private int E;//边数
    private int[][] matrix;//相邻矩阵
    private int[] mark;//判定一个结点是否被访问过
    public static final int VISITED = -2;//已经访问
    public static final int UNVISITED = -3;//未访问
    public static final int UNCONNECTED = 1000;//未访问

    /**
     * 初始化无向图
     *
     * @param n 结点数
     */
    public GraphMatrix(int n) {
        matrix = new int[n][n];
        mark = new int[n];
        for (int i = 0; i < n; i++) {
            mark[i] = UNVISITED;
            for (int j = 0; j < n; j++) {
                matrix[i][j] = UNCONNECTED;
            }
        }
        V = new ArrayList<String>(n);

    }

    /**
     * 添加结点
```

```
     *
     * @param vertex 结点值
     */
    public void insertVertex(String vertex) {
        V.add(vertex);
    }

    /**
     * 插入边
     *
     * @param v1     起点下标
     * @param v2     终点下标
     * @param weight 权值，这里0表示不相邻，1表示相邻
     */
    public void insertEdge(int v1, int v2, int weight) {
        matrix[v1][v2] = weight;
        matrix[v2][v1] = weight;
        E++;//边数+1
    }

    /**
     * 获得v1->v2的权值
     *
     * @param v1
     * @param v2
     * @return
     */
    public int getWeight(int v1, int v2) {
        return matrix[v1][v2];
    }

    /**
     * 获得边的数量
     *
     * @return
     */
    public int getEdgeNum() {
        return E;
    }

    /**
     * 获得结点数
     *
     * @return
     */
    public int getVertexNum() {
        return V.size();
    }

    public String getVertexValue(int index) {
        return V.get(index);
    }

    /**
     * 获得某个结点的第一个邻接点下标
     *
     * @param v 某个顶点下标
     * @return 第一个邻接点下标
```

```
     */
    public int getFirst(int v) {
        for (int i = 0; i < V.size(); i++) {
            if (matrix[v][i] > 0) {
                return i;
            }
        }
        return -1;
    }

    /**
     * @param v1 某个顶点坐标
     * @param v2 前一个邻接结点坐标
     * @return 下一个邻接结点坐标
     */
    public int getNext(int v1, int v2) {
        for (int i = v2 + 1; i < V.size(); i++) {
            if (matrix[v1][i] > 0) {
                return i;
            }
        }
        return -1;
    }

    public boolean isEdge(int v1, int v2) {
        return isLegalIndex(v1) && isLegalIndex(v2) && matrix[v1][v2] > 0;
    }

    public boolean isLegalIndex(int v) {
        return v >= 0 && v < V.size();
    }

    public void DFS() {
        for (int i = 0; i < getVertexNum(); i++) {
            setMark(i, UNVISITED);
        }//初始化所有顶点
        for (int i = 0; i < getVertexNum(); i++) {
            if (getMark(i) == UNVISITED) {
                DFSHelp(i);
            }
        }
    }

    public void DFSHelp(int v) {
        System.out.print(V.get(v) + " ");
        setMark(v, VISITED);
        for (int edge = getFirst(v); isEdge(v, edge); edge = getNext(v, edge)) {
            if (getMark(edge) == UNVISITED) {
                DFSHelp(edge);
            }
        }
    }

    public void BFS() {
        for (int i = 0; i < getVertexNum(); i++) {
            setMark(i, UNVISITED);
        }//初始化所有顶点
        for (int i = 0; i < getVertexNum(); i++) {
```

```
            if (getMark(i) == UNVISITED) {
                BFSHelp(i);
            }
        }
    }

    public void BFSHelp(int v) {
        LQueue<Integer> queue = new LQueue<>();
        queue.enqueue(v);
        setMark(v, VISITED);
        while (!queue.isEmpty()) {
            int temp = queue.dequeue();
            System.out.print(V.get(temp) + " ");
            for (int index = getFirst(temp); isEdge(temp, index); index =
getNext(temp, index)) {
                if (getMark(index) == UNVISITED) {
                    queue.enqueue(index);
                    setMark(index, VISITED);
                }
            }
        }
    }

    public void setMark(int v, int val) {
        mark[v] = val;
    }

    public int getMark(int v) {
        return mark[v];
    }

    //打印邻接矩阵
    public void print() {
        for (int[] edge : matrix) {
            System.out.println(Arrays.toString(edge));
        }
    }

    public static void main(String[] args) {
        //定义图的所有顶点
        String[] vertexs = {"A", "B", "C", "D", "E", "F"};
        //创建图
        GraphMatrix graph = new GraphMatrix(vertexs.length);
        //添加顶点到图中
        for (String vertex : vertexs) {
            graph.insertVertex(vertex);
        }
        //添加边到图中
        graph.insertEdge(0, 2, 1);
        graph.insertEdge(0, 4, 1);
        graph.insertEdge(1, 2, 1);
        graph.insertEdge(1, 5, 1);
        graph.insertEdge(2, 3, 1);
        graph.insertEdge(2, 5, 1);
        graph.insertEdge(3, 5, 1);
        graph.insertEdge(4, 5, 1);
        graph.print();
        graph.DFS();
```

```
        System.out.println();
        graph.BFS();
    }
}
```

```
public class Prim {
    private GraphMatrix originalGraph;//用相邻矩阵实现的原图
    private ArrayList<Integer> N;//不在MST集合中的元素下标
    private ArrayList<Integer> S;//在MST集合中的元素下标
    private int vertexNum;//顶点数


    /**
     * 初始化
     * @param originalGraph
     */
    public void createOriginalGraph(GraphMatrix originalGraph) {
        this.originalGraph = originalGraph;
        this.vertexNum = originalGraph.getVertexNum();
        this.N = new ArrayList<>(vertexNum);
        for (int i = 0; i < vertexNum; i++) {
            N.add(i);
        }
        this.S = new ArrayList<>(0);
    }

    public void showOriginalGraph() {
        originalGraph.print();
    }

    /**
     * 从start开始使用prim算法生成MST
     * @param start 起点的下标
     */
    public void prim(int start) {
        int v1 = start;
        int v2 = originalGraph.getFirst(v1);
        originalGraph.setMark(v1, GraphMatrix.VISITED);
        S.add(v1);//加入MST
        N.remove(v1);//同时从N中移除
        for (int i = 0; i < vertexNum - 1; i++) {
            int miniWeight = GraphMatrix.UNCONNECTED;
            for (int j = 0; j < S.size(); j++) {//从MST中的顶点
                for (int k = 0; k < N.size(); k++) {//到非MST中的顶点
                    if (originalGraph.getMark(S.get(j)) == GraphMatrix.VISITED
&&
                            originalGraph.getMark(N.get(k)) ==
GraphMatrix.UNVISITED
                            && originalGraph.getWeight(S.get(j), N.get(k)) <
miniWeight) {
                        //只需要看从已访问结点到未访问结点的边
                        miniWeight = originalGraph.getWeight(S.get(j),
N.get(k));
                        v1 = S.get(j);
                        v2 = N.get(k);
                    }
                }
```

```
            }
            System.out.println(originalGraph.getVertexValue(v1) + "-" +
miniWeight + "->"
                    + originalGraph.getVertexValue(v2));
            //更新
            S.add(v2);//加入MST
            N.remove((Object) v2);//同时从N中移除
            originalGraph.setMark(v2, GraphMatrix.VISITED);
        }
    }

    public static void main(String[] args) {
        //定义图的所有顶点
        String[] vertexs = {"A", "B", "C", "D", "E", "F"};
        //创建图
        GraphMatrix graph = new GraphMatrix(vertexs.length);
        //添加顶点到图中
        for (String vertex : vertexs) {
            graph.insertVertex(vertex);
        }
        //添加边到图中
        graph.insertEdge(0, 2, 7);
        graph.insertEdge(0, 4, 9);
        graph.insertEdge(1, 2, 5);
        graph.insertEdge(1, 5, 6);
        graph.insertEdge(2, 3, 1);
        graph.insertEdge(2, 5, 2);
        graph.insertEdge(3, 5, 2);
        graph.insertEdge(4, 5, 1);
        Prim test = new Prim();
        test.createOriginalGraph(graph);
        test.showOriginalGraph();
        test.prim(0);
    }
}
```

```
[1000, 1000, 7, 1000, 9, 1000]
[1000, 1000, 5, 1000, 1000, 6]
[7, 5, 1000, 1, 1000, 2]
[1000, 1000, 1, 1000, 1000, 2]
[9, 1000, 1000, 1000, 1000, 1]
[1000, 6, 2, 2, 1, 1000]
A-7->C
C-1->D
C-2->F
F-1->E
C-5->B
```

Kruskal算法——从边出发

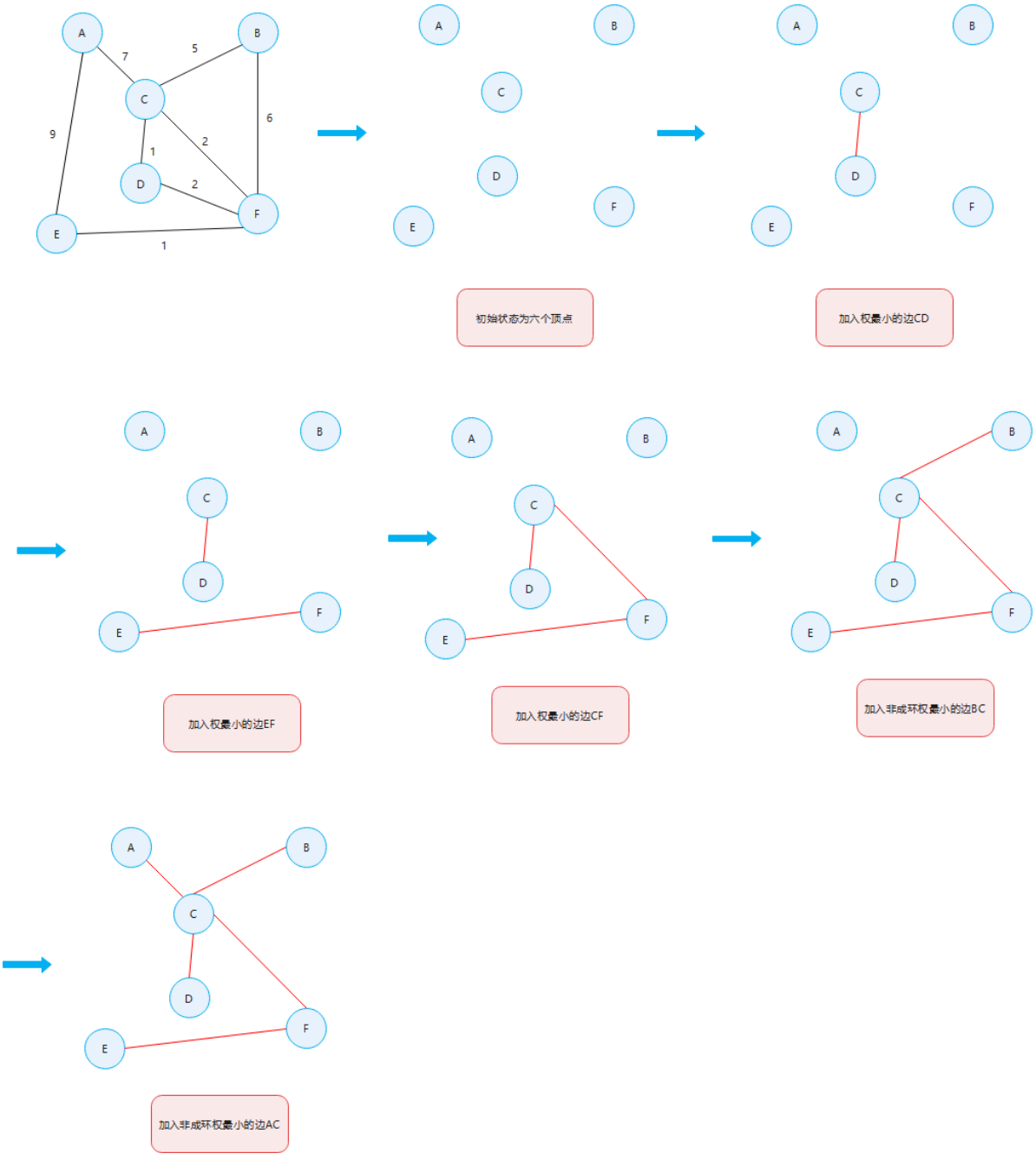
算法步骤

[1]设连通网 N = (V, E)，令最小生成树初始状态为只有 n 个顶点而无边的非连通图 T=(V, { })，每个顶点自成一个连通分量。

[2]在 E 中选取代价最小的边，若该边依附 的顶点落在 T 中不同的连通分量上（即： 不能形成环），则将此边加入到 T 中；否 则，舍去此边，选取下一条代价最小的边。

[3] 依此类推，直至 T 中所有顶点都在同一 连通分量上为止。

示例



代码实现

[1] 利用并查集树判断结点是否连通，以及连通两个非连通分量

```
public class GTNodeA {
    private int parent;//父结点下标
    private Object element;//存储元素内容
```

```
    public GTNodeA() {
        this(null, -1);
    }

    public GTNodeA(Object element) {
        this(element, -1);
    }

    public GTNodeA(Object element, int parent) {
        this.element = element;
        this.parent = parent;
    }

    public int getParent() {
        return parent;
    }

    public int setParent(int parent) {
       return this.parent = parent;
    }

    public Object getElement() {
        return element;
    }

    public void setElement(Object element) {
        this.element = element;
    }
}

public class UnionFindTree {
    //用树实现并查集
    public GTNodeA[] set;

    public UnionFindTree(GTNodeA[] set) {
        this.set = set;
    }

    /**
     * 返回包含给定元素的集合名字
     *
     * @param i 元素下标
     * @return 以根结点下标作为集合名字
     */
    public int find(int i) {
        GTNodeA current = set[i];
        if (current.getParent()<0) return i;
        return current.setParent(find(current.getParent()));
    }//使用路劲压缩法：在查找某个元素是否属于某个集合时，将该结点到根结点路径上
    // 所有结点的父指针全部改为指向根结点，这种方式可以产生极浅的树

    /**
     * 生成一个新的集合，该集合是i所属的集合set1和j所属的集合set2的并集
     *
     * @param i
     * @param j
     */
    public void union(int i, int j) {
```

```
        int root1 = find(i);
        int num1 = set[root1].getParent();
        int root2 = find(j);
        int num2 = set[root2].getParent();
        if (num1 <= num2) {
            set[root1].setParent(num1+num2);
            set[root2].setParent(root1);
            //将其中结点数少的一棵树的根结点的父结点设置为
            //结点数多的一棵树的根结点
        } else {
            set[root2].setParent(num1+num2);
            set[root1].setParent(root2);
            //将其中结点数少的一棵树的根结点的父结点设置为
            //结点数多的一棵树的根结点
        }
    }//使用了重量平衡原则


    /**
     * 判断元素i和元素j是否在同一个分组中
     * @param i
     * @param j
     * @return
     */
    public boolean isConnected(int i,int j){
        return find(i)==find(j);
    }
    public void print() {
        for (int i = 0; i < set.length; i++) {
            System.out.print(set[i].getElement() + " ");
        }
    }

    public static void main(String[] args) {
        GTNodeA node_A = new GTNodeA("A");
        GTNodeA node_C = new GTNodeA("C");
        GTNodeA node_H = new GTNodeA("H");
        GTNodeA node_K = new GTNodeA("K");
        GTNodeA node_E = new GTNodeA("E");
        GTNodeA node_B = new GTNodeA("B");
        GTNodeA node_D = new GTNodeA("D");
        GTNodeA node_F = new GTNodeA("F");
        GTNodeA node_J = new GTNodeA("J");
        GTNodeA node_L = new GTNodeA("L");
        GTNodeA node_N = new GTNodeA("N");
        GTNodeA node_M = new GTNodeA("M");
        GTNodeA node_I = new GTNodeA("I");
        GTNodeA node_G = new GTNodeA("G");
        GTNodeA[] test = {node_A, node_B, node_C, node_D,
                node_E, node_F, node_G, node_H,
                node_I, node_J, node_K, node_L,
                node_M, node_N};
        UnionFindTree testTree = new UnionFindTree(test);
        System.out.print("initialized forest: ");
        testTree.print();
        System.out.println();
        System.out.println("find(1): " + testTree.find(1));
        System.out.println("find(3): " + testTree.find(3));
```

```
        System.out.println("find(4): " + testTree.find(4));
        testTree.union(0, 4);
        testTree.union(0, 1);
        testTree.union(0, 3);
        System.out.println("union(0,4), union(0,1), union(0,3)");
        System.out.println("find(1): " + testTree.find(1));
        System.out.println("find(3): " + testTree.find(3));
        System.out.println("find(4): " + testTree.find(4));
    }
}
```

2 利用最小优先队列优化边的查找与删除

```
public class Edge implements Comparable<Edge> {
    private int v1;//起点
    private int v2;//终点
    private int weight;//权重

    public Edge(int v1, int v2) {
        this.v1 = v1;
        this.v2 = v2;
    }

    public Edge(int v1, int v2, int weight) {
        this.v1 = v1;
        this.v2 = v2;
        this.weight = weight;
    }

    public int getV1() {
        return v1;
    }

    public void setV1(int v1) {
        this.v1 = v1;
    }

    public int getV2() {
        return v2;
    }

    public void setV2(int v2) {
        this.v2 = v2;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }


    @Override
    public int compareTo(@NotNull Edge edge) {
```

```
        if (this.weight < edge.weight) {
            return -1;
        } else if (this.weight == edge.weight) {
            return 0;
        } else {
            return 1;
        }
    }

    @Override
    public String toString() {
        return "Edge{" +
                 v1 +
                "->" + v2 +
                " weight=" + weight +
                '}';
    }
}

public class MinPriorityQueue<T extends Comparable<T>> {
    //最小优先队列
    private static final int DEFAULT_CAPACITY = 10;//默认大小
    private int currentSize;//当前堆的大小
    private T[] array;//堆数组

    public MinPriorityQueue() {
        this.array = (T[]) new Comparable[DEFAULT_CAPACITY + 1];
        currentSize = 0;
    }

    public MinPriorityQueue(int size) {
        this.array = (T[]) new Comparable[size + 1];
        currentSize = 0;
    }

    public MinPriorityQueue(T[] array) {

        this.array = (T[]) new Comparable[array.length + 1];
        for (int i = 0; i < array.length; i++) {
            this.array[i + 1] = array[i];
        }//从1开始算
        currentSize = array.length;
        buildHeap();
    }

    /**
     * 往堆中插入元素
     *
     * @param element 元素值
     */
    public void insert(T element) {
        try {
            if (isFull()) {
                throw new Exception("Array is full!");
            }
            int temp = ++currentSize;
            array[temp] = element;//将element放在数组最后
```

```
            while ((temp != 1) && (array[temp].compareTo(array[getParent(temp)])
< 0)) {
                swap(array, temp, getParent(temp));
                temp = getParent(temp);//如果比父结点的值小则于其交换
            }//注意根结点的下标为1，不是0
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public T findMin() {
        return array[1];
    }

    /**
     * 删除堆顶元素
     */
    public void deleteMin() {
        try {
            if (isEmpty()) {
                throw new Exception("Array is empty!");
            } else {
                swap(array, 1, currentSize--);//将堆顶元素放到最后,同时删除
                if (currentSize != 0) siftDown(1);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 大的值下沉
     *
     * @param pos 当前位置
     */
    private void siftDown(int pos) {
        try {
            if (pos < 0 || pos > currentSize) {
                throw new Exception("Illegal position!");
            }
            while (!isLeaf(pos)) {
                int j = getLeft(pos);
                if ((j < currentSize) && (array[j].compareTo(array[j + 1])) > 0)
j++;
                //跟子树中最小的值交换
                if (array[pos].compareTo(array[j]) < 0) return;
                //当前值已经比子树中的值都小，则返回
                swap(array, pos, j);//交换
                pos = j;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void print() {
        for (int i = 1; i <= currentSize; i++) {
            System.out.print(array[i] + " ");
```

```
        }
    }

    public void heapSort() {
        while (currentSize != 0) {
            System.out.print(findMin() + " ");
            deleteMin();
        }
    }

    public boolean isEmpty() {
        return currentSize == 0;
    }

    public boolean isFull() {
        return currentSize == array.length - 1;
    }


    private boolean isLeaf(int i) {
        return i > currentSize / 2;
    }

    private void buildHeap() {
        for (int i = currentSize / 2; i > 0; i--) {
            siftDown(i);//对每个非叶子结点进行下沉操作
            //从右到左，从下到上
        }
    }

    private int getLeft(int i) {
        return 2 * i;
    }

    private int getRight(int i) {
        return 2 * i + 1;
    }

    private int getParent(int i) {
        return i / 2;
    }

    private void swap(T[] array, int x, int y) {
        T temp = array[y];
        array[y] = array[x];
        array[x] = temp;
        return;
    }

    public static void main(String[] args) throws Exception {
        Edge edge1 = new Edge(1, 2, 1);
        Edge edge2 = new Edge(1, 2, 2);
        Edge edge3 = new Edge(1, 2, 6);
        Edge edge4 = new Edge(1, 2, 4);
        Edge edge5 = new Edge(1, 2, 5);
        Edge edge6 = new Edge(1, 2, 7);
        Edge edge7 = new Edge(1, 2, 3);
//        Integer[] elements = {1, 2, 6, 4, 5, 7, 3};
```

```
//        MinPriorityQueue<Integer> test = new MinPriorityQueue<Integer>
(elements);
        Edge[] edges = {edge1, edge2, edge3, edge4, edge5, edge6, edge7};

        MinPriorityQueue<Edge> test = new MinPriorityQueue<Edge>(edges);

        test.print();
        System.out.println();
        test.heapSort();
        System.out.println();
        test.print();
    }
}
```

3 算法具体实现

```
public class Kruskal {
    private ArrayList<Edge> MST;//最小生成树中的所有边
    private UnionFindTree ufTree;//并查集树，放索引
    private MinPriorityQueue<Edge> allEdges;//图中所有的边
    private GraphMatrix originalGraph;//用相邻矩阵实现的原图
    private int vertexNum;//结点数

    public Kruskal(GraphMatrix originalGraph) {
        this.originalGraph = originalGraph;
        this.vertexNum = originalGraph.getVertexNum();
    }

    public void generateMST() {
        this.MST = new ArrayList<Edge>();//初始化MST
        GTNodeA[] nodes = new GTNodeA[vertexNum];
        for (int i = 0; i < vertexNum; i++) {
            nodes[i] = new GTNodeA(i);
        }
        this.ufTree = new UnionFindTree(nodes);//初始化并查集
        this.allEdges = new MinPriorityQueue<>(originalGraph.getEdgeNum());//初始
化优先队列
        for (int i = 0; i < vertexNum; i++) {
            for (int j = i + 1; j < vertexNum; j++) {
                if (originalGraph.getWeight(i, j) != GraphMatrix.UNCONNECTED) {
                    allEdges.insert(new Edge(i, j, originalGraph.getWeight(i,
j)));
                }
            }
        }//将所有的边加入优先队列中
        while (!allEdges.isEmpty() && MST.size() < vertexNum - 1) {
            Edge e = allEdges.findMin();
            allEdges.deleteMin();
            int v1 = e.getV1();
            int v2 = e.getV2();
            //判断v1和v2是否已经连通
            if (ufTree.isConnected(v1, v2)) continue;
            ufTree.union(v1, v2);//不连通则连接
            MST.add(e);//将边并入MST
        }
    }
```

```
    public ArrayList<Edge> getMST() {
        return MST;
    }

    /**
     * 打印MST的所有边
     */
    public void printMST() {
        for (Edge edge : MST) {
            System.out.println(originalGraph.getVertexValue(edge.getV1()) + "-"
                    + originalGraph.getWeight(edge.getV1(), edge.getV2())
                    + "->" + originalGraph.getVertexValue(edge.getV2()));
        }
    }

    public static void main(String[] args) {
        String[] vertexs = {"A", "B", "C", "D", "E", "F"};
        //创建图
        GraphMatrix graph = new GraphMatrix(vertexs.length);
        //添加顶点到图中
        for (String vertex : vertexs) {
            graph.insertVertex(vertex);
        }
        //添加边到图中
        graph.insertEdge(0, 2, 7);
        graph.insertEdge(0, 4, 9);
        graph.insertEdge(1, 2, 5);
        graph.insertEdge(1, 5, 6);
        graph.insertEdge(2, 3, 1);
        graph.insertEdge(2, 5, 2);
        graph.insertEdge(3, 5, 2);
        graph.insertEdge(4, 5, 1);
        Kruskal test = new Kruskal(graph);
        test.generateMST();
        test.printMST();
    }
}
```

```
C-1->D
E-1->F
C-2->F
B-5->C
A-7->C
```

散列（哈希）

Hashing(散列)

是对直接寻址法的一种改进

这是一个技术，通过它将直接寻址中的关键字通过一个“黑盒子”，把其值压缩到另外一个范围中的某个地址值

平均情形下获得**常数级别的查找数据和储存数据的时间**

Hash Function(哈希函数)

接收待查找的关键字

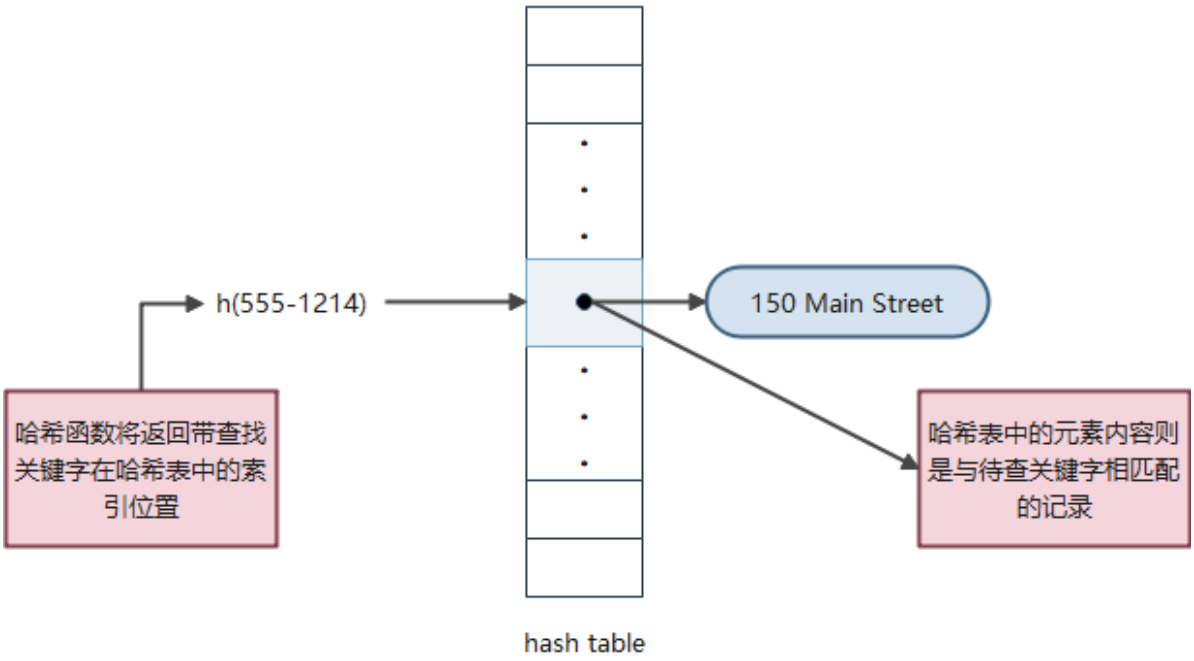
返回一个数组中的索引

> 这个索引就是储存关键字所对应纪录数组中的储存位置，称这个数组为**Hash Table**，称HashFunction返回的索引值为**Hash Index**
>
> 🏷 **哈希表是一个数组，哈希函数返回的是哈希表的索引值**

完美哈希函数

> 该函数能够将查找的关键字映射到互不相同的哈希表上的哈希索引忠
>
> 但是基本不存在此类函数

示例



在找关键字为555-1214这个关键字的位置的时候，没有用到比较，不用从哈希表中一个一个去比，通过将关键字**用哈希函数运算**出一个值，这个值就是关键字为555-1214在哈希表中的索引位置

映射

如何通过哈希函数将待查关键字和哈希表的索引号进行映射?

> 可以通过取模运算

如果关键字不是整数类型，我们可以将不是整数类型的关键字通过各种方式转换成整数类型，转换后的结果称之为**hash code**

数字分析法

假设关键字集合中的每个关键字都是由s位数字组成(k1,k2,..., kn)，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址

```
H(key)=(last 7 digits of key) % n
```

> 比如:出生年月日
>
> 仅限于
>
> > 能预先估计出全体关键字的每一位上各种数字出现的频度
> >
> > 关键字中的某几部分总是相等的

平方取中法

若关键字的每一位都有某些数字重复出现频度很高的现象，则先求关键字的平方值，以通过"平方"扩大差别，同时平方值的中间几位受到整个关键字中各位的影响

```
H(key) = (middle N digits of key^2) % n
```

哈希函数构建

1 使哈希函数尽可能的完美

2 采用何种构造哈希函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，总的原则是**使产生冲突的可能性降到尽可能地小**

3 要计算容易和速度快

4 确定性:对于同一个关键码，不管什么时候计算出来的hash index都应该是确定的

5 散列函数的定义域必须包括需要存储的全部关键码，如果散列表允许有m个地址时，其值域必须在0到m-1之间

Hash冲突

哈希冲突：不同的关键字通过哈希函数运算后得到哈希索引一致，就会引起哈希冲突

解决办法如下

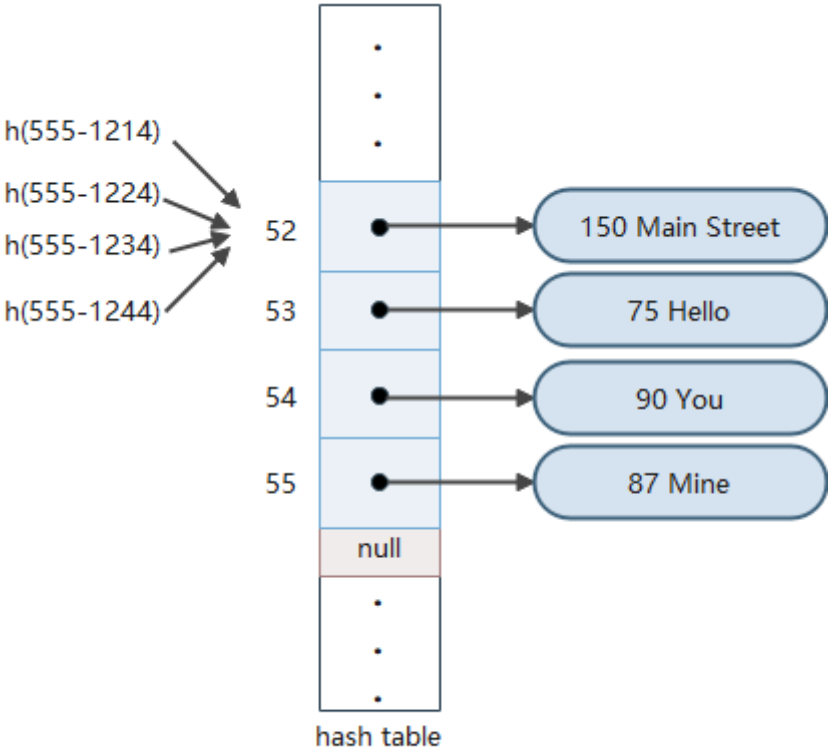
开地址法（open addressing)

在哈希表中重新找一个位置

线性探查（linear probing)

如果在哈希表中的第k个位置发生了冲突，那么就依次试探其后继位置k+1，k+2...

> 会产生基本聚集



DEMO

假设给出一组表项，它们的关键码为Burke,Eat, Bird， Blue,Art, Alert, Height, Eight.采用的散列函数是:**取其第一个字母在字母表中的位置**，散列表的长度为26，试采用线性探查法处理冲突，给出散列表的结果

哈希索引	空表	插入Burke	插入Eat	插入Bird	插入Blue	插入Art	插入Alert	插入Heigh	插入Eight
0						Art	Art	Art	Art
1		Burke	Burke	Burke	Burke	Burke	Burke	Burke	Burke
2				Bird	Bird	Bird	Bird	Bird	Bird
3			发生冲突 索引后移		Blue	Blue	Blue	Blue	Blue
4			Eat	Eat	Eat	Eat	Eat	Eat	Eat
5							Art	Art	Art
6									Eight
7								Heigh	Heigh
8									
...									
25									

性能分析

[1] 搜索成功的平均搜索长度ASLsucc

> 指的是找到表中已有表项的平均探查次数
>
> 如上图，其$ASL_{succ}=\frac{1}{8}\sum_{i=1}^{8}=\frac{1}{8}(1+1+2+3+1+6+3+1)=\frac{9}{4}$
>
> 分母是已有关键字个数

[2] 搜索不成功的平均搜索长度ASLunsucc

> 指在表中搜索不到待查表项，但找到插入位置的平均探查次数,它是表中**所有可能散列到的位置上要插入新元素时为找到空桶**的探查次数的平均直
>
> 到空位置才能明确
>
> 如上图，其$ASL_{unsucc} = \frac{9+8+7+6+5+4+3+2+18}{26} = \frac{31}{13}$
>
> 🏷 还要算剩下的18个位置的
>
> 🏷 分母是散列表长度

BST的有序性比散列表强

需要注意的问题

1 不能真正删除表中已有表项，删除表项会影响其他表项的搜索

> 🏷 若把关键码为Bird的表项真正删除，把它所在位置的info域置为Empty，以后在搜索关键码为Blue 和Eat的表项时就查不下去，会错误地判断表中没有关键码为Blue 和Eat的表项

2 若想删除一个表项,只能给它做一个**删除标记deleted**进行逻辑删除,不能把它真正删去

> 逻辑删除的副作用是:在执行多次副除后,表面上看起来散列表很满,实际上有许多位置没有利用

3 线性探查方法容易产生“堆积”，不同探查序列的关键码占据可用的空桶,为寻找某一关键码要经历不同的探查序列,导致搜索时间增加

> 在某一个范围内的数据聚集。

平方探查（quadratic probing）

如果在哈希表的第k个位置发生了冲突，那么就依次试探其后继位置k+1^2、k+2^2、k+3^2...

如果在哈希表的第k个位置发生了冲突，那么就依次试探其后继位置k+1^2、k-1^2、k+2^2、k-2^2、k+3^2...

	k	k+1			k+4				

相对于线性探测法，会产生二次聚集

> 散列表的长度尽可能定义为素数

注意

> 删除元素的问题如同线性探查一样，不能真删除元素，而应该打标记
>
> 如果有两个关键字值有相同的基位置，那么它们就会有同样的探查序列，这个问题称为二次聚集

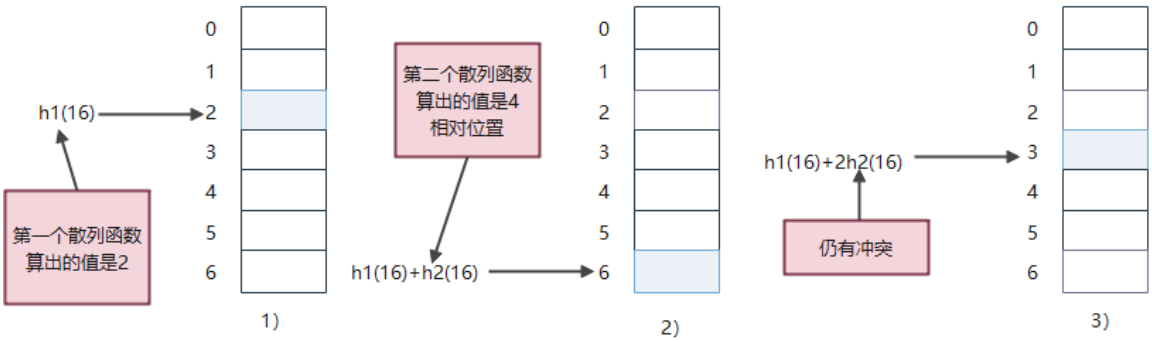
双散列探查（double hashing probing）

当通过第一个哈希函数得到的哈希索引发生冲突之后，获得的下一个哈希索引应该是**第一个哈希索引加上通过第二个哈希函数求得的哈希索引之和**

对第二个哈希函数的要求

> **有别于第一个哈希函数**
>
> 所求的值也要依赖于关键字
>
> 不能返回0值

🏷 避免了基本聚集和二次聚集



这种方法会使关键字的探查序列不同，从而避免了聚集

开散列法（open hashing）

Java API中的HashMap用到的就是开散列法。每一个桶是用红黑树实现的

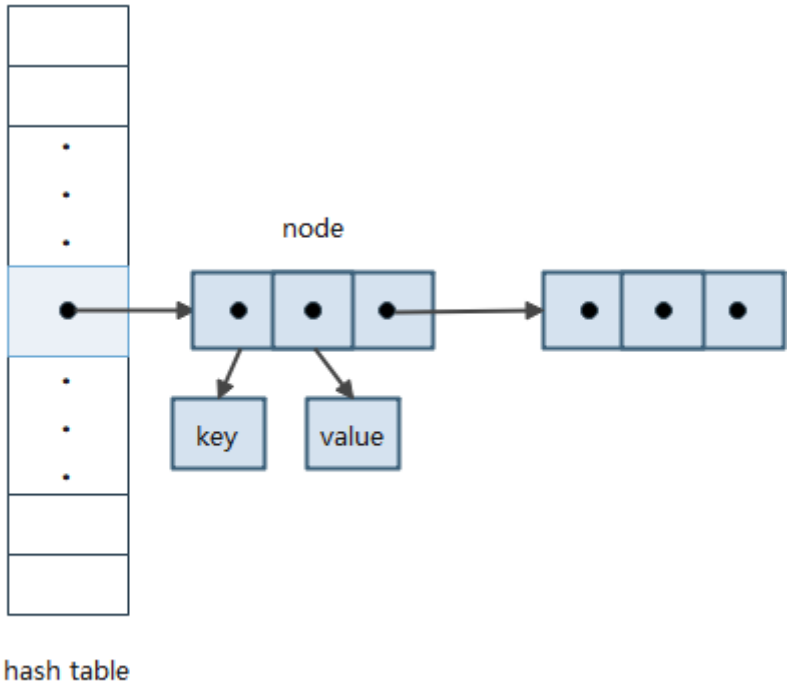
改变哈希表 的结构，使哈希表的每一个位置上不再只容纳一个元素，而是可以容纳多个

哈希表中的每一个位置都不止代表一个元素,而可以代表多个

> 我们把能够代表多个元素的位置形象的称为桶
>
> 我们称同一子集（桶）中的关键码互为同义词

桶可以表示为

> 线性表、有序线性表等



哈希效率衡量

装载因子

完美的哈希函数并不总是实际的，因此冲突的发生是不可避免的

为了能够度量哈希的效率，我们需要借用一个称为Load Factor的因子

$$\lambda = \frac{N}{M} \qquad N：实际记录个数 \quad M：哈希表长度$$

> 线性探查法：$\lambda < 0.3$效率比较好
>
> 平方探查法或双哈希法：$\lambda < 0.5$比较好
>
> 开散列法：$\lambda < 1$比较好