

算法4

1. 基础

1.1 栈

下压栈API:

```
public class Stack<Item> implements Iterable<Item>

    • boolean isEmpty();
    • int size();
    • void push(Item item);
    • Item pop();
```

1.1.1 使用数组实现的下压栈

```
1 import edu.princeton.cs.algs4.Stdout;
2 import java.util.Iterator;
3
4 public class ResizingArrayStack<Item> implements Iterable<Item> {
5     private Item[] array = (Item[]) new Object[1];
6     private int N = 0;
7
8     public boolean isEmpty() {
9         return N == 0;
10    }
11
12    public int size() {
13        return N;
14    }
15
16    public void push(Item item) {
17        if (N == array.length) resize(array.length * 2);
18        array[N++] = item;
19    }
20
21    public Item pop() {
22        Item item = array[--N];
23        array[N] = null;
24        if (N > 0 && N == array.length / 4)
25            resize(array.length / 2);
26        return item;
27    }
28
29    public Iterator<Item> iterator() {
30        return new ReverseArrayIterator();
31    }
32
33    private void resize(int max) {
34        Item[] temp = (Item[]) new Object[max];
35        for (int i = 0; i < N; i++)
36            temp[i] = array[i];
37        array = temp;
```

```

38     }
39
40     private class ReverseArrayIterator implements Iterator<Item> {
41         private int i = N;
42
43         public boolean hasNext() {
44             return i > 0;
45         }
46
47         public Item next() {
48             return array[~i];
49         }
50
51         public void remove() {
52             throw new UnsupportedOperationException();
53         }
54     }
55
56     public static void main(String[] args) {
57         ResizingArrayStack<Integer> resizingArrayStack = new
58         ResizingArrayStack<Integer>();
59
60         resizingArrayStack.push(20);
61         resizingArrayStack.push(32);
62         resizingArrayStack.push(321);
63         StdOut.println(resizingArrayStack.pop());
64         StdOut.println(resizingArrayStack.pop());
65         StdOut.println(resizingArrayStack.pop());
66     }

```

1.1.2 使用链表实现的下压栈

```

1 import edu.princeton.cs.algs4.StdOut;
2 import java.util.Iterator;
3
4 public class ListStack<Item> implements Iterable<Item> {
5     private Node first;
6     private int N = 0;
7
8     private class Node {
9         Item item;
10        Node next;
11    }
12
13     private class ListIterator implements Iterator<Item> {
14         private Node current = first;
15
16         public boolean hasNext() {
17             return current != null;
18         }
19
20         public Item next() {
21             Item item = current.item;
22             current = current.next;

```

```

23         return item;
24     }
25
26     public void remove() {
27         throw new UnsupportedOperationException();
28     }
29 }
30
31     public boolean isEmpty() {
32         return N == 0;
33     }
34
35     public int size() {
36         return N;
37     }
38
39     public void push(Item item) {
40         Node oldnode = first;
41         first = new Node();
42         first.item = item;
43         first.next = oldnode;
44         ++N;
45     }
46
47     public Item pop() {
48         Item item = first.item;
49         first = first.next;
50         --N;
51         return item;
52     }
53
54     public Iterator<Item> iterator() {
55         return new ListIterator();
56     }
57
58     public static void main(String[] args) {
59         ListStack<Integer> listStack = new ListStack<Integer>();
60
61         listStack.push(12);
62         listStack.push(2);
63         listStack.push(234);
64         StdOut.println(listStack.pop());
65         StdOut.println(listStack.pop());
66         StdOut.println(listStack.pop());
67     }
68 }
69

```

1.2 队列

队列API:

```

public class Queue<Item> implements Iterable<Item>

    • int size();
    • boolean isEmpty();

```

- `void enqueue(Item item);`
- `Item dequeue();`

```
1 import edu.princeton.cs.algs4.Stdout;
2
3 import java.util.Iterator;
4 import java.util.List;
5
6 public class Queue<Item> implements Iterable<Item> {
7     private Node first;
8     private Node last;
9     int N = 0;
10
11     private class Node {
12         Item item;
13         Node next;
14     }
15
16     private class ListIterator implements Iterator<Item> {
17         private Node current = first;
18
19         public boolean hasNext() {
20             return current != null;
21         }
22
23         public Item next() {
24             Item item = current.item;
25             current = current.next;
26             return item;
27         }
28
29         public void remove() {
30             throw new UnsupportedOperationException();
31         }
32     }
33
34     public boolean isEmpty() {
35         return N == 0;
36     }
37
38     public int size() {
39         return N;
40     }
41
42     public void enqueue(Item item) {
43         Node oldlast = last;
44         last = new Node();
45         last.item = item;
46         last.next = null;
47         if (isEmpty()) first = last;
48         else oldlast.next = last;
49         N++;
50     }
51
52     public Item dequeue() {
53         Item item = first.item;
54         first = first.next;
```

```

55     N--;
56     if (isEmpty()) last = null;
57     return item;
58 }
59
60 public ListIterator iterator() {
61     return new ListIterator();
62 }
63
64 public static void main(String[] args) {
65     Queue<Integer> queue = new Queue<Integer>();
66
67     queue.enqueue(32);
68     queue.enqueue(53);
69     queue.enqueue(2340);
70     for (Integer elem : queue) {
71         StdOut.println(elem);
72     }
73     StdOut.println(queue.dequeue());
74     StdOut.println(queue.dequeue());
75     StdOut.println(queue.dequeue());
76     StdOut.println("final size: " + queue.size());
77 }
78 }
```

1.3 背包

背包API:

```

public class Bag<Item> implements Iterable<Item>

    • int size();
    • boolean isEmpty();
    • void add(Item item);
```

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.ST;
3 import edu.princeton.cs.algs4.StdOut;
4
5 import java.util.Iterator;
6
7 public class Bag<Item> implements Iterable<Item> {
8     private Node first;
9     private int N = 0;
10
11     private class Node {
12         Item item;
13         Node next;
14     }
15
16     private class ListIterator implements Iterator<Item> {
17         private Node current = first;
18
19         public boolean hasNext() {
20             return current != null;
```

```

21     }
22
23     public Item next() {
24         Item item = current.item;
25         current = current.next;
26         return item;
27     }
28
29     public void remove() {
30         throw new UnsupportedOperationException();
31     }
32 }
33
34     public boolean isEmpty() {
35         return N == 0;
36     }
37
38     public int size() {
39         return N;
40     }
41
42     public void add(Item item) {
43         Node oldfirst = first;
44         first = new Node();
45         first.item = item;
46         first.next = oldfirst;
47         N++;
48     }
49
50     public Iterator iterator() {
51         return new ListIterator();
52     }
53
54     public static void main(String[] args) {
55         Bag<Integer> bag = new Bag<Integer>();
56
57         bag.add(120);
58         bag.add(32);
59         bag.add(42);
60         for (Integer elem : bag)
61             StdOut.println(elem);
62         StdOut.println("current size: " + bag.size());
63     }
64 }
```

1.4 链表

使用C实现的单链表：

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5
6 struct Node{
```

```
7     struct Node*next;
8     char value[1];/* 虽然value数组只有一个字节，但可以通过malloc扩展
9          字节大小，而且还能很好的获取其指针 */
10    };
11
12
13 struct List{
14     struct Node*first;
15     struct Node*last;
16     int ListSize;
17 };
18
19
20 /*
21 * 链表初始化
22 */
23 void ListInit(struct List*list){
24     list->first=NULL;
25     list->last=NULL;
26     list->ListSize=0;
27 }
28
29
30 /*
31 * 链表销毁
32 */
33 void ListDestroy(struct List*list){
34     struct Node*node,*temp;
35
36     node=list->first;
37     while(node!=NULL){
38         temp=node;
39         node=node->next;
40         free(temp);
41     }
42     list->first=NULL;
43     list->last=NULL;
44     list->ListSize=0;
45 }
46
47
48 /*
49 * 返回链表长度
50 */
51 static inline
52 int Size(const struct List*list){
53     return list->ListSize;
54 }
55
56
57 /*
58 * 链表是否为空
59 */
60 static inline
61 int isEmpty(const struct List*list){
62     return list->ListSize==0;
63 }
64
```

```
65
66 /* 
67 * 链表前向插入数据
68 */
69 struct Node* FInsert(struct List*list,void*value,size_t sz){
70     struct Node*newNode;
71
72     if((newNode=malloc(sizeof(struct Node)+sz-1))==NULL)
73         return NULL;
74     memcpy(newNode->value,value,sz);
75     newNode->next=list->first;
76     if(isEmpty(list)) list->last=newNode;
77     list->first=newNode;
78     list->ListSize++;
79     return newNode;
80 }
81
82
83 /*
84 * 链表后向插入
85 */
86 struct Node* BInsert(struct List*list,void*value,size_t sz){
87     struct Node*newNode;
88
89     if((newNode=malloc(sizeof(struct Node)+sz-1))==NULL)
90         return NULL;
91     memcpy(newNode->value,value,sz);
92     newNode->next=NULL;
93     if(isEmpty(list)) list->first=newNode;
94     else list->last->next=newNode;
95     list->last=newNode;
96     list->ListSize++;
97     return newNode;
98 }
99
100
101 /*
102 * 链表普通插入，当node==NULL等同于FInsert()
103 */
104 struct Node* Insert(struct List*list,struct Node*node,void*value,size_t sz)
105 {
106     struct Node*newNode;
107
108     if(node==NULL)
109         return FInsert(list,value,sz);
110     else if(list->first==NULL)
111         return NULL;
112     if((newNode=malloc(sizeof(struct Node)+sz-1))==NULL)
113         return NULL;
114     memcpy(newNode->value,value,sz);
115     newNode->next=node->next;
116     node->next=newNode;
117     if(list->last==node)
118         list->last=newNode;
119     list->ListSize++;
120     return newNode;
121 }
```

```
122 /*
123 * 链表前向删除元素
124 */
125 int FRemove(struct List*list,void*retval,size_t sz){
126     struct Node*temp;
127
128     if(isEmpty(list))
129         return -1;
130     if(retval!=NULL)//若retval非空才选择拷贝待删除元素中的数据
131         memcpy(retval,list->first->value,sz);
132     temp=list->first;
133     list->first=list->first->next;
134     free(temp);
135     if(list->ListSize==1)
136         list->last=NULL;
137     list->ListSize--;
138     return 0;
139 }
140
141
142 //不支持bremove()
143
144
145 /*
146 * 普通链表元素删除，当node==NULL时等同于FRemove()
147 */
148 int Remove(struct List*list,struct Node*node,void*retval,size_t sz){
149     struct Node*temp;
150
151     if(node==NULL)
152         return FRemove(list,retval,sz);
153     if(node->next==NULL)
154         return NULL;
155     temp=node->next;
156     node->next=temp->next;
157     if(list->last==temp)
158         list->last=node;
159     if(retval!=NULL)
160         memcpy(retval,temp->value,sz);
161     free(temp);
162     list->ListSize--;
163     return 0;
164 }
165
166
167 /*
168 * 链表遍历函数，使用用法类似于strtok()
169 */
170 struct Node* Iterator(struct List*list){
171     static struct Node*node=NULL;
172     struct Node*ret=NULL;
173
174     if(list!=NULL)
175         node=list->first;
176     if(node!=NULL){
177         ret=node;
178         node=node->next;
179     }
```

```
180     }
181     return ret;
182 }
183
184
185 /**
186 * 返回链表首元素指针
187 */
188 inline static
189 struct Node*First(struct List*list){
190     return list->first;
191 }
192
193
194 /**
195 * 返回链表尾元素指针
196 */
197 inline static
198 struct Node*Last(struct List*list){
199     return list->last;
200 }
201
202
203 /**
204 * 在链表中查找指定元素
205 */
206 struct Node*Find(const struct List*list,const void*val,size_t sz,
207                   int(*pf)(const void*,const void*)){
208     struct Node*node=list->first;
209
210     while(node!=NULL){
211         if(pf(node->value,val))
212             return node;
213         node=node->next;
214     }
215     return NULL;
216 }
217
218
219 int main(void){
220     struct List list;
221     double darr[10]={
222         1.2,1.4,3.12,3.1,8.3,
223         2.4,8.3,3.2,6.4,1.6
224     };
225
226     ListInit(&list);
227     for(int i=0;i<5;i++)
228         Insert(&list,NULL,(void*)&darr[i],sizeof(double));
229     for(int i=5;i<10;++i)
230         Insert(&list,list.last,(void*)&darr[i],sizeof(double));
231
232     for(const struct Node*p=Iterator(&list);
233          p!=NULL;p=Iterator(NULL)){
234         printf("%.2f\n",*(double*)(p->value));
235     }
236     printf("current list size: %d\n",Size(&list));
237     ListDestroy(&list);
```

```

238     printf("final list size: %d\n", size(&list));
239
240     return 0;
241 }

```

1.5 算法分析

表 1.4.7 对增长数量级的常见假设的总结

描述	增长的数量级	典型的代码	说 明	举 例
常数级别	1	a = b + c;	普通语句	将两个数相加
对数级别	$\log N$	(请见 1.1.10.2 节, 二分查找)	二分策略	二分查找
线性级别	N	<pre>double max = a[0]; for (int i = 1; i < N; i++) if (a[i] > max) max = a[i];</pre>	循环	找出最大元素
线性对数级别	$N \log N$	[请见算法 2.4]	分治	归并排序
平方级别	N^2	<pre>for (int i = 0; i < N; i++) for (int j = i+1; j < N; j++) if (a[i] + a[j] == 0) cnt++;</pre>	双层循环	检查所有元素对
立方级别	N^3	<pre>for (int i = 0; i < N; i++) for (int j = i+1; j < N; j++) for (int k = j+1; k < N; k++) if (a[i] + a[j] + a[k] == 0) cnt++;</pre>	三层循环	检查所有三元组
指数级别	2^N	(请见第 6 章)	穷举查找	检查所有子集

1.6 union-find 算法

union-find 的 API: `public class UF`

- `UF(int N)`
- `void find(int p)`
- `int find(int pos)`
- `boolean connected(int p, int q)`
- `int count()`

UF1: 使用数组遍历式方式进行连通实现

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.StdIn;
3 import edu.princeton.cs.algs4.StdOut;
4
5 public class UF1 {
6     private int[] id;
7     private int count;
8
9     public UF1(int N) {
10         id = new int[N];
11         count = N;
12         for (int i = 0; i < N; i++)
13             id[i] = i;
14     }

```

```

15
16     public int count() {
17         return count;
18     }
19
20     public int find(int pos) {
21         return id[pos];
22     }
23
24     public boolean connected(int p, int q) {
25         return find(p) == find(q);
26     }
27
28     public void union(int p, int q) {
29         int pID = find(p);
30         int qID = find(q);
31
32         if (pID == qID) return;
33         for (int i = 0; i < id.length; ++i)
34             if (id[i] == pID) id[i] = qID;
35         count--;
36     }
37
38     public static void main(String[] args) {
39         int[] array = In.readInts(args[0]);
40         int cnt = 0, N = array[cnt++];
41         UF1 uf1 = new UF1(N);
42
43         while (cnt < array.length) {
44             int p = array[cnt++];
45             int q = array[cnt++];
46             if (uf1.connected(p, q)) continue;
47             uf1.union(p, q);
48             StdOut.println(p + " " + q);
49         }
50         StdOut.println(uf1.count() + " components");
51     }
52 }
```

UF2: 使用节点连接到同一个树根的方式完成连通分量的来链接

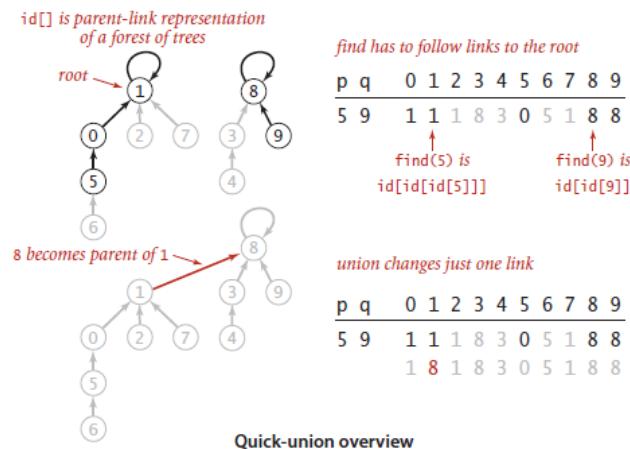
```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.StdOut;
3
4 public class UF2 {
5     private int[] id;
6     private int count;
7
8     public UF2(int N) {
9         id = new int[N];
10        count = N;
11        for (int i = 0; i < N; i++)
12            id[i] = i;
13    }
14 }
```

```

15     public int count() {
16         return count;
17     }
18
19     public int find(int pos) {
20         while (pos != id[pos]) pos = id[pos];
21         return pos;
22     }
23
24     public boolean connected(int p, int q) {
25         return find(p) == find(q);
26     }
27
28     public void union(int p, int q) {
29         int pID = find(p);
30         int qID = find(q);
31
32         if (pID != qID) {
33             id[pID] = qID;
34             count--;
35         }
36     }
37
38     public static void main(String[] args) {
39         int[] array = In.readInts(args[0]);
40         int cnt = 0, N = array[cnt++];
41         UF2 uf2 = new UF2(N);
42
43         while (cnt < array.length) {
44             int p = array[cnt++];
45             int q = array[cnt++];
46             if (uf2.connected(p, q)) continue;
47             uf2.union(p, q);
48             StdOut.println(p + " " + q);
49         }
50         StdOut.println(uf2.count() + " connected");
51     }
52 }
```

图示链接过程：



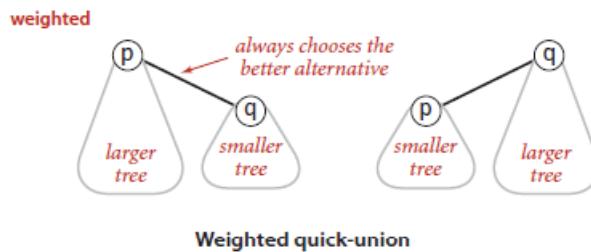
UF3：在UF2的基础上增加一个记录连通分量的节点数的数组，每次进行挂载（链接）的时候根据连通分量树的大小将小树挂在大树的根节点上

```
1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.StdOut;
3
4 public class UF3 {
5     private int[] id;
6     private int[] sz;
7     private int count;
8
9     public UF3(int N) {
10         id = new int[N];
11         sz = new int[N];
12         count = N;
13         for (int i = 0; i < N; i++) {
14             id[i] = i;
15             sz[i] = 1;
16         }
17     }
18
19     public int count() {
20         return count;
21     }
22
23     public int find(int pos) {
24         while (pos != id[pos]) pos = id[pos];
25         return pos;
26     }
27
28     public boolean connected(int p, int q) {
29         return find(p) == find(q);
30     }
31
32     public void union(int p, int q) {
33         int pID = find(p);
34         int qID = find(q);
35
36         if (pID != qID) {
37             //将小树挂在大树的根节点上
38             if (sz[pID] < sz[qID]) {
39                 id[pID] = qID;
40                 sz[qID] += sz[pID];
41             } else {
42                 id[qID] = pID;
43                 sz[pID] += sz[qID];
44             }
45             count--;
46         }
47     }
48
49     public static void main(String[] args) {
50         int[] array = In.readInts(args[0]);
51         int cnt = 0, N = array[cnt++];
52         UF3 uf3 = new UF3(N);
53
54         while (cnt < array.length) {
```

```

55         int p = array[cnt++];
56         int q = array[cnt++];
57         if (uf3.connected(p, q)) continue;
58         uf3.union(p, q);
59         Stdout.println(p + " " + q);
60     }
61     Stdout.println(uf3.count() + " connected");
62 }
63 }
```

链接图示：



2. 排序

公共类API: `public class Sort`

- `private static boolean less(Comparable lhs, Comparable rhs)`
- `private static void swap(Comparable[] a, int i, int j)`
- `private static void show(Comparable[] a)`
- `public static boolean issorted(Comparable[] a)`
- `public static void sort(Comparable[] a)`

封装sort静态方法的类展示如下，后面只展示sort()方法

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Stdout;
3 import edu.princeton.cs.algs4.StdRandom;
4 import jdk.nashorn.internal.ir.LiteralNode;
5
6 public class Sort {
7     private static boolean less(Comparable lhs, Comparable rhs) {
8         return lhs.compareTo(rhs) < 0;
9     }
10
11     private static void swap(Comparable[] a, int i, int j) {
12         Comparable temp = a[i];
13         a[i] = a[j];
14         a[j] = temp;
15     }
16
17     private static void show(Comparable[] a) {
18         for (int i = 0; i < a.length; ++i)
19             Stdout.println(a[i]);
20     }
21
22     public static boolean isSorted(Comparable[] a) {
23         for (int i = 1; i < a.length; i++)
```

```

14         if (less(a[i], a[i - 1]))
15             return false;
16     return true;
17 }
18
19 public static void sort(Comparable[] a) {
20     //见下
21 }
22
23 public static void main(String[] args) {
24     Integer[] array = new Integer[100];
25     for (int i = 0; i < array.length; ++i)
26         array[i] = (Integer) StdRandom.uniform(0, 100);
27
28     sort(array);
29     show(array);
30 }
31
32 }
```

2.1 初级排序

2.1.1 冒泡排序

慢慢的将当前剩余最大的值交换到相对最后位置，如同气泡向水面上冒一样。时间复杂度\$N^2\$

```

1 public static void sort(Comparable[] a) {
2     for (int i = a.length; i > 0; i--) {
3         for (int j = 1; j < i; ++j)
4             if (less(a[j], a[j - 1]))
5                 swap(a, j, j - 1);
6     }
7 }
```

2.1.2 选择排序

核心思想就是将依次最小的元素放在数组最前面。时间复杂度\$N^2\$

```

1 public static void sort(Comparable[] a) {
2     for (int i = 0; i < a.length; ++i) {
3         int min = i;
4         for (int j = i + 1; j < a.length; ++j) {//选取最小下标
5             if (less(a[j], a[min]))
6                 min = j;
7         }
8         swap(a, i, min);
9     }
10 }
```

2.1.3 插入排序

核心思想就是将第j个元素插入到数组前j-1个元素的某个恰当位置中。时间复杂度\$N^2\$

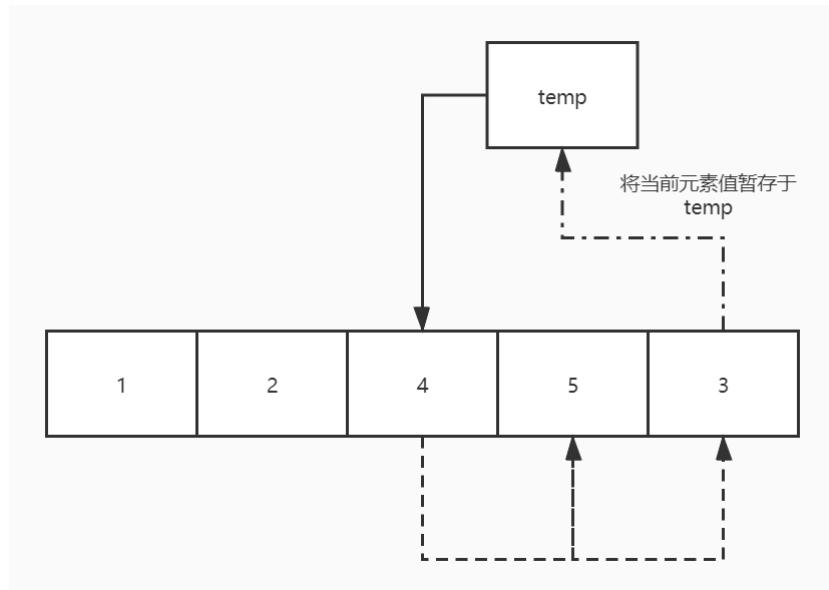
解法1：

```
1 public static void sort(Comparable[] a) {  
2     for (int i = 1; i < a.length; ++i) {  
3         for (int j = i; j > 0 && less(a[j], a[j - 1]); --j)  
4             swap(a, j, j - 1);  
5     }  
6 }
```

解法2：采用移动而不是交换，节省交换成本

```
1 public static void sort(Comparable[] a) {  
2     for (int i = 1, j; i < a.length; ++i) {  
3         Comparable temp = a[i];  
4         for (j = i; j > 0 && less(temp, a[j - 1]); j--)  
5             a[j] = a[j - 1];  
6         a[j] = temp;  
7     }  
8 }
```

图示：



2.14 希尔排序

希尔排序的核心思想就是：先对相隔一段间距的元素进行插入排序使得数组部分有序，然后减少间隔量最终变成完全的插入排序，此时达到最终的有序。时间复杂度小于\$N^2\$，递增序列\$1/2(3^k-1)\$可以做到\$N^{3/2}\$

解法1：

```

1  public static void sort(Comparable[] a) {
2      for (int h = a.length / 3; h >= 1; h /= 3) {
3          for (int i = h; i < a.length; i++)
4              for (int j = i; j >= h && less(a[j], a[j - h]); j -= h)
5                  swap(a, j, j - h);
6      }
7  }

```

解法2：

```

1  public static void sort(Comparable[] a) {
2      for (int h = a.length / 3; h >= 1; h /= 3) {
3          for (int i = h, j; i < a.length; ++i) {
4              Comparable temp = a[i];
5              for (j = i; j >= h && less(temp, a[j - h]); j -= h)
6                  a[j] = a[j - h];
7              a[j] = temp;
8          }
9      }
10 }

```

2.2 归并排序

自顶向下的归并排序，即使用了分而治之的方法将大数组分解成小数组然后进行归并

```

1  private static Comparable[] aux;
2
3  //归并操作
4  public static void merge(Comparable[] a, int low, int mid, int high) {
5      int i = low, j = mid + 1;
6      for (int k = low; k <= high; ++k)
7          aux[k] = a[k];
8
9      for (int k = low; k <= high; ++k) {
10         if (i > mid)
11             a[k] = aux[j++];
12         else if (j > high)
13             a[k] = aux[i++];
14         else if (less(aux[i], aux[j]))
15             a[k] = aux[i++];
16         else
17             a[k] = aux[j++];
18     }
19 }
20
21 //递归式归并排序操作
22 private static void msot(Comparable[] a, int low, int high) {
23     if (high - low <= 0) return;
24     // else if (high - low < 15)
25     //     InsertSort(a, low, high);
26
27     int mid = low + (high - low) / 2;
28     msot(a, low, mid);
29     msot(a, mid + 1, high);

```

```

30         merge(a, low, mid, high);
31     }
32
33     //封装了msort的归并排序操作
34     public static void sort(Comparable[] a) {
35         aux = new Comparable[a.length];
36         msort(a, 0, a.length - 1);
37     }

```

改进版归并排序：当分割后的数组规模小于15时采用插入排序进行解决，而不再继续使用分割（因为插入排序对于小数组而言可能比归并排序更快）

```

1  private static Comparable[] aux;
2
3  //归并操作
4  public static void merge(Comparable[] a, int low, int mid, int high) {
5      int i = low, j = mid + 1;
6      for (int k = low; k <= high; ++k)
7          aux[k] = a[k];
8
9      for (int k = low; k <= high; ++k) {
10         if (i > mid)
11             a[k] = aux[j++];
12         else if (j > high)
13             a[k] = aux[i++];
14         else if (less(aux[i], aux[j]))
15             a[k] = aux[i++];
16         else
17             a[k] = aux[j++];
18     }
19 }
20
21 //插入排序
22 public static void InsertSort(Comparable[] a, int low, int high) {
23     for (int i = low, j; i <= high; ++i) {
24         Comparable temp = a[i];
25         for (j = i; j > low && less(temp, a[j - 1]); --j)
26             a[j] = a[j - 1];
27         a[j] = temp;
28     }
29 }
30
31 //递归式的归并排序
32 private static void msort(Comparable[] a, int low, int high) {
33     if (high - low <= 0) return;
34     else if (high - low < 15)
35         InsertSort(a, low, high);
36     else {
37         int mid = low + (high - low) / 2;
38         msort(a, low, mid);
39         msort(a, mid + 1, high);
40         //若该局部数组已经有序，则不需要再进入merge之中
41         if (less(a[mid + 1], a[mid]))
42             merge(a, low, mid, high);
43     }
44 }
45

```

```

46     //封装递归之后的归并排序
47     public static void sort(Comparable[] a) {
48         aux = new Comparable[a.length];
49         msot(a, 0, a.length - 1);
50     }

```

自顶向下的归并排序，其运行方向与上面相反，从低端从顶端运行，先两两归并，然后归并。。。最后整体归并

```

1  private static Comparable[] aux;
2
3  //归并操作
4  public static void merge(Comparable[] a, int low, int mid, int high) {
5      int i = low, j = mid + 1;
6      for (int k = low; k <= high; ++k)
7          aux[k] = a[k];
8
9      for (int k = low; k <= high; ++k) {
10         if (i > mid)
11             a[k] = aux[j++];
12         else if (j > high)
13             a[k] = aux[i++];
14         else if (less(aux[i], aux[j]))
15             a[k] = aux[i++];
16         else
17             a[k] = aux[j++];
18     }
19 }
20
21 //封装递归之后的归并排序
22 public static void sort(Comparable[] a) {
23     int N = a.length;
24     aux = new Comparable[N];
25     for (int sz = 1; sz < N; sz *= 2) {
26         for (int lo = 0; lo < N - sz; lo += sz * 2)
27             merge(a, lo, lo + sz - 1, Math.min(N - 1, lo + sz * 2 - 1));
28     }
29 }

```

2.3 快速排序

2.3.1 普通快速排序

```

1  private static int partition(Comparable[] a, int low, int high) {
2      int i = low, j = high + 1;
3      Comparable key = a[low];
4
5      while (true) {
6          while (less(a[++i], key))
7              if (i == high) break;
8          while (less(key, a[--j]))
9              if (j == low) break;
10         if (i >= j) break;
11         swap(a, i, j);

```

```

12     }
13     swap(a, low, j);
14     return j;
15 }
16
17 public static void qsort(Comparable[] a, int low, int high) {
18     if (low >= high) return;
19
20     int pos = partition(a, low, high);
21     qsort(a, low, pos - 1);
22     qsort(a, pos + 1, high);
23 }
24
25 public static void sort(Comparable[] a) {
26     qsort(a, 0, a.length - 1);
27 }

```

快速排序的改进版1：面对小数组使用插入排序进行代替

```

1  public static void InsertionSort(Comparable[] a, int low, int high) {
2      for (int i = low, j; i < high; i++) {
3          Comparable temp = a[low];
4          for (j = i; j > 0 && less(a[j], a[j - 1]); --j)
5              a[j] = a[j - 1];
6          a[j] = temp;
7      }
8  }
9
10 public static void qsort(Comparable[] a, int low, int high) {
11     if (high - low <= 0) return;
12     else if (high - low < 10) {
13         Insertionsort(a, low, high);
14         return;
15     }
16
17     int pos = partition(a, low, high);
18     qsort(a, low, pos - 1);
19     qsort(a, pos + 1, high);
20 }
21
22 public static void sort(Comparable[] a) {
23     qsort(a, 0, a.length - 1);
24 }

```

快速排序改进版2：使用三取样+小数组转而用插入排序完成的方式来加速数组排序

```

1 //三取样确定切分元素位置，保证key值不大不小
2 private static int ThreeMid(Comparable[] a, int low, int high) {
3     Integer[] temp = {low, low + (high - low) / 2, high};
4     for (int i = 1, j; i < temp.length; ++i) {
5         Integer t = temp[i];
6         for (j = i; j > 0 && less(a[j], a[j - 1]); --j)
7             temp[j] = temp[j - 1];
8         temp[j] = t;
9     }
10    return temp[1];

```

```

11    }
12
13    public static int partition(Comparable[] a, int low, int high) {
14        int i = low, j = high + 1;
15        swap(a, ThreeMid(a, low, high), low);
16        Comparable key = a[low];
17
18        while (true) {
19            while (less(a[++i], key))
20                if (i == high) break;
21            while (less(key, a[--j]))
22                if (j == low) break;
23            if (i >= j) break;
24            swap(a, i, j);
25        }
26        swap(a, low, j);
27        return j;
28    }
29
30    public static void InsertionSort(Comparable[] a, int low, int high) {
31        for (int i = low, j; i <= high; ++i) {
32            Comparable t = a[i];
33            for (j = i; j > low && less(a[j], a[j - 1]); --j)
34                a[j] = a[j - 1];
35            a[j] = t;
36        }
37    }
38
39    public static void qsort(Comparable[] a, int low, int high) {
40        if (high - low <= 0) return;
41        else if (high - low < 10) {
42            InsertionSort(a, low, high);
43            return;
44        }
45
46        int mid = partition(a, low, high);
47        qsort(a, low, mid - 1);
48        qsort(a, mid + 1, high);
49    }
50
51    public static void sort(Comparable[] a) {
52        qsort(a, 0, a.length - 1);
53    }

```

2.3.2 三切分快速排序

```

1 //3取样
2 private static int ThreeMid(Comparable[] a, int low, int high) {
3     Integer[] temp = {low, low + (high - low) / 2, high};
4     for (int i = 1, j; i < temp.length; ++i) {
5         Integer t = temp[i];
6         for (j = i; j > 0 && less(a[j], a[j - 1]); --j)
7             temp[j] = temp[j - 1];
8         temp[j] = t;
9     }
10    return temp[1];
11}

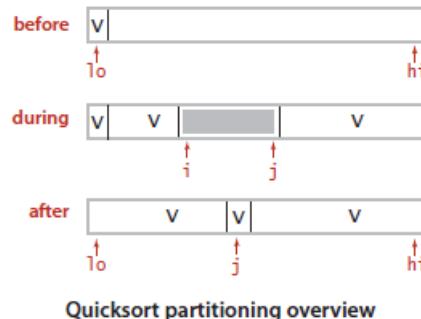
```

```

12
13     public static void qsort(Comparable[] a, int low, int high) {
14         if (high - low <= 0) return;
15
16         swap(a, low, ThreeMid(a, low, high)); //用到了3取样方法
17         int lt = low, i = low + 1, gt = high;
18         Comparable v = a[low];
19
20         while (i <= gt) {
21             int cmp = a[i].compareTo(v);
22             if (cmp < 0)
23                 swap(a, lt++, i++);
24             else if (cmp > 0)
25                 swap(a, gt--, i);
26             else
27                 i++;
28         }
29         qsort(a, low, lt - 1);
30         qsort(a, gt + 1, high);
31     }
32
33     public static void sort(Comparable[] a) {
34         qsort(a, 0, a.length - 1);
35     }

```

图示：



2.4 优先队列

优先队列API：

```

public class MaxPQ<Key extends Comparable<Key>>

    • MaxPQ()
    • MaxPQ(int max)
    • MaxPQ(Key[] a)
    • void insert(Key v)
    • Key max()
    • Key delMax()
    • boolean isEmpty()
    • int size()

```

2.4.1 最大堆

一个允许动态调整的最大二叉堆：

```
1 import edu.princeton.cs.algs4.StdOut;
2
3 public class MaxPQ<Key extends Comparable<Key>> {
4     private Key[] pq;
5     private int capacity;
6     private int N = 0;
7
8     public MaxPQ(int maxN) {
9         pq = (Key[]) new Comparable[maxN + 1];
10        capacity = maxN;
11    }
12
13     public boolean isEmpty() {
14         return N == 0;
15     }
16
17     public int size() {
18         return N;
19     }
20
21     //插入操作
22     public void insert(Key v) {
23         if (isFull())
24             resize(capacity * 2);
25         pq[++N] = v;
26         swim(N);
27     }
28
29     //删除最大堆中最大值元素，并返回
30     public Key delMax() {
31         //这里可以加入越界检查，不过也无所谓
32         Key max = pq[1];
33         pq[1] = pq[N--];
34         pq[N + 1] = null;
35         sink(1);
36
37         if ((N + 1) < capacity / 4)
38             resize(capacity / 2);
39         return max;
40     }
41
42     /* 工具方法： */
43
44     private boolean less(int i, int j) {
45         return pq[i].compareTo(pq[j]) < 0;
46     }
47
48     private void swap(int i, int j) {
49         Key temp = pq[i];
50         pq[i] = pq[j];
51         pq[j] = temp;
52     }
53 }
```

```

54     //使指定位置元素上浮，直到合理位置为止
55     private void swim(int k) {
56         while (k > 1 && less(k / 2, k)) {
57             swap(k, k / 2);
58             k /= 2;
59         }
60     }
61
62     //使指定位置元素下沉，直到合理位置为止
63     private void sink(int k) {
64         while (2 * k <= N) {
65             int j = k * 2;
66             if (j < N && less(j, j + 1)) j++;
67             if (!less(k, j)) break;
68             swap(k, j);
69             k = j;
70         }
71     }
72
73     private void resize(int maxN) {
74         Key[] temp = (Key[]) new Comparable[maxN];
75         for (int i = 1; i <= N; ++i)
76             temp[i] = pq[i];
77         capacity *= 2;
78         pq = temp;
79     }
80
81     private boolean isFull() {
82         return N + 1 == capacity;
83     }
84
85     public static void main(String[] args) {
86         MaxPQ<Integer> maxPQ = new MaxPQ<Integer>(110);
87
88         for (int i = 0; i < 200; i++)
89             maxPQ.insert(i);
90         while (!maxPQ.isEmpty())
91             Stdout.println(maxPQ.delMax());
92     }
93 }
94

```

2.4.2 索引优先队列

索引优先队列的作用在于允许用户引用；已经进入优先队列的元素，其关键就是给其中每一个元素添加一个索引。我们可以认为这种数据结构就是一种能够快速访问其中最小元素的数组。其API如下所示：

```

public class IndexMinPQ<Item extends Comparable<Item>>

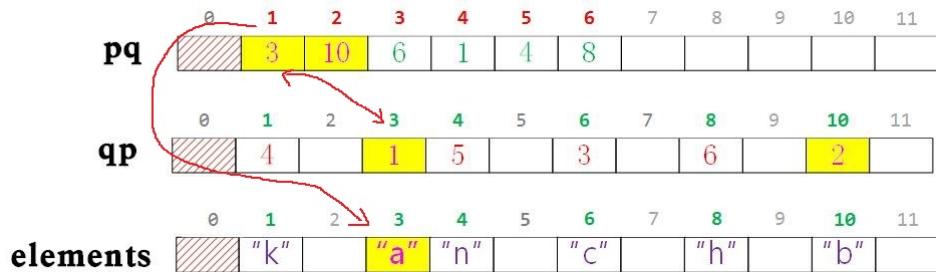
• IndexMinPQ(int maxN)
• void insert(int k, Item item)
• void change(int k, Item item)
• boolean contains(int k)
• void delete(int k)
• Item min()
• int minIndex()

```

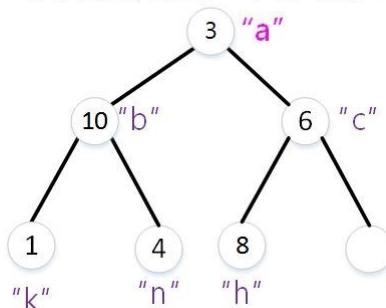
- `int delMin()`
- `boolean isEmpty()`
- `int size()`

具体的原理可以参看如下这篇博文：

[索引优先队列的工作原理与简易实现\)](#)



等价的完全二叉树形式



```

1 public class IndexMinPQ<Key extends Comparable<Key>> {
2     private int[] pq; //索引二叉堆，保存进行优先排序的元素在keys数组中的索引
3     private int[] qp; //存储在某一个索引在索引二叉堆中的下标
4     private Key[] keys; //保存索引二叉堆中的真正优先级元素，若值为-1表示没有存储元素
5     int maxN;
6     int n;
7
8     public IndexMinPQ(int maxN) {
9         this.maxN = maxN;
10        this.n = 0;
11        pq = new int[maxN + 1];
12        qp = new int[maxN + 1];
13        keys = (Key[]) new Comparable[maxN + 1];
14        for (int i = 0; i <= maxN; ++i)
15            qp[i] = -1;
16    }
17
18    public int size() {
19        return n;
20    }
21
22    public boolean isEmpty() {
23        return n == 0;
24    }
25
26    public boolean contains(int i) {
27        return qp[i] != -1;
28    }
29
  
```

```

30     public void insert(int i, Key key) {
31         validateIndex(i);
32         if (!contains(i))
33             throw new IllegalArgumentException("");
34
35         n++;
36         qp[i] = n;
37         pq[n] = i;
38         keys[i] = key;
39         swim(n);
40     }
41
42     public void delete(int i) {
43         validateIndex(i);
44         if (!contains(i))
45             throw new IllegalArgumentException("");
46
47         int index = qp[i];
48         swap(index, n--);
49         swim(index);
50         sink(index);
51         keys[i] = null;
52         qp[i] = -1;
53     }
54
55     public void changeKey(int i, Key key) {
56         validateIndex(i);
57         if (!contains(i))
58             throw new IllegalArgumentException("");
59
60         keys[i] = key;
61         swim(qp[i]);
62         sink(qp[i]);
63     }
64
65     public Key min() {
66         if (isEmpty())
67             throw new IllegalArgumentException("");
68         return keys[pq[1]];
69     }
70
71     public int minIndex() {
72         if (isEmpty())
73             throw new IllegalArgumentException("");
74         return pq[1];
75     }
76
77     public int delMin() {
78         if (isEmpty())
79             throw new IllegalArgumentException("");
80
81         int minIndex = pq[1];
82         swap(1, n--);
83         sink(1);
84         qp[minIndex] = -1;
85         keys[minIndex] = null;
86         pq[n + 1] = -1;
87         return minIndex;

```

```

88 }
89
90     private void validateIndex(int i) {
91         if (i < 0 && i >= maxN)
92             throw new IllegalArgumentException("");
93     }
94
95     private boolean greater(int i, int j) {
96         return keys[pq[i]].compareTo(keys[pq[j]]) > 0;
97     }
98
99     //交换优先队列中的两个元素的索引，因此需要交换pq和qp
100    private void swap(int i, int j) {
101        int tmp = pq[i];
102        pq[i] = pq[j];
103        pq[j] = tmp;
104        qp[pq[i]] = i;
105        qp[pq[j]] = j;
106    }
107
108    private void sink(int k) {
109        while (2 * k <= n) {
110            int j = 2 * k;
111            if (j < n && greater(j, j + 1)) j++;
112            if (!greater(k, j)) break;
113            swap(k, j);
114            k = j;
115        }
116    }
117
118    private void swim(int k) {
119        while (k > 1 && greater(k / 2, k)) {
120            swap(k / 2, k);
121            k /= 2;
122        }
123    }
124 }

```

2.4.3 堆排序

核心思想：先将需要排序的数组构建成为最大堆，然后与最后一个元素进行交换、递减堆大小并重新对根节点做下层操作，重复上述操作直到堆大小变成为0。构建最大堆是最初最重要的一步：从底($N/2-1$ 位置开始)向上对每一个元素执行下层操作。

```

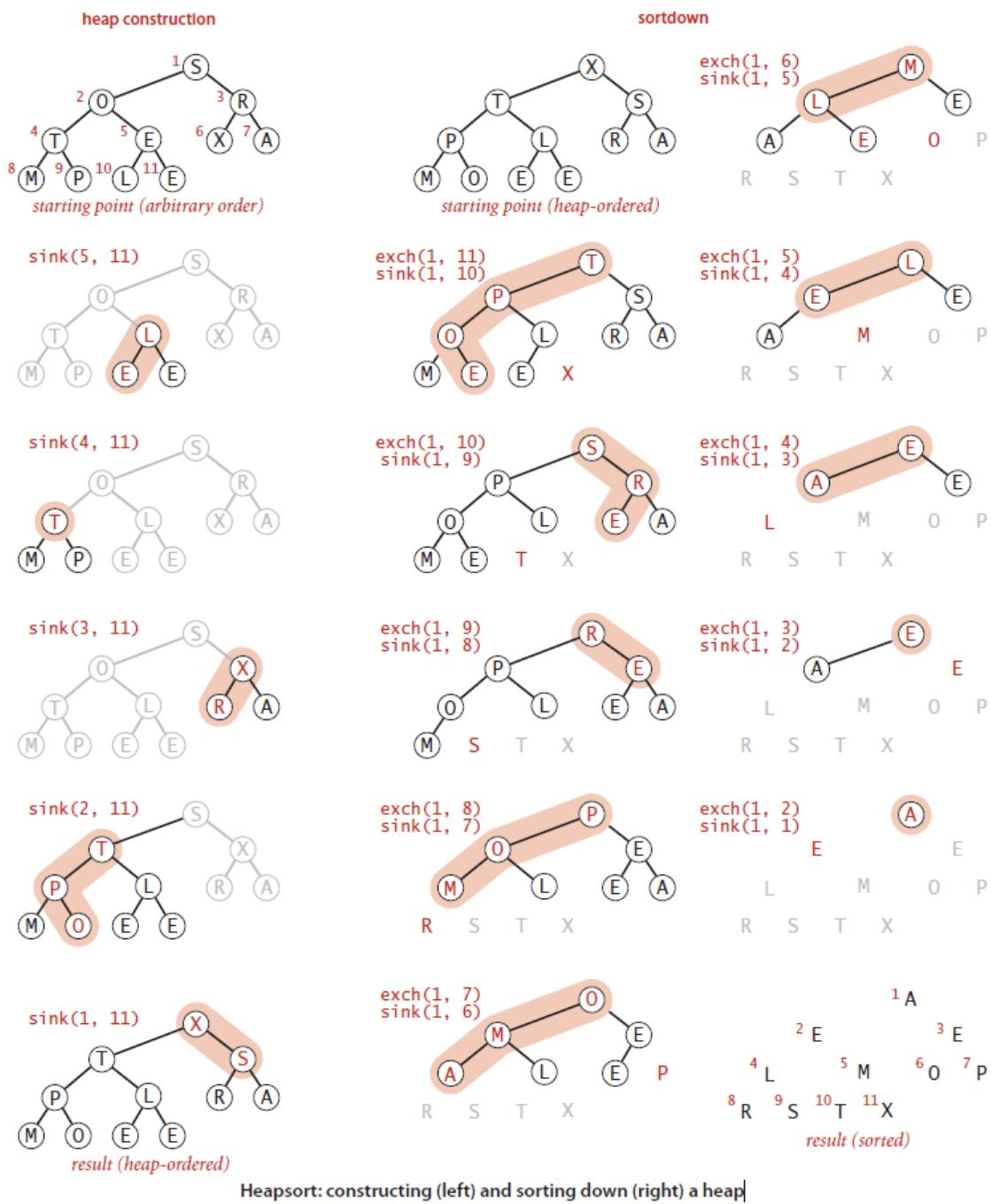
1 import edu.princeton.cs.algs4.StdOut;
2
3 public class Sort {
4     private static void swap(Comparable[] a, int i, int j) {
5         Comparable temp = a[j];
6         a[j] = a[i];
7         a[i] = temp;
8     }
9

```

```

10     private static boolean less(Comparable lhs, Comparable rhs) {
11         return lhs.compareTo(rhs) < 0;
12     }
13
14     //元素下沉函数
15     private static void sink(Comparable[] a, int k, int N) {
16         while (2 * k + 1 < N) {
17             int j = 2 * k + 1;
18             if (j < N - 1 && less(a[j], a[j + 1])) j++;
19             if (!less(a[k], a[j])) break;
20             swap(a, k, j);
21             k = j;
22         }
23     }
24
25     //堆排序
26     public static void sort(Comparable[] a) {
27         int N = a.length;
28         /* 从底向上使用sink()函数构建最大堆（注意是从那个最后一个
29             右子节点的节点开始sink()最大堆有序化） */
30         for (int k = N / 2 - 1; k >= 0; --k)
31             sink(a, k, N);
32         while (N > 0) {
33             swap(a, 0, --N);
34             sink(a, 0, N);
35         }
36     }
37
38     public static void show(Comparable[] a) {
39         for (Comparable item : a)
40             Stdout.println(item);
41     }
42
43     public static void main(String[] args) {
44         Integer[] array = new Integer[100];
45         for (int i = 0; i < array.length; ++i)
46             array[i] = array.length - i;
47         sort(array);
48         show(array);
49     }
50 }
```

图示：



C语言实现：

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define ARRAYLENGTH 100
6
7 static void swap(int array[],int i,int j){
8     int temp=array[i];
9     array[i]=array[j];
10    array[j]=temp;
11 }
12
13
14 /* 最大堆元素下沉操作 */
15 static void sink(int array[],int k,int N{
16     int lchild,rchild,maxchild;

```

```

17     while(2*k+1<N){
18         lchild=2*k+1;
19         rchild=2*k+2;
20         maxchild=lchild;
21         //想要实现逆序排序只要改变1、2两点的比较运算符即可
22         if(rchild<N&&array[lchild]<array[rchild])//1
23             maxchild=rchild;
24         if(array[maxchild]<array[k])//2
25             break;
26         swap(array,k,maxchild);
27         k=maxchild;
28     }
29 }
30
31
32
33 /* 堆排序 */
34 void HeapSort(int array[],int N){
35     for(int k=N/2-1;k>=0;--k)
36         sink(array,k,N);
37     while(N>0){
38         swap(array,0,--N);
39         sink(array,0,N);
40     }
41 }
42
43
44 void show(int array[],int N){
45     for(int i=0;i<N;++i)
46         printf("%d\n",array[i]);
47 }
48
49
50 int main(void)
51 {
52     int array[ARRAYLENGTH];
53     for(int i=0;i<ARRAYLENGTH;++i)
54         array[i]=ARRAYLENGTH-i;
55     HeapSort(array,ARRAYLENGTH);
56     show(array,ARRAYLENGTH);
57 }
```

3. 查找

3.1 符号表

无序符号表API (也同样是有序符号表共同拥有的) : `public class ST<Key,Value>`

- `ST()`
- `void put(Key key,value value)`
- `value get(Key key)`
- `void delete(Key key)`
- `boolean contains(Key key)`
- `boolean isEmpty()`

- `int size()`
- `Iterable<Key> keys()`

有序符号表增加的API:

- `Key min()`
- `Key max()`
- `Key floor(Key key)`
- `Key ceiling(Key key)`
- `int rank(Key key)`
- `Key select(int k)`
- `void deleteMin()`
- `void deleteMax()`
- `int size(Key lo, Key hi)`
- `Iterable<Key> keys(Key lo, Key hi)`
- `Iterable<Key> keys()`

对于符号表（键-值对容器，在C++对应于关联容器`std::map`）来说，最重要的两个操作为 `void put(Key key, value val)` 和 `value get(Key key)`，分别对应着符号表的插入和搜索操作，其时间复杂度关乎着该容器的好坏。

3.1.1 无序链表符号表

容器插入操作 `put()` 时间复杂度: \$N\$

容器查找操作 `get()` 时间复杂度: \$N\$

```

1 import edu.princeton.cs.algs4.Queue;
2 import edu.princeton.cs.algs4.StdOut;
3
4 import java.util.Collection;
5 import java.util.Iterator;
6
7 public class SeqSearchST<Key, value> {
8     private Node first;
9     private int N = 0;
10
11     private class Node {
12         Key key;
13         value val;
14         Node next;
15
16         public Node(Key key, value val, Node next) {
17             this.key = key;
18             this.val = val;
19             this.next = next;
20         }
21     }
22
23     public SeqSearchST() {
24     }
25
26     //获取指定键key对应的值val
27     public value get(Key key) {
28         for (Node x = first; x != null; x = x.next) {

```

```
29             if (key.equals(x.key))
30                 return x.val;
31         }
32     }
33 }
34
35 //添加键值对
36 public void put(Key key, Value val) {
37     for (Node x = first; x != null; x = x.next) {
38         if (key.equals(x.key)) {
39             x.val = val;
40             return;
41         }
42     }
43     first = new Node(key, val, first);
44     N++;
45 }
46
47 public void delete(Key key) {
48     put(key, null);
49     N--;
50 }
51
52 public boolean contains(Key key) {
53     return get(key) != null;
54 }
55
56 public boolean isEmpty() {
57     return N == 0;
58 }
59
60 public int size() {
61     return N;
62 }
63
64 /* Iterable指的是一个可迭代的容器（它必然实现了一个iterator()成员）
   ，而Iterator是作用于其上的迭代器 */
65
66 public Iterable<Key> keys() {
67     Queue<Key> queue = new Queue<Key>();
68     for (Node x = first; x != null; x = x.next)
69         queue.enqueue(x.key);
70     return queue;
71 }
72
73 public static void main(String[] args) {
74     SeqSearchST<String, Integer> seqSearchST =
75         new SeqSearchST<String, Integer>();
76
77     seqSearchST.put("hello", 32);
78     seqSearchST.put("show", 3);
79     seqSearchST.put("world", 5);
80     seqSearchST.put("code", 6);
81     for (String key : seqSearchST.keys())
82         Stdout.println(key + " " + seqSearchST.get(key));
83     Stdout.println("size: " + seqSearchST.size());
84 }
85 }
```

3.1.2 有序数组符号表

容器插入操作 put() 时间复杂度: $\$N\$$

容器查找操作 get() 时间复杂度: $\$logN\$$

```
1 import edu.princeton.cs.algs4.Queue;
2 import edu.princeton.cs.algs4.Stdout;
3
4 public class BinarySearchST<Key extends Comparable<Key>, Value> {
5     private Key[] keys;
6     private Value[] vals;
7     private int capacity = 0;
8     private int N = 0;
9
10    //重新调整符号表容器大小
11    private void resize(int capacity) {
12        this.capacity = capacity;
13        Key[] newkeys = (Key[]) new Comparable[capacity];
14        Value[] newvals = (Value[]) new Object[capacity];
15        for (int i = 0; i < N; ++i) {
16            newkeys[i] = keys[i];
17            newvals[i] = vals[i];
18        }
19        keys = newkeys;
20        vals = newvals;
21    }
22
23    public BinarySearchST(int capacity) {
24        this.capacity = capacity;
25        this.keys = (Key[]) new Comparable[capacity];
26        this.vals = (Value[]) new Object[capacity];
27    }
28
29    public boolean contains(Key key) {
30        return get(key) != null;
31    }
32
33    public boolean isEmpty() {
34        return N == 0;
35    }
36
37    public int size() {
38        return N;
39    }
40
41    //返回小于等于指定键key的键的数量
42    public int rank(Key key) {
43        int low = 0, high = N - 1;
44        while (low <= high) {
45            int mid = low + (high - low) / 2;
46            int cmp = key.compareTo(keys[mid]);
47            if (cmp < 0) high = mid - 1;
48            else if (cmp > 0) low = mid + 1;
49            else return mid;
50        }
51    }
52}
```

```

51     return low;
52 }
53
54 public value get(key key) {
55     if (isEmpty()) return null;
56     int i = rank(key);
57     if (i < N && keys[i].compareTo(key) == 0)
58         return vals[i];
59     return null;
60 }
61
62 public void put(key key, value val) {
63     int i = rank(key);
64     if (i < N && keys[i].compareTo(key) == 0) {
65         vals[i] = val;
66         return;
67     }
68     if (N == capacity)
69         resize(capacity * 2);
70     for (int j = N; j > i; --j) {
71         keys[j] = keys[j - 1];
72         vals[j] = vals[j - 1];
73     }
74     keys[i] = key;
75     vals[i] = val;
76     N++;
77 }
78
79 public void delete(key key) {
80     if (isEmpty()) return;
81     int i = rank(key);
82     if (i < N && keys[i].compareTo(key) == 0) {
83         for (int j = i; j < N - 1; j++) {
84             keys[j] = keys[j + 1];
85             vals[j] = vals[j + 1];
86         }
87         keys[N - 1] = null;
88         vals[N - 1] = null;
89         N--;
90     }
91 }
92
93 public Key min() {
94     if (isEmpty()) return null;
95     return keys[0];
96 }
97
98 public Key max() {
99     if (isEmpty()) return null;
100    return keys[N - 1];
101 }
102
103 public void deleteMin() {
104     delete(min());
105 }
106
107 public void deleteMax() {
108     delete(max());

```

```
109     }
110
111     //取大于等于指定键的键
112     public Key ceiling(Key key) {
113         return keys[rank(key)];
114     }
115
116     //取小于等于指定键的键
117     public Key floor(Key key) {
118         if (isEmpty()) return null;
119         int i = rank(key);
120         if (i < N) {
121             if (keys[i].compareTo(key) == 0)
122                 return keys[i];
123             return keys[i - 1];
124         }
125         return null;
126     }
127
128     public Key select(int k) {
129         if (k < 0 || k >= size()) return null;
130         return keys[k];
131     }
132
133     public int size(Key low, Key high) {
134         if (high.compareTo(low) < 0)
135             return 0;
136         else if (contains(high))
137             return rank(high) - rank(low) + 1;
138         else
139             return rank(high) - rank(low);
140     }
141
142     //返回指定键范围的键集合容器，该容器提供了一个iterator()方法
143     public Iterable<Key> keys(Key low, Key high) {
144         Queue<Key> q = new Queue<Key>();
145         for (int i = rank(low); i < rank(high); ++i)
146             q.enqueue(keys[i]);
147         if (contains(high))
148             q.enqueue(keys[rank(high)]);
149         return q;
150     }
151
152     public Iterable<Key> keys() {
153         return keys(min(), max());
154     }
155
156     public static void main(String[] args) {
157         BinarySearchST<String, Integer> binarySearchST =
158             new BinarySearchST<String, Integer>(10);
159
160         binarySearchST.put("a", 32);
161         binarySearchST.put("b", 5);
162         binarySearchST.put("e", 43);
163         binarySearchST.put("d", 64);
164         for (String str : binarySearchST.keys())
165             StdOut.println(str + " " + binarySearchST.get(str));
166         binarySearchST.delete("a");
167     }
168 }
```

```
167     StdOut.println(binarySearchST.size());
168     StdOut.println(binarySearchST.min());
169     StdOut.println(binarySearchST.max());
170     StdOut.println(binarySearchST.floor("c"));
171     StdOut.println(binarySearchST.ceiling("c"));
172 }
173 }
```

3.1.3 不同类型符号表对比

使用不同数据结构构造符号表（键-值对关联数组）的优缺点：

使用的数据结构	实现	插入时间复杂度	查找时间复杂度	优点	缺点
链表 (顺序查找)	SequentialSearchST	\$N\$	\$N\$	适用于小型问题	当问题规模变大时处理很慢
有序数组 (二分查找)	BinarySearchST	\$N\$	\$\log N\$	最优的查找效率和空间需求，能够进行有序性相关的操作	插入操作很慢
二叉查找树	BST	\$\log N\$~\$N\$	\$\log N\$~\$N\$	实现简单，能够进行有序性相关的操作	没有性能上的保证，链接需要额外的空间
平衡二叉树查找树	RedBlackBST	\$\log N\$	\$\log N\$	最优的查找和插入效率，能够进行有序性相关的操作	链接需要额外的空间
散列表	SeparateChain HashST LinearProbing HashST	\$1\$~\$\log N\$	\$1\$~\$\log N\$	能够快速地查找和插入常见类型的数据	需要计算每种类型数据的散列，无法进行有序性相关的操作，链接和空节点需要额外的空间

3.2 二叉查找树

容器插入操作 `put()` 时间复杂度: $\log N$~$N$$

容器查找操作 `get()` 时间复杂度: $\log N$~$N$$

```

1 import edu.princeton.cs.algs4.Queue;
2 import edu.princeton.cs.algs4.StdOut;
```

```
3
4 public class BST<Key extends Comparable<Key>, Value> {
5     private Node root;
6
7     //二叉树节点私有类定义
8     private class Node {
9         private Key key;
10        private Value val;
11        private Node left, right;
12        private int N;
13
14        public Node(Key key, Value val, int N) {
15            this.left = this.right = null;
16            this.key = key;
17            this.val = val;
18            this.N = N;
19        }
20    }
21
22     private int size(Node x) {
23         if (x == null) return 0;
24         return x.N;
25     }
26
27     public int size() {
28         return size(root);
29     }
30
31     public boolean contains(Key key) {
32         return get(key) != null;
33     }
34
35     public boolean isEmpty() {
36         return size() == 0;
37     }
38
39     private Value get(Node x, Key key) {
40         if (x == null) return null;
41
42         int cmp = key.compareTo(x.key);
43         if (cmp < 0)
44             return get(x.left, key);
45         else if (cmp > 0)
46             return get(x.right, key);
47         else return x.val;
48     }
49
50     //在二叉搜索树中查找指定键的元素
51     public Value get(Key key) {
52         return get(root, key);
53     }
54
55     private Node put(Node x, Key key, Value val) {
56         if (x == null) return new Node(key, val, 1);
57
58         int cmp = key.compareTo(x.key);
59         if (cmp < 0)
60             x.left = put(x.left, key, val);
```

```

61         else if (cmp > 0)
62             x.right = put(x.right, key, val);
63         else x.val = val;
64         //对于当前节点而言，更新N没什么用，但是对其父节点及其祖先节点是必要的
65         // x.N=size(x.left)+size(x.right)+1;
66         x.N++;
67     return x;
68 }
69
70 //向二叉搜索树元素插入操作
71 public void put(Key key, Value val) {
72     root = put(root, key, val);
73 }
74
75 private Node min(Node x) {
76     if (x.left == null) return x;
77     return min(x.left);
78 }
79
80 public Key min() {
81     return min(root).key;
82 }
83
84 private Node max(Node x) {
85     if (x.right == null) return x;
86     return max(x.right);
87 }
88
89 public Key max() {
90     return max(root).key;
91 }
92
93 //下取整
94 private Node floor(Node x, Key key) {
95     if (x == null) return null;
96
97     int cmp = key.compareTo(x.key);
98     if (cmp == 0) return x;
99     else if (cmp < 0) //若key小于当前节点的key，则继续到该节点的左子树去找
100        return floor(x.left, key);
101     Node t = floor(x.right, key);
102     if (t != null) return t;
103     else return x;
104 }
105
106 public Key floor(Key key) {
107     Node t = floor(root, key);
108     if (t == null) return null;
109     else return t.key;
110 }
111
112 //上取整
113 private Node ceiling(Node x, Key key) {
114     if (x == null) return null;
115
116     int cmp = key.compareTo(x.key);
117     if (cmp == 0) return x;
118     else if (cmp > 0)

```

```

119         return ceiling(x.right, key);
120     Node t = ceiling(x.left, key);
121     if (t != null) return t;
122     else return x;
123 }
124
125 public Key ceiling(Key key) {
126     Node t = ceiling(root, key);
127     if (t == null) return null;
128     return t.key;
129 }
130
131 //选取排位顺序的键
132 private Node select(Node x, int k) {
133     if (x == null) return null;
134
135     int t = size(x.left);
136     if (t > k) return select(x.left, k);
137     else if (t < k) return select(x.right, k - t - 1);
138     else return x;
139 }
140
141 public Key select(int k) {
142     Node t = select(root, k);
143     if (t == null) return null;
144     else return t.key;
145 }
146
147 //获取指定键的位置
148 private int rank(Node x, Key key) {
149     if (x == null) return 0;
150
151     int cmp = key.compareTo(x.key);
152     if (cmp < 0)
153         return rank(x.left, key);
154     else if (cmp > 0)
155         return size(x.left) + 1 + rank(x.right, key);
156     else return size(x.left);
157 }
158
159 public int rank(Key key) {
160     return rank(root, key);
161 }
162
163 private void keys(Node x, Queue<Key> queue, Key low, Key high) {
164     if (x == null) return;
165
166     int lcomp = low.compareTo(x.key);
167     int hcomp = high.compareTo(x.key);
168     if (lcomp < 0) keys(x.left, queue, low, high); //先将左子树中符合要求的
元素加入queue中
169     if (lcomp <= 0 && hcomp >= 0) queue.enqueue(x.key); //再将自己也加入到
queue中
170     if (hcomp > 0) keys(x.right, queue, low, high); //最后将右子树中符合要
求的元素加入到queue中
171 }
172
173 //返回存储指定范围键的可迭代容器

```

```

174     public Iterable<Key> keys(Key low, Key high) {
175         Queue<Key> queue = new Queue<Key>();
176         keys(root, queue, low, high);
177         return queue;
178     }
179
180     public Iterable<Key> keys() {
181         return keys(min(), max());
182     }
183
184     private Node deleteMin(Node x) {
185         if (x.left == null) return x.right;
186         x.left = deleteMin(x.left);
187         x.N = size(x.left) + size(x.right) + 1;
188         return x;
189     }
190
191     public void deleteMin() {
192         root = deleteMin(root);
193     }
194
195     private Node deleteMax(Node x) {
196         if (x.right == null) return x.left;
197         x.right = deleteMax(x.right);
198         x.N = size(x.left) + size(x.right) + 1;
199         return x;
200     }
201
202     public void deleteMax() {
203         root = deleteMax(root);
204     }
205
206     private Node delete(Node x, Key key) {
207         if (x == null) return null;
208
209         int cmp = key.compareTo(x.key);
210         if (cmp < 0)
211             x.left = delete(x.left, key);
212         else if (cmp > 0)
213             x.right = delete(x.right, key);
214         else {
215             //待删结点只有一个子树
216             if (x.right == null) return x.left;
217             else if (x.left == null) return x.right;
218             Node t = x;//保存待删结点
219             x = min(x.left);//获取待删结点右子树最小结点引用x
220             x.right = deleteMin(t.right);//将待删结点右子树最小结点删除，并将剩余
部分挂载到x的右边
221             x.left = t.left;//待删结点的左子树挂在新结点的左边
222         }
223         return x;
224     }
225
226     public void delete(Key key) {
227         root = delete(root, key);
228     }
229
230     private void show(Node x) {

```

```

231     if (x == null) return;
232     show(x.left);
233     StdOut.println(x.key);
234     show(x.right);
235   }
236
237   public void show() {
238     show(root);
239   }
240 }
```

C语言实现：

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAXLINE 64
6 #undef min
7 #undef max
8
9 struct Node {
10     char str[MAXLINE];
11     int val;
12     struct Node* left, * right;
13 };
14
15 struct BST {
16     struct Node* root;
17 };
18
19
20 struct Node* CreateNode(const char* buf, int value) {
21     struct Node* node;
22
23     if ((node = malloc(sizeof(struct Node))) == NULL)
24         return NULL;
25     strcpy(node->str, buf);
26     node->val = value;
27     node->left = NULL;
28     node->right = NULL;
29     return node;
30 }
31
32
33 void InitBST(struct BST* bst) {
34     bst->root = NULL;
35 }
36
37
38 void destroy(struct Node* h) {
39     if (h == NULL) return;
40
41     if (h->left) destroy(h->left);
42     if (h->right) destroy(h->right);
43     free(h);
44 }
```

```
45
46
47 void BSTDestroy(struct BST* bst) {
48     if (bst == NULL) return;
49     destroy(bst->root);
50     bst->root = NULL;
51 }
52
53
54 //插入操作
55 struct Node* insert(struct Node* h, const char* buf, int value) {
56     if (h == NULL)
57         return CreateNode(buf, value);
58
59     int cmp = strcmp(buf, h->str);
60     if (cmp < 0)
61         h->left = insert(h->left, buf, value);
62     else if (cmp > 0)
63         h->right = insert(h->right, buf, value);
64     else
65         h->val = value;
66     return h;
67 }
68
69
70 void BSTInsert(struct BST* bst, const char* buf, int value) {
71     bst->root = insert(bst->root, buf, value);
72 }
73
74
75 //查找操作
76 int get(const struct Node* h, const char* buf) {
77     int cmp;
78
79     if (h == NULL) return -1;
80     if ((cmp = strcmp(buf, h->str)) < 0)
81         return get(h->left, buf);
82     else if (cmp > 0)
83         return get(h->right, buf);
84     else return h->val;
85 }
86
87
88 int BSTGet(const struct BST* bst, const char* buf) {
89     if (bst == NULL) return -1;
90     return get(bst->root, buf);
91 }
92
93
94 //最小值
95 struct Node* min(struct Node* h) {
96     if (h == NULL) return NULL;
97     if (h->left != NULL)
98         return min(h->left);
99     else return h;
100}
101
102
```

```
103 const char* BSTMin(const struct BST* bst) {
104     if (bst == NULL) return NULL;
105     return min(bst->root)->str;
106 }
107
108
109 //最大值
110 struct Node* max(struct Node* h) {
111     if (h == NULL) return NULL;
112     if (h->right != NULL)
113         return max(h->right);
114     else return h;
115 }
116
117
118 const char* BSTMax(const struct BST* bst) {
119     if (bst == NULL) return NULL;
120     return max(bst->root)->str;
121 }
122
123
124 //删除最小值
125 struct Node* deleteMin(struct Node* h) {
126     struct Node* t;
127
128     if (h == NULL) return NULL;
129     if (h->left != NULL) {
130         h->left = deleteMin(h->left);
131         return h;
132     }
133     else {
134         t = h->right;
135         free(h);
136         return t;
137     }
138 }
139
140
141 void BSTDeleteMin(struct BST* bst) {
142     if (bst == NULL) return;
143     bst->root = deleteMin(bst->root);
144 }
145
146
147 //删除最大值
148 struct Node* deleteMax(struct Node* h) {
149     struct Node* t;
150
151     if (h == NULL) return NULL;
152     if (h->right != NULL) {
153         h->right = deleteMax(h->right);
154         return h;
155     }
156     else {
157         t = h->left;
158         free(h);
159         return t;
160     }
}
```

```

161 }
162
163
164 void BSTDeleteMax(struct BST* bst) {
165     if (bst == NULL) return;
166     bst->root = deleteMax(bst->root);
167 }
168
169
170 //任意删除
171 struct Node* delete(struct Node* h, const char* buf) {
172     //t表示替代删除结点的结点指针,right用来表示右子树的根节点（可能跟原来不一样）
173     struct Node* t/*, * right*/;
174     int cmp;
175
176     if (h == NULL) return NULL;
177     if ((cmp = strcmp(buf, h->str)) < 0) {
178         h->left = delete(h->left, buf);
179         return h;
180     }
181     else if (cmp > 0) {
182         h->left = delete(h->right, buf);
183         return h;
184     }
185     else {
186         //若当前结点的右子树为空
187         if (h->right == NULL) {
188             t = h->left;
189             free(h);
190             return t;
191         }
192         //若当前结点的右子树非空
193         else {
194             strcpy(h->str, min(h->right)->str);
195             h->val = min(h->right)->val;
196             h->right = deleteMin(h->right);
197             return h;
198         }
199     }
200 }
201
202
203 void BSTDelete(struct BST* bst, const char* buf) {
204     if (bst == NULL) return;
205     bst->root = delete(bst->root, buf);
206 }
207
208
209 //打印操作（中序）
210 void print(const struct Node* h) {
211     if (h == NULL) return;
212
213     if (h->left != NULL)
214         print(h->left);
215     printf("str: %s, value: %d\n", h->str, h->val);
216     if (h->right != NULL)
217         print(h->right);
218 }
```

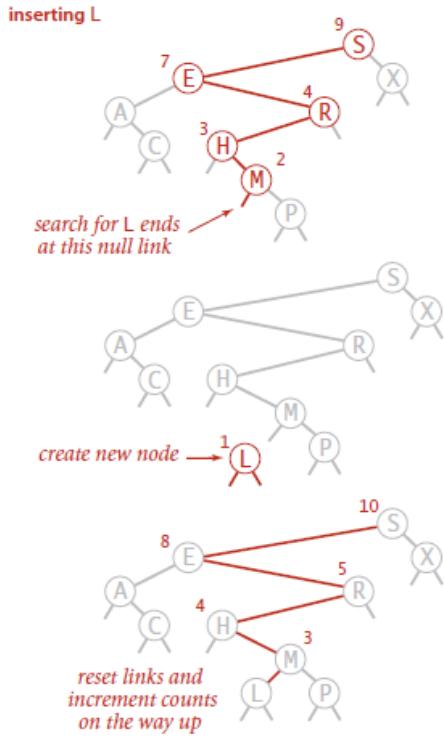
```
219  
220  
221 void BSTPrint(const struct BST* bst) {  
222     print(bst->root);  
223     putchar('\n');  
224 }
```

3.2.1 插入操作

在递归前沿着树向下走寻找合适的位置，然后以递归创建了新结点，递归调用返回途中沿着树向上爬时更新结点的计数值

```
1  private Node put(Node x, Key key, value val) {  
2      if (x == null) return new Node(key, val, 1);  
3  
4      int cmp = key.compareTo(x.key);  
5      if (cmp < 0)  
6          x.left = put(x.left, key, val);  
7      else if (cmp > 0)  
8          x.right = put(x.right, key, val);  
9      else x.val = val;  
10     //对于当前节点而言，更新N没什么用，但是对其父节点及其祖先节点是必要的  
11     //    x.N=size(x.left)+size(x.right)+1;  
12     x.N++;  
13     return x;  
14 }  
15  
16 //向二叉搜索树元素插入操作  
17 public void put(Key key, value val) {  
18     root = put(root, key, val);  
19 }
```

图示：



Insertion into a BST

注意：这里采用的是使用return返回更新好结点的指针（引用）的方式递归向上传递给父结点，通知其最新的左/右子结点的指针（引用），从而来完成子结点的更新（删除或者删除）。例如如下形式就是典型的使用形式：

```
struct Node *deleteMin(struct Node*h);
h->left=deleteMin(h->left);
```

通过递归返回的时候更新父结点的指向子结点的指针（引用）会使用很方便。但若使用二级指针的方式来更新的话，会显得比较麻烦。可能需要如下的形式：

```
voide deleteMin(struct Node**h);
```

3.2.2 查找操作

类似于插入操作

```

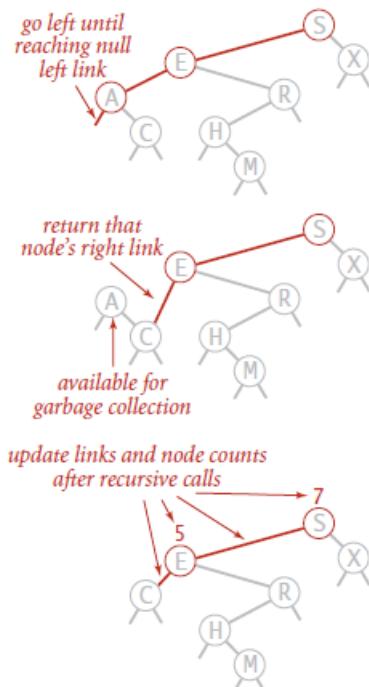
1 private value get(Node x, Key key) {
2     if (x == null) return null;
3
4     int cmp = key.compareTo(x.key);
5     if (cmp < 0)
6         return get(x.left, key);
7     else if (cmp > 0)
8         return get(x.right, key);
9     else return x.val;
10 }
11
12 //在二叉搜索树中查找指定键的元素
13 public value get(Key key) {
14     return get(root, key);
15 }
```

3.2.3 删除操作

删除最小结点的关键在于：将待删除结点的右结点接到待删除结点的父结点的左边

```
1  private Node deleteMin(Node x) {
2      if (x.left == null) return x.right;
3      x.left = deleteMin(x.left);
4      x.N = size(x.left) + size(x.right) + 1;
5      return x;
6  }
7
8  public void deleteMin() {
9      root = deleteMin(root);
10 }
```

图示：



Deleting the minimum in a BST

删除任意结点的关键在于：区分只有一个或者无子树的结点（可以看作是像`deleteMin()`一样的操作）和左右子树同时存在的结点。无子树或者只有一个子树的结点只要将左子树（若存在）或者右子树接到待删结点的父结点的左/右边。而左右子树同时存在的结点，需要在删除时暂时记录待删结点的引用，然后取出待删结点右子树中的最小结点用其来替代待删结点（需要将其执行`deleteMin()`操作），然后将调整后的左右子树挂在该替代节点的左右两边。

```
1  private Node delete(Node x, Key key) {
2      if (x == null) return null;
3
4      int cmp = key.compareTo(x.key);
5      if (cmp < 0)
6          x.left = delete(x.left, key);
7      else if (cmp > 0)
8          x.right = delete(x.right, key);
9      else {
```

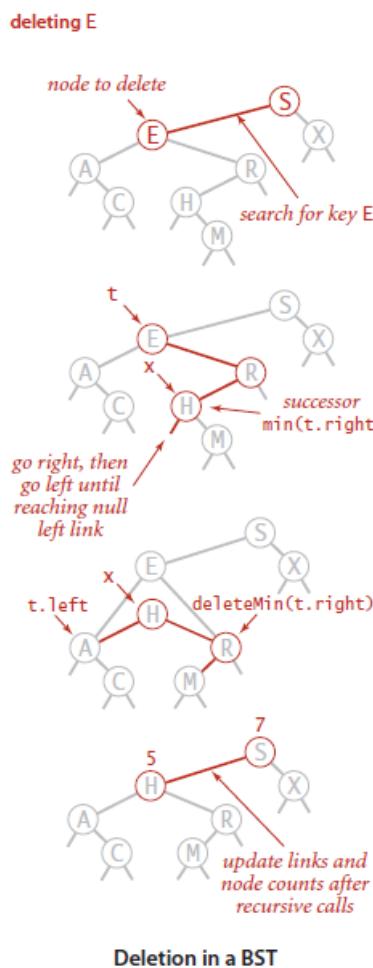
```

10 //待删结点只有一个子树
11 if (x.right == null) return x.left;
12 else if (x.left == null) return x.right;
13 //保存待删结点
14 Node t = x;
15 //获取待删结点右子树最小结点引用x
16 x = min(x.left);
17 //将待删结点右子树最小结点删除，并将剩余部分挂载到x的右边
18 x.right = deleteMin(t.right);
19 //待删结点的左子树挂在新结点的左边
20 x.left = t.left;
21 }
22 return x;
23 }

25 public void delete(Key key) {
26     root = delete(root, key);
27 }

```

图示：



Deletion in a BST

3.2.4 范围(遍历)操作

这里的范围操作关键就是使用中序遍历，将二叉搜索树中的结点按照从小到大的顺序加入到队列之中

```

1 private void keys(Node x, Queue<Key> queue, Key low, Key high) {
2     if (x == null) return;

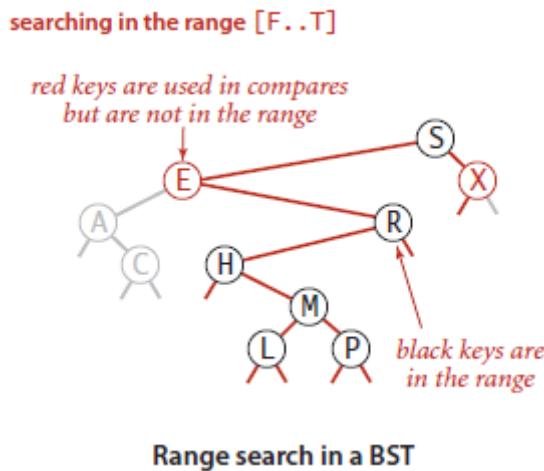
```

```

3     int lcomp = low.compareTo(x.key);
4     int hcomp = high.compareTo(x.key);
5     //先将左子树中符合要求的元素加入queue中
6     if (lcomp < 0) keys(x.left, queue, low, high);
7     //再将自己也加入到queue中
8     if (lcomp <= 0 && hcomp >= 0) queue.enqueue(x.key);
9     //最后将右子树中符合要求的元素加入到queue中
10    if (hcomp > 0) keys(x.right, queue, low, high);
11
12 }
13
14 //返回存储指定范围键的可迭代容器
15 public Iterable<Key> keys(Key low, Key high) {
16     Queue<Key> queue = new Queue<Key>();
17     keys(root, queue, low, high);
18     return queue;
19 }
20
21 public Iterable<Key> keys() {
22     return keys(min(), max());
23 }

```

图示：



3.2.5 上下取整操作

以下取整为例，其关键点在于：若在结点遍历过程中遇到一个比自己小的结点，就先暂时记录它然后在它的右子树中继续查找（试图找到比这个节点更合适的结点）。若找不到就仍然使用这个暂存的点进行返回，否则使用找到的合适点进行返回。

```

1 //下取整
2 private Node floor(Node x, Key key) {
3     if (x == null) return null;
4
5     int cmp = key.compareTo(x.key);
6     if (cmp == 0) return x;
7     else if (cmp < 0) //若key小于当前节点的key，则继续到该节点的左子树去找
8         return floor(x.left, key);
9     Node t = floor(x.right, key);
10    if (t != null) return t;
11    else return x;

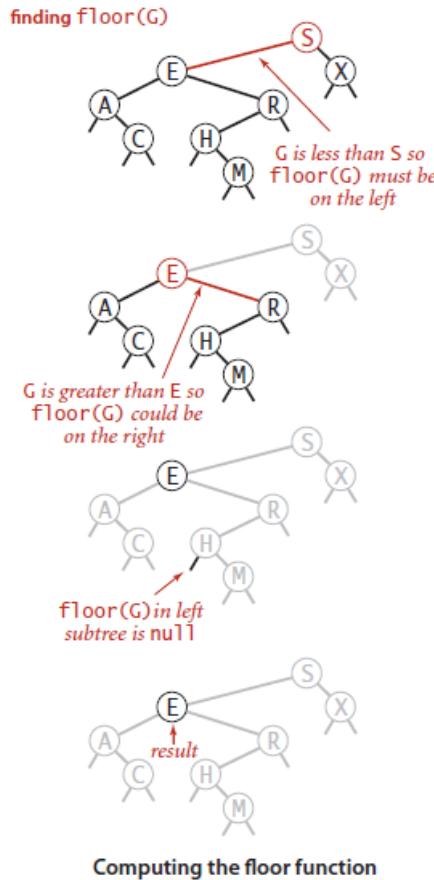
```

```

12     }
13
14     public Key floor(Key key) {
15         Node t = floor(root, key);
16         if (t == null) return null;
17         else return t.key;
18     }

```

图示：



3.2.6 排位选择操作

选取指定位顺序的键select()方法的关键在于：根据每一个结点中的N成员与欲取结点的位置进行比较，若欲取结点位置大于当前节点的N则到该节点的右结点去寻找（不过欲取结点位置要减去左边的结点数量）；若小于则在左边递归寻找；若相等则返回当前结点。

```

1 //选取排位顺序的键
2 private Node select(Node x, int k) {
3     if (x == null) return null;
4
5     int t = size(x.left);
6     if (t > k) return select(x.left, k);
7     else if (t < k) return select(x.right, k - t - 1);
8     else return x;
9 }
10
11 public Key select(int k) {
12     Node t = select(root, k);
13     if (t == null) return null;

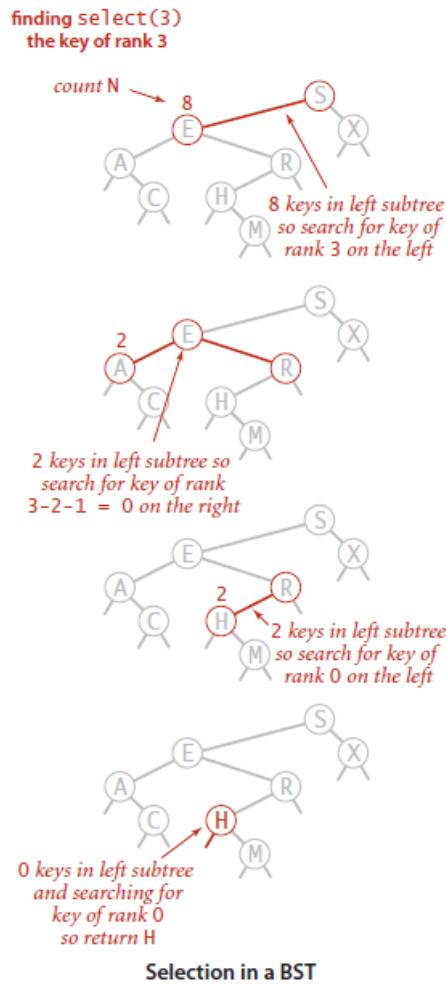
```

```

14         else return t.key;
15     }

```

图示：



而返回指定键位置的rank()方法，则很容易用size()方法递归计算出来

```

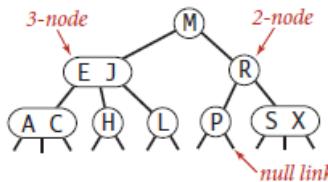
1 //获取指定键的位置
2 private int rank(Node x, Key key) {
3     if (x == null) return 0;
4
5     int cmp = key.compareTo(x.key);
6     if (cmp < 0)
7         return rank(x.left, key);
8     else if (cmp > 0)
9         return size(x.left) + 1 + rank(x.right, key);
10    else return size(x.left);
11 }
12
13 public int rank(Key key) {
14     return rank(root, key);
15 }

```

3.3 平衡查找树

3.3.1 2-3树

2-3树指的是由2-结点和3-结点共同构成的二叉树，其中2-结点具有两个指向孩子结点的链接（左孩子比它小，右孩子比它大），而3-结点具有三个指向孩子结点的链接（左孩子比它小，中间孩子键值介于3-结点中两个键之间，右孩子比它大）。一棵完美平衡的2-3查找树中的所有空链接null到根结点的距离总是相同的，且查找/插入操作总是能够在 $\log N$ 时间复杂度内完成。



Anatomy of a 2-3 search tree

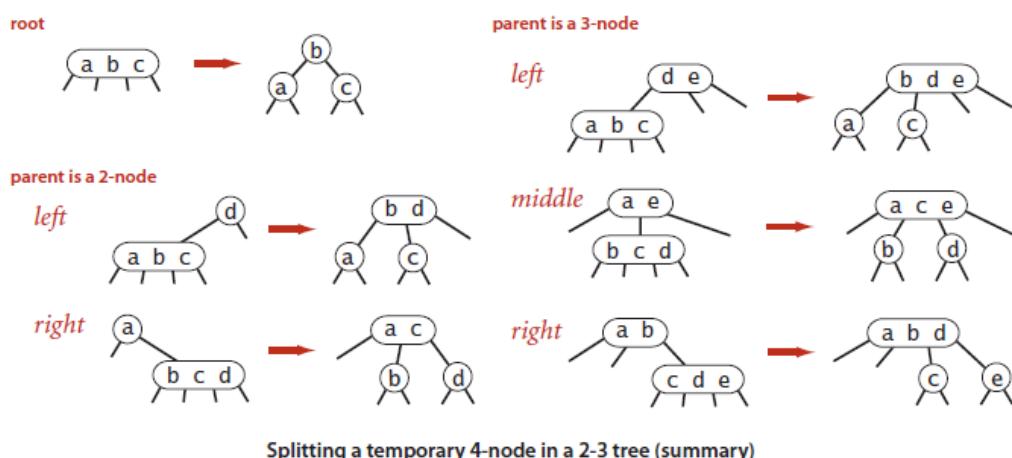
2-3树的插入操作可以总体分成如下两种情况：

1. 向2-结点进行插入：

此时的处理很简单，2-结点直接变成3-结点即可。

2. 向3-结点进行插入：

则操作时会临时产生一个临时的4-结点，该4-结点然后就会分解将中键（中间结点）提出给父结点（此时等效于向其父结点又进行了一次插入操作）。①若父结点原来是2-结点，则其结果就如同1) 的结果相同（父结点变成了一个3-结点），此时插入操作就到此为止；②若父结点原来是3-结点，则父结点也同样的会临时变成一个4-结点，此时该父结点又一次提出一个中间结点给它的父结点...若此递归下去，直到其遇到一个为2-结点的父结点（一种比较特殊的情况就是若该中间结点向上插入的过程中遇到了根结点，使得根结点变成了一个临时4-结点，此时该临时4-结点会直接分解成3个2结点，使树增高1层）。



3.3.2 红黑树

红黑树(左倾)的本质就是通过普通二叉搜索树来实现完美平衡2-3树，而通过这种方式实现的2-3树可以保证我们的查找和插入操作都维持在 $\log N$ 级别

```
1 import edu.princeton.cs.algs4.BlockFilter;
2 import edu.princeton.cs.algs4.Queue;
3 import edu.princeton.cs.algs4.Stdout;
```

```
4
5  public class RedBlackBST<Key extends Comparable<Key>, Value> {
6      private static final boolean RED = true;
7      private static final boolean BLACK = false;
8
9      private Node root;
10
11     private class Node {
12         Key key;
13         Value val;
14         Node left, right;
15         boolean color;
16         int size;
17
18         public Node(Key key, Value val, boolean color, int size) {
19             this.key = key;
20             this.val = val;
21             this.color = color;
22             this.size = size;
23         }
24     }
25
26     public RedBlackBST() {
27     }
28
29     private boolean isRed(Node x) {
30         if (x == null) return false;
31         return x.color == RED;
32     }
33
34     private int size(Node x) {
35         if (x == null)
36             return 0;
37         return x.size;
38     }
39
40     //左旋
41     private Node rotateLeft(Node h) {
42         Node x = h.right;
43         h.right = x.left;
44         x.left = h;
45
46         x.color = h.color;
47         h.color = RED;
48         x.size = h.size;
49         h.size = size(h.left) + size(h.right) + 1;
50         return x;
51     }
52
53     //右旋
54     private Node rotateRight(Node h) {
55         Node x = h.left;
56         h.left = x.right;
57         x.right = h;
58
59         x.color = h.color;
60         h.color = RED;
61         x.size = h.size;
```

```

62     h.size = size(h.left) + size(h.right) + 1;
63     return x;
64 }
65
66 //翻转当前结点和左右孩子结点的颜色
67 private void flipColors(Node h) {
68     h.color = !h.color;
69     h.left.color = !h.left.color;
70     h.right.color = !h.right.color;
71 }
72
73 /* 从当前结点的右子结点中借一个结点给左子结点，使左子结点
74 变成一个3-结点；或者3者合并成为3-结点 */
75 private Node moveRedLeft(Node h) {
76     flipColors(h);
77     /* 若右子结点是一个3-结点，则提取一个结点给左子结点
78     使其成为3-结点 */
79     if (isRed(h.right.left)) {
80         h.right = rotateRight(h.right);
81         h = rotateLeft(h);
82         flipColors(h);
83     }
84     return h;
85 }
86
87 /* 从当前结点的左子结点中借一个结点给右子结点，使右子结点
88 变得有剩余结点使得删除一个不影响红黑树的平衡 */
89 private Node moveRedRight(Node h) {
90     flipColors(h);
91     /* 若左子结点是一个3-结点，则提取一个结点给右子结点
92     使其成为3-结点 */
93     if (isRed(h.left.left)) {
94         h = rotateRight(h);
95         flipColors(h);
96     }
97     return h;
98 }
99
100 //在删除后做局部平衡处理
101 private Node balance(Node h) {
102     if (isRed(h.right))
103         h = rotateLeft(h);
104     if (isRed(h.left) && isRed(h.left.left))
105         h = rotateRight(h);
106     if (isRed(h.left) && isRed(h.right))
107         flipColors(h);
108     h.size = size(h.left) + size(h.right) + 1;
109     return h;
110 }
111
112 private Node put(Node h, Key key, Value val) {
113     if (h == null)
114         return new Node(key, val, RED, 1);
115
116     int cmp = key.compareTo(h.key);
117     if (cmp < 0)
118         h.left = put(h.left, key, val);
119     else if (cmp > 0)

```

```

120         h.right = put(h.right, key, val);
121     else h.val = val;
122
123     /* 红黑树比普通二叉查找树多就多在如下部分: */
124     if (!isRed(h.left) && isRed(h.right))
125         h = rotateLeft(h);
126     if (isRed(h.left) && isRed(h.left.left))
127         h = rotateRight(h);
128     if (isRed(h.left) && isRed(h.right))
129         flipColors(h);
130     h.size = size(h.left) + size(h.right) + 1;
131     return h;
132 }
133
134 private Value get(Node h, Key key) {
135     while (h != null) {
136         int cmp = key.compareTo(h.key);
137         if (cmp < 0)
138             h = h.left;
139         else if (cmp > 0)
140             h = h.right;
141         else return h.val;
142     }
143     return null;
144 }
145
146 private Node deleteMin(Node h) {
147     if (h.left == null)
148         return null;
149
150     /* 左子结点不是3-结点的情况下需要进行moveRedLeft
        局部调整作业，使得左子结点变成一个3结点 */
151     if (!isRed(h.left) && !isRed(h.left.left))
152         h = moveRedLeft(h);
153     h.left = deleteMin(h.left);
154     return balance(h);
155 }
156
157
158 private Node deleteMax(Node h) {
159     if (isRed(h.left))
160         h = rotateRight(h);
161     if (h.right == null)
162         return null;
163
164     /* 右子结点不是3-结点的情况下需要进行moveRedRight
        局部调整作业，使得右子结点变成一个3-结点 */
165     if (!isRed(h.right) && !isRed(h.right.left))
166         h = moveRedRight(h);
167     h.right = deleteMax(h.right);
168     return balance(h);
169 }
170
171
172 private Node delete(Node h, Key key) {
173     //欲删除结点在左子树中
174     if (key.compareTo(h.key) < 0) {
175         if (!isRed(h.left) && !isRed(h.left.left))
176             h = moveRedLeft(h);
177         h.left = delete(h.left, key);
178     }
179     else if (key.compareTo(h.key) > 0) {
180         if (!isRed(h.right) && !isRed(h.right.right))
181             h = moveRedRight(h);
182         h.right = delete(h.right, key);
183     }
184     else {
185         if (h.left == null)
186             return null;
187         Node min = h.left;
188         while (min.right != null)
189             min = min.right;
190         min.right = h.right;
191         if (isRed(h.left))
192             h = rotateRight(h);
193         h.left = null;
194         h.size = 1;
195         return h;
196     }
197     if (isRed(h))
198         h = balance(h);
199     return h;
200 }

```

```

178     }
179     /* 欲删除结点为当前结点或在右子树上。其中最需要注意的是要在前往右边
180      的删除路径上让途径的结点变成向右偏的3-结点（即红链接只存在于父结
181      点和其右子结点之间，而不是父结点和其左子结点之间），这样递归下去
182      方便从无子3-结点中删除一个结点*/
183     else {
184         if (isRed(h.left))
185             h = rotateRight(h);
186         if (key.compareTo(h.key) == 0 && h.right == null)
187             return null;
188         if (!isRed(h.right) && !isRed(h.right.left))
189             h = moveRedRight(h);
190         if (key.compareTo(h.key) == 0) {
191             h.val = get(h.right, min(h.right).key);
192             h.key = min(h.right).key;
193             h.right = deleteMin(h.right);
194         } else h.right = delete(h.right, key);
195     }
196     return balance(h);
197 }
198
199 private void keys(Node h, Queue<Key> queue, Key low, Key high) {
200     if (h == null) return;
201
202     int lcmp = low.compareTo(h.key);
203     int hcmp = high.compareTo(h.key);
204     if (lcmp < 0)
205         keys(h.left, queue, low, high);
206     if (lcmp <= 0 && hcmp >= 0)
207         queue.enqueue(h.key);
208     if (hcmp > 0)
209         keys(h.right, queue, low, high);
210 }
211
212 private Node min(Node h) {
213     if (h.left == null) return h;
214     return min(h.left);
215 }
216
217 private Node max(Node h) {
218     if (h.right == null) return h;
219     return max(h.right);
220 }
221
222 private int rank(Node h, Key key) {
223     if (h == null) return 0;
224
225     int cmp = key.compareTo(h.key);
226     if (cmp < 0)
227         return rank(h.left, key);
228     else if (cmp > 0)
229         return rank(h.right, key) + size(h.left) + 1;
230     else return size(h.left);
231 }
232
233 private Node select(Node h, int k) {
234     if (h == null) return null;
235

```

```

236         int t = size(h.left);
237         if (t > k)
238             return select(h.left, k);
239         else if (t < k)
240             return select(h.right, k - t - 1);
241         else return h;
242     }
243
244     private Node floor(Node h, Key key) {
245         if (h == null) return null;
246
247         int cmp = key.compareTo(h.key);
248         if (cmp < 0)
249             return floor(h.left, key);
250         Node t = floor(h.right, key);
251         if (t != null) return t;
252         return h;
253     }
254
255     private Node ceiling(Node h, Key key) {
256         if (h == null) return null;
257
258         int cmp = key.compareTo(h.key);
259         if (cmp > 0)
260             return ceiling(h.right, key);
261         Node t = ceiling(h.left, key);
262         if (t != null) return t;
263         return h;
264     }
265
266     public boolean isEmpty() {
267         return root == null;
268     }
269
270     public int size() {
271         return size(root);
272     }
273
274     public int size(Key low, Key high) {
275         if (low.compareTo(high) >= 0)
276             return 0;
277         else if (!contains(high))
278             return rank(high) - rank(low);
279         else
280             return rank(high) - rank(low) + 1;
281     }
282
283     public boolean contains(Key key) {
284         return get(key) != null;
285     }
286
287     public int rank(Key key) {
288         if (!contains(key))
289             throw new RuntimeException("No this key");
290         return rank(root, key);
291     }
292
293     public Key select(int k) {

```

```
294     Node t = select(root, k);
295     if (t == null) return null;
296     return t.key;
297 }
298
299 public Key floor(Key key) {
300     Node t = floor(root, key);
301     if (t == null) return null;
302     return t.key;
303 }
304
305 public Key ceiling(Key key) {
306     Node t = ceiling(root, key);
307     if (t == null) return null;
308     return t.key;
309 }
310
311 public Value get(Key key) {
312     if (root == null)
313         return null;
314     return get(root, key);
315 }
316
317 public void put(Key key, Value val) {
318     root = put(root, key, val);
319     root.color = BLACK;
320 }
321
322 public void deleteMin() {
323     if (!isRed(root.left) && !isRed(root.right))
324         root.color = RED;
325     root = deleteMin(root);
326     if (isEmpty()) root.color = BLACK;
327 }
328
329 public void deleteMax() {
330     if (!isRed(root.left) && !isRed(root.right))
331         root.color = RED;
332     root = deleteMax(root);
333     if (isEmpty()) root.color = BLACK;
334 }
335
336 public void delete(Key key) {
337     if (!isRed(root.left) && !isRed(root.right))
338         root.color = RED;
339     root = delete(root, key);
340     if (!isEmpty()) root.color = BLACK;
341 }
342
343 public Iterable<Key> keys(Key low, Key high) {
344     Queue<Key> queue = new Queue<Key>();
345     keys(root, queue, low, high);
346     return queue;
347 }
348
349 public Iterable<Key> keys() {
350     return keys(min(), max());
351 }
```

```

352
353     public Key min() {
354         if (root == null) return null;
355         return min(root).key;
356     }
357
358     public Key max() {
359         if (root == null) return null;
360         return max(root).key;
361     }
362
363     public static void main(String[] args) {
364         RedBlackBST<String, Integer> redBlackBST =
365             new RedBlackBST<String, Integer>();
366
367         redBlackBST.put("a", 22);
368         redBlackBST.put("b", 32);
369         redBlackBST.put("c", 5);
370         redBlackBST.put("d", 24);
371         redBlackBST.put("h", 2);
372         for (String str : redBlackBST.keys())
373             StdOut.print(str + " ");
374         StdOut.println();
375
376         StdOut.println("ceiling of k: " + redBlackBST.ceiling("k"));
377         StdOut.println("floor of k: " + redBlackBST.floor("k"));
378
379         redBlackBST.delete("d");
380         redBlackBST.deleteMin();
381         redBlackBST.deleteMax();
382         for (String str : redBlackBST.keys())
383             StdOut.print(str + " ");
384         StdOut.println();
385     }
386 }

```

C语言实现：

```

1 /**
2  * 左倾红黑树实现
3 */
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7
8 #define NODESTRLEN 64
9 #define RED 1
10#define BLACK 0
11
12#define MAX(l,r) ((l)>(r)?(l):(r))
13
14 struct Node {
15     char str[NODESTRLEN];
16     int val;
17     int color;
18     struct Node* left, * right;
19 };

```

```
20
21 struct RBT {
22     struct Node* root;
23 };
24
25
26 //结点创建
27 struct Node* NodeCreate(const char* buf, int value) {
28     struct Node* node;
29     if ((node = malloc(sizeof(struct Node))) == NULL)
30         return NULL;
31     strncpy(node->str, buf, NODESTRLEN - 1);
32     node->str[NODESTRLEN - 1] = '\0';
33     node->val = value;
34     node->color = RED;
35     node->left = NULL;
36     node->right = NULL;
37     return node;
38 }
39
40
41 //红黑树初始化
42 void RBTInit(struct RBT* rbt) {
43     rbt->root = NULL;
44 }
45
46
47 void NodeDestroy(struct Node* h) {
48     if (h == NULL) return;
49
50     if (h->left) NodeDestroy(h->left);
51     if (h->right) NodeDestroy(h->right);
52     free(h);
53 }
54
55
56 //红黑树销毁
57 void RBTDestroy(struct RBT* rbt) {
58     if (rbt == NULL) return;
59     NodeDestroy(rbt->root);
60     rbt->root = NULL;
61 }
62
63
64 int isEmpty(const struct RBT* rbt) {
65     return (rbt == NULL || rbt->root == NULL) ? 1 : 0;
66 }
67
68
69 //判断当前结点是否为红结点
70 int isRed(const struct Node* node) {
71     return node == NULL ? 0 : node->color ? 1 : 0;
72 }
73
74
75 int NodeSize(const struct Node* h) {
76     if (h == NULL) return 0;
77     return NodeSize(h->left) + NodeSize(h->right) + 1;
```

```
78 }
79
80
81 //结点总数
82 int RBTSIZE(const struct RBT* rbt) {
83     if (rbt == NULL) return 0;
84     return NodeSize(rbt->root);
85 }
86
87
88 int height(const struct Node* node) {
89     if (node == NULL) return 0;
90     int hs = MAX(height(node->left), height(node->right));
91     return hs + ((node->color == BLACK) ? 1 : 0);
92 }
93
94
95 //树高
96 int RBTHIGHT(const struct RBT* rbt) {
97     return rbt == NULL ? 0 : height(rbt->root);
98 }
99
100
101 //颜色翻转
102 void flipColors(struct Node* h) {
103     h->color = !h->color;
104     h->left->color = !h->left->color;
105     h->right->color = !h->right->color;
106 }
107
108
109 //左旋
110 struct Node* rotateLeft(struct Node* h) {
111     struct Node* x = h->right;
112
113     h->right = x->left;
114     x->left = h;
115     x->color = h->color;
116     h->color = RED;
117     return x;
118 }
119
120
121 //右旋
122 struct Node* rotateRight(struct Node* h) {
123     struct Node* x = h->left;
124
125     h->left = x->right;
126     x->right = h;
127     x->color = h->color;
128     h->color = RED;
129     return x;
130 }
131
132
133 /**
134 * 从当前结点的右子树中提出个结点使左子结点变成非2-结点
135 */
```

```

136 struct Node* removeLeft(struct Node* h) {
137     flipColors(h);
138     if (isRed(h->right->left)) {
139         h->right = rotateRight(h->right);
140         h = rotateLeft(h);
141         flipColors(h);
142     }
143     return h;
144 }
145
146
147 /**
148 * 从当前结点的左子树中提出个结点使右子结点变成非2-结点
149 */
150 struct Node* removeRight(struct Node* h) {
151     flipColors(h);
152     if (isRed(h->left->left)) {
153         h = rotateRight(h);
154         flipColors(h);
155     }
156     return h;
157 }
158
159
160 struct Node* NodeMin(struct Node* node) {
161     if (node == NULL) return NULL;
162     return node->left ? NodeMin(node->left) : node;
163 }
164
165
166 //返回最小键
167 const char* RBTMin(const struct RBT* rbt) {
168     return isEmpty(rbt) ? NULL : NodeMin(rbt->root)->str;
169 }
170
171
172 struct Node* NodeMax(struct Node* node) {
173     if (node == NULL) return NULL;
174     return node->right ? NodeMax(node->right) : node;
175 }
176
177
178 //返回最大键
179 const char* RBTMax(const struct RBT* rbt) {
180     return isEmpty(rbt) ? NULL : NodeMax(rbt->root)->str;
181 }
182
183
184 struct Node* NodePut(struct Node* h, const char* buf, int value) {
185     int cmp;
186
187     if (h == NULL)
188         return NodeCreate(buf, value);
189     if ((cmp = strcasecmp(buf, h->str)) < 0) {
190         h->left = NodePut(h->left, buf, value);
191     }
192     else if (cmp > 0) {
193         h->right = NodePut(h->right, buf, value);

```

```

194     }
195     else h->val = value;
196
197     if (!isRed(h->left) && isRed(h->right))
198         h = rotateLeft(h);
199     if (isRed(h->left) && isRed(h->left->left))
200         h = rotateRight(h);
201     if (isRed(h->left) && isRed(h->right))
202         flipColors(h);
203     return h;
204 }
205
206
207 //插入操作
208 void RBTPut(struct RBT* rbt, const char* buf, int value) {
209     if (rbt == NULL) {
210         fprintf(stderr, "rbt is null\n");
211         return;
212     }
213     rbt->root = NodePut(rbt->root, buf, value);
214     rbt->root->color = BLACK;
215 }
216
217
218 int get(const struct Node* h, const char* buf) {
219     int cmp;
220
221     if (h == NULL) return -1;
222     if ((cmp = strcmp(buf, h->str)) < 0)
223         return get(h->left, buf);
224     else if (cmp > 0)
225         return get(h->right, buf);
226     return h->val;
227 }
228
229
230 //查找
231 int RBTGet(const struct RBT* rbt, const char* buf) {
232     return rbt == NULL ? -1 : get(rbt->root, buf);
233 }
234
235
236 //检测是否存在该键
237 int contains(const struct RBT* rbt, const char* buf) {
238     return RBTGet(rbt, buf) != -1;
239 }
240
241
242 //在删除结点后调用该函数来重新对结点进行调整，使其保持平衡
243 struct Node* balance(struct Node* h) {
244     if (isRed(h->right))
245         h = rotateLeft(h);
246     if (isRed(h->left) && isRed(h->left->left))
247         h = rotateRight(h);
248     if (isRed(h->left) && isRed(h->right))
249         flipColors(h);
250     return h;
251 }

```

```

252
253
254 struct Node* deleteMin(struct Node* h) {
255     if (h == NULL) return NULL;
256     if (h->left == NULL) {
257         free(h);
258         return NULL;
259     }
260     if (!isRed(h->left) && !isRed(h->left->left))
261         h = removeLeft(h);
262     h->left = deleteMin(h->left);
263     return balance(h);
264 }
265
266
267 //删除最小结点
268 void RBTDeleteMin(struct RBT* rbt) {
269     if (rbt == NULL) return;
270     if (!isRed(rbt->root->left) && !isRed(rbt->root->right))
271         rbt->root->color = RED;
272     rbt->root = deleteMin(rbt->root);
273     if (!isEmpty(rbt)) rbt->root->color = BLACK;
274 }
275
276
277 struct Node* deleteMax(struct Node* h) {
278     if (h == NULL) return NULL;
279     if (isRed(h->left))
280         h = rotateRight(h);
281     if (h->right == NULL) {
282         free(h);
283         return NULL;
284     }
285     if (!isRed(h->right) && !isRed(h->right->left))
286         h = removeRight(h);
287     h->right = deleteMax(h->right);
288     return balance(h);
289 }
290
291
292 //删除最大结点
293 void RBTDeleteMax(struct RBT* rbt) {
294     if (rbt == NULL) return;
295     if (!isRed(rbt->root->left) && !isRed(rbt->root->right))
296         rbt->root->color = RED;
297     rbt->root = deleteMax(rbt->root);
298     if (!isEmpty(rbt)) rbt->root->color = BLACK;
299 }
300
301
302 struct Node* delete(struct Node* h, const char* buf) {
303     if (h == NULL) return NULL;
304
305     if (strcmp(buf, h->str) < 0) {
306         if (!isRed(h->left) && !isRed(h->left->left))
307             h = removeLeft(h);
308         h->left = delete(h->left, buf);
309     }

```

```

310     else {
311         if (isRed(h->left))
312             h = rotateRight(h);
313         /* 算法4书中使用的key.compareTo(h.key)其实没有什么用,
314            其比较顶多就是确保就是我们需要找的结点,但实际上即使
315            不适用这个我们也可以确保。因为我们要删的结点肯定会通过
316            上面执行的局部调整变成一个3-结点中的一个,这样才可以
317            使删除操作变得方便,而且它一定是既没有左子结点也肯定
318            没有右子结点!☺*/
319         if (/*strcmp(buf,h->str)==0 && */h->right == NULL) {
320             free(h);
321             return NULL;
322         }
323         if (!isRed(h->right) && !isRed(h->right->left))
324             h = removeRight(h);
325
326         if (strcmp(buf, h->str) == 0) {
327             struct Node* t = NodeMin(h->right);
328             strcpy(h->str, t->str);
329             h->val = t->val;
330             h->right = deleteMin(h->right);
331         }
332         else h->right = delete(h->right, buf);
333     }
334     return balance(h);
335 }
336
337
338 //任意删除
339 void RBTDelete(struct RBT* rbt, const char* buf) {
340     if (isEmpty(rbt)) {//检查红黑树是否为空?
341         fprintf(stderr, "rbt is null or is empty\n");
342         return;
343     }
344     if (!contains(rbt, buf)) {//检测红黑树中是否包含该键
345         fprintf(stderr, "key: %s is not contain!\n", buf);
346         return;
347     }
348
349     if (!isRed(rbt->root->left) && !isRed(rbt->root->right))
350         rbt->root->color = RED;
351     rbt->root = delete(rbt->root, buf);
352     if (!isEmpty(rbt)) rbt->root->color = BLACK;
353 }
354
355
356 static void printNode(const struct Node* node) {
357     if (node == NULL) return;
358     if (node->left)
359         printNode(node->left);
360     printf("str: %s, val: %d, color: %d\n", node->str, node->val,
361           isRed(node));
362     if (node->right)
363         printNode(node->right);
364     }
365
366 //打印

```

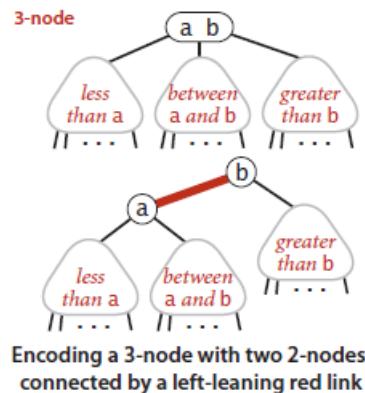
```

367 static void RBTPrint(const struct RBT* rbt) {
368     if (rbt == NULL) {
369         fprintf(stderr, "rbt is null\n");
370         return;
371     }
372     printNode(rbt->root);
373 }

```

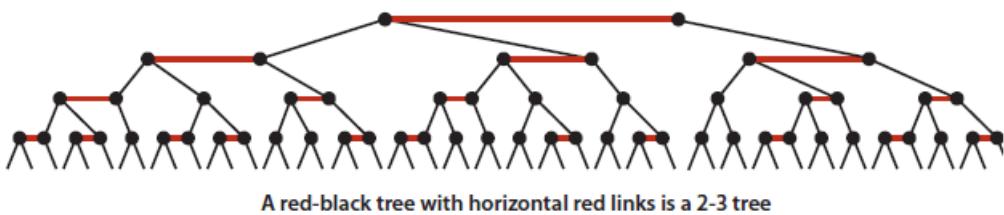
3.3.2.1 二叉搜索树改造

为了使得上述普通的二叉搜索树能够支持红黑树的实现，我们需要在 `private class Node` 中加入一个名为color的布尔类型变量，当`color==RED(true)`表示该结点是一个红结点，这意味着该结点和其父结点组成了一个逻辑上的3-结点；否则为黑结点，这意味着该结点单独组成一个2-结点。



除此之外我们还对红黑树做出如下的规定（这样我们的红黑树就如同下图一样）：

- 红链接均为左链接（纯粹是为了处理的情况少点，方便）
- 没有任何一个结点同时与两条红链接同时相连
- 该树是完美黑色平衡的，即任意空链接到它的黑链接数量都是相同的
- 根结点永远都是黑结点，空结点也永远都是黑结点



这样做好处在于代码复用性高，只要稍微对二叉搜索树进行改进，基本上只要对结点类定义、`put()` 和 `delete()` 等方法进行改造就可以。

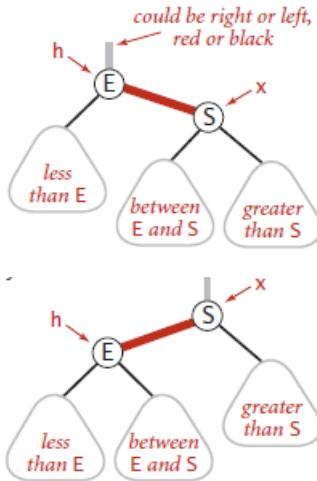
3.3.2.1 左右旋操作

为了实现结点的插入，我们必须在之前了解下结点的左旋和右旋操作。**左旋**指的是以当前结点为中心进行逆时针方向旋转，该操作本质就是将该子树的根结点设置为原根结点的右子结点。比如它可以使一个红链接为右链接的3-结点变成一个红链接为左链接的3-结点。

```

1  private Node rotateLeft(Node h) {
2      Node x = h.right;
3      h.right = x.left;
4      x.left = h;
5
6      x.color = h.color;
7      h.color = RED;
8      x.size = h.size;
9      h.size = size(h.left) + size(h.right) + 1;
10     return x;
11 }

```

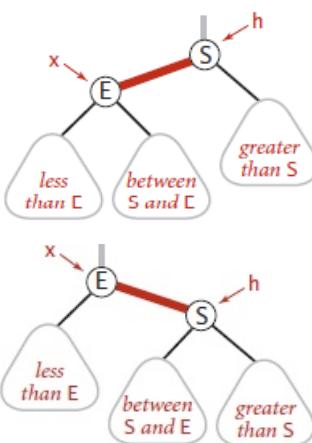


而**右旋**与之相反，它以当前节点为中心进行顺时针方向旋转，该操作本质上就是将该指针的根结点设置为原根结点的左子结点。比如它可以将一个红链接为左链接的3-结点变成一个红链接为右链接的3-结点。

```

1  private Node rotateRight(Node h) {
2      Node x = h.left;
3      h.left = x.right;
4      x.right = h;
5
6      x.color = h.color;
7      h.color = RED;
8      x.size = h.size;
9      h.size = size(h.left) + size(h.right) + 1;
10     return x;
11 }

```

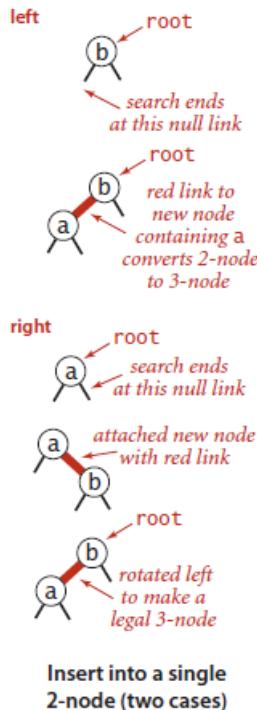


3.3.2.2 插入操作

在此之前，我们需要做如下规定：新插入的结点总是红结点

1. 向2-结点进行插入

若向一个2-结点进行插入，则2-结点会直接变成3-结点。对于一个向左插入的红结点而言，新的结点对符合我们的要求，因此我们不需要做什么；但是对于一个向右插入的红结点而言，由于其插入的结点的红链接是右链接不符合我们“红链接必须是右链接”的前提假设，因此我们需要对其根结点进行左旋操作。



2. 向3-结点进行插入

若向一个3-结点进行插入，则按照2-3树的插入规则该结点会临时变成一个4-结点，然后取出其中间结点向上给原3-结点的父结点（本质就是将中间结点插入到其父结点中。而插入操作又等同于新来的结点的color==RED，所以在处理3-结点插入的时候特别需要面对子结点是红色的多种情况）。向3-结点插入新结点有如下3种情况：

- 向3-结点的右边插入

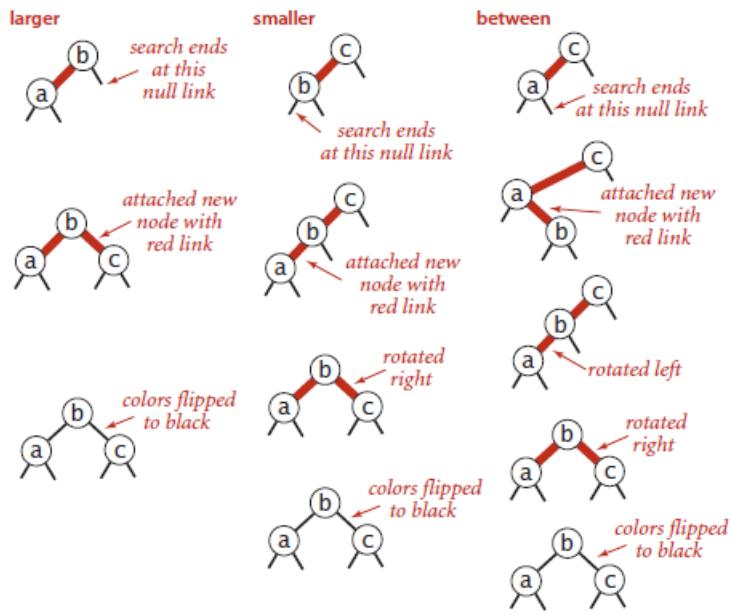
此时的处理就是直接变成临时4-结点，然后向上传递中间结点，而左右结点被分解变成了两个单独的2-结点。在程序中操作就只是简单的翻转颜色 `flipColors()` 而已

- 向3-结点的左边插入

此时的处理是让其直接变成临时4-结点，然后做右旋操作，最后向上传递中间结点，然后左右结点被分解成两个独立的2-结点。在程序中操作为左旋 `rotateLeft()` +翻转颜色 `flipColors()`

- 向3-结点的中间插入

此时的处理是让其直接变成临时的4-结点，然后做左旋操作，然后再右旋，最后向上传递中间结点，左右结点被分解成两个独立的2-结点。在程序中操作为左旋 `rotateLeft()` +右旋 `rotateRight()` +翻转颜色 `flipColors()`

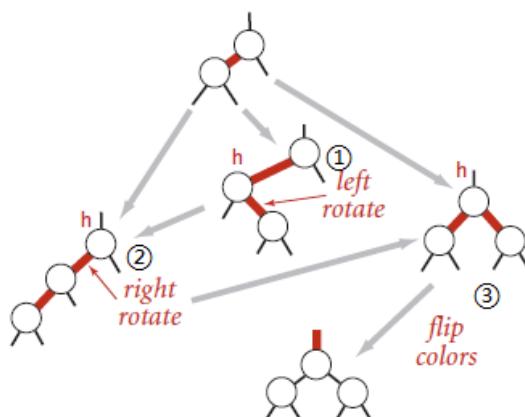


Insert into a single 3-node (three cases)

3. 中间结点的向上（插入）传递

因为我们上面提到过，特别是对于向3-结点插入的时候必然存在一个向父结点插入中间结点的过程，因此简简单单找到合适的插入点然后做相应的旋转操作什么的显然是不够的。我们必须使用递归的方式，通过递归不断向下查找到合适的位置插入，然后再递归返回的途中使用相应的处理手段（旋转、翻转颜色使红结点变黑，黑结点变红）不断从下向上调整好所有路径上的结点，这样即使途中有向上传递中间结点的情况这样也能够完美的处理。

由于只有向3-结点插入新结点才会发生向上传递的情况，所以我们可以以向3-结点插入的三种情况（左边插入、中间插入、右边插入）为模板做出 `if () { /*...*/ }` 的调整，下图是3-结点插入后状态转移图（其中囊括了所有的处理情况），其中的①②③是我们的处理顺序：



Passing a red link up a red-black BST

```

1  private Node put(Node h, Key key, value val) {
2      if (h == null)
3          return new Node(key, val, RED, 1);
4
5      int cmp = key.compareTo(h.key);
6      if (cmp < 0)
7          h.left = put(h.left, key, val);
8      else if (cmp > 0)
9          h.right = put(h.right, key, val);
10     else h.val = val;
11

```

```

12     /* 红黑树比普通二叉查找树多就多在如下部分: */
13     if (!isRed(h.left) && isRed(h.right))
14         h = rotateLeft(h);
15     if (isRed(h.left) && isRed(h.left.left))
16         h = rotateRight(h);
17     if (isRed(h.left) && isRed(h.right))
18         flipColors(h);
19     h.size = size(h.left) + size(h.right) + 1;
20     return h;
21 }

```

3.3.2.2 删除最小/大值操作

在2-3树删除最小/大值过程中，我们能够发现若最左边或最右边的结点是一个3-结点，那么我们以普通二叉搜索树的删除方式直接删除这个结点就行了，此时3-结点变成了2-结点，但丝毫不影响我们红黑树的完美平衡。但是当最左边或者最右边的结点为2-结点的时候就不一样了，此时若直接删除这个结点就会影响到红黑树的完美平衡性。因此删除最小/大值操作的关键在于：**沿着左链接或者右链接向下，保证不能直接删除一个2-结点，要确保当前结点总不是2-结点**（对于向左删除一定是向左偏的非2-结点，对于向右删除的一定向右偏的非2-结点，其中临时产生的4-结点一定是既向右又向左）。

1. 删除最小值

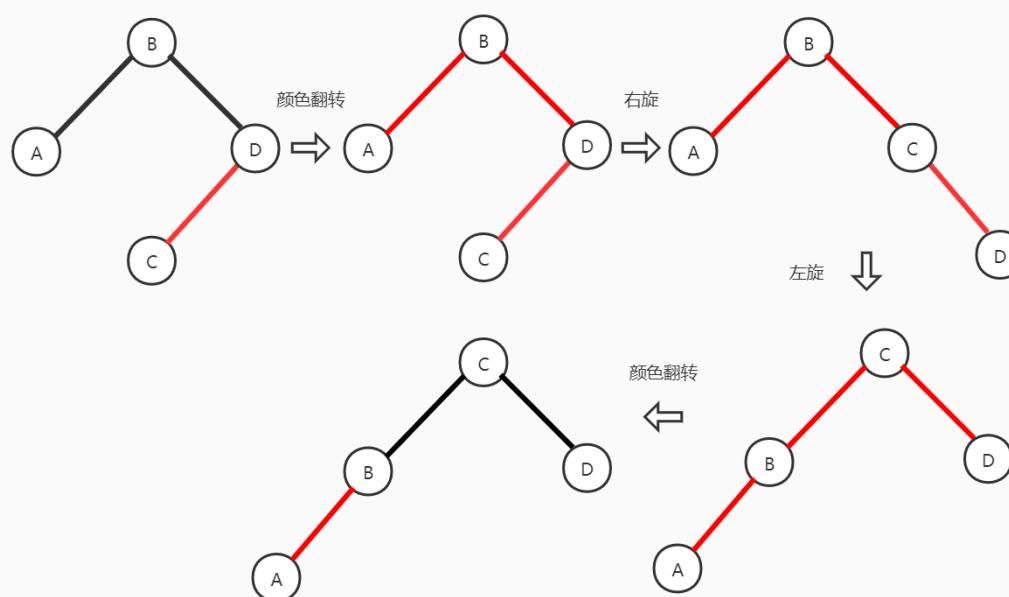
因此在沿着左链接向下的过程中，当前结点有如下的3种情况：

- 若当前结点的左子结点是3-结点

则此时我们并不需要特别的处理，因为它本身就一定是一个向左偏的3-结点

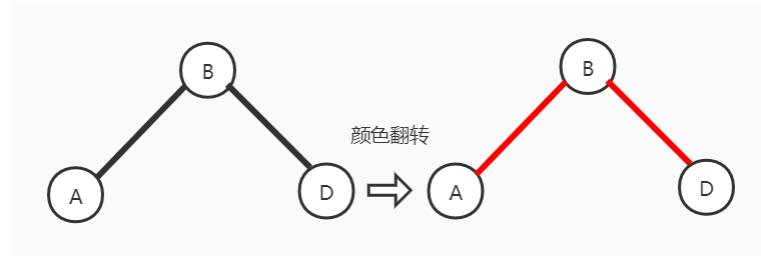
- 若当前节点的左子结点是2-结点，但右子结点不是2-结点

则此时左子结点需要到右子结点中“借”一个键移动到左子结点。因此在递归之前需要做出如下操作：①先进行颜色翻转处理（联想到插入时分解一个4-结点的时候也有一个颜色翻转的操作，这个和其下面的颜色翻转可以是其逆操作，不止一次是因为插入时本身也存在中间结点上提而多次调用的过程）；②然后对当前结点的右子结点进行右旋操作；③接着对当前结点做左旋操作，使得右子结点成为新的根结点（替代当前结点）；④最后再对根结点进行颜色翻转操作。



- 若当前节点的左子结点和其右子结点都是2-结点

则此时需要将左子结点、右子结点和父结点中的最小键（其实这是从2-3树角度考虑的，在二叉搜索树实现的红黑树中不需要这个考虑，直接称当前结点即可）合并成一个4-结点（这里也不用像书上那样过多的考虑2-3-4树那样）。因此在递归向下前需要做出如下的操作：直接进行颜色翻转使得当前结点变成4-结点。



当完成上述的递归左链接向下的过程后，`deleteMin()` 就可以被递归调用以对真正的欲删除的结点进行删除操作。需要注意的是：当完成删除操作之后，需要在递归返回的路径中依次对每一个结点执行再平衡操作，以保持我们对红黑树的预先规定。（当然其中很多细节需要不断查看源码进行体悟）

```

1 //翻转当前结点和左右孩子结点的颜色
2 private void flipColors(Node h) {
3     h.color = !h.color;
4     h.left.color = !h.left.color;
5     h.right.color = !h.right.color;
6 }
7
8 /* 从当前结点的右子结点中借一个结点给左子结点，使左子结点
9    变成一个3-结点；或者3者合并成为3-结点 */
10 private Node moveRedLeft(Node h) {
11     flipColors(h);
12     /* 若右子结点是一个3-结点，则提取一个结点给左子结点
13        使其成为3-结点 */
14     if (isRed(h.right.left)) {
15         h.right = rotateRight(h.right);
16         h = rotateLeft(h);
17         flipColors(h);
18     }
19     return h;
20 }
21
22 private Node deleteMin(Node h) {
23     if (h.left == null)
24         return null;
25
26     /* 左子结点不是3-结点的情况下需要进行moveRedLeft
27        局部调整作业，使得左子结点变成一个3结点 */
28     if (!isRed(h.left) && !isRed(h.left.left))
29         h = moveRedLeft(h);
30     h.left = deleteMin(h.left);
31     return balance(h);
32 }
33
34 public void deleteMin() {
35     if (!isRed(root.left) && !isRed(root.right))
36         root.color = RED;
37     root = deleteMin(root);
38     if (isEmpty()) root.color = BLACK;
39 }
```

2. 删除最大值

其操作类似于删除最小值，但可能有一些细节上的不同。

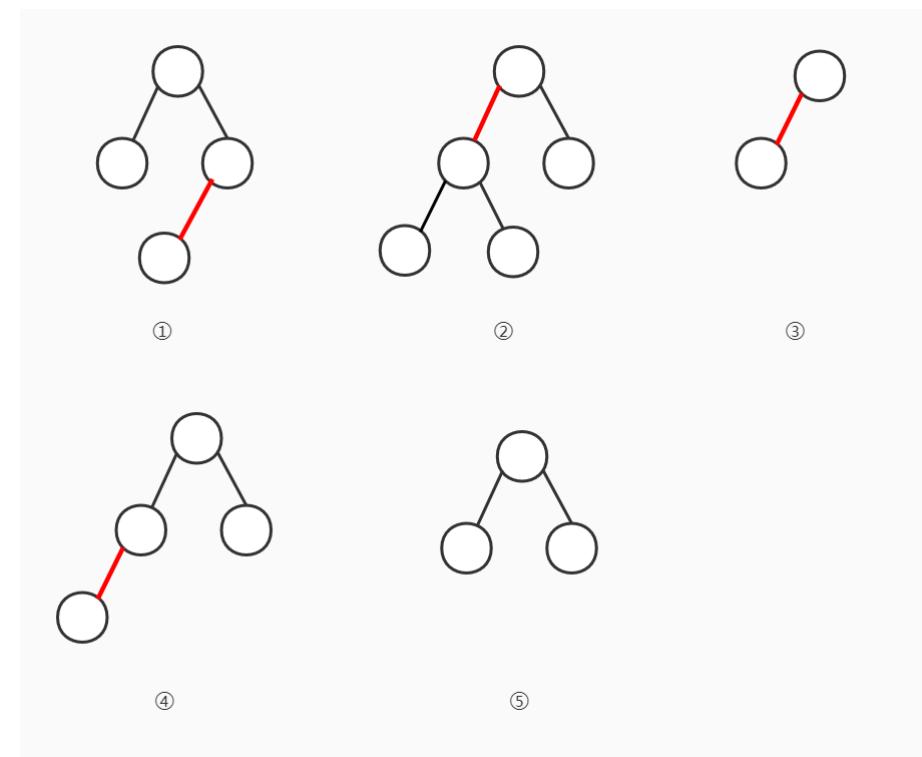
```
1 //翻转当前结点和左右孩子结点的颜色
2 private void flipColors(Node h) {
3     h.color = !h.color;
4     h.left.color = !h.left.color;
5     h.right.color = !h.right.color;
6 }
7
8 /* 从当前结点的左子结点中借一个结点给右子结点，使右子结点
9    变得有剩余结点使得删除一个不影响红黑树的平衡 */
10 private Node moveRedRight(Node h) {
11     flipColors(h);
12     /* 若左子结点是一个3-结点，则提取一个结点给右子结点
13        使其成为3-结点 */
14     if (isRed(h.left.left)) {
15         h = rotateRight(h);
16         flipColors(h);
17     }
18     return h;
19 }
20
21 //在删除后做局部平衡处理
22 private Node balance(Node h) {
23     if (isRed(h.right))
24         h = rotateLeft(h);
25     if (isRed(h.left) && isRed(h.left.left))
26         h = rotateRight(h);
27     if (isRed(h.left) && isRed(h.right))
28         flipColors(h);
29     h.size = size(h.left) + size(h.right) + 1;
30     return h;
31 }
32
33 private Node deleteMax(Node h) {
34     if (isRed(h.left))
35         h = rotateRight(h);
36     if (h.right == null)
37         return null;
38
39     /* 右子结点不是3-结点的情况下需要进行moveRedRight
40        局部调整作业，使得右子结点变成一个3-结点 */
41     if (!isRed(h.right) && !isRed(h.right.left))
42         h = moveRedRight(h);
43     h.right = deleteMax(h.right);
44     return balance(h);
45 }
46
47 public void deleteMax() {
48     if (!isRed(root.left) && !isRed(root.right))
49         root.color = RED;
50     root = deleteMax(root);
51     if (isEmpty()) root.color = BLACK;
52 }
```

3.3.2.3 删除操作

红黑树的任意删除操作其实本质上就是删除最小值操作和删除最大值操作的扩展版，它结合向左删除或者向右删除的特征。若我们想要删除的结点在当前结点的左子树上，则要确保当前结点一定是一个向左偏的非2-结点；若我们想要删除的结点在当前结点的右子树上（或欲删除结点在就是当前结点），则要确保当前结点一定是向右偏的非2-结点。我们会惊奇地发现：删除当前结点和删除右子树上的结点都被归为了一类处理。

即使在删除的过程中红黑树也仍然保持着完美平衡（因为我们删除前的操作都是通过局部微调的方式维持着平衡的状态，顶多就是出现某一个非2-结点的红链接方向向右而已，这都是为了要在3-结点身上删除一个键值对使其变成2-结点做准备）。所以我们有理由相信当即将删除那个指定结点的时候，它一定是一个向左偏的3-结点（向左删除）或者是一个向右偏的3-结点（向右删除）或者是一个临时4-结点或者本来就单单一个结点。

当我们需要向右删除或者删除当前结点时仍然是基本上如同处理一个删除最小值的情况：①当前结点的右子结点本身就是3-结点；②当前结点的右子结点为2-结点，但是左子结点为3-结点；③当前结点、左右子结点都是2-结点。所以处理它们的方法基本上相同，就是让当前结点变成向右偏的非2-结点。



还是通过代码进行理解更好：

```
1  private Node delete(Node h, Key key) {
2      //欲删除结点在左子树中
3      if (key.compareTo(h.key) < 0) {
4          if (!isRed(h.left) && !isRed(h.left.left))
5              h = moveRedLeft(h);
6          h.left = delete(h.left, key);
7      }
8      //欲删除结点在右子树或者当前结点就是
9      else {
10         if (isRed(h.left))
11             h = rotateRight(h);
12         /* 若当前结点确实是欲删除结点，若右边没有子结点，则由于红黑树在删除
13          的过程中仍然是平衡的，所以当前结点左边是绝对没有子结点，即使有
```

```

14            也被上一步的右旋操作处理掉了。所以这里直接删除返回null本质上与
15             上面的deleteMax()我个人觉得区别不大。*/
16     if (key.compareTo(h.key) == 0 && h.right == null)
17         return null;
18     if (!isRed(h.right) && !isRed(h.right.left))
19         h = moveRedRight(h);
20
21     if (key.compareTo(h.key) == 0) {
22         h.val = get(h.right, min(h.right).key);
23         h.key = min(h.right).key;
24         h.right = deleteMin(h.right);
25     } else h.right = delete(h.right, key);
26 }
27
28     return balance(h);
29 }
30
31     public void delete(Key key) {
32         if (!isRed(root.left) && !isRed(root.right))
33             root.color = RED;
34         root = delete(root, key);
35         if (!isEmpty()) root.color = BLACK;
36     }

```

3.4 散列表

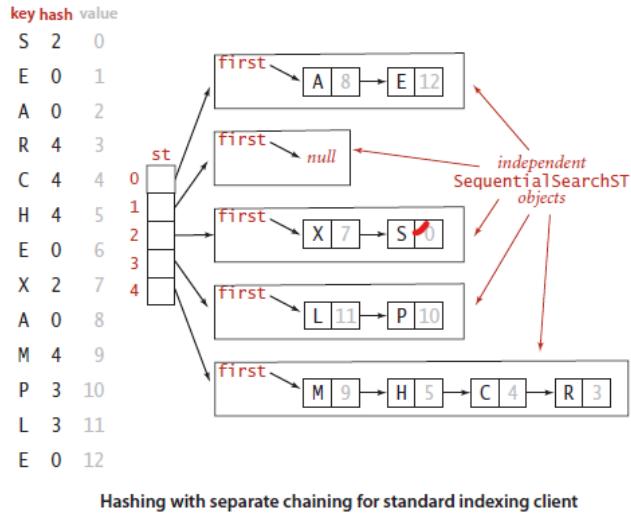
3.4.1 散列函数

散列的查找算法分为两步：

1. 使用散列函数（主要是除留余数法）将被查找的键转化为数组中的一个索引；
2. 然后使用一些方法（主要有拉链法和线性探测法）处理碰撞冲突问题

3.4.2 基于拉链法的散列表

基于拉链法的散列表使用一个大小为M（素数）、每一个元素指向一条链表的数组，将多个散列值相同的键-值对存放在同一个元素指向链表之中，从而避免碰撞冲突。



插入时间复杂度：\$N/M\$(在散列值均匀的前提下)\$\sim \log N\$

查找时间复杂度: N/M \$(在散列值均匀的前提下)~\$ $\log N$ \$

```
1 import edu.princeton.cs.algs4.Queue;
2 import edu.princeton.cs.algs4.SquentialSearchST;
3 import edu.princeton.cs.algs4.Stdout;
4
5 public class SeparateChainingHashST<Key, Value> {
6     private SquentialSearchST<Key, Value>[] st;
7     private int N;//键值对数
8     private int M;//散列表大小
9
10    //根据键key, 使用散列函数计算该键在散列表中的索引
11    private int hash(Key key) {
12        return (key.hashCode() & 0xffffffff) % M;
13    }
14
15    private void resize(int newM) {
16        SeparateChainingHashST<Key, Value> temp =
17            new SeparateChainingHashST<Key, Value>(newM);
18        for (int i = 0; i < M; i++)
19            for (Key key : st[i].keys())
20                temp.put(key, st[i].get(key));
21        this.M = temp.M;
22        this.N = temp.N;
23        this.st = temp.st;
24    }
25
26    public SeparateChainingHashST(int M) {
27        this.M = M;
28        this.N = 0;
29        st = (SquentialSearchST<Key, Value>[][]) new SquentialSearchST[M];
30        for (int i = 0; i < M; ++i)
31            st[i] = new SquentialSearchST<>();
32    }
33
34    public SeparateChainingHashST() {
35        this(997);
36    }
37
38    public void put(Key key, Value val) {
39        if (N >= M * 10) resize(2 * M);
40
41        st[hash(key)].put(key, val);
42        N++;
43    }
44
45    public Value get(Key key) {
46        return (Value) st[hash(key)].get(key);
47    }
48
49    public void delete(Key key) {
50        st[hash(key)].delete(key);
51        N--;
52
53        if (N <= 2 * M) resize(M / 2);
54    }
55}
```

```

56     public boolean isEmpty() {
57         return N == 0;
58     }
59
60     public int size() {
61         return N;
62     }
63
64     public boolean contains(Key key) {
65         return get(key) != null;
66     }
67
68     Iterable<Key> keys() {
69         Queue<Key> queue = new Queue<Key>();
70
71         for (int i = 0; i < M; ++i) {
72             if (!st[i].isEmpty()) {
73                 Iterable<Key> t = st[i].keys();
74                 for (Key key : t)
75                     queue.enqueue(key);
76             }
77         }
78         return queue;
79     }
80
81     public static void main(String[] args) {
82         SeparateChainingHashST<String, Integer> st = new
SeparateChainingHashST<String, Integer>();
83
84         st.put("a", 43);
85         st.put("b", 32);
86         st.put("s", 23);
87         st.put("k", 12);
88         st.put("p", 97);
89         StdOut.println("size: " + st.size());
90         StdOut.println("a: " + st.get("a"));
91         StdOut.println("k: " + st.get("k"));
92         for (String str : st.keys())
93             StdOut.println(str);
94     }
95 }
```

3.4.3 基于线性探测法的散列表

实现散列表的另一种方式使用大小为M的数组保存N个键-值对，其中M>N。这种方法依靠数组中的空位解决碰撞冲突。基于这种策略的所有方法统称为开放地址散列表。而开放地址散列表中最简单的一种方法就是线性探测法：当碰撞（指定索引上已经被别的键-值对占据），此时我们直接检测下一个位置（通过索引+1），若没有使用就存放在这个位置（可以重复）。

插入操作时间复杂度：常数阶 $\sim \log N$

查找操作时间复杂度：常数阶 $\sim \log N$

```

1 import edu.princeton.cs.algs4.Queue;
2 import edu.princeton.cs.algs4.StdOut;
```

```

3 import javax.sound.sampled.Line;
4
5
6 public class LinearProbingHashST<Key, Value> {
7     private int N;
8     private int M = 16;
9     private Key[] keys;
10    private Value[] vals;
11
12    private int hash(Key key) {
13        return (key.hashCode() & 0xffffffff) % M;
14    }
15
16    private void resize(int cap) {
17        LinearProbingHashST<Key, Value> t =
18            new LinearProbingHashST<Key, Value>(cap);
19        for (int i = 0; i < M; i++)
20            if (keys[i] != null)
21                t.put(keys[i], vals[i]);
22        keys = t.keys;
23        vals = t.vals;
24        M = t.M;
25    }
26
27    public LinearProbingHashST() {
28        keys = (Key[]) new Object[M];
29        vals = (Value[]) new Object[M];
30    }
31
32    public LinearProbingHashST(int m) {
33        keys = (Key[]) new Object[m];
34        vals = (Value[]) new Object[m];
35        M = m;
36    }
37
38    public void put(Key key, Value val) {
39        if (N >= M / 2) resize(M * 2);
40
41        int i = hash(key);
42        for (; keys[i] != null; i = (i + 1) % M)
43            if (keys[i].equals(key)) {
44                vals[i] = val;
45                return;
46            }
47        keys[i] = key;
48        vals[i] = val;
49        N++;
50    }
51
52    public Value get(Key key) {
53        for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
54            if (keys[i].equals(key))
55                return vals[i];
56        return null;
57    }
58
59    public void delete(Key key) {
60        if (!contains(key)) return;

```

```

61
62     //删除指定的键-值对
63     int i = hash(key);
64     while (!key.equals(keys[i]))
65         i = (i + 1) % M;
66     keys[i] = null;
67     vals[i] = null;
68
69     /* 将删除键-值对右边的键-值对重新插入，这样防止后续的键-值对
70      无法查找到（因为这些键-值对很有可能是通过探针的方式插入的） */
71     i = (i + 1) % M;
72     for (; keys[i] != null; i = (i + 1) % M) {
73         Key keyToRedo = keys[i];
74         Value valToRedo = vals[i];
75         keys[i] = null;
76         vals[i] = null;
77         N--;
78         put(keyToRedo, valToRedo);
79     }
80     N--;
81     if (N > 0 && N == M / 8)
82         resize(M / 2);
83 }
84
85 public Iterable<Key> keys() {
86     Queue<Key> queue = new Queue<Key>();
87     for (int i = 0; i < M; ++i)
88         if (keys[i] != null)
89             queue.enqueue(keys[i]);
90     return queue;
91 }
92
93 public boolean contains(Key key) {
94     return get(key) != null;
95 }
96
97 public int size() {
98     return N;
99 }
100
101 public boolean isEmpty() {
102     return N == 0;
103 }
104 }
```

4.图

4.1 无向图

使用邻接表组织的无向图类API: `public class Graph`

- `Graph(int v)`
- `Graph(In in)`
- `int V()`
- `int E()`

- `void addEdge(int v, int w)`
- `Iterable<Integer> adj(int v)`
- `String toString()`

邻接表组织无向图：

所需空间： $E + V$

添加一条边时间复杂度： 1

遍历指定顶点v所有边时间复杂度： $\text{degree}(v)$

检查某一条边的存在性： $\text{degree}(v)$

```

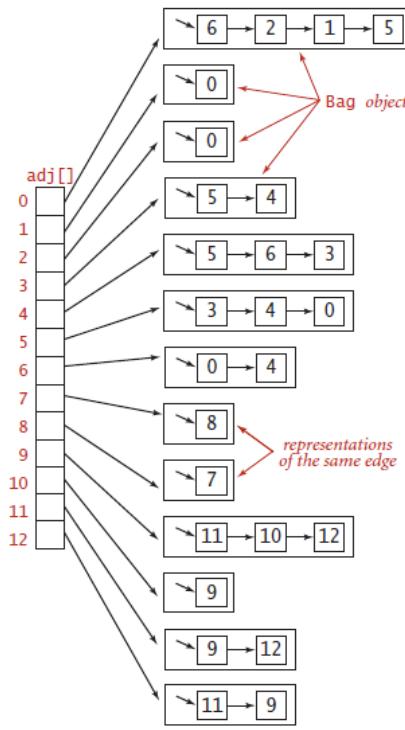
1 import edu.princeton.cs.algs4.Bag;
2 import edu.princeton.cs.algs4.In;
3 import edu.princeton.cs.algs4.Stdout;
4
5 public class Graph {
6     private final int V;
7     private int E;
8     private Bag<Integer>[] adj;
9
10    public Graph(int V) {
11        this.V = V;
12        this.E = 0;
13        adj = (Bag<Integer>[]) new Bag[V];
14        for (int v = 0; v < V; v++)
15            adj[v] = new Bag<Integer>();
16    }
17
18    public Graph(In in) {
19        this(in.readInt());
20        int E = in.readInt();
21        for (int i = 0; i < E; i++) {
22            int v = in.readInt();
23            int w = in.readInt();
24            addEdge(v, w);
25        }
26    }
27
28    //图中顶点数
29    public int V() {
30        return V;
31    }
32
33    //图中边数
34    public int E() {
35        return E;
36    }
37
38    //向图中添加一条边
39    public void addEdge(int v, int w) {
40        adj[v].add(w);
41        adj[w].add(v);
42        E++;
43    }
44

```

```

45     //返回顶点v的所有的邻接点
46     public Iterable<Integer> adj(int v) {
47         return adj[v];
48     }
49
50     public String toString() {
51         String s = V + " vertices, " + E + "edges\n";
52         for (int v = 0; v < V; v++) {
53             s += v + ": ";
54             for (int w : this.adj(v))
55                 s += w + " ";
56             s += "\n";
57         }
58         return s;
59     }
60
61     //计算指定顶点的入度
62     public static int degree(Graph G, int v) {
63         int degree = 0;
64         for (int w : G.adj(v)) degree++;
65         return degree;
66     }
67
68     //计算有向图中的最大顶点入度
69     public static int maxDegree(Graph G) {
70         int max = 0;
71         for (int v = 0; v < G.V(); v++)
72             if (degree(G, v) > max)
73                 max = degree(G, v);
74         return max;
75     }
76
77     //计算有向图中的平均入度
78     public static double avgDegree(Graph G) {
79         return 2.0 * G.E() / G.V();
80     }
81
82     //计算有向图中的自环个数
83     public static int numberofSelfLoops(Graph G) {
84         int count = 0;
85         for (int v = 0; v < G.V(); v++)
86             for (int w : G.adj(v))
87                 if (v == w) count++; //有一点连到自己的顶点存在
88         return count / 2;
89     }
90 }
```

邻接表如图所示：



4.1.1 深度优先搜索

深度优先搜索也可以称为深度优先遍历，因为它实际上是通过对图中的每一个与指定顶点连通的顶点进行遍历的方式获知图中的哪些顶点与之相连。其步骤很简单：

1. 将起点标记为已访问；
2. 然后递归方法问它的所有没有被标记过的邻居顶点。

深度优先搜索的原理是通过栈（递归的本质就是栈）来实现对图中顶点的完全遍历

```

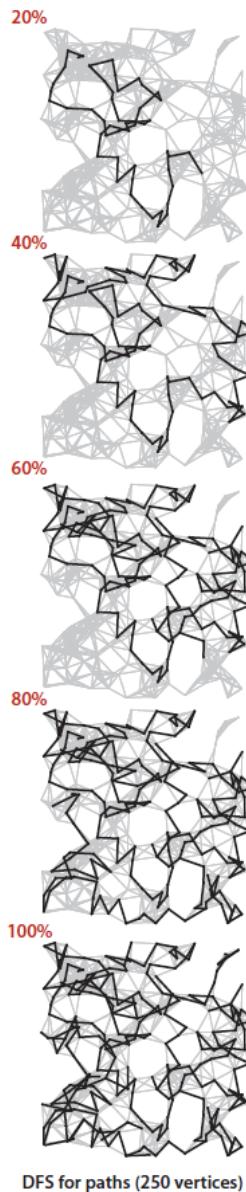
1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Stack;
3 import edu.princeton.cs.algs4.Stdout;
4
5 public class DepthFirstSearch {
6     private boolean[] marked;
7     private int count;
8
9     //实现递归实现的深度优先搜索
10    private void dfs(Graph G, int v) {
11        marked[v] = true;
12        count++;
13        for (int w : G.adj(v))
14            if (!marked[w]) dfs(G, w);
15    }
16
17    //使用栈实现的深度优先搜索
18    private void dfs1(Graph G, int v) {
19        Stack<Integer> stack = new Stack<Integer>();
20        marked[v] = true;
21        stack.push(v);
22        count++;
23
24        while (!stack.isEmpty()) {

```

```

25         int t = stack.pop();
26         for (int vertex : G.adj(t)) {
27             if (!marked[vertex]) {
28                 marked[vertex] = true;
29                 stack.push(vertex);
30                 count++;
31             }
32         }
33     }
34 }
35 }
36
37 public DepthFirstSearch(Graph G, int s) {
38     marked = new boolean[G.v()];
39     dfs1(G, s);
40 }
41
42 public boolean marked(int w) {
43     return marked[w];
44 }
45
46 public int count() {
47     return count;
48 }
49
50 public static void main(String[] args) {
51     Graph G = new Graph(new In(args[0]));
52     int s = Integer.parseInt(args[1]);
53     DepthFirstSearch search = new DepthFirstSearch(G, s);
54
55     StdOut.println("start vertex(" + s + ") connects: ");
56     for (int v = 0; v < G.v(); v++) {
57         if (search.marked(v))
58             StdOut.print(v + " ");
59     }
59     StdOut.println();
60     StdOut.println("connected count: " + search.count());
61 }
62 }
```

书中的图很好的展示了DFS所谓的“深度”是为何意，因为这孩子比较皮，老喜欢跑远的跑到更深的地方😊



4.1.1.1 使用DFS路径搜索

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Queue;
3 import edu.princeton.cs.algs4.Stack;
4 import edu.princeton.cs.algs4.Stdout;
5
6 public class DepthFirstPaths {
7     private boolean[] marked;
8     private int[] edgeTo; //存储递归过程中当前点的前一个点的索引
9     private final int s;
10
11     //使用栈实现的深度优先搜索路径方法
12     private void dfs(Graph G, int s) {
13         Stack<Integer> stack = new Stack<Integer>();
14         stack.push(s);
15         marked[s] = true;
16         edgeTo[s] = s;
17
18         while (!stack.isEmpty()) {
19             int t = stack.pop();

```

```

20         for (int ver : G.adj(t)) {
21             if (!marked[ver]) {
22                 stack.push(ver);
23                 marked[ver] = true;
24                 edgeTo[ver] = t;
25             }
26         }
27     }
28 }
29
30 //使用递归实现的深度优先搜索路径方法
31 private void dfs1(Graph G, int s) {
32     marked[s] = true;
33     for (int w : G.adj(s)) {
34         if (!marked[w]) {
35             edgeTo[w] = s;
36             dfs1(G, w);
37         }
38     }
39 }
40
41 public DepthFirstPaths(Graph G, int s) {
42     marked = new boolean[G.V()];
43     edgeTo = new int[G.V()];
44     this.s = s;
45     //    edgeTo[s]=s;
46     //    dfs1(G,s);
47     dfs(G, s);
48 }
49
50 public boolean hasPathTo(int v) {
51     return marked[v];
52 }
53
54 //返回从s->v的路径
55 public Iterable<Integer> pathTo(int v) {
56     if (!hasPathTo(v)) return null;
57
58     Stack<Integer> stack = new Stack<Integer>();
59     for (int x = v; x != s; x = edgeTo[x])
60         stack.push(x);
61     stack.push(s);
62     return stack;
63 }
64
65 public static void main(String[] args) {
66     Graph graph = new Graph(new In(args[0]));
67     int s = Integer.parseInt(args[1]);
68     DepthFirstPaths paths = new DepthFirstPaths(graph, s);
69
70     for (int v = 0; v < graph.V(); ++v) {
71         StdOut.print(s + " to " + v + ": ");
72         if (paths.hasPathTo(v))
73             for (int x : paths.pathTo(v))
74                 if (x == s) StdOut.print(x);
75                 else StdOut.print("-" + x);
76         StdOut.println();
77     }

```

```
78     }
79 }
```

4.1.1.2 使用DFS寻找连通分量

虽然理论上使用DFS实现图中连通分量的寻找与 $V+E$ 成正比，看上去和union-find算法应该相近，但是由于其在使用前必须对图进行构造，因此它的性能实际上没有比union-find并查集算法更好。

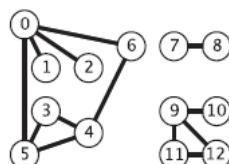
检测所需时间与 $V+E$ 成正比。

```
1 import edu.princeton.cs.algs4.Bag;
2 import edu.princeton.cs.algs4.In;
3 import edu.princeton.cs.algs4.Stack;
4 import edu.princeton.cs.algs4.Stdout;
5
6 public class CC {
7     private boolean[] marked;
8     private int[] id;
9     private int count;
10
11     //使用栈实现的DFS
12     private void dfs(Graph G, int v) {
13         Stack<Integer> stack = new Stack<Integer>();
14         stack.push(v);
15         marked[v] = true;
16         id[v] = count;
17
18         while (!stack.isEmpty()) {
19             int t = stack.pop();
20             for (int w : G.adj(t)) {
21                 if (!marked[w]) {
22                     stack.push(w);
23                     marked[w] = true;
24                     id[w] = count;
25                 }
26             }
27         }
28     }
29
30     //使用递归实现的DFS
31     private void dfs1(Graph graph, int v) {
32         marked[v] = true;
33         id[v] = count;
34         for (int w : graph.adj(v))
35             if (!marked[w])
36                 dfs(graph, w);
37     }
38
39     public CC(Graph G) {
40         marked = new boolean[G.V()];
41         id = new int[G.V()];
42         /* 将同一个连通分量下的顶点全部标记为同一个id,
43            若已经标记过就不再执行dfs, 而是遍历剩下的顶
44            点让它们归入到新的连通分量组之中 */
45         for (int s = 0; s < G.V(); s++)
46             if (!marked[s]) {
```

```

47         dfs(G, s);
48         count++;
49     }
50 }
51
52 public boolean connected(int v, int w) {
53     return id[v] == id[w];
54 }
55
56 public int id(int v) {
57     return id[v];
58 }
59
60 public int count() {
61     return count;
62 }
63
64 public static void main(String[] args) {
65     Graph graph = new Graph(new In(args[0]));
66     CC cc = new CC(graph);
67
68     int M = cc.count();
69     StdOut.println(graph.v() + " vertexes");
70     StdOut.println(M + " components");
71
72     Bag<Integer>[] components = (Bag<Integer>[]) new Bag[M];
73     for (int i = 0; i < M; ++i)
74         components[i] = new Bag<Integer>();
75     for (int i = 0; i < graph.v(); i++)
76         components[cc.id[i]].add(i);
77
78     for (int i = 0; i < M; i++) {
79         for (int v : components[i])
80             StdOut.print(v + " ");
81         StdOut.println();
82     }
83 }
84 }
```

图中有3个连通分量：



4.1.1.3 使用DFS检测环

我们可以使用DFS判断一个从一个起点到终点是否有环：假设在DFS遍历的过程中遇到了一个已经标记过的顶点，这就说明从起点到这个顶点至少有多条路径到终点，即有环。

```

1 public class Cycle {
2     private boolean[] marked;
3     private boolean hasCycle;
4
5     private void dfs(Graph G, int v, int u) {
6         marked[v] = true;
```

```

7     for (int w : G.adj(v))
8         if (!marked[w])
9             dfs(G, w, v);
10        else if (w != u)
11            hasCycle = true;
12    }
13
14    public Cycle(Graph G) {
15        marked = new boolean[G.V()];
16        for (int s = 0; s < G.V(); ++s)
17            if (!marked[s])
18                dfs(G, s, s);
19    }
20
21    public boolean hasCycle() {
22        return hasCycle;
23    }
24}

```

4.1.2 广度优先搜索

广度优先搜索BFS与深度优先搜索DFS相反，它是通过队列来实现对图中顶点的完全遍历，这种方法的好处在于它在顶点遍历的过程中实际上用到了贪心的思想，总是会先去找它（顶点）身边最近的那几个顶点进行遍历。其主要步骤如下：

1. 从队列中弹出下一个顶点并标记（其实也可以在入队的时候标记）
2. 然后将与v相邻的所有未被标记的顶点加入到队列之中

4.1.2.1 使用BFS路径搜索

此时搜寻到的路径必然是到达指定定点最短的。

```

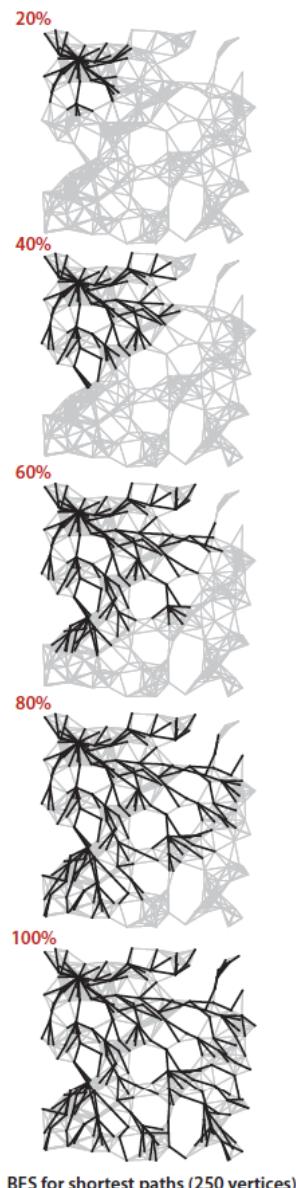
1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Queue;
3 import edu.princeton.cs.algs4.Stack;
4 import edu.princeton.cs.algs4.Stdout;
5
6 public class BreadthFirstPaths {
7     private boolean[] marked;
8     private int[] edgeTo;//存储递归过程中当前点的前一个点的索引
9     private final int s;
10
11    private void bfs(Graph G, int s) {
12        Queue<Integer> queue = new Queue<Integer>();
13        queue.enqueue(s);
14        edgeTo[s] = s;
15        marked[s] = true;
16
17        while (!queue.isEmpty()) {
18            int t = queue.dequeue();
19            for (int w : G.adj(t))
20                if (!marked[w]) {
21                    queue.enqueue(w);
22                    edgeTo[w] = t;
23                }
24            }
25        }
26    }
27
28    public Iterable<Integer> pathTo(int t) {
29        if (!marked[t])
30            return null;
31        Stack<Integer> path = new Stack<Integer>();
32        for (int x = t; x != s; x = edgeTo[x])
33            path.push(x);
34        path.push(s);
35        return path;
36    }
37
38    public int shortestPathLength() {
39        return edgeTo[s];
40    }
41
42    public static void main(String[] args) {
43        In in = new In(args[0]);
44        Graph G = new Graph(in);
45        BreadthFirstPaths bfs = new BreadthFirstPaths(G);
46        for (int v = 0; v < G.V(); v++)
47            if (bfs.hasPathTo(v))
48                StdOut.println("v " + v + " " + bfs.pathTo(v));
49    }
50}

```

```

23             marked[w] = true;
24         }
25     }
26 }
27
28 public BreadthFirstPaths(Graph G, int s) {
29     marked = new boolean[G.V()];
30     edgeTo = new int[G.V()];
31     this.s = s;
32     bfs(G, s);
33 }
34
35 public boolean hasPathTo(int v) {
36     return marked[v];
37 }
38
39 //返回从s->v的路径
40 public Iterable<Integer> pathTo(int v) {
41     if (!hasPathTo(v)) return null;
42
43     Stack<Integer> stack = new Stack<Integer>();
44     for (int x = v; x != s; x = edgeTo[x])
45         stack.push(x);
46     stack.push(s);
47     return stack;
48 }
49
50 public static void main(String[] args) {
51     Graph graph = new Graph(new In(args[0]));
52     int s = Integer.parseInt(args[1]);
53     DepthFirstPaths paths = new DepthFirstPaths(graph, s);
54
55     for (int v = 0; v < graph.V(); ++v) {
56         Stdout.print(s + " to " + v + ": ");
57         if (paths.hasPathTo(v))
58             for (int x : paths.pathTo(v))
59                 if (x == s) Stdout.print(x);
60                 else Stdout.print("-" + x);
61         Stdout.println();
62     }
63 }
64 }
```

书中的图很好的展示了BFS所谓的“广度”是为何意，因为这孩子比较乖有比较谨慎，起初总是在近的地方玩，之后熟悉了才到远的地方玩😊



BFS for shortest paths (250 vertices)

4.2 有向图

有向图API: `public class Digraph`

- `Digraph(int v)`
- `Digraph(In in)`
- `int v()`
- `int E()`
- `void addEdge(int v,int w)`
- `Iterable<Integer> adj(int v)`
- `Digraph reverse()`
- `String toString()`

```

1 import edu.princeton.cs.algs4.Bag;
2 import edu.princeton.cs.algs4.In;
3 import edu.princeton.cs.algs4.Stdout;
4
5 public class Digraph {
6     private final int V;
7     private int E;
```

```
8     private Bag<Integer>[] adj;
9
10    public Digraph(int v) {
11        this.v = v;
12        this.E = 0;
13        adj = (Bag<Integer>[]) new Bag[v];
14        for (int v = 0; v < V; v++)
15            adj[v] = new Bag<Integer>();
16    }
17
18    public Digraph(In in) {
19        this(in.readInt());
20        int E = in.readInt();
21        for (int i = 0; i < E; i++) {
22            int v = in.readInt();
23            int w = in.readInt();
24            addEdge(v, w);
25        }
26    }
27
28    public int V() {
29        return V;
30    }
31
32    public int E() {
33        return E;
34    }
35
36    public void addEdge(int v, int w) {
37        adj[v].add(w);
38        E++;
39    }
40
41    //返回指定点的指出点集合
42    public Iterable<Integer> adj(int v) {
43        return adj[v];
44    }
45
46    //返回一个反向图
47    public Digraph reverse() {
48        Digraph R = new Digraph(V);
49        for (int v = 0; v < V; v++)
50            for (int w : adj[v])
51                R.addEdge(w, v);
52        return R;
53    }
54
55    public String toString() {
56        String s = V + " vertices, " + E + " edges\n";
57        for (int v = 0; v < V; v++) {
58            s += v + ": ";
59            for (int w : this.adj(v))
60                s += w + " ";
61            s += "\n";
62        }
63        return s;
64    }
65}
```

```

66     public static void main(String[] args) {
67         Digraph digraph = new Digraph(new In(args[0]));
68         StdOut.println(args[0]);
69
70         StdOut.println(digraph);
71     }
72 }
```

4.2.1 可达性问题

4.2.1.1 使用DFS解决可达性问题

```

1 import edu.princeton.cs.algs4.Bag;
2 import edu.princeton.cs.algs4.In;
3 import edu.princeton.cs.algs4.Stack;
4 import edu.princeton.cs.algs4.StdOut;
5
6 public class DirectedDFS {
7     private boolean[] marked;
8
9     //使用递归实现的深度优先搜索方法
10    private void dfs(Digraph G, int v) {
11        marked[v] = true;
12        for (int w : G.adj(v))
13            if (!marked[w]) dfs(G, w);
14    }
15
16    //使用栈实现的深度优先搜索方法
17    private void dfs1(Digraph G, int v) {
18        Stack<Integer> stack = new Stack<Integer>();
19        stack.push(v);
20        marked[v] = true;
21
22        while (!stack.isEmpty()) {
23            int t = stack.pop();
24            for (int w : G.adj(t)) {
25                if (!marked[w]) {
26                    stack.push(w);
27                    marked[w] = true;
28                }
29            }
30        }
31    }
32
33    public boolean marked(int v) {
34        return marked[v];
35    }
36
37    public DirectedDFS(Digraph G, int s) {
38        marked = new boolean[G.V()];
39        dfs1(G, s);
40    }
41
42    public DirectedDFS(Digraph G, Iterable<Integer> sources) {
43        marked = new boolean[G.V()];
```

```

44     for (int s : sources)
45         if (!marked[s]) dfs1(G, s);
46     }
47
48     public static void main(String[] args) {
49         Digraph digraph = new Digraph(new In(args[0]));
50
51         Bag<Integer> sources = new Bag<Integer>();
52         for (int i = 1; i < args.length; ++i)
53             sources.add(Integer.parseInt(args[i]));
54
55         DirectedDFS reachable = new DirectedDFS(digraph, sources);
56         for (int v = 0; v < digraph.V(); v++)
57             if (reachable.marked(v)) Stdout.print(v + " ");
58         Stdout.println();
59     }
60 }
```

4.2.1.2 使用DFS路径搜索

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Stack;
3 import edu.princeton.cs.algs4.StdOut;
4
5 public class DepthFirstDirectedPaths {
6     private boolean[] marked;
7     private int[] edgeTo;
8     private final int s;
9
10    //使用递归实现的深度优先搜索方法
11    private void dfs(Digraph G, int s) {
12        marked[s] = true;
13        for (int w : G.adj(s))
14            if (!marked[w]) {
15                edgeTo[w] = s;
16                dfs(G, w);
17            }
18    }
19
20    //使用栈实现的深度优先搜索方法
21    private void dfs1(Digraph G, int s) {
22        Stack<Integer> stack = new Stack<Integer>();
23        stack.push(s);
24        edgeTo[s] = s;
25        marked[s] = true;
26
27        while (!stack.isEmpty()) {
28            int t = stack.pop();
29            for (int w : G.adj(t)) {
30                if (!marked[w]) {
31                    stack.push(w);
32                    edgeTo[w] = t;
33                    marked[w] = true;
34                }
35            }
36        }
37    }
38 }
```

```

36         }
37     }
38
39     public DepthFirstDirectedPaths(Digraph G, int s) {
40         marked = new boolean[G.V()];
41         edgeTo = new int[G.V()];
42         this.s = s;
43         dfs1(G, s);
44     }
45
46     public boolean hasPathTo(int v) {
47         return marked[v];
48     }
49
50     //返回从起点s到指定点v的路径容器引用
51     public Iterable<Integer> pathTo(int v) {
52         if (!hasPathTo(v)) return null;
53
54         Stack<Integer> stack = new Stack<Integer>();
55         for (int p = v; p != s; p = edgeTo[p])
56             stack.push(p);
57         stack.push(s);
58         return stack;
59     }
60
61     public static void main(String[] args) {
62         Digraph digraph = new Digraph(new In(args[0]));
63         int s = Integer.parseInt(args[1]);
64         DepthFirstDirectedPaths paths = new DepthFirstDirectedPaths(digraph,
65         s);
66
67         for (int v = 0; v < digraph.v(); ++v) {
68             Stdout.print(s + " to " + v + ": ");
69             if (paths.hasPathTo(v))
70                 for (int x : paths.pathTo(v))
71                     if (x == s) Stdout.print(x);
72                     else Stdout.print("-" + x);
73             Stdout.println();
74         }
75     }

```

4.2.1.3 使用BFS路径搜索

```

1 import edu.princeton.cs.algs4.*;
2
3 public class BreadFirstDirectedPaths {
4     private boolean[] marked;
5     private int[] parents;
6     private final int s;
7
8     //广度优先搜索路径方法
9     private void bfs(Digraph G, int s) {
10        Queue<Integer> queue = new Queue<Integer>();
11        queue.enqueue(s);

```

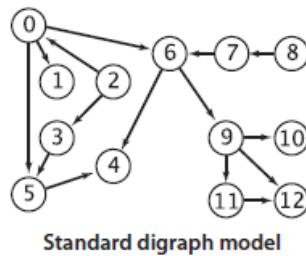
```

12     marked[s] = true;
13     parents[s] = s;
14
15     while (!queue.isEmpty()) {
16         int t = queue.dequeue();
17         for (int w : G.adj(t)) {
18             if (!marked[w]) {
19                 queue.enqueue(w);
20                 marked[w] = true;
21                 parents[w] = t;
22             }
23         }
24     }
25 }
26
27 public BreadFirstDirectedPaths(Digraph G, int s) {
28     marked = new boolean[G.V()];
29     parents = new int[G.V()];
30     this.s = s;
31     bfs(G, s);
32 }
33
34 public boolean hasPathTo(int v) {
35     return marked[v];
36 }
37
38 //返回从起点s到指定点v的路径容器引用
39 public Iterable<Integer> pathTo(int v) {
40     if (!hasPathTo(v)) return null;
41
42     Stack<Integer> stack = new Stack<Integer>();
43     for (int p = v; p != s; p = parents[p])
44         stack.push(p);
45     stack.push(s);
46     return stack;
47 }
48
49 public static void main(String[] args) {
50     Digraph digraph = new Digraph(new In(args[0]));
51     int s = Integer.parseInt(args[1]);
52     BreadFirstDirectedPaths paths = new BreadFirstDirectedPaths(digraph,
53     s);
54
55     for (int v = 0; v < digraph.V(); ++v) {
56         StdOut.print(s + " to " + v + ": ");
57         if (paths.hasPathTo(v))
58             for (int x : paths.pathTo(v))
59                 if (x == s) StdOut.print(x);
60                 else StdOut.print("-" + x);
61         StdOut.println();
62     }
63 }

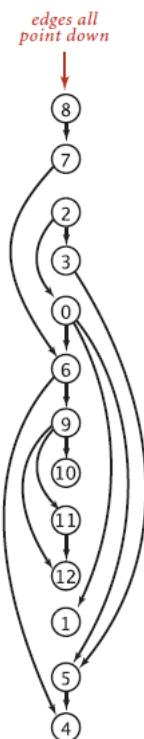
```

4.2.2 拓扑排序

拓扑排序指的是给定一幅有向图，将所有顶点进行排序，使得所有的有向边均从排在前面的元素指向排在后面的元素。需要注意的是，拓扑排序只能在有向无环图上进行，所以我们在拓扑排序之前就是先要判定这个有向图是否有环存在。



下面的排序顺序就是按照拓扑排序而行。



4.2.2.1 有向环检测

其判断的方法就是使用深度优先搜索DFS算法遍历整个图中的结点，若在遍历到顶点v时发现有一个顶点w已经在栈中了，则说明本来就有一条路径w->v，现在又发现一条路径v->w，即有环。

```
1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Stack;
3 import edu.princeton.cs.algs4.Stdout;
4
5 public class DirectedCycle {
6     private boolean[] marked;
7     private int[] edgeTo;
8     private Stack<Integer> cycle;
9     private boolean[] onStack;
10
11     private void dfs(Digraph G, int v) {
12         onStack[v] = true;
```

```

13     marked[v] = true;
14     for (int w : G.adj(v)) {
15         //若之前已经找到一个环了，则既然已经判定了有环就没有必要再找了
16         if (this.hasCycle()) return;
17         else if (!marked[w]) {
18             edgeTo[w] = v;
19             dfs(G, w);
20         }
21         /* 若w在栈中，说明有个v->w有个环，那么我们就可以沿着edgeTo数组向回找
22            以将这个环中的所有结点(v->w->v)加入到cycle中 */
23         else if (onStack[w]) {
24             cycle = new Stack<Integer>();
25             for (int p = v; p != w; p = edgeTo[p])
26                 cycle.push(p);
27             cycle.push(w);
28             cycle.push(v);
29         }
30     }
31     onStack[v] = false;
32 }

33

34     public DirectedCycle(Digraph G) {
35         marked = new boolean[G.V()];
36         edgeTo = new int[G.V()];
37         onStack = new boolean[G.V()];
38         for (int v = 0; v < G.V(); ++v)
39             if (!marked[v]) dfs1(G, v);
40     }

41

42     public boolean hasCycle() {
43         return cycle != null;
44     }

45

46     public Iterable<Integer> cycle() {
47         return cycle;
48     }

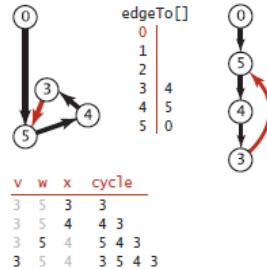
49

50     public static void main(String[] args) {
51         Digraph digraph = new Digraph(new In(args[0]));
52         DirectedCycle cycle = new DirectedCycle(digraph);

53

54         if (cycle.hasCycle()) {
55             StdOut.println("This Digraph has cycle:");
56             for (int v : cycle.cycle()) {
57                 StdOut.print(v + " ");
58             }
59             StdOut.println();
60         }
61     }
62 }
```

当当前顶点v发现自己身边有一个顶点w已经存放在栈中的时候，这就说明v有路径到w，而且w也有另一条路径到v：



Trace of cycle computation

4.2.2.2 深度优先排序

实际上由于深度优先搜索DFS算法总是会从起点一个劲的向自己深层次所指向的顶点进行搜索，因此在其遍历的过程中有向边的指出顶点是先于有向边的指入顶点而得到遍历。因此我们按照这一规则将先得到遍历的顶点最后放入到容器中，而最极端的顶点先放入到容器之中，那么我们容器中的元素的顺序就是典型的拓扑排序！（而实际中的拓扑排序也正是采用了这种方法来解决）。

排序时间与\$V+E\$成正比。

```

1 //伪代码
2 func dfs(G,v):
3     mark(v)
4     for every adjacency vertex of v:
5         dfs(G,adj)
6     put v to stack
7 end

```

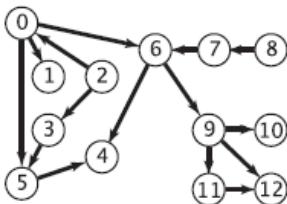
```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Queue;
3 import edu.princeton.cs.algs4.Stack;
4 import edu.princeton.cs.algs4.Stdout;
5
6 public class DepthFirstOrder {
7     private boolean[] marked;
8     private Queue<Integer> pre;
9     private Queue<Integer> post;
10    private Stack<Integer> reversePost;
11
12    private void dfs(Digraph G, int v) {
13        pre.enqueue(v); //前序遍历
14
15        marked[v] = true;
16        for (int w : G.adj(v))
17            if (!marked[w]) dfs(G, w);
18
19        post.enqueue(v); //后序遍历
20        reversePost.push(v); //逆后序遍历，得到的正是深度优先排序
21    }
22
23    public DepthFirstorder(Digraph G) {
24        pre = new Queue<Integer>();
25        post = new Queue<Integer>();
26        reversePost = new Stack<Integer>();
27        marked = new boolean[G.V()];
28    }

```

```
28     for (int v = 0; v < G.V(); ++v)
29         if (!marked[v]) dfs(G, v);
30     }
31
32     //返回前序遍历
33     public Iterable<Integer> pre() {
34         return pre;
35     }
36
37     //返回后序遍历
38     public Iterable<Integer> post() {
39         return post;
40     }
41
42     //返回逆后序遍历
43     public Iterable<Integer> reversePost() {
44         return reversePost;
45     }
46
47     public static void main(String[] args) {
48         Digraph digraph = new Digraph(new In(args[0]));
49         DepthFirstOrder order = new DepthFirstOrder(digraph);
50
51         for (int v : order.reversePost())
52             StdOut.print(v + " ");
53         StdOut.println();
54     }
55 }
```

前序、后序和逆后序遍历的不同之处：



*preorder
is order of
dfs() calls*

↓

	pre	post	reversePost
dfs(0)	0		
dfs(5)	0 5		
dfs(4)	0 5 4	queue	
4 done			
5 done	0 5 4 1		
dfs(1)		queue	
1 done	0 5 4 1	4 5	
dfs(6)	0 5 4 1 6		
dfs(9)	0 5 4 1 6 9		
dfs(11)	0 5 4 1 6 9 11	queue	
dfs(12)	0 5 4 1 6 9 11 12		stack
12 done		4 5 1 12	12 1 5 4
11 done		4 5 1 12 11	11 12 1 5 4
dfs(10)	0 5 4 1 6 9 11 12 10		
10 done		4 5 1 12 11 10	10 11 12 1 5 4
check 12			
9 done		4 5 1 12 11 10 9	9 10 11 12 1 5 4
check 4			
6 done		4 5 1 12 11 10 9 6	6 9 10 11 12 1 5 4
0 done		4 5 1 12 11 10 9 6 0	0 6 9 10 11 12 1 5 4
check 1	0 5 4 1 6 9 11 12 10 2		
dfs(2)			
check 0	0 5 4 1 6 9 11 12 10 2 3		
dfs(3)			
check 5		4 5 1 12 11 10 9 6 0 3	3 0 6 9 10 11 12 1 5 4
3 done			
2 done		4 5 1 12 11 10 9 6 0 3 2	2 3 0 6 9 10 11 12 1 5 4
check 3			
check 4			
check 5			
check 6			
dfs(7)	0 5 4 1 6 9 11 12 10 2 3 7		
check 6		4 5 1 12 11 10 9 6 0 3 2 7	7 2 3 0 6 9 10 11 12 1 5 4
7 done		0 5 4 1 6 9 11 12 10 2 3 7 8	
dfs(8)			
check 7		4 5 1 12 11 10 9 6 0 3 2 7 8	8 7 2 3 0 6 9 10 11 12 1 5 4
8 done			
check 9			
check 10			
check 11			
check 12			

Computing depth-first orders in a digraph (preorder, postorder, and reverse postorder)

4.2.2.3 拓扑排序

一幅有向无环图的拓扑排序即为所有顶点的逆后序排列。

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.Stdout;
3
4 public class Topological {
5     private Iterable<Integer> order;
6
7     public Topological(Digraph G) {
8         DirectedCycle cyclefinder = new DirectedCycle(G);
9         if (!cyclefinder.hasCycle()) {
10             DepthFirstOrder dfs = new DepthFirstOrder(G);
11             order = dfs.reversePost();
12         }
13     }

```

```

14
15     public Iterable<Integer> order() {
16         return order;
17     }
18
19     public boolean isDAG() {
20         return order != null;
21     }
22
23     public static void main(String[] args) {
24         Digraph digraph = new Digraph(new In(args[0]));
25         Topological topological = new Topological(digraph);
26
27         if (topological.isDAG()) {
28             for (int v : topological.order())
29                 Stdout.print(v + " ");
30             Stdout.println();
31         }
32     }
33 }

```

图示：

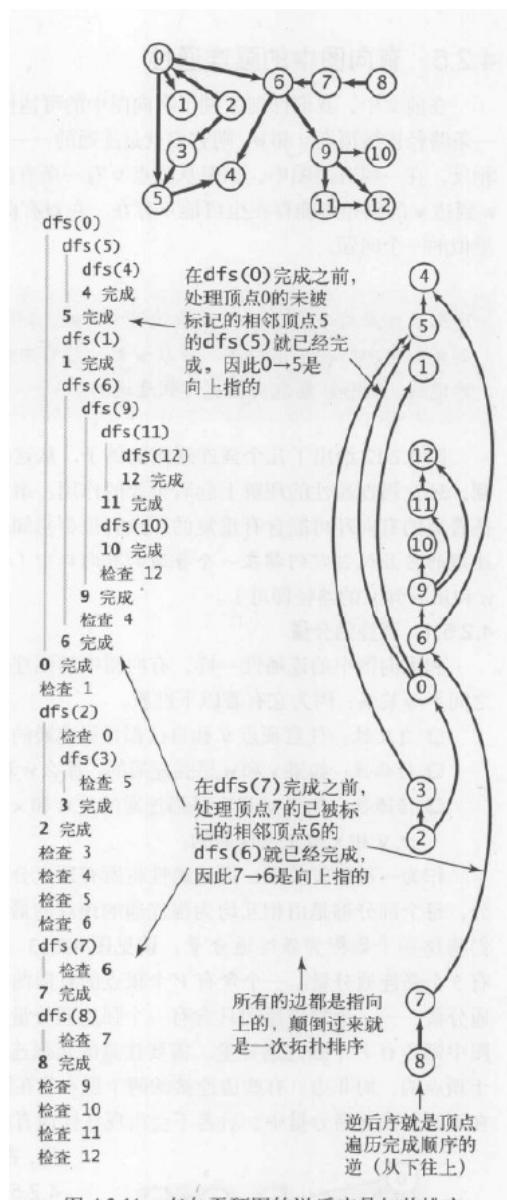


图 4.2.11 有向无环图的逆后序是拓扑排序

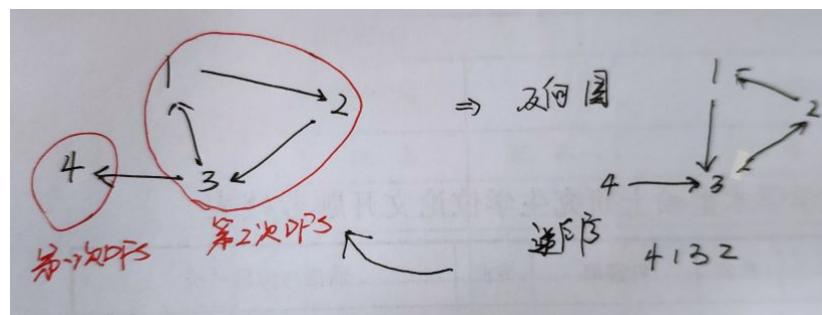
4.2.3 有向图中的强连通性

若有向图中的两个顶点 v, w , 它们互相可达, 则我们称它们为强连通。若在一个顶点集中每一个顶点相互强连通, 且这个顶点集是上述条件成立的最大顶点集, 则我们称这样的顶点集为强连通分量。更进一步, 当有向图中只有一个强连通分量的时候, 我们称这样的图为强连通图。

4.2.3.1 Kosaraju算法

对于有向图中的强连通分量的判断, 我们只需要记住Kosaraju算法的结论即可: **使用深度优先搜索算法查找①给定图\$G\$的反向图\$G^R\$, 根据②由此得到的所有顶点的逆后序③再用深度优先搜索处理有向图\$G\$, ④其构造函数中的每一次递归调用所标记的顶点都在同一个强连通分量之中。**

不过我们也可以细想这是为什么? 首先, 对给定图\$G\$的反向图\$G^R\$计算的逆后序, 其原图中的某一个强连通分量中的最深层次的总是会被放置到前面(这里仅仅相对于这个强连通分量而言), 因为反向图中的原本处于图中深层次位置的顶点变化了根部位置的顶点。若按照这个顺序遍历, 则从属于自己强连通分量之中的深层次顶点开始DFS, 则当显然会遍历到属于同一连通分量之中的顶点, 但是对处别的强连通分量之中的顶点它是无法遍历到的。因此有了上面④的结果。



```
1 import edu.princeton.cs.algs4.Bag;
2 import edu.princeton.cs.algs4.In;
3 import edu.princeton.cs.algs4.Stack;
4 import edu.princeton.cs.algs4.Stdout;
5
6 public class KosarajuCC {
7     private boolean[] marked;
8     private int[] id;
9     private int count;
10
11     // 使用递归实现的深度优先遍历算法, 以将同一个强连通分量中的顶点进行标识
12     private void dfs(Digraph G, int v) {
13         marked[v] = true;
14         id[v] = count;
15         for (int w : G.adj(v))
16             if (!marked[w])
17                 dfs(G, w);
18     }
19
20     // 使用栈实现的深度优先遍历算法, 以将同一个强连通分量中的顶点进行标识
21     private void dfs1(Digraph G, int v) {
22         Stack<Integer> stack = new Stack<Integer>();
23         stack.push(v);
24         marked[v] = true;
25         id[v] = count;
26
27         while (!stack.isEmpty()) {
28             int t = stack.pop();
```

```

29         for (int w : G.adj(t)) {
30             if (!marked[w]) {
31                 stack.push(w);
32                 marked[w] = true;
33                 id[w] = count;
34             }
35         }
36     }
37 }
38
39 public KosarajuCC(Digraph G) {
40     marked = new boolean[G.V()];
41     id = new int[G.V()];
42     DepthFirstOrder order = new DepthFirstOrder(G.reverse());
43     for (int s : order.reversePost())
44         if (!marked[s]) {
45             dfs1(G, s);
46             count++;
47         }
48 }
49
50 public boolean stronglyConnected(int v, int w) {
51     return id[v] == id[w];
52 }
53
54 public int id(int v) {
55     return id[v];
56 }
57
58 public int count() {
59     return count;
60 }
61
62 public static void main(String[] args) {
63     Digraph digraph = new Digraph(new In(args[0]));
64     KosarajuCC cc = new KosarajuCC(digraph);
65
66     int M = cc.count();
67     StdOut.println(digraph.V() + " vertexes, " + M +
68                   " strongly connected components:");
69
70     Bag<Integer>[] bags = new Bag[M];
71     for (int i = 0; i < M; ++i)
72         bags[i] = new Bag<Integer>();
73     for (int i = 0; i < digraph.V(); ++i)
74         bags[cc.id(i)].add(i);
75
76     for (int i = 0; i < M; ++i) {
77         for (int v : bags[i])
78             StdOut.print(v + " ");
79         StdOut.println();
80     }
81 }
82 }
```

4.2.3.2 有向图中顶点对的可达性问题

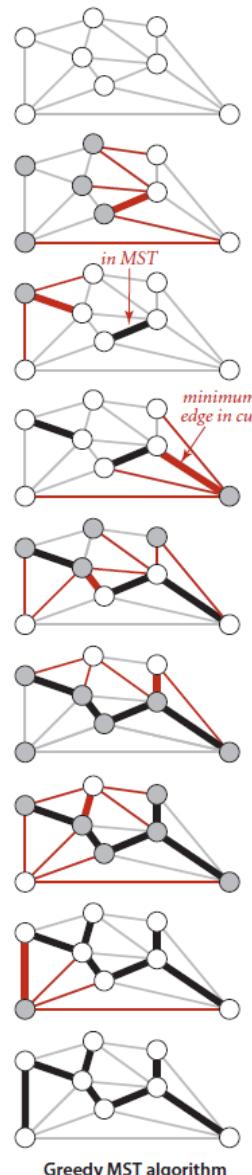
在无向图中问顶点v是否可到达顶点w，其本质就是连通性问题。但是在有向图中，这个问题并非如此，只有两个相互可达才能被称为强连通，所以这本质上是两个问题。解决能够随机查询点v到点w是否可达，可以通过对每一个顶点执行深度优先搜索算法，并把DFS结果存放在一个容器之中（在这里使用到一个存放DirectedDFS类对象的引用数组）。

```
1 public class TransitiveClosure {
2     private DirectedDFS[] all;
3
4     TransitiveClosure(Digraph G){
5         all=new DirectedDFS[G.V()];
6         for(int v=0;v<G.V();++v)
7             all[v]=new DirectedDFS(G,v);
8     }
9
10    boolean reachable(int v,int w){
11        return all[v].marked(w);
12    }
13 }
```

4.3 最小生成树

图的生成图指的是它的一棵含有其所有顶点的无环连通子图。一幅加权无向图的最小生成树(MST)是它的一棵权值（树中所有边的权值之和）最小的生成树。

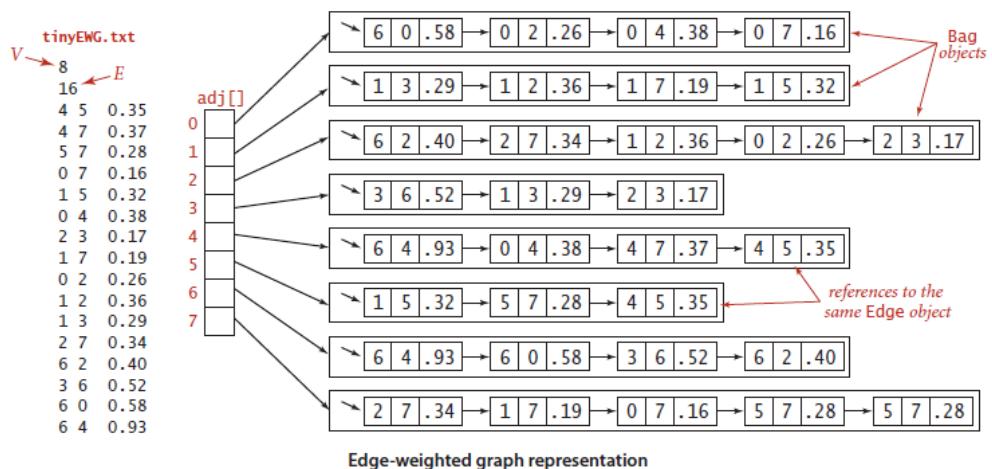
切分定理：在一幅加权图中，给定任意的切分，它的横截边中权重最小者必然是属于图的最小生成树。按照这一规则，我们有了最小生成树的贪心算法：将图中初始状态下的所有边标记为灰色，找到其中一种切分，它产生的横截边均不为黑色。将它的权重最小横截边标记为黑色。反复，知道标记V-1条黑色边为止。如下图：



Greedy MST algorithm

4.3.1 加权无向图的表示

我们所有的最小生成树计算都是基于加权无向图这一数据结构，其与无向图数据结构最大的区别在于它使用类对象Edge来代替了原始临界表中的原始数据类型int。



```

1 import edu.princeton.cs.algs4.Bag;
2 import edu.princeton.cs.algs4.In;
3 import edu.princeton.cs.algs4.StdOut;
4

```

```

5  public class EdgeweightedGraph {
6      private final int V;
7      private int E;
8      private Bag<Edge>[] adj;
9
10     public EdgeweightedGraph(int v) {
11         this.V = v;
12         this.E = 0;
13         adj = (Bag<Edge>[]) new Bag[v];
14         for (int v = 0; v < V; ++v)
15             adj[v] = new Bag<Edge>();
16     }
17
18     public EdgeweightedGraph(In in) {
19         this(in.readInt());
20         int E = in.readInt();
21         for (int i = 0; i < E; i++) {
22             Edge edge = new Edge(in.readInt(), in.readInt(),
in.readDouble());
23             addEdge(edge);
24         }
25     }
26
27     public void addEdge(Edge e) {
28         int v = e.either(), w = e.other(v);
29         adj[v].add(e);
30         adj[w].add(e);
31         E++;
32     }
33
34     //返回某一顶点的所有邻边
35     public Iterable<Edge> adj(int v) {
36         return adj[v];
37     }
38
39     //返回存放加权无向图中的所有的边的容器
40     public Iterable<Edge> edges() {
41         Bag<Edge> b = new Bag<Edge>();
42         for (int v = 0; v < V; v++)
43             for (Edge e : adj[v])
44                 /* 因为边Edge会在加权无向图中的两个顶点的邻接表中存放两次,
45                    所以我们在将边存放到容器中时应当避免存放两次。在这里只
46                    获取存放在索引较小的顶点邻接表中的边 */
47                 if (e.other(v) > v) b.add(e);
48         return b;
49     }
50
51     public int V() {
52         return V;
53     }
54
55     public int E() {
56         return E;
57     }
58
59     public static void main(String[] args) {
60         EdgeweightedGraph graph = new EdgeweightedGraph(new In(args[0]));
61     }

```

```
62     for (Edge edge : graph.edges())  
63         Stdout.println(edge);  
64     }  
65 }
```

4.3.2 Prim算法

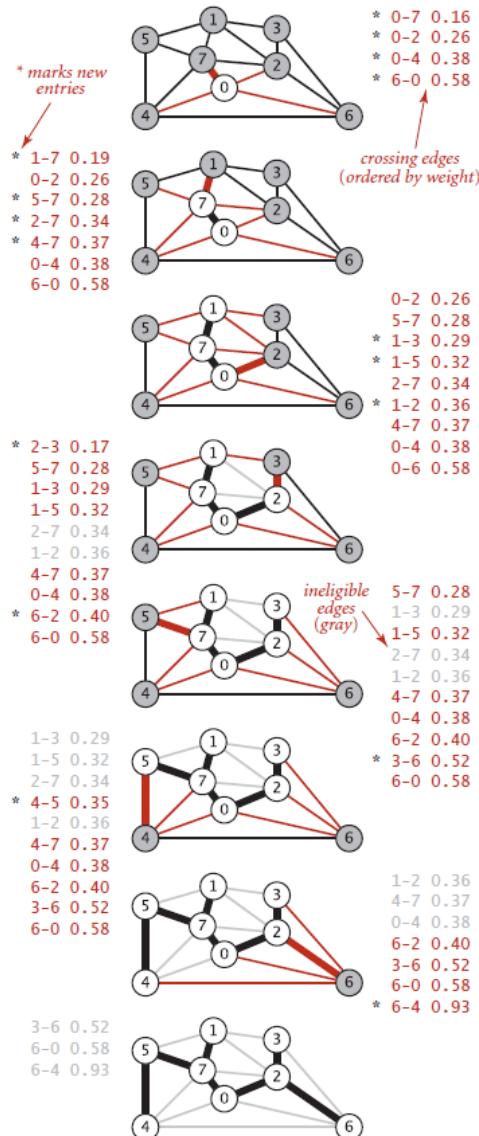
Prim算法的核心思想就是：在最小生成树MST生长的过程中，每一次迭代总是将不在MST中但与之连接的顶点且权重最小的边加入到MST之中。

4.3.2.1 Prim算法延时实现

其具体的做法就是：

1. 先将图中的某一点加入到MST之中（通过marked标记），并将其所有的邻边加入到最小优先队列MinPQ中；
2. 接着MinPQ从中取出最小连接边，若对端的顶点没有被标记，然后紧接着对对端的顶点执行步骤1，将其所有有效邻边加入到MinPQ；
3. 实际上MST不仅维护着一群顶点（通过marked布尔类型数组），而且还维护着一群边（通过一个容器）。

这样我们就可以不断地重复1-2步骤，就可以生成最小生成树MST了，具体如下图所示：



Trace of Prim's algorithm (lazy version)

时间复杂度: $E \log V$

空间复杂度: V

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.MinPQ;
3 import edu.princeton.cs.algs4.Queue;
4 import edu.princeton.cs.algs4.Stdout;
5
6 public class LazyPrimMST {
7     private boolean[] marked;
8     private Queue<Edge> mst;
9     private MinPQ<Edge> pq;
10
11    public LazyPrimMST(EdgeWeightedGraph G) {
12        pq = new MinPQ<Edge>();
13        marked = new boolean[G.V()];
14        mst = new Queue<Edge>();
15
16        visit(G, 0);
17        while (!pq.isEmpty()) {

```

```

18     Edge e = pq.delMin();
19
20     int v = e.either(), w = e.other(v);
21     /*之所以需要跳过一些无效的边，是因为这些边之前就被添加到
22      *最小优先队列之中（当时它们确实是有效的），但是随着两端
23      *顶点陆续加入MST就导致这些边变得无效，且仍然存在于MinPQ
24      *之中。因此我们必须通过检查优先队列之中每一条取出边，通过
25      *检查两端的顶点是否加入到MST的方法来检测这条边是否有效。
26      *只有在有效的前提下加入到MST之中*/
27     if (marked[v] && marked[w]) continue;
28     mst.enqueue(e);
29     if (!marked[v]) visit(G, v);
30     if (!marked[w]) visit(G, w);
31
32 }
33 }
34
35 //将v的所有有效邻边加入到MinPQ之中
36 private void visit(EdgeweightedGraph G, int v) {
37     marked[v] = true;
38     for (Edge edge : G.adj(v))
39         if (!marked[edge.other(v)])
40             pq.insert(edge);
41 }
42
43 //返回最小生成树中的所有边
44 public Iterable<Edge> edges() {
45     return mst;
46 }
47
48 public double weight() {
49     double ret = 0.0;
50     for (Edge e : mst)
51         ret += e.weight();
52     return ret;
53 }
54
55 public static void main(String[] args) {
56     EdgeweightedGraph graph = new EdgeweightedGraph(new In(args[0]));
57     LazyPrimMST mst = new LazyPrimMST(graph);
58
59     for (Edge e : mst.edges())
60         Stdout.println(e);
61 }
62 }
```

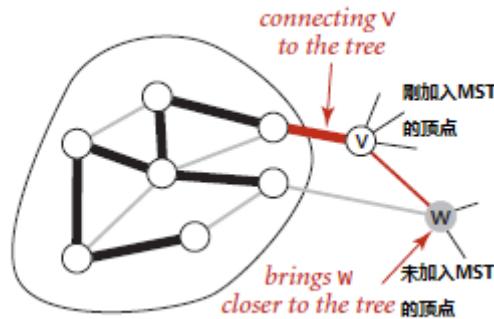
4.3.2.2 Prim算法即时实现

Prim算法延时实现和及时实现的区别在于：

- Prim算法的延时实现的关键在于让最小优先队列MinPQ维护着MST中点的所有未加入MST邻边（这些邻边中本身还包括无效的边，即它们的两端顶点已经加入到MST之中），然后在每一次迭代中取出权值最小的有效邻边和那个对端点加入到MST之中；
- 而Prim算法的及时实现的关键在于让最小索引优先队列IndexMinPQ维护着未加入到MST的剩余顶点到MST的最小权重距离，这些权重距离（有些是多条边权重的叠加，有些是直接的

邻边的权重）。然后我们在每一次迭代中取出连接到MST代价最小的未加入点，让该点加入到MST之中，随便添加那条连接边。

所以我们可以看到，延时实现的关键在于维护未加入MST的邻边，而问题在于邻边可能有些是无效的；而及时实现的关键在于维护未加入MST的所有点以及它们到MST的距离，这些距离对点而言都是唯一的，当这个点到MST距离最小时一定这个距离指的就是连接两者的那条邻边。



每次将一个点加入到MST之后，就需要利用现有的信息更新所有为加入到MST顶点到MST权重距离。

最小生成树生成时间复杂度：\$E\log V\$

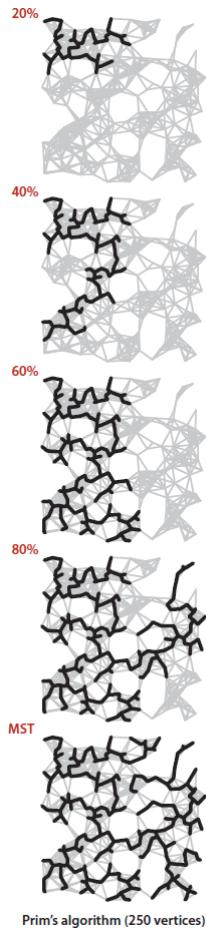
空间复杂度：\$V\$

```
1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.IndexMinPQ;
3 import edu.princeton.cs.algs4.Queue;
4 import edu.princeton.cs.algs4.Stdout;
5
6 public class PrimMST {
7     private Edge[] edgeTo;
8     private double[] distTo;
9     private boolean[] marked;
10    private IndexMinPQ<Double> pq;
11
12    /* 访问结点v的所有邻边，若该边有效且它的权重比现有w到最小生成树的权
13       重还要低，则将其替换为最新边edgeTo[w]=...和权重值distTo[w]=... */
14    private void visit(EdgeWeightedGraph G, int v) {
15        marked[v] = true;
16        for (Edge e : G.adj(v)) {
17            /* 每加入一个顶点到MST之后，就遍历它的所有临近顶点，
18               更新加入到MST的权重路径信息 */
19            int w = e.other(v);
20            if (!marked[w] && e.weight() < distTo[w]) {
21                edgeTo[w] = e;
22                distTo[w] = e.weight();
23                if (pq.contains(w)) pq.change(w, distTo[w]);
24                else pq.insert(w, distTo[w]);
25            }
26        }
27    }
28
29    public PrimMST(EdgeWeightedGraph G) {
30        edgeTo = new Edge[G.V()];
31        distTo = new double[G.V()];
```

```

32     marked = new boolean[G.V()];
33     for (int v = 0; v < G.V(); ++v)
34         distTo[v] = Double.POSITIVE_INFINITY;
35     pq = new IndexMinPQ<Double>(G.V());
36
37     distTo[0] = 0.0;
38     pq.insert(0, 0.0);
39     while (!pq.isEmpty())
40         visit(G, pq.delMin());
41 }
42
43 //返回最小生成树中的所有边
44 public Iterable<Edge> edges() {
45     Queue<Edge> queue = new Queue<Edge>();
46     for (Edge edge : edgeTo)
47         if (edge != null)
48             queue.enqueue(edge);
49     return queue;
50 }
51
52 public double weight() {
53     double ret = 0.0;
54     for (Edge edge : edges())
55         ret += edge.weight();
56     return ret;
57 }
58
59 public static void main(String[] args) {
60     EdgeweightedGraph graph = new EdgeweightedGraph(new In(args[0]));
61     PrimMST mst = new PrimMST(graph);
62
63     for (Edge edge : mst.edges())
64         Stdout.println(edge);
65     StdOut.printf("Total price: %.2f", mst.weight());
66 }
67 }
```

过程图示:



Prim's algorithm (250 vertices)

4.3.3 Kruskal算法

算法原理：按照权重顺序（从小到大）处理它们，将边加入到最小生成树MST之中，加入到的边不会形成环，直到树中不会含有 $V-1$ 条边为止。即由一片森林变成一棵树。

时间复杂度： $E \log E$

空间复杂度： E

```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.MinPQ;
3 import edu.princeton.cs.algs4.StdOut;
4 import edu.princeton.cs.algs4.UF;
5
6 import java.util.ArrayDeque;
7 import java.util.Queue;
8
9 public class KruskalMST {
10     private Queue<Edge> mst;
11
12     public KruskalMST(EdgeWeightedGraph G) {
13         mst = new ArrayDeque<Edge>();
14         MinPQ<Edge> pq = new MinPQ<Edge>();
15         UF uf = new UF(G.V()); //并查集用来检测换的存在性
16         for (Edge e : G.edges()) pq.insert(e);
17
18         while (!pq.isEmpty() && mst.size() < G.V() - 1) {
19             Edge e = pq.delMin();
20             int v = e.either(), w = e.other(v);

```

```

21     /* 若这两条边之前就已经相连了，则此时再添加这条v-w的新边，
22      * 必然会导致MST形成一个环，所以我们应该跳过这条边，将其
23      * 从最小优先队列中删除 */
24     if (uf.connected(v, w)) continue;
25     uf.union(v, w);
26     mst.add(e);
27   }
28 }
29
30 public Iterable<Edge> edges() {
31   return mst;
32 }
33
34 public double weight() {
35   double ret = 0.0;
36   for (Edge e : mst)
37     ret += e.weight();
38   return ret;
39 }
40
41 public static void main(String[] args) {
42   EdgeWeightedGraph graph = new EdgeWeightedGraph(new In(args[0]));
43   KruskalMST mst = new KruskalMST(graph);
44   for (Edge edge : mst.edges())
45     StdOut.println(edge);
46   StdOut.printf("total weight: %.2f", mst.weight());
47 }
48 }
```

各个最小生成树算法总结（假设为V个顶点E条边的连通图，在最坏情况下）：

算法	空间复杂度	时间复杂度
延时的Prim算法	\$E\$	\$E \log E\$
即时的Prim算法	\$V\$	\$E \log V\$
Kruskal算法	\$E\$	\$E \log E\$

其中即时的Prim算法是这里面最好的。

4.4 最短路径

加权有向图的数据结构表示: `public EdgeWeightDigraph`

- `EdgeWeightDigraph(int v)`
- `EdgeWeightDigraph(In in)`
- `void addEdge(DirectedEdge edge)`
- `int V()`
- `int E()`
- `Iterable<DirectedEdge> adj(int v)`
- `Iterable<DirectedEdge> edges()`

```

1 import edu.princeton.cs.algs4.Bag;
2 import edu.princeton.cs.algs4.In;
3 import edu.princeton.cs.algs4.StdOut;
```

```

4
5  public class Edgeweightingraph {
6      private final int V;
7      private int E;
8      private Bag<DirectedEdge>[] adj;
9
10     public Edgeweightingraph(int v) {
11         this.V = v;
12         this.E = 0;
13         adj = (Bag<DirectedEdge>[]) new Bag[V];
14         for (int v = 0; v < V; ++v)
15             adj[v] = new Bag<DirectedEdge>();
16     }
17
18     public Edgeweightingraph(In in) {
19         this(in.readInt());
20         this.E = in.readInt();
21         while (!in.isEmpty()) {
22             DirectedEdge edge = new DirectedEdge(in.readInt(), in.readInt(),
23             in.readDouble());
24             addEdge(edge);
25         }
26     }
27
28     public int V() {
29         return V;
30     }
31
32     public int E() {
33         return E;
34     }
35
36     public void addEdge(DirectedEdge edge) {
37         adj[edge.from()].add(edge);
38         // E++;
39     }
40
41     public Iterable<DirectedEdge> adj(int v) {
42         return adj[v];
43     }
44
45     public Iterable<DirectedEdge> edges() {
46         Bag<DirectedEdge> bag = new Bag<DirectedEdge>();
47         for (int v = 0; v < V; ++v)
48             for (DirectedEdge edge : adj[v])
49                 bag.add(edge);
50         return bag;
51     }
52
53     public static void main(String[] args) {
54         Edgeweightingraph digraph = new Edgeweightingraph(new In(args[0]));
55
56         for (int v = 0; v < digraph.V(); ++v) {
57             for (DirectedEdge edge : digraph.adj(v))
58                 Stdout.print(edge + " ");
59             Stdout.println();
60         }
61     }

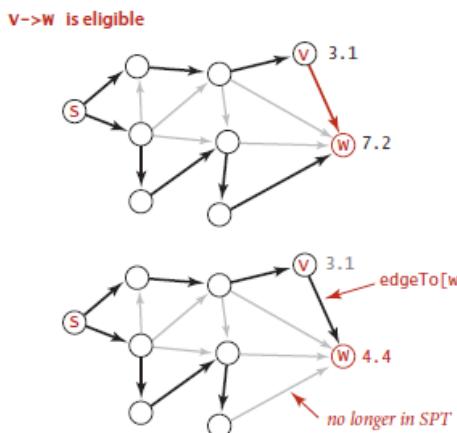
```

4.4.1 Dijkstra算法

Dijkstra算法主要用来解决非负加权有向图的最路径问题，Dijkstra算法的核心思想其实与Prim算法类似，Prim的核心在于每一次迭代的过程中添加离MST最近的未加入MST顶点，而Dijkstra算法的核心在于每次迭代的过程中添加离起点最近的未加入SPT的顶点（其中MST指的是最小生成树，SPT指的是最短路径树）。

在Prim算法中，我们最重要的操作就是`visit()`：我们每次向最小生成树MST加入一个顶点之后，就会通过它的所有邻边去更新剩余未加入顶点到MST的距离信息`distTo[]`和`edgeTo[]`（`distTo[]`也必然是在索引最小优先队列`IndexMinPQ`之中）。完成这些更新操作之后，Prim算法就会从队列中取出下一个最小顶点重复上面的操作。这里的`visit()`操作的本质就是①加入一个顶点和②更新剩余的顶点信息

同样的，在Dijkstra算法中也有类似的操作，可以说基本上与Prim算法类似，即书中所谓的放松`relax()`操作：我们每次向最短路径树加入一个顶点之后，就会通过它的所有邻边去更新剩余未加入顶点到起点的距离信息`distTo[]`和指入边`edgeTo[]`（`distTo[]`同样的也是在索引最小优先队列`IndexMinPQ`之中）。完成这些操作之后，Dijkstra算法就会从队列中取出下一个最小顶点重复上面的操作。因此，我们可以认为Dijkstra算法其实本质上和Prim算法有很多相似的地方⑨。下面演示的是顶点v加入到SPT之后对`distTo[w]`和`edgeTo[w]`的`relax()`更新操作：



```

1 import edu.princeton.cs.algs4.In;
2 import edu.princeton.cs.algs4.IndexMinPQ;
3 import edu.princeton.cs.algs4.Stack;
4 import edu.princeton.cs.algs4.StdOut;
5
6 public class DijkstrasP {
7     private DirectedEdge[] edgeTo;
8     private double[] distTo;
9     private IndexMinPQ<Double> pq;
10
11    public DijkstrasP(EdgeWeightDigraph G, int s) {
12        edgeTo = new DirectedEdge[G.V()];
13        distTo = new double[G.V()];
14        pq = new IndexMinPQ<Double>(G.V());
15
16        for (int v = 0; v < G.V(); ++v)
17            distTo[v] = Double.POSITIVE_INFINITY;
18        distTo[s] = 0.0;
19
20        pq.insert(s, 0.0);

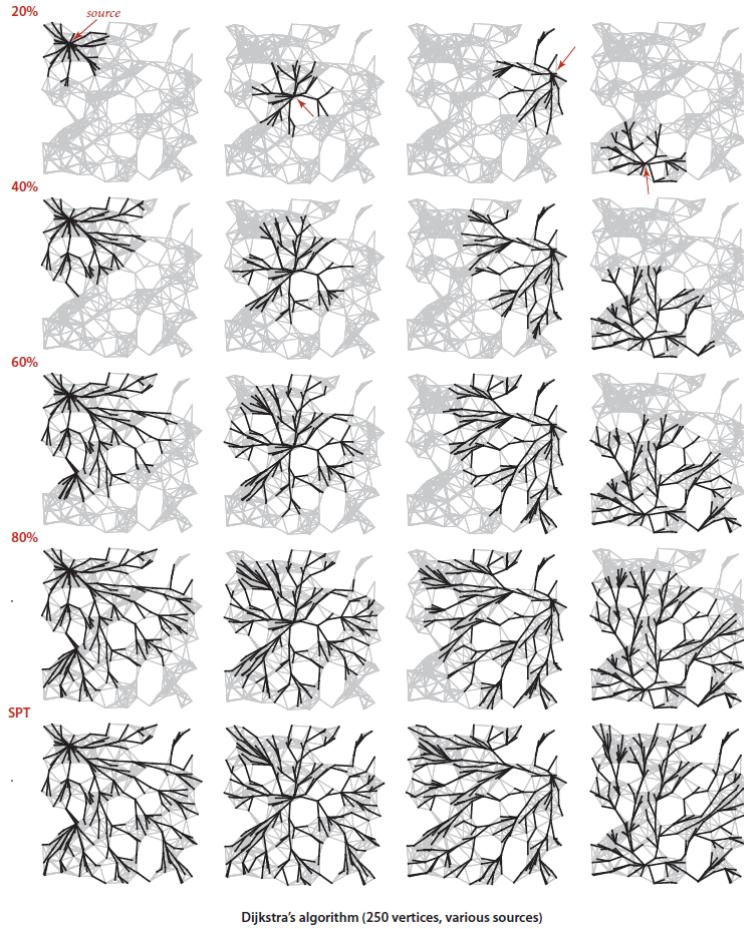
```

```

21     while (!pq.isEmpty())
22         relax(G, pq.delMin());
23     }
24
25     private void relax(EdgeweightedDigraph G, int v) {
26         for (DirectedEdge e : G.adj(v)) {
27             int w = e.to();
28             if (distTo[w] > distTo[v] + e.weight()) {
29                 distTo[w] = distTo[v] + e.weight();
30                 edgeTo[w] = e;
31                 if (pq.contains(w)) pq.changeKey(w, distTo[w]);
32                 else pq.insert(w, distTo[w]);
33             }
34         }
35     }
36
37     public double distTo(int v) {
38         return distTo[v];
39     }
40
41     public boolean hasPathTo(int v) {
42 //        return edgeTo[v] != null;
43 //        return distTo[v] < Double.POSITIVE_INFINITY;
44     }
45
46     public Iterable<DirectedEdge> pathTo(int v) {
47         if (!hasPathTo(v)) return null;
48         Stack<DirectedEdge> path = new Stack<DirectedEdge>();
49         for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
50             path.push(e);
51         return path;
52     }
53
54     public static void main(String[] args) {
55         EdgeweightedDigraph digraph = new EdgeweightedDigraph(new In(args[0]));
56         DijkstrasP dijkstrasP = new DijkstrasP(digraph, 0);
57
58         for (int v = 1; v < digraph.v(); ++v) {
59             Stdout.print(0 + " -> " + v + ": ");
60             double weight = 0.0;
61             for (DirectedEdge edge : dijkstrasP.pathTo(v)) {
62                 weight += edge.weight();
63                 if (edge.from() != 0)
64                     Stdout.printf("->%d", edge.to());
65                 else
66                     Stdout.printf("%d->%d", edge.from(), edge.to());
67             }
68             Stdout.printf(" (total weight: %.2f)\n", weight);
69         }
70     }
71 }

```

计算过程图示：



4.4.2 无环加权有向图中的最短路径算法

计算无环加权有向图中的最短路径的核心思想就是：按照无环图中的拓扑顺序对图中的所有顶点进行relax松弛操作（沿路更新 $\text{distTo}[w]$ 和 $\text{edgeTo}[w]$ ）。这样我们就可以简单的获得其中的最短路径。

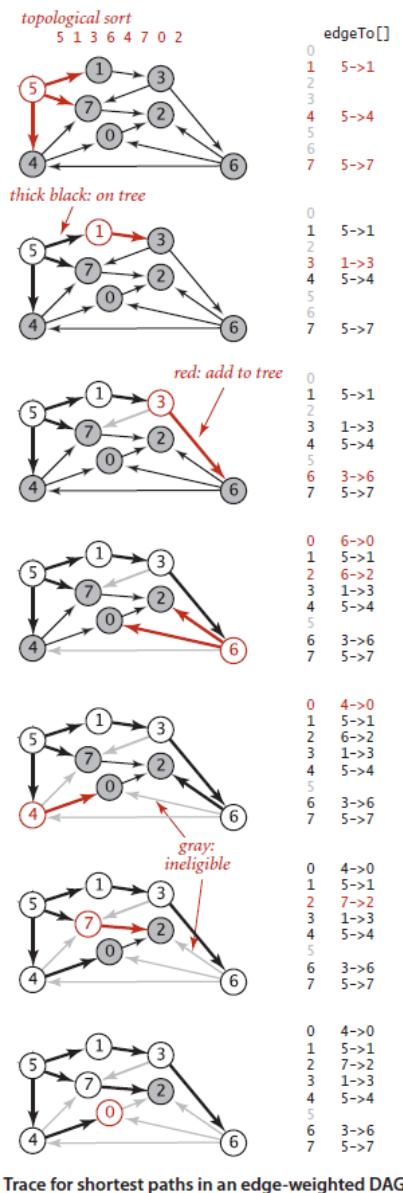
```

1 import edu.princeton.cs.algs4.*;
2 import edu.princeton.cs.algs4.DirectedEdge;
3
4 public class AcyclicSP {
5     private DirectedEdge[] edgeTo;
6     private double[] distTo;
7
8     private void relax(EdgeWeightedDigraph G, int v) {
9         for (DirectedEdge e : G.adj(v)) {
10             int w = e.to();
11             if (distTo[w] > distTo[v] + e.weight()) {
12                 distTo[w] = distTo[v] + e.weight();
13                 edgeTo[w] = e;
14             }
15         }
16     }
17
18     public AcyclicSP(EdgeWeightedDigraph G, int s) {
19         edgeTo = new DirectedEdge[G.V()];
20         distTo = new double[G.V()];
21         for (int v = 0; v < G.V(); ++v)
22             distTo[v] = Double.POSITIVE_INFINITY;
23         distTo[s] = 0.0;
24     }
25 }
```

```

24
25     /* 按照拓扑排序的顺序对每一个顶点进行relax放松操作，即
26     沿着拓扑顺序对路径上的每一个顶点的distTo[w]和edgeTo[w]
27     进行更新操作。 */
28     Topological top = new Topological(G);
29     for (int v : top.order())
30         relax(G, v);
31 }
32
33 public double distTo(int v) {
34     return distTo[v];
35 }
36
37 public boolean hasPathTo(int v) {
38     return distTo[v] < Double.POSITIVE_INFINITY;
39 }
40
41 public Iterable<DirectedEdge> pathTo(int v) {
42     if (!hasPathTo(v)) return null;
43     Stack<DirectedEdge> stack = new Stack<DirectedEdge>();
44     for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
45         stack.push(e);
46     return stack;
47 }
48
49 public static void main(String[] args) {
50     In in = new In(args[0]);
51     int s = Integer.parseInt(args[1]);
52     EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
53
54     AcyclicSP sp = new AcyclicSP(G, s);
55     for (int v = 0; v < G.V(); v++) {
56         if (sp.hasPathTo(v)) {
57             StdOut.printf("%d to %d (%.2f)\n", s, v, sp.distTo(v));
58             for (DirectedEdge e : sp.pathTo(v)) {
59                 StdOut.print(e + " ");
60             }
61             StdOut.println();
62         } else {
63             StdOut.printf("%d to %d          no path\n", s, v);
64         }
65     }
66 }
67 }
```

执行过程图示：



Trace for shortest paths in an edge-weighted DAG

与上面相反的是，使用拓扑排序+松弛操作组合还可以计算无环加权有向图中的最长路径，它的操作基本上与前者相反，尽可能取大一点。如下所示：

```

1 import edu.princeton.cs.algs4.*;
2 import edu.princeton.cs.algs4.DirectedEdge;
3
4 public class AcyclicLP {
5     private DirectedEdge[] edgeTo;
6     private double[] distTo;
7
8     private void relax(EdgeweightedDigraph G, int v) {
9         for (DirectedEdge e : G.adj(v)) {
10             int w = e.to();
11             if (distTo[w] < distTo[v] + e.weight()) {
12                 distTo[w] = distTo[v] + e.weight();
13                 edgeTo[w] = e;
14             }
15         }
16     }
17
18     public AcyclicLP(EdgeweightedDigraph G, int s) {

```

```

19     edgeTo = new DirectedEdge[G.V()];
20     distTo = new double[G.V()];
21     for (int v = 0; v < G.V(); ++v)
22         distTo[v] = Double.NEGATIVE_INFINITY;
23     distTo[s] = 0.0;
24
25     Topological top = new Topological(G);
26     for (int v : top.order())
27         relax(G, v);
28 }
29
30 public boolean hasPathTo(int v) {
31     return distTo[v] > Double.NEGATIVE_INFINITY;
32 }
33
34 public double distTo(int v) {
35     return distTo[v];
36 }
37
38 public Iterable<DirectedEdge> pathTo(int v) {
39     if (!hasPathTo(v)) return null;
40     Stack<DirectedEdge> stack = new Stack<DirectedEdge>();
41     for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
42         stack.push(e);
43     return stack;
44 }
45
46 public static void main(String[] args) {
47     In in = new In(args[0]);
48     int s = Integer.parseInt(args[1]);
49     EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
50
51     AcyclicLP lp = new AcyclicLP(G, s);
52
53     for (int v = 0; v < G.V(); v++) {
54         if (lp.hasPathTo(v)) {
55             StdOut.printf("%d to %d (%.2f) ", s, v, lp.distTo(v));
56             for (DirectedEdge e : lp.pathTo(v)) {
57                 StdOut.print(e + " ");
58             }
59             StdOut.println();
60         } else {
61             StdOut.printf("%d to %d          no path\n", s, v);
62         }
63     }
64 }
65 }
```

4.4.3 Bellman-Ford算法(一般加权有向图)

4.4.3.1 通用Bellman-Ford算法

Bellman-Ford算法主要是用来解决一般加权有向图中的最短路径问题。算法核心思想为：在任意含有 V 个顶点的加权有向图中给定起点 s ，从 s 无法到任何负权重环，则我们可以通过如下的方式计算单点最短路径问题：**将 $s \rightarrow s$ 的距离权重 $distTo[s]$ 初始化为0.0，其他的 $distTo[]$ 元素初始化为无穷大，然后以任意顺序对有向图中的所有边进行发送 relax 操作，重复 V 轮。**

因此我们很容易的就可以推断出这样的算法的时间复杂度为： $\$EV\$$ ，空间复杂度为： $\$V\$$ 。我们可以按照这个算法进行实现，因为它没有任何严格的预处理、处理顺序的要求，代码如下：

```
1 import edu.princeton.cs.algs4.DirectedEdge;
2 import edu.princeton.cs.algs4.EdgeWeightedDigraph;
3 import edu.princeton.cs.algs4.In;
4 import edu.princeton.cs.algs4.StdOut;
5
6 import java.util.Arrays;
7 import java.util.Stack;
8
9 public class BellmanFordSP {
10     private double[] distTo;
11     private DirectedEdge[] edgeTo;
12
13     //对指定的边进行松弛操作
14     private void relax(DirectedEdge e) {
15         int w = e.to(), v = e.from();
16         if (distTo[w] > distTo[v] + e.weight()) {
17             distTo[w] = distTo[v] + e.weight();
18             edgeTo[w] = e;
19         }
20     }
21
22     //算法正常执行的前提是没有负权重环
23     public BellmanFordSP(EdgeWeightedDigraph G, int s) {
24         distTo = new double[G.V()];
25         edgeTo = new DirectedEdge[G.V()];
26         Arrays.fill(distTo, Double.POSITIVE_INFINITY);
27
28         //VE次执行松弛操作
29         distTo[s] = 0.0;
30         for (int v = 0; v < G.V(); ++v)
31             for (DirectedEdge e : G.edges())
32                 relax(e);
33     }
34
35     public boolean hasPathTo(int v) {
36         return distTo[v] < Double.POSITIVE_INFINITY;
37     }
38
39     public double distTo(int v) {
40         return distTo[v];
41     }
42
43     public Iterable<DirectedEdge> pathTo(int v) {
44         if (!hasPathTo(v)) return null;
45         Stack<DirectedEdge> stack = new Stack<DirectedEdge>();
46         for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
47             stack.push(e);
```

```

48         return stack;
49     }
50
51     public static void main(String[] args) {
52         In in = new In(args[0]);
53         int s = Integer.parseInt(args[1]);
54         EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
55
56         BellmanFordSP sp = new BellmanFordSP(G, s);
57
58         for (int v = 0; v < G.V(); v++) {
59             if (sp.hasPathTo(v)) {
60                 StdOut.printf("%d to %d (%.2f) ", s, v, sp.distTo(v));
61                 for (DirectedEdge e : sp.pathTo(v))
62                     StdOut.print(e + " ");
63                 StdOut.println();
64             } else {
65                 StdOut.printf("%d to %d          no path\n", s, v);
66             }
67         }
68     }
69 }
```

负权重环对加权有向图计算最短路径的影响：若一个一般性加权有向图中出现了负权重环，那么这样的图中显然是计算不出最终的最短路径，因为我们完全可以在其中的路径中通过无限次绕着这个负权重环来无限的趋于负无穷，最终得到任意数目的更短路径。因此在对一个一般加权有向图计算最短路径的时候一定要检测其中是否存在负权重环。若图中不存在负权重环，则我们就可以使用Bellman-Ford算法来计算最短路径。

4.4.3.2 基于队列的Bellman-Ford算法

基于队列的Bellman-Ford算法不再粗暴的采用一个双重循环，而是基于一个基本事实：只有上一轮中的`distTo[]`值发生变化的顶点指出的边才能够改变其他`distTo[]`元素的值。它的意思就是说，**对于一个顶点w以及它的距离权重`distTo[w]`，只有在它的指入父亲顶点的`distTo[v]`发生改变的前提下才会发生改变，否则再多次的循环也对这个顶点最短路径的正确指入边没什么影响**。因此我们可以改进随机进行`relax`操作的Bellman-Ford算法，将其改成按照一定的“由父到子”的顺序（这里处理为队列顺序）执行`relax`操作。这样就可以加快最短路径算法的执行速度。（**其实你可以发现这种思想就是广度优先遍历BFS算法**，但不同的是这里的顶点可能被重新加入队列之中，被重新加入的原因只有一个，那就是图中有负权重边）

不过此时的时间复杂度仍然是： $\$EV\$$ ，空间复杂度为： $\$V\$$ 。

```

1 import edu.princeton.cs.algs4.*;
2 import edu.princeton.cs.algs4.DirectedEdge;
3
4 import java.util.Arrays;
5 import java.util.Stack;
6
7 public class BellmanFordSPQ {
8     private double[] distTo;
9     private DirectedEdge[] edgeTo;
10    private Queue<Integer> queue;
11    private boolean[] onQ;
12
13    //对v->w进行松弛操作
```

```

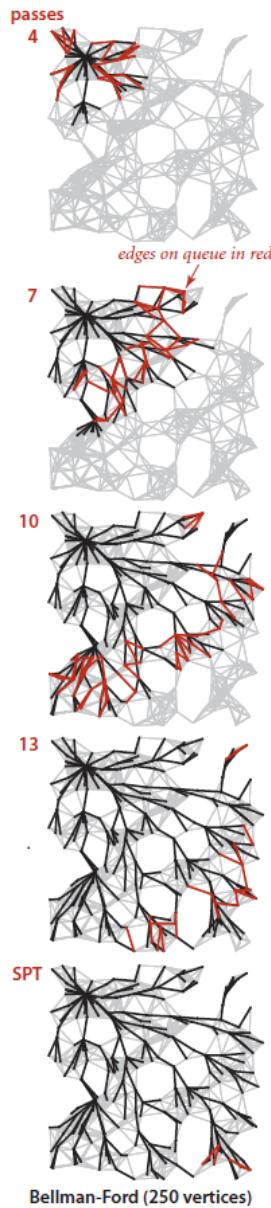
14     private void relax(EdgeWeightedDigraph G, int v) {
15         for (DirectedEdge e : G.adj(v)) {
16             int w = e.to();
17             if (distTo[w] > distTo[v] + e.weight()) {
18                 distTo[w] = distTo[v] + e.weight();
19                 edgeTo[w] = e;
20                 //若指出顶点没有在队列之中，那么就将其加入到队列中
21                 if (!onQ[w]) {
22                     onQ[w] = true;
23                     queue.enqueue(w);
24                 }
25             }
26         }
27     }
28
29     //正常执行的前提是该加权有向图中没有负权重环
30     public BellmanFordSPQ(EdgeWeightedDigraph G, int s) {
31         distTo = new double[G.V()];
32         edgeTo = new DirectedEdge[G.V()];
33         queue = new Queue<Integer>();
34         onQ = new boolean[G.V()];
35         Arrays.fill(distTo, Double.POSITIVE_INFINITY);
36
37         /* 这里所用的算法思想（在图中不存在负权重环情况下）就是
38            广度优先遍历BFS算法 */
39         distTo[s] = 0.0;
40         queue.enqueue(s);
41         while (!queue.isEmpty()) {
42             int v = queue.dequeue();
43             onQ[v] = false;
44             relax(G, v);
45         }
46     }
47
48     public boolean hasPathTo(int v) {
49         return distTo[v] < Double.POSITIVE_INFINITY;
50     }
51
52     public double distTo(int v) {
53         return distTo[v];
54     }
55
56     public Iterable<DirectedEdge> pathTo(int v) {
57         if (!hasPathTo(v)) return null;
58         Stack<DirectedEdge> stack = new Stack<DirectedEdge>();
59         for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
60             stack.push(e);
61         return stack;
62     }
63
64     public static void main(String[] args) {
65         In in = new In(args[0]);
66         int s = Integer.parseInt(args[1]);
67         EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
68         BellmanFordSPQ sp = new BellmanFordSPQ(G, s);
69
70         for (int v = 0; v < G.V(); v++) {
71             if (sp.hasPathTo(v)) {

```

```

72         Stdout.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
73         for (DirectedEdge e : sp.pathTo(v))
74             Stdout.print(e + " ");
75         Stdout.println();
76     } else {
77         Stdout.printf("%d to %d           no path\n", s, v);
78     }
79 }
80 }
81 }
```

算法执行过程图示：



4.4.3.3 负权重环检测

在上面的讨论中我们已经提到过加权有向图中负权重环的存在会严重的影响最短路径的计算（对于上面没有负权重环检测能力的基于队列的Bellman-Ford算法，若用来计算一个含有负权重环有向图的话会陷入一个死循环）。因此我们必须在图最短路径计算的过程中需要证明负权重环的存在性。若存在则没有最短路径，我们此时应该从循环中退出并得到那个环的具体路径；否则我们可以获得最短路径树SPT。

对于负权重环的检测我们直接使用书中的结论：若将所有边放松 V 轮之后当且仅当队列非空时有向图中才存在从起点可达的负权重环。也就是说我们没调用 V 次relax()操作就应该检查下当前找到的最短路径树SPT中是否存在负权重环（这里只要检测到有环就必然是负权重环的，因为只有负权重环才会让一个最短路径树SPT从一点又回到自己本身）

```
1 import edu.princeton.cs.algs4.*;
2 import edu.princeton.cs.algs4.DirectedEdge;
3
4 import java.util.Arrays;
5 import java.util.Stack;
6
7 public class BellmanFordSPQ1 {
8     private double[] distTo;
9     private DirectedEdge[] edgeTo;
10    private Queue<Integer> queue;
11    private boolean[] onQ;
12    private Iterable<DirectedEdge> cycle;
13    private int count;
14
15    //松弛操作
16    private void relax(EdgeweightedDigraph G, int v) {
17        for (DirectedEdge e : G.adj(v)) {
18            int w = e.to();
19            if (distTo[w] > distTo[v] + e.weight()) {
20                distTo[w] = distTo[v] + e.weight();
21                edgeTo[w] = e;
22                if (!onQ[w]) {
23                    queue.enqueue(w);
24                    onQ[w] = true;
25                }
26            }
27            //每V轮就调用一次findNegativeCycle()检测负权重环的存在
28            if (++count % G.V() == 0)
29                findNegativeCycle();
30        }
31    }
32
33    //在Bellman-Ford算法当前找到的最短路径树SPT中寻找负权重环（使用深度优先遍历算法
34    DFS）
35    private void findNegativeCycle() {
36        int V = edgeTo.length;
37        EdgeweightedDigraph spt = new EdgeweightedDigraph(V);
38        for (int v = 0; v < V; ++v)
39            if (edgeTo[v] != null)
40                spt.addEdge(edgeTo[v]);
41
42        EdgeweightedDirectedCycle finder = new
43        EdgeweightedDirectedCycle(spt);
44        cycle = finder.cycle();
45    }
46
47    public BellmanFordSPQ1(EdgeweightedDigraph G, int s) {
48        distTo = new double[G.V()];
49        edgeTo = new DirectedEdge[G.V()];
50        queue = new Queue<Integer>();
51        onQ = new boolean[G.V()];
52        Arrays.fill(distTo, Double.POSITIVE_INFINITY);
```

```

51
52     distTo[s] = 0.0;
53     queue.enqueue(s);
54     onQ[s] = true;
55     while (!queue.isEmpty() && !hasNegativeCycle()) {
56         int v = queue.dequeue();
57         onQ[v] = false;
58         relax(G, v);
59     }
60 }
61
62 public boolean hasPathTo(int v) {
63     return distTo[v] < Double.POSITIVE_INFINITY;
64 }
65
66 public double distTo(int v) {
67     return distTo[v];
68 }
69
70 public Iterable<DirectedEdge> pathTo(int v) {
71     if (!hasPathTo(v)) return null;
72     Stack<DirectedEdge> stack = new Stack<DirectedEdge>();
73     for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
74         stack.push(e);
75     return stack;
76 }
77
78 public boolean hasNegativeCycle() {
79     return cycle != null;
80 }
81
82 public Iterable<DirectedEdge> negativeCycle() {
83     return cycle;
84 }
85
86 public static void main(String[] args) {
87     In in = new In(args[0]);
88     int s = Integer.parseInt(args[1]);
89     EdgeWeightedDigraph G = new EdgeWeightedDigraph(in);
90     BellmanFordSPQ1 sp = new BellmanFordSPQ1(G, s);
91
92     if (sp.hasNegativeCycle()) {
93         StdOut.println("Negative Cycle: ");
94         for (DirectedEdge e : sp.negativeCycle())
95             StdOut.print(e + " ");
96         StdOut.println();
97     } else {
98         for (int v = 0; v < G.V(); v++) {
99             if (sp.hasPathTo(v)) {
100                 StdOut.printf("%d to %d (%.2f)\n", s, v,
101                             sp.distTo(v));
102                 for (DirectedEdge e : sp.pathTo(v)) {
103                     StdOut.print(e + " ");
104                 }
105                 StdOut.println();
106             } else {
107                 StdOut.printf("%d to %d          no path\n", s, v);
108             }
109         }
110     }
111 }
```

```

108 }
109 }
110 }
111 }

```

最短路径算法总结：

算法	局限	一般复杂度	最坏复杂度	空间复杂度	优势
Dijkstra算法	边权重必须为正	$E \log V$	$E \log V$	V	最坏情况下仍有较好的性能
拓扑排序	只适用于无环加权有向图	$E + V$	$E + V$	V	是无环图中的最优算法
Bellman-Ford算法	不能存在负权重环	$E + V$	VE	V	适用领域广泛

5. 字符串

5.1 字符串排序

5.1.1 键索引计数排序

键索引计数法实质上是一种排序方法，适用于小整数键的简单排序。执行4个步骤如下：

1. 使用一个int数组count[]计算出每一个键出现的频率；
2. 使用count[]来计算每一个键在排序结果中的起始索引位置；
3. 将原来数组中的元素移动到一个辅助数组aux[]之中进行排序，且每一个元素在aux[]中的位置是由它的键对应的count[]（此时count[i]表示该键当前在辅助数组中的起始下标位置）值决定；
4. 将在aux[]数组中排序好的数据回写到原来的数组之中。

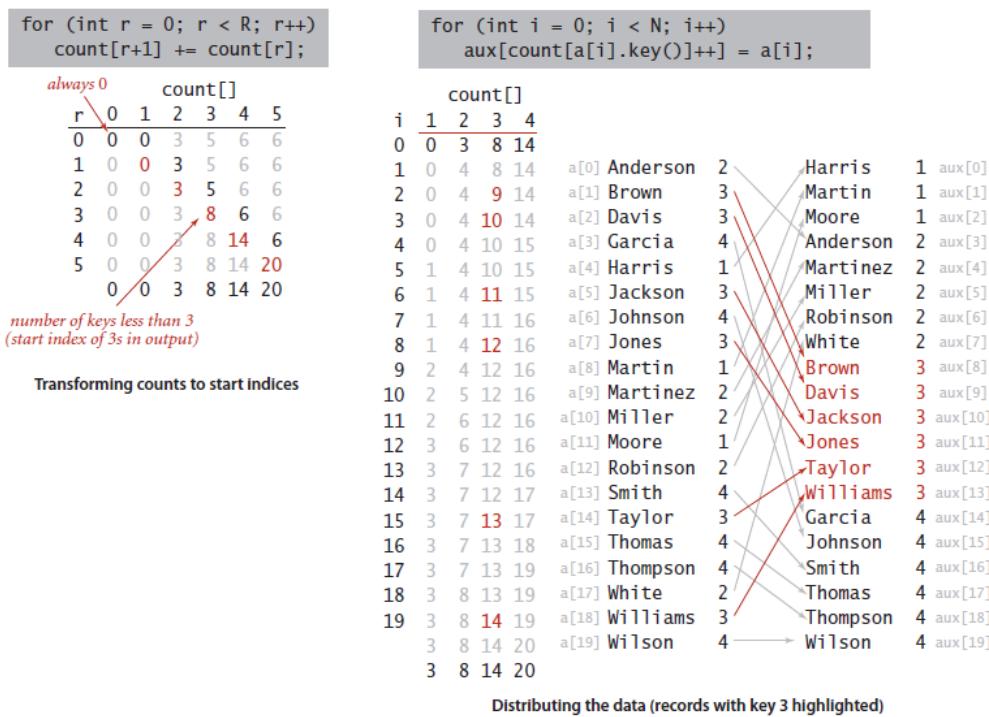
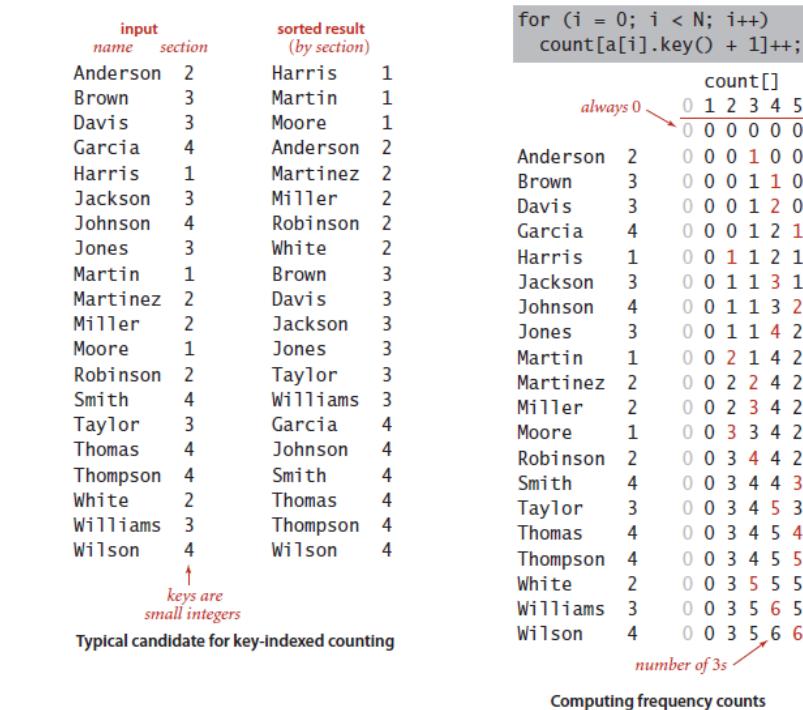
代码如下所示：

```

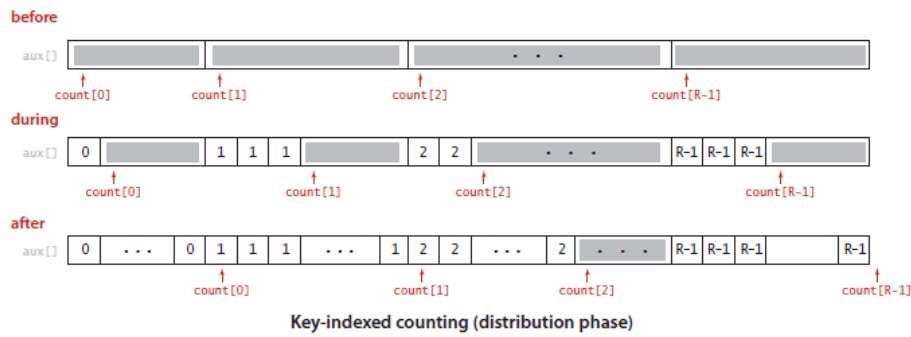
1 int N = a.length;
2
3 String[] aux = new String[N];
4 int[] count = new int[R + 1];
5
6 //计算键出现的频率
7 for (int i = 0; i < N; i++)
8     count[a[i].key() + 1]++;
9 //将频率转换为索引
10 for (int r = 0; r < R; r++)
11     count[r + 1] += count[r];
12 //将元素分类
13 for (int i = 0; i < N; i++)
14     aux[count[a[i].key()]++] = a[i];
15 //回写
16 for (int i = 0; i < N; i++)

```

下图演示了一组名字字符串按照组号（键）进行从小到大排序的过程：



而下面的过程演示了从原数组按照count[]移动到辅助数组的过程：



5.1.2 低位优先的字符串排序

低位优先的字符串排序实际上就是针对字符串数组从它们的低位开始到高位进行字符串长度次的键索引计数排序。

```

1 import edu.princeton.cs.algs4.StdOut;
2
3 public class LSD {
4     public static void sort(String[] a, int w) {
5         int N = a.length;
6         int R = 256;
7         String[] aux = new String[N];
8
9         for (int d = w - 1; d >= 0; d--) {
10             int[] count = new int[R + 1];
11             for (int i = 0; i < N; i++)
12                 count[a[i].charAt(d) + 1]++;
13             //count[a[i][d]+1]++;
14             //count[i+1]=n表示字符码为i的字符出现了n次
15
16             for (int r = 0; r < R; r++)
17                 count[r + 1] += count[r];
18             //count[r+1]+=count[r];
19             //此时count[i]表示字符码为i的字符要移动到辅助数组中的下标位置
20
21             for (int i = 0; i < N; i++)
22                 aux[count[a[i].charAt(d)]++] = a[i];
23             //aux[count[a[i][d]]++]=a[i]
24
25             for (int i = 0; i < N; i++)
26                 a[i] = aux[i];
27         }
28     }
29
30     public static void main(String[] args) {
31         String[] a = new String[]{
32             "4PGC938", "2IYE230",
33             "3CIO720", "1ICK750",
34             "1OHV845", "4JZY524"
35         };
36
37         for (String str : a)
38             StdOut.println(str);
39     }
40 }
```

C++实现：

```

1 #include <algorithm>
2 #include <cstring>
3 #include <iostream>
4 #include <string>
5 #include <vector>
6 using namespace std;
7
8 const vector<string> &
9 strsort(vector<string> &strvec, int w) {
10     size_t N = strvec.size(), R = 256;
11     vector<string> aux(N);
12     int *count = new int[R + 1];
13
14     for (int d = w - 1; d >= 0; --d) {
15         memset(count, 0, sizeof(int) * (R + 1));
16         for (int i = 0; i < N; ++i)
17             count[strvec[i][d] + 1]++;
18         for (int r = 0; r < R; ++r)
19             count[r + 1] += count[r];
20         for (int i = 0; i < N; ++i)
21             aux[count[strvec[i][d]]++] = strvec[i];
22         copy(aux.begin(), aux.end(), strvec.begin());
23     }
24     delete[] count;
25     return strvec;
26 }
27
28 int main() {
29     vector<string> strvec{
30         "4PGC938", "2IYE230",
31         "3CI0720", "1ICK750",
32         "10HV845", "4JZY524"};
33     for (const string &str : strsort(strvec, strvec[0].size()))
34         cout << str << endl;
35     return 0;
36 }
```

图示：

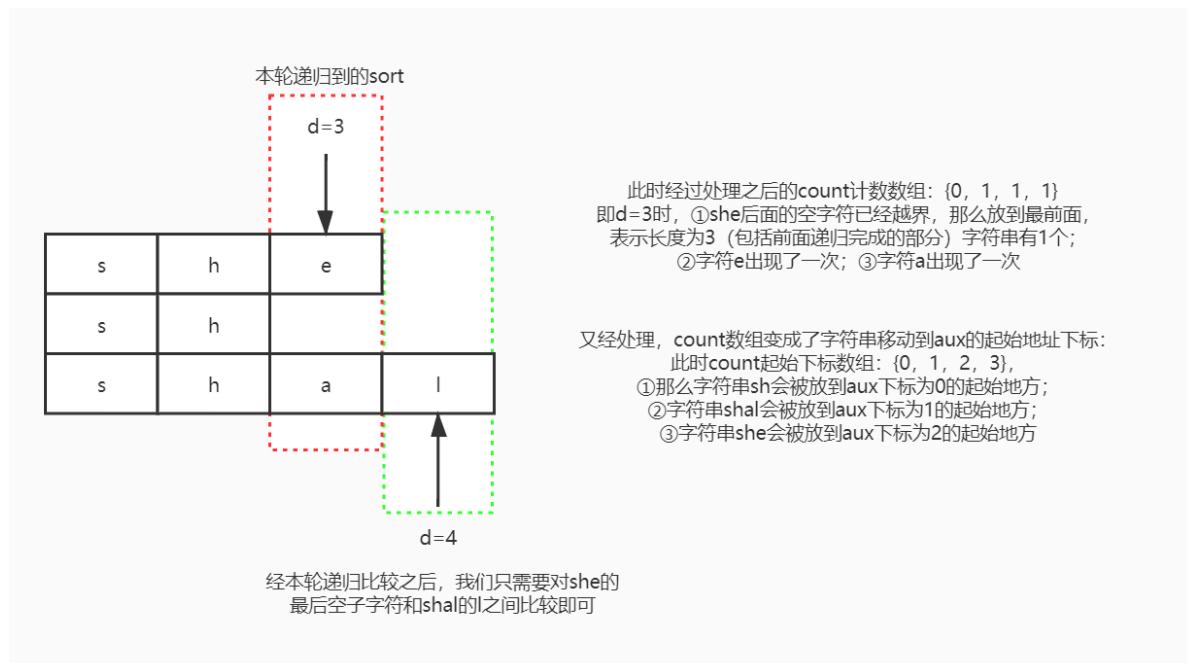
input (W=7)	d=6	d=5	d=4	d=3	d=2	d=1	d=0	output
4PGC938	2IYE230	3CI0720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CI0720	3CI0720	4JZY524	2RLA629	1ICK750	3CI0720	1ICK750	1ICK750
3CI0720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CI0720	10HV845	10HV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	10HV845	1ICK750	10HV845	10HV845
10HV845	3CI0720	2RLA629	3CI0720	1ICK750	10HV845	1ICK750	10HV845	10HV845
4JZY524	3ATW723	2RLA629	3CI0720	1ICK750	10HV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CI0720	3CI0720	4JZY524	2RLA629	2RLA629
3CI0720	10HV845	4PGC938	1ICK750	3CI0720	3CI0720	10HV845	2RLA629	2RLA629
10HV845	10HV845	10HV845	1ICK750	10HV845	2RLA629	10HV845	3ATW723	3ATW723
10HV845	10HV845	10HV845	10HV845	10HV845	2RLA629	10HV845	3CI0720	3CI0720
2RLA629	4PGC938	10HV845	10HV845	10HV845	3ATW723	4PGC938	3CI0720	3CI0720
2RLA629	2RLA629	1ICK750	10HV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

5.1.3 高位优先的字符串排序

高位优先的字符串排序的基本思想很简单：首先用键索引计数法将所有字符串按照首字母排序，然后（递归地）再将每个首字母所对应的子数组排序（忽略首字母，因为每一类中的所有字符串的首字母都是相同的）。

高位优先排序的难点在于处理不同长度的字符串。在这里我们使用的处理方法就是：不断地递归调用sort，而sort每一次都只根据子字符串中第一个字符对字符串数组进行排序，完成之后对首字符相同的字符串数组部分根据它们的首字符递归调用sort进行排序，直到遇到这个起始字符后面的子字符串数组为空。

如果字符串之间长度不同，那么随着递归的推进，最终总会有些字符串在sort函数中呈现出的子字符串为空（实际上，我们用字符下标 \geq 字符串长度时，表示检测到子字符串为空），那么显然这些字符串应该自动地放到源字符串数组的前面位置。接着再对剩下的子字符串进行递归比较。如图所示：



```
1 import edu.princeton.cs.algs4.Alphabet;
2 import edu.princeton.cs.algs4.Stdout;
3
4 public class MSD {
5     private static int R = Alphabet.EXTENDED_ASCII.R(); //256
6     private static final int M = 10;
7     private static String[] aux;
8
9     private static void swap(String[] a, int i, int j) {
10        String temp = a[i];
11        a[i] = a[j];
12        a[j] = temp;
13    }
14
15    /* 若下标d已经超过了str的最大下标, 则返回-1, 这样count[1]这个位置
16       就可以记录在本轮不再比较的字符串数量。它是String charAt方法的改造
17       而原来的方法在越界之后会直接抛出一个异常 */
18    private static int charAt(String str, int d) {
19        if (d < str.length()) return str.charAt(d);
20        return -1;
21    }
22
23    private static boolean less(String v, String w, int d) {
```

```

24         return v.substring(d).compareTo(w.substring(d)) < 0;
25     }
26
27     /* 要知道插入排序是可以一次就将字符串数组进行排序，不需要像最高位优先
28      字符串排序那样对每一位字符都进行一次键索引计数排序 */
29     public static void insertion_sort(String[] a, int low, int high, int d)
30     {
31         for (int i = low, j; i <= high; ++i) {
32             String temp = a[i];
33             for (j = i; j > low && less(temp, a[j - 1], d); --j)
34                 a[j] = a[j - 1];
35             a[j] = temp;
36         }
37     }
38
39     public static void msd_sort(String[] a, int low, int high, int d) {
40         if (high - low <= M) {
41             insertion_sort(a, low, high, d);
42             return;
43         }
44         int[] count = new int[R + 2];
45         for (int i = low; i <= high; ++i)
46             count[charAt(a[i], d) + 2]++;
47         for (int r = 0; r < R; ++r)
48             count[r + 1] += count[r];
49         for (int i = low; i <= high; ++i)
50             aux[count[a[i].charAt(d) + 1]++] = a[i];
51         for (int i = low; i <= high; ++i)
52             a[i] = aux[i - low];
53
54         //递归根据下一个字符位置进行比较
55         for (int r = 0; r < R; ++r)
56             msd_sort(a, low + count[r], low + count[r + 1] - 1, d + 1);
57     }
58
59     public static void msd_sort(String[] a) {
60         int N = a.length;
61         aux = new String[N];
62         msd_sort(a, 0, N - 1, 0);
63     }
64
65     public static void main(String[] args) {
66         String[] strarr = new String[]{
67             "she", "shells", "seashells",
68             "by", "what", "how", "code", "look",
69             "the", "the", "are", "surely", "talk",
70             "cheap", "joker", "bubble", "fuck"
71         };
72         msd_sort(strarr);
73         for (String str : strarr)
74             stdout.println(str);
75     }
76 }

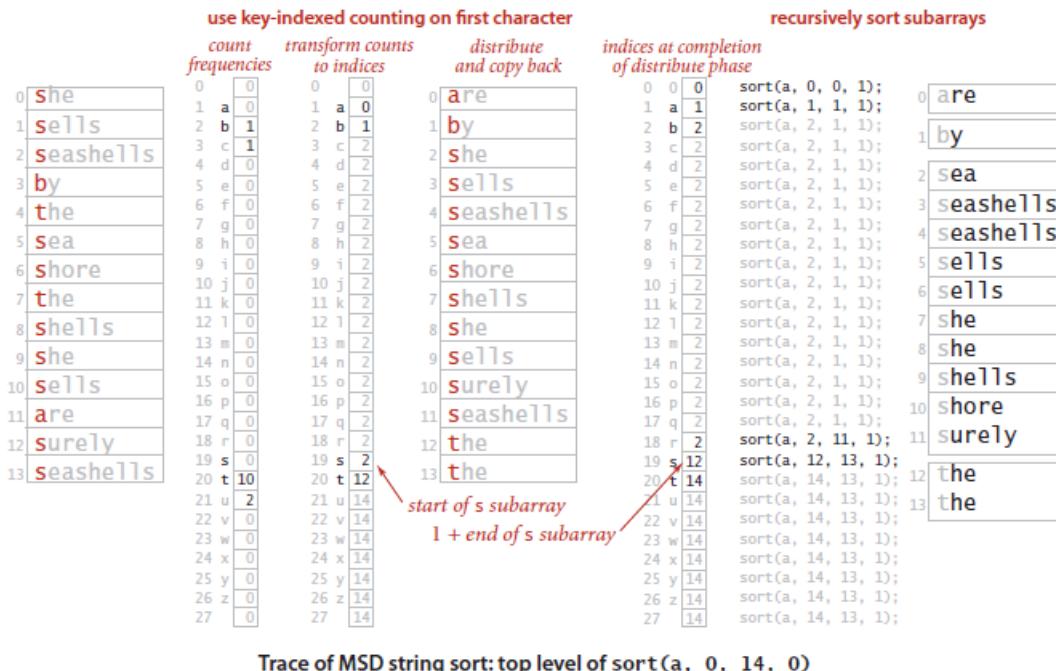
```

这个算法妙就妙在count[]数组的设计上！count[]在不同阶段下其元素表示的意义如下图所示：

表 5.1.1 高位优先的字符串排序中 count[] 数组的意义

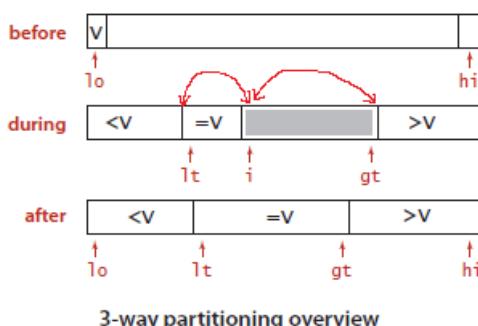
第 d 个字符排序的完成阶段	count[r] 的值				
	r=0	r=1	r 在 2 与 R-1 之间	r=R	r=R+1
频率统计	0 (未使用)	长度为 d 的字符串数量	第 d 个字符的索引值是 r-2 的字符串的数量		
将频率转化为索引	长度为 d 的字符串的子数组的起始索引	第 d 个字符的索引值是 r-1 的字符串的子数组的起始索引		未使用	
数据分类	第 d 个字符的索引值为 r 的字符串的子数组的起始索引		未使用		
	1+ 长度为 d 的字符串的子数组的结束索引	1+ 第 d 个字符串的索引值是 r-1 的字符串的子数组的结束索引		未使用	

下图演示了一次键索引计数排序前后发生的过程：



5.1.4 三向字符串快速排序

三向字符串快速排序的思想和普通的三切分快速排序没什么太大的不同，仍然是将字符串数组切分成3个部分，第一部分放置当前字符小于中枢字符的字符串；第二部分放置当前字符串等于中枢字符的字符串；第三部分放置当前字符大于中枢字符的字符串。我将快速排序中使用过的一张图放置在这里会很好的加深理解：



Java实现：

```

1 import edu.princeton.cs.algs4.StdOut;
2
3 public class Quick3String {

```

```

4     private static int charAt(String s, int d) {
5         if (d < s.length()) return s.charAt(d);
6         return -1;
7     }
8
9     private static void swap(String[] a, int i, int j) {
10        String temp = a[i];
11        a[i] = a[j];
12        a[j] = temp;
13    }
14
15    private static void sort(String[] a, int low, int high, int d) {
16        if (high <= low) return;
17
18        int lt = low, gt = high, i = low + 1;
19        int v = charAt(a[low], d); //中枢字符
20
21        /** i的起始位置
22         *      |
23         *      v      gt
24         * [v,.....]
25         *
26         */
27        while (i <= gt) {
28            int t = charAt(a[i], d);
29            if (t < v) swap(a, lt++, i++);
30            else if (t > v) swap(a, i, gt--);
31            else i++;
32        }
33
34        sort(a, low, lt - 1, d);
35        if (v >= 0) sort(a, lt, gt, d + 1);
36        sort(a, gt + 1, high, d);
37    }
38
39    public static void sort(String[] a) {
40        sort(a, 0, a.length - 1, 0);
41    }
42
43    public static void main(String[] args) {
44        String[] strarr = new String[]{
45            "she", "shells", "seashells",
46            "by", "what", "how", "code", "look",
47            "the", "the", "are", "surely", "talk",
48            "cheap", "joker", "bubble", "fuck"
49        };
50        sort(strarr);
51        for (String str : strarr)
52            Stdout.println(str);
53    }
54}

```

C++实现：（其实我更喜欢用C写😊）

```

1 #include <algorithm>
2 #include <iostream>
3 #include <string>

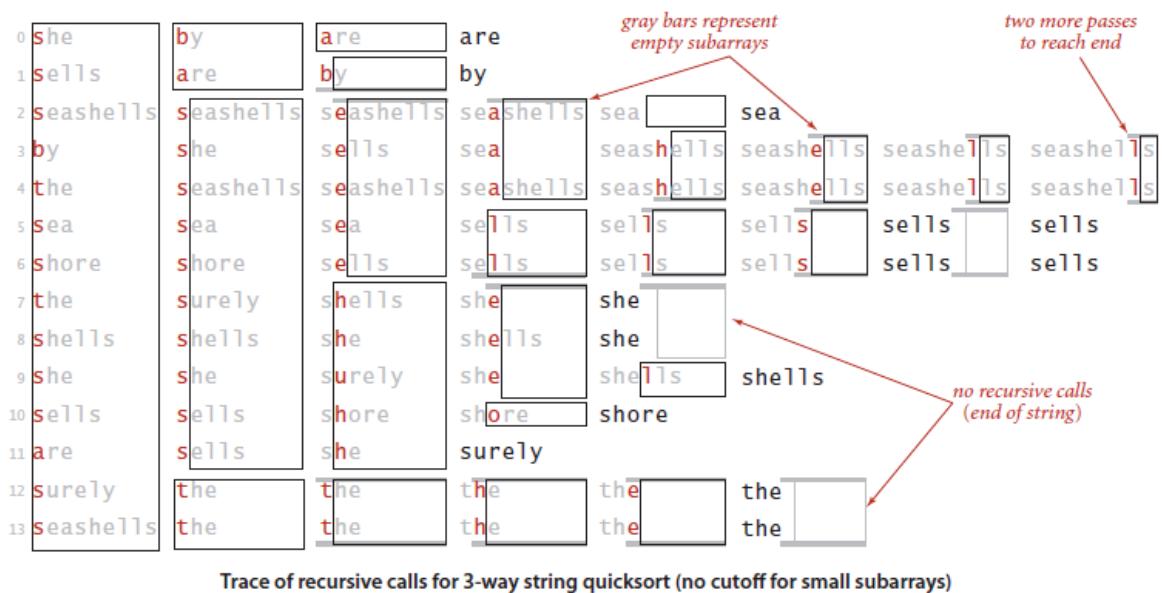
```

```

4 #include <vector>
5 using namespace std;
6
7 class Quick3String {
8 public:
9     static const vector<string> &
10    sort(vector<string> &strvec) {
11        return (sort(strvec, 0, strvec.size() - 1, 0), strvec);
12    }
13
14 private:
15     static void sort(vector<string> &strvec, int low, int high, int d) {
16         if (low >= high)
17             return;
18
19         int lt = low, gt = high, i = low + 1;
20         int key = (d < strvec[low].size()) ? strvec[low][d] : -1;//中枢字符
21
22         while (gt >= i) {
23             if (strvec[i][d] < key)
24                 swap(strvec, lt++, i++);
25             else if (strvec[i][d] > key)
26                 swap(strvec, i, gt--);
27             else
28                 i++;
29         }
30
31         sort(strvec, low, lt - 1, d);
32         if (key >= 0)
33             sort(strvec, lt, gt, d + 1);
34         sort(strvec, gt + 1, high, d);
35     }
36
37     static void swap(vector<string> &strvec, int i, int j) {
38         ::swap(strvec[i], strvec[j]);
39     }
40 };
41
42 int main() {
43     vector<string> strvec{
44         "she", "shells", "seashells",
45         "by", "what", "how", "code", "look",
46         "the", "the", "are", "surely", "talk",
47         "cheap", "joker", "bubble", "fuck"};
48     for (const string &str : Quick3String::sort(strvec))
49         cout << str << endl;
50     return (0);
51 }

```

如下是三向字符串快速排序一次完整的过程图演示：



总结：各种字符串排序算法的性能特点

算法	是否稳定	是否原地排序	时间复杂度	空间复杂度	适用领域
插入排序	✓	✓	$\$N\$ \sim \$N^2\$$	$\$1\$$	小数组或者已经有序的数组
归并排序	✓	✗	$\$N \log^2 N \$$	$\$ \log N \$$	稳定的通用排序算法
快速排序	✗	✓	$\$N \log^2 N \$$	$\$N \$$	通用排序算法，特别适合空间不足的情况
三向快速排序	✗	✓	$\$N \$ \sim \$N \log N \$$	$\$ \log N \$$	大量重复键
低位优先字符串排序	✓	✗	$\$NW\$$	$\$N \$$	较短的定长字符串
高位优先字符串排序	✓	✗	$\$N \$ \sim \$Nw \$$	$\$N + WR \$$	随机字符串
三向字符串字符串排序	✗	✓	$\$N \$ \sim \$Nw \$$	$\$W + \log N \$$	通用排序算法，特别适合含有较长公共前缀的字符串

5.2 单词查找树

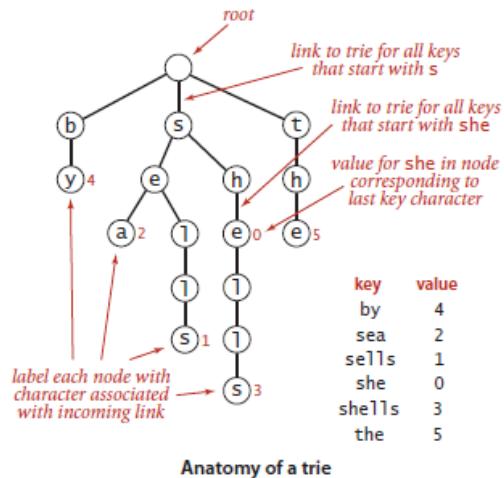
5.2.1 单词查找树

以字符串为键的符号表API: `public class StringST<Value>`

- `StringST()`
- `void put(String key, Value val)`
- `Value get(String key)`
- `void delete(String key)`

- `boolean contains(String key)`
- `boolean isEmpty()`
- `String longestPrefixof(String s)`
- `Iterable<String> keysWithPrefix(String s)`
- `Iterable<String> keysThatMatch(String s)`
- `int size()`
- `Iterable<String> keys()`

结构图示：



这种数据结构的关键原理在于：让每一个单词的字符挂接在基于字符集的R（拓展ASCII为256）叉树上，若树上的某一个字符所对应的值非null，那么表示该字符与之前路径上所有字符组成的字符串存在，且对应值就是这个字符结点所对应的值。

```

1 import edu.princeton.cs.algs4.Queue;
2 import edu.princeton.cs.algs4.StdOut;
3
4
5 public class Triest<value> {
6     private static int R = 256;
7     private Node root;
8
9     private static class Node {
10         private Object val;
11         private Node[] next = new Node[R];
12     }
13
14     private Node put(Node x, String key, value val, int d) {
15         if (x == null) x = new Node();
16         if (d == key.length()) {
17             x.val = val;
18             return x;
19         }
20
21         char c = key.charAt(d);
22         x.next[c] = put(x.next[c], key, val, d + 1);
23         return x;
24     }
25
26     private Node delete(Node x, String key, int d) {
27         if (x == null) return null;

```

```

28         if (d == key.length())
29             x.val = null;
30         else {
31             //递归查找并删除
32             char c = key.charAt(d);
33             x.next[c] = delete(x.next[c], key, d + 1);
34         }
35
36         if (x.val != null) return x;
37         for (char c = 0; c < R; c++)
38             if (x.next[c] != null) return x;
39         return null;
40     }
41
42     private Node get(Node x, String key, int d) {
43         if (x == null) return null;
44         if (d == key.length()) return x;
45
46         char c = key.charAt(d);
47         return get(x.next[c], key, d + 1);
48     }
49
50     private void collect(Node x, String pre, Queue<String> queue) {
51         if (x == null) return;
52         if (x.val != null) queue.enqueue(pre);
53         for (char c = 0; c < R; c++)
54             collect(x.next[c], pre + c, queue);
55     }
56
57     /* pre表示先前经由单词查找树找到的前缀字符串，只有当这个前缀字符串与
58      * 匹配字符串长度相同且结点值不为null时才表示匹配成功，故将其加入到队列中 */
59     private void collect(Node x, String pre, String pat, Queue<String>
queue) {
60         int d = pre.length();
61         if (x == null) return;
62         if (d == pat.length() && x.val != null)
63             queue.enqueue(pre);
64         if (d == pat.length()) return;
65
66         char next = pat.charAt(d);
67         for (char c = 0; c < R; c++)
68             if (next == '.' || next == c)
69                 collect(x.next[c], pre + c, pat, queue);
70     }
71
72     private int search(Node x, String s, int d, int length) {
73         if (x == null) return length;
74         if (x.val != null) length = d;
75         if (d == s.length()) return length;
76         char c = s.charAt(d);
77         return search(x.next[c], s, d + 1, length);
78     }
79
80     private int size(Node x) {
81         if (x == null) return 0;
82
83         int cnt = 0;
84         if (x.val != null) cnt++;

```

```

85         for (char c = 0; c < R; c++)
86             cnt += size(x.next[c]);
87         return cnt;
88     }
89
90     public int size() {
91         return size(root);
92     }
93
94     public boolean isEmpty() {
95         return size() == 0;
96     }
97
98     public boolean contains(String key) {
99         return get(key) != null;
100    }
101
102    public void put(String key, value val) {
103        root = put(root, key, val, 0);
104    }
105
106    public void delete(String key) {
107        root = delete(root, key, 0);
108    }
109
110    public value get(String key) {
111        Node x = get(root, key, 0);
112        if (x == null) return null;
113        return (value) x.val;
114    }
115
116    public Iterable<String> keysWithPrefix(String pre) {
117        Queue<String> queue = new Queue<String>();
118        collect(get(root, pre, 0), pre, queue);
119        return queue;
120    }
121
122    public Iterable<String> keys() {
123        return keysWithPrefix("");
124    }
125
126    public Iterable<String> keysThatMatch(String pat) {
127        Queue<String> queue = new Queue<String>();
128        collect(root, "", pat, queue);
129        return queue;
130    }
131
132    public String longestPrefixOf(String s) {
133        int length = search(root, s, 0, 0);
134        return s.substring(0, length);
135    }
136
137    public static void main(String[] args) {
138        String[] strarr = new String[]{
139            "sea", "what", "fuck", "show",
140            "how", "good", "code", "dance"
141        };
142        TrieST<Integer> triest = new TrieST<Integer>();

```

```
143
144     for (int i = 0; i < strarr.length; ++i)
145         triest.put(strarr[i], i);
146     StdOut.println(triest.size());
147     for(String str: triest.keys())
148         StdOut.println(str);
149     }
150 }
```

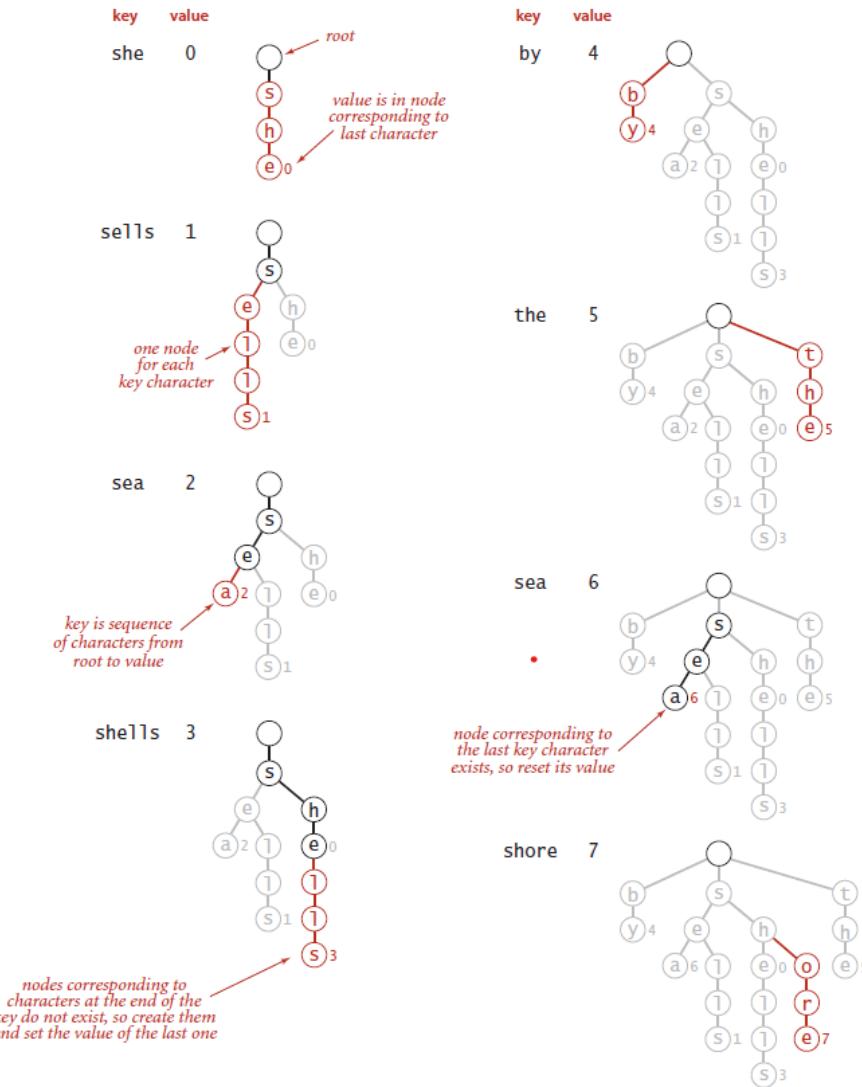
5.2.1.1 相关操作分析

5.2.1.1.1 插入操作

为了达到插入的目的，我们必须解决先前路径上的字符结点不存在的问题。若先前路径上的字符结点不存在，虽然我们不会为这样的结点赋值，但是我还是必须要创建这样的结点。再创建之后必须检查字符的长度是否与指定插入键字符串的长度相同，若是则将值赋予；否则递归处理到下一个结点（若先前的结点是新建的，那么下一个结点也必然需要新创建）。

```
1 private Node put(Node x, String key, value val, int d) {
2     if (x == null) x = new Node(); //若当前字符结点不存在，则新建
3     if (d == key.length()) {      //若到达了指定的字符结点，则赋值
4         x.val = val;
5         return x;
6     }
7         //否则递归处理到下一个结点再处理
8     char c = key.charAt(d);
9     x.next[c] = put(x.next[c], key, val, d + 1);
10    return x;
11 }
12
13 public void put(string key, value val) {
14     root = put(root, key, val, 0);
15 }
```

下面演示了一个简单单词查找树的构造过程：



Trie construction trace for standard indexing client

5.2.1.1.2 查找操作

查找操作和插入操作类似，也是使用递归，我们可以用下面一个简单的算法指示完成相应的操作：

```

1 NodeType get(NodeType x, Key key):
2     若当前结点 ==null: 返回null
3
4     若当前结点复合要求: 返回当前结点
5     否则 get(x.next[c], key)

```

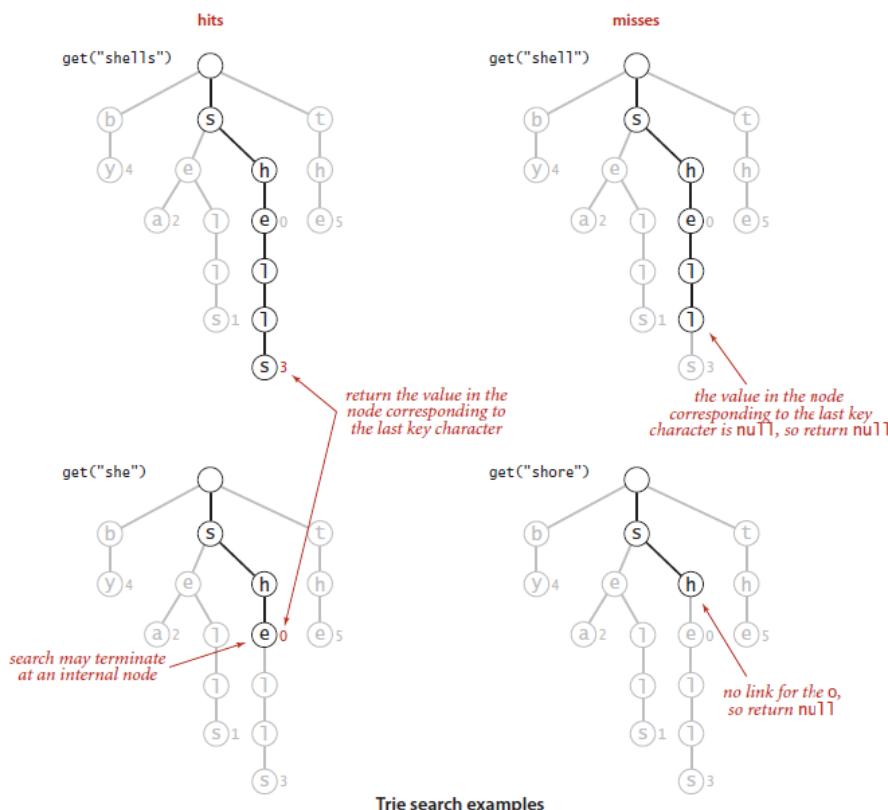
实际真正的代码如下，在这里结点是否符合要求完全依赖的是字符串的长度是否等于先前路径上字符的个数，若相等，我们直接返回这个结点并不管这个结点的值是否为null，是否是null这个工作可以交给下面的封装函数完成。

```

1  private Node get(Node x, String key, int d) {
2      if (x == null) return null;
3      if (d == key.length()) return x;
4
5      char c = key.charAt(d);
6      return get(x.next[c], key, d + 1);
7  }
8
9  public Value get(String key) {
10     Node x = get(root, key, 0);
11     if (x == null) return null;
12     return (Value) x.val;
13 }

```

下面演示了一些实际的查找过程：



5.3.1.1.3 删除操作

删除操作实际上很简单，只需要将单词查找树中对应字符串最后一个字符的结点的值设置为null即可，不过若这个结点在删除之前就没有任何子结点，那么就需要对这个结点本身以及其父字符结点+祖先结点中不再有孩子的结点（且自己本身的值为null）沿路径向上一并删除，这里主要由递归的return返回null起作用。（这里的套路与左倾红黑树中的删除方法相同！）

```

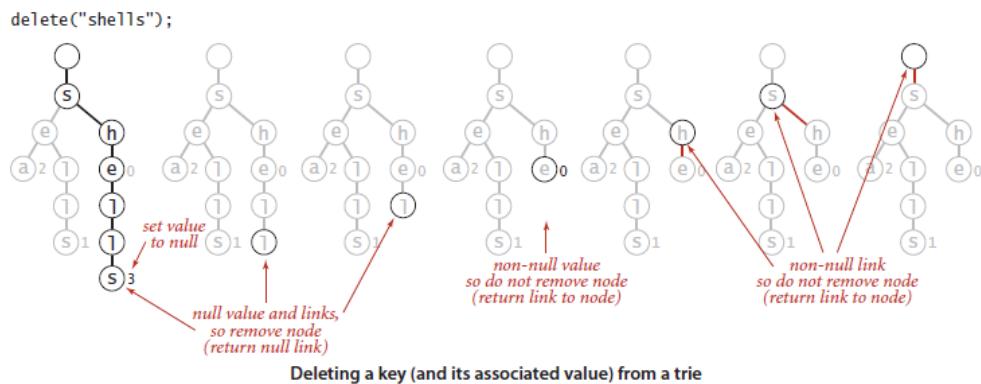
1  private Node delete(Node x, String key, int d) {
2      if (x == null) return null;
3
4      //1、若当前结点即是指定结点，则值置为null
5      if (d == key.length())
6          x.val = null;
7
8      //2、否则，递归查找
9      else {

```

```

9         char c = key.charAt(d);
10        x.next[c] = delete(x.next[c], key, d + 1);
11    }
12
13    /* 置结点值为null后，在返回的路径上对每一个字符结点
14       进行处理。将没有子结点且自己的值为null的结点删除 */
15    if (x.val != null) return x;
16    for (char c = 0; c < R; c++)
17        if (x.next[c] != null) return x;
18    return null;
19}
20
21 public void delete(String key) {
22     root = delete(root, key, 0);
23 }
```

下图展示了一个指定字符串删除的递归前、低轨到指定字符结点将其值置为null以及递归返回的过程中沿路径向上对结点的尾后处理过程：



5.2.1.1.3 返回具有指定前缀键

返回具有指定前缀键的作用就是可以将一个单词查找树中存储的具有指定前缀的字符串加入到一个容器之中，然后将这个容器的引用进行返回。在上述类中对应的方法就是 `keyWithPrefix()`。

为了完成这一目标我们引入了一个收集函数 `collect()`，它是一个类中的私有辅助方法，它可以从指定的字符结点开始将路径上（直到最后的末尾）具有指定前缀的、且存在于单词查找树中的字符串加入到一个容器之中。这里使用的方法仍然是递归！下面的算法指示描述了这个递归方法的范式：

```

1 NodeType collect(NodeType x, Key key, Queue queue):
2     若当前结点 == null: return
3
4     若当前结点值非空：加入这个结点前面路径所有字符组成的字符串到容器之中
5     否则 collect(x.next[c], key, queue)
```

根据上面这个 `collect()` 辅助方法我们可以轻易的让封装函数将单词查找树的根结点 `root` 和前缀字符串、容器传入到这个方法之中，最终通过递归将所有符合的字符串加入到容器之中，从而返回具有指定前缀键的容器引用。

更进一步，若我们向这个函数传入的前缀字符串是一个空字符串""，那么我们就可以让 `keyWithPrefix()` 方法实际返回单词查找树中的所有字符串键！我们将这个方法命名为 `keys()`。

```

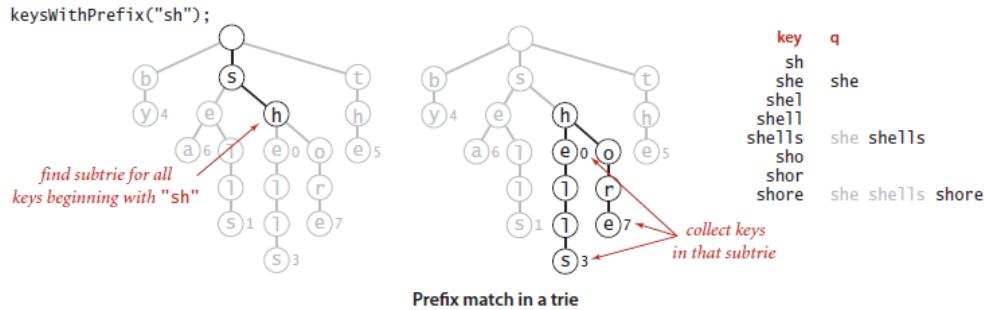
1 private void collect(Node x, String pre, Queue<String> queue) {
2     if (x == null) return;
```

```

3     if (x.val != null) queue.enqueue(pre);
4     for (char c = 0; c < R; c++)
5         collect(x.next[c], pre + c, queue);
6     }
7
8 //返回所有单词查找树中存在的具有指定前缀的字符串键
9 public Iterable<String> keysWithPrefix(String pre) {
10    Queue<String> queue = new Queue<String>();
11    collect(get(root, pre, 0), pre, queue);
12    return queue;
13 }
14
15 //返回所有单词查找树中存在的字符串键
16 public Iterable<String> keys() {
17     return keysWithPrefix("");
18 }

```

下图演示了一次对具有指定前缀“sh”的字符串查找：



5.2.1.1.4 通配符匹配

通配符匹配的思想其实也很简单，类似于上面的查找过程。例如对于一个单词查找树 {"shell", "she", "what"}，我们试图从中找出符合下面通配符字符串，即pattern，“shel..”，那么我们必然知道指定通配符需要查找的字符串长度必然与通配符字符串的长度相同（这里为5）！，显然我们只需要沿着前面“she”字符串前缀找剩下路径中长度为5的字符结点，若这个结点的值非空，那么将其加入到实现安排的容器之中即可，否则不加入。我们可以用如下的伪代码指示这一过程：

```

1 NodeType collect(NodeType x, String pattern, Queue queue):
2     若当前结点为空: return
3
4     若当前结点表示字符串长度与通配模式字符串长度相同 && 结点值非空:
5         将这个字符串加入到容器之中
6     若当前结点表示的字符串长度与通配模式字符串长度相同但结点值为空:
7         return
8     对当前结点中符合要求下一个字符要求的子结点:
9         collect(x.next[c], pattern, queue)

```

具体的代码如下所示：

```

1     private void collect(Node x, String pre, String pat, Queue<String>
queue) {
2         int d = pre.length();
3         if (x == null) return;
4         if (d == pat.length() && x.val != null)
5             queue.enqueue(pre);

```

```

6     if (d == pat.length()) return;
7
8     char next = pat.charAt(d);
9     for (char c = 0; c < R; c++)
10        if (next == '.' || next == c)
11            collect(x.next[c], pre + c, pat, queue);
12    }
13
14    public Iterable<String> keysThatMatch(String pat) {
15        Queue<String> queue = new Queue<String>();
16        collect(root, "", pat, queue);
17        return queue;
18    }

```

5.2.1.15 最长前缀

最长前缀问题指的是给定一个字符串从单词查找树中找出最长的前缀，且这个前缀字符串本身也存储于单词查找树中，一种最特别的情况就是给定字符串本身就是最大前缀，那么可以料想这个单词字符串本身也必然是存储于单词查找树中。

解决这个问题的最好方法仍然是递归，不过递归方法的目的是为了找到那个最长前缀字符串的长度，而不是直接获取最长前缀字符串。这个递归式有如下几种情况需要处理：①若当前结点为空，则返回之前保存的最长前缀字符串长度；②若当前结点非空，则将保存的最长前缀字符串长度修改为当前字符串的长度；③若当前结点的长度已经等于指定字符串的长度，则直接返回这个保存的最长前缀字符串长度；④剩下的情况就是递归了。算法流程如下所示：

```

1 int search(NodeType x, String s, int d, int length):
2     若当前结点为空: return length
3
4     若当前结点值非空: 修改length长度
5     若当前结点表示的字符串长度等于查找字符串的长度: return length
6     return search(x.next[c], s, d, length)

```

```

1 private int search(Node x, String s, int d, int length) {
2     if (x == null) return length;
3     if (x.val != null) length = d;
4     if (d == s.length()) return length;
5     char c = s.charAt(d);
6     return search(x.next[c], s, d + 1, length);
7 }
8
9 public String longestPrefixof(String s) {
10    int length = search(root, s, 0, 0);
11    return s.substring(0, length);
12 }

```

5.2.1.2 单词查找树性质

- 会命中的查找在单词查找树中所访问的字符结点数最多为键的长度+1，而与单词查找树中键的数量无关；
- 不会命中的查找在单词查找树中的随机模型中需要访问 $\sim \log_R N$ (N 表示总随机键数， R 表示所基于的字符数量) 个字符节点数，也就是说查找未命中的成本与键的长度无关；

- 一棵单词查找树中的链接总数在\$RN\$到\$RNw\$之间，其中\$w\$为键的平均长度。这意味着所有键比较短时，链接的总数接近于\$RN\$；当所有键比较长时，链接的总数接近于\$RNw\$。所以缩小\$R\$可以节省大量的空间。

总结就是：**命中查找时间成本与字符串的长度有关，而未命中查找时间成本与单词查找树中的键数量有关；而整个单词查找树的空间成本与键的数量、所基于字符集中的字符数量有关。**

5.2.2 三向单词查找树

三向单词查找树可以认为是R向单词查找树的紧凑表示，其最大的好处在于可以很好的解决R向单词查找树所允许空间严重依赖于字符集中的字符R数量而造成巨大空间需求。在三向单词查找树中，每个结点都含有一个字符、三条链接和一个值。这3条链接分别对应着当前字符小于、等于和大于结点字母的所有键。按照我们R向单词查找树的实现我们可以很好的完成上述数据结构的实现：

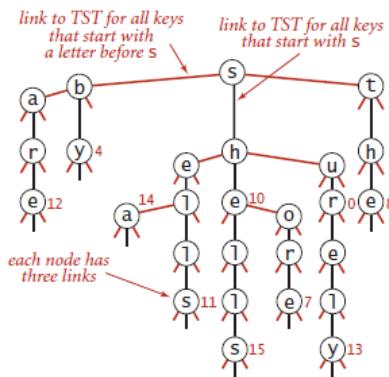
```

1  public class TST<Value> {
2      private Node root;
3
4      private class Node {
5          char c;
6          Node left, mid, right;
7          Value val;
8      }
9
10     private Node get(Node x, String key, int d) {
11         if (x == null) return null;
12
13         char c = key.charAt(d);
14         if (c < x.c)
15             return get(x.left, key, d);
16         else if (c > x.c)
17             return get(x.right, key, d);
18         else if (d < key.length() - 1)
19             return get(x.mid, key, d + 1);
20         else return x;
21     }
22
23     public Value get(String key) {
24         Node x = get(root, key, 0);
25         if (x == null) return null;
26         else return (Value) x.val;
27     }
28
29     private Node put(Node x, String key, Value val, int d) {
30         char c = key.charAt(d);
31         if (x == null) {
32             x = new Node();
33             x.c = c;
34         }
35
36         if (c < x.c)
37             x.left = put(x.left, key, val, d);
38         else if (c > x.c)
39             x.right = put(x.right, key, val, d);
40         else if (d < key.length() - 1)
41             x.mid = put(x.mid, key, val, d + 1);
42     }
43 }
```

```

42         else x.val = val;
43     return x;
44 }
45
46 public void put(String key, Value val) {
47     root = put(root, key, val, 0);
48 }
49 }
```

三向单词查找树的形状如下：



TST representation of a trie

对于三向单词查找树有如下的性质：

- 由\$N\$个平均长度为\$w\$的字符串构造的三向单词查找树中的链接总数在\$3N\$到\$3Nw\$之间；
- 在一棵由\$N\$个随机字符串构造的三向单词查找树中，查找未命中平均需要比较字符\$\sim \ln N\$次。除了\$\sim \ln N\$次外，一次插入或命中的查找会比较一次被查找的键中的每个字符。

各种字符串查找算法的性能比较：

算法(数据结构)	未命中查找检查的字符数量	内存使用	优点
二叉树查找(BST)	$c_1(\lg N)^2$	$64N$	适用于随机排列的键
2-3树查找(红黑树)	$c_2(\lg N)^2$	$64N$	有性能保障
线性探测法(并行数组)	w	$32N \sim 128N$	内置类型，缓存散列值
字典树查找(R向单词查找树)	$\log_R(N)$	$(8R+56)N \sim (8R+56)Nw$	适用于较短的键和较小的字母表
字典树查找(三向单词查找树)	$1.39\lg N$	$64N \sim 64Nw$	适用于非随机的键

5.3 子字符串查找

5.3.1 暴力子字符串查找算法

时间复杂度: \$NM\$

空间复杂度: \$N+M\$

```
1 import edu.princeton.cs.algs4.Stdout;
2
3 public class StrSearch {
4     public static int BFSearch(String txt, String pat) {
5         int M = txt.length(), N = pat.length();
6         for (int i = 0; i <= M - N; ++i) {
7             int j;
8             //并不显式的回溯跟踪文本串的下标
9             for (j = 0; j < N; ++j)
10                 if (txt.charAt(i + j) != pat.charAt(j))
11                     break;
12             if (j == N) return i;
13         }
14         return -1;
15     }
16
17     public static int BFSearch1(String txt, String pat) {
18         int i, M = txt.length();
19         int j, N = pat.length();
20         for (i = 0, j = 0; i < M && j < N; ++i) {
21             if (txt.charAt(i) == pat.charAt(j)) ++j;
22             else {
23                 /* 若未匹配成功，显式回溯跟踪文本串的下标，
24                  并复位跟踪匹配模式串的下标 */
25                 i -= j;
26                 j = 0;
27             }
28         }
29         if (j == N) return i - N;
30         else return -1;
31     }
32
33     public static void main(String[] args) {
34         String txt = "hello world", pat = "world";
35         Stdout.println(BFSearch1(txt, pat));
36         Stdout.println(txt.substring(BFSearch1(txt, pat)));
37     }
38 }
```

5.3.2 KMP字符串查找算法

因为算法4中使用的有限状态自动机的方式来讲解KMP算法，我个人不是很喜欢那种理解方式，所以采用一般算法书籍中常用的“最大公共前后缀计算”的方式来讲解这一内容，并使用C++实现。

在这里推荐一个up主讲解的视频：[KMP字符串匹配算法](#)

时间复杂度: \$N/M\\$ \sim \\$NM\\$

空间复杂度: \$N+M\$

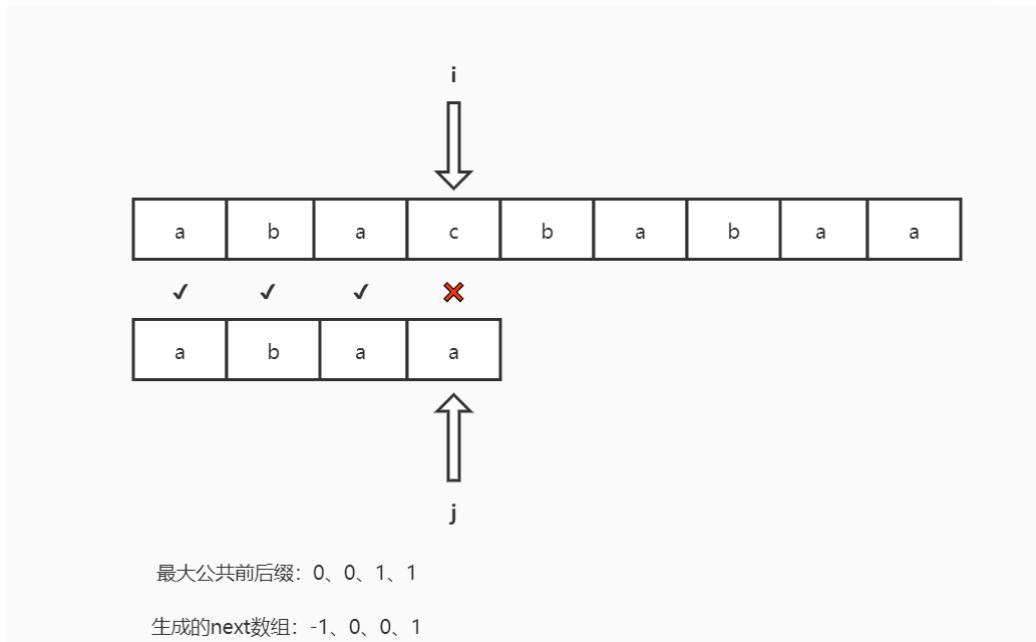
```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;
5
6 //计算模式字符串的最大公共前缀表
7 void prefix_table(const string &pattern, int prefix[], int n) {
8     int len = 0, i = 1;
9     prefix[0] = 0;
10    while (i < n) {
11        if (pattern[i] == pattern[len]) {
12            len++;
13            prefix[i] = len;
14            i++;
15        } else {
16            /* 若当前最大公共前后缀无法虽然i的增加而增长，而退而寻找它之前的
17             * 第二大公共前后缀字符串，若当len为0时也找不到则直接将最大公共
18             * 前后缀数组prefix中的当前位置设置为0即可 */
19            if (len > 0)
20                len = prefix[len - 1];
21            else {
22                prefix[i] = len;
23                i++;
24            }
25        }
26    }
27 }
28
29 //向后移动一位最大公共前缀表，并将第0位设置为-1
30 void move_prefix(int prefix[], int n) {
31     for (int i = n - 1; i > 0; --i)
32         prefix[i] = prefix[i - 1];
33     prefix[0] = -1;
34 }
35
36 //kmp算法执行子字符串查找
37 int kmp_search(const string &txt, const string &pat) {
38     int i, M = txt.size();
39     int j, N = pat.size();
40     int *next = new int[N];
41     prefix_table(pat, next, N);
42     move_prefix(next, N);
43
44     for (i = 0, j = 0; i < M && j < N;) {
45         if (txt[i] == pat[j]) {
46             i++;
47             j++;
48         } else {
49             j = next[j];
50             if (j == -1) {
51                 j = 0;
52                 i++;
53             }
54         }
55     }
56 }
```

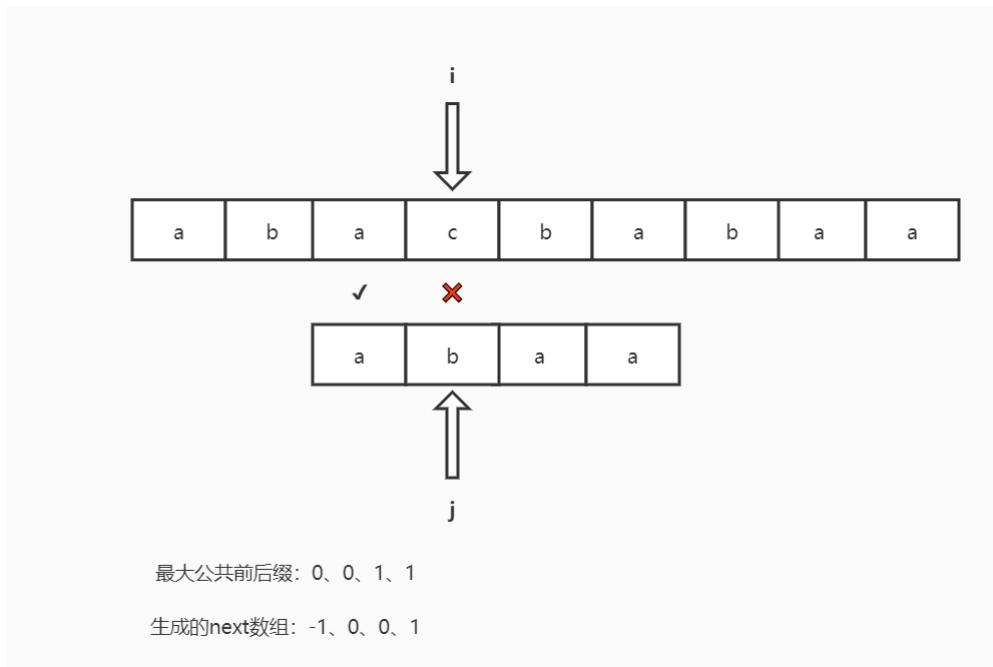
```

56     delete[] next;
57     if (j == N)
58         return i - N;
59     else
60         return -1;
61 }
62
63 int main() {
64     string txt("hello world"), pat("world");
65     cout << kmp_search(txt, pat) << endl;
66     cout << txt.substr(kmp_search(txt, pat), pat.size()) << endl;
67     return 0;
68 }
```

5.3.2.1 最大公共前后缀

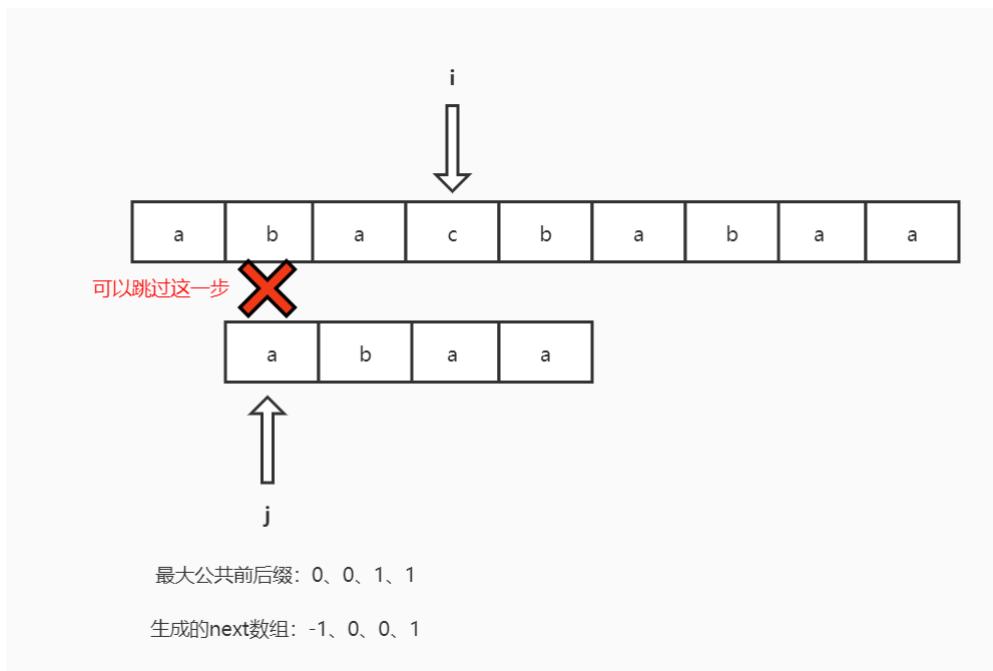
众所周知，KMP算法的核心就是避免在文本串中的下标（指向当前待判的字符）在与模式字符串发生比较失败的时候进行回溯，防止不必要的重复比较。下面演示了KMP使用最大公共前后缀数组生成的next数组进行匹配时发生的前两步：



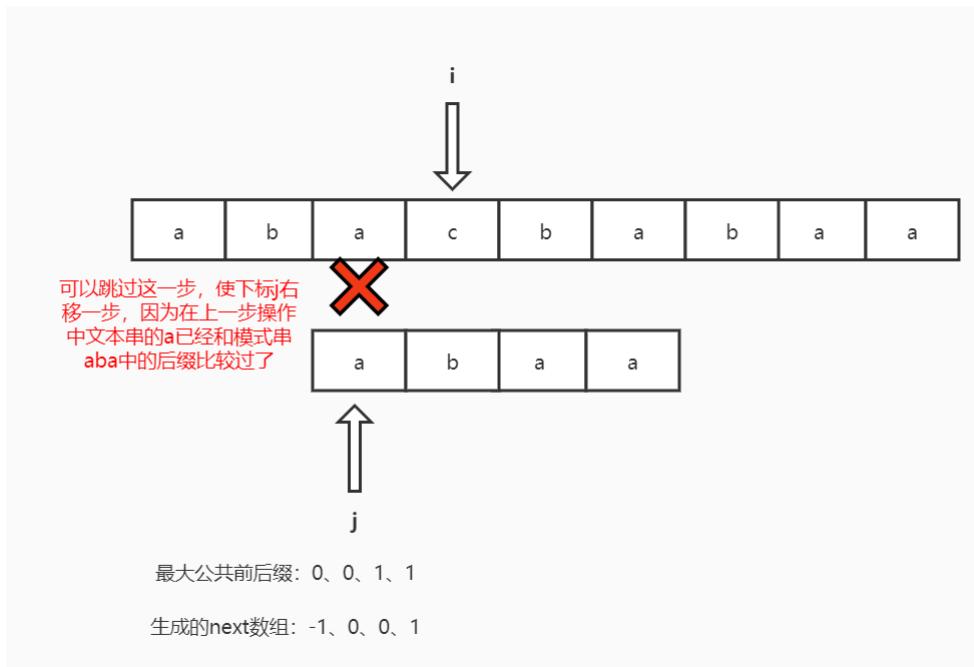


从上面的kmp匹配的过程中你可以看到文本串中的i在匹配的过程中并没有像暴力排序那样发生指针回溯的过程，其中两个最主要与BF算法不同的两个细节在于：①模式字符串pattern跳过了对文本串txt第2、3个字符开始的匹配；②模式字符串pattern中的下标j移动到了一个另一个位置。

对于第一个细节，如此操作的理由在于：由于模式字符串和上面的文本字符串在最后一个a字符前面的部分已经执行过了比较，我们知道下面abaa中前三个字符aba与上面的abac中的aba相同，而模式字符串第一个字符a却与后面的b不相同，因此我们显然可以知道若按照暴力匹配，下一步的abaa与bacb比较必然在第一个字符就会失败。因此我们可以直接跳过这个字符，让abaa与acba进行比较。



对于第二个细节，如此操作的理由在于：由于根据上面的推断abaa现在应该让文本串中的acba进行比较，但是由于aba这个字符串有着最大公共前后缀“a”。因此上一步的abaa与abac比较已经可以让我们知道前缀“a”和acba的第一个字符a的比较也是可以跳过的！所以我们可以直接将j设置成最大前后缀后边字符的下标，即：j=len(当前j前面字符串的最大前后缀)。



由上我们可以知道只要，只要我们计算出一个模式字符串的最大公共前后缀数组，我们可以使得在子字符串匹配的过程中只发生模式字符串中的下标j回退而不发生文本字符串下标i的回退。而真正绝对到底回退到那个模式字符串数组中的哪一个位实际上取决于由最大公共前后缀数组prefix_table生成的next数组。其算法伪代码大致如下：

```

1 def kmp(txt, pattern):
2     根据pattern生成最大公共前后缀数组prefix
3     进而由prefix生成next数组
4
5     for i < txt.len && j < pattern.len:
6         if 当前字符与模式串中的字符相同:
7             i++, j++
8             #当前字符没有成功，试试模式串中最大公共前后缀后面开始的字符
9             #我们规定若next[j]等于-1表示这个当前字符串前面的部分没有
10            #最大公共前后缀，此时直接开始试文本串中下一个字符串，j复位
11            else:
12                j = next[j]
13                if j == -1:
14                    i++, j = 0
15
16            if j == pattern.len:
17                return i - pattern.len
18            else return -1;

```

5.3.2.2 计算最大公共前后缀

计算最大公共前后缀需要用到动态规划的思想。通过下面逐步计算最大公共前后缀数组的过程可以发现：当前最大公共前后缀的值可以通过前一个子字符串的结果进一步计算得到。

例如①“aba”的下一个最大前缀前后缀值若想成为2，那么只需要比较位置为1的b和当前字符（aba后面的那个字符）是否相等，若相等则将前一个最大公共前后缀的值+1放置到当前位置；②否则用前面字符串的第二长公共前后缀字符串后的第一个字符与当前字符比较，若相同用这个第二公共前后缀字符串长度+1放置到当前prefix数组位置。例如“abacdabab”需要计算当前位置的prefix值，而“abacdaba”的最大公共前后缀为“aba”，既然当前的“b”与“aba”后面的“c”不相同，那么当前最大公共前后缀显然不能+1。只能退而求其次看看“abacdaba”的第二公共前后缀“a”后面的b是否与当前字符相同，显然这里是相

同的，所以我们可以用第二公共前后缀长度 $l+1$ 放置到当前的prefix之中。如果上面的都不成立，则继续比较第3大公共前后缀。。。直到没有子前后缀就从头比较。

a	b	a	c	d	a	b	a	b
0	a							
0	ab							
1	aba							
0	abac							
0	abacd							
1	abacda							
2	abacdab							
3	abacdaba							
2	abacdabab							

我们可以用下面的伪代码来描述这一算法：

```
1 def prefix(str,prefixArr):
2     prefixArr[0]
3     当前最大公共前后缀len=0
4
5     while i < len(str):
6         if str[len] == str[i]:
7             将当前prefixArr的最大公共前后缀值设置为len+1
8             增加len、i
9         else:
10            将len设置前面字符串的次公共前后缀长度
11            if 前面已经没有次公共前后缀:
12                prefixArr[i++] = 0
```

使用C++语言对此进行实现：

```
1 void prefix_table(const string &pattern, int prefix[], int n) {
2     int len = 0, i = 1;
3     prefix[0] = 0;
4
5     while (i < n) {
6         if (pattern[i] == pattern[len]) {
7             len++;
8             prefix[i] = len;
9             i++;
10        }
11        else {
12            if (len > 0)
13                len = prefix[len - 1];
14            else
15                prefix[i++] = len;
16        }
17    }
```

5.3.2.3 计算next数组

只需要对上面的prefix数组每一个元素右移一位即可，不过第一个元素需要设置为-1，表示当前模式字符串第一个字符和文本字符串当前字符不相同，此时需要对文本字符串中的下标*i*+1，使得游标向前进一格。

5.3.3 Boyer-Moore字符串查找算法

Boyer-Moore字符串查找算法的核心就是**基于模式串的字符集合生成一个字符不匹配时的跳跃表（辅助数组）**，这个表可以告诉程序在字符发生不匹配时文本串的下标该跳跃到哪一个位置，然后再开始从右向左的匹配（注意：Boyer-Moore算法是从右向左开始匹配的）。虽然程序看起来文本串的下标*i*一直都是处在跳跃的状态，但实际上由于文本串中的字符是基于*i+j*进行匹配的，所以Boyer-Moore算法是存在文本串字符重复比较（即真正的回溯）的现象的。

下面是以前不好的甚至是错误的观点：

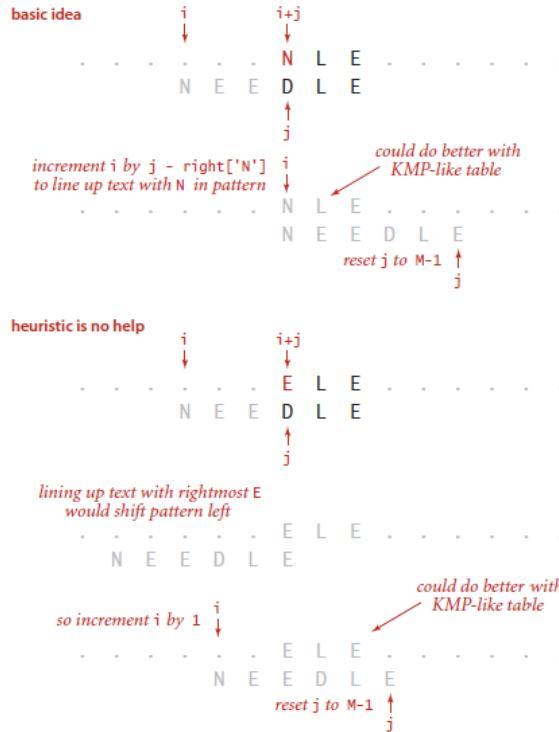
与KMP算法类似，这些算法都是**基于模式字符串中生成一个辅助数组**（KMP中是next数组，告诉模式字符串在不匹配的时候下标该跳到哪里；对于Boyer-Moore数组指的是跳跃表，它用来告诉文本串在发生不匹配的时候该跳到哪个位置），**并在发生字符不匹配的时候避免文本串下标回溯**。两者的算法核心的不同在于：**KMP利用的是模式串最大公共前后缀有无的特点，而Boyer-Moore利用的是当前未匹配成功的字符是否存在于模式字符串这一特点**。其表现形式的不同在于：KMP中下标跳跃的模式串中的下标（向前跳），而在Boyer-Moore算法中跳跃的是文本串中的下标（向后跳）。

5.3.3.1 跳跃的过程

该算法的另一个特点在于它的比较过程是从右向左的，而不是从左向右的！在比较的过程中，若当前文本串中的字符和模式串中的字符不同，则需要通过如下的处理让文本串中的下标进行跳跃式步进，而模式串中的下标也可能发生更新：

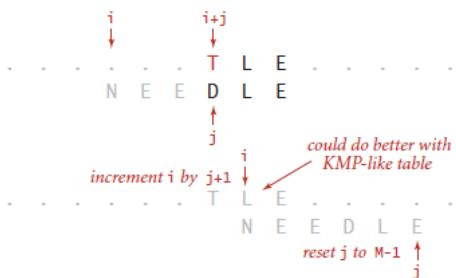
1. 若当前匹配文本串字符存在于模式串的字符集合中，且模式串当前位置-跳跃表查询结果>0，则让文本串下标增大这个差值；
2. 若当前匹配文本串字符存在于模式串的字符集合中，但模式串当前位置-跳跃表查询结果<0，则让文本串下标递增1步；
3. 若当前匹配文本串字符不存在于模式串的字符集合中，故跳跃表查询结果为-1，则让文本串下标递增1步。

下图展示了当前字符没有匹配成功，但该字符在模式串的字符集合之中，此时本文串下标跳跃的情况：



Mismatched character heuristic (mismatch in pattern)

下图展示了当前字符没有匹配成功，且该字符不在模式串的字符集合之中，此时的文本串下标跳跃情况：



Mismatched character heuristic (mismatch not in pattern)

5.3.3.2 跳跃表的计算

跳跃表其实就是一个字符集合数组，若模式字符串出现了哪一个字符（遍历时最后出现）就在相应的额下标位置上记录该字符在模式字符串中的下标即可。若没有出现则在数组中初始化-1即可。

java代码实现：

```

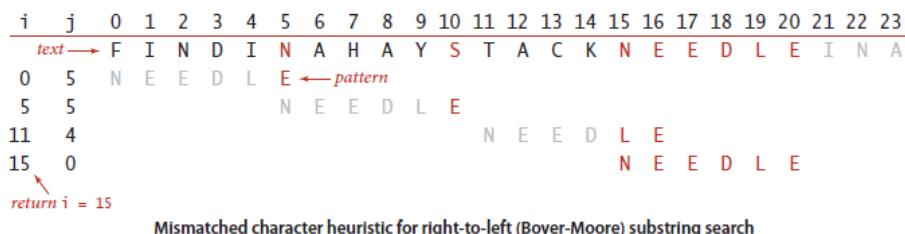
1 import edu.princeton.cs.algs4.StdOut;
2 import java.util.Arrays;
3
4 public class BoyerMoore {
5     private int[] right;
6     private String pat;
7
8     //计算跳跃表
9     BoyerMoore(String pat) {
10         this.pat = pat;
11         int M = pat.length();
12         int R = 256;
```

```

13     right = new int[R];
14     Arrays.fill(right, -1);
15     for (int j = 0; j < M; j++)
16         right[pat.charAt(j)] = j;
17 }
18
19 public int search(String txt) {
20     int M = txt.length();
21     int N = pat.length();
22     int skip;//文本串需要向右步进的量
23
24     for (int i = 0; i <= M - N; i += skip) {
25         skip = 0;
26         for (int j = N - 1; j >= 0; j--)
27             /* 若不匹配，则skip会被设置为正整数
28             * 1. 一种情况是当前匹配的文本串字符存在于模式串字符集合中,
29             *      且模式串当前位置-跳跃表查询得到的值>0;
30             * 2. 一种情况是当前匹配的文本串字符存在于模式串字符集合中,
31             *      但模式串当前位置-跳跃表查询得到的值<0;
32             * 3. 一种情况是当前匹配的文本串字符不存在于模式串字符集合中,
33             *      则查询跳跃表的结果<0。
34             */
35         if (pat.charAt(j) != txt.charAt(i + j)) {
36             skip = j - right[txt.charAt(i + j)];
37             if (skip < 1) skip = 1;
38             break;
39         }
40         if (skip == 0) return i;
41     }
42     return -1;
43 }
44
45 public static void main(String[] args) {
46     String txt = "hello world", pat = "world";
47     BoyerMoore boyerMoore = new BoyerMoore(pat);
48     int b = boyerMoore.search(txt);
49     StdOut.println(b);
50     if (b != -1) StdOut.println(txt.substring(b));
51 }
52 }
53

```

下面展示了一次完整的匹配过程：



C++实现：

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;

```

```

5
6 static void jump_table(const string &pat, vector<int> &jump) {
7     fill(jump.begin(), jump.end(), -1);
8     for (int i = 0; i < pat.length(); ++i)
9         jump[i] = static_cast<int>(pat[i]);
10 }
11
12 int BoyerMooreSearch(const string &txt, const string &pat) {
13     int M = txt.size(), N = pat.size();
14     vector<int> jump(256);
15     int skip;
16
17     jump_table(pat, jump);
18     for (int i = 0; i < M - N; i += skip) {
19         skip = 0;
20         for (int j = N - 1; j >= 0; --j)
21             if (txt[i + j] != pat[j]) {
22                 skip = jump[txt[i + j]];
23                 if (skip < 1)
24                     skip = 1;
25                 break;
26             }
27         if (skip == 0)
28             return i;
29     }
30     return -1;
31 }
32
33 int main() {
34     string txt("talk is cheap, show me the code"), pat("show");
35     int res = BoyerMooreSearch(txt, pat);
36     cout << res << endl;
37     if (res >= 0)
38         cout << txt.substr(res, pat.size());
39
40     return 0;
41 }

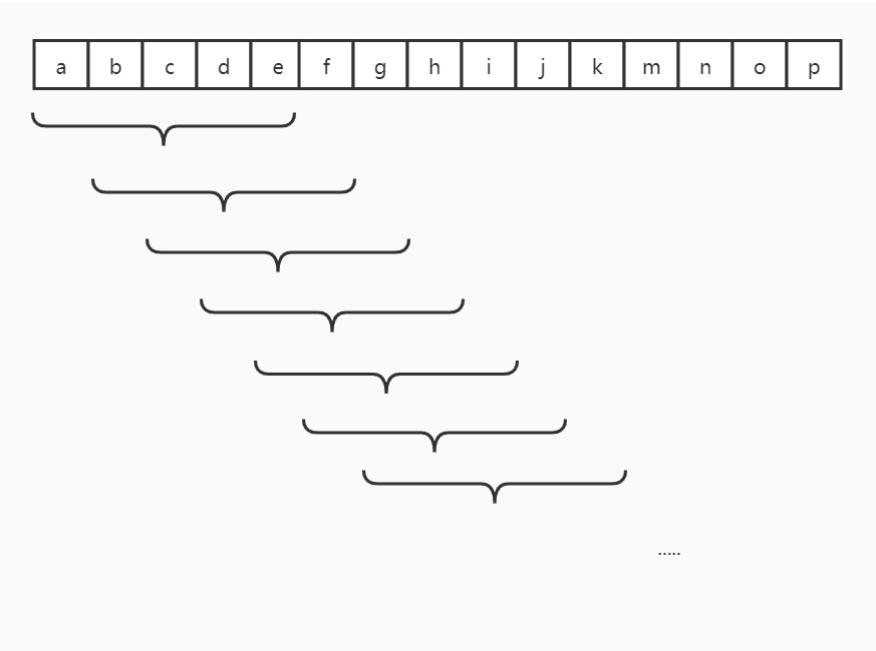
```

5.3.4 Rabin-Karp字符串查找算法

Rabin-Karp算法是一种基于Hash函数的字符串查找算法。其核心思想就是：计算模式字符串的Hash值，然后用相同的Hash函数计算文本中所有可能的M个字符长度的子字符串的Hash值并一一匹配。若匹配成功，则再进行一次检验（可能两两计算使用更大质数得到的Hash值），然后返回对应的子字符串起始下标。

对于Hash值的计算主要是通过Horner算法逐位累加计算：

而对于文本串中的Hash值是通过滑动窗口来计算得到的



我们可以将其中一串子字符串的数值通过如下的表达式进行表达：

而通过滑动窗口得到的下一个字符串的数值表达式为：

因此我们可以从上面推导出来的表达式得知：我们可以从之前计算得到关于 x_i 的hash值，进而通过上述的公式计算下一个子字符串的hash值。

这样我们就可以根据模式字符串的hash和文本串在滑动过程产生的hash值——比较，然后找到那个想要的子字符串的起始下标位置。

```

1 import edu.princeton.cs.algs4.Stdout;
2
3 import java.math.BigInteger;
4 import java.util.Random;
5
6 public class RobinKarp {
7     private long patHash;
8     private int R = 256;
9     private long Q;
10    private int M;
11    private long RM;      //R^(M-1) % Q
12
13    //对Hash值匹配一致的字符串再次执行检验，不过我们这里并没有真正这么做
14    private boolean check(int i) {
15        return true;
16    }
17
18    //计算一个长素数
19    private static long makeprime() {
20        BigInteger prime = new BigInteger(31, new Random());
21        return prime.longValue();
22    }
23
24    //使用Horner方法计算R进制M长字符串的Hash值
25    private long hash(String str, int M) {
26        long h = 0;
27        for (int i = 0; i < M; ++i)
28            h = (R * h + str.charAt(i)) % Q;
29        return h;

```

```

30     }
31
32     RobinKarp(String pat) {
33         M = pat.length();
34         Q = makeprime();
35         RM = 1;
36         for (int i = 1; i < M; ++i)
37             RM = (R * RM) % Q;
38         pathash = hash(pat, M);
39     }
40
41     public int search(String txt) {
42         int N = txt.length();
43         long txthash = hash(txt, M);
44
45         //若txt中的第一个子字符串就是想要找的，那么直接检验后返回
46         if (txthash == pathash && check(0)) return 0;
47         for (int i = M; i < N; ++i) {
48             //计算: [(x_i + R^(M - 1)) * R + t_(i + M)] mode Q
49             txthash = (txthash + Q - RM * txt.charAt(i - M) % Q) % Q;
50             txthash = (txthash * R + txt.charAt(i)) % Q;
51             if (txthash == pathash && check(i - M + 1))
52                 return i - M + 1;
53         }
54         return -1;
55     }
56
57     public static void main(String[] args) {
58         String txt = "hello world", pat = "world";
59         RobinKarp robinKarp = new RobinKarp(pat);
60         int index = robinKarp.search(txt);
61
62         Stdout.println(index);
63         if (index != -1)
64             Stdout.println(txt.substring(index));
65     }
66 }
```

各种子字符串查找算法对比总结：

算法	时间复杂度 (最坏)	时间复杂度 (一般)	空间复杂度	是否存在文本重复比较 (回溯)
暴力匹配	\$MN\$	\$1.1N\$	\$1\$	✓
KMP	\$3N\$	\$1.1N\$	\$M\$	✗
Boyer-Moore	\$MN\$	\$N/M\$	\$R\$	✗
Rabin-Karp	\$7N\$	\$7N\$	\$1\$	✓

其中KMP和Rabin-Karp算法的时间复杂度在线性阶，Boyer-Moore算法的时间复杂度在亚线性阶，但两者都需要额外的辅助空间。

5.4 正则表达式

常见的正则表达式符号：

	或操作
()	括号
*	闭包操作：前面字符串重复0次或多次
+	闭包操作：前面字符串至少重复1次
?	闭包操作：前面字符串重复0次或1次
{n}	闭包操作：前面字符串重复指定n次
{m-n}	闭包操作：前面字符串重复m到n次
.	通配符，表示一个任意字符
[]	指定的字符集合中的任一个
-	范围指示，表示从哪个字符开始到哪个字符结束的范围
^	补集，[^ab]表示不包括a、b的所有字符集合
\	转义符号

NFA代码：

```
1 import edu.princeton.cs.algs4.*;
2
3 public class NFA {
4     private char[] re;
5     private Digraph G;
6     private int M;
7
8     public NFA(String regexp) {
9         Stack<Integer> ops = new Stack<Integer>();
10        re = regexp.toCharArray();
11        M = re.length;
12        G = new Digraph(M + 1);
13
14        for (int i = 0; i < M; i++) {
15            int lp = i;
16            if (re[i] == '(' || re[i] == '|')
17                ops.push(i);
18            else if (re[i] == ')') {
19                int or = ops.pop();
20                if (re[or] == '|') {
21                    lp = ops.pop();
22                    G.addEdge(lp, or + 1);
23                    G.addEdge(or, i);
24                } else lp = or;
25            }
26        }
27    }
28}
```

```

26             if (i < M - 1 && re[i + 1] == '*') {
27                 G.addEdge(lp, i + 1);
28                 G.addEdge(i + 1, lp);
29             }
30             if (re[i] == '(' || re[i] == '*' || re[i] == ')')
31                 G.addEdge(i, i + 1);
32         }
33     }
34
35     public boolean recognizes(String txt) {
36         Bag<Integer> pc = new Bag<Integer>();
37         DirectedDFS dfs = new DirectedDFS(G, 0);
38         for (int v = 0; v < G.V(); v++)
39             if (dfs.marked(v)) pc.add(v);
40
41         for (int i = 0; i < txt.length(); ++i) {
42             Bag<Integer> match = new Bag<Integer>();
43             for (int v : pc)
44                 if (v < M)
45                     if (re[v] == txt.charAt(i) || re[v] == '.')
46                         match.add(v + 1);
47             pc = new Bag<Integer>();
48             dfs = new DirectedDFS(G, match);
49             for (int v = 0; v < G.V(); ++v)
50                 if (dfs.marked(v)) pc.add(v);
51         }
52
53         for (int v : pc) if (v == M) return true;
54         return false;
55     }
56
57     public static void main(String[] args) {
58         String regexp = "(.*" + args[0] + ".*)";
59         NFA nfa = new NFA(regexp);
60         In in = new In(args[1]);
61
62         while (in.hasNextLine()) {
63             String txt = in.readLine();
64             if (nfa.recognizes(txt))
65                 Stdout.println(txt);
66         }
67     }
68 }
```

5.5 数据压缩

5.5.1 读写二进制

将文本中的数据以01形式二进制的形式展示：

```

1 import edu.princeton.cs.algs4.BinaryIn;
2 import edu.princeton.cs.algs4.StdOut;
3
4 public class BinaryDump {
5     public static void main(String[] args) {
```

```

6     int cnt, width = Integer.parseInt(args[0]);
7     BinaryIn bin = new BinaryIn(args[1]);
8
9     for (cnt = 0; !bin.isEmpty(); cnt++) {
10        if (width != 0) {
11            if (cnt != 0 && cnt % width == 0)
12                StdOut.println();
13            if (bin.readBoolean())
14                StdOut.print("1");
15            else StdOut.print("0");
16        } else bin.readBoolean();
17    }
18    StdOut.println();
19    StdOut.println(cnt + " bits");
20}
21}

```

将文本中的数据以十六进制形式输出展示：

```

1 import edu.princeton.cs.algs4.BinaryIn;
2 import edu.princeton.cs.algs4.StdOut;
3
4 public class HexDump {
5     public static void main(String[] args) {
6         int i, bytesPerLine = Integer.parseInt(args[0]);
7         BinaryIn binaryIn = new BinaryIn(args[1]);
8
9         for (i = 0; !binaryIn.isEmpty(); ++i) {
10            if (i == 0) StdOut.print("");
11            else if (i % bytesPerLine == 0)
12                StdOut.println();
13            else
14                StdOut.print(" ");
15
16            char c = binaryIn.readChar();
17            StdOut.printf("%02x", c & 0xff);
18        }
19        StdOut.println("\n" + (i * 8) + " bits");
20    }
21}

```

将文本中的数据以图片形式展示其中的每一个比特位：

```

1 import edu.princeton.cs.algs4.BinaryIn;
2 import edu.princeton.cs.algs4.Picture;
3
4 import java.awt.*;
5
6 public class PictureDump {
7     public static void main(String[] args) {
8         int width = Integer.parseInt(args[0]);
9         int height = Integer.parseInt(args[1]);
10        BinaryIn bin = new BinaryIn(args[2]);
11        Picture picture = new Picture(width, height);
12
13        for (int row = 0; row < height; ++row) {

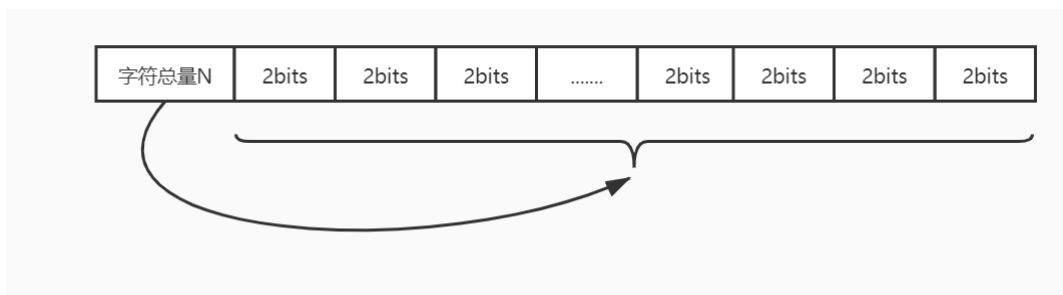
```

```

14         for (int col = 0; col < width; ++col) {
15             if (!bin.isEmpty()) {
16                 boolean bit = bin.readBoolean();
17                 if (bit) picture.set(col, row, Color.BLACK);
18                 else picture.set(col, row, Color.WHITE);
19             } else picture.set(col, row, Color.RED);
20         }
21     }
22     picture.show();
23 }
24 }
```

5.5.1.1 基因数据双位编码

由于生物学中使用ACTG这4个字符来表示DNA中不同的碱基，因此我们对于每一个字符实际上只需要用2个比特就可以实现数据的表示。这样压缩的原理就是：遍历原先表示DNA的字符串，找出每一个字符在ACTG字母表中的下标，然后对这个下标进行2位编码，并在编码后的数据头部加入字符总数以方便解码使用。由于原先每一个字符需要8个比特进行，现在却只需要2个比特就可以完成相同的功能，因此压缩比接近25%。



不过这种编码方式压缩和解码所需要的字母表都是实现两者约定好的，但是在实际的一些情况下可能未必如此，很有可能对方并不知道这些字母表，因此实际编码后的数据中不仅需要指出压缩数据的总量，还需要指出字母表的内容等。

```

1 import edu.princeton.cs.algs4.*;
2
3 public class Genome {
4     public static void compress(BinaryIn bin, BinaryOut bout) {
5         Alphabet DNA = new Alphabet("ACTG");
6         String s = bin.readString();
7         int N = s.length();
8
9         bout.write(N);
10        for (int i = 0; i < N; ++i) {
11            /* 按照DNA的基数对字符在字母表中的下标进行编码，在这里
12             * 基数为2，表示只需要2个比特就可以表示一个原始的字符，
13             * 然后将这2个比特写入到比特流中 */
14            int d = DNA.toIndex(s.charAt(i));
15            bout.write(d, DNA.lgR());
16        }
17        bout.close();
18    }
19
20    public static void expand(BinaryIn bin) {
21        Alphabet DNA = new Alphabet("ACTG");
22        int w = DNA.lgR();
```

```

23     int N = bin.readInt();
24
25     for (int i = 0; i < N; ++i) {
26         //从bin中读入2位比特，然后将其按照DNA字母表的下标返回原始的字符
27         char c = bin.readChar(w);
28         Stdout.print(DNA.toChar(c));
29     }
30 }
31
32 public static void main(String[] args) {
33     BinaryIn bin = new BinaryIn(args[1]);
34     if (args[0].equals("+"))
35         expand(bin);
36     if (args[0].equals("-")) {
37         BinaryOut bout = new BinaryOut(args[2]);
38         compress(bin, bout);
39     }
40 }
41 }
```

5.5.2 游程编码

游程编码的思想非常简单，就是依次记录连续的0或者连续的1的个数，例如000000011111111000，则可以被游程编码压缩成用783（其中这些压缩后的每一个数根据实际情况采用多少位来表示，书中假设用8位bits来表示一个数）。

由于游程编码本身就非常依赖数据文件中存在大量连续0或者连续1事实条件，因此在实际中并不适合文本文件的压缩，因为文本文件出现连续0或者连续1的情况太少了，采用游程编码反而会产生更大的压缩文件。

```

1 import edu.princeton.cs.algs4.BinaryIn;
2 import edu.princeton.cs.algs4.BinaryOut;
3
4 public class RunLength {
5     public static void expand(BinaryIn bin, BinaryOut bout) {
6         boolean b = false;
7         while (!bin.isEmpty()) {
8             char cnt = bin.readChar();
9             for (int i = 0; i < cnt; ++i)
10                 bout.write(b);
11             b = !b;//翻转比特值
12         }
13         bout.close();
14     }
15
16     public static void compress(BinaryIn bin, BinaryOut bout) {
17         char cnt = 0;//相同比特计数
18         for (boolean b, old = false; !bin.isEmpty(); ++cnt) {
19             b = bin.readBoolean();
20             if (b != old) {
21                 bout.write(cnt);
22                 cnt = 0;
23                 old = !old;
24             } else {
25                 if (cnt == 255) {
```

```

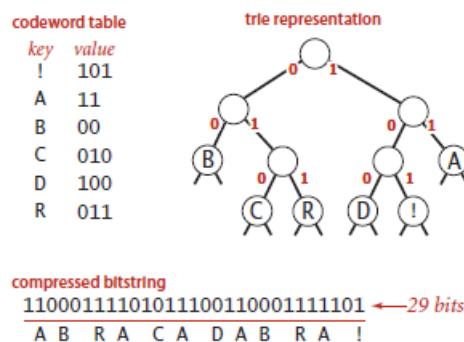
26             bout.write(cnt);
27             cnt = 0;
28         bout.write(cnt);
29     }
30 }
31 bout.write(cnt);
32 bout.close();
33 }
34 }
35
36 public static void main(String[] args) {
37     BinaryIn bin = new BinaryIn(args[1]);
38     BinaryOut bout = new BinaryOut(args[2]);
39     if (args[0].equals("-"))
40         compress(bin, bout);
41     else if (args[0].equals("+"))
42         expand(bin, bout);
43 }
44 }
```

5.5.3 霍夫曼编码

霍夫曼树的核心思想很简单，即：不再使用7位或8位的二进制数表示每一个字符，而是使用较少的比特表示表示频率较高的字符，用较多的比特表示出现频率较低的字符。并使用变长前缀码来避免表示不同字符的编码前缀重复的问题。

5.5.3.1 构建霍夫曼树

实现霍夫曼编码的关键在于根据字符的频率构建霍夫曼树。在霍夫曼树中，出现频率较低的字符所处的叶节点距离树的根结点更远，出现频率较高的字符所处的叶节点距离树的根结点更近，且这些字符绝对不会位于树中间路径上的结点中，这样就可以防止某些字符的编码变成了其他字符编码的前缀。



为了构建霍夫曼树，我们必须先遍历一遍整个文本，将文本中的字符频率进行统计。然后根据这个频率表依次为每一个字符创建一个树节点（即一个森林），每一个树节点记录着表示字符的出现频率，然后加入到一个优先队列之中。然后每一次从优先队列中取出频率最小的两个树进行合并成新的树（同时频率也进行相加），然后重新加入到队列之中，直到优先队列中只剩下一个树。此时这个剩下的树便是霍夫曼树。下面演示了霍夫曼树的构建过程：



```

1  /* 构建霍夫曼树 */
2  private static Node buildTries(int[] freq) {
3      MinPQ<Node> pq = new MinPQ<Node>();
4
5      /* 为频率数组中的每一个（频率）元素构建一群单树节点
6      （或者说是一群树，即森林），并将其加入到优先队列之中 */
7      for (char c = 0; c < R; ++c)
8          if (freq[c] > 0)
9              pq.insert(new Node(c, freq[c], null, null));
10
11     /* 从优先队列中取出两棵树合并成一颗新的树，然后重新加入到
12     优先队列之中。需要注意的是：其中的两棵树的频率也需要合并 */
13     while (pq.size() > 1) {
14         Node x = pq.delMin();
15         Node y = pq.delMin();
16         Node parent = new Node('\0', x.freq + y.freq, x, y);
17         pq.insert(parent);
18     }
19     return pq.delMin();

```

5.5.3.2 构建字符-变长前缀码映射表

为了方便将文本中的字符编码成边长前缀码而不至于为每一个字符去霍夫曼树中查找相应的字符，我们需要为每一个霍夫曼中的字符构建一个字符->变长前缀码的映射表。其实际的操作就是前序遍历，记录每一个到叶节点路径上的编码，然后保存在一个数组之中（我们实际上还是以0101的字符去记录）。

```

1  /* 建立一个字符到二进制编码（仍然以字符串形式保存）的映射表格 */
2  private static void buildCode(String[] st, Node x, String s) {
3      if (x.isLeaf()) {
4          st[x.ch] = s;
5          return;
6      }
7      buildCode(st, x.left, s + '0');
8      buildCode(st, x.right, s + '1');
9  }
10
11 private static String[] buildCode(Node root) {
12     String[] st = new String[R];
13     buildCode(st, root, "");
14     return st;
15 }
```

5.5.3.3 写入/读入霍夫曼树二进制数据

为了能够让解码端能够根据压缩后的变长前缀码集合反向解压出原始的文本数据，霍夫曼压缩算法需要将构建的霍夫曼树一同写入到压缩文件之中，并且放置在文件的头部处。同样的，对于解压的一方，解压算法需要从解压文件中获取霍夫曼树的二进制数据，反向构建出霍夫曼树。

对于霍夫曼树的写入，其所采用的方法为：采用**前序遍历**。每当访问到一个中间路径的树节点就向压缩输出文件写入一个比特0；当它访问当一个叶节点，就会写入一个比特1，紧接着写入叶节点中记录的8位ASCII码。

```

1  /* 将构建的霍夫曼树以二进制数据写入到压缩文件中。写入方式为
2   * 中序遍历，若遍历到的结点为中间路径节点，就写入一个0；若
3   * 遍历到的结点是叶节点，则写入一个1，然后写入对应的字符 */
4  private static void writeTries(Node x, BinaryOut bout) {
5      if (x.isLeaf()) {
6          bout.write(true);
7          bout.write(x.ch);
8          return;
9      }
10     bout.write(false);
11     writeTries(x.left, bout);
12     writeTries(x.right, bout);
13 }
```

对于反向构建霍夫曼树，其所采用的方法为：首先读取一个比特以获知当前树节点的类型，若是1则表示是叶节点，此时就创建一个叶节点然后读取后面8位比特获得字符信息然后返回引用；若是0则表示是中间路径节点，此时就创建一个中间路径节点然后递归的构造它的左子树和右子树。这里的代码还是值得学习的

```
1  /* 从压缩文件中读取出数据并构建霍夫曼树 */
2  private static Node readTries(BinaryIn bin) {
3      /* 碰到1，则表示后面的数据是叶节点的字符数据，  

4       * 此时新建一个叶节点返回给上一层 */
5      if (bin.readBoolean())
6          return new Node(bin.readChar(), 0, null, null);
7      //碰到0，递归处理下一个二进制数
8      return new Node('\0', 0, readTries(bin), readTries(bin));
9  }
```

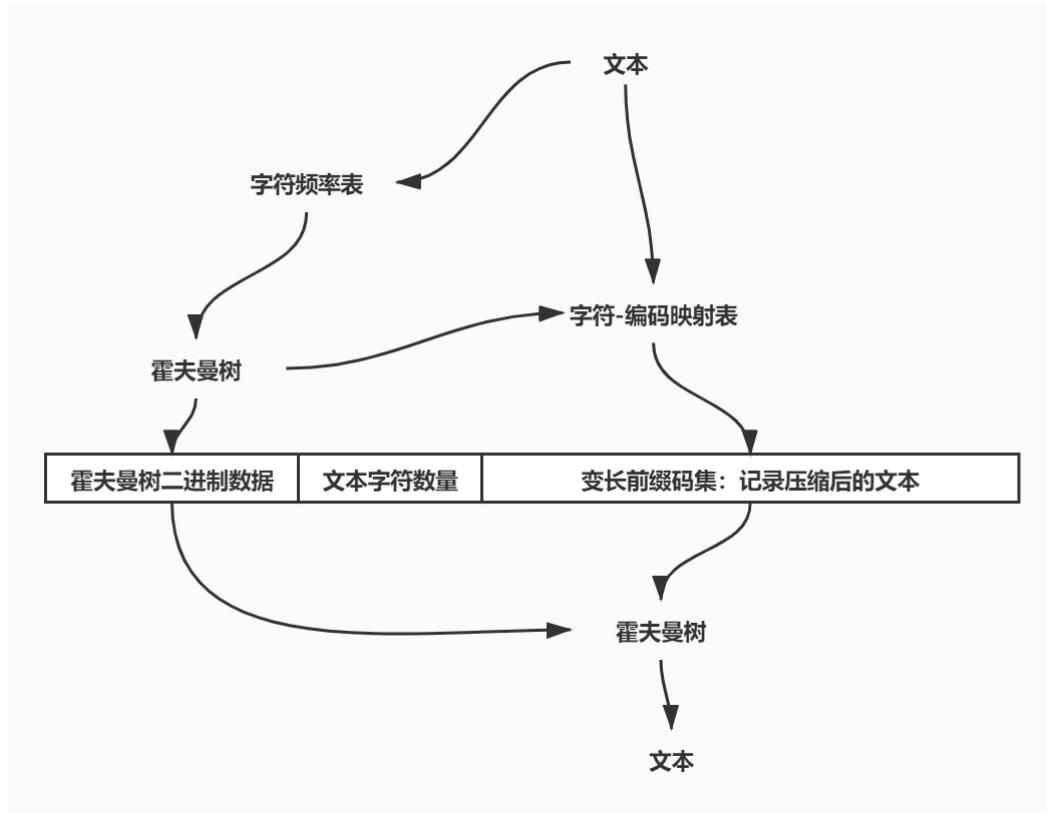
5.5.3.4 压缩/解压算法流程

霍夫曼编码的大致过程：

1. **构建频率表**：遍历文本中的所有字符，记录每一个出现字符的频率
2. **构建霍夫曼树**：遍历文本中所有的字符，根据字符频率使用优先队列构建霍夫曼编码树；
3. **建立字符-编码映射表**：根据上述编码树构建出字符到变长前缀码（仍以字符串形式记录）的对应表；
4. **将霍夫曼树比特化写入到输出文件头部**：对树使用前序遍历，将霍夫曼编码树比特化，方便解压的一方获取编码树信息；
5. **将文本中的字符数量写入到压缩文件**；
6. **为每一个字符执行编码工作**：将变长前缀码写入到压缩文件之中。

霍夫曼解压缩的大致过程：

1. **根据重建霍夫曼树**：根据压缩文件头部信息反向构建霍夫曼编码树；
2. **读取文件字符串长度**；
3. **根据霍夫曼树解码剩余的文件数据**。



```

1 import edu.princeton.cs.algs4.BinaryIn;
2 import edu.princeton.cs.algs4.BinaryOut;
3 import edu.princeton.cs.algs4.MinPQ;
4
5 public class Huffman {
6     private static int R = 256;
7
8     private static class Node implements Comparable<Node> {
9         private char ch;           //当前结点表示字符，若为中间路径结点则直接设置
10        为'\0';
11         private int freq;        //当前字符出现的频率
12         private final Node left, right;
13
14         Node(char ch, int freq, Node left, Node right) {
15             this.ch = ch;
16             this.freq = freq;
17             this.left = left;
18             this.right = right;
19         }
20
21         public boolean isLeaf() {
22             return left == null && right == null;
23         }
24
25         public int compareTo(Node that) {
26             return this.freq - that.freq;
27         }
28
29     /* 构建霍夫曼树 */
30     private static Node buildTries(int[] freq) {
31         MinPQ<Node> pq = new MinPQ<Node>();
32
33         /* 为频率数组中的每一个（频率）元素构建一群单树节点
34          （或者说是一群树，即森林），并将其加入到优先队列之中 */
35         for (char c = 0; c < R; ++c)
36             if (freq[c] > 0)
37                 pq.insert(new Node(c, freq[c], null, null));
38
39         /* 从优先队列中取出两棵树合并成一颗新的树，然后重新加入到
40          优先队列之中。需要注意的是：其中的两棵树的频率也需要合并 */
41         while (pq.size() > 1) {
42             Node x = pq.delMin();
43             Node y = pq.delMin();
44             Node parent = new Node('\0', x.freq + y.freq, x, y);
45             pq.insert(parent);
46         }
47         return pq.delMin();
48     }
49
50     /* 将构建的霍夫曼树以二进制数据写入到压缩文件中。写入方式为
51      * 中序遍历，若遍历到的结点为中间路径节点，就写入一个0；若
52      * 遍历到的结点是叶节点，则写入一个1，然后写入对应的字符 */
53     private static void writeTries(Node x, BinaryOut bout) {
54         if (x.isLeaf()) {
55             bout.write(true);
56             bout.write(x.ch);
57             return;
58         }
59
60         bout.write(false);
61         if (x.left != null)
62             writeTries(x.left, bout);
63         if (x.right != null)
64             writeTries(x.right, bout);
65     }
66
67     public static void main(String[] args) {
68         String s = BinaryIn.readAll();
69         Node root = buildTries(s);
70         writeTries(root, new BinaryOut("huffman.out"));
71     }
72 }
```

```
57     }
58     bout.write(false);
59     writeTries(x.left, bout);
60     writeTries(x.right, bout);
61 }
62
63 /* 从压缩文件中读取出数据并构建霍夫曼树 */
64 private static Node readTries(BinaryIn bin) {
65     /* 碰到1，则表示后面的数据是叶节点的字符数据，  

66      此时新建一个叶节点返回给上一层 */
67     if (bin.readBoolean())
68         return new Node(bin.readChar(), 0, null, null);
69     //碰到0，递归处理下一个二进制数
70     return new Node('\0', 0, readTries(bin), readTries(bin));
71 }
72
73 /* 建立一个字符到二进制编码（仍然以字符串形式保存）的映射表格 */
74 private static void buildCode(String[] st, Node x, String s) {
75     if (x.isLeaf()) {
76         st[x.ch] = s;
77         return;
78     }
79     buildCode(st, x.left, s + '0');
80     buildCode(st, x.right, s + '1');
81 }
82
83 private static String[] buildCode(Node root) {
84     String[] st = new String[R];
85     buildCode(st, root, "");
86     return st;
87 }
88
89
90 /* 压缩 */
91 public static void compress(BinaryIn bin, BinaryOut bout) {
92     String s = bin.readString();
93     char[] input = s.toCharArray();
94     int[] freq = new int[R];
95
96     //1、构建字符出现频率数组
97     for (int i = 0; i < input.length; ++i)
98         freq[input[i]]++;
99     //2、根据频率数组构建霍夫曼树
100    Node root = buildTries(freq);
101    //3、根据霍夫曼树构建字符到二进制编码（该编码还是以字符串的形式记录）的映射表
102    String[] st = new String[R];
103    buildCode(st, root, "");
104
105    //4、先向压缩文件中写入霍夫曼树
106    writeTries(root, bout);
107    //5、再向压缩文件写如实际记录的字符数量
108    bout.write(input.length);
109    //6、最后正式写入由字符翻译成的变长前缀码
110    for (int i = 0; i < input.length; ++i) {
111        String code = st[input[i]];
112        for (int j = 0; j < code.length(); ++j) {
113            if (code.charAt(j) == '1')
114                bout.write(true);
```

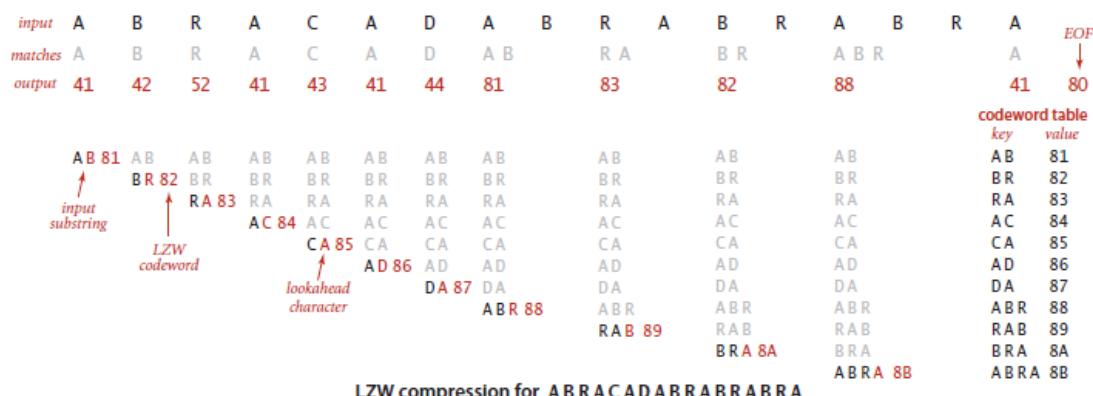
```

115             else bout.write(false);
116         }
117     }
118     bout.close();
119 }
120
121 /* 解压 */
122 public static void expand(BinaryIn bin, BinaryOut bout) {
123     //1、先从压缩文件中头部读入二进制数据，构建霍夫曼树
124     Node root = readTries(bin);
125     //2、再从压缩文件中读入数据字符数量值
126     int N = bin.readInt();
127     //3、此时根据霍夫曼树翻译其中存储的变长前缀码
128     for (int i = 0; i < N; i++) {
129         Node x = root;
130         while (!x.isLeaf()) {
131             if (bin.readBoolean())
132                 x = x.right;
133             else x = x.left;
134         }
135         bout.write(x.ch);
136     }
137     bout.close();
138 }
139
140 public static void main(String[] args) {
141     BinaryIn bin = new BinaryIn(args[1]);
142     BinaryOut bout = new BinaryOut(args[2]);
143     if (args[0].equals("-")) compress(bin, bout);
144     else if (args[0].equals("+")) expand(bin, bout);
145 }
146 }
```

5.5.4 LZW编码

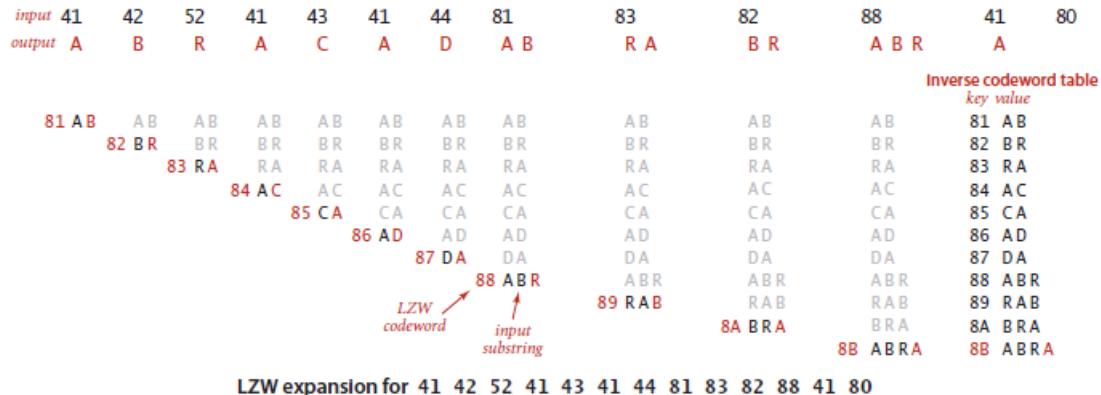
LZW编码压缩的大致过程，图示见下：

1. 使用128个ASCII字符初始化LZW编译表（使用三向单词查找树）；
2. 读入输入文件，找到输入字符串在单词查找树中的最长前缀匹配，然后输出（这个前缀字符串）对应的LZW编码；
3. 使用匹配的键和前瞻字符相连得到一个新键，将其与下一个LZW编码相关联并加入到三项查找树中。重复上述过程直到输入文本字符串的末尾。



LZW编码解压的大致过程，图示见下：

1. 读入当前编码，查找编译表，输出当前编码对应的字符或者字符串；
2. 读入下一个编码（它一定能在编译表中找到），然后找到其对应的字符串，取其首字母；
3. 将当前编码对应的字符或字符串与上面的首字符相连，产生新的字符，然后将下一个LZW编码与之对应，加入到编译表中；
4. 将当前编码设置为下一个编码的值，重新迭代。



```
1 import edu.princeton.cs.algs4.BinaryIn;
2 import edu.princeton.cs.algs4.BinaryOut;
3 import edu.princeton.cs.algs4.TST;
4
5 public class LZW {
6     private static final int R = 156;
7     private static final int L = 4096;
8     private static final int W = 12;
9
10    public static void compress(BinaryIn bin, BinaryOut bout) {
11        String input = bin.readString();
12        TST<Integer> st = new TST<Integer>();
13
14        /* 1、将128个字符与其对应的编码加入到三向单词查找树中 */
15        for (int i = 0; i < R; i++)
16            st.put(""+(char)i, i);
17        int code = R + 1;//code记录当前编译表的最大编码
18
19        while (input.length() > 0) {
20            /* 2、读入输入文件数据，找到输入在单词查找树中的最长匹配，
21             * 然后将对应的编码写入到压缩文件之中， */
22            String s = st.longestPrefixOf(input);
23            bout.write(st.get(s), W);
24
25            /* 3、然后将匹配字符串+前瞻（下一个）字符合成的字符串加入到
26             * 单词查找树中，并为其编入新的LZW码 */
27            int t = s.length();
28            if (t < input.length() && code < L)
29                st.put(input.substring(0, t + 1), code++);
30            input = input.substring(t); //将输入字符串的起始位置向后移动
31        }
32
33        bout.write(R, W);
34        bout.close();
35    }
36
37    public static void expand(BinaryIn bin, BinaryOut bout) {
```

```
38     String[] st = new String[L];
39     int i;
40
41     /* 1、构建LZW编码到ASCII字符的反向映射表 */
42     for (i = 0; i < R; ++i)
43         st[i] = "" + (char) i;
44     st[i++] = " ";
45
46     /* 2、从压缩文件中读取第一个LZW编码，并从反向映射表中
47      * 查询相应的ASCII字符 */
48     int codeward = bin.readInt(W);
49     String val = st[codeward];
50     while (true) {
51         /* 3、将当前LZW编码对应的字符写入到解压文件之中 */
52         bout.write(val);
53
54         /* 4、读入下一个LZW编码，根据反向映射表查询其表示字符
55          * 或字符串，然后取其第一个字符与当前字符组成新的字符串
56          * 加入到反向映射表（并为其编码，这个反向映射表也即编译表） */
57         codeward = bin.readInt(W);
58         if (codeward == R) break;
59         String s = st[codeward];
60         if (i == codeward)
61             s = val + val.charAt(0);
62         if (i < L)
63             st[i++] = val + s.charAt(0);
64         /* 5、将下一个字符串的设置为当前字符串，重新迭代 */
65         val = s;
66     }
67
68     bout.close();
69 }
70
71 public static void main(String[] args) {
72     BinaryIn bin = new BinaryIn(args[1]);
73     BinaryOut bout = new BinaryOut(args[2]);
74     if (args[0].equals("-"))
75         compress(bin, bout);
76     else if (args[0].equals("+"))
77         expand(bin, bout);
78 }
79 }
```