算法

原作者github: https://github.com/CyC2018/Interview-Notebook

PDF离线制作github: https://github.com/sjsdfg/Interview-Notebook-PDF

希望各位不吝star

一、前言

本文实现代码以及测试代码放在 Algorithm

二、算法分析

数学模型

1. 近似

$N^3/6-N^2/2+N/3$ \~ $N^3/6$。使用 \~f(N) 来表示所有随着 N 的增大除以 f(N) 的结果趋近于 1 的函数。

2. 增长数量级

$N^3/6-N^2/2+N/3$ 的增长数量级为 $O(N^3)$。增长数量级将算法与它的实现隔离开来，一个算法的增长数量级为 $O(N^3)$ 与它是否用 Java 实现，是否运行于特定计算机上无关。

3. 内循环

执行最频繁的指令决定了程序执行的总时间，把这些指令称为程序的内循环。

4. 成本模型

使用成本模型来评估算法，例如数组的访问次数就是一种成本模型。

ThreeSum

ThreeSum 用于统计一个数组中和为 0 的三元组数量。

```
public interface ThreeSum {
    int count(int[] nums);
}
```

```
public class ThreeSumSlow implements ThreeSum {
    @Override
    public int count(int[] nums) {
        int N = nums.length;
        int cnt = 0;
        for (int i = 0; i < N; i++)
            for (int j = i + 1; j < N; j++)
                for (int k = j + 1; k < N; k++)
                    if (nums[i] + nums[j] + nums[k] == 0)
                        cnt++;
        return cnt;
    }
}
```

该算法的内循环为 `if (nums[i] + nums[j] + nums[k] == 0)` 语句，总共执行的次数为 $N(N-1)(N-2) = N^3/6-N^2/2+N/3$，因此它的近似执行次数为 $\sim N^3/6$，增长数量级为 $O(N^3)$。

改进

通过将数组先排序，对两个元素求和，并用二分查找方法查找是否存在该和的相反数，如果存在，就说明存在三元组的和为 0。

应该注意的是，只有数组不含有相同元素才能使用这种解法，否则二分查找的结果会出错。

该方法可以将 ThreeSum 算法增长数量级降低为 $O(N^2 \log N)$。

```
public class ThreeSumFast {
    public static int count(int[] nums) {
        Arrays.sort(nums);
        int N = nums.length;
        int cnt = 0;
        for (int i = 0; i < N; i++)
            for (int j = i + 1; j < N; j++) {
                int target = -nums[i] - nums[j];
                int index = BinarySearch.search(nums, target);
                // 应该注意这里的下标必须大于 j，否则会重复统计。
                if (index > j)
                    cnt++;
            }
        return cnt;
    }
}
```

```
public class BinarySearch {
    public static int search(int[] nums, int target) {
        int l = 0, h = nums.length - 1;
        while (l <= h) {
            int m = l + (h - l) / 2;
            if (target == nums[m])
                return m;
            else if (target > nums[m])
                l = m + 1;
            else
                h = m - 1;
        }
        return -1;
    }
}
```

倍率实验

如果 $T(N) \sim aN^b \log N$，那么 $T(2N)/T(N) \sim 2^b$。

例如对于暴力的 ThreeSum 算法，近似时间为 $\sim N^3/6$。进行如下实验：多次运行该算法，每

次取的 N 值为前一次的两倍，统计每次执行的时间，并统计本次运行时间与前一次运行时间的比值，得到如下结果：

N	Time(ms)	Ratio
500	48	/
1000	320	6.7
2000	555	1.7
4000	4105	7.4
8000	33575	8.2
16000	268909	8.0

可以看到，T(2N)/T(N) \~ 2^3，因此可以确定 T(N) \~ aN^3logN。

```
public class RatioTest {
    public static void main(String[] args) {
        int N = 500;
        int loopTimes = 7;
        double preTime = -1;
        while (loopTimes-- > 0) {
            int[] nums = new int[N];
            StopWatch.start();
            ThreeSum threeSum = new ThreeSumSlow();
            int cnt = threeSum.count(nums);
            System.out.println(cnt);
            double elapsedTime = StopWatch.elapsedTime();
            double ratio = preTime == -1 ? 0 : elapsedTime / preTime;
            System.out.println(N + "  " + elapsedTime + "  " + ratio);
            preTime = elapsedTime;
            N *= 2;
        }
    }
}
```

```
public class StopWatch {
    private static long start;

    public static void start(){
```

```
        start = System.currentTimeMillis();
    }

    public static double elapsedTime() {
        long now = System.currentTimeMillis();
        return (now - start) / 1000.0;
    }
}
```

注意事项

1. 大常数

在求近似时，如果低级项的常数系数很大，那么近似的结果就是错误的。

2. 缓存

计算机系统会使用缓存技术来组织内存，访问数组相邻的元素会比访问不相邻的元素快很多。

3. 对最坏情况下的性能的保证

在核反应堆、心脏起搏器或者刹车控制器中的软件，最坏情况下的性能是十分重要的。

4. 随机化算法

通过打乱输入，去除算法对输入的依赖。

5. 均摊分析

将所有操作的总成本除于操作总数来将成本均摊。例如对一个空栈进行 N 次连续的 push() 调用需要访问数组的元素为 N+4+8+16+...+2N=5N-4（N 是向数组写入元素，其余的都是调整数组大小时进行复制需要的访问数组操作），均摊后每次操作访问数组的平均次数为常数。

三、栈和队列

栈

First-In-Last-Out

```
public interface MyStack<Item> extends Iterable<Item> {
    MyStack<Item> push(Item item);

    Item pop() throws Exception;

    boolean isEmpty();

    int size();
}
```

1. 数组实现

```
public class ArrayStack<Item> implements MyStack<Item> {
    // 栈元素数组，只能通过转型来创建泛型数组
    private Item[] a = (Item[]) new Object[1];
    // 元素数量
    private int N = 0;

    @Override
    public MyStack<Item> push(Item item) {
        check();
        a[N++] = item;
        return this;
    }

    @Override
    public Item pop() throws Exception {
        if (isEmpty())
            throw new Exception("stack is empty");

        Item item = a[--N];
        check();
        a[N] = null; // 避免对象游离
        return item;
```

```
23.         }
24.
25.         private void check() {
26.             if (N >= a.length)
27.                 resize(2 * a.length);
28.             else if (N > 0 && N <= a.length / 4)
29.                 resize(a.length / 2);
30.         }
31.
32.         /**
33.          * 调整数组大小，使得栈具有伸缩性
34.          */
35.         private void resize(int size) {
36.             Item[] tmp = (Item[]) new Object[size];
37.             for (int i = 0; i < N; i++)
38.                 tmp[i] = a[i];
39.             a = tmp;
40.         }
41.
42.         @Override
43.         public boolean isEmpty() {
44.             return N == 0;
45.         }
46.
47.         @Override
48.         public int size() {
49.             return N;
50.         }
51.
52.         @Override
53.         public Iterator<Item> iterator() {
54.             // 返回逆序遍历的迭代器
55.             return new Iterator<Item>() {
56.                 private int i = N;
57.
58.                 @Override
59.                 public boolean hasNext() {
60.                     return i > 0;
61.                 }
62.
63.                 @Override
64.                 public Item next() {
65.                     return a[--i];
66.                 }
67.             };
```

```
        }
    }
```

2. 链表实现

需要使用链表的头插法来实现，因为头插法中最后压入栈的元素在链表的开头，它的 next 指针指向前一个压入栈的元素，在弹出元素时就可以通过 next 指针遍历到前一个压入栈的元素从而让这个元素称为新的栈顶元素。

```
public class ListStack<Item> implements MyStack<Item> {
    private Node top = null;
    private int N = 0;

    private class Node {
        Item item;
        Node next;
    }

    @Override
    public MyStack<Item> push(Item item) {
        Node newTop = new Node();
        newTop.item = item;
        newTop.next = top;
        top = newTop;
        N++;
        return this;
    }

    @Override
    public Item pop() throws Exception {
        if (isEmpty())
            throw new Exception("stack is empty");
        Item item = top.item;
        top = top.next;
        N--;
        return item;
    }

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

```
34.
35.     @Override
36.     public int size() {
37.         return N;
38.     }
39.
40.     @Override
41.     public Iterator<Item> iterator() {
42.         return new Iterator<Item>() {
43.             private Node cur = top;
44.
45.             @Override
46.             public boolean hasNext() {
47.                 return cur != null;
48.             }
49.
50.             @Override
51.             public Item next() {
52.                 Item item = cur.item;
53.                 cur = cur.next;
54.                 return item;
55.             }
56.         };
57.     }
58. }
```

队列

First-In-First-Out

下面是队列的链表实现，需要维护 first 和 last 节点指针，分别指向队首和队尾。

这里需要考虑 first 和 last 指针哪个作为链表的开头。因为出队列操作需要让队首元素的下一个元素成为队首，所以需要容易获取下一个元素，而链表的头部节点的 next 指针指向下一个元素，因此可以让 first 指针链表的开头。

```
1.  public interface MyQueue<Item> extends Iterable<Item> {
2.      int size();
3.
4.      boolean isEmpty();
5.
```

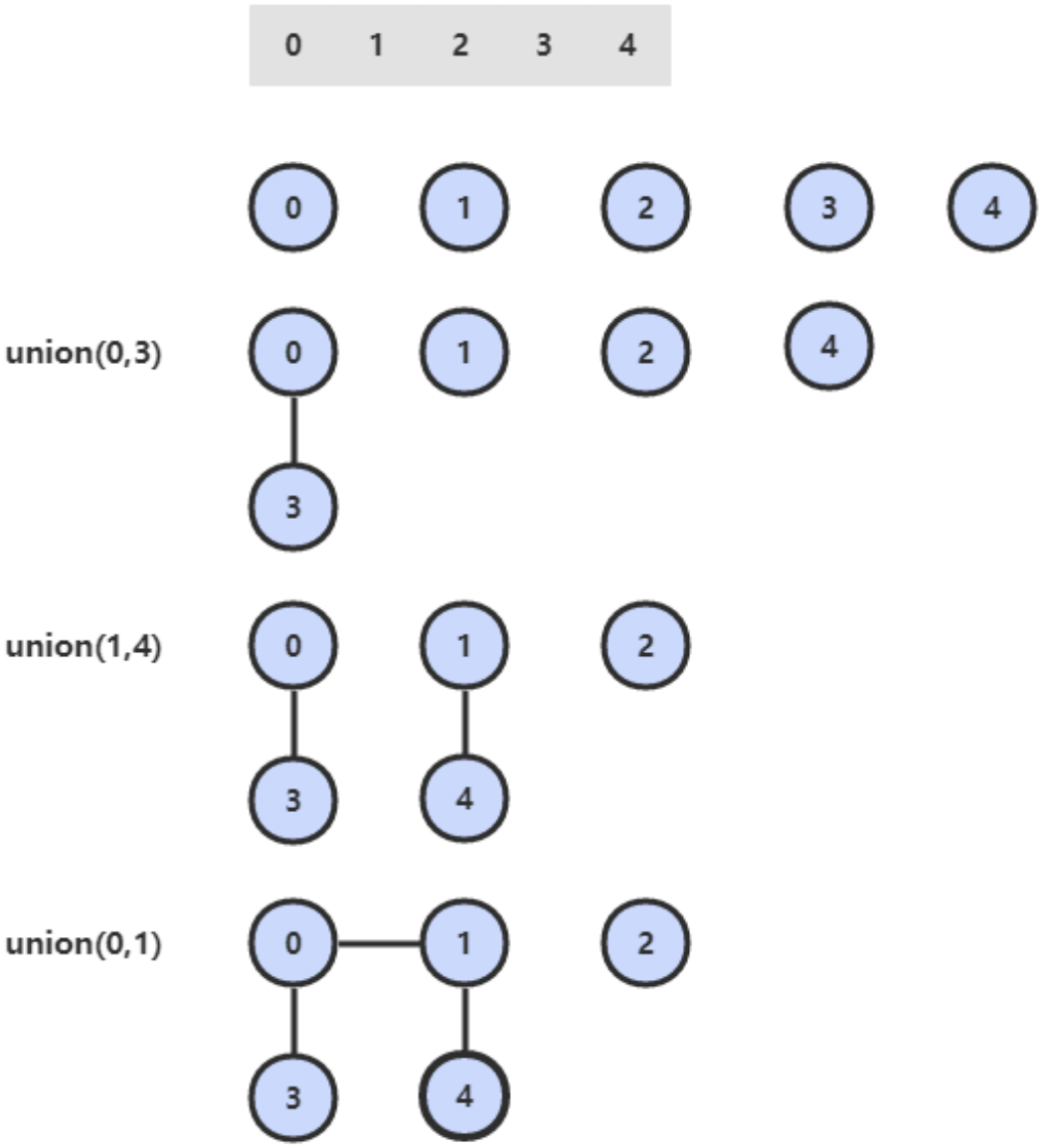
```
    MyQueue<Item> add(Item item);

    Item remove() throws Exception;
}
```

```
public class ListQueue<Item> implements MyQueue<Item> {
    private Node first;
    private Node last;
    int N = 0;

    private class Node {
        Item item;
        Node next;
    }

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

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

    @Override
    public MyQueue<Item> add(Item item) {
        Node newNode = new Node();
        newNode.item = item;
        newNode.next = null;
        if (isEmpty()) {
            last = newNode;
            first = newNode;
        } else {
            last.next = newNode;
            last = newNode;
        }
        N++;
        return this;
    }

    @Override
    public Item remove() throws Exception {
        if (isEmpty())
```

```
            throw new Exception("queue is empty");
        Node node = first;
        first = first.next;
        N--;
        if (isEmpty())
            last = null;
        return node.item;
    }

    @Override
    public Iterator<Item> iterator() {
        return new Iterator<Item>() {
            Node cur = first;

            @Override
            public boolean hasNext() {
                return cur != null;
            }

            @Override
            public Item next() {
                Item item = cur.item;
                cur = cur.next;
                return item;
            }
        };
    }
}
```

四、并查集

用于解决动态连通性问题，能动态连接两个点，并且判断两个点是否连通。




```java
public abstract class UF {
    protected int[] id;

    public UF(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public abstract int find(int p);
```

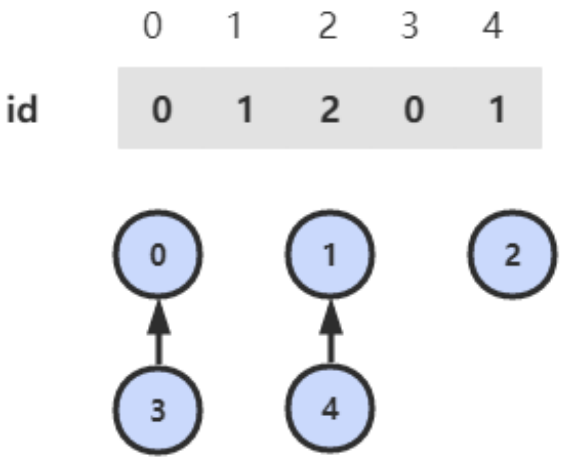
```
15.
16.         public abstract void union(int p, int q);
17.     }
```

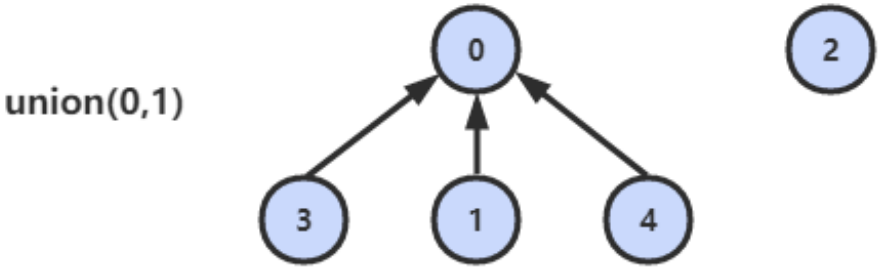
quick-find

可以快速进行 find 操作，即可以快速判断两个节点是否连通。

同一连通分量的所有节点的 id 值相等。

但是 union 操作代价却很高，需要将其中一个连通分量中的所有节点 id 值都修改为另一个节点的 id 值。



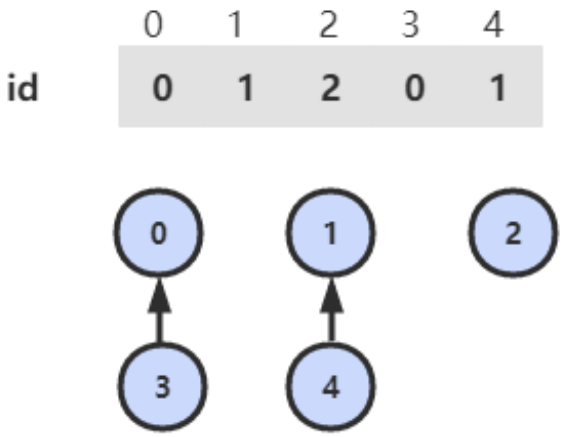


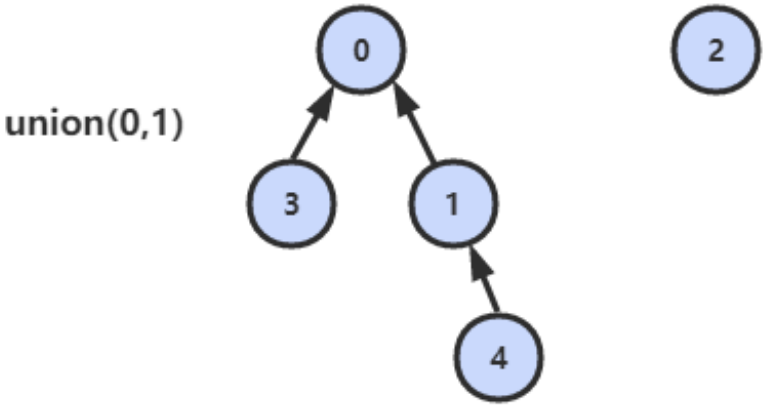
```java
public class QuickFindUF extends UF {
    public QuickFindUF(int N) {
        super(N);
    }

    @Override
    public int find(int p) {
        return id[p];
    }

    @Override
    public void union(int p, int q) {
        int pID = find(p);
        int qID = find(q);

        if (pID == qID)
            return;

        for (int i = 0; i < id.length; i++)
            if (id[i] == pID)
                id[i] = qID;
    }
}
```

quick-union

可以快速进行 union 操作，只需要修改一个节点的 id 值即可。

但是 find 操作开销很大，因为同一个连通分量的节点 id 值不同，id 值只是用来指向另一个节点。因此需要一直向上查找操作，直到找到最上层的节点。






```java
public class QuickUnionUF extends UF {
    public QuickUnionUF(int N) {
        super(N);
    }

    @Override
    public int find(int p) {
        while (p != id[p])
            p = id[p];
        return p;
    }

    @Override
```

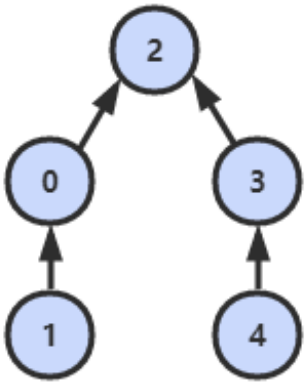
```
14.     public void union(int p, int q) {
15.         int pRoot = find(p);
16.         int qRoot = find(q);
17.         if (pRoot != qRoot)
18.             id[pRoot] = qRoot;
19.     }
20. }
```

这种方法可以快速进行 union 操作，但是 find 操作和树高成正比，最坏的情况下树的高度为触点的数目。

加权 quick-union

为了解决 quick-union 的树通常会很高的问题，加权 quick-union 在 union 操作时会让较小的树连接较大的树上面。

理论研究证明，加权 quick-union 算法构造的树深度最多不超过 logN。




```java
public class WeightedQuickUnionUF extends UF {

    // 保存节点的数量信息
    private int[] sz;

    public WeightedQuickUnionUF(int N) {
        super(N);
        this.sz = new int[N];
        for (int i = 0; i < N; i++)
            this.sz[i] = 1;
    }

    @Override
    public int find(int p) {
        while (p != id[p])
            p = id[p];
        return p;
    }

    @Override
    public void union(int p, int q) {
        int i = find(p);
        int j = find(q);
        if (i == j) return;
        if (sz[i] < sz[j]) {
            id[i] = j;
            sz[j] += sz[i];
        } else {
            id[j] = i;
            sz[i] += sz[j];
        }
    }
```

```
33.     }
```

路径压缩的加权 quick-union

在检查节点的同时将它们直接链接到根节点，只需要在 find 中添加一个循环即可。

各种 union-find 算法的比较

算法	union	find
quick-find	N	1
quick-union	树高	树高
加权 quick-union	logN	logN
路径压缩的加权 quick-union	非常接近 1	非常接近 1

五、排序

待排序的元素需要实现 Java 的 Comparable 接口，该接口有 compareTo() 方法，可以用它来判断两个元素的大小关系。

研究排序算法的成本模型时，计算的是比较和交换的次数。

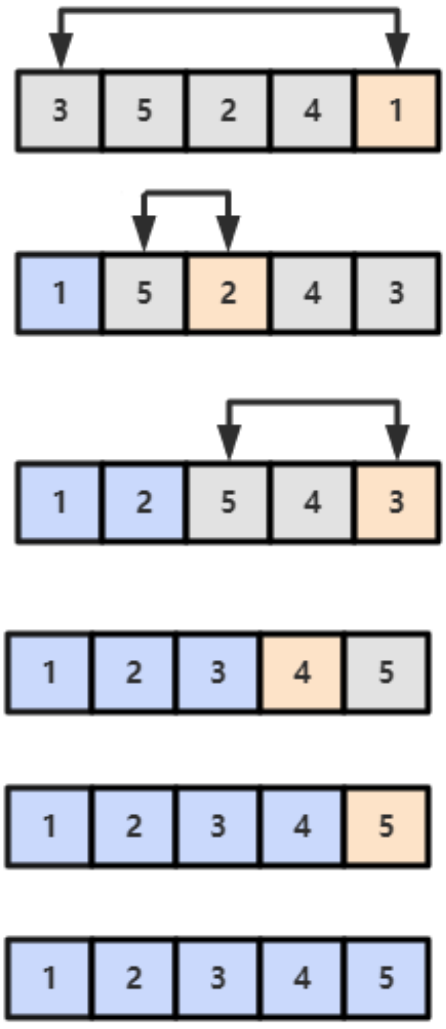
使用辅助函数 less() 和 swap() 来进行比较和交换的操作，使得代码的可读性和可移植性更好。

```
1.  public abstract class Sort<T extends Comparable<T>> {
2.
3.      public abstract void sort(T[] nums);
4.
5.      protected boolean less(T v, T w) {
6.          return v.compareTo(w) < 0;
7.      }
8.
```

```
    protected void swap(T[] a, int i, int j) {
        T t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

选择排序

选择出数组中的最小元素，将它与数组的第一个元素交换位置。再从剩下的元素中选择出最小的元素，将它与数组的第二个元素交换位置。不断进行这样的操作，直到将整个数组排序。




```
public class Selection<T extends Comparable<T>> extends Sort<T> {
```

```
    @Override
    public void sort(T[] nums) {
        int N = nums.length;
        for (int i = 0; i < N; i++) {
            int min = i;
            for (int j = i + 1; j < N; j++)
                if (less(nums[j], nums[min]))
                    min = j;
            swap(nums, i, min);
        }
    }
}
```

选择排序需要 \~$N^2/2$ 次比较和 \~N 次交换，它的运行时间与输入无关，这个特点使得它对一个已经排序的数组也需要这么多的比较和交换操作。

冒泡排序

通过从左到右不断交换相邻逆序的相邻元素，在一轮的交换之后，可以让未排序的元素上浮到右侧。

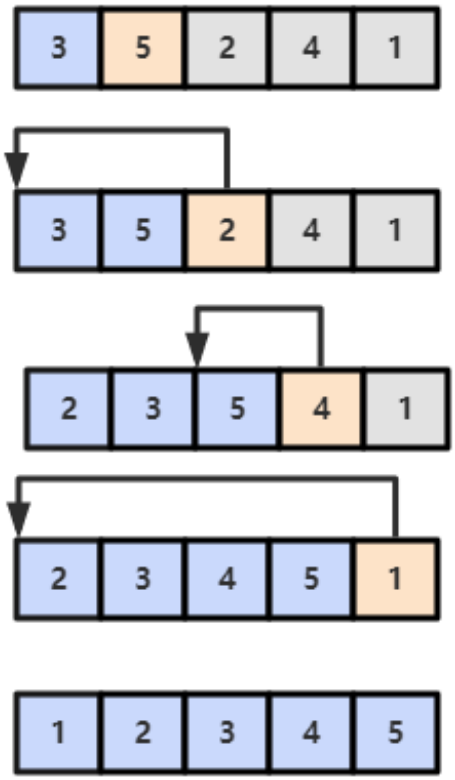
在一轮循环中，如果没有发生交换，就说明数组已经是有序的，此时可以直接退出。

```
public class Bubble<T extends Comparable<T>> extends Sort<T> {
    @Override
    public void sort(T[] nums) {
        int N = nums.length;
        boolean hasSorted = false;
        for (int i = 0; i < N && !hasSorted; i++) {
            hasSorted = true;
            for (int j = 0; j < N - i - 1; j++) {
                if (less(nums[j + 1], nums[j])) {
                    hasSorted = false;
                    swap(nums, j, j + 1);
                }
            }
        }
    }
}
```

插入排序

插入排序从左到右进行，每次都将当前元素插入到左侧已经排序的数组中，使得插入之后左部数组依然有序。

第 j 元素是通过不断向左比较并交换来实现插入过程：当第 j 元素小于第 j - 1 元素，就将它们的位置交换，然后令 j 指针向左移动一个位置，不断进行以上操作。




```
public class Insertion<T extends Comparable<T>> extends Sort<T> {
    @Override
    public void sort(T[] nums) {
        int N = nums.length;
        for (int i = 1; i < N; i++)
            for (int j = i; j > 0 && less(nums[j], nums[j - 1]); j--)
                swap(nums, j, j - 1);
    }
}
```

对于数组 {3, 5, 2, 4, 1}，它具有以下逆序：(3, 2), (3, 1), (5, 2), (5, 4), (5, 1), (2, 1), (4, 1)，插入排序每次只能交换相邻元素，令逆序数量减少 1，因此插入排序需要交换的次数为逆序数量。

插入排序的复杂度取决于数组的初始顺序，如果数组已经部分有序了，逆序较少，那么插入排序会很快。

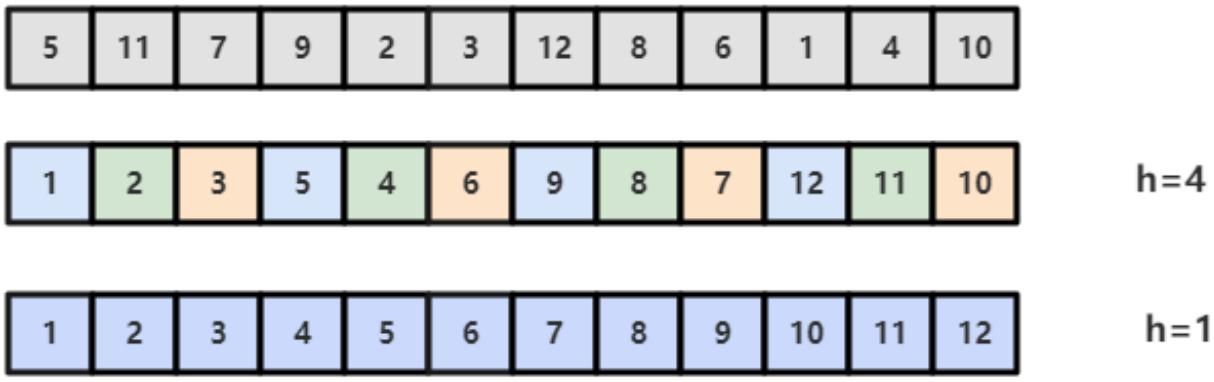
- 平均情况下插入排序需要 \~$N^2/4$ 比较以及 \~$N^2/4$ 次交换；
- 最坏的情况下需要 \~$N^2/2$ 比较以及 \~$N^2/2$ 次交换，最坏的情况是数组是倒序的；
- 最好的情况下需要 N-1 次比较和 0 次交换，最好的情况就是数组已经有序了。

希尔排序

对于大规模的数组，插入排序很慢，因为它只能交换相邻的元素，每次只能将逆序数量减少 1。

希尔排序的出现就是为了改进插入排序的这种局限性，它通过交换不相邻的元素，每次可以将逆序数量减少大于 1。

希尔排序使用插入排序对间隔 h 的序列进行排序。通过不断减小 h，最后令 h=1，就可以使得整个数组是有序的。



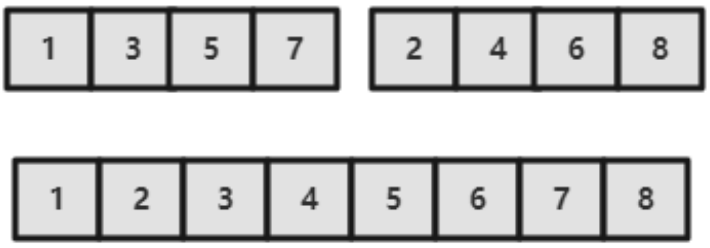

```
1.  public class Shell<T extends Comparable<T>> extends Sort<T> {
2.      @Override
```

```
    public void sort(T[] nums) {
        int N = nums.length;
        int h = 1;
        while (h < N / 3)
            h = 3 * h + 1;  // 1, 4, 13, 40, ...

        while (h >= 1) {
            for (int i = h; i < N; i++)
                for (int j = i; j >= h && less(nums[j], nums[j - h]); j -= h)
                    swap(nums, j, j - h);
            h = h / 3;
        }
    }
}
```

希尔排序的运行时间达不到平方级别，使用递增序列 1, 4, 13, 40, ... 的希尔排序所需要的比较次数不会超过 N 的若干倍乘于递增序列的长度。后面介绍的高级排序算法只会比希尔排序快两倍左右。

归并排序

归并排序的思想是将数组分成两部分，分别进行排序，然后归并起来。



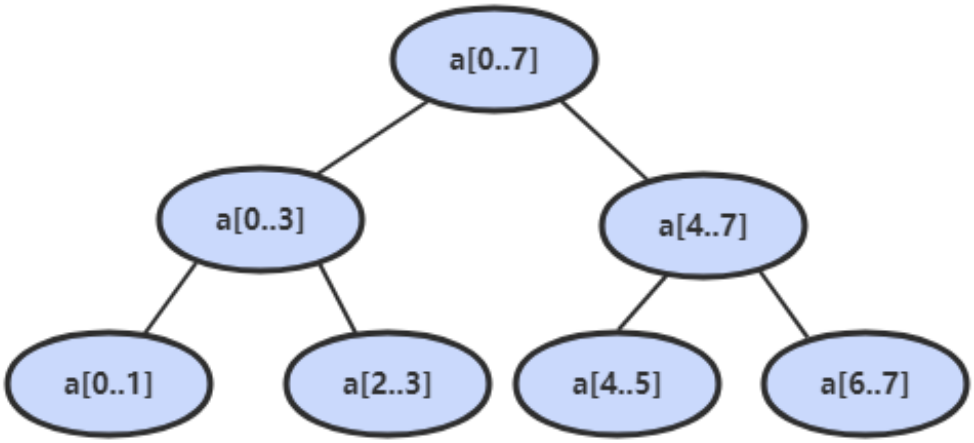
1. 归并方法

归并方法将数组中两个已经排序的部分归并成一个。

```
public abstract class MergeSort<T extends Comparable<T>> extends Sort<T
```

```
> {

    protected T[] aux;

    protected void merge(T[] nums, int l, int m, int h) {
        int i = l, j = m + 1;

        for (int k = l; k <= h; k++)
            aux[k] = nums[k];         // 将数据复制到辅助数组

        for (int k = l; k <= h; k++) {
            if (i > m)
                nums[k] = aux[j++];
            else if (j > h)
                nums[k] = aux[i++];
            else if (aux[i].compareTo(nums[j]) <= 0)
                nums[k] = aux[i++];  // 先进行这一步，保证稳定性
            else
                nums[k] = aux[j++];
        }
    }
}
```

2. 自顶向下归并排序




```
public class Up2DownMergeSort<T extends Comparable<T>> extends MergeSort<T> {
```

```
    @Override
    public void sort(T[] nums) {
        aux = (T[]) new Comparable[nums.length];
        sort(nums, 0, nums.length - 1);
    }

    private void sort(T[] nums, int l, int h) {
        if (h <= l)
            return;
        int mid = l + (h - l) / 2;
        sort(nums, l, mid);
        sort(nums, mid + 1, h);
        merge(nums, l, mid, h);
    }
}
```

因为每次都将问题对半分成两个子问题，而这种对半分的算法复杂度一般为 O(NlogN)，因此该归并排序方法的时间复杂度也为 O(NlogN)。

3. 自底向上归并排序

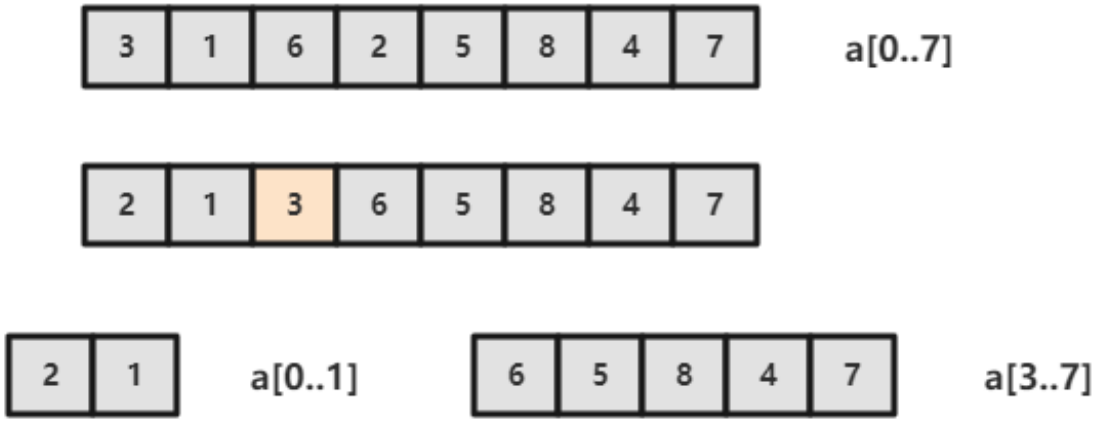
先归并那些微型数组，然后成对归并得到的微型数组。

```
public class Down2UpMergeSort<T extends Comparable<T>> extends MergeSort<T> {
    @Override
    public void sort(T[] nums) {
        int N = nums.length;
        aux = (T[]) new Comparable[N];
        for (int sz = 1; sz < N; sz += sz)
            for (int lo = 0; lo < N - sz; lo += sz + sz)
                merge(nums, lo, lo + sz - 1, Math.min(lo + sz + sz - 1, N - 1));
    }
}
```

快速排序

1. 基本算法

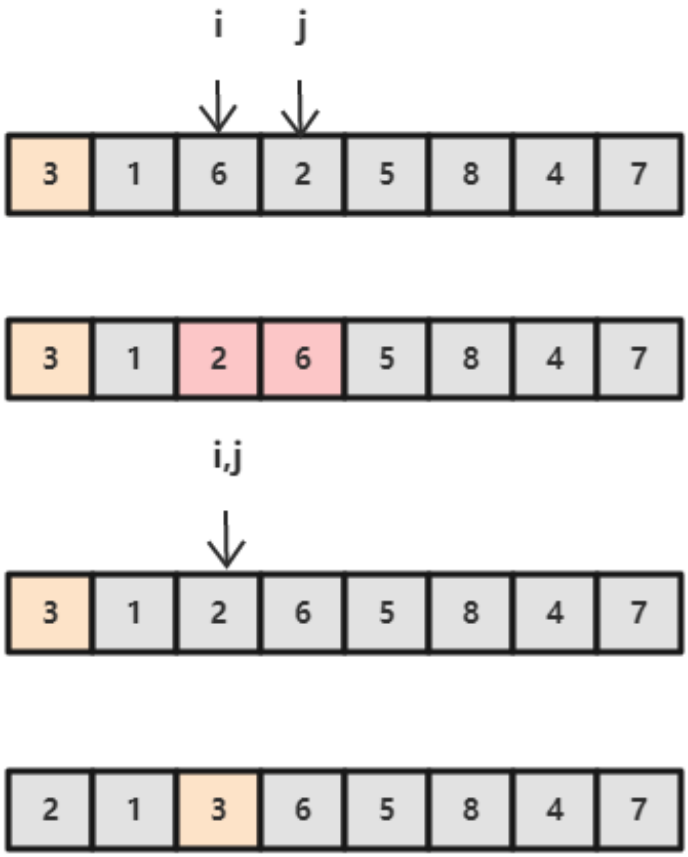
- 归并排序将数组分为两个子数组分别排序，并将有序的子数组归并使得整个数组排序；
- 快速排序通过一个切分元素将数组分为两个子数组，左子数组小于等于切分元素，右子数组大于等于切分元素，将这两个子数组排序也就将整个数组排序了。




```
public class QuickSort<T extends Comparable<T>> extends Sort<T> {
    @Override
    public void sort(T[] nums) {
        shuffle(nums);
        sort(nums, 0, nums.length - 1);
    }

    private void sort(T[] nums, int l, int h) {
        if (h <= l)
            return;
        int j = partition(nums, l, h);
        sort(nums, l, j - 1);
        sort(nums, j + 1, h);
    }

    private void shuffle(T[] nums) {
        List<Comparable> list = Arrays.asList(nums);
        Collections.shuffle(list);
        list.toArray(nums);
    }
}
```

2. 切分

取 a[lo] 作为切分元素，然后从数组的左端向右扫描直到找到第一个大于等于它的元素，再从数组的右端向左扫描找到第一个小于等于它的元素，交换这两个元素，并不断进行这个过程，就可以保证左指针 i 的左侧元素都不大于切分元素，右指针 j 的右侧元素都不小于切分元素。当两个指针相遇时，将切分元素 a[lo] 和 a[j] 交换位置。




```
private int partition(T[] nums, int l, int h) {
    int i = l, j = h + 1;
    T v = nums[l];
    while (true) {
        while (less(nums[++i], v) && i != h) ;
        while (less(v, nums[--j]) && j != l) ;
        if (i >= j)
            break;
        swap(nums, i, j);
    }
    swap(nums, l, j);
    return j;
}
```

3. 性能分析

快速排序是原地排序，不需要辅助数组，但是递归调用需要辅助栈。

快速排序最好的情况下是每次都正好能将数组对半分，这样递归调用次数才是最少的。这种情况下比较次数为 $C_N=2C_{N/2}+N$，复杂度为 O(NlogN)。

最坏的情况下，第一次从最小的元素切分，第二次从第二小的元素切分，如此这般。因此最坏的情况下需要比较 $N^2/2$。为了防止数组最开始就是有序的，在进行快速排序时需要随机打乱数组。

4. 算法改进

（一）切换到插入排序

因为快速排序在小数组中也会递归调用自己，对于小数组，插入排序比快速排序的性能更好，因此在小数组中可以切换到插入排序。

（二）三数取中

最好的情况下是每次都能取数组的中位数作为切分元素，但是计算中位数的代价很高。人们发现取 3 个元素并将大小居中的元素作为切分元素的效果最好。

（三）三向切分

对于有大量重复元素的数组，可以将数组切分为三部分，分别对应小于、等于和大于切分元素。

三向切分快速排序对于只有若干不同主键的随机数组可以在线性时间内完成排序。

```
1.  public class ThreeWayQuickSort<T extends Comparable<T>> extends
    QuickSort<T> {
2.      @Override
3.      protected void sort(T[] nums, int l, int h) {
4.          if (h <= l)
5.              return;
6.          int lt = l, i = l + 1, gt = h;
```

```
            T v = nums[l];
            while (i <= gt) {
                int cmp = nums[i].compareTo(v);
                if (cmp < 0)
                    swap(nums, lt++, i++);
                else if (cmp > 0)
                    swap(nums, i, gt--);
                else
                    i++;
            }
            sort(nums, l, lt - 1);
            sort(nums, gt + 1, h);
        }
    }
```

5. 基于切分的快速选择算法

快速排序的 partition() 方法，会返回一个整数 j 使得 a[l..j-1] 小于等于 a[j]，且 a[j+1..h] 大于等于 a[j]，此时 a[j] 就是数组的第 j 大元素。

可以利用这个特性找出数组的第 k 个元素。

```
public T select(T[] nums, int k) {
    int l = 0, h = nums.length - 1;
    while (h > l) {
        int j = partition(nums, l, h);
        if (j == k)
            return nums[k];
        else if (j > k)
            h = j - 1;
        else
            l = j + 1;
    }
    return nums[k];
}
```

该算法是线性级别的。因为每次能将数组二分，那么比较的总次数为 (N+N/2+N/4+..)，直到找到第 k 个元素，这个和显然小于 2N。

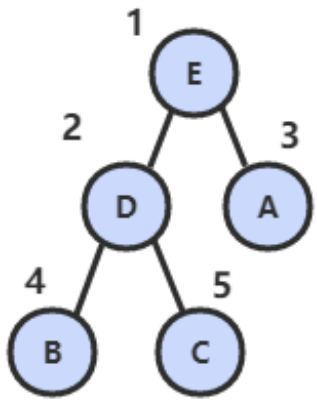
堆排序

1. 堆

堆的某个节点的值总是大于等于子节点的值，并且堆是一颗完全二叉树。

堆可以用数组来表示，因为堆是完全二叉树，而完全二叉树很容易就存储在数组中。位置 k 的节点的父节点位置为 k/2，而它的两个子节点的位置分别为 2k 和 2k+1。这里不使用数组索引为 0 的位置，是为了更清晰地描述节点的位置关系。

1	2	3	4	5
E	D	A	B	C

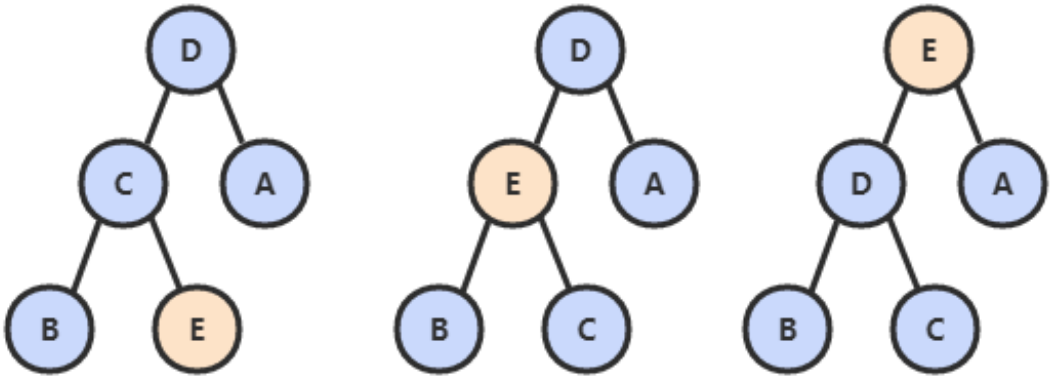



```
public class Heap<T extends Comparable<T>> {

    private T[] heap;
    private int N = 0;

    public Heap(int maxN) {
        this.heap = (T[]) new Comparable[maxN + 1];
    }

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

    public int size() {
```

```
        return N;
    }

    private boolean less(int i, int j) {
        return heap[i].compareTo(heap[j]) < 0;
    }

    private void swap(int i, int j) {
        T t = heap[i];
        heap[i] = heap[j];
        heap[j] = t;
    }
}
```

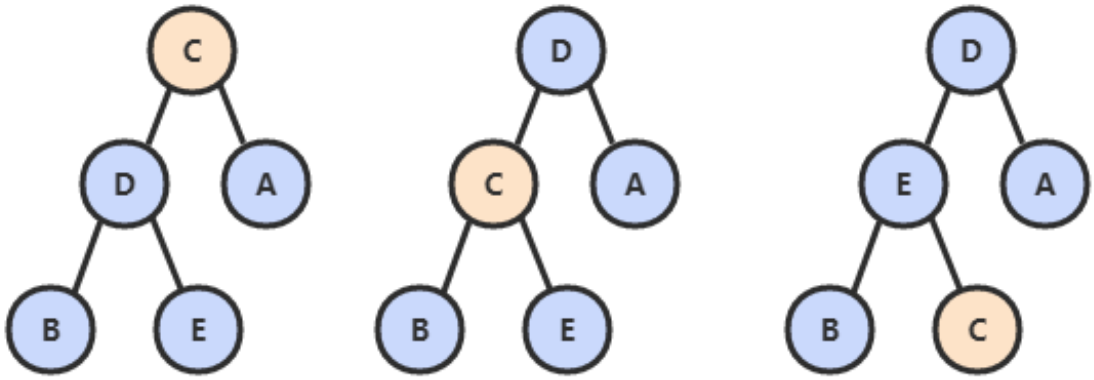
2. 上浮和下沉

在堆中，当一个节点比父节点大，那么需要交换这个两个节点。交换后还可能比它新的父节点大，因此需要不断地进行比较和交换操作，把这种操作称为上浮。




```
private void swim(int k) {
    while (k > 1 && less(k / 2, k)) {
        swap(k / 2, k);
        k = k / 2;
    }
}
```

类似地，当一个节点比子节点来得小，也需要不断地向下进行比较和交换操作，把这种操作称为下沉。一个节点如果有两个子节点，应当与两个子节点中最大那么节点进行交换。




```
private void sink(int k) {
    while (2 * k <= N) {
        int j = 2 * k;
        if (j < N && less(j, j + 1))
            j++;
        if (!less(k, j))
            break;
        swap(k, j);
        k = j;
    }
}
```

3. 插入元素

将新元素放到数组末尾，然后上浮到合适的位置。

```
public void insert(Comparable v) {
    heap[++N] = v;
    swim(N);
}
```

4. 删除最大元素

从数组顶端删除最大的元素，并将数组的最后一个元素放到顶端，并让这个元素下沉到合适的位置。

```
public T delMax() {
    T max = heap[1];
    swap(1, N--);
    heap[N + 1] = null;
    sink(1);
    return max;
}
```

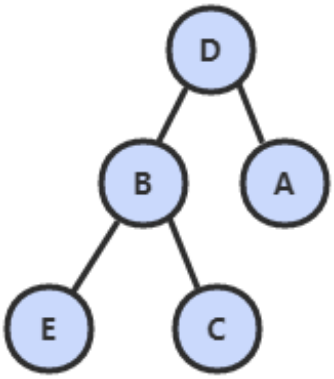
5. 堆排序

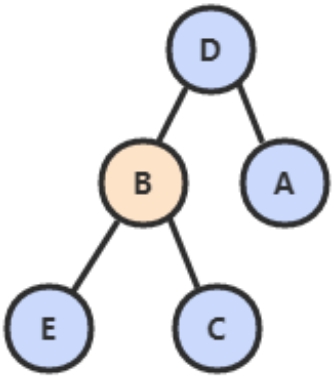
由于堆可以很容易得到最大的元素并删除它，不断地进行这种操作可以得到一个递减序列。如果把最大元素和当前堆中数组的最后一个元素交换位置，并且不删除它，那么就可以得到一个从尾到头的递减序列，从正向来看就是一个递增序列。因此很容易使用堆来进行排序。并且堆排序是原地排序，不占用额外空间。

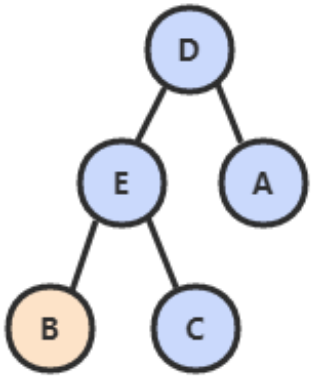
（一）构建堆

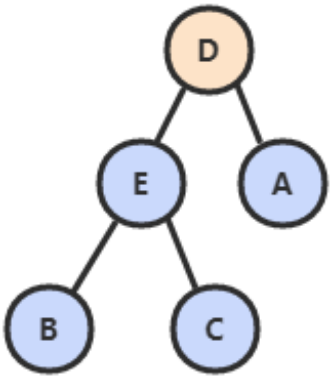
无序数组建立堆最直接的方法是从左到右遍历数组，然后进行上浮操作。一个更高效的方法是从右至左进行下沉操作，如果一个节点的两个节点都已经是堆有序，那么进行下沉操作可以使得这个节点为根节点的堆有序。叶子节点不需要进行下沉操作，可以忽略叶子节点的元素，因此只需要遍历一半的元素即可。

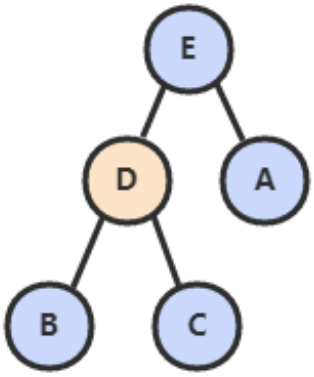
1	2	3	4	5
D	B	A	E	C

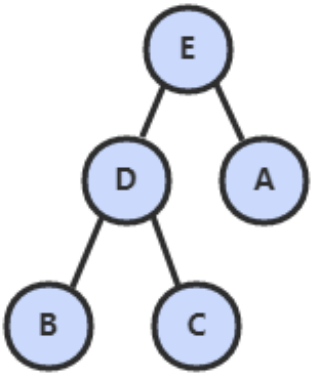


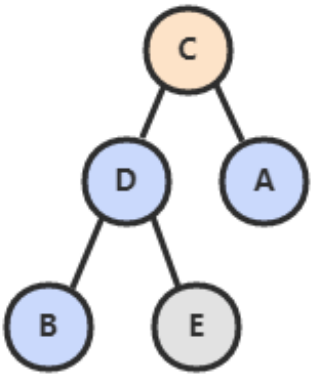


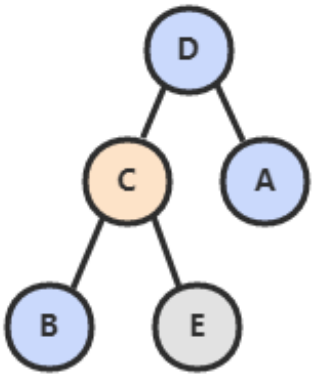


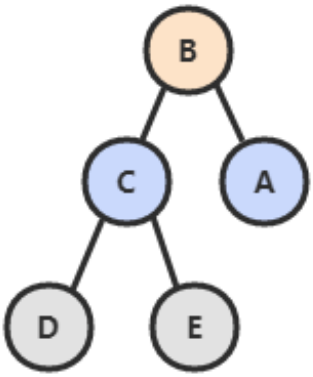


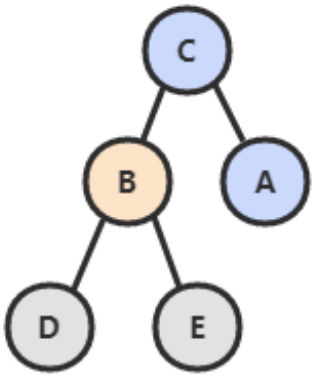


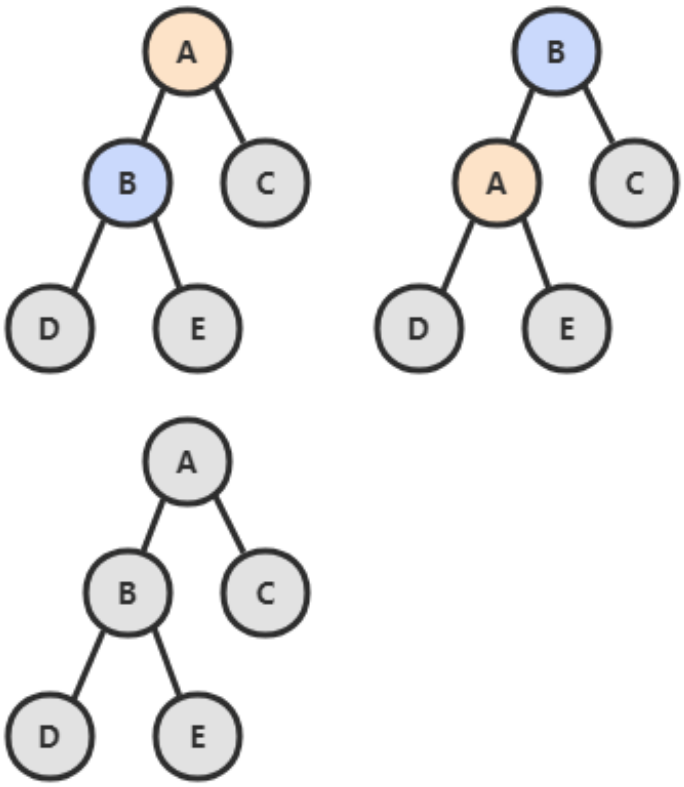

E
D
A
B
C

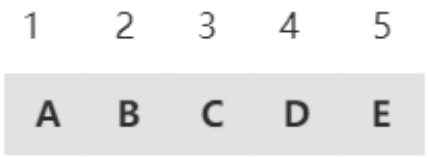

C
D
A
B
E


D
C
A
B
E


B
C
A
D
E


C
B
A
D
E






```java
public class HeapSort<T extends Comparable<T>> extends Sort<T> {
    /**
     * 数组第 0 个位置不能有元素
     */
    @Override
    public void sort(T[] nums) {
        int N = nums.length - 1;
        for (int k = N / 2; k >= 1; k--)
            sink(nums, k, N);

        while (N > 1) {
            swap(nums, 1, N--);
            sink(nums, 1, N);
        }
    }

```

```
    private void sink(T[] nums, int k, int N) {
        while (2 * k <= N) {
            int j = 2 * k;
            if (j < N && less(nums, j, j + 1))
                j++;
            if (!less(nums, k, j))
                break;
            swap(nums, k, j);
            k = j;
        }
    }

    private boolean less(T[] nums, int i, int j) {
        return nums[i].compareTo(nums[j]) < 0;
    }
}
```

6. 分析

一个堆的高度为 logN，因此在堆中插入元素和删除最大元素的复杂度都为 logN。

对于堆排序，由于要对 N 个节点进行下沉操作，因此复杂度为 NlogN。

堆排序时一种原地排序，没有利用额外的空间。

现代操作系统很少使用堆排序，因为它无法利用局部性原理进行缓存，也就是数组元素很少和相邻的元素进行比较。

小结

1. 排序算法的比较

算法	稳定	时间复杂度	空间复杂度	备注
选择排序	no	N^2	1	
冒泡排序	yes	N^2	1	

算法	稳定	时间复杂度	空间复杂度	备注
插入排序	yes	N \~ N^2	1	时间复杂度和初始顺序有关
希尔排序	no	N 的若干倍乘于递增序列的长度	1	
快速排序	no	NlogN	logN	
三向切分快速排序	no	N \~ NlogN	logN	适用于有大量重复主键
归并排序	yes	NlogN	N	
堆排序	no	NlogN	1	

快速排序是最快的通用排序算法，它的内循环的指令很少，而且它还能利用缓存，因为它总是顺序地访问数据。它的运行时间近似为 \~cNlogN，这里的 c 比其他线性对数级别的排序算法都要小。使用三向切分快速排序，实际应用中可能出现的某些分布的输入能够达到线性级别，而其它排序算法仍然需要线性对数时间。

2. Java 的排序算法实现

Java 主要排序方法为 java.util.Arrays.sort()，对于原始数据类型使用三向切分的快速排序，对于引用类型使用归并排序。

六、查找

符号表（Symbol Table）是一种存储键值对的数据结构，可以支持快速查找操作。

符号表分为有序和无序两种，有序符号表主要指支持 min()、max() 等根据键的大小关系来实现的操作。

有序符号表的键需要实现 Comparable 接口。

```
1.  public interface UnorderedST<Key, Value> {
```

```
    int size();

    Value get(Key key);

    void put(Key key, Value value);

    void delete(Key key);
}
```

```
public interface OrderedST<Key extends Comparable<Key>, Value> {

    int size();

    void put(Key key, Value value);

    Value get(Key key);

    Key min();

    Key max();

    int rank(Key key);

    List<Key> keys(Key l, Key h);
}
```

链表实现无序符号表

```
public class ListUnorderedST<Key, Value> implements UnorderedST<Key, Value> {

    private Node first;

    private class Node {
        Key key;
        Value value;
        Node next;

        Node(Key key, Value value, Node next) {
            this.key = key;
            this.value = value;
```

```
            this.next = next;
        }
    }

    @Override
    public int size() {
        int cnt = 0;
        Node cur = first;
        while (cur != null) {
            cnt++;
            cur = cur.next;
        }
        return cnt;
    }

    @Override
    public void put(Key key, Value value) {
        Node cur = first;
        // 如果在链表中找到节点的键等于 key 就更新这个节点的值为 value
        while (cur != null) {
            if (cur.key.equals(key)) {
                cur.value = value;
                return;
            }
            cur = cur.next;
        }
        // 否则使用头插法插入一个新节点
        first = new Node(key, value, first);
    }

    @Override
    public void delete(Key key) {
        if (first == null)
            return;
        if (first.key.equals(key))
            first = first.next;
        Node pre = first, cur = first.next;
        while (cur != null) {
            if (cur.key.equals(key)) {
                pre.next = cur.next;
                return;
            }
            pre = pre.next;
            cur = cur.next;
        }
```

```
        }

        @Override
        public Value get(Key key) {
            Node cur = first;
            while (cur != null) {
                if (cur.key.equals(key))
                    return cur.value;
                cur = cur.next;
            }
            return null;
        }
    }
```

二分查找实现有序符号表

使用一对平行数组，一个存储键一个存储值。

rank() 方法至关重要，当键在表中时，它能够知道该键的位置；当键不在表中时，它也能知道在何处插入新键。

复杂度：二分查找最多需要 logN+1 次比较，使用二分查找实现的符号表的查找操作所需要的时间最多是对数级别的。但是插入操作需要移动数组元素，是线性级别的。

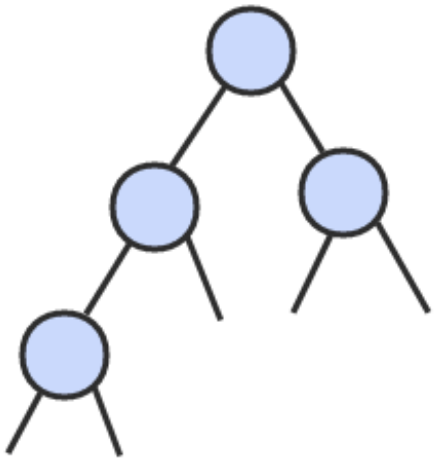
```
public class BinarySearchOrderedST<Key extends Comparable<Key>, Value>
implements OrderedST<Key, Value> {

    private Key[] keys;
    private Value[] values;
    private int N = 0;

    public BinarySearchOrderedST(int capacity) {
        keys = (Key[]) new Comparable[capacity];
        values = (Value[]) new Object[capacity];
    }

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

```

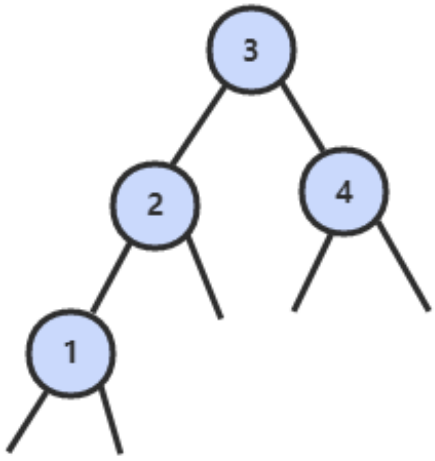
```java
    @Override
    public int rank(Key key) {
        int l = 0, h = N - 1;
        while (l <= h) {
            int m = l + (h - l) / 2;
            int cmp = key.compareTo(keys[m]);
            if (cmp == 0)
                return m;
            else if (cmp < 0)
                h = m - 1;
            else
                l = m + 1;
        }
        return l;
    }

    @Override
    public List<Key> keys(Key l, Key h) {
        int index = rank(l);
        List<Key> list = new ArrayList<>();
        while (keys[index].compareTo(h) <= 0) {
            list.add(keys[index]);
            index++;
        }
        return list;
    }

    @Override
    public void put(Key key, Value value) {
        int index = rank(key);
        // 如果找到已经存在的节点键位 key，就更新这个节点的值为 value
        if (index < N && keys[index].compareTo(key) == 0) {
            values[index] = value;
            return;
        }
        // 否则在数组中插入新的节点，需要先将插入位置之后的元素都向后移动一个位置
        for (int j = N; j > index; j--) {
            keys[j] = keys[j - 1];
            values[j] = values[j - 1];
        }
        keys[index] = key;
        values[index] = value;
        N++;
    }

```

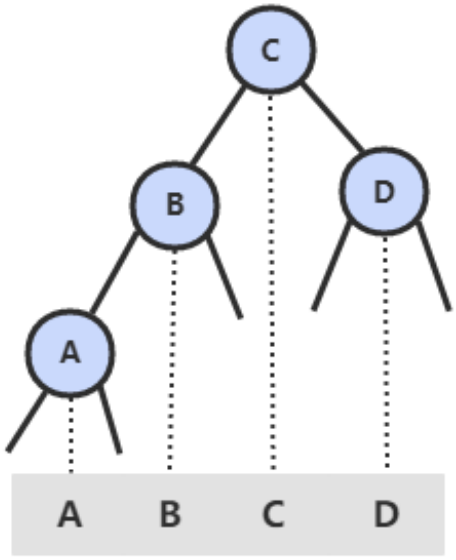
```
62.     @Override
63.     public Value get(Key key) {
64.         int index = rank(key);
65.         if (index < N && keys[index].compareTo(key) == 0)
66.             return values[index];
67.         return null;
68.     }
69. 
70.     @Override
71.     public Key min() {
72.         return keys[0];
73.     }
74. 
75.     @Override
76.     public Key max() {
77.         return keys[N - 1];
78.     }
79. }
```

二叉查找树

二叉树 是一个空链接，或者是一个有左右两个链接的节点，每个链接都指向一颗子二叉树。








```java
public class BST<Key extends Comparable<Key>, Value> implements
OrderedST<Key, Value> {

    protected Node root;

    protected class Node {
        Key key;
        Value val;
        Node left;
        Node right;
        // 以该节点为根的子树节点总数
        int N;
        // 红黑树中使用
```

```
        boolean color;

        Node(Key key, Value val, int N) {
            this.key = key;
            this.val = val;
            this.N = N;
        }
    }

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

    private int size(Node x) {
        if (x == null)
            return 0;
        return x.N;
    }

    protected void recalculateSize(Node x) {
        x.N = size(x.left) + size(x.right) + 1;
    }
}
```

（为了方便绘图，二叉树的空链接不画出来。）

1. get()

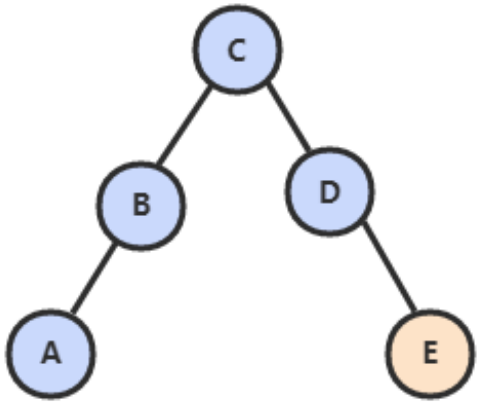
- 如果树是空的，则查找未命中；
- 如果被查找的键和根节点的键相等，查找命中；
- 否则递归地在子树中查找：如果被查找的键较小就在左子树中查找，较大就在右子树中查找。

```
@Override
public Value get(Key key) {
    return get(root, key);
}

private Value get(Node x, Key key) {
    if (x == null)
        return null;
```

```
        int cmp = key.compareTo(x.key);
        if (cmp == 0)
            return x.val;
        else if (cmp < 0)
            return get(x.left, key);
        else
            return get(x.right, key);
    }
```

2. put()

当插入的键不存在于树中，需要创建一个新节点，并且更新上层节点的链接使得该节点正确链接到树中。

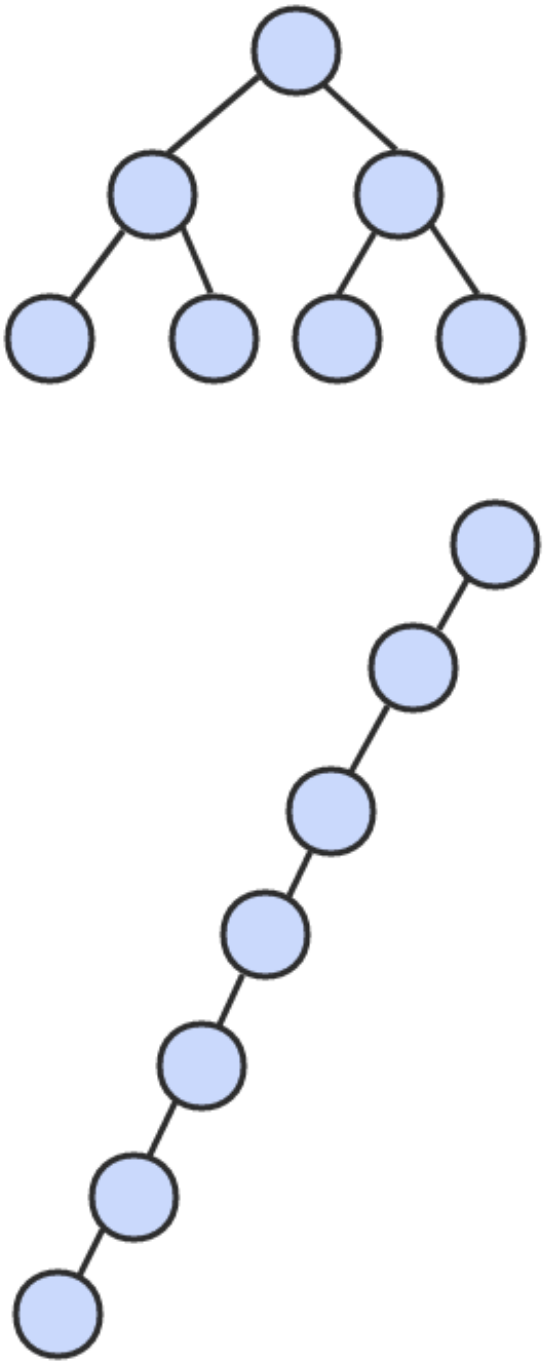



```
    @Override
    public void put(Key key, Value value) {
        root = put(root, key, value);
    }

    private Node put(Node x, Key key, Value value) {
        if (x == null)
            return new Node(key, value, 1);
        int cmp = key.compareTo(x.key);
        if (cmp == 0)
            x.val = value;
        else if (cmp < 0)
            x.left = put(x.left, key, value);
        else
            x.right = put(x.right, key, value);
        recalculateSize(x);
```

```
17.         return x;
18.     }
```

3. 分析

二叉查找树的算法运行时间取决于树的形状，而树的形状又取决于键被插入的先后顺序。最好的情况下树是完全平衡的，每条空链接和根节点的距离都为 logN。



4. floor()

floor(key)：小于等于键的最大键

- 如果键小于根节点的键，那么 floor(key) 一定在左子树中；
- 如果键大于根节点的键，需要先判断右子树中是否存在 floor(key)，如果存在就找到，否则根节点就是 floor(key)。

```
public Key floor(Key key) {
    Node x = floor(root, key);
    if (x == null)
        return null;
    return x.key;
}

private Node floor(Node x, Key key) {
    if (x == null)
        return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0)
        return x;
    if (cmp < 0)
        return floor(x.left, key);
    Node t = floor(x.right, key);
    return t != null ? t : x;
}
```

5. rank()

rank(key) 返回 key 的排名。

- 如果键和根节点的键相等，返回左子树的节点数；
- 如果小于，递归计算在左子树中的排名；
- 如果大于，递归计算在右子树中的排名，并加上左子树的节点数，再加上 1（根节点）。

```
@Override
public int rank(Key key) {
    return rank(key, root);
}
```

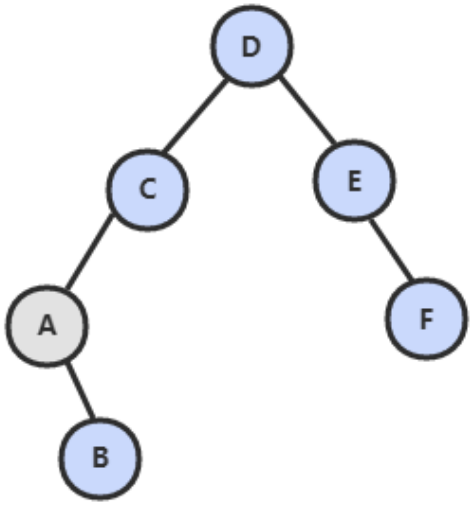
```
5.
6.  private int rank(Key key, Node x) {
7.      if (x == null)
8.          return 0;
9.      int cmp = key.compareTo(x.key);
10.     if (cmp == 0)
11.         return size(x.left);
12.     else if (cmp < 0)
13.         return rank(key, x.left);
14.     else
15.         return 1 + size(x.left) + rank(key, x.right);
16. }
```

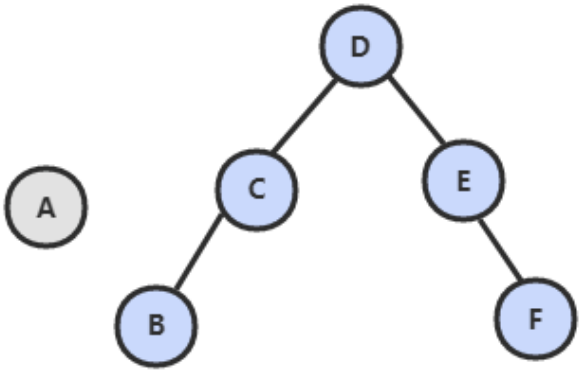
6. min()

```
1.  @Override
2.  public Key min() {
3.      return min(root).key;
4.  }
5.
6.  private Node min(Node x) {
7.      if (x == null)
8.          return null;
9.      if (x.left == null)
10.         return x;
11.     return min(x.left);
12. }
```

7. deleteMin()

令指向最小节点的链接指向最小节点的右子树。

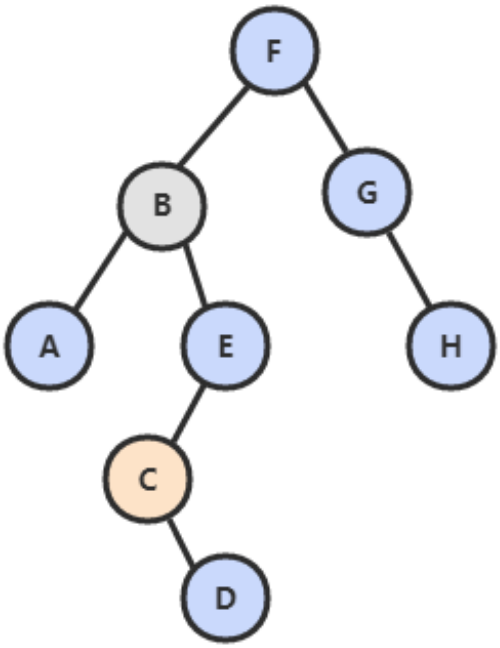


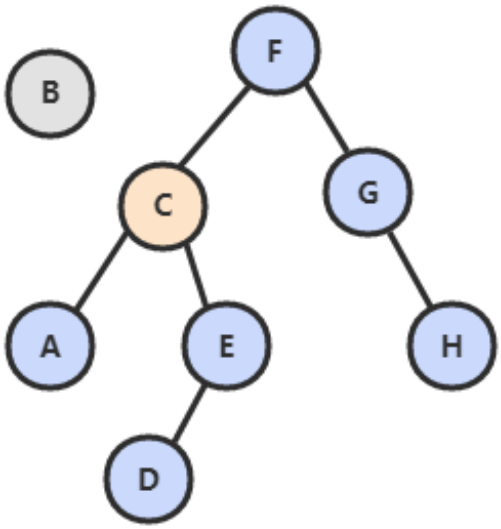



```java
public void deleteMin() {
    root = deleteMin(root);
}

public Node deleteMin(Node x) {
    if (x.left == null)
        return x.right;
    x.left = deleteMin(x.left);
    recalculateSize(x);
    return x;
}
```

8. delete()

- 如果待删除的节点只有一个子树，那么只需要让指向待删除节点的链接指向唯一的子树即可；
- 否则，让右子树的最小节点替换该节点。






```java
public void delete(Key key) {
    root = delete(root, key);
}
private Node delete(Node x, Key key) {
    if (x == null)
        return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = delete(x.left, key);
    else if (cmp > 0)
        x.right = delete(x.right, key);
    else {
        if (x.right == null)
            return x.left;
        if (x.left == null)
            return x.right;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    recalculateSize(x);
    return x;
}
```

9. keys()

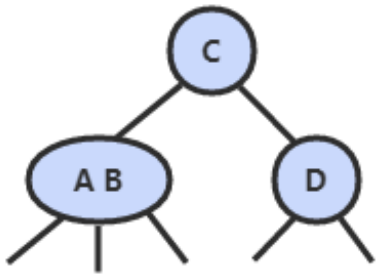
利用二叉查找树中序遍历的结果为递增的特点。

```
@Override
public List<Key> keys(Key l, Key h) {
    return keys(root, l, h);
}

private List<Key> keys(Node x, Key l, Key h) {
    List<Key> list = new ArrayList<>();
    if (x == null)
        return list;
    int cmpL = l.compareTo(x.key);
    int cmpH = h.compareTo(x.key);
    if (cmpL < 0)
        list.addAll(keys(x.left, l, h));
    if (cmpL <= 0 && cmpH >= 0)
        list.add(x.key);
    if (cmpH > 0)
        list.addAll(keys(x.right, l, h));
    return list;
}
```

10. 性能分析

复杂度：二叉查找树所有操作在最坏的情况下所需要的时间都和树的高度成正比。

2-3 查找树

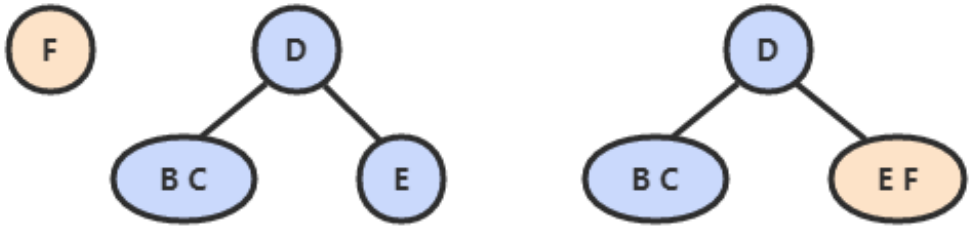


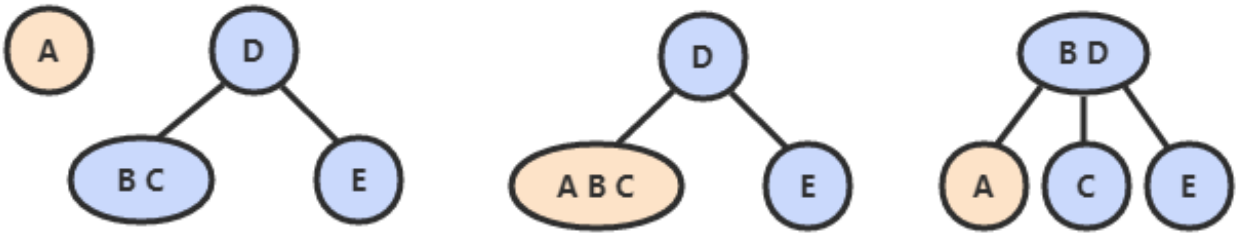
1. 插入操作

插入操作和 BST 的插入操作有很大区别，BST 的插入操作是先进行一次未命中的查找，然后再将节点插入到对应的空链接上。但是 2-3 查找树如果也这么做的话，那么就会破坏了平衡性。它是将新节点插入到叶子节点上。

根据叶子节点的类型不同，有不同的处理方式：

- 如果插入到 2- 节点上，那么直接将新节点和原来的节点组成 3- 节点即可。





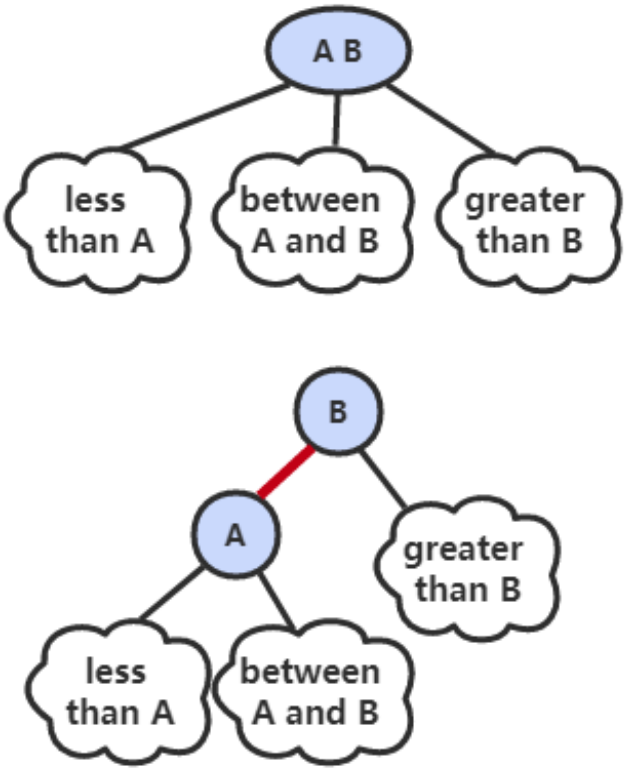
2. 性质

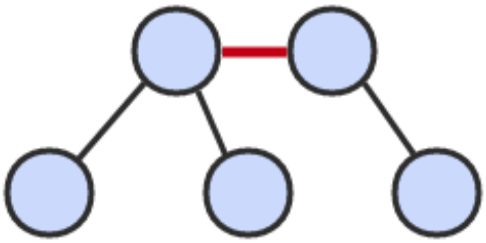
2-3 查找树插入操作的变换都是局部的，除了相关的节点和链接之外不必修改或者检查树的其它部分，而这些局部变换不会影响树的全局有序性和平衡性。

2-3 查找树的查找和插入操作复杂度和插入顺序无关，在最坏的情况下查找和插入操作访问的节点必然不超过 logN 个，含有 10 亿个节点的 2-3 查找树最多只需要访问 30 个节点就能进行任意的查找和插入操作。

红黑树

2-3 查找树需要用到 2- 节点和 3- 节点，红黑树使用红链接来实现 3- 节点。指向一个节点的链接颜色如果为红色，那么这个节点和上层节点表示的是一个 3- 节点，而黑色则是普通链接。

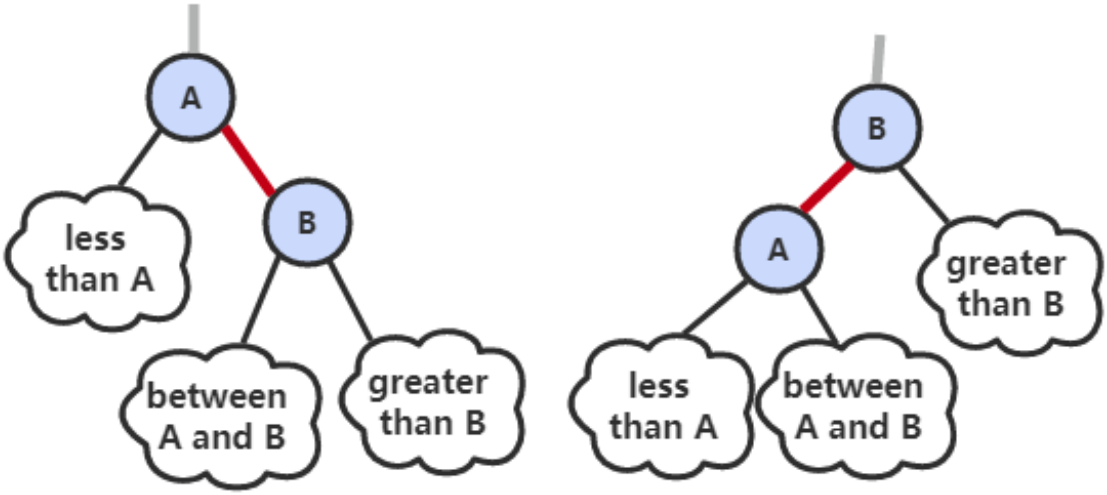




```java
public class RedBlackBST<Key extends Comparable<Key>, Value> extends BST<Key, Value> {
    private static final boolean RED = true;
    private static final boolean BLACK = false;

    private boolean isRed(Node x) {
        if (x == null)
            return false;
        return x.color == RED;
    }
}
```

1. 左旋转

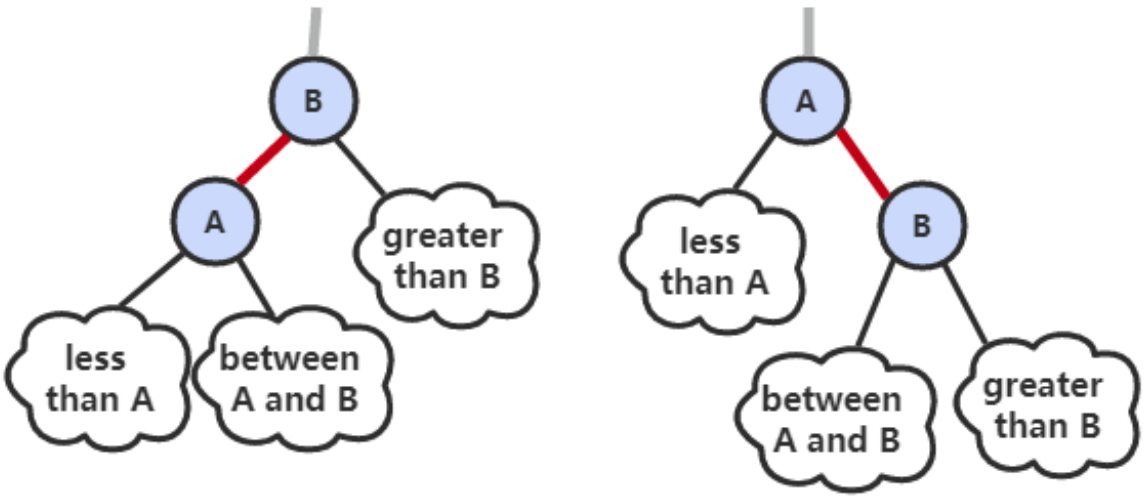
因为合法的红链接都为左链接，如果出现右链接为红链接，那么就需要进行左旋转操作。



```java
public Node rotateLeft(Node h) {
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    recalculateSize(h);
    return x;
}
```

2. 右旋转

进行右旋转是为了转换两个连续的左红链接，这会在之后的插入过程中探讨。

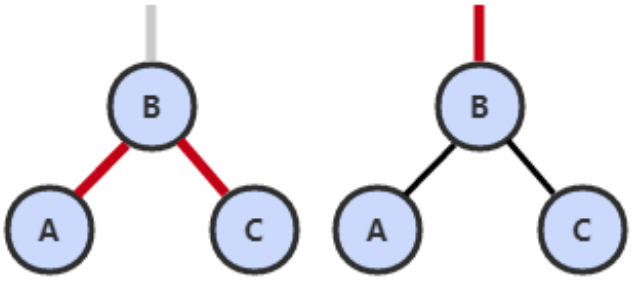



```java
public Node rotateRight(Node h) {
    Node x = h.left;
    h.left = x.right;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    recalculateSize(h);
    return x;
```

```
}
```

3. 颜色转换

一个 4- 节点在红黑树中表现为一个节点的左右子节点都是红色的。分裂 4- 节点除了需要将子节点的颜色由红变黑之外，同时需要将父节点的颜色由黑变红，从 2-3 树的角度看就是将中间节点移到上层节点。

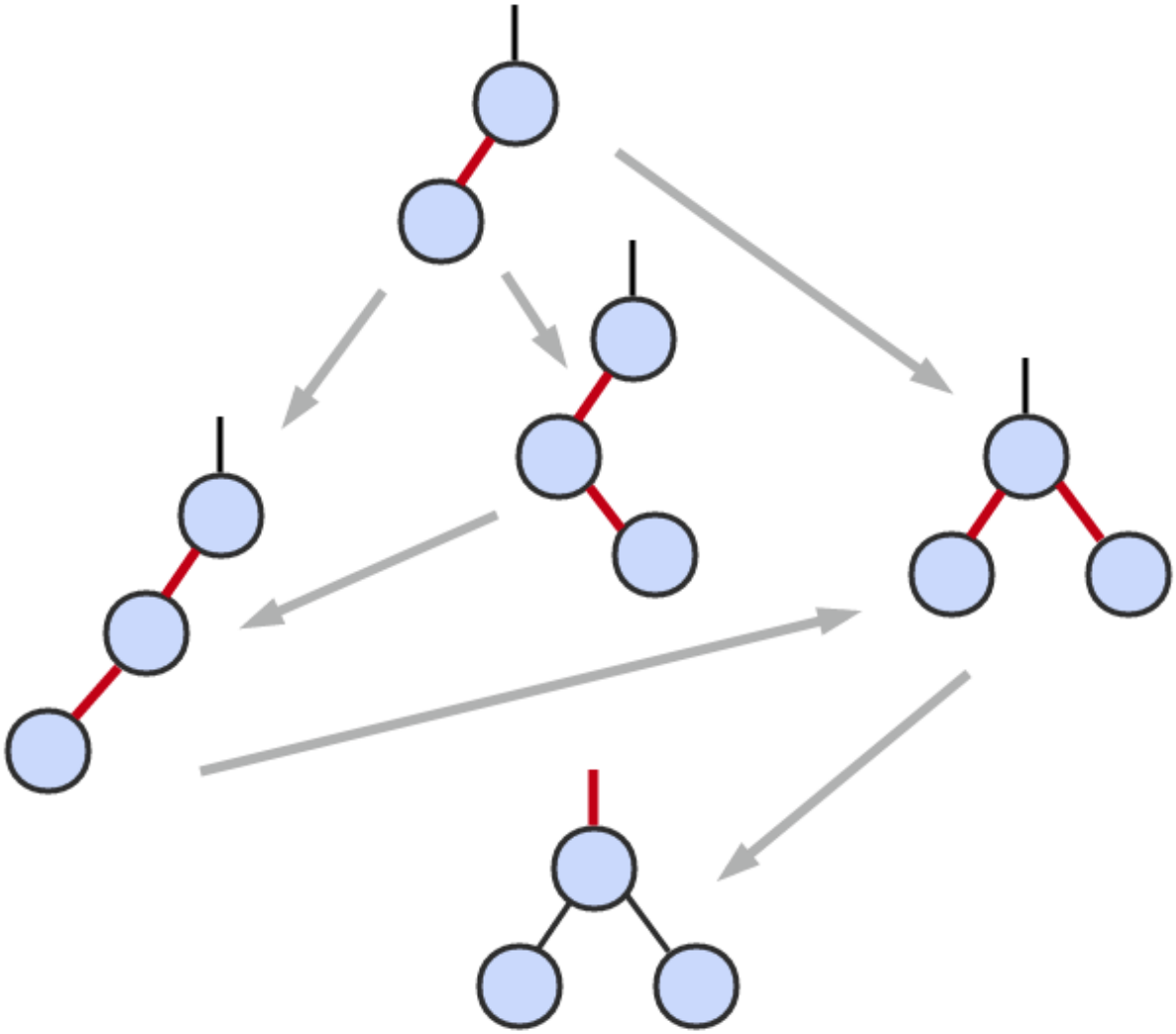



```
void flipColors(Node h) {
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

4. 插入

先将一个节点按二叉查找树的方法插入到正确位置，然后再进行如下颜色操作：

- 如果右子节点是红色的而左子节点是黑色的，进行左旋转；
- 如果左子节点是红色的，而且左子节点的左子节点也是红色的，进行右旋转；
- 如果左右子节点均为红色的，进行颜色转换。



```java
@Override
public void put(Key key, Value value) {
    root = put(root, key, value);
    root.color = BLACK;
}

private Node put(Node x, Key key, Value value) {
    if (x == null) {
        Node node = new Node(key, value, 1);
        node.color = RED;
        return node;
    }
    int cmp = key.compareTo(x.key);
    if (cmp == 0)
        x.val = value;
    else if (cmp < 0)
        x.left = put(x.left, key, value);
```

```
        else
            x.right = put(x.right, key, value);

        if (isRed(x.right) && !isRed(x.left))
            x = rotateLeft(x);
        if (isRed(x.left) && isRed(x.left.left))
            x = rotateRight(x);
        if (isRed(x.left) && isRed(x.right))
            flipColors(x);

        recalculateSize(x);
        return x;
    }
```

可以看到该插入操作和二叉查找树的插入操作类似，只是在最后加入了旋转和颜色变换操作即可。

根节点一定为黑色，因为根节点没有上层节点，也就没有上层节点的左链接指向根节点。flipColors() 有可能会使得根节点的颜色变为红色，每当根节点由红色变成黑色时树的黑链接高度加 1.

5. 分析

一颗大小为 N 的红黑树的高度不会超过 2logN。最坏的情况下是它所对应的 2-3 树，构成最左边的路径节点全部都是 3- 节点而其余都是 2- 节点。

红黑树大多数的操作所需要的时间都是对数级别的。

散列表

散列表类似于数组，可以把散列表的散列值看成数组的索引值。访问散列表和访问数组元素一样快速，它可以在常数时间内实现查找和插入操作。

由于无法通过散列值知道键的大小关系，因此散列表无法实现有序性操作。

1. 散列函数

对于一个大小为 M 的散列表，散列函数能够把任意键转换为 [0, M-1] 内的正整数，该正整数即为 hash 值。

散列表存在冲突，也就是两个不同的键可能有相同的 hash 值。

散列函数应该满足以下三个条件：

- 一致性：相等的键应当有相等的 hash 值，两个键相等表示调用 equals() 返回的值相等。
- 高效性：计算应当简便，有必要的话可以把 hash 值缓存起来，在调用 hash 函数时直接返回。
- 均匀性：所有键的 hash 值应当均匀地分布到 [0, M-1] 之间，这个条件至关重要，直接影响到散列表的性能。

除留余数法可以将整数散列到 [0, M-1] 之间，例如一个正整数 k，计算 k%M 既可得到一个 [0, M-1] 之间的 hash 值。注意 M 必须是一个素数，否则无法利用键包含的所有信息。例如 M 为 10^k，那么只能利用键的后 k 位。

对于其它数，可以将其转换成整数的形式，然后利用除留余数法。例如对于浮点数，可以将其表示成二进制形式，然后使用二进制形式的整数值进行除留余数法。

对于有多部分组合的键，每部分都需要计算 hash 值，并且最后合并时需要让每部分 hash 值都具有同等重要的地位。可以将该键看成 R 进制的整数，键中每部分都具有不同的权值。

例如，字符串的散列函数实现如下

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (R * hash + s.charAt(i)) % M;
```

再比如，拥有多个成员的自定义类的哈希函数如下：

```
int hash = (((day * R + month) % M) * R + year) % M;
```

R 通常取 31。

Java 中的 hashCode() 实现了 hash 函数，但是默认使用对象的内存地址值。在使用 hashCode() 函数时，应当结合除留余数法来使用。因为内存地址是 32 位整数，我们只需要

31 位的非负整数，因此应当屏蔽符号位之后再使用除留余数法。

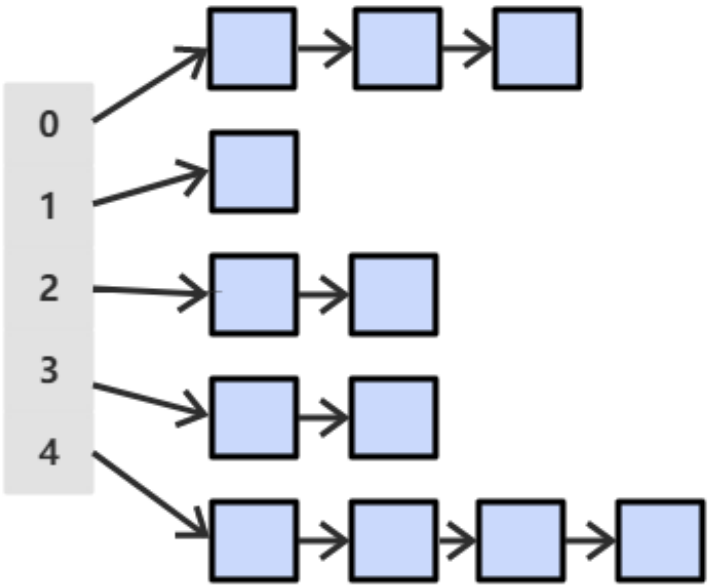
```
int hash = (x.hashCode() & 0x7fffffff) % M;
```

使用 Java 自带的 HashMap 等自带的哈希表实现时，只需要去实现 Key 类型的 hashCode() 函数即可。Java 规定 hashCode() 能够将键均匀分布于所有的 32 位整数，Java 中的 String、Integer 等对象的 hashCode() 都能实现这一点。以下展示了自定义类型如何实现 hashCode()。

```
public class Transaction {
    private final String who;
    private final Date when;
    private final double amount;

    public Transaction(String who, Date when, double amount) {
        this.who = who;
        this.when = when;
        this.amount = amount;
    }

    public int hashCode() {
        int hash = 17;
        int R = 31;
        hash = R * hash + who.hashCode();
        hash = R * hash + when.hashCode();
        hash = R * hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

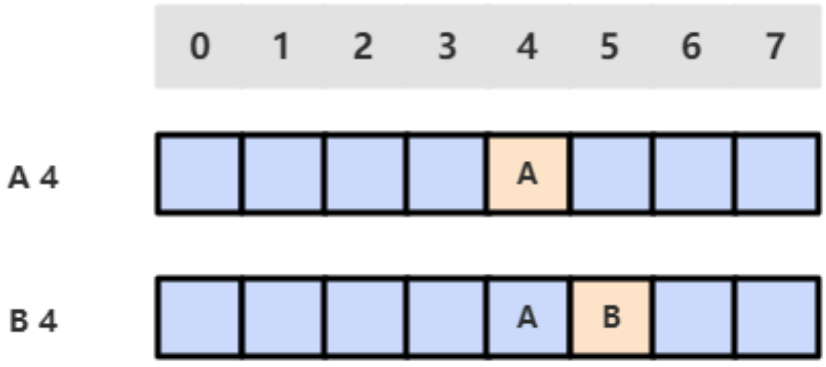
2. 基于拉链法的散列表

拉链法使用链表来存储 hash 值相同的键，从而解决冲突。此时查找需要分两步，首先查找 Key 所在的链表，然后在链表中顺序查找。



3. 基于线性探测法的散列表

线性探测法使用空位来解决冲突，当冲突发生时，向前探测一个空位来存储冲突的键。使用线性探测法，数组的大小 M 应当大于键的个数 N（M>N)。




```
public class LinearProbingHashST<Key, Value> implements UnorderedST<Key
, Value> {
    private int N = 0;
    private int M = 16;
    private Key[] keys;
```

```
    private Value[] values;

    public LinearProbingHashST() {
        init();
    }

    public LinearProbingHashST(int M) {
        this.M = M;
        init();
    }

    private void init() {
        keys = (Key[]) new Object[M];
        values = (Value[]) new Object[M];
    }

    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % M;
    }
}
```

（一）查找

```
public Value get(Key key) {
    for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
        if (keys[i].equals(key))
            return values[i];

    return null;
}
```

（二）插入

```
public void put(Key key, Value value) {
    resize();
    putInternal(key, value);
}

private void putInternal(Key key, Value value) {
    int i;
    for (i = hash(key); keys[i] != null; i = (i + 1) % M)
        if (keys[i].equals(key)) {
            values[i] = value;
```

```
                return;
            }

        keys[i] = key;
        values[i] = value;
        N++;
    }
```

（三）删除

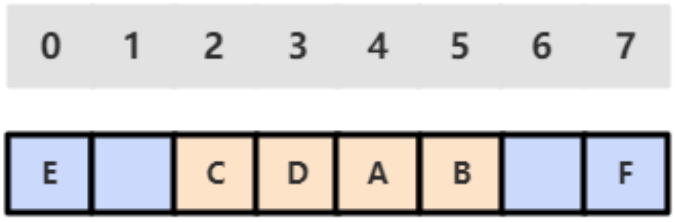
删除操作应当将右侧所有相邻的键值对重新插入散列表中。

```
public void delete(Key key) {
    int i = hash(key);
    while (keys[i] != null && !key.equals(keys[i]))
        i = (i + 1) % M;

    // 不存在，直接返回
    if (keys[i] == null)
        return;

    keys[i] = null;
    values[i] = null;

    // 将之后相连的键值对重新插入
    i = (i + 1) % M;
    while (keys[i] != null) {
        Key keyToRedo = keys[i];
        Value valToRedo = values[i];
        keys[i] = null;
        values[i] = null;
        N--;
        putInternal(keyToRedo, valToRedo);
        i = (i + 1) % M;
    }
    N--;
    resize();
}
```

（四）调整数组大小

线性探测法的成本取决于连续条目的长度，连续条目也叫聚簇。当聚簇很长时，在查找和插入

时也需要进行很多次探测。例如下图中 2\~5 位置就是一个聚簇。




```
private void resize() {
    if (N >= M / 2)
        resize(2 * M);
    else if (N <= M / 8)
        resize(M / 2);
}

private void resize(int cap) {
    LinearProbingHashST<Key, Value> t = new LinearProbingHashST<Key, Value>(cap);
    for (int i = 0; i < M; i++)
        if (keys[i] != null)
            t.putInternal(keys[i], values[i]);

    keys = t.keys;
    values = t.values;
    M = t.M;
}
```

小结

1. 符号表算法比较

算法	插入	查找	是否有序
二分查找实现的有序表	N	logN	yes
二叉查找树	logN	logN	yes

算法	插入	查找	是否有序
2-3 查找树	logN	logN	yes
链表实现的有序表	N	N	no
拉链法实现的散列表	N/M	N/M	no
线性探测法实现的散列表	1	1	no

应当优先考虑散列表，当需要有序性操作时使用红黑树。

2. Java 的符号表实现

- java.util.TreeMap：红黑树
- java.util.HashMap：拉链法的散列表

3. 稀疏向量乘法

当向量为稀疏向量时，可以使用符号表来存储向量中的非 0 索引和值，使得乘法运算只需要对那些非 0 元素进行即可。

```
public class SparseVector {
    private HashMap<Integer, Double> hashMap;

    public SparseVector(double[] vector) {
        hashMap = new HashMap<>();
        for (int i = 0; i < vector.length; i++)
            if (vector[i] != 0)
                hashMap.put(i, vector[i]);
    }

    public double get(int i) {
        return hashMap.getOrDefault(i, 0.0);
    }

    public double dot(SparseVector other) {
        double sum = 0;
        for (int i : hashMap.keySet())
            sum += this.get(i) * other.get(i);
        return sum;
```

```
20.         }
21.     }
```

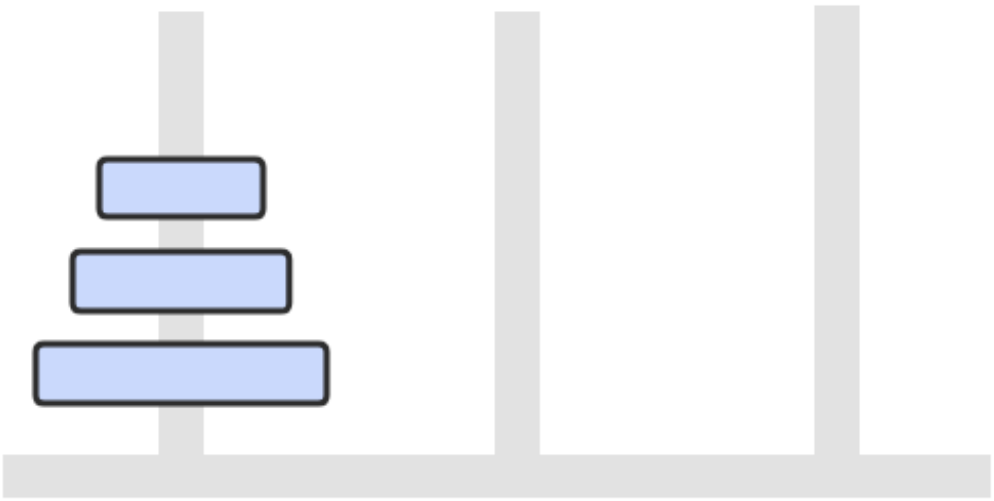
七、其它

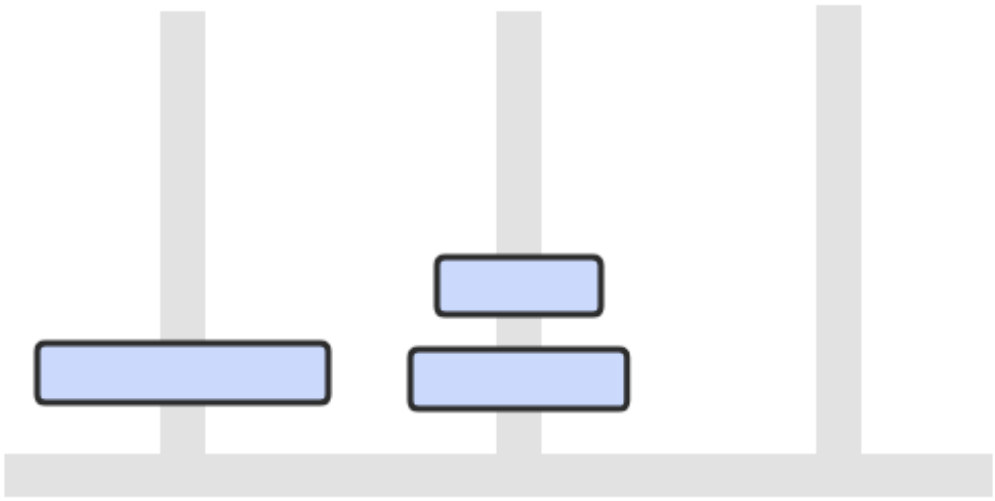
汉诺塔

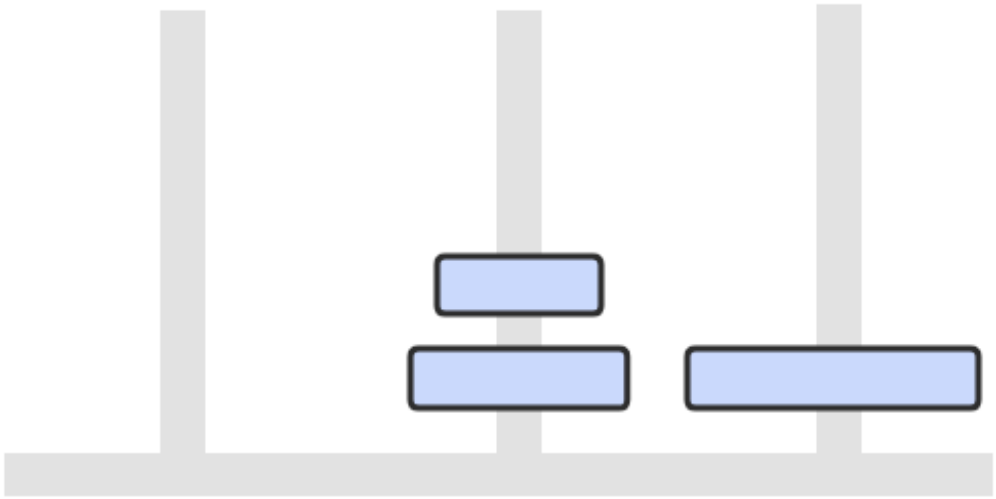
这是一个经典的递归问题，分为三步求解：

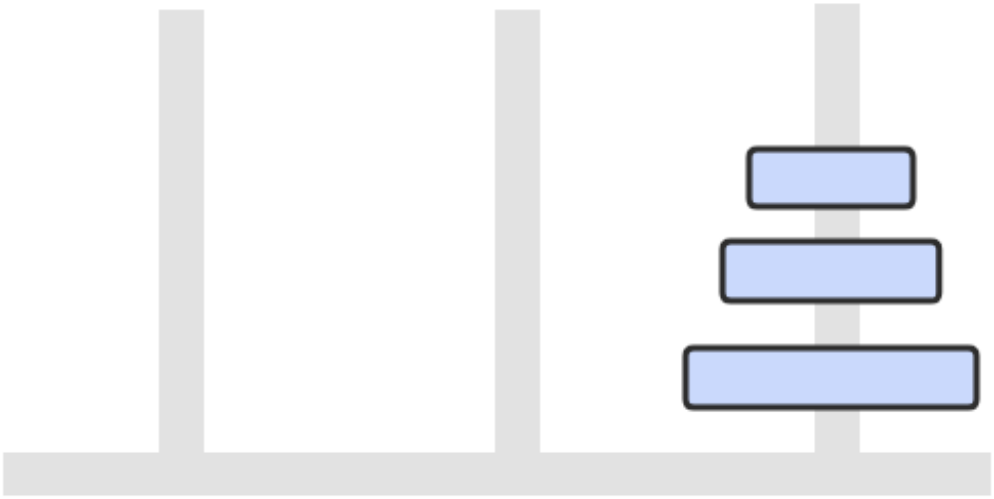
1. 将 n-1 个圆盘从 from -> buffer
2. 将 1 个圆盘从 from -> to
3. 将 n-1 个圆盘从 buffer -> to

如果只有一个圆盘，那么只需要进行一次移动操作，从上面的移动步骤可以知道，n 圆盘需要移动 (n-1)+1+(n-1) = 2n-1 次。









```java
public class Hanoi {
    public static void move(int n, String from, String buffer, String to) {
        if (n == 1) {
            System.out.println("from " + from + " to " + to);
            return;
        }
        move(n - 1, from, to, buffer);
        move(1, from, buffer, to);
        move(n - 1, buffer, from, to);
    }

    public static void main(String[] args) {
        Hanoi.move(3, "H1", "H2", "H3");
    }
}
```

```
from H1 to H3
from H1 to H2
from H3 to H2
from H1 to H3
from H2 to H1
from H2 to H3
from H1 to H3
```

哈夫曼编码

哈夫曼编码根据数据出现的频率对数据进行编码，从而压缩原始数据。

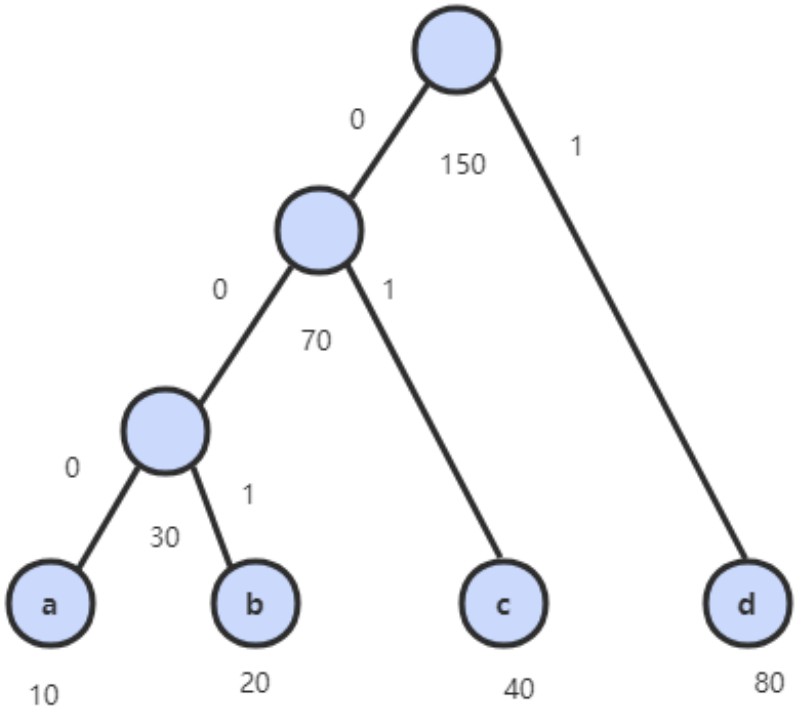
例如对于文本文件，其中各种字符出现的次数如下：

- a : 10
- b : 20
- c : 40
- d : 80

可以将每种字符转换成二进制编码，例如将 a 转换为 00，b 转换为 01，c 转换为 10，d 转换为 11。这是最简单的一种编码方式，没有考虑各个字符的权值（出现频率）。而哈夫曼编码能让出现频率最大的字符编码最短，从而保证最终的编码长度最短。

首先生成一颗哈夫曼树，每次生成过程中选取频率最少的两个节点，生成一个新节点作为它们的父节点，并且新节点的频率为两个节点的和。选取频率最少的原因是，生成过程使得先选取的节点在树的最底层，那么需要的编码长度更长，频率更少可以使得总编码长度更少。

生成编码时，从根节点出发，向左遍历则添加二进制位 0，向右则添加二进制位 1，直到遍历到根节点，根节点代表的字符的编码就是这个路径编码。




```
public class Huffman {

    private class Node implements Comparable<Node> {
        char ch;
        int freq;
        boolean isLeaf;
        Node left, right;

        public Node(char ch, int freq) {
            this.ch = ch;
            this.freq = freq;
            isLeaf = true;
        }

        public Node(Node left, Node right, int freq) {
            this.left = left;
            this.right = right;
            this.freq = freq;
            isLeaf = false;
        }

        @Override
        public int compareTo(Node o) {
```

```
            return this.freq - o.freq;
        }
    }

    public Map<Character, String> encode(Map<Character, Integer> frequencyForChar) {
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
        for (Character c : frequencyForChar.keySet()) {
            priorityQueue.add(new Node(c, frequencyForChar.get(c)));
        }
        while (priorityQueue.size() != 1) {
            Node node1 = priorityQueue.poll();
            Node node2 = priorityQueue.poll();
            priorityQueue.add(new Node(node1, node2, node1.freq + node2.freq));
        }
        return encode(priorityQueue.poll());
    }

    private Map<Character, String> encode(Node root) {
        Map<Character, String> encodingForChar = new HashMap<>();
        encode(root, "", encodingForChar);
        return encodingForChar;
    }

    private void encode(Node node, String encoding, Map<Character, String> encodingForChar) {
        if (node.isLeaf) {
            encodingForChar.put(node.ch, encoding);
            return;
        }
        encode(node.left, encoding + '0', encodingForChar);
        encode(node.right, encoding + '1', encodingForChar);
    }
}
```

参考资料

- Sedgewick, Robert, and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 2011.

github: https://github.com/sjsdfg/Interview-Notebook-PDF