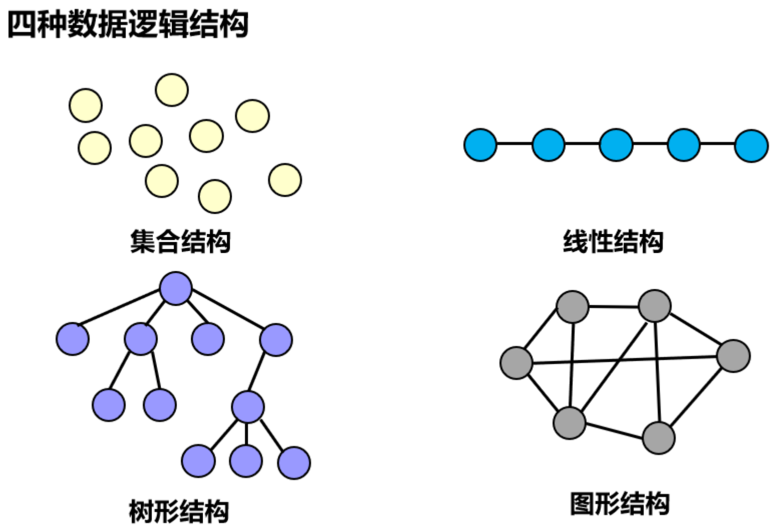
绪论

1. 数据结构：相互之间存在一种或多种特定关系的数据元素的集合
2. 数据结构三元素：**逻辑结构、存储结构、数据的运算**
   - 逻辑结构：元素集合和关系({<a,b>,<b,c>,...})
     - 

     - 四种可以总结为：线性结构、非线性结构
   - 存储结构：**顺序存储、链式存储、索引存储、散列存储**
   - 同一逻辑结构可以对应不同的存储结构
   - 邻接表和邻接矩阵都是既可以顺序存储也可以链式存储
3. 抽象数据类型三元组(D,R,P)
   - D-数据元素的集合、R-关系的集合、P-操作的集合
4. 渐进上界O

   渐进下界Ω

   渐进紧界Θ

   O与Θ经常混用

   时间复杂度本身是空间复杂度的上界
5. 算法复杂度分析实例

```
int SumI (int A[], int n){
   return n < 1 ? 0 : sumI(A, n-1)+A[n-1];
}
```

复杂度递推方程：

$$T(n) = T(n-1) + O(1)$$
$$T(0) = O(1)$$

求解：

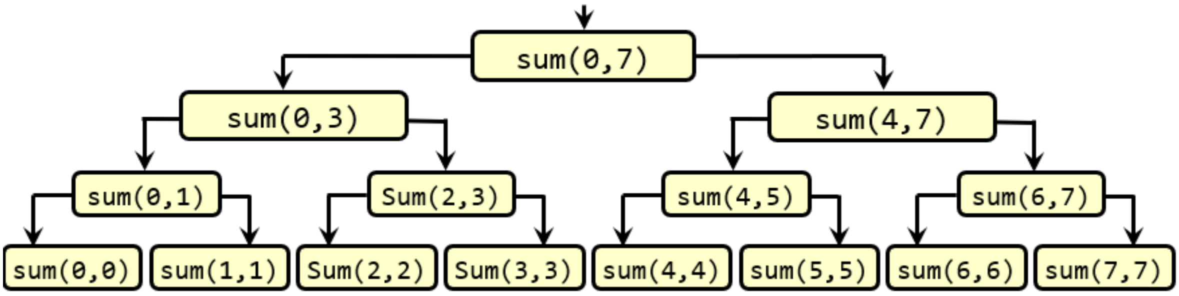
$$\begin{aligned} T(n) - n &= T(n-1) - (n-1) \\ &= T(1) - 1 = T(0) \end{aligned}$$
$$T(n) = O(1) + n = O(n)$$

```
sum(int A[], int lo, int hi){   // 区间范围A[lo, hi];
    if(lo==hi)
        return A[lo];
    int mi = (lo + hi) >> 1;              // 入口形式sum(A, 0, n-1)
    return sum(A, lo, mi) + sum(A, mi+1, hi);
}
```

- **复杂度计算方法：复杂度递推方程**

$$T(n) = 2 \times T(n/2) + O(1),\quad T(1) = O(1)$$
$$T(n) + c = 2 \times (T(n/2)+c) = \dots = 2^{\log n}(T(1)+c) = n(T(1)+c)$$
$$T(n) = O(n)$$

- **递归调用实例图：计算8元素组成的数组A[]元素之和**



时间复杂度： $T(n) = O(1) \times (2^0 + 2^1 + 2^2 + \dots + 2^{\log n})$

$= O(1) \times (2^{1+\log n} - 1) = O(n)$

- **递归实现：**

```
int fib(n) {return (n<2) ? n : fib(n-1)+fib(n-2);}
```

- **复杂度：** $T(n) = T(n-1) + T(n-2) + 1;\ T(0) = T(1) = 1;$

令 $S(n) = (T(n) + 1) / 2$

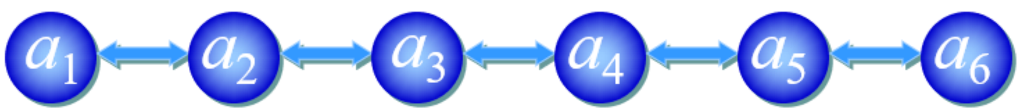
则 $S(0) = (T(0)+1)/2 = 1 = fib(1),\ S(1) = (T(1)+1)/2 = 1 = fib(2)$

故 $S(n) = S(n-1) + S(n-2) = fib(n+1)$

$T(n) = 2*S(n) - 1 = 2*fib(n+1) - 1 = O(fib(n+1))$

$= O(\phi^{n+1})$

向量

1. 线性表：是由n (≥0) 个数据元素的有限序列，记作(a0, a1, ..., an-1)
    - 

    - 除第一个和最后一个节点，每个节点有且仅有一个直接前驱/后继
    - 是一种逻辑结构，既可以链式存储，也可以顺序存储

2. 向量：线性表基于数组的存储表示

3. 向量ADT接口

■ 向量运算

- ✓ 建立[构造函数]：建立向量数据结构
- ✓ 清除[析构函数]：把某个指定的数据结构置为空
- ✓ 求长[size()]：报告向量当前的规模
- ✓ 判空[isempty]：判定向量是否为空
- ✓ 判满[isfull]：判定向量是否达到存储最大允许容量
- ✓ 获取[get(r)]：获取向量中秩为r的数据元素
- ✓ 更新[put(r,e)]：用元素e替换秩为r元素的数值
- ✓ 插入[insert(r,e)]：e作为秩为r元素插入，原后继元素一次后移
- ✓ 删除[remove(r)]：删除秩为r的元素，返回该元素中原存放的对象
- ✓ 查找[find(e)]：查找等于e且秩最大的元素
- ✓ 遍历[traverse]：遍历向量中所有元素，处理方法由函数对象指定
- ✓ 判序[disordered()]：判断所有元素是否已按非降序排序
- ✓ 排序[sort()]：调整各元素的位置，使之按非降序排序
- ✓ 去重[deduplicate()]：剔除重复元素

■ 针对*有序向量*的额外运算

- ✓ 去重[unquify()]：剔除重复元素
- ✓ 查找[search(e)]：查找目标元素e，返回不大于e且秩最大的元素

有元素值返回的接口：get、remove、find（秩最大，倒着找，找不到返回-1）、search、disordered（返回相邻元素逆序对个数，排好序返回0）

4. 扩容时间复杂度分析（扩容要考虑复制）

- 每次翻倍：总复杂度2N，均摊2
- 每次增加x：总复杂度$N^2/2x$，均摊$N/2x$
- insert时需要调用expand()，判断size有没有超过capacity
- remove时需要调用shrink()，判断有没有必要缩容

5. 置乱器

■ 置乱器

```
template <typename T> void permute ( Vector<T>& V )
//随机置乱向量，使各元素等概率出现于各位置
   for ( int i = V.size(); i > 0; i-- ) //自后向前
      swap ( V[i - 1], V[rand() % i] );
      //V[i - 1]与V[0, i)中某一随机元素交换
}
```

6. 唯一化deduplicate（注意，当前元素最多与前面的一个元素重复，因此复杂度仍为O(n^2)）

■ 唯一化

```
template <typename T>
int Vector<T>::deduplicate() {              //删除无序向量中重复元素
   int oldSize = _size;                     //记录原规模
   Rank i = 1;                              //从_elem[1]开始
   while ( i < _size )                      //自前向后逐一考查各元素_elem[i]
      ( find ( _elem[i], 0, i ) < 0 ) ? //在其前缀中找与之雷同者（至多一个）
      i++ : remove ( i ); //若无雷同则继续考查其后继，否则删除雷同者
   return oldSize - _size; //向量规模变化量，即被删除元素总数
}
```

7. 归并排序

```
static void merge(int data[], int first, int mid, int last, int sorted[]){
    int i = first, j = mid;
    int k = 0;
    while (i < mid && j < last)
        if (data[i] < data[j])              // 归并
            sorted[k++] = data[i++];
        else
            sorted[k++] = data[j++];
    while (i < mid) sorted[k++] = data[i++];    // 拷贝两个子序列中的剩余元素
    while (j < last) sorted[k++] = data[j++];
    for (int v = 0; v < k; v++)
        data[first + v] = sorted[v];        // 将排序后结果覆盖原数组
}
static void mergeSort(int data[], int first, int last, int sorted[]){
    if (first + 1 < last){
        int mid = (first + last) / 2;
        mergeSort(data, first, mid, sorted);
        mergeSort(data, mid, last, sorted);
        merge(data, first, mid, last, sorted);
    }
}
```

- ✓ 设排序时间为T(n)
- ✓ 对长度为n的向量归并排序，需完成2长度为n/2向量的归并排序和一两路归并:
  $T(n) = 2*T(n/2) + O(n)$
- ✓ 边界条件:
  $T(1) = 1$
- ✓ 可以解得:
  $T(n) = O(n\log n)$

对包含20个关键字序列进行归并排序，共需要进行 [填空1] 趟归并。//5

8. 插入排序

```
template <typename T>
void Vector<T>::insertSort(Rank lo, Rank hi) { //assert: 0 < lo <= hi <= size
    for (int j = lo+1; j < hi; j++){ // 循环新插入的元素
        T key = _elem[j];             // 缓存新插入元素
        int i = j - 1;                // 已排序序列最后一个元素
        while (i >= lo && _elem[i] > key){ // 对已排序序列从后往前比较
            _elem[i + 1] = _elem[i];    // 大于新插入元素则往后移
            i--;
        }
        _elem[i + 1] = key;   //插入新元素
    }
}
```

9. 冒泡排序

```
template <typename T>
void Vector<T>::bubbleSort(Rank lo, Rank hi) {
        int times = 0; bool exchange = true; // 从第一趟开始
        int nSort = hi - lo;
        while (times < nSort && exchange) {
                exchange = false;       // 某趟是否有交换的标志，初始为无交换
                for (int j = hi-1; j > lo + times; j--)
                                        //从最后元素开始到第一个未排序元素
                        if (_elem[j - 1]>_elem[j]) {//若需要交换则置换元素
                                T temp = _elem[j - 1];
                                _elem[j - 1] = _elem[j];
                                _elem[j] = temp;
                                exchange = true;
                        }
                times++;
        }
}
```

遇到相邻大小一样时，不进行交换，因此保证起泡排序为稳定的排序方法，复杂度O(n^2)

10. 选择排序：**不稳定！！！**

```
template <typename T>
void Vector<T>::selectSort(Rank lo, Rank hi) {
        for (Rank i = hi - 1; i > lo; i--) { // 从后往前
                int max = i;
                for (Rank j = 0; j < i + 1; j++) {
                                        // 遍历前面未排序，选择最大元素
                        if (_elem[j] > _elem[max])
                                max = j;
                }
                if (max != i) {                 // 交换
                        T temp = _elem[i];
                        _elem[i] = _elem[max];
                        _elem[max] = temp;
                }
        }
}
```

11. 排序算法比较

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
插入	O(n)	O(n^2)	O(n^2)	O(1)	稳定
冒泡	O(n)	O(n^2)	O(n^2)	O(1)	稳定
选择	O(n^2)	O(n^2)	O(n^2)	O(1)	不稳定
归并	O(nlgn)	O(nlgn)	O(nlgn)	O(n)	稳定

12. Fibonacci查找

fib(n)-1=(fib(n-1)-1)+(fib(n-2)-1)+1，其中fib(n-1)-1是左边查找范围，fib(n-2)-1是右边查找范围

13. ~~回字的八种...~~二分查找的三种写法

- 重复元素随机选取一个

```
// 二分查找算法（版本A）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
template <typename T>
static Rank binSearch ( T* A, T const& e, Rank lo, Rank hi ) {
    while ( lo < hi ) {                                      //每步迭代可能做两次比较判断，三个分支
        Rank mi = ( lo + hi ) >> 1;                          //以中点为轴点
        if( e < A[mi] ) hi = mi;                             //深入前半段[lo, mi)继续查找
        else if ( A[mi] < e ) lo = mi + 1;                   //深入后半段(mi, hi)继续查找
        else return mi;                                      //在mi处命中
    }                                                        //成功查找可以提前终止
    return -1;                                               //查找失败
} //多个命中元素时，不能保证返回秩最大者；查找失败时，简单地返回-1，而不能指示失败位置
```

- 重复元素最大秩，否则返回-1

```
// 二分查找算法（版本B）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
template <typename T>
static Rank binSearch ( T* A, T const& e, Rank lo, Rank hi ) {
    while ( 1 < hi - lo ) {                  //每步迭代仅有两个分支；成功查找不能提前终止
        Rank mi = ( lo + hi ) >> 1;          //以中点为轴点
        ( e < A[mi] ) ? hi = mi : lo = mi;   //经比较后确定深入[lo, mi)或[mi, hi)
    }                                        //出口时hi = lo + 1，查找区间仅含一个元素A[lo]
    return ( e == A[lo] ) ? lo : -1 ;        //查找成功时返回对应的秩；否则统一返回-1
}
```

- 返回不大于key的元素最大秩

```
template <typename T>
static Rank binSearch ( T* A, T const& e, Rank lo, Rank hi ) {
    while ( lo < hi ) {                            //每步迭代仅需做一次比较判断，有两个分支
        Rank mi = ( lo + hi ) >> 1;                //以中点为轴点
        ( e < A[mi] ) ? hi = mi : lo = mi + 1;     //经比较后确定深入[lo, mi)或(mi, hi)
    }                                              //成功查找不能提前终止
    return --lo; //循环结束时，lo为大于e的元素的最小秩，故lo - 1即不大于e的元素的最大秩
} //有多个命中元素时，总能保证返回秩最大者；查找失败时，能够返回失败的位置
```

- 总结（注意输入的l~r是左闭右开区间）

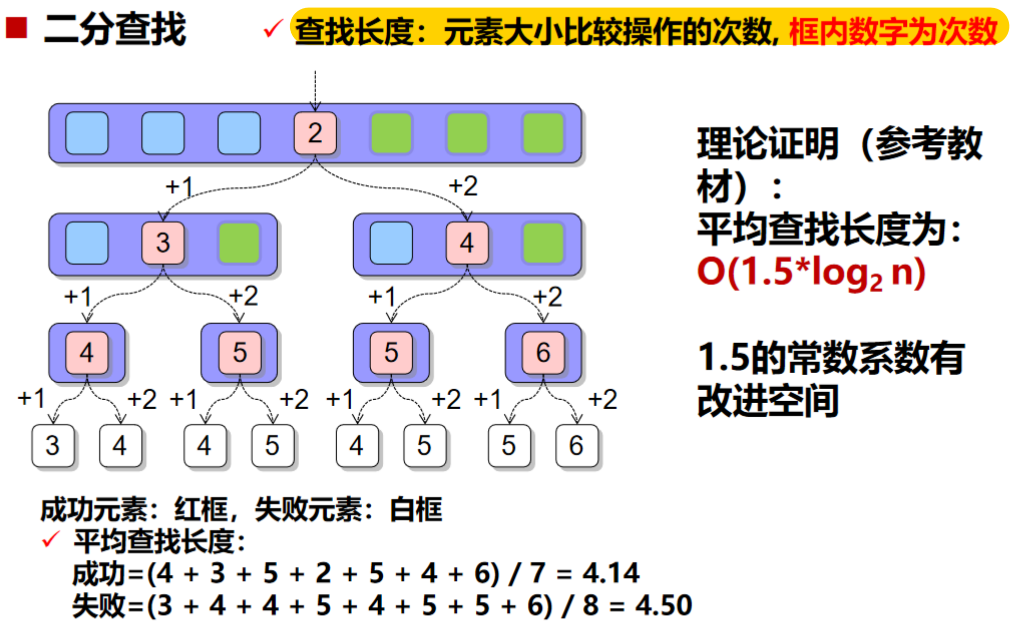
```
int search1(int a[],int l,int r,int key)
{
    while(l<r)
    {
        int mid=l+r>>1;
        if(a[mid]>key)r=mid;
        else l=mid+1;
    }
    /*  或者写作
        if(a[mid]<=key)l=mid+1;
        else r=mid;
    */
    return l-1;//与r-1相等
}// 不大于的最大秩(大于的最小秩则return l或return r)
```

```c
int search2(int a[],int l,int r,int key)
{
    while(l<r)
    {
        int mid=l+r>>1;
        if(a[mid]>=key)r=mid;
        else l=mid+1;
    }
    /*  或者写作
        if(a[mid]<key)l=mid+1;
        else r=mid;
    */
    return l-1;//与r-1相等
}// 小于的最大秩(不小于的最小秩则return l或return r)
```

14. 对于长度为9的有序顺序表，若采用折半搜索，在等概率情况下搜索成功的平均**搜索长度**为（ ）//25/9

    解析：这里我们采用的折半搜索是三段式，mid命中则返回，命不中则递归（不考虑长度是否满足循环条件，直接递归）直至命中，九个元素的搜索次数分别为323413234

15. **查找长度**：元素比较次数



列表

1. 线性表- 顺序存储：静态/动态空间；链式存储：动态空间
2. 插入（表头first是空节点）

```
bool List::Insert (int i, int& x) //将新元素x插入在链表中第 i 个结点之后
{    ListNode* current = Locate(i);
     if (current == NULL) return false;      //无插入位置
     ListNode* newNode = new ListNode(x);    //创建新结点
     newNode->next = current->next;          //链入, 核心操作
     current->next = newNode;
     return true;    //插入成功
}; //在第 i 个结点之后插入, 只需定位第 i 个结点, i=0则插在表头结点之后,
作为新的首元。
LinkNode* List::Locate (int i) {  //函数返回表中第 i 个元素的地址, 若i<0或i
超出表中结点个数, 则返回NULL。i=0返回表头结点地址
 if (i < 0) return NULL;                        //i不合理
 ListNode* current = first;  int k = 0;         // 见下
 while (current != NULL && k < i )
        { current = current->next;k++; }
     return current;                           //返回第 i 号结点地址或NULL
}; // 初始为*current = first->link; int k = 1; 是否可以
    //考察 i=0 情形如何, 插入和删除有时需要定位i=0或i-1
```

3. 删除

```
bool List::Remove (int i, T& x ) {
                         //删除链表第i个元素, 通过引用参数x返回元素值
    ListNode* current = Locate(i-1);
    if ( current == NULL)
         return false;                         //表中无第i个元素, 无法删除
    ListNode* del = current->next;                               //核心操作
    current->next = del->next;
    x = del->data;
    delete del;
    return true;
};                                        //删除第i个结点, 需定位第i-1个结点
```

L为带表头节点的链表, P不为首元素, 请排序以下代码删除P的直接前驱 [填空1]

(a)P->next = P->next->next;
(b)While(P->next->next!=Q) P=P->next;
(c)Q=P;
(d)free(Q);
(e)Q=P->next;
(f)P=L;

//cfbead

4. 析构

```
template <typename T> List<T>::~List() //列表析构器
{
     int oldSize = _size;
     while ( 0 < _size ) remove ( header->succ );
     //反复删除首节点, 直至列表变空
     delete header;
     delete trailer;
} //清空列表, 释放头、尾哨兵节点
```

5. deduplicate:O(n^2)

uniquify:O(n)，考察相邻的节点对，相同则删去后面那个

search:O(n^2)

插入排序：search+insert

选择排序：末尾插入最大点并删去原始最大点

6. 双向链表（并非是循环链表！！！）
   - ADT接口

     首节点插入[insertAsFisrt(e)]：将e作为首节点插入
     末节点插入[insertAsLast(e)]：将e作为末节点插入
     前插入[insertB(p,e)]：将p作为当前节点的前驱插入
     后插入[insertA(p,e)]：将p作为当前节点的后继插入

   - 哨兵节点(header+tailer，两个不可见的空节点)：外部不可见；可见的任何一个节点都有后继和前驱
   - 向前插入

```
template <typename T> ListNodePosi(T) ListNode<T>::insertAsPred ( T const& e ) {
   ListNodePosi(T) x = new ListNode ( e, pred, this ); //创建新节点
   pred->succ = x; pred = x; //设置正向链接
   return x; //返回新节点的位置
}
```

   - 删除（异于单向链表通过前驱删除节点）

```
template <typename T> T List<T>::remove ( ListNodePosi(T) p ) {
   T e = p->data;                          //备份待删除节点的数值（假定T类型可直接赋值）
   p->pred->succ = p->succ; p->succ->pred = p->pred;      //后继、前驱
   delete p; _size--;                              //释放节点，更新规模
   return e;                                        //返回备份的数值
}
```

7.

数据结构	无序向量	无序列表	有序向量	有序列表
search(x)	O(n)	O(n)	O(logn) 二分查找	O(n)
insert(x)	O(1) 末端插入	O(1) 末端插入	O(n) 查找|移位	O(n) 查找
remove(x)	O(n) 查找|移位	O(n) 查找	O(n) 查找|移位	O(n) 查找

栈

1. 括号匹配

策略：后开先闭！

- ✓ 从左往右扫描
- ✓ 碰到开（左）括号，入栈
- ✓ 碰到闭（右）括号，若栈顶不是对应的开括号，返回错误；若是，则出栈
- ✓ 最后必须栈空

2. 给出波兰表达式、逆波兰表达式，借助栈求值：注意是 后出栈的数 op 先出栈的数
3. 给出中缀表达式，借助栈求逆波兰表达式：栈内放运算符，遇到比自己优先级大的和同级的弹出，优先级小的则将自己压入栈

操作符优先级：

1) 阶乘运算符

2) 指数运算符

3) 乘除运算符

4) 加减运算符

5) 最低优先级：),], }

注意：括号是来打酱油的，前括号单纯入栈，不会造成任何操作符弹出，后括号只会弹出栈顶第一个前括号之前所有的操作符

注意：单目运算符！的运算优先级最高，入栈后下一刻，后面的运算符就会让它出栈

（（A+B）× C−D）× E -> AB+C×D-E×

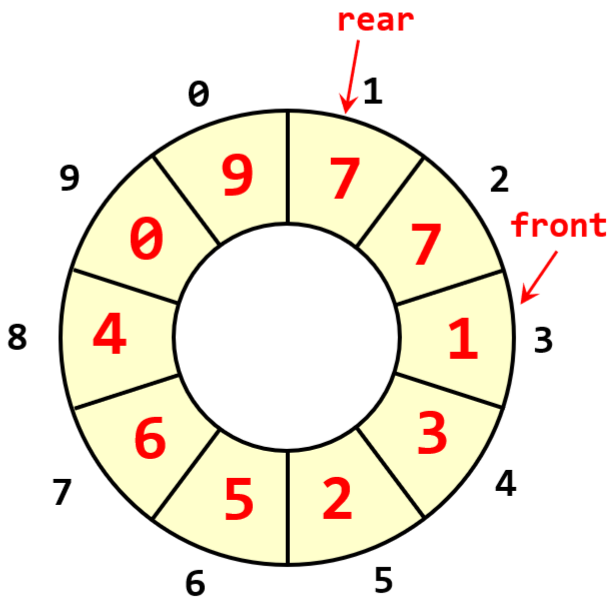
9+(3−1)×3−10÷2-> 9 3 1 - 3 ×10 2 ÷ −

4. 栈混洗：通过出栈入栈获得一个新序列. 可能的序列总数是卡特兰数 $C_{2n}^{n} - C_{2n}^{n-1} = C_{2n}^{n}/(n+1)$
   - 卡特兰数性质：$h(n)=h(n-1)h(0)+h(n-2)h(1)+...+h(1)h(n-2)+h(0)h(n-1)$
   - **12个高矮不同的人,排成两排,每排必须是从矮到高排列,而且第二排比对应的第一排的人高,问排列方式有多少种？[填空1]**

     //132
5. 用链表实现栈，头指针不需要留空

队列

1. 数组实现- 循环队列（注意front->rear是顺时针）



2. 链表实现（加入尾指针，尾部访问和删除O(1)；头指针和尾指针不需要空白）

树

1. 节点的度：某节点的孩子总数；树的度：树节点的度的最大值

   叶节点：没有孩子的节点

   内部节点：除叶节点之外的节点

   堂兄弟：具有相同的祖父

   节点深度：从根到该节点的边数

   节点高度：从该节点到该点的 叶 子节点的最大边数；树的高度：树根的高度（未特别声明，**单节点的树高度为0，空树高度为-1**）

   祖先/后代

2. 二叉树的层：具有相同深度的节点所在的层相同

   完全二叉树：除了最后一层外，其他各层全是满的，并且最后一层的节点尽可能往左靠

   满二叉树：所有叶子节点都处于最底层的二叉树（树中所有层都是满的）

   真二叉树：每个节点的孩子数目为0个或2个

   平衡二叉树：树中任意节点的左右子树的高度差不超过k（通常k为1）

   n个节点的完全二叉树高度：h=$\lfloor log_2 n \rfloor$

3. 树中节点数$N = 1 + \sum_{i=1}^{\infty} in_i$，其中$n_i$是度数为i的节点个数；特别地，对于二叉树，

   $n_0 = 1 + n_2$

4. 多叉树ADT接口

节点成员函数	功能
root()	节点对应的根节点
parent()	节点的父节点
firstChild()	节点的长子
nextSibling()	节点的兄弟
insert(i,e)	将e作为节点的第i个孩子插入
remove(i)	删除第i个孩子及其后代
traverse()	遍历

5. 父亲表示法：查兄弟和儿子不理想

   儿子表示法：查兄弟和父亲不理想

   父亲+孩子：动态维护复杂

   长子+兄弟：好！！！！！！！！！！

6. 多叉树转二叉树：左孩子-长子，右孩子-下一个兄弟

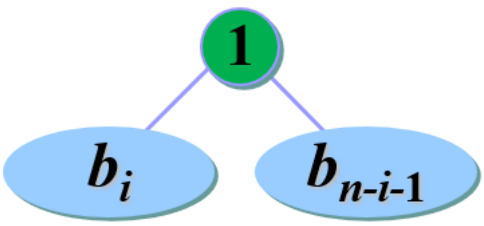
   二叉树转多叉树：自己揣摩吧，不难

   森林转二叉树：把各个树根节点当兄弟，进行多叉树转二叉树操作

7. 二叉树的棵树

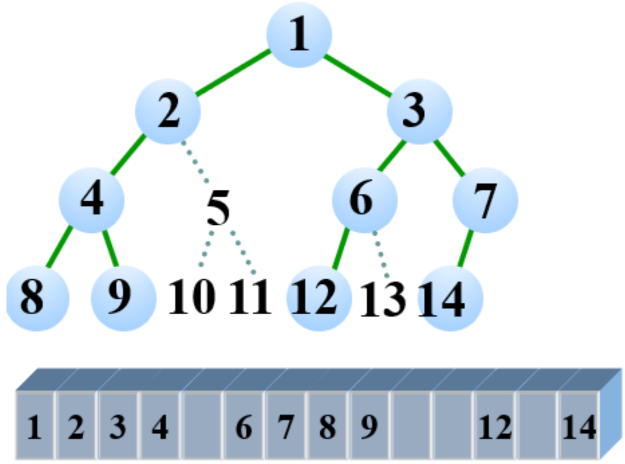
■ 二叉树的棵树

- ✓ n节点二叉树棵树形态数目?
- ✓ 卡特兰数



$$b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1} = \frac{(2n)!}{(n+1)! \cdot n!}$$

8.



一般二叉树顺序表示

9. 二叉树的遍历：bfs（层次遍历）、dfs（先/中/后序遍历）
    - 先序：根左右；中序：左根右；后序：左右根
    - bfs：头结点出列，放入两个孩子
10. 重构二叉树
    - 前序/后序+中序：可重构
    - 前序+后序：不可重构，如果一个子树为空，无法分辨是左子树还是右子树；除非已知是真二叉树
    - 前序遍历给定，问有多少中序遍历可能：卡特兰数
11. 树的路径长度(PL)：所有节点从根到该节点的路径长度之和

    带权路径长度(WPL)

    节点的带权路径长度：节点路径长度与节点权重之积；树的带权路径长度：所有叶子节点的带权路径长度之和
12. 哈夫曼编码：哈夫曼树左孩子路径赋1，右孩子路径赋0，前缀无歧义

二叉搜索树

1. 二叉搜索树（二叉排序树）
    - 空树或任意节点的左子树（若非空）的关键码都小于等于该节点关键码且任意节点的右子树（若非空）的关键码都大于等于该节点关键码的树
    - 任意一棵二叉树是二叉搜索树，当且仅当其中序遍历序列单调非降
2. 插入：O(h)，先查询，查询到则return false，否则作为叶子节点插入. 运用_ hot，_ hot指向要插入部位的父亲，new时直接以此为父亲. _hot起到提醒更新树高的意义

    删除O(h)

    - 若叶子节点，则直接删去，hot取父亲
    - 若左子树为空，则右子树直接接上来；右子树为空，左子树接，hot取父亲
    - 若左右子树均不为空，则寻找直接后继；对于此种情况，直接后继是右子树的最左，设为v'（右子树最左），然后将当前删除节点的值v替换为v'，在右子树中递归删除v'，hot取叶子节点的父亲
        - 直接后继：右子树的最左，没右子树则寻找将包含当前节点的子树作为左子树的点（向左上直到右上有路）

```
template <typename T> BinNodePosi(T)
BinNode<T>::succc() { //定位节点v直接后继
    BinNodePosi(T) s = this; //记录后继的临时变量
       if ( rc ) {
    //若有右孩子，则直接后继必在右子树中，具体地就是
           s = rc; //右子树中
           while ( HasLChild ( *s ) )
              s = s->lc; //最靠左（最小）的节点
       }
       else {
//否则，直接后继应是“将当前节点包含于其左子树中的
最低祖先”，具体地就是
          while ( IsRChild ( *s ) )
             s = s->parent;
              //逆向地沿右向分支不断朝左上方移动
          s = s->parent;
//最后再朝右上方移动一步，即抵达直接后继（如果存在）
       }
       return s;
}
```

3. 随机生成：关键码随机排列并插入，平均高度Θ(logn)

   随机组成：树形个数为卡特兰数，各形态概率均等，可证明二叉搜索树平均高度Θ(n^(1/2))

   等价：中序遍历相同

   平衡因子：左右子树高度差，balFac(v)=height(lc(v))-height(rc(v))

4. AVL调整

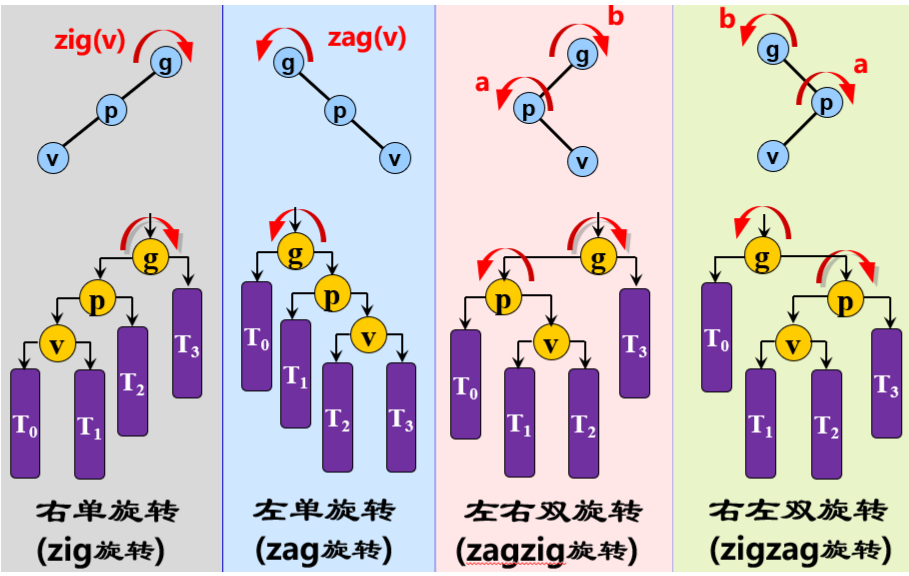
■ **平衡化旋转分类（四种情况）**

✓ **左单旋(zag旋转，逆时针旋转)**

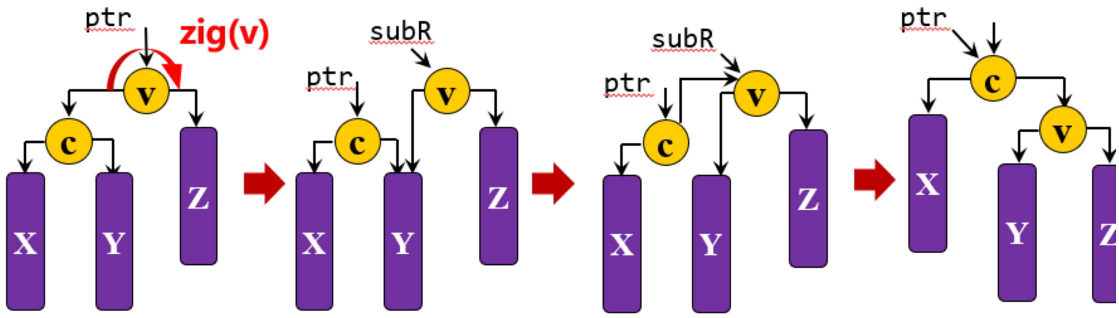
✓ **右单旋(zig旋转，顺时针旋转)**

✓ **先左后右双旋(zagzig旋转)**

✓ **先右后左双旋(zigzag旋转)**



■ 核心操作：左单旋与右单旋

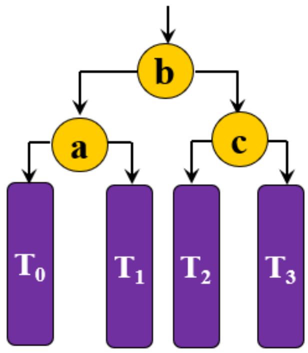



```
void RotateR(Node *& ptr) {//左子树比右子树高，旋转后新根在ptr
    Node *subR = ptr;//要右旋转的结点
    ptr = subR->left;
    subR->left = ptr->right;//转移ptr右边负载
    ptr->right = subR;//ptr成为新根
    ptr->bf = subR->bf = 0; // 修改v和c的平衡因子
};
```


3+4 组装

■ 统一重平衡算法实现

```
template <typename T> BinNodePosi(T) BST<T>::connect34 (
   BinNodePosi(T) a, BinNodePosi(T) b, BinNodePosi(T) c,
   BinNodePosi(T) T0, BinNodePosi(T) T1, BinNodePosi(T) T2,
BinNodePosi(T) T3
) { // 在确定a,b,c节点及四棵子树情况下组装重构平衡子树
   a->lc = T0; if ( T0 ) T0->parent = a;
   a->rc = T1; if ( T1 ) T1->parent = a;
   updateHeight ( a ); // 组装左子树并更新高度
   c->lc = T2; if ( T2 ) T2->parent = c;
   c->rc = T3; if ( T3 ) T3->parent = c;
   updateHeight ( c ); // 组装右子树并更新高度
   b->lc = a; a->parent = b;
   b->rc = c; c->parent = b;
   updateHeight ( b ); // 组装子树根节点
   return b; //返回该子树新的根节点
}
```



■ 统一重平衡算法实现

```
template <typename T> BinNodePosi(T)
BST<T>::rotateAt ( BinNodePosi(T) v ) { //v为非空孙辈节点
    BinNodePosi(T) p = v->parent;
    BinNodePosi(T) g = p->parent; //视v、p和g相对位置分四种情况
    if ( IsLChild ( *p ) ) /* zig */
      if ( IsLChild ( *v ) ) { /* zig-zig */
        p->parent = g->parent; //向上联接
        return connect34 (v, p, g, v->lc, v->rc, p->rc, g->rc);
      }
      else { /* zig-zag */
        v->parent = g->parent; //向上联接
        return connect34 (p, v, g, p->lc, v->lc, v->rc, g->rc);
      }
    else  /* zag */
       … … … …
}
```

插入：插入最多34connect一次（最多调整两次），但最多nlogn回溯

```
template <typename T> BinNodePosi(T) AVL<T>::insert ( const T& e ) {
   BinNodePosi(T) & x = search ( e ); if ( x ) return x; //确认目标节点不存在
   BinNodePosi(T) xx = x = new BinNode<T> ( e, _hot ); _size++; //插入新节点x
// 此时，x的父亲_hot若增高，则其祖父有可能失衡
   for ( BinNodePosi(T) g = _hot; g; g = g->parent ) {
                                       //从x之父出发向上，逐层检查各代祖先g
      if ( !AvlBalanced ( *g ) ) { //发现g失衡
         FromParentTo ( *g ) = rotateAt (tallerChild(tallerChild(g)));
             //3+4组装重新接入原树，FromParentTo(*g)获得g的父亲节点到g的指针
         break; //g复衡后，局部子树高度必然复原；其祖先亦必如此，故调整随即结束
      } else //否则（g依然平衡），只需简单地
         updateHeight ( g ); //更新其高度（注意：即便g未失衡，高度亦可能增加）
   } //至多只需一次调整；若果真做过调整，则全树高度必然复原
   return xx; //返回新节点位置
} //无论e是否存在于原树中，总有AVL::insert(e)->data == e
```

删除：删除最多34connect nlogn次，回溯nlogn

```
template <typename T> bool AVL<T>::remove ( const T& e ) {
   BinNodePosi(T) & x = search ( e ); if ( !x ) return false; //确认目标存在
   removeAt ( x, _hot ); _size--;
                      //先按BST规则删除之（此后，原节点之父_hot及其祖先均可能失衡）
   for ( BinNodePosi(T) g = _hot; g; g = g->parent ) {
                                        //从_hot出发向上，逐层检查各代祖先g
      if ( !AvlBalanced ( *g ) )                  //一旦发现g失衡
         g = FromParentTo (*g)=rotateAt(tallerChild(tallerChild(g))); //原父亲
      updateHeight ( g ); //并更新其高度（注意：即便g未失衡，高度亦可能降低）
   }          //可能需做Omega(logn)次调整——无论是否做过调整，全树高度均可能降低
   return true; //删除成功
} //若目标节点存在且被删除，返回true；否则返回false
```

网址：https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

尝试：30,20,10,40,50,60,70,100,90,80,65,63,25

高级搜索树

1. splay：除了拥有二叉查找树的性质之外，伸展树还具有的一个特点是：当某个节点被访问时，伸展树会通过旋转使该节点成为树根。

简单splay、双层splay

2. 局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用
3. m阶B-树，节点允许分支数为[$\lceil \boldsymbol{m/2} \rceil$, $\boldsymbol{m}$]，节点允许储存数据个数为[$\lceil \boldsymbol{m/2} \rceil$-1, $\boldsymbol{m}$-1] (根节点数据下限为1). 通常以($\lceil \boldsymbol{m/2} \rceil$, $\boldsymbol{m}$)树命名B树

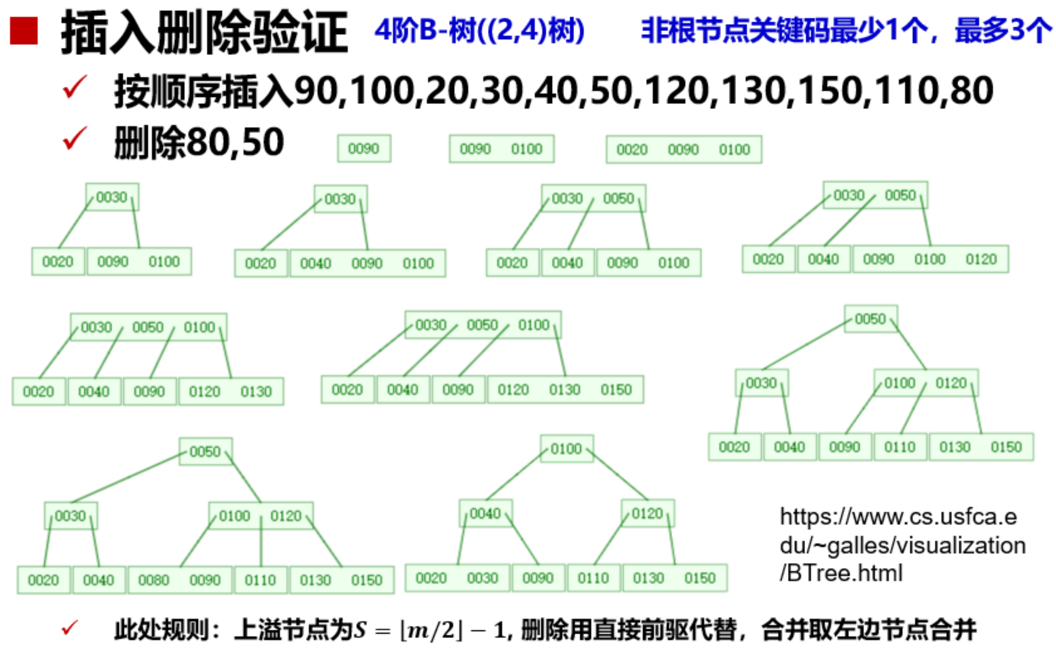
   第k层关键码总数N_k，其上一层分支总数L_{k-1}，本层分支总数L_k，满足：$L_{k-1}+N_k=L_k$ ，可递推得$L_{-1}+N_0+N_1+N_2+...+N_k=L_k$

   B-树的叶子结点均在最低点

   注意节点数和关键码数的区别
4. 设关键码总数为N，则有$2\times\lceil m/2\rceil^{h-1}\leq N+1\leq m^h$

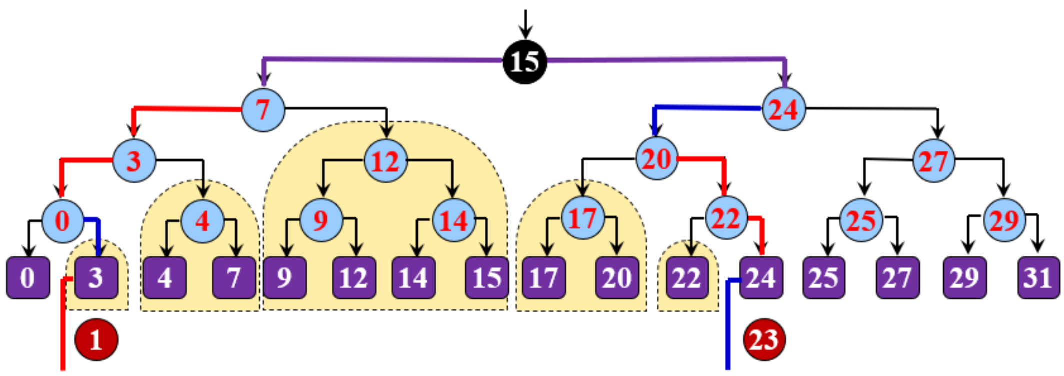
   $\Rightarrow \log_m(N+1)\leq h\leq 1+\log_{\lceil m/2\rceil}\lfloor\frac{N+1}{2}\rfloor$ （不要画图qwq）
5. 插入：未出现的值插入在底层然后上溢（注意初值时的处理）

   删除：若其在叶子结点直接删，否则找直接后继删；删完下溢

   - 下溢：1）有左兄弟且可以借，就旋转左兄弟最大值过来；2）有右兄弟可以借，就旋转右兄弟最小值过来；3）均不可借，就借父节点一个值拼合左兄弟（若没左兄弟则拼合有兄弟，必有一个兄弟）和该节点，继续递归下溢父节点；4）下溢到根节点，若根节点被删空，则删除根节点，用唯一儿子替代根节点，高度--
   - 

6. k-d tree
   - 一维范围查询（也不知道为什么要用这么蠢的方法qwq）

     先找lca，然后分成两边进行查询，O(r+logn)，r是查询范围长度



   - 2-d tree 构造

用排序（也可以别的玩意）找中位数，注意是对x还是对y排，中间的数留下，两边分别递归，递归选择与当前基不同的基(x/y)；

- 范围查询

  与查询区域相交则进入子树，否则不进（剪枝）

- 最近邻查询

  对于每个子树树根，递归时均进入查询点P所在的半平面对应的子树；更新minDist；dfs回溯时比较未进入的子树与查询点P的水平/垂直距离（Δx或Δy），如果距离小于minDist则进入，否则不进入，直接剪枝.

堆

1. 堆：完全二叉树；只需维护最大值，而无需维护其他元素的全局有序性
2. 向量id约定

$$\boxed{\boldsymbol{Parent(i) = (i-1) \gg 1}}$$

$$\boxed{\boldsymbol{lChild(i) = (i \ll 1) + 1}}$$

$$\boxed{\boldsymbol{rChild(i) = (i \ll 1) + 2}}$$

   大小顶堆约定：优先级队列默认大顶堆

3. 插入：插入队尾上滤

   删除队首：队尾元素与队首交换，下滤

4. 堆构建

   暴力插入：$\boldsymbol{O}(\boldsymbol{log1}+\boldsymbol{log2}+...+\boldsymbol{logn})=\boldsymbol{O}(\boldsymbol{logn!})=\boldsymbol{O}(\boldsymbol{nlogn})$

   Floyd堆合并：0-$\lfloor \boldsymbol{n/2} \rfloor-\boldsymbol{1}$(n是节点总数) 每点均递归下滤，共从($\lfloor \boldsymbol{n/2} \rfloor$)个节点开始进行递归下滤操作

✓ **蛮力算法：每个节点时间正比于其深度**

$$\sum_j \boldsymbol{depth(j)} = S(h) = \sum_{i=0}^{h} i \times 2^i$$

$$\rightarrow 2S(h) = \sum_{i=0}^{h} i \times 2^{i+1} = \sum_{i=1}^{h+1} (i-1) \times 2^i$$

$$\rightarrow S(h) = \sum_{i=1}^{h+1} (i-1) \times 2^i - \sum_{i=0}^{h} i \times 2^i = h \cdot 2^{h+1} - 2^{h+1} + 2$$

$$\rightarrow S(h) = (\log_2(n+1) - 2)(n+1) + 2 = O(nlogn)$$

✓ **堆合并法：每个节点时间正比于其高度**

$$\sum_j \boldsymbol{height(j)} = \sum_{i=0}^{h}\left((h-i) \times 2^i\right) = 2^{h+1} - (h+2)$$

$$= n - log_2(n+1) = O(n)$$

图

1. 路径：由边连接的顶点路径

   环路：起点和终点相同的路径

简单路径：若路径上各顶点 v1, v2, ..., vm 均不互相重复, 则称这样的路径为简单路径

连通图与连通分量：在无向图中, 若从顶点v1到顶点v2有路径, 则称顶点v1与v2是连通的。如果图中任意一对顶点都是连通的, 则称此图是连通图。非连通图的极大连通子图叫做连通分量

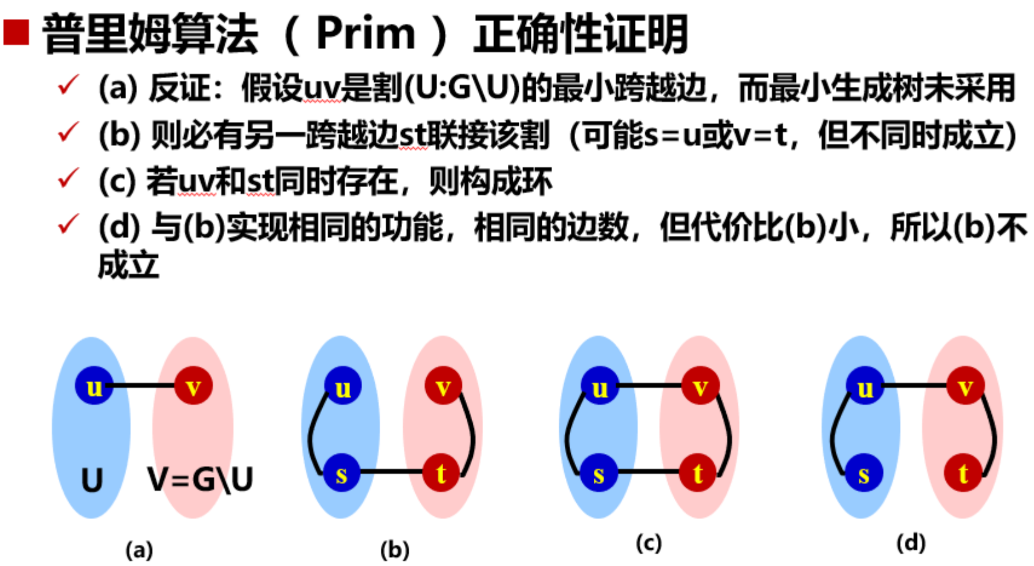
强连通图与强连通分量：在有向图中, 若对于每一对顶点vi和vj, 都存在一条从vi到vj和从vj到vi的路径, 则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量.

2. 图的表示
   - 邻接矩阵：使用方阵A[n][n]表示由n个顶点构成的图；对于不存在的边，通常统一取值为无穷或0
   - 邻接表：空间O(n+e)
3. 图的遍历：图的每个顶点被访问一次且仅一次；每个边至多访问一次

   遍历树：遍历形成的树

   树边：遍历树的边

   各顶点被访问到（将顶点标记为**DISCOVERED**）的次序，类似于树的先序遍历；各顶点被访问完（将顶点标记为**VISITED**）的次序，类似于树的后序遍历
   教材要求记录访问到的时间 dTime （DISCOVERED）及访问完的时间 fTime (VISITED)
4. dfs边的种类：树边TREE（undiscovered）、前向边FORWARD（已经visited但进入时间戳小）、后向边BACKWARD（discovered）、跨边CROSS（已经visited但进入时间戳大）；以上边的标记可以在遍历树中直观看出
5. Prim：朴素O(n)，每次找最短割边



6. 最短路径树SPT：

   单调性：最短路径的任意前缀也是最短路径；S到v的最短路径经过u，则沿着该路径从S到u也是u的最短路径（可反证）

   BFS优先考虑当前所有被发现点中，最早被发现的点；
   DFS优先考虑当前所有被发现点中，最后被发现的点；
   Prim和Dijkstra考虑当前被发现点中，优先级最高的点；
7. Dijikstra

   邻接矩阵、邻接表$O(n^2)$；优先队列O(elogn)

   处理不了负边
8. 也不知道是干啥有可能是水字数用的但看起来很厉害的亚子

```
template <typename Tv, typename Te> struct BfsPU { //针对BFS算法的顶点优先级更新器
   virtual void operator() ( Graph<Tv, Te>* g, int uk, int v ) {
      if ( g->status ( v ) == UNDISCOVERED ) //对uk每一尚未被发现的邻接顶点v
         if ( g->priority ( v ) > g->priority ( uk ) + 1 ) { //优先级比父亲降低1
            g->priority ( v ) = g->priority ( uk ) + 1; //更新优先级（数）
            g->parent ( v ) = uk;                       //更新父节点
         } //如此效果等同于，先被发现者优先
   }
};
```

时间复杂度O(n²)，可通过优先级队列降低

```
template <typename Tv, typename Te> struct BfsPU { //针对BFS算法的顶点优先级更新器
   virtual void operator() ( Graph<Tv, Te>* g, int uk, int v ) {
      if ( g->status ( v ) == UNDISCOVERED ) //对uk每一尚未被发现的邻接顶点v
         if ( g->priority ( v ) > g->priority ( uk ) - 1 ) { //优先级比父亲提高1
            g->priority ( v ) = g->priority ( uk ) - 1; //更新优先级（数）
            g->parent ( v ) = uk;                       //更新父节点
         } //如此效果等同于，后被发现者优先
   }
};
```

时间复杂度O(n²)，可通过优先级队列降低

```
template <typename Tv, typename Te> struct PrimPU {//针对Prim算法的顶点优先级更新器
   virtual void operator() ( Graph<Tv, Te>* g, int uk, int v ) {
      if ( g->status ( v ) == UNDISCOVERED ) //对uk每一尚未被发现的邻接顶点v
         if ( g->priority ( v ) > g->weight ( uk, v ) ) { //优先级为边的权重
            g->priority ( v ) = g-> weight ( uk, v ); //更新优先级（数）
            g->parent ( v ) = uk;                      //更新父节点
         }
   }
};
```

时间复杂度O(n²)，可通过优先级队列降低

```
template <typename Tv, typename Te> struct PrimPU {//针对Dijkstra的顶点优先级更新器
   virtual void operator() ( Graph<Tv, Te>* g, int uk, int v ) {
      if ( g->status ( v ) == UNDISCOVERED ) //对uk每一尚未被发现的邻接顶点v
         if (g->priority(v) > g->priority(uk)+g->weight(uk,v)){//原长度加新增长度
            g->priority ( v ) = g->priority(uk)+g->weight(uk,v); //更新优先级（数）
            g->parent ( v ) = uk;                                //更新父节点
         }
   }
};
```

时间复杂度O(n²)，可通过优先级队列降低

◻题目：

- 无向图G=(V, E)，其中：V={a, b, c, d, e, f}，E={(a, b)，(a, e)，(a, c)，(b, e)，(c, f)，(f, d)，(e, d)}，对该图进行深度优先遍历（优先访问编号小的结点），得到的顶点序列为？

//abedfc

9. **图搜索统一框架**：选点更新邻居，更新完优先级再选点（估计补全代码必考）
10. Belleman-Ford(SPFA前体物，没队列优化的SPFA)：O(ne)，可以判负环

```
bool Bellman_Ford(){
    for (int i = 1; i <= nodenum; ++i) //初始化
        dist[i] = (i == original ? 0 : MAX);
    for (int k = 1; k <= nodenum - 1; ++k) //k次迭代
        for (int j = 1; j <= edgenum; ++j)
                        // 对原公式n²次松弛，简化为e次边的松弛
            if (dist[edge[j].v]>dist[edge[j].u]+edge[j].cost){
                dist[edge[j].v]=dist[edge[j].u]+edge[j].cost;
                pre[edge[j].v] = edge[j].u;
            }
    bool negative = false; //判断是否含有负权回路
    for (int j = 1; j <= edgenum; ++j)
                // 再做一次迭代看是否有任意边可改进，若是则有负权和回路
        if(dis[edge[j].v] > dis[edge[j].u]+edge[j].cost){
            negative = true; break;}
    return negative;
}
```

证了个寂寞的证明

Bellman和Ford算法正确性证明

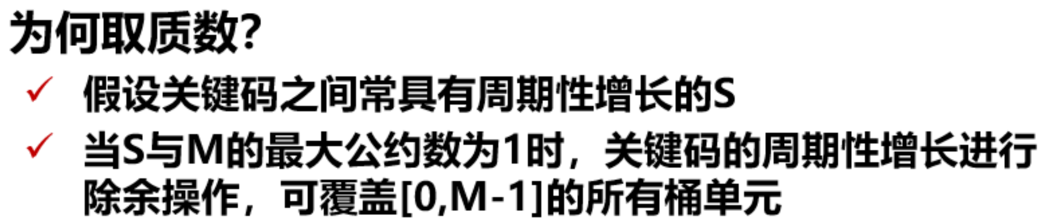
- ✓ **为何能检测负权和环路并返回正确的negative判断?**
  negative=0， 无负权回路； negative=1, 有负权回路;

- ✓ **若图中有负权和环路，而negative=0**
- ✓ **则设该环路为包含m个节点**$c = \{v_0, v_1, v_2 \dots v_m\}$, $v_m = v_0$，**则有** $\sum_{i=1}^{m} cost(v_{i-1}, v_i) < 0$ **。假设算法返回negative=0，则对任意** i **都有** $dist(v_i) \le dist(v_{i-1}) + cost(v_{i-1}, v_i)$**，这里** $i = 1, 2 \dots, m$**。将环路**c**上的所有这些不等式相加，有** $\sum_{i=1}^{m} dist(v_i) \le \sum_{i=1}^{m} dist(v_{i-1}) + \sum_{i=1}^{m} cost(v_{i-1}, v_i)$**，由于** $v_m = v_0$**，上式得到**$\sum_{i=1}^{m} cost(v_{i-1}, v_i) \ge 0$**，与环路为负权和的结论矛盾，故算法必定返回1**

11. Floyd：kij
12. 拓扑排序

散列

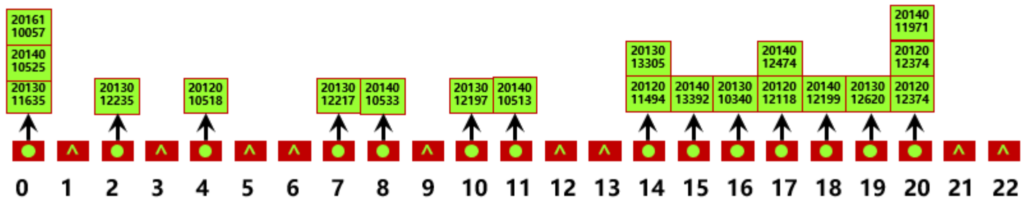
1. 散列技术在记录的存储位置和关键码之间建立一确定的对应关系f， f 称为散列函数（哈希函数）；散列技术将所有记录存储在一片连续空间（使用向量作为支撑结构），这块连续空间称为散列表（哈希表）
2.
3. 散列函数
    - 除余法：$\textbf{hash} = f(\textbf{key}) = \textbf{key} \ \% \ \textbf{M}$



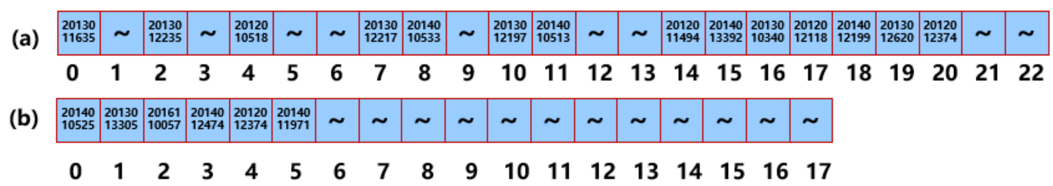
- MAD法：**hash=*f*(key)=(a × key +b)** % **M**，可以克服连续局部聚集
- 数字分析法：抽取关键码的若干位作为散列函数；如学号、电话号码后几位，特征更明显
- 平方取中法：~~1~~5227~~56~~ = 227
- 折叠法：将关键码从左至右分为位数相等的几部分，然后将几部分叠加求和，并按照散列表表长取后几位作为地址；f(9876543210) = 987+654+321+0 = ~~1~~962 = 962
- （伪）随机数法：**hash=*f*(key)=rand(key)** % **M**，用自带rand()，注意平台移植

4. 冲突排解
    - 多槽位法

- 独立链法



- 公共溢出区法：在原散列表(a)之外另设一词典结构(b)作为公共溢出区；适用于冲突极少的情况



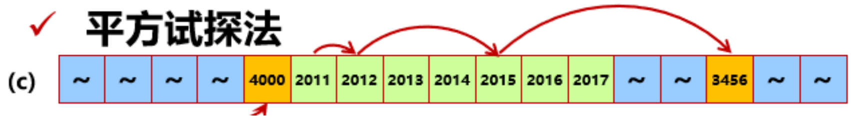
- 线性试探法（每个词条均有可能落到任意的散列地址，称作**开放定址**；闭散列策略）

动态删除词条：标记lazyRemoval，可以标示继续向后试探，此处也可装填

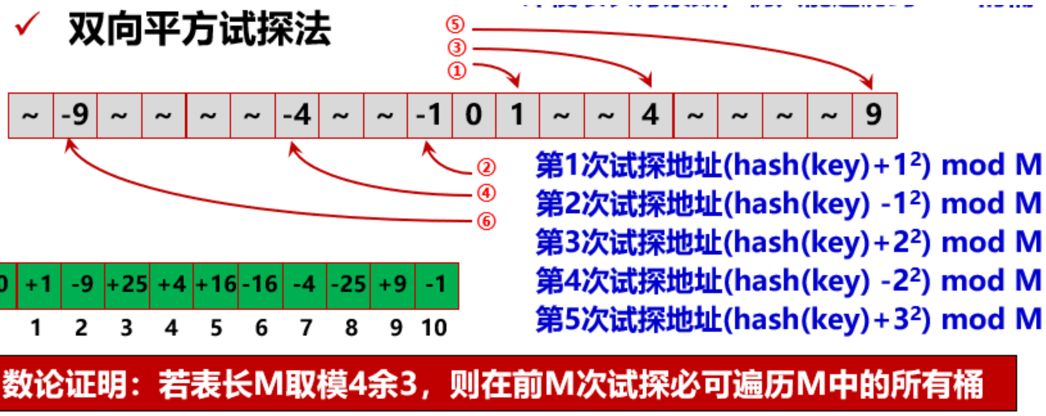
- 因可用的散列地址仅限于散列表所覆盖的范围内，所以称为闭散列策略

- 平方试探法（开放定址）

第k次试探的地址是（hash(key)+k2) mod M，k=0,1,2....



- 双向平方试探法



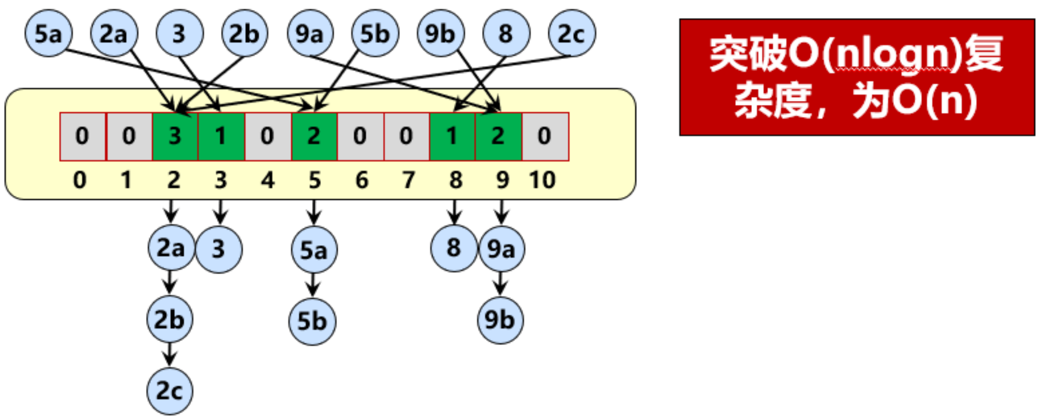
表长M%4==3

- 散列码转换

如对字符串，$x_0a^(n-1)+x_1a^(n-2)+\ldots+x_(n-2)a^1+a^0$

5. 桶排序，[0~M)范围的数

- ✓ **算法占用空间O(M)**
- ✓ **算法创建散列表时间O(M)，关键码插入耗时O(n)，依次读取关键码耗时O(max(M,n))，整体O(max(M,n))**
- ✓ **若允许输入整数重复**
- ✓ **可采用独立链方法排解冲突，读取排序时按每独立链读取**



基数排序

输入序列	441	276	320	214	698	280	112
以个位排序	320	280	441	112	214	276	698
以十位排序	112	214	320	441	276	280	698
以百位排序	112	214	276	280	320	441	698

先低后高

■ **复杂度：各字段取值范围[0,M_i), i<=t，若M=max(M_i)**

$O(n+M_1)+O(n+M_2)+\ldots+O(n+M_t)=O(t*(n+M))$

串

1. 基本概念
   - 长度为0的串称为空串“”，注意与空白串“ ”的区别
   - ■ **子串：S.substr(i,k)= “$a_i\ a_{i+1} \dots a_{i+k-1}$” =S[i, i+k),**
     - ✓ **S[i]起的连续k个字符**　[0,i) [i, i+k) [i+k,n)

     ■ **前缀：S.prefix(k) = S.substr(0,k)=S[0, k)**
     - ✓ **S中最靠前的k个字符**　[0, k) [k,n)

     ■ **后缀：S.suffix(k) = S.substr(n-k,k)= S[n-k, n)**
     - ✓ **S中最靠后的k个字符**　[0,n-k) [n-k, n)

     注意substr(i,k)：k是长度
   - ADT

操作接口	功能
length()	查询串的长度
charAt(i)	返回第i个字符
substr(i,k)	返回从第i个字符起，长度为k的子串
prefix(k)	返回长度为k的前缀
suffix(k)	返回长度为k的后缀
equal(T)	判断T是否与当前字符串相等
concat(T)	将T串接在当前字符串的后面
indexOf(P)	若P是当前字符串的一个子串，则返回该子串的位置；否则返回-1

   - 库函数

strcpy	**字符串复制 int strcpy(char* string1, char * string2)**
	复制字符串string2的内容到string1中
	char str1[] = “word1”, char str2[] = “word2”, 调用strcpy(str1, str2)后， str1 = “word2”, str2 = “word2”
strncpy	**字符串部分复制 int strncpy(char* string1, char * string2，int n)**
	字符串string2的前n个字符覆盖掉字符串string1的前n个字符
	char str1[] = “Hello”, char str2[] = “world”, 调用strcpy(str1, str2，2)后， str1 = “Wollo”, str2 = “world”
strcat	**字符串连接 int strcat(char* string1, char * string2)**
	连接字符string2到字符串string1后面，存储于string1中
	char str1[] = “Tsing”, char str2[] = “hua”, 调用strcat(str1, str2)后， str1 = “Tsinghua”, str2 = “hua”
strncat	**将特定数量字符串连接到另一字符串 int strncat(char* string1, char * string2，int n)**
	连接字符串string2中的前n个字符到字符串string1后面，存储于string1中
	char str1[] = “Tsing”, char str2[] = “hua”, 调用strcat(str1, str2, 2)后， str1 = “Tsinghu”, str2 = “hua”

strchr	**在给定字符串中搜索给定字符第一次出现的地址 char * strchr (char* string1, char ch)**
	在字符串string1中搜索字符ch并返回指向字符ch的指针，失败返回NULL
	char str[100] = “The Dog Barked at the Cat”, char * p = strchr(str, ‘B’)后，指针p得到字符‘B’的存储位置
strcspn	**在给定字符串中搜索指定字符第一次出现的位置 int strcspn (char* string1, char ch)**
	在字符串string1中搜索字符ch并返回第一次出现位置（0开始计数），失败返回字符串长度
	char str[100] = “The Dog Barked at the Cat”, int d = strcspn (str, ‘a’)后，d=9
strrchr	**在给定字符串中搜索指定字符最后一次出现的地址 char * strrchr (char* string1, char ch)**
	在字符串string1中搜索字符ch并返回最后一次出现位置的指针，失败返回NULL
	char str[100] = “The Dog Barked at the Cat”, char * d = strcrhr (str, ‘a’)后，d得到最后一次出现a的地址
strpbrk	**两字符串中寻找首次共同出现的字符 char * strpbrk(char *string1, char *string2)**
	返回该字符在string1中的地址，若找不到则返回NULL
	char str1[] = “University”, char str2[] = “converse”,char * p = strpbrk (str1,str2) 后，在指针p内得到首先出现共同字符’v’在str1中的地址
strstr	**两字符串中寻找首次共同出现子字符串 char * strstr(char *string1, char *string2)**
	返回该子字符串在string1中的地址，若找不到则返回NULL
	char str1[] = “University”, char str2[] = “converse”,char * p = strstr (str1,str2) 后，在指针p内得到首先出现共同子字符串’ver’在str1中的地址
strlen	**计算字符串长度 int strlen (char* string1)**
	在字符串中字符的个数，串结束符’ \0’不计入内
	char str[] = “Tsinghua”, int d = strlen(str)后，d=8
strcmp	**字符串比较大小 int strcmp (char* string1, char* string2)**
	比较两字符串大小，返回值小于0，等于0，大于0分别表示string1小于,等于,大于string2 两个字符串自左向右逐个字符相比（按ASCII值大小相比较），直到出现不同的字符或遇’\0’为止
	int k = strcmp(“Joe”, “Joseph”)后，k得到值小于0

2. 蛮力算法

性能分析

- ✓ **最好情况：只经过一次匹配即找到，O(m)**
- ✓ **最坏情况：尝试所有的循环**

  每轮循环比对m-1次成功，1次失败，即m次比对n-m+1次循环

  因此，比对次数为m×(n-m+1)=O(n×m)

3. 朴素KMP：next表

$P[0, j)$ 中长度为 t 的真前缀，应与长度为 t 的真后缀完全匹配，故 t 来自集合：

$$N(P,j) = \{0 \le t < j \mid P[0,t) = P[j-t,j)\}$$

为保证不遗漏可能的匹配，应在N(P,j)中挑选最大 t： $next[j] = max(N(P,j))$

一旦发生P[j]与T[i]失配，可转而将P[next[j]]与T[i]对齐，进行下一轮匹配

$$Next[j] = \begin{cases} -1, & j = 0 \\ Max\{t \mid 0 \le t < j, P[0,t) = P[j-t,j)\} & \text{该集合非空} \\ 0 & \text{其它} \end{cases}$$

4. 代码实现

- 查询

```
int match ( char* P, char* T ) {
   int* next = buildNext ( P );
   int n = ( int ) strlen ( T ), i = 0;
   int m = ( int ) strlen ( P ), j = 0;
   while ( j < m  && i < n )
      if ( T[i] == P[j] )
         { i ++;  j ++; }
      else
         j = next[j];
   delete [] next;
   return i - j;
}
```

- 建表

```
int* buildNext ( char* P ) { //构造模式串P的next表
   size_t m = strlen ( P ), j = 0; //“主”串指针
   int* N = new int[m]; //next表
   int t = N[0] = -1; //模式串指针
   while ( j < m - 1 )
      if ( 0 > t || P[j] == P[t] ) { //匹配
         j ++; t ++;
         N[j] = t;
      }
      else //失配
         t = N[t];
   return N;
}
```

- 复杂度分析

令k=2*i-j
由while循环的分析，每进行一次while循环，k至少+1
k的最大值可能为$2*i_{max}-j_{min}$=2n+1，故while循环至多执行2n+1次。为此，复杂度为O(n)

5. 改进KMP

$$N(P,j) = \{0 \leq t < j \mid P[0,t) = P[j-t,j), \text{且} P(t) \neq P(j)\}$$

$$next[j] = max(N(P,j))$$

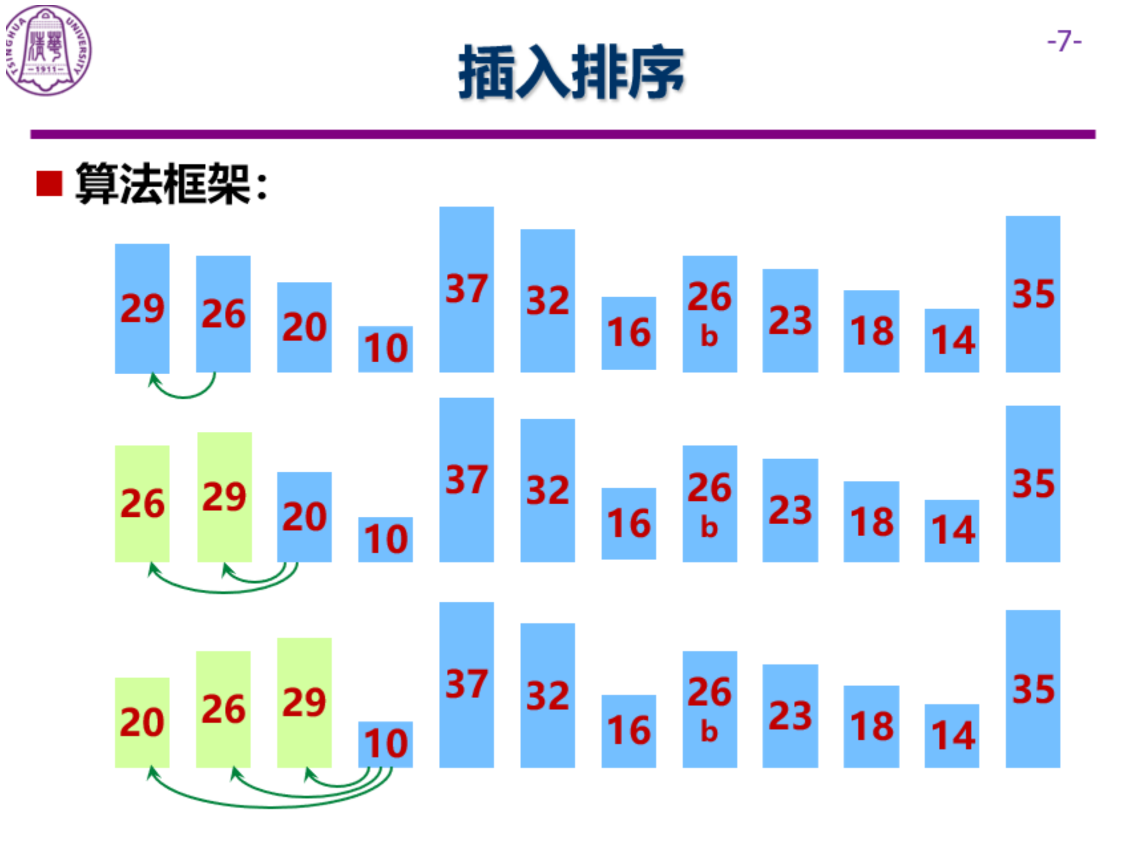
手算法宝：首先先算出朴素KMP，再从前向后更新，比对s[next[i]]和s[i]

排序

1. 排序码：排序的依据

   内排序：不需要外部空间；外排序：需要辅助外部空间

2. 插入排序：O(n^2)稳定，对于几乎有序的序列几乎是线性的




```
void insertSort(int data[], int l, int r) {
    int auxiliary = 0;
    for (int i = l+1; i <= r; i++) {
        if (data[i] < data[i - 1]) {
            auxiliary = data[i];
            int j = i - 1;
            while (j >= l && data[j] > auxiliary) {
                data[j + 1] = data[j];
                j -= 1;
            }
            data[j + 1] = auxiliary;
        }
}
```

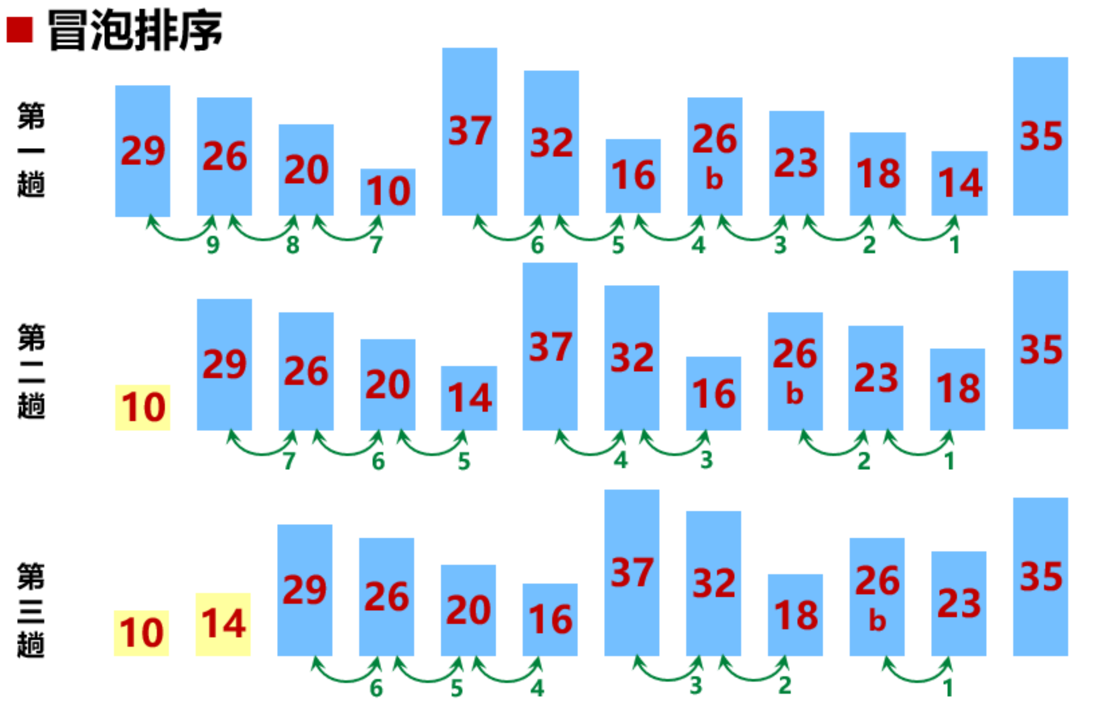
3. 希尔排序：每次$gap=\lceil gap/2\rceil$，每个gap划分的类中进行插入排序

```
void ShellSort(int data[], int count){
    int step = 0;
    int auxiliary = 0;
    for (step = count / 2; step > 0; step /= 2){  // 从数组第step个元素开始
        for (int i = step; i < count; i++){      // 每个元素与自己组内的数据进行直接插入排序
            if (data[i] < data[i - step]){      // 插入排序的第一次判断
                auxiliary = data[i];            // 需要往前插入，对待插入数据进行缓存
                int j = i - step;
                while (j >= 0 && data[j] > auxiliary){ //对同组前面数据检测，所大则循环后移
                    data[j + step] = data[j];
                    j -= step;
                }
                data[j + step] = auxiliary;           // 插入数据
            }
        }
    }
}
```

4. 归并排序：稳定

```
static void merge(int data[], int first, int mid, int last, int sorted[]){
    int i = first, j = mid;
    int k = 0;
    while (i < mid && j < last)
        if (data[i] < data[j])                // 归并
            sorted[k++] = data[i++];
        else
            sorted[k++] = data[j++];
    while (i < mid) sorted[k++] = data[i++];   // 拷贝两个子序列中的剩余元素
    while (j < last) sorted[k++] = data[j++];
    for (int v = 0; v < k; v++)
        data[first + v] = sorted[v];        // 将排序后结果覆盖原数组
}
static void mergeSort(int data[], int first, int last, int sorted[]){
    if (first + 1 < last){
        int mid = (first + last) / 2;
        mergeSort(data, first, mid, sorted);
        mergeSort(data, mid, last, sorted);
        merge(data, first, mid, last, sorted);
    }
}
```

5. 冒泡排序：稳定



```
void bubbleSort(int data[], int count) {
    int temp;
    int pass = 1; bool exchange = true;             // 从第一趟开始
    while (pass < count && exchange) {
        exchange = false;                    // 某趟是否有交换的标志，初始为无交换
        for (int j = count - 1; j >= pass; j--) //从最后元素开始到第一个未排序元素
            if (data[j - 1]>data[j]) {              //若需要交换则置换元素
                temp = data[j - 1];
                data[j - 1] = data[j];
                data[j] = temp;
                exchange = true;
            }
        pass++;
    }
}
```

6. 快速排序：不稳定

Partition

```
void quickSort(int data[], int l, int r){
    if (l< r){
        int pivotL = l, pivotR = r, x = data[l];
        while (pivotL < pivotR){
            while (pivotL<pivotR && data[pivotR]>x) pivotR --;
                        //从右向左找第一个小于x的数
            if (pivotL<pivotR) data[pivotL ++] = data[pivotR];

            while (pivotL<pivotR && data[pivotL]<x) pivotL++;
                       //从左向右找第一个大于等于x的数
            if (pivotL<pivotR) data[pivotR --] = data[pivotL];
        }
        data[pivotL] = x;
        quickSort(data, l, pivotL - 1);    // 递归调用处理左子序列
        quickSort(data, pivotL + 1, r);    // 递归调用处理右子序列
    }
}
```

假设待排序的元素服从独立均匀随机分布。Partition算法在经过n-1次比较和至多n+1次移动操作后，对规模为n的向量的划分结果无非n种可能，划分所得左侧子序列的长度分别为0,1,2,…,n-1，按假定条件，每种情况的概率均为1/n，故若将算法的平均运行时间记为T(n)，则有

$$T(n) = (n+1) + (1/n) \times \sum_{k=1}^{n} [T(k-1) + T(n-k)]$$

$$= (n+1) + (2/n) \times \sum_{k=1}^{n} T(k-1)$$

等式两边同乘以n，则有

$$nT(n) = n(n+1) + 2\sum_{k=1}^{n} T(k-1)$$

同理有

$$(n-1)T(n-1) = (n-1)n + 2\sum_{k=1}^{n-1} T(k-1)$$

✓ **以上两式相减，即得：**

$$nT(n) - (n-1)T(n-1) = 2n + 2T(n-1)$$

$$nT(n) = (n+1)T(n-1) + 2n$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{T(0)}{1} + \frac{2}{2} + \cdots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= 2 \cdot \sum_{k=1}^{n+1}(1/k) - 1 = 2 \cdot \mathcal{O}(lnn) = \mathcal{O}(2 \cdot ln2 \cdot log_2 n) = \mathcal{O}(1.386 \cdot log_2 n)$$

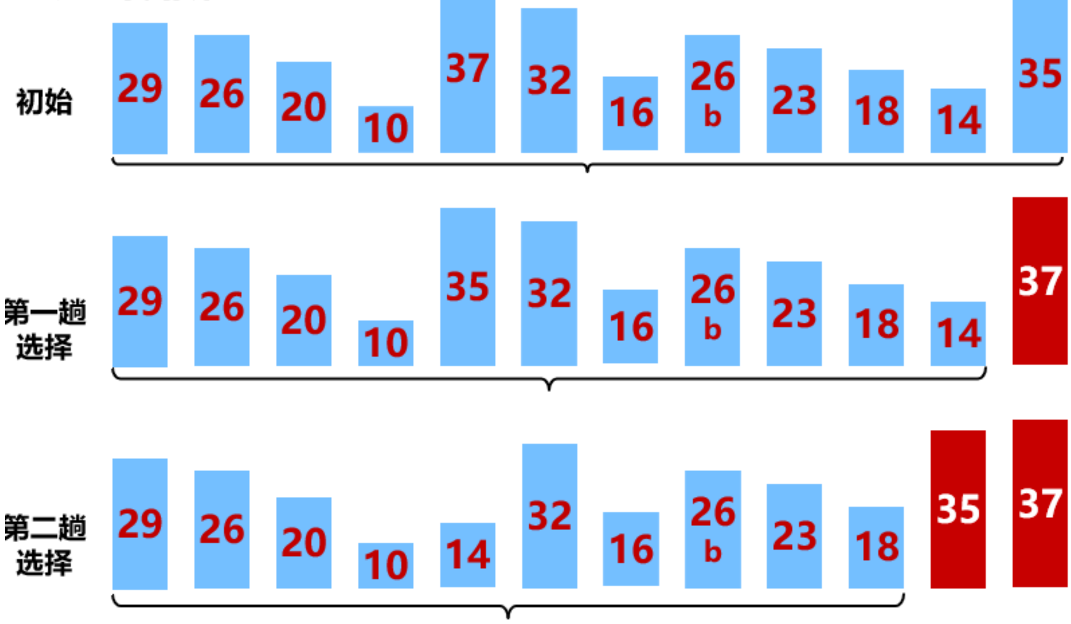
实验结果表明，平均计算时间而言，快速排序是最好的内部排序方法

改进1：对于特别小规模的子序列直接插入排序

改进2：选头、尾、中三个元素中大小在中间的元素，交换到头部进行划分

7. 选择排序：直接交换，所以直接选择排序**不稳定！！！**

■ **选择排序**




```
void selectSort(int Data[], int count){
    int i, j, index, temp;
    for (i = count - 1; i > 0; i--){ // 从后往前
        index = i;
        for (j = 0; j < i + 1; j++){ // 遍历前面未排序，选择最大元素
            if (Data[j] > Data[index]) index = j;
        }
        if (index != i){            // 交换
            temp = Data[i];
            Data[i] = Data[index];
            Data[index] = temp;
        }
    }
}
```

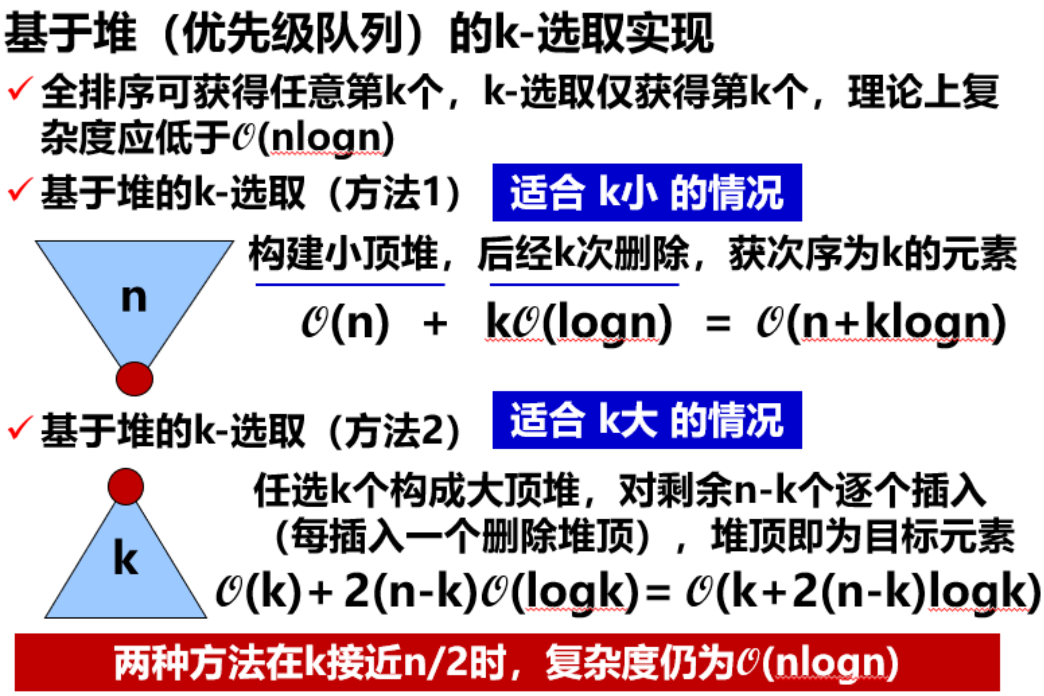
8. 堆排序：不稳定

9.

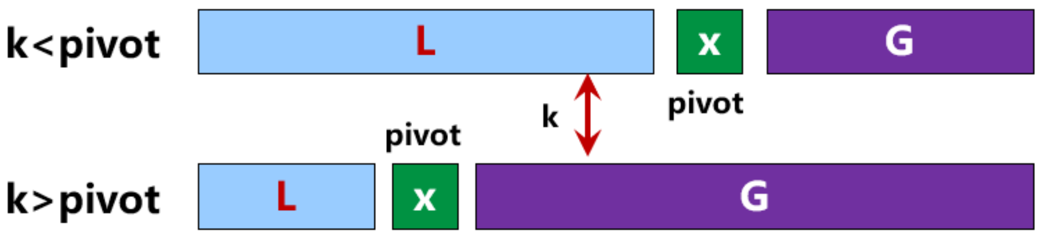
排序方法	平均情况	最好情况	最差情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	(不)稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log^2 n)$~$O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$~$O(n)$	不稳定

10. k-选取

- 基于堆的方法



- 基于快速划分的方法O(n)




```
quickSelect (int data[], int k, int l, int r){
    int pivot = partition(data, l, r);
    if(k==pivot) return data[pivot];
    else if(k<pivot) return quickSelect(data, k, l, pivot - 1);
    else return quickSelect(data, k-pivot-1, pivot+1, r);
}
```