散列表

散列表的概念

散列表也被称为哈希表(Hash Table)：是根据关键字的值直接访问元素存储位置的存储结构。也就是说，在元素的存储地址和它的关键字之间建立一个确定的对应关系H，使得每个关键字和一个唯一的存储位置相对应，即：$\text{addr}(R_i)=H(\text{key}_i)$。其中，$R_i$为查找表中的某个元素，$\text{key}_i$是其关键字的值。我们称对应关系$H$为散列函数或哈希函数(Hash Function)，$H(\text{key}_i)$的值为散列地址，按此思想建立的查找表为散列表。

散列表既是一种存储方式，又是一种查找方式。散列技术的核心是散列函数的设计，首先确定一个散列函数H，将各元素的关键字作为自变量，求出其对应的函数值，作为各元素的存储地址，将各元素存储在相应的存储位置中。查找时仍然按照函数H进行计算，得到待查元素的存储地址。

例如，关键字集合为：$S=\{\text{apple, april, james, jeff, moon, safe, sail, sophia}\}$。假设以关键字的首字母在英文字母中的编号作为其散列地址，即：$H(\text{key}_i)=S[i][0]-\text{'a'}$。在计算散列地址时发现：
$H(\text{apple})=H(\text{april})=0, H(\text{james})=H(\text{jeff})=9, H(\text{moon})=12, H(\text{safe})=H(\text{sail})=H(\text{sophia})=18$，这种不同的关键字通过相同的散列函数计算得到同一地址的现象称为冲突或碰撞(Collision)，即$\text{key}_1\neq\text{key}_2$，但是$H(\text{key}_1)=H(\text{key}_2)$。称这些发生冲突的关键字为相对于散列函数的同义词(Synonym)。在一般情况下，冲突只能尽量减少，而不能完全避免。因此，散列函数必须有解决冲突的办法。

设计散列函数的要求如下：

1. 散列函数本身简单高效。
2. 散列函数计算出来的地址应均匀分布在地址空间内，极少出现冲突。
3. 有合适的解决冲突的方法。

构造散列函数的方法

直接定址法

直接定址法就是取关键字或关键字的某个线性函数值作为散列地址，即：

$$H(\text{key})=\text{key}\quad 或\quad H(\text{key})=a\times\text{key}+b\quad 其中a和b为常数$$

由于直接定址不是压缩映象，地址集合和关键字集合的大小相同，因此，对于不同的关键字不会发生冲突。但实际上能适用这种散列函数的情况很少。

折叠法

将较长的关键字从左到右分割成位数相同的几段，每段的位数应与散列地址的位数相同，最后一段的位数可以少一些。，然后把这几段叠加并舍去进位，得到的结果作为散列地址，这种方法称为折叠法。折叠法适用于关键字位数很多，且每位的数字分布大致均匀的情况。有两种叠加的方法：

1. 移位叠加：把各段的最低位对齐相加。
2. 间界叠加：从左到右沿分割界把各段来回折叠，然后对齐相加。

数字分析法

对关键字进行分析，取关键字的数值分布均匀的若干位或其组合作为散列地址，尽可能构造冲突概率较低的散列地址，这种方法称为数字分析法。数字分析法适用于关键字位数比散列地址位数大，切可能出现的关键字事先知道的情况。

平方取中法

平方取中法是一种比较常见的构造散列函数的方法。关键字求平方后，按散列表的表长，取中间的若干位作为散列地址。这是因为关键字求平方后的中间几位数和关键字的每一位都有关，能反映关键字每一位的变化，使得随机分布的关键字对应到随机的散列地址上。平方取中法比较适用于事先不知道关键字每一位的分布，且关键字的值域比散列表长度大，关键字的位数又不是很大的情况。

除留余数法

除留余数法采用取模运算mod，把关键字除以某个不大于散列表长度的证书得到的余数作为散列地址。散列函数形式为：

$$H(\mathrm{key}) = \mathrm{key} \bmod p \quad p \leq m$$

其中，散列地址$H(\mathrm{key})$的值域为$[0, p-1]$，要求散列表的长度至少为p。若运算结果为负值，则需要加上p。

除留余数法的关键是p的选取，如果p选不好，就容易产生同义词。在实际应用中，通常取p为一个小于散列表长度m的最大质数，这样可以减少冲突的发生。

解决冲突的方法

尽管我们构造散列函数的目的是使得每个关键字与一个唯一的存储位置相对应，但是冲突是无法避免的，因此散列函数必须包括解决冲突的策略。有两种常用的解决冲突的方法：闭散列法（开放地址法）和开散列法（拉链法）。

闭散列法

闭散列法的基本思想是：对于一个待插入散列表的元素，若按照给定的散列函数求得的基地址$H(\mathrm{key})$已经被占用，则根据某种策略寻求另一种散列地址，重复这一过程，直到找到一个可用的地址空间来保存该元素。在查找元素时，也要遵循和插入一样的解决冲突的策略。闭散列法也叫开放地址法，有两个含义：

1. 数组空间是封闭的，发生冲突时不再使用额外的存储单元，因此称为闭散列法。
2. 数组中的每个地址都有可能被任何基地址的元素占用，即每个地址对所有元素都是开放的，因此也称为开放地址法。

用闭散列法解决冲突的散列表被称为闭散列表。当发生冲突时，寻找下一个散列地址的过程称为探测。探测序列的计算公式如下：

$$H_i = (H(\mathrm{key}) + d_i) \bmod m \quad i = 1, 2, \ldots, k \ (k \leq m-1)$$

其中$H(\mathrm{key})$为基地址，m为散列表长，d_i为增量序列。根据d_i的取值不同，可以分成几种探测方法，下面介绍常用的三种方法。

线性探测法

增量序列$d_i = i$，即d_i的取值为$1, 2, \ldots, m-1$的线性序列。线性探测法的基本思想是，当发生冲突时，依次探测地址$(H(\mathrm{key}) + 1) \bmod m, (H(\mathrm{key}) + 2) \bmod m, \ldots, (H(\mathrm{key}) + (m-1)) \bmod m$，直到找到一个空单元，把数据放入该空单元中。顺序查找时，我们把散列表看成一个循环表，如果探测到了表尾都没有找到空单元，则回到表头开始继续探测。如果探测了所有单元后仍然没有找到空单元，则说明散列表已满，需要进行“溢出”处理。

线性探测法计算简单，只要有空单元就可以将元素存入，但是容易产生堆积(Clustering)，即不同基地址的元素争夺同一个单元的现象。堆积也称为聚集，实际上是在处理同义词之间的冲突时引发的非同义词的冲突。显然，这种现象对查找不利。

二次探测法

线性探测很容易出现堆积，使得散列表中形成一些较长的连续被占单元，从而导致很长的探测序列，降低散列表的查找和插入运算的效率。因此，理想的方法是加大探测序列的步长，使得发生冲突的元素的位置比较分散，这种可行的方法是二次探测法。二次探测法的增量序列d_i为$1^2, -1^2, 2^2, -1^2, \ldots, \pm k^2$ $(k \leq \lfloor \frac{m}{2} \rfloor)$。当发生冲突时，依次探测地址
$(H(\text{key}) + 1) \bmod m, (H(\text{key}) - 1) \bmod m, (H(\text{key}) + 4) \bmod m, (H(\text{key}) - 4) \bmod m, \ldots$，直到找到一个空单元，把数据放入该空单元中。

该方法的优点是能够减少堆积的产生，缺点是不容易探测到整个散列空间。

双重散列法

双重散列法以关键字的另一个散列函数值作为增量序列。设两个散列函数为H_1, H_2，其中H_1用于计算基地址，H_2用于计算增量序列。双重散列法的增量序列$d_i = i \times H_2(\text{key})$，探测序列的计算公式如下：

$$d_i = (d + i \times H_2(\text{key})) \bmod m \quad (1 \leq i \leq m - 1)$$

当发生冲突时，依次探测地址
$(H_1(\text{key}) + 1 \times H_2(\text{key})) \bmod m, (H_1(\text{key}) + 2 \times H_2(\text{key})) \bmod m, (H_1(\text{key}) + 3 \times H_2(\text{key})) \bmod m, \ldots$，直到找到一个空单元，把数据放入该空单元中。

在设置第二个散列函数H_2时，通常要使H_2的值和m互质，才能使得发生冲突的同义词地址均匀地分布在整个散列表中，否则可能造成同义词地址的循环计算。若m为指数，则H_2取$1 \sim m - 1$之间的任何数均与m互质。

双重散列法的优点是不易产生堆积，但是这种方法的计算量稍大。

开散列法

开散列法（也称拉链法，或链地址法）的一种简单形式是，将所有关键字为同义词的元素存储在同一单链表中，单链表中每个结点包含两个域：数据域存储集合中的元素，指针域指向下一个同义词。单链表的头指针按照这些同义词的基地址存储在散列表数组中。由于链表结点是动态申请的，因此，只要存储器空间足够，就不会发生存储溢出问题，这也是“开”散列法名称的来源。用开散列法解决冲突的散列表称为开散列表。

散列表的实现

下面给出散列表的抽象数据类型定义，抽象类`hashTable`规定了散列表必须支持的操作，包括插入、查找和删除等。抽象类`hashTable`提供了一个基于除留余数法设计的默认散列函数`defaultHash`；还提供一个函数指针`hash`方便用于指定自己的散列函数。散列表的长度往往设置为质数，这是因为在设计散列函数的时候，需要一个大小合适的质数来进行取模运算，函数`nextPrime`的功能是获取大于表长的第一个质数：

```
template <class Type>
class hashTable{
public:
    virtual int size() = 0; // 散列表中元素的个数
    virtual int capacity() = 0; // 散列表的容量
    virtual bool search(const Type& k) const = 0; // 查找元素
    virtual bool insert(const Type& k) = 0; // 插入元素
    virtual bool remove(const Type& k) = 0; // 删除元素
    virtual void print() = 0; // 输出散列表
    virtual ~hashTable(){}
```

```
protected:
    int nextPrime(int n); // 求大于n的最小质数
    int (*hash)(const Type& k, int maxSize); // 散列函数
    static int defaultHash(const Type& k, int maxSize = capacity()); // 默认散列函数
};
```

默认散列函数：形参k是关键字，形惨maxSize是质数，可以用散列表的长度作为实参。函数范围值是一个$[0, \text{maxSize})$之间的散列地址。

```
template <class Type>
int hashTable<Type>::defaultHash(const Type& k, int maxSize) {
    int hashVal = k % maxSize; // 使用模运算计算散列值
    if(hashVal < 0) hashVal += maxSize; // 确保散列值为非负
    return hashVal;
}
```

获得距离n最近的一个大于n的质数：

```
template <class Type>
int hashTable<Type>::nextPrime(int n) {
    int i;
    if(n % 2 == 0) n++; // 确保n为奇数
    for( ; ; n += 2){
        for(i = 3; i * i <= n; i += 2){
            if(n % i == 0) break;
        }
        if(i * i > n) return n; // 找到大于n的最小质数
    }
}
```

闭散列表的表示和实现

闭散列表是用顺序存储结构表示和实现的。在顺序存储结构中删除元素不能像链式存储结构那样动态释放，并且散列表中国呢的元素不能随意移动，这些因素导致删除元素的时间代价较大，这时可以采用懒惰删除(Lazy Deletion)的方法。因此，在闭散列表中除要保存元素本身之外，还需要保存每个元素的状态。枚举类型NodeState列举了三种可能的状态：空(EMPTY)、使用中(ACTIVE)和已删除(DELETED)。

散链表的平均查找长度和装填因子有关，装填因子越大，表明散列表中空单元越少，发生冲突的可能性，在查找时所耗费的时间就越多；装填因子越小，表明散列表中还有很多的空单元，发生冲突的可能性就越小。增加散列表的长度可以使得装填因子变小，那么如何确定散列表的长度将非常关键，选大了可能会造成空间浪费，选小了容易发生冲突从而影响查找性能。已经有人证明，在装填因子为0.5左右时，闭散列表的综合性能达到最优。因此，当装填因子达到0.5时，应该调用resize函数，增加散列表的长度。resize函数重新申请了一块更大的存储空间，将原空间中标记为ACTIVE的元素插入新的空间中，而那些被标记为DELETED的元素不再存储，至此完成了删除操作。

下面给出闭散列表的类型定义及运算实现。在闭散列表中定义了结点类型Node，每个结点除了存储关键字key之外，还存储了结点的状态state。闭散列表的长度maxSize是根据用户指定的数值计算出来的大于该数的第一个质数。

```
template <class KeyType>
class closeHashTable : public hashTable<KeyType> {
    enum NodeState{EMPTY, ACTIVE, DELETED}; // 节点状态
    struct Node{
        KeyType key; // 键
        NodeState state; // 节点状态
        Node() : state(EMPTY) {} // 默认构造函数，状态为EMPTY
```

```
    };
    Node* data; // 散列表
    int maxSize; // 散列表的容量
    int curLength; // 散列表中元素的个数
    void resize(); // 扩展散列表
public:
    closeHashTable(int len = 11, int (*h)(const KeyType& k, int maxSize) = defaultHash);
    ~closeHashTable(){delete [] data;}
    int size() {return curLength;} // 返回散列表中元素的个数
    int capacity() {return maxSize;} // 返回散列表的容量
    bool search(const KeyType& k) const; // 查找元素
    bool insert(const KeyType& k); // 插入元素
    bool remove(const KeyType& k); // 删除元素
    void print(); // 输出散列表
};
```

构造函数：形参len是用户指定的数值，利用nextPrime(len)函数求出大于该数值的第一个质数作为散列表的长度；形参h是函数指针，用于可以通过实参来指定自己的散列函数。

```
template <class KeyType>
closeHashTable<KeyType>::closeHashTable(int len, int (*h)(const KeyType& k, int maxSize)){
    maxSize = nextPrime(len); // 确保容量为质数
    data  = new Node[maxSize]; // 分配内存
    curLength = 0; // 初始化元素个数为0
    hash = h; // 设置散列函数
}
```

查找操作1：查找关键字为k的元素是否在散列表中，查找成功返回true，查找失败返回false。闭散列表的查找算法描述如下：

首先计算关键字为k的元素的基地址。

1. 若该地址的状态为使用中，则分为两种情况：
    - 若该地址中元素的关键字不等于k，则表示发生冲突，利用线性探测法计算下一个散列地址。回到步骤1处继续判断散列地址的状态。
    - 若该地址中元素的关键字等于k，则查找成功，返回true
2. 若该地址的状态为空或意删除，则表示查找失败，返回false

查找操作2：查找关键字为k的元素是否在散列表中，查找成功返回其地址，查找失败返回-1。

```
template <class KeyType>
bool closeHashTable<KeyType>::search(const KeyType& k) const {
    int offset = 1;
    int pos = hash(k, maxSize); // 关键字为k的元素的基地址
    while(data[pos].state == ACTIVE){ // 该位置状态为ACTIVE
        if(data[pos.key] != k){ // 关键字不匹配
            pos = (pos + offset) % maxSize; // 线性探测法，计算下一个位置
        }else{
            return true; // 找到元素，返回位置
        }
        return false; // 没有找到元素
    }
}
```

```
template <class KeyType>
int closeHashTable<KeyType>::getPos(const KeyType& k) const {
    int offset = 1;
    int pos = hash(k, maxSize); // 关键字为k的元素的基地址
    while(data[pos].state == ACTIVE){ // 该位置状态为ACTIVE
        if(data[pos].key != k){ // 关键字不匹配
            pos = (pos + offset) % maxSize; // 线性探测法，计算下一个位置
        }else{
            return pos; // 找到元素，返回位置
        }
    }
    return -1; // 没有找到元素
}
```

插入操作：插入关键字为k的元素（不允许重复）到散列表中，若该关键字已经存在，则退出程序并返回false，否则插入该元素并返回true。闭散列表的插入算法描述如下：

首先计算装填因子，若大于0.5，则调用resize扩大表空间。然后计算关键字为k的元素的基地址。

1. 若该地址的状态为使用中，则分为两种情况：
    - 若该地址中元素关键字等于k，则退出程序并返回false
    - 若该地址中元素关键字不等于k，则表示发生冲突，利用线性探测法计算下一个散列地址。回到步骤1处继续判断新散列地址的状态。
2. 若该地址的状态为空或意删除，则插入该元素并返回true

```
template <class KeyType>
bool closeHashTable<KeyType>::insert(const KeyType& k) {
    int offset = 1, pos;
    if(curLength >= maxSize / 2) resize(); // 如果装填因子过半，扩展散列表
    pos = hash(k, maxSize); // 关键字为k的元素的基地址
    while(data[pos].state == ACTIVE){ // 该位置状态为ACTIVE
        if(data[pos].key == k) return false; // 关键字已存在，插入失败
        else pos = (pos + offset) % maxSize; // 线性探测法，计算下一个位置
    }
    data[pos].key = k; // 插入元素
    data[pos].state = ACTIVE; // 设置状态为ACTIVE
    curLength++; // 元素个数加1
    return true; // 插入成功
}
```

删除操作：删除散列表中关键字为k的元素。若删除成功则返回true，否则返回false。删除算法和插入算法相似，都要先查找元素的位置，若能够找到元素，则采用懒惰删除法，将状态改为DELETED。

```
template <class KeyType>
bool closeHashTable<KeyType>::remove(const KeyType& k) {
    int pos = getPos(k); // 获取元素k的散列位置
    if(pos == -1) return false; // 没有找到元素，删除失败
    data[pos].state = DELETED; // 设置状态为DELETED
    curLength--; // 元素个数减1
    return true; // 删除成功
}
```

扩大散列表空间：该函数重置散列表长度为大于2*maxSize的第一个质数。首先重新申请一块存储空间，然后调用insert函数将原来空间中标记为ACTIVE的元素依次插入新的空间中，最后释放原有空间。注意，不能像普通的顺序表一样直接复制元素，应根据新的散列表长度重新计算散列地址，再将元素插入表空间中。

```
template <class KeyType>
void closeHashTable<KeyType>::resize(){
    Node* tmp = data;
    int oldSize = maxSize; // 记录旧的散列表容量
    maxSize = nextPrime(2 * maxSize); // 扩展容量为原来的两倍
    data = new Node[maxSize]; // 分配新的内存
    curLength = 0; // 重置元素个数为0
    for(int i = 0; i < oldSize; i++){
        if(tmp[i].state == ACTIVE){
            insert(tmp[i].key);
            curLength++; // 插入旧散列表中的元素
        }
    }
    delete [] tmp; // 释放旧散列表的内存
}
```

输出散列表：遍历散链表，输出标记为ACTIVE的元素。

```
template <class KeyType>
void closeHashTable<KeyType>::print(){
    int pos;
    cout << "输出闭散链表中的内容:" << endl;
    for(pos = 0; pos < maxSize; pos++){
        if(data[pos].state == ACTIVE){
            cout << pos << ": " << data[pos].key << "\t\t";
        }
    }
    cout << endl;
}
```

开散列表的表示和实现

开散列表是用链式结构表现和实现的，所有关键字为同义词的记录存储在同一单链表中，单链表的头指针按照这些同义词的基地址保存到散列表数组中，因而查找、插入和删除操作都是在同义词单链表中进行的。尽管开散列表中的结点是动态申请的，只要存储器空间足够，就不会发生存储溢出的问题，但是当装填因子过大时，运算效率会降低。例如，一个极端情况：当集合中有n个元素，而散列表数组的大小为1时，开散列表将退化为一个单链表，所有元素都是同义词，查找、插入和删除操作的时间复杂度均为$O(n)$。因此在实现开散列表时，当装填因子达到1时将调用resize扩大空间，防止单链表中元素过多，影响运算速度。

下面给出开散链表的类型定义及运算实现。在开散列表中定义了结点类型Node，每个结点包含两个域：数据域key用于存储关键字，指针域next用于存储指向下一个同义词的指针。散列表数组data中每个元素都是一个指针，指向对应的单链表的首地址。整型变量maxSize存储表长（即散列表数组的大小）。整型变量curLength存储表中当前元素的个数。

```
template <class Type>
class openHashTable : public hashTable<Type> {
    struct Node {
        Type key; // 键
        Node* next; // 指向下一个节点的指针
        Node() : next(nullptr) {} // 默认构造函数
```

```
        Node(const Type& k) : key(k), next(nullptr) {} // 构造函数
    };
    Node** data; // 散列表
    int maxSize; // 散列表的容量
    int curLength; // 散列表中元素的个数
    void resize(); // 扩展散列表
public:
    openHashTable(int len = 11, int (*h)(const Type& k, int maxSize) = defaultHash);
    ~openHashTable();
    int size() { return curLength; } // 返回散列表中元素的个数
    int capacity() { return maxSize; } // 返回散列表的容量
    bool search(const Type& k) const; // 查找元素
    bool insert(const Type& k); // 插入元素
    bool remove(const Type& k); // 删除元素
    void print(); // 输出散列表
};
```

构造函数：形参len是用户指定的数值，利用nextPrime(len)函数求出大于该数值的第一个质数作为散列表的长度；形参h是函数指针，用于可以通过实参来指定自己的散列函数。

```
template <class Type>
openHashTable<Type>::openHashTable(int len, int (*h)(const Type& k, int maxSize)) {
    curLength = 0; // 初始化元素个数为0
    hash = h; // 设置散列函数
    maxSize = nextPrime(len); // 确保容量为质数
    data = new Node*[maxSize]; // 分配内存
    for (int i = 0; i < maxSize; ++i)
        data[i] = new Node; // 为每个链表申请头结点
}
```

析构函数：释放每个单链表及散列表数组。

```
template <class Type>
openHashTable<Type>::~openHashTable() {
    for (int i = 0; i < maxSize; ++i) {
        Node* current = data[i];
        while (current) {
            Node* toDelete = current;
            current = current->next;
            delete toDelete; // 释放每个节点的内存
        }
    }
    delete[] data; // 释放散列表数组的内存
}
```

查找操作：查找关键字为k的元素是否在散列表中，查找成功返回true，查找失败返回false。开散列表的查找算法描述如下：

首先计算关键字为k的元素的基地址$H(\text{key})$，然后遍历该基地址对应的单链表，若单链表中某个结点的关键字为k，则查找成功返回true；若直到单链表为空都没有找到，则查找失败返回false。

```
template <class Type>
bool openHashTable<Type>::search(const Type& k) const {
    int pos = hash(k, maxSize); // 计算散列位置
    Node* current = data[pos]->next; // 跳过头结点
    while (current) {
        if (current->key == k) {
            return true; // 找到元素
        }
        current = current->next; // 移动到下一个节点
    }
    return false; // 没有找到元素
}
```

插入操作：插入关键字为k的元素（不允许重复）到散列表中，若该关键字已经存在，则退出程序并返回false，否则插入该元素并返回true。开散列表的插入算法描述如下：

若插入新元素使得装填因子大于1，则调用resize扩大表空间。然后计算关键字为k的元素的基地址$H(\text{key})$，遍历该基地址对应的单链表，若在单链表中找到某个结点的关键字为k，则退出程序并返回false；若直到单链表为空，都没找到关键字为k的结点，则在表头（或表尾）插入该元素并返回true

```
template <class Type>
bool openHashTable<Type>::insert(const Type& k){
    if(search(k)) return false; // 如果元素已存在，返回false
    if(curLength+1 >= maxSize) resize(); // 如果元素个数超过容量的一半，扩展散列表
    int pos = hash(k, maxSize); // 计算散列位置
    Node* newNode = new Node(k); // 创建新节点
    newNode->next = data[pos]->next; // 将新节点插入到链表头部
    data[pos]->next = newNode; // 更新头结点的指针
    curLength++; // 增加元素个数
    return true; // 插入成功
}
```

删除操作：删除散列表中关键字为k的元素。若删除成功则返回true，否则返回false。删除算法和插入算法相似，都要先查找元素的位置，若能够找到元素，则删除它并返回true，否则查找失败返回false

```
template <class Type>
bool openHashTable<Type>::remove(const Type& k) {
    int pos = hash(k, maxSize); // 计算散列位置
    Node *pre = data[pos], *p;
    while(pre->next != nullptr && pre->next->key != k)
        pre = pre->next; // 找到要删除的节点的前一个节点
    if(pre->next == nullptr) return false; // 如果没有找到，返回false
    else{
        p = pre->next; // 找到要删除的节点
        pre->next = p->next; // 更新前一个节点的指针
        delete p; // 释放要删除的节点内存
        curLength--; // 减少元素个数
        return true; // 删除成功
    }
}
```

扩大散列表空间：该函数重置散列表数组大小为大于2*maxSize的第一个质数。首先重新申请散列表数组，然后遍历每个单链表的结点，计算结点的新散列地址，然后通过改变指针的指向，将结点插入新散列表数组的单链表中（无需申请新结点），最后释放原油散列表数组。请注意：不能将各单链表直接连接到新散列表数组上，应根据新的散列表大小重新计算散列地址，再将元素插入表空间中。

```
template <class Type>
void openHashTable<Type>::resize(){
    Node **tmp = data, *p, *q;
    int i, pos, oldSize = maxSize;
    maxSize = nextPrime(2 * maxSize); // 扩展容量为原来的两倍
    data = new Node*[maxSize];
    for(i = 0; i < maxSize; i++)
        data[i] = new Node; // 为每个链表申请头结点
    for(i = 0; i < oldSize; i++){
        p = tmp[i]->next; // 获取旧链表的第一个节点
        while(p != nullptr) {
            pos = hash(p->key, maxSize); // 计算新位置
            q = p->next; // 保存下一个节点
            p->next = data[pos]->next; // 在新hash地址的表头插入p结点
            data[pos]->next = p; // 更新头结点的指针
            p = q; // 移动到下一个节点
        }
    }
    for(i = 0; i < oldSize; i++) delete tmp[i]; // 释放旧链表的内存
    delete[] tmp; // 释放旧散列表数组的内存
}
```

输出散列表：遍历散列表的每个单链表，输出所有结点。

```
template <class Type>
void openHashTable<Type>::print() {
    Node* p;
    cout << "输出开散列表中的内容: " << endl;
    for(int i = 0; i < maxSize; i++){
        p = data[i]->next; // 跳过头结点
        cout << i << ":";
        while(p != nullptr) {
            cout << "-->" << p->key; // 输出当前节点的键
            p = p->next; // 移动到下一个节点
        }
    }
    cout << endl; // 换行
}
```

闭散列表与开散列表的比较

闭散列表域开散列表的比较类似于顺序表域单链表的比较。开散列表用链接方法存储同义词，其优点是无堆积现象，其平均查找长度较短，查找、插入和删除操作易于实现。其缺点是指针需要额外空间。闭散列表无序附加指针，因而存储效率较高，当结点空间较小时，较为节省空间。但由此带来的问题是更容易产生堆积现象，而且由于空单元是查找不成功的条件，因此实现删除操作时不能简单地将待删除元素所在单元置空，否则将截断该元素后继散列地址序列的查找路径。因此闭散列表的删除操作只是在待删除元素所在的单元上做标记。当运行到一定阶段并整理后，才能真正删除有标记的单元。

散列表的查找性能分析

当查找关键字为key的元素时，首先计算散列地址$H(\text{key})$，散列表的查找过程描述如下：

1. 若散列地址为$H(\text{key})$的单元为空（或闭散列表中的已删除），则所查元素不存在，查找失败。
2. 若散列地址为$H(\text{key})$的单元中元素的关键字等于key，则找到所查元素，查找成功。
3. 若散列地址为$H(\text{key})$的单元中元素的关键字不等于key，则发生冲突，需要按照解决冲突的方法，找出下一个散列地址$H'(\text{key})$，重复步骤$1 \sim 3$，直到因查找成功或查找失败而结束查找过程。

虽然散列表在关键字和存储位置之间建立了直接映象，但是由于存在冲突，散列表的查找过程仍然是一个和关键字比较的过程，因此，仍然可用平均查找长度来衡量散列表的检索效率。为了讨论一般情况的平均查找长度，首先介绍装填因子的概念。装填因子定义为：

$$\alpha = \text{表中的元素数} \div \text{散列表长度}$$

装填因子α表示表的填满程度，装填因子越大，表明散列表中空单元越少，发生冲突的可能性，在查找时所耗费的时间就越多；装填因子越小，表明散列表中还有很多的空单元，发生冲突的可能性就越小。

处理冲突的办法	平均查找长度	平均查找长度
处理冲突的办法	ASL_{succ}	ASL_{un}
线性探测法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
二次探测法	$\frac{1}{2}\left(\ln\frac{1}{1-\alpha}\right)$	$\frac{1}{1-\alpha}$
拉链法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$