红黑树

概述

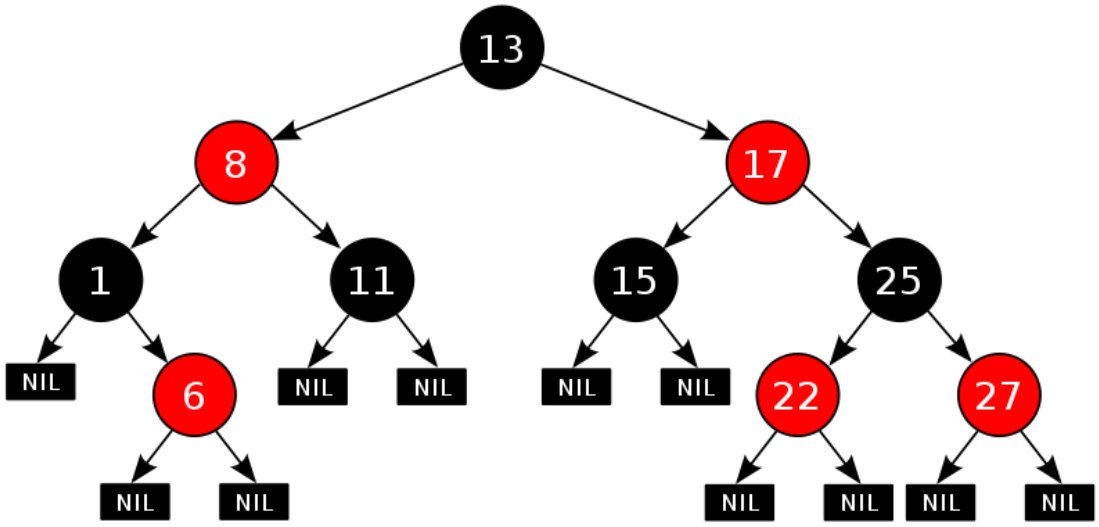
红黑树是计算机科学中的一种自平衡二元搜索树。二进制树的每个节点都有一个额外的位，该位通常被解释为节点的颜色（红色或黑色）。这些颜色位用于确保树在插入和删除期间保持大致平衡。

通过以满足某些属性的方式将树的每个节点绘制两种颜色，从而保留平衡，这些属性共同限制了树在最坏的情况下变得不平衡的程度。修改树时，新树随后将重新排列和重新绘制以恢复着色属性。属性的设计方式使这种重新排列和重新着色可以有效地执行。

树的平衡并不完美，但它足够好，可以保证在O（log n）时间中搜索，其中n是树中的元素总数。插入和删除操作，以及树重新排列和重新着色，也执行在 O（log n）时间。 [3]

跟踪每个节点的颜色只需要每个节点 1 位信息，因为只有两种颜色。树不包含任何其他特定于它是红-黑树的数据，因此其内存占用量几乎与经典（未着色）二进制搜索树相同。在许多情况下，无需额外的内存成本即可存储额外的信息位。

数据结构




```
Red–black tree ADT{
    enum color_t { BLACK, RED };
    struct Node {
        Node* parent;
        Node* left;
        Node* right;
        enum color_t color;
        int key;
    };
    Node* GetParent(Node* n) ;
    Node* GetGrandParent(Node* n) ;
    Node* GetUncle(Node* n);
```

```
    void RotateLeft(Node* n) ;
    Node* Insert(Node* root, Node* n);
    void InsertRecurse(Node* root, Node* n);
    void InsertRepairTree(Node* n) ;
    void ReplaceNode(Node* n, Node* child) ;
    void DeleteOneChild(Node* n) ;
    void DeleteCase1(Node* n);
}
```

关键的基本操作的具体实现概述

红黑树的插入和插入修复

现在我们了解了二叉查找树的插入，接下来，咱们便来具体了解红黑树的插入操作。红黑树的插入相当于在二叉查找树插入的基础上，为了重新恢复平衡，继续做了插入修复操作。

假设插入的结点为 z，红黑树的插入伪代码具体如下所示：

```
RB-INSERT(T, z){
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z) }
```

我们把上面这段红黑树的插入代码，跟我们之前看到的二叉查找树的插入代码，可以看出，RB-INSERT(T, z)前面的第 1-13 行代码基本就是二叉查找树的插入代码，然后第 14-16 行代码把 z 的左孩子、右孩子都赋为叶结点 nil，再把 z 结点着为红色，最后为保证红黑性质在插入操作后依然保持，调用一个辅助程序 RB-INSERT-FIXUP 来对结点进行重新着色，并旋转。

换言之

如果插入的是根结点，因为原树是空树，此情况只会违反性质 2，所以直接把此结点涂为黑色。

如果插入的结点的父结点是黑色，由于此不会违反性质 2 和性质 4，红黑树没有被破坏，所以此时也是什么也不做。

但当遇到下述 3 种情况时：

插入修复情况 1：如果当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）

是红色

插入修复情况 2：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的右子

插入修复情况 3：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的左子

又该如何调整呢？答案就是根据红黑树插入代码 RB-INSERT(T, z)最后一行调用的 RB-INSERT-FIXUP（T,z）所示操作进行，具体如下所示：

```
RB-INSERT-FIXUP（T,z）{
 1 while color[p[z]] = RED
 2       do if p[z] = left[p[p[z]]]
 3             then y ← right[p[p[z]]]
 4                  if color[y] = RED
 5                       then color[p[z]] ← BLACK                    ▷ Case 1
 6                            color[y] ← BLACK                      ▷ Case 1
 7                            color[p[p[z]]] ← RED                  ▷ Case 1
 8                            z ← p[p[z]]                           ▷ Case 1
 9                       else if z = right[p[z]]
10                                 then z ← p[z]                    ▷ Case 2
11                                      LEFT-ROTATE(T, z)           ▷ Case 2
12                                 color[p[z]] ← BLACK              ▷ Case 3
13                                 color[p[p[z]]] ← RED             ▷ Case 3
14                                 RIGHT-ROTATE(T, p[p[z]])         ▷ Case 3
15             else (same as then clause
                          with "right" and "left" exchanged)
16 color[root[T]] ← BLACK
}
```

下面，咱们来分别处理上述 3 种插入修复情况。

插入修复情况 1：当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色。

即如下代码所示：

```
 1 while color[p[z]] = RED
 2       do if p[z] = left[p[p[z]]]
 3             then y ← right[p[p[z]]]
 4                  if color[y] = RED
```

此时父结点的父结点一定存在，否则插入前就已不是红黑树。

与此同时，又分为父结点是祖父结点的左子还是右子，对于对称性，我们只要解开一个方向就可以了。

在此，我们只考虑父结点为祖父左子的情况。

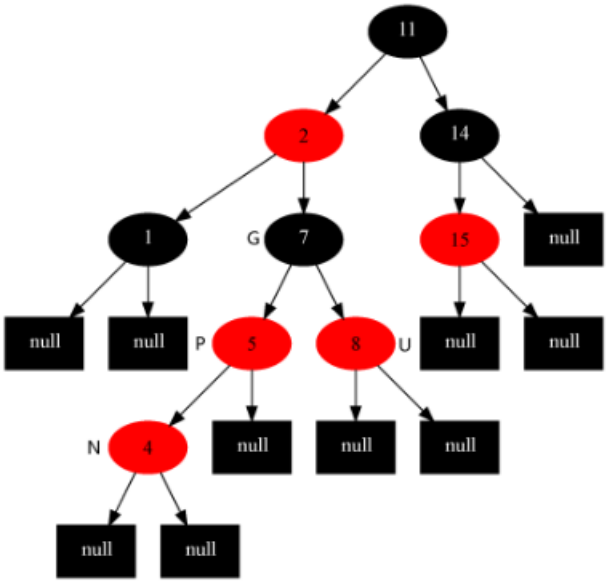
同时，还可以分为当前结点是其父结点的左子还是右子，但是处理方式是一样的。我们将此归为同一类。

对策：将当前结点的父结点和叔叔结点涂黑，祖父结点涂红，把当前结点指向祖父结点，从新的当前结点重新开始算法。即如下代码所示：

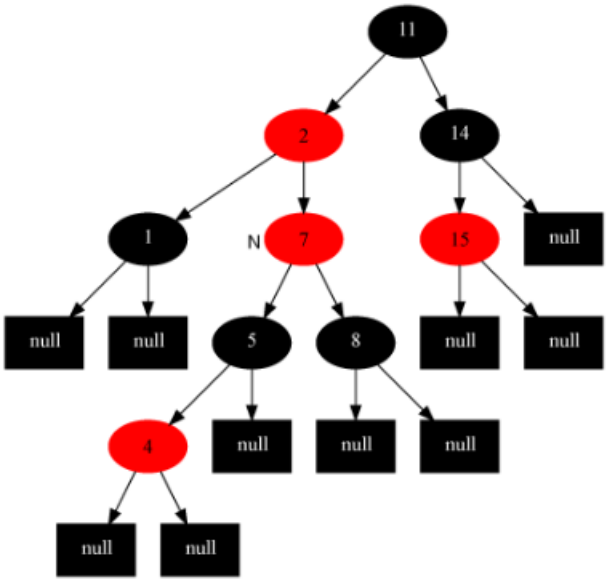
```
 5                       then color[p[z]] ← BLACK                    ▷ Case 1
 6                            color[y] ← BLACK                      ▷ Case 1
```

```
7                color[p[p[z]]] ← RED                ▷ Case 1
8                z ← p[p[z]]                         ▷ Case 1
```

针对情况1，变化前（图片来源：saturnman）[当前结点为4结点]：



变化后：

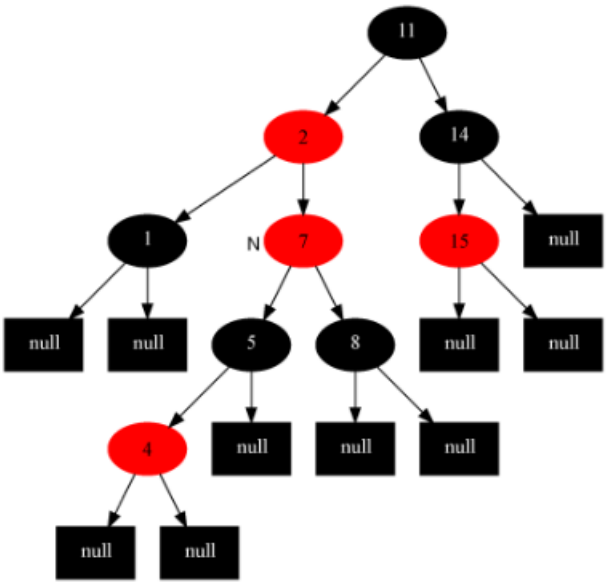


插入修复情况 2：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的右子

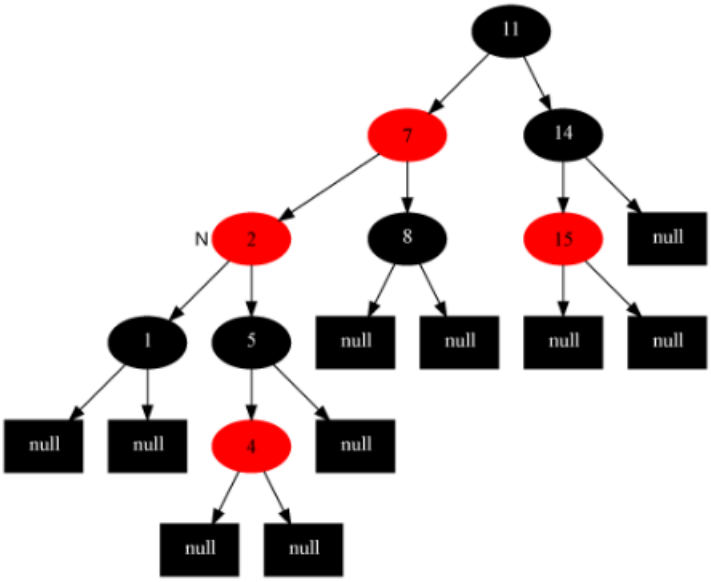
对策：当前结点的父结点做为新的当前结点，以新当前结点为支点左旋。即如下代码所示：

```
9                else if z = right[p[z]]
10                    then z ← p[z]                  ▷ Case 2
11                         LEFT-ROTATE(T, z)         ▷ Case 2
```

如下图所示，变化前[当前结点为7结点]：



变化后：



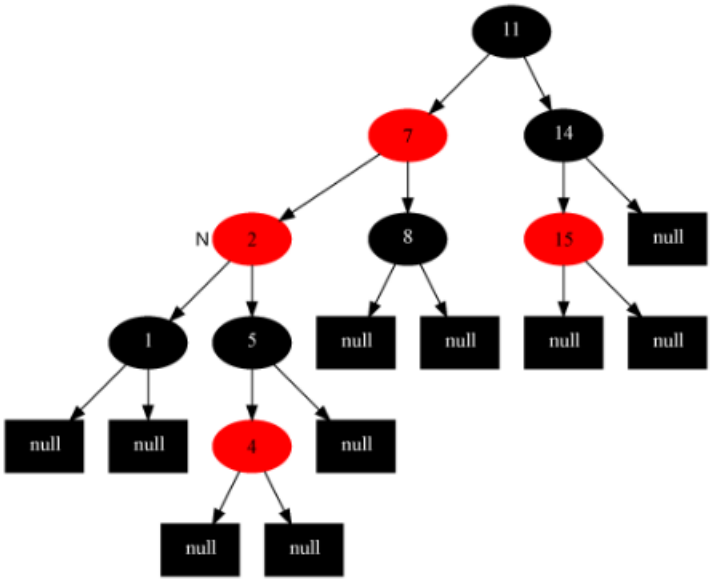
插入修复情况 3：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的左子

解法：父结点变为黑色，祖父结点变为红色，在祖父结点为支点右旋，操作代码为：

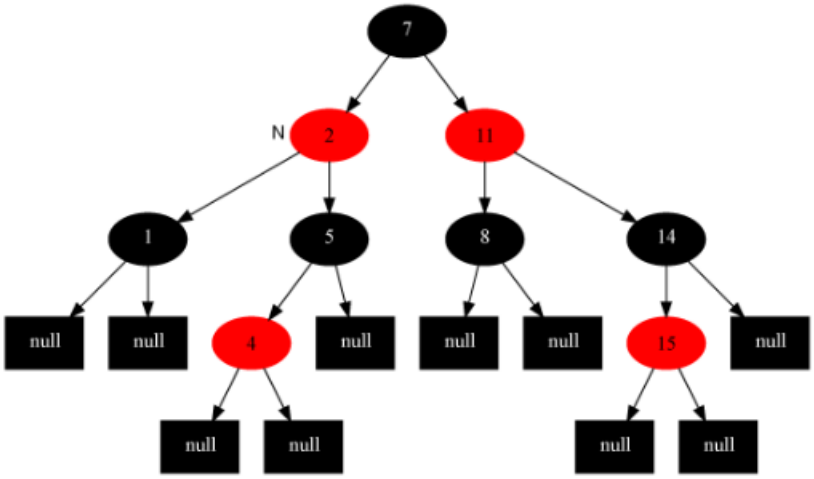
```
12                    color[p[z]] ← BLACK              ▷ Case 3
13                    color[p[p[z]]] ← RED             ▷ Case 3
14                    RIGHT-ROTATE(T, p[p[z]])         ▷ Case 3
```

最后，把根结点涂为黑色，整棵红黑树便重新恢复了平衡。

如下图所示[当前结点为2结点]



变化后：



应用

红黑树和 AVL 树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。这不只是使它们在时间敏感的应用如即时应用(real time application)中有价值，而且使它们有在提供最坏情况担保的其他数据结构中作为建造板块的价值；例如，在计算几何中使用的很多数据结构都可以基于红黑树。

红黑树在函数式编程中也特别有用，在这里它们是最常用的持久数据结构之一，它们用来构造关联数组和集合，在突变之后它们能保持为以前的版本。除了 O(log n)的时间之外，红黑树的持久版本对每次插入或删除需要 O(log n)的空间。

红黑树是 2-3-4 树的一种等同。换句话说，对于每个 2-3-4 树，都存在至少一个数据元素是同样次序的红黑树。在 2-3-4 树上的插入和删除操作也等同于在红黑树中颜色翻转和旋转。这使得 2-3-4 树成为理解红黑树背后的逻辑的重要工具，这也是很多介绍算法的教科书在红黑树之前介绍 2-3-4 树的原因，尽管 2-3-4 树在实践中不经常使用。

哈希表

概述

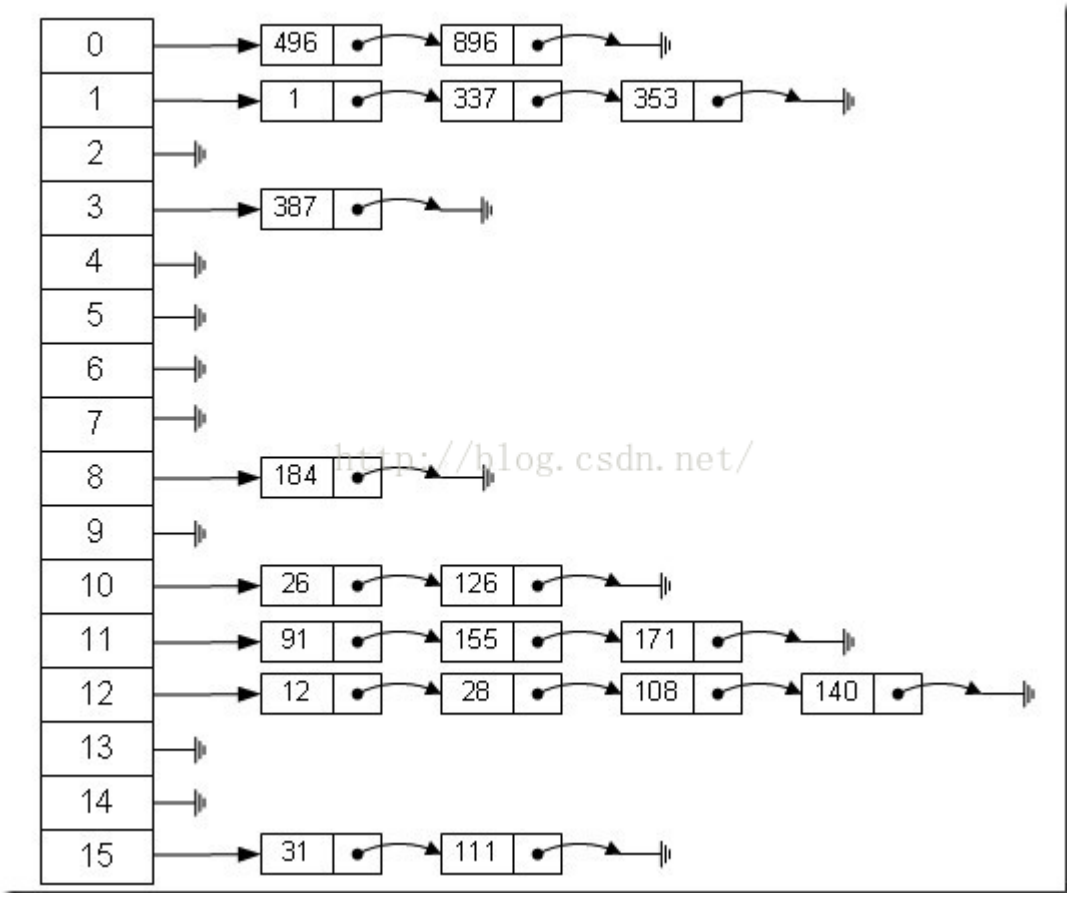
哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

记录的存储位置=f(关键字)

这里的对应关系 f 称为散列函数，又称为哈希（Hash 函数），采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表（Hash table）。

哈希表 hashtable(key，value) 就是把 Key 通过一个固定的算法函数既所谓的哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将 value 存储在以该数字为下标的数组空间里。(或者：把任意长度的输入（又叫做预映射，pre-image)，通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。)　　而当使用哈希表进行查询的时候，就是再次使用哈希函数将 key 转换为对应的数组下标，并定位到该空间获取 value，如此一来，就可以充分利用到数组的定位性能进行数据定位。

数据结构



```
Hash ADT{
    struct HashTable{
        int TableSize;
        Position *TheList;
    };
    struct ListNote{
        int element;
        Position next;
    };
    int Hash(int key,int tablesize);
    int NextPrime(int x);
    HashTbl InitalizeTable(int TableSize);
    void DestroyTable(HashTbl H);
    Position Find(int key,HashTbl H);
    void Insert(int key, HashTbl H);
    void Delete(int key,HashTbl H);
    int Hash(int key,int tablesize);
    int NextPrime(int x);
    HashTbl InitalizeTable(int TableSize);
    Position Find(int key,HashTbl H);
    void Insert(int key,HashTbl H);
    void Delete(int key,HashTbl H);
}
```

关键的基本操作的具体实现概述

初始化：

对于哈希表，最影响它性能的就是按什么创建哈希表。因此，创建它有如下方法。

平方取中法:先通过求关键字的平方值扩大相近数的差别，然后根据表长度取中间的几位数作为散列函数值。

除余法:它是以表长 m 来除关键字，取其余数作为散列地址，即 h(key)=key％m。

相乘取整法:首先用关键字 key 乘上某个常数 A（A 大于 0 小于 1）并抽取出 key.A 的小数部分；然后用 m 乘以该小数后取整。

随机数法:选择一个随机函数，取关键字的随机函数值为它的散列地址。

```
Hash init(int bucket){
    assert(bucket>0);
    printf("Enter init \n");
    Hash hash;
    int i = 0;
    hash = (Hash)malloc(sizeof(struct HASH));
    if(hash == NULL)
    {
        printf("init mem    error ! \n");
        exit(-1);
    }
```

```
hash->bucket = bucket;
hash->list = (Node)malloc(sizeof(struct NODE)*hash->bucket);
if(hash->list == NULL)
{
        printf("init mem    error ! \n");
        exit(-1);
}
for(i=0;i<bucket;i++)
{
    hash->list[i].data = 0;
    hash->list[i].key= 0;
    hash->list[i].next= NULL;
}
return hash;}
```

应用

基因测试

题目描述

现代的生物基因测试已经很流行了。现在要测试色盲的基因组。有 N 个色盲的人和 N 个非色盲的人参与测试。

基因组包含 M 位基因，编号 1 至 M。每一位基因都可以用一个字符来表示，这个字符是‘A’、'C'、'G'、'T'四个字符之一。

例如： N = 3, M = 8。

色盲者 1 的 8 位基因组是: AATCCCAT

色盲者 2 的 8 位基因组是: ACTTGCAA

色盲者 3 的 8 位基因组是: GGTCGCAA

正常者 1 的 8 位基因组是: ACTCCCAG

正常者 2 的 8 位基因组是: ACTCGCAT

正常者 3 的 8 位基因组是: ACTTCCAT

通过认真观察研究，生物学家发现，有时候可能通过特定的连续几位基因，就能区分开是正常者还是色盲者。

例如上面的例子，不需要 8 位基因，只需要看其中连续的 4 位基因就可以判定是正常者还是色盲者，这 4 位基因编号分别是：

（第 2、3、4、5）。也就是说，只需要看第 2,3,4,5 这四位连续的基因，就能判定该人是正常者还是色盲者。

比如：第 2,3,4,5 这四位基因如果是 GTCG，那么该人一定是色盲者。

生物学家希望用最少的连续若干位基因就可以区别出正常者和色盲者, 输出满足要求的连续基因的最少位数。

输入格式

第一行，两个整数： N 和M。 1 <= N <= 500， 3 <= M <= 500。

接下来有 N 行，每一行是一个长度为 M 的字符串，第 i 行表示第 i 位色盲者的基因组。

接下来又有 N 行，每一行是一个长度为 M 的字符串，第 i 行表示第 i 位正常者的基因

组。

输出格式

一个整数。

输入样例

3 8

AATCCCAT

ACTTGCAA

GGTCGCAA

ACTCCCAG

ACTCGCAT

ACTTCCAT

输出样例

4

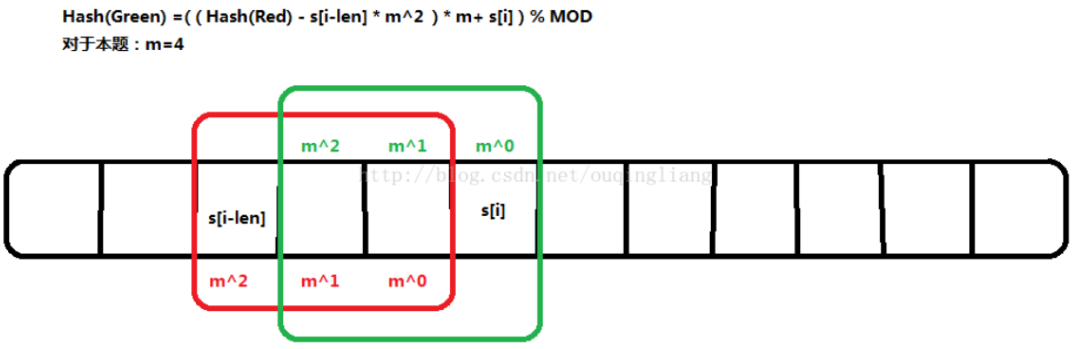
【做法 1】 很容易就想到大暴力。枚举区间[i,j]，然后暴力判断是否合法，判断两个字符串是否不等是 O(m)的，则判断 n*n 次的时间就是 O(n^2*m)。则总的时间复杂度为 O(n^2*m^3)。

【做法 2】可以想到哈希。在判断两个字符串是否相等时可以把字符串哈希。把色盲者的 n 个 Hash 值存到表里，然后把正常人的 Hash 值在哈希表里查找即可。时间复杂度为四次方级别。

【做法 3】因为，枚举区间时是顺着枚举的。则考虑可以把上一次的结果记住，这一次就可以避免重复计算了。具体细节这里略去。则时间可降低一维。为 O(n^3)≈O(125000000)，在极限数据时还会超时。

【做法 4】

如果答案再大一点，就不那么好了。如果答案小一点，就不符合情况。那么，我们就可以二分答案!



二分答案 len，再从小到大枚举起始点 i，则区间变成了[i,len+i]，每一次区间都会向右移一位。应该怎样快速的计算 Hash 值呢?

```
#include<cstdio>
#include<iostream>
#include <cstring>
#include <algorithm>
#include <string>
#include <cstdlib>
```

```
#include<vector>
using namespace std;
typedef long long LL;
#define INF 0x3fffffff
#define Maxn 510
#define MOD 3000007LL
int n,m,n2;
short a[Maxn<<1][Maxn];LL h[Maxn<<1];
int hash[MOD];
bool pd(int i)
{
    bool flag;
    for(int j=0;j<n;j++)
        hash[h[j]] = i;
    flag = true;
    for(int j=n;j<n2;j++)
        if(hash[h[j]]==i)
        {
            flag = false;
            break;
        }
    return flag;
}

bool check(int len)
{
    memset(h,0,sizeof(h));
    memset(hash,0,sizeof(hash));
    for(int i=0;i<len;i++)
        for(int j=0;j<n2;j++)
            h[j] = ( (h[j]*4) + a[j][i] ) % MOD;
    LL Pow = 1;
    for(int i=1;i<len;i++)
        Pow = (Pow*4LL) % MOD;
    if(pd(len-1))
        return true;
    for(int i=len;i<m;i++)
    {
        for(int j=0;j<n2;j++)
            h[j] = ( ( ( h[j] - ((0LL+a[j][i-len]) * Pow) % MOD + MOD) *4LL ) + a[j][i] ) %
MOD;
            if(pd(i))
                return true;
    }
```

```
        return false;
    }
     int main()
    {
        freopen("1810.in","r",stdin);
        freopen("1810.out","w",stdout);
        scanf("%d%d",&n,&m);
        n2 = n<<1;
            char ch;
        for(int i=0;i<n2;i++)
        {
            getchar();
            for(int j=0;j<m;j++)
            {
                ch = getchar();
                if(ch == 'C') a[i][j] = 1;
                if(ch == 'T') a[i][j] = 2;
                if(ch == 'G') a[i][j] = 3;
            }
        }
        int l = 1;
        int r = m;while(l+1<r)
        {
            int mid = (l+r)>>1;
            if(check(mid))
                 r = mid;
            else
                l = mid;
        }
        printf("%d\n",r);
        return 0;
}
```

BK 树

概述

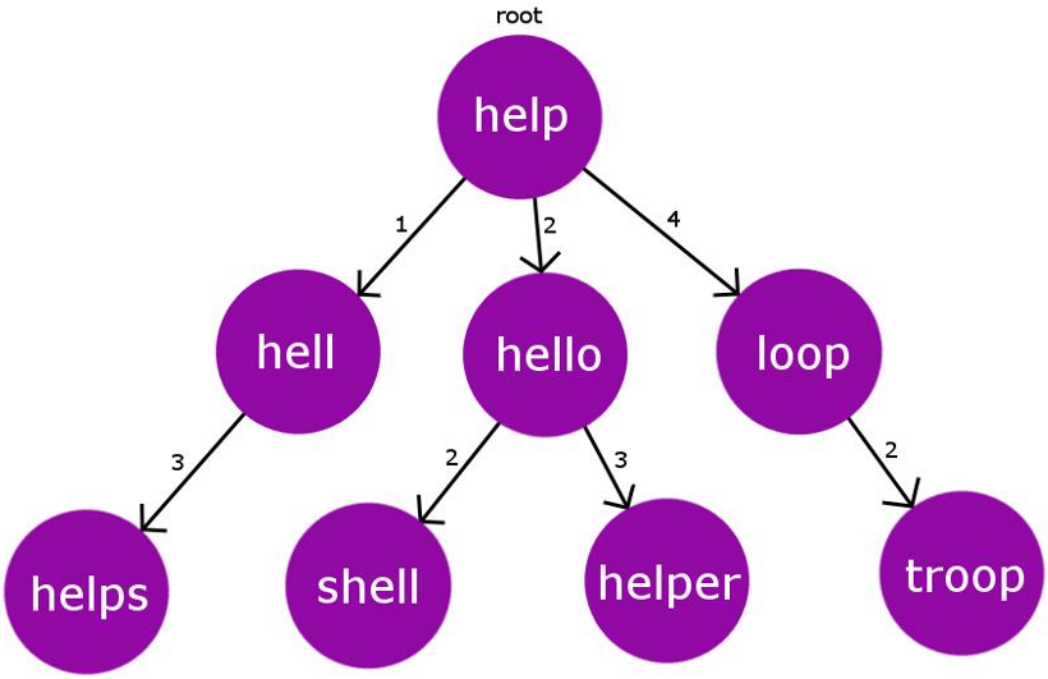
BK Tree 或 Burkhard Keller Tree 是一种数据结构，用于根据编辑距离（Levenshtein 距离）概念执行拼写检查。 BK 树也用于近似字符串匹配。基于该数据结构，可以实现许多软件中的各种自动校正特征。

假设我们有一个单词字典，然后我们有一些其他的单词要在字典中检查拼写错误。我们需要收集字典中与给定单词非常接近的所有单词。例如，如果我们检查一个单词“ruk”，我们将有{“truck”，“buck”，“duck”，.....}。因此，拼写错误可以通过删除单词中的字符或在单词中添加新字符或通过将单词中的字符替换为适当的字符来更正。因此，我们将使用编辑距离作为衡量错误拼写单词与字典中单词的正确性和匹配的度量。

数据结构

BK 树中的节点将代表我们字典中的单个单词，并且节点的数量与字典中单词的数量完全相同。边将包含一些整数权重，它将告诉我们从一个节点到另一个节点的编辑距离。假设我们有一个从节点 u 到节点 v 的边缘有一些边缘权重 w，那么 w 是将字符串 u 转换为 v 所需的编辑距离。

考虑字典集合：{“help”，“hell”，“hello”}。因此，对于这一字典集合，我们的 BK 树将如下




```
public class BKTree<T>{
    private final MetricSpace<T> metricSpace;
    private Node<T> root;
    public BKTree(MetricSpace<T> metricSpace);
    public static <E> BKTree<E> mkBKTree(MetricSpace<E> ms, Collection<E> elems);
    public void put(T term);
    public Set<T> query(T term, double radius);
    private static final class Node<T>();
    public void add(MetricSpace<T> ms, T value);
    public void query(MetricSpace<T> ms, T term, double radius, Set<T> results);
    }
```

关键的基本操作的具体实现概述

编辑距离：

又称 Levenshtein 距离，是指两个字串之间，由一个转成另一个所需的最少编辑操作次数。该类中许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。

```
 *
 * 使用动态规划算法。算法复杂度：m*n。
*/
public class LevensteinDistance implements MetricSpace<String>{
    private double insertCost = 1;          // 可以写成插入的函数，做更精细化处理
    private double deleteCost = 1;          // 可以写成删除的函数，做更精细化处理
    private double substitudeCost = 1.5;// 可以写成替换的函数，做更精细化处理。比如使用键盘距离。

    public double computeDistance(String target,String source){
        int n = target.trim().length();
        int m = source.trim().length();

        double[][] distance = new double[n+1][m+1];

        distance[0][0] = 0;
        for(int i = 1; i <= m; i++){
            distance[0][i] = i;
        }
        for(int j = 1; j <= n; j++){
            distance[j][0] = j;
        }

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <=m; j++){
                double min = distance[i-1][j] + insertCost;

                if(target.charAt(i-1) == source.charAt(j-1)){
                    if(min > distance[i-1][j-1])
                        min = distance[i-1][j-1];
                }else{
                    if(min > distance[i-1][j-1] + substitudeCost)
                        min = distance[i-1][j-1] + substitudeCost;
                }

                if(min > distance[i][j-1] + deleteCost){
                    min = distance[i][j-1] + deleteCost;
                }

                distance[i][j] = min;
            }
        }

        return distance[n][m];
```

```
    }

    @Override
    public double distance(String a, String b) {
        return computeDistance(a,b);
    }

    public static void main(String[] args) {
        LevensteinDistance distance = new LevensteinDistance();
        System.out.println(distance.computeDistance("你好","好你"));
    }
}
```

应用

【问题描述】

给定一组单词，如何找到拼写错误的单词最接近的正确单词？

【问题分析】

首先，我们需要设置容差值。此容差值只是从拼写错误的单词到字典中正确单词的最大编辑距离。因此，要在容差限度内找到符合条件的正确单词，Naive 方法将迭代字典中的所有单词并收集容差限制内的单词。但是这种方法具有 O（n * m * n）时间复杂度（n 是 dict []中的单词数，m 是正确单词的平均大小，n 是拼写错误单词的长度），对于较大的字典大小超时。

假设我们有一个拼写错误的单词“oop”，容差限制为 2.现在，我们将看到我们将如何收集给定拼写错误单词的预期正确值。

迭代 1：我们将开始检查根节点的编辑距离。 D（“oop” - >“help”）= 3.现在我们将迭代其编辑距离范围[D-TOL，D + TOL]，即[1,5]

迭代 2：让我们从最高可能的编辑距离中开始迭代，即节点“循环”与编辑距离 4.现在我们将再次从拼写错误的单词中找到它的编辑距离。 D（“oop”，“loop”）= 1。

这里 D = 1，即 D <= TOL，所以我们将“循环”添加到预期的正确单词列表并处理其子节点的编辑距离在[D-TOL，D + TOL]中，即[1,3]

迭代 3：现在，我们处于节点“troop”。我们将再一次检查拼写错误单词的编辑距离。 D（“oop”，“troop”）= 2.再次 D <= TOL，因此我们再次将“troop”添加到预期的正确单词列表中。

对于从根节点开始直到最底部叶节点的[D-TOL，D + TOL]范围内的所有单词，我们将继续相同。这类似于树上的 DFS 遍历，有选择地访问边缘权重在某个给定范围内的子节点。

因此，最后我们将只留下拼写错误的单词“oop”的 2 个预期单词，即{“loop”，“troop”}

```
// C++ program to demonstrate working of BK-Tree
#include "bits/stdc++.h"
using namespace std;
Node RT;
Node tree[MAXN];
// index for current Node of tree
int ptr;
int min(int a, int b, int c)
```

```
{
	return min(a, min(b, c));
}
// Edit Distance
// Dynamic-Approach O(m*n)
int editDistance(string& a,string& b)
{
	int m = a.length(), n = b.length();
	int dp[m+1][n+1];

	// filling base cases
	for (int i=0; i<=m; i++)
		dp[i][0] = i;
	for (int j=0; j<=n; j++)
		dp[0][j] = j;

	// populating matrix using dp-approach
	for (int i=1; i<=m; i++)
	{
		for (int j=1; j<=n; j++)
		{
			if (a[i-1] != b[j-1])
			{
				dp[i][j] = min( 1 + dp[i-1][j], // deletion
								1 + dp[i][j-1], // insertion
								1 + dp[i-1][j-1] // replacement
							);
			}
			else
				dp[i][j] = dp[i-1][j-1];
		}
	}
	return dp[m][n];
}
vector <string> getSimilarWords(Node& root,string& s)
{
	vector < string > ret;
	if (root.word == "")
	return ret;

	// calculating editdistance of s from root
	int dist = editDistance(root.word,s);

	// if dist is less than tolerance value
```

```
        // add it to similar words
        if (dist <= TOL) ret.push_back(root.word);

        // iterate over the string havinng tolerane
        // in range (dist-TOL , dist+TOL)
        int start = dist - TOL;
        if (start < 0)
        start = 1;

        while (start < dist + TOL)
        {
                vector <string> tmp =
                        getSimilarWords(tree[root.next[start]],s);
                for (auto i : tmp)
                        ret.push_back(i);
                start++;
        }
        return ret;
}

// driver program to run above functions
int main(int argc, char const *argv[])
{
        // dictionary words
        string dictionary[] = {"hell","help","shel","smell",
                                        "fell","felt","oops","pop","oouch","halt"
                                        };
        ptr = 0;
        int sz = sizeof(dictionary)/sizeof(string);

        // adding dict[] words on to tree
        for(int i=0; i<sz; i++)
        {
                Node tmp = Node(dictionary[i]);
                add(RT,tmp);
        }

        string w1 = "ops";
        string w2 = "helt";
        vector < string > match = getSimilarWords(RT,w1);
        cout << "similar words in dictionary for : " << w1 << ":\n";
        for (auto x : match)
                cout << x << endl;
```

```
        match = getSimilarWords(RT,w2);
        cout << "Correct words in dictionary for " << w2 << ":\n";
        for (auto x : match)
            cout << x << endl;

        return 0;
    }
```

二项堆

概述

二项堆：它是一组由二项树组成的结构。

1）二项树 Bk 是一棵具有以下性质的树(这是一个递归定义)[1]:

A: 共有 2^k 个节点

B: 树的高度为 k

C:在深度 i 处恰有个节点，其中 i = 0, 1, 2, ... k

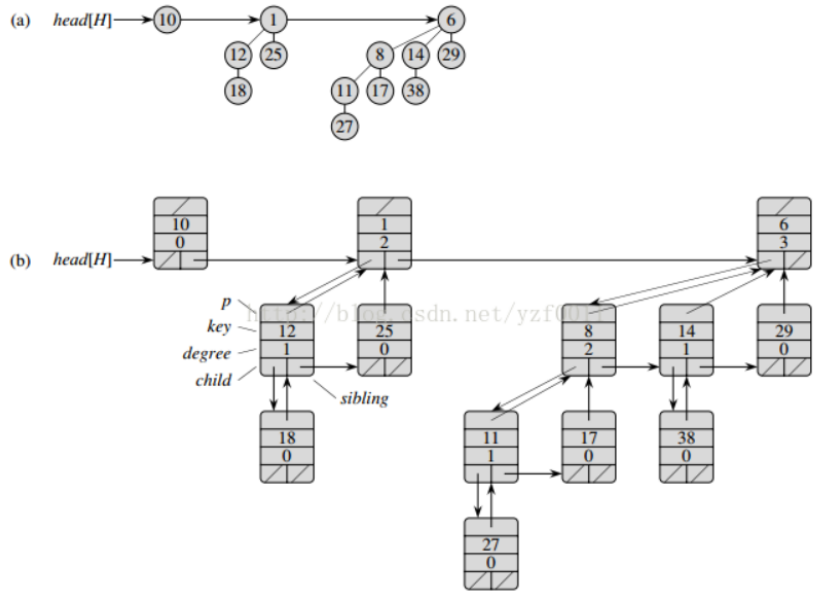
D: 根的度数为 k，它大于任何其它节点的度数，并且如果根的子女从左到右编号为 k - 1, k - 2, ..., 0，子女 i 是子树 Bi 的根。

2）二项堆由一组满足下面性质的二项树组成[3]:

A: 每个二项树遵循最小堆性质：节点的关键字大于或等于其父节点的关键字。

B: 对于任意的非负整数 k，在二项堆中至多有一棵二项树的根具有度数 k

具体形式如图:



[4]

上图（a）给出的二项堆 H 的 head 节点指向它所包含的的度数最小的一棵二项树，最小二项树再指向度数次最小的一棵二项树，同时我们会发现一个拥有 n 个节点的二项堆，它所拥有的二项树刚好是它的二进制表示形式。图 (a) 中共有 13 个节点，即 n = 13 = (1101)，所以这个二项堆包含的二项树的度数分别是 0，2，3，每个对应度数二项树包含的节点个数为 2^0, 2^2, 2^ 3。

通过图（b）我们可得知，每棵二项树通过左孩子右兄弟方式进行链接。同时二项树的

根作为二项堆的根列表，并通过兄弟链进行连接。

数据结构

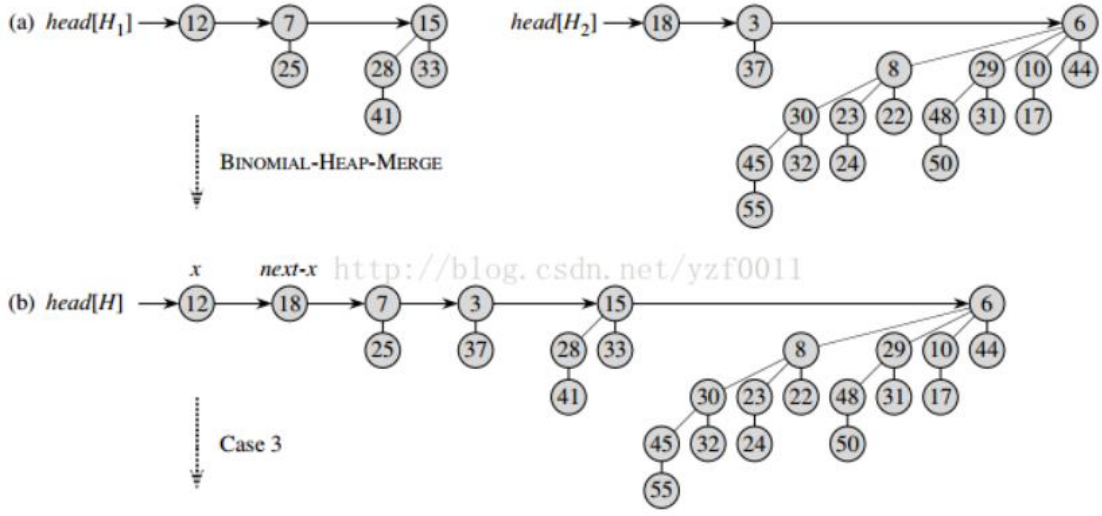
```
// basic structure of binomial heap node
typedef struct Node
{
    ElemType key;
    int degree;
    struct Node *p;
    struct Node *child;
    struct Node *sibling;
}Node;
// the structure of binomial heap
typedef struct heap
{
    Node * head;
}*BinomialHeap;
```

关键的基本操作的具体实现概述

合并两个二项堆 h1 和 h2

1）步骤

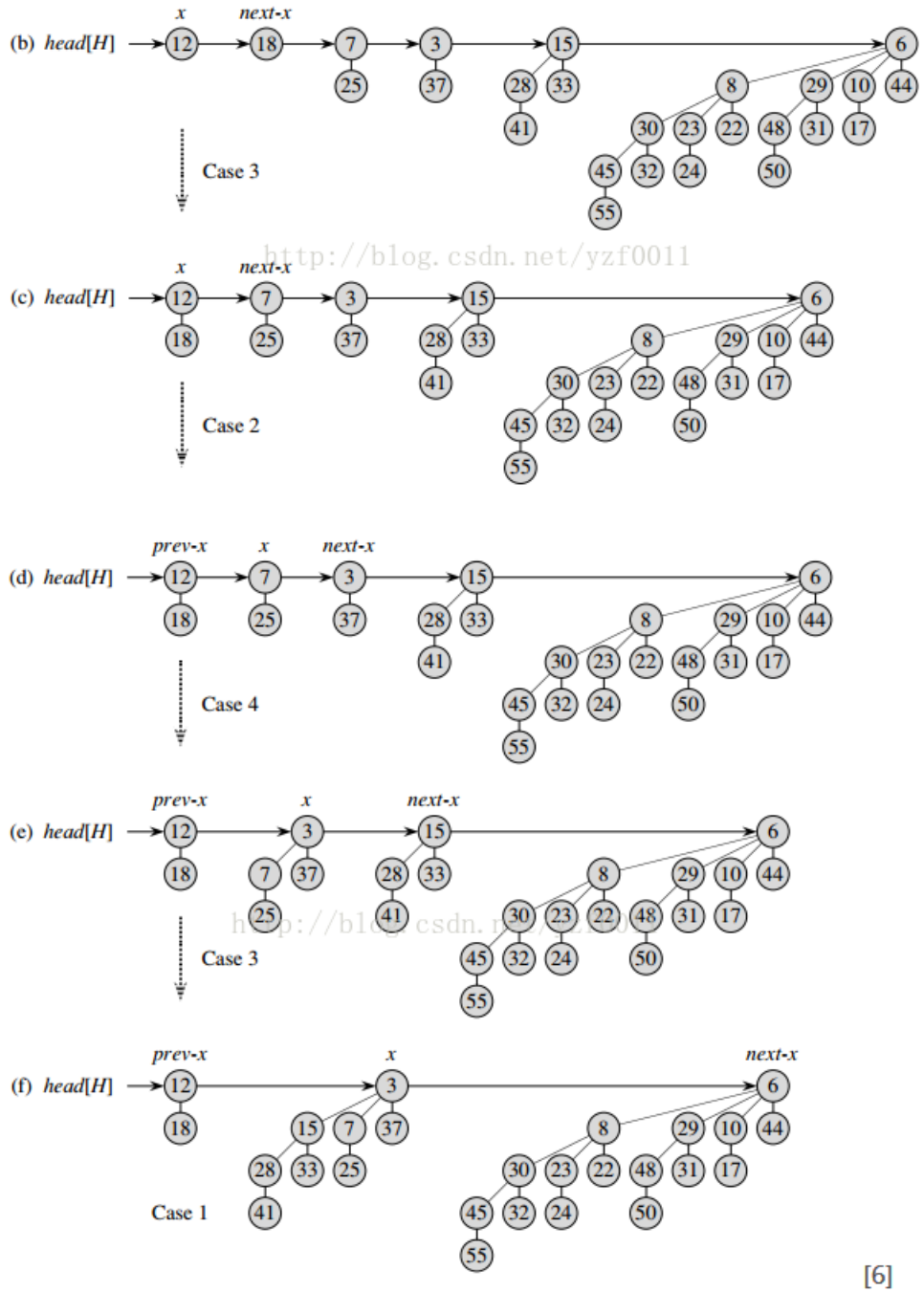
A: 将两个二项堆所有包含的二项树拆分，然后按度数递增的方式将所有的根列表进行串联，并创建一棵新的二叉树，使得其 head 节点指向这串联的根列表的第一个节点。




```
// merge the root lists of h1 and h2 into a single linked list
// that is sorted into monotonically increasing order.
Node * binomialMerge(BinomialHeap h1, BinomialHeap h2)
{
    Node * firstNode = NULL;
    Node * p = NULL;
    Node * p1 = h1->head;
```

```
    Node * p2 = h2->head;
    if (p1 == NULL || p2 == NULL)
    {
        if(p1 == NULL)
            firstNode = p2;
        else
            firstNode = p1;
        return firstNode;
    }
    if (p1->degree < p2->degree)
    {
        firstNode = p1;
        p = firstNode;
        p1 = p1->sibling;
    }
    else
    {
        firstNode = p2;
        p = firstNode; p2 = p2->sibling
    }
    while (p1 != NULL && p2 != NULL)
    {
        if (p1->degree < p2->degree)
        {
            p->sibling = p1;
            p = p1;
            p1 = p1->sibling;
        }
        Else
        {
            p->sibling = p2;
            p = p2;
            p2 = p2->sibling;
        }
    }
    if (p1 != NULL)
        p->sibling = p1;
    else
        p->sibling = p2;
    return firstNode;
}
```

B:串连后，将其中有相同度数的二项树进行合并，并使之符合最小堆性质，也要修改其中有的二项堆的父亲指针（即 node 数据结构中的 p）。



[6]

由上面的图我们可以看出 case3 和 case1 是需要游动的 prev-x, x 和 nex-x 等指针的，剩下的情况就是 degree[x] == degree[next-x]所以要尽心 Binomial heap link，但必须使得 link 后的二项树满足最小堆性质。

```
// unite heaps h1 and h2, returning the resulting heap
BinomialHeap binomialHeapUnion(BinomialHeap *h1, BinomialHeap *h2)
{
    BinomialHeap h = makeBinomialHeap();
    h->head = binomialMerge(*h1, *h2);
    free(*h1);
    *h1 = NULL;
    free(*h2);
```

```
    *h2 = NULL;
    if (h->head == NULL)
        return h;
    Node * prev = NULL;
    Node * x = h->head;
    Node * next = x->sibling;
    while (next != NULL)
    {
        if (x->degree!= next->degree|| (next->sibling != NULL&&x->degree==
    next->sibling->degree))
         {
            // case 3 or case 2, just move forward
            prev = x;
            x = next;
        }
        else if (x->key <= next->key)
        {
            x->sibling = next->sibling;
            binomialLink(next, x);
        }
        Else
        {
            if (prev == NULL)
                h->head = next;
            else
                prev->sibling = next;
            binomialLink(x, next);
        }
        // change the next
        next = x->sibling;
    }
    return h;
}
```

应用

【问题描述】

找到一组数中的最小值

由最小堆性质，我们可以得出二项堆 h 的最小值

```
    // find the minimum key from the binomial heap
    // and return the minimum node's pointer
    Node * binomialHeapMinimum(BinomialHeap h)
    {
        Node * y = NULL;
        Node * x = h->head;
```

```
        if(x != NULL)
        {
            int min = x->key;
            y = x;
            x = x->sibling;
            while (x != NULL)
            {
                if(x->key < min)
                {
                    min = x->key;
                    y = x;
                }
                x = x->sibling;
                }
            }
        return y;
    }
```

参考文献

[1] fourye007，https://blog.csdn.net/yzf0011/article/details/56004750[DB/OL]

[2] SeanC52111，https://www.jianshu.com/p/cedbd94f4f45 [DB/OL]

[3] Wikipedia，https://en.wikipedia.org/wiki/Search_tree [DB/OL]

[4] 流浪的 Coder，https://www.cnblogs.com/big-devil/p/8590242.html，哈希表及其常用算法（代码实例），[DB/OL]