树结构调研资料

一、Search trees

B 树

1. 概述：

B 树是一种多路平衡查找树，它的每一个节点最多包含 K 个孩子，K 被称为 B 树的阶。K 的大小取决于磁盘页的大小。

下面来具体介绍一下 B-树（Balance Tree），一个 m 阶的 B 树具有如下几个特征：

1.根结点至少有两个孩子。

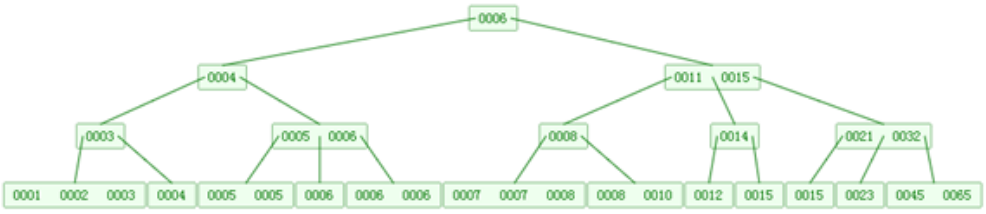
2.每个中间节点都包含 k-1 个元素和 k 个孩子，其中 m/2 <= k <= m

3.每一个叶子节点都包含 k-1 个元素，其中 m/2 <= k <= m

4.所有的叶子结点都位于同一层。

5.每个节点中的元素从小到大排列，节点当中 k-1 个元素正好是 k 个孩子包含的元素的值域分划。[1]

下图是一个 M=4 阶的 B 树:



2. 数据结构：

ADT BTree{

数据对象：D 是具有相同特性的数据元素的集合

数据关系：R1={<ai-1,ai>|ai-1,ai∈D，i=2，...,n}

R2={<ptr[i-1],ptr[i]>|i=1...,n}

约定 a1|key[1]为关键字数组头，an|key[p-<keynum]为关键字数组尾

约定 ptr[i]为结点的第 i 个子树

基本操作：

InitBTree(t);

//初始化 B 树

SearchBTNode(BTNode *p,KeyType k);

//在结点 p 中查找关键字 k 的插入位置 i

Result SearchBTree(BTree t,KeyType k);

//在 B 树查找关键字 k 的插入位置，返回查找结果

InsertBTNode(BTNode *&p,int i,KeyType k,BTNode *q);

//将关键字 k 和结点 q 分别插入到 p->key[i+1]和 p->ptr[i+1]中

SplitBTNode(BTNode *&p,BTNode *&q);

//将结点 p 分裂成两个结点,前一半保留,后一半移入结点 q

NewRoot(BTNode *&t,KeyType k,BTNode *p,BTNode *q);

//生成新的根结点 t,原 p 和 q 为子树指针

InsertBTree(BTree &t,int i,KeyType k,BTNode *p);

//在 B 树 t 中插入关键字 k

Remove(BTNode *p,int i);

//p 结点删除 key[i]和它的孩子指针 ptr[i]

Substitution(BTNode *p,int i);

//查找替代值

MoveRight(BTNode *p,int i);

//结点调整右移操作

MoveLeft(BTNode *p,int i);

//结点调整左移操作

Combine(BTNode *p,int i);

//结点调整合并操作

AdjustBTree(BTNode *p,int i);

//B 树调整操作

BTNodeDelete(BTNode *p,KeyType k);

//在结点 p 中删除关键字 k

BTreeDelete(BTree &t,KeyType k);

//在 B 树 t 中删除关键字 k

DestroyBTree(BTree &t);

//递归释放 B 树

PrintBTree(BTree t);

//遍历打印 B 树

}

3. 关键的基本操作的具体实现概述：

B 树的查找操作

查找操作的理解不难。我直接用几张图来阐述一下。

在 B 树上进行查找和二叉树的查找很相似，二叉树是每个节点上有一个关键子和两个分支。B 树上每个节点有 K 个关键字和 K+1 个分支。二叉树的查找只考虑向左还是向右走，而 B 树中需要由多个分支决定。

B 树的查找分两步，首先查找节点，由于 B 树通常是在磁盘上存储的所以这步需要进行磁盘 IO 操作。第二步是查找关键字，当找到某个节点后将该节点读入内存中然后通过顺序或者折半查找来查找关键字。若没有找到关键字，则需要判断大小来找到合适的分支继续查找。

B 树的插入操作

对高度为 k 的 m 阶 B 树，新结点一般是插在叶子层。通过检索可以确定关键码应插入的结点位置。然后分两种情况讨论：

1、 若该结点中关键码个数小于 m-1，则直接插入即可。

2、 若该结点中关键码个数等于 m-1，则将引起结点的分裂。以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点(k -1 层)中

重复上述工作，最坏情况一直分裂到根结点，建立一个新的根结点，整个 B 树增加一层。[2]

4. 应用：

题目：辅存上的栈[3]

考虑在一个有着相对少量的快速主存但有着相对大量较慢的磁盘存储空间的计算机上实现一个栈的问题。操作 PUSH 和 POP 操作的对象为单字。我们希望计算机支持的栈可以增长得很大，以至于无法全部装人主存中，因此它的大部分都要放在磁盘上。

一种简单但低效的栈实现方法是将整个栈存放在磁盘上。在主存中保持一个栈的指针，它指向栈顶元素的磁盘地址。如果该指针的值为 p，则栈顶元素是磁盘的[p/m]页上的第(p modm)个字,这里 m 为每页所含的字数。

为了实现 PUSH 操作，我们增加栈指针，从磁盘将适当的页读到主存中后，复制要被压入栈的元素到该页上适当字的位置，最后将该页写回到磁盘。POP 操作与之类似。我们减小栈指针，从磁盘上读入所需的页，再返回栈顶元素。我们不需要写回该页，因为它没有被修改。

因为磁盘操作代价相对较高，我们统计任何实现的两部分代价:总的磁盘存取次数和总的 CPU 时间。任何对一个包含 m 个字的页面的磁盘存取，都会引起一次磁盘存取和(m)的 CPU 时间。

a.从渐近意义上看，使用这种简单实现，在最坏的情况下，n 个栈操作需要多少次磁盘存取?CPU 时间又是多少?(用 m 和 n 来表示这个问题及后面几个问题的答案。)

现在考虑栈的另一种实现，即在主存中始终保持存放栈中的一页。(还用少量的主存来记录当前哪一页在主存中。)只有相关的磁盘页驻留在主存中，才能执行栈操作。如果需要，可以将当前主存中的页写回磁盘，并且可以从磁盘向主存读入新的一页。如果相关的磁盘页已经在主存，那么就无需任何磁盘存取。

b.最坏情况下，n 个 PUSH 操作所需的磁盘存取次数是多少?所需的 CPU 时间是多少?

c.最坏情况下，n 个栈操作所需的磁盘存取次数是多少?所需的 CPU 时间是多少?假设现在是在主存中保持栈的 2 页(此外还有少量的字来记录哪些页在主存中)的实现。

d.请描述如何管理栈页，使得任何栈操作的摊还磁盘存取次数为 O(1/m)，摊还 CPU 时间为 O(1)。

解答

a) 最坏情况下，假设是 n 次 PUSH 操作，共需 2n 次磁盘存取，Θ(2m*n)的 CPU 时间。

b) 每页有 m 个字，n 次 PUSH 磁盘存取次数为 n/m，CPU 时间为 Θ(n)。

c) 假设当前内存中页已满，第一次栈操作为 PUSH，则需要换页，需要 2 次磁盘存取；接着进行两次 POP，需要再次换页，需进行 1 次磁盘存取。接着进行 2 次 PUSH，两次 POP，……。

存取次数大致为 n*(3/2)，CPU 时间为 Θ(mn(3/2)).

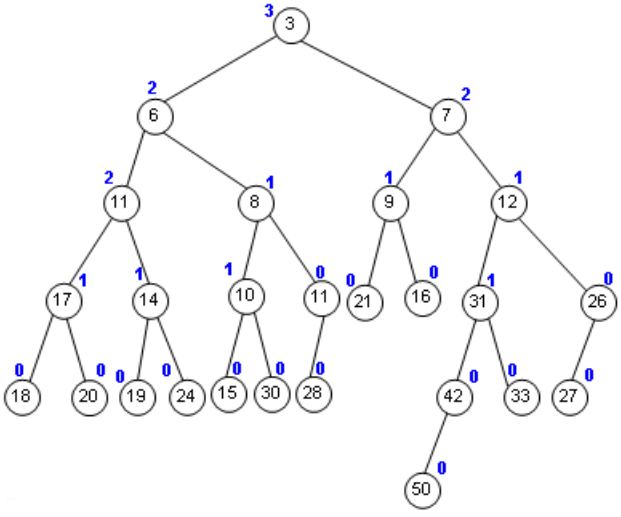
d) 使用磁盘中相邻两页放在内存中实现栈，这样当一页满或空时可以使用另外一页。最坏情况下连续 m 次 PUSH 或 POP 才需换一页。平摊磁盘存取为 O(1/m)，平摊 CPU 时间为 O(1)。

二、Heaps

左偏树

1. 概述：

左偏树（英语：leftist tree 或 leftist heap），也可称为左偏堆、左倾堆，是计算机科学中的一种树，是一种优先队列实现方式，属于可并堆，在信息学中十分常见，在统计问题、最值问题、模拟问题和贪心问题等等类型的题目中，左偏树都有着广泛的应用。斜堆是比左偏树更为一般的数据结构。[4]



左偏树（Leftist Tree）

2. 数据结构：

```
struct node {//左偏树节点
    int key, dis;
    node *l, *r;
    bool operator < (const node &b) const {
```

```
        return key > b.key;
    }
};
node *tree[maxn];

void init (node *&t, int key);
int dis (node *a);//计算左偏树的 dis 值
node *merge (node *a, node *b);//合并两棵左偏树
node *pop (node *a);//删除左偏树 key 值最小的节点
node *insert (node *root, int num);//左偏树中插入 key 值为 num 的节点
int top (node *a);//去取左偏树 key 值最大的元素
```

3. 关键的基本操作的具体实现概述：

合并 (merge)

合并操作是左偏树最重要的操作，必须深刻理解。

假设我们已经得到了两个左偏树，我们考虑如果将其合并。我们要使整棵树的时间复杂度得到保证，也就是说要让树的层数尽量小。由于我们维护了每一个右儿子 disdis 值尽量的小，所以我们可以将一个合并问题 merge(x,y)merge(x,y)转化为合并一棵树的右儿子和另一棵树，即 merge(r(x),y)merge(r(x),y)。当然，为了维护堆性质，我们还要保证 val(x)<val(y)val(x)<val(y)。

完成合并后，由于根节点的右儿子可能发生了变化，所以我们要对右儿子的父亲重新进行更新。

为了维护左偏性质 11 和 22，以便之后的合并操作，我们可能还需要再对左右子树进行交换，并顺带更新 disdis 值，即令 dis(x)=dis(r(x))+1dis(x)=dis(r(x))+1。

由之前的证明可知，节点数为 nn 的左偏树，其最大 disdis 值至多为 log2(n+1)−1log2(n+1)−1，那么我们的合并操作的最大时间复杂度即为

O(maxi∈tree(x)disi+maxj∈tree(y)disj)\=O(2log2(n+1)−2)=O(log2n)O(maxi∈tree(x)disi+maxj∈tree(y)disj)\=O(2log2(n+1)−2)=O(log2n)，符合我们的需要。

```
inline int merge(int x,int y)//返回值的含义为合并后新树的根节点编号
{
    if(!x||!y)return x|y;//如果 x，y 有一棵是空树，返回另一棵的编号
    if(val(x)>val(y)||(val(x)==val(y)&&x>y))//维护堆性质，把权值大的合并到权值小的上
        swap(x,y);
    r(x)=merge(r(x),y);//递归合并 x 的右子树与 y
    f(r(x))=x;//更新右子树父亲
    if(dis(l(x))<dis(r(x)))//维护左偏性质 1
        swap(l(x),r(x));
    dis(x)=dis(r(x))+1;//维护左偏性质 2
        return x;
}
```

取出堆顶 (find)

除了合并外，左偏树当然要实现它的本职工作，查询最值。

由于我们已经维护好了左偏树，只要直接输出树根的权值即为最小值。

Code:

```
inline int find(int x)
{
    return f(x)==x?x:f(x)=find(f(x));
    //由于树的最大深度未知，直接查询可能会导致时间复杂度退化为 O(n)
    //所以要用并查集的路径压缩写法
}
int Min=val(find(p));//查询节点 p 所在左偏树的最小值，p 已经被删除则返回-1。[5]
```

4. 应用：

题目：猴王 Monkey King[6]

题目描述

很久很久以前，在一个广阔的森林里，住着 n 只好斗的猴子。起初，它们各干各的，互相之间也不了解。但是这并不能避免猴子们之间的争吵，当然，这只存在于两个陌生猴子之间。当两只猴子争论时，它们都会请自己最强壮的朋友来代表自己进行决斗。显然，决斗之后，这两只猴子以及它们的朋友就互相了解了，这些猴子之间将再也不会发生争论了，即使它们曾经发生过冲突。假设每一只猴子都有一个强壮值，每次决斗后都会减少一半(比如 10 会变成 5，5 会变成 2.5)。并且我们假设每只猴子都很了解自己。就是说，当它属于所有朋友中最强壮的一个时，它自己会站出来，走向决斗场。

输入

输入分为两部分。

第一部分，第一行有一个整数 n(n<=100000)，代表猴子总数。

接下来的 n 行，每行一个数表示每只猴子的强壮值(小于等于 32768)。

第二部分，第一行有一个整数 m(m<=100000)，表示有 m 次冲突会发生。

接下来的 m 行，每行包含两个数 x 和 y，代表第 x 个猴子和第 y 个猴子之间发生冲突。

输出

输出每次决斗后在它们所有朋友中的最大强壮值。数据保证所有猴子决斗前彼此不认识。

样例输入

```
5
20
16
10
10
4
4
2 3
```

3 4

3 5

1 5

样例输出

8

5

5

10

解析

首先，看到题目我们首先会想到，并查集和堆，因为我们每一次都要从一群猴子中找出最强壮的，暴力搜一遍显然不行，因此要用到堆，然后两只猴子打架后需要将两群猴子合并，因此要用到并查集，但是同时写这两种数据结构太麻烦，所以要借助一种既具有查找功能又具有合并功能的数据结构－左偏树。

```
#include<stdio.h>
#include<string.h>
#include<iostream>
#include<algorithm>
using namespace std;
int scan(){
    char c=getchar();
    int x=0;
    while(c>'9'||c<'0')
        c=getchar();
    while(c>='0'&&c<='9')
        x=x*10+c-'0',
        c=getchar();
    return x;
}           //读入优化
struct node{
```

```
    int dis;                    //结点距离
    int fa;                     //父亲结点
    int rchild,lchild;     //左右子结点
    int key;                    //强壮值
};                              //一只猴子的属性结构体
int n,m;
node monkey[100010];     //100000 只猴子
int find(int x){
    if(monkey[x].fa!=x)
        monkey[x].fa=find(monkey[x].fa);
    return monkey[x].fa;
}       //找到编号为 x 的猴子所属群最强壮的猴子(编号),并压缩路径
        //压缩路径可能写错了(欢迎指点)
int merge(int a,int b){       //a,b 均为猴子编号
    if(a==0) return b;
    if(b==0) return a;                  //达到最底层
    if(monkey[a].key<monkey[b].key) swap(a,b);//维护大顶树的性质，强行让猴子 a 强壮
    monkey[a].rchild=merge(monkey[a].rchild,b);//递归合并操作
    //似乎没有更新父节点
    if(monkey[monkey[a].rchild].dis>monkey[monkey[a].lchild].dis)
        swap(monkey[a].rchild,monkey[a].lchild);
       //不满足左偏树性质，维护，交换指针
    monkey[a].dis=monkey[monkey[a].rchild].dis+1;   //更新距离
    return a;       //返回最强壮猴子编号
}               //核心操作，合并两群猴子
int main(){
    int i,j,k,x,y;
    n=scan();
    for(i=1;i<=n;i++)
```

```
        monkey[i].key=scan(),
        monkey[i].fa=i;       //猴子数据读入
    m=scan();
    for(i=1;i<=m;i++){
        x=scan();
        y=scan();         //x,y 号猴子要决斗
        j=find(x);k=find(y);   //寻找各自最强壮的猴子朋友 j,k
        monkey[j].key/=2.0;
        monkey[k].key/=2.0;     //战斗力减半

         int fa1=merge(monkey[j].lchild,monkey[j].rchild);
         int fa2=merge(monkey[k].lchild,monkey[k].rchild);
         //删除操作，先把决斗的猴子各自从自己的猴群中删除(最强壮的猴子是根结点),之后各自合并自己的左右子猴群，例如 fa1 是猴子 j 的猴群再次合并后的根结点， fa2 则是 k 的。(合并操作将返回最强壮的猴子编号)
        monkey[j].lchild=monkey[j].rchild=monkey[k].lchild=monkey[k].rchild=0;
//猴子 j 和 k 逐出猴群，子结点清空为 0
        int t=merge(fa1,fa2);
        //合并参战猴群，无 j 和 k 猴子
        int p=merge(j,k);
        //合并 j 和 k 猴子
        int final_fa=merge(t,p);
        //重新纳入 j 和 k 猴子，final_fa 为最终最强壮的猴子，即两棵左偏树在经过一番操作后形成的新左偏树的根结点。
        monkey[fa1].fa=monkey[fa2].fa=monkey[t].fa=final_fa;
        monkey[j].fa=monkey[k].fa=monkey[p].fa=monkey[final_fa].fa=final_fa;
//更新可能是猴王的猴子的父结点
        printf("%d\n",monkey[final_fa].key);//输出猴王的强壮值
        //下一次战斗 循环
    }
```

}

三、Tries

字典树

1. 概述：

又称单词查找树，Trie 树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。[7]

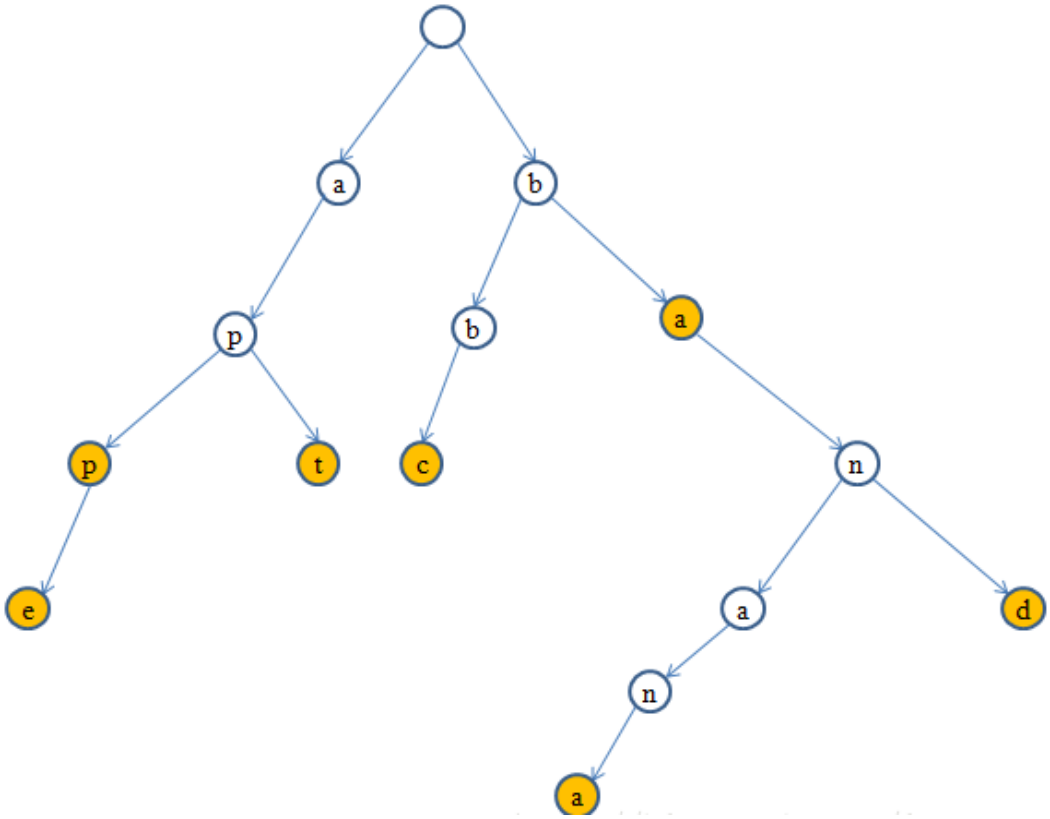
2. 数据结构：

```
// trie ADT
typedef struct trie trie_t;
struct trie
{
    trie_node_t *root;
    int count;
};
// Returns new trie node (initialized to NULLs)
trie_node_t *getNode(void);
Initializes trie (root is dummy node)
void initialize(trie_t *pTrie);
// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(trie_t *pTrie, char key[]);
// Returns non zero, if key presents in trie
int search(trie_t *pTrie, char key[]);
```

3. 关键的基本操作的具体实现概述：

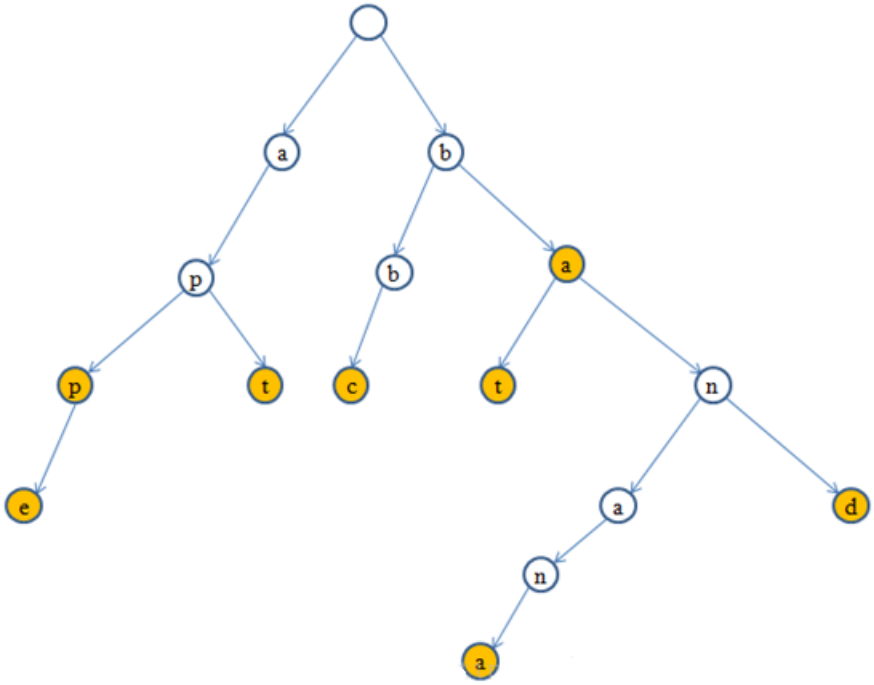
构建 TrieTree

给定多个字符串，如 {banana,band,apple,apt,bbc,app,ba}，那么所构建的一棵 TrieTree 形状如下：



其中，黄色的节点代表从根节点通往该节点的路径上所经过的节点的字符构成的一个字符串出现在原来的输入文本中，如以 d 为例，路径上的字符为：b-a-n-d，对应输入的字符串集合中的”band”。TrieTree 可以很方便的扩展，当来了新的字符串时，只要把新的字符串按照原本的规则插入到原来的树中，便可以得到新的树。如需要加入新的单词”bat”，那么树的结构只需简单的拓展成如下

的形式：



可以看出，TrieTree 充分利用字符串与字符串间拥有公共前缀的特性，而这种特性在字符串的检索与词频统计中会发挥重要的作用。

利用 TrieTree 进行字符串检索

利用上一节中构造的 TrieTree，我们可以很方便的检索一个单词是否出现在原来的字符串集合中。例如，我们检索单词”banana”，那么我们从根节点开始，逐层深入，由路径 b-a-n-a-n-a 最终到达节点 a，可以看出此时的节点 a 是黄色的，意味着“从根节点到该节点的路径形成的字符串出现在原来的字符串集合中”，因此单词”banana”的检索是成功的。又如，检索单词”application”，从根节点沿路径 a-p-p，到达节点 p 后，由于节点 p 的后代并没有’l’，这也意味着检索失败。再举一个例子，检索单词”ban”，沿着路径 b-a-n 到达节点 n，然而，当前的节点 n 并不是黄色的，说明了“从根节点到该节点的路径形成的字符串“ban”没有出现在原来的字符串集合中，但该字符串是原字符串集合中某个(些)单词的前缀”。

可以看出，利用 TrieTree 进行文本串的单词统计十分方便，当我们要检索一个单词的词频时，不用再去遍历原来的文本串，从而实现高效的检索。这在搜索引擎中统计高频的词汇是十分有效的。[8]

4. 应用：

秘密信息 SecretMessage[9]

【题目描述】

贝茜正在领导奶牛们逃跑。为了联络，奶牛们互相发送秘密信息。

信息是二进制的，共有 M(1≤M≤50000)条。反间谍能力很强的约翰

已经部分拦截了这些信息，知道了第 i 条二进制信息的前 bi(1≤bi≤10000)位。他同时知道，奶牛使用 N(1≤N≤50000)条密码。但是，他仅仅了解第 j 条密码的前 cj(1≤cj≤10000)位。

对于每条密码 j，他想知道有多少截得的信息能够和它匹配。也就是说，有多少信息和这条密码有着相同的前缀。当然，这个前缀长度必须等于密码和那条信息长度的较小者。

在输入文件中，位的总数（即∑Bi+∑Ci）不会超过 500000.

【输入】SecretMessage.in

第 1 行输入 N 和 M，之后 N 行描述秘密信息，之后 M 行描述密码。每行先输入一个整数表示信息或密码的长度，之后输入这个信息或密码。所有数字之间都用空格隔开。

【输出】SecretMessage.out

共 M 行，输出每条密码的匹配信息数。

【样例输入】

4 5

3 0 1 0

1 1

3 1 0 0

3 1 1 0

1 0

1 1

2 0 1

5 0 1 0 0 1

2 1 1

【样例输出】

1

3

1

1

2

思路叙述

这道题在字典树上稍加修改即可

1.先记录下这个字符串走过的全部路径 book[ ]，再记录下它的尾节点即最后一个字符 en[ ]即可，多个字符串通过同一节点时，这个节点要增加记录.

2.在查找时，当密码还未匹配完时且前面的都匹配. 有秘密信息匹配完了时 ans 加上以当前节点为末尾的字符串数即 en[ ]，若密码不匹配且未匹配完了，那么就直接返回 ans.若密码匹配完了，则令 ans 加上 book[ ]在减去 en[ ]. ~~end 在前已记录~~.

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=500010;
int n,m,k,a[10010];
struct ac{
        int trie[maxn][2],tot,book[maxn],en[maxn];
        void ins(int a[],int len){
                int p=0;
                for(int i=0;i<len;++i){
                        int ch=a[i];
                        if(!trie[p][ch]) trie[p][ch]=++tot;
                        p=trie[p][ch];
                        book[p]++;
                }
                en[p]++;
        }
        int found(int a[],int len){
```

```
                int p=0,add=0,ch;
                for(int i=0;i<len;++i){
                        ch=a[i];
                        if(trie[p][ch]){
                                p=trie[p][ch];
                                add+=en[p];
                        }
                        else{
                                return add;
                        }
                }
                add+=book[p]-en[p];
                return add;
        }
}ac;
int main(){
        //freopen("SecretMessage.in","r",stdin);
        //freopen("SecretMessage.out","w",stdout);
        scanf("%d %d",&n,&m);
        for(int i=1;i<=n;++i){
                scanf("%d",&k);
                for(int j=0;j<k;++j){
                        scanf("%1d",&a[j]);
                }
                ac.ins(a,k);
        }
        for(int i=1;i<=m;++i){
                scanf("%d",&k);
                for(int j=0;j<k;++j){
                        scanf("%1d",&a[j]);
```

```
                }
                int ans=ac.found(a,k);
                printf("%d\n",ans);
        }
        return 0;
}
```

四、Spatial data partitioning trees

R 树

1. 概述：

R 树是用来做空间数据存储的树状数据结构。例如给地理位置，矩形和多边形这类多维数据建立索引。R 树是由 Antonin Guttman 于 1984 年提出的。［1］　人们随后发现它在理论和应用方面都非常实用。 在现实生活中，R 树可以用来存储地图上的空间信息，例如餐馆地址，或者地图上用来构造街道，建筑，湖泊边缘和海岸线的多边形。然后可以用它来回答“查找距离我 2 千米以内的博物馆”，“检索距离我 2 千米以内的所有路段”（然后显示在导航系统中）或者“查找（直线距离）最近的加油站”这类问题。R 树还可以用来加速使用包括大圆距离在内的各种距离度量方式的最邻近搜索。[10]

2. 数据结构：

R 树是 B 树在高维空间的扩展，是一棵平衡树。每个 R 树的叶子结点包含了多个指向不同数据的指针，这些数据可以是存放在硬盘中的，也可以是存在内存中。

3. 关键的基本操作的具体实现概述：

搜索

R 树的搜索操作很简单，跟 B 树上的搜索十分相似。它返回的结果是所有符合查找信息的记录条目。而输入是不仅仅是一个范围了，它更可以看成是一个空间中的矩形。也就是说，输入的是一个搜索矩形。

先给出伪代码：

Function：Search

描述：假设 T 为一棵 R 树的根结点，查找所有搜索矩形 S 覆盖的记录条目。

S1:[查找子树]如果 T 是非叶子结点，如果 T 所对应的矩形与 S 有重合，那么检查所有 T 中存储的条目，对于所有这些条目，使用 Search 操作作用在每一个条目所指向的子树的根结点上（即 T 结点的孩子结点）。

S2:[查找叶子结点]如果 T 是叶子结点，如果 T 所对应的矩形与 S 有重合，那么直接检查 S 所指向的所有记录条目。返回符合条件的记录。

插入

R 树的插入操作也同 B 树的插入操作类似。当新的数据记录需要被添加入叶子结点时，若叶子结点溢出，那么我们需要对叶子结点进行分裂操作。显然，叶子结点的插入操作会比搜索操作要复杂。插入操作需要一些辅助方法才能够完成。

来看一下伪代码：

Function：Insert

描述：将新的记录条目 E 插入给定的 R 树中。

I1：[为新记录找到合适插入的叶子结点]开始 ChooseLeaf 方法选择叶子结点 L 以放置记录 E。

I2：[添加新记录至叶子结点]如果 L 有足够的空间来放置新的记录条目，则向 L 中添加 E。如果没有足够的空间，则进行 SplitNode 方法以获得两个结点 L 与 LL，这两个结点包含了所有原来叶子结点 L 中的条目与新条目 E。

I3：[将变换向上传递]开始对结点 L 进行 AdjustTree 操作，如果进行了分裂操作，那么同时需要对 LL 进行 AdjustTree 操作。

I4：[对树进行增高操作]如果结点分裂，且该分裂向上传播导致了根结点的分裂，那么需要创建一个新的根结点，并且让它的两个孩子结点分别为原来那个根结点分裂后的两个结点。

Function：ChooseLeaf

描述：选择叶子结点以放置新条目 E。

CL1：[Initialize]设置 N 为根结点。

CL2：[叶子结点的检查]如果 N 为叶子结点，则直接返回 N。

CL3：[选择子树]如果 N 不是叶子结点，则遍历 N 中的结点，找出添加 E.I 时扩张最小的结点，并把该结点定义为 F。如果有多个这样的结点，那么选择面积最小的结点。

CL4：[下降至叶子结点]将 N 设为 F，从 CL2 开始重复操作。

Function：AdjustTree

描述：叶子结点的改变向上传递至根结点以改变各个矩阵。在传递变换的过程中可能会产生结点的分裂。

AT1：[初始化]将 N 设为 L。

AT2：[检验是否完成]如果 N 为根结点，则停止操作。

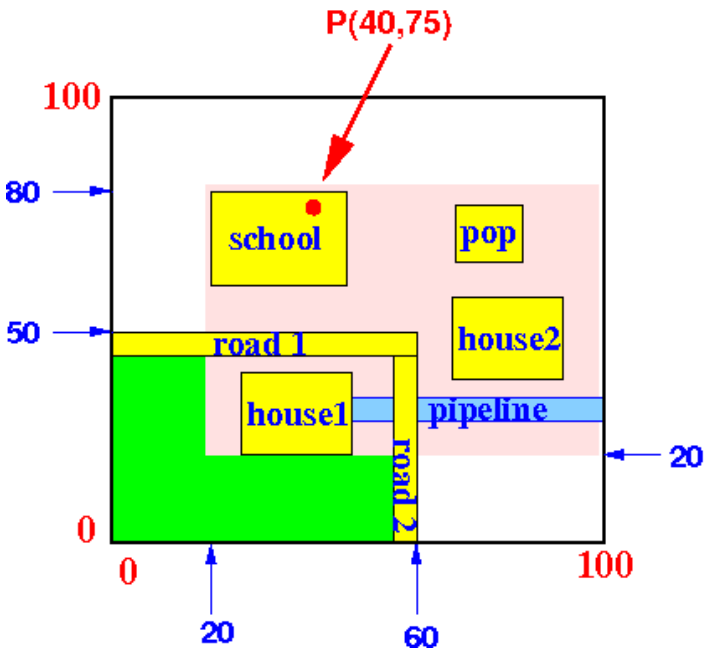
AT3：[调整父结点条目的最小边界矩形]设 P 为 N 的父节点，EN 为指向在父节点 P 中指向 N 的条目。调整 EN.I 以保证所有在 N 中的矩形都被恰好包围。

AT4：[向上传递结点分裂]如果 N 有一个刚刚被分裂产生的结点 NN，则创建一个指向 NN 的条目 ENN。如果 P 有空间来存放 ENN，则将 ENN 添加到 P 中。如果没有，则对 P 进行 SplitNode 操作以得到 P 和 PP。

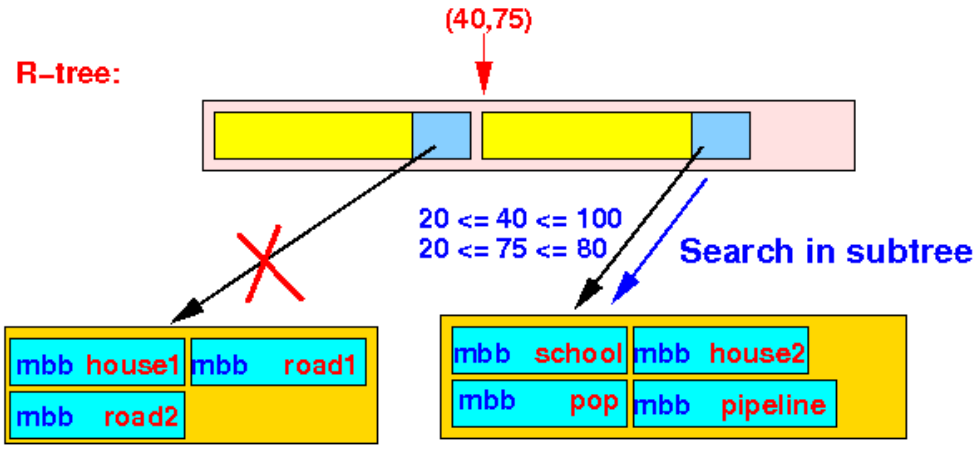
AT5：[升高至下一级]如果 N 等于 L 且发生了分裂，则把 NN 置为 PP。从 AT2 开始重复操作。[11]

4. 应用：

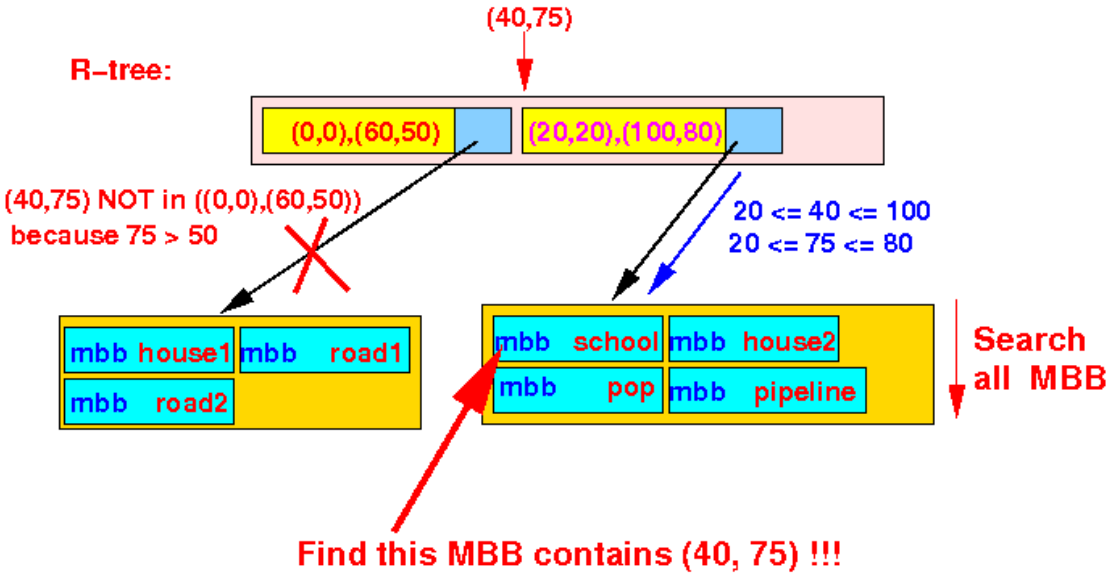
下面来看一个具体的例子，现在的任务是要在前面已经建立好的 R 树中搜索点 P(40,75)。[12]



从根结点开始，因为它不是叶子结点，所以逐个搜索其中的条目，点 P(40,75)并不位于((0,0), (60,50))这个 BB 中，但它位于((20,20), (100,80))这个 BB 中，所以递归地搜索该子树。如下图所示：



此时，我们已经达到了一个叶子结点，因此逐个搜索其中的对象，发现P(40,75)位于对象 school 的 MBB 中，因此最后应该返回该对象作为结果。



参考资料

1. 许卓群. 数据结构与算法[M]. 高等教育出版社, 2004.
2. 彭军、向毅. 数据结构与算法[M]. 人民邮电出版社, 2013.
3. Budd T. 经典数据结构（Java 语言版）[M]. 清华大学出版社, 2005.
4. 王晓东. 算法设计与分析[M]. 清华大学出版社, 2003. 5.
5. 严蔚敏. 数据结构 C 语言版[M]. 清华大学出版社, 2007.

6. 王广芳.数据结构算法与应用-C++语言描述[M].机械工业出版社,2006.
7. 严蔚敏等.数据结构题集(C 语言版)[M].清华大学出版社,1999.2.
8. 严太山，郭观七，李文彬.基于字典树的词频统计[J].湖南理工学院学报：自然科学版，2015（1）：81-83.
9. Frank M.Carrano.数据结构与 C++高级教程[M].清华大学出版社,2004.
10. 南淑萍.关于数据结构中的 R 树[J].湖北科技学院学报，2014（10）：13-14.
11. 贾丹，周军.基于 C 语言的 R 树基本操作实现 [J].辽宁工业大学学报：社会科学版，2015（2）：132-134
12. 高贤强，化希耀，陈立平.引入计算思维的《数据结构》教学改革研究[J].现代计算机：专业版，2015（7）：16-19.