21级实验室暑假第二讲

目录

- 字符串hash
- KMP算法
- Manacher算法（马拉车算法）
- 字典树（Trie树）

字符串hash

哈希：哈希就是将所要处理的数据转化成数字，且这个数字能**唯一地**去对应上这个数据。若这个数字对应上了多个数字，则称作哈希冲突。比如

$$k_1 \neq k_2, 但 hash(k_1) = hash(k_2)$$

字符串哈希：字符串哈希是指将一个任意长度的字符串映射成一个**非负整数**,并且冲突的概率几乎为0.

字符串hash

字符串哈希的思想

- ✓ 取一固定值P,把字符串看成**P进制数**,并分配一个大于0的数值,代表每种字符。一般来说，分配的数值都远小于P。例如，对于小写字母构成的字符串，可以令a=1,b=2,...,z=26。取一固定值M,求出该P进制数对M的**余数**,作为该字符串的Hash值。

令a=1,b=2,...,z=26

$$hash("abcb") = (1 \times P^3 + 2 \times P^2 + 3 \times P^1 + 2 \times P^0)\%M$$

P和M的取法

- 一般来说，取P=131或P=13331(**取素数**)，此时Hash值产生冲突的概率极低，只要Hash值相同,就可以认为与原字符串相等的。
- 通常取M=2^{64}，即直接使用unsigned long long类型存储这个Hash值，在计算时不处理算法溢出问题，产生溢出时相当于自动对2^{64}取模，这样可以避免低效的取模（mod）运算。
- 除了在极特殊构造的数据上，上述Hash算法很难产生冲突。还可以多取一些恰当的P和M的值(如大质数)，多进行几组Hash运算，当结果都相同时，才认为原字符串相等，就更加难易构造出使这个Hash产生错误的数据。

字符串hash

常用的几个字符串hash法

- 1. unsigned long long hash[N];　　hash[i]=hash[i-1]*P+idx(s[i])（自动取模）
- 2. hash[i]=(hash[i-1]*P+idx(s[i]))%M　（P取较大素数，M取大素数）
- 3.双hash

  hash1[i]=(hash1[i-1]*P+idx(s[i]))%M1

  hash2[i]=(hash2[i-1]*P+idx(s[i]))%M2

  pair<hash1,hash2>表示一个字符串

  M1一般取1e9+7，M2一般取1e9+9（原因：1000000007和1000000009是一对孪生素数，取它们，冲突的概率极低）

从速度上来看，应该是： 自然溢出 > 单Hash > 双Hash（也就是自然溢出时间更小）。从安全性上来看，应该：双Hash方法 > 单Hash方法。因为双Hash方法相当于是用两次 单 Hash的结果来比较，这样子冲突的概率会变得更低。

字符串hash

字符串哈希的基本运算

已知某一字符串的哈希值,在其字符串的尾巴继续添置一个字母,求新组成的字符串的哈希值.

令a=1,b=2,...,z=26

$$hash("abcb") = (1 \times P^3 + 2 \times P^2 + 3 \times P^1 + 2 \times P^0)\%M$$

$$\begin{aligned} & hash("abcb"+'d') \\ &= hash("abcbd") \\ &= (1 \times P^4 + 2 \times P^3 + 3 \times P^2 + 2 \times P^1 + 4 \times P^0)\%M \\ &= (hash("abcb") \times P + d)\%M \end{aligned}$$

字符串hash

字符串哈希的基本运算

进而,已知字符串S和字符串T的哈希值,求S+T的哈希值,其中len(T)表示字符串T的长度

$$Hash(S+T)=(Hash(S)\times P^{len(T)}+Hash(T))\%M$$

已知字符串S和字符串S+T的哈希值,求T的哈希值

$$Hash(T)=(Hash(S+T)-Hash(S)\times P^{len(T)})\%M$$

此处取余运算这样写更好（减法取模运算保证Hash为正）

$$Hash(T)=((Hash(S+T)-Hash(S)\times P^{len(T)})\%M+M)\%M$$

字符串hash

字符串哈希的基本运算

```
const int MAXN=100010,P=131,M=1e9+7;
typedef unsigned long long ull;
typedef long long ll;
char s[MAXN];//已知字符串s从1开始存储
ull Hash1[MAXN],p1[MAXN];//此处用自动取模
ll Hash2[MAXN],p2[MAXN];//此处采用单Hash方法
//Hash1和Hash2共同构成双Hash方法

p1[0]=p2[0]=1;
int len=strlen(s+1);
for(int i=1;i<=len;i++)
{//此处为了方便,字符串的字符分配值就是该字符的ascii码
        p1[i]=p1[i-1]*P;//求P的次方
        Hash1[i]=Hash1[i-1]*P+s[i];//利用类似前缀和的思想求Hash
        p2[i]=((p2[i-1]%M)*P)%M;//(nm)%mod=(n%mod * m%mod) %mod;
        Hash2[i]=(((Hash2[i-1]%M)*P)%M+s[i])%M;
}
```

字符串hash

求任意子串的Hash值

$$Hash(T) = ((Hash(S+T) - Hash(S) \times P^{len(T)})\%M + M)\%M$$

求字符串区间[l,r]的Hash值
例如abcdaa求[3,5]区间的Hash值
1 2 3 4 5 6
a b c d a a ---> $1 \times P^5 + 2 \times P^4 + 3 \times P^3 + 4 \times P^2 + 1 \times P^1 + 1 \times P^0$

a b c d a ----> $1 \times P^4 + 2 \times P^3 + 3 \times P^2 + 4 \times P^1 + 1 \times P^0$

a b ---> $1 \times P^1 + 2 \times P^0$

c d a ---> $3 \times P^2 + 4 \times P^1 + 1 \times P^0 = Hash("abcda") - Hash("ab") \times P^3$

故字符串区间[l,r]的Hash值为 $Hash(r) - Hash(l-1) \times P^{r-l+1}$

字符串hash

求任意子串的Hash值 $Hash(r)-Hash(l-1)\times P^{r-l+1}$

```
ull get_hash1(int l,int r)
{
        return Hash1[r]-Hash1[l-1]*p1[r-l+1];
}

ll get_hash2(int l,int r)
{
        return ((Hash2[r]-Hash2[l-1]*p2[r-l+1])%M+M)%M;
}
```

KMP算法

字符串匹配中的几个概念

- 模式串：等待被在主串中匹配的串
- (文本串)主串：被模式串匹配的串
- 字符串匹配：在主串中查找模式串的过程
  - 比如对于主串: aabbcc 与模式串：aa 来说，一般所要进行的匹配是在主串中查找模式串第一次出现的位置，比如模式串：aa 在主串：aabbcc 中第一次出现的位置是1。

✓ **什么是KMP算法?**

- KMP算法是一种改进的字符串匹配算法，由D.E.Knuth与J.H.Morris和V.R.Pratt同时发现，因此人们称它为克努特——莫里斯——普拉特操作（简称KMP算法）。KMP算法的关键是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。具体实现就是实现一个next()函数，函数本身包含了模式串的局部匹配信息。

KMP算法

✓ KMP算法的作用

- KMP主要应用在字符串匹配上。
- KMP的主要思想是当出现字符串不匹配时，可以知道一部分之前已经匹配的文本内容，可以利用这些信息避免从头再去做匹配了。
- 所以如何记录已经匹配的文本内容，是KMP的重点，也是next数组肩负的重任。

✓ 什么是前缀表

- next数组就是一个前缀表（prefix table）

✓ 前缀表有什么作用呢?

前缀表是用来回退的，它**记录了模式串与主串(文本串)不匹配的时候，模式串应该从哪里开始重新匹配**。

KMP算法

✓ 前缀表

举例:

要在文本串：aabaabaafa 中查找是否出现过一个模式串：aabaaf。

- 可以看出，文本串中第六个字符b 和 模式串的第六个字符f，不匹配了。如果暴力匹配，会发现不匹配，此时就要**从头匹配**了。
- 但如果使用前缀表，就不会从头匹配，而是从**上次已经匹配的内容开始匹配**，找到了模式串中第三个字符b继续开始匹配。

KMP算法

✓ 前缀表是如何记录的

前缀表的任务是当前位置匹配失败，找到之前已经匹配上的位置，在重新匹配，此也意味着在某个字符失配时，前缀表会告诉你下一步匹配中，模式串应该跳到哪个位置。

那么什么是前缀表：记录下标i之前（包括i）的字符串中，有多大长度的相同前缀后缀。

前缀: 指不包含最后一个字符的所有以第一个字符开头的连续子串。

例如: 模式串： aabaaf 的前缀有: **a** **aa** **aab** **aaba** **aabaa**
后缀: 指不包含第一个字符的所有以最后一个字符结尾的连续子串。

模式串： aabaaf 的后缀有： **f** **af** **aaf** **baaf** **abaaf**

KMP算法

前缀表要求的就是最长的相同前后缀的长度

例如:
a的最长的相同前后缀为0（由于第一个字符和最后一个字符都是a,无前缀和后缀）
aa的最长的相同前后缀为1(aa的前缀有**a**;aa的后缀有**a**)
aaa的最长的相同前后缀为2(aaa的前缀有**a**,**a**a;aaa的后缀有**a**,a**a**)

KMP算法

为什么一定要用前缀表

回顾一下，刚刚匹配的过程在下标5的地方遇到不匹配，模式串是指向f，如图：

下标	0	1	2	3	4	5
模式串	a	a	b	a	a	f

KMP算法

然后就找到了下标2，指向b，继续匹配：如图：

下标	0	1	2	3	4	5
模式串	a	a	b	a	a	f

KMP算法

下标5之前这部分的字符串（也就是字符串**aa**b**aa**）的最长相等的前缀 和 后缀字符串是 子字符串aa ， 因为找到了最长相等的前缀和后缀，匹配失败的位置是后缀子串的后面，那么我们找到与其相同的前缀的后面从新匹配就可以了。
所以前缀表具有告诉我们当前位置匹配失败，跳到之前已经匹配过的地方的能力。

KMP算法

如何计算前缀表

下标	0	1	2	3	4	5
模式串	a	a	b	a	a	f

长度为前1个字符的子串a，最长相同前后缀的长度为0

（注意: **字符串的前缀是指不包含最后一个字符的所有以第一个字符开头的连续子串；后缀是指不包含第一个字符的所有以最后一个字符结尾的连续子串。**）

KMP算法

如何计算前缀表

下标	0	1	2	3	4	5
模式串	a	a	b	a	a	f

长度为前2个字符的子串aa，最长相同前后缀的长度为1

下标	0	1	2	3	4	5
模式串	a	a	b	a	a	f

长度为前3个字符的子串aab，最长相同前后缀的长度为0

KMP算法

如何计算前缀表

以此类推：

✓ 长度为前4个字符的子串aaba，最长相同前后缀的长度为1

✓ 长度为前5个字符的子串aabaa，最长相同前后缀的长度为2

注意: 此处前缀aab和后缀baa并不相同

✓ 长度为前6个字符的子串aabaaf，最长相同前后缀的长度为0

下标	0	1	2	3	4	5
模式串	a	a	b	a	a	f
前缀表	0	1	0	1	2	0

可以看出模式串与前缀表对应位置的数字表示的就是：下标i之前（包括i）的字符串中，有多大长度的相同前缀后缀。

KMP算法

再来看一下如何利用 前缀表找到 当字符不匹配的时候应该指针应该移动的位置。如动画所示：

KMP算法

前缀表和next数组

- 很多KMP算法的实现都是使用next数组来做回退操作，那么next数组与前缀表有什么关系呢?
- next数组既可以是**前缀表**，但是很多实现都是把**前缀表统一减一**（**右移一位，初始位置为-1**）之后作为next数组。
- 其实这并不涉及到KMP的原理，而是具体实现，next数组既可以是**前缀表**，也可以是**前缀表统一减一**（**右移一位，初始位置为-1**）

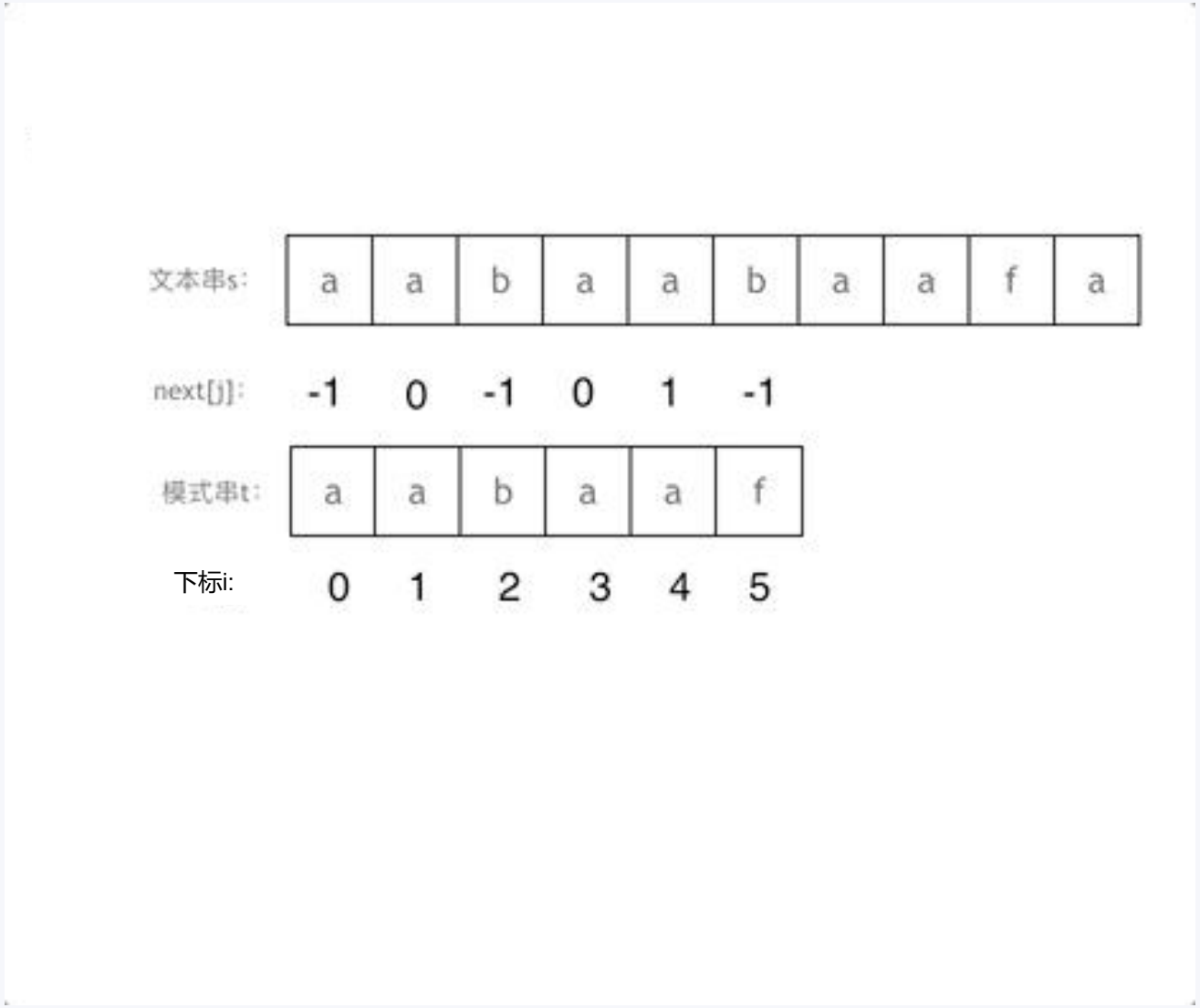
KMP算法

使用next数组来匹配

以下以前缀表统一减一之后的next数组来做演示。

- ✓ 有了next数组，就可以根据next数组来匹配文本串s，和模式串t了。
- ✓ 注意next数组是新前缀表（旧前缀表统一减一了）。

匹配过程动画如右图所示：



KMP算法

时间复杂度的分析

其中n为文本串长度，m为模式串长度，因为在匹配的过程中，根据前缀表不断调整匹配的位置，可以看出匹配的过程是O(n)，之前还要单独生成next数组，时间复杂度是O(m)。所以整个KMP算法的时间复杂度是O(n+m)的。

暴力的解法显而易见是O(n * m)，所以**KMP在字符串匹配中极大的提高的搜索的效率。**

KMP算法

构造next数组

定义一个函数getNext来构建next数组，函数参数一个字符串，其中next数组用全局变量表示。代码如下:

```
int nex[MAXN];

void get_next(string s)
```

构造next数组其实就是计算模式串s前缀表的过程。 主要有如下三步：

1. 初始化
2. 处理前后缀不相同的情况
3. 处理前后缀相同的情况

KMP算法

构造next数组

1.初始化：

定义两个指针i和j，j指向前缀末尾位置，i指向后缀末尾位置。然后还要对next数组进行初始化赋值，如下：

```
//初始化
int j=-1;
nex[0]=j;
```

j 为什么要初始化为 -1呢，因为之前说过 前缀表要统一减一的操作仅仅是其中的一种实现，我们这里选择j初始化为-1，next[i] 表示 i（包括i）之前最长相等的前后缀长度（其实就是j）所以初始化next[0] = j

KMP算法

构造next数组

2. 处理前后缀不相同的情况

- ✓ 因为j初始化为-1，那么i就从1开始(模式串的第一个字符已作处理)，进行s[i]与 s[j+1]的比较。所以遍历模式串s的循环下标i 要从 1 开始，代码如下：

```
for(int i=1;i<s.size();i++)
```

KMP算法

构造next数组

2. 处理前后缀不相同的情况

✓ 如果 s[i] 与 s[j+1]不相同，也就是遇到 前后缀末尾不相同的情况，就要向前回退。
✓ next[j]就是记录着j（包括j）之前的子串的相同前后缀的长度。

✓ 那么 s[i] 与 s[j+1] 不相同，就要**找 j+1前一个元素在next数组里的值（就是next[j]）**。
✓ 所以，处理前后缀不相同的情况代码如下：

此处一定是while,不是if

```
while(j>=0&&s[i]!=s[j+1])
    j=nex[j];//向前回退
```

KMP算法

构造next数组

3. 处理前后缀相同的情况

- ✓ 如果s[i] 与 s[j + 1] 相同，那么就同时向后移动i 和j 说明找到了相同的前后缀，同时还要将j（前缀的长度）赋给next[i], 因为next[i]要记录相同前后缀的长度。

- ✓ 代码如下:

```
if(s[i]==s[j+1])
    j++;
nex[i]=j;//将j（前缀的长度）赋给nex[i]
```

KMP算法

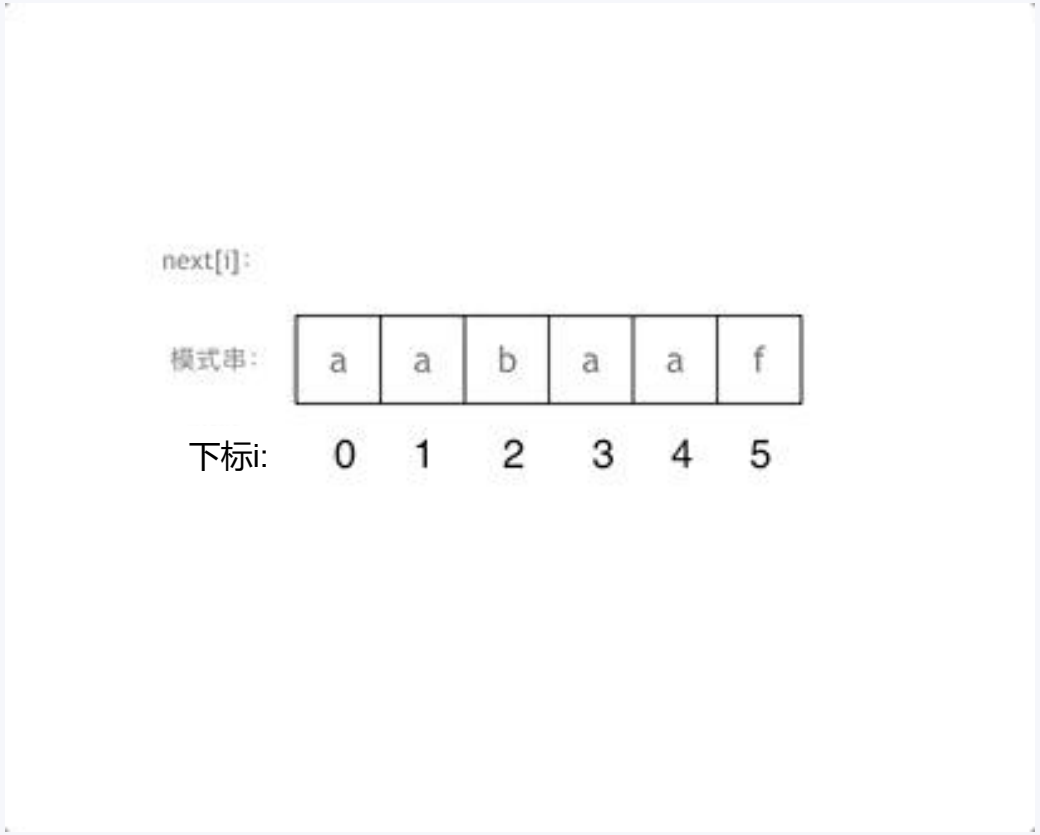
构造next数组

整体构建next数组的函数代码如下：

```
int nex[MAXN];

void get_next(string s){
//初始化
    int j=-1;
    nex[0]=j;
    for(int i=1;i<s.size();i++){
        while(j>=0&&s[i]!=s[j+1])
            j=nex[j];//向前回退
        if(s[i]==s[j+1])
            j++;
        nex[i]=j;//将j（前缀的长度）赋给nex[i]
    }
}
```

代码构造next数组的逻辑流程动画如下：



KMP算法

使用next数组来做匹配

- ✓ 在文本串s里 找是否出现过模式串t。
- ✓ 定义两个下标，j 指向模式串起始位置，i指向文本串起始位置。那么j初始值依然为-1(因为next数组里记录的起始位置为-1)。
- ✓ i就从0开始，遍历文本串，代码如下：

```
int j=-1;//因为nex数组里记录的起始位置为-1
for(int i=0;i<s.size();i++)//注意i从0开始
```

KMP算法

使用next数组来做匹配

- ✓接下来就是 s[i] 与 t[j + 1] （因为j从-1开始的） 进行比较。
- ✓如果 s[i] 与 t[j + 1] 不相同，j就要从next数组里寻找下一个匹配的位置。

代码如下:

```
while(j>=0&&s[i]!=t[j+1])//不匹配
    j=nex[j];//j寻找之前匹配的位置
```

KMP算法

使用next数组来做匹配

- ✓如果 s[i] 与 t[j + 1] 相同，那么i 和 j 同时向后移动， 代码如下：

```
if(s[i]==t[j+1])//匹配,j和i同时向后移动
    j++;//i的增加在for循环中
```

- ✓如何判断在文本串s里出现了模式串t呢，如果j指向了模式串t的末尾，那么就说明模式串t完全匹配文本串s里的某个子串了。代码如下：

```
if(j==(t.size()-1))//所有字符完全匹配
    return (i-t.size()+1);
```

KMP算法

完整KMP代码链接(前缀表统一减一)

使用next数组来做匹配

用模式串匹配文本串的整体代码如下：

```
int KMP(string s,string t)
{
    int j=-1;//因为nex数组里记录的起始位置为-1
    for(int i=0;i<s.size();i++){//注意i从0开始
        while(j>=0&&s[i]!=t[j+1])//不匹配
            j=nex[j];//j寻找之前匹配的位置
        if(s[i]==t[j+1])//匹配,j和i同时向后移动
            j++;//i的增加在for循环中
        if(j==(t.size()-1))//所有字符完全匹配
            return (i-t.size()+1);
    }
    return -1;//匹配失败
}
```

KMP算法

构造next数组(使用前缀表)

整体构建next数组的函数代码如下：

```
void get_next(string s){
    //初始化
    int j=0;
    nex[0]=j;
    for(int i=1;i<s.size();i++){
        while(j>0&&s[i]!=s[j])
        // j要保证大于0，因为下面有取j-1作为数组下标的操作
            j=nex[j-1];//向前回退，注意这里，是要找前一位的对应的回退位置了
        if(s[i]==s[j])
            j++;
        nex[i]=j;//将j（前缀的长度）赋给nex[i]
    }
}
```

KMP算法

[完整KMP代码链接(前缀表)]

使用next数组来做匹配

用模式串匹配文本串的整体代码如下：

```
int KMP(string s,string t)
{
    int j=0;
    for(int i=0;i<s.size();i++)//注意i从0开始
    {
        while(j>0&&s[i]!=t[j])//不匹配
            j=nex[j-1];//j寻找之前匹配的位置
        if(s[i]==t[j])//匹配,j和i同时向后移动
            j++;//i的增加在for循环中
        if(j==t.size())//所有字符完全匹配,注意此处是size()
            return (i-t.size()+1);
    }
    return -1;//匹配失败
}
```

KMP算法

给定一个模式串 S，以及一个模板串 P，所有字符串中只包含大小写英文字母以及阿拉伯数字。

模板串 P 在模式串 S 中多次作为子串出现。

求出模板串 P 在模式串 S 中所有出现的位置的起始下标。

对KMP函数稍作修改即可实现，其中**next数组实现用前缀表统一减一**的方式表示

```
void KMP(string s,string t){
    int j=-1;//因为nex数组里记录的起始位置为-1
    for(int i=0;i<s.size();i++){//注意i从0开始
        while(j>=0&&s[i]!=t[j+1])//不匹配
            j=nex[j];//j寻找之前匹配的位置
        if(s[i]==t[j+1])//匹配,j和i同时向后移动
            j++;//i的增加在for循环中
        if(j==(t.size()-1)){//所有字符完全匹配
            printf("%d ",i-j);
            j=nex[j];
        }
    }
}
```

Manacher算法(马拉车算法)

Manacher 算法是处理回文串的利器，由一位名叫 Manacher 的大佬在 1975 年提出，可在线性时间复杂度(即 O(n))内求解出最长回文子串。

问题:给定一个字符串 S，求出它的最长回文子串长度。
比如 **ab**abba**c** 的最长回文子串为abba，长度为4。

Manacher算法(马拉车算法)

字符串预处理

为了解决奇数长度的回文串，与偶数长度的回文串的不统一的问题，我们对给定字符串 S 预处理，即间隔插入分隔符#，如下所示：

已知长度为n的字符串,它的空隙有n+1个,往n+1个空隙填入分隔符#,故n+(n+1)=2*n+1必然为奇数

原串 abaa -> #a#b#a#a# --> !#a#b#a#a#@
最长回文子串 aba -> #a#b#a#

原串 abaabc -> #a#b#a#a#b#c# --> !#a#b#a#a#b#c#@
最长回文子串 baab -> #b#a#a#b#

有时为了方便,在预处理字符串时，其实是将原串，比如abab，处理成了**!#a#b#a#b#@**。下标0位置和末尾取了没有出现过的字符且首尾不相等，作为**哨兵**，就不用考虑边界啦

- 扩展后的原串长度=新串半径(包含中心点)-1

(例: baab 的长度 4 = #b#a#a#b# 半径为5 - 1； aba 的长度 3 = #a#b#a# 的半径 4 - 1)

Manacher算法(马拉车算法)

字符串预处理

预处理代码如右所示:

```
int len;//变形后字符串的长度
char b[MAXN];//存变形后的字符串
string s;//原始字符串

void init(){
    len=0;
    b[len++]='!';//起点哨兵
    b[len++]='#';
    for(int i=0;i<s.size();i++){
        b[len++]=s[i];
        b[len++]='#';
    }
    b[len++]='@';//末尾哨兵
}
```

Manacher算法(马拉车算法)

引入p数组, rt, mid

预处理后的新串表示为 b，定义数组p，p[i]表示 b 中**以下标i为回文中心的最大回文半径**。
以字符串abab为例：**!#a#b#a#b#@**,一般哨兵不用考虑p[i]

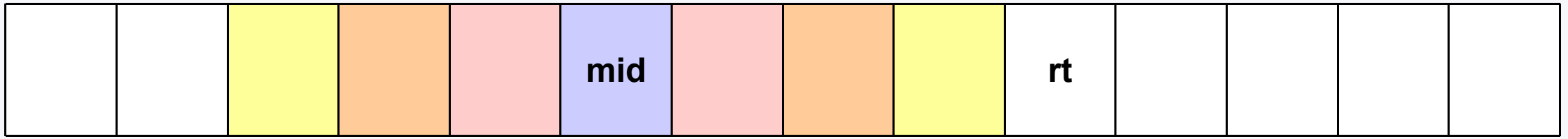
i	0	1	2	3	4	5	6	7	8	9	10
b	!	#	a	#	b	#	a	#	b	#	@
p[i]		1	2	1	4	1	4	1	2	1	

如果我们得到了p[i]，那么**p[i] - 1就是原串 S 以i为回文中心的最大回文长度**（可根据上表验证一下）。

Manacher算法(马拉车算法)

引入p数组, rt, mid

现在我们来求解p[i]，定义**rt表示已经计算过的回文串能达到的最远右边界的下一个位置**。即 $rt = \max(j + p[j]), j \in [1, i-1]$ ，mid表示rt所对应的最左侧的回文中心，有 $p[mid] + mid = rt$ 。如下图所示：



现在，已经计算完了p[1]~p[i-1]，发现，以mid为中心的回文子串右边界最远，能达到rt-1位置

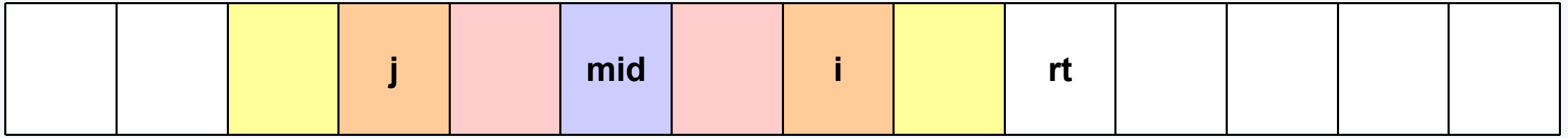
Manacher算法(马拉车算法)

计算p数组,更新 rt, mid

如何计算p[i]呢，显然有i > mid（因为p[mid]已经计算过）下面再分两种情况讨论：

情况一：i < rt

现在，求解p[i]，即以i为中心的最大回文半径。已知i<rt，我们找到i关于mid的对称点j



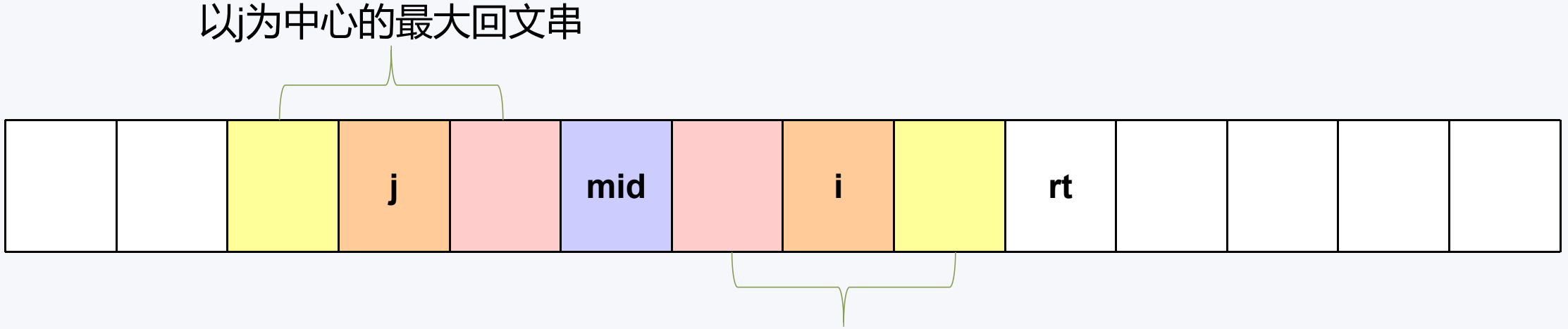
Manacher算法(马拉车算法)

计算p数组,更新 rt, mid

情况一：i < rt

此时我们发现，我们可以用已知的p[j]的信息来辅助计算p[i]。p[j]又有两种情况，讨论如下：

一、以j为中心的最大回文串被**包含**在以mid为中心的最大回文串中



由对称性可知，这一段也是回文，因此，p[i]最小也能达到**p[j]**，但rt及更右边是否能对以i为中心的回文串的扩展作出贡献，只能通过**暴力枚举**来检验。

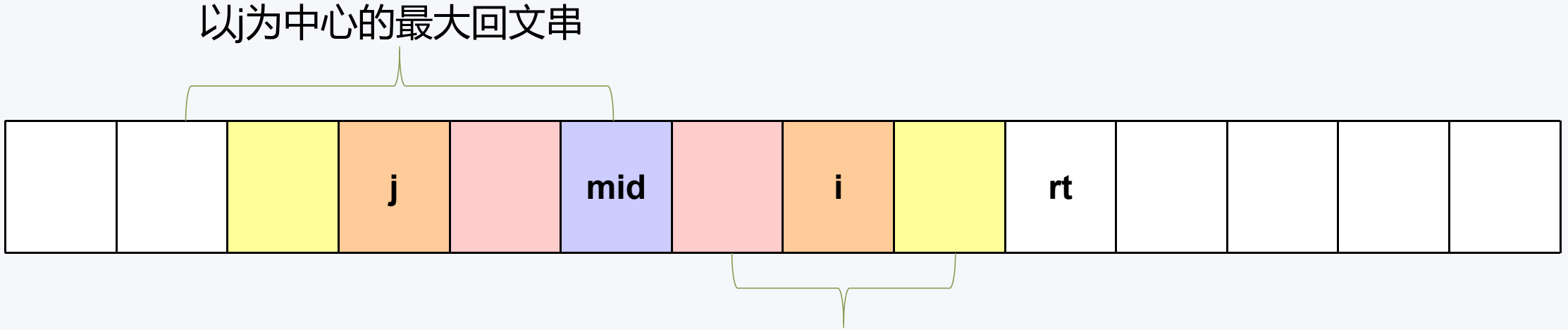
Manacher算法(马拉车算法)

计算p数组,更新 rt, mid

情况一：i < rt

此时我们发现，我们可以用已知的p[j]的信息来辅助计算p[i]。p[j]又有两种情况，讨论如下：

二、以j为中心的最大回文串**没有被包含**在以mid为中心的最大回文串中



由对称性可知，这一段也是回文，但不能触碰到rt
也就是说p[i]未必能达到p[j]的大小，但p[i]至少能达到rt-i同样地，对于rt及更右侧的部分，需要**暴力枚举**来检测是否能扩展回文

Manacher算法(马拉车算法)

计算p数组,更新 rt, mid

情况一：i < rt

根据以上两种情况，可以写出如下代码：

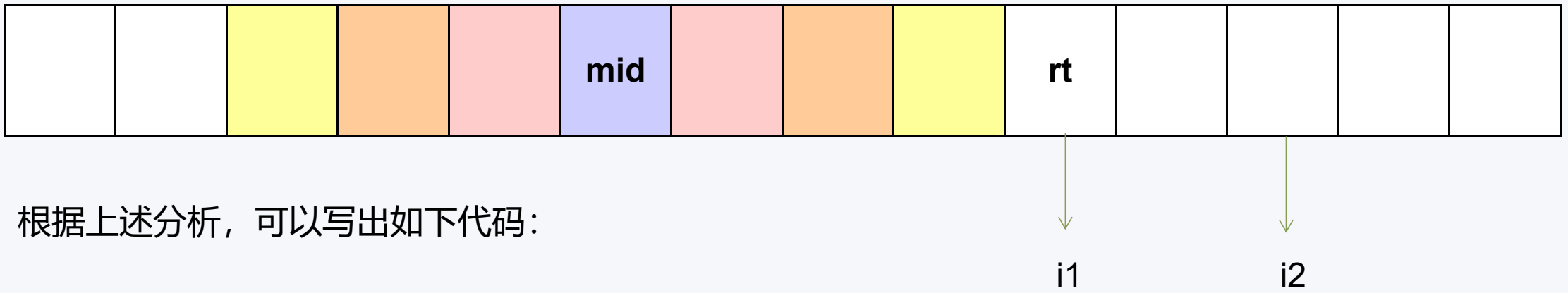
```
//由于i,j关于mid对称,(i+j)/2=mid,j=2*mid-i
if(i<rt)
        p[i]=min(p[2*mid-i],rt-i);
while(b[i+p[i]]==b[i-p[i]])//(此处存在哨兵,不用考虑边界问题)
//以i为中心p[i]为半径不断往外扩展,如果相等继续扩展
        p[i]++;
```

Manacher算法(马拉车算法)

计算p数组,更新 rt, mid

情况二：i >= rt

此时，没有已知信息可以辅助计算了，令p[i] = 1然后暴力扩展。



根据上述分析，可以写出如下代码：

```
if(i>=rt)
        p[i]=1;
while(b[i+p[i]]==b[i-p[i]])
//以i为中心p[i]为半径不断往外扩展,如果相等继续扩展
        p[i]++;
```

Manacher算法(马拉车算法)

计算p数组,更新 rt, mid

求解出p[i]以后，需要更新mid和rt，代码如下：

```
if(i+p[i]>rt){//如果以i为中心的最大回文串能更新rt
        rt=i+p[i];
        mid=i;//更新rt对应的mid
}
```

Manacher算法(马拉车算法)

马拉车代码链接

完整Manacher的代码如下：

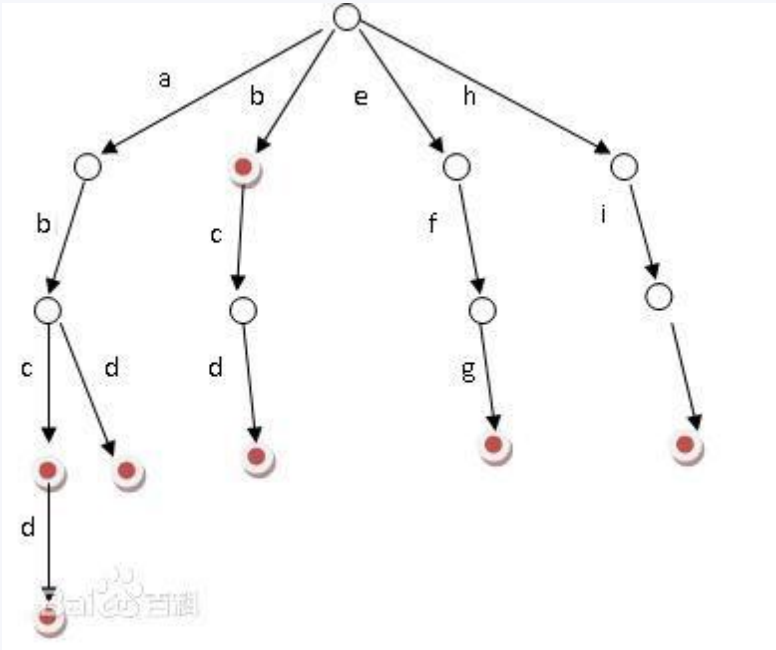
```
int manacher()
{
    int rt=0,mid=0,res=0;
    for(int i=1;i<=len-2;i++){//哨兵位置可不用考虑
        if(i<rt)
            p[i]=min(p[2*mid-i],rt-i);
        else p[i]=1;
        while(b[i+p[i]]==b[i-p[i]])
        //以i为中心p[i]为半径不断往外扩展,如果相等继续扩展
            p[i]++;
        if(i+p[i]>rt){//如果以i为中心的最大回文串能更新rt
            rt=i+p[i];
            mid=i;//更新rt对应的mid
        }
        res=max(res,p[i]-1);//注意此处是p[i]-1
    }
    return res;
}
```

字典树(Trie树)

字典树又称单词查找树，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，查询效率比哈希表高。

三大性质:

- 根节点不包含字符，除根节点外每一个节点都只包含一个字符；
- 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；
- 每个节点的所有子节点包含的字符都不相同。



字典树(Trie树)

Trie(字典树)是一种常用于实现字符串快速检索的多叉树结构。Trie的每个结点都拥有若干个字符指针,若在插入或检素字符串扫描到一个字符c,就会沿着当前结点的字符指针,指向该指针指向的结点。

初始化：一棵空的Trie仅包含一个根结点，该点的字符指针均指向空。

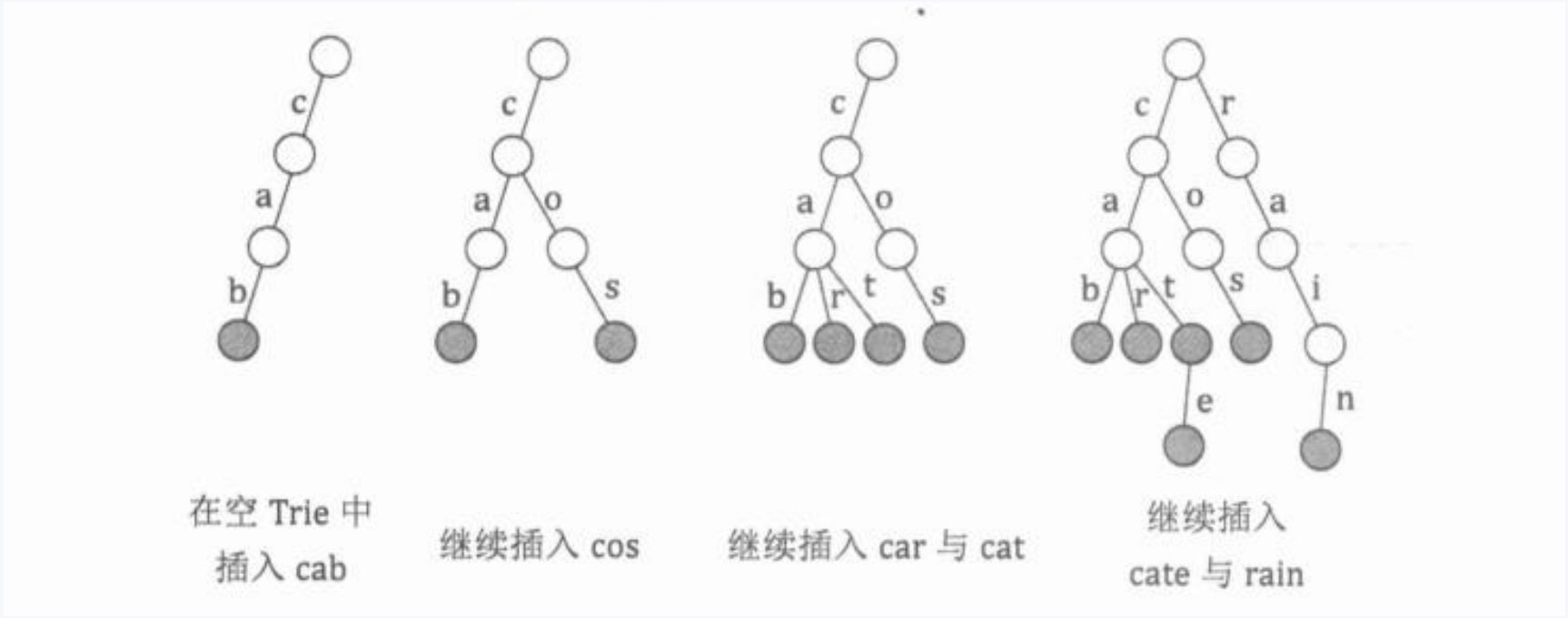
字典树(Trie树)

插入操作:

当需要插入一个字符串S时，令一个指针P指向根结点。然后，一次扫描S中的每个字符c。

- 如果P的c字符指针指向一个已经存在的节点Q，则令P=Q。
- 如果P的c字符指针指向空，则新建一个结点Q，令P的c字符指针指向Q,然后令P=Q

当S中的字符扫描完毕时，在当前结点P上标记它是字符串的末尾或者计算出现次数



字典树(Trie树)

插入操作代码:

```
int trie[MAXN][26],idx=0;//此处以小写字符(a~z)为例
int cnt[MAXN],vis[MAXN];
//cnt用来记录该字符串末尾出现次数;vis用来记录是否存在该字符串
void insert(char *s){//插入字符串
    int p=0;//指向根结点
    int len=strlen(s);
    for(int i=0;i<=len-1;i++){//遍历字符串s所有字符
        int u=s[i]-'a';
        if(trie[p][u]==0)//当前结点为空,新建
            trie[p][u]=++idx;
        p=trie[p][u];
    }
    cnt[p]++;//记录该字符串末尾出现的次数
    vis[p]=1;//为查询是否存在出现该字符串做准备,也可根据cnt数组判断
}
```

字典树(Trie树)

检索操作:
当需要检索一个字符串S在Trie树中是否存在(出现的次数)时，令一个指针P指向根结点。然后，一次扫描S中的每个字符c。

- 如果P的c字符指针指向一个已经存在的节点Q，则令P=Q。
- 如果P的c字符指针指向空，说明S没有被插入Trie，结束检索。

当S中的字符扫描完毕时，在当前结点P被标记为一个字符串的末尾,说明S在Trie中存在，否则说明S中没有插入过Trie。

字典树(Trie树)

字典树代码链接

检索操作代码:

```
int query(char *s){//查询字符串出现次数（或查询字符串是否出现过）
    int p=0;
    int len=strlen(s);
    for(int i=0;i<=len-1;i++){
        int u=s[i]-'a';
        if(trie[p][u]==0)
            return 0;//不存在
        p=trie[p][u];
    }
    return cnt[p];
    //return vis[p];
}
```