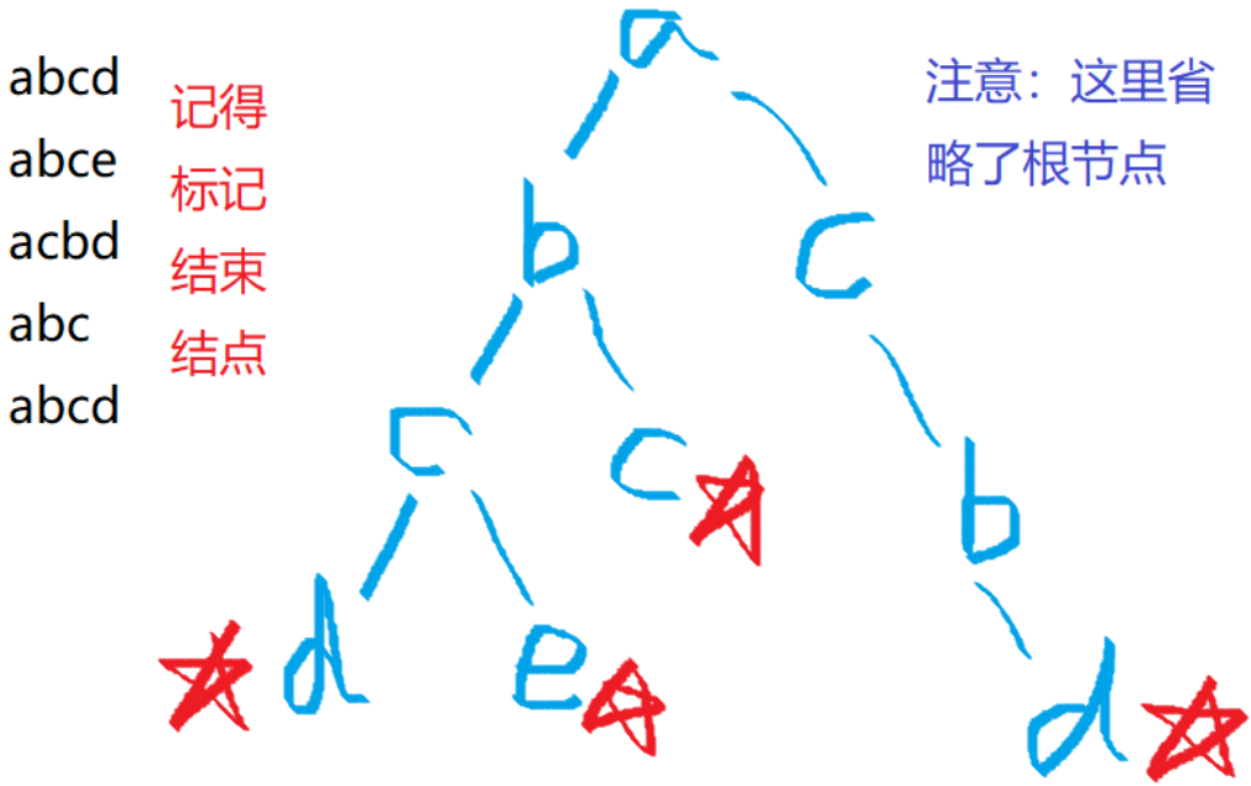
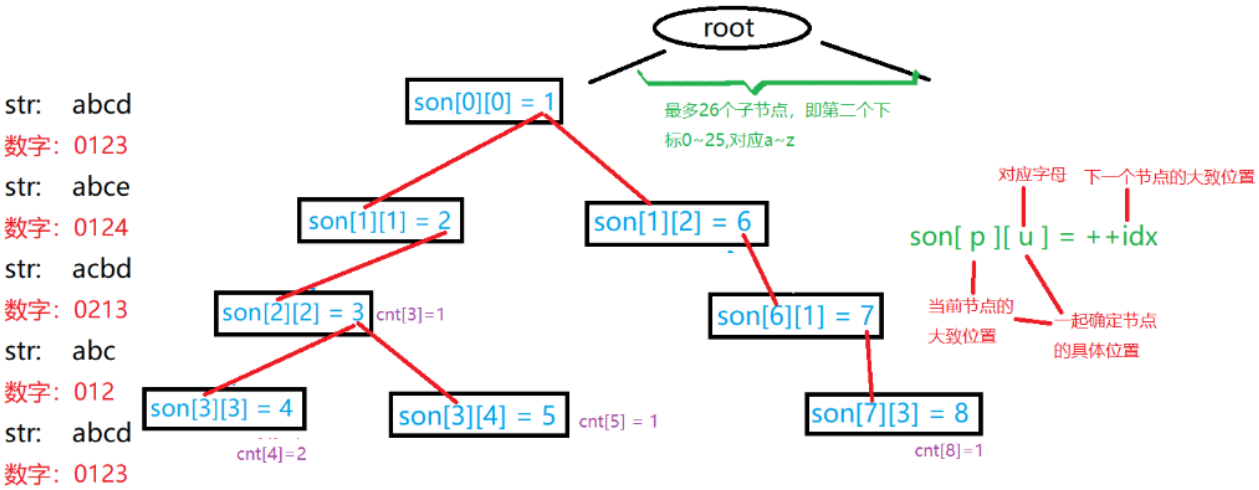
1：字典树： （Trie 树）

是一种高效的存储和查找字符串集合的数据结构，存储形式如下：



用数组来模拟Trie 树的具体分析：



插入操作代码：

```
public static void insert(char[] str){
        int p = 0;
        for(int i = 0; i < str.length; i++){
            int u = str[i] - 'a';
            if(son[p][u] == 0){
                son[p][u] = ++idx;
            }
```

```
            p = son[p][u];
        }
        cnt[p] ++;
    }
```

查找操作代码：

```
public static int query(char[] str){
        int p = 0;
        for(int i = 0; i < str.length; i++){
            int u = str[i] - 'a';
            if(son[p][u] == 0){
                return 0;
            }
            p = son[p][u];
        }
        return cnt[p];
    }
```

例题：

维护一个字符串集合，支持两种操作：

1：I x：向集合中插入一个字符串 x；

2：Q x：询问一个字符串在集合中出现了多少次

输入格式：

第一行包含整数 N，表示操作数

接下来 N 行，每行包含一个操作指令，指令为 I x 或 Q x 中的一种

输出格式：

对于每个询问指令 Q x，都要输出一个整数作为结果，表示 x 在集合中出现的次数，每个结果占一行

输入样例：

```
5
I abc
Q abc
Q ab
I ab
Q ab
```

输出样例：

```
1
0
1
```

```
import java.io.*;
public class Main{
    public static int idx; // idx表示当前要插入的节点是第几个,每创建一个节点值+1;
    public static int[][] son = new int[100010][26]; // son[][]存储子节点的位置，分支最多26条;
    public static int[] cnt = new int[100010]; // cnt[]存储以某节点结尾的字符串个数(同时也起标记作用);
    public static char[] str = new char[100010];
    public static void insert(char[] str){
        int p = 0;
        for(int i = 0; i < str.length; i++){
            int u = str[i] - 'a';
            if(son[p][u] == 0){
                son[p][u] = ++idx;
            }
            p = son[p][u];
        }
        cnt[p] ++;
    }
    public static int query(char[] str){
        int p = 0;
        for(int i = 0; i < str.length; i++){
            int u = str[i] - 'a';
            if(son[p][u] == 0){
                return 0;
            }
            p = son[p][u];
        }
        return cnt[p];
    }
    public static void main(String[] args) throws IOException{
        BufferedReader bw = new BufferedReader(new InputStreamReader(System.in));
```

```
        int n = Integer.parseInt(bw.readLine());
        while(n -- > 0){
            String[] strings = bw.readLine().split(" ");
            String op = strings[0];
            String str = strings[1];
            if(op.equals("I")){
                insert(str.toCharArray());
            }else{
                System.out.println(query(str.toCharArray()));
            }
        }
    }
}
```

2：并查集：

1：将两个集合合并

2：询问两个元素是否在一个集合当中

基本原理：

每个集合用一棵树来表示，树根的编号就是整个集合的编号，每个节点存储它的父节点，p [x] 表示 x 的父节点

问题 1：如何判断树根：

```
if(p[x] == x)
```

问题 2：如何求 x 的集合编号：

```
while(p[x] != x){
    x = p[x];
}
```

问题 3：如何合并两个集合：

px 是 x 的集合编号，py 是 y 的集合编号

```
p[x] = y;
```

例题：

一共有 n 个数，编号是 1 - n，最开始每个数在各自的集合中

现在要进行 m 个操作，操作共有两种：
1：M，a，b：将编号为a和b的两个数所在的集合合并，如果两个数已经在同一个集合中，则忽略这个操作
2：Q，a，b：询问编号为a和b的两个数是否在同一个集合中
输入格式：
第一行输入整数 m 和 n
接下来 m 行，每行包含一个操作指令，指令为M a b 或 Q a b中的一种
输出格式：
对于每个询问指令”Q a b”，都要输出一个结果，如果a和b在同一集合内，则输出“Yes”，否则输出“No”
每个结果占一行
输入样例：

```
4 5
M 1 2
M 3 4
Q 1 2
Q 1 3
Q 3 4
```

输出样例：

```
Yes
No
Yes
```

```java
import java.util.Scanner;
public class Main{
    public static int[] p = new int[100010];
    // find函数十分重要，要记住
    public static int find(int x){
        if(p[x] != x){
            p[x] = find(p[x]);
        }
```

```
        return p[x];
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            p[i] = i;
        }
        while(m -- > 0){
            String str = sc.next();
            char op = str.charAt(0);
            int a = sc.nextInt();
            int b = sc.nextInt();
            if(op == 'M'){
                p[find(a)] = find(b);
            }else{
                if(find(a) == find(b)){
                    System.out.println("Yes");
                }else{
                    System.out.println("No");
                }
            }
        }
    }
}
```

并查集可以扩展到维护点的数量：

例题：

给定一个包含n个点（编号为1~n）的无向图，初始时图中没有边

现在要进行m个操作，操作共有三种：

C a b，在点a和点b之间连一条边，a和b可能相等；

Q1 a b，询问点a和点b是否在同一个连通块中，a和b可能相等；

Q2 a ，询问点a所在连通块中点的数量；

输入格式

第一行输入整数n和m

接下来m行，每行包含一个操作指令，指令为 C a b，Q1 a b 或 Q2 a 中的一种

输出格式

对于每个询问指令 Q1 a b，如果a和b在同一个连通块中，则输出 Yes，否则输出 No
对于每个询问指令 Q2 a ，输出一个整数表示点a所在连通块中点的数量
每个结果占一行
输入样例：

```
5 5
C 1 2
Q1 1 2
Q2 1
C 2 5
Q2 5
```

输出样例：

```
Yes
2
3
```

```
import java.util.Scanner;
public class Main{
    public static int[] p = new int[100010];
    public static int[] size = new int[100010];
    public static int find(int x){
        if(p[x] != x){
            p[x] = find(p[x]);
        }
        return p[x];
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            p[i] = i;
            size[i] = 1;
        }
```

```
        while(m -- > 0){
            String str = sc.next();
            if(str.equals("C")){
                int a = sc.nextInt();
                int b = sc.nextInt();
                if(find(a) == find(b)){
                    continue;
                }else{
                    size[find(b)] += size[find(a)];
                    p[find(a)] = find(b);
                }
            }else if(str.equals("Q1")){
                int a = sc.nextInt();
                int b = sc.nextInt();
                if(find(a) == find(b)){
                    System.out.println("Yes");
                }else{
                    System.out.println("No");
                }
            }else{
                int a = sc.nextInt();
                System.out.println(size[find(a)]);
            }
        }
    }
}
```

3：堆：（从小到大 / 从大到小输出前 k 个数）

如何手写一个堆：

1：插入一个数：heap [++ size]，up (size) ；

2：求集合当中的最小值：heap [1]

3：删除最小值：heap [1] = heap [size]，size --，down (1)

4：删除任意一个元素：heap [k] = heap [size]，size -- , down (k)，up (k);

5：修改任意一个元素：heap [k] = x，down (k)，up (k);

例题：

输入一个长度为 n 的整数数列，从小到大输出前 m 小的数

输入格式：

第一行包含整数 n 和 m

第二行包含 n 个整数，表示整数数列

输出格式：

共一行， 包含 m 个整数，表示整数数列中前 m 小的数

输入样例：

```
5 3
4 5 3 1 2
```

输出样例：

```
1 2 3
```

```
import java.util.Scanner;
public class Main{
    public static int[] h = new int[100010];
    public static int size;
    public static void swap(int x, int y){
        int temp = h[x];
        h[x] = h[y];
        h[y] = temp;
    }
    public static void down(int u){
        int t = u;
        if(u * 2 <= size && h[u * 2] < h[t]){
            t = u * 2;
        }
        if(u * 2 + 1 <= size && h[u * 2 + 1] < h[t]){
            t = u * 2 + 1;
        }
        if(u != t){
            swap(u, t);
            down(t);
        }
    }
    public static void main(String[] args){
```

```
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            h[i] = sc.nextInt();
        }
        size = n;
        // 为什么从n/2开始down
        // 首先要明确要进行down操作时必须满足左儿子和右儿子已经是个堆
        // 开始创建堆的时候，元素是随机插入的，所以不能从根节点开始down，而是要找到满足下面三个性质的结点：
        // 1.左右儿子满足堆的性质
        // 2.下标最大（因为要往上遍历）
        // 3.不是叶结点（叶节点一定满足堆的性质）
        for(int i = n / 2; i >= 0; i--){
            down(i);
        }
        while(m -- > 0){
            System.out.printf("%d ", h[1]);
            h[1] = h[size];
            size --;
            down(1);
        }
    }
}
```

4：哈希表：

1：存储结构：

开放寻址法，拉链法

2：字符串哈希方式----------字符串前缀哈希法

例题：（利用拉链法）

维护一个集合，支持如下几种操作：

1：I x：插入一个数 x

2：Q x：询问数 x 是否在集合中出现过

现在要进行 N 次操作，对于每个询问操作输出对应的结果

输入格式：

第一行包含整数 N，表示操作数量

接下来 N 行，每行包含一个操作指令， 操作指令为 I，x，Q，x 中的一种

输出格式：

对于每个询问指令 Q x，输出一个询问结果，如果 x 在集合中出现过，则输出 yes，否则输出 no

输入样例：

```
5
I 1
I 2
I 3
Q 2
Q 5
```

输出样例：

```
Yes
No
```

```
import java.util.Scanner;
public class Main{
    public static int idx;
    public static int N = 100003; // 一般为质数
    public static int[] h = new int[100003];
    public static int[] e = new int[100003];
    public static int[] ne = new int[100003];
    public static void add(int x){
        int k = (x % N + N) % N;
        e[idx] = x;
        ne[idx] = h[k];
        h[k] = idx;
        idx++;
    }
    public static boolean find(int x){
        int k = (x % N + N) % N;
        for(int i = h[k]; i != -1; i = ne[i]){
            if(e[i] == x){
                return true;
```

```
            }
        }
        return false;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for(int i = 0; i < N; i++){
            h[i] = -1;
        }
        while(n -- > 0){
            String str = sc.next();
            if(str.equals("I")){
                int a = sc.nextInt();
                add(a);
            }else{
                int b = sc.nextInt();
                if(find(b)){
                    System.out.println("Yes");
                }else{
                    System.out.println("No");
                }
            }
        }
    }
}
```

3：字符串哈希：

字符串前缀哈希法：（用来比较某一个字符串中两段字符相等不相等）

把字符串变成一个p进制数字（哈希值），实现不同的字符串映射到不同的数字

对形如 X1X2X3…Xn−1XnX1X2X3…Xn−1Xn 的字符串,采用字符的ascii 码乘上 P 的次方来计算哈希值

映射公式 (X1×Pn−1+X2×Pn−2+…+Xn−1×P1+Xn×P0) modQ

注意点：

1：任意字符不可以映射成0，否则会出现不同的字符串都映射成0的情况，比如A,AA,AAA皆为0

2：冲突问题：通过巧妙设置P (131 或 13331) , Q (2^ 64)的值，一般可以理解为不产生冲突

求一个字符串的哈希值就相当于求前缀和，求一个字符串的子串哈希值就相当于求部分和
前缀和公式 h[i+1]=h[i]×P+s[i]h[i+1]=h[i]×P+s[i] i∈[0,n−1]i∈[0,n−1] h为前缀和数组，s为字符串数组
区间和公式 h[l , r]=h[r]−h[l−1]×P^ (r - l + 1)
例题：
给定一个长度为n的字符串，再给定m个询问，每个询问包含四个整数l1,r1,l2,r2，请你判断[l1,r1]和[l2,r2]这两个区间所包含的字符串子串是否完全相同
字符串中只包含大小写英文字母和数字
输入格式：
第一行包含整数n和m，表示字符串长度和询问次数，第二行包含一个长度为n的字符串，字符串中只包含大小写英文字母和数字，接下来m行，每行包含四个整数l1,r1,l2,r2，表示一次询问所涉及的两个区间
注意，字符串的位置从1开始编号
输出格式：
对于每个询问输出一个结果，如果两个字符串子串完全相同则输出“Yes”，否则输出“No”
每个结果占一行
输入样例：

```
8 3
aabbaabb
1 3 5 7
1 3 6 8
1 2 1 2
```

输出样例：

```
Yes
No
Yes
```

```
import java.util.Scanner;
public class Main{
    public static int N = 100010;
```

```
    public static int P = 131;
    public static long[] h = new long[100010];
    public static long[] p = new long[100010];
    public static long get(int l, int r){
        return h[r] - h[l - 1] * p[r - l + 1];
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        String str = sc.next();
        p[0] = 1;
        for(int i = 1; i <= n; i++){
            p[i] = p[i - 1] * P;
            h[i] = h[i - 1] * P + str.charAt(i - 1);
        }
        while(m -- > 0){
            int l1 = sc.nextInt();
            int r1 = sc.nextInt();
            int l2 = sc.nextInt();
            int r2 = sc.nextInt();
            if(get(l1, r1) == get(l2, r2)){
                System.out.println("Yes");
            }else{
                System.out.println("No");
            }
        }
    }
}
```