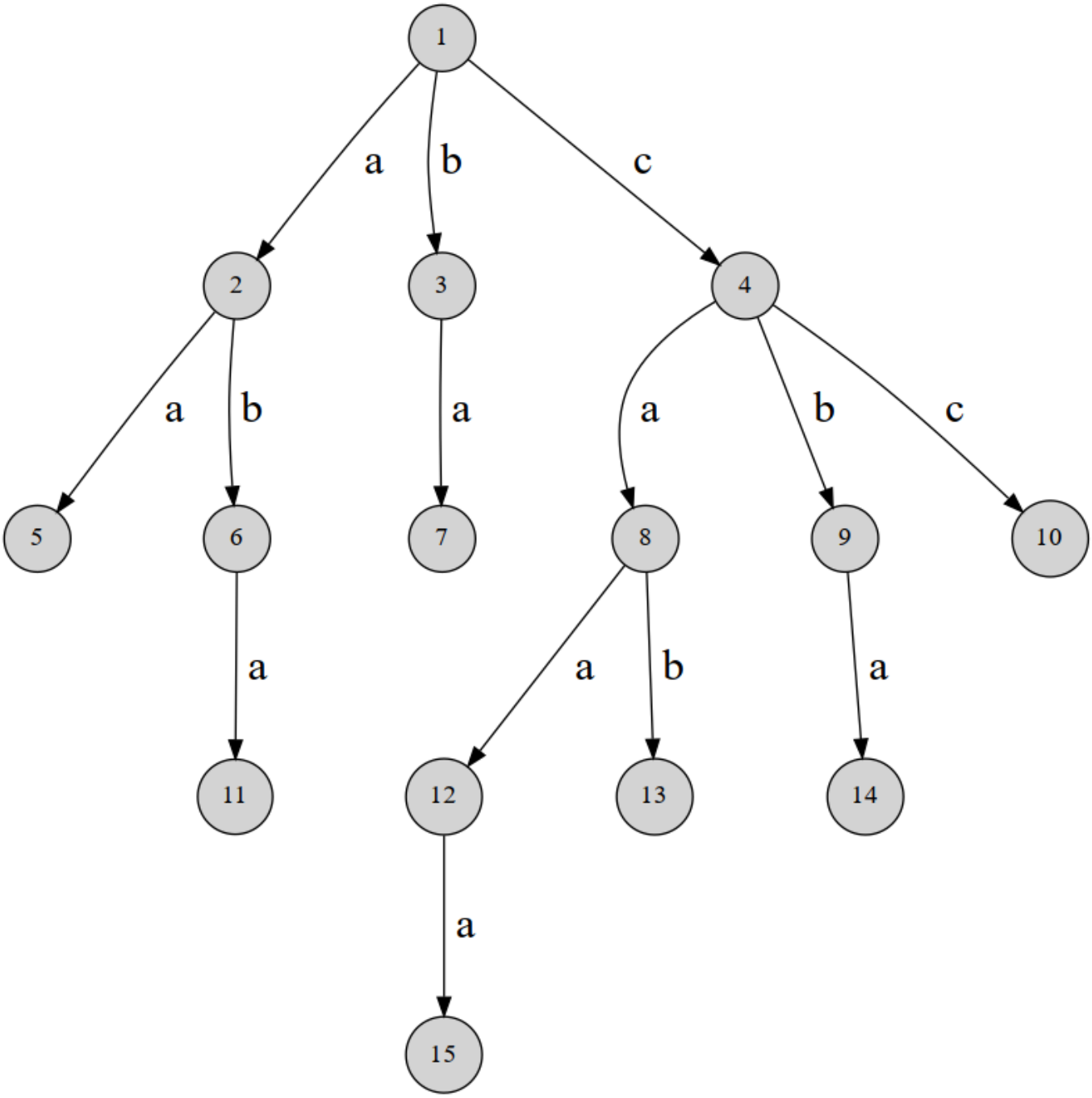
字典树(Trie)

字典树，英文名 trie，顾名思义，就是一个像字典一样的树。

简介

先放一张图：



Trie

可以发现，这棵字典树用边来代表字母，而从根结点到树上某一结点的路径就代表了一个字符串。举个例子， $1\rightarrow4\rightarrow8\rightarrow12$ 表示的就是字符串 caa。

trie 的结构非常好懂，我们用 $\delta(u,c)$ 表示结点 u 的 c 字符指向的下一个结点，或者说是结点 u 代表的字符串后面添加一个字符 c 形成的字符串的结点□□c 的取值范围和字符集大小有关，不一定是 $0\sim26$□□

有时需要标记插入进 trie 的是哪些字符串，每次插入完成时在这个字符串所代表的结点处打上标记即可。

代码实现

放一个结构体封装的模板：

```
struct trie {
  int nex[100000][26], cnt;
  bool exist[100000];  //该结点结尾的字符串是否存在

  void insert(char *s, int l) {  //插入字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) nex[p][c] = ++cnt;  //如果没有，就添加结点
      p = nex[p][c];
    }
    exist[p] = 1;
  }
  bool find(char *s, int l) {  //查找字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) return 0;
      p = nex[p][c];
    }
    return exist[p];
  }
};
```

应用

检索字符串

字典树最基础的应用——查找一个字符串是否在“字典”中出现过。

于是他错误的点名开始了

给你 n 个名字串，然后进行 m 次点名，每次你需要回答“名字不存在”、“第一次点到这个名字”、“已经点过这个名字”之一。

$1\le n\le 10^4,1\le m\le 10^5$□所有字符串长度不超过 50。

> 题解
>
> 对所有名字建 trie，再在 trie 中查询字符串是否存在、是否已经点过名，第一次点名时标记为点过名。

参考代码

```
#include <cstdio>

const int N = 500010;

char s[60];
int n, m, ch[N][26], tag[N], tot = 1;

int main() {
 scanf("%d", &n);

 for (int i = 1; i <= n; ++i) {
   scanf("%s", s + 1);
   int u = 1;
   for (int j = 1; s[j]; ++j) {
     int c = s[j] - 'a';
     if (!ch[u][c]) ch[u][c] = ++tot;
     u = ch[u][c];
   }
   tag[u] = 1;
 }

 scanf("%d", &m);

 while (m--) {
   scanf("%s", s + 1);
   int u = 1;
   for (int j = 1; s[j]; ++j) {
     int c = s[j] - 'a';
     u = ch[u][c];
     if (!u) break;  //不存在对应字符的出边说明名字不存在
   }
   if (tag[u] == 1) {
     tag[u] = 2;
     puts("OK");
   } else if (tag[u] == 2)
     puts("REPEAT");
   else
     puts("WRONG");
 }

 return 0;
}
```

AC 自动机

trie 是 AC 自动机的一部分

维护异或极值

将数的二进制表示看做一个字符串，就可以建出字符集为 $\{0,1\}$ 的 trie 树。

BZOJ1954 最长异或路径

给你一棵带边权的树，求 (u,v) 使得 u 到 v 的路径上的边权异或和最大，输出这个最大值。

点数不超过 10^5，边权在 $[0,2^{31})$ 内。

> 题解
>
> 随便指定一个根 $root$，用 $T(u,v)$ 表示 u 和 v 之间的路径的边权异或和，那么 $T(u,v)=T(root,u)\oplus T(root,v)$，因为 LCA 以上的部分异或两次抵消了。
>
> 那么，如果将所有 $T(root,u)$ 插入到一棵 trie 中，就可以对每个 $T(root,u)$ 快速求出和它异或和最大的 $T(root,v)$。
>
> 从 trie 的根开始，如果能向和 $T(root,u)$ 的当前位不同的子树走，就向那边走，否则没有选择。
>
> 贪心的正确性：如果这么走，这一位为 1；如果不这么走，这一位就会为 0。而高位是需要优先尽量大的。

参考代码：

```
#include <algorithm>
#include <cstdio>

const int N = 100010;

int head[N], nxt[N << 1], to[N << 1], weight[N << 1], cnt;
int n, dis[N], ch[N << 5][2], tot = 1, ans;

void insert(int x) {
  for (int i = 30, u = 1; i >= 0; --i) {
    int c = ((x >> i) & 1);
    if (!ch[u][c]) ch[u][c] = ++tot;
    u = ch[u][c];
  }
}

void get(int x) {
  int res = 0;
  for (int i = 30, u = 1; i >= 0; --i) {
    int c = ((x >> i) & 1);
```

```
    if (ch[u][c ^ 1]) {
      u = ch[u][c ^ 1];
      res |= (1 << i);
    } else
      u = ch[u][c];
  }
  ans = std::max(ans, res);
}

void add(int u, int v, int w) {
  nxt[++cnt] = head[u];
  head[u] = cnt;
  to[cnt] = v;
  weight[cnt] = w;
}

void dfs(int u, int fa) {
  insert(dis[u]);
  get(dis[u]);
  for (int i = head[u]; i; i = nxt[i]) {
    int v = to[i];
    if (v == fa) continue;
    dis[v] = dis[u] ^ weight[i];
    dfs(v, u);
  }
}

int main() {
  scanf("%d", &n);

  for (int i = 1; i < n; ++i) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    add(u, v, w);
    add(v, u, w);
  }

  dfs(1, 0);

  printf("%d", ans);

  return 0;
}
```

维护异或和

01-trie 是指字符集为 ${0,1}$ 的trie□01-trie 可以用来维护一些数字的异或和，支持修改（删除+重新插入），和全局加一（即：让其所维护所有数值递增 1，本质上是一种特殊的修改操作）。

如果要维护异或和，需要按值从低位到高位建立 trie□

一个约定：文中说当前结点**往上**指当前结点到根这条路径，当前结点**往下**指当前结点的子树。

插入 & 删除

如果要维护异或和，我们**只需要**知道某一位上0和1个数的**奇偶性**即可，也就是对于数字1来说，当且仅当这一位上数字1的个数为奇数时，这一位上的数字才是1，请时刻记住这段文字：如果只是维护异或和，我们只需要知道某一位上1的数量即可，而不需要知道 trie 到底维护了哪些数字。

对于每一个结点，我们需要记录以下三个量：

- `ch[o][0/1]`指结点o的两个儿子，`ch[o][0]`指下一位是0，同理`ch[o][1]`指下一位是1。
- `w[o]`指结点o到其父亲结点这条边上数值的数量（权值）。每插入一个数字x，x二进制拆分后在 trie 上路径的权值都会+1。
- `xorv[o]`指以o为根的子树维护的异或和。

具体维护结点的代码如下所示。

```
void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  // w[o] = w[o] & 1;
  //只需知道奇偶性即可，不需要具体的值。当然这句话删掉也可以，因为上文就只利用了他的奇偶性。
}
```

插入和删除的代码非常相似。

需要注意的地方就是：

- 这里的MAXH指 trie 的深度，也就是强制让每一个叶子结点到根的距离为MAXH。对于一些比较小的值，可能有时候不需要建立这么深（例如：如果插入数字4，分解二进制后为100，从根开始插入001这三位即可），但是我们强制插入MAXH位。这样做的目的是为了便于全局+1时处理进位。例如：如果原数字是3（11），递增之后变成4（100），如果当初插入3时只插入了2位，那这里的进位就没了。
- 插入和删除，只需要修改叶子结点的`w[]`即可，在回溯的过程中一路维护即可。

```
namespace trie {
const int MAXH = 21;
int ch[_ * (MAXH + 1)][2], w[_ * (MAXH + 1)], xorv[_ * (MAXH + 1)];
int tot = 0;
int mknode() {
  ++tot;
  ch[tot][1] = ch[tot][0] = w[tot] = xorv[tot] = 0;
  return tot;
```

```
}
void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  w[o] = w[o] & 1;
}
void insert(int &o, int x, int dp) {
  if (!o) o = mknode();
  if (dp > MAXH) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}
void erase(int o, int x, int dp) {
  if (dp > 20) return (void)(w[o]--);
  erase(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}
}  // namespace trie
```

全局加一

所谓全局加一就是指，让这棵 trie 中所有的数值+1。

形式化地讲，设 trie 中维护的数值有 $V_1,V_2,V_3,\cdots,V_n$。全局加一后其中维护的值应该变成 $V_1+1,V_2+1,V_3+1,\cdots,V_n+1$。

```
void addall(int o) {
  swap(ch[o][0], ch[o][1]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
```

我们思考一下二进制意义下+1是如何操作的。

我们只需要从低位到高位开始找第一个出现的0，把它变成1，然后这个位置后面的1都变成0即可。

下面给出几个例子感受一下：（括号内的数字表示其对应的十进制数字）

```
1000(10)  + 1 = 1001(11)  ;
10011(19) + 1 = 10100(20) ;
11111(31) + 1 = 100000(32);
10101(21) + 1 = 10110(22) ;
100000000111111(16447) + 1 = 100000001000000(16448);
```

对应 trie 的操作，其实就是交换其左右儿子，顺着**交换后**的0边往下递归操作即可。

回顾一下w[o]的定义：w[o]指结点o到其父亲结点这条边上数值的数量（权值）。

有没有感觉这个定义有点怪呢？如果在父亲结点存储到两个儿子的这条边的边权也许会更接近于习惯。但是在这里，在交换左右儿子的时候，在儿子结点存储到父亲这条边的距离，显然更加方便。

01-trie 合并

待补

可持久化字典树

待补

练习题

luogu-P6018-Fusion tree

luogu-P6623-树

参考链接

Oi Wiki