AC自动机

- AC自动机：`Trie`+`KMP`的组合。
- AC自动机是在KMP的基础上进行扩展，在KMP中，我们存在模式串p以及被匹配的串s，我们可以通过KMP算法在`O(n)`的时间内判断p是否在s中出现过、出现的位置以及出现的次数。AC自动机实质上是将模式串p换成了trie树。

算法流程

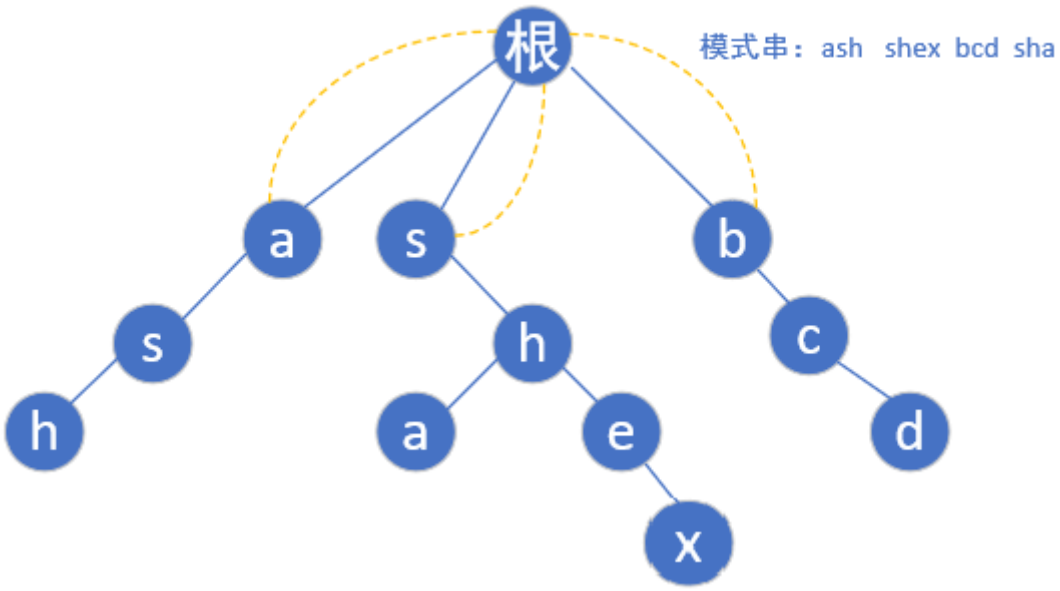
- 构造一棵 Trie，作为AC自动机的搜索数据结构。
- 构造 next 指针，使当前字符失配时跳转到具有最长公共前后缀的字符继续匹配。如同 KMP算法一样， AC自动机在匹配时如果当前字符匹配失败，那么利用 next 指针进行跳转。由此可知如果跳转，跳转后的串的前缀，必为跳转前的模式串的后缀并且跳转的新位置的深度（匹配字符个数）一定小于跳之前的节点。所以我们可以利用 bfs 在 Trie 上面进行每一层节点 next 指针的求解。
- 扫描主串进行匹配。

算法图解

建立Trie

- 首先给定模式串`ash,shex,bcd,sha`,然后我们根据模式串建立如下Trie树:

```
void insert(const char* str) {
    int p = 0;
    for (int i = 0; str[i]; i++) {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++idx;
        p = son[p][u];
    }
    cnt[p]++;
}
```



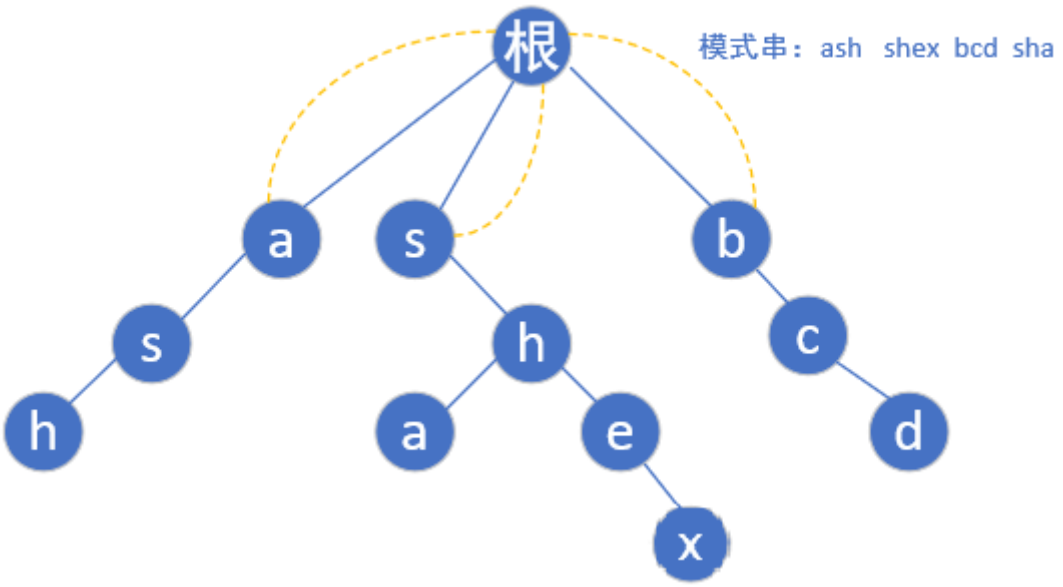
构造next

- AC自动机就是在Trie树的基础上，增加一个失配时候用的next指针，如果当前点匹配失败，则将指针转移到next指针指向的地方，这样就不用回溯,而可以继续匹配下去了。一般，next指针的构建都是用 bfs 实现的。

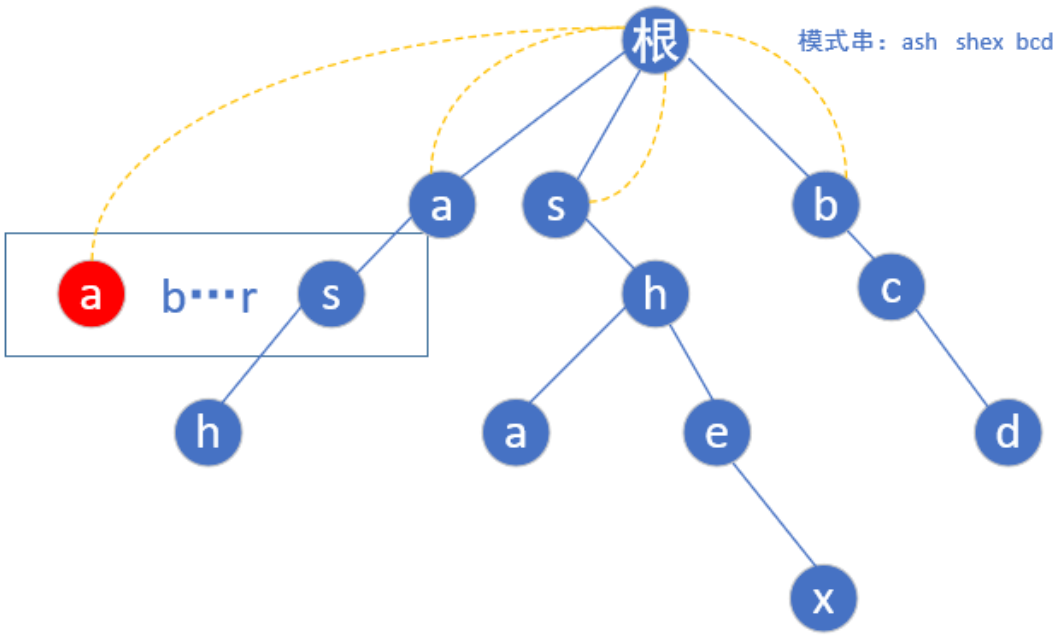
  核心： 当前模式串后缀和 next 指针指向的模式串部分前缀相同。

```
void build() {
    int hh = 0, tt = -1;

    for (int i = 0; i < 26; i++)
        if (son[0][i]) q[++tt] = son[0][i];

    while (hh <= tt) {
        int t = q[hh++];
        for (int i = 0; i < 26; i++) {
            int& p = son[t][i];
            if (!p) p = son[ne[t]][i];
            else {
                ne[p] = son[ne[t]][i];
                q[++tt] = p;
            }
        }
    }
}
```

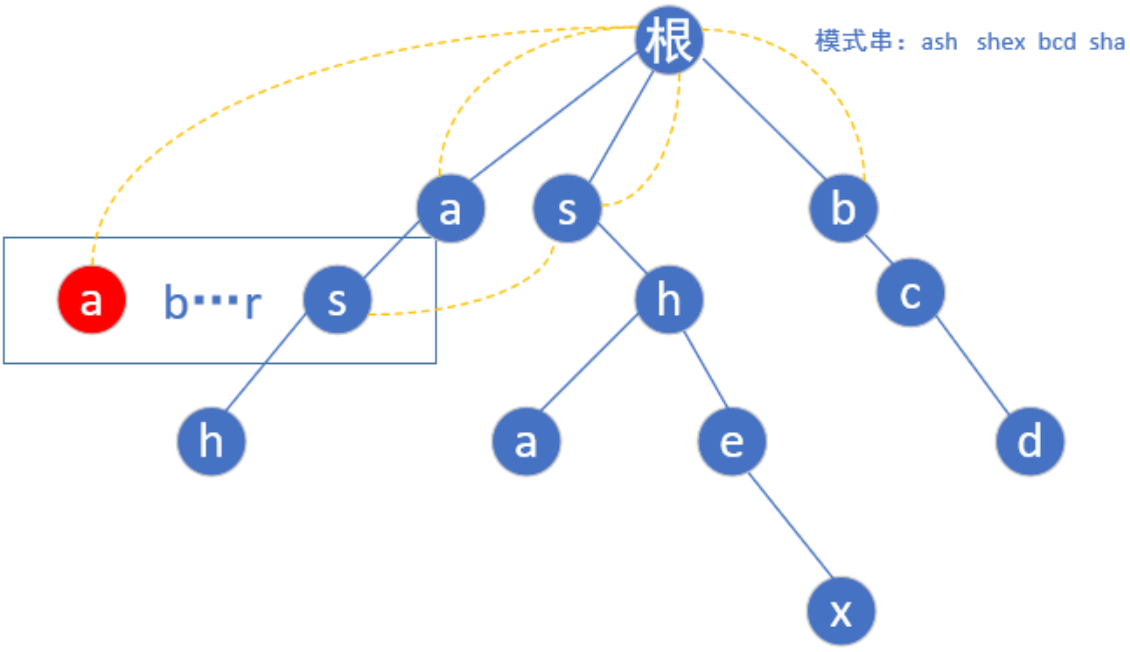
- 每个模式串的首字母肯定是指向根节点的：



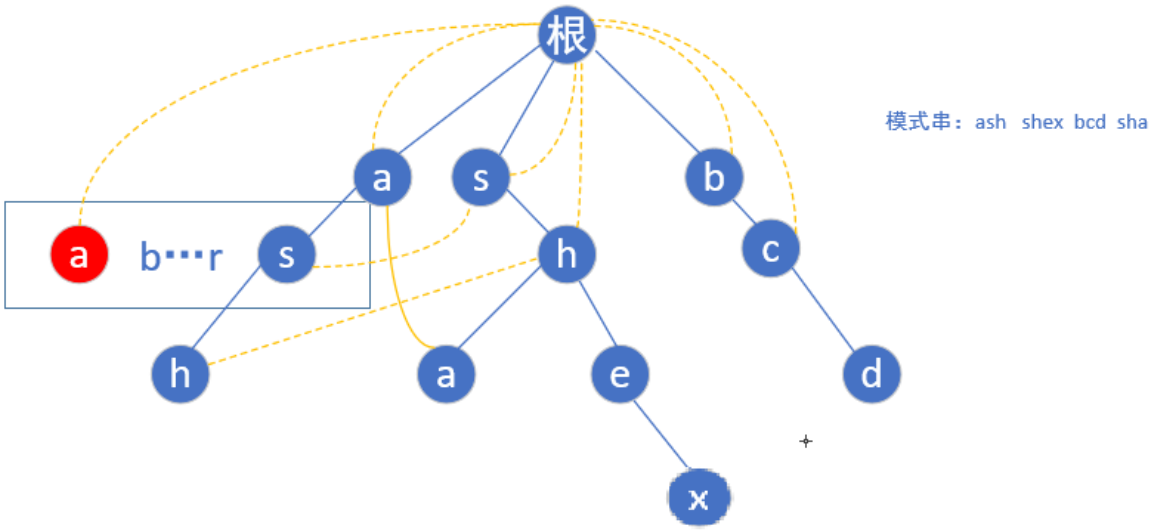
- 第一层 bfs 遍历完，开始第二层（根节点为第0层）第二层节点 s 是第一层节点 a 的子节点，但是我们还是要从 `a - z` 遍历。如果不存在这个子节点，树节点值也等于父节点的 next 指向的节点中具有相同字母的子节点（如下图中的红色 a ）。



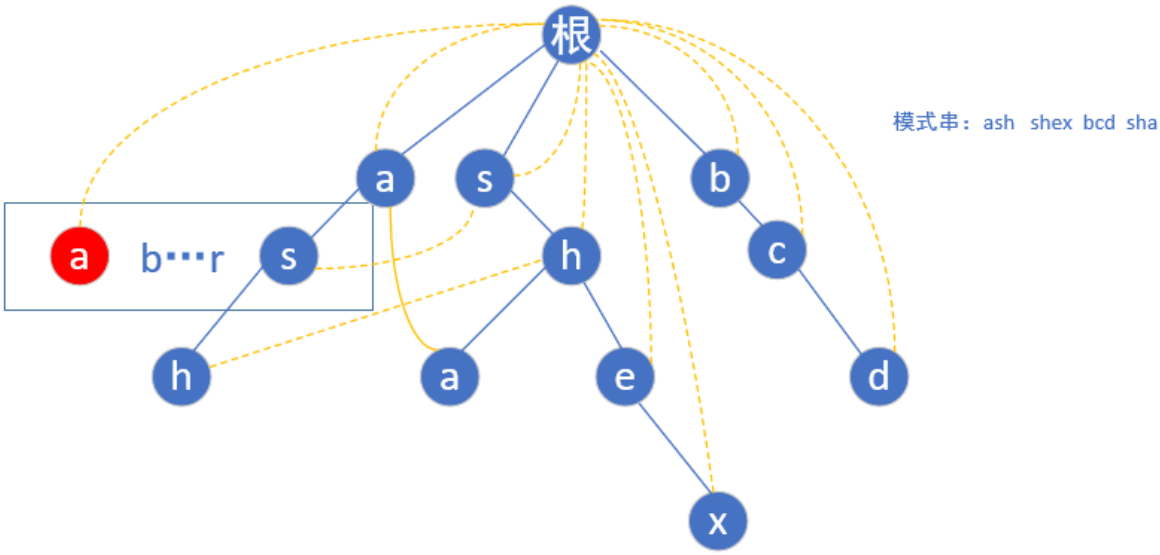
- 当我们遍历到 s 的时候，由于存在 s 这个节点，我们就让它的 next 指针指向他父亲节点 a 的 next 指针指向的那个节点（根）的具有相同字母的子节点（第一层的 s）。



- 按照相同规律构建第二层后，到了第三层的 h 点，还是按照上面的规则，我们找到 h 的父亲节点 s 的 next 指针指向的那个位置（第一层的 s），然后指向它所指向的相同字母 `根 -> s -> h` 的这个链的 h 节点。

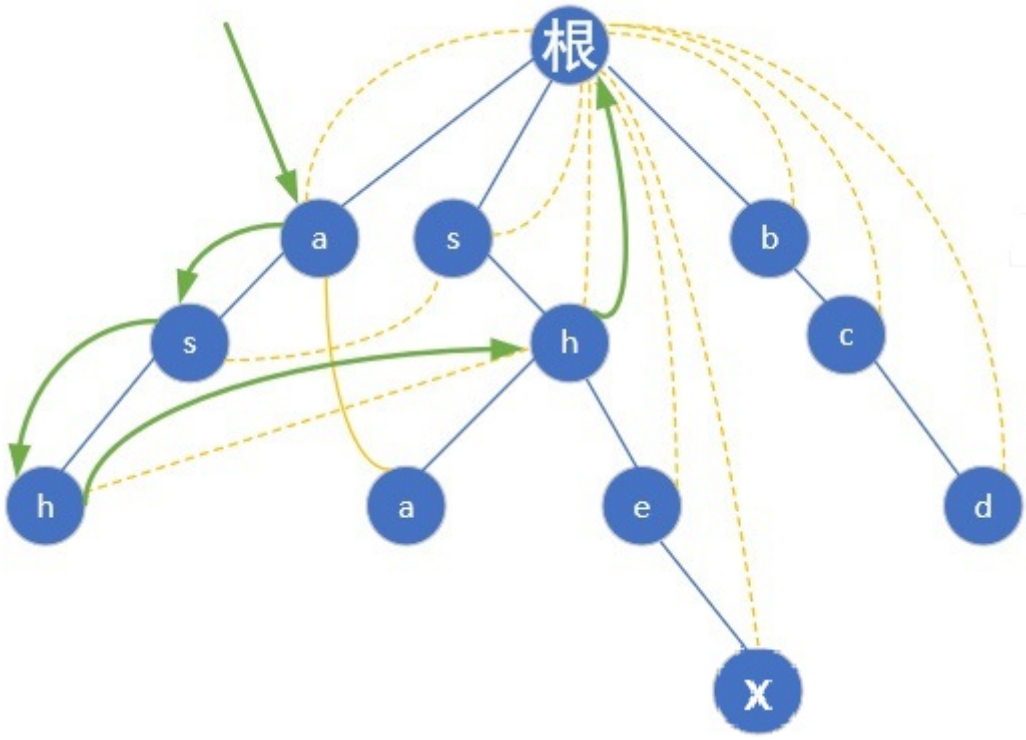


- 按照如上的规律，完全构造好后的树如下所示：



匹配过程

```
for (int i = 0, j = 0; str[i]; i++) {
    int u = str[i] - 'a';
    j = son[j][u];
    int p = j;
    while (p) {
        res += cnt[p];
        cnt[p] = 0;
        p = ne[p];
    }
}
```



快乐刷题

- P73 Keywords Search
- P454 单词