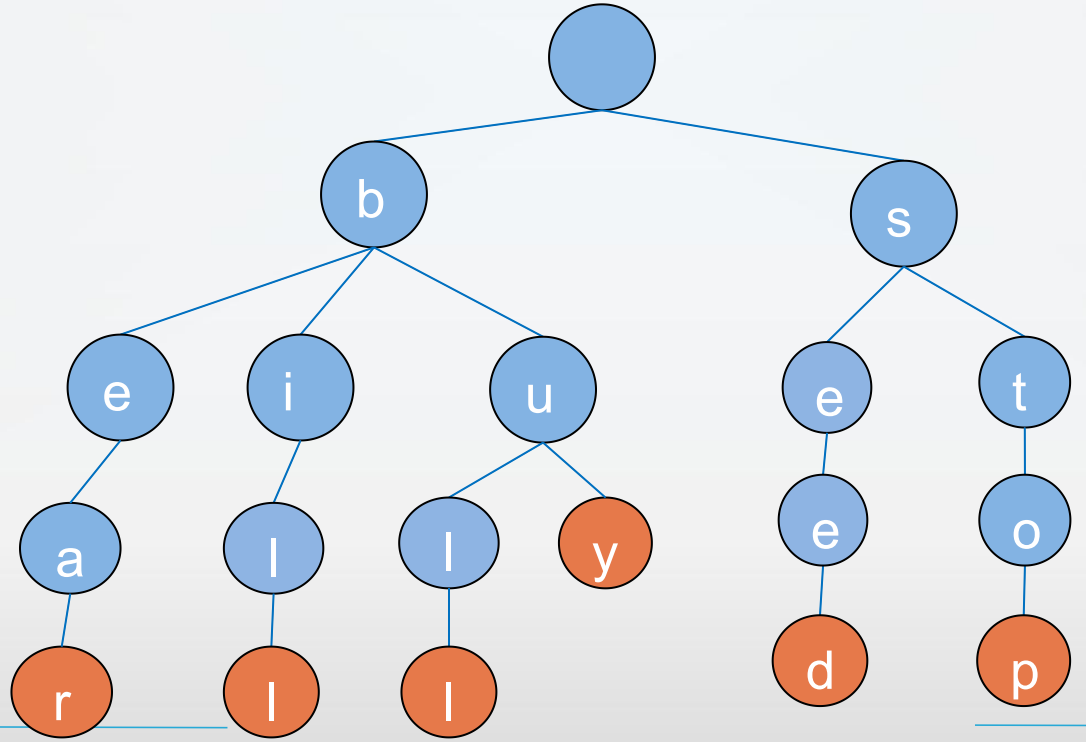
Trie树

Trie树

Trie树，又叫前缀树、字典树、键树等，在很多字符串相关的问题中应用广泛，比如存储字典、字符串的快速检索、求最长公共前缀、快速统计和排序大量字符串等，典型应用是搜索引擎系统用于文本词频统计。



Trie树

基本性质：

- 根结点不包含字符，除根节点以外每个结点只包含一个字符。
- 从根结点到某一个结点，路径上经过的字符连接起来，为该结点对应的字符串。
- 每个结点的所有子结点包含的字符都不相同。

核心思想：

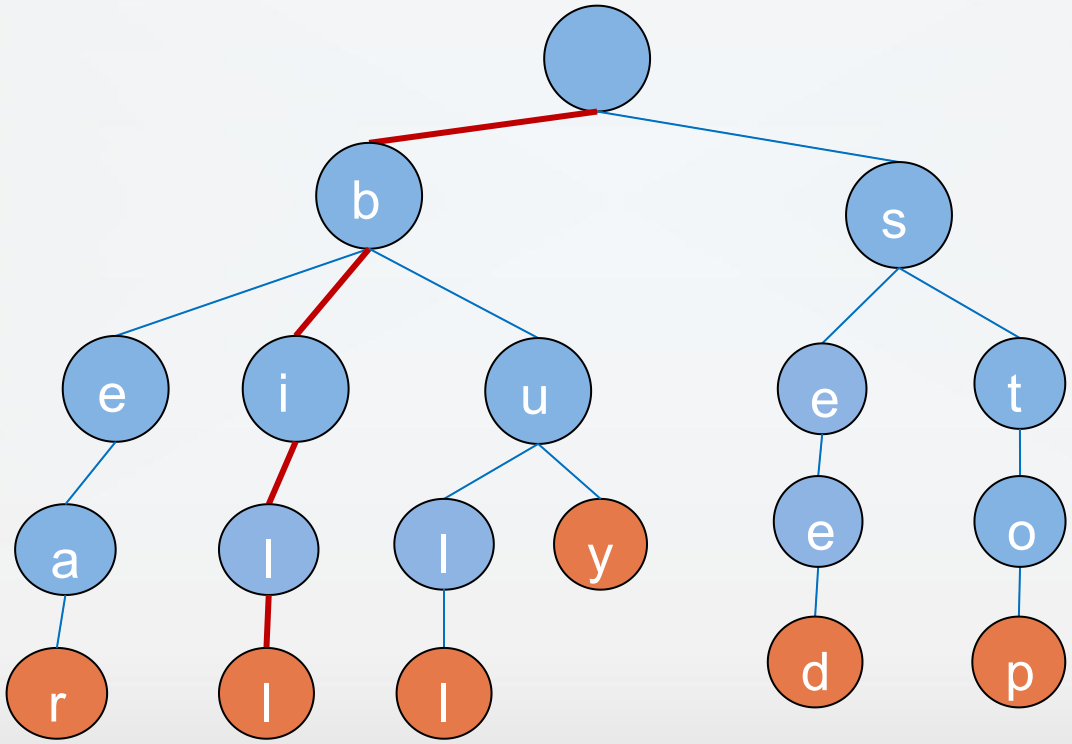
通过最大限度地减少字符串的比较，「用空间换时间」，利用共同前缀来提高查询效率。

Trie树

- 基本操作
  - 创建
  - 查找
  - 插入
  - 删除

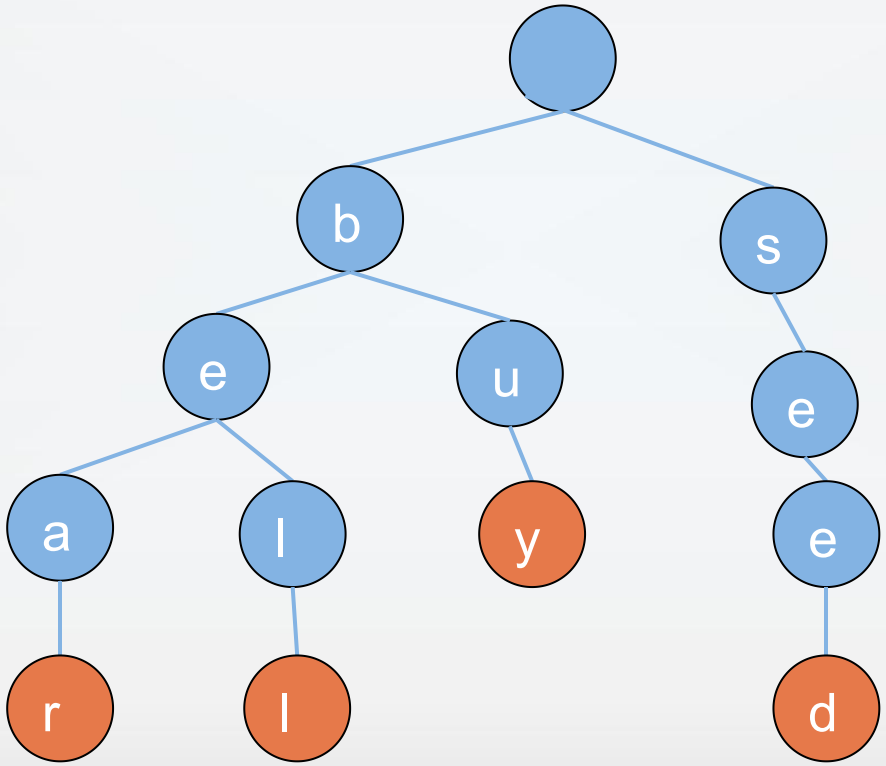
Trie树

- **查询：** 将待查找字符串分割成单个的字符，然后从Trie树的根结点开始逐个字符匹配。如图所示，红色的路径就是在Trie树中匹配的路径。



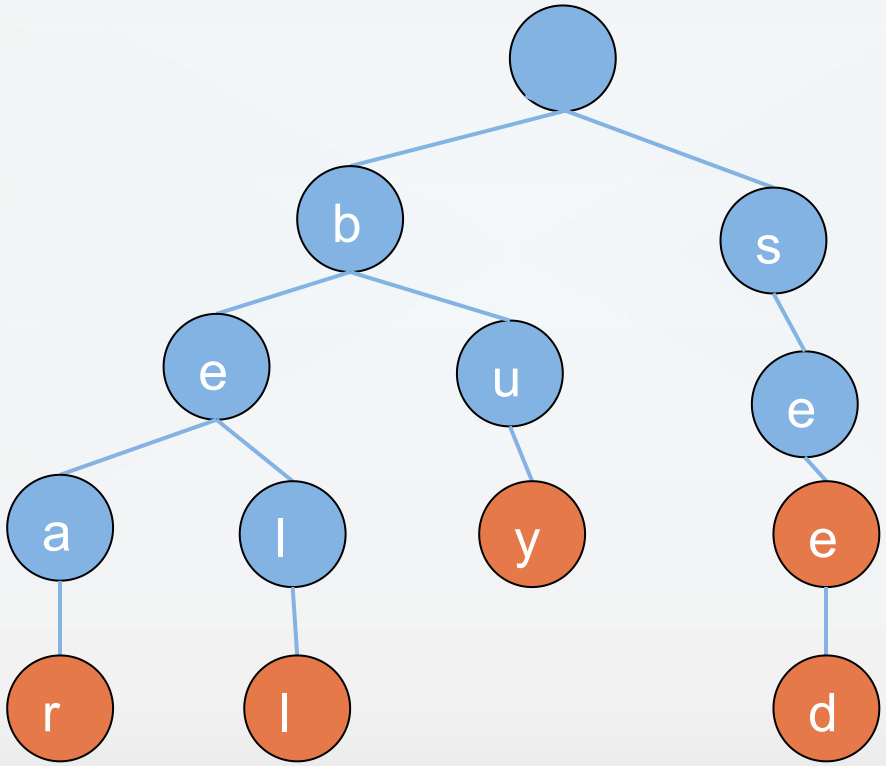
Trie树

- **插入：** 往Trie树中插入一个字符串，例如，插入单词buy。



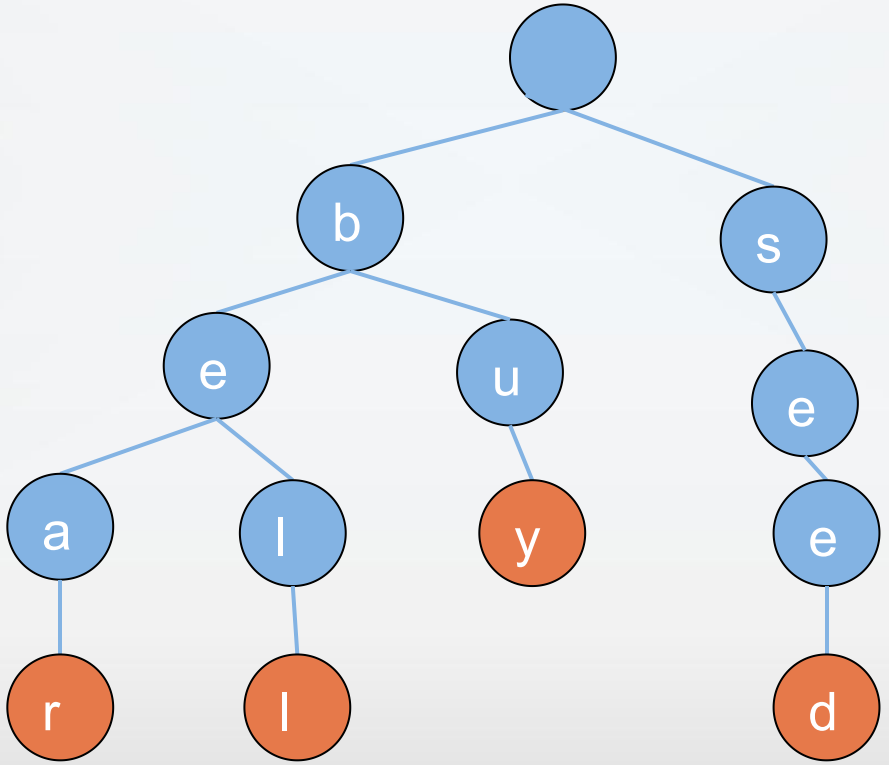
Trie树

- **插入：** 往Trie树中插入一个字符串，例如，插入单词see。



Trie树

- **创建：** 在构造过程中的每一步，都相当于往Trie树中插入一个字符串，直到所有字符串都插入完成。例如，依次插入单词bear, seed, buy, bell。

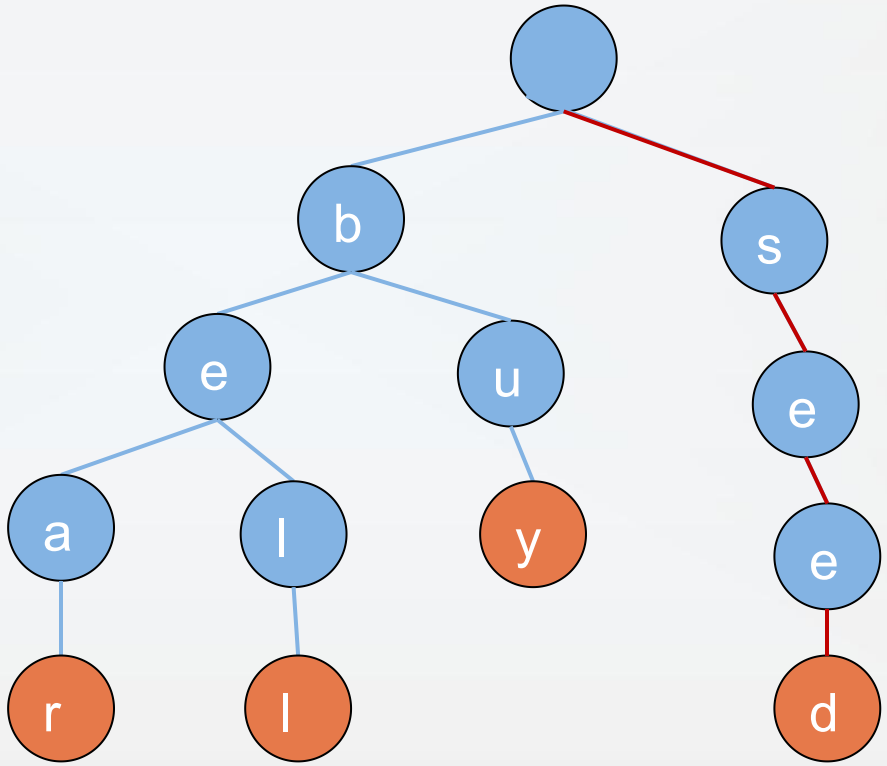


Trie树

- 删除

1. 删除整个单词

例如，删除seed



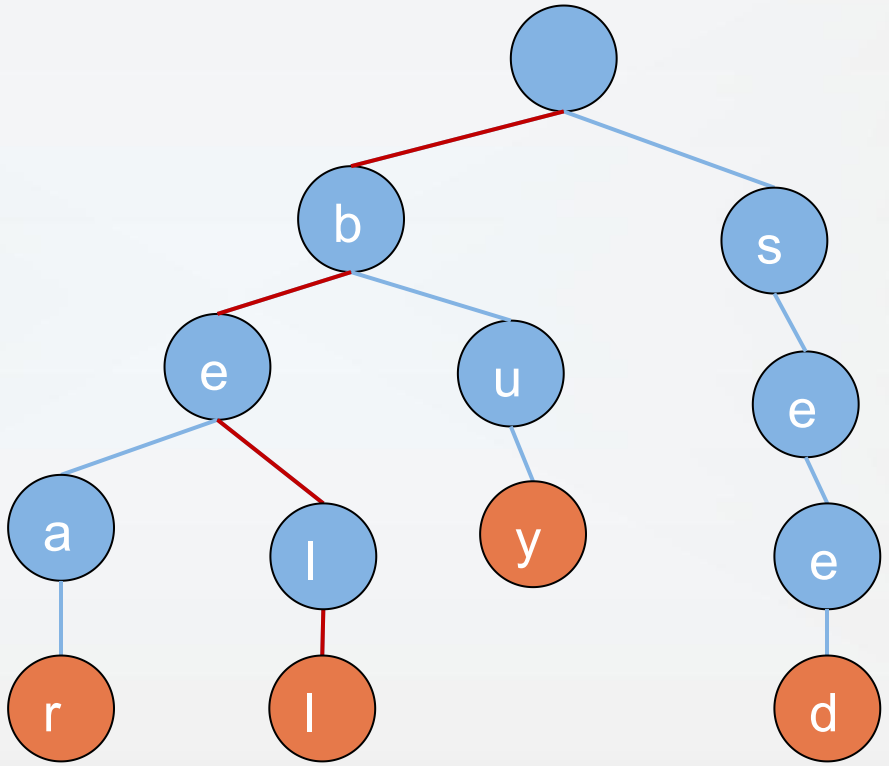
Trie树

- 删除

2. 删除分支单词

例如，删除bell

※ 需保留公共前缀



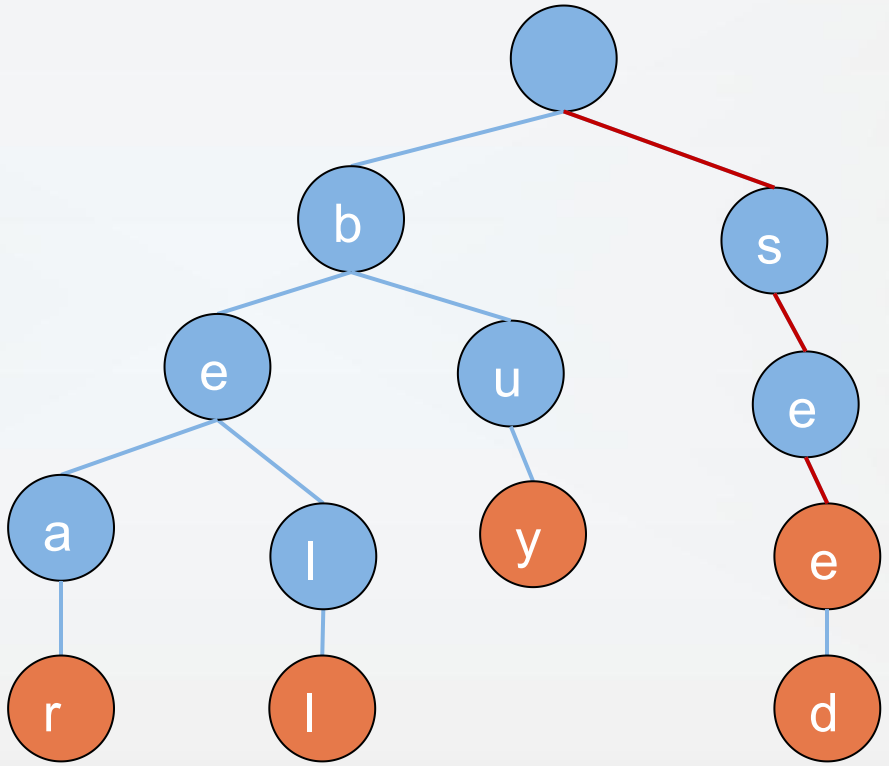
Trie树

- 删除

3. 删除前缀单词

例如，删除单词see

※ 需修改字符标记



Trie树

- 存储表示

Trie树的结点实现：

```
#define SIZE 26 //字符集大小
typedef struct TrieNode {
    char val;  //当前结点的值
    TrieNode* son[ SIZE ];    //子结点数组
    bool bWordEnd;   //是否是单词结束结点
    int num;   //有多少单词含有当前字符
} TrieNode , *TrieTree;
```

或

```
typedef struct TrieNode {
    char val;  //当前结点的值
    TrieNode* son, *brother;  //孩子兄弟表示法
    bool bWordEnd;   //是否是单词结束结点
    int num;   //有多少单词含有当前字符
} TrieNode , *TrieTree;
```

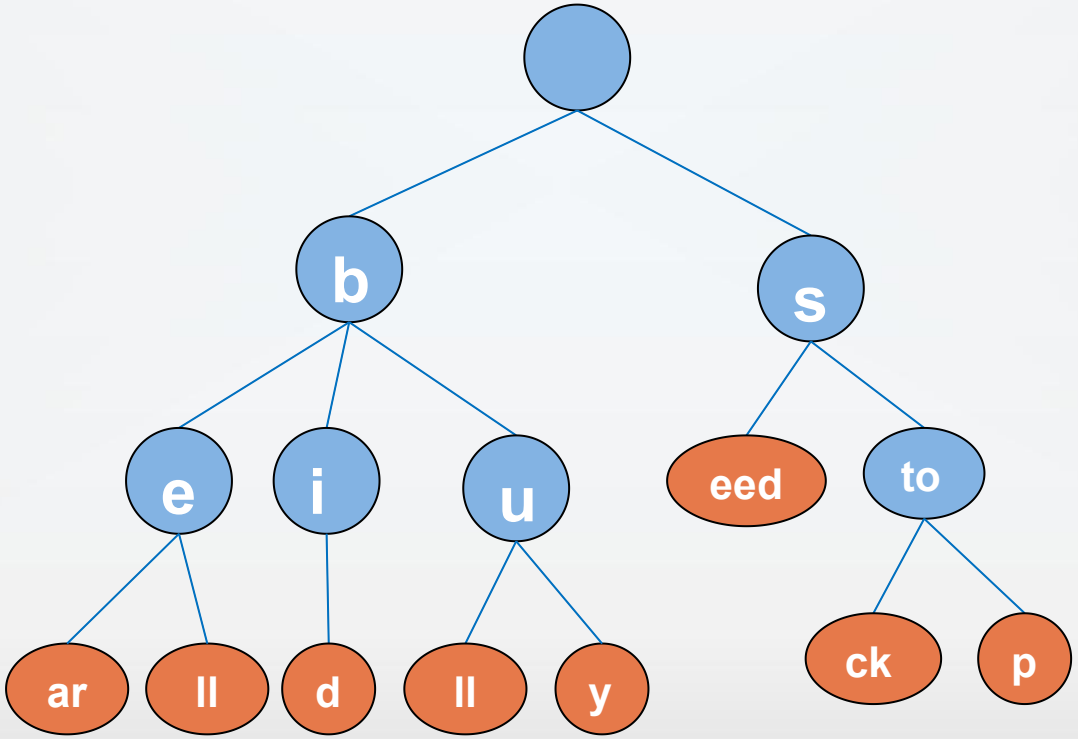
Trie树

- 性能分析
  - **时间复杂度：** 假设所有字符串长度之和为n，构建字典树的时间复杂度为O(n)；假设要查找的字符串长度为k，查找的时间复杂度为O(k)。
  - **空间复杂度：** 字典树每个结点只存储一个字符，还要保存指向的子结点的指针，比较耗内存，空间复杂度较高。

“空间换时间”，利用字符串的公共前缀来降低查询时间的开销，以达到提高效率的目的。

Trie树

压缩字典树(Compressed Trie)，若子结点都是单支树，压缩为1个结点，可以优化一定的空间，但是增加维护成本。



Trie树

- 应用

1. 字符串检索（前缀匹配）
2. 文本预测、拼写检查
3. 词频统计
4. 排序
5. 作为其他数据结构和算法的辅助结构

Trie树

- 扩展

1. 后缀树
2. 三分字典树
3. Radix Tree/Trie
4. PATRICIA tree
5. bitwise版本的crit-bit tree