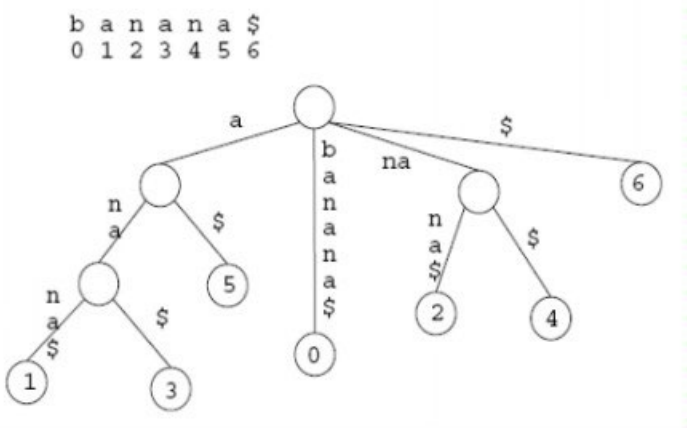


学号：202200400053	姓名：王宇涵	班级：2202
上机学时：4	实验日期：2024-04-03	
课程设计题目： 后缀树的构造		
软件开发环境： Clion 2023.1.1		
报告内容： <div>           1. 需求描述           <div>             1.1 问题描述             <p>后缀树是一种数据结构，一个具有 <math>m</math> 个字符的字符串 <math>S</math> 的后缀树 <math>T</math>，就是一个包含一个根节点的有向树，该树恰好带有 <math>m+1</math> 个叶子(包含空字符)，这些叶子被赋予从 0 到 <math>m</math> 的标号。每一个内部节点，除了根节点以外，都至少有两个子节点，而且每条边都用 <math>S</math> 的一个子串来标识。出自同一节点的任意两条边的标识不会以相同的字符开始。后缀树的关键特征是：对于任何叶子 <math>i</math>，从根节点到该叶子所经历的边的所有标识串联起来后恰好拼出 <math>S</math> 的从 <math>i</math> 位置开始的后缀，即 <math>S[i, \dots, m]</math>。后缀树的图示：</p>  </div> </div>		
1.2 基本要求 <div>           1)对任意给定的字符串 <math>S</math>，建立其后缀树；           2)查找一个字符串 <math>S</math> 是否包含子串 <math>T</math>；           3)统计 <math>S</math> 中出现 <math>T</math> 的次数；           4)找出 <math>S</math> 中最长的重复子串。所谓重复子串是指出现了两次以上的子串；           5)分析以上各个算法的时间复杂性。           6)应用后缀树，查找两个字符串 <math>Q</math> 和 <math>R</math> 中最长的共有子串。分析时间复杂性并通过实验结果验证。         </div>		
1.3 输入说明 <div>           输入界面设计         </div>		

```
请输入命令：
--》输入create,创建新后缀树
--》输入find,进行后缀树中字符串的查询
--》输入count,进行后缀树中字符串的查询
--》输入maxsub,进行最长重复子串的查询
--》输入maxpub,进行最长公共子串的查询
--》输入exit,退出
请输入命令：
```

### 输入样例

```
Create abcabx
Find ab
Find e
Count ab
Count abx
Count e
Maxsub
Maxpub abcb
Maxpub abe
Maxpub p
```

## 1.4 输出说明

### 输出界面设计

对于 create 和 maxpub 输出对应树的结构，末尾编号为对应的字符串位置  
否则输出结果.

### 输出样例

```
请输入需要创建后缀树的字符串:abcbax
[root]
|---ab
|   |---cabx$   ||0
|   |---x$      ||3
|---b
|   |---cabx$   ||1
|   |---x$      ||4
|---cabx$      ||2
|---x$         ||5
|---$          ||6
```

```
请输入命令:find
请输入想要查找的字符串:ab
存在子串
```

```
请输入命令:find
请输入想要查找的字符串:e
不存在子串
```

```
请输入命令:count
请输入想要查找的字符串:ab
子串重复个数2
请输入命令:count
请输入想要查找的字符串:abx
子串重复个数1
请输入命令:count
请输入想要查找的字符串:e
不存在此子串
```

```
请输入命令:maxsub
最长重复子串ab
```

```
请输入命令:maxpub
请输入另一个字符串:abe
ab
```

```
请输入命令:maxpub
请输入另一个字符串:p
不存在公共字符串
```

## 2. 分析与设计

### 2.1 问题分析

1)对任意给定的字符串  $S$ ，建立其后缀树；

答：利用 ukkonen 算法，将时间复杂度优化为  $O(n)$

2)查找一个字符串  $S$  是否包含子串  $T$ ；

答：遍历树，查找某个结点的字符串是否为  $T$ ，若有该结点，包含  $T$ ，若未找到字符串  $T$  则不包含。

3)统计  $S$  中出现  $T$  的次数；

答：找到某个结点的字符串为  $T$ ，它的孩子结点的叶结点个数为  $S$  中出现  $T$  的次数。

4)找出  $S$  中最长的重复子串。所谓重复子串是指出现了两次以上的子串；

答：找到最深的有孩子结点的结点，其字符串就是  $S$  中最长的重复子串

5)分析以上各个算法的时间复杂性。

答：设  $n$  为字符串长度。

建立后缀树  $O(n)$ ；查找  $O(n)$ ；统计  $O(n)$ ；最长重复子串  $O(n)$ 。

6)应用后缀树，查找两个字符串  $Q$  和  $R$  中最长的共有子串。分析时间复杂性并通过实验结果验证。

答：时间复杂度  $O(n+m)$ ， $n$  和  $m$  分别为  $Q$  和  $R$  的字符串长度。验证方法：固定一个字符串长度  $n$ ，按正比增大另一个字符串长度  $m$ ，统计函数运行时间判断是否和  $m$  大小成正比。

### 2.2 主程序设计

我们设计后缀树分为 `suffixTree.h`, `suffixTree.cpp`, `main.cpp` 三个文件，分别定义了后缀树的类，函数具体实现，主函数，先通过 `main()` 函数验证后缀树的基本功能，再通过测试函数 `testMaxSub()` 中动态增大  $m$  的大小来分析验证函数的时间复杂度。

### 2.3 设计思路

主体思路是先实现基础功能，再通过给定的样例进行测试功能的正确性，最终自己实现动态序列，调整输入数据的规模，来观察后缀树 `maxSub()` 函数性能的变化规律。

### 2.4 数据及数据类型定义

`SuffixNode` 类

**Child:** 指向该节点的左孩子（子节点）。

**Brother:** 指向该节点的右兄弟（右孩子）。

**suffixNode:** 后缀链接，指向当前节点的后缀节点。

**str:** 节点上对应的字符串。

**iiflag:** 叶子节点：对应字符串开始位置；其它情况：标识子节点中是否有 \$、#。

此外，该结构还定义了以下方法：

**init(string str):** 用于初始化节点的方法，传入一个字符串参数进行初始化。

**print(string s):** 打印函数，用于打印节点的信息以及以此节点为根节点的整棵子树的结构。

### *SuffixTree 类*

**后缀树节点 (SuffixNode):** 这是后缀树的节点类，用于构建后缀树的各个节点。它可能包含指向子节点的指针、字符等信息。

**活动点 (ActiveNode):** 用于表示后缀树的构建过程中的当前活动状态。

**构建后缀树:** 通过 **create(string str)** 函数构建后缀树，传入一个字符串作为参数。

**节点分裂:** **Split(string str, int currentIndex, SuffixNode \*prefixNode)** 函数用于处理剩余的等待插入的后缀，可能涉及到节点的分裂操作。

**查找字符:** **find(char w)** 函数用于寻找当前活动点子节点中是否包括后缀字符的节点（边）。

**查找子串:** **search(string sentence)** 函数用于查找给定字符串是否为其子串。

**打印后缀树:** **print()** 函数用于打印后缀树。

**计数:** **count(string subsen)** 函数用于查找字符串中出现字符串 **subsen** 的次数。

**最长重复子串:** **maxsub(SuffixNode\* sn)** 函数用于寻找最长的重复子串。

**最长公共子串:** **maxpub(string s)** 函数用于寻找同 **s** 的最长公共子串。

**两个子串中最长公共子串:** **searchpub(SuffixNode\* sn)** 函数用于寻找两个子串中最长公共子串。

**获取叶子节点数量:** **getleaf(SuffixNode\* sn)** 函数用于得到某节点的叶子数。

**根节点 (root):** 后缀树的根节点。

**剩余后缀数 (remainder):** 需要插入多少后缀。

**标志 (iflag):** 记录当前后缀树的节点。

**后缀树表示的字符串 (treeword):** 后缀树所代表的字符串。

## 2.5. 算法设计及分析

### *详细介绍两个主要的构造后缀树函数*

#### **后缀树的构造函数 void create(string s)**

1. 从字符串的第一个字符开始，逐个向后处理，直到处理完整个字符串。
2. 对于每个字符，首先检查当前活动点的子节点中是否存在包含该字符的节点（即查找函数 **find()** 的调用），如果存在，则将剩余后缀数 **remainder** 加一，并结束当前循环，继续处理下一个字符。
3. 如果不存在包含该字符的节点，且当前活动边为空，并且剩余后缀数不为零，则需要分割操作。具体步骤为：
  - 创建一个新节点，将当前字符及其后面的字符作为该节点的字符串内容。
  - 将该新节点插入到当前活动节点的子节点中。
  - 根据规则 3，更新活动点为当前活动节点的后缀节点，活动边和长度置空。
  - 调用 **Split()** 函数继续处理剩余后缀。
4. 如果不存在包含该字符的节点，并且剩余后缀数为零，则直接在当前活动节点插入一个新节点。
5. 如果不存在包含该字符的节点，但剩余后缀数大于零，则需要分割操作。具体步骤为：
  - 根据当前活动边进行分割，创建新节点，并将剩余的字符作为其字符串内容。
  - 将新节点插入到分割出来的节点的子节点中。

- 根据规则 1 和规则 3，更新活动点为根节点或其后缀节点，活动边和长度置空。
- 调用 `Split()` 函数继续处理剩余后缀。

#### 结点分裂函数 **void Split(string str,int currentIndex,SuffixNode \*prefixNode)**

1. 首先计算剩余待插入的后缀的起始位置，以确定需要分割的字符范围。
2. 从起始位置开始，逐个查找后缀字符是否已经存在于后缀树中。如果存在，则继续查找下一个字符。
3. 如果某个字符不存在，则需要进行分割操作：
  - 如果当前活动边为空，则直接在当前活动节点下插入一个新节点，新节点的内容为剩余的后缀字符串。
  - 否则，进行分割操作：
    - 创建一个新节点，将待分割节点的后缀部分作为新节点的内容。
    - 将新节点插入到待分割节点的子节点中。
    - 创建另一个新节点，将剩余的后缀字符串作为其内容。
    - 将另一个新节点插入到刚才分割出来的节点的子节点的兄弟节点中。
    - 更新待分割节点的字符串内容。
    - 根据规则 2，连接分割节点与其后缀节点。
4. 更新剩余后缀数 `remainder`，表示成功插入一个后缀。
5. 根据规则 1 和规则 3，更新活动点、活动边和活动边长度。
6. 如果剩余后缀数仍大于零，则递归调用 `Split` 函数处理剩余的后缀。

#### 简略介绍功能函数

##### 查找函数 **bool find(char w)**

思想：该函数实现了在后缀树中查找字符 `w` 的功能。它通过遍历当前活动点 `activenode->point` 的子节点，查找是否存在字符 `w`，并根据查找结果更新活动边和活动边长度。

时间复杂度：在最坏情况下，需要遍历当前活动点的所有子节点才能确定字符是否存在。因此时间复杂度为 `O(m)`，其中 `m` 是当前活动点的子节点数量。

##### 判断子串是否出现函数 **bool search(string sentence)**

思想：该函数实现了在后缀树中搜索字符串 `sentence` 的功能。它通过遍历后缀树，逐字符匹配输入的字符串 `sentence`，如果能够匹配成功直到结束，则返回 `true`，否则返回 `false`。

时间复杂度：在最坏情况下，需要遍历整个待搜索的字符串 `sentence` 以及后缀树的路径。因此时间复杂度为 `O(n)`，其中 `n` 是待搜索字符串的长度。

##### 统计子串出现次数函数 **int count(string subsen)**

思想：该函数实现了统计子串 `subsen` 在原始字符串中出现的次数的功能。它通过在后缀树中查找子串 `subsen` 的起始位置，然后计算以这些位置为根的子树中叶子节点的个数，即为子串出现的次数。

时间复杂度：在最坏情况下，需要遍历整个待搜索的子串 `subsen` 以及后缀树的路径，并且需要遍历每个匹配到的位置的子树来统计叶子节点个数。因此时间复杂度为 `O(n)`，其中 `n` 是原始字符串的长度。

##### 求解最长重复子串函数 **string maxsub(SuffixNode\* sn)**

思想：该函数实现了求解后缀树中最长重复子串的功能。它通过深度优先搜索后缀树，找到具有相同字符串前缀的节点，并根据这些节点的深度得到最长重复子串。

时间复杂度：在最坏情况下，需要遍历整个后缀树来查找最长重复子串。因此时间复杂度为 `O(n)`，其中 `n` 是原始字符串的长度。

##### 求解最长公共子串函数 **string maxpub(string s)**

思想：该函数实现了求解后缀树中与给定字符串 `s` 具有最长公共前缀的子串的功能。它通过比较后缀树中每个节点的字符串与给定字符串 `s` 的公共前缀，找到最长的公共前缀并返回。

时间复杂度：在最坏情况下，需要遍历整个后缀树来查找具有最长公共前缀的子串。因此时间复杂度为

$O(n)$ ，其中  $n$  是后缀树的节点数，即  $O(m+n)$ ；

### 搜寻最长公共子串函数 `string searchpub(SuffixNode* sn)`

思想：该函数实现了在后缀树中搜索与给定节点 `sn` 具有最长公共前缀的子串的功能。它通过比较给定节点 `sn` 的后缀链接以及其父节点的字符串，找到最长的公共前缀并返回。

时间复杂度：由于后缀树中的每个节点都具有后缀链接，因此在最坏情况下，需要遍历从给定节点 `sn` 开始一直追溯到根节点的路径来查找具有最长公共前缀的子串。因此时间复杂度为  $O(h)$ ，其中  $h$  是后缀树中从给定节点 `sn` 到根节点的路径长度，即  $O(m+n)$ 。

### 3. 测试

1. 基础功能测试：已在输入实例和输出实例进行演示，最终得到了正确的结果。

2. `maxPub` 函数时间复杂度  $O(m+n)$  验证：

固定字符串 1 长度  $n$  不变，字符串 2 的长度  $m$  进行规律变化，探究平均运行时间和  $m$  增长的关系。

```
newWord的长度为：900；maxpub函数总执行时间为：4ms；
newWord的长度为：1200；maxpub函数总执行时间为：4ms；
newWord的长度为：1500；maxpub函数总执行时间为：6ms；
newWord的长度为：1800；maxpub函数总执行时间为：8ms；
newWord的长度为：2100；maxpub函数总执行时间为：12ms；
newWord的长度为：2400；maxpub函数总执行时间为：15ms；
newWord的长度为：2700；maxpub函数总执行时间为：19ms；
newWord的长度为：3000；maxpub函数总执行时间为：25ms；
newWord的长度为：3300；maxpub函数总执行时间为：31ms；
newWord的长度为：3600；maxpub函数总执行时间为：34ms；
newWord的长度为：3900；maxpub函数总执行时间为：40ms；
```

固定字符串 2 长度  $m$  不变，字符串 1 的长度  $n$  进行规律变化，探究平均运行时间和  $n$  增长的关系。

```
Word的长度为：300；maxpub函数总执行时间为：1ms；
Word的长度为：600；maxpub函数总执行时间为：2ms；
Word的长度为：900；maxpub函数总执行时间为：2ms；
Word的长度为：1200；maxpub函数总执行时间为：4ms；
Word的长度为：1500；maxpub函数总执行时间为：6ms；
Word的长度为：1800；maxpub函数总执行时间为：10ms；
Word的长度为：2100；maxpub函数总执行时间为：12ms；
Word的长度为：2400；maxpub函数总执行时间为：17ms；
Word的长度为：2700；maxpub函数总执行时间为：21ms；
Word的长度为：3000；maxpub函数总执行时间为：26ms；
Word的长度为：3300；maxpub函数总执行时间为：33ms；
Word的长度为：3600；maxpub函数总执行时间为：38ms；
Word的长度为：3900；maxpub函数总执行时间为：43ms；
Word的长度为：4200；maxpub函数总执行时间为：49ms；
```

可以看出，总执行时间随着  $m$  的增大总体呈正比趋势，总执行时间随着  $n$  的增大总体呈正比趋势，因此可以验证时间复杂度为  $O(m+n)$ 。

### 4. 分析与探讨

本次实验我通过学习后缀树的数据结构，了解它巧妙的构造方法 `ukkonen` 和高效处理的几种字符串问题方法，并通过代码实现，成功完成了基础功能的测试。

此外，我也成功通过自己生成动态的字符串，改变输入数据的规模，成功简洁地完成了查找两字符串最长公共子串的性能测量与分析。

### 5. 附录：实现源代码



## 后缀树的构造函数

```
void SuffixTree::create(string str){//对字符串 str 构造后缀树
    int index = 0;//
    treeword=str; // 整颗树表示的单词
    while (index < str.length()){// 循环建立后缀
        int currentIndex = index++;// 保存当前的位置
        char w =str[currentIndex];// 得到当前的后缀字符 abcabcx 依此为 a b c a b c x
        bool f=find(w); // 查找这棵树中是否有
        //存在,则 remainder++即可
        if (f){// 查找是否存在保存有当前后缀字符的节点
            // f 函数已经完成了对于活动点的修改
            remainder++;// 存在, 则将 remainder+1, activenode->length+1, 结束本次循环
            continue;//跳过本次循环, 即找到一个相同, 在 find 方法内利用活动点记录一次, 暂不处
            理
        }

        //此时 index=null, 且需要插入额外后缀 (remainder=0)
        else if (!f&&!activenode->index&&remainder != 0) {
            SuffixNode *child = activenode->point->Child;
            SuffixNode *splitNode = activenode->point; // 分裂点为 point
            SuffixNode *newNode = new SuffixNode();//即将插入的节点
            string s = str.substr(currentIndex, str.length() - currentIndex);//idx~末尾
            newNode->init(s);
            newNode->iiflag = iflag++;//newNode 为新建立的叶子节点, 对应有字符串起始位置
            //遍历活动节点孩子的兄弟节点, 插到最后
            while (child->Brother != NULL) {
                child = child->Brother;
            }
            child->Brother = newNode;

            //以上为一次分割过程。
            //插入了 abx
            // 在非 root 节点上进行操作, 分割完成之后需根据规则 3 对待
            if (NULL == activenode->point->suffixNode){// 无后缀节点, 则活动节点变为 root
                activenode->point = root;
            } else{// 否则活动节点变为当前活动节点的后缀节点
                activenode->point = activenode->point->suffixNode;
            }
            activenode->index = NULL;
            activenode->length = 0;
            // abx
            Split(str, currentIndex, splitNode);//eg:需要插入 abx,bx,x,则处理完 abx, 后需要继续处理 bx,
            进行分裂操作
            continue;
        }
        // 找不到而且 remainder==0 表示之前在该字符之前未剩余有其他待插入的后缀字符, 所以直接
```

插入该后缀字符即可

```
else if (!f&&remainder == 0) { // abcabxabcd  bcabxabcd cabxabcd
    // 直接在当前活动节点插入一个节点即可,
    SuffixNode *node = new SuffixNode();
    string tmp=str.substr(currentIndex,str.length()-currentIndex); // idx~末尾的字符
    // 如果当前活动点无子节点,则将新建的节点作为其子节点即可,否则循环遍历子节点(通过兄弟节点进行保存)
    node->init(tmp);
    node->iiflag=iiflag++;
    SuffixNode* child = activenode->point->Child; //插入 point 的最后一个孩子
    if (NULL== child) {
        activenode->point->Child = node;
    } else {
        while (NULL!= child->Brother) {
            child = child->Brother;
        }
        child->Brother = node;
    }
}
else {
    // 如果 remainder>0,则说明该字符之前存在剩余字符,需要进行分割,然后插入新的后缀
    SuffixNode *splitNode = activenode->index;// 待分割的节点即为活动边(active_edge)
    // 创建切分后的节点,放到当前节点的子节点
    // 该节点继承了当前节点的子节点以及后缀节点信息
    SuffixNode *node = new SuffixNode();
    // abcabx 分为 ab cabx
    // 从活动边长度开始截取剩余字符作为子节点,这里从活动边截取
    string tmp=splitNode->str.substr(activenode->length,splitNode->str.length()-activenode->length); // length 到末尾的字符
    //复制 splitNode 的信息
    node->init(tmp);
    node->iiflag=splitNode->iiflag;
    splitNode->iiflag=-1;
    node->Child = splitNode->Child;
    node->suffixNode = splitNode->suffixNode;//后缀节点转移

    splitNode->Child = node;
    splitNode->suffixNode = NULL;

    // 创建新插入的节点,放到当前节点的子节点(通过子节点的兄弟节点保存)
    SuffixNode *newNode = new SuffixNode();// 插入新的后缀字符
    string tmp2=str.substr(currentIndex,str.length()-currentIndex); // 需要插入的结点
    newNode->init(tmp2);
    newNode->iiflag=iiflag++;
    splitNode->Child->Brother = newNode;
```



```

        splitNode->str = splitNode->str.substr(0,activenode->length);// 修改当前节点的字符为
0~length 的字符

        //以上为一次分割过程。
        // 分割完成之后需根据规则 1 和规则 3 进行区分对待
        // 按照规则 1 进行处理
        if (root == activenode->point) { // 活动节点是根节点的情况
            activenode->point == root; //活动节点保留为根节点
            //activenode->index    root,a,2->root,b,1
            // 按照规则 3 进行处理
        }
        //
        activenode->point = root;

    }
    else if (NULL == activenode->point->suffixNode) { // 无后缀节点，则活动节点变为 root
        activenode->point = root;
    }
    else { // 否则活动节点变为当前活动节点的后缀节点
        activenode->point = activenode->point->suffixNode;
    }
    // 活动边和活动边长度都重置
    activenode->index = NULL;
    activenode->length = 0;
    // 递归处理剩余的待插入后缀
    Split(str, currentIndex, splitNode);
}
}
}

```

### 分割结点操作

void SuffixTree::Split(string str,int currentIndex,SuffixNode \*prefixNode){//相当于把分割部分单独写出来

// 此处计算剩余待插入的后缀的开始位置，例如剩余后缀数为 2，已处理过 abx，需要处理 bx 和 x 时，下面计算 b，相当于新分支节点

int start=currentIndex-remainder+1;//bx 中 b 的位置

// dealStart 表示本次插入进行查找的开始字符位置（规则二，请看代码首部）

int a=activenode->point->str.length();//区分后缀节点与 root，root 时，此值为 0

int b=activenode->length;//和之前 length 用法类似

int dealStart=a+b+start;//后缀节点，直接从子节点开始比对，非后缀节点，需要先找出拥有下一个插入字符为首字符的子节点

// 从 dealStart 开始查找所有后缀字符是否都存在(相对与活动点) // bx x

for (int index = dealStart; index <= currentIndex; index++) {

char w =str[index];

bool f=find(w);

//以下步骤和 create 方法中相似

if (f) { // 存在，则查找下一个，activePoint.length+1，这里不增加 reminder

continue;

```
}
```

```
SuffixNode *splitNode = NULL;// 被分割的节点
```

if(NULL==activenode->index){// 如果 activePoint.index==null, 说明没有找到活动边, 那么只需要在活动节点下插入一个节点即可

```
SuffixNode *node = new SuffixNode();
string tmp1=str.substr(index,str.length()-index);
node->init(tmp1);
node->iiflag=iiflag++;
SuffixNode *child = activenode->point->Child;
if(NULL==child){
    activenode->point->Child = node;
}else{
    while (NULL != child->Brother) {
        child = child->Brother;
    }
    child->Brother = node;
}
}
```

```
}
```

```
else{
```

```
// 开始分割, 分割部分同上面的分割
```

```
splitNode = activenode->index;
```

```
// 创建切分后的节点, 放到当前节点的子节点
```

```
// 该节点继承了当前节点的子节点以及后缀节点信息
```

```
SuffixNode *node = new SuffixNode();
```

```
string tmp2;
```

```
tmp2=splitNode->str.substr( activenode->length,splitNode->str.length()-activenode->length);
```

```
node->init(tmp2);
```

```
node->iiflag=splitNode->iiflag;//新分裂出的子节点得到原来的 iiflag 值
```

```
splitNode->iiflag=-1;//原本是叶子节点, 因分裂变成非叶子节点, iiflag 置-1
```

```
node->Child = splitNode->Child;
```

```
node->suffixNode = splitNode->suffixNode;
```

```
splitNode->Child = node;
```

```
splitNode->suffixNode = NULL;
```

```
// 创建新插入的节点, 放到当前节点的子节点(通过子节点的兄弟节点保存)
```

```
SuffixNode *newNode = new SuffixNode();
```

```
string tmp3;
```

```
tmp3=str.substr(index,str.length()-index);
```

```
newNode->init(tmp3);
```

```
newNode->iiflag=iiflag++;//新叶子节点
```

```
splitNode->Child->Brother = newNode;
```

```
// 修改当前节点的字符数
```

```
string tmp4;
```

```
tmp4=splitNode->str.substr(0,activenode->length);
```

```
splitNode->str = tmp4;
```

```
// 规则 2, 连接后缀节点
```

```

        prefixNode->suffixNode = splitNode;
    }
    remainder--; // 插入成功, 进行更新

    // 按照规则 1 进行处理
    if (root == activenode->point) { // 活动节点是根节点的情况
        // activenode->point == root
        // 按照规则 3 进行处理
    } else if (NULL == activenode->point->suffixNode) { // 无后缀节点, 则活动节点变为 root
        activenode->point = root;

    } else {
        activenode->point = activenode->point->suffixNode;
    }

    activenode->index = NULL;
    activenode->length = 0;
    if (remainder > 0) { // 如果 remainder == 0 则不需要继续递归插入后缀, 插入已完成
        Split(str, currentIndex, splitNode); // 递归处理即将插入的后缀
    }
}
}

```

#### 查找字符是否出现

```

bool SuffixTree::find(char w){
    const SuffixNode* start=activenode->point; // 在活动点处向后寻找。
    SuffixNode* current=activenode->index;
    bool exist=false; // 是否存在, 存在的话, 用三元组活动边、活动点等存位置
    if (current==NULL) { // current 为空, 说明无活动边, 则在其子节点中进行查找, 如 root, null, 1
        // 寻找遍历子节点, 只找一层
        SuffixNode* child=start->Child;
        while (child!=NULL) {
            if (child->str[0]==w) // 存在的话
            {
                activenode->index=child; // 在无活动边情况下 eg: root, 0, 0->root, a, 1
                activenode->length++; //
                exist=true;
                break;
            }
            else {
                child=child->Brother; // 得以遍历所有的孩子
            }
        }
    }
    // 不需要改变活动边
    else if (activenode->length!=current->str.length() && current->str[activenode->length]==w) { // 有活动边且
        没有移到活动边末尾
    }
}

```

```

        activenode->length++;
        exist=true;
    }
    //需要改变活动边
    else if(activenode->length==current->str.length()){//有活动边，且此时坐标位置移到活动边末尾，且更换活动节点
        activenode->point=current;
        activenode->index=NULL;
        activenode->length=0;//更换活动点后，length 清 0
        // 遍历新活动点的所有孩子(一层)，判断有无对应的字符
        SuffixNode* grantchild=current->Child;
        while(grantchild!=NULL){
            if(grantchild->str[0]==w)//存在的话
            {
                activenode->index=grantchild;//有，则更新 index 和 length
                activenode->length++;//
                exist=true;
                break;
            }
            else{
                grantchild=grantchild->Brother;//得以遍历所有的孩子
            }
        }
    }
    else{ // 啥都没有，则不存在
        exist=false;
    }

    return exist;
}

```

### 查找子串是否出现

```

bool SuffixTree::search(string sentence){//O(查询的字符串长度)
    int index = 0;// 查找到的节点的匹配的位置
    // 查找从根节点开始，遍历子节点
    SuffixNode *start = root;
    for (int i = 0; i < sentence.length(); i++) {
        if (start->str.length() < index + 1) {// 如果当前节点已匹配完，则从子节点开始，同时需重置 index==0
            index = 0;
            start = start->Child;
            while (NULL != start) {
                // 比较当前节点指定位置(index)的字符是否与待查找字符一致
                // 由于是遍历子节点，所以如果不匹配换个子节点继续
                if (start->str[index] == sentence[i]) {
                    index++;
                }
            }
        }
    }
}

```

```

        break;
    }
    else {
        start = start->Brother;
    }
}
if (NULL== start) { // 子节点遍历完都无匹配则返回 false
    return false;
}
}
else if (start->str[index] == sentence[i]) {
    // 如果当前查找到的节点的还有可比较字符，则进行比较，如果不同则直接返回 false
    index++;
}
else {
    return false;
}
}
return true;
}

```

#### 统计子串出现次数

```

int SuffixTree::count(string subsen){ //遍历所有的节点，找到的时间+数的时间，节点数
    //经过证明，在最坏情况下，后缀树的节点数也不会超过 2N (N 为文本的长度). O(n)
    int index = 0; // 查找到的节点的匹配的位置
    int count=0; //记录相同子串的个数
    // 查找从根节点开始，遍历子节点
    SuffixNode *start = root;
    SuffixNode *child=NULL; //得到当前比对到的节点
    SuffixNode *tmp=NULL;
    for (int i = 0; i < subsen.length(); i++) {
        if (start->str.length() < index + 1) { // 如果当前节点已匹配完，则从子节点开始，同时需重置
            index==0
            index = 0;
            start = start->Child;
            while (NULL != start) {
                // 比较当前节点指定位置(index)的字符是否与待查找字符一致
                // 由于是遍历子节点，所以如果不匹配换个子节点继续
                if (start->str[index] == subsen[i]) {
                    child=start;
                    index++;
                    break;
                } else {
                    start = start->Brother;
                }
            }
        }
        if (NULL== start) { // 子节点遍历完都无匹配则返回 false

```

```

        return -1;
    }
}
else if (start->str[index] == subsen[i]) {
    // 如果当前查找到的节点的还有可比较字符，则进行比较，如果不同则直接返回 false
    child=start;
    index++;
} else {
    return -1;
}
}
if(child->Child==NULL)
    return ++count;
return getleaf(child);
}

```

#### 查找最长重复子串

```

string SuffixTree::maxsub(SuffixNode* sn){//最长重复子串 O(n)
    if(sn->Child==NULL)//自己为叶子
        return "";

    SuffixNode* tmp=sn->Child;
    int maxlength=0;//记录子串中最长的长度
    string maxstring="";//记录子串中最长的重复子串（最深的非叶节点）
    while(tmp!=NULL){
        string tmpstr=maxsub(tmp);//利用递归方法，求子串中最长的重复子串
        if(tmpstr.length()>maxlength){//如果遇到更长的，则进行更换
            maxlength=tmpstr.length();
            maxstring=tmpstr;
        }//只可以查 1 个
        tmp=tmp->Brother;
    }
    return sn->str+maxstring;
}

```

#### 查找最长公共子串

```

string SuffixTree::maxpub(string s){//O(m+n)构建
    string newword=treeword+s+"#";//形成 s=S$S#形式
    string tmp=treeword;
    root->init("");
    activenode->init(root,NULL,0);
    iflag=0;
    int remainder=0;
    create(newword);//构造新的后缀树
//    this->print();
    this->treeword=tmp;//搜索时需要用到，因此不能改变 treeword 的值
    return searchpub(this->root);
}

```

```
}
```

### 搜索最长公共子串

string SuffixTree::searchpub(SuffixNode\* sn){//局限，只能两个。合成新子串。考虑临近叶子子节点和不临近叶子子节点

//找到最深的且后缀有\$#和#的节点

```
if(sn->Child==NULL)
```

```
    return "";
```

```
else{
```

```
    int maxlength=0;
```

```
    string maxstring="";
```

```
    int sl=0;
```

```
    int jing=0;
```

```
    SuffixNode* tmp=sn->Child;
```

```
    while(tmp!=NULL){
```

```
        if(tmp->Child==NULL)//如果为叶子
```

```
            if(tmp->iiflag<=treeword.length()-1)
```

```
                sl++;
```

```
            if(tmp->iiflag>treeword.length()-1)
```

```
                jing++;
```

```
            if(tmp->iiflag==3)//不是叶子，子节点有#
```

```
                jing++;
```

```
            if(tmp->iiflag==2)//不是叶子，子节点有$
```

```
                sl++;
```

```
            string tmpstr=searchpub(tmp);
```

```
            if(tmpstr.length()>maxlength){
```

```
                maxlength=tmpstr.length();
```

```
                maxstring=tmpstr;
```

```
            }//只可以查 1
```

```
            tmp=tmp->Brother;
```

```
    }
```

```
    if(jing!=0&sl==0)//后缀有#
```

```
        sn->iiflag=-3;
```

```
    if(sl!=0&jing==0)//后缀有$
```

```
        sn->iiflag=-2;
```

```
    if(sl!=0&jing!=0|maxstring!="")
```

```
        return sn->str+maxstring;
```

```
    else
```

```
        return "";
```

```
}
```

```
}
```



