**（第一页）**

大家好,我是第十小组的XXX，本次由将我代表我们小组为大家展示我们的成果。

本次的课题是AVL树和字典树（trie）的实现。

**（第二页）**

下面是我们本次讲解的主要内容（看情况来确定某些重复的部分讲不讲）

**（第三页）**

本次课题的主要内容是AVL树和字典树的实现，以及两种数据结构的比较。从插入字符串的长度、数量、顺序、方式等多方面分析比较两种数据结构的复杂性并对其性质做出相应的评价。

**（第四页、第五页）**

程序主要的结构有AVL节点和TRIE节点：

AVL节点同我们所学习的AVL节点结构类似，所不同的是原来的整形变量被替换为现在的字符数组。其中字符数组所能容纳的字符最多为99个，即每个节点上的字符串最多为99位。

TRIE节点中主要包括edw和一个指针数组，其中edw为词尾标记，可用于判断以该点为结尾的字符串是否存，而指针数组分别指向26个节点，每个节点代表一个英文字母。

**（第六页、第七页、第八页）**

以下为关键代码：

这是AVL树插入的实现方法：

与由数字构成的AVL树方法相同，区别在于用strcmp函数比较字符串大小，即原先对整形数字的操作替换为对字符串的操作。

**（第九页、第十页、第十一页）**

AVL树删除的实现方法：（重点）

每个节点的删除同搜索二叉树的删除方法相同，每次的删除通过递归方法将被删除节点不断传递，最终传递到叶节点。

具体：

若树中只有一个根节点，直接删除。

若该节点只有左儿子，则将左儿子赋值给该节点后删除左儿子。

若该节点只有右儿子，则将右儿子赋值给该节点后删除右儿子。

当某个节点被删除后，我们找到其右子树中的最小字符串赋值到该节点，并在右子树中删除最小的字符串，这样最终可以将被删除的节点传递到一个叶节点。叶节点的删除同上述的几种情况。在叶节点被删除后更新节点高度，若打破AVL平衡，则进行对应的旋转操作。

具体旋转操作以因删除左子树中字符串而造成AVL平衡被破坏为例。若该节点右子树的左子树高度比该节点右子树的右子树高度要高，则进行双旋转，否则进行单旋转（具体说明可通过代码展示和画图进行说明）。（第九页、第十页）

字典树（trie）的实现：

**（第十二页）**

1.由于在具体实现过程中，trie节点结构中的指针数组中每一个成员代表一个英文字母，必须将每个字母转换成对应的数字进行操作，从0到25分别代表a到z，不区分大小写。用一个简单wordtonumber函数就可以实现这一转换。

**（第十三页）**

2.字典树的插入

通过wordtonumber函数将字符串转换后，每读入一个字母，就在相应的字母的路径上进行查找，如果该节点已经存在，则继续向下寻找，否则申请动态内存分配开辟新的节点，直到被插入字符串的所有字符都已经搜索完毕，在最后一个字母的节点中将词尾标志eow的值置为1.

**（第十四页）**

3.字典树的查找

方法和插入几乎相同，若在查找过程中遇到空指针，则返回NULL，若所有的字母都已找到，则只需确定词尾eow的值是否唯一，以确定该字符串是否存在。若存在返回地址。

**（第十五页）**

4.字典树的删除

只需将相应的词尾eow值置为0即可。

**（第十六页~第二十三页）**

下面是我们的测试结果：

图中蓝色线条代表AVL树运行时间，红色代表字典树运行时间。

**（16页）**第一张图为增序输入字符串并以相同顺序删除的运行结果。

**（17页）**第二张图为增序输入字符串并以逆序删除的运行结果。

**（18页）**第三张图为随机顺序输入字符串并以随机顺序删除的运行结果。

**（19~21页）**第四到第六张图分别为三种情况下两种树的运行时间和字符串长度的关系。

**（22页）**从图中可以看出，显然AVL树运行速度更快。

另外，从后三图可以看出，字符串长度对AVL树的操作时间影响几乎为0.

根据拟合图像可以看出，相同字符串长度时，对两种树进行N次插入操作的复杂度为O(NlogN),而对于字典树，插入字符串复杂度还正比于字符串长度，即O(N)

**（第二十四页）**

下面我们来分享下研究过程中小组成员的思考过程，遇到的困难和一些心得：

过程中遇到的问题：

1. 首先是trie删除的方法，我们主要想到了惰性删除和count删除方法，那么他们各自有什么优缺点呢

**（第二十五页）**

1. 惰性删除

这是本小组采用的删除方法，显然，这种方法实现十分容易（从代码的行数就可以看出，才7行）

另外时间复杂度低，只需将词尾的标记eow置为0就可删除一个词。

最重要的是实际应用中删除一些词后，这些节点仍旧保留，再次插入拥有相同前缀字母的字符串是，这些节点不需要再次申请分配内存，大大降低了时间复杂度。

但这个优点同时也是缺点，即一个节点一旦被插入，就永远无法彻底删除，这样就会造成一些无用的节点一直占用内存，大大增加了空间复杂度。

**（第二十六页）**

1. count删除

count删除原理主要是给每个节点的结构内增加一个count变量，以此来监测字典书中该节点被多少字符串使用。每次删除-1，插入+1.一旦该节点的count值等于0，就释放内存。

一个节点占用的内存远大于一个count变量占用的内存，理论上来讲是可以节省内存的。

但这种方法的劣势就是前一种方法的优势。首先算法实现复杂，并且每次更新count值需要O(N)操作。另外再次插入带有相同字母前缀的字符串又要重新分配内存，相当于是一种以牺牲时间复杂度为代价来降低空间复杂度的方法。

另外，在实际应用中这种算法的价值有待商榷。我们不清楚一个节点被彻底删除的概率大小，也就无法确定该算法能为我们降低多少空间复杂度了。

因此，基于前一种方法实现比较简单，我们选择了惰性删除方法。

下面是过程中我们遇到的一些问题：

1. **（27页）**测试文档会出现重复的字符串，虽然测试要求要用不同的字符串，不过重复的字符串大多数是一位或者两位的，且占总数字符串的量极小，对测试结果影响不大，可以忽略。
2. **（28页）**可能会遇到指针内存访问失败，同样的测试用例前后测试结果不一的情况。本小组调试了很久，最后发现居然是旋转函数中的height写错了。虽然是一个比较低级的错误，但是正因为没有想到会在这种地方犯错，我们没有去关注，希望大家引起境界。另外遇到同样的运行错误可以参考我们的经历。
3. **（29页）**Trie插入失败，除第一次外其余测试结果运行时间为0。

因为trie的根节点是不带任何字符串的，如果在删除一棵字典树的时候将根节点也一起删除了，再次测试的时候一定要声明，否则就会出现上述错误。

1. 第一次测试后无法正常测试，没有rewind

文档已经读完，继续测试会没有数据，需将文档指针重置。