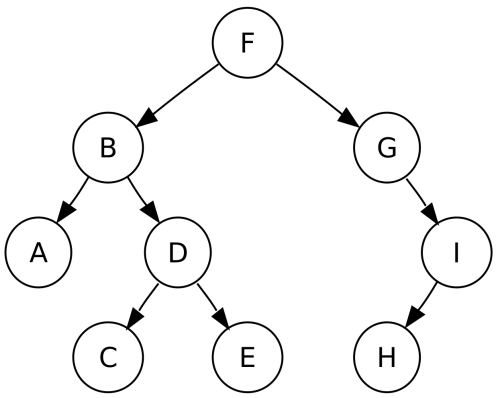
第二次研讨课作业

Author: 林日中(1951112), Date: 11/28/2020

1．二叉排序树

> 它或者是一棵空树；或者是具有下列性质的二叉树：
>
> - 若左子树不空，则左子树上所有结点的值均小于它的根结点的值
> - 若右子树不空，则右子树上所有结点的值均大于它的根结点的值
> - 左、右子树也分别为二叉排序树
>
> 请思考：
>
> （1） 请给出一棵二叉排序树。
> （2） 如何利用该二叉排序树得到一个递增序列?
> （3） 如何利用该二叉排序树得到一个递减序列?

（1） 请给出一棵二叉排序树。



（2） 如何利用该二叉排序树得到一个递增序列?

中序遍历。此二叉排序树的中序遍历序列为 `ABCDEFGHI`，即为该树对应的递增序列。

（3） 如何利用该二叉排序树得到一个递减序列?

逆中序遍历。此二叉排序树的逆中序遍历序列为 `IHGFEDCBA`，即为该树对应的递减序列。

在实际编程过程中，可以先得到中序遍历序列，再通过一个栈将该序列倒置，即得到逆中序遍历序列。

2． 树形结构在文件管理中的应用

> （1） 怎样利用树形结构来管理文件目录，并能够将文件和文件夹加以区分；
>
> （2） 如何统计一个结点下文件夹和文件的数目；
>
> （3） 从目录树的管理上看，要实现文件夹或文件的删除、复制、移动，请描述算法的实现思路；
>
> （4） 地址路径和目录树结构怎么映射？给定一个地址路径，例如 `D:\Program Files\Microsoft Office\Office14`，怎么实现定位？反之，给定一个结点，获得相应路径地址。请描述算法实现的思路。

设计枚举结构 `FNodeType` 如下。

```
enum FNodeType
{
    FILE,
    DOCUMENT
};
```

构建`struct FTNode`（即`File Tree Node`），如下。

```
struct FTNode
{
    char *name = nullptr;                 // name of this node
    char *addressFromRoot = nullptr;      // address from file root
    FNodeType nodetype;                   // type of the node
    int countOfChildFile = 0;             // total number of child files
    int countOfChildDocument = 0;         // total number of child documents
    int size = 0;                         // size of this document/file, unit: byte
    struct FTNode *firstChild = nullptr;  // pointer to first child
    struct FTNode *nextSibling = nullptr; // pointer to next siblin`g
    /* other possible variables */
}; // struct FTNode
```

该结构使用了“孩子兄弟表示法”，即其本质为一个二叉链表的结点，左指针指向第一个孩子，右指针指向下一个兄弟。这个方法是多叉树中应用最广泛的方法。

- 变量`name`和`addressFromRoot`分别记录了该结点名称，和上一次进行`update()`操作时，该结点到根目录的路径。
- 变量`nodetype`记录了当前结点是否为文件夹，若值为`FILE`，则该结点是文件夹；若值为`DOCUMENT`，则该结点是文件。
- 变量`countOfChildFile`和`countOfChildDocument`分别记录了上一次进行`update()`操作时，整个树结构中，该结点的子文件夹和子文件数目。
- 变量`size`记录了上一次进行`update()`操作时当前结点（文件/文件夹）的大小，单位是`byte`。
- 变量`firstChild`和`nextSibling`分别记录了第一个子结点和下一个兄弟结点的地址。

构建`class FileTree`如下，相关函数功能已用注释方式伪码描述。

```
class FileTree
{
    struct FTNode *root = nullptr;
    FileTree() : root(new FTNode)
    {
        /* actions to initialize a file tree */
    }
    int judge_address_validity(const char *addr)
    {
        /* actions to judge validity */
        /* check if the input address is grammatically illegal, e.g. illegal characters, pointed to an inaccessible path */
    }
    struct FTNode *locate_node(const char *addr)
    {
        struct FTNode *cur = root->firstChild;
        /* reach and return the corresponding node according to the input address */
        return cur;
    }
    int insert_node(const char *addr)
    {
        /* actions to insert a new node */
        /* insert a node according to the input address */
    }
    int delete_node(const char *addr)
    {
        /* actions to delete a node */
        /* delete a node according to the input address */
    }
    int move_node(const char *dst, const char *src)
    {
        /* actions to move a node from src to dst */
        /* locate the node, delete it from src, insert it to dst */
    }
    int copy_node(const char *dst, const char *src)
    {
        /* actions to copy a node from src to dst */
```

```
        /* locate and recursively traverse the node, copy son
documents and files to the destination */
    }
    int update(struct FTNode *cur = nullptr)
    {
        if (!cur)
        {
            cur = root->firstChild;
        }
        /* actions to update information of nodes of whole tree */
        /* from the node cur, recursively traverse its first child
*/
    }
}; // class FileTree
```

- 成员变量 `root` 存储了指向整个文件树的根结点的指针。它的 `firstChild` 指向真正的根结点，`nextSibling` 应为 `nullptr`， `size` 存储了当前结点文件夹或文件的大小。
- 成员函数 `FileTree()` 包装了构建文件目录树的各种操作。
- 成员函数 `judge_address_validity()` 判断地址字符串是否合法，如是否含有非法字符、是否指向不可访问的目录等。
- 成员函数 `locate_node()` 用于定位地址对应的结点的地址并返回。若结点不存在，则返回 `nullptr`。
- 成员函数 `insert_node()` 用于根据地址字符串插入一个新结点。
- 成员函数 `delete_node()` 用于根据地址字符串递归删除一个既有结点及其子结点。
- 成员函数 `move_node()` 用于根据 `src` 和 `dst` 两个字符串移动既有结点。
- 成员函数 `copy_node()` 用于根据 `src` 和 `dst` 两个字符串复制既有结点。需要注意的是，需要**递归遍历原结点的所有子结点**，一起复制到新路径中。
- 成员函数 `update()` 用于递归遍历结点 `cur` 的第一个子结点和下一个兄弟结点，更新每个结点的相关信息（子文件和子文件夹数目、当前文件/文件夹大小、当前路径名称等）。

以上函数均设置了**错误反馈机制**，能判断 路径合法性 和 路径指向的结点是否存在 ，主要由 `judge_address_validity()` 和 `locate_node()` 实现。

（1） 怎样利用树形结构来管理文件目录，并能够将文件和文件夹加以区分?

如上所述，采用孩子兄弟表示法建立结点结构，构建文件目录树。设置具有标记作用的变量，区分文件夹和文件。

（2） 如何统计一个结点下文件夹和文件的数目?

```
int count(const struct FTNode *curNode, const enum FNodeType type)
{
    if (!curNode)
    {
        return 0;
    }
    if (FILE == curNode->nodetype)
    {
```

```
9           return (int)(type == curNode->nodetype) + count(curNode-
>firstChild, type) + count(curNode->nextSibling, type);
10      }
11      else // if (DOCUMENT == curNode->nodetype)
12      {
13          return (int)(type == curNode->nodetype);
14      }
15  }
```

如函数 `count()` 所示，递归遍历该结点的第一个子结点和下一个兄弟结点，分别统计子文件夹和子文件数目。即可得到一个结点下的子文件夹和子文件数目。

（3）从目录树的管理上看，如何实现文件夹或文件的删除、复制、移动?

如以上结构所示。

1. 删除：利用 `delete_node()` 函数。先定位结点，然后递归将该结点和子结点依次删除。
2. 复制：利用 `copy_node()` 函数。先定位原结点，然后在新位置递归地创建和原结点的各子结点值相同的新结点。
3. 移动：利用 `move_node()` 函数。先定位原结点，然后将该结点从它的上一个兄弟结点的 `nextSibling` 指针抹除；若为其父结点的第一个子结点，同样将其从其父结点的 `firstChild` 指针抹除。然后将其移入新位置，并更新相关指针信息。最后在该结点调用 `update()` 函数，更新相关信息。

（4）地址路径和目录树结构怎么映射?

1. 由地址路径映射到目录树结构：该功能由 `locate_node()` 函数实现，该函数最后会返回一个指向对应结点的指针。函数首先调用 `judge_address_validity()` 函数判断地址的合法性，然后将地址复制到一个新字符串 `str`，并将其中的正斜杠（“ / ”）都修改为反斜杠（“ \ ”）。函数开始遍历 `str`，每次读取到反斜杠为止，并将反斜杠丢弃；判断读到的下一级目录是否存在，若存在则继续遍历 `str`，直到找到对应结点并返回；否则返回 `nullptr`。
2. 由目录树结构映射到地址路径：该功能由 `update()` 函数实现，得到的结果字符串为 `addressFromRoot`。从目录树结构的根结点开始遍历到需要获得地址路径的结点，依次存储各结点的 `name` 信息，并用反斜杠（“ \ ”）分隔。最后得到的字符串即为当前结点所在地址路径。

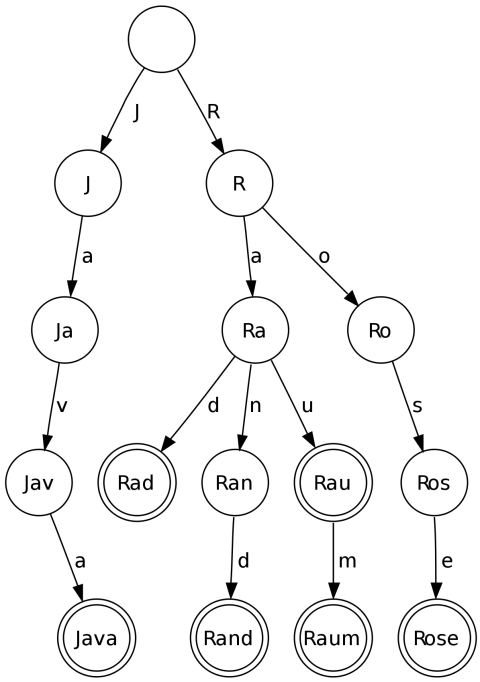
3. 找出重复次数最多的短信

> 有一千条短信，有重复，以文本文件（ASCII）形式保存，一行一条，请找出重复出现次数最多的前 10 条。

应用字典树实现。

字典树又称单词查找树， `Trie` 树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计、排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。

它的优点是：利用字符串的公共前缀来节约存储空间，最大限度地减少无谓的字符串比较，查询效率比哈希表高。



建立Trie树的大致步骤如下：

1. 遍历字符串，并根据字符串的每一个字符创建新结点/访问既有结点，直到该字符串遍历完成。在每个叶结点中设置一个计数器，用于统计各短信的出现次数。
2. 按照此方法遍历所有字符串（即短信）。

Trie树结点的大致结构如下。

```
struct TrieNode
{
    char value = '\0';
    char *str = nullptr;
    union MyData
    {
        struct MyPointers
        {
            struct TrieNode *firstChild = nullptr;
            struct TrieNode *nextSibling = nullptr;
        } relative;
        size_t freq = 0;
    } data;
}; // struct TrieNode
```

- 变量`value`和`str`分别储存了当前结点代表的字符和由根结点出发至当前结点位置所有字符组成的字符串。
- `union MyData`包含了两种结构：一是储存左右两个指针的结构体`struct Pointers`，二是存储叶结点出现频数的`size_t freq`。利用共用体，既实现了不同结点所需的功能，又达到了节省内存的目的。

读入所有短信后，从根结点开始遍历Trie树，将各叶结点放入排序规则为频数越高优先级越高的优先队列中，取出前10个结点并进行后续操作。

Trie树结构和一般的树结构大致相同，操作相似。

优缺点分析

- Trie树的优点是插入和查询的效率很高，都为$O(m)$，其中m是待插入/查询的字符串的长度。
- Trie树的遍历时间代价低，但由于需要储存的数据较多，占用的空间很多，空间复杂度较高。这是一种典型的以空间换时间的数据结构。

心得体会

此次研讨课别开生面，相比于上一次，同学们更加熟练也更加踊跃，在其他课上很难见到这样酣畅淋漓的学生讨论。

通过此次讨论，我对树形结构及其应用、对数据结构这门课程有了更深的认识。同学的发言也使我深受启发，拓展思维平时没有注意过得许多问题也在讨论中认识到并进行了思考讨论。非常期待在今后的算法分析等后续课程中更深入透彻地学习本研讨课中提到的相关算法。