第二次研讨课报告

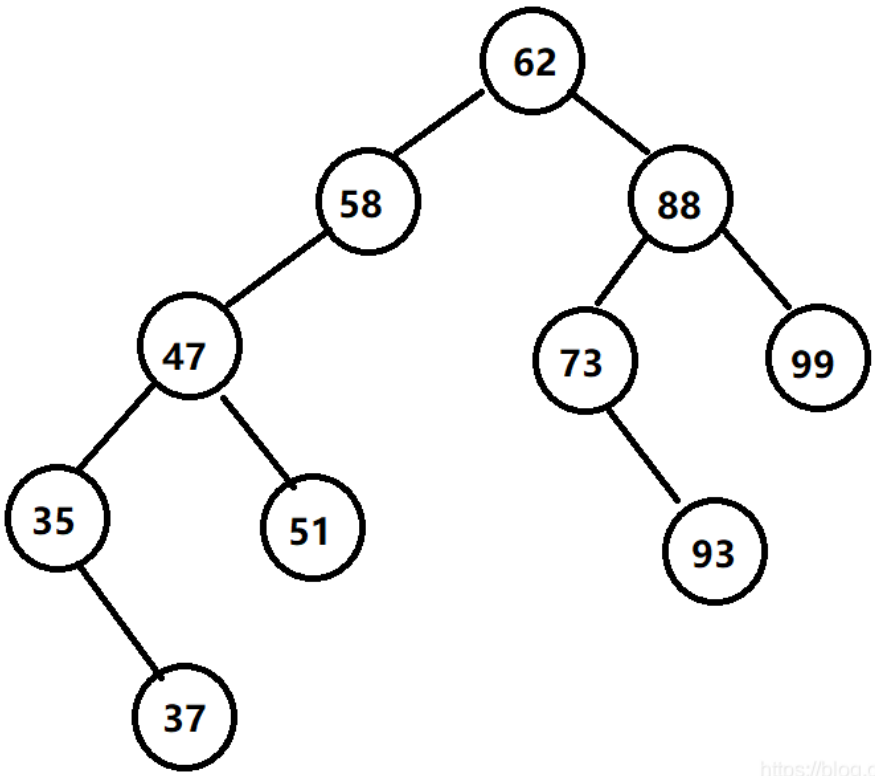
1 题目一

- 题目：
  - 二叉排序树具有如下性质：
    - 它或者是一棵空树，或者是具有下列性质的二叉树：
    - 若左子树不为空，则左子树上所有结点的值均小于它的根节点的值
    - 若右子树不为空，则右子树上所有结点的值均大于它的根节点的值
    - 左、右子树也分别为二叉排序树
  - 请思考：
    - 请给出一棵二叉排序树。
    - 如何利用该二叉排序树得到一个递增序列？
    - 如何利用该二叉排序树得到一个递减序列？
    - 建树过程中，每一次插入结点时如何调整结点位置才能维持二叉排序树的平衡？

1.1 二叉排序树举例

根据上述二叉排序树的定义，我们需要保证所有节点的值比其左子树中的所有节点

都大，比其右子树中的所有节点都小。如下图所示，该树就是二叉排序树，其每一个节点均满足左节点比根节点小，右节点比根节点大



1.2 二叉排序树的递增序列和递减序列

- 由二叉树的性质，根节点所存的值应该是在两个孩子节点之间，而每个节点的左子树中的元素一定小于根节点，右子树中的元素一定大于根节点，
- 递增序列
  - 所以当此树以中序遍历输出时，左子树均在根节点左边，均比根节点小，右子树均在根节点右边，均比右节点大，递归到每一个最小单元同样成立，所以该树的中序遍历为一个递增序列
- 递减序列
  - 相当于将树做一个镜像，再中序遍历输出
  - 也相当于在中序遍历中先输出右孩子节点，'这样输出的序列正好和递增序列相反，成为递减序列
- 递减序列

```
void ascending(node* ptr)
{
    if (ptr->left != NULL)
        ascending(pos->left);        //走到左子树的尽头
    cout << *ptr << " ";
    if (ptr->right != NULL)
        ascending(pos->right);
}
```

- 递增序列

```
void decending(node* ptr)
{
    if (ptr->right != NULL)
        decending(pos->left);        //走到右子树的尽头
    cout << *ptr << " ";
    if (ptr->left != NULL)
        decending(pos->right);
}
```

1.3 二叉排序树插入

- 根据二叉排序树的性质，二叉排序树在插入节点时，需要逐层递归判断被插入节点与插入位置处节点 的大小，如果被插入节点的值小于插入位置处节点的值就向左插入，反之向右插入。
- 二叉排序树的插入操作

```
void insertNode(int element, Node* & node)
{
    if (node == NULL)
    {
        Node* noder = new Node;
```

```
        noder->value = element;
        noder->left = noder->right = NULL;
        node = noder;
    }
    //插入
    else if (element >= node->value)
        insertNode(element, node->right);
    else
        insertNode(element, node->left);
}
```

- 此时会有一种极端情况，当一组递增或者递减序列插入到二叉排序树当中时，该二叉排序树将只会有左孩子节点，相当于退化成了一个链表，此时的增删改查操作将相对于链表没有优势，
- 所以我们可以通过调整二叉树的根节点与孩子节点的关系，在保证二叉搜索树本身性质不变的前提下维护每一个节点的左右子树的高度差绝对值不超过1，此时形成的二叉搜索树称为**平衡二叉树**（`AVL`）

1.3.1 平衡二叉树相关概念

AVL树本质上是一颗二叉查找树，但是它又具有以下特点：

- 可以是空树，假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过 1
- 左右两个子树 也都是一棵平衡二叉树。
- 在AVL树中，任何节点的两个子树的高度最大差别为 1 ,所以它也被称为平衡二叉树

- 此时平衡后进行查找删除等操作的时间复杂度都是对数，因此时二叉树的深度得到了调整

1.3.2 平衡二叉树的调整方式

- 实际进行平衡调整的过程中有以下概念
    - **平衡因子**(Balance Factor)
        - 左子树和右子树高度差
    - **最小不平衡子树**
        - 距离插入节点最近的，并且 BF 的绝对值大于 1 的节点为根节点的子树。
        - ***旋转纠正只需要纠正「最小不平衡子树」即可***
- 节点定义

```
class AVLNode {
public:
    /** 数据 **/
```

```
     int data;
    /** 相对高度 **/
    int height;
    /** 父节点 **/
    AVLNode parent;
    /** 左子树 **/
    AVLNode left;
    /** 右子树 **/
    AVLNode right;
    AVLNode(int data)
    {
        this.data = data;
        this.height = 1;
    }
}
```

- 计算高度

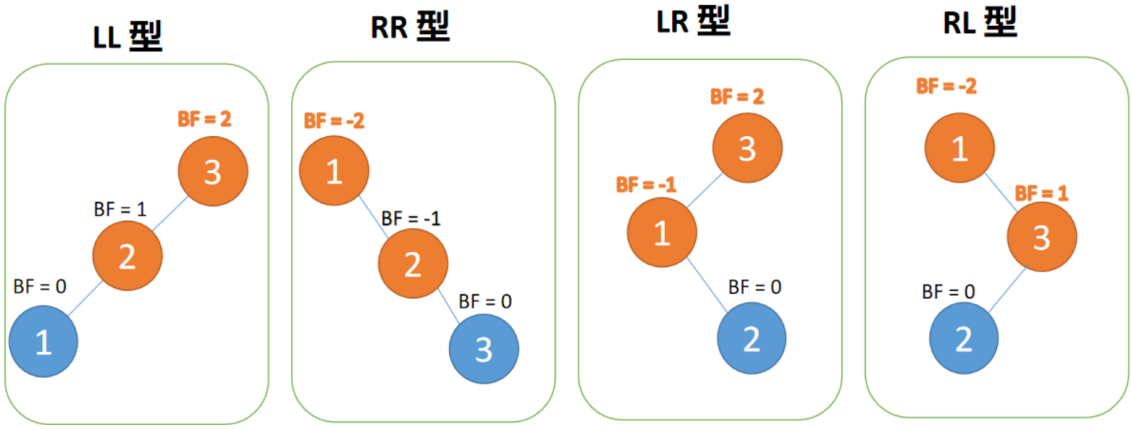
```
/** 通过子树高度 计算高度 **/
int calcHeight(AVLNode root)
{
    if (root.left == NULL && root.right == NULL) {
        return 1;
    }
    else if (root.right == NULL)
    {
        return root.left.height + 1;
    }
    else if (root.left == NULL)
    {
        return root.right.height + 1;
    }
    else
    {
        return root.left.height > root.right.height ? root.left.height + 1 :
root.right.height + 1;
    }
}
```

- 计算平衡因子

```
private int calcBF(AVLNode root)
{
    if (root == null)
    {
        return 0;
    }
    else if (root.left == null && root.right == null)
    {
        return 0;
    }
```

```
    else if (root.right == null)
    {
        return root.left.height ;
    }
    else if (root.left == null)
    {
        return - root.right.height;
    }
    else
    {
        return root.left.height - root.right.height;
    }
}
```

- 针对最小不平衡子树，旋转方式具有两大类和四种方式



- 其中前两种将中间节点变为根节点即可
- 右边两种需要变为左侧两中情况(`LR` 型先左旋，`RL` 型先右旋)
- 在变化过程中，需要注意节点高度变化，也即平衡因子的更新

- 左旋

```
AVLNode leftRotate(AVLNode root)
{
    AVLNode oldRoot = root;
    AVLNode newRoot = root.right;
    AVLNode parent = root.parent;

    //1.newRoot 替换 oldRoot 位置
    if (parent ! = NULL)
    {
        if (oldRoot.parent.data > oldRoot.data) {
            parent.left = newRoot;
        }
        else
```

```
        {
            parent.right = newRoot;
        }
    }
    newRoot.parent = parent;


    //2.重新组装 oldRoot (将 newRoot 的左子树 给 oldRoot 的右子树)
    oldRoot.right = newRoot.left;
    if (newRoot.left != NULL)
    {
        newRoot.left.parent = oldRoot;
    }


    //3. oldRoot 为 newRoot 的左子树
    newRoot.left = oldRoot;
    oldRoot.parent = newRoot;


    //刷新高度
    oldRoot.height = calcHeight(oldRoot);
    newRoot.height = calcHeight(newRoot);
    return newRoot;
}
```

- 右旋

```
AVLNode rightRotate(AVLNode root)
{
    AVLNode oldRoot = root;
    AVLNode newRoot = root.left;
    AVLNode parent = root.parent;

    //1.newRoot 替换 oldRoot 位置
    if (parent != NULL)
    {
        if (oldRoot.parent.data > oldRoot.data)
        {
            parent.left = newRoot;
        }
        else
        {
            parent.right = newRoot;
        }
    }
    newRoot.parent = parent;


    //2.重新组装 oldRoot (将 newRoot 的右子树 给 oldRoot 的左子树)
    oldRoot.left = newRoot.right;
    if (newRoot.right != NULL)
    {
```

```
        newRoot.right.parent = oldRoot;
    }

    //3. oldRoot 为 newRoot 的左子树
    newRoot.right = oldRoot;
    oldRoot.parent = newRoot;


    //刷新高度
    oldRoot.height = calcHeight(oldRoot);
    newRoot.height = calcHeight(newRoot);
    return newRoot;
}
```

- AVL插入

```
AVLNode insert(AVLNode root, int data)
{
    //插入左子树
    if (data < root.data)
    {
        if (root.left == NULL)
        {
            root.left = new AVLNode(data);
            root.left.parent = root;
        }
        else
        {
            insert(root.left,data);
        }
    }

    //插入右子树
    else if (data > root.data)
    {
        if (NULL == root.right)
        {
            root.right = new AVLNode(data);
            root.right.parent = root;
        }
        else
        {
            insert(root.right,data);
        }
    }

    //刷新高度
    root.height = calcHeight(root);

    //旋转
    //1. LL 型 右旋转
    if (calcBF(root) == 2)
    {
```

```
        //2. LR 型 先左旋转
        if (calcBF(root.left) == -1)
        {
            root.left = leftRotate(root.left);
        }
        root = rightRotate(root);
    }

    //3. RR型 左旋转
    if (calcBF(root) == -2)
    {
        //4. RL 型 先右旋转
        if (calcBF(root.right)== 1)
        {
            root.right = rightRotate(root.right);
        }
        root = leftRotate(root);
    }

    return root;
}
```

- 在插入元素的过程中不断使用左旋、右旋、先左旋再右旋、先右旋再左旋等操作来满足二叉平衡树的性质（每个结点的左子树与右子树的高度差不超过 1）。
  但是二叉平衡树的要求太严格，若出现频繁的插入、删除操作对二叉平衡树的性能会大打折扣，为了避免这种情况的发生，又引申出了**红黑树**
- 红黑树的出现是为了解决二叉平衡树频繁的插入和删除导致的性能降低的问题
- 此处再不讨论红黑树

2. 题目二

- 题目
  - 树形结构在文件管理中的应用：
    - 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。
    - 如何统计一个节点下的文件夹和文件的数目。
    - 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。
    - 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:\ProgramFiles\MicrosoftOffice\Office14`，怎么实现定位。反之，给定一个节点，获得相应路径的地址。请描述算法的实现思路

2.1 树形结构管理文件目录

- 类型区分
- 首先我们可以将每个磁盘 (根目录) 看作一棵树的根节点，不同的磁盘是相互独立的树结构。一个目录下的文件和子目录可以看作是该节点下的子节点。
- 由于空文件夹和文件都没有子节点，因此不能仅仅通过是否有子节点来区分文件夹和文件，我们可以直接在存储文件信息的结构体中设置一个变量来代表类型。文件夹和文件的数目即为子树节点的数目，因此可以直接在每个子树的根节点上使用变量直接记录。

- 关于文件下子文件夹的数目
- 方法一：文件树形成的树状结构不一定是二叉树，每个文件夹节点下的子节点很可能超过两个。为了方便维护，可以使用指针数组或指针顺序表的方式进行维护，指针顺序表的长度即为当前目录下的所有文件和文件夹的数目总和。
- 方法二：我们可以将子目录中文件数量，文件信息线性表都封装到一个包含子文件信息的结构体中。并将该文件信息结构体的指针放入文件信息结构体中。此时，如果该指针的值为 `NULL` 则表示文件，如果该指针的值为非空则表示文件夹

2.2 实现文件夹或文件的删除、复制、移动

- 文件的删除、复制、移动只需要移动单个节点即可。文件夹的删除、复制、移动过程需要对整个子树进行操作。
- 文件夹的删除可以看作是一棵树的全部节点的释放。为了不丢失内存，我们要采用后序遍历的方式进行内存释放，即先释放子节点，再释放根节点。
- 文件夹的复制可以看作是树结构的复制，此处注意我们要将树的所有数据都复制一遍 (深拷贝)，而不是简单的进行指针的移动 (浅拷贝)。树的复制要借助前序遍历的思想，先复制根节点，再依次复制左右节点。文件的移动时只需要将相应的指针移动即可。

2.3 目录树结构查址

- 对于给定的地址路径，进行逐级查找，以 `\` 作为分隔符，作为每一级节点名的查找依据，查找到相应节点
- 对于给出的一个节点，只要递归返回它的上级路径直到查找到根节点即可

```
string findRoot(FileTree* file, string path)
{
    if (file == Root)
        return file;
    else
        return findRoot(file->parent) + file.path;
}
```

3. 题目三

- 题目：有一千万条短信，有重复，以文本文件(ASCII)的形式保存，一行一条，请找出重复出现最多的前十条
- 分析：该问题的本质是大数据查重问题，关键在于如何数据映射到不同的元素（构造单射函数），进而进行进一步的查找操作。尤其对于相似度极高的数据，如何进行合理映射提高查找效率是关键问题（同济大学核酸提醒等极为相似的短信信息等）

Solution 1: 哈希表

- **散列表**（**Hash table**，也叫**哈希表**），是根据键（Key）而直接访问在内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做**散列表**。
  - 通过设置哈希函数，将某一条数据映射到某一键值，通过键值进行查重，可以极大概率得到准确结果
  - 遇到哈希冲突时可以进行二次比较，若哈希函数合理，每一键值对应的数据数据量较小，比较较快

- 对文本文件(ASCII)码进行处理可以应用字符串哈希，即构造一个数字使之唯一代表一个字符串。
- 我们先定义：给定一个字符串$S = s_1s_2s_3\dots s_n$，对于每一个 s_i就是一个字母，规定 $idx(s_i) = s_i -' a' + 1$（也可以直接用ASCII值）字符串哈希使用 Base 和 MOD（都要求是素数），一般都是 Base ＜ MOD，同时将Base和MOD尽量取大，这种情况下，冲突概率很低。

(获取前i个字符的哈希值)

unsigned long long int 自然溢出

- 对于自然溢出方法，我们定义 Base ，而MOD对于自然溢出方法，就是 unsigned long long 整数的自然溢出（相当于MOD 是 $2^{64}-1$）

```
unsigned long long Base;
unsigned long long hash[MAXN], p[MAXN];

hash[0] = 0;
p[0] = 1;
```

定义了上面的两个数组，首先 $hash[i]$表示$[0,i]$字串的hash 值。而$p[i]$表示 $Base^i$，也就是底的 i次方。对应的Hash公式为

$$hash[i] = hash[i-1] * Base + idx(s[i])$$

单Hash方法

- 同样定义Base和MOD，用Long long 实现

```
long long Base;
long long hash[MAXN], p[MAXN];

hash[0] = 0;
p[0] = 1;
```

- 定义了上面的两个数组，首先$hash[i]$表示$[0,i]$字串的hash 值。而$p[i]$表示 $Base^i$，也就是底的i次方。对应的Hash公式为

$$ash[i] = (hash[i-1] * Base + idx(s[i]))\%MOD$$

对于此种Hash方法，将Base和MOD尽量取大即可，这种情况下，冲突的概率是很低的。

双Hash方法

- 用字符串Hash可能出现冲突的情况，即不同字符串却有着相同的hash值，为了降低冲突的概率，可以用双Hash方法。
- 将一个字符串用不同的Base和MOD，hash两次，将这两个结果用一个二元组表示，作为一个总的Hash结果。
- 相当于我们用不同的Base和MOD，进行两次 单Hash方法 操作，然后将得到的结果，变成一个二元组结果，这样子，我们要看一个字符串，就要同时对比两个 Hash 值，这样子出现冲突的概率大大降低

- 对应的Hash公式为

$$hash1[i] = (hash[i-1] * Base1 + idx(s[i]))\%MOD$$
$$hash2[i] = (hash[i-1] * Base2 + idx(s[i]))\%MOD$$

- 映射的哈希结果为$< hash1[i], hash2[i] >$

进制哈希（BKDRHash）

- 首先我们定一个 素数 seed (常见 31,313,3131,3131313 等等)，则 hash值就等于 每一个字符的ASCII码乘以seed的n次方 累加 之后再对哈希数组大小取余得到余数就是hash值，n就是位数。
- 对应Hash键值(n为字符串的长度)

$$key = (\sum_{i=0}^{n} s[i] * seed^{n-1-i})\%n$$

- 代码实现

```
 unsigned int BKDRHash(char* str)
{
    unsigned int seed = 31;    // 31 131 1313 13131 131313 etc.. 37

    unsigned int key = 0;

    while (*str)
    {
        key = key * seed + (*str++);
    }

    return ( key & 0x7fffffff ) % size;
}
```

- 可以将短信数据的每一条数据映射到某一哈希值，通过统计哈希值来进行计数，找出其中重复前10的数据

`c++STL` `unordered_set`、`unordered_map`、`unordered_multiset`、`unordered_multimap`

- 实现时可以用把每条数据先映射成一个数值，再用除留余数法把每条数据分配到一个和余数对应的小文件中，这样相同的数据一定在一个小文件当中
- 这样若内存空间受限，可以通过一次读入一个小文件的方式，进行统计

4. 总结

本次讨论课主要讨论了树型结构中

- 二叉查找树
  - 增删改查
  - 二叉搜索树排序
  - 平衡(`AVL`)
    - 旋转操作
- 树形结构在操作系统中文件管理的应用
- 哈希表与映射
  - 哈希函数
  - 字符串哈希