数据结构第二次上机

物理学院 沈鹏飞 PB17081531

平衡排序树

需求分析

一个二叉排序树满足对于任意结点A，其左子树所有结点的key小于A的key，右子树所有结点大于A的key。如此存储的排序树仅需要进行中序遍历就可以得到其有序结果，并且在对于某一个固定的键值查找时可以获得更高的速度（预期为O(logN)）。再插入和删除的过程中仅需要保持此性质即可。

然而，对于这种排序树，还有可能会有一些不太理想的情况出现。比如原始输入数据比较有序，那马就有可能出现"歪脖子树"的情况。最极端的情况比如原始数据就是从小到大顺序的，那么初始的排序树中所有的子树都会出现在右节点上。这样并没有起到提高搜索速度的作用。因此通过设计一些算法来使得排序树左右子树尽量等高，由此来提高排序的效率。

概要设计

本次代码的基本设计采用课程内容中所介绍的方法。在每一个节点设置一个平衡因子，表示左子树高减右子树高的值。一棵正常的平衡排序树允许的平衡因子为(-1,0,1)。再插入和删除结点时会对平衡因子造成影响，当取值超出这个范围时就要进行调整。调整的方法采用课程介绍的旋转法。

每一个节点的设计如下：

```
struct Node
{
    Node* left;
    Node* right;
    Node* parent;
    Node* link(Node*, bool);//0链在左边，1链在右边
    int data;
    int balance_index;
};
```

程序将LL，LR，RL，LL四种情况各自封装为函数，其函数原型如下：

```
Node* LL_rotate(Node* node)
```

其返回值是旋转之后子树的根节点。

最后将平衡搜索树封装为一个类：

```
class BssTree
{
public:
    BssTree() :rootnode(INT_MAX) {}
```

```
    ~BssTree();

    int insert(int i);
    void insert(std::initializer_list<int>);
    int remove(int i);
    void clear() { this->~BssTree(); }

    void mid_traverse(void(*func)(int));
    void print(ostream& os);
private:

    void TreePrint(Node* T, int level,ostream&);
    Node* LL_rotate(Node* node);
    Node* RR_rotate(Node* node);
    Node* LR_rotate(Node* node);
    Node* RL_rotate(Node* node);
    void delete_node(Node*);
    void mid_traverse(void(*func)(int), Node* node);//传入

    Node rootnode;

};
```

存储根结点，根节点的左孩子作为树的根。根结点的key值设置为最大整数，这样在加入结点时就始终会落在其左孩子上，平衡调整时不要触及根节点即可。

详细设计

需要详细介绍的主要是两个旋转算法和插入和删除寻找调整节点的过程。

由于LL和RR对称，LR和RL对称，就选取其中的两个来进行介绍。

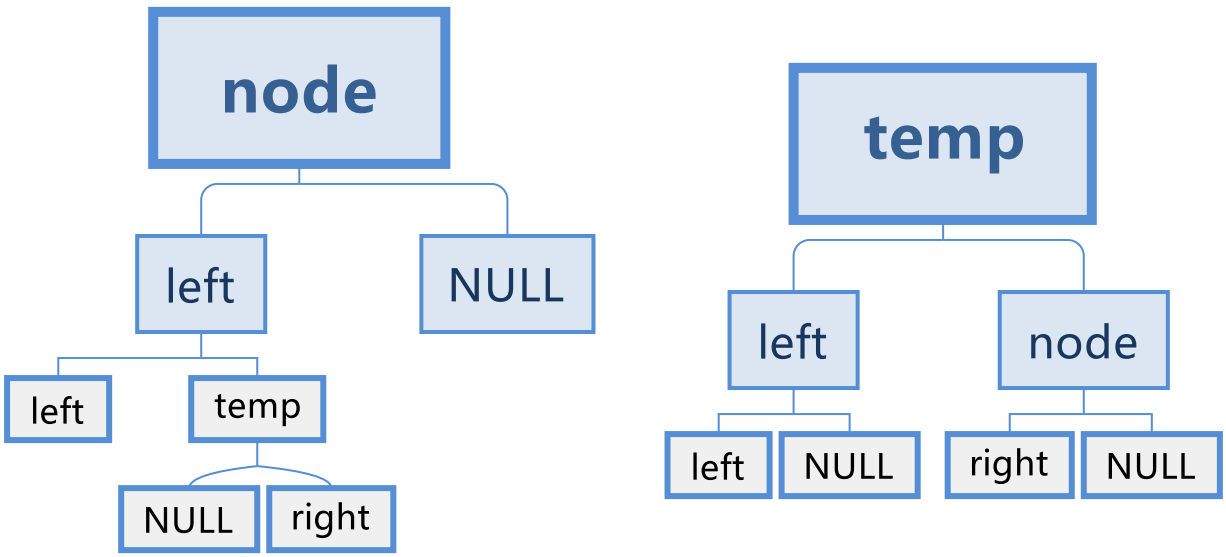
```
Node* BssTree::LL_rotate(Node* node)
{
    Node* temp = node->left;
    node->link(temp->right,0);
    node->balance_index -= (temp->balance_index+1);
    //当左孩子的平衡因子原始为0时，旋转后根节点的平衡因子下降1.初始为1时，旋转后下降2
    temp->link(node, 1);
    temp->balance_index -= 1;
    //旋转后左孩子的平衡因子总是下降1
    return temp;
}
```

```
Node* BssTree::LR_rotate(Node* node)
{
    auto temp = node->left->right;
    switch (temp->balance_index)//此处进行了一个复杂的分情况讨论。直接通过重新连接结点来实现LR旋转。其正确性通过分类尝试一一检验是最容易的办法。以下给出几个简单的例子
    {
    case -1:
        node->left->balance_index = +1;
```

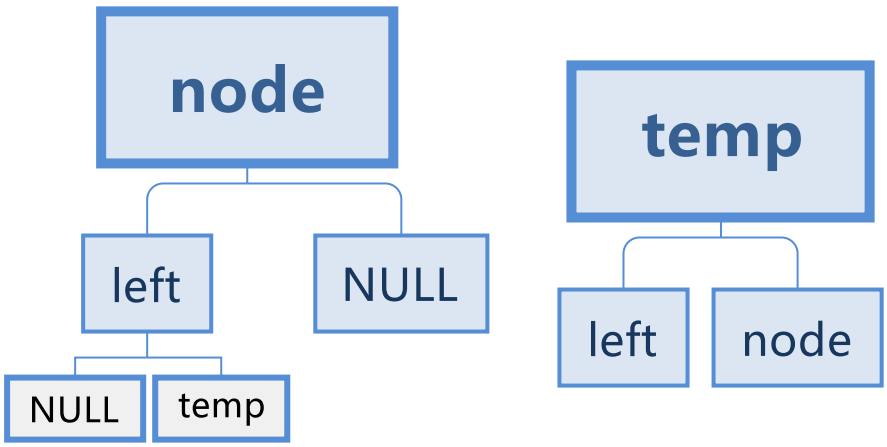
```
            node->balance_index = 0;
            break;
        case 1:
            node->left->balance_index = 0;
            node->balance_index = -1;
            break;
        case 0:
            node->left->balance_index = 0;
            node->balance_index = 0;
        }
        auto debug = node->left;
        debug->link(temp->left, 1);

        auto temp2 = node->left;
        node->link(temp->right, 0);
        temp->link(node, 1);
        temp->link(temp2, 0);
        temp->balance_index = 0;
        return temp;
    }
```

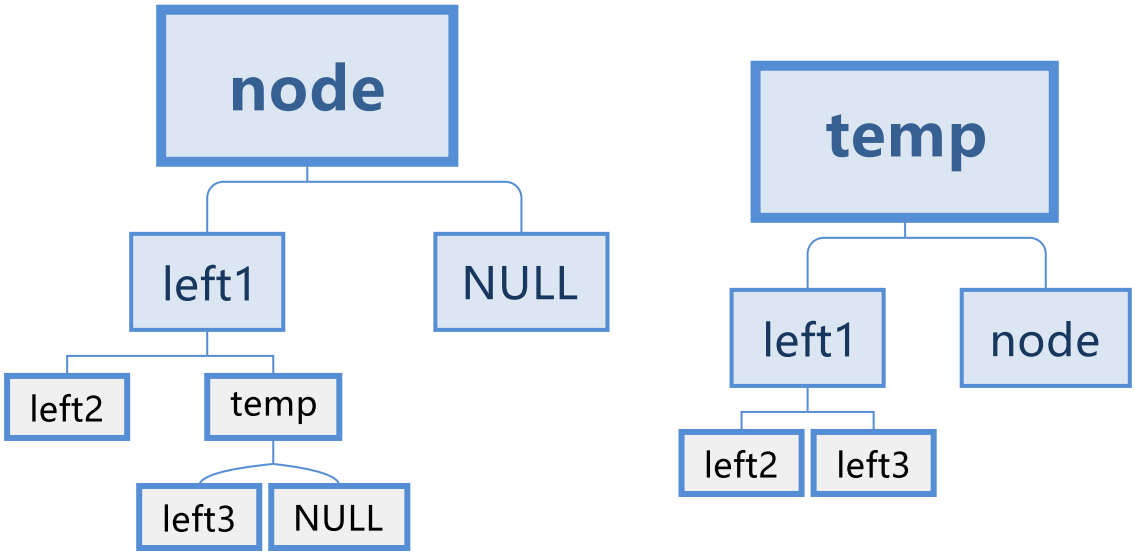
case -1:



case 0:



case 1:



只讨论了这几种代表情况。实际上对所有情况都成立。

插入结点相对来说较为简单。只需要寻找到离插入位置最近的可能失衡的结点，并从此结点开始改变平衡因子。之后判断此结点的平衡因子，若已失衡就进行旋转。旋转之后一定恢复平衡，并且与先前的树高相同，即不会对更上层树的平衡因子造成影响，但删除有可能反复降低树高，并对上层的结点的平衡因子造成影响。此处就分析相对难度更大的删除结点的过程。

```
int BssTree::remove(int i)
{
    Node* follower, * node, * to_be_delete = nullptr;
    follower = &rootnode;
    node = rootnode.left;

    while (node)
    {
        follower = node;
        if (i > node->data) node = node->right;
        else if (i < node->data)node = node->left;
        else {
```

```
            to_be_delete = node;//标记待删结点
            if (node->left && node->right)
                if (node->balance_index >= 0) node = node->left;
                else node = node->right;
            else
                follower = node; break;
        }
    }//follower此时指向真正在空间上会被删除的结点
    if (to_be_delete) {
        to_be_delete->data = follower->data;//实现了交换的目的
        if (follower->parent->left == follower)
        {……}
        else
        {……}//删除节点的过程，不再赘述
        while (follower != &rootnode && follower->balance_index == 0)
        //经调整后平衡因子为0的结点树高降低了，需要不断向上调整同时寻找至发现调整之后平衡因子不为0的结点
        {
            follower->parent->balance_index +=
                   (follower->parent->left == follower ? -1 : 1);
            follower = follower->parent;
        }
        while (follower != &rootnode) {
            if (follower && follower->balance_index == 2)
            {……}
            else if (follower && follower->balance_index == -2)
            {……}//调用旋转函数进行调整的分支，不再赘述
            else break; //当某一个结点没有再失衡就说明这个结点的树高没有减小。因此不需要再向上搜索
            while (follower != &rootnode && follower->balance_index == 0)
            {//过程同上
                follower->parent->balance_index +=
                    (follower->parent->left == follower ? -1 : 1);
                follower = follower->parent;
            }
        }
    }
    else
    {
        return 0;//未删除
    }
    return 1;//成功删除
}
```

算法时空分析

查找为二叉查找，耗时为(O(logN))

插入和删除结点的过程都是先查找再回溯修正结点。其耗时都依赖于树高，所以也都是(O(logN))级别。

算法测试分析

每一次插入结点和删除结点后打印整棵树，观察发现保持排序树的性质，并且能够保持平衡。并且对于大数据量的情况也进行了测试，进行百万量级数据反复插入和删除，保持平衡因子和树性质的正确性。

最短路径

需求分析

在一个带权有向图中找出两个结点之间的最短路径，输入起点终点，并且输入一条拥塞道路，要求能够给出最短路径。拥塞道路实际上就是改变了原始图的邻接矩阵。所有原始矩阵中不邻接的两点都可以视为拥塞。

概要设计

采用Floyd算法，一次求出对于所有结点互相到达的最短路径长度，并且给出其方向。

对于邻接矩阵的每一个结点，定义类型：

```
struct Node
{
    int next;
    int distance;
};
```

distance表示距离，next表示如果要到达此结点下一次应该前往的位置。

以-1来表示无穷远距离，因此要自己定义路径长度相加和比较路径长度所用的函数。

详细设计

路径长度相加函数：

```
int add(int i, int j)
{
    if ((i == -1) || (j == -1)) return -1;
    else return i + j;
}
```

路径长度比较函数：

```
bool compare(int i, int j)//i<j时返回值为true
{
    if (i == -1) return 0;
    if (j == -1)return 1;
    return i <= j;
}
```

Floyd算法：

```
void Floyd(Node head[][size])
{
    for (int i = 0; i < size; i++)
    {
```

```
        for (int j = 0; j < size; j++)
        {
            for (int k = 0; k < size; k++)
            {
                if ((!compare(head[j][k].distance, add(head[i][k].distance, head[j]
[i].distance)))&& add(head[i][k].distance, head[j][i].distance)!=-1)
                    //如果新的距离仍为无穷远就不更新道路(next元素)
                {
                    head[j][k].distance = add(head[i][k].distance, head[j][i].distance);
                    head[j][k].next = head[j][i].next;
                }
            }
        }
    }
}
```

算法时空分析

Floyd算法具有三重循环，每一层都是size量级，因此总的时间复杂度为O(n^3)。

算法测试分析

对于给定初始邻接矩阵，在输入拥塞节点后，可以对于输入的起点和终点给出正确的路径。

快速排序

需求分析

要求利用快速排序算法对于一个给定数组进行排序。

概要设计

对于一个数组挑出一个key值（可以选用第一个值），通过一次排序，将所有小于key值的元素放在key的左边，大于key值的元素放在key的右边。然后对首元素到key和key到尾元素递归地进行此操作。

详细设计

```
void fast_sort(int* beg, int* end)
{

    if (beg < end)
    {
        swap(beg, beg + rand() % (end - beg));//随机选取一个元素作为key值
        int* head = beg, * tail = end;
        int key = *beg;
        while (head != tail)
        {
            while (*tail >= key && tail != head) tail--;
```

```
            *head = *tail;
            if (head == tail) break;
            while (*head < key && tail != head) head++;
            *tail = *head;
        }//任何情况下只要头尾相遇就跳出循环
        *head = key;

        fast_sort(beg, head);
        fast_sort(head + 1, end);

    }
}
```

算法时空分析

其时间复杂度为O(NlogN),空间复杂度主要是栈的调用，为O(logN)。

算法测试分析

对于任意大数据量可以快速给出正确的结果。