1 手写代码练习

2 二叉树的相关代码——80%考察

2.1 遍历

```
// 先序遍历
// 递归
template <class ElemType>
void BinaryTree<ElemType>::PreOrderHelp(BinTreeNode<ElemType>*r,
void(*visit)(const ElemType&))const{
    if(r!=NULL){
        (*visit)(r->data);
        PreOrderHelp(r->leftChild, visit);
        PreOrderHelp(r->rightChild, visit);
    }
}

// 非递归
// 就去想一步步到底怎么走下去的，每一个分叉点都有几个选择
template<class ElemType>
void BinaryTree<ElemType>::PreOrderHelp(BinTreeNode<ElemType>*r,
void(*visit)(const ElemType&))const{
    BinTreeNode<ElemType>*cur = r;
    LinkStack<BinTreNode<ElemType>*> s;

    while(cur != NULL){
        (*visit)(cur->data);
        s.Push(cur);

        if(cur->leftChild != NULL){
            cur = cur->leftChild;
        }
        else if(!s.Empty()){
            while(!s.Empty()){
                s.Pop(cur);
                cur = cur->rightChild;
                if(cur != NULL) break;
            }
```

```
        }
        else{
            cur = NULL;
        }
    }
}

// PreOrder 成型函数
template<class ElemType>
void BinaryTree<ElemType>::PreOrder(BinaryTree&bt, void(*visit)
(const ElemType&))const{
    PreOrderHelp(bt.GetRoot(), visit);
}


// 中序
// 递归
template <class ElemType>
void BinaryTree<ElemType>::InOrder(BinTreeNode<ElemType>*r,
void(*visit)(const ElemType&))const{
    if(r!=NULL){
        InOrder(r->leftChild, visit);
        (*visit)(r->data);
        InOrder(r->rightChild, visit);
    }
}

// 非递归 -- 找到最左边的一个点后 第一步要做的肯定是访问第一个点 然后看一看它的
右子树有没有东西，有的话同理，直到某节点的右子树没东西了，回溯
template<class ElemType>
BinTreeNode<ElemType>
BinaryTree<ElemType>::GetFarLeft(BinTreeNode<ElemType>*r,
LinkStack<ElemType>&s){
    if(r == NULL){
        return NULL;
    }
    else{
        BinTreeNode<ElemType>* cur = r;
```

```
        while(cur->leftChild!=NULL){
            s.Push(cur);
            cur = cur->leftChild;
        }
        return cur;
    }
}
template<class ElemType>
void BinaryTree<ElemType>::InOrder(BinTreeNode<ElemType>*r,
void(*visit)(const ElemType&)){
    BinTreeNode<ElemType> *cur = r;
    LinkStack<BinTreeNode<ElemType>* > s;

    cur = GetFarLeft(cur, s);
    while(cur != NULL){
        (*visit)(cur->data);
        if(cur->rightChild != NULL){
            cur = GetFarLeft(cur->rightChild, s);
        }
        else if(!s.Empty()){
            s.Pop(cur);
        }
        else {
            cur = NULL;
        }
    }
}

// InOrder 成型函数
template<class ElemType>
void BinaryTree<ElemType>::InOrder(BinaryTree<ElemType>&bt,
void(*visit)(const ElemType&))const{
    InOrderHelp(bt.GetRoot(), visit);
}

// 后续遍历二叉树
// 递归
template <class ElemType>
```

```
void BinaryTree<ElemType>::PostOrderHelp(BinTreeNode<ElemType>*r,
void(*visit)(const ElemType&)) const {
    if(r!=NULL){
        PostOrderHelp(r->leftChild, visit);
        PostOrderHelp(r->rightChild, visit);
        (*visit)(r->data);
    }
}
template <class ElemType>
void BinaryTree<ElemType>::PostOrder(BinaryTree<ElemType>&bt,
void(*visit)(const ElemType&))const{
    PostOrderHelp(bt.GetRoot(), visit);
}

// 非递归 -- 赌一把不考 -- 怂了怂了不敢赌了
// 其实它的本质和中序是一样的 只不过是多了一个 表示右子树是否读取的标志罢了
template <class ElemType>
struct MidiNode{
    BinTreeNode<ElemType> *node;
    bool rightSubTreeVisited;
}
template <class ElemType>
MidiNode<ElemType>
BinaryTree<ElemType>::GetFarLeft(BinTreeNode<ElemType>*r,
LinkStack<MidiNode<ElemType>* > s){
    if(r == NULL){
        return NULL;
    }
    else{
        BinTreeNode<ElemType> *cur = r;
        MidiNode<ElemType> *newPtr;
        while(cur->leftChild != NULL){
            newPtr->node = cur;
            newPtr->rightSubTreeVisited = false;
            s.Push(newPtr);
            cur = cur->leftChild;
        }
        newPtr->node = cur;
```

```
        newPtr->rightSubTreeVisited = false;
        return newPtr;
    }
}
template<class ElemType>
void BinaryTree<ElemType>::PostOrderHelp(BinTreeNode<ElemType> *r,
void(*visit)(const ElemType&))const{
    if(r!=NULL){
        ModiNode<ElemType>*cur;
        LinkStack<MidiNode<ElemType>* >s;
        cur = GetFarLeft(r, s);

        while(cur!=NULL){
            if(cur->node->rightChild == NULL || cur-
>rightSubTreeVisited){
                (*visit)(cur->node->data);
                delete cur;

                if(!s.Empty()){
                    s.Pop(cur);
                }
                else{
                    cur = NULL;
                }
            }
            else{
                cur->rightSubTreeVisited = true;
                s.Push(cur);
                cur = GoFarLeft<ElemType>(cur->node->rightChild,
s);
            }
        }
    }
}


// 层次遍历
// 只有非递归 -- 送分
```

```
template <class ElemType>
void BinaryTree<ElemType>::LevelOrder(BinTreeNode<ElemType>*r,
void(*visit)(const ElemType&))const{
    BinTreeNode<ElemType>*cur = r;
    LinkQueue<BinTreeNode<ElemType>*> q;
    if(cur!=NULL) q.InQueue(cur);
    while(!q.Empty()){
        q.OutQueue(cur);
        (*visit)(cur->data);
        if(cur->leftChild != NULL)
            q.InQueue(cur->leftChild);
        if(cur->rightChild != NULL)
            q.InQueue(cur->rightChild);
    }
}
```

2.2 遍历的应用

```
// 应用一 二叉排序树的输出
// 递归算法 从大到小遍历一个二叉排序树 这个太 easy 了无非就是先访问右孩子再访
问左孩子的前序罢了
template <class ElemType>
void InOrderHelp(BinTreeNode<ElemType>*r) const{
    if(r != NULL){
        InOrderHelp(r->rightChild);
        cout << r->data <<" ";
        InOrderHelp(r->leftChild);
    }
}
// 上面的题目的一个衍生题目 递归算法从大到小展示出二叉排序树中小于key的值
template<class ElemType, class KeyType>
void BinarySortTree<ElemType>::InOrderHelp(BinTreeNode<ElemType>*r,
KeyType key){
    if(r != NULL){
        InOrderHelp(r->rightChild, key);
        if(r->data < key){
            cout << r->data << " ";
        }
        InOrderHelp(r->leftChild, key);
```

```
    }
}
template<class ElemType, class KeyType>
void BinarySortTree<ElemType>::InOrder(BinSortTree<ElemType>&bst,
KeyType key){
    InOrderHelp(bst.GetRoot(), key);
}

// 应用二 二叉树的各种操作
// 二叉树的拷贝
// 递归
template <class ElemType>
void BinaryTree<ElemType>::CopyTreeHelp(BinTreeNode<ElemType>*r){
    if(r == NULL){
        rerurn NULL;
    }
    else{
        BinTreeNode<ElemType> *leftChild = CopyTreeHelp(r-
>leftChild);
        BinTreeNode<ElemType> *rightChild = CopyTreeHelp(r-
>rightChild);
        BinTreeNode<ElemType> *tn = new BinTreeNode(r->data,
leftChild, rightChild);
        return tn;
    }
}
template <class ElemType>
BinaryTree<ElemType>
BinaryTree<ElemType>::CopyTree(BinaryTree<ElemType>&bt){
    BinTreeNode<ElemType>*newTreeRoot = CopyTreeHelp(bt.GetRoot());
    return BinaryTree(newTreeRoot);
}

// 非递归 -- 基于层次遍历 -- 在访问的当口能做太多的事情了
template<class ElemType>
void CopyTree(BinaryTree<ElemType> &fromBT, BinaryTree<ElemType>
*&toBT){ // 注意这个地方指针的引用
    if(toBT != NULL) delete toBT;
```

```
    if(froBT.Empty()) toBT = NULL;
    else{
        LinkQueue<BinTreeNode<ElemType>*> fromQ, toQ;
        BinTreeNode* fromPtr, toPtr, fromRoot, toRoot;
        fromRoot = fromBT.GetRoot();
        toRoot = new BinTreeNode<ElemType>(fromRoot);
        fromQ.InQueue(fromRoot);
        toQ.InQueue(toRoot);
        while(!fromQ.Empty()){
            fromQ.OutQueue(fromPtr);
            toQ.outQueue(toPtr);
            if(fromPtr->leftChild != NULL){
                toPtr->leftChild = new BinTreeNode<ElemType>
(fromPtr->leftChild->data);
                fromQ.InQueue(fromPtr->leftChild);
                toQ.InQueue(toPtr->leftChild);
            }
            if(fromPtr->rightChild != NULL){
                toPtr->rightChild = new BinTreeNode<ElemType>
(fromPtr->rightChild->data);
                fromQ.InQueue(fromPtr->rightChild);
                toQ.InQueue(toPtr->rightChild);
            }

        toBT = new BinaryTee<ElemType>(toRoot);
    }
}

// 二叉树的销毁
template <class ElemType>
void BinaryTree<ElemType>::DestroyHelp(BinTreeNode<ElemType> *&r){
// 这个地方也是写指针引用
    if(r != NULL){
        DestroyHelp(r->leftChild);
        DestroyHelp(r->rightChild);
        delete r;
        r = NULL;
    }
```

```
}
template <class ElemType>
void BinaryTree<ElemType>::Destroy(BinaryTree<ElemType> &bt){
    DestroyHelp(bt.GetRoot());
}

// 统计二叉树中结点的个数 —— 递归非递归均可 不就是遍历一下嘛
// 以先序为内核 —— 复习一下上午所记得
template <class ElemType>
int BinaryTree<ElemType>::CountNodeHelp(BinTreeNode<ElemType>*r){
   if(r == NULL) return 0;
    else return CountNodeHelp(r->leftChild) + CountNodeHelp(r-
>rightChild) + 1;
}
template <class ElemType>
int BinaryTree<ElemType>::CountNode(BinaryTree<ElemType> &bt){
    return CountNodeHelp(bt.GetRoot());
}

template <class ElemType>
int BinaryTree<ElemType>::CountNodeHelp(BinTreeNode<ElemType>*r){
    int count = 0;
    BinTreeNode<Type> * cur = r;
    LinkStack<BinTreNode<ElemType>*> s;

    while(cur!=NULL){
        count++;
        s.Push(cur);

        if(r->leftChild != NULL){
            r = r->leftChild;
        }
        else if(!s.Empty()){
            while(!s.Empty()){
                s.Pop(cur);
                cur = cur -> rightChild;
                if(cur != NULL) break;
            }
```

```
        }
        else cur = NULL;
    }
}

// 统计二叉树中叶子结点的个数 —— 当然可以用上面的方法只需要在 Count++ 前加一个条件就可以了
// 但是我们可以变得更加简单
template <class Elemtype>
long BinaryTree<ElemType>::leafCountHelp(BinTreeNode<ElemType>*r){
    if(r == NULL){
        return 0;
    }
    else{
        if(r->rightChild == NULL && r->leftChild == NULL){
            return 1;
        }
        else{
            return (leafCountHelp(r->leftChild) + leafCountHelp(r->rightChild))
        }
    }
}
template <class ElemType>
long LeafCount(const BinaryTree<ElemType> &bt)
// 操作结果：计算二叉树中叶子结点数目
{
    return LeafCountHelp(bt.GetRoot()); // 调用辅助函数实现计算二叉树中叶子结点数目
}

// 求二叉树的高
template <class ElemType>
int BinaryTree<ElemType>::HeightHelp(BinTreeNode<ElemType> *r){
    if(r == NULL){
        return 0;
    }
    else{
```

```
        int lHeight = HightHelp(r->leftChild);
        int rHeight = HightHelp(r->rightChild);
        return (lHeight > rHeight?lHeight:rHeight) + 1;
    }
}
template <class ElemType>
int BinaryTree<ElemType>::Height(BinaryTree<ElemType> &bt){
    return HeightHelp(bt.GetRoot())
}

// 二叉树中某一个结点的双亲 从根结点处出发找到 cur 的双亲结点
template <class ElemType>
BinTreeNode<ElemType>
BinaryTree<ElemType>::ParentHelp(BinTreeNode<ElemType>*r,
BinTreeNode<ElemType> *cur){
    if(r == NULL){
        return NULL;
    }
    else {
        if(r->rightChild == cur || r->leftChild == cur){
            return r;
        }
        else{
            BinTreeNode<ElemType> *tmp;
            tmp = ParentHelp(r->leftChild);
            if(tmp != NULL) return tmp;
            tmp = ParentHelp(r->rightChild);
            if(tmp != NULL) return tmp;

            return NULL;
        }
    }
}

// 展示二叉树
    // way 1 以左右孩子表示法 注意这个 level 是 Node 的层次数
template <class ElemType>
```

```
void BinaryTree<ElemType>::DisplayHelp(BinTreeNode<ElemType>*r, int
level){
    if(r != NULL){
        DisPlayHelp(r->rightChild, level + 1);
        cout << endl;            // 很关键
        for(int i = 0; i < level - 1; i++){
            cout << " ";
        }
        cout << r->data;                 //显示结点
        DisPlayHelp(r->leftChild, level + 1);
    }
}
template <class ElemType>
void BinaryTree<ElemType>::Display(BinaryTree<ElemType> &bt){
    DisplayHelp(bt.GetRoot(), 1);
}
    // way 2 以另一种奇怪的方式展示 只不过将 cout 的代码换一下位置就ok了 整
体架构还是不变的


// 创造二叉树 —— 通过前序和中序
template <class ElemType>
void BinaryTree<ElemType>::CreatBTHelp(BinTreeNode<ElemType>*&r,
ElemType pre[], ElemType in[], int preLeft, int preRight, int
inLeft, int inRight){
    if(preLeft > preRight){
        r = NULL;
    }
    else{
        r = new BinTreeNode<ElemType>(pre[preLeft]);// 根据preLeft
创建根
        int mid = inLeft;
        while(in[mid] != pre[preLeft]){
            mid++;
        }
        CreatBTHelp(r->leftChild, pre, in, preLeft+1, preLeft +
mid-inLeft, inLeft, mid-1);
```

```
        CreatBTHelp(r->rightChild, pre, in, preLeft + mid-inleft+1,
preRight, mid + 1, inRight);
    }
}
```

3 列表相关代码

```
// 链表的合并与转置
// 先将两个递增的链表合并 并消除重复，再变为递减
template <class ElemType>
void reverse(LinkList<ElemType> &l){
    int mid = l.Length() / 2;
    int lPos = 1;
    int rPos = l.Length();

    ElemType lItem, rItem;
    while(lPos <= mid){
        l.GetElem(lPos, lItem);
        l.GetElem(rPos, rItem);
        l.setElem(lPos, rItme);
        lPos++;
        l.setElem(rPos, lItem);
        rPos--;
    }
}

template<class Elemtype>
void Merge(LinkList<ElemType> &la, LinkList<ElemType>&Lb,
LinkList<ElemType>&lc){
    ElemType aItem, bItem, cItem;
    int aPos = bPos = 1;
    lc.Clear();         // 这一句千万别忘了！！！

    while(aPos <= la.Length() && bPos <= lb.Length){
        la.GetItem(aPos, aItem);
        lb.GetItem(bPos, bItem);
        if(aItem < bItem){
            lc.Insert(lc.Length() + 1, aItem);
            aPos++;
```

```
        }
        else if(aItme > bItem){
            lc.Insert(lc.Length() + 1, bItem);
            bPos++;
        }
        else{
            lc.Insert(lc.Length() + 1, aItem);
            aPos++;
            bPos++;
        }
    }

    while(aPos <= la.Length()){
        la.GetItem(aPos, aItem);
        lc.Insert(lc.Length() + 1, aItem);
        aPos++;
    }
    while(bPos <= lb.Length()){
        lb.GetItem(bPos, bItem);
        lc.Insert(lc.Length() + 1, bItem);
        bPos++;
    }

    reverse(lc);
}
```

3.1 Part 3 排序

```
// 快速排序
// 一定要想好手写是怎么手写的奥
/* - 第一个函数 找到核心轴
   - 第二个函数 做递归
   - 第三个函数 真正的接口 */

template <class ElemType>
int Partition<ElemType elem[], int low, int high>{
    while(low < high){
        while(low < high && elem[low] <= elem[high]){
            high--;
```

```
        }
        Swap(elem[low], elem[high]);

        while(low < high && elem[low] <= elem[high]){
            low++;
        }
        Swap(elem[low], elem[high]);
    }
    return low;
}

template <class ElemType>
void QuickSortHelp(ElemType elem[], int low, int high){
    if(low < high){
        int pivotLoc = Partition(elem, low, high);
        QuickSortHelp(elem, low, pivotLoc - 1);
        QuickSortHelp(elem, pivotLoc, high);
    }
}

template <class ElemType>
void QuickSort(ElemType elem[], int n){
    QuickSort(elem, 0, n-1);
}

// 堆排序
/* 主函数中只需要两步
    - Step 1 调整顶堆
    - Step 2 排序
*/
template <class ElemType>
void SiftAdjust(ElemType elem[], int low, int high){
    for(int f = low, i = 2*low + 1; i <= high; i = 2*i+1){
        if(i < high && elem[i] < elem[i+1]){
            i++;
        }
        if(elem[f] >= elem[i]){
            break;
```

```
        }
        Swap(elem[f], elem[i]);
        f = i;
    }
}

template<class ElemType>
void HeapSort(ElemType elem[], int n){
    for(int i = (n-2)/2; i >= 0; --i){
        SiftAdjust(elem, i, n-1);
    }
    for(i = n -1; i > 0; --i){
        Swap(elem[0], elem[i]);
        SiftAdjust(elem, 0, i-1);
    }
}
```