南開大學

《数据结构》

实验报告

实验（ 四 ）

年　　级: 2023 级

专　　业: 计算机科学与技术

姓　　名: 袁田

学　　号: 2314022

一．实验内容

实现以下内容：

1. 实现一个基于链表的二叉树，可以按照给定格式字符串生成对应二叉树。
2. 删除二叉树第 K 层所有节点后重新调整成二叉树，调整方式不作要求。
3. 给定一个元素值，找出所有和给定元素值节点距离为 K 的节点集合（节点中元素可能重复）。
4. 使用二叉树前序遍历和中序遍历生成一个二叉树。（节点中元素不重复）

二．设计思想

T1.由于样例中给出的是关于二叉树层次遍历的结果，其中为空的结点记作 null，其余给出了结点的具体数值。因此我利用 node 结构体储存数值，左指针(指向左子树)，右指针(指向右子树)，将给出的字符串取得其中为数值和为 null 的放置到数组中，且 null 的值记作-1.构建 Insert 函数用于链接。

T2.在 node 中新增了 id 元素，记录每个节点的 id，id 值从 1 开始，id 为 i 的结点对应到数组中的位置为 i-1，这样第 k 层的元素的 id 值范围为 2^(k-1)到 2^(k)-1，构建新的一棵树，插入时如果 id 为上述范围中则不进行插入。

T3.设计一个函数确定任意两个节点之间的最短路径，然后将书中所有结点的指针放置到数组 a 中，将给定值的结点指针放到数组 b 中，进行遍历，确定 a 中任意节点和 b 中任意节点的距离为 k，则输出其对应值

T4.利用递归的思想，通过前序和中序遍历依次找到左右子树的前序遍历和中序遍历，再分别重复上述操作，便可以构造得到二叉树，再输出其后序遍历由于检验

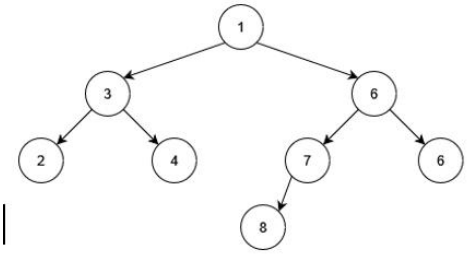
三．程序效果

程序的运行效果，输入及输出的相关要求和具体执行结果如下所示：

Part1. 实现一个基于链表的二叉树，可以按照给定格式字符串生成对应二叉树。

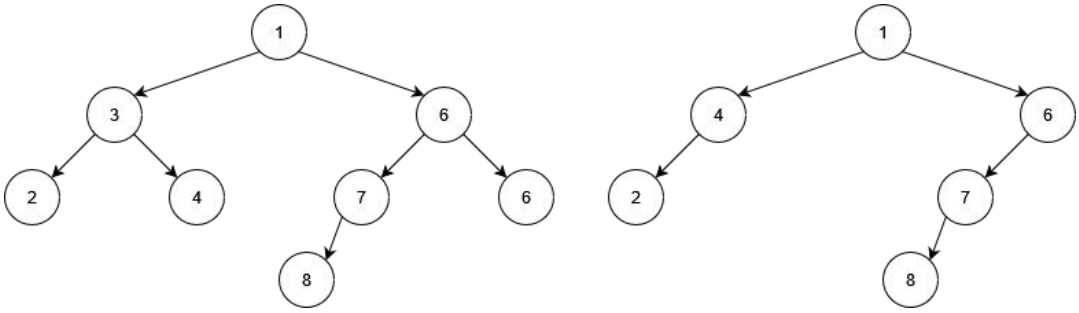
输入：[[1],[3,6],[2,4,7,6],[null,null,null,null,8,null,null,null]]

输出：

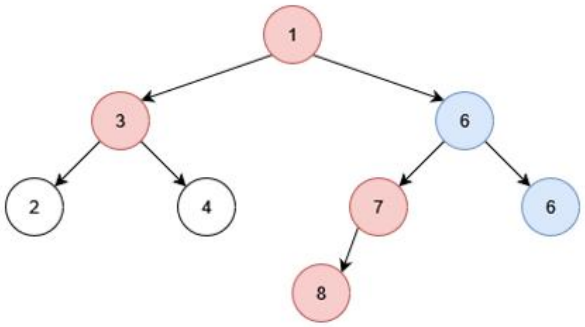


一般通过其前序遍历和中序遍历值来确定其是否正确

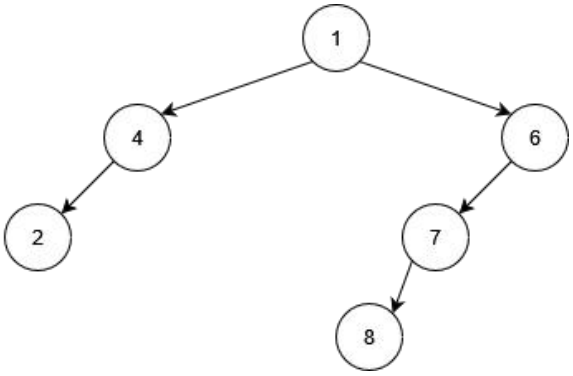
Part2. 删除二叉树第 K 层所有节点后重新调整成二叉树，调整方式不作要求。
输入：层数，假设为 2
输出：也是通过前序遍历和中序遍历来判断正确性的



Part3. 给定一个元素值，找出所有和给定元素值节点距离为 K 的节点集合（节点中元素可能重复）.
输入：如 6，2(其中 6 为给定元素值，2 为节点间距离)
输出：



Part4. 使用二叉树前序遍历和中序遍历生成一个二叉树。（节点中元素不重复）
输入：前序遍历：1 4 2 6 7 8 ； 中序遍历：2 4 1 8 7 6
输出：



通过后序遍历来检验正确性

四．核心代码

T1．part1.构建 node 结构体和 Insert 函数

```
struct node{
    int data;
    int id;
    int layer;
    node* left;
    node* right;
    node* father;
    node(int val,int new_id,int new_layer)
    {
        data=val;
        id=new_id;
        layer=new_layer;
        left=NULL;
        right=NULL;
    }
};
```

```
void Insert(node* root,int val)
{
    queue< node* >q;
    q.push(root);
    while(!q.empty())
    {
        node* tmp=q.front();
        q.pop();
        if(tmp->left==NULL)
        {
            int tmp_layer=tmp->layer;
            int tmp_id=tmp->id;
            tmp->left=new node(val,tmp_id*2,tmp_layer+1);
            tmp->left->father=tmp;
            return;
        }
        else
        {
            q.push(tmp->left);
        }
        if(tmp->right==NULL)
        {
            int tmp_layer=tmp->layer;
            int tmp_id=tmp->id;
            tmp->right=new node(val,tmp_id*2+1,tmp_layer+1);
            tmp->right->father=tmp;
            return;
        }
        else
        {
            q.push(tmp->right);
        }
```

Part2.其中 tarray 为放置了节点数值的数组，依次进行插入操作

这里是在前序遍历和中序遍历函数中进行了分类：若值为-1，视作该节点为空跳过

```
int root_id=1;
int root_layer=1;
string s;
getline(cin,s);
int tarray[100]={};
int len=0;;
for(int i=0;i<s.size();i++)
{
    if(s[i]=='['||s[i]==']'||s[i]==',') continue;
    int tmp=0;
    while(s[i]>='0'&&s[i]<='9')
    {
        tmp=tmp*10+s[i]-'0';
        i++;
    }
    if(s[i]=='n'&&s[i+1]=='u'&&s[i+2]=='l'&&s[i+3]=='l')
    {
        tmp=-1;
        i+=4;
    }
    tarray[len]=tmp;
    len++;
}

node* root=new node(tarray[0],root_id,root_layer);
for(int i=1;i<len;i++)
{
    Insert(root,tarray[i]);
}
FrontOrder(root);
cout<<endl;
MiddleOrder(root);
cout<<endl;
BackOrder(root);
```

用 FrontOrder 和 MiddleOrder 函数来检验树的正确性

T2.相比于 t1 没有较大区别，只是判定了一下层数对应的相关节点，不进行插入操作

```
if(k==1)
{
    new_root=new node(tarray[1],root_id,root_layer);
    for(int i=2;i<len;i++)
    {
        Insert(new_root,tarray[i]);
    }
}
else
{
    new_root=new node(tarray[0],root_id,root_layer);
    int sta=pow(2,k-1)-1;
    cout<<sta<<" "<<tarray[sta]<<endl;
    int end=pow(2,k)-1;
    cout<<end<<" "<<tarray[end]<<endl;
    for(int i=1;i<sta;i++)
    {
        Insert(new_root,tarray[i]);
    }
    for(int i=end;i<len;i++)
    {
        Insert(new_root,tarray[i]);
    }
}
```

T3.part1.构建 GetDistance 函数，确定任意两个结点之间的距离

```
vanode[valen]=tmp->left;
valen++;
```

```
vanode[valen]=tmp->right;
valen++;
```

```
int GetDistance(node* a,node* b)
{
    int aid[100]={};
    int bid[100]={};
    node* atmp=a;
    int alen=0;
    while(atmp!=NULL)
    {
        aid[alen]=atmp->id;
        atmp=atmp->father;
        alen++;
    }

    node* btmp=b;
    int blen=0;
    while(btmp!=NULL)
    {
        bid[blen]=btmp->id;
        btmp=btmp->father;
        blen++;
    }
    for(int i=0;i<alen;i++)
    {
        for(int j=0;j<blen;j++)
        {
            if(aid[i]==bid[j])
            {
                int d=i+j;
                return d;
            }
        }
    }
    return -1;
```

并且在插入函数中，将每个结点的指针都放入到指针数组 vanode 中

Part2.将值为 val 的结点指针放到数组 val_arr 中

```
node* val_arr[100]={};
int val_len=0;
for(int i=0;i<valen;i++)
{
    if(vanode[i]->data==val)
    {
        val_arr[val_len]=vanode[i];
        val_len++;
    }
}
```

Part3.进行 for 循环遍历，若 vanode 中的结点 a 和 val_arr 中的结点 b 的距离恰好为给定值 k，则将 a 指针指向的结点的值输出出来。 且为了避免输出重复，构建 flag 数组确定输出不重复。

```
for(int i=0;i<val_len;i++)
{
    for(int j=0;j<valen;j++)
    {
        if(GetDistance(val_arr[i],vanode[j])==k)
        {
            if(vanode[j]->data!=-1&&flag[j]==0)
            cout<<vanode[j]->data<<" ";
            flag[j]=1;
        }
    }
}
```

T4.设计 build 函数，进行递归操作

```
//先构建左子树中的前序和中序遍历结果
for(int i=0;i<len;i++)
{
    if(mid[i]!=root_val)
    {
        mid_left[len_left]=mid[i];
        len_left++;
    }
    else
    {
        break;
    }
}
for(int i=1;i<=len_left;i++)
{
    pre_left[i-1]=pre[i];
}
build(pre_left,mid_left,new_root,len_left,'l');
```

```
//再构建右子树中的前序和中序遍历结果
for(int i=len_left+1;i<len;i++)
{
    mid_right[len_right]=mid[i];
    len_right++;
}
len_right=0;
for(int i=len_left+1;i<len;i++)
{
    pre_right[len_right]=pre[i];
    len_right++;
}
build(pre_right,mid_right,new_root,len_right,'r');
```

五．总结

通过本次实验，我对于树的各项功能有了更加深入的理解：

1. 并查集可以很容易的得到两个结点是否具有亲戚关系，但是并不能获得两个节点的最短路径，给树的每个节点都设定一个 id 可以通过得到每个结点的祖先 id 数组来获得结点最短路径
2. 树是一个递归的数据结构，无论是前中后序遍历还是构建一棵树，大部分时候都利用了树递归得到的性质