二叉树——数据结构

树

树的定义

树(Tree)是n(n>=0)个结点的有限集。n=0时称为空树。在任意一颗非空树中：

1. 有且仅有一个特定的称为根(root)的结点；
2. 当n>1时，其余结点课分为m(m>0)个互不相交的有限集，其中每一个集合本身又是一棵树，并且称为根的子树。

结点

- 结点拥有的子树数称为结点的度(Degree)。
- 度为0的结点称为叶结点或者终端结点。
- 树的度是树内各结点的度的最大值。
- 结点的子树的根称为该结点的孩子(Child)，相应的，该结点称为孩子的双亲(Parent)。
- 同一个双亲的孩子之间互称兄弟。

树的其他概念

- 结点的层次从根开始定义起，根为第一层，根的孩子为第二层。
- 树中结点的最大层次称为树的深度(Depth)或高度。
- 森林是m(m>=0)棵互不相交的树的集合。

树的表示

孩子兄弟表示法

任意一棵树，它的结点的第一个孩子如果存在就是唯一的，它的右兄弟如果存在也是唯一的。因此，我们设置两个指针，分别指向该结点的第一个孩子和此节点的右兄弟。

```
typedef int DataType;
typedef struct Node
{
    struct Node* firstchild; //第一个孩子结点
    struct Node* nextbrother; //下一个兄弟结点
    DataType data; //结点的数据域
}
```

二叉树

二叉树的定义

二叉树是n(n>=0)个结点的有限集合，该集合或者为空集(称为空二叉树)，或者由一个根节点和两课互不相交的、分别称为根节点的左子树和右子树的二叉树组成。

二叉树的特点

- 每个结点最多有两棵子树，所以二叉树中不存在度大于2的结点。
- 左子树和右子树是有顺序的，次序不能任意颠倒。
- 即使树中某结点只有一颗子树，也要区分它是左子树还是右子树。

二叉树的性质

1. 在二叉树的第i层上至多有2^(i-1)个结点。(i>=1)
2. 深度为k的二叉树至多有2^k -1个结点。(k>=1)
3. 对任何一颗二叉树T，如果其终端结点数为n0，度为2的节点数为n2，则n0=n2+1。
4. 具有n个结点的完全二叉树的深度为log2n +1.

二叉树的遍历

二叉树的遍历是指从根节点出发，按照某种次序依次访问儿二叉树中的所有结点，使得每个结点都被访问且仅被访问一次。

1. 前序遍历

   若二叉树为空，则空操作返回，否则先访问根结点，然后遍历左子树再遍历右子树。

2. 中序遍历

   若二叉树为空，则空操作返回，否则从根节点开始，中序遍历根节点的左子树，然后是访问根结点，最后中序遍历右子树。

3. 后序遍历

   若二叉树为空，则空操作返回，否则从左到右先叶子后结点的方式遍历访问左右子树，最后是访问根结点。

4. 层序遍历

   若二叉树为空，则空操作返回，否则从数的第一层，也就是根结点开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序队结点逐个访问。

特殊的二叉树

1. 斜树

   所有的结点都只有左子树的二叉树叫左斜树。所有结点都是只有右子树的二叉树叫右斜树。

2. 满二叉树

   在一颗二叉树中，如果所有的分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。

3. 完全二叉树

   对一颗具有n个结点的二叉树按层序编号，如果编号为i(1<=i<=n)的结点与同样深度的满二叉树中编号为i的结点在二叉树中位置完全相同，则这颗二叉树称为完全二叉树。

二叉树的顺序存储

物理结构上用数组存储，一个结点的下标为i，则这个结点的左孩子下标：2×i+1，右孩子下标：2×i+2，父亲下标：(i-1)/2;

二叉树的链式存储

```
typedef char DataType;
typedef struct BinaryTreeNode
{
    DataType data;
```

```c
    struct BinaryTreeNode* left;
    struct BinaryTreeNode* right;
}BTNode;

// 通过前序遍历的数组"ABD##E#H##CF##G##"构建二叉树
BTNode* BinaryTreeCreate(DataType* a, int n, int* pi)
{
    //判断当前结点是否为空，若为空结点则返回NULL
    if (a[*pi] == '#' || *pi >= n)
    {
        (*pi)++;
        return NULL;
    }
    //当前结点非空，创建当前结点
    BTNode* root = (BTNode*)malloc(sizeof(BTNode));
    root->data = a[*pi];
    //字符位置向后移动一个位置
    (*pi)++;
    //创建左子树
    root->left = BinaryTreeCreate(a, n, pi);
    //创建右子树
    root->right = BinaryTreeCreate(a, n, pi);
    return root;
}
// 二叉树销毁
void BinaryTreeDestory(BTNode** root)
{
    //如果树不为空
    if (*root)
    {
        //销毁左子树
        BinaryTreeDestory(&(*root)->left);
        //销毁右子树
        BinaryTreeDestory(&(*root)->right);
        //释放结点
        free(*root);
        //置空
        *root = NULL;
    }
}
// 二叉树节点个数
int BinaryTreeSize(BTNode* root)
{
    //当树为空时，结点个数为0，否则为根结点个数 加上根的左子树中结点个数
    //再加上根的右子树结点个数
    int count = 0;
    if (root)
    {
        count = 1 + BinaryTreeSize(root->left) + BinaryTreeSize(root->right);
    }
    else
    {
        return 0;
    }
    return count;
}
// 二叉树叶子节点个数
```

```c
int BinaryTreeLeafSize(BTNode* root)
{
    //当树为空时，叶子结点个数为0
    //当某个结点的左右子树均为空时，此结点是叶子结点，返回1
    int count = 0;
    if (!root)
    {
        return 0;
    }
    else if (root->left == NULL && root->right == NULL)
    {
        return 1;
    }
    else
    {
        count = BinaryTreeLeafSize(root->left) + BinaryTreeLeafSize(root->right);
    }
    return count;
}
// 二叉树第k层节点个数
int BinaryTreeLevelKSize(BTNode* root, int k)
{
    //如果树为空或者K小于等于0，返回0
    if (root == NULL || k <= 0)
    {
        return 0;
    }
    //树不为空且K等于1，返回1
    if (root != NULL && k == 1)
    {
        return 1;
    }
    return BinaryTreeLevelKSize(root->left, k - 1) + BinaryTreeLevelKSize(root->right, k - 1);
}
// 二叉树查找值为x的节点
BTNode* BinaryTreeFind(BTNode* root, DataType x)
{
    //当前结点是否为空
    if (!root)
    {
        return NULL;
    }
    if (root->data == x)
    {
        return root;
    }
    //当前结点不为空也不等于x,遍历左子树
    BTNode* tmp = BinaryTreeFind(root->left, x);
    if (!tmp)
    {
        return tmp;
    }
    else
    {
        //左子树返回空，遍历右子树
        return tmp = BinaryTreeFind(root->right, x);
```

```c
    }
}
// 二叉树前序遍历
void BinaryTreePrevOrder(BTNode* root)
{
    //如果树不为空
    if (root)
    {
        //访问根结点
        putchar(root->data);
        //遍历左子树
        BinaryTreePrevOrder(root->left);
        //遍历右子树
        BinaryTreePrevOrder(root->right);
    }
}
// 二叉树中序遍历
void BinaryTreeInOrder(BTNode* root)
{
    //如果树不为空
    if (root)
    {
        //中序遍历根节点的左子树
        BinaryTreeInOrder(root->left);
        //访问根结点
        putchar(root->data);
        //中序遍历右子树
        BinaryTreeInOrder(root->right);
    }
}
// 二叉树后序遍历
void BinaryTreePostOrder(BTNode* root)
{
    if (root)
    {
        //遍历左子树
        BinaryTreePostOrder(root->left);
        //遍历右子树
        BinaryTreePostOrder(root->right);
        //访问根结点
        putchar(root->data);
    }
}
```

堆

堆是一颗完全二叉树，堆中某个结点的值总是不大于或不小于其父节点的值。根结点最大的堆叫做大根堆，根结点最小的堆叫做小根堆。

堆的性质

- 堆中某个节点的值总是不大于或不小于其父节点的值；
- 堆总是一棵完全二叉树；

堆的实现

从0开始对结点进行编号，寻找其中父子结点之间索引的对应关系。

首先，通过父结点的索引找出子结点的索引，设父结点的索引为i，则其左孩子结点的索引：2×i+1，右孩子结点的索引：2×i+2；

然后通过子结点的索引来找父结点的索引，设子结点的索引为i，则其父结点的索引为(i-1)/;

这样通过子结点与父结点之间的索引关系，便相当于建立了父结点和子结点之间的指针，实现了用数组来存储堆。

```c
typedef int DataType;
typedef struct Heap
{
    DataType* arr;
    int size;
    int capacity;
}Heap;

//向上调整算法
void AdjustUp(DataType* a, int n, int child)
{
    int parent = (child-1)/2;
    while (child > 0)
    {
        //如果孩子大于双亲，进行交换
        if (a[child] > a[parent])
        {
            DataType tmp = a[parent];
            a[parent] = a[child];
            a[child] = tmp;
            //调整，进行下一次交换
            child = parent;
            parent= (child - 1) / 2;
        }
        else
        {
            break;
        }
    }
}
//向下调整算法:左子树是小堆，右子树也是小堆
void AdjustDown(DataType* a, int n, int root)
{
    int parent = root;
    int child = parent * 2 + 1;
    while (child < n)
    {
        //找出左右孩子中小的那一个
        if (child + 1 < n && a[child + 1] < a[child])
        {
            child++;
        }
        //如果孩子比双亲还小，则将小的一个孩子结点与双亲结点进行交换
        if (a[parent] > a[child])
        {
            DataType tmp = a[parent];
            a[parent] = a[child];
            a[child] = tmp;
```

```
            //调整，进行下一次交换
            parent = child;
            child = parent * 2 + 1;
        }
        else//孩子比双亲大，则终止调整
        {
            break;
        }
    }
}
// 堆的构建
void HeapCreate(Heap* hp, DataType* a, int n)
{
    hp->arr = (DataType*)malloc(sizeof(DataType)*n);
    hp->size = n;
    hp->capacity = n;

    //建堆:调用向下调整算法，从最后一个结点的双亲开始
    for (int i = (n - 1 - 1) / 2; i >= 0; i--)
    {
        AdjustDown(hp->arr, hp->size, i);
    }
}
// 堆的销毁
void HeapDestory(Heap* hp)
{
    free(hp->arr);
    hp->arr = NULL;
    hp->size = hp->capacity = 0;
}
// 堆的插入
void HeapPush(Heap* hp, DataType x)
{
    //检查容量
    if (hp->size == hp->capacity)
    {
        hp->capacity *= 2;
        hp->arr = (DataType*)realloc(hp->arr, sizeof(DataType)*hp->capacity);
    }
    //尾插
    hp->arr[hp->size] = x;
    hp->size++;
    //向上调整
    AdjustUp(hp->arr, hp->size, hp->size-1);
}
// 堆的删除
void HeapPop(Heap* hp)
{
    //交换
    DataType tmp = hp->arr[0];
    hp->arr[0] = hp->arr[hp->size - 1];
    hp->arr[hp->size - 1] = tmp;
    //向下调整
    AdjustDown(hp->arr, hp->size, 0);
}
// 取堆顶的数据
DataType HeapTop(Heap* hp)
```

```c
{
    return hp->arr[0];
}
// 堆的数据个数
int HeapSize(Heap* hp)
{
    return hp->size;
}
// 堆的判空
int HeapEmpty(Heap* hp)
{
    if (hp->size == 0)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
// 对数组进行堆排序
// 升序建小堆，降序建大堆
void HeapSort(int* a, int n)
{
    //排序需要建大堆：
    //因为每次都会把堆顶元素拿出来放到当前堆的最后一个位置
    //相对于每次都会把剩余元素当中的最大值(即堆顶元素)找出来，放到当前堆的最后一个位置
    for (int i = (n - 1 - 1) / 2; i >= 0; i--)
    {
        AdjustDown(a, n, i);
    }
    while (n - 1 > 0)
    {
        DataType tmp = a[0];
        a[0] = a[n - 1];
        a[n - 1] = tmp;
        //调堆，选次大的数
        AdjustDown(a, n - 1, 0);
        n--;
    }
}
```