考研数据结构

绪论

基本概念

数据

保存信息的载体

数据元素

数据对象的基本单位。

数据项

数据元素的最小单位。

数据对象

具有相同性质的数据元素。

数据类型

原子类型

其值不可再分

结构类型

其值可再分

抽象数据类型

抽象数据和其对应的操作所组成

数据结构的定义

其数据元素之间具有一种或多种**特定关系的集合**。且数据元素之间的关系称之为**结构。**

数据结构的定义由**逻辑、存储结构和数据的运算**三部分组成，所以其数据类型表示最好是抽象数据类型。

注意：

一个**算法的设计**基于数据的逻辑结构；**算法的实现**基于数据的存储结构。

数据结构三要素

数据的逻辑结构

线性结构

线性表：普通顺序表、栈和队列、数组、串。

非线性结构

非线性表：集合、树形结构、图的结构。

从逻辑上来说：线性结构就是一个萝卜一个坑；非线性结构就是不同品种萝卜的种在不同的田。

数据的存储结构

顺序存储结构

数据对象的数据元素在逻辑上和物理上的存储是连续的。

链式存储结构

数据元素之间逻辑上连续，物理上不连续。

索引存储结构

利用索引表，构建的索引项（由数据的关键字和地址组成）

散列存储结构

又称Hash存储，使用关键字确定数据元素的存储地址。

数据的运算

在数据上的运算的定义

作用于逻辑结构，指出运算的功能。

在数据上的运算的实现

作用于存储结构，指出具体的操作步骤。

算法

五个特征：有穷性（算法有在有限时间内执行和退出、可行性（算法的操作可理解性、确定性（步骤明确、输入、输出

五个目标：正确性、可读性、健壮性、效率和低存储量要求。

时间复杂度

一个语句的频度：指一个算法中一条语句被重复执行的次数。

$T(n)$：指的算法中所有语句的频度之和。

通常来说，我们指算法复杂度，是指算法中，依赖问题规模n(输入量)，增长速度最快的函数$f(n)$的频度$O(f(n))$，且**与问题规模n成正比**，所以说$T(n) = O(f(n))$

例子：递归调用

对于线性递归函数，其$f(n) = f(n-1) + c$，其$O(f(n+c) = O(n)$

对于非线性递归函数，其$f(n) = f(n-1) + f(n-2)$，其调用结构更类似于**执行树**，那么久类似于完全二叉树计算结点数,当树具有n层，其结点总数=2n-1,故其时间复杂度为$O(2n-1)$

空间复杂度

算法所需耗费的存储空间$S(n) = O(g(n))$

其包含了算法中的变量、常量、指令和输入数据外，还包含了对数据进行操作时所需的辅助空间。

线性表

顺序表

是相同数据类型的有限序列。

存储结构为逻辑和物理上元素都是相邻。

位序和数组下标区别

位序：是指数组索引从1开始。

数组下标：是指数组索引从0开始。

静态分配

使用数组进行建立。

缺点：

数组大小不可更改。

动态分配

使用指针，malloc开辟连续堆空间存储，不使用时需要使用free释放空间。

缺点：

增加拓展顺序表时，需要使用辅助空间，导致空间复杂度高。

特点

随机访问：访问元素的时间复杂为O(1)。

存储密度高：动态分配中不仅有data还有该data指针(存储的地址)。

拓展容量不方便。

插入、删除元素不方便。

操作

创建表并初始化、删除表

createlits

initatelist

destroylist

插入、删除某元素

insertlist

deletelist

查找

按位查找

按值查找

总代码

```
#include<iostream>
using namespace std;
#define Maxsize 10


//创建动态分配顺序表
typedef  struct {
    int* data;
    int length;
}sqlist;

//创建静态分配顺序表
typedef  struct {
    int data[Maxsize];
    int length;
}matrix_sqlist;

//初始化顺序表
//静态类型
template<typename T> void InitialList(T& l,int Init_len)
{
    for (int i = 0; i < Init_len; i++) {
        l.data[i] = 0;
    }
    l.length = Init_len;
    cout << "Initial matrix_sqlist is ok!" << endl;
}

//销毁动态顺序表
template < class T > bool DestroyList(T& l) {
```

```
    free(l.data);
    return true;
}

//求表长
template<class T> int LenList(T l) {
    return l.length;
}

//查询表是否为空
template < class T > bool isEmptyList(T l) {
    if (l.length == 0) {
        return true;
    }
    return false;
}


//插入新元素于顺序表中，使用从尾部移动，腾出位序i位置的方法
template <class T> bool InsertElem(T& l, int index, int e) {
    //如果插入元素的位序小于1或大于数组最大长度，返回false
    if (index < 1 || index > Maxsize - 1) {
        return false;
    }
    //如果顺序表的长度已经达到最大，则无法插入新元素
    if (l.length == Maxsize) {
        return false;
    }

    for (int j = l.length; j > index; j--) {
        l.data[j] = l.data[j - 1];
    }
    l.data[index] = e;
    ++l.length;
    return true;
}

//删除元素
template <class T> bool DeleteElem(T& l, int index) {
    if (index < 1 || index > Maxsize - 1) {
        return false;
    }
    for (int j = index; j < l.length - 1; ++j) {
        l.data[j] = l.data[j + 1];
    }
    --l.length;
    return true;
}

//按位查找
template < class T> int GetElem(T l, int index) {
    if (index < 1 || index > Maxsize - 1) {
        return 0;
    }
    return l.data[index-1];
```

```
}



//按值查找
template <class T > auto LocateElem(T l, int value) -> decltype(value) {
    ;//return  -- > index
    for (int i = 0; i < l.length; ++i) {
        if (l.data[i] == value) {
            return i;
        }
    }
 }

//打印顺序表中已有元素
template<class T> void PrintList(T l) {
    for (int i = 0; i < l.length ; ++i) {
        cout << l.data[i];
    }
    cout << endl;
}


int main() {
    //测试静态顺序表
    matrix_sqlist mq;
    InitialList(mq,3);
    InsertElem(mq, 1, 1);
    InsertElem(mq, 2, 2);
    cout <<"顺序表长度:"<< LenList(mq) << endl;
    PrintList(mq);
    DeleteElem(mq, 1);
    PrintList(mq);
    cout <<"Find ELemt index:"<< LocateElem(mq, 2) << endl;;
    cout <<"The index is value in the sqlist:"<< GetElem(mq, 2) << endl;;

    //测试动态顺序表
    sqlist ql;
    ql.data = (int*)malloc(sizeof(int) * Maxsize);
    InitialList(ql,3);
    PrintList(ql);
}
```

树

区分:

m叉树和度为m的树：

度为m的树	m叉树
其至少有一个结点的度为m，任意结点的度≤m	任意节点的度≤m
其高度h时，至少有m+h-1个结点。	允许所有结点的度都小于<m
一定是非空树，至少有m+1个结点	可以为空树

高度为h的m叉树，至少有h个结点。

高度为h、度为m的树，至少有h+m-1个结点

树常考性质

1. 结点数的总度数=结点数-1
2. 度为m的树种第i层上至多有m^{i-1}个结点。
3. 高度为h的m叉树至多有$\frac{m^h-1}{m-1}$个结点(完全m叉树)
4. 具有n个结点的m叉树的最小高度$h=\log_m(n(m-1)+1)$

二叉树

完全二叉树

满二叉树可以是完全二叉树。

其特点是，若有度为1的结点，其只有左子树，没有右子树。

当前结点编号i=偶数时，父结点编号为$\frac{i}{2}$,左子树为$2i$，右子树为$2i-1$，当前结点为i/2的左孩子;

当前结点编号i=奇数时，父结点编号为$\frac{i-1}{2}$,左子树为$2i$，右子树为$2i-1$，当前结点为$\frac{i-1}{2}$的右孩子;

满二叉树

其所有结点的度为2，总结点个数为2^h-1(h为高度)。

二叉树常考性质

1. 在非空的二叉树的前提下，叶子结点个数=度为2的结点数+1。$n_0=n_2+1$
2. 非空二叉树第k层上至多有2^{k-1}个结点。
3. 高度为h的二叉树至多有2^h-1个结点。
4. 结点数为n的**完全二叉树**的高度$h=log_2(n+1)或者log_2n+1.$

性质4，前一个从二叉树的总结点数角度看(当前结点应比h-1层结点总数多，比h层结点总数小或等于($2^{h-1}-1\leq n<2^h-1)$)，后者从二叉树高度当前h的结点数看($2^{h-1}\leq n<2^h$)。

二叉树访问节点方式

对二叉树的结点进行先中后序遍历。

使用递归分支法

从根开始使用遍历法，对有子树的结点进行同方法展开。

使用递归调用时访问根结点次数法

对根节点记录的第一次访问是，前序遍历；

对根节点记录的第二次访问是，中序遍历；

对根节点记录的第三次访问是，后序遍历；

层序遍历

利用队列，保存每次访问结点的左右子树，从而保证根结点对子树是横向的节点访问。

```
while(! IsEmpty(p)){
    Dequeue(Q,p); //Q为队列，p为当前结点;
    operater(p);//对当前结点进行的操作
    if(p->lchild != NULL){
        EnQueue(Q,p->lchild);//将左孩子加入队列
    }
    if(p->rchild != NULL){
        EnQueue(Q,p->rchild);//将右孩子加入队列
    }
}
```

由二叉树的遍历序列构造二叉树

仅有一个遍历序列，是无法构造二叉树。且每个遍历序列组合应都有**中序**。

前序+中序遍历序列

由前序确定根结点（从左往右，中序确定左右子树结点集合

后序+中序遍历序列

由后序确定根结点（从右往左，中序确定左右子树结点集合

层序+中序遍历序列

由层序遍历（从左往右确定第一层的根结点和子树），依靠中序确定左右子树。

前序、后序、层序序列两两组合无法唯一确定二叉树。

线索二叉树

根据前、中、后序某一个序列，将二叉树n个结点的n+1个空指针域连其前驱和后继。

利用ltag和rtag标志，标识其左右子树是否为前驱或后继。

土方法寻找前驱：

使用pre树节点变量，存储当前面访问节点q的前驱。当访问节点q与所求节点p相等时，此时的pre为p的前驱。

```
void LDR(BiTree T){
if(T != NULL){
LDR(T->lchild);//遍历左子树
visit(T);//访问根结点
LDR(T->rchild);//遍历右子树
    }
}
void visit(BiTree *q){
    if(q == p)//是否递归到需要查找的结点
        final = pre;//是，此时的前驱为pre
    else
        pre = q;//否，将q作为前驱
}
```

线索化建立

```
//当前根结点处理 ， pre 保存当前结点前驱
void visit(BiTree* p,BiTree &pre){
    if(p->lchild == NULL){//左子树线索处理
        p->lchild = pre;
        p->ltag = 1;
    }

    if(pre != NULL && pre->rchild != NULL){//右子树线索处理
        pre->rchild  = p;
        pre->rtag = 1;
    }
    pre = p;

}
```

树和森林

树的先根遍历

类似于二叉树的先序遍历，先访问根结点，再类似于递归分支法访问左子树和右子树的根结点，循环往复。

树的后根遍历

类似于二叉树的后序遍历，先访问左子树和右子树的根结点，再根类似于递归分支法访问根结点结点，循环往复。

树的层序遍历

类似于二叉树的层序遍历

树边二叉树：

树的兄弟连线，并放在右孩子处。长子在左边。

口诀：左长子右兄弟

森林变二叉树：

先将每个树变为二叉树

哈夫曼树

又称最优二叉树。

是二叉树的应用:它的具有编码场景,将元素集的元素按权值排序,依次从权值集合中选择权值最小的两个.

在构造树时的口诀：左小右大。

在对已经构造好的哈夫曼树下进行编码标注：左0右1.

带权路径长度WPL

1.通过叶子结点计算

$WPL_{总} = \sum 叶子结点权值 * 所在层数$

2.通过数学思维，将所有子树节点相加

$WPL_{总} = \sum 除层数1的根结点外，其余所有结点权值新加$

哈夫曼编码长度

将每个字符的编码位数X权值的和。

等长编码长度：2^n是二进制能表示所有字符数，其长度为nx总权值,

图

一定是非空集合。线性表和树可以为空。

概念：

子图：普通子图：可以不包含全部的顶点和边。

生成子图：必须包含所有顶点，边可以不全部包含。

极大连通子图：指无向图的连通分量。

强连通子图：指有向图的强连通分量。

生成树：包含全部顶点的极小连通子图。

树

常考点：

n个顶点，|E|>n-1，则该图必有回路。

动态查找和静态查找的区别是：

动态查找会添加删除结点。而静态查找只会对原表仅查找操作。