数据结构与算法

一. 基础知识

术语:

1. 数据: 存储在计算机中可以用二进制表示的内容
2. 数据元素：在计算机中作为一个整体考虑和处理的对象 是组成数据的基本单位 由若干数据项组成
3. 数据对象: 性质相同的数据元素的集合
4. 数据结构: 研究数据与数据之间关系的一门学科
    1. 数据结构的逻辑结构: DS = (D, R) 数据集加关系集
    2. 数据结构的存储结构:
       四种基本的存储结构
        1. 顺序存储
        2. 链式存储
        3. 索引存储
        4. 哈希存储
    3. 数据的操作实现
5. 抽象数据类型(ADT): ADT = （D, R, P) 数据对象 + 关系 + 操作

算法的特性:

1. 有穷性
2. 确定性
3. 可行性
4. 有输入输出
5. 通用性
6. 可读性
7. 健壮性

算法分析

- 时间复杂度
- 空间复杂度

二. 线性表

1. **定义**: 由长度为n的一组结点组成的有限序列
2. **顺序存储结构**: 也称为顺序表
    1. 特点:
        - 逻辑顺序与物理顺序一致
        - 元素之间的关系用物理位置的相邻来体现
    2. 形式定义:

```
template <class Elem>
class  Alist: public list<Elem> {
```

```
    private:
        int maxSize, listSize, curr; // 线性表最大长度、表长、当前元素位置
        Elem * listArray;
    public:
        Alist(int size=DefaultListSize) {    //构造函数
            maxsize=size; listSize = curr =0;
                listArray = new Elem[ maxSize ];
        }
        ~Alist() {  delete [ ] listArray; }      // 析构函数
        void clear(){ listSize = curr =0; }      //清空线性表
        void Prev( ){ if (curr>0) curr--; }      //当前位置curr前移到前驱
        void Next( ){ if (curr<listSize-1) curr++;} //当前位置curr后移到后
继
        bool setPos(int pos);            //任意指定当前数据元素的位置
        bool insert(const Elem it);          //在当前位置插入元素
        bool append(const Elem it);      //在表尾插入元素
        bool remove( Elem &it);      //删除当前位置元素并返回其值
        ….        //其它操作的实现
};
```

3. 操作实现
    1. 插入操作
        1. 实现
        2. 时间复杂度分析
    2. 追加操作
        1. 实现
        2. 时间复杂度分析
    3. 删除操作
        1. 实现
        2. 时间复杂度分析

3. **链式存储结构：** 简称链表
    1. 特点：
        1. 插入和删除操作时不需要移动元素
        2. 不要求逻辑上相邻的元素在物理位置上也必须相邻
    2. 分类：
        1. **单链表**：
            1. 简述：单链表中每个元素用一个结点来存储，每个结点包括数据域和指针域，单链表的指针域中只包含一个指针
            2. 一些概念：
                1. 头结点：在链表最前端不存储任何数据元素的结点
                2. 头指针：指向头结点
                3. 空表：仅有一个头结点的单链表是空表
            3. 操作实现：
                1. 单链表节点以及单链表的定义
                2. 单链表的建立
                3. 读取元素
                4. 插入
                5. 删除

2. **双向链表**：
    1. 简述：双向链表的指针域中包括两个指针(前驱和后继)
    2. 操作实现:
        1. 双向链表节点以及双向链表的定义
        2. 插入
        3. 删除
3. **循环链表**：
    1. 如果是单向循环链表，尾节点的next指向头节点，如果是双向循环链表，尾节点的next指向头节点，头节点的pre指向尾节点
    2. 循环条件判定：当前节点是否等于头节点
    3. 操作实现：
        1. 定义
        2. 应用: 约瑟夫环
4. **两种方法的对比及改进**
    1. 对比：
        1. 空间上：顺序表没有浪费空间，链表存在指针域开销
        2. 访问上：顺序表可以随机访问，仅需O(1)复杂度，链表只能顺序访问，需要O(n)复杂度
        3. 插入删除上：顺序表插入删除复杂度为O(n), 链表插入删除复杂度为O(1)
    2. 改进：
        1. 空闲链管理空闲的空间
        2. 静态链表：静态链表的空间是一组地址连续的内存空间，数据元素包括数值域和下标域，每个元素所存的下标即为next。需要用一个空闲链来管理连续空间中未使用的空间
5. **线性表的应用**：
    1. 一元多项式的表示:
        1. 表示方法一：采用顺序表，用下标来表示指数，数据来表示相应系数，适用于密集多项式
        2. 表示方法二：采用链表，每个节点中存储指数和系数，适用于稀疏多项式
    2. 商品链的更新：采用链表

三.受限线性表

1. **栈:**
    1. 特点: 仅在表的尾部进行插入和删除的线性表，具有先进后出(FILO)、后进先出(LIFO)的特点
    2. 一些属性
        1. 栈顶：进行插入和删除操作的一端
        2. 出栈、入栈
    3. 存储结构：
        1. 顺序栈 -- 栈的顺序存储结构
            1. 简述：用数组存储数据，两个指针一个指向栈顶，一个指向栈底
            2. 溢出：上溢 -- 对满栈进行push 下溢 -- 对空栈进行pop **为了防止溢出，入栈出栈前要检查** 采用栈顶对栈顶，两个栈共用空间的方法可以提高空间利用率、降低上溢的发生概率
            3. 操作实现：

        1. 创建
        2. 入栈出栈 push() pop()
        3. 返回栈顶元素 top()
    2. 链栈 -- 栈的链式存储结构
        1. 简述：本质是一个单链表，但是它所能进行的操作是有限制的，只能在表头插入删除
        2. 头节点：若有头节点，那么栈顶元素存在头节点的next节点、栈顶指针指向头节点，否则，栈顶指针指向栈顶元素 **默认链栈不带头节点**
        3. 操作实现：
            1. 链栈节点以及链栈的定义
            2. 销毁
            3. 求长度
            4. 返回栈顶
            5. 入栈出栈
    4. **栈的应用**
        1. 括号匹配
            1. 思路简述：如果碰到左括号，就将其入栈；如果碰到右括号，就取出栈顶元素，判断二者是否匹配
            2. 代码实现：
        2. 递归和栈
            1. 每一个函数被调用时，为它在栈顶分配一个存储区，当函数返回时，就从栈顶释放存储区
            2. 函数返回的过程：1. 从栈顶弹出工作记录 2. 将工作记录中的参数赋给对应变量 3.将函数的返回值赋给对应变量 4. 转移到返回地址
        3. **表达式求值**：
            1. 将中序表达式转换为后序表达式
            2. 对后续表达式进行求值
2. **队列：**
    1. 特点：仅能在一端进行插入操作，而在另一端进行删除操作的线性表；具有先进先出(FIFO)、后进后出(LILO)的特点
    2. 一些属性和成员：
        1. 队尾：进行插入操作的一端
        2. 队头：进行删除操作的一端
        3. 入队、出队
    3. 操作实现：
        1. 创建
        2. 销毁
        3. 插入、删除
        4. 返回队头
        5. 返回长度
    4. 存储结构：
        1. 顺序队列：
            1. 简述：利用头指针(front)和尾指针(rear)来记录队列头和队列尾的位置
            2. 操作实现：

1. 入队：将元素放置rear指向的位置，然后将rear + 1
2. 出队：将front + 1
3. 判断是否为空：front == rear

2. 循环队列:
    1. 简述：头尾相连的顺序队列
    2. 操作实现：
        1. 入队：区别在于 rear = (rear + 1) % max_size
        2. 出队: 区别在于 front = (front + 1) % max_size
        3. 判断是否为空：同上
        4. 判断是否为满：front = (rear + 1) % max_size
3. 链队列

5. 队列的应用：
    1. 计算杨辉三角
    2. 火车车厢重排

3. **字符串**
    1. 定义：n个字符的一个有序序列
    2. 操作实现：
        1. 求长度
        2. 赋值
        3. 将两个串连接
        4. 比较大小
        5. 查找是否存在某字符或某字符串
        6. 截取生成子串
        7. 插入字符或字符串
        8. 删除字符或子串
    3. 存储结构：
        1. 静态存储(顺序存储)：
            1. 缺点：
                1. 需要预先定义大小，如果定义的max_length过大，则会浪费空间；
                2. 因为有最大字符数，所以会使某些可能改变长度的操作受到限制
        2. 动态存储(链式存储) :
            1. 类型：
                1. 每个节点存一个字符：优点是便于插入删除，缺点是浪费空间
                2. 每个节点存K个字符：优点是提高了存储密度，缺点是插入删除很不方便
    4. **模式匹配**：
        1. 定义：模式匹配是指 子串(模式串)在主串中的定位
        2. 算法：
            - Brute-Force算法：依次遍历，判断是否匹配
            - KMP算法：
                - 前缀表：
                    - 作用：在匹配发生错误时，回退到上次已经匹配过的地方，减少从头匹配

- 意义：用next数组来表示前缀表，对于一个字符串来说，它对应的next数组中，next[i]是从头开始的长度为i + 1的子串的最长相等前后缀的长度
- 如何求next数组

```
/*
求next数组 可以看成是模式串与模式串进行匹配的一个过程 当未匹配时利用前面已经完成的next数组进行跳转
*/
vector<int> next;
void getNext(string s) {
    int j = 0;// j是前缀末尾的下标
    next[0] = 0; // 显然next[0] = 0

    for (int i = 1; i < s.size(); ++i) { // i是后缀末尾的下标
        // 当s[i]和s[j]不匹配时 j就回退到之前已经匹配的位置
        while (j > 0 && s[i] != s[j]) j = next[j - 1];
        // 匹配成功时 将长度加一
        if (s[i] == s[j]) ++j;
        // 每个下标处都进行记录
        next[i] = j;
    }
}
```

- 如何查找

```
/*
haystack为主串 needle为模式串
*/
int strStr(string haystack, string needle) {
    int n = needle.size(), j = 0; // j是指向模式串当前位置的指针
    next.assign(n, 0);
    getNext(needle);
    // i是指向主串当前位置的指针
    for (int i = 0; i < haystack.size(); ++i) {
        // 匹配失败 回退
        while (j > 0 && haystack[i] != needle[j]) j = next[j - 1];
        // 匹配成功
        if (haystack[i] == needle[j]) ++j;
        // 成功找到模式串 返回起始位置
        if (j == needle.size()) return i - j + 1;
    }

    return -1;
}
```

四. 扩展线性表

1. **数组**:
2. 定义：由n个具有相同数据类型的数据元素组成的有序序列，且该序列必须存储在一块地址连续的存储单元中
3. 特点：元素类型相同　支持随机存取　元素个数固定
4. 操作实现：　随机存取　随机修改　初始化
5. 存储结构:
   . 静态数组(定义数组):
   1. 维数：一维　多维
   2. 压缩存储：多个值相同的矩阵元素分配到一个存储空间，值为0的矩阵元素不分配空间的存储方式
6. **广义表**:
7. 定义：广义表L是由n个元素组成的有穷序列，其中每个元素可以是原子项或者广义表
8. 概念:
   - 表头：非空广义表的第一个元素为表头，表头可以是原子或者广义表
   - 表尾：除去第一个元素外其他元素组成的表为表尾，表尾一定是广义表
9. 存储方式:
   1. 头尾链表存储:
      1. 节点分类:
         1. 表节点 由标识域 头指针域和尾指针域组成 (标识域是用来区分表节点和原子节点的)
         2. 原子节点 由标识域和值域组成
      2. 特点:
      3. 实现:
   2. 扩展线性链表存储:
      1. 节点组成:
         1. 标志域：用来表明节点类型　为0代表是附加头节点　为1代表是原子节点　为2代表是子表
         2. 信息域：节点为附加头节点时，存储引用计数；为原子节点时，存储数据值；为子表节点时，存储子表的表头指针
         3. 尾指针域：节点为附加头节点时，存放头指针；否则，存放同层下一个节点的地址
      2. 特点:
10. **递归操作:**
    1. 取表头 取表尾
    2. 复制
    3. 求长度
    4. 求深度

五. 树和二叉树

1. **树**:
   1. 定义：树是包含n个节点的有限集合，n = 0时为空树，n不为0是非空树，
      在一棵非空树T中，有以下两个特点:
      - 有且仅有一个特定的节点R作为树的根节点

- 除根节点外的节点可以被分为m个互不相交的有限集合T1 --> Tm, 其中每个集合本身也是一棵树，称为树T的子树，每个子树都有对应的根节点R1 -- > Rm，称为R的孩子

2. 术语：

1. 节点的度：节点拥有的子树个数
2. 树的度：树内节点的度的最大值
3. 叶子节点：度为0的节点
4. 分支节点：度不为0的节点
5. 孩子节点、双亲节点：树中节点的子树的根是该节点的孩子节点，该节点称为其孩子节点的双亲节点
6. 兄弟节点：父节点相同的节点
7. 堂兄弟节点：父节点在同一层的节点
8. 祖先：一个节点的祖先是从根节点到该节点所经分支上的所有节点
9. 子孙：一个节点的子孙是以该节点为根的子树中的所有节点
10. 节点的层次：根节点的层次为1，其余节点的层次为其双亲节点的层次加一
11. 树的深度：树中节点的最大层次
12. 树的高度：树的总层数
13. 有序树：将树中每个节点的各子树看成是从左到右有顺序的
14. 无序树：将树中每个节点的各子树看成是无顺序的
15. 森林：m(m >= 0)棵互不相交的树的集合

2. **二叉树**

1. 定义：二叉树是包含n个节点的有限集合，n = 0时为空二叉树，n != 0时为非空二叉树
   在一棵非空树T中，有以下两个特点:

- 有且仅有一个特定的节点R 称为二叉树T的根节点
- 除根结点外的其余节点 被分为两个互不相交的有限集合T1、T2，其中T1、T2也是一棵二叉树，这两棵二叉树分别称为 根节点R的左子树和右子树，左子树的根节点称为R的左孩子，右子树的根节点称为R的右孩子

2. 性质：

1. 在二叉树的第i层上至多有pow(2, i - 1)个节点 (i >= 1)
2. 高度为k的二叉树至多有pow(2, k) - 1个节点
3. 对于任意一个非空二叉树T, 如果其叶子节点的个数为n0，度为2的节点数为n2，则有 n0 = n2 + 1　证明： 记n为节点树，e为边数， 度为1的节点数为n1，
   则有 n = n0 + n1 + n2 = e + 1 = n1 + 2 * n2 + 1
4. 对一颗具有n个节点的完全二叉树的节点按层序从左往右编号，则对于任意一个节点 i (0 <= i <= n - 1)有：
   1. 如果2 * i + 1 >= n，则节点i为左孩子，否则，其左孩子的序号为2 * i + 1
   2. 如果2 * i + 2 >= n，则节点i为右孩子，否则，其右孩子的序号为2 * i + 2
   3. 如果 i = 0，则节点i是二叉树的根节点；如果 i > 0，则其双亲节点的序号为

$$\lfloor (i-1)/2 \rfloor$$

5. 具有n个节点的完全二叉树的高度 k =

$$\lfloor \log_2 n \rfloor + 1$$

3. **满二叉树和完全二叉树**

1. 满二叉树
   1. 定义: 一棵高度为k并且有 2^k - 1个结点的二叉树
   2. 性质:

        1. 每一层上的节点数都到达最大值
        2. 不存在度为1的节点
        3. 每个分支节点都有两棵高度相同的子树，所有叶子节点都在最下一层上
    3. 编号: 从0开始编号
2. 完全二叉树:
    1. 定义: 一棵二叉树每个节点的层序编号与对应位置的满二叉树的层序编号完全一致，则该树为完全二叉树
    2. 性质：
        1. 从满二叉树中连续删除m个元素，其层序编号为2^k - 1 - i，i：from 1 to m，所得二叉树也是完全二叉树
        2. 完全二叉树中至多只有最下面两层的上的节点的度数小于2，且最下层节点都集中在该层的最左边
        3. 完全二叉树中如果一个节点没有左孩子，那么它也没有右孩子
4. 二叉树的存储结构
    1. 顺序存储结构
        1. 简述：将二叉树中的所有节点按照一定的次序存储到一组地址连续的存储单元中
        2. 性质：编号从0开始，则对于节点 i 来说，
            1. 如果2 * i + 1 >= n，则节点i为左孩子，否则，其左孩子的序号为2 * i + 1
            2. 如果2 * i + 2 >= n，则节点i为右孩子，否则，其右孩子的序号为2 * i + 2
            3. 如果 i = 0，则节点i是二叉树的根节点；如果 i > 0，则其双亲节点的序号为

$$\lfloor (i-1)/2 \rfloor$$

        3. 如何存储一般的二叉树：对于一般的二叉树来说，将其按照完全二叉树的形式进行存储，其中如果某节点为空，则用特定值表示不存在
    2. 链式存储结构
        1. 简述：指二叉树的各节点随机的存储在内存空间中，节点之间的关系用指针表示
        2. 节点组成：至少包括 数据域，左指针域，右指针域
        3. 操作实现
            1. 节点类的声明
            2. 二叉树类的定义
5. **二叉树的遍历**

    二叉树节点定义

```
class TreeNode {
    TreeNode* left;
    TreeNode* right;
    int val;

    TreeNode(int v): val(v), left(nullptr), right(nullptr) { };
};
```

    1. 前序遍历
        1. 递归版本

```
void preOrder(TreeNode* root) {
    if (!root) return ;

    visit(root);
    preOrder(root->left);
    preOrder(root->right);
}
```

2. 非递归版本一

```
void preOrder(TreeNode* root) {
    stack<TreeNode*> ts;
    TreeNode* cur = root;
    if (cur) ts.push(cur);

    while (!ts.empty()) {
        TreeNode* tmp = ts.top();
        ts.pop();
        visit(tmp);

        if (tmp->right) {
            ts.push(tmp->right);
        }
        if (tmp->left) {
            ts.push(tmp->left);
        }
    }
}
```

3. 非递归版本二

```
void preOrder(TreeNode* root) {
    stack<TreeNode*> ts;
    TreeNode* cur = root;

    while (!ts.empty() || cur) { // 当指针为空且栈为空时 停止循环
        if (cur) { // 中 左
            visit(cur);
            ts.push(cur);
            cur = cur->left;
        }
        else { // 右
            cur = ts.top();
            ts.pop();
            cur = cur->right;
        }
    }
}
```

2. 中序遍历

1. 递归版本

```
void inOrder(TreeNode* root) {
    if (!root) return;

    inOrder(root->left);
    visit(root);
    inOrder(root->right);
}
```

2. 非递归版本

```
void inOrder(TreeNode* root) {
    stack<TreeNode*> ts;
    TreeNode* cur = root;

    while (!ts.empty() || cur) { // 当指针为空且栈为空时 停止循环
        if (cur) { // 左
            ts.push(cur);
            cur = cur->left;
        }
        else { // 中 右
            cur = ts.top();
            ts.pop();
            visit(cur);
            cur = cur->right;
        }
    }
}
```

3. 后序遍历

1. 递归版本

```
void postOrder(TreeNode* root) {
    if (!root) return 0;

    postOrder(root->left);
    postOrder(root->right);
    visit(root);
}
```

2. 非递归版本

1. 实现

```
// 对TreeNode 新增了一个flag用于标记
void postOrder(TreeNode* root) {
    TreeNode* cur = root; // 将根节点root赋值给cur
    stack<TreeNode*> ts;

    if (cur) {
        ts.push(cur);
        cur->flag = 1; // 如果cur不为空，将cur入栈，将cur->flag设置
为1，
```

```
        while (!ts.empty()) { // 当栈不为空时
            TreeNode* tmp = ts.top();
            ts.pop(); // 取出栈顶元素赋值给tmp

            if (tmp->flag == 1) { // 如果tmp->flag为1
                ts.push(tmp);
                tmp->flag = 2;  // 则将其再次入栈 并设置tmp->flag为
2
                if (tmp->right) {
                    ts.push(tmp->right);
                    tmp->right->flag = 1;
                } // 如果tmp->right不为空 将其入栈 并设置tmp->right-
>flag为1
                if (tmp->left) {
                    ts.push(tmp->left);
                    tmp->left->flag = 1;
                } // 如果tmp->left不为空 将其入栈 并设置tmp->left-
>flag为1
            }
            else { // 如果tmp->flag为2 则访问tmp
                visit(tmp);
            }
        }
    }
}
```

6.

平衡旋转化:

RR LL 旋转 无需处理

RL旋转 如果节点r有右子树 作为r的父节点的左子树

LR旋转 如果节点r有左子树 作为r的父节点的右子树 如果节点r有右子树 作为r的父节点的父节点的左子树

考试复习:

1. 各种定义
2. 三大类型的数据结构 线性结构 树形结构 网状结构
3. 性能
4. 算法
    1. 算法思想
    2. 时间空间复杂度分析
    3. 适用范围: 最好情况 & 最坏情况