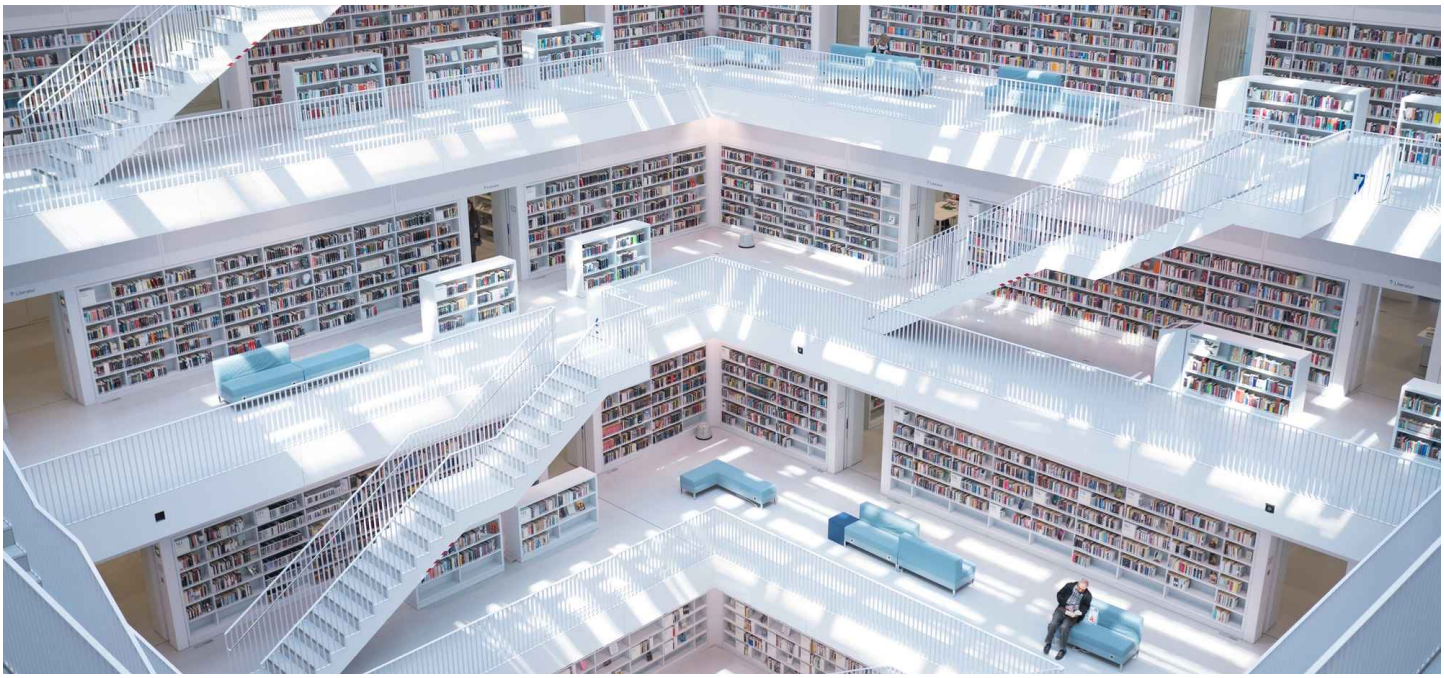

数据结构基础知识

1.数据结构

1. 对象 Object：现实世界的各种事物。
2. 数据对象 Data Object：性质相同的数据元素的集合，即经过抽象用符号形式表示为可被计算机程序处理的对象。
3. 数据 Data：所有数据对象。
4. 数据元素 Data Element：数据的基本单位。
5. 数据项 Data Item：组成数据元素的最小单位。

2.线性结构

1. 线性结构：栈Stack，队Queue，线性表Linear List
2. 栈：只允许在序列末端操作
3. 队：只允许在序列两端操作，队头删除，队尾插入
4. 线性表：允许在序列任意位置操作
5. 平均查找长度（Average Search Length）:查找成功时比较次数的期望值。
6. 线性结构的存储表示分为：顺序存储Sq，链式存储L。在链表中，头指针指向删除端，如栈顶、队头。

2.1.1栈

2.1.1.1顺序栈 SqStack

```
typedef struct{
    ElemType *elem;
    int top;//栈顶的下一个位置
    int size;//存储容量
    int increment;//扩容时，增加的存储容量
}SqStack;
```

2.1.1.2链栈 LStack

```
typedef struct LSNode{
    ElemType data;
    struct LSNode* next;
}LSNode,*LStack;
//
```

2.1.2队

2.1.2.1队列的顺序表示

```
typedef struct {
    Elemtype *elem;
    int front;    //队头位标
    int rear;    //队尾的下一位置
    int maxSize;
}SqQueue;
```

2.1.2.2 链队列 LQueue

```
typedef struct LQNode{
    Elemtype data;
    struct LQNode * next;
}LQNOde, *QueuePtr;

typedef struct{
    QueuePtr front;         //队头指针，指向队头结点
    QueuePtr rear;         //队尾指针，指向队尾结点
}LQueue;
```

2.1.3 线性表 Linear List

2.1.3.1 顺序表 SqList

```
typedef struct {
    ElemType * elem;
    int length;    //当前长度
    int size;
    int increment;
}SqList;
```

2.1.3.2 线性表的链式表示（链表）

头指针：指向链表的第一个结点的指针。

首元结点：放第一个数据元素的结点。

头结点：首元结点之前的一个附设结点。

表长：链表中的数据元素的个数。

1. 单链表 LinkList

```
typedef struct LNode{
    ElemType data;
    struct LNode* next;
} LNode, *LinkList;
```

2. 双向链表 DuLinkList

```
typedef struct DuLNode{
    ElemType data;
    struct DuLNode* prior,*next;    //直接前驱，直接后继
}DuLNode, * DuLinkList;
```

3. 单循环链表：尾元结点的指针域指向头结点 。

4. 双向循环链表：尾元的结点的next指向头结点，头结点的prior指向尾元结点。

3.排序基础

3.1排序的概念与分类

1. 记录：含有多个数据项的数据元素。
2. 关键字（域）：用作记录唯一标识的数据项。
3. 排序：将无序的记录序列按关键字调整为有序记录序列的操作。
4. 排序分类：两大类->内部排序和外部排序。
5. 内部排序：将待排序列完全放在内存中进行的排序。
6. 外部排序：是对大文件的排序，大文件无法一次装入内存，在排序过程需要在内存和辅存之间多次数据交换。
7. 内部排序分为五大类：交换排序、选择排序、插入排序、归并排序和基数排序。

4.哈希表

4.1 哈希表的概念

1. 关键字集合K到一个有限的连续的地址集（区间）D的映射关系H表示为

H (key)：K -> D，（key属于K）

K为主关键字集合，H为哈希函数（散列函数），按哈希函数构建的表即哈希表，D的大小m即哈希表的地址区间长度。

2. 冲突：关键字不同，但哈希值相同的现象。
3. 同义词：哈希值相同的关键字。

4.2 哈希函数的构造方法

1. 直接定址法：H（key）= a*key+b，a为缩放系数，b为平移系数。
2. 除留余数法：H（key）= key % p（p<=m），m为地址区间长度。

……

4.3 处理冲突的方法

4.3.1 链地址法

1. 链地址法：哈希表定义为一个由m个头指针组成的指针数组 T[0...m-1]，凡是哈希值为 i 的记录，均插入以 T[i] 头指针的单链表，即 i 同义词链表。T数组初始值均为空指针。

4.3.2 开放定址法

1. 开放定址法：在哈希表的地址空间内解决冲突，插入产生冲突时，使用探测技术在计算出另外一个哈希值，若不冲突则插入，否则继续求下一个地址，直到探测到空闲地址为止。在探测过程中，求得的一系列地址称为探测地址序列。

2. 常用的开放定址法：线性探测法和二次探测法。

3. 线性探测法：探测地址序列：Hi = （H(key) + i）% m， 1 <=i<=m-1

Hi 表示第 i 次冲突时探测的哈希值（地址）。

4. 二次探测法：Hi = （H(key) + Di）% m， 1 <=i<=m-1

Di = 1，-1，4，-4，……，k^2 , -k^2 （k <= m/2）

4.4 哈希表的实现

4.4.1 链地址哈希表的实现

```
typedef struct Node{
    RcdType r;
    struct Node * next;
}Node;

typedef struct {
    Node ** rcd;    //指针数组
    int size;
    int count;
    int (*hash) (KeyType key,,int hashSize);//函数指针变量，即哈希函数，
                                            //hashSize为哈希表长度
}HashTable;
```

4.4.2 开放定址哈希表的实现

```
typedef struct {
    RcdType * rcd;
    int size;
    int count;
    int * tag;    //标记，0：空；1：有效；-1：已删除
    int (*hash) (KeyType key,int hashSize);
    void (* collision) (int & hashValue, int hashSize);
    //函数指针变量，用于处理的冲突的函数
}HashTable;
```

4.5 哈希表的查找性能

1. 装填因子 a = 表中填入的记录数 / 地址区间长度

2. 性能上，链地址 > 二次探测 > 线性探测

3. 完美哈希函数：没有冲突的哈希函数，K的大小 n ，D的大小 m ，且m>=n，没有同义词。

若 m=n，则为最小完美哈希函数。

5.递归

5.1 递归基础

1. 递归函数： 含有递归调用的函数。递归调用是用相同的策略解决规模更小的问题，直到问题规模小到某个边界条件，则不再递归调用，而是直接处理。

5.2 递归与分治

5.3 递归与迭代

5.4 广义表

5.4.1 广义表的定义

1. 广义表 Generalized List：L = （a1, a2,..... an），ai可以是广义表，也可以是不可细分的元素，分别是L的子表和原子。
2. 表头和表尾：对非空广义表，第一个元素是表头，其余元素序列构成表尾，表尾一定是广义表（单独拿出表尾时，要另外加一个括弧）。
3. 长度：元素个数。
4. 深度：广义表中括弧的最大嵌套层数。

5.4.2 广义表的存储结构

```
typedef char AtomType;
typedef enum{
    ATOM,LIST;//0为原子，1为子表
}ElemTag;
typedef struct GLNode{
    ElemTag tag;
    union{
        AtomType atom;
        struct {
            struct GLNode * hp;
            struct GLNode * tp;
        }ptr;
    }un;
}GLNode, *GList;
```

6.二叉树

6.1 二叉树的定义

1. 二叉树 Binary Tree：是含有 n （n>=0）个结点的有限集合。
2. n=0时为空二叉树。
3. 结点的度Degree：结点的孩子个数。度为0的结点是叶子结点，其他结点是分支（内部）结点。
4. 树的度：即最大的结点度。
5. 层次Level：从根结点开始计算，最大层次即深度或高度。
6. 满二叉树、完全二叉树……

6.2二叉树的存储结构

6.2.1顺序存储结构

6.2.2链式存储结构

6.3遍历二叉树

1. 先序、中序、后序遍历
2. 递归遍历
3. 非递归遍历
4. 使用栈的非递归遍历
5. 使用队列的非递归遍历（层次遍历）

6.4堆

6.4.1堆的定义

1. 堆：所有非叶子结点均不大于（或不小于）其左右孩子结点的完全二叉树。
2. 根最小的为小顶堆，根最大的为大顶堆。
3. 堆的存储结构：采用与完全二叉树一致的顺序存储结构，并增加堆长度域。

```
typedef struct {
    RcdType * rcd;
    int length;
    int size;
    int tag;    //小（大）顶堆 的标记
    int (* prior)(KeyType ,KeyType);//函数变量，用于关键字优先级比较
}Heap;
```

4. 堆的筛选：是指将堆中指定的以 p 结点为根的子树调整为子堆，其前提是 p 的左右子树均为子堆。

5. 筛选的操作过程：先比较左右孩子得出其优先者，然后 p 与优先者比较，若 p 优先则结束；否则交换位置，重复以上步骤，直到 p 成为叶子结点。
6. 堆 的插入：将插入元素加到堆尾，再逐步往上筛选（直到插入元素成为堆顶）。
7. 删除堆顶结点：堆尾与堆顶交换，堆长度减1，对新的堆顶进行筛选。
8. 建堆：先初始化堆，再依次对 length/2 , length/2-1 , ……1的结点进行筛选。（使用循环 for(i=n/2; i>0; i--) ）。

6.5 二叉查找（排序）树

6.5.1二叉查找树的定义

1. 二叉查找树的定义：空二叉树 或者 所有结点均满足 左小根中右大 的二叉树。（结点的值不可重复）。

6.6平衡二叉树

6.6.1平衡二叉树

1. 平衡二叉树 的定义：空树或者任意结点的左右子树的高度差的绝对值<=1的二叉查找树。
2. 平衡因子：左子树的高度 - 右子树的高度
3. 最小失衡子树：在平衡二叉树插入新结点后，往上找第一个不平衡的结点（平衡因子为-2或者2）的结点，以该结点为根的子树即最小失衡子树。
4. 失衡调整：插入结点失衡后，将最小失衡子树调整为平衡的子树而且其高度与原树高度相同。

7.树

7.1树的定义

1. 树：是含有 n (n>=0) 个结点的有限集合。
2. 森林： m (m>=0) 棵互不相交的子树的集合。

7.2树的存储结构

1. 双亲表示法

```
typedef struct PTNode{
    ElemType data;
    int parent;
} PTNode;
typedef struct {
    PTNode * nodes;
```

```
    int r,nodeNum;    //r 是根位置
}
```

2. 双亲孩子表示法

```
typedef struct ChildNode{
    int childIndex;    //孩子在数组中的下标
    struct ChildNode * nextChild;
}ChildNode;
typedef struct{
    TElemType data;
    int parent;
    struct ChildNode* firstChild;
}PCTreeNode;
typedef struct{
    PCTreeNode * nodes;
    int nodeNum,r;
}PCTree;
```

3. 孩子兄弟表示法->该结构实际上是表示成二叉树

```
typedef struct CSTNode{
    TElemType data;
    struct CSTNode* firstChild,*nextSibling;
}CSTNode,*CSTree,*CSForest;
```

7.3树和森林的遍历

1. 由于树的根结点的子树数目不确定，所以一般不考虑中序遍历。
2. 树的先序遍历结结果与其对应的二叉树的先序一致，树的后序遍历结果与其对应的二叉树的中序一致。
3. 给定一棵树（或者森林），可以找到唯一的一棵二叉树与之对应。将森林转化为二叉树时，森林分为3部分：根结点、根结点的子树森林（对应二叉树的左子树）、剩余森林（二叉树的右子树）。

其操作是将每棵树的右兄弟变为右孩子。

7.4并查集

1. 并查集（Union Find Sets）的定义：是由一组不相交子集所构成的集合。其中，任意两个子集两两不相交。
2. 代表元：每个子集选取某个元素作为标识。

3. 并查集的常见操作：查找Find和合并Union
4. 查找：查找某元素所属的子集。
5. 合并：合并两个元素所属的子集。
6. 使用森林F表示并查集S，根结点为代表元。
7. 采用森林的双亲表示法存储结构

```
typedef struct {
    int *parent; //双亲数组，数组下标表示元素，数组存储双亲的下标,为-1时表示是树的根结点
    int n;
}PForest,MFSet;
```

8. 并查集的初始化：先使每个元素自成一个子集（树），即 parent = -1。
9. 合并的操作（将其中一个代表元的双亲置为另外一个代表元）时间主要取决于树 的高度，有改进方法可降低树的高度：加权合并规则法和路径压缩法
10. 加权合并规则法：合并时，小树合并到大树（结点数多的树），可将根结点存储为 -1 改为结点个数的负数。
11. 路径压缩法：更加高效。在查找结点的代表元（根结点）时，置查找路径上的某个结点的双亲为根结点。

7.5 B树（平衡多叉查找树）

7.5.1B树的定义

1. 一棵 m 阶B树，或为空树，或为满足以下特性的 m 叉树。

（1）树中每个结点最多有 m 棵子树。

（2）若根不是终端结点，则至少有2棵子树。

（3）除根结点外的所有非终端结点至少有 m/2 棵子树。

（4）每个非终端结点中包含信息：（n, A0, K1, A1, K2, ……, Kn, An ）。

其中 Ki （1<=i<=n）为关键字，且按升序排序。

Ai 指针（0<=i<=n）指向子树的根结点

关键字的个数 n 满足：m/2 -1 <= n <= m-1 。

所有叶子结点都在同一层，且不包含任何信息（实际上不存在这些结点，指向他们的结点为空）

2. 存储结构

```
#define m 3
```

```
typedef struct BTNode{
    int keynum;     //当前的结点个数
    KeyType key[m+1];//关键字数组，0号未用
    struct BTNode *parent;//双亲结点指针
    struct BTNode *ptr[m+1];//孩子结点指针数组
    Record *recptr[m+1];     //记录指针向量，0号未用
}BTNode,*BTree;
```

7.6 B+树

8.图

8.1图的定义

1. 图：是由有限顶点集V和有限边集E组成，记为 G = （V , E ）

在图中，通常将数据元素称为顶点vertex，顶点之间的关系称为边edge。

2. <v,w> 有向边（即弧），（v,w）无向边（即边）
3. 邻接顶点：（v,w）v,w互为邻接顶点；<v,w> v,w，w是v的邻接顶点，反之不然。
4. 度：顶点的边数。出度、入度
5. 权和网：在边或弧的附加信息，即权。带权的图称为带权图，即网。
6. 路径：从v到w的边（弧）均存在，则构成了从v到w的路径。
7. 路径长度：路径包含 的边数（在带权图中，路径长度是指路径上各边权值之和）。
8. 连通图：在无向图中任意两个顶点都是连通的（v到w有路径，则v和w是连通）
9. 连通分量：无向图的极大连通子图。
10. 强连通图：在有向图中任意两个顶点既有v到w的路径，又有w到v的路径。
11. 强连通分量：有向图的极大强连通子图。
12. 连通图的生成树：含有所有顶点且仅有 n-1 条边的连通子图。

8.2 图的存储结构

1. 使用邻接矩阵存储边（弧）
2. 邻接数组 存储 图

顶点数组和关系数组

3. 邻接表 存储 图

顶点数组和邻接链表

8.3 图的遍历

1. 图的遍历：从某一顶点开始，访问所有顶点，且使每一个顶点仅被访问一次。
2. 深度优先遍历（DFS 类似树的 先序遍历）：从图中某点出发，先访问它，然后 对其所有邻接顶点依次检查，若某邻接顶点未被访问，则以它为新起点递归DFS。
3. 广度优先遍历（BFS 类似于树的层次遍历）：从图中某点出发，先访问它，再依次分为其所有未被访问的邻接顶点，然后再按之前邻接顶点被访问的顺序依次访问他们的邻接顶点。（借助队列，访问、入队、判空、出队、循环）

8.4 最小生成树：权值总和最小的生成树。构造最小生成树的两种方法：普里姆算法和克鲁斯卡尔算法

1. 普里姆算法
2. 克鲁斯卡尔算法

8.5 最短路径（在带权图中，路径长度是指路径上各边权值之和）：路径长度最小的路径。

1. 单源点最短路径：
2. 顶点之间的最短路径：

8.6 拓扑排序

8.7 关键路径