数据结构与算法(C++)

第一章　概论

1.1　引言

1.2 数据结构

- 抽象数据类型(ADT)

1.3 算法

- 算法复杂度
  - 空间复杂度　S(n)
  - 时间复杂度　T(n)
  - 两段算法串联在一起的复杂度，取决于比较慢的算法
  - 两段算法嵌套在一起的复杂度，为相乘的复杂度
  - 若是关于n的k阶多项式，复杂度为theta(n^k)
  - 一个for循环的复杂度等于循环次数乘以循环体代码的复杂度
  - if条件判断的复杂度等于判断复杂度和两个分支部分的复杂度的最大值

1.4 最大子列和问题

- 分而治之（分治法）的基本思路是将原问题拆分成若干小型问题，分别解决后再将结果和而治之，用递归实现很方便
- 最大子列和的分治法：　O(NlogN)
- 在线处理：每输入一个数据就进行即时处理，得到结果是对于当前已经读入的所有数据都成立的解，即在任何一个地方终止输入，算法都能给出当前正确的解，复杂度只有O(N)
- 关键是让计算机记住一些关键的中间结果，避免重复计算

第2章　数据结构实现基础

2.1

- 数据组织的基本存储方式主要是利用数组和链表方式来实现
- 数据结构的存储实现跟所需要的操作密切相关，没有最好的存储方式，只有最合适的存储方式
- 求集合中的第**K**大数问题

2.2　数据存储基础

- 数组
- 类型定义typedef
- 指针

    - 动态内存分配: void * malloc (unsigned size)
    - 动态存储释放函数: void free (void * ptr)
- 结构
- 共用体
- 链表
    - 每一个结点里保存着下一个结点的地址
    - 单向链表
        - 通常使用结构的嵌套来定义单向链表结点的数据类型，结构的递归定义

```
typedef struct Node * PtrToNode;
struct Node{
    ElementType Data;
    PtrToNode  Next;
}
```

```
PtrToNode p=(PtrToNode) malloc (sizeof(struct Node));
```

        - 插入结点

```
t->Next=p-Next;
p->Next=t;
```

        - 删除结点

```
t=p->Next;
p->Next=t->Next;
free(t)
```

        - 单向链表的遍历
        - 带头结点的单向链表
    - 双向链表
        - 多了前驱单元指针
        - 双向循环链表：
    - 单链表反转问题：

2.3 流程控制基础

- 分支控制
- 循环控制
- 选择排序: 从待排序列中找出值最大的元素，然后将该最大值元素与待排序列的第一个元素交换，一直重复该过程，直到待排序列只剩一个元素
- 函数与递归
    - 递归出口：即递归的结束条件
    - 递归子式：当前函数结果与准备调用的函数结果之间的关系

- 汉诺塔问题
- 第**K**大数：递归方法求解
  - 选取s中的元素e将e分解为大于e的集合S1和小于e的集合S2,转换为在S1和S2中的查找问题。
  - 如何分解为两个集合：从左右两端扫描，直到中间相遇，时间复杂度为O(n)且不需要额外空间
  - 基于冒泡排序和简单选择排序，时间复杂度为o(n*k)
  - 基于最小堆，o(nlogk)
  - 基于快速排序，o(n)

第三章　线性结构

3.1

3.2 线性表的定义与实现

- 定义
  - 由同一类型的数据元素构成的有序序列的线性结构
  - 初始化
  - 查找
  - 插入
  - 删除
- 线性表的顺序存储实现(在内存中用地址连续的一块存储空间顺序存储各元素)

```
typedef int Position;
typedef struct LNode* PtrToLNode;
struct LNode {
    ElementType Data[MAXSIZE];
    Position Last;
};
typedef PtrToLNode List;
```

- 初始化

```
List MakeEmpty()
{
    List L;
    L=(List)malloc(sizeof(struct LNode));
    L->Last=-1;
    return L
}
```

- 查找,时间复杂度为O(n)

```
#define ERROR -1
Position Find(List L, ElemenType X)
{
    Position i=0;
    while(i<=L->Last && L->Data[i]!=X)
        i++;
    if(i>L->Last)
        return ERROR;
    else
        return i;
}
```

- 插入,时间复杂度为O(n)

```
bool Insert(List L, ElemenType X, int i)
{
    Position j;
    if(L->Last==MAXSIZE-1)
    {   printf('满')
        return false;
    }
     if(i<1 ||i>L->Last+2)
     {  printf('位序不合法');
        return false;
     }
    for(j=L->Last;j>=i-1;j--)
        L->Data[j+1]=L->Data[j];
    L->Data[i]=X;
    L->Last++;
    return true;
}
```

- 删除，时间复杂度为O(n)

```
bool delete (List L, int i)
{
    Position j;
    if(i<1 || i>L->Last+1)
    {   printf('位序不存在元素');
        return false;
    }
    for(j=i;j<=L->Last;j++){
        L->Data[j-1]=L->Data[j];
    }
    L->Last--;
    return true;
}
```

- 线性表的链式存储实现(不需要地址连续的存储单元，不要求逻辑上相邻的两个元素物理上也相邻)

```
typedef struct LNode* PtrToLNode;
struct LNode {
    ElementType Data;
    PtrToLNode Next;
};
typedef PtrToLNode List;
typedef PtrToLNode Position;
```

- 求表长,时间复杂度为O(n)

```
int Length(List L){
    Position p;
    int cnt=0;
    p=L;
    while(p){
        p=p->Next;
        cnt++;
    }
    return cnt
}
```

- 查找，时间复杂度为O(n)
  - 按序号查找
  - 按值查找
- 插入,时间复杂度为O(n)

```
bool Insert(List L,ElemenType X, int i){
    //默认有头节点
    Position tmp,pre;
    int cnt=0;
    pre=L;
    while(pre && cnt<i-1){
        pre=pre->Next;
        cnt++;
    }
    if(pre==NULL || cnt!=i-1)
        return false;
    else{
        tmp=(Position)malloc(sizeof(struct LNode));
        tmp->Data=X;
        tmp->Next=pre->Next;
        pre->Next=tmp;
        return true;
    }

}
```

- 删除，时间复杂度为O(n)

```
bool Delete(List L,int i){
```

```
    Position tmp,pre;
    int cnt=0;

    pre=L;
    while(pre && cnt<i-1){
        pre=pre->Next;
        cnt++;
    }
    if(pre==NULL || cnt!=i-1 || pre->Next=NULL){
        return false;
    }
    else{
        tmp=pre->Next;
        pre-Next=tmp->Next;
        free(tmp);
        return true
    }
}
```

- 广义表与多重链表
  - 广义表中，元素不仅可以是单元素也可以是另一个广义表
  - 采用共用体实现两个域的复用

```
typedef struct GNode* PtrToGNode;
typedef PtrToGNode GList;
struct GNode{
    int Tag;
    union{
        ElementType Data;
        GList Sublist;
    }URegion;
    PtrToGNode Next;
};
```

  - 双向链表不是多重链表
  - 树、图可以用多重链表实现

3.3 堆栈

- 具有一定约束的线性表，插入和删除都作用在栈顶的端点位置
- LIFO表，压入栈:Push 弹出栈:Pop
- 堆栈的实现(C++中有STL实现):
  - 可采用顺序和链式两种形式
  - 栈的顺序存储实现：

```
typedef int Position;
typedef struct SNode* PtrToSNode;
struct SNode {
    ElementType * Data;
    Position Top;  //栈顶指针
    int MaxSize;
}
typedef PtrToSNode Stack
```

```
//创建一个空的堆栈
Stack CreateStack(int MaxSize){
    Stack S=(Stack)malloc(sizeof(struct SNode));
    S->Data=(ElemenType*)malloc(MaxSize*sizeof(ElemenType));
    S->Top=-1;
    S->MaxSize=MaxSize;
    return S
}
```

- 入栈操作

```
bool IsFull(Stack S){
    return(S->Top==S->MaxSize-1);
}
bool Push(Stack S, ElementType X){
    if(IsFull(S)){
        printf('堆栈满');
        return false
    }
    else{
        S->Data[++(S->Top)]=X;
        return true;
    }
}
```

- 出栈操作

```
bool IsEmpty(Stack S){
    return (S->Top==-1);
}
ElementType Pop(Stack S){
    if(IsEmoty(S)){
        printf('空');
        return ERRROR;
    }
    else{
        return (S->Data[(S->Top)--]);
    }
}
```

- 用一个数组实现两个堆栈

```
typedef int Position;
typedef struct SNode* PtrToSNode;
struct SNode{
    ElemenType* Data;
    Position Top1;
    Position Top2;
    int MaxSize;
}
typedef PtrToSNode Stack;

bool push(Stack S,ElementType X, int Tag){
    if(S->Top1-S->Top2==1){
        return false;
    }
    else{
        if(Tag==1)
            S->Data[++(S->Top1)]=X;
        else
            S->Data[--(S->Top2)]=X;
    return True;
    }

}

bool Pop(Stack S,int Tag){
    if(Tag==1){
        if(S->Top1==-1)
            return false;
        else
            return S->Data[(S->Top1)--];
    }
    else{
        if(S->Top2==MaxSize)
            return false;
        else
            return S->Data[(S->Top2)++];
    }
}
```

- 堆栈的链式存储实现
  - 与单链表类似，但插入和删除只能在链栈的栈顶进行，栈顶指针就是链表的头指针

```
typedef struct SNode* PtrToSNode;
struct SNode{
    ElementType Data;
    PtrToSNode Next;
};
typedef PtrToSNode Stack;

Stack CreateStack(){
    Stack S;
    S=malloc(sizeof(struct SNode))
```

```
    S->Next=Null;
    return S;
}

bool IsEmpty(Stack S){
    return(S->Next==NULL);
}

bool push(Stack S, ElementType X){
    PtrToSNode TnpCell;

    TmpCell=(PtrToSNode)malloc(sizeof(struct SNode));
    TmpCell->Data=X;
    TmpCell->Next=S->Next;
    S->Next=TmpCell;
    return true;
}

ElemenType Pop(Stack S){
    if(IsEmpty(S)){
        return false;
    }
    esle{
        PtrToSNode FirstCell;
        ElementType TopElem;
        FirstCell=S->Next;
        TopElem=FirstCell->Data;
        S->Next=FirstCell->Next;
        free(FirstCell);
        return TopElem;
    }
}
```

- 利用堆栈实现后缀表达式

3.4 队列

- 队列也是一个有序线性表，但队列的插入和删除操作是在线性表的两个不同端点进行的
- FIFO
- 队列的顺序存储实现
  - Rear(或front)%数组长度　取余运算就可以实现折返到起始单元
  - 实际使用队满的条件是(Rear+1)%数组长度==Front

```
typedef int Position;
typedef struct QNode* PtrToQNode;
struct QNode{
    ElemenType* Data;
    Position Front,Rear;
    int MaxSize;
};
typedef PtrToQNode Queue;
```

```
//循环队列的创建插入和删除
Queue CreateQueue(int MaxSize){
    Queue Q=(Queue)malloc(sizeof(struct QNode));
    Q->Data=(ElementType*)malloc(MaxSize*sizeof(ElementType));
    Q->front=Q->Rear=0;
    Q->MaxSize=MaxSize;
    return Q
}
```

- 队列的链式存储实现
  - 队列的头指向链表的头节点，队列的尾指向链表的尾结点
- 总结

  堆栈的应用：表达式求值，函数调用、递归实现、深度优先搜索

  队列的应用：广度优先搜索、操作系统中竞争性资源的管理、服务资源的获得

第四章 树

4.1

- 静态查找、动态查找
- 静态查找
  - 集合中的记录是固定的，不涉及对记录的插入和删除操作，而仅仅是按关键字查找记录
  - 顺序查找：从线性表的一端开始，向另一端逐个取出数据元素的关键字，并与要找的关键字K比较，以判断是否存在要找的数据元素。时间复杂度为O(n)
  - 二分查找：数据是有序存放的，折半查找，二分查找具有对数时间复杂度O(logn)
  - ```
    int binarysearch(int a[], int n, int K){
        //数组从小大排列
        int left=0,right=n;
        int mid;
        while(left<=right){
            mid=(left+right)/2;
            if(K>a[mid])
                left=mid+1;
            elif(K<a[mid]){
                right=mid-1;
            }
            else
                return mid;
        }
        return -1
    }
    ```

4.2 树的定义、表示和术语

- 树是一种非线性数据结构
- 结点的层次：规定根结点在第一层
- 结点的度：子树的个数
    ```

4.3 二叉树

- 定义：左子树和右子树也为二叉树
- 性质
  - 斜二叉树：已退化为线性表
  - 完美二叉树（满二叉树）
  - 完全二叉树：已有的结点与完美二叉树的编号相同，叶结点只能存在于最下层和次下层，深度为O(logN)
  - 一个二叉树第i层的最大结点树为2^{i-1}
  - 深度为k的二叉树有最大结点数$2^{k}-1$
  - $n_0=n_2+1$ 度为0的个数以及度为2的个数
- 二叉树的存储结构
  - 顺序存储结构：通常用于完全二叉树，先编号后存储 i/2向下取整为父节点 2i为左孩子， 2i+1为右孩子。数组下标起始单元为1
  - 二叉树的链表存储 ，每个结点由数据和左右指针三个数据成员组成

```
tyoedef struct TNode* Position;
typedef Position BinTree;
struct TNode{
    ElementType Data;
    BinTree left;
    BinTree right;
};
```

- 二叉树的操作
  - 遍历:先序遍历、中序遍历、后序遍历、层次遍历
  - 先序、中序和后序遍历过程：遍历过程中经过结点的路线一样，只是访问各结点的时机不同。
  - 中序遍历:中序遍历左子树、访问根结点、中序遍历右子树
    - ```
      Void InorderTraversal(Bintree BT){
          if(BT){
              InorderTraversal(BT->left);
              printf("%d",BT->data);
              InorderTraversal(BT->Right);
          }
      }
      ```
  - 先序遍历：访问根结点、先序遍历左子树、先序遍历右子树
    - ```
      void PreorderTraversal(Bintree BT){
          if(BT){
              printf("%d",BT->data);
              PreorderTraversal(BT->left);
              preorderTraversal(BT->right);
          }
      }
      ```
  - 后序遍历:后序遍历左子树、后序遍历右子树、访问根节点

```
void PostorderTraversal(Bintree BT){
    if(BT){
        PostorderTraversal(BT->left);
        PostorderTraversal(BT->right);
        printf("%d",BT->data);
    }
}
```

    - 先序遍历是在深入时遇到结点就访问、中序遍历是在从左子树返回时遇到结点访问，后序遍历是在从右子树返回时遇到结点访问
    - 可以借助堆栈实现遍历路线
    - 二叉树遍历的非递归算法可以用堆栈实现：
    - 层序遍历： 按树的层次遍历，队列实现
        - 从队列中取出一个元素
        - 访问该元素所指结点
        - 若元素所指结点的左右孩子非空，则将左右孩子的指针顺序入队
- 应用
    - 输出二叉树的所有叶节点
    - 求二叉树的高度： 后序遍历，递归求解
    - 表达式树及其遍历：可以按不同的遍历方式得到不同的表达式
    - 由两种遍历序列确定二叉树
        - 可由先序遍历序列和中序遍历唯一地确定一颗二叉树
        - 可由后序遍历序列和中序遍历唯一地确定一颗二叉树
- 二叉树的创建：先序创建和层序创建
    - 层序创建：按树的从上至下从左到右的顺序形成的，各层的空结点输入数值0，需要一个队列暂时存储各结点地址

4.4 二叉搜索树(二叉排序树、二叉查找树)

- 满足条件
    - 非空左子树的所有键值小于根节点的键值
    - 非空右子树的所有键值大于根节点的键值
    - 左右子树都是二叉搜索树
    - 对二叉搜索树中序遍历，将得到一个从小到大的输出序列
- 二叉树的查找某个元素
    - 递归实现
    - 非递归实现
- 查找最大和最小元素
    - 最小元素在最左分支的端结点上
    - 最大元素在最右分支的端节点上
    - 递归实现
    - 非递归实现
- 插入
    - 若没有找到该元素，则查找终止的位置就是X应插入的位置

    - 插入算法的递归实现
- 删除
    - 要删除的是叶节点：直接删除
    - 要删除的结点只有一个孩子结点：改变父节点的指针，指向要删除结点的孩子结点
    - 要删除的结点有左右两颗子树：
        - 选取右子树的最小元素填充删除结点的位置
        - 选取左子树的最大元素填充结点的位置
        - 被选择的结点必定最多只有一个孩子
        - 删除操作的实现

4.5　平衡二叉树

- 平均查找长度（ASL）：查找所有结点的比较次数的平均值
- 一棵树的ASL越小，它的结构越好，与完全二叉树越接近，查找时间复杂度也越接近O(logN)
- 平衡二叉树的定义(AVL树)　　为二叉搜索树
    - AVL树的插入、删除、查找均可在O(logN)时间内完成
    - 任一结点的左右子树均为AVL树
    - 根结点左右子树高度差的绝对值不超过１
    - 平衡因子：　BF(T)=h_L-h_R
- 平衡二叉树的调整（平衡因子是自底向上调整的）
    - 单旋调整
        - RR型不平衡，右单旋，逆时针旋转，　以距离产生问题结点最近的且平衡因子大于１或小于－１的结点
        - LL型不平衡，左单旋，顺时针
    - 双旋调整
        - LR型不平衡，左－右双旋，相当于先右单旋，再左单旋
        - RL型不平衡，右－左单旋，相当于先左单旋，再右单旋

4.6 堆

- 适合于特权需求的数据结构，也称为优先队列
- 堆是特殊的队列，从堆中取出元素是依照元素的优先级大小
- 堆的定义和表示
    - 二叉搜索树的插入和删除操作代价为O(log N)
    - 堆常用的结构表示是二叉树，一般是一颗完全二叉树
    - 通常不必用指针，而是用数组实现堆的存储，数组的起始单元为１
    - 用数组表示完全二叉树是堆的第一个特性
    - 另一特性为部分有序性：
    - 最大堆：任一结点的值大于或等于其子结点的值
    - 最小堆：任一结点的值小于或等于其子结点的值
    - 兄弟结点之间不存在约束关系
- 最大堆的操作
    - 是一颗完全二叉树，每个结点上的元素值不小于其子节点元素的值
    - 最大堆的创建：

        - 可以给第 0 个元素赋值一个特殊的值
    - 最大堆的插入：
        - 新增结点要保证最大堆仍是完全二叉树，也要满足最大堆的性质
        - 从新增的最后一个结点的父节点开始，用要插入的元素从下过滤上层结点
    - 最大堆的删除
        - 取走根节点的元素
        - 堆的最后一个元素必须重新放置
        - 从根结点开始，用最大堆的最后一个元素从上过滤下层结点
    - 最大堆的建立
        - 在线性时间复杂度下建立最大堆
        - 将N个元素按输入顺序存入二叉树中，调整各结点元素，满足最大堆的有序性
        - 从第N/2个结点开始，及前面的所有结点，逐一进行向下过滤操作，直到根节点过滤完毕
        - 某一结点向下过滤要与其子孙结点比较键值，最大的比较次数为树中各结点高度的和
        - 最大堆建立算法的复杂度与结点个数呈线性关系

4.7 哈夫曼树

- 哈夫曼树的定义(Huffman)
    - 带权路径长度最小的二叉树，最优二叉树、哈夫曼树
- 哈夫曼树的构造
    - 贪心算法:每一步执行两棵树的合并，每次选择权重最小的两棵树进行合并
    - 同一组给定权值的结点，构造的哈弗曼树的形状可能不同，但带权路径长度是相同的
- Huffman算法的复杂度
    - O(NlogN)
- 哈夫曼编码: 对字符进行编码，使字符串的编码存储空间最少
    - 不产生二义性的关键: 任何一个字符都不能是另一个字符的编码的前缀
    - 哈夫曼编码也称为前缀编码
    - 采用哈夫曼树生成方法可以保证正确的文本前缀编码
    - 左分支记为 0，右分支记为 1，前缀编码树的特定是每个字符必定是叶节点，且没有度为 1 的结点

4.8 集合及其运算

- 集合的表示
    - 交、并、补、差，判断元素所属的集合
    - 可以用树结构表示集合
    - C++ STL中标准关联容器set, multiset, map, multimap内部采用的就是一种非常高效的平衡检索二叉树：红黑树，也成为RB树(Red-Black Tree)

第 5 章散列查找

5.1

- 顺序查找　O(N)
- 二分查找　O(logN) 需要已排序，且是静态查找

- 二叉搜索树　O(h) 最好情况是O(logN) 最坏情况是O(N)
- 平衡二叉树　O(logN)
- 散列查找：
    - 如何构造散列函数
    - 如何解决冲突
    - 散列查找广泛应用于数据库的信息搜索

5.2 基本概念

- 符号表：
    - 名字－属性对的集合
    - 最核心的操作是：查找、插入和删除
    - 也称为散列表(Hash Table，即哈希表)
    - 散列是一种重要的查找方法，以数据对象的关键字key为自变量，通过一个确定的函数关系h,计算出对应的函数值h(key),把这个值解释为数据对象的存储地址，即存储位置=h(key)
    - h　　哈希函数
- 一般设散列表空间大小为m,填入表中的元素个数为n 则a=n/m 称为装填因子，　实际上将a设为0.5-0.8
- 映射到同一散列地址上的关键字称为同义词
- 散列表溢出
- 关键问题
    - 如何设计散列函数，使得发生冲突的概率尽可能小
    - 当冲突或溢出不可避免时，如何使得表中没有空单元被浪费、同时插入、删除、查找操作都能正确完成

5.3 散列函数构造

- 好的散列函数
    - 计算简单
    - 关键词对应的地址空间均匀、以尽量减少冲突
- 数字关键词的散列函数构造
    - 直接定址法
        - h(key)=a*key+b
        - 计算简单，分布均匀，但要求地址集合与关键词集合大小相同，对于较大的关键词集合不适用
    - 除留余数法
        - h(key)=key mod p p一般选取为小于或等于散列表表长的某个最大素数
    - 数字分析法
        - 若数字关键词的位数较多，有些位数容易相同，有的位数比较随机
        - 一般来说，不够随机的位不适合参与散列运算
    - 折叠法
        - 把关键词分割成位数相同的几个部分，然后叠加
    - 平方取中法
- 字符串关键词的散列函数的构造
    - 把字符串映射到整数后再比较
    - ASCII码加和法

        - h(key)=(sum(key[i])) mod TableSize
    - 前３个字符移位法
    - 移位法
        - 涉及关键词的所有n个字符

5.4 处理冲突的方法

- 开放地址法：一旦产生了冲突，去寻找另一个空的散列地址
    - h(key)=(h(key)+d) mod Table Size
    - 线性探测法
        - d选为i,即以增量序列1,2,...,(TableSize-1) 循环探测下一个存储地址
        - 容易产生一次聚集现象
    - 平方探测法
        - d选为+_i^2
        - 若散列表是某个4k+3形式的素数时，平方探测法就可以探查到整个散列表空间
        - 平方探测法一定程度上减轻了聚集效应
        - 散列到同一地址的数据对象会产生二次聚集
        - 开放地址散列表不能真的删除，只能增加一个删除标记，由于它可能已经引起了冲突
    - 双散列探测法
        - d选为　i*h2(key)
        - 对任意的key,h2(key)不能为０
        - 应保证所有的散列存储单元都能被探测到
    - 再散列法
        - 当装填因子过大时，加倍扩大散列表，再散列
        - 再散列需要新建一个两倍大的散列表
- 分离链接法
    - 将所有关键词为同义词的数据对象通过结点链接存储在同一个单链表中
    - 带头结点的空链表

5.5 散列表的性能分析

- 平均查找长度度量散列表查找效率
- 影响冲突的因素:
    - 散列函数是否均匀
    - 处理冲突的方法
    - 散列表的装填因子a
- 散列方法的存储是随机的，它不便于顺序查找，也不适合于范围查找，也不适合查找最小值、最大值等
- 散列方法是依据关键字直接计算出对应数据对象的位置，即要求关键字与数据元素间存在一定对应关系，通过这个关系，可以得到关键字对应的数据对象位置
- 数组的特点是：寻址容易，插入和删除困难；
- 而链表的特点是：寻址困难，插入和删除容易

- 一种寻址容易，插入删除也容易的数据结构:哈希表

5.6 应用(文件的词频统计)

- 若已存在，增加单词的词频，若不存在，则插入该单词并记词频为 1

第6章　图

6.1

6.2 图的基本概念

- 图的结构是任意两个数据对象之间都可能存在某种特定关系的数据结构
- 图的定义和术语
  - 稠密度: 2|E|/|V|
  - 连通分量: 无向图的极大连通子图
  - 强连通图、强连通分量
  - 生成树
- 图的抽象数据类型
  - 操作集:
    - 创建
    - 增加边
    - 删除边
    - 是否为空
    - DFS 深度优先遍历
    - BFS 广度优先遍历

6.3 图的存储结构

- 邻接矩阵
  - 无向图的邻接矩阵为对称矩阵
  - 浪费时间、浪费空间
  - 对于稠密图是一种高效的方法
- 邻接表
  - 对于顶点v,将所有邻接于v的顶点链成一个单链表，称为v的邻接表，再将所有点的邻接表表头放在数组中，构成邻接表
  - 在边稀疏时，用邻接表表示图比邻接矩阵节省空间
  - 为了求入度，可以建立逆邻接表
  - 建立容易，但要判定任意两个顶点之间是否有边或有弧相连，则需搜索第i个和第j个链表，不及邻接矩阵方便

6.4 图的遍历

- 从图的所有顶点出发，对图中的所有顶点访问一次且只访问一次的次序序列
- 深度优先搜索
  - 树的先序遍历的推广
  - 从顶点v出发，依次从v的未访问邻接点出发递归地进行同样的深度优先搜索，直到图中所有和v有路径想通的顶点都被访问到
  - 图的遍历过程实质上是对每个顶点查找其邻接点的过程
  - 当用邻接矩阵作为图的存储结构时，查找所有顶点的邻接点所需时间为O(|V|^2)
  - 当以邻接表作存储结构时，DFS的时间复杂度为O(|V|+|E|)
- 广度优先搜索
  - 类似于按树的层次遍历的过程
  - 从顶点v出发，依次访问v的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，直至图中所有已被访问的顶点的邻接点都被访问到
  - 即以v出发，由近至远，依次访问和v有路径想通且路劲长度为1,2,...的顶点
  - 需要一个队列把访问过的顶点依次保存下来
  - 广度优先搜索遍历图的时间复杂度和深度优先搜索遍历相同
- 图的遍历的应用:求连通分量、欧拉回路、生成树、DAG的判定、DAG的根、桥边、关节点

6.5 最小生成树

- Prim算法
  - 当前树的生长过程
  - Prim算法的时间复杂度为O(|V|^2)
  - 从根结点长出一棵树
- Kruskal算法
  - 把初始仅含**|V|**个孤立顶点的森林逐步合成一颗生成树
  - 选择一条权重最小的边
  - 把权值按升序排列，时间复杂度是O(|E|log|E|)
  - 判断一条边的两端是否属于同一颗树
  - 总体时间复杂度为O(|E|log|V|)

6.6 最短路径

- 单源最短路径
  - D算法，时间复杂度: O(|V|^2)
- 每一对顶点之间的最短路径
  - F算法，时间复杂度:O(|V|^3)

6.7 拓扑排序

- 有向无环图

6.8 关键路径计算

第7章　排序

7.1

- 没有一种排序算法在任何情况下是最优的

7.2 选择排序

- 简单选择排序
  - 在未排序的序列中选出最小的元素和序列的首位元素交换，接下来在剩下的未排序序列中再选出最小元素与序列的第二位元素交换，依次类推
  - 时间复杂度为O(N^2)
- 堆排序
  - 利用最大堆(最小堆)输出堆顶元素，将剩余元素重新生成最大堆(最小堆)，继续输出堆顶元素
  - 时间复杂度O(NlogN)
  - 额外空间复杂度为O(1)

7.3 插入排序

- 简单插入排序
  - 数个额外空间
  - 时间复杂度为:O(N^2)
- 希尔排序
  - 将待排序的一组元素按一定间隔分为若干个序列，分别进行插入排序，在每轮排序中将间隔逐步减小，直到为1，即最后一步进行简单插入排序
  - 希尔排序算法的整体时间复杂度和增量序列的选取有关

7.4 交换排序

- 冒泡排序
  - 在第k次循环中，对从第1到第N-k个元素从前往后进行比较，每次比较相邻的两个元素，若前一个元素大于后一个，互换位置，一次循环就把第K大的元素移到了第N-K的位置上，即第k趟的冒泡，一共进行N-1次冒泡
  - 增加一个flag标识，若一趟没有交换元素，则序列全部有序，直接退出
  - 时间复杂度为O(N^2)
- 快速排序
  - 将未排序元素根据主元分为两个子序列，其中一个都大于主元，另一个都小于主元，再递归地对两个子序列排序
  - 平均时间复杂度为O(NlogN)
  - 主元的选取影响时间复杂度
  - 至少O(logN)的栈空间

7.5 归并排序

- 将两个已排序的子序列归并合成一个有序序列的过程

- 时间复杂度为O(NlogN)
- 归并排序由于要将两个数组来回复制耗时，一般用于外部排序

7.6 基数排序

- 桶排序
  - 需要已知关键字的范围，且是可列的
  - O(N+M)的时间
- 基数排序
  - 桶排序的推广，主要对有多关键字的对象进行排序
  - 主位优先法
  - 次位优先法
  - 时间复杂度为O(D(N+R))

7.7 外部排序

- 待排序的数据记录以文件的形式存储在外存储上
- 归并排序

7.8 比较

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定