数据结构 课程重难点分析及考试方向预判

本文档以课程复习纲要为蓝本，进行重难点分析及相关考试方向预判，其中**☆为一级要求，□为二级要求，△为三级要求，单独出题以[S]表示，混合出题以[M]表示**，所有页码标注均为英文课本。

Chapter 0 引论（仅作提纲使用）

a) 数据结构的表示

二维矩阵、栈、队列 ***逻辑表示***

数组、单链表、双链表 ***存储表示***

b) 数据逻辑结构的分类

一位数组、队列、栈 ***线性结构***

图和树 多元关系 递归定义 ***动态结构***

c) 数据存储结构的分类

数组 ***顺序存储***

链表与树 ***链式存储***

d) 要求

1 **可以给出 ADT 及其相关描述**

（2 可以给出 C++代码）

本书五大重点：

1 线性表（单链表）的表示和基本操作

2 矩阵（稀疏矩阵等特殊矩阵）的表示和基本操作

3 二叉树及其搜索树的的表示、定义及基本操作以及 AVL、B、B+的扩展理解

4 堆与优先队列的表示、基本操作和相关应用

5 图论基本定义以及相关算法（最小生成树 x2、单源最短路径 x1、全局最短路径 x1）

Chapter 1 C++

要求范围：**类定义方法、递归程序定义、new 使用、友元函数**
考试要求：尽量使用 C++进行描述与实现

重难点分析：

模板类、引用声明、const 的使用(包括引用的 const 定义、函数返回 const 的定义)

考点分析：

[M□] C++面向对象的程序设计（包括模板类、引用等）

Chapter 2 程序性能评估

a) 测度分类
时间复杂度 空间复杂度

b) 时间复杂性
知道Ω(n) O(n) Θ(n) 的理论定义，即上界、下界、精确界；
要求：估计简单算法的时间复杂性、找出最坏与期望时间复杂性

c) 空间复杂性
了解基本存储单元关于 n 的逼近渐进函数

d) **排序要求**
   i. **基于比较的排序：**
      1. **内排序：冒泡、快速、插入、堆、归并**
      2. **外排序：赢者树排序**
   ii. **基于储存的排序：桶排序与基数排序（稳定排序）**

重难点分析：

1 常见排序算法三种时间复杂性、空间复杂性以及稳定性表

注：本部分收纳后面的所有排序种类，即后面的排序重点在此一并列出，后面即不在混合列出，方便复习。

（注：前半部分为比较排序，后半部分为非比较排序）

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$ on average, worst case is n	typical in-place sort is not stable; stable versions exist	Partitioning	Quicksort is usually done in place with O(log(*n*)) stack space. Most implementations are unstable, as stable in-place partitioning is more complex. Naïve variants use an O(*n*) space array to store the partition.
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends[further explanation needed]; worst case is n	Yes	Merging	Highly parallelizable (up to O(log(*n*)) using the Three Hungarian's Algorithm or more practically, Cole's parallel merge sort) for processing large amounts of data.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	O(*n* + *d*), where *d* is the number of

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
							inversions
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, for example using lists.[3]
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes	Insertion	When using a self-balancing binary search tree
Tournament sort	—	$n \log n$	$n \log n$	n[4]		Selection	

Name	Best	Average	Worst	Memory	Stable	$N << 2^k$	Notes
Bucket sort (uniform keys)	—	$n + k$	$n^2 \cdot k$	$n \cdot k$	Yes	No	Assumes uniform distribution of elements from the domain in the array.[6]
Bucket sort (integer keys)	—	$n + r$	$n + r$	$n + r$	Yes	Yes	r is the range of numbers to be sorted. If r = $\mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$ [7]
Counting sort	—	$n + r$	$n + r$	$n + r$	Yes	Yes	r is the range of numbers to be sorted. If r = $\mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$ [6]

Name	Best	Average	Worst	Memory	Stable	$N \ll 2^k$	Notes
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	n	Yes	No	[6][7]
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + \frac{k}{d} \cdot 2^d$	Yes	No	Stable version uses an external array of size n to hold all of the bins
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$\frac{k}{d} \cdot 2^d$	No	No	In-Place. k / d recursion levels, 2^d for count array

2 Ω(n) O(n) Θ(n) 的理论定义，即上界、下界、精确界，并可以对简单的程序进行时间复杂性和空间复杂性估算

考点分析：

[S☆] 排序算法的考查，题型可以涉及问答题、选择题、编程题方向，考查范围涉及各排序算法的稳定性、时间复杂性空间复杂性的比较问答、选择正确或者是直接完成一个排序算法的实现。

[M□] 程序或程序块的时间复杂性判定，题型涉及问答题、编程题

方向，主要考查对给定程序块或者自定义程序的整体复杂度分析，特别是在精确界情况下的分析(即判定上界与下界同时需要在同一个多项式阶次下)。

Chapter 3 数据表示

a)线性表的两种表示方法（公式化、链式化）

要求：C++的表示与基本操作（search/insert/delete）

能用图与程序描述与实现

***指针的使用 和 insert delete 的辅助指针实现**

b)**链表函数功能扩展**

append iterator reverse alternate merge

c)**循环链表与双向链表的结构与实现**

1）利弊分析

2）**空满判断**

3）**哨兵节点的使用**

d)**Chain 的表示与实现**

1）模板类的替换与使用

2）扩展至 LinkedWDigraph

e)**桶排序与基数排序**

O(n)的时间复杂性、算法的整数受限性

f) **并查集**（了解）

union find

重难点分析：

1 线性表的表示及扩展（Linearlist&Chain）

主要在于两种表示方法的书面表述和代码表述,主要考虑的是数据表示的基本操作 **insert search**（含 **find**） **delete** 三种基本操作

这样总计有 **6** 个重点方法需要掌握，另有 **3** 个扩展方法，以下列出框架。

注：请注意链式表示的首尾节点的额外判定问题，可以参见课本程序 **3.3-3.5** 以及 **3.10-3.17**，。

另外需要考虑的是 **chain** 的迭代器表示，需要掌握 **Initialize**、**next** 两个方法。

```
template<class T>//数组表示
class LinearList    {
    public:
        LinearList(int MaxListSize = 10); // constructor
        ~LinearList() {delete [] element;} // destructor
        bool IsEmpty() const {return length == 0;}
        int Length() const {return length;}
        bool Find(int k, T& x) const;
            // return the k'th element of list in variable x
        int Search(const T& x) const; // return position of x
        AbstractList<T>& Delete(int k, T& x);
            // delete k'th element of list and return in x
        AbstractList<T>& Insert(int k, const T& x);
            // insert x just after k'th element
        void Output(ostream& out) const;
    private:
        int length;
        int MaxSize;
        T *element; // dynamic array
};
template<class T>//链表表示  请注意辅助指针
class Chain {
    friend ChainIterator<T>;
    public:
        Chain() {first = 0;}
        ~Chain();
        bool IsEmpty() const {return first == 0;}
        int Length() const;
        bool Find(int k, T& x) const;
        int Search(const T& x) const;
        Chain<T>& Delete(int k, T& x);
        Chain<T>& Insert(int k, const T& x);
        Chain<T>& Reverse();
        void Zero() {first = 0;}
        void Erase(); // delete all nodes
        Chain<T>& Append(const T& x);
        void Output(ostream& out) const;
        Chain<T>& Merge(Chain<T>& A, Chain<T>& B);
    private:
        ChainNode<T> *first,    // pointer to first node
```

```
                    *last;      // pointer to last node
};
template<class T>//迭代器
class ChainIterator {
public :
        T*        Initialize(const Chain<T>& c)
                        {location = c.first;
                         if (location) return &location->data;
                         return 0;}
       T*           Next()
                     {if (!location)      return 0;
                      location = location->link;
                      if (location)    return      &location->data;
                      return 0;}
private :
              ChainNode<T> *location;
} ;
```

2 循环链表与双向链表

1）与普通链表不同的是这里增添了更多的指针元素，具体说来：

循环链表在尾节点处的 link 节点指向了 first 节点；

双向链表在每个节点处增加了 prev 节点；

而双向循环链表是二者兼有之，为了简化操作，在其中设置了一个空的哨兵节点（dummy 节点），里面不存有数值，是为了方便链表的操作设置的，其后继为头结点，前驱为尾节点。

2）对于它们的空满判定：

空：循环或双向链表是 x.first=0 ，双向循环链表是哨兵前后均指向自身。

满：循环和双向是 nomem 判定，但是查找是 link==first 判定为查找结束标志。

3）利弊分析：每种操作相对方便，但是复杂度提高。

3 稳定排序与并查集

此部分的排序部分请参见第二章的相关内容，并查集部分只需要作了解目标，union/find 的概念理解即可。

考点分析:

[S/M☆]线性表的考查，考题类型涉及选择、问答、读程、写程所有题型，考查范围为对于选择、问答考查关于各个方法的复杂度问题，读程序一般会和图论混合考查链式的网络表示，写程序会考查基本线性表尤其是链表的基本操作表示，属于本书五大重点之一，需要高度掌握。

[S□]循环及双向链表的考查，考题类型涉及选择及问答，考查范围是和普通链表的比较（包括复杂度的比较和操作难度以及具体实现差异的比较），两种特殊链表的搜索终点确定，空表判定等等。

[M△]对于等价类，也就是并查集的考查，考题类型涉及选择、问答，考查范围是对其基本操作，也就是 find 和 union 的描述的解释，以及一些简单的优化方法的描述。

Chapter 4 数组与矩阵

a) **array1D array2D 的 ADT 与 C++ 标准表示**
b) **matrix 的行主映射与列主映射以及逻辑位置与映射函数**
c) **matrix 的 ADT 与 C++标准表示**
d) **特殊矩阵存储（对称、上下、稀疏）及映射函数**
e) **稀疏矩阵的链表表示与图的邻接链表表示的异同点**

重难点分析：

1 一维/二维数组的基本表示

主要考查的是在数组条件下的数组基本构架和一些基本运算的构建,包括常见的加减乘除单目或多目运算,由于已经有最基本的动态指针数组实现,实际这种考查难度不高。

程序请参见：程序 4.1-4.11 都有基本描述，难度低，不再赘述。

2 矩阵的基本表示以及运算实现及逻辑位置确定

注：以下的所有矩阵表示的外部参数输入的起点是 1，但内部使用数组从 0 开始，所以会有一个对应关系。

这部分主要在于将二维矩阵转换成矩阵，并实现矩阵乘法，当然事实不会这么简单。在课本中，我们使用一维数组实现了整个矩阵，那么，我们需要考虑映射关系：即行主映射与列主映射。

行主映射的对应关系为：element[(i-1)*cols+j-1]

列主映射的对应关系为：element[(j-1)*rows+i-1]

当然其物理本质依然是由指针声明的动态一维数组。

具体程序可以参看程序 4.12-4.15

3 特殊矩阵的映射与表示

总体原则：在进行插入及删除及改变的操作时请判定是否越界，因为在此特殊矩阵基本都已一维数组存储，除稀疏矩阵的链表表示法。

1)对角矩阵 m[i][i]=e[i]

2)三对角矩阵 m[i+1][i]=e[i-2] m[i][i]=e[n+i-2] m[i][i+1]=e[2*n+i-2]

3)三角矩阵

下三角矩阵 **m[i][j]=e[i*(i-1)/2+j-1]**

上三角矩阵 **m[i][j]=e[(j-1)*j/2+i-1]**

4)稀疏矩阵**[**数组表示**]**

分为三个一维数组分开表示，分别记录 **row column value**

实质上将矩阵虚级化，有些运算对数组表示的稀疏矩阵影响不大，但是对于矩阵的转置会比较高的要求，具体说来实现的方式是按列对每列进行首元素及每列元素数量标记方便转置时的标注与转换（因为，转置后变为实际列优先），接下来读入行的方式将每个列的数据依次读入，读后列下标的映射位置后移，循环完成转置。

而对于加法操作，需要进行判定，分单边有值，双边有值，双边和为 **0** 等 **3** 个类型，分开操作，难度不大不再赘述。

程序参看 **4.17-4.25**

5）稀疏矩阵**[**链表表示**]**

实际是由不等长二维链表实现，即由一个一维（行列双向）链表，行向连缀本行的元素，列向连缀首列元素，从而实现一个二维链表。关于其转置的实现，使用的是桶排序的思路，具体见程序 **4.28-4.30**

3 稀疏矩阵的链表表示和后面图的邻接链表表示的异同点

相同点：两个的物理结构基本相同，是由行列双向链表实现的不等长二维链表。

不同点：

1）两者意义不同，一个表示有值，不是 **0**，另一个表示的是之间有连接而不是没有直接通路；

2）稀疏矩阵的要求更加松散，稀疏矩阵对于全零行可以直接省略行标识，而对于图来说，即使其为孤立点，也不能删除这个点的列向表示，故图邻接链表的首列表示也常用一维数组。

考点分析：

[S/M△]一维二维数组的考查，涉及的题型有选择、问答，当然这一部分渗透于每个题型，难度不大，而且后矩阵部分重叠，考查要求较低。

[S/M□]矩阵的考查，一般涉及的内容包含矩阵的映射关系、矩阵的表示与实现，涉及的题型包括选择、问答、读程序等，这个方面的考查渗透亦比较宽泛，对于选择问答一般考查映射关系与基础实现，而对于读程序题会考查表示与实现的相关问题，另外直接的考查一般被包含在我们的特殊矩阵考查中。

[S☆]特殊矩阵的考查，一般的涉及的内容包括三角矩阵、对角矩阵与稀疏矩阵的考查，特别是稀疏矩阵的考查，考查的是映射关系与具体实现方法，因为对于特殊矩阵而言，一维实现需要很高技巧性，所以也是本书的五大重点之一，需要重点掌握。

[S□]链式稀疏矩阵与图论相关，一般涉及的内容包括链式稀疏矩阵与图论邻接链表的比较，考查题型一般只会涉及问答题类型，但是需要对此有清晰的理解。

Chapter 5 栈（**LIFO**）

a) 栈的定义与 ADT
b) 栈的**自定义**描述（C++&ADT）
c) 栈的**存储实现：一位数组及链表**
   栈的**操作实现：add delete**
d) **栈的应用：括号配对、前后缀表达式等等**

重难点分析:

1 栈的基本描述与自定义描述及其操作实现

对于栈的基本描述由于并不常见也不实用所以不作为考查对象，主要是理解自定义的栈的描述及其实现。其中需要实现的方法有空栈判定、满栈判定、栈的推入、栈的弹出即 **add &delete**。另外其实现方法也有数组和链表两种表示方法，具体框架如下。

```
template<class T>
class Stack {
// LIFO objects
    public:
        Stack(int MaxStackSize = 10);
        ~Stack() {delete [] stack;}
        bool IsEmpty() const {return top == -1;}
        bool IsFull() const {return top == MaxTop;}
        T Top() const;
        Stack<T>& Add(const T& x);
        Stack<T>& Delete(T& x);
    private:
        int top;       // current top of stack
        int MaxTop; // max value for top
        T *stack;      // element array
};
template<class T>
class LinkedStack {
    public:
        LinkedStack() {top = 0;}
        ~LinkedStack();
        bool IsEmpty() const {return top == 0;}
        bool IsFull() const;
        T Top() const;
        LinkedStack<T>& Add(const T& x);
```

```
        LinkedStack<T>& Delete(T& x);
    private:
        Node<T> *top; // pointer to top node
};
```

具体程序可见程序 **5.2 5.4**

2 堆栈的应用

包括括号匹配、前后缀表达式互换、图的 **DFS** 等。认知到堆栈的真正应用于计算机的递归实现中，知道 **hanoi** 的实现也是堆栈。具体而言括号匹配的掌握实际被包含于前后缀表达式转换中，而前后缀表达式转换请参阅实验指导书，在其中有完整而详尽的描述。

考点分析：

[S□]由于堆栈在很大程度上和递归有等价关系，而为了明确不混乱，递归的内容包括 **DFS** 不算为堆栈的考查范围。这样堆栈由于难度一般，考点混合度较小，只需掌握基本的实现与定义即可。考题类型涉及选择、问答、读程序，写程序可能会出小的实现（和队列一起），考查的内容也仅仅涉及基本的实现即 **pop/push**，那么整体难度一般。

[S/M□]堆栈的应用，作为堆栈的衍生类型，整体要求也是一般，涉及题型一般仍然是选择、问答，一般考查的是这些的算法描述，同样难度重要度一般。

Chapter 6 队列(**FIFO**)

a) 队列的定义与 ADT
b) 队列的**形式化**描述（C++&ADT）
c) 队列的**存储实现：一位数组及链表**
   队列的**操作实现：add delete**

队列的优劣判定
循环队列的空满判定

d) **链表队列的 add、delete 的实现**
表头、表尾的依据

e) **队列的应用：BFS**

重难点分析:

1 队列的形式化描述和实现

对于队列基本的形式化表示方法即被常用，与栈类似，其考查的是队列的基本操作 add 和 delete，即出队与入队，另外还需要关注的是队列的优劣判定与空满判定；当然还需要关注的是一类特别的队列循环队列的判定；另外其实现方法也有数组和链表两种表示方法，具体框架如下。

```
class Queue {
// FIFO objects
    public:
        Queue(int MaxQueueSize = 10);
        ~Queue() {delete [] queue;}
        bool IsEmpty() const {return front == rear;}
        bool IsFull() const {return (
                ((rear + 1) % MaxSize == front) ? 1 : 0);}
        T First() const; // return front element
        T Last() const; // return last element
        Queue<T>& Add(const T& x);
        Queue<T>& Delete(T& x);
    private:
        int front;      // one counterclockwise from first
        int rear;       // last element
        int MaxSize; // size of array queue
        T *queue;       // element array
};
template<class T>
class LinkedQueue {
    friend ostream& operator<<
            (ostream&, const LinkedQueue<T>&);
    friend istream& operator>>
```

```
        (istream&, LinkedQueue<T>&);
    public:
        LinkedQueue() {front = rear = 0;} // constructor
        ~LinkedQueue() {Erase();}
        bool IsEmpty() const
              {return ((front) ? false : true);}
        bool IsFull() const;
        T First() const; // return first element
        T Last() const; // return last element
        LinkedQueue<T>& Add(const T& x);
        LinkedQueue<T>& Delete(T& x);
        int Size() const;
    private:
        Node<T> *front;    // pointer to first node
        Node<T> *rear;     // pointer to last node
        void Erase();       // delete all nodes
};
```

具体程序可见程序 **6.2-6.6**

需要注意的是循环队列的判定，其中空队的判定是头尾指针相同，满队的判定是两者在对 **Maxsize** 取模后 **front'-rear'=1**。

链式队列的头尾依据依然有所区别，空是 **front** 指向空，满是尝试增添节点返回异常视为已满。

2 队列的应用

主要提及的仍然是队列在图论的 **BFS** 上的应用，**BFS** 在相关的图论算法中使用也是比较多的，在这里不统一作介绍，将放在图论处统一介绍。

考点分析：

[M□]由于队列在某种程度上与栈相似，是 **BFS** 的基础，如不算相关图论问题，难度不大。考题类型涉及选择、问答、读程序，写程序可

能会出小的实现（和堆栈一起），考查的内容也仅仅涉及基本的实现即 **push/pop**，那么整体难度一般。

[M□]队列的应用，考察的基本都是 **BFS** 相关应用，涉及题型一般仍然是选择、问答，一般考查的是这些的算法描述，同样难度重要度一般。

Chapter 7 跳表（△）与 hash

a) sortedChain
b) Skiplist
c) **Hash 的基本的思想**
   常用 hash 函数
   hash 表的基本表示（open addressing/linked representation）
   hash 表的方法实现：search/insert/delete
   hash 的冲突处理

重难点分析：

1 明确了对跳表的无要求性

2 Hash 的基本表示、实现与冲突处理

首先应该明确哈希函数实际上大空间数据到小空间存储空间的映射，常用的哈希函数包括理想哈希和线性开放选址两类，在这种下物理实现是数组，开放选址使用的是取模操作，如遇冲突就进行逐位后移寻找插入位置，当然理想哈希几乎不考，但是需要注意的是其方法实现包括查找、插入和删除是比较重要的，程序可参考 **7.11-7.15**，以下给出链式 **hash** 的框架。

```
class hashChains : public dictionary<K,E>
{
    public:
```

```
        hashChains(int theDivisor = 11)
        ~hashChains(){delete [] table;}
        bool empty() const {return dSize == 0;}
        int size() const {return dSize;}
        pair<const K, E>* find(const K& theKey) const
            {return table[hash(theKey) % divisor].find(theKey);}
        void insert(const pair<const K, E>& thePair)
        void erase(const K& theKey)
            {table[hash(theKey) % divisor].erase(theKey);}
        void output(ostream& out) const;
    protected:
        sortedChain<K, E>* table;    // hash table
        hash<K> hash;                // maps type K to nonnegative integer
        int dSize;                   // number of elements in list
        int divisor;                 // hash function divisor
};
```

考点分析：

[S□]作为非重点部分，我们只需要掌握散列表的基础即可，既包括表示与实现，也包括特色的冲突处理，但是要求与难度实际上都不高，涉及题型一般会涉及选择、问答，考查问题为散列表实现的复杂度分析、一些实现与冲突处理细节等。

Chapter 8 二叉树及其他树

a) **树**的定义、**基本术语**
   根、树的高度、孩子、双亲、叶子、顶点的度
b) **二叉树及基本术语**
   根、左子树、右子树、空树、满二叉树（一维数组存储）、完全二叉树(一维数组存储)
c) 数学表达式的二叉树表示
d) 二叉树的**性质**
e) 二叉树的**存储（线性、链表）**&C++表示
f) 动态存在结构：
   递归编程、三种遍历方式、树的高度计算
   以上的编程实现

g) 其他树（森林）的二叉树表示

利用二叉树表示集合与集合操作的实现

重难点分析：

1 树和二叉树的基本定义与术语

主要掌握此段的各种术语，包括树的相关术语（根、树的高度、孩子、双亲、叶子、顶点的度）、二叉树的相关术语（根、左子树、右子树、空树、满二叉树、完全二叉树），对其可以有文字上和图形表示上的理解，而这些属于记忆内容在这里不再赘述。

2 二叉树的性质（需要掌握实质及证明）[8.3 节]

1） n 元素的二叉树有 n-1 条边

2） 高为 h 的二叉树，有至少 h 至多 2^h-1 个节点

3） 节点数为 n 的二叉树，树的高度至多为 n，高度至少为 ⌐$\log_2$（n+1）¬

4）节点数为 n 的完全二叉树，对于标号为 i 的节点，有以下三个性质成立；

一）i=1 此节点为根节点

i>1 此节点的父节点的下标为[i/2]

二）2i>n 此节点没有左孩子 否则孩子节点下标为 2i

三）2i+1>n 此节点没有右孩子 否则孩子节点下标为 2i+1

3 二叉树的存储

对于二叉树的存储实际分为一位数组存储和链式存储两种形式，因此我们都需要掌握。但是需要注意的是，数组存储的主要适用范围是完全二叉树、满二叉树等数组使用密度相对较高的类型，而对于我们的

普通二叉树一般采用的形式是链式形式，数组形式比较简洁是按层次遍历的方式进行了标号，方法同性质 4，链表形式在书中给出，可以见程序 8.1。

4 二叉树的遍历及数学表达式的表示

这是二叉树的重点，我们考虑到遍历实际上有前序遍历、中序遍历、后序遍历、层次遍历四类，应了解到四类遍历的实质前三类是递归或者说 DFS，而后一类可以说是 BFS 即队列实现。具体的程序请看 8.2-8.4 以及 8.7，另外需要注意的是 visit 函数的重写。

关于其数学表达式的表示，计算机操作层面使用的是堆栈，物理表示层面使用的是二叉树，所以也就是说这个数学表达式的前中后序表示是非常重要的，当然要求一般局限于绘图问答，所有要求不算很高。

5 二叉树的各种操作实现

除了最基础的定位根节点，合并树、拆分树，还有四个遍历外，我们实际上还有四个遍历的输出，当然那个不难，主要考虑的是高度与节点数的求法，各位可以参考 8.9.3 节与 8.9.4 节，用的是递归求极值和递归求和的方法重写 visit 函数实现的。

6 其他树形结构的二叉树表示与在线等价类

这一部分主要是考虑森林的二叉树表示、多叉树与二叉树的转换，以及在线等价类（并查集）的深入，关于并查集前面已经描述，这里不再赘述。关于多叉树转二叉树的方案，主要抓住“左儿子右兄弟”的原则，即最左边的儿子变为左儿子，其右侧兄弟变为其右儿子，依次递推即可；而对于森林转二叉树，先把每棵树转化为二叉树，然后进

行左儿子右兄弟的操作，即儿子均放在左侧，将其他树变为其右子树，即可变为二叉树，反之亦然。

考点分析：

[S□]树的定义及术语的考查，这部分的考查术语基础考查，属于记忆性内容，考题类型涉及选择、问答，考查范围涉及术语的定义、含义，与图的区别，与多叉树的区别等等。

[S□]二叉树的性质考查，二叉树的性质以四个性质为蓝本，经常会涉及一些证明，考题类型也限定在选择与问答方向，考查范围对性质的理解的问答，或者问答式的简单相关推论证明。

[S/M□]二叉树的存储与简单实现，包括两种存储方式和求树高等操作，需要认真阅读相关代码，理解要义，考题类型一般涉及读程序、写程序两类，考查范围比较广，与二叉树基本表示相关的都会考查。

[S/M☆]二叉树的遍历，这是一个重要部分，应清楚理解四个遍历的实质和操作方法，学会对 **visit** 函数的重写，以及其衍生产物数学表达式的理解，考题类型为读程序和写程序两块，考查范围是其实现和过程实现的具体理解。

[S△]其他树类型和在线等价类，这些都是需要理解地记忆的内容，要求仅仅是画出，所以考题类型基本定位于选择与问答，而考题内容定位于森林、多叉树、二叉树转换等等。

Chapter 9 优先队列

a) 最大堆 最小堆 **的定义与 ADT （实质：完全二叉树）**
b) **Heap 的一维数组存储、类定义、基本操作**
   search delete insert Initialize（时间复杂性）
c) 高度优先左高树（HBLT）的定义
d) **堆排序实现算法与程序**
e) **Huffman code 的思想、huffman tree 实现算法、对字符频率表建立 huffman tree**

重难点分析：

1 堆的分类定义及基本操作实现

首先注意到，堆实际上是分最大堆和最小堆两种，也称大根堆和小根堆，对于每个节点都有左右儿子（如果有的话）小于（或大于）父节点的情况，并且其实质是一个有序的完全的二叉树。另外，我们需要考虑的是其四个基本操作，当然我们以课本的最大堆为例，分为搜索、删除、插入和初始化四个，搜索是针对完全有序的堆排序的结果而言的，从根节点开始向下递归寻找目标值，当然这个用的范围并不是很广，主要需要考虑的是删除与插入，删除一般是删除极值，也就是取出根的值，并且将最后一个节点移动至根节点位置，然后进行堆维护操作，而插入操作实际上就是把放在树末尾，通过维护操作递推向上直到不满足上升要求为止；初始化是将一个无序一维数组的前半部分进行逆向堆维护，这样可以保证最后的变成堆的形态。程序可以参见程序 9.1-9.5，另外需要注意的是复杂度问题，前三个都是 logn 的，后一个是 n 的，有证明请看课本。

2 堆排序的实现与算法描述

堆排序算法是建立在建堆完成的基础上的，那么其基础就是

initialize 方法，在建堆完成之后，堆排序相对显得简单，做法是做类似删除操作，将极值取出，那么取出 **n** 次之后，我们得到的就是一个有序序列，正确性不再赘述，请读者自己理解。具体代码并未给出，请各位在英文维基百科上查询 **heapsort**，即可得到相关代码表示。而且堆实质也就是优先队列的一种实现形式。

3 左高树的定义及简单表述

作为本章的次要内容，左高树我们需要把握的是 **s**、**w** 两个值的含义，还有外部节点的定义，另外对实现相关的操作步骤需要有个基础了解。

4 堆或优先队列的应用：哈夫曼相关

考查的是哈夫曼树、哈夫曼编码等，我们需要了解其思想，并会实现基本编码。给定 **n** 个权值作为 **n** 个叶子结点，构造一棵二叉树，若带权路径长度达到最小的树称为哈夫曼树，其带权路径长度最短，权值较大的结点离根较近，注意到结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积，而树的带权路径长度规定为所有叶子结点的带权路径长度之和，所以树的带权路径长度最小即满足要求，一般也不唯一，下面借用维基百科的描述来说明一下哈夫曼编码：

在计算机数据处理中，霍夫曼编码使用变长编码表对源符号（如文件中的一个字母）进行编码，其中变长编码表是通过一种评估来源符号出现机率的方法得到的，出现机率高的字母使用较短的编码，反之出现机率低的则使用较长的编码，这便使编码之后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。

例如，在英文中，**e** 的出现机率最高，而 **z** 的出现概率则最低。当利用霍夫曼编码对一篇英文进行压缩时，**e** 极有可能用一个比特来表示，而 **z** 则可能花去 **25** 个比特（不是 **26**）。用普通的表示方法时，每个英文字母均占用一个字节（**byte**），

即 8 个比特。二者相比，e 使用了一般编码的 1/8 的长度，z 则使用了 3 倍多。倘若我们能实现对于英文中各个字母出现概率的较准确的估算，就可以大幅度提高无损压缩的比例。

这也是我们实现霍夫曼编码的原则，当然仅需了解含义可以画图即可，编码参见维基百科哈夫曼树部分。

考点分析：

[S/M☆]堆的理解与实现以及其优先队列应用，对于这部分实际上就是对堆这种特殊树的考查，在处理中一般考查的各种操作的实现，包括查找、插入、删除或者其他内容，以及维护队的性质以及建堆操作，虽然这些操作看似简单，可是实现需要一定要求，考试题型涉及选择、问答、读程序、写程序的所有题型，考查范围涉及堆的各个方面的实现，特别是初始化的实现，和它们的时间复杂性分析，是五大重点之一，所以重点掌握是必要的。

[S/M☆]堆排序及其实现，恰与前述的排序算法相似，这里对堆排序进行了详尽而又充分的描述，作为综合表现最优秀的比较排序算法之一，其实现步骤是很重要的，由于其建立在基本堆操作之上，所以堆操作熟悉的情况下，才会掌握堆排序的实质，考题类型基本为问答和写程序，考点还是排序算法的实现和时间复杂性分析。

[S△]左高树相关，我们需要考虑的是左高树的几个重要定义和基本实现方法，考题类型集中于选择或问答，考题范围基本为左高树基础理解与实现原理。

[S□]哈夫曼相关，主要考查的是哈夫曼树和哈夫曼编码及自定义内容，考题类型涉及写程序和选择、问答，需要考查的范围基本是哈夫

曼的几个相关定义，以及如何用字频作为权重因子建立哈夫曼编码，（当然需要先建立对应的哈夫曼树，再加权得到，）可能在考试中被重点提及需要注意。

Chapter 10 竞赛树（△）

赢者（输者）树的定义；
采用上述树的排序算法思想描述
采用赢者树实现外部排序的算法描述；

[S△]此段已明确没有重难点，也没有重点考点。所以不再赘述，只做了解即可。

Chapter 11 搜索树

a) 二叉搜索树（☆）、AVL 树（平衡性）、B-树（多叉树、根节点的分裂与合并）、B+树定义
b) 二叉搜索树的表示与基本操作:
   search insert delete
c) AVL 树的性质、平衡因子的定义、旋转
d) B-/B+树的定义表述

重难点分析：

1 二叉搜索树的定义、ADT 表示及基本操作

对于二叉搜索树其定义并不难理解，实质仍然是一棵二叉树，只不过就像堆一样，是一个有特性的二叉树，其性质是对于每一个节点，左右儿子（若存在）其左儿子必在定义下的顺序中（或普遍约定的顺序中）比父节点小，右儿子必比父节点大。如此构成了一棵二叉搜索树。其 ADT 表示中涉及插入、删除、查找三个基本操作，但因为其没有优化，复杂度变化程度大，树的深度也差异很大，不过其期望复

杂度都是 **logn**，最坏情况下均会到达 **n** 的水平。需要注意的是有一个 **DBST** 类，只不过要求很低，而且便于理解，在此不进行赘述。程序可以参见程序 **11.2-11.4**，尤其需要注意删除操作。

2 AVL 树的定义、性质及旋转操作

AVL 树是一种经过修正的二叉查找树，具体说来其实现了普通二叉搜索树的复杂度不稳定问题。通过途中修正旋转实现整个树的相对平衡，具体说来需要有以下性质：

1） **n** 个节点的 **AVL** 树的高度是 **logn** 级别的
2） 对于每一个值 **n**，如果其 **n>=0**，那么存在一棵 **AVL** 树
3） 一个 **n** 元素的 **AVL** 树搜索时间复杂性是 **O（logn）**
4） 一个新元素插入原有 **n** 元素的 **AVL** 树，时间复杂性是 **O（logn）**
5） 一个元素从原有 **n** 元素的 **AVL** 树中删除，时间复杂性是 **O（logn）**

对于我们来说，实际的要求是要理解这个数据结构而不是代码实现，也就是会用图形表示这个 **AVL** 树，那么需要考虑的是 **AVL** 树的自调整问题，也就是旋转操作的实现，具体说来分为插入调整与删除调整两种，插入调整分为 **LL**、**LR**、**RL**、**RR** 四种调整，删除操作分为 **R0**、**R1**、**R-1** 共三种调整，具体实现方式请参见课本图表 **11.8-11.13**，另外注意调整的原则的一个重要参数：平衡因子，即左子树深度减去右子树深度的值，当这个值在插入删除后出现了绝对值大于一的情况，那么即需要调整。

3 B-树（包括 **m** 叉树）的定义、性质及基本操作与树的调整

（**B+**树简介）

类似于 **AVL** 树，这也是一种二叉平衡搜索树，我们需要基本了解 **m** 叉树的定义与性质：

1) 在对应的 **m** 叉树中，每个内部节点最多有 **m** 个孩子并且含有 **1** 到 **m-1** 个元

素

2) 每个含有 p 个元素的节点拥有 p+1 个孩子
3) 考虑任何一个有 p 个元素的节点，节点的这 p 个元素按照严格升序排列，而此点的 p+1 个孩子，第 i 个孩子应该比第 i-1 号（如果存在）的元素大，而比第 i 号元素（如果存在）小

这是一个 m 叉树的基本定义，下面介绍基本操作，m 叉树实际上与二叉树类似，实际上也是有查找、插入、删除操作在内的三个基本操作，可以证明 m 叉树的高度是 $\log_m$（n+1），其插入和删除的时候实际上需要考虑到节点的拆分和合并，但是由于后面介绍 B-树的类似操作，在这里不再赘述。

那么 B-树的相关定义与性质是这样的：

1) 它是一棵 m 叉树
2) 对于根节点它最多含有两个孩子节点
3) 所有的内部节点（不包含根）最少含有 ⌈m/2⌉ 个孩子
4) 所有外部节点都在同一层（也就是同一高度）

另外还可以证明，设 d= ⌈m/2⌉ ，那么有以下关于节点数与高度的性质：

5) $2d^{h-1}-1<=n<=m^h-1$
6) $\log_m(n+1)<=h<=\log_d((n+1)/2)+1$

在以上的性质基础上，介绍我们的 B-树基本操作其基本操作包含查找、插入、删除，课本 11.4.5-11.4.7 介绍了这一部分的内容，具体说来查找与普通 BST 没有什么差异，对于插入与删除我们需要考虑的是节点的合并与拆分，在插入时如果此点已经超过 m-1 个节点上限需要对这个节点进行拆分，将其二分位两个节点挂载在父节点下，依次向上递推直至所有节点都符合要求，删除是其逆操作，换句话说叫不足则合并，不再详述。

B+树的出现是因可能会涉及 ISAM 的操作数计算，B+树是一种常用于文件组织的 B-树的变形树，有以下性质：

1）所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指

针，且叶子结点的本身依关键字的大小从小到大顺序链接。
2）所有中间结点可以看成是索引部分，结点中仅含有其子树（根结点）中最大（或最小）关键字。

对于 B+树我们要求不高，仅做了解即可，其基本操作可以参见 slides 或者维基百科，在这里不再赘述。

考点分析：

[S/M☆]二叉搜索树的考查，作为五大重点之一，自然有其特殊的重要性，考题类型涉及选择、问答、读程序、写程序所有题型，考查范围一般包括但不局限于二叉搜索树的定义与性质，其算法实现与理解，特别是要考虑查找、删除以及插入的具体实现，还有相关高度、节点数以及时间空间复杂性的表述等。

[S/M☆]AVL 树的相关特性的考查，作为平衡二叉搜索树，考查题型一般只会涉及选择与问答，但是考查的范围一般涉及的会比较难，不局限于基本定义，而是考查对 **AVL** 的五种旋转的表示，特别是书面表示，重点需要把握的就是平衡因子的定义、平衡因子的变化对应的旋转形式是什么、旋转的具体操作是怎么样的等等，是一个拉分的主要考点。

[S□]B-/B+/m-way 树的考查，对于 **m** 叉树的考查一般被包含于 **B-**树之中，所以我们需要重点考虑的是 **B-**树的相关操作，与 **AVL** 与类似，重点除了那些基本定义，也就是构造的基本定义以外，需要重点关注的是 **B-**树的插入删除中需要实现的节点的合并与拆分，掌握其实现方法并可以在书面表示出来。而对于 **B+**树，由于不是相对重点，需要掌握的主要是性质、与 **B-**树的区别以及一些相关的结论等等，不

做非常重点的要求。

Chapter 12 图

a) 图的定义与分类
   网络、有向（赋权）图、无向（赋权）图、
   顶点度、路径、回路
   树在本章与第八章定义的区别
   最短路径、最小生成树
b) 图的表示：
   邻接矩阵 邻接链表
   以上的图表述、C++表述
c) 图的性质（顶点数与边数、握手定理）、稀疏图表示方法
d) BFS 与 DFS 的描述与程序，时间空间复杂性分析
e) 图遍历算法、图联通分支、BFS/DFS 树

重难点分析：

1 图的定义、相关术语与性质

首先应该明确的是图的是一个点集合边集构成的二元组，即 **G=（V,E）**，下面罗列以下术语，各位需要理解，最好记忆：

1） 图的构成组件：节点、边、邻接关系、自环、路径（简单路径）、路径长度、回路、连通分量、强连通分量、度、入度、出度、边的头尾、最短路径

2） 图的相关分类：有向图、无向图、有向带权图、无向带权图、网络（带权图）、子图、生成树（最小生成树、深度优先生成树、广度优先生成树）、二分图、连通图、完全图、稀疏图、稠密图

需要注意的是，其中这里树的定义变为了连通无向无环图，需要和第八章的表述加以区分。

下面介绍图的相关性质：

1） 握手定则 无向图的边数的两倍为无向图各顶点度数之和

2） 无向图的边数介于（含）**0** 与 **n**（**n-1**）**/2** 之间

3） 有向图的边数介于（含）**0** 与 **n**（**n-1**）之间

4） 有向图的边数为各顶点的入度或出度之和

当然实际会有很多推论，并在课本上以课后习题形式予以了提供。

2 图的表示方法

图的表示方法分为理论物理表示和代码实现物理表示两种，对于理论上的物理表示分为邻接矩阵、邻接链表、邻接线式表等，但是对于线式表实际不常用，需要掌握的是它的实现原理以及书面表达即可，下面需要重点解释是邻接矩阵和邻接链表的描述：邻接矩阵顾名思义是使用矩阵（二维数组）表示各节点之间的邻接关系，对于无权图，使用 **1** 表示存在连接，**0** 表示不存在，而对于有权图，**1** 被替换为权值，**0** 依然表示没有连接，这种表示适合稠密图，是一个 **v*v** 的二维数组，利用率相对较高；而对于稀疏图，我们则一般比较常使用的是邻接链表，利用链式表示出度情况，对每一个节点建一个一维链表，每个节点按列定义为一维数组，如此建立一张邻接表。它们的 **C++** 表述可以参见程序 **12.1-12.10**，由于课本在处理邻接链表时采用了分类型的处理，程序会略显杂乱，但是基本都包含在 **linkedbase** 中，实际的差距并不大，只有小细节调整。

3 BFS 与 **DFS**

即广度优先搜索和深度优先搜索，是一种图的遍历方法，对于这两个

来说，是图论所有算法的基础，具体而言，DFS 的实质就是递归，对于存在的可遍历点就不断往下递归直到下层无可遍历点为止，也可被称作“钻牛角尖”式的遍历方法，BFS 的实质是层次遍历，实现亦然使用的是队列存储本层的所有访问，然后逐渐轮换至下一层，如此变换，不需要递归实现遍历，两种搜索方式各具特色，两种的后面使用也比较丰富，具体代码参见程序 12.17-12.21.

4 图遍历、连通分量与生成树

这部分图遍历即为 BFS 和 DFS，需要掌握的是两种搜索产生的生成树形式；连通分量的求法实际上有 Tarjan 强连通分量算法，具体可以参见维基百科，由于要求不高，在这里不再赘述；对于生成树将在后面章节描述，这里不在同一表述。

考点分析：

[S/M□]图的定义、性质与术语的考查，这一部分的考查准确说属于记忆内容，并且考察广度与细致度，题型范围涉及选择与问答，内容范围就是各种性质定义的考查，特别是关键定义的辨析与理解，但是需要主要的是，与树的辨析一直是重点，大家需要重点掌握，另外不排除会有一些引理或推论的证明等等。

[S/M☆]图的表示是基础也是重点，因为在涉及链式表示的时候，分为了有向图、无向图、有向带权图、无向带权图这四种图的分别表示，所以考试题型涉及问答、读程序、写程序等，考试范围主要是图的各种表示写法、表示的一些具体问题等等。

[M☆]图的深度广度优先搜索，这个是后面各种图论算法的基础，所以虽然表面上没有那么重要，理解也比较容易，但是会渗透在算法部分中常常出现，所以需要注意，考查题型一般确定于写程序、读程序等，但是一般不会单独出题，考试的范围就是两者的应用与比较。

[M□] 图遍历、连通分量与生成树的考查，这方面的考查前两者基本都是基础理解与书面表述，考试题型涉及选择、问答，考试范围一般涉及定义理解，实现的算法描述，实现结果的表述与书写，后者则放在算法部分统一讨论。

Chapter 13 贪婪算法

a) 贪婪算法思想及特点
b) 实现：**拓扑排序、二分图覆盖、单源最短路径、最小生成树（Prim 和 Kruskal）**

Chapter 14 分治算法

应用：**merge**
qsort
search no. K element

Chapter 15 动态规划

应用：**矩阵链的计算**
所有顶点之间最短路算法(floyd)

注脚与考点分析概述：本段做同一性处理的原因是，这段讲的比较快也比较散，基本的数据结构已包含于之前章节之中，这一部分是对前面的提升与部分总结提高，故作为算法章节统一处理。

这一部分不再进行分类考点分析，因为基本都是单独考查算法或者是考查算法比较，而且考查题型基本只有问答中的算法简述、读过程理解与写一些算法程序实现，另外考查的内容就是算法的思想分析、实

现与复杂度分析，故不再赘述。需要注意的是，我们的排序类算法与图论类算法的重点程度与要求难度，均是一级要求，也就是**[S/M☆]**，需要各位重点掌握相关内容，这一部分的所有内容也均是是五大重点涉及，各位更需高度关注。

重难点分析：

1 排序搜索类算法

这其中主要涉及的是归并排序、快速排序与二分查找等。

1）归并排序：

归并操作步骤如下：

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
4. 重复步骤 3 直到某一指针达到序列尾
5. 将另一序列剩下的所有元素直接复制到合并序列尾

当然这只是一层的归并操作的处理方法，我们做每个归并排序前会不断分治，（实质就是一个分治递归过程，）直到两个已经排序序列为单元素时（单元素序列必有序），（即终止条件达成，）那么不断利用已经完成的排序子序列进行合并操作，最终实现排序全部过程。相关代码请参见维基百科。

2）快速排序

首先应该明确的是，它的平均与期望复杂度 **nlogn**，它的最坏复杂度却是 n^2 的，下面介绍步骤：

1. 从数列中挑出一个元素，称为 "基准"（**pivot**），

2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（**partition**）操作。
3. 递归地把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归的最底部情形，是子序列的大小是零或一，也就是必然有序的状态。虽然一直递归下去，但是这个算法会在子序列大小不足时退出，因为在每次的迭代中，它至少会把一个元素摆到它最后的位置。通过这样的递归实现快速排序，当然一般选择序列首元素为基准，所以实际上有一个覆盖的简化方法，具体见维基百科相关代码。

3）寻找第 **k** 大的元素

首先需要明确的是，它的实现方法在某种程度上是和快速排序类似，一般实现的需求依然是寻找基准，只不过这个基准是中位数就是了，下面介绍步骤：

1. 将序列重新分组，一般分为 **5-7** 组，然后从每组中找到中位数，然后形成中位数集，再对这个集合中找出最终的中位数作为基准
2. 重新分配数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边，但是一般会保证不相同）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（**partition**）操作。
3. 确定基准的位置，确定其是否是我们的目标位置，或是小于、大于目标位置，然后递归地把小于（大于）目标值元素的子数列进行再次分组，直至找到我们的目标位置元素。

这个最终的代码需要参看课本程序 **14.7**，那里有比较准确的描述。

2 图论类算法

在这个部分中，我们需要关注的有拓扑排序、二分图覆盖、两类最短路径的求法以及最小生成树的求法，下面我们分开介绍这几种算法：

1）拓扑排序

拓扑排序作为一个图论的基础算法，不是一个线性排序算法，其实现的是能否确定一个有向图所代表的的 **AOE**（活动网络）是否存在一种先后关系，即先完成一个，再完成一个，依此类推最终实现所有节点（活动）的完结，当然如果存在一个环（或者称作环形依赖关系）那就无法实现拓扑排序，会返回错误信号，具体实现步骤如下（英文、伪代码）：

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

通过以上的步骤实现了拓扑排序，其 **AOE** 的拓扑排序序列存在于 **L** 中，需要注意的是有一个判定，就是是否存在环，当然上面的伪代码亦有描述，代码可以参见作者的相关附件与课本描述程序 **13.2**，需要说明的是这个算法的实质是贪心算法，并且正确性已经得到了证明。

2）二分图匹配

对于二分图匹配部分，课本给出的是贪心算法，实际这并不是最优解，最多只能视为可接受解，所以考查的可能性低，程序 **13.3** 给出了初步描述，但是由于不是最优解，意义不大，因此考查概率与难度都不会很高。

3）最小生成树

对于最小生成树我们有点优先的 **Prim** 算法和边优先的 **Kruskal** 算法，并且其算法实质都是贪心算法，并且正确性已经得到了证明，下面分别介绍两种算法的应用：

1>Prim 算法

Prim 算法是以点作为变元的，每一次添加一个点进入整个生成树中，如果生成树存在（即图连通），那么可以得到最小生成树，需要注意的是最小生成树实际上是不唯一的，所以不同算法得出的最小生成树可能会有差异，具体步骤如下：

1. 输入：一个加权连通图，其中顶点集合为 V，边集合为 E；
2. 初始化：V_{new} = {x}，其中 x 为集合 V 中的任一节点（起始点），E_{new} = {}；
3. 重复下列操作，直到 V_{new} = V：
    1. 在集合 E 中选取权值最小的边（u, v），其中 u 为集合 V_{new} 中的元素，而 v 则不是（如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中之一）；
    2. 将 v 加入集合 V_{new} 中，将（u, v）加入集合 E_{new} 中；
4. 输出：使用集合 V_{new} 和 E_{new} 来描述所得到的最小生成树。

需要注意的是如果图不连通，实际在判定是会报错并返回没有最小生成树的信息，另外多条满足时的挑选原则也直接影响最后的生成树样式，代码可以参见维基百科。

2>Kruskal 算法

Kruskal 算法是一种边优先的算法，每次加入的是权值最小的边，但是为了防止出现环，实际上我们用到了之前在线等价类的相关内容，也就是并查集的相关操作，包括 **union** 和 **find**，来确定我们的点集会不会因此而产生环，具体实现步骤如下：

1. 新建图 G，G 中拥有原图中相同的节点，但没有边
2. 将原图中所有的边按权值从小到大排序
3. 从权值最小的边开始，如果这条边连接的两个节点于图 G 中不在同一个连通分量中，则添加这条边到图 G 中
4. 重复 3，直至图 G 中所有的节点都在同一个连通分量中

需要注意的是，实际操作中每次步骤三这样判定之后无法像 **Prim** 算法一样在中间就确定整个图的连通性，而需要在步骤四判定最后的点并查集已经合并为一个，否则视为整个图不连通，同样多条满足的挑选原则也直接影响最后的生成树样式，代码需要参见维基百科或者课本程序 **13.6-13.8**。

4）单源最短路径

在这部分我们现在仅介绍 **Dijkstra** 算法，这也是一个贪心算法，类似于 **Prim** 算法，也是以点作为优先判定元的，最后通过不断添加节点形成单源最短路径的路径集，具体步骤如下（伪代码）：

u = Extract_Min(Q) 在顶点集合 *Q* 中搜索有最小的 *d*[*u*] 值的顶点 *u*。这个顶点被从集合 *Q* 中删除并返回给用户。

```
 1  function Dijkstra(G, w, s)
 2      for each vertex v in V[G]                                  // 初始化
 3            d[v] := infinity       // 将各点的已知最短距离先设成无穷大
 4            previous[v] := undefined    // 各点的已知最短路径上的前趋都未知
 5      d[s] := 0     // 因为出发点到出发点间不需移动任何距离，所以可以直接将 s 到 s 的最小距离设为 0
 6      S := empty set
 7      Q := set of all vertices
 8      while Q is not an empty set    // Dijkstra 算法主体
 9            u := Extract_Min(Q)
10            S.append(u)
11            for each edge outgoing from u as (u,v)
12                  if d[v] > d[u] + w(u,v// 拓展边（u,v）。w(u,v)为从 u 到 v 的路径长度。
```

```
13                          d[v] := d[u] + w(u,v)     // 更新路径长度到更小的那个和值。
14                          previous[v] := u          // 纪录前趋顶点
```

如果我们只对在 ***s*** 和 ***t*** 之间查找一条最短路径的话，我们可以在第 **9** 行添加条件如果满足 ***u*** = ***t*** 的话终止程序。

需要注意的是我们的非连通图，实质上也是可以正常结束点的，只不过与源点 **u0** 不连通的点路径距离均会为无穷大，另外之所以成为单源最短路径，可知只有源点到其他各点的最短路径距离是最短的，而并不是其他点之间也是最短的，所以可利用的只有 **d** 数组，不可以使用 **d** 数组去估算其他最短距离，对于多源最短距离的求法需要使用 **floyd** 算法。

5）多源最短路径

对于多源最短路径的算法，我们介绍的是 **floyd** 算法，这是一个动态规划算法，也是我们需要现阶段掌握的唯一动态规划算法，其具体步骤如下：

设$D_{i,j,k}$为从i到j的只以$(1..k)$集合中的节点为中间节点的最短路径的长度。

1. 若最短路径经过点 k，则$D_{i,j,k} = D_{i,k,k-1} + D_{k,j,k-1}$;
2. 若最短路径不经过点 k，则$D_{i,j,k} = D_{i,j,k-1}$。

因此，$D_{i,j,k} = \min(D_{i,k,k-1} + D_{k,j,k-1}, D_{i,j,k-1})$。

通过这样一个 **O**（**n**^{3}）算法，实际上实现了全局的最短路径求法，需要注意的是，这里面实际就利用到了动态规划中的记忆化搜索过程，希望有能力的同学可以以此将动态规划的实质有一些基础了解，详细代码可以参见程序 **15.9** 或者维基百科的相关内容。

后记：

至此，整个课本《数据结构、算法与应用》就总结到这里了，其中作者按照重要度和课程要求对整本书进行了简析，希望对读者的数据结构课程考试有所帮助，在此感谢对本书提供宝贵意见和建议的各位同学，希望各位新年愉快、新春愉快、考试顺利、归家旅途一路顺风！

作者：圆形方孔钱

2013 年 12 月

附录

题型设置：
问答
选择
按算法走例子
编程实现

老师原有复习纲：

总复习

1) 第 **0** 章引论:
a) 理解数据结构的逻辑表示和物理表示，能结合具体的数据结构解释，如 **2** 维数矩阵、栈、队列的逻辑描述和对应数组和采用数组（链表）的存储表示。
b) 知道线性结构和动态结构的区别，前者要求元素总有前、后面的元素的概念，如，一维数组、队列和栈；后者则没有前后的概念，如图和树，一个元素可以和多于 **2** 个的元素有位置上的关系。
c) 存储结构上分为顺序存储（数组）和链接存储（链表和树）；
d) 对给定的数据，知道并有能力给出其抽象的数据结构，即给出数据说明和在其上的操作说明。最好能给出 **C++**的类表示。
2) 第一章 **C++**

a) 类的定义方法；基本的程序设计、递归函数的设计（知道递归程序是通过栈实现的）；new 的使用和友元函数。我们的理想是大家能用 C++描述数据结构和程序实现，但若做不到，可以和自然语言等混合描述,称为伪代码。

3) 第二章:程序性能评估（算法性能）

a) 知道程序有两个重要的测度，即时间复杂性和空间复杂性。

b) 时间复杂性：指定一个基本操作，把算法输入数量 n 作为参数，用算法需要的基本操作数量的关于 n 的渐进逼近函数来描述算法复杂性的级。知道符号 (n)，O(n),和 (n)函数的理论上的定义，以及实际上的作用（下界，上界和精确界）。对简单的算法自己能估计 O(n). 时间复杂性又分最坏和期望时间复杂性，若不特殊说明，一般指最坏情况下的时间复杂性。即找一个特例，算法中基本操作最多。

c) 空间复杂性：指一个算法（程序）所需的内存空间，主要是指数据所使用的空间（虽然也包含指令和程序栈等）。也是用算法需要的基本存储单元的关于 n 的渐进逼近函数来描述算法复杂性的级表示。在表示上也用符号 (n)，O(n),和 (n)，但精确时，要和数据的类型联系起来计算，如整型（2byte）、实数类型(4byte)等。

d) 掌握基本的排序算法和它们的最坏和期望时间复杂性，即能用自然语言给出下面算法的思想和具体的实现步骤。最好能给出 C++程序。

i. 基于比较的排序

1. 内排序：冒泡、快速、插入、堆排序、归并排序

2. 外排序：赢着树排序

ii. 基于存储

1. BOX 排序和基数排序

4) 第三章:数据表示: （重点是单链表上的操作）

a) 线形表的两种表示方法（基于公式的和基于链接的），C++类的表示和对应的基本操作（search, insert 和 delete 实现的算法。能用图和程序给出基本操作的描述和实现。特别是对指针的使用和具体操作中对指针的修改要熟练,知道需要辅助的指针帮助实现 insert, delete。

b) 对链表函数功能的扩展，有实现能力（append, iterator, reverse ,alternate）以及两个有序链表的归并等。

c) 掌握循环链表和双向链表的结构和对应基本操作的实现。知道它们的特点和在什么情形下用它们比较好,对编程带来的简化。知道循环链表的空和满的判断,和增加一个虚拟头节点在编程带来的便利。

d) 掌握采用 C++描述的 Chain 的表示和基本操作的实现。知道通过对类 T 的不同类的替换，得到后面应用赋权图的邻接链表的表示。

e) 掌握箱子排序和基数排序算法和程序。知道这是一种不需要比较操作并且具有 O(n)时间复杂性的排序算法，但也是一种应用受限算法（元素为整数）。

f) 集合的表示和操作 Union, Find 的实现方法。

5) 第四章：数组与矩阵

a) 一维和二维数组的抽象数据表示方法，建立在类定义下的操作和对应的 C++实现。

b) 2 维矩阵的行主映射和列主映射的存储方式和对应的数组元素的逻辑位置和具体存储位置的映射函数。

c) 矩阵抽象数据描述和物理存储。

d) 特殊矩阵的存储（对称、上下矩阵和稀疏矩阵）及映射函数。

e) 稀疏矩阵的链表表示和后面图的邻接链表表示的异同点是什么？

6) 第五章：栈 (LIFO)

a) 栈的定义和逻辑上的功能描述
b) 栈的抽象数据描述和 C++类的表示（自定义）
c) 采用一维数组或链表的栈的表示和操作的实现 （add, delete）
d) 栈的应用，算数表达式中的括号配对，表达式的前缀和后缀表达式的计算，Hanoi 塔的可视化描述。知道递归函数的实现是通过栈实现的。以及后面的图的深度优先遍历的实现等。
7) 第六章：队列 (FIFO)
a) 队列的定义和逻辑上的功能描述
b) 队列的抽象数据描述和 C++类的表示
c) 采用一维数组或链表的栈的表示和操作的实现 （add, delete），各自的优缺点是什么？采用一维数组时，如何把数组看做是环状的数据结构，此时队列满和空是如何判断的？
d) 当采用链表表示队列时，add 和 delete 是怎么实现的？在表头和表尾的选择根据是什么？
e) 列举一些书中有哪些队列的应用。知道对图的广度优先遍历是通过队列实现的。
8) 第七章 跳表和 hash
a) SortedChain 的面向对象的描述、特点和基本函数 search, insert, delete 的实现。
b) SkipList 的组织结构、主要的特点 (利用链表元素已经排好序的特点，采用和一维数组折半查找的思想，缩小检索空间。在实现技术上是采用链表实现的)。
c) 采用 Hash 的基本思想(大空间数据到小空间存储空间的映射)，常用 hash 函数是什么？Hash 表的两种基本的表示（ hashing with linear open addressing 线性开型寻址散列和链表散列）方法能叙述。对应两种表示各自对 search, insert 和 delete 的实现方法，特别是发出冲突时的处理方法。
9) 第八章：二叉树和其他树
a) 关于树的定义和基本术语（根、树的高度、孩子、双亲、叶子、顶点的度）；
b) 二叉树的定义和基本术语 （根、左子树、右子树、空树、满二叉树和完全二叉树）
c) 数学表达式的二叉树表示（相互转化的对应关系）
d) 二叉树的性质（高度和顶点个数的关系，满和完全二叉树高度和顶点个数关系、双亲和孩子标号的关系，以及采用一维数组的存储对应方式）。
e) 二叉树的存储 （线性数组和链表）和 C++的类的表示方法；
f) 知道二叉树是一种递归的动态存在结构，能够熟练地采用递归程序解决二叉树的编程，如二叉树的三种遍历方式（会走例子）和写出对应的程序，树的高度计算等；
g) 其他树（森林）的二叉树表示，以及利用二叉树表示集合时，对集合操作的实现。
10) 第九章：优先队列(Priority Queues) heap
a) 最大（最小）树以及 Heap (堆)的定义和表示；、
b) Heap 的一维数组存储方式、类定义和基本操作（search, delete, insert）和 Initialize 的实现;
c) 高度优先左高树 (HBLT)定义以研究意义是什么（树的合并等操作更加方便）
d) Heap sort 实现算法和程序；
e) 霍夫曼编码的思想、霍夫曼树及实现算法。给定一组字符和出现的频率,有能力建立霍夫曼树.

11) 竞赛树 Tournament Tree
a) 赢者（输者）树的定义；
b) 采用上述树的排序算法思想描述

c) 采用赢者树实现外部排序的算法描述；

12) 搜索树 Search trees

a) 二叉搜索树、AVL 树 、B 树 和 B+树的定义；

b) 二叉搜索树的表示和基本操作（search, insert, delete） 的实现算法即走例子描述其过程；

c) AVL 树的性质（高度和节点数关系）、平衡因子的定义，在 insert 后不平衡的类型及对应的旋转修正方法描述（可以用图）；

d) m 叉搜索树和 B 树的定义，性质和关于 insert,delete 操作后对树的修正算法的思想描述（图描述）

e) B 树和 B+树的异同点是什么？B+树的搜索和插入和删除操作实现的思想。

13) 图 （graph）

a) 图的定义和分类（网络、有向（赋权）图、无向（赋权）图、术语（顶点度、路径、回路）,和第八章定义的区别。最短路径和最小生成树。

b) 图的两种表示方法和对应的采用图和 C++描述。

c) 图的性质（顶点数和边数），稀疏图适合的表示方法；

d) 图的深度和广度优先遍历算法的描述和实现程序。算法的时间和空间复杂性分析。

e) 图遍历算法的应用：图的连通分支的计算、深度和广度优先生成树。

14) The Greedy Method

a) 贪婪算法的思想及特点；

b) 算法及实现：拓扑排序、二部图顶点覆盖、单原点到其他所有顶点的最短路径 Dijkstra 算法、最小生成树算法（Prim 和 Kruskal）的思想描述（伪代码）。

15) 分而治之算法

a) 该方法的思想描述

b) 主要应用算法：归并排序、快速排序和找无序数组中第 k 大（小）元素。

16) 动态规划

a) 该算法的思想已经和分而治之算法的区别

b) 矩阵链的计算和所有顶点之间最短路算法。

题型:问答,选择,按算法走例子,编程实现