

中国地质大学（北京）

《数据结构》课程设 计报告

课程名称：数据结构课程设计

任课教师：郑春梅

学 时：48 学时

开课院系：信息工程学院

开课时间：2018 年 3 月

《数据结构》课程设计报告

目录

第 1 章 线性表实验	6
1.1 实验完成情况	6
题目一：顺序表各项操作	6
题目二：单链表各项操作	7
题目三：双链表的各项操作	7
实训项目：	8
1.2 约瑟夫环问题	8
1.2.1 问题再现	8
1.2.2 需求分析	8
1.2.3 功能分析	8
1.2.4 详细设计	12
1.2.5 运行结果	13
1.2.6 总结与展望	13
第 2 章：栈、队列实验	14
2.1 实验完成情况	14
题目一：栈的各项操作	14
题目二：队列的各项操作	14
实训项目：	15
2.2 非递归方式求解迷宫问题	16
2.2.1 问题再现	16
2.2.2 需求分析	16
2.2.3 功能分析	16
2.2.4 详细设计	20
2.2.5 运行结果	22
2.2.6 总结与展望	22

第 3 章：串和数组实验	22
3.1 实验完成情况	22
题目一：最长重复子串求解	22
题目二：kmp 算法匹配	23
题目三：稀疏矩阵转置	23
题目四：广义表基本运算	23
题目五：求解马鞍点	23
题目六：课后习题	24
3.2 最长重复子串的求解	24
3.2.1 问题再现	24
3.2.2 需求分析	24
3.2.3 功能分析	24
3.2.4 详细设计	26
3.2.5 运行结果	27
3.2.6 总结与展望	27
第 4 章：树实验	28
4.1 实验完成情况	28
题目一：二叉树的各项操作	28
题目二：根据前、中序列构造二叉树	28
题目三：非递归方式查找祖先节点	29
题目四：线索二叉树各项操作	29
题目五：哈夫曼树的构造及编码	29
4.2 哈夫曼树构造	29
4.3 中序非递归遍历二叉树	30
4.3.1 问题再现	30
4.3.2 需求分析	30
4.3.3 功能分析	30
4.3.4 详细设计	31
4.3.5 运行结果	31

4.3.6 总结与展望.....	32
第5章：图实验	32
5.1 实验完成情况	32
题目一：图的存储方式	32
题目二：图的遍历算法	32
题目三：顶点的入度及出度	33
题目四：prim 算法以及 kruskal 算法	33
题目五：迪杰斯特拉算法	33
实训项目：	33
5.2 kruskal 求最小生成树.....	33
5.2.1 问题再现	33
5.2.2 需求分析	34
5.2.3 功能分析	34
5.2.4 详细设计	36
5.2.5 运行结果	37
5.2.6 总结与展望.....	38
第六章：查找表实验	38
6.1 实验完成情况	38
题目一：顺序查找.....	38
题目二：折半查找.....	38
题目三：哈希表操作	38
题目四：二叉排序树操作.....	39
6.2 哈希表相关操作	39
6.2.1 问题再现	39
6.2.2 需求分析	39
6.2.3 功能分析	39
6.2.4 详细设计	40
6.2.5 运行结果	42
6.2.6 总结与展望.....	42

第七章：内排序实验	43
7.1 实验完成情况	43
排序操作	43
7.2 堆排序	43
7.2.1 问题再现	43
7.2.2 需求分析	43
7.2.3 功能分析	43
7.2.4 详细设计	44
7.2.5 运行结果	46
7.2.6 总结与展望	46
第八章：课程总结与展望	46
第九章：参考文献	47

摘 要：通过本学期的数据结构的学习，了解了线性表、队列、栈等经典的数据结构及相关的算法。本报告从课程所安排的七个章节出发，将详细的介绍每一章所完成的实验内容，以及从每一章中都挑选出感触较深的内容，对问题进行剖析，提出解决方案并对方案进行评价。

关键词：数据结构 需求分析 概要设计 详细设计 展望

引 言：在计算机领域，经常能够听到这样的话：程序就是合适的数据结构搭配合适的算法。生活中的问题总是千变万化，问题数据在计算机中的存储方式也不能一概而论。由此，数据结构在程序设计中的重要性及其优势就被凸显出来。数据结构作为计算机专业的基础课程，要研究计算机加工对象的逻辑结构、在计算机中的表示形式以及实现各种基本操作的算法。而在本学期中主要学习了以下七个章节的内容。

第 1 章 线性表实验

1.1 实验完成情况

题目一：顺序表各项操作

题目要求	题目完成情况	对应文件函数名称
初始化顺序表	完成	array.cpp: clear()
使用尾插法插入元素	完成	array.cpp: push_back()
输出顺序表及其长度	完成	array.cpp: outPut(), getSize()
判断顺序表是否为空	完成	array.cpp: isEmpty()
输出顺序表的第 k 个元素	完成	array.cpp: locate()
输出某个元素的位置	完成	array.cpp: getLo()
在某个位置上插入元素	完成	array.cpp: insert()
删除第 k 个元素	完成	array.cpp: erase()
输出顺序表	完成	array.cpp: outPut()

释放顺序表	完成	array.cpp:(类析构函数)
-------	----	-------------------

完成题目：10/10

题目二：单链表各项操作

题目要求	题目完成情况	对应文件函数名称
初始化单链表	完成	forwardList.cpp:init()
用尾插法插入元素	完成	forwardList.cpp:push_back()
输出单链表及长度	完成	forwardList.cpp:outPut(),getSize()
判断单链表是否为空	完成	forwardList.cpp:isEmpty()
输出单链表的某个元素	完成	forwardList.cpp:locate()
输出元素位置	完成	forwardList.cpp:getLo()
在特定位置插入元素	完成	forwardList.cpp:insert()
删除元素	完成	forwardList.cpp:erase()
输出单链表	完成	forwardList.cpp:outPut()
释放单链表	完成	forwardList.cpp:clear()

完成题目:10/10

题目三：双链表的各项操作

题目要求	题目完成情况	对应文件函数名称
初始化双链表	完成	fList.cpp:init()
用尾插法插入元素	完成	List.cpp:push_back()
输出双链表及长度	完成	List.cpp:outPut(),getSize()
判断双链表是否为空	完成	fList.cpp:isEmpty()
输出双链表的某个元素	完成	List.cpp:locate()
输出元素位置	完成	List.cpp:getLo()
在特定位置插入元素	完成	List.cpp:insert()
删除元素	完成	List.cpp:erase()
输出双链表	完成	List.cpp:outPut()
释放双链表	完成	List.cpp:clear()

完成题目:10/10

实训项目：

题目要求	题目完成情况	对应文件函数名称
约瑟夫环问题	完成	circlelist:test()
列车时刻调度问题	完成	forwardlist:welcome()等

1.2 约瑟夫环问题

1.2.1 问题再现

约瑟夫环是一个数学的应用问题：已知 n 个人（以编号 1, 2, 3... n 分别表示）围坐在一张圆桌周围。从编号为 k 的人开始报数，数到 m 的那个人出列；他的下一个人又从 1 开始报数，数到 m 的那个人又出列；依此规律重复下去，直到圆桌周围的人全部出列。通常解决这类问题时我们把编号从 $0 \sim n-1$ ，最后结果+1 即为原问题的解。^[1]

1.2.2 需求分析

在约瑟夫环问题中，每轮报数结束后都会有一个人出队，而在 $n-1$ 轮之后，就只剩下了一个人。这个问题的需求即是求解这最后一个人在所有人中的序列。

1.2.3 功能分析

要解决这个问题，那么就需要完成下面的几个功能：

1. 对于这个问题，首先要确定初始时的人数，并且要能为每一个人保存其在原序列中的序号，即能按照人数及其序号建立线性表。

2. 由于在每一轮报数中，在序列中最后一个人报数结束后，应是序列中国年第一个人再次开始报数，所以从表中任一位置开始都能访问到所有的元素。

3. 在每一轮报数中，都会删除一个人，即删除一个节点，由此可见，这个问题会进行 $n-1$ 次删除节点的操作。

从功能 3 出发，可以确定存储的方式：

1. 顺序存储：

每一次删除都移动元素，而在顺序存储的过程中，删除元素的时间复杂度达到了 $\Theta(n)$ 。
因而在这种方式中，总的空间复杂度计算如下：

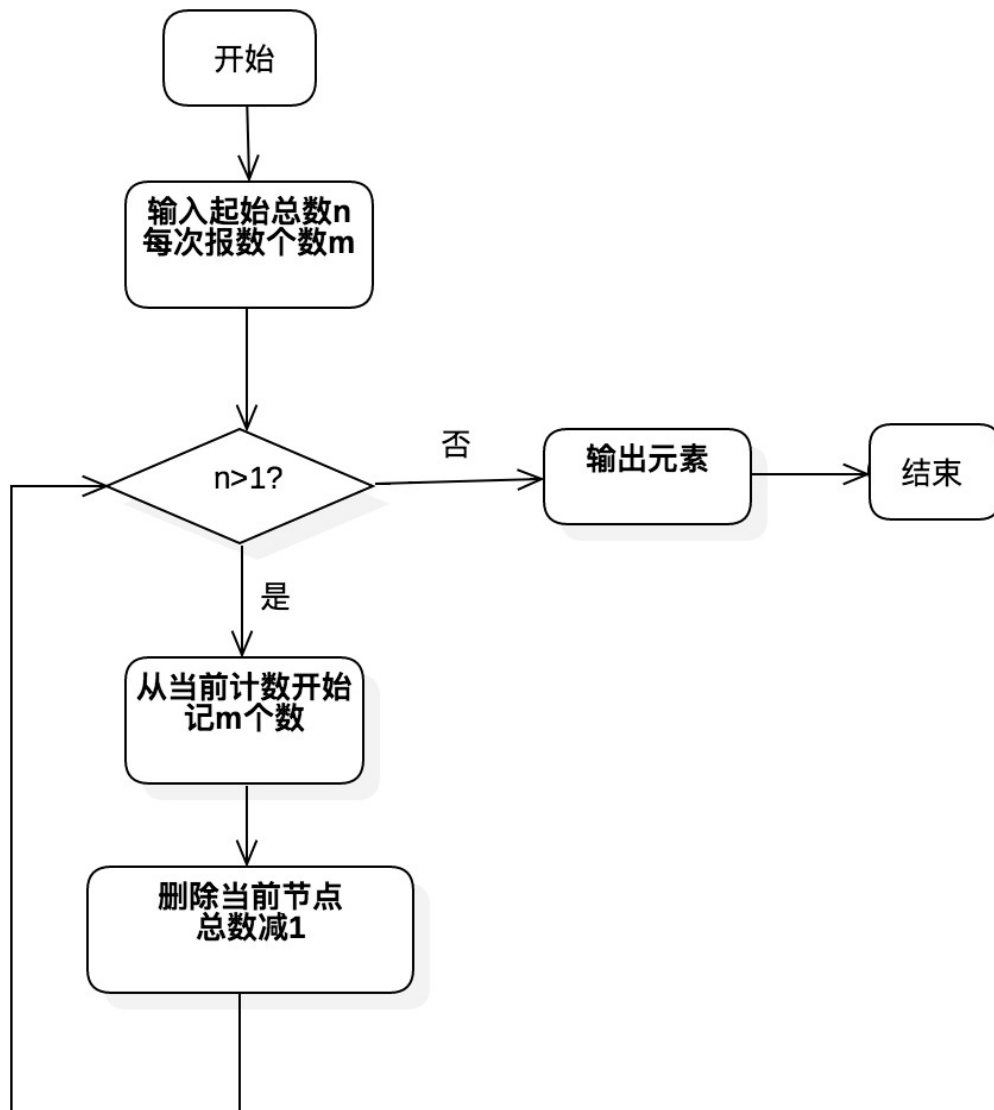
$$\Theta\left((n-1) * \left(m + \frac{n-1}{2}\right)\right) = \Theta(n^2 + nm)$$

在此方式中，由于需要申请原先元素的辅助空间，因此空间复杂度为：

$$\Theta(n)$$

并且在程序运行时，始终占据所申请的空间。

具体的流程图如下：



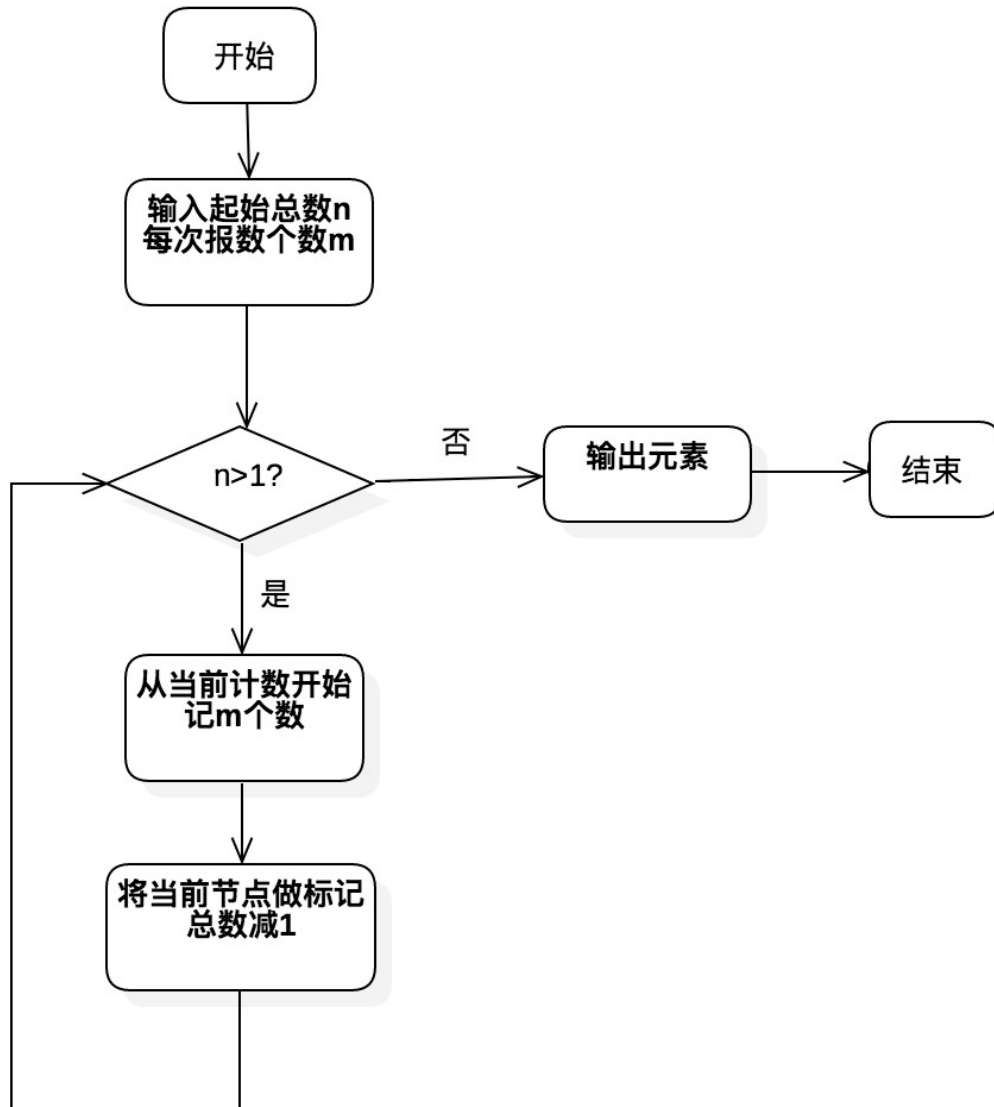
在顺序存储中，如果每一次删除不在移动元素，而是将每个元素的序号改为一个没有在原序列中出现的数字作为标记，那么这时时间复杂度可优化为：

$$\Theta((n-1) * m) = \Theta(nm)$$

同样的，空间复杂度为：

$$\Theta(n)$$

具体的流程图如下：



2. 链式存储：

在链式存储中，每一次删除操作的时间复杂度是 $\Theta(1)$ 的，同样也需要循环 $n-1$ 次，则时间复杂度的计算如下：

$$\Theta((n-1) * (m)) = \Theta(nm)$$

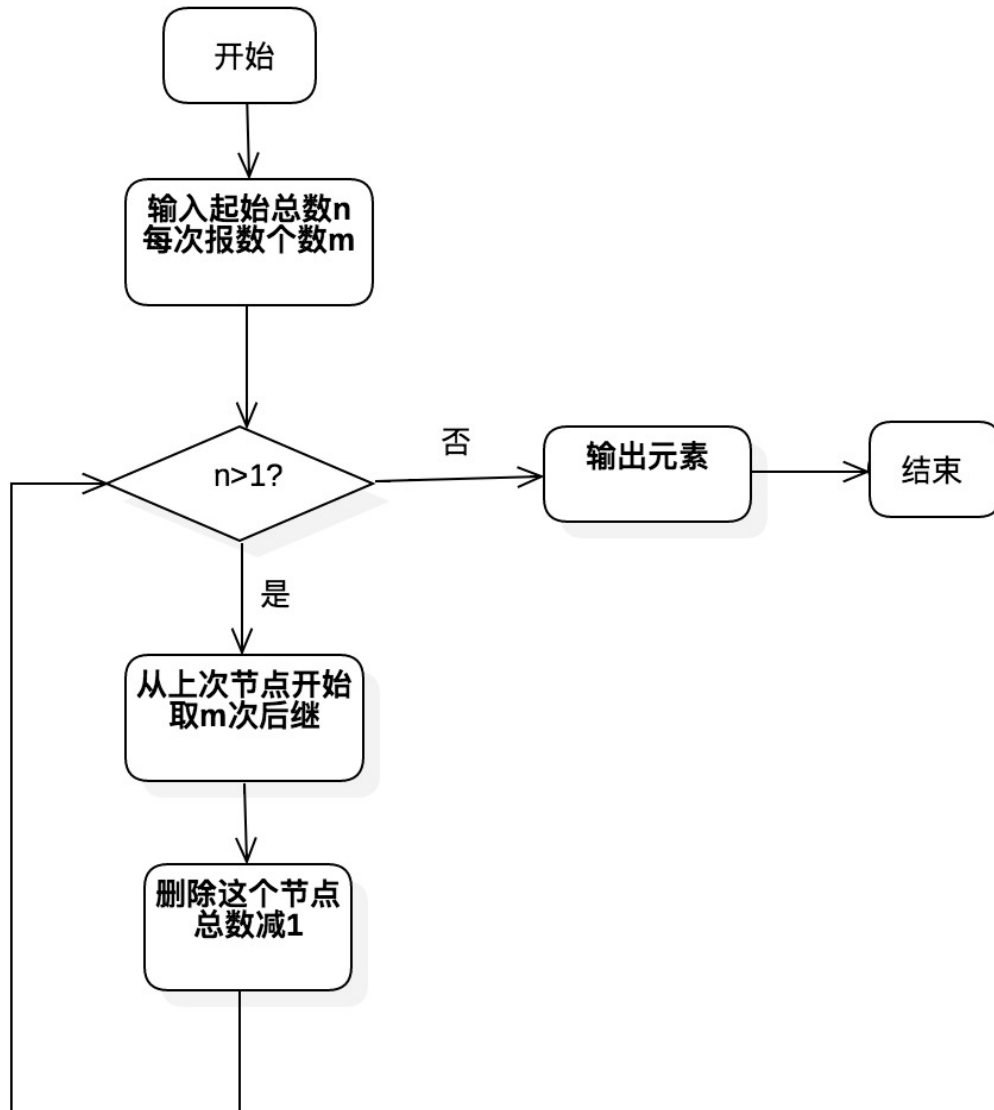
可见在链式存储的最低复杂度就已经达到了 $\Theta(nm)$ ，那么继续对空间复杂度进行分析。

假定指针类型所需要的空间与一个 `int` 型的大小相同，那么对于这个问题而言，所需的
空间大小取决于原来序列的长度，有：

$$\Theta(n * 2) = \Theta(n)$$

到这里，时间复杂度与空间复杂度与顺序存储都已经相同，可是在运行时，由于每一轮都需要删除一个元素，那么最后链式存储所占的空间仅为 $\Theta(1)$ 。比顺序存储更为高效。

具体的流程图如下：



由此可以看出，在顺序存储与链式存储中，链式存储的优势更大，因此在这个问题中选择链式存储。

从功能 2 出发，由于从每一个节点出发都要能够访问到所有的元素，因此选择循环链表作为实际的结构来解决这个问题。

1.2.4 详细设计

该实验是建立在循环链表的基础之上的，使用到的循环链表的函数以及功能如下：

void init(): 初始化循环链表

node<int> push_back(int k): 使用尾插法向循环链表中插入元素

int getSize(): 获取队列的长度

node<int> erase(int n): 删除第 *n* 个元素

node<int> getStart()*: 获取循环链表的头指针

node<int> getRight()*: 获取当前节点的后继

int getNum(): 获取当前节点在原序列中的序号

具体代码：

在代码中是选择了带头节点的循环链表，因此在取后继时需要判断是否是头结点，如果是头结点，直接跳过。

```
int fun() {  
    int m, n;  
    cin >> n >> m; // 输入队列长度以及间隔数  
    if (n <= 0 || m <= 0) return -1; // 判断数据合法性  
  
    auto now = list.getStart();  
    for (int i = 1; i <= n; ++i) {  
        list.push_back(i);  
    } // 创建循环链表  
  
    while (list.getSize() > 1) { // 外层控制链表长度  
        for (int i = 0; i < m - 1; ++i) { // 取 m 次后继  
            if (now == list.getStart()) now = now->getRight();  
            now = now->getRight();  
            if (now == list.getStart()) now = now->getRight();  
        }  
        int n = list.getLo(now->getNum());  
        cout << now->getNum() << "->"; // 输出此次删除的节点序号  
        now = list.erase(n); // 删除节点  
    }  
}
```

```

        if (now == list.getStart()) now = now->getRight();
    }

    return list.getStart()->getRight()->getNum();//返回结果
}

```

1.2.5 运行结果

```

% ./a.out
2 2
2->1
% ./a.out
5 2
2->4->1->5->3
% ./a.out
5 4
4->3->5->2->1
% █

```

1.2.6 总结与展望

在这个问题中,如果不将其只仅仅局限于从数据结构来解决它,则从数学角度进行分析,可以得到更优复杂度的解决方案。分析如下:

首先,如果只有一个节点,那么输出元素,程序结束。即

$$F(1) = 1$$

当元素个数大于 1 时,则可以将上一次的元素看作 0,其后继看作 1……,那么这个时候就有:

$$F(n) = (F(n-1) + m) \% n$$

这个时候由于每一次循环中都是一个常量,时间复杂度为:

$$\Theta(n-1) = \Theta(n)$$

可见,这个时候的时间复杂度要远远优于前面的方式,而且只需要记录上一次删除的序号,时间复杂度也只有 $\Theta(1)$,可见要优于前面的方式。但这种方式已经超出了课程学习的本质目的,所以不再进行实现。

对于这个问题,在数据结构方面我了解到了顺序表以及链表在不同环境下复杂度会有很大的差别。更加体会到对于不同的问题,要选取合适的数据结构来处理这个问题,不同的数据结构在处理问题时复杂度会有很大的不同。在链表中,增加头结点的目的是统一操作,不必再对首元结点进行特殊处理,但是在这个问题中,增加头结点之后,由于头结点不含有

原来的序号，即不参与报数，处理时却恰恰需要为其增加特殊处理，显得更加画蛇添足。除此之外，在程序设计方面，我更了解到程序应该是数据结构与算法相辅相成，不能仅仅只将目光局限与其中一个上，比如在这个问题中，如果不加限制，我认为最后的数学解法比其他几种解法更优。在编写程序时，两者都应该兼顾，力求最优。在调试代码的过程中，链表的错误主要是非法访问内存（segment fault），这时就需要调试人静下心来慢慢的去寻找其中的非法操作。

第 2 章：栈、队列实验

2.1 实验完成情况

题目一：栈的各项操作

题目要求	题目完成情况	对应文件函数名称
初始化栈	完成	stack.cpp:类构造函数
判断栈是否为空	完成	stack.cpp:isEmpty()
栈元素依次进栈	完成	stack.cpp:push()
输出栈长度	完成	stack.cpp:getSize()
输出从栈顶到栈底的元素	完成	stack.cpp:showstack()
输出出栈序列	完成	stack.cpp:getTop()
释放栈	完成	stack.cpp:类析构函数

完成题目:7/7

题目二：队列的各项操作

题目要求	题目完成情况	对应文件函数名称
初始化队列	完成	circlequeue.cpp:init()
判断队列是否为空	完成	circlequeue.cpp:isEmpty()
元素依次进队	完成	circlequeue.cpp:push_back()
出队一个元素并输出	完成	circlequeue.cpp:pop_front()

输出元素个数	完成	circqueue.cpp:getSize()
输出出队序列	完成	circlequeue.cpp:showqueue()
释放栈	完成	circlequeue.cpp:类析构函数

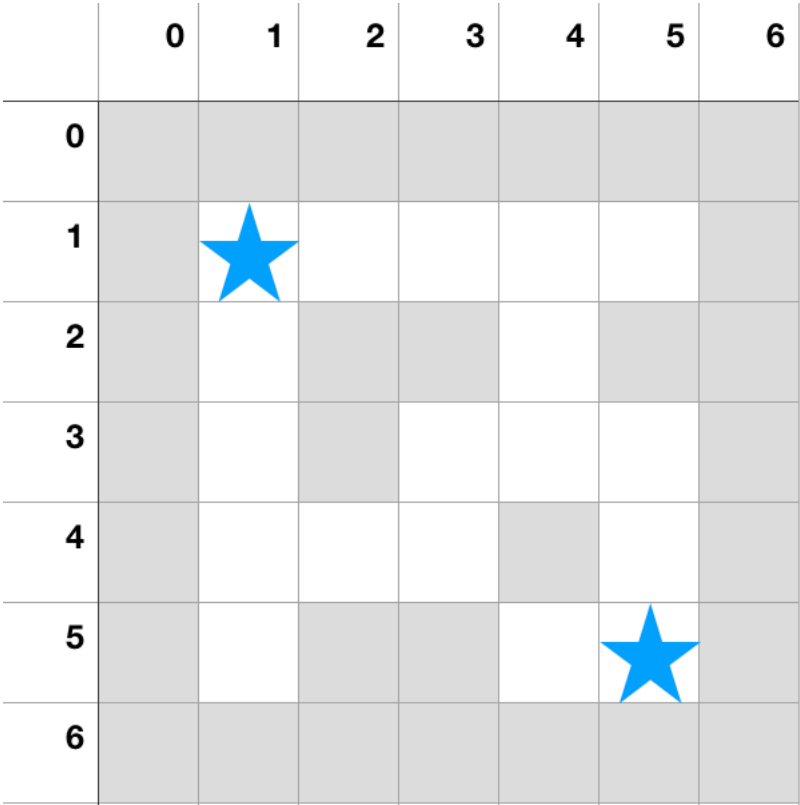
完成题目:7/8 (题目要求重复)

实训项目：

题目要求	题目完成情况	对应文件函数名称
中缀表达式转后缀表达式	完成	stack/equal.cpp
非递归方式求解迷宫	完成	stack/maze.cpp
八皇后问题	完成	stack/queen.cpp
病人看病模拟程序	完成	queue/hos.cpp

2.2 非递归方式求解迷宫问题

2.2.1 问题再现



给定这样的迷宫，从起点(1, 1)出发，终点为(5, 5)，使用深度优先搜索求解在这个图中一共有多少条道路，并且输出每条道路的路径。

2.2.2 需求分析

在这个问题中，需求关系较为简单，即从(1, 1)出发，利用深度优先搜索来求解到(5, 5)的路径。

2.2.3 功能分析

1. 图的存储

对于这个问题，首先要考虑存储该图，由于图的规模并不是很大，所以直接手动模拟存储这个图，空间复杂度为：

$$\Theta(n^2)$$

2. 求解方式

使用深度优先搜索进行求解。

首先考虑时间复杂度，在这个问题中，在图中的每一个点，最多只有 4 个方向，而深度优先搜索中，对于每一个点，他所有的方向都要遍历一边，那么最后的时间复杂度即为：

$$\Theta(4v)$$

而在这个问题中，时间复杂度为：

$$\Theta(4n^2) = \Theta(n^2)$$

其次考虑空间复杂度，在这个问题中，对于每一个点，需要记录其遍历的方向，而每一个点最多的方向个数为 4，那么只需利用一个数字来代表其遍历的方向即可，这里所申请的辅助空间的大小为：

$$\Theta(n^2)$$

其次，由于在遍历过程中，需要记录此次搜索路径上的点已防止重复遍历，而在最差的情况下，所有的点都可能在此路径中，那么在这里所需要申请的辅助空间大小为：

$$\Theta(n^2)$$

初次之外，由于需要计算所有的路径，那么就需要另外的空间来标记所有的点是否已经完成遍历，这里的辅助空间为：

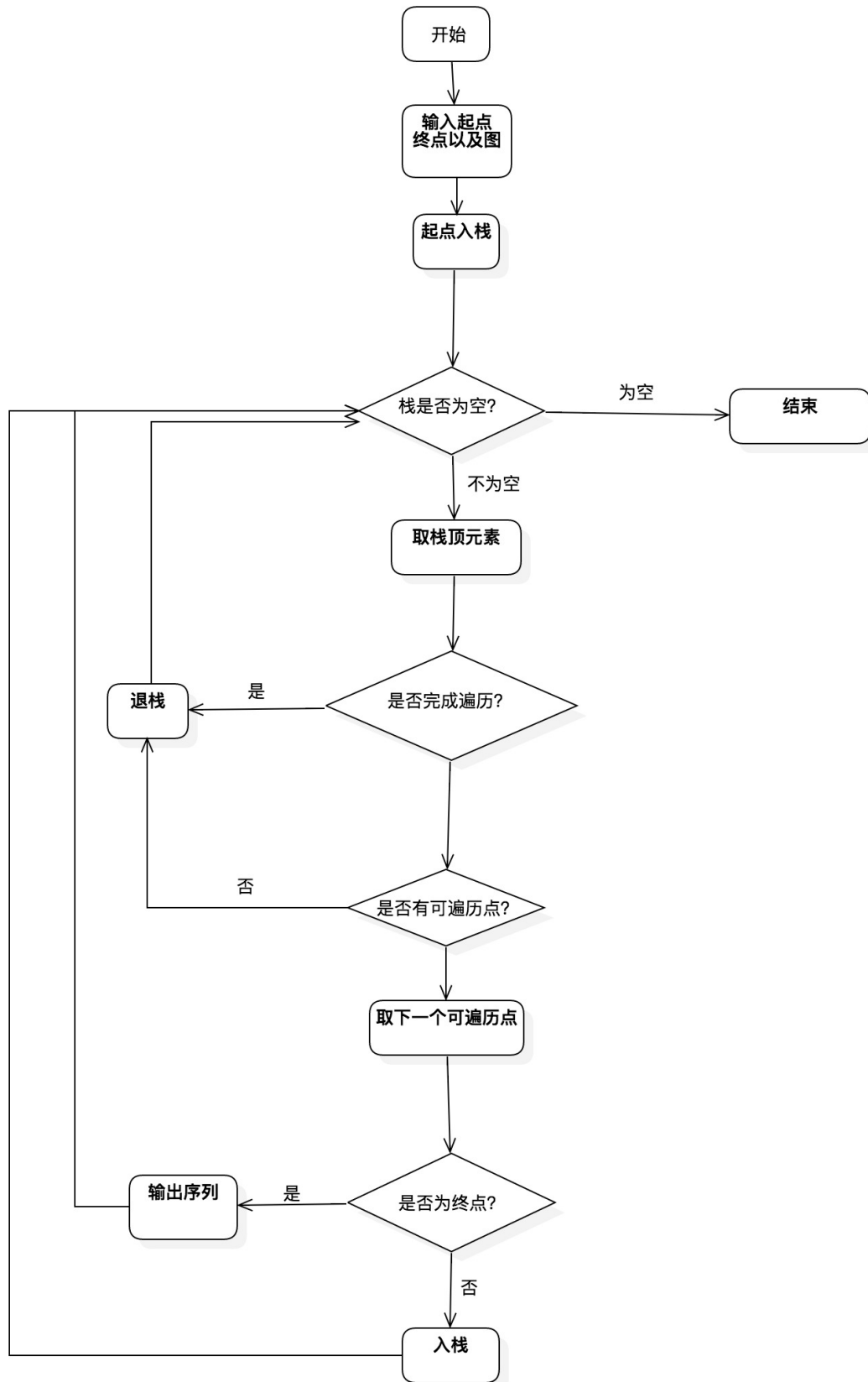
$$\Theta(n^2)$$

综上，对于这个问题，所需要的空间复杂度为：

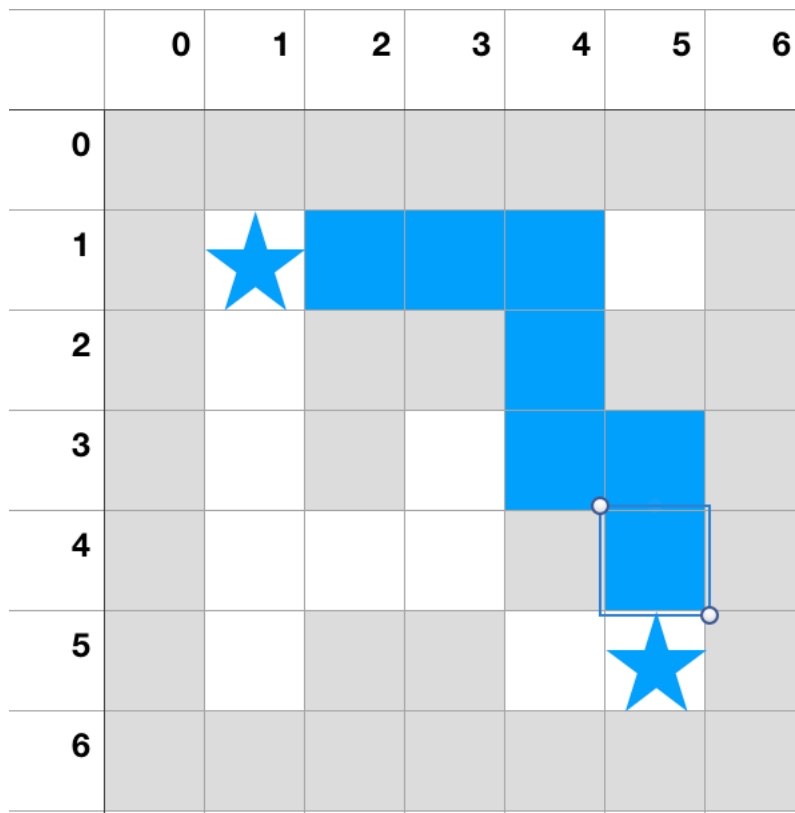
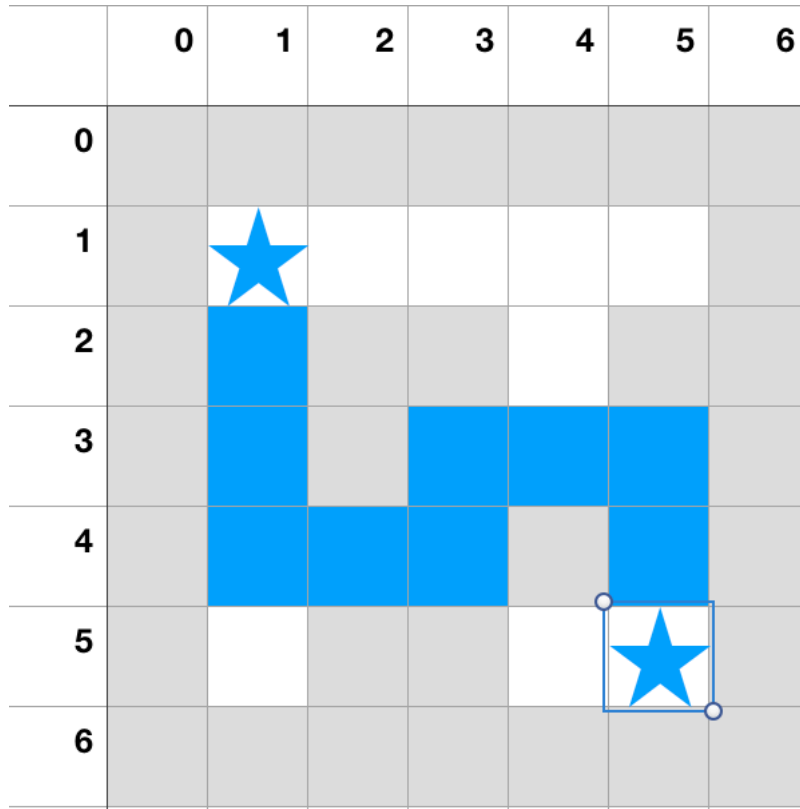
$$\Theta(3n^2) = \Theta(n^2)$$

可见这个问题的时间复杂度与空间复杂度都已经达到了 $\Theta(n^2)$ 。

3. 求解流程：



4. 问题答案



共如图两种方式以及完成遍历。

2.2.4 详细设计

在这个问题中需要用到栈来完成求解，栈的基本函数及功能如下：

`bool isEmpty()` //判断栈是否为空

`pair<int, int> getTop()` //取栈顶元素

`bool pop()` //退栈元素，返回是否退栈成功

`bool push()` //入栈操作

具体代码：

`//pii = pair<int, int>`

`void dfs() {`

`stack<pii> st; //st 即为搜索时路径上的顶点栈`

`st.push(pii(1, 1)); //起点入栈`

`mark[1][1] = 1;`

`{`

`stack<int> tem;`

`tem.push(-1);`

`ss.push(tem);`

`} //保存起点的遍历方向, 这里使用了一个栈来保存每个节点的遍历方向, 空间复杂`

度更大, 是一个可以优化的地方。

`while (!st.isEmpty()) {`

`auto c = st.getTop(); //取栈顶元素`

`if (c.first == 5 && c.second == 5) { //已访问到终点, 输出路径`

`anss.insert(st);`

`st.pop();`

`ss.pop();`

`mark[c.first][c.second] = 0;`

`continue;`

`}`

`int start = ss.getTop().getTop() + 1; //取当前点遍历方向`

`if (start >= 4) { //如果当前点已完成遍历`

`st.pop();`

`ss.pop(); //退栈`

```

        mark[c.first][c.second] = 0; //当前点已不在路径中
        continue;
    }

    int cc = 0;
    for (; start < 4; ++start) { //判断该点是否还有下一个可达点
        int nx = c.first + dir[start].first;
        int ny = c.second + dir[start].second;
        ss.getTop().push(start);
        if (test(nx, ny)) { //如果是可达点，将其入栈，并且进行标记
            mark[nx][ny] = 1;
            st.push(pii(nx, ny));
            stack<int> tem;
            tem.push(-1);
            ss.push(tem);
            ++cc;
            break;
        }
    }

    if (cc == 0) { //如果该点没有下一个可达点，退栈
        st.pop();
        ss.pop();
        mark[c.first][c.second] = 0;
    }
}

return;
}

//用于判断是否是可达点
bool test(int x, int y) {
    return (G[x][y] == 1) && (mark[x][y] == 0);
}

```

2.2.5 运行结果

```
% ./a.out
Total:2
1,1 1,2 1,3 1,4 1,5 3,4 3,5 4,5 5,5
1,1 1,2 1,3 1,4 1,5 3,4 3,5 4,5 5,5 4,1 5,1
%
```

2.2.6 总结与展望

在这个问题中的代码实现上，由于自己思路不够严谨，浪费了一些空间，只需用一个数字来记录每个点的遍历方向即可，但是在实现上每一个点都使用了一个栈来记录每一个点的遍历方向，最坏情况下可能浪费了 3 倍的空间，确实是自己实现上的问题。非递归的实现方式相较与递归方式的实现而言，在实现方式上更加复杂，但是却在系统层面上省去了内存中断的过程，也是需要掌握的一项技术。

在以后实现代码的过程中，要做到尽可能的不再浪费空间，以求达到复杂度最优，正如上面的情况，如果原问题的输入数据在 $1e^9$ 以上，那么就会浪费 $3e^9$ 的空间，因为不知道将来问题的规模究竟会有多大，所以才要做到最优，否则即使是很小浪费的复杂度，经过成百上千万的数据的放大作用，最终也会是一个庞大的浪费。

第 3 章：串和数组实验

3.1 实验完成情况

题目一：最长重复子串求解

题目要求	题目完成情况	对应文件函数名称
求解最长重复子串	完成	mls.cpp

题目二：kmp 算法匹配

题目要求	题目完成情况	对应文件函数名称
kmp 算法	完成	kmp.cpp

题目三：稀疏矩阵转置

题目要求	题目完成情况	对应文件函数名称
生成三元组	完成	adt.cpp:main() 中生成 , output()输出
输出转置矩阵	完成	adt.cpp:get()
实现 a+b 三元组	完成	adt.cpp:add()

题目四：广义表基本运算

题目要求	题目完成情况	对应文件函数名称
建立广义表	完成	test.cpp:create()
输出深度	完成	test.cpp:getheight()
输出长度	完成	test.cpp:getLength()

题目五：求解马鞍点

题目要求	题目完成情况	对应文件函数名称
求解马鞍点	完成	horse.cpp:getans()

题目六：课后习题

题目要求	题目完成情况	对应文件函数名称
递归方式求最大值，和，平均值	完成	dfs/3.14.cpp

3.2 最长重复子串的求解

3.2.1 问题再现

给定一个字符串，求解其中出现过两次的最长的子串。例如：s=“aababcabcdabcde”，最长重复子串为：abcd

3.2.2 需求分析

在这个问题中，输入，所求问题相较而言都比较清晰。即给定一个字符串，在这个字符串中求解出重复出现过的最长的子串。

3.2.3 功能分析

1. 存储字符串。
2. 能快速截取子串，
3. 子串能与原字符串进行快速对比

首先对于这个字符串而言，由于需要多次与原串进行比对，所以需要保存原字符串。但是在这个问题中，更多的是需要随机对原串进行访问而非对其进行删除或者插入。所以在存储方面选择更简单而且更加节约空间的顺序存储即可。

最长公共子串是一个非常经典的问题，对于其上的算法也有很多。

1. 暴力求解

穷举子串所有的起点与终点的搭配来获得子串，在获得子串之后再对子串与原串进行匹配。逻辑最为简单，时间复杂度最高。复杂度的推导过程如下：

对于每一个起点而言，需要穷举所有的终点来获得子串，那么这里的时间复杂度为：

$$\Theta\left(n * \frac{n}{2}\right) = \Theta(n^2)$$

再获得子串之后，仍需要将子串与原串进行比对，如果只使用朴素匹配的，时间复杂度为：

$$\Theta(n * m) = \Theta(nm) = \Theta(n^2)$$

那么这就是最为朴素的算法，总的时间复杂度为：

$$\Theta(n^2 * n^2) = \Theta(n^4)$$

由于只需要为子串申请额外的空间，即只需要记录其首尾，则空间复杂度为：

$$\Theta(1)$$

如果在比对的过程中使用了 kmp 快速模式串匹配，则比对时的时间复杂度为：

$$\Theta(n + m)$$

那么此时总的时间复杂度为：

$$\Theta(n^2 * (n + m)) = \Theta(n^3)$$

此时空间复杂度即为 kmp 算法的空间复杂度：

$$\Theta(n)$$

2. 二分查找区间长度，再进行模式串匹配

对于上面的暴力查找而言，二分子串长度则是在获取子串上进行了优化。其复杂度推导如下：

首先需要二分子串长度，其时间复杂度为：

$$\Theta(\log n)$$

其次则需要穷举所有这个长度的子串来获得子串，其时间复杂度为：

$$\Theta(n - m + 1)$$

那么获取子串的总时间复杂度为：

$$\Theta(\log n * (n - m + 1)) = \Theta(n \log n)$$

在获取子串之后，仍需要对其进行匹配，其复杂度与前面相同，那么有：

朴素匹配时间复杂度：

$$\Theta(n^3 \log n)$$

空间复杂度：

$$\Theta(1)$$

Kmp 匹配时间复杂度：

$$\Theta(n \log n * (n + m)) = \Theta(n^2 \log n)$$

空间复杂度：

$$\Theta(n)$$

3. 后缀数组解法

由于字符串的所有后缀即为字符串的所有子串，那么要求最长重复子串，只需要将后缀字符串数组按照字典序进行排序然后再比对相邻字符串的相同的字符长度。其复杂度推导如下：

首先要获取所有的后缀，其时间复杂度为：

$$\Theta(n)$$

然后需要保存所有的后缀，其空间复杂度为：

$$\Theta\left(\sum_{m=1}^n m\right) = \Theta\left(\frac{n^2 + n}{2}\right) = \Theta(n^2)$$

接下来需要对其进行排序，由于是字符串的比较，所以排序的时间复杂度为：

$$\Theta(n^2 \log n)$$

接下来是对所有相邻字符串的比较：

$$\Theta(n^2 * n) = \Theta(n^3)$$

最后，总的时间复杂度为：

$$\Theta(n^3 + n + n^2 \log n) = \Theta(n^3)$$

而总的空间复杂度为：

$$\Theta(n^2 + n \log n) = \Theta(n^2)$$

对于后缀数组，还有更为快速的倍增算法和 DC3 算法来求解这个问题，在此不在进行分析。

3.2.4 详细设计

```
//str 为原字符串
int nfind() {
    string back(str);
    vector<string> strs;
    while (back.size() != 0) { //获取所有后缀，在这里的写法复杂度较高，可以优化
        strs.push_back(back);
        back.erase(back.begin());
    }
    sort(strs.begin(), strs.end()); //所有的后缀排序
    int maxn = -1;
    for (int i = 0; i < strs.size() - 1; ++i) { //从所有的相邻后缀中获得最长重
```

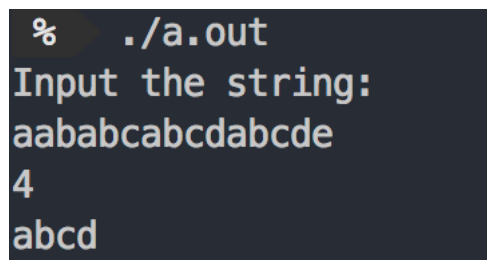
复子串

```
int j;  
string tem;  
tem.clear();  
for (j = 0; j < strs[i].size() && j < strs[i + 1].size(); ++j) { //子串
```

比对

```
    if (strs[i][j] != strs[i + 1][j])  
        break;  
    tem = tem + strs[i][j];  
}  
if (j > maxn) {  
    ans = tem;  
    maxn = j;  
}  
}  
return maxn;  
}
```

3.2.5 运行结果



A terminal window with a dark background. The prompt is '% ./a.out'. The user has entered 'Input the string: aababcbcdabcde'. The program has output '4' and 'abcd'.

3.2.6 总结与展望

最长重复子串是一个非常经典的问题，但是对于这个问题，可以看出以上给出的三个解法时间复杂度都相对比较高，即使是一个简单的问题，其解决方案也可能会十分的复杂，正如经典的 TSP 问题，即使问题描述非常的简单，但其复杂度也非常的高。所以在以后的实践中，对于一个问题应该仔细分析而不是盲目下手，否则只会事倍功半，白费努力。同时对于一个问题也有很多不同的解法，应当从其中选取合适的解法来解决问题。

最后这个问题也可以用动态规划法来解决，但是我认为这种解法已经远远超出了数据结构所研究的范围，所以不再加以赘述。

第 4 章：树实验

4.1 实验完成情况

题目一：二叉树的各项操作

题目要求	题目完成情况	对应文件函数名称
创建二叉链表存储结构	完成	bintree.cpp:create()
二叉树的前、中、后序递归算法	完成	bintree.cpp:pre(), in(), aft()
二叉树中序非递归算法	完成	bintree.cpp:inno()
二叉树前序后序非递归算法	未完成	
二叉树广义表形式输出	完成	bintree.cpp:outlist()
输出指定结点左右孩子	完成	bintree.cpp:outlr()
输出叶结点数目	完成	bintree.cpp:getleave()
输出树的高度	完成	bintree.cpp:getheight()
输出结点的层次	完成	bintree.cpp:getfloor()
释放二叉树	完成	bintree.cpp:类析构函数

完成必做题目:6/6 完成选做题目:2/4

题目二：根据前、中序列构造二叉树

题目要求	题目完成情况	对应文件函数名称
构造二叉树	完成	qzh.cpp:build()
输出后序序列	完成	qzh.cpp:aft()

完成题目:2/2

题目三：非递归方式查找祖先节点

题目要求	题目完成情况	对应文件函数名称
非递归方式查找祖先节点	完成	bintree.cpp:parent.cpp()

题目四：线索二叉树各项操作

题目要求	题目完成情况	对应文件函数名称
二叉树线索化	完成	bintree.cpp:inThread()
非递归方式输出中序序列	完成	bintree.cpp:in2()
寻找前驱节点	未完成	
寻找后继节点	完成	bintree.cpp:next()

题目五：哈夫曼树的构造及编码

题目要求	题目完成情况	对应文件函数名称
哈夫曼树的构造	完成	haffman.cpp:buildtree()
哈夫曼树的编码	完成	haffman.cpp:code()

4.2 哈夫曼树构造

在所提交的代码中，哈夫曼树的构造方式略有不同。在代码中使用了优先队列即堆结构来维护整棵哈夫曼树，时间复杂度达到了：

$$\Theta(n \log n)$$

而哈夫曼树的另两种方式有：

1. 顺序遍历其现有节点获取其权值最小的两个节点，合并节点之后再放入原集合中。其时间复杂度为：

$$\Theta(n * (n - 1)) = \Theta(n^2)$$

2. 首先将节点集合进行排序，每次再取权值最小的两个节点，然后再使用插入排序方法

将其插入到对应位置，其时间复杂度为：

$$\Theta(n \log n + n * (n - 1)) = \Theta(n^2)$$

可见，对于以上三种方式而言，复杂度最低的为优先队列方式的实现。

4.3 中序非递归遍历二叉树

4.3.1 问题再现

利用非递归方式求解二叉树中序序列并输出。

4.3.2 需求分析

输出二叉树中序遍历序列

4.3.3 功能分析

1. 对于这个问题而言，需要满足下面几个功能：

1. 对于二叉树而言，首先需要选择合适的方式进行存储。
2. 对于每一个节点而言，能够直接访问到它的左右孩子节点。
3. 能够进行二叉树的遍历。

1. 数组存储

在这个问题中，由于树的节点有可能会非常多，如果选用数组模拟的话，首先要确定树结点的大小，而且在节点数很多的情况下可能会浪费很多的空间，而且在这个问题中，并不需要随机访问节点，所以利用数组顺序存储并没有很大的优势。

2. 二叉链表

利用二叉链表存储节点的话，每一个节点则多了左右孩子指针，但是在扩展方面是比较方便的，对于不定大小的树可以快速增加节点，动态分配内存，灵活性高，而且在这个问题中，并不需要随机访问节点，而且在实现方面相对于数组也更为简单，因此在这个问题中选取了二叉链表作为实现方式。

2. 求解方法

对于中序遍历二叉树而言，首先需要取每一个节点左子树中最左边的节点，在访问过其之后，再访问该节点，最后再访问其右子树。由于每一个节点都仅被访问一次，那么最后的

时间复杂度为:

$$\Theta(n)$$

而由于不需要申请其他的辅助空间, 那么空间复杂度为:

$$\Theta(1)$$

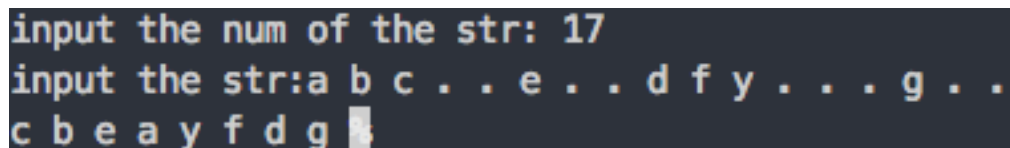
在递归的遍历中, 只需要先对每个节点的左子树进行递归遍历, 再访问这个节点, 最后再对每个节点的右子树进行递归遍历, 即完成了中序遍历。

但是在非递归的遍历中, 需要先利用栈来保存在该节点之前访问过的节点, 当其左子树都遍历完成之后, 再遍历这个节点, 遍历结束后再退栈, 访问其右子树。

4.3.4 详细设计

```
void inno() { //中序非递归排序
    stack<node<T>*> ss; //用来保存访问过的节点
    node<T> *now = root;
    do {
        while (now != nullptr) {
            ss.push(now);
            now = now->left;
        } //遍历其左子树的最左端叶结点
        if (!ss.empty()) {
            now = ss.top(); //取其栈顶节点
            ss.pop(); //退栈
            cout << now->data << " "; //访问节点
            now = now->right; //访问其右子树
        }
    } while (now != nullptr || !ss.empty());
}
```

4.3.5 运行结果



```
input the num of the str: 17
input the str: a b c . . e . . d f y . . . g . .
c b e a y f d g
```

4.3.6 总结与展望

对于这个问题而言，仍然是递归方式与非递归方式的对比，在这个问题中，递归方式写法较为简单，只需要短短几行，但是在操作系统层面上而言却需要进行寄存器的中断与恢复。而非递归方式在写法上较为复杂，但是在系统上没有更多的消耗。而且在编写非递归遍历的过程中，也能更加深入的了解到树的中序遍历的流程，掌握将递归过程该为非递归的方法。

第5章：图实验

5.1 实验完成情况

题目一：图的存储方式

题目要求	题目完成情况	对应文件函数名称
建立邻接矩阵	完成	graph/1.cpp:getG1()
由邻接矩阵产生邻接表	完成	graph/1.cpp:getG3()
由邻接表产生邻接矩阵	完成	graph/1.cpp:getG2()

题目二：图的遍历算法

题目要求	题目完成情况	对应文件函数名称
递归深度优先遍历	完成	graph/2.cpp:dfs()
非递归深度优先遍历	完成	实验二已完成
广度优先遍历		graph/2.cpp:bfs()

题目三：顶点的入度及出度

题目要求	题目完成情况	对应文件函数名称
顶点入度及出度	完成	graph/3. cpp

题目四：prim 算法以及 kruskal 算法

题目要求	题目完成情况	对应文件函数名称
prim 算法	完成	graph/4. cpp:prim()
kruskal (选做)	完成	graph/4. cpp:kru()

题目五：迪杰斯特拉算法

题目要求	题目完成情况	对应文件函数名称
迪杰斯特拉算法	完成	graph/6. cpp

实训项目：

题目要求	题目完成情况	对应文件函数名称
拓扑排序	完成	graph/5. cpp

5.2 kruskal 求最小生成树

5.2.1 问题再现

一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。^[2]

5.2.2 需求分析

利用 kruskal 算法求解最小生成树。

5.2.3 功能分析

1. 图的存储方式

对于无向图而言，常用的存储方式一般有两种，即邻接表和邻接矩阵。在邻接表的存储方式中，数组中保留的是边的权重或者是否邻接其空间复杂度为：

$$\Theta(n^2)$$

而图的邻接矩阵则是保存边的信息，其顶点表需要的空间复杂度为：

$$\Theta(3n)$$

其邻接表中，每一个节点都需要保存其终点以及权重信息，如果它指向后继指针的大小也计算在内，它的空间复杂度为：

$$\Theta(3E * 2) = \Theta(6E)$$

其中 n 为顶点数目， E 为无向图的边数。

当 $3n + 6E < n^2$ 时，采用邻接矩阵存储方式，反之则采用邻接表的存储方式。

但是在 kruskal 算法中，由于只需要用到所有的边，所以只需要存储所有的边即可，此时空间复杂度为：

$$\Theta(3E * 2) = \Theta(6E)$$

2. 求解方法

kruskal 是根据每一条边来求解最小生成树的，那么对于每一条边，首先要判断这条边的两个顶点是否已经联通，如果两个顶点已经联通，如果将这条边在加入到生成树的集合中，那么这个生成树就含有了环，其判断方法往往是依靠并查集来实现的，判断图的连通性也可以使用深度优先搜索和广度优先搜索。

首先分析并查集的复杂度：

由于并查集需要为每一个顶点再申请一个辅助数组来记录其集合编号，那么它的空间复杂度为：

$$\Theta(n)$$

朴素的并查集，每次查找都需要不断的去寻找它的父节点，知道找到最后的根节点，这样，最坏情况是：

$$\Theta(n)$$

在进行优化后，并查集的平均复杂度可以达到：

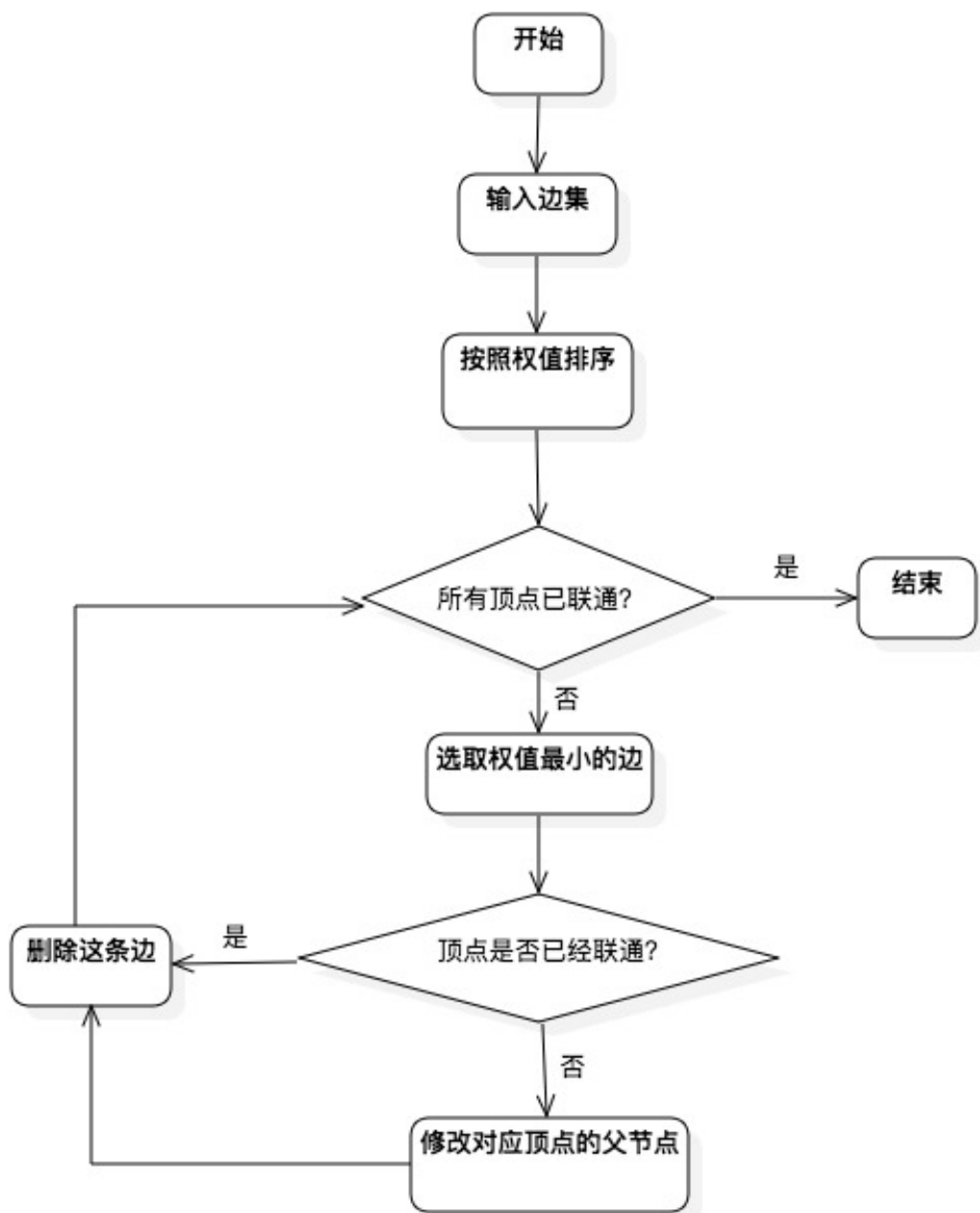
$$\Theta(\log n)$$

而在 kruskal 算法中，由于每一次选择边的时候都需要进行并查集的查找，那么最后的时间复杂度为：

$$\Theta(2E * 2 \log n) = \Theta(4E \log n) = \Theta(E \log n)$$

其中 E 为边的数目，n 为顶点数目。

程序流程图如下：



除此之外，kruskal 算法还可以通过优先队列即小顶堆来实现，每次只取堆顶的边，判断其端点的连通性，然后再决定是否将其加入到生成树边集中。再分析时间复杂度的时候需要考虑到堆的复杂度，即为 $\Theta(\log E)$ ，再加上并查集的复杂度 $\Theta(\log n)$ ，最后的时间复杂度为：

$$\Theta(2 * E * (\log E + \log n)) = \Theta(E \log n)$$

5.2.4 详细设计

```
//并查集递归寻找其父节点算法
int findroot(int n) {
    if (root[n] != n) {
        root[n] = findroot(root[n]); //路径压缩操作
    }
    return root[n];
}

T kru() {
    for (int i = 0; i < maxn; ++i) {
        root[i] = i;
    } //初始化并查集

    sort(Es.begin(), Es.end(), [=](edge<T> a, edge<T> b) { return a.cost <
b.cost; }); //将边按照权值排序

    int ans = 0; //最小生成树总权重
    int k = 0; //已经加入到最小生成树中的边的数量

    for (auto c : Es) {
        int l = findroot(c.start);
        int r = findroot(c.end); //取两个端点的根节点
        if (l == r) continue; //已经联通，直接跳过
        ans += c.cost; //将边加入到最小生成树中

        cout << c.start << ", " << c.end << ", " << c.cost << " ";
        root[l] = r; //修改端点父节点

        ++k;

        if (k == v.size() - 1) return ans; //最小生成树已完成构建
    }
}
```

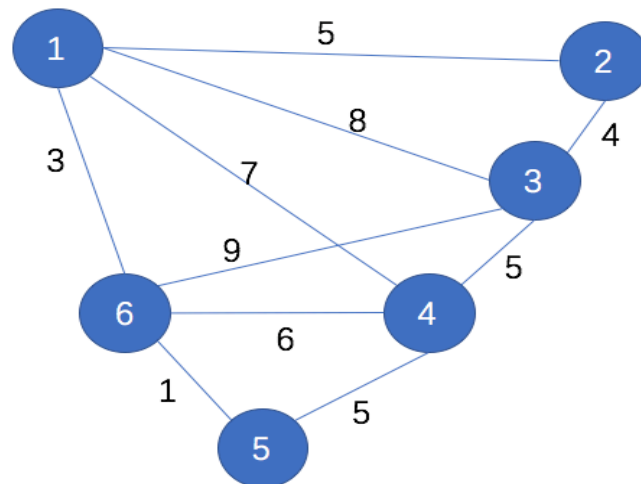
```

    if (k != v.size() - 1) return -1; //不是连通图
}

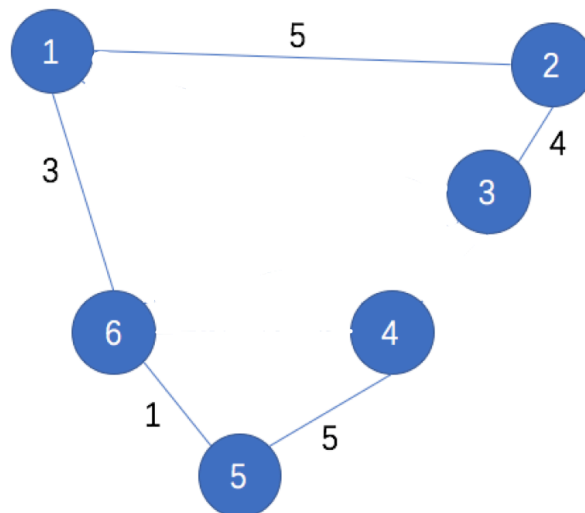
```

5.2.5 运行结果

测试数据：



其最小生成树为：



运行结果为：

格式为： 起点， 终点， 权重， 最后为最小生成树的总权重。

6,5,1 1,6,3 2,3,4 2,1,5 5,4,5 18

5.2.6 总结与展望

通过这个实验，对于最小生成树有了比较深刻的了解，对于无向图而言，首先要保障无向图是联通图，在联通图的基础上才能进行最小生成树的构建。Kruskal 算法在边较为稀疏的图上能发挥更大的作用。在处理图的问题时，有的问题并不需要完整的存储整个图，比如在这个问题中，只需要将图中所有的边存储即可。在编程处理问题的时候，不能盲目的将所有的数据都进行存储，应当只选择需要的数据，避免不必要的工作。在这个方法中也使用到了并查集，可见不同的算法与数据结构之间可能也会有交错，包含等关系，不能只是着重于一个数据结构的应用，同时也应该考虑多个数据结构的搭配使用，不论如何，一切都应向最优的方式进行。

第六章：查找表实验

6.1 实验完成情况

题目一：顺序查找

题目要求	题目完成情况	对应文件函数名称
输出顺序查找序列	完成	search/1.cpp:search()

题目二：折半查找

题目要求	题目完成情况	对应文件函数名称
输出折半查找序列	完成	search/2.cpp:binsearch()

题目三：哈希表操作

题目要求	题目完成情况	对应文件函数名称
------	--------	----------

线性探测法建立哈希表	完成	search/3.cpp:init()
查找记录	完成	search/3.cpp:search()
删除记录	完成	search/3.cpp:del()
插入记录	完成	search/3.cpp:insert()

题目四：二叉排序树操作

题目要求	题目完成情况	对应文件函数名称
用括号表示法输出	未完成	
判断是否为二叉排序树	完成	search/4.cpp:check()
输出查找路径	完成	search/4.cpp:search()
删除节点	未完成	

6.2 哈希表相关操作

6.2.1 问题再现

哈希表的各项操作

6.2.2 需求分析

在使用线性探测法构造的哈希表上，能够完成查找记录，删除已有记录，插入新纪录。

6.2.3 功能分析

哈希表的空间复杂度分析：

由于哈希表需要对所有的元素都进行哈希操作，所以需要为每一个元素都申请额外的辅助空间，表的空间复杂度为：

$$\theta(n)$$

其中 n 为元素的个数。

线性探测法构造哈希表的探测次数分析：

线性探测法在哈希函数的到的地址已经被填充时，顺序后移，如果超过表预设纪录数，则从表头开始继续后移。

最优探测次数：

$$\Theta(n)$$

最差情况探测次数：

$$\Theta\left(\sum_{k=1}^{k=n} k\right) = \Theta\left(\frac{n^2 + n}{2}\right) = \Theta(n^2)$$

引入装填因子

$$\alpha = \frac{\text{已经含有的纪录数}}{\text{表预设的纪录数}}$$

线性探测法在查找的时，会从哈希得到的地址开始逐次后移，如果遇到表中对应地址没有被标记或者找到了元素，那么就停止查找，返回对应地址。

查找成功时的平均查找长度：

$$\frac{1}{2}\left(1 + \frac{1}{1 - \alpha}\right)$$

查找失败时的平均查找长度为：

$$\frac{1}{2}\left(1 + \frac{1}{(1 - \alpha)^2}\right)$$

删除，插入的平均探测次数均与以上两个平均查找长度有关。

在以上几种操作中，需要注意的是删除操作，在删除操作中，不能直接将表中的元素直接删除，这样会影响以后元素的查找，所以应该是为其做好标注，这样的话，就需要额外的空间来记录标记，这里的空间复杂度为：

$$\Theta(n)$$

那么最后，整个哈希表的空间复杂度为：

$$\Theta(n + n) = \Theta(n)$$

6.2.4 详细设计

```
int search(int t) { //查找操作
    int key = t % mod; //首先通过哈希函数获取起始地址
    int ok = 0;
    while (list[key] != t && list[key] != -1) { //如果表中相应元素没有被插入
```


过并且也不是所需要查找的元素，那么就应该继续查找。

```
        cout << key << " "; //输出查找路径
        ++key; //移动地址
        ++ok;
        key %= mod; //保证地址不会越界
        if (ok == len) {
            cout << "Search falied" << endl;
            return -1;
        }
    }
    return key;
}

void del() { //删除操作
    int t;
    cout << "Input the delete element:" << endl;
    cin >> t;
    mark[search(t)] = 1; //将表中对应元素做好标记，不能直接删除
}
```

6.2.5 运行结果

```
Input the mod:
13
Input the length of the data:
11
Input the data:
16 74 60 43 54 90 46 31 29 88 77
-1 -1 -1 16 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 16 -1 -1 -1 -1 -1 74 -1 -1 -1
-1 -1 -1 16 -1 -1 -1 -1 60 74 -1 -1 -1
-1 -1 -1 16 43 -1 -1 -1 60 74 -1 -1 -1
-1 -1 54 16 43 -1 -1 -1 60 74 -1 -1 -1
-1 -1 54 16 43 -1 -1 -1 60 74 -1 -1 90
-1 -1 54 16 43 -1 -1 46 60 74 -1 -1 90
-1 -1 54 16 43 31 -1 46 60 74 -1 -1 90
-1 -1 54 16 43 31 29 46 60 74 -1 -1 90
-1 -1 54 16 43 31 29 46 60 74 88 -1 90
77 -1 54 16 43 31 29 46 60 74 88 -1 90
Input the element to seach:

77
12 0
Input the delete element:
90
12
77 -1 54 16 43 31 29 46 60 74 88 -1 90
Input the element to seach:
77
12 0
Input the element to seach:
90
The element has alread been deleted.
Input the element to add:
90
77 -1 54 16 43 31 29 46 60 74 88 -1 90
Input the element to seach:
90
12
```

6.2.6 总结与展望

在哈希表的实验中，对于不同的散列函数，哈希表的各项操作的平均查找次数也会有所不同。这个时候应该从数据出发，选取合适的散列函数，不能一概而论。除此以外，在哈希表的各项操作中，尤为需要注意删除操作，删除操作不能影响到后续的查询等操作，在编写

代码的过程中也是相同，对于每一个操作，都需要考虑其对以后操作的影响，尽量只将变化局限到该操作中，减少外界的变动，增强代码的健壮性。

第七章：内排序实验

7.1 实验完成情况

排序操作

题目要求	题目完成情况	对应文件函数名称
插入排序	完成	sort/1.cpp:insertsort()
希尔排序	完成	sort/1.cpp:shellsort()
快速排序	完成	sort/1.cpp:quicksort()
堆排序	完成	sort/1.cpp:heapsort()

7.2 堆排序

7.2.1 问题再现

堆排序是指利用堆积树这种数据结构所设计的一种排序算法，它是选择排序的一种。可以利用数组的特点快速定位指定索引的元素。^[3]堆分为大根堆和小根堆，是完全二叉树。

7.2.2 需求分析

完成排序操作。

7.2.3 功能分析

首先对于元素序列，要能完成建堆操作，以小顶堆为例，在建堆的时候，从 $\frac{n}{2}$ 处开始，与它左右孩子中的最小元素交换，然后不断向上调整。其建堆的时间复杂度推导如下：

假定最下层节点的高度为 1，那么这一层的节点数最多为 $\frac{n}{2}$ 个，第二层的节点数最多为

$\frac{n}{2^2}$ 个, ……，而每一层调整的次数不会超过该层的高度，那么最后的时间复杂度为：

$$\Theta\left(\sum_{m=1}^{\log n} \frac{hn}{2^m}\right) = \Theta(n)$$

而在排序的过程中，由于堆的性质，堆顶一定是最小的元素，所以在输出堆顶元素之后，将其与序列尾部的元素互换然后在进行从上往下的调整，直到恢复堆的性质。那这里的时间复杂度为：

$$\Theta(n * \log n) = \Theta(n \log n)$$

而在此过程中，由于不需要其余的辅助空间，那么空间复杂度为：

$$\Theta(1)$$

7.2.4 详细设计

```
void shiftDown(int now) { //自顶向下调整
    int lchild=now *2+1, rchild=lchild+1; //左孩子和右孩子的下标
    while(rchild<llen) { //如果左右孩子均存在
        if(data[now]<=data[lchild]&&data[now]<=data[rchild]) {
            return;
        } //判断是否已经结束调整
        if(data[lchild]<=data[rchild]) {
            int tem = data[now];
            data[now] = data[lchild];
            data[lchild] = tem;
            now = lchild; //如果最小的元素是左孩子，与其互换
        } else {
            int tem= data[now];
            data[now] = data[rchild];
            data[rchild] = tem;
            now =rchild; //否则与右孩子互换
        }
        lchild=now *2+1;
        rchild=lchild+1; //更新左右孩子下标
    }
    if(lchild<llen&&data[lchild]<data[now]) {
```

```

        int tem = data[lchild];
        data[lchild] = data[now];
        data[now] = tem;
    } // 如果只有左孩子并且需要调整，与左孩子互换，互换结束后显然已经到达最底
    层

    return;
}

```

```

void heapsort() {
    getback();
    int i;
    for (i = llen - 1; i >= 0; i--) {
        shiftDown(i);
    } // 建堆操作
    cout << "The heap:" << endl;
    out();
    cout << endl;
    while (llen > 0)
    {
        cout << data[0] << " ";
        data[0] = data[llen - 1]; // 将最后一个元素放到堆顶
        --llen;
        shiftDown(0); // 重新调整
    }
    return;
}

```

7.2.5 运行结果

```
Input the length of the data:
9
Input the data:
1 2 3 9 5 4 6 7 8
the ans of the heapsort:
The heap:
1 2 3 7 5 4 6 9 8
1 2 3 4 5 6 7 8 9
```

7.2.6 总结与展望

排序问题是一个老生常谈的话题，排序操作由于需要经常的使用，由此也演化出来很多不同的排序方式。对于不同的排序操作，不仅仅需要明白每一个排序操作的大体思路，而且也应该理解每一趟排序后的性质，根据不同的性质，再结合数据本身的特点，再选择最终使用的排序算法，这样才能实现最优，比如对于有序序列，如果使用快速排序则会遇到最差的情况。因此，算法应当结合实际使用，不能脱离实际情况泛泛而谈。

第八章：课程总结与展望

通过一个学期的数据结构的学习，首先在理论上，我学习到了很多经典的数据结构思想，对于不同的数据，选择合适的数据结构是程序高效的一个关键因素。同时，理论也不能脱离实践，二者应当相辅相成相互支持，过于注重理论却无法实现到代码上也是自己能力的欠缺，但是实践却需要理论的支持。

其次，在编程时应注意细节，从第一章的链表开始就体现出细节的重要性，对于每一个链表的节点，最基本的要注意节点是否为空以及后继是否初始化为 `nullptr` 等，如果有所差错，在程序运行时就会经常段错误，异常退出，而在后续的一些算法中，细节更加体现在复杂度的优化上，比如在最小生成树中，树只有 $n-1$ 条边，那么如果在树的边集已经足够 $n-1$ 条边，直接返回即可，这是时间复杂度上的小小优化。

再次，在编程的时候往往需要结合实际情况来选择合适的数据结构以及算法，比如在哈希表的操作中，选择不同的散列函数以及探测方式往往会有不同的复杂度，因此在编程的时候，不能脱离实际。尤其是数据量的范围，这往往会决定数据的存储方式。

最后，在编程的过程中，同伴的帮助也是不可缺少的，对于一份代码，不同的人可能会有不同的着重点，对于一个算法或者数据结构而言，不同的人可能也会有不同的关注点，在遇到困难的时候，需要及时向他人寻求帮助，力求提高效率。

《数据结构》这门课程作为计算机专业的专业课程，是其他很多课程的基本课程，它更加侧重于数据的存储方式以及在数据结构上的基本操作，在数据结构的支持下，再搭配以合适的算法，才能保证程序正确、高效的运行。数据结构的内容不仅仅只局限于课本上所列的内容，更多更优秀的数据结构需要我们去发掘。

第九章：参考文献

[1] 百度百科 约瑟夫环问题[Z]

[2] 严蔚敏.《数据结构与算法分析》[M]: 清华大学出版社, 2011

[3] 百度百科 堆排序[Z]