期中大作业

姓名：闫浩扬 学号：2253156 日期：2023 年 11 月 9 日

一、 基本概念

(一) 对于数据结构，逻辑结构，物理结构，算法等概念的理解与认识

1. 数据结构

数据结构是指在计算机上处理某种存储，组织数据的方法，它是相互之间存在一种或多种特定关系的数据元素的集合。

数据结构包括三部分：逻辑结构，物理结构和数据的运算。

2. 逻辑结构

简单来说，逻辑结构就是数据元素之间的逻辑关系。

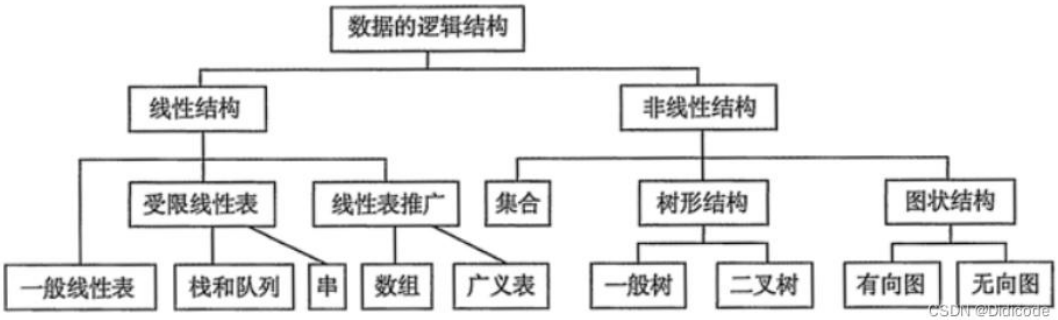
逻辑结构的分类：可以分为线性结构和非线性结构。再具体分类，可以将非线性结构再分为集合，树形和图状。

① 集合：结构中的数据元素之间除了同属于一个集合外，没有其他的关系。

② 线性结构：线性结构中的数据元素之间是一对一的关系。

③ 树形结构：树形结构中的数据元素之间是一对多的关系。

④ 图状结构或网状结构：结构中的元素之间是多对多的关系。



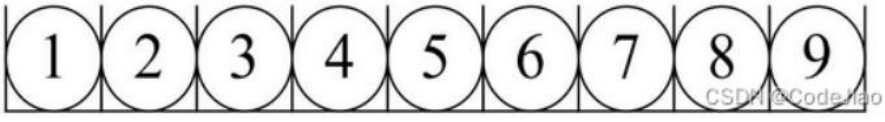
3. 物理结构

数据结构在计算机中的表示（又称映像）成为数据的物理结构，或称存储结构。物理结构研究的是数据结构在计算机中的实现方法，包括数据结构中元素的表示以及元素间关系的表示。

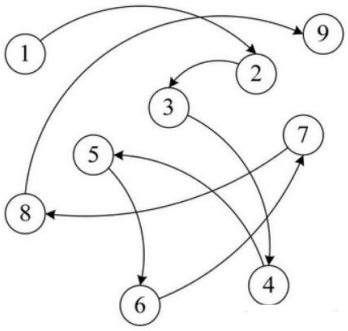
物理结构一般有四种：顺序存储，链式存储，散列，索引。

① 顺序存储结构：利用数据元素在存储器中的相对位置来表示数据元素之间的逻辑顺序。

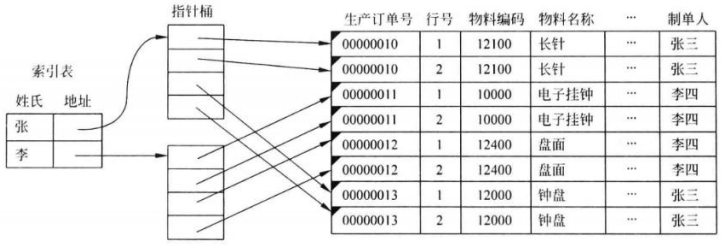
顺序存储结构是把数据元素放在地址连续的存储单元中，程序设计中使用数组类型来实现。（逻辑相邻物理相邻）



② 链式存储结构：利用结点中指针来表示数据元素之间的关系。把数据元素存储在任意的存储单元里，这组存储单元可以是连续的，也可以是连续的，程序设计中使用指针类型来实现。（逻辑相邻物理不一定相邻）



③ 索引存储：类似于目录，以后可以联系操作系统的文件系统章节来理解。



④ 散列存储：通过关键字直接计算出元素的物理地址。

4. 算法

算法：解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，每条指令表示一个或多个操作。

基本特性：

- 输入输出：可以有 0 到多个输入，但至少会有一个输出。
- 有穷性：执行了有限步骤后会自动结束而不是无限循环，每个步骤的时间合理。
- 确定性：每一个步骤对具有确定的含义，不能出现二义性。
- 可行性：每一步都能够通过执行有限次数完成。

算法的设计应该遵循正确性，可读性，健壮性，高效率低存储。

算法效率的度量：常用时间复杂度来表示时间量度，用空间复杂度表示空间消耗。$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

(二) 对于数据结构，逻辑结构，物理结构，算法等概念之间关系的理解与认识

1. 算法同逻辑结构与物理结构

一个算法的设计取决于所选定的逻辑结构，而算法的实现依赖于所采用的物理结构。

2. 算法与数据结构

数据结构是算法的基础，算法是数据结构的应用。不同的数据结构适合不同的算法，不同的算法需要不同的数据结构。因此，在学习和使用数据结构和算法时，要根据具体的问题和场景，选择合适的数据结构和算法，以达到最优的效果。

3. 数据结构与逻辑结构

数据结构是对数据元素及其之间关系的组织方式，包括逻辑关系和存储关系，而逻辑结构描述了这些数据元素之间的逻辑关系，如线性结构或非线性结构。数据结构的选择直接影响程序的运行效率和对数据的操作便捷性，而逻辑结构通过定义数据元素关系为数据结构的设计提供了指导。在程序设计中，我们需要综合考虑逻辑关系和数据组织的需求，选择合适的数据结构。

4. 数据结构与物理结构

数据结构的设计会影响数据在内存中的存储形式，选择不同的数据结构会产生不同的物理结构。

5. 逻辑结构与物理结构

逻辑结构从逻辑关系上描述数据，它与数据的存储无关，是独立于计算机存储器的。

同一种逻辑结构可以用不同的物理结构来实现，不同的逻辑结构也可以用相同的物理结构来实现。例如，线性表可以用顺序存储结构或链式存储结构来实现，树和图也可以用顺序存储结构或链式存储结构来实现。逻辑结构是数据结构的抽象层面，物理结构是数据结构的具体层面。两者相互依赖，相互影响。

二、 线性表的应用

(一) 线性表的基础知识

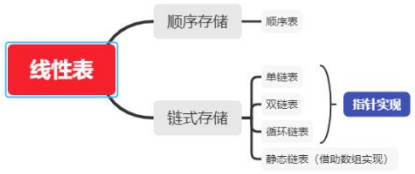
线性表是具有相同数据类型的 n（n>=0）个数据元素的有限序列，其中 n 为表长，当 n=0 时，线性表是一个空表。若用 L 命名线性表，则其一般表示为 $L=(a_1, a_2, a_3, \cdots\cdots, a_i, a_{i+1}, \cdots\cdots, a_n)$

线性表是一种逻辑结构，表示元素之间一对一的相邻关系。顺序表和链表是指存储结构，两者属于不同层面的概念。

线性表的顺序存储又称为书序表，使用一组连续的存储单元一次存储线性表中的数据元素，最大的特点是随机访问，可在 O(1)的时间内找到指定元素，并且存储密度高，因为每个结点只存储数据元素，但是由于逻辑上相邻的元素物理上也相邻，所以插入和删除操作需要移动大量元素。

顺序表也可以采用链式存储结构，不需要使用地址连续的存储单元，不要求逻辑上相邻的元素在物理位置上也相邻，可以通过指针将元素之间连接起来，因此插入删除操作不需要移动元素，但是失去了随机存取的优点。另一优点就是可以动态内存申请，方便对动态问题的求解。

在选择哪种存储结构时，我们需要根据题目的要求，考虑哪种空间消耗小，操作效率高，更可以满足实际需求。



(二) 题目描述

实现一个图书信息管理系统，包括图书的添加、删除、修改、排序、查询和显示等基本操作。每本书的信息包括书名、作者、ISBN 号等。

(三) 思考与分析

图书详情

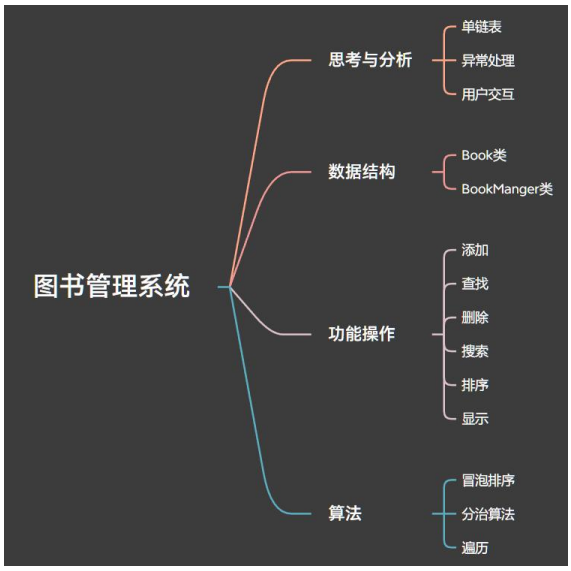
书名	作者	出版社	ISBN	价格	剩余数量	详情	编辑	删除
大雪中的山庄	东野圭吾	北京十月文艺出版社	9787530216835	35.00	0	详情	编辑	删除
三生三世 十里桃花	唐七公子	沈阳出版社	9787544138000	26.80	4	详情	编辑	删除
11处特工皇妃	潇湘冬儿	江苏文艺出版社	9787539943893	74.80	3	详情	编辑	删除
经济学原理（上下）	[美] 曼昆	机械工业出版社	9787111126768	88.00	3	详情	编辑	删除
画的秘密	马克-安托万·马修	北京联合出版公司·后浪出版公司	9787550265608	60.00	7	详情	编辑	删除
造彩虹的人	东野圭吾	北京十月文艺出版社	9787530216859	39.50	5	详情	编辑	删除
控方证人	阿加莎·克里斯蒂	新星出版社	9787513325745	35.00	4	详情	编辑	删除
方向	马克-安托万·马修	后浪丨北京联合出版公司	9787020125265	99.80	1	详情	编辑	删除
追寻生命的意义	[奥] 维克多·弗兰克	新华出版社	9787501162734	12.00	5	详情	编辑	删除
秘密花园	乔汉娜·贝斯福	北京联合出版公司	9787550252585	42.00	5	详情	编辑	删除

上述图示为图书管理系统的实现图，我们可以观察到每本书都有一个独一无二的 ISBN 编号，同时具备书名和作者两个基本属性。在系统管理中，以单本书为一个基本单位，因此需要将这本书的各项属性集合在一起。所以引入了类的概念，将每本书抽象为一个类，便于对这个类进行多种操作。

考虑到实际情况中图书数量庞大且涉及的操作灵活多变。由于顺序表是静态的，且元素在内存中顺序

存储，虽然可以采用倍增的方式扩充内存，但是不便于频繁的增删改操作，所以顺序表并不最优。因此，我们考虑使用链式存储结构实现图书管理系统。链式存储结构的优势在于快捷方便地移动元素，同时动态内存申请也方便了内存的管理，可以使系统更具灵活性和效率。这样的设计能够更好地适应图书管理系统的动态特性和复杂操作需求。

下图为设计的图书管理系统框架:



(四) 存储结构及理由

通过上述分析，我们初步确定了采用链式结构来实现图书管理系统中的线性表。然而，链表的选择还有多种方式，包括单链表、双链表和循环链表等。在确定使用链式结构的同时，我们需要进一步考虑选择哪种链表结构来最好地满足系统需求。

- 单链表：单链表是最简单的链表结构，每个节点包含数据和一个指向下一个节点的指针。适用于图书管理系统的简单增删查操作，对内存的利用较为高效。然而，单链表无法快速反向遍历，对于一些需要频繁的逆向操作可能不够高效。
- 双链表：双链表在单链表的基础上增加了一个指向前一个节点的指针，提高了反向遍历的效率。适用于需要频繁双向遍历的情况，但相应地增加了内存开销。
- 循环链表：循环链表是一种特殊的链表，尾节点指向头节点，形成一个闭环。适用于需要循环访问的场景，但在图书管理系统中可能并不是首选，因为图书信息的自然顺序并不需要形成闭环。

因此在综合考虑后，采用单链表的数据结构，不仅可以满足增删查改的需要，还可以高效地利用内存。但是由于链表只能顺序读取的特点，如果图书量庞大，在进行查找时，每次都要遍历链表，时间效率可能较低，为此，可以采用索引的方式，或者对书籍进行分类来缩小查找规模来实现。

以下为 Book 类和 BookManger 类的实现:

```
class Book {
public:
    string title;  //书名
    string author; //作者
    string isbn;   //ISBN 编号
    Book* next;    //指向下一节点
    Book(string title, string author, string isbn)
        : title(title), author(author), isbn(isbn), next(NULL) {}
};
```

Book 类用于存储每本书的相关信息，以及使用 Book* next 来存储下个节点的地址。

```
class BookManager {
private:
    Book* head;   //头节点
public:
    BookManager() : head(NULL) {}
    void addBook(string title, string author, string isbn);  // 添加图书
    void deleteBook(string isbn);   // 删除图书
      //修改图书信息
    void modifyBook(string isbn, string newTitle, string newAuthor);
    void sortBooks();               // 排序图书
    Book* searchBook(string isbn);  // 查询图书
    void displayBooks();            // 显示所有图书
    ~BookManager();                 // 析构函数
```

BookManger 类实现了图书单链表，以及一些相关的链表操作。

```
};
```

(五) 基本操作与实现

基于 Book 类，我们可以将每一个书看做一个结点，然后构建带表头单链表，对单链表的构建，删除，排序，查询，修改就需要 BookManger 内的方法实现。通过这些操作，就实现了对图书的添加，删除，修改，查询，排序，显示等操作。以下为具体实现与分析。

```
void BookManager::addBook(string title, string author, string isbn) {
    // 检查是否存在相同 ISBN 的图书
    if (searchBook(isbn) != NULL)
        错误输出并返回;
    // 创建新图书节点
     Book* newBook = new Book(title, author, isbn); //动态内存申请
     if (head == NULL) //链表为空
            head = newBook; // 新节点为头结点
    else {
        Book* temp = head;
        while (temp->next != NULL) //遍历到链表尾
            temp = temp->next;
        temp->next = newBook;         // 不为空，在链表末尾添加新节点
    }
```

addBook 方法实际为单链表添加节点操作，通过动态内存申请，创建一个新结点，为其存入新值并链接到链表尾部。同时避免重复添加，需要先检查一下 ISBN 号是否已存在。由于需要遍历单链表，所以时间复杂度为 O(n).

```
}
void BookManager::deleteBook(string isbn) {
    Book* temp = head; Book* prev = NULL;  //创建两个临时节点，用于存放当前指针和上次指针
    // 如果要删除的图书是头结点
```

```
    if (temp != NULL && isbn 编号相同) {
        head = temp->next;  //头指针指向下一结点
        delete temp;        //释放原先头结点
        return;
    }
    // 遍历链表查找要删除的图书
    while (temp != NULL && isbn 编号不同) {
        prev = temp;          //循环遍历，直至找到相同 isbn 编号的书籍
        temp = temp->next;
    }
    if (temp != NULL) {    // 如果找到要删除的图书，前驱跳过本节点指向后驱，并释放结点
        prev->next = temp->next;
        delete temp;
        输出提示
    else
        cout << "未找到,删除失败." << endl;
}
```

deleteBook 方法实际为单链表删除节点操作，所以需要考虑下删除的是否为头结点，然后遍历链表查找要删除的图书，若找到释放该结点并对链表指针进行修改。由于单链表通过指针相连接，所以我们需要注意对于空链表，首尾结点的判断，避免越界访问等。时间复杂度为 O(n).

```
void BookManager::modifyBook(string isbn, string newTitle, string newAuthor) {
    Book* book = searchBook(isbn);   //调用寻找函数，根据 isbn 号找到本书
    if (如果找到)
        修改名称，作者，isbn 等相关信息
    else
        cout << "未找到." << endl;
}
```

modifyBook 方法实现了结点信息修改的功能，首先调用 SearchBook 函数，查找到所要修改的节点，随后进行信息的修改，时间复杂度同 Search 函数相同，应该都为 O(n).

下面考虑图书的排序，按照 isbn 号对图书进行排序，对于一般的数据结构，我们常用冒牌排序（实现如下），快速排序，但是这种频繁交换数据的方法在链表中效率较低。因此转而思考线性表的特点，尤其是在这里链式存储结构的特点，每个结点靠指针链接起来，如果我们排序的话，自然有两种思路，一种是不换位置换元素，另一种就是不换元素换位置。显然，交换结点位置是空间开销小，效率高的方法。

再搜索学习后发现，对于链式存储结构，分治算法效率较高，算法思想简单来说就是分而治之，

步骤如下：①找出中间节点，用单步和双步两个指针可以快速找出，并拆分为左边链表，右边链表；

②递归调用继续拆分左边链表和右边链表，直到只有 null 或者只有一个节点为止；

③逐个合并节点：定义结果空链表，对比合并的两个链表，小的值逐个拼接到结果链表中。

伪代码如下

```
MergeSortList(head):
```

```
    // 如果链表为空或只有一个节点，已经有序
    if head == null or head.next == null:
        return head
    // 分解链表为两半
    middle = FindMiddle(head)
    left = head
    right = middle.next
    middle.next = null
    // 递归地对两个子链表进行排序
    left = MergeSortList(left)
    right = MergeSortList(right)
    // 合并两个有序子链表
    sortedList = MergeLists(left, right)
    return sortedList
FindMiddle(head):    // 使用快慢指针法找到链表中点
    slow = head
    fast = head.next
    while fast != null and fast.next != null:
        slow = slow.next
        fast = fast.next.next
    return slow
MergeLists(left, right):
    // 创建哑结点作为合并后链表的头结点
    dummy = new Node(0)
    current = dummy
    // 合并两个有序链表
    while left != null and right != null:
        if left.data <= right.data:
            current.next = left
            left = left.next
        else:
            current.next = right
            right = right.next
        current = current.next
    // 处理剩余的节点
    if left != null:
        current.next = left
    else:
        current.next = right
    return dummy.next
```

分治算法排序是一种归并排序，总体时间复杂度为 O(nlogn)。分为分解，解决，合并三步，分解需要将链表切半，时间复杂度为 O(n)，解决调用递归对两个子链表进行排序，在每一层递归中，链表长度减半，直到长度为 1。所以递归树的高度为 Logn，每一层的合并操作都需要 O(n)的时间，所以总的时间复杂度为 O(nlogn).

归并排序在链表上的性能较好，因为合并两个有序链表的操作简单，仅需要对指针进行修改，与数组不同，链表上的归并排序不需要额外的空间复杂度。

冒泡排序代码如下:

```
void BookManager::sortBooks() {
    if (空链表 or 到尾节点)
        return;
    // 使用冒泡排序按 ISBN 号排序图书 冒泡排序时间复杂度为 O(n*n)
    bool swapped;
    Book* temp;
    Book* prev = NULL;
    do {
        swapped = false;
        temp = head;
        while (temp->next != prev) {
            if (temp->isbn > temp->next->isbn) {
                交换信息
                swapped = true;
            }
            temp = temp->next;
        }
        prev = temp;
    } while (swapped);
```

冒泡排序是一种基础的比较排序算法，通过多次遍历待排序序列，两两比较，这个过程重复 n 次。时间复杂度为 O(n*n)，但是在本例中，由于链表需要交换信息，效率很低，但是可以不交换信息，交换结点指针稍加修改。

```
}
Book* BookManager::searchBook(string isbn) {  //遍历链表搜索图书
    Book* temp = head;
    while (temp != NULL) {
        if (temp->isbn == isbn)
            return temp;
        temp = temp->next;
    }
    return NULL;
```

SearchBook 方法采用简单的遍历方法，寻找所求结点，时间复杂度为 O(n)

```
}
```

(六) 总结

通过上述数据结构的构造，我们定义了 Book 类和 BookManger 类，并在 BookManger 类中实现了多种对单链表的操作，实现了图书管理系统的基本操作，但是这仅是数据结构的构造，具体实现还需要补充许多。

但是，在图书管理系统设计的过程中，我又巩固了线性表的基本概念和基本操作，尤其是单链表的操作，并且，更深刻地理解了算法和数据结构选择的重要性，需要根据具体问题的要求和效率选择更好的数据结构和算法，正如上述所言，一个好的程序由好的数据结构和好的算法构成，例如在数据结构的选择上，链表就更适用于图书管理系统这种动态信息管理问题，在排序算法的选择上，归并排序就更适用于链表操作。

三、 栈的应用

(一) 栈的基础知识

栈(Stack)是一种线性存储结构，具有先进后出(LIFO)的特点，限定只能在栈顶进行插入和删除操作。

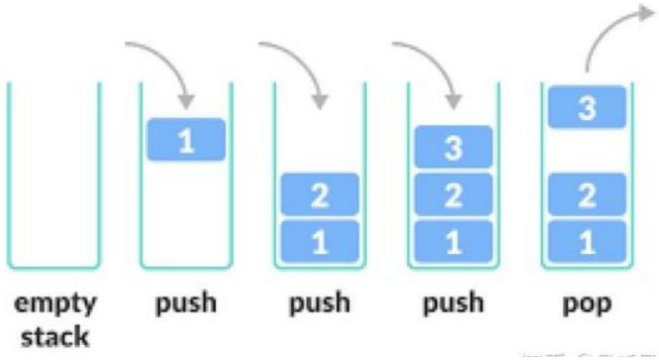
换言之，栈同上述线性表的区别就是，线性表是一种逻辑结构，二数组，链表都是物理结构，是线性表的一种实现。栈相当于一个筒，只能由上而下进行操作，所以按照线性表存储结构的实现方式，栈也可以分为基于数组的栈和基于链表的栈。

1) 基于数组的栈：以数组为底层结构时，通常以数组头作为栈底，数组头到数组尾为栈顶的生长方向
2) 基于单链表的栈：以链表为底层的数据结构时，以链表头为栈顶，便于节点的插入与删除，压栈产生的新节点将一直出现在链表的头部

栈的基本操作：创建栈，销毁栈，压入，弹出，输出，以及取最小元素等。

栈是一种用于解决事情发生特定顺序的数据结构，解决问题前可先判断是否符合先进后出的规律。典型例子有数制转换，树的非递归遍历，括号匹配，表达式求值以及迷宫问题汉诺塔问题。

在这里，我以迷宫问题为例，并实现栈的一些基本操作。



(二) 题目描述

给定一个迷宫，指明起点和终点，找出从起点出发到终点的最短有效可行路径，就是迷宫问题（maze problem）。迷宫可以以二维数组来存储表示。0 表示通路，1 表示障碍。注意这里规定移动可以从上、下、左、右四方方向移动。

(三) 思考与分析

求迷宫从入口到出口的所有路径是一个经典的程序设计问题，由于计算机解迷宫时，通常用的是“穷举求解”的方法，即从入口出发，顺某一方向向前探索，若能走通，则继续走，否则沿原路返回，换一个方向探索，直到所有路径都探索完。为了保证可以在任何位置都能沿原路返回，显然需要一个后进先出的结

构来保存从入口到当前位置的路径，因此在迷宫求解时应用"栈"更为合适。

在本题中，使用二维数组表示地图，0 表示通路，1 表示障碍，地图四周全为 1 表示围墙防止异常。在规范化后，再进行算法的设计，利用栈的后进先出的特点，每次进入一个可通过的点，就把这个点存入到栈当中，直到五通路，返回上一个点，即进行出栈操作，然后走另一个方向，直到走到终点。

对于算法的选择，采用深度优先算法（DFS）寻找最短通路，DFS 的求解思路分两部分：①先判断是否到大目标位置，如果到达目标位置，再试探有无其他更短的路径 ②如果没有到达目标位置，则找到下一步可以到达的位置，直到找到目标位置。

(四) 存储结构及理由

1	1	1	1	1	1	1	1	1	1
1	0	0	1	0	0	0	1	0	1
1	0	0	1	0	0	0	1	0	1
1	0	0	0	0	1	1	0	0	1
1	0	1	1	1	0	0	0	0	1
1	0	0	0	1	0	0	0	0	1
1	0	1	0	0	0	1	0	0	1
1	0	1	1	1	0	1	1	0	1
1	1	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1

对于地图的存储，采用二维数组 maze[M+2][N+2]来表示，图中红框部分为迷宫，而迷宫四周值设为 1，这样就可以保证迷宫中各店的试探方向都为 4 个，可以根据四个方向的增量来修改坐标。

方向的存储结构如下:

```
//试探方向存储结构
typedef struct {
    int xx, yy;
}Direction;
Direction dire[4] = { {0,1},{1,0},{0,-1},{-1,0} }; //上下左右四个方向
```

对于路径的存储，采用栈的形式，因为栈后进先出特性非常适合深度优先搜索，它允许在探索路径时简单而自然地回溯到之前的状态。又知栈有两种形式，数组栈或链表栈，随着地图扩大，路径长度也可能很大，而链表允许动态分配和释放元素，对于处理未知路径长度的问题(如 DFS)很有用，同时链表中插入删除元素相对高效，也符合 DFS 需要频繁修改的要求，所以优先采用链表栈，方便内存管理。

数据类型定义如下:

```
typedef struct datatype {// 数据类型
    int x, y, di; //横纵坐标及方向
} ElemType;
```

栈中节点的存储结构如下:

```
typedef struct node {// 节点
    ElemType data;      //节点信息
    struct node* next;  //指向栈中下一节点
} Node;
```

栈的存储结构定义如下:

```
class Stack { // 栈类
private:
    int count; //步数统计
    Node* top; //栈顶
    Node* createNode(ElemType data);    // 私有辅助函数，用于创建节点
public:
```

```
    Stack():count(0),top(NULL){};  // 构造函数
    ~Stack(){while(!isEmpty){pop();}; // 析构函数
    bool isEmpty();    // 检查栈是否为空
    void push(ElemType data); // 将元素推入栈
    Node* pop();   // 从栈中弹出元素
    void showPath(Node* node); // 递归显示路径
};
```

(五) 基本操作与实现

在确定数据结构后，接下来将完善各个数据结构的具体操作，然后进行应用。

以下为 Stack 类的方法实现:包括入栈，出栈，构造，销毁，输出等操作

```
Node* Stack::createNode(ElemType data) {// 私有辅助函数，用于创建节点
      Node* newNode = new Node; //为新节点申请内存
      if(!newNode)  {申请内存失败，退出}
      newNode->data = data;
      newNode->next = NULL;
      return newNode;
}
```

入栈操作

```
void Stack::push(ElemType data) {// 将元素推入栈
    Node* newNode = createNode(data);
    count++;
    if (count == 1)  //是首节点
        top = newNode;
    else {
        newNode->next = top; //作为新栈顶，并更新栈顶指针
        top = newNode;
    }
}
```

出栈操作

```
// 从栈中弹出元素
Node* Stack::pop() {
    Node* popNode = NULL;
    if (栈非空) {
        popNode = top; // 弹出栈顶节点
        top = top->next;
        count--;
    }
    return popNode;
}
```

递归显示路径(栈)

```
void Stack::showPath(Node* node) { // 递归显示路径
    if (node) {
        showPath(node->next);
```

```
        std::cout << "(" << node->data.x << "," << node->data.y << ")" << std::endl;
    }
}
```

判空

```
// 检查栈是否为空
bool Stack::isEmpty() {  return count == 0; }
```

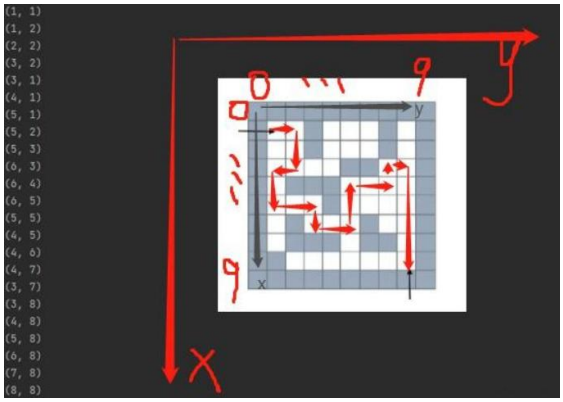
可见栈的操作简单易实现，并且时间复杂度较低，因为没有遍历查找等操作。

下面将实现 DFS 算法，DFS 通过栈完成对迷宫的深度优先搜索。利用先前定义的 Stack 类来辅助实现这一算法。在深度优先搜索中，我们从起点开始，通过不同的路径一直深入到无法继续为止，然后回退到之前的状态，选择另一条路径，直到遍历完整个图或者找到目标位置。

```
//使用 DFS 判断是否是通路，若是加入栈中
bool Findpath(int maze[][6],Stack* stack ,Direction dir[],int startx,int starty,int endx,int
endy) { //startx,starty 是起点的坐标;endx、endy 是终点的坐标.
          //初始化
          assert(stack);int x, y, di;int line, col;
          maze[startx][starty] = -1;
          ElemType start = { startx,starty,-1 };
          push(stack, start);//将起点入栈
          while (栈非空) {
                Node* po = pop(stack);ElemType temp = po->data;//取栈顶节点进行判断
                x = temp.x;y = temp.y;
                di = temp.di++;//换一方向，从 0-4，使用后++再赋值，正好可以满足
                while (di < 4) {
                        line = x + dire[di].xx;col = y + dire[di].yy;//通过偏移量移动坐标
                          if (当前位置为 0) {
                                //储存上一个节点的位置，入栈
                                temp = { x,y,di };
                                push(stack, temp);
                                x = line;y = col;
                                maze[line][col] = -1;
                                if (当前位置是终点) {
                                    //把终点的位置入栈
                                    temp = { x,y,-1 };
                                    push(stack, temp);
                                    return true;
                                }
                                else
                                    di = 0;
                          }
                          else
                              di++;
                    }
          }
          return false;
```

```
}
```

这个函数采用深度优先搜索（DFS）算法，使用栈作为辅助数据结构，函数从起点开始，深度优先地探索迷宫中的路径。每当找到可行的方向时，将当前位置信息入栈，并标记该位置为已访问，以避免重复探索。若新位置为终点，则将终点入栈，表示找到了一条通路。在无法继续探索的情况下，通过栈的回溯操作回退到上一个状态，尝试其他方向。实现结果如下:



这个算法完美地体现了栈先进后出的特点。由于在每个位置都需要考虑多个方向的可能性，因此时间复杂度随着迷宫大小的增加而成平方增长。假设迷宫的大小为 n（n x n），在最坏情况下，我们需要访问所有 n^2 个位置。对于每个位置，可能需要尝试上、下、左和右四个方向。因此，总的时间复杂度可以表示为 O(4n^2)，简化为 O(n^2)。

(六) 总结

在解决迷宫问题时，算法的设计和数据结构的选择是关键的。由于问题的本质是在迷宫中寻找路径，需要不断尝试，可行则继续向前，不可行则回退，这与栈的“后进先出”特点相吻合。因此，我选择了栈作为存储结构，从而在优先搜索中可以方便地进行回溯操作。

考虑到路径的长度可能很大，且需要频繁操作，选择链式存储结构，存储路径的方式。这样就可以动态地管理内存，适应不同长度的路径，并避免固定大小的数组可能带来的限制。

此外，对于算法的选择，选择了深度优先搜索（DFS）。DFS 在其实现中具有回溯的思想，与栈的先进后出特性相吻合，使得在搜索路径的过程中能够自然而直观地进行回溯。虽然 DFS 也可以通过递归实现，但在地图较大时，递归深度可能增加，从而导致程序效率降低。相比之下，使用栈可以更好地兼顾算法的效率和空间消耗。

对于算法的优化，在实践中，可以通过合理的算法设计和剪枝等优化策略，有效提高搜索空间的利用率，从而改善算法的性能。

同时，我也更牢固地掌握了栈的应用，理解了算法和数据结构的关系。此外，我发现，一种算法可以应用在不同的问题中，我们可以把实际问题抽象化后再寻找更合理的算法。

四、 实验总结

本次实验，我分析了数据结构，逻辑结构，物理结构与算法的概念以及它们之间的关系，认识到一个好的程序需要好的数据结构和好的算法，同时一个数据结构可以有多个实现方法，例如，栈可以使用数组实现也可以使用链表实现，具体采用哪种存储结构需要我们结合问题分析。同时，一个好的算法应该综合考虑正确性、效率、可读性、健壮性和灵活性等多个方面，但是评价一个算法的好坏最中要的还是看它对于问题的适用程度，根据具体问题和应用场景的不同，选择适当的算法搭配数据结构是至关重要的。

对于数据结构的具体应用，我选取了线性表和栈，在线性表的应用中，我选择了图书信息管理系统，因为图书信息有统一的特点，并且两两之间有线性的结构，符合线性表的特点，同时，我采用链表的存储结构，因为链表相比顺序表更方便数据的移动和空间的扩充。对于算法，我也比较了冒泡排序和分治排序的区别和效率。在栈的应用中，我选择了迷宫问题，因为路径搜素过程中的回溯和栈的特点相吻合。同时，

也具体实现了栈的基本操作，如入栈出栈等，更好地理解了栈的应用情形。同时，搭配 DFS 算法用于搜索路径，提高了效率，这个应用可以很好地体现数据结构和算法的结合。

对于队列的应用，也有很多，可以将其看作两端开的栈，也有许多应用情形。

通过本次实验，我更深刻地理解了数据结构和算法的关系，以及逻辑结构，物理结构的关系。