西安电子科技大学

数据结构上机实验

题　目： 数据结构上机实验

姓　名： 汤栋文

学　号： 22009200601

专　业： 人工智能

目录

摘要

以下是本学期数据结构课程所有上机实验内容报告。每一部分按照要求补全程序，在报告中包含了每一部分对算法的解释，算法的执行过程。对部分算法探讨了其更简单或更高效的实现方法，包括了我自己的对该算法的想法或改进。本报告中所有的相关代码已在Github公开：https://github.com/MTDoven/ProgramLearning/tree/main/DataStructures/Homework

一、基本信息

姓名：汤栋文
学号：22009200601
时间：2023年12月22日

二、上机作业

（一）线性表

线性表：

线性表是最基本、最简单、也是最常用的一种数据结构，它是n个数据元素的有限序列。线性表中数据元素之间的关系是一对一的关系，即除首元素外，其他元素有且仅有一个直接前驱元素，除尾元素外，其他元素有且仅有一个直接后继元素。换句话说，线性表的数据元素是线性排列的，可以用顺序表或者链表的形式来实现。

实验目的：

1. 熟悉线性表的顺序和链式存储结构
2. 掌握线性表的基本运算
3. 能够利用线性表的基本运算完成线性表应用的运算

1. 顺序表逆置

程序说明：

这个 invert 函数中设置了两个指针，i 从数组的开始位置，j 从数组的结束位置。当 i 小于 j 时（也就是两个指针还没遇到的时候），交换 L->data[i] 和 L->data[j] 的值 （也就是对应数据域中的数据）。然后将 i 向后移动一个，将 j 向前移动一个，继续交换两端的元素，直到 i 和 j 相遇或交叉。

实验结果：

```
45  //添加顺序表逆置算法
46  void invert(sequenlist* L)
47  {
48      int i = 1, j = L->last; // i是起始位置，j是结束位置
49      while(i < j) {
50          // 交换两端的元素
51          datatype temp = L->data[i];
52          L->data[i] = L->data[j];
53          L->data[j] = temp;
54
55          // 移动指针
56          i++;
57          j--;
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
abcdefg*
 a b c d e f g
 g f e d c b a
```

2. 单链表逆置

程序说明：

这个函数的目的是将链表中的元素顺序颠倒。我使用三个指针：prev（前一个节点）、current（当前节点）和 next（下一个节点），通过改变节点间的指向来实现链表的逆置。在逆置过程中，每个节点的 next 指针被重新设置，指向它的前一个节点。最后，将链表的头结点的 next 指针指向新的第一个节点。

实验结果：

```
52  //添加单链表逆置算法
53  void invert(linklist* head) {
54      if (head == NULL || head->next == NULL) {
55          return; // 空链表或只有一个元素的链表不需要逆置
56      }
57
58      linklist* prev = NULL;
59      linklist* current = head->next;
60      linklist* next = NULL;
61
62      while (current != NULL) {
63          next = current->next;  // 储存下一个位置
64          current->next = prev;  // 反转当前指针
65          prev = current;        // 移动指针
66          current = next;        // 移动指针
67      }
68      head->next = prev; // 更新头节点
69  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
abcdefg*
 a b c d e f g
 g f e d c b a
```

算法分析：

这种逆置方法一大好处是不消耗额外的空间，空间复杂度是$O(1)$，时间复杂度是$O(n)$也并不太复杂。顺序表的逆置本来就是一个很简单的算法，有一种情况如果顺序表中每个元素的数据量太大其实这种深拷贝式的移动也是比较低效的，但这是顺序表本身存储结构的问题，与逆置算法无关。

3. 分解单链表

程序说明：

函数开始时，通过一个指针p遍历原始的单链表。对于单链表中的每一个节点，函数会根据节点中数据的类型（字母、数字或其他字符）来决定将其加入哪一个循环链表。为了加入循环链表，首先为节点数据创建一个新的节点newNode，然后依次判断类型并插入。继续遍历原始链表中的下一个节点，直到所有节点都被处理。

实验结果：

```
75  //添加按字母、数字、其它字符分解单链表算法
76  void resolve(linklist *head, linklist *letter, linklist *digit, linklist *other) {
77      linklist *p = head->next;
78      while (p != NULL) {
79          // 创建一个新节点把当前节点数据存入
80          linklist *newNode = (linklist*)malloc(sizeof(linklist));
81          newNode->data = p->data;
82          newNode->next = NULL;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
1a!2b#3d*$
1a!2b#3d*
abd
123
!#*
```

（二）栈和队列

栈：

栈是一种特殊的线性表，其特殊之处在于它只允许在表的一端进行插入和删除操作。这一端被称为栈顶，另一端则被称为栈底。由于栈的这种特点，最后进入栈的元素必定是第一个被取出的。在栈中进行插入操作，称为压栈或进栈，在栈中进行删除操作，称为弹栈或出栈。由于栈只允许在栈顶进行操作，这使得栈的操作非常快速且容易管理。栈经常被用于存储临时数据，特别是在递归算法中非常有用。

队列：

队列遵循先进先出的原则，即先进入队列的元素将首先被移出。队列类似于现实生活中排队的概念，先到先服务。队列有两个主要操作：入队和出队。当一个元素被加入到队列时，它被添加到队列的尾部；而当一个元素被从队列中移出时，它是从队列的头部移除的。除了基本的入队和出队操作外，队列还可能支持其他操作，如获取队列头部的元素但不移除、检查队列是否为空等。

实验目的：

1. 熟悉栈和队列的顺序和链式存储结构
2. 掌握栈和队列的基本运算
3. 能够利用栈和队列的基本运算完成栈和队列应用的运算

4. 判字符串中心对称

程序说明：

判断输入的链串是否是中心对称。我们可以借助栈这种数据结构达到目的。栈有后入先出的特性，因此我们可以把链表前半部分的字符都放入栈中，然后逐一与后半部分进行比较。在这个函数中：首先计算链表的长度n，把链表前半部分的字符入栈，与后半部分的字符依次进行比较，如果比较过程中发现有不同的字符，说明字符串不是中心对称，返回0并结束函数。如果和后半部分的每个字符都相同，那么字符串才是中心对称，函数返回1。

实验结果：

```
98  int symmetry(linklist*head,stack*s) {
99      linklist *p;
100     int i,n;
101     datatype e;
102     p=head->next;
103     n=length(head);
104     for(i=1;i<=n/2;i++) {
105         push(s,p->data); p=p->next; }
106     if(n%2==1) p=p->next;
107     while(p!=NULL) {
108         e=pop(s);
109         if(e!=p->data)
110             return 0;
111         p=p->next; }
112     return 1;
113 }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

abcdedcba
abcdedcba
字符串"abcdedcba"中心对称

abcdefg
abcdefg
字符串"abcdefg"不是中心对称
```

5. 循环队列

程序说明：

在这个实现中，队尾元素的位置是由rear标记的。对于队头元素，我们通过队尾元素的位置和队列长度计算得出。取模操作是为了实现循环队列，当索引走到队列的末端时，取模就可以回到队列的头部。

入队算法：

首先检查队列是否已满，如果已满，打印一条消息并不进行任何操作。如果队列未满，更新队尾标记，将 rear 指针向“前”移动一个位置，并更新队列长度，存入新元素。通过取模运算确保当索引走到队列的末端时可以回到队列的头部。

```
43  //添加入队算法
44  void enqueue(qu *sq, datatype x) {
45      if (sq->quelen >= m) {
46          printf("Queue is full\n");
47      } else {
48          sq->rear = (sq->rear + 1) % m;
49          sq->sequ[sq->rear] = x;
50          sq->quelen++;
51      }
52  }
```

出队算法：

首先检查队列是否为空，即队列长度是否为0，如果为空，打印一条消息并返回 NULL。如果非空，计算队头的位置这是通过第61行的代码实现的。如果队列没有出现循环的情况，队尾减队长加一总是小于m的，但如果出现循环，队尾减去队长加一可能出现负数，这时候我们需要加m来修正这个数字，因此出现了第61行的表达形式。

```
55  //添加出队算法
56  datatype *dequeue(qu *sq) {
57      if (sq->quelen <= 0) {
58          printf("Queue is empty\n");
59          return NULL;
60      } else {
61          int front = (sq->rear - sq->quelen + 1 + m) % m;  // 计算队列头部元素的位置
62          datatype *result = (datatype*)malloc(sizeof(datatype));
63          *result = sq->sequ[front];
64          sq->quelen--;
65          return result;
66      }
67  }
```

存在问题：

这里比较有趣的一点在于出队列的时候进行的深拷贝（62和63行），一开始我直接将相应位置的指针返回，但后来发现这样的话后续往同一位置入队列的时候就会复写掉原来的数据，而且刚刚返回的数据实际上也被一并复写了（因为二者是同一块空间）。因此此处在返回指针时先创建了一块空间将内容深拷贝后返回新空间的指针，这样做时比较安全的。后来我又发现，其实这里入队列和出队列的操作模式不一致，应该都传递数据值或者都传递指针，这样一致的传递不会发生上述问题。

实验结果：

```
1.Enter Queue   2.Delete Queue   -1.Quit:1
Enter the Data:1
1.Enter Queue   2.Delete Queue   -1.Quit:1
Enter the Data:2
1.Enter Queue   2.Delete Queue   -1.Quit:1
Enter the Data:3
1.Enter Queue   2.Delete Queue   -1.Quit:1
Enter the Data:4
1.Enter Queue   2.Delete Queue   -1.Quit:1
Enter the Data:5
1.Enter Queue   2.Delete Queue   -1.Quit:1
Enter the Data:6
Queue is full
```

```
1.Enter Queue   2.Delete Queue   -1.Quit:2
1
1.Enter Queue   2.Delete Queue   -1.Quit:2
2
1.Enter Queue   2.Delete Queue   -1.Quit:2
3
1.Enter Queue   2.Delete Queue   -1.Quit:2
4
1.Enter Queue   2.Delete Queue   -1.Quit:2
5
1.Enter Queue   2.Delete Queue   -1.Quit:2
Queue is empty
1.Enter Queue   2.Delete Queue   -1.Quit:-1
```

（三）串

串：

串是由字符组成的有限序列，是一种特殊的线性表。字符可以是字母、数字、符号或其他特殊字符，而串则是这些字符按照特定顺序排列而成的数据结构。串是一种非常常见的数据类型，用于表示文本或任何字符序列。串的基本操作包括串的连接、截取、查找、替换等。

实验目的：

1. 熟悉串的顺序存储结构
2. 掌握串的基本运算及应用

6. 模式匹配

程序说明：

使用两个变量 i 和 j 分别跟踪主串 S 和子串 subS 中的当前字符位置。当 S->str[i] 与 subS->str[j] 匹配时，i 和 j 都向前移动一位，继续比较下一个字符。如果当前字符不匹配，i 退回到这次匹配尝试的起始位置的下一位置，j 重置为 0，即子串从头开始与主串的下一位置进行匹配。如果 j 达到了子串 subS 的长度，说明子串完全匹配，返回子串在主串中的起始位置。如果主串 S 结束而子串 subS 未能完全匹配，则返回 -1 表示匹配失败。这只是朴素的模式匹配算法，算法存在冗余，实际上可以有更快速的模式匹配方法。

实验结果：

```
29  //添加顺序串的朴素模式匹配算法
30  int Index(seqstring *S, seqstring *subS) {
31      int i = 0, j = 0;
32
33      while (i < S->len && j < subS->len) {
34          if (S->str[i] == subS->str[j]) { // 当前字符匹配
35              i++; j++;
36          } else {
37              i=i-j+1; j=0;  // 主串回溯，子串从头开始
38          }
39      }
40
41      if (j == subS->len) return i - subS->len; // 匹配成功，返回子串在主串中的位置
42      else return -1; // 匹配失败
43  }
```

```
输入串：abcdefg
输入子串：cde
匹配成功！

输入串：abcdefg
输入子串：acg
匹配失败！
```

7. 删除子串

程序说明：

首先检查参数 i 和 m 是否合法。如果参数不合法，函数返回。使用一个循环从位置 i + m 开始，将每个后续字符向前移动 m 个位置，以此来“覆盖”掉要删除的字符。在完成字符移动后，更新字符串的长度。

实验结果：

```
39  //添加删除子串算法
40  void strDelete(seqstring *S, int i, int m) {
41      if (i < 0 || i >= S->len || m < 0 || i + m > S->len) {
42          printf("参数不合法\n"); return;
43      }
44      for (int j = i + m; j < S->len; j++) {
45          S->str[j - m] = S->str[j]; // 将后面的字符前移
46      }
47      S->len -= m; // 更新长度
48  }
```

```
输入串：abcdefg
abcdefg
删除的开始位置:3
删除的字符个数:2
abcfg
```

（四）数组

数组：

数组是一种存储相同类型元素的线性数据结构。这些元素可以通过索引或下标来访问，索引通常是整数。数组的特点是它们在内存中是连续存储的，这使得按索引直接访问元素非常高效。数组的一些基本操作包括初始化、读取、修改、添加元素、删除元素等。

实验目的：

1. 熟悉数组的结构
2. 掌握矩阵的压缩存储
3. 能够对数组和矩阵的压缩存储进行运算

8. 对称矩阵相乘

程序说明：

要实现对称矩阵相乘的算法，我们需要考虑到对称矩阵的特殊性质：它们的上三角和下三角是对称的。在这个程序中，我们假设矩阵A和矩阵B都是对称的，并且它们的下三角部分以行为主序存储在一维数组中。算法的核心部分就是从矩阵A和矩阵B中取元素时的规则，当我想取下三角位置元素时，直接计算位置取元素即可；但是如果要取得元素位于上三角其实我们要取下三角对应位置得元素，取对称位置的方式也非常简单只需要把行号和列号交换一下然后取交换后位置的元素即可，也就是代码52和53行的内容。除此之外取元素时还需要计算我们要取的元素在一维表中的位置，而这只需要根据等差数列求和公式找到在此之前有多少个元素即可。

实验结果：

```
46  //添加对称矩阵相乘算法
47  void mult(array *pa) {
48      // 计算矩阵乘积
49      for (int i = 0; i < n; i++) {
50          for (int j = 0; j < n; j++) {
51              for (int k = 0; k < n; k++) {
52                  int a_ik = (i >= k) ? pa->A[i * (i + 1) / 2 + k] : pa->A[k * (k + 1) / 2 + i];
53                  int b_kj = (k >= j) ? pa->B[k * (k + 1) / 2 + j] : pa->B[j * (j + 1) / 2 + k];
54                  pa->C[i][j] += a_ik * b_kj;
55  }   }   }   }
```

```
以行为主序输入矩阵A的下三角：
1 2 3 4 5 6
以行为主序输入矩阵B的下三角：
1 2 1 2 1 2
   13    8   12
   18   12   17
   26   19   25
```

9. 找马鞍点

程序说明：

首先遍历矩阵以确定最大最小值。然后再遍历矩阵，检查每个元素是否是马鞍点。如果是，那么这个元素就是一个马鞍点，打印其位置和值。如果在整个矩阵中没有找到马鞍点，打印相应的消息。在大多数情况下马鞍点不会出现多个，除非一些相等的情况，没有很高的要求时，其实找到马鞍点立即退出，是比较高效的做法。

实验结果：

```
24  //添加找马鞍点算法
25  void minmax(array *pa) {
26      int i, j, k, saddlePointExists = 0;
27
28  >   for (i = 1; i <= m; i++) {…
34      }// 初始化每行的最大值
35
36  >   for (j = 1; j <= n; j++) {…
42      }// 初始化每列的最小值
43
44  >   for (i = 1; i <= m; i++) {…
51      }// 查找马鞍点
52
53      if (!saddlePointExists) {
54          printf("没有马鞍点\n");
```

```
1 2 1
0 1 0
1 2 1
马鞍点在 [2, 2] , 值是 1
```

（五）树

树：

树是一种非常常见且重要的数据结构，树简化到极致就得到了线性表，换句话说树是线性表的拓展。树的结构类似于真实世界中的树，有树根、枝干、叶子，其中：根节点是树的顶端节点，是树的起点；叶子节点是没有子节点的节点，位于树的末端。树的每个节点都可以有零个或多个子节点，这些子节点本身也可以有自己的子节点也是一棵树，形成了一种递归的层级结构，树的很多算法也是利用递归的思想实现的。

二叉树：

树有许多不同的类型，其中常见的包括二叉树和二叉搜索树：二叉树每个节点最多有两个子节点，分别称为左子节点和右子节点。二叉搜索树是一种特殊的二叉树，它满足左子节点的值小于父节点的值，右子节点的值大于父节点的值，这种性质使得二叉搜索树可以进行高效的查找、插入和删除操作。

实验目的：

1. 熟悉二叉树的链式存储结构
2. 掌握二叉树的建立、深度优先递归遍历等算法
3. 能够利用遍历算法实现一些应用

10. 交换左右子树

程序说明：

这个函数接收一个指向二叉树根节点的指针，并返回处理后的树的根节点指针。保存当前节点的左子树到临时变量。然后，调用swap函数递归地交换当前节点的右子树，并将返回的新树作为当前节点的左子树。接着，调用swap函数递归地交换之前保存在临时变量中的原左子树，将返回的新树作为当前节点的右子树。最后，返回左右子树已经被交换的当前节点。当递归调用达到叶节点时遇到空的左右子树，递归开始回溯，最终全都被交换。

实验结果：

```
80  //添加交换左右子树算法
81  bitree* swap(bitree* root) {
82      if (root == NULL) return NULL;
83      // 递归地交换左右子树
84      bitree* temp = root->lchild;
85      root->lchild = swap(root->rchild);
86      root->rchild = swap(temp);
87      // 返回
88      return root;
89  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
按层次输入二叉树，虚结点输入'@'，以'#'结束输入：
ABC@DE@#
A(B(,D),C(E))
A(C(,E),B(D))
```

11. 统计二叉树节点

程序说明：

与上一个程序类似，我使用递归的方式来统计二叉树的节点。分别统计出左孩子的节点和右孩子的节点，然后再加上自己的一个节点就得到总节点数。统计叶子数也类似，分别统计出左孩子和右孩子的叶子数，加起来就是总的叶子数。

实验结果：

```
84  //添加统计结点个数算法
85  int countnode(bitree *root) {
86      if(root == NULL) return 0; //当前节点为空，返回0
87      else {
88          int left = countnode(root->lchild);   //统计左子树节点个数
89          int right = countnode(root->rchild);  //统计右子树节点个数
90          return left + right + 1;  //返回左右子树节点个数之和再加上自身一个节点，算一个树有多少节点
91      }
92  }
93
94  //添加统计叶子结点个数算法
95  int countleaf(bitree *root) {
96      if(root == NULL) return 0; //如果节点为空，则返回0
97      else if(root->lchild == NULL && root->rchild == NULL) return 1; //如果节点为叶子节点，返回1
98      else {
99          int left = countleaf(root->lchild);   //统计左子树中叶子节点的个数
100         int right = countleaf(root->rchild);  //统计右子树中叶子节点的个数
101         return left + right;  //叶子节点的总数为左子树叶子节点个数加右子树叶子节点个数
102     }
103 }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

按层次输入结点值，虚结点输入'@'，以换行符结束：ABC@DE@
删除子树之前的二叉树：A(B(,D),C(E))
结点总数是：5
叶子结点总数是：2
```

（六）图

图：

图是一种由节点和连接这些节点的边组成的数据结构，它可以用来表示各种关系和网络结构，图是树的拓展，树是没有环路的图。图的应用非常广泛，包括社交网络中的好友关系、计算机网络中的节点连接等。图算法涉及许多重要的问题，如最短路径、最小生成树等。

实验目的：

1. 熟悉图的邻接矩阵和邻接表的存储结构
2. 熟悉图的邻接矩阵和邻接表的建立算法
3. 掌握图的遍历算法

12. 无向图邻接矩阵

程序说明：

这种搜索算法以递归的方式探索图的顶点，访问所有可达的顶点。visited 是一个全局数组，用来跟踪哪些顶点已经被访问过。如果访问过就不再访问。遍历所有顶点，检查哪些顶点与当前顶点邻接且没有被访问过。则在这个新节点递归地调用dfsa函数，以新节点作为新的出发点进行深度优先搜索。总的来说，dfsa函数首先访问并标记当前顶点，然后对每一个与之相邻且未被访问的顶点递归地执行相同的过程。这样，函数会深入到每一个可达的顶点，直到所有连通的顶点都被访问过。

实验结果：

```
49  //添加深度优先搜索遍历算法
50  void dfsa(int v) {
51      printf("%c ", g->vexs[v]); // 访问顶点
52      visited[v] = 1;
53      for (int i = 0; i < n; i++) {
54          if (g->arcs[v][i] == 1 && !visited[i]) {
55              dfsa(i); // 对未访问的邻接顶点递归调用
56          }
57      }
58  }
```

```
输入8个顶点的字符数据信息：
ABCDEFGH
输入10条边的起、终点i,j:
0,1 0,2 1,4 1,5 2,5 2,6 3,7 4,7 5,7 6,7

输入出发点序号（0-7），输入-1结束：0
A B E H D F C G
输入出发点序号（0-7），输入-1结束：7
H D E B A C F G
输入出发点序号（0-7），输入-1结束：4
E B A C F H D G
输入出发点序号（0-7），输入-1结束：-1
```

（七）排序

排序：

排序是一种将元素序列按照特定顺序排列的过程。常见的排序算法包括冒泡排序、快速排序、归并排序、插入排序等。排序是数据处理中的一个基本任务，用于数据组织、搜索和优化等方面。排序算法本身有很多种，他们适用于不同的情况，有着不同的时间和空间复杂度，没有绝对的好和差，实际应用时应该根据需要选择最适合的排序算法。

实验目的：

1. 熟悉各种内部排序算法
2. 能够编写程序显示排序过程中各趟排序的结果
3. 能够编写一些排序的算法

13. 希尔排序

程序说明：

希尔排序是一种基于插入排序的算法，通过将原始数据分割成多个子序列来提高排序的效率。先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下，效率是很高的，因此希尔排序在时间效率较高。

实验结果：

```
45  //添加希尔排序算法
46  void shellsort(rectype r[], int d[]) {
47      int i, j, k, m;
48      rectype temp;
49      for (m = 0; m < 3; m++) { // 逐个取出增量值
50          int dk = d[m]; // 当前增量
51          for (i = dk; i < N; i++) { // 按当前增量进行插入排序
52              temp = r[i + D1]; // 取出无序序列中的第i个作为待插入元素
53              j = i - dk; // 序列中上一个位置
54              while (j >= 0 && temp.key < r[j + D1].key) { // 寻找插入位置
55                  r[j + dk + D1] = r[j + D1]; // 数据移动
56                  j -= dk; // 继续向前寻找
57              }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

输入10个整型数: 1 10 9 2 8 3 7 4 6 5
排序前的数据:     1   10    9    2    8    3    7    4    6    5
排序后的数据:     1    2    3    4    5    6    7    8    9   10
```

14. 双向起泡排序

程序说明：

双向起泡排序通常比传统的冒泡排序更快，它对元素的处理方式更加高效。传统的冒泡排序只在一个方向上进行，每次遍历只能将一个元素移动到正确的位置。双向起泡排序在一个完整的遍历周期中，首先从低到高进行一次排序，然后再从高到低进行一次排序。这意味着在每个完整的遍历周期中，可以将两个元素（一个最大，一个最小）移动到它们各自正确的位置。然而，需要注意的是，双向起泡排序并不总是比传统冒泡排序快。在最坏情况下，两者的时间复杂度都是$O(n^2)$，但总的来说双向起泡排序的平均性能可能更好。实际性能提升取决于具体的数据分布和数组的初始状态。

实验结果：

```
44  //添加双向起泡排序算法
45  void dbubblesort(sequenlist r[], int n) {
46      int low = 1, high = n; // 设置排序范围的上下界
47      int flag = 1; // 标志变量，用于优化排序过程
48      while (low < high && flag == 1) {
49          flag = 0;
50          for (int j = low; j < high; j++) {…
57          } // 从低到高进行冒泡排序
58          high--; // 尾部已经固定不需要再排序
59          for (int j = high; j > low; j--) {…
66          } // 从高到低进行冒泡排序
67          low++; // 头部已经固定不需要再排序
68      }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
排序前的数据:    33   48   54   80   51   74   86   87   45   36
排序后的数据:    33   36   45   48   51   54   74   80   86   87
```

（八）查找

查找：

查找指的是在数据结构中寻找特定元素的过程。查找可以在各种数据结构中进行，如数组、链表、树、哈希表等。查找的目标是确定一个特定的值是否存在于数据集中，如果存在，通常还需要找出它的位置。线性查找从数据结构的第一个元素开始，逐个检查每个元素，直到找到目标元素或遍历完所有元素。线性查找不需要数据预先排序，但效率较低，尤其是在数据量大的情况下。非线性查找通常依赖于数据的某种组织方式来加快查找速度。比如二分查找，比较中间元素与目标值，根据比较结果选择数组的一半继续查找。在实际应用中，选择哪种查找方法取决于数据的类型、结构以及是否已排序等因素。

实验目的：

1. 熟悉线性表、二叉排序树和散列表的查找
2. 能够编写一些查找的算法

15. 分块查找

程序说明：

希尔排序是一种基于插入排序的算法，通过将原始数据分割成多个子序列来提高排序的效率。先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下，效率是很高的，因此希尔排序在时间效率较高。

实验结果：

```
36  //添加折半查找索引表，块内顺序查找算法
37  int blksearch(record r[], index idx[], keytype key, int idxN) {
38      int i, block = -1;
39      for (i = 0; i < idxN; i++) { // 在索引表中定位块
40          if (key <= idx[i].key) {
41              block = i;  break; } }
42      if (block != -1) // 在找到的块内进行线性查找
43          for (int k = idx[block].low; k <= idx[block].high; k++)
44              if (r[k].key == key)
45                  return k; // 找到目标，返回位置
46      return -1; // 未找到
47  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
待查找的记录关键字表:
   22   12   13    8    9   20   33   42   44   38   24   48   60   58   74   49   86   53
输入所要查找记录的关键字: 33
查找到，是第7个记录。
```

16. 判断二叉排序树

程序说明：

这个程序用来判断一个二叉树是否为二叉排序树。二叉排序树中每个节点的左子树中的所有节点的键值都小于该节点的键值，而右子树中的所有节点的键值都大于该节点的键值。这个程序使用了递归的方法进行判断。如果根节点为空，则直接返回是BST。然后依次检查当前节点、左子树、右子树，有任何一部分不是二叉排序树就返回0，否则返回1。

实验结果：

```
92   //判断二叉排序树
93   int isBST(bitree *root) {
94       if (root==NULL) return 1;
95       if (root->lchild!=NULL && root->lchild->key > root->key) return 0;
96       if (root->rchild!=NULL && root->rchild->key < root->key) return 0;
97       if (!isBST(root->lchild)) return 0;
98       if (!isBST(root->rchild)) return 0;
99       return 1;
100  }
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
按层次输入二叉排序树的整型结点数据,0表示虚结点,-1表示结束:
3 2 5 1 0 4 0 -1
3(2(1),5(4))
1 2 3 4 5
是二叉排序树!
按层次输入二叉排序树的整型结点数据,0表示虚结点,-1表示结束:
1 2 3 0 4 5 0 -1
1(2(,4),3(5))
2 4 1 5 3
不是二叉排序树!
```

其他方法分析：

就这个问题而言，其实存在一种比较偷懒的方法，我们观察上面的程序输出结果，对二叉排序树来说，中序遍历的结果一定是有序的。因此我们可以利用中序遍历函数，用一个全局变量来缓存“上一个访问的节点数值”，访问当前节点时，就拿着当前节点与上一个节点的数值比较，对二叉排序树来说，它理应比上一个节点大，因此只要比上一个节点小就可以退出程序判定不是二叉排序树，如果指导遍历结束都符合条件那么说明是二叉排序树。这两种方法的比较顺序有区别，在不同的数据下会有略微不同的表现，但总体上复杂度相同没有太大的差别。

三、总结

在本学期的数据结构课程上机实验中，我锻炼了编程技能，加深了对各种数据结构的深刻理解。在上机作业中，我完成了实验任务，而且深入探讨了一部分算法的优缺点，分析算法复杂度，思考了算法更简单和高效实现方法，提出了改进的可能方法。总而言之，我在数据结构课程上机实验中受益匪浅，提高了专业素养和综合能力。最后，取得如此的进步，还要多谢老师本学期的辛勤教导。