**本科课程实验报告**



**（2021-2022学年第二学期）**

课程名称： 数据结构与算法实验

作 者： 刘宇诺

学 号： 20201210207

所在学院： 信息科学与工程学院

专业班级： 计算机科学与技术20-1

指导教师： 曹为民

实验一

|  |  |
| --- | --- |
| **实验名称** | 实验一 线性表的顺序存储及相应操作 |
| **实验目的和要求 ：** | |
| 实现线性表的顺序存储结构，并实现相应的插入、删除、定位等操作 | |
| **实验实现思路及步骤：** | |
| 1．顺序表插入操作的基本步骤：要在顺序表中的第i个数据元素之前插入一个数据元素x,首先要判断插入位置i是否合法，假设线性表的表长为n，则i的合法值范围：1≤i≤n+1，若是合法位置，就再判断顺序表是否满，如果满，则增加空间或结束操作，如果不满，则将第i个数据元素及其之后的所有数据元素都后移一个位置，此时第i个位置已经腾空，再将待插入的数据元素x插入到该位置上，最后将线性表的表长增加1。  2．顺序表删除操作的基本步骤：要删除顺序表中的第i个数据元素，首先仍然要判断i的合法性，i 的合法范围是1≤i≤n，若是合法位置，则将第i个数据元素之后的所有数据元素都前移一个位置，最后将线性表的表长减1。  3．顺序表查找操作的基本步骤：要在顺序表中查找一个给定值的数据元素，则可以采用顺序查找的方法，从顺序表中第1个数据元素开始依次将数据元素值与给定值进行比较，若相等则返回该数据元素在顺序表中的位置，否则返回0值。 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| #include <stdio.h>  #include <stdlib.h>  #include "Seqlist.h"  SeqList\* SL\_Create(int maxlen)  // 创建一个顺序表。  // 与SqLst\_Free()配对。  {      SeqList\* slist=(SeqList\*)malloc(sizeof(SeqList));      slist->data = (T\*)malloc(sizeof(T)\*maxlen);      slist->max=maxlen;      slist->len=0;      return slist;  }  void SL\_Free(SeqList\* slist)  // 释放/删除 顺序表。  // 与SqLst\_Create()配对。  {      free(slist->data);      free(slist);  }  void SL\_MakeEmpty(SeqList\* slist)  // 置为空表。  {      slist->len=0;  }  int SL\_Length(SeqList\* slist)  // 获取长度。  {      return slist->len;  }  bool SL\_IsEmpty(SeqList\* slist)  // 判断顺序表是否空。  {      return 0==slist->len;  }  bool SL\_IsFull(SeqList\* slist)  // 判断顺序表是否满。  {      return slist->len==slist->max;  }  T SL\_GetAt(SeqList\* slist, int i)  // 获取顺序表slist的第i号结点数据。  // 返回第i号结点的值。  {      if(i<0||i>=slist->len) {          printf("SL\_GetAt(): location error when reading elements of the slist!\n");          SL\_Free(slist);          exit(0);      }      else          return slist->data[i];  }  void SL\_SetAt(SeqList\* slist, int i, T x)  // 设置第i号结点的值（对第i号结点的数据进行写）。  {      if(i<0||i>=slist->len) {          printf("SL\_SetAt(): location error when setting elements of the slist!\n");          SL\_Free(slist);          exit(0);      }      else          slist->data[i]=x;  }  bool SL\_InsAt(SeqList\* slist, int i, T x)  // 在顺序表的位置i插入结点x, 插入d[i]之前。  // i 的有效范围[0,plist->len]。  {      // 请在下面的Begin-End之间补充代码，插入结点。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (i > slist->len || i < 0) return false;      int n = slist->len - i - 1;      for (int j = slist->len + 1; j > i; j -- )      {          slist->data[j] = slist->data[j - 1];      }      slist->data[i] = x;      slist->len ++ ;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  T SL\_DelAt(SeqList\* slist, int i)  // 删除顺序表plist的第i号结点。  // i的有效范围应在[0,plist->len)内，否则会产生异常或错误。  // 返回被删除的数据元素的值。  {      // 在下面的Begin-End之间补充代码，删除第i号结点。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (i < 0 || i >= slist->len) return 0;      for (int j = i; j < slist->len; j ++ )      {          slist->data[j] = slist->data[j + 1];      }      slist->len -- ;      return slist->data[i];      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  int SL\_FindValue(SeqList\* slist, T x)  // 在顺序表表中查找第一个值为x的结点，返回结点的编号。  // 返回值大于等于0时表示找到值为x的结点的编号，-1表示没有找到。  {      int i=0;      while(i<slist->len && slist->data[i]!=x) i++;      if (i<slist->len) return i;      else return -1;  }  int SL\_DelValue(SeqList\* slist, T x)  // 删除第一个值为x的结点。  // 存在值为x的结点则返回结点编号, 未找到返回－1。  {      // 在下面的Begin-End之间补充代码，删除第一个值为 x 的结点。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      int n = slist->len;      for (int i = 0; i <= n; i ++ )      {          if (slist->data[i] == x)          {              for (int j = i; j < slist->len; j ++ )                  slist->data[j] = slist->data[j + 1];              slist->len -- ;              return i;          }      }      return -1;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  void SL\_Print(SeqList\* slist)  // 打印整个顺序表。  {      if (slist->len==0) {          printf("The slist is empty.\n");          return;      }      //printf("The slist contains: ");      for (int i=0; i<slist->len; i++) {          printf("%d  ", slist->data[i]);      }      printf("\n");    } | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 顺序表优点：（1）空间利用率高，数据是连续存放，命中率比较高 （2）存取速度高效，通过下标来直接访问。  缺点：（1）插入和删除比较慢，在插入或者删除一个元素的时候，需要遍历整个顺序表移动前后的数据（2）需要预先分配足够大的空间，估计的过大，会导致顺序表后的大量空间被闲置；但是估计的过小，又会造成溢出。 | |
| **指导教师评语及评分建议：** | |
|  | |

实验二

|  |  |
| --- | --- |
| **实验名称** | 实验二 线性表的链式存储及操作 |
| **实验目的和要求 ：** | |
| 实现线性表的链式存储结构，并实现相应的插入、删除、定位等操作 | |
| **实验实现思路及步骤：** | |
| 头插法：  创建单链表的过程就是一个动态生成链表的过程，单链表头插法的思路是：  5. 声明一个节点p和计数变量i；  6. 初始化一空链表L；  7. 让L的头节点的指针指向null，建立一个带头结点的单链表。  8. 循环：  ①生成一新结点赋值给P；  ②随机生成艺术字赋值给p的数据域；  ③将p插入到头节点与前一新结点之间。  尾插法：  尾插法即我们在标为添加，我们设 r 为指向尾结点的变量，r 会随着循环不断的变化结点，而L我们定义为整个单链表，我们把原先的 r 的指针域存放新的尾结点的地址，然后我们最后让 r 拿到新的尾结点的地址，这样就完成了尾插法。 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| // 单链表实现文件  #include <stdio.h>  #include <stdlib.h>  #include "LinkList.h"  // 1）  LinkList\* LL\_Create()  // 创建一个链接存储的线性表，初始为空表，返回llist指针。  {      LinkList\* llist=(LinkList\*)malloc(sizeof(LinkList));      llist->front=NULL;      llist->rear=NULL;      llist->pre=NULL;      llist->curr=NULL;      llist->position=0;      llist->len=0;      return llist;  }  // 2）  void LL\_Free(LinkList\* llist)  // 释放链表的结点，然后释放llist所指向的结构。  {      LinkNode\* node=llist->front;      LinkNode\* nextnode;      while(node){          nextnode=node->next;          free(node);          node=nextnode;      }      free(llist);  }  // 3）  void LL\_MakeEmpty(LinkList\* llist)  // 将当前线性表变为一个空表，因此需要释放所有结点。  {      LinkNode\* node=llist->front;      LinkNode\* nextnode;      while(node){          nextnode=node->next;          free(node);          node=nextnode;      }      llist->front=NULL;      llist->rear=NULL;      llist->pre=NULL;      llist->curr=NULL;      llist->position=0;      llist->len=0;  }  // 4）  int LL\_Length(LinkList\* llist)  // 返回线性表的当前长度。  {      return llist->len;  }  // 5）  bool LL\_IsEmpty(LinkList\* llist)  // 若当前线性表是空表，则返回true，否则返回 False。  {      return llist->len==0;  }  // 6）  bool LL\_SetPosition(LinkList\* llist, int i)  // 设置线性表的当前位置为i号位置。  // 设置成功，则返回true，否则返回false（线性表为空，或i不在有效的返回）。  // 假设线性表当前长度为len，那么i的有效范围为[0,len]。  {      int k;      /\* 若链表为空，则返回\*/      if (llist->len==0) return false;      /\*若位置越界\*/      if( i < 0 || i > llist->len)      {   printf("LL\_SetPosition(): position error");          return false;      }      /\* 寻找对应结点\*/      llist->curr = llist->front;      llist->pre = NULL;      llist->position = 0;      for ( k = 0; k < i; k++)    {          llist->position++;          llist->pre = llist->curr;          llist->curr = (llist->curr)->next;      }        /\* 返回当前结点位置\*/      return true;  }  // 7)  int LL\_GetPosition(LinkList\* llist)  // 获取线性表的当前位置结点的编号。  {      return llist->position;  }  // 8）  bool LL\_NextPosition(LinkList\* llist)  // 设置线性表的当前位置的下一个位置为当前位置。  // 设置成功，则返回true，否则返回false（线性表为空，或当前位置为表尾）。  {      if (llist->position >= 0 && llist->position < llist->len)      /\* 若当前结点存在，则将其后继结点设置为当前结点\*/      {          llist->position++;          llist->pre = llist->curr;          llist->curr = llist->curr->next;          return true;      }      else          return false;  }  // 9）  T LL\_GetAt(LinkList\* llist)  // 返回线性表的当前位置的数据元素的值。  {      if(llist->curr==NULL)      {          printf("LL\_GetAt(): Empty list, or End of the List.\n");          LL\_Free(llist);          exit(1);      }      return llist->curr->data;  }  // 10）  void LL\_SetAt(LinkList\* llist, T x)  // 将线性表的当前位置的数据元素的值修改为x。  {      if(llist->curr==NULL)      {          printf("LL\_SetAt(): Empty list, or End of the List.\n");          LL\_Free(llist);          exit(1);      }      llist->curr->data=x;  } | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 优点：（1）插入和删除速度快，保留原有的物理顺序，在插入或者删除一个元素的时候，只需要改变指针指向即可。（2）没有空间限制,存储元素无上限,只与内存空间大小有关. （3）动态分配内存空间，不用事先开辟内存（4)是内存的利用率变高  缺点：（1）占用额外的空间以存储指针，比较浪费空间，不连续存储，malloc函数开辟空间碎片比较多 （2）查找速度比较慢，因为在查找时，需要循环链表。 | |
| **指导教师评语及评分建议：** | |
|  | |

实验三

|  |  |
| --- | --- |
| **实验名称** | 实验三 栈和队列 |
| **实验目的和要求 ：** | |
| 实现顺序栈、链式栈、循环队列、链式队列的相关操作 | |
| **实验实现思路及步骤：** | |
| （1）栈：只允许在一端进行插入或删除操作的[线性表](https://so.csdn.net/so/search?q=%E7%BA%BF%E6%80%A7%E8%A1%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_43752469/article/details/_blank)。特点：先进后出。  栈顶：允许进行插入删除。  栈底：不允许进行插入和删除，固定的。  （2）队列：也是一种操作受限的线性表，只允许在表的一端进行插入，在表的另一端进行删除。  队头：允许删除的一端，出队。  队尾：允许插入的一端，入队。 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| /\*判断栈是否为满\*/  bool SS\_IsFull(SeqStack\* ss)  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      if (ss->top + 1 == ss->max) return true;      return false;      /\*\*\*\*\*\*END\*\*\*\*\*\*/  }  /\*判断栈是否为空\*/  bool SS\_IsEmpty(SeqStack\* ss)  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      if (ss->top == -1) return true;      return false;      /\*\*\*\*\*\*END\*\*\*\*\*\*/  }  /\*获取栈元素个数\*/  int SS\_Length(SeqStack\* ss)  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      return ss->top + 1;      /\*\*\*\*\*\*END\*\*\*\*\*\*/  }  /\*将x进栈,满栈则无法进栈(返回false)\*/  bool SS\_Push(SeqStack\* ss, T x)  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      if (ss->top + 1 == ss->max) return false;      ss->data[ ++ ss->top] = x;      return true;      /\*\*\*\*\*\*END\*\*\*\*\*\*/  }  /\*出栈，出栈的元素放入item,空栈则返回false\*/  bool SS\_Pop(SeqStack\* ss, T &item)  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      if (SS\_IsEmpty(ss)) return false;      item = ss->data[ss->top];      ss->top -- ;      return true;      /\*\*\*\*\*\*END\*\*\*\*\*\*/  }  /\*判断栈是否为空\*/  bool LS\_IsEmpty(LinkStack\* ls)  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (ls->len == 0) return true;      return false;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  /\*获取栈的长度\*/  int LS\_Length(LinkStack\* ls)  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      return ls->len;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  /\*将x进栈\*/  void LS\_Push(LinkStack\* ls, T x)  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      LNode\* p = (LNode\*)malloc(sizeof(LNode\*));      p->data = x;      ls->len += 1;      p->next = ls->top;      ls->top = p;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  /\*出栈。出栈元素放入item；如果空栈，将返回false\*/  bool LS\_Pop(LinkStack\* ls, T& item)  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (LS\_IsEmpty(ls)) return false;      item = ls->top->data;      LNode\* p = ls->top;      ls->top = p->next;      ls->len -= 1;      return true;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  /\*读栈顶元素放入item。如果空栈，将返回false\*/  bool LS\_Top(LinkStack\* ls, T& item)  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (LS\_IsEmpty(ls)) return false;      item = ls->top->data;      return true;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  bool SQ\_IsEmpty(SeqQueue\* sq)  // 判断队列是否为空，为空返回true，否则返回false。  {      // 请在Begin-End之间补充代码，完成队列是否为空的判断。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (sq->front == sq->rear)          return true;      return false;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  bool SQ\_IsFull(SeqQueue\* sq)  // 判断队列是否为满。为满返回true,否则返回false。  {      // 请在Begin-End之间补充代码，完成队列是否为满的判断。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if ((sq->front == 0 && sq->rear == sq->max - 1) || sq->front == sq->rear + 1)          return true;      return false;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  int SQ\_Length(SeqQueue\* sq)  // 队列长度。  {      // 请在Begin-End之间补充代码，获取队列长度。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      int len = 0;      len = (sq->rear - sq->front + sq->max) % sq->max;      return len;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  bool SQ\_In(SeqQueue\* sq, T x)  // 将x入队。若入队失败(队列满)，则返回false，否则返回true。  {      // 请在Begin-End之间补充代码，将 x 入队。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (SQ\_IsFull(sq)) return false;      sq->data[sq->rear] = x;      sq->rear ++ ;      if (sq->rear > sq->max - 1) sq->rear = 0;      return true;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  bool SQ\_Out(SeqQueue\* sq, T& item)  // 从队列sq出队一个元素，返回时item为出队的元素的值。若出队成功(队列不为空)，则返回true，否则(队列空)，返回false，此时item不会返回有效值。  {      // 请在Begin-End之间补充代码，完成元素出队操作。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (SQ\_IsEmpty(sq)) return false;      item = sq->data[sq->front];      sq->front += 1;      //printf("%d\n", item);      if (sq->front > sq->max - 1) sq->front = 0;      return true;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  bool CLQ\_IsEmpty(LNode\* rear)  // 判断队列是否为空。  {      // 请在Begin-End之间补充代码，完成队列是否为空的判断。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (rear->next == rear) return true;      return false;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  int CLQ\_Length(LNode\* rear)  // 返回队列长度，rear指向尾结点。  {      // 请在Begin-End之间补充代码，获取队列长度。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      int len = 0;      LNode\* p = rear->next;      while (p != rear)      {          len ++ ;          p = p->next;      }      //return len;      return rear->next->data;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  void CLQ\_In(LNode\* & rear, T x)  // 入队列, 新结点加入链表尾部。rear指向尾结点。  {      // 请在Begin-End之间补充代码，完成新结点入队操作。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      LNode\* p = (LNode\*)malloc(sizeof(LNode\*));      p->data = x;      p->next = rear->next;      rear->next = p;      rear = p;      rear->next->data += 1;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  bool CLQ\_Out(LNode\* & rear, T& item)  // 出队列。空队列时，返回值为false。rear指向尾结点。  {      // 请在Begin-End之间补充代码，完成结点出队操作。      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (CLQ\_IsEmpty(rear)) return false;      if (rear->next->data == 1)      {          item = rear->data;          LNode\* p = rear;          rear = p->next;          rear->data -= 1;          rear->next = rear;          free(p);          return true;      }      LNode\* p = rear->next->next;      item = p->data;      rear->next->next = p->next;      rear->next->data -= 1;      free(p);      return true;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  } | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 栈是一种特殊的线性表，仅能在线性表的一端操作，栈顶允许操作，栈底不允许操作。 栈的特点是：先进后出，或者说是后进先出，从栈顶放入元素的操作叫入栈，取出元素叫出栈。  栈的结构就像一个集装箱，越先放进去的东西越晚才能拿出来，所以，栈常应用于实现递归功能方面的场景，例如斐波那契数列。  [队列](https://so.csdn.net/so/search?q=%E9%98%9F%E5%88%97&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_39595164/article/details/_blank)与栈一样，也是一种线性表，不同的是，队列可以在一端添加元素，在另一端取出元素，也就是：先进先出。从一端放入元素的操作称为入队，取出元素为出队，示例图如下：  使用场景：因为队列先进先出的特点，在多线程阻塞队列管理中非常适用。 | |
| **指导教师评语及评分建议：** | |
|  | |

实验四

|  |  |
| --- | --- |
| **实验名称** | 实验四 串与稀疏矩阵 |
| **实验目的和要求 ：** | |
| 实现串与稀疏矩阵的基本操作 | |
| **实验实现思路及步骤：** | |
| 串（或字符串）是由零或多个字符组成的有限序列，记为：s='a1a2...an'(n>=0)  在描述串时，要求串的值必须用一对单引号括起来，但单引号本身不属于串  模式匹配：在当前串中寻找某个子串的过程，该子串称为模式串  特殊矩阵：矩阵中有很多值相同的元素并且他们的分布有一定的规律  稀疏矩阵：矩阵中有很多零元素  稀疏矩阵由三元组表来表示，在第0行可以表示“总体”信息，即总行数，总列数、非零元素总个数 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| #include <cstdio>  #include <cstring>  #include <iostream>  #include <algorithm>  using namespace std;  struct SString  {  char\* data;  int len;  };  void init(SString &s); // 清空串 也是初始化  void setstr(SString &s, char\* ch); // 给串赋值  int getlen(SString s); // 获得串的长度  int SStrcmp(SString s, SString t); // 比较串s和串七  void SStrcat(SString &str, SString s, SString t); // 拼接两个串 返回一个新串  void subStr(SString &t, SString s, int pos, int len); // 求子串  int index(SString s, SString t); // 求子串位置 朴素匹配  void delStr(SString &s, int pos, int len); // 删除串中的某个子串  int main()  {  char ch1[100], ch2[100];  SString s, t;    // 初始化  init(s), init(t);    cout << "分别对串 s 和串 t 进行赋值: " << endl;  scanf("%s%s", ch1, ch2);    puts("");  // 赋值操作  setstr(s, ch1), setstr(t, ch2);    // 输出  cout << "test 输出" << endl;  printf("%s\n%s\n", s.data, t.data);    puts("");  // 测试 获取字符串长度 函数  cout << "test 获取字符串长度" << endl;  cout << "s len: " << getlen(s) << '\n' << "t len: " << getlen(t) << endl;    puts("");  // 测试 字符串比较 函数  cout << "test 字符串比较" << endl;  cout << SStrcmp(s, t) << endl;    puts("");  // 将串清空  cout << "test 清空串 t 并输出被清空的串的内容和长度" << endl;  init(t);  printf("%s\t%d\n", t.data, t.len);    puts("");  init(s);  // 拼接串  cout << "test 拼接串 请输入两个串" << endl;  SString str;  scanf("%s%s", ch1, ch2);  setstr(s, ch1), setstr(t, ch2);  SStrcat(str, s, t);  printf("s: %s, t: %s, s + t: %s\n", s.data, t.data, str.data);    puts("");  // 求子串  cout << "test 求子串 请输入一个串" << endl;  scanf("%s", ch1);  setstr(s, ch1);  cout << "请输入要求子串的位置和长度" << endl;  int pos, len;  cin >> pos >> len;  subStr(t, s, pos, len);  printf("substring: %s\n", t.data);    puts("");  // 字符串匹配  cout << "test 字符串匹配(输出-1 表示未匹配) 请输入要进行匹配的俩个串" << endl;  scanf("%s%s", ch1, ch2);  init(s), init(t);  setstr(s, ch1), setstr(t, ch2);  cout << index(s, t) << endl;    puts("");  // 删除某个子串  cout << "test 删除子串 请输入一个主串" << endl;  scanf("%s", ch1);  init(s), setstr(s, ch1);  cout << "请输入要删除的位置和长度" << endl;  cin >> pos >> len;  delStr(s, pos, len);  printf("%s\n", s.data);    return 0;  }  void init(SString &s)  {  s.data = NULL;  s.len = 0;  }  void setstr(SString &s, char\* ch)  {  int n = strlen(ch);  s.len = n;  s.data = (char\*)malloc(sizeof(char) \* (n + 1));  for (int i = 0; i < n; i ++ )  s.data[i] = ch[i];  s.data[n] = '\0';  }  int getlen(SString s) // 在给字符串赋值的函数中最后设置了一个 '\0'  {  return s.len;  }  int SStrcmp(SString s, SString t)  {  int n = s.len, m = t.len;  for (int i = 0; i < n && i < m; i ++ )  if (s.data[i] != t.data[i])  return s.data[i] - t.data[i];  if (n == m) return 0;  return n - m;  }  void SStrcat(SString& str, SString s, SString t)  {  int n = s.len + t.len;  str.data = (char\*)malloc(sizeof(char) \* (n + 1));  for (int i = 0; i < s.len; i ++ )  str.data[i] = s.data[i];  for (int i = s.len, j = 0; j < t.len; i ++ , j ++ )  str.data[i] = t.data[j];  str.data[n] = '\0';  }  void subStr(SString &t, SString s, int pos, int len)  {  pos = pos - 1;  init(t);  t.data = (char\*)malloc(sizeof(char) \* (len + 1));  int cnt = 0;  for (int i = pos; i <= pos + len - 1; i ++ )  t.data[cnt ++ ] = s.data[i];  t.data[cnt] = '\0';  t.len = cnt;  }  int index(SString s, SString t) // 朴素匹配 未找到返回 -1  {  int n = s.len, m = t.len;  for (int i = 0; i <= n - m; i ++ )  {  int j = 0, k = i;  for ( ; j < m; j ++ , k ++ )  {  if (t.data[j] != s.data[k])  break;  }  if (j == m)  return i + 1;  }  return -1;  }  void delStr(SString &s, int pos, int len)  {  char ch[100];  pos -= 1;  int n = s.len;  int cnt = n - len;  int i = 0;  for (i = 0; i < pos; i ++ )  ch[i] = s.data[i];  if (pos + len - 1 >= n - 1)  {  ch[i] = '\0';  init(s);  setstr(s, ch);  return;  }  for (int j = pos + len; j < n; j ++ )  ch[i ++ ] = s.data[j];  ch[i] = '\0';  init(s);  setstr(s, ch);  }  //稀疏矩阵的三元组顺序表示方法及基本操作的实现（建立、输出、转置）并实现一个主菜单来//实现。  #include <cstdio>  #include <cstring>  #include <iostream>  #include <algorithm>  using namespace std;  const int N = 100;  int q[N][N];  struct Val  {  int r, c, data;  };  struct TSMatris  {  Val data[100];  int mrow;  int mcol;  int cnt;  };  void init(TSMatris &t); // 初始化  void print(TSMatris t); // 解压 打印 输出  void Tran(TSMatris t, TSMatris &tb); // 转置  int main()  {  int cnt, cnt1;  TSMatris t, tb;  cout << "1: 建立三元组; 2: 输出三元组; 3: 转置三元组; 4: 初始化三元组（要更换三元请先初始化）" << endl;  cout << "0: 退出程序" << endl;  int key;  while (cin >> key, key != 0)  {  switch(key)  {  case 4 :  init(t);  init(tb);  cout << "完成初始化" << endl;  break;  case 1 :  cnt = 0, cnt1 = 0;  cout << "请输入三元组的个数" << endl;  cin >> cnt;  while (cnt -- ) // 稀疏矩阵的读入  {  int a, b, c;  cout << cnt1 + 1<<' ' << "input row col value: " << endl;  cin >> a >> b >> c;  t.data[cnt1 ++ ] = {a, b, c};  t.mrow = max(t.mrow, a);  t.mcol = max(t.mcol, b);  t.cnt ++ ;  }  cout << "完成三元组的建立" << endl;  break;  case 2 :  print(t);  cout << "完成输出" << endl;  break;  case 3 :  Tran(t, tb);  t = tb;  cout << "完成转置" << endl;  break;  }  }    return 0;  }  void init(TSMatris &t)  {  t.mcol = t.mrow = 0;  t.cnt = 0;  }  void print(TSMatris t)  {  memset (q, 0, sizeof q);  for (int i = 0; i < t.cnt; i ++ )  {  int x = t.data[i].c - 1, y = t.data[i].r - 1;  q[x][y] = t.data[i].data;  }  for (int i = 0; i < t.mrow; i ++ )  {  for (int j = 0; j < t.mcol; j ++ )  {  printf("%d ", q[i][j]);  }  puts("");  }  }  void Tran(TSMatris t, TSMatris &tb)  {  for (int i = 0; i < t.cnt; i ++ )  {  Val tmp = t.data[i];  tb.data[i] = {tmp.c, tmp.r, tmp.data};  }  tb.mrow = t.mcol;  tb.mcol = t.mrow;  tb.cnt = t.cnt;  } | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 串，即[字符串](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E4%B8%B2&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_46354489/article/details/_blank)，是由零个和多个 字符 组成的有序序列  串中字符个数称为 串的长度，n=0时串称为 空串  子串：串中任意个 连续的字符组成的子序列  主串：包含子串的串  字符在主串的位置：字符在串中的序号  子串在主串中的位置：子串的第一个字符在主串中的位置 | |
| **指导教师评语及评分建议：** | |
|  | |

实验五

|  |  |
| --- | --- |
| **实验名称** | 实验五 二叉树及哈夫曼编码 |
| **实验目的和要求 ：** | |
| 建立二叉树，并实现二叉树遍历的递归和非递归算法，哈夫曼树的建立，哈夫曼编码及译码 | |
| **实验实现思路及步骤：** | |
| 哈弗曼编码  含义  通过对数据重新的编码，减少数据占用的空间存储；使用的时候再进行解压缩，恢复数据的原有特性。  类别  无损压缩——压缩过程没有数据丢失，解压得到原有数据特性。  有损压缩——压缩过程会丢失数据的部分信息，如压缩BMP位图为JPEG会导致精度损失  编码类型  定长编码方案：每个字符的编码长度一样，如ASCII码，128个字符，都是用8位二进制码表示的，最高位为0，每个字符的编码与频率无关；  变长编码方案：每个字符的编码长度与其出现的频率有关，出现的频率越高，其二进制编码的长度越越短；频率越低，则二进制编码长度越长；  最后总数据的编码长度最短，实现压缩数据。 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| BiTreeNode\* CreatBiTree(char\* s, int &i, int len)  // 利用先序遍历创建二叉树  // 参数：先序遍历字符串s，字符串初始下标i=0，字符串长度len。  // 返回：二叉树  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      BiTreeNode\* root = (BiTreeNode\*)malloc(sizeof(BiTreeNode));      char tmp = s[i ++ ];      if (i >= len || tmp == '#')      {          root = NULL;      }      else      {          root->data = tmp;          root->left = CreatBiTree(s, i, len);          root->right = CreatBiTree(s, i, len);      }      return root;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  void InOrder(BiTreeNode\* root)  // 二叉树的中序遍历  // 参数：二叉树根节点root  // 输出：中间没有空格，末尾不换行。  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (root)      {          InOrder(root->left);          printf("%c", root->data);          InOrder(root->right);      }      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  BiTreeNode\* BiTreeChange(BiTreeNode\* root)  // 实现二叉树左右子树的交换（递归法）  // 参数：二叉树根节点root  // 返回：二叉树  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (root == NULL) return NULL;      BiTreeNode\* tmp = root->left;      root->left = root->right;      root->right = tmp;      if (root->left != NULL)          BiTreeChange(root->left);      if (root->right != NULL)          BiTreeChange(root->right);        return root;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  void PreOrder(BiTreeNode\* root)  // 二叉树的前序遍历  // 参数：二叉树根节点root  // 输出：二叉树的前序遍历，中间没有空格，末尾不换行。  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (root)      {          printf("%c", root->data);          PreOrder(root->left);          PreOrder(root->right);      }      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  BiTreeNode\* BiTreeChangeStack(BiTreeNode\* root)  // 实现二叉树左右子树的交换（栈实现）  // 参数：二叉树根节点root  // 返回：二叉树  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      stack<BiTreeNode\*> st;      BiTreeNode\* tmp = NULL;      if (root == NULL) return NULL;      st.push(root);      BiTreeNode\* p = root;      while (!st.empty())      {          tmp = p->left;          p->left = p->right;          p->right = tmp;          if (p->right != NULL)              st.push(p->right);          if (p->left != NULL)              p = p->left;          else          {              p = st.top();              st.pop();          }      }        return root;      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  void PostOrder(BiTreeNode\* root)  // 二叉树的后序遍历  // 参数：二叉树根节点root  // 输出：二叉树的后序遍历，中间没有空格，末尾不换行。  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      if (root)      {          PostOrder(root->left);          PostOrder(root->right);          printf("%c", root->data);      }        /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  void HierarchyOrder(BiTreeNode\* root)  // 二叉树的层次遍历（队列实现）  // 参数：二叉树根节点root  // 输出：二叉树的层次遍历，中间没有空格，末尾不换行。  {      // 请在这里补充代码，完成本关任务      /\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/      queue<BiTreeNode\*> q;      BiTreeNode\* tmp = NULL;      q.push(root);      while (!q.empty())      {          tmp = q.front();          q.pop();          printf("%c", tmp->data);          if (tmp->left != NULL)              q.push(tmp->left);          if (tmp->right != NULL)              q.push(tmp->right);      }      /\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/  }  假设用于通信的电文仅为a,b,c,d,e,f,g,h等8个字母组成，字母在电文中出现的频率分别为0.07, 0.19, 0.02, 0.06, 0.32, 0,03, 0.21, 0.1。试为这8个字母设计哈夫曼树及哈夫曼编码，并计算带权路径长度WPL。  IMG_256 | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 构造哈夫曼树  规则就是  1.在频率集合中找出字符中最小的两个；小的在左边，大的在右边；这两个兄弟组成二叉树。他们的双亲为他们的频率（权值）之和。  2.在频率表中删除此次找到的两个数，并加入此次最小两个数的频率和（把他们的双亲加入，然后排序）。 | |
| **指导教师评语及评分建议：** | |
|  | |

实验六

|  |  |
| --- | --- |
| **实验名称** | 实验六 图的建立与遍历 |
| **实验目的和要求 ：** | |
| 实现图的建立 、利用图的遍历、最短路径等重要算法解决具体应用 | |
| **实验实现思路及步骤：** | |
| 深度优先遍历：以一个顶点为起点入栈，如果该点没有指向的点或指向的点都被访问过则出栈，如果有则将该点入栈；然后继续以栈顶元素为顶点重复上述操作，直至栈空为止。  广度优先遍历：以一个顶点为起点入队，然后队首元素出队，将刚刚出队的元素所有指向且未被访问的点都入队，然后继续对队首元素进行操作，直至队列为空。 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| void Creat\_ALG(ALGraph \*m){      VertexType a,b;      for(int i=0;i<m->arcnum;i++){          scanf("%d%d",&a,&b);          /\*\*\*\*\*\*\*\*\*\*\*\*begin\*\*\*\*\*\*\*\*\*\*\*\*/          ArcNode\* p = (ArcNode\*)malloc(sizeof(ArcNode));          p->num = b - 1;          p->next = NULL;          ArcNode\* tmp = m->vertices[a - 1].firstarc;          if (tmp == NULL)          {              m->vertices[a - 1].firstarc = p;          }          else          {              while (tmp->next != NULL) tmp = tmp->next;              tmp->next = p;          }          /\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*/      }  }  void dfs(stack<int> &s,ALGraph m,int visit[]){      /\*\*\*\*\*\*\*\*\*\*\*\*\*begin\*\*\*\*\*\*\*\*\*\*\*\*\*/      int v = s.top();      if (cnt1 != 0) printf(" ");      cnt1 ++ ;      printf("%d", m.vertices[v].data);      ArcNode\* p = m.vertices[v].firstarc;  // 第一个指向的节点      bool flag = false;      ArcNode\* p1 = p;      if (p == NULL)  // 没有指向的节点      {          s.pop();          return;      }      if (!visit[p1->num]) flag = true;  // 如果有没有访问过的节点设置为true      // 有指向的节点 看看指向的节点是否都被访问过      while (p1->next != NULL)      {          p1 = p1->next;          if (!visit[p1->num]) flag = true;      }      if (flag == false)  // 该节点连接的节点都访问过了      {          s.pop();          return;      }      // 能到这一步一定是有连接的节点 而且连接的节点有没有被访问过的      while (p != NULL)      {          int temp = p->num;          if (!visit[temp])          {              visit[temp] = 1;              s.push(temp);              dfs(s, m, visit);          }          p = p->next;      }      /\*\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*\*/  }  int cnt2 = 0;  void bfs(queue<int> &q,ALGraph m,int visit[]){        /\*\*\*\*\*\*\*\*\*\*\*\*\*begin\*\*\*\*\*\*\*\*\*\*\*\*\*/      int n = m.vexnum;      while (!q.empty())      {          int v = q.front();          q.pop();          cnt2 ++ ;          printf("%d", m.vertices[v].data);          if (cnt2 < n) printf(" ");          ArcNode\* p = m.vertices[v].firstarc;          while (p != NULL)          {              int temp = p->num;              if (!visit[temp])              {                  visit[temp] = 1;                  q.push(temp);              }              p = p->next;          }      }      /\*\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*\*/  }  void FindInDegree(ALGraph G,int a[]){      /\*\*\*\*\*\*\*\*\*\*\*\*\*begin\*\*\*\*\*\*\*\*\*\*\*\*\*/      int n = G.vexnum;      for (int i = 0; i < n; i ++ )      {          ArcNode\* p = G.vertices[i].firstarc;          if (p == NULL)              continue;          else          {              while (p != NULL)              {                  a[p->num] ++ ;                  p = p->next;              }          }      }        /\*\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*\*/  } | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 深度优先遍历4部曲(dfs):  先访问给定下标为v的顶点  再标记已访问顶点下标  遍历已访问顶点的边表  边表的顶点下标未曾被标记,则递归调用dfs(G,v)  广度优先遍历6部曲(bfs) :  先访问给定顶点下标v的顶点  再标记下标为v的顶点  接着将下标为v的顶点入队  然后遍历下标为v的顶点的边表  随后将边表中未曾标记的顶点依次入队  一个一个出队,重复执行以上步骤 | |
| **指导教师评语及评分建议：** | |
|  | |

实验七

|  |  |
| --- | --- |
| **实验名称** | 实验七 查找 |
| **实验目的和要求 ：** | |
| 实现各种检索算法并解决实际问题 | |
| **实验实现思路及步骤：** | |
| 折半查找通常是针对顺序存储的线性表，线性表的结点按关键码从小到大排序，后面称之为折半查找的顺序表。为了简化讨论，假设折半查找的顺序表中每个结点只含一个关键码，关键码为整数。 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| int BSL\_FindKey(BSeqList\* blist, int key)  //在排序的顺序表中查找关键码值为key的结点，返回结点的编号  //返回值大于等于0时表示找到值为key的结点的编号，-1表示没有找到  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      int len = blist->len;      if (len >= blist->max)          return -1;        int l = 0, r = blist->len - 1;      while (l <= r)      {          int mid = (l + r) >> 1;          if (blist->pkey[mid] == key) return mid;          else if (blist->pkey[mid] > key) r = mid - 1;          else l = mid + 1;      }      return -1;      /\*\*\*\*\*\*END\*\*\*\*\*\*/      /\*请不要修改[BEGIN,END]区域外的代码\*/  }  bool ILH\_InsKey(LHTable\* pt, int x)  //插入关键码x  //返回true，表示插入成功  //返回false，表示插入失败(关键码已经存在)  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      if (ILH\_FindKey(pt, x)) return false;      int mod = pt->n;      int m = x % mod;      HNode\* tmp = (HNode\*)malloc(sizeof(HNode));      tmp->key = x;      tmp->next = NULL;      HNode\* s = pt->pn[m].next;      if (pt->pn[m].key == 0)      {          pt->pn[m].key = x;          return false;      }      else      {          if (s == NULL)          {              pt->pn[m].next = tmp;              return true;          }          else          {              while (s->next)              {                  s = s->next;              }              s->next = tmp;          }          return true;      }      /\*\*\*\*\*\*END\*\*\*\*\*\*/      /\*请不要修改[BEGIN,END]区域外的代码\*/  }  bool ILH\_DelKey(LHTable\* pt, int x)  //删除关键码  //返回true表示该关键码存在，且成功删除  //返回false表示该关键码不存在  {      /\*请在BEGIN和END之间实现你的代码\*/      /\*\*\*\*\*BEGIN\*\*\*\*\*/      HNode\* p;      if (ILH\_FindKey(pt, x))      {          int mod = pt->n;          int m = x % mod;          if (pt->pn[m].key == x)          {              pt->pn[m].key = 0;              return false;          }          else          {              HNode\* tmp = pt->pn[m].next;              if (tmp->key == x)              {                  pt->pn[m].next = tmp->next;                  return true;              }              else              {                  while (tmp->key != x)                  {                      tmp = tmp->next;                  }                  HNode\* p = tmp;                  tmp = p->next;                  free(p);                  return true;              }          }          return false;      }      /\*\*\*\*\*\*END\*\*\*\*\*\*/      /\*请不要修改[BEGIN,END]区域外的代码\*/  } | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 散列查找的平均查找长度影响因素有：  1.散列函数是否均匀，散列函数的好坏直接决定着出现冲突的频繁程度。  2.处理冲突的方法，处理方法不同，也会影响平均查找长度。比线性探测处理冲突可能会产生堆积，不如二次探测法；而链地址探测法不会产生任何堆积。  3.散列表的装填因子。所谓的装填因子α=填入表中的记录个数/散列表长度。α越大则产生冲突的可能性就越大。通常将散列表的空间设置得比查找集合大，虽然浪费了一定的空间，但是得到了较高的查找效率。 | |
| **指导教师评语及评分建议：** | |
|  | |

实验八

|  |  |
| --- | --- |
| **实验名称** | 实验八 排序 |
| **实验目的和要求 ：** | |
| 实现各种内部排序算法并解决实际问题 | |
| **实验实现思路及步骤：** | |
| 冒泡排序重复地遍历待排序的数列，每次比较两个相邻元素，如果它们的顺序错误就把它们交换。重复地进行遍历直到没有再需要交换时表示数列已经排序完成。  选择排序是一种简单直观的排序算法，首先在未排序序列中找到最小元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。  插入排序的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。  归并排序（MERGE-SORT）是利用归并的思想实现的排序方法，是采用分治法Divide and Conquer的一个非常典型的应用。分Divide：将问题分成一些小的问题然后递归求解；治Conquer：将分的阶段得到的各答案合并在一起。  快速排序是最常用的一种排序算法，它的特点是速度快、效率高。快速排序的基本思想：选择一个关键值作为基准值。比基准值小的都在左边序列（一般是无序的），比基准值大的都在右边（一般是无序的）。一般选择序列的第一个元素作为基准值。  排序Heapsort是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。  计数排序不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。 | |
| **主要开发工具** | Dev C++ |
| **实验效果及主要实现代码：** | |
| // 编写简单选择排序算法。  // void SelectSort(Rectype R[ ], int n)  void SelectSort(int r[ ], int n)  {  int min;  for (int i = 1; i < n - 1; i ++ )  {  min = i;  for (int j = i + 1; j < n; j ++ )  if (r[j] < r[min])  min = j;    if (min != i)  {  r[0] = r[i];  r[i] = r[min];  r[min] = r[0];  }  }  }  编写直接插入排序算法。  void  InsertSort(Rectype R[ ], int n)  void InsertSort(int r[], int n)  {  for (int i = 2; i <= n; i ++ )  {  r[0] = r[i];  int j = i - 1;    while (r[j] > r[0])  {  r[j + 1] = r[j];  j -- ;  }    r[j + 1] = r[0];  }  }  编写冒泡排序算法。  void BubbleSort(Rectype R[ ], int n)  void BubbleSort(int r[], int n)  {  for (int i = 1; i < n; i ++ )  {  int swap = 0;  for (int j = 1; j < n - i; j ++ )  {  if (r[j] > r[j + 1])  {  r[0] = r[j + 1];  r[j + 1] = r[j];  r[j] = r[0];  swap = 1;  }  }    if (swap == 0) break;  }  }  编写快速排序算法。  void QuickSort(Rectype R[ ], int left, int right)  void QuickSort(int q[], int l, int r)  {  if (l >= r) return;    int x = q[l + r >> 1];  int i = l - 1, j = r + 1;  while (i < j)  {  do i ++ ; while (q[i] < x);  do j -- ; while (q[j] > x);  if (i < j) swap(q[i], q[j]);  }    QuickSort(q, l, j);  QuickSort(q, j + 1, r);  } | |
| **遇到的问题和解决方法：** | |
| 要到调试不通的bug。  通过搜索引擎搜索，若果还不能解决多设计几组测试用例通过debug来查找错误。 | |
| **心得体会：** | |
| 1.排序算法的稳定性：如果序列中两个元素相等，排序后两个元素的相对顺序不变，则称所用的排序方法是稳定的，反之，是非稳定的。（有人会说，两个元素相等为什么还考虑相对顺序？ 在单一属性元素的排序中确实不重要，但在结构体数据的排序中，算法的稳定性很重要），一般来说，若存在不相邻元素间交换，则很可能是不稳定的排序。  2.算法的复杂度：一个算法的质量优劣将影响到算法乃至程序的效率，主要从时间复杂度和空间复杂度两个维度考虑。  （算法的复杂度一般表示的是平均复杂度，但有的算法平均复杂度不好求取，可以求取最坏情况时的复杂度和最好情况时的复杂度，两者平均近似平均复杂度，时间复杂度分析主要分析比较次数和移动次数）  3.排序算法分类  排序原则分类：内部排序算法主要包括 插入排序、交换排序、选择排序、归并排序和基数排序等五类  排序时间复杂度分类：1）简单的排序方法,时间复杂度O(n2)  2）先进的排序方法,时间复杂度O(nlogn)  3）基数排序,时间复杂度O(d\*n) | |
| **指导教师评语及评分建议：** | |
|  | |