中南大学

数据结构试验报告

题 目 实验四

学生姓名 王云鹏

学 号 8213180228

指导老师 郑 瑾

学 院 计算机学院

专业班级 物联网1802

完成时间 2020.6

指导老师评定 签名

实验四

1. 需求分析

本实验继续突出了数据结构加操作的程序设计观点，但根据树结构的非线性特点，将操作进一步

集中在遍历操作上，因为遍历操作是其他众多操作的基础。本实验还希望达到使读者熟悉各种存储结

构的特征，以及掌握如何应用树解决具体问题（即原理与应用的结合）等目的。

1．二叉树的建立与遍历（验证性实验）

问题描述

建立一棵二叉树，并对其进行遍历（先序、中序和后序），打印输出遍历结果。

基本要求

从键盘接受输入（先序），以二叉链表作为存储结构，建立二叉树（以先序来建立），并采用递归

算法对其进行遍历（先序、中序和后序），将遍历结果打印输出。

测试数据

*ABC*□□*DE*□*G*□□*F*□□□（其中□表示空格字符），则输出结果为：先序为*ABCDEGF*，中序为

*CBEGDFA*，

z

后序为*CGBFDBA*。

3．打印二叉树结构（设计性实验）

问题描述

按凹入表形式横向打印二叉树结构，即二叉树的根在屏幕的最左边，二叉树的左子树在屏幕的下

边，二叉树的右子树在屏幕的上边。例如：

*A*

*C*

*B*

*D*

*E*

*F*

*C*

*F*

*E*

*A*

*D*

*B*

测试数据

由读者依据软件工程的测试技术自己确定。注意测试边界数据，如空二叉树。

实现提示

(1) 利用中序遍历方法。

(2) 利用结点的深度控制横向位置。

4．赫夫曼编码（综合性实验）

问题描述

设某编码系统共有*n*个字符，使用频率分别为*w*1, *w*2, …, *wn*，设计一个不等长的编码方案，使得该

编码系统的空间效率最好。

基本要求

一个完整的系统应具有以下功能。

(1) I：初始化（Initialization）。从终端读入字符集大小*n*，以及*n*个字符和*n*个权值，建立赫夫曼树，

并将它存于文件hfmTree中。

(2) E：编码（Encoding）。利用已建好的赫夫曼树对文件ToBeTran中的正文进行编码，然后将结果

存入文件CodeFile中。

(3) D：译码（Decoding）。利用已建好的赫夫曼树将文件CodeFile中的代码进行译码，结果存入文

件TextFile中。

(4) P：打印代码文件（Print）。将文件CodeFile以紧凑格式显示在终端上，每行50个字符。同时将

此字符形式的编码文件写入文件CodePrint中。

(5) T：打印赫夫曼树（Tree printing）。将已在内存中的赫夫曼树以直观的方式显示在终端上，同

时将此字符形式的赫夫曼树写入文件TreePrint中。

测试数据

由读者依据软件工程的测试技术自己确定。注意测试边界数据。

实现提示

利用赫夫曼编码树求得最佳的编码方案。

(1) 文件CodeFile的基类型可以设为字节型。

(2) 用户界面可以设计为“菜单”方式，除显示上述功能符号外，还应显示“Q”（Quit），表示退

出运行。请用户键入一个选择功能符。此功能执行完毕后再显示此菜单，直至某次用户选择了“E”

为止。

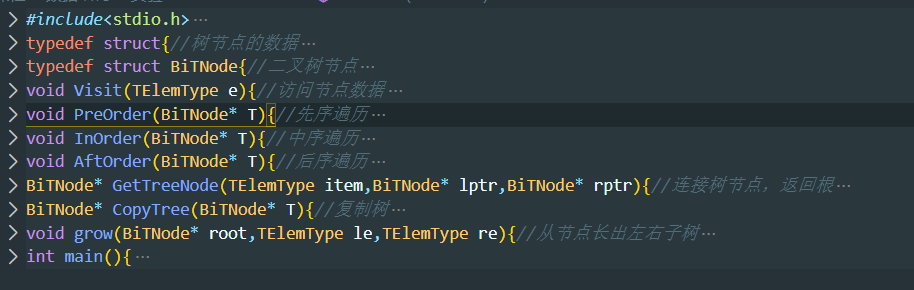
(3) 在程序的一次执行过程中，第一次执行I、D或C命令之后，赫夫曼树已经在内存了，不必再

读入。每次执行时不一定执行I命令，因为文件hfmTree可能早已建好。

1. 概要设计

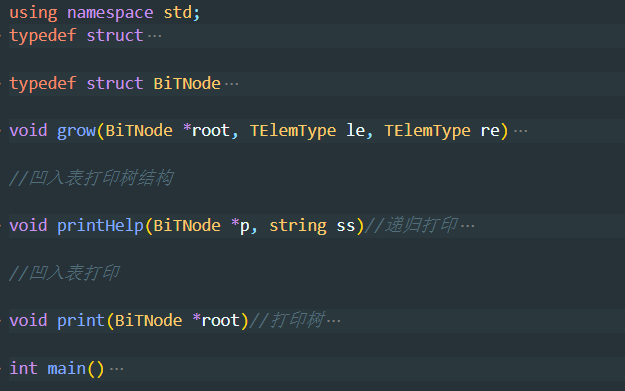
* 验证性实验

函数：



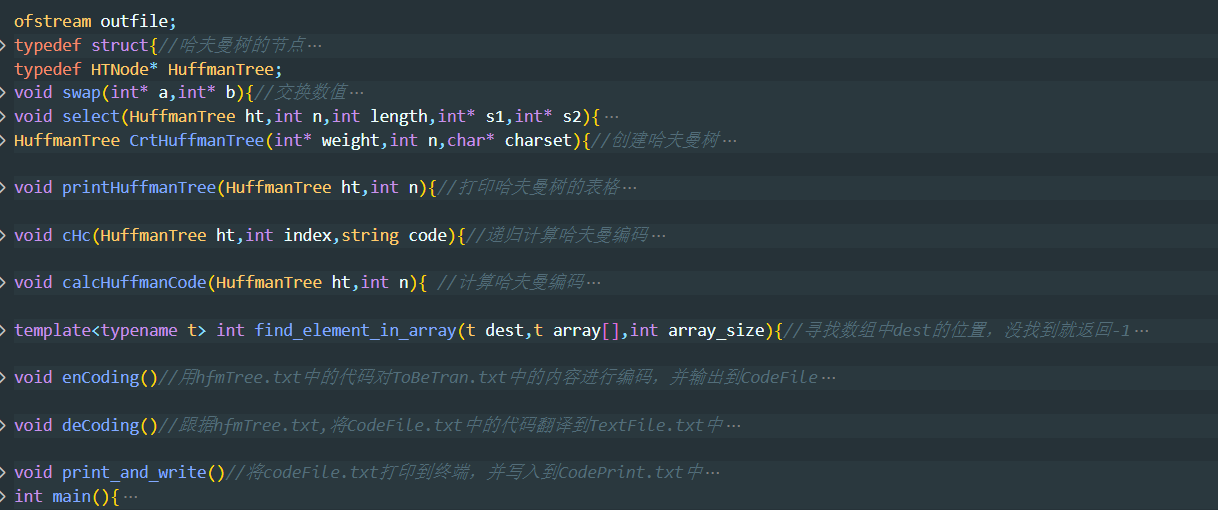
* 设计性实验

函数：



* 综合性实验

函数：



1. 详细设计

* 验证性实验

|  |
| --- |
| *#include*<stdio.h>  *#include*<stdlib.h>  typedef struct{*//树节点的数据*      char data;  }TElemType;  typedef struct BiTNode{*//二叉树节点*      TElemType data;      struct BiTNode \*lchild,\*rchild;*//左右孩子节点*  }BiTNode;  void Visit(TElemType e){*//访问节点数据*      printf("%c\n",e.data);  }  void PreOrder(BiTNode\* T){*//先序遍历*  *if*(T==NULL){*//空则返回*  *return*;      }      Visit(T->data);*//先访问根节点*      PreOrder(T->lchild);*//再以先序访问左子树*      PreOrder(T->rchild);*//最后以先序访问右子树*  }  void InOrder(BiTNode\* T){*//中序遍历*  *if*(T==NULL){  *return*;      }      InOrder(T->lchild);      Visit(T->data);      InOrder(T->rchild);  }  void AftOrder(BiTNode\* T){*//后序遍历*  *if*(T==NULL){  *return*;      }      AftOrder(T->lchild);      AftOrder(T->rchild);      Visit(T->data);  }  BiTNode\* GetTreeNode(TElemType item,BiTNode\* lptr,BiTNode\* rptr){*//连接树节点，返回根*      BiTNode\* T=(BiTNode\*)malloc(sizeof(BiTNode));*//根*  *if*(T==NULL){          printf("malloc fail in GetTreeNode for T\n");  *return* NULL;      }      T->data=item;*//根的数据*      T->lchild=lptr;      T->rchild=rptr;  *return* T;  }  BiTNode\* CopyTree(BiTNode\* T){*//复制树*  *if*(T==NULL){*//空则返回*  *return* NULL;      }      BiTNode\* newT=(BiTNode\*)malloc(sizeof(BiTNode));*//根*  *if*(newT==NULL){          printf("malloc fail in CopyTree for newT\n");  *return* NULL;      }      BiTNode\* newlptr=(BiTNode\*)malloc(sizeof(BiTNode));*//左子*  *if*(newlptr==NULL){          printf("malloc fail in CopyTree for newlptr\n");  *return* NULL;      }      BiTNode\* newrptr=(BiTNode\*)malloc(sizeof(BiTNode));*//右子*  *if*(newrptr==NULL){          printf("malloc fail in CopyTree for newrptr");  *return* NULL;      }  *if*(T->lchild!=NULL){*//左子树不为空就复制左子树*          newlptr=CopyTree(T->lchild);      }  *else*{          newlptr=NULL;*//否则置空*      }  *if*(T->rchild!=NULL){          newrptr=CopyTree(T->rchild);*//as*      }  *else*{          newrptr=NULL;*//as*      }      newT=GetTreeNode(T->data,newlptr,newrptr);*//将复制好的左子树和右子树连接到根节点上*  *return* newT;  }  void grow(BiTNode\* root,TElemType le,TElemType re){*//从节点长出左右子树*      root->lchild=(BiTNode\*)malloc(sizeof(BiTNode));*//左子*  *if*(root->lchild==NULL){          printf("malloc fail in grow for root->lchild\n");  *return*;      }      root->rchild=(BiTNode\*)malloc(sizeof(BiTNode));*//右子*  *if*(root->rchild==NULL){          printf("malloc fail in grow for root->lchild\n");  *return*;      }      root->lchild->data=le;*//le为左子的数据*      root->lchild->lchild=NULL;      root->lchild->rchild=NULL;      root->rchild->data=re;*//re为右子的数据*      root->rchild->lchild=NULL;      root->rchild->rchild=NULL;  }  int main(){      BiTNode root;*//树根*      TElemType rootE={'r'};      root.data=rootE;      root.lchild=NULL;      root.rchild=NULL;      TElemType a={'a'};*//测试*      TElemType b={'b'};      TElemType c={'c'};      TElemType d={'d'};      TElemType e={'e'};      TElemType f={'f'};      grow(&root,a,b);      grow(root.lchild,c,d);      grow(root.rchild,e,f);      printf("PreOrder\n");      PreOrder(&root);      printf("InOrder\n");      InOrder(&root);      printf("AftOrder\n");      AftOrder(&root);      printf("PreOrder of CopyTree\n");      PreOrder(CopyTree(&root));  *return* 0;  } |

* 设计性实验

|  |
| --- |
| *#include*<iostream>  using namespace std;  typedef struct  {*//树节点的数据*      char data;  } TElemType;  typedef struct BiTNode  {*//二叉树节点*      TElemType data;      struct BiTNode \*lchild, \*rchild;*//左右孩子节点*  } BiTNode;  void grow(BiTNode \*root, TElemType le, TElemType re)  {*//从节点长出左右子树*      root->lchild = (BiTNode \*)malloc(sizeof(BiTNode));*//左子*  *if* (root->lchild == NULL)      {          printf("malloc fail in grow for root->lchild\n");  *return*;      }      root->rchild = (BiTNode \*)malloc(sizeof(BiTNode));*//右子*  *if* (root->rchild == NULL)      {          printf("malloc fail in grow for root->lchild\n");  *return*;      }      root->lchild->data = le;*//le为左子的数据*      root->lchild->lchild = NULL;      root->lchild->rchild = NULL;      root->rchild->data = re;*//re为右子的数据*      root->rchild->lchild = NULL;      root->rchild->rchild = NULL;  }  *//凹入表打印树结构*  void printHelp(BiTNode \*p, string ss)*//递归打印*  {  *if* (p == NULL)  *return*;      ss += "   ";*//每往下一层就加一个空位*      printHelp(p->rchild, ss);      cout << ss << p->data.data << endl;*//中序遍历*      printHelp(p->lchild, ss);  }  *//凹入表打印*  void print(BiTNode \*root)*//打印树*  {      string ss = "";      printHelp(root, ss);  }  int main()  {      BiTNode root;*//树根*      TElemType rootE = {'r'};      root.data = rootE;      root.lchild = NULL;      root.rchild = NULL;      TElemType a = {'a'};*//测试*      TElemType b = {'b'};      TElemType c = {'c'};      TElemType d = {'d'};      TElemType e = {'e'};      TElemType f = {'f'};      grow(&root, a, b);      grow(root.lchild, c, d);      grow(root.rchild, e, f);      print(&root);  *return* 0;  } |

|  |
| --- |
| *#include*<iostream>  *#include*<io.h>  *#include*<string>  *#include*<fstream>  using namespace std;  ofstream outfile;  typedef struct{*//哈夫曼树的节点*      char ch;      int weight;      int parent,lchild,rchild;  }HTNode;  typedef HTNode\* HuffmanTree;  void swap(int\* a,int\* b){*//交换数值*      int temp=\*a;      \*a=\*b;      \*b=temp;  }  void select(HuffmanTree ht,int n,int length,int\* s1,int\* s2){  *//在ht［1］~ht［i-1］的范围内选择两个parent为0且weight最小的结点*  *//其序号分别赋值给s1、 s2返回*  *//先将parent为0的挑出其下标组成数组，按weight排序，取前两位*      int\* index=(int\*)malloc((2\*n-length)\*sizeof(int));*//数组*      int p=0;  *for*(int i=1;i<=length;i++){*//将parent为0的挑出其下标组成数组*  *if*(ht[i].parent==0){              index[p++]=i;          }      }  *for*(int i=0;i<2\*n-length;i++){*//选择排序*  *for*(int j=i+1;j<2\*n-length;j++){  *if*(ht[index[i]].weight>ht[index[j]].weight){                  swap(&index[i],&index[j]);              }          }      }      \*s1=index[0];      \*s2=index[1];  }  HuffmanTree CrtHuffmanTree(int\* weight,int n,char\* charset){*//创建哈夫曼树*      HuffmanTree ht;      int ArrayLength=2\*n-1;*//树长*      ht=(HuffmanTree)malloc((ArrayLength+1)\*sizeof(HTNode));  *for*(int i=0;i<=n;i++){*//叶子节点初始化*          ht[i].ch=charset[i];          ht[i].weight=weight[i];          ht[i].parent=0;          ht[i].lchild=0;          ht[i].rchild=0;      }  *for*(int i=n+1;i<=ArrayLength;i++){*//其他节点初始化*          ht[i].ch=0;          ht[i].weight=0;          ht[i].parent=0;          ht[i].lchild=0;          ht[i].rchild=0;      }      int s1=0;      int s2=0;  *for*(int i=n+1;i<=ArrayLength;i++){          select(ht,n,i-1,&s1,&s2);*//挑出weight最小的两个下标*  *// printf("%d %d %d\n",i,s1,s2);*          ht[s1].parent=i;*//叶子的父节点*          ht[s2].parent=i;          ht[i].lchild=s1;*//父节点的孩子节点*          ht[i].rchild=s2;          ht[i].weight=ht[s1].weight+ht[s2].weight;*//父节点的weight*      }  *return* ht;  }  void printHuffmanTree(HuffmanTree ht,int n){*//打印哈夫曼树的表格*      printf("char\tweight\tparent\tlchild\trchild\n");  *for*(int i=1;i<=2\*n-1;i++){          printf("%c\t%d\t%d\t%d\t%d\n",ht[i].ch,ht[i].weight,ht[i].parent,          ht[i].lchild,ht[i].rchild);      }  }  void cHc(HuffmanTree ht,int index,string code){*//递归计算哈夫曼编码*  *if*(ht[index].lchild==0 && ht[index].rchild==0){*//到达树叶就输出*  *// cout<<ht[index].ch<<"\t"<<code<<endl;*          outfile<<ht[index].ch<<"\t"<<code<<endl;  *return*;      }      cHc(ht,ht[index].lchild,code+"0");*//左边加0*      cHc(ht,ht[index].rchild,code+"1");*//右边加1*  }  void calcHuffmanCode(HuffmanTree ht,int n){*//计算哈夫曼编码*      string left="0";      string right="1";      cHc(ht,ht[2\*n-1].lchild,left);*//左边0*      cHc(ht,ht[2\*n-1].rchild,right);*//右边1*  }  template<typename t> int find\_element\_in\_array(t dest,t array[],int array\_size){*//寻找数组中dest的位置，没找到就返回-1*  *for*(int i=0;i<array\_size;i++){  *if*(dest==array[i]){  *return* i;          }      }      cout<<"did not find char in array"<<endl;  *return* -1;  }  void enCoding()*//用hfmTree.txt中的代码对ToBeTran.txt中的内容进行编码，并输出到CodeFile*  {  *//打开文件*      ifstream hfmTree;      hfmTree.open("./hfmTree.txt",ios::in);      ifstream ToBeTran;      ToBeTran.open("./ToBeTran.txt",ios::in);      ofstream codeFile;      codeFile.open("./CodeFile.txt",ios::out);      char ch[36];      string ch\_code[36];      int i=0;  *while*(!hfmTree.eof()){          hfmTree>>ch[i]>>ch\_code[i];*//从文件中读取编码到数组*          i++;      }      char temp;  *while*(!ToBeTran.eof()){          ToBeTran>>temp;  *if*(find\_element\_in\_array(temp,ch,sizeof(ch)/sizeof(char))==-1)*continue*;          codeFile<<ch\_code[find\_element\_in\_array(temp,ch,sizeof(ch)/sizeof(char))];*//根据数组中的编码，将字符替换为01存入文件*      }      hfmTree.close();*//关闭文件*      ToBeTran.close();      codeFile.close();  }  void deCoding()*//跟据hfmTree.txt,将CodeFile.txt中的代码翻译到TextFile.txt中*  {      ifstream hfmTree;*//打开文件*      hfmTree.open("./hfmTree.txt",ios::in);      ifstream codeFile;      codeFile.open("./CodeFile.txt",ios::in);      ofstream textFile;      textFile.open("./TextFile.txt",ios::out);      char ch[36];      string ch\_code[36];      int i=0;  *while*(!hfmTree.eof()){          hfmTree>>ch[i]>>ch\_code[i];*//从文件读入编码*          i++;      }      string code;      getline(codeFile,code);      string temp;      temp=code[0];      int j=0;  *while*(true){*//根据数组中的编码，将01..译为字符*  *if*(j>=code.length())*break*;  *if*(find\_element\_in\_array(temp,ch\_code,sizeof(ch\_code)/sizeof(string))==-1){              temp=temp+code[++j];          }*else*{              textFile<<ch[find\_element\_in\_array(temp,ch\_code,sizeof(ch\_code)/sizeof(string))];              temp=code[++j];          }      }      hfmTree.close();*//关闭文件*      codeFile.close();      textFile.close();  }  void print\_and\_write()*//将codeFile.txt打印到终端，并写入到CodePrint.txt中*  {      ifstream codeFile;*//打开文件*      codeFile.open("./CodeFile.txt",ios::in);      ofstream codePrint;      codePrint.open("./CodePrint.txt",ios::out);      string code;      getline(codeFile,code);*//读入编码*  *for*(int i=0;i<code.length();i++){          cout<<code[i];          codePrint<<code[i];  *if*((i+1)%50==0){*//每输出50个字符，换行*              cout<<endl;              codePrint<<endl;          }      }  }  int main(){  *// int weight[]={0,7,5,2,4};*  *// int n=4;*      int n=0;      ifstream source;      source.open("./source.txt",ios::in);*//打开要编码的测试文件*      source>>n;  *// cout<<"请输入要编码的字符个数"<<endl;*  *// cin>>n;*      int weight[n+1];      weight[0]=0;      char charset[n+1];      charset[0]=0;  *// cout<<"请输入要编码的字符和其权重"<<endl;*  *for*(int i=1;i<n+1;i++){          source>>charset[i]>>weight[i];  *// cin>>charset[i]>>weight[i];*      }      source.close();      HuffmanTree ht=CrtHuffmanTree(weight,n,charset);*//创建哈夫曼树*      printHuffmanTree(ht,n);      outfile.open("./hfmTree.txt",ios::out);      calcHuffmanCode(ht,n);*//计算哈夫曼码*      outfile.close();      enCoding();*//测试*      deCoding();      print\_and\_write();  *return* 0;  } |

* 综合性实验

4.调试分析

1. 采用IDE中自带的调试功能进行调试，手动添加断点和查看程序。
2. 对设计和编码的讨论和分析。该程序实现了顺序栈的操作。分析程序代码的质量，主要从以下几个方面考虑。

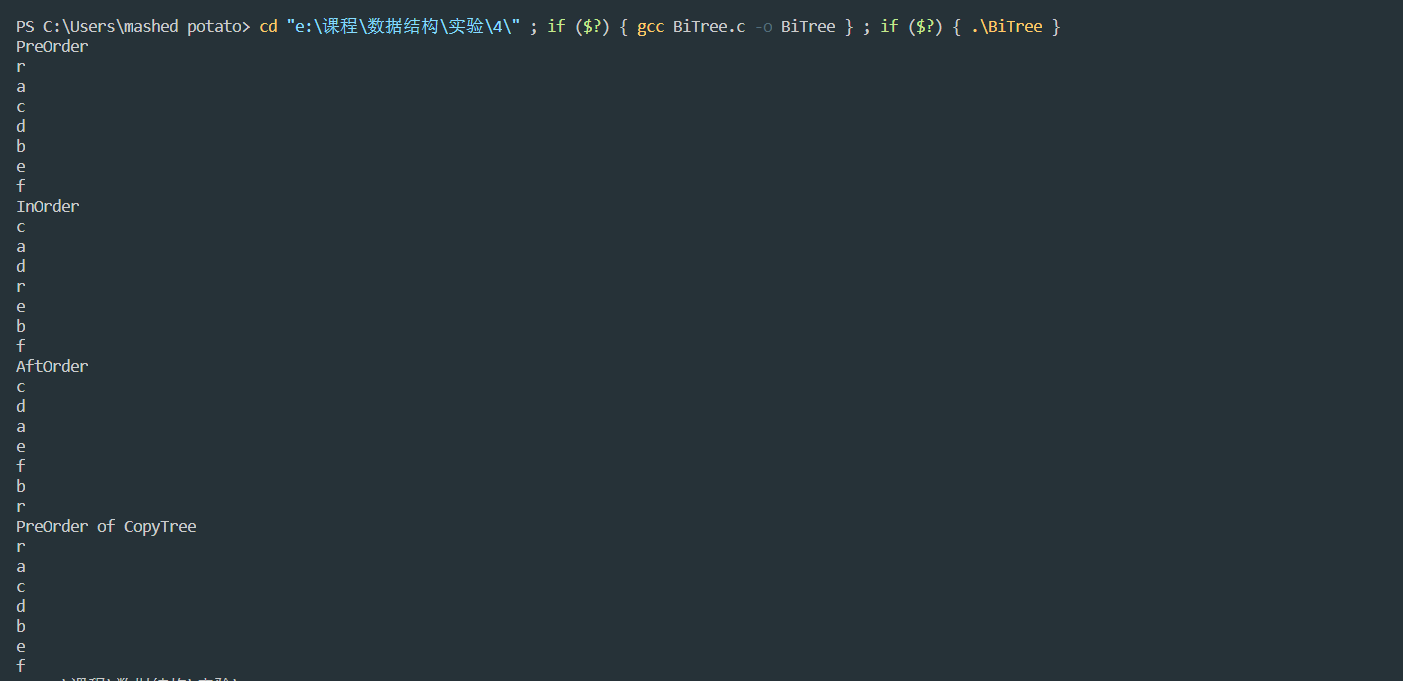
* 正确性。在一定的数据范围内，该程序能实现所需功能，所以正确性是没有问题的。
* 健壮性。在一定的数据输入范围内，该程序能较好的实现操作。但是如果输入数 据非法，该程序还是可能会产生一些预想不到的输出结构，或是不做任何处理。所以， 该程序的健壮性有待进一步的提高。要综合考虑一些情况，当输入有误时，应返回一个 表示错误的值，并中止程序的执行，以便在更高的抽象层次上进行处理。

5.使用说明

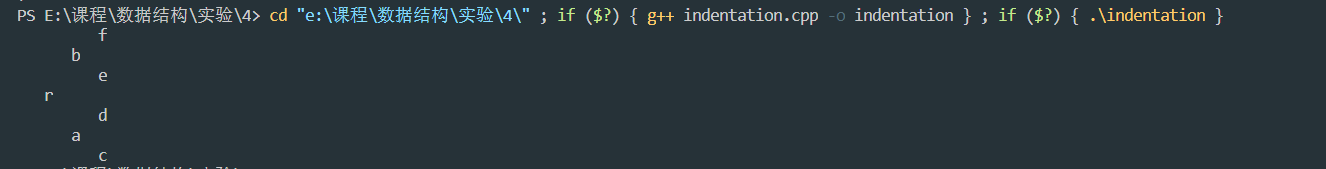
按照屏幕提示，选择想要的功能并输入对应数字，按下ENTER键后，根据屏幕提示进行输入，即可得到想要的结果。

6.测试程序运行结果

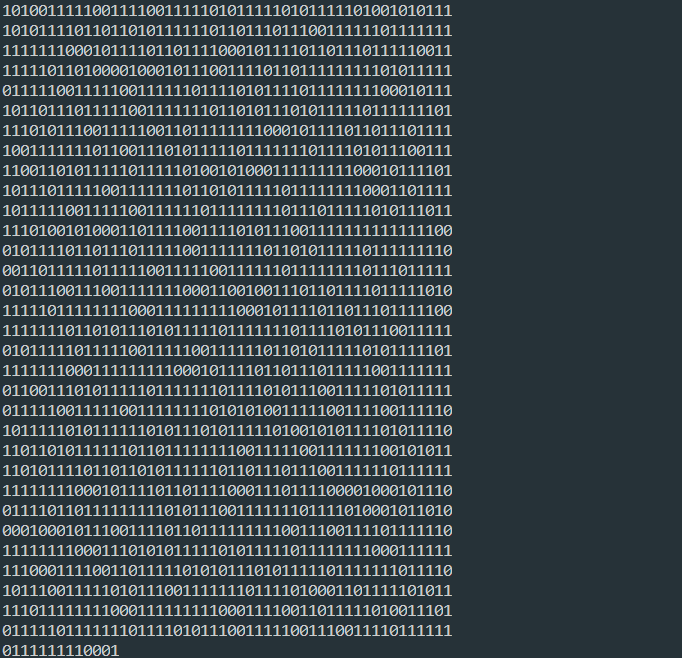
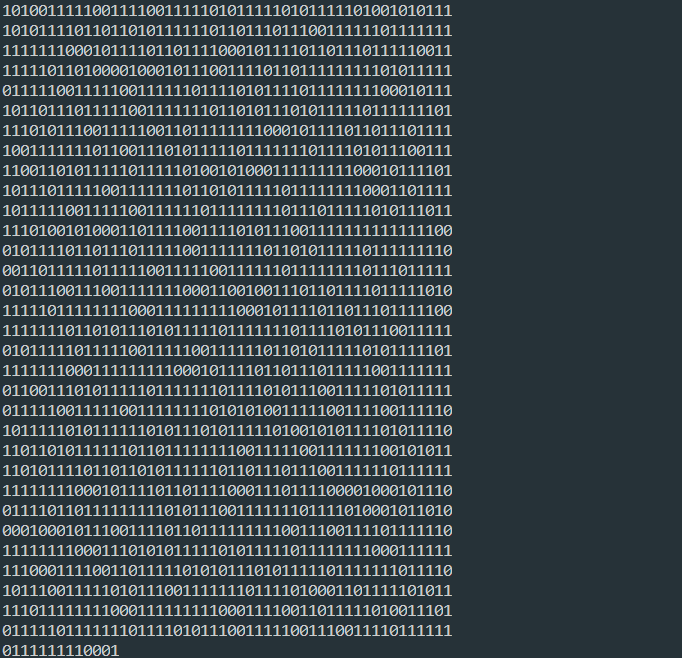
* 验证性实验



* 设计性实验



* 综合性实验



1. **心得体会**

通过本次实验，使我对数据结构有了更深的理解，对指针的运用更加熟练，熟悉了对函数的定义和操作。