哈尔滨工程大学

实 验 报 告

实 验 名 称： 约瑟夫环 （A档，1/7）

班 级： 20210616

学 号： 2021065220

姓 名： 陈贺喜

实 验 时 间： 2022.10.28

成 绩：

指 导 教 师： 张泽宝

实验室名称： 21B计算中心276室

**哈尔滨工程大学实验室与资产管理处 制**

|  |
| --- |
| 实验名称： 约瑟夫环 |
| 一、问题描述  设有编号为1,2,…,n的n（n＞0）个人围成一个圈，每个人持有一个密码m。从第一个人开始报数，报到m时停止报数，报m的人出圈，再从他的下一个人起重新报数，报到m时停止报数，报m的出圈，……，如此下去，直到所有人全部出圈为止。当任意给定n和m后，设计算法求n个人出圈的次序。   1. 数据结构设计   约瑟夫环可以看成一个循环链表，从1每隔m将元素移出链表  typedef struct LinkList  {  int data; //存储每个人的编号  struct LinkList \*next; //指向下一个人的指针  }LinkList;  三、算法设计  /\*初始化约瑟夫环\*/  LinkList \* InitLoop(int n)  {  LinkList \*p = (LinkList\*)malloc(sizeof(LinkList));  p->data = 1;  p->next = NULL;  LinkList \*q, \*r;  q = p;  for(int i = 1; i < n; i++)  {  LinkList \*s = (LinkList\*)malloc(sizeof(LinkList));  s->data = i+1;  s->next = NULL;  q->next = s;  q = q->next;  }  r = q;  q->next = p;  return r;//返回最后元素地址  }  首先初始化n个人的约瑟夫环，使用尾插法，最后将最后一个人的指针指向第一个人形成一个环，使用r记录最后一个人的地址，返回最后元素的地址，因为移除一个元素要知道此元素之前的元素地址，而当m为1时，从1开始，如果不是返回的最后元素的地址，那么要将指针移动n-1次到达尾端，直接返回尾地址可规避此问题。  /\*输出出圈顺序\*/  void OutPut(LinkList \*L, int m)  {  if(!L) exit(0);  LinkList \*p, \*q; p = L;  int i;  while(p->next !=p)  {  for(i = 1;i<m;i++){p = p->next;}  printf("%d ", p->next->data);  q = p->next;  p->next = p->next->next;  free(q);  }  printf("%d\n", p->data);  }  然后开始出圈，当p->next != p时代表环中还有人未出圈，i从1开始，指针后移m-1次后到达出圈人位置前，打印p->next->data输出出圈人的编号，q = q->next 出圈元素为q，p->next = p->next->next并free(q)删除元素。循环直到p->next == p，环中只剩一人，最后打印此人编号，屏幕输出结果即为出圈次序。  int main()  {  int n, m;  scanf("%d%d", &n, &m);  LinkList \*L = InitLoop(n);  OutPut(L, m);  return 0;  }  主函数，输入n和m，输出结果。  四、界面设计  无界面，直接输入n,m得到结果   1. 运行测试与分析     输入n=10,m=3，输出结果3 6 9 2 7 1 8 5 10 4  六、实验收获与思考  学会了链表的创建，尾插法，链表元素的删除等方法，提高了编程能力。 |

哈尔滨工程大学

实 验 报 告

实 验 名 称： 表达式求值 （A档，1/7）

班 级： 20210616

学 号： 2021065220

姓 名： 陈贺喜

实 验 时 间： 2022.11.2

成 绩：

指 导 教 师： 张泽宝

实验室名称： 21B计算中心276室

**哈尔滨工程大学实验室与资产管理处 制**

|  |
| --- |
| 实验名称： 表达式求值问题 |
| 一、问题描述  表达式是数据运算的基本形式。人们的书写习惯是中缀式，如：11 + 22 \* ( 7 - 4 ) / 3。中缀式的计算按运算符的优先级及括号优先的原则，相同级别从左到右进行计算。表达式还有后缀式（如：22 7 4 - \* 3 / 11 +）和前缀式（如：+ 11 / \* 22 – 7 4 3）。后缀表达式和前缀表达式中没有括号，给计算带来方便。如后缀式计算时按运算符出现的先后进行计算。本设计的主要任务是进行表达式形式的转换及不同形式的表达式计算。   1. 数据结构设计   #define INIT\_SIZE 100 //初始大小  #define INCREMENT 10 //当栈不够时增加大小  //运算符栈，存放字符形式的运算符  typedef struct {  char \*base;  char \*top;  int size;  } OpStack;  //运算数栈，存放浮点数  typedef struct {  double \*base;  double \*top;  int size;  } NuStack;   1. 算法设计   //初始化运算符栈  OpStack InitOpStack()  {  OpStack p;  p.base = (char \*)malloc(INIT\_SIZE \* sizeof(char));  p.top = p.base;  p.size = INIT\_SIZE;  return p;  }  //初始化运算数栈  NuStack InitNuStack()  {  NuStack p;  p.base = (double \*)malloc(INIT\_SIZE \* sizeof(double));  p.top = p.base;  p.size = INIT\_SIZE;  return p;  }  //运算符入栈  void PushOp(OpStack &p, char ch)  {  if (p.top - p.base >= p.size)  {  p.base = (char \*)realloc(p.base, (p.size + INCREMENT) \* sizeof(char));  p.top = p.base + p.size;  p.size += INCREMENT;  }  \*p.top = ch;  p.top++;  }  //运算数入栈  void PushNu(NuStack &p, double ch)  {  if (p.top - p.base >= p.size)  {  p.base = (double \*)realloc(p.base, (p.size + INCREMENT) \* sizeof(double));  p.top = p.base + p.size;  p.size += INCREMENT;  }  \*p.top = ch;  p.top++;  }  #运算符出栈  char PopOp(OpStack &p)  {  if (p.base == p.top)  return 0;  char ch = \*(p.top - 1);  p.top--;  return ch;  }  //运算数出栈  double PopNu(NuStack &p)  {  if (p.base == p.top)  return 0;  double ch = \*(p.top - 1);  p.top--;  return ch;  }  //返回栈顶元素（运算符）  char GetopOp(OpStack &p)  {  if (p.top == p.base)  {  return 0;  }  else  {  return \*(p.top - 1);  }  }  //返回栈顶元素（运算数）  double GetopNu(NuStack &p)  {  if (p.top == p.base)  {  return 0;  }  else  {  return \*(p.top - 1);  }  }  //比较运算符优先级大小  char Precede(char a, char b)  {  int i, j;  char pre[][8] = {/\*运算符之间的优先级制作成一张表格\*/  {'>', '>', '<', '<', '<', '>', '>', '<'},  {'>', '>', '<', '<', '<', '>', '>', '<'},  {'>', '>', '>', '>', '<', '>', '>', '>'},  {'>', '>', '>', '>', '<', '>', '>', '>'},  {'<', '<', '<', '<', '<', '=', '0', '<'},  {'>', '>', '>', '>', '0', '>', '>', '>'},  {'<', '<', '<', '<', '<', '0', '=', '<'},  {'>', '>', '>', '>', '<', '>', '>', '>'}};  switch (a) {  case '+':i = 0;break;  case '-':i = 1;break;  case '\*':i = 2;break;  case '/':i = 3;break;  case '(':i = 4;break;  case ')':i = 5;break;  case '#':i = 6;break;  case '%':i = 7;break;}  switch (b) {  case '+':j = 0;break;  case '-':j = 1;break;  case '\*':j = 2;break;  case '/':j = 3;break;  case '(':j = 4;break;  case ')':j = 5;break;  case '#':j = 6;break;  case '%':j = 7;break;}  return pre[i][j];  }  //运算符优先级大小  int prio(char op) //给运算符优先级排序  {  int priority;  if (op == '\*' || op == '/' || op == '%')  priority = 2;  if (op == '+' || op == '-')  priority = 1;  if (op == '(' || op == ')' || op == '#')  priority = 0;  return priority;  }  //运算加、减、乘、除、取模  double Operate(double i, char b, double j)  {  double result;  switch (b) {  case '+':result = i + j;break;  case '-':result = i - j;break;  case '\*':result = i \* j;break;  case '/':result = i / j;break;  case '%':result = fmod(i, j);break;  }  return result;  }  //中缀表达式求值，从左到有读取，运算数入栈，运算符比较与运算符栈顶运算符优先级大小，大于栈顶运算符则入栈，小于取出运算数栈顶两个元素进行运算再入栈，等于也入栈，最终运算数栈之剩最终结果。  void Infix()  {  OpStack OPTR = InitOpStack(); //运算符栈  NuStack OPND = InitNuStack(); //运算数栈  PushOp(OPTR, '#');  char chx, op;  double num;  int i, j;  printf("输入中缀表达式(以#结束,不加=):\n");  scanf("%c", &chx);  while (chx != '#' || GetopOp(OPTR) != '#')  {  num = 0;  i = 0;  char tmpn[100];  while (chx <= '9' && chx >= '0' || chx == '.')  {  tmpn[i] = chx;  scanf("%c", &chx);  i++;  }  if (i)  {  num = atof(tmpn);  PushNu(OPND, num);  for (int j = 0; j < i; j++) tmpn[j] = '\0';  }  switch (Precede(GetopOp(OPTR), chx))  {  case '<':  PushOp(OPTR, chx);  scanf("%c", &chx);  break;  case '>':  op = PopOp(OPTR);  double a, b;  b = PopNu(OPND);  a = PopNu(OPND);  PushNu(OPND, Operate(a, op, b));  break;  case '=':  PopOp(OPTR);  scanf("%c", &chx);  break;  }  }  printf("%g\n", GetopNu(OPND));  }  //前缀表达式求值，从右向左读取，遇到运算数入栈，遇到运算符将运算数栈栈顶两元素出栈进行运算，最终运算数栈只剩最终结果。  void Polish()  {  NuStack OPND = InitNuStack(); //运算数栈  char chx[1000];  double num;  printf("输入前缀表达式(以#结束,不加=):\n");  scanf("%[^\n]", chx);  char \*q = chx;  while (\*q != '#') q++;  while (q != chx)  {  num = 0;  char tmpn[100], tmpnn[100]; //tmpnn反转tmpn  int j = 0;  while (\*q <= '9' && \*q >= '0' || \*q == '.')  {  tmpn[j] = \*q;  q--;  j++;  }  if (j)  {  for (int i = 0; i < j; i++)  {  tmpnn[i] = tmpn[j - i - 1];  }  num = atof(tmpnn);  PushNu(OPND, num);  for (int k = 0; k < j; k++)  {  tmpn[k] = '\0';  tmpnn[k] = '\0';  }  }  if (\*q == '+' || \*q == '-' || \*q == '\*' || \*q == '/' || \*q == '%')  {  double a, b;  a = PopNu(OPND);  b = PopNu(OPND);  PushNu(OPND, Operate(a, \*q, b));  q--;  }  q--;  }  double a, b;  a = PopNu(OPND);  b = PopNu(OPND);  PushNu(OPND, Operate(a, \*q, b));  printf("%g\n", GetopNu(OPND));  }  //后缀表达式求值，从左向右读取，遇到运算数入栈，遇到运算符弹出栈顶两个元素计算结果入栈。  void RePolish()  {  OpStack OPTR = InitOpStack(); //运算符栈  NuStack OPND = InitNuStack(); //运算数栈  char chx;  double num;  int i, j;  printf("输入后缀表达式(以#结束,不加=):\n");  scanf("%c", &chx);  while (chx != '#')  {  num = 0;  i = 0;  char tmpn[100];  if (chx == '+' || chx == '-' || chx == '\*' || chx == '/' || chx == '%')  {  PushNu(OPND, Operate(PopNu(OPND), chx, PopNu(OPND)));  } else {  while (chx <= '9' && chx >= '0' || chx == '.')  {  tmpn[i] = chx;  scanf("%c", &chx);  i++;  }  if (i)  {  num = atof(tmpn);  PushNu(OPND, num);  for (int j = 0; j < i; j++) tmpn[j] = '\0';  }  }  scanf("%c", &chx);  }  printf("%g\n", GetopNu(OPND));  }  //中缀转前缀，从右向左读取，遇到运算数存到中间结果栈，遇到右括号存入运算符栈，遇到左括号出栈并入中间结果栈直到遇到右括号，遇到运算符比较与运算符栈栈顶运算符优先级，当前运算符大于等于运算符栈顶运算符，当前运算符入运算符栈，小于则运算符栈顶出栈并入中间结果栈，最终中间结果依次出栈打印即为前缀表达式。  void Infix\_to\_Polish()  {  OpStack OPTR = InitOpStack(); //运算符栈  OpStack OPTR2 = InitOpStack(); //中间结果  PushOp(OPTR, '#');  PushOp(OPTR2, '#');  char chx[1000];  printf("中缀->前缀:\n输入中缀表达式(以#结束,不加=):\n");  scanf("%[^\n]", chx);  int n = strlen(chx) - 2;  while (n != -1)  {  int i = 0;  while (chx[n] <= '9' && chx[n] >= '0' || chx[n] == '.')  {  PushOp(OPTR2, chx[n]);  --n;  i++;  }  if (i) PushOp(OPTR2, '|');  if (chx[n] == ')')  {  PushOp(OPTR, chx[n]);  --n;  }  if (chx[n] == '(')  {  while (GetopOp(OPTR) != ')') PushOp(OPTR2, PopOp(OPTR));  PopOp(OPTR);  --n;  }  if (chx[n] == '+' || chx[n] == '-' || chx[n] == '\*' || chx[n] == '/' || chx[n] == '%')  {  if (prio(chx[n]) >= prio(GetopOp(OPTR)))  {  PushOp(OPTR, chx[n]);  --n;  } else if (prio(chx[n]) < prio(GetopOp(OPTR)))  {  PushOp(OPTR2, PopOp(OPTR));  }  }  }  while(GetopOp(OPTR) != '#')  {  if(OPTR.top - OPTR.base != 2)  {  printf("%c ", PopOp(OPTR));  }  else  {  printf("%c", PopOp(OPTR));  }  }  while (GetopOp(OPTR2) != '#')  {  if (GetopOp(OPTR2) == '|')  {  PopOp(OPTR2);  printf(" ");  }  if(GetopOp(OPTR2) <= '9' && GetopOp(OPTR2) >= '0' || GetopOp(OPTR2) == '.')  {  printf("%c", PopOp(OPTR2));  }  else  {  printf(" %c", PopOp(OPTR2));  }  }  printf("\n");  }  //中缀转后缀，从左向右读取，遇到运算数直接打印，遇到左括号入栈，遇到右括号运算符栈出栈并打印直到遇到左括号，最终得到后缀表达式。  void Infix\_to\_RePolish()  {  OpStack OPTR = InitOpStack(); //运算符栈  PushOp(OPTR, '#');  char chx;  double num;  int i, j;  printf("中缀->后缀:\n输入中缀表达式(以#结束,不加=):\n");  scanf("%c", &chx);  while (chx != '#')  {  if (chx == ' ')  {  scanf("%c", &chx);  }  num = 0;  i = 0;  char tmpn[100];  while (chx <= '9' && chx >= '0' || chx == '.')  {  tmpn[i] = chx;  scanf("%c", &chx);  i++;  }  if (i)  {  num = atof(tmpn);  printf("%g ", num);  for (int j = 0; j < i; j++) tmpn[j] = '\0';  }  if (chx == ')')  {  while (GetopOp(OPTR) != '(') printf("%c ", PopOp(OPTR));  PopOp(OPTR);  scanf("%c", &chx);  }  if (chx == '(')  {  PushOp(OPTR, chx);  scanf("%c", &chx);  }  if (chx == '+' || chx == '-' || chx == '\*' || chx == '/' || chx == '%')  {  if (prio(chx) > prio(GetopOp(OPTR)))  {  PushOp(OPTR, chx);  scanf("%c", &chx);  } else if (prio(chx) <= prio(GetopOp(OPTR)))  {  printf("%c ", PopOp(OPTR));  }  }  }  while (GetopOp(OPTR) != '#')  {  printf("%c ", PopOp(OPTR));  }  printf("\n");  }  int main() {  // Infix();  // Polish();  // RePolish();  // Infix\_to\_Polish();  // Infix\_to\_RePolish();  return 0;  }  四、界面设计    输入表达式，以#结束并回车   1. 运行测试与分析   计算中缀表达式11+22\*(7-4)/3  中缀转前缀  计算前缀表达式  中缀转后缀  计算后缀表达式  六、实验收获与思考  学到了栈的应用，学习了各种表达式之间的转换，提高了编程能力。 |

哈尔滨工程大学

实 验 报 告

实 验 名 称： 魔方阵 （A档，3/7）

班 级： 20210616

学 号： 2021065220

姓 名： 陈贺喜

实 验 时 间： 2022.10.28

成 绩：

指 导 教 师： 张泽宝

实验室名称： 21B计算中心276室

**哈尔滨工程大学实验室与资产管理处 制**

|  |
| --- |
| 实验名称： 魔方阵 |
| 一、问题描述  魔方阵是一个古老的智力问题，它要求在一个m×m的矩阵中填入1～的 数字（m为奇数），使得每一行、每一列、每条对角线的累加和都相等，如图1所示。    二、数据结构设计  int a[m][m];  三、算法设计  #include <stdio.h>  int main()  {  int m;  printf("输入奇数m:");  scanf("%d", &m);  while(m % 2 == 0) scanf("%d", &m);  int a[m][m];  int i;  int col,row;  col = (m-1)/2;  row = 0;  a[row][col] = 1; //第一行中间为1  for(i = 2; i <= m\*m; i++) // i比i-1行减1,列加1，i-1是m倍数则行加1列不变  {  if((i-1)%m == 0 )  {  row++;  }  else  {  row--;  row = (row+m)%m;  col ++;  col %= m;  }  a[row][col] = i;  }  for(row = 0;row<m;row++)  {  for(col = 0;col < m; col ++)  {  printf("%6d",a[row][col]);  }  printf("\n");  }  return 0;  }  四、界面设计    五、运行测试与分析  输入奇数5得如下结果    输入奇数7得如下结果    六、实验收获与思考  学会了魔方阵的生成，提高了编程能力。 |

哈尔滨工程大学

实 验 报 告

实 验 名 称： 文件压缩 （A档，4/7）

班 级： 20210616

学 号： 2021065220

姓 名： 陈贺喜

实 验 时 间： 2022.10.28

成 绩：

指 导 教 师： 张泽宝

实验室名称： 21B计算中心276室

**哈尔滨工程大学实验室与资产管理处 制**

|  |
| --- |
| 实验名称： 文件压缩 |
| 一、问题描述  哈夫曼编码是一种常用的数据压缩技术，对数据文件进行哈夫曼编码可大大 缩短文件的传输长度，提高信道利用率及传输效率。要求采用哈夫曼编码原理， 统计文本文件中字符出现的词频，以词频作为权值，对文件进行哈夫曼编码以达 到压缩文件的目的，再用哈夫曼编码进行译码解压缩。   统计待压缩的文本文件中各字符的词频，以词频为权值建立哈夫曼树， 并将该哈夫曼树保存到文件 HufTree.dat 中。   根据哈夫曼树（保存在 HufTree.dat 中）对每个字符进行哈夫曼编码，并 将字符编码保存到 HufCode.txt 文件中。   压缩：根据哈夫曼编码，将源文件进行编码得到压缩文件 CodeFile.dat。   解压：将 CodeFile.dat 文件利用哈夫曼树译码解压，恢复为源文件。  二、数据结构设计  typedef struct Node  {  int weight; //权值  int parent; //父节点的序号，为-1的是根节点  int lchild, rchild; //左右孩子节点的序号，为-1的是叶子节点  char data; //字符  string code; //字符编码  }HTNode, \*HuffmanTree; //用来存储赫夫曼树中的所有节点  三、算法设计  #include <bits/stdc++.h>  using namespace std;  typedef struct Node  {  int weight; //权值  int parent; //父节点的序号，为-1的是根节点  int lchild, rchild; //左右孩子节点的序号，为-1的是叶子节点  char data; //字符  string code; //字符编码  }HTNode, \*HuffmanTree; //用来存储赫夫曼树中的所有节点  void Get\_min(HuffmanTree HT, int &min1, int &min2)  {  int i = 0, tmp;  while(HT[i].parent != -1) i++;  tmp = HT[i].weight;  min1 = i;  for(; HT[i].weight > 0; i++)  {  if(HT[i].parent == -1 && HT[i].weight < tmp)  {  tmp = HT[i].weight;  min1 = i;  }  }  HT[min1].parent = 1;  i = 0;  while(HT[i].parent != -1) i++;  tmp = HT[i].weight;  min2 = i;  for(; HT[i].weight > 0; i++)  {  if(HT[i].parent == -1 && HT[i].weight < tmp)  {  tmp = HT[i].weight;  min2 = i;  }  }  HT[min1].parent = 1;  }  HuffmanTree Get\_HuffmanTree(string chars,int \*weights, int n)  {  int total = 2\*n-1;  HuffmanTree HT = new HTNode[total];  int i;  for(i = 0; i < n; i++)  {  HT[i].weight = weights[i];  HT[i].data = chars[i];  HT[i].parent = -1;  HT[i].lchild = -1;  HT[i].rchild = -1;  }  for(; i< total; i++)  {  HT[i].weight = 0;  HT[i].parent = -1;  HT[i].lchild = -1;  HT[i].rchild = -1;  }  int min1, min2;  for(i = n; i < total; i++)  {  Get\_min(HT,min1,min2);  HT[min1].parent = i;  HT[min2].parent = i;  HT[i].lchild = min1;  HT[i].rchild = min2;  HT[i].weight = HT[min1].weight + HT[min2].weight;  }  return HT;  }  void Save\_HuffmanTree(HuffmanTree HT, int n, string filename)  {  ofstream file(filename);  if(file.good())  {  for(int i = 0; i < n; i++)  {  file << HT[i].weight << " "  << HT[i].parent << " "  << HT[i].lchild << " "  << HT[i].rchild << endl;  }  file.close();  cout << "写入成功! " << filename << endl;  }  else  {  cout << "写入失败! " << filename << endl;  }  }  HuffmanTree Read\_HufTree(string filename)  {  HuffmanTree HT;  ifstream file(filename);  if(file.good())  {  int i = 0, tmp;  while(!file.eof())  {  file >> tmp;  i++;  }  file.close();  int n = i / 4;  file.open(filename);  HT = new HTNode[n];  for(i = 0; i < n; i++)  {  file >> HT[i].weight  >> HT[i].parent  >> HT[i].lchild  >> HT[i].rchild;  }  file.close();  cout << "读取成功! " << filename << endl;  }  else  {  cout << filename << " 文件不存在 " << filename << endl;  }  return HT;  }  string\* Get\_HuffmanCode(HuffmanTree HT, int n)  {  int cur, i = 0, code\_len = 0;  cur = 2 \* n - 2;  string \*HC = new string[n];  char \*code = new char[n];  for(i = 0; i < cur + 1; i++) HT[i].weight = 0; //weight = 0 未遍历 1 遍历左孩子 2遍历右孩子  while(cur != -1)  {  if(HT[cur].weight == 0)//左右孩子均未被遍历，先向左遍历  {  HT[cur].weight = 1;  if (HT[cur].lchild != -1)//如果当前节点不是叶子节点，则记下编码，并继续向左遍历  {  code[code\_len++] = '0';  cur = HT[cur].lchild;  }  else//如果当前节点是叶子节点，则终止编码，并将其保存起来  {  code[code\_len] = '\0';  HC[cur] = code; //复制编码串  }  }  else if (HT[cur].weight == 1)//左孩子已被遍历，开始向右遍历右孩子  {  HT[cur].weight = 2;  if (HT[cur].rchild != -1)//如果当前节点不是叶子节点，则记下编码，并继续向右遍历  {  code[code\_len++] = '1';  cur = HT[cur].rchild;  }  }  else//左右孩子均已被遍历，退回到父节点，同时编码长度减1  {  HT[cur].weight = 0;  cur = HT[cur].parent;  --code\_len;  }  }  return HC;  }  void Save\_HuffmanCode(string \*HC, string chars, string filename)  {  ofstream file(filename);  if(file.good())  {  for(int i = 0; i < chars.length(); i++)  {  file << chars[i] << " " << HC[i] << endl;  }  cout << "写入成功! " << filename << endl;  file.close();  }  else  {  cout << "写入失败! " << filename << endl;  }  }  void Save\_Zip(HuffmanTree HT, string str, string filename)  {  int len = 0;  ofstream file1(filename, ios::trunc);  file1.close();  ofstream file(filename, ios::app);  while(HT[len].data > 0) len++;  if(file.good())  {  for(int i = 0; i < str.length(); i++)  {  for(int j = 0; j < len; j++)  {  if(str[i] == HT[j].data)  {  file << HT[j].code ;  }  }  }  file.close();  cout << "写入成功! " << filename << endl;  }  else  {  cout << "写入失败! " << filename << endl;  }  }  string Unzip(string filename, HuffmanTree HT)  {  string str;  ifstream file(filename);  if(file.good())  {  string code\_read, code = "";  file >> code\_read;  int i, j, len;  len = code\_read.length();  for(i = 0; i < len; i++)  {  code += code\_read[i];  for(j = 0; HT[j].data > 0; j++)  {  if(HT[j].code == code)  {  str += HT[j].data;  code = "";  break;  }  }  }  file.close();  }  else  {  str = "error";  }  return str;  }  void Save\_Uzip\_Str(string txt, string filename)  {  ofstream file(filename);  if(file.good())  {  file << txt << endl;  file.close();  cout << "写入成功! " << filename << endl;  }  else  {  cout << "写入失败! " << filename << endl;  }  }  int main()  {  string str, chars, str\_filename;  bool f = true;  cout << "输入需要压缩字符串的文件：";  cin >> str\_filename;  while(f)  {  ifstream file(str\_filename);  if(file.good())  {  string tmp; bool if\_enter = false;  while(getline(file,tmp))  {  if(if\_enter)  {  str += "\n" + tmp;  }  else  {  str += tmp;  if\_enter = true;  }  }  file.close();  f = false;  }  else  {  cout << "文件不存在，请重新输入：";  cin >> str\_filename;  }  }  cout << str << endl;  chars += str[0];  for(int i = 1; i < str.length(); i++)  {  bool flag = true;  for(int j = 0; j < chars.length(); j++)  {  if(str[i] == chars[j])  {  flag = false;  break;  }  }  if(flag)  {  chars += str[i];  }  }  int weights[chars.length()];  for(int i = 0; i < chars.length(); i++)  {  weights[i] = 0;  for(int j = 0; j < str.length(); j++)  {  if(str[j] == chars[i])  {  weights[i]++;  }  }  }  HuffmanTree HT = Get\_HuffmanTree(chars, weights, chars.length());  Save\_HuffmanTree(HT, 2\*chars.length()-1, "HufTree.dat");  HuffmanTree HT2 = Read\_HufTree("HufTree.dat");  string \*HC = Get\_HuffmanCode(HT2, chars.length());  Save\_HuffmanCode(HC, chars, "HufCode.txt");  for(int i = 0; i < chars.length(); i++) //将字符和对应的编码写入HT  {  HT[i].code = HC[i];  HT2[i].data = chars[i];  HT2[i].code = HC[i];  }  Save\_Zip(HT2, str, "CodeFile.dat");  string uzip\_str = Unzip("CodeFile.dat",HT2);  Save\_Uzip\_Str(uzip\_str, "uzip." + str\_filename);  pause();  }   1. 界面设计   输入字符串文件，下一行输出字符串内容     1. 运行测试与分析     字符串文件内容    哈夫曼树文件    CodeFile.txt    HufCode.txt    解压后    六、实验收获与思考  学会了二叉树的应用，学会了文件压缩的算法，提高了编程能了。 |

哈尔滨工程大学

实 验 报 告

实 验 名 称： 邻接矩阵 （B档，5/7）

班 级： 20210616

学 号： 2021065220

姓 名： 陈贺喜

实 验 时 间： 2022.10.28

成 绩：

指 导 教 师： 张泽宝

实验室名称： 21B计算中心276室

**哈尔滨工程大学实验室与资产管理处 制**

|  |
| --- |
| 实验名称： 邻接矩阵 |
| 一、问题描述  邻接矩阵   邻接矩阵的 C 语言描述   基本运算的算法——建立无向网的邻接矩阵、求图中与顶点 i 邻接的第 一个顶点、求图中顶点 i 相对于顶点 j 的下一个邻接点、若图 G 中存在 顶点 u，则返回该顶点在图中的位置、图的广度优先遍历、图的深度优 先遍历  二、数据结构设计  typedef struct  {  char vexs[MAX\_VERTEX]; //顶点信息  int arcs[MAX\_VERTEX][MAX\_VERTEX]; //邻接矩阵  int vexnum,arcnum; //顶点数、边数  }Graph;  三、算法设计  #include<bits/stdc++.h>  using namespace std;  #define OK 1  #define Maxint INT\_MAX  #define MAX\_VERTEX 100  typedef int status ;  typedef struct  {  char vexs[MAX\_VERTEX];  int arcs[MAX\_VERTEX][MAX\_VERTEX];  int vexnum,arcnum;  }Graph;  status LocateVex(Graph G,int u)  {  int i;  for(i = 0; i < G.vexnum; i++)  if(u == G.vexs[i]) return i;  return -1;  }  status CreateUDN(Graph &G)  {  cout << "输入总顶点数，总边数 " << endl;  cin >> G.vexnum >> G.arcnum;  cout << "依次输入点的信息" << endl;  for(int i = 0; i < G.vexnum; i++)  cin >> G.vexs[i];  for(int i = 0;i < G.vexnum; i++) //初始化邻接矩阵，边的权值均为Maxint  for(int j = 0; j < G.vexnum; j++)  G.arcs[i][j] = Maxint;    for(int k = 0; k < G.arcnum; k++) //构造邻接矩阵  {  char v1, v2;  int w;  cin >> v1 >> v2 >> w; //输入一条边依附的顶点及权值  int i = LocateVex(G,v1);  int j = LocateVex(G,v2); //确定v1，v1在G中的位置，即顶点数组的下标  G.arcs[i][j] = w; //边<v1,v2>的权值置为w  G.arcs[j][i] = G.arcs[i][j]; //置<v1,v2>的对称边<v2,v1>的权值为w  }  return OK;  }  status FirstNeightbor(Graph &G, int i)  {  for(int j = 0; j < G.vexnum; j++)  {  if(G.arcs[i][j] > 0 && G.arcs[i][j] != Maxint)  {  return j;  }  }  return -1;  }  status NextNeightBor(Graph &G, int i, int j)  {  for(int k = j + 1; k < G.vexnum; k++)  {  if(G.arcs[i][k] > 0 && G.arcs[i][k] != Maxint)  {  return k;  }  }  return -1;  }  int FindVex(Graph &G, char u)  {  for(int i = 0; i < G.vexnum; i++)  {  if(G.vexs[i] == u)  {  return i;  }  }  return -1;  }  void DFS(Graph &G, int \*visited, int i)  {  int j;  cout << G.vexs[i] << " ";  visited[i] = 1;  for(j = 0; j < G.vexnum; j++)  {  if(G.arcs[i][j] != 0 && G.arcs[i][j] != Maxint && !visited[j])  {  DFS(G, visited, j);  }  }  }  void DFST(Graph &G)  {  int i, visited[G.vexnum];  for(i = 0; i < G.vexnum; i++)  {  visited[i] = 0;  }  for(i = 0; i < G.vexnum; i++)  {  if(!visited[i])  {  DFS(G, visited, i);  }  }  cout << endl;  }  typedef struct  {  char data[MAX\_VERTEX];  int front;  int rear;  }SqQueue;  // 初始化队列  void InitQueue(SqQueue &Q)  {  Q.front = 0;  Q.rear = 0;  }  bool EnQueue(SqQueue &Q, char e)  {  if( ( Q.rear+1 ) % MAX\_VERTEX == Q.front )  return false;  Q.data[Q.rear]=e;  Q.rear = (Q.rear+1)%MAX\_VERTEX;  return true;  }  char\* DeQueue(SqQueue &Q, char \*e)  {  if( Q.front == Q.rear )  return NULL;  \*e = Q.data[Q.front];  Q.front = (Q.front+1)%MAX\_VERTEX;  return e;  }  bool isEmptyQueue(SqQueue &Q)  {  return Q.front == Q.rear ? true : false;  }  void BFST(Graph &G)  {  SqQueue Q;  int i, j, mark, visited[G.vexnum];  char data;  for(i = 0; i < G.vexnum;i++)  visited[i] = false;  InitQueue(Q);  for(i=0; i < G.vexnum; i++)  {  if(!visited[i])  {  visited[i] = 1;  EnQueue(Q,G.vexs[i]);  while(!isEmptyQueue(Q))  {  DeQueue(Q,&data); //队首顶点出队，并赋值给data  cout << data << " ";  //找所删除顶点的下标，更新该下标值，以便正确找到与出队元素相连的其他顶点  for( j = 0; j < G.vexnum; j++)  if(G.vexs[j] == data )  mark = j ;  //找寻与此顶点相连且未被访问的顶点，逐次标记、打印，并入队  for(j=0;j<G.vexnum;j++)  {  if(G.arcs[mark][j]==1 && !visited[j])  {  visited[j] = 1;  printf("%c ",G.vexs[j]);  EnQueue(Q,G.vexs[j]);  }  }  }  }  }  cout << endl;  }  int main()  {  cout << "\033[36m\033[1m";  Graph G;  CreateUDN(G);  cout << "邻接矩阵为：" << endl;  for(int i = 0; i < G.vexnum; i++)  {  for(int j = 0; j < G.vexnum; j++)  cout << G.arcs[i][j] << " ";  cout << endl;  }  cout << FirstNeightbor(G , 0) << endl;  cout << NextNeightBor(G, 0, 2) << endl;  cout << FindVex(G, 'A') << endl;  DFST(G);  BFST(G);  cout << "\033[0m";  return 0;  }  四、界面设计    五、运行测试与分析    输入如图所示的图  邻接矩阵  ∞∞ 6 2  ∞∞ 3 5  6 3 ∞∞  2 5 ∞∞  输出A（vexs[0]）邻接的第一个顶点为vexs[2],即C  A相对于C的第一个顶点为D  深度优先遍历为A C B D  广度优先遍历为A B C D  六、实验收获与思考  学会了图的邻接矩阵表示，提高了编程能力。 |

哈尔滨工程大学

实 验 报 告

实 验 名 称：查找最高分与次高分 （A档，6/7）

班 级： 20210616

学 号： 2021065220

姓 名： 陈贺喜

实 验 时 间： 2022.10.28

成 绩：

指 导 教 师： 张泽宝

实验室名称： 21B计算中心276室

**哈尔滨工程大学实验室与资产管理处 制**

|  |
| --- |
| 实验名称： 查找最高分与次高分 |
| 一、问题描述  有512人参与玩某游戏，从 1~512 给每个人分配一个编号，每个人的游戏得 分在 0~999 之间，现要用不同方法查找出游戏参与者的最高分和次高分。  要求：自行产生 512 个的随机整数作为所有游戏参与者的得分。   输出所有游戏参与者（用编号表示）及其得分。   用顺序查找方法查找出其中取得最高分和次高分者及其分数，并输出。   锦标赛法查找出其中取得最高分和次高分者及其分数，并输出。   通过无序序列建堆和堆调整得到取得最高分者和次高分者及其分数，并 输出。   比较不同方法的查找效率和各自的特点  二、数据结构设计  typedef struct  {  int num; //编号  int score; //分数  }Person;  三、算法设计  #include <bits/stdc++.h>  using namespace std;  typedef struct  {  int num;  int score;  }Person;  int random(int max, Person \*check, int n)//生成0-max的不重复随机数  {  int rand\_num;  while(true)  {  rand\_num = rand() % (max+1);  bool flag = false;  for(int i = 0; i < n; i++)  {  if(check[i].score == rand\_num)  {  flag = true;  break;  }  }  if(flag){continue;}  else{break;}  }  return rand\_num;  }  void Find\_Max(Person \*arr, int n)  {  int i;  int max1, max2, max1\_num, max2\_num;  max1 = 0;  for(i = 0; i < n; i++)  {  if(max1 < arr[i].score)  {  max1 = arr[i].score;  max1\_num = i;  }  }  max2 = 0;  for(i = 0; i < n; i++)  {  if(max2 < arr[i].score && arr[i].score < max1)  {  max2 = arr[i].score;  max2\_num = i;  }  }  cout << "顺序查找：" << endl;  printf("最高分：%3d，编号：%3d\n最低分：%3d，编号：%3d\n", arr[max1\_num].score, arr[max1\_num].num, arr[max2\_num].score, arr[max2\_num].num);  }  void Tournament(Person arr[], int len)  {  int k = log2(len);//k为比赛场次  int p = 1, q;//p表示和比赛对手的间隔，q表示两场比赛的间隔  for (int i = 0; i < k; i++) //i可表示当前比赛的小跟班个数  {  q = p \* 2;//两场比赛的间隔是和对手间隔的两倍  for (int j = 0; j < len; j += q)//模拟每场比赛  {  if (arr[j].score < arr[j + p].score)//如果后面的大  {  for (int t = 0; t < i + 1; t++)//比赛的数字和他的小跟班一起交换，所以是交换i+1次  {  Person temp = arr[j + t];  arr[j + t] = arr[j + p + t];  arr[j + p + t] = temp;  }  }  Person temp = arr[j + p];//把进行比较的数字中后者换到前者后面做跟班  arr[j + p] = arr[j + i + 1];  arr[j + i + 1] = temp;  }  p = q;//上一场比赛的间隔等于下一场比赛和对手的间隔  }  Person max1 = arr[0]; //最大的被换到开头  Person max2 = arr[1]; //第二大的在最大的跟班中产生  for (int i = 1; i <= k; i++)  if (arr[i].score > max2.score)  max2 = arr[i];  cout << "锦标赛法：" << endl;  printf("最高分：%3d，编号：%3d\n最低分：%3d，编号：%3d\n", max1.score, max1.num, max2.score, max2.num);  }  void max\_heap(Person \*arr, int start, int end) {  //建立父节点指标和子节点指标  int father = start;  int son = father \* 2 + 1;  while (son <= end) //若子节点指标在范围内才做比较  {  if (son + 1 <= end && arr[son].score < arr[son + 1].score) //先比较两个子节点大小，选择最大的  son++;  if (arr[father].score > arr[son].score) //如果父节点大于子节点代表调整完毕，直接跳出函数  {  return;  }  else //否则交换父子内容再继续子节点和孙节点比较  {  swap(arr[father], arr[son]);  father = son;  son = father \* 2 + 1;  }  }  }  void heap(Person \*arr, int len) {  //初始化，i从最后一个父节点开始调整  for (int i = len / 2 - 1; i >= 0; i--)  max\_heap(arr, i, len - 1);  // 先将第一个元素和已经排好的元素前一位做交换，再从新调整刚调整的元素之前的元素，直到排序完毕  for (int i = len - 1; i > 0; i--)  {  swap(arr[0], arr[i]);  max\_heap(arr, 0, i - 1);  }  cout << "堆排序：" << endl;  printf("最高分：%3d，编号：%3d\n最低分：%3d，编号：%3d\n", arr[len-1].score, arr[len-1].num, arr[len-2].score, arr[len-2].num);  }  int main()  {  Person \* players = new Person[512];  srand((int)time(0));  for(int i = 0; i < 512; i++)  {  players[i].score = random(999, players, i);  players[i].num = i + 1;  printf("\033[34m\033[1m%4d：%-4d\033[0m", players[i].num, players[i].score);  if((i+1) % 8 == 0) cout << endl;  }  cout << "\033[33m";  Find\_Max(players, 512);  cout << "\033[0m\033[32m";  Tournament(players, 512);  cout << "\033[0m\033[35m";  heap(players, 512);  cout << "\033[0m";  }  四、界面设计    五、运行测试与分析    六、实验收获与思考  学会了顺序查找、锦标赛法计算次大值、堆排序，提高了编程能力。 |

哈尔滨工程大学

实 验 报 告

实 验 名 称：直接插入、快速、直接选择排序算法验证 （C档，1/7）

班 级： 20210616

学 号： 2021065220

姓 名： 陈贺喜

实 验 时 间： 2022.10.28

成 绩：

指 导 教 师： 张泽宝

实验室名称： 21B计算中心276室

**哈尔滨工程大学实验室与资产管理处 制**

|  |
| --- |
| 实验名称： 直接插入排序、快速排序、直接选择排序算法验证 |
| 一、问题描述  （1）直接插入排序算法验证。  （2）快速排序算法验证。  （3）直接选择排序算法验证。  二、数据结构设计  一维数组int arr[8] = {9,3,6,2,8,1,6,4};  三、算法设计  #include <bits/stdc++.h>  using namespace std;  void print(int a[], int n)  {  for(int i = 0; i < n; i++)  {  cout << a[i] << " ";  }  cout << endl;  }  void insert\_sort(int a[], int n) //直接插入排序  {  int i, j, k;  for (i = 1; i < n; i++)  {  for (j = i - 1; j >= 0; j--) //为a[i]在前面的a[0...i-1]有序区间中找一个合适的位置  if (a[j] < a[i])  break;  if (j != i - 1) //如找到了一个合适的位置  {  int temp = a[i]; //将比a[i]大的数据向后移  for (k = i - 1; k > j; k--)  a[k + 1] = a[k];  a[k + 1] = temp; //将a[i]放到正确位置上  }  }  }  void QuickSort(int \*a, int low, int high) //快速排序  {  int i = low;  int j = high;  int key = a[low];  if (low >= high) //如果low >= high说明排序结束了  {  return ;  }  while (low < high) //该while循环结束一次表示比较了一轮  {  while (low < high && key <= a[high])  {  --high; //向前寻找  }  if (key > a[high])  {  swap(a[low], a[high]);  ++low;  }  while (low < high && key >= a[low])  {  ++low; //向后寻找  }  if (key < a[low])  {  swap(a[low], a[high]);  --high;  }  }  QuickSort(a, i, low-1); //用同样的方式对分出来的左边的部分进行同上的做法  QuickSort(a, low+1, j); //用同样的方式对分出来的右边的部分进行同上的做法  }  void SelectionSort(int a[], int n) //直接选择排序  {  int i,j,min;  for (i = 0; i < n; i++) {  min = i;  for (j = i + 1; j < n; j++)  {  if (a[j] < a[min]) min = j;  }  swap(a[min], a[i]);  }  }  int main()  {  int arr[8] = {9,3,6,2,8,1,6,4};  // insert\_sort(arr, 8);  // QuickSort(arr, 0, 7);  //SelectionSort(arr, 8);  print(arr, 8);  }  四、界面设计  无界面  五、运行测试与分析    六、实验收获与思考  学会了三种排序算法，提高了编程能力。 |