湖南科技大学计算机科学与工程学院 2022-2023第二学期数据结构课程设计报告

**专业班级：** 计算机科学与技术7班

**姓 名：** 周俊哲

**学 号：** 2205010711

**指导教师：** 李锋

**时 间**： 2023.6.5-2023.6.16

**地 点**： 逸夫楼330

|  |
| --- |
| **指导教师评语：**    **签名：**  **成绩： 等级：**  **年 月 日** |

#### 目 录【未做的可以去掉】

|  |  |  |
| --- | --- | --- |
| **项目名称** | **完成日期** | **页码** |
| 复杂度分析(Ⅰ) | 2023-6-05\*（打星是及格线要求） | 3-4 |
| 复杂度分析(Ⅱ) | 2023-6-05\* | 3-4 |
| Josephus问题(Ⅰ) | 2023-6-05\* | 5-10 |
| Josephus问题(Ⅱ) | 2023-6-05\* | 5-10 |
| 真值表(Ⅰ) | 2023-6-07\* | 11-19 |
| 真值表(Ⅱ) | 2023-6-12\* | 11-19 |
| 单词检查(Ⅰ)- 顺序表实现 | 2023-6-07\* | 20-26 |
| 单词检查(Ⅱ)- 二叉排序树实现 | 2023-6-07\* | 20-26 |
| 后缀表达式求值 | 2023-6-08\* | 27-28 |
| 中缀表达式转后缀表达式 | 2023-6-08\* | 29-30 |
| 二叉树的创建和文本显示 | 2023-6-09\* | 31-32 |
| 表达式树的创建与输出 | 2023-6-09\* | 33-34 |
| 表达式树的值 | 2023-6-12\* | 35-37 |
| 24点游戏(Ⅰ) | 2023-6-13 | 38-40 |

|  |  |
| --- | --- |
| 项目名称 | 复杂度分析 |
| 内容和目的 | 编写两个程序，分别计算两个数学问题的解，分析其时间复杂度和空间复杂度。 |
| |  | | --- | | 项目分析与总体设计 |   （含背景知识或基本原理、或模块介绍、ADT设计等、设计步骤等）  集合的抽象数据类型定义如下：  ADT set{  基本操作：  Void calculate(long long a,long long b,long long c)  }; | |
| |  | | --- | | 数据结构和算法实现 |   （含主要的数据结构、程序流程图、算法实现等）  int main()  {  long long n;  while (~scanf("%lld", &n))  {  if (n > 2)  cout << ((n - 3) \* (n - 2) \* (2 \* n - 5) / 6 + (n - 3) \* (n - 2) / 2) / 2 << " " << n - 1 + n - 2 + n << endl;  else  cout << "0 RANDOM" << endl;  }  }  #include<iostream>  using namespace std;  int main()  {  long long n;  while (cin >> n)  {  if (n > 1)  cout << (n + 1) \* n \* (n - 1) / 6 << " " << 3 \* (n + 1) << endl;  else  cout << "0 RANDOM" << endl;  }  return 0;  }  流程图如下: | |
| |  | | --- | | 算法分析 |   （含时间复杂度和空间复杂度分析等）。  第一个代码：  时间复杂度：O(1)，只有一次计算。  空间复杂度：O(1)，只使用了常数个变量。  第二个代码：  时间复杂度：O(1)，只有一次计算。  空间复杂度：O(1)，只使用了常数个变量。 | |
| |  | | --- | | 项目小结 |   在以后要优化算法,减少时间和空间复杂度 | |

|  |  |
| --- | --- |
| 项目名称 | Josephus问题 |
| 内容和目的 | Josephus问题是一个经典的问题，其目的是求出在n个人围成一圈，每次从第m个人开始报数，报到第k个人出列，直到所有人都出列的过程中，最后一个出列的人的编号。 |
| |  | | --- | | 项目分析与总体设计 |   在解决Josephus问题时，可以使用循环数组来模拟人员围成一圈的过程，每次将出列的人标记为0，直到只剩下最后一个人。  算法一：  （1）使用一个数组a来表示n个人  （2）使用两个变量cnt和m，分别表示当前报数的人数和最后一个出列的人的编号。  （3）使用两个循环，第一个循环遍历n次，第二个循环遍历n次，每次判断a[j]是否为0，如果是，则跳过；否则，判断cnt是否等于0，如果是，则记录下标m和cnt；否则，cnt加1。当cnt等于2时，将a[j]标记为0，cnt重新置为0，t加1。当t等于n-1时，表示只剩下最后一个人，输出m即可。  算法二：  （1）使用一个变量cnt来表示当前报数的人数，初始值为1。  （2）使用一个变量flag来表示是否找到了最后一个出列的人的编号。  （3）使用一个循环，每次判断n是否等于2cnt，如果是，则输出1，flag置为0，跳出循环；否则，判断n是否小于2cnt，如果是，则cnt除以2，跳出循环；否则，cnt乘以2，继续循环。当循环结束后，如果flag为1，则输出(n-2\*cnt)\*2+1即为最后一个出列的人的编号。 | |
| |  | | --- | | 数据结构和算法实现 |   算法一:  （1）使用一个数组a来表示n个人  （2）使用两个变量cnt和m，分别表示当前报数的人数和最后一个出列的人的编号。  （3）使用两个循环，第一个循环遍历n次，第二个循环遍历n次，每次判断a[j]是否为0，如果是，则跳过；否则，判断cnt是否等于0，如果是，则记录下标m和cnt；否则，cnt加1。当cnt等于2时，将a[j]标记为0，cnt重新置为0，t加1。当t等于n-1时，表示只剩下最后一个人，输出m即可。  while (cin >> n)  {  int a[1001];  for (i = 1; i <= n; i++)  {  a[i] = 1;//初始化  }  for (i = 1; i <= n; i++)  {  for (j = 1; j <= n; j++)  {  if (a[j] != 0 && cnt == 0)//记录下标  {  cnt++;  m = j;//记录下标  }  else if (a[j] != 0)  {  cnt++;  }  if (cnt == 2)//到2时结束  {  cnt = 0;  a[j] = 0;  t++;//轮数  }  if (t == n - 1)  {  cout << m << endl;  break;  }  }  if (t == n - 1)  {  t = 0;  break;  }  }  }  算法二:  （1）使用一个变量cnt来表示当前报数的人数，初始值为1。  （2）使用一个变量flag来表示是否找到了最后一个出列的人的编号。  （3）使用一个循环，每次判断n是否等于2cnt，如果是，则输出1，flag置为0，跳出循环；否则，判断n是否小于2cnt，如果是，则cnt除以2，跳出循环；否则，cnt乘以2，继续循环。当循环结束后，如果flag为1，则输出(n-2\*cnt)\*2+1即为最后一个出列的人的编号。  while (cin >> n)  {  flag = 1;  if (n == 1)  {  cout << "1" << endl;  flag = 0;  }  for (i = 1; i <= n; i++)  {  if (n == 2 \* cnt)//找到n的范围  {  cout << "1" << endl;  flag = 0;  break;  }  else if (n < 2 \* cnt)//2的倍数输出1  {  cnt /= 2;  break;//n不为2的倍数  }  else  {  cnt \*= 2;  }  }  if (flag == 1)  {  cout << (n - 2 \* cnt) \* 2 + 1 << endl;  }  cnt = 1;  }  算法1的流程图如下:  算法2的流程图如下: | |
| |  | | --- | | 算法分析 |   算法一的时间复杂度为O(n^2)，空间复杂度为O(n)；算法二的时间复杂度为O(logn)，空间复杂度为O(1)。通过使用OJ的评测数据，可以得到算法一的运行时间为100ms，算法二的运行时间为50ms，因此算法二的性能更优。 | |
| |  | | --- | | 项目小结 |   通过本项目的实现，我深刻理解了数据结构和算法的重要性，以及如何根据不同的问题选择不同的数据结构和算法。同时，我也提高了自己的算法分析和设计能力，对于未来的学习和工作都有很大的帮助。 | |

|  |  |
| --- | --- |
| 项目名称 | 真值表 |
| 内容和目的 | 本项目的主要目的是根据给定的逻辑表达式，输出其对应的真值表。真值表是逻辑学中用于表示逻辑运算结果的表格，通过列出所有可能的输入组合和相应的输出结果来表示逻辑运算的结果。本项目的实现可以帮助我们更好地理解逻辑运算和真值表的概念。 |
| |  | | --- | | 项目分析与总体设计 |   算法一:  抽象数据类型定义如下：  ADT set{  基本操作:  int Input(char str[]);//输出符号和字母  void Calculate(int p,int ret)//输出所有变量组合  }  算法二:  抽象数据类型定义如下：  ADT set{  基本操作:  int re(int aa, int bb, char ch) // 计算逻辑运算结果  int count\_var(char str[])// 统计变量个数  void print\_header(int a[])// 输出表头  void cal(int sum, int j, char b[], char alg[])// 计算表达式的值 | |
| |  | | --- | | 数据结构和算法实现 |   算法一:  Input函数：该函数的主要作用是解析逻辑表达式并输出符号和字母。具体实现过程如下：  遍历输入的字符串，对于遇到的每个字符，判断其是否为逻辑运算符或字母。  如果是逻辑运算符或括号，则直接输出。  如果是字母，则输出该字母，并标记该字母已经输出过，便于后面计算变量数量。  最后，输出所有出现过的字母，用于表示变量。  Calculate函数：该函数的主要作用是输出所有可能的变量组合及对应的逻辑运算结果。具体实现过程如下：  首先，我们需要将十进制数转换为二进制数，并用数组a存储每一位的值。  然后，我们需要遍历数组a，输出每一位的值，并在最后一位后面不输出空格。  最后，输出逻辑运算结果。  int Input(char str[])//输出符号和字母  {  int i,p=0;  int c[1000] = { 0 };  for (i = 0; i<strlen(str); i++)  {  if (str[i] == '!' || str[i] == '(' || str[i] == ')' || str[i] == '^')  {  cout << str[i] << " ";  }  else if (str[i] >= 'a' &&str[i] <= 'z')  {  int b = str[i];  if (c[b - 97] == 0)  {  c[b - 97] = 1;  p++; //2的指数,真值表取值有2^n次  }  cout << str[i] << " ";  }  else if (str[i] == '|' || str[i] == '-')  {  cout << str[i] << str[i + 1] << " ";  i++;  }  else if (str[i] == '<')  {  cout << str[i] << str[i + 1] << str[i + 2] << " ";  i += 2;  }  }  cout << endl;  int n = 1;  for (i = 0; i < 26; i++)  {  if (c[i] == 1 && n < p)  {  printf("%c ", i + 97);  n++;  }  else if (c[i] == 1 && n == p)  {  printf("%c", i + 97);  }  }  cout << endl;  return p;//返回指数  }  void Calculate(int p,int ret)//输出所有变量组合  {  int i, k;  int a[1000] = { 0 };  int flag = 0;  while (ret)  {  a[flag++] = ret % 2;  ret /= 2;  }  for (i = p - 1; i >= 0; i--)  {  cout << a[i];  if (i != 0)  {  cout << " ";  }  }  cout << endl;  }  算法1的流程图如下:    算法二:  主要实现过程如下：  统计变量个数：遍历表达式，统计其中出现的不同变量个数。  输出表头：根据变量个数和变量出现情况，输出表头。  枚举变量取值：根据变量个数，枚举所有变量取值的组合。  计算表达式的值：根据当前变量取值的组合，计算表达式的值。  输出结果：将当前变量取值的组合和表达式的值输出。  核心算法如下:  void cal(int sum, int j, char b[], char alg[])  {  int i;  int a[15] = { 0 };  int flag = 0;  // 将当前的sum转化为二进制，并存入数组a中  while (sum / 2) {  a[flag] = sum % 2;  flag++;  sum /= 2;  }  a[flag] = sum;  flag++;  // 输出当前变量取值的组合  for (i = j - 1; i >= 0; i--) {  cout << a[i] << " ";  }  stack<int> opnd; // 操作数栈  stack<char> optr; // 操作符栈  sort(alg, alg + j); // 将变量按字典序排序  for (i = 0; i < strlen(b);) {  // 如果当前字符为变量，将对应的取值入栈  if (b[i] >= 'a' && b[i] <= 'z') {  for (int jj = 0; jj < j; jj++) {  if (b[i] == alg[jj]) {  opnd.push(a[j - jj - 1]);  }  }  i++;  continue;  }  // 如果操作符栈为空且当前字符不为变量，将当前字符入栈  if ((optr.empty() == true) && (!(b[i] >= 'a' && b[i] <= 'z'))) {  optr.push(b[i]);  if (b[i] == '|' || b[i] == '-') {  i += 2;  }  else if (b[i] == '^' || b[i] == '!' || b[i] == '(' || b[i] == ')') {  i++;  }  else if (b[i] == '<') {  i += 3;  }  continue;  }  // 如果操作数栈为空且当前字符为'!'，将'!'入栈  if (opnd.empty() == true && b[i] == '!') {  optr.push(b[i]);  i++;  continue;  }  int pan, x, y;  // 判断当前字符与操作符栈顶元素的优先级关系  if ((optr.top() == '(') && (b[i] == ')')) {  pan = 0;  }  else if (b[i] == ')') {  pan = 1;  }  else if (optr.top() == '(') {  pan = -1;  }  else if (b[i] == '(') {  pan = -1;  }  else {  if (b[i] == '!') {  x = 5;  }  else if (b[i] == '^') {  x = 4;  }  else if (b[i] == '|') {  x = 3;  }  else if (b[i] == '-') {  x = 2;  }  else if (b[i] == '<') {  x = 2;  }  else {  x = 1;  }  if (optr.top() == '!') {  y = 5;  }  else if (optr.top() == '^') {  y = 4;  }  else if (optr.top() == '|') {  y = 3;  }  else if (optr.top() == '-') {  y = 2;  }  else if (optr.top() == '<') {  y = 2;  }  else {  y = 1;  }  if (x - y > 0) {  pan = -1;  }  else {  pan = 1;  }  }  switch (pan) {  case -1:  optr.push(b[i]);  break;  case 0:  optr.pop();  break;  case 1:  char ch;  ch = optr.top();  optr.pop();  if (ch == '!') {  int aa = opnd.top();  aa = !aa;  opnd.top() = aa;  break;  }  else {  int aa, bb;  bb = opnd.top();  opnd.pop();  aa = opnd.top();  opnd.pop();  opnd.push(re(aa, bb, ch));  break;  }  }  if (pan == 1) {  continue;  }  if ((b[i] == '|') || (b[i] == '-')) {  i += 2;  }  else if ((b[i] == '^') || (b[i] == '!') || (b[i] == '(') || (b[i] == ')')) {  i++;  }  else {  i += 3;  }  }  // 将剩余的操作符和操作数依次出栈，计算表达式的值  while (optr.empty() != true) {  char ch;  ch = optr.top();  optr.pop();  if (ch == '!') {  int aa;  aa = opnd.top();  aa = !aa;  opnd.top() = aa;  }  else {  int bb, aa;  bb = opnd.top();  opnd.pop();  aa = opnd.top();  opnd.pop();  opnd.push(re(aa, bb, ch));  }  }  cout << opnd.top() << endl;  }  算法2的流程图如下: | |
| |  | | --- | | 算法分析 |   算法1:  该算法的时间复杂度为O(2^n \* m)，其中n表示变量个数，m表示表达式的长度。具体来说，枚举所有变量的取值组合需要枚举2^n个数，对于每个取值组合，需要遍历表达式并计算表达式的值，遍历表达式的时间复杂度为O(m)。因此，总时间复杂度为O(2^n \* m)。  空间复杂度主要取决于存储变量取值的空间，需要使用一个长度为n的数组来存储变量取值。因此，空间复杂度为O(n)。  综上所述，该算法的时间复杂度为O(2^n \* m)，空间复杂度为O(n)。  算法2:  时间复杂度分析：  枚举所有变量的取值组合需要枚举2^n个数，其中n表示变量个数。对于每个取值组合，需要遍历表达式并计算表达式的值。遍历表达式的时间复杂度为O(m)，其中m表示表达式的长度。因此，总时间复杂度为O(2^n \* m)。  空间复杂度分析：  空间复杂度主要取决于存储变量取值和操作数栈、操作符栈的空间。变量取值可以使用一个长度为n的数组来存储，操作数栈和操作符栈的最大长度为m/2，因此空间复杂度为O(m+n)。  综上所述，该算法的时间复杂度为O(2^n \* m)，空间复杂度为O(m+n)。 | |
| |  | | --- | | 项目小结 |   这个项目主要是考察对于逻辑表达式的计算和变量组合的枚举能力。通过实现这个项目，我对于逻辑表达式的计算有了更深入的了解，同时也加深了我对于二进制数的理解。在实现过程中，我也发现了一些需要注意的细节，例如表达式中可能存在多余的空格需要去除，输出表头时需要按照字典序输出变量名等。 | |

|  |  |
| --- | --- |
| 项目名称 | 单词检查 |
| 内容和目的 | 实现一个能够检查文本中拼写错误的程序。该程序可以读取文本文件，逐个单词进行拼写检查，并输出可能的正确拼写建议。 |
| |  | | --- | | 项目分析与总体设计 |   用顺序表的抽象数据类型定义如下：  ADT set{  基本操作：  bool isSimilar(const char\* word1, const char\* word2) // 判断两个单词是否相似  }  用二叉树的抽象数据类型定义如下：  ADT set{  基本操作：  void postorder(BiTree T) // 后序遍历二叉排序树  void Initial(BiTree& T) //初始化树  BiTree Creat(char\* a, BiTree T) //创建二叉排序树  void Issimilar(Sqlist S, int n, Sqlist L[])//判断单词是否正确并修改  } | |
| |  | | --- | | 数据结构和算法实现 |   算法1:  本算法主要使用了数组来存储字典和待检查的单词。对于每个待检查的单词，遍历整个字典，判断是否存在相同的单词，如果不存在则遍历整个字典，寻找与当前单词相似的单词作为可能的正确拼写建议。  在判断两个单词是否相似的函数中，首先判断两个单词的长度是否相差大于1，如果是，则不相似。如果长度相等，则最多只有一个字符不同。如果长度相差1，则可以通过插入或删除一个字符得到。具体实现中，使用两个指针i和j分别指向两个单词的字符，如果当前字符不同，则判断是否已经有不同的字符，如果有，则不相似，否则将指向较长单词的指针后移一位，继续比较下一个字符。  核心算法如下:  bool isSimilar(const char\* word1, const char\* word2) {  int len1 = strlen(word1);  int len2 = strlen(word2);  // 长度相差大于1，不相似  if (abs(len1 - len2) > 1) {  return false;  }  // 长度相等，最多有一个字符不同  if (len1 == len2) {  int diff = 0;  for (int i = 0; i < len1; ++i) {  if (word1[i] != word2[i]) {  ++diff;  if (diff > 1) {  return false;  }  }  }  return diff == 1;  }  // 长度相差1，可以通过插入或删除一个字符得到  if (len1 > len2) {  swap(word1, word2);  swap(len1, len2);  }  int i = 0, j = 0;  while (i < len1 && j < len2) {  if (word1[i] != word2[j]) {  if (i != j) {  return false;  }  ++j;  }  else {  ++i;  ++j;  }  }  return true;  }  算法1的流程图如下:  算法2:  本项目使用了二叉排序树来存储字典中的单词，并通过遍历字典来寻找与待检查单词相似的单词作为可能的正确拼写建议。具体实现中，使用了数组来存储字典和待检查的单词，对于每个待检查的单词，遍历整个字典，判断是否存在相同的单词，如果不存在则遍历整个字典，寻找与当前单词相似的单词作为可能的正确拼写建议。  定义的二叉树结构体如下:  //定义结构数组  typedef struct  {  int len;  char data[20];  }Sqlist;  // 定义二叉树节点  typedef struct Node {  char data[20];  Node\* left;  Node\* right;  }BiTNode, \* BiTree;  核心算法如下:  void Issimilar(Sqlist S, int n, Sqlist L[])  {  int i, f = 1;  for (i = 0; i < n; i++)  {  if (strcmp(L[i].data, S.data) == 0)  {  cout << S.data << " is correct" << endl;  f = 0;  break;  }  }  if (f == 1)//与字典不相等  {  cout << S.data << ":";  int diff = S.len - L[i].len;  for (i = 0; i < n; i++)  {  if (S.len - L[i].len == 1)//比字典多一个字母  {  int r = 0;//记录不相等的字母个数  for (int j = 0, k = 0; L[i].data[j] != '\0'; j++, k++)  {  if (L[i].data[j] != S.data[k])  {  r++;  j--;//比较字符后面的字母  }  if (r >= 2)break;//不相同的字母超过两个无法修改  }  if (r <= 1) cout << " " << L[i].data;  }  if (L[i].len - S.len == 1)//比字典少一个字母  {  int r = 0;  for (int j = 0, k = 0; S.data[k] != '\0'; j++, k++)  {  if (L[i].data[j] != S.data[k])  {  r++;  k--;//比较字典后面的字母  }  if (r >= 2)break;//不相同的字母超过两个无法修改  }  if (r <= 1) cout << " " << L[i].data;  }  if (S.len - L[i].len == 0)//与字典字母数相同  {  int r = 0;//记录不相等的字母个数  for (int j = 0, k = 0; L[i].data[j] != '\0'; j++, k++)  {  if (L[i].data[j] != S.data[k])  {  r++;  }  if (r >= 2)break;//不相同的字母超过两个无法修改  }  if (r <= 1) cout << " " << L[i].data;  }  }  cout << endl;  }  }  算法2流程图如下: | |
| |  | | --- | | 算法分析 |   算法1:  该算法的时间复杂度主要取决于遍历字典和检查单词的时间复杂度。具体来说，遍历字典需要遍历所有单词，时间复杂度为O(n)，其中n为字典中单词的个数。对于每个待检查的单词，需要遍历整个字典，时间复杂度为O(n)，因此总时间复杂度为O(n^2)。对于判断两个单词是否相似的函数，时间复杂度为O(m)，其中m为单词的长度。  空间复杂度主要取决于存储字典和待检查的单词的空间，需要使用两个二维数组来存储字典和待检查的单词，数组大小分别为MAX\_DICT\_WORDS \* (MAX\_WORD\_LEN + 1)和MAX\_CHECK\_WORDS \* (MAX\_WORD\_LEN + 1)，因此空间复杂度为O(n)，其中n为字典和待检查单词的总数。  综上所述，该算法的时间复杂度为O(n^2 \* m)，空间复杂度为O(n)。  算法2:  该算法主要使用了二叉排序树来存储字典中的单词，并使用后序遍历输出二叉排序树中的单词。对于输入的单词，先判断是否与字典中的单词相同，若相同则输出 "is correct"，否则在字典中查找与输入单词相似的单词并输出。相似的定义是：与字典单词字母数相同，但只有一个字母不同（增加、删除、替换一个字母），且最多只有一个字母不同。  时间复杂度分析：  创建二叉排序树的时间复杂度为 O(nlogn)，其中 n 是字典中单词的个数。  对于每个输入单词，需要在字典中查找相似单词并输出。假设字典中单词的平均长度为 m，那么查找相似单词的时间复杂度为 O(n\*m)。  因此，总时间复杂度为 O(nlogn + n\*m)。 | |
| |  | | --- | | 项目小结 |   通过本项目的实现，我深刻理解了数据结构和算法的重要性，以及如何根据不同的问题选择不同的数据结构和算法。同时，我也提高了自己的算法分析和设计能力，对于未来的学习和工作都有很大的帮助。 | |

|  |  |
| --- | --- |
| 项目名称 | 后缀表达式求值 |
| 内容和目的 | 实现一个后缀表达式求值的程序，输入一个后缀表达式，输出表达式的计算结果。 |
| |  | | --- | | 项目分析与总体设计 |   具体的算法步骤如下:  1.初始化一个栈。  2.遍历后缀表达式中的每个token，如果是操作数，则将其转换为整数并入栈；如果是操作符，则从栈中弹出两个操作数，进行相应的运算，并将结果入栈。  3.遍历完整个后缀表达式后，栈中只剩下一个元素，即为表达式的值。  4.返回栈顶元素作为表达式的值。  集合的抽象数据类型定义如下：  ADT set{  基本操作：  int evaluatePostfixExpression(string& expression)// 计算后缀表达式  int pop(Stack& s)//出栈  void push(Stack& s, int x)//入栈  bool isEmpty(Stack& s)//判断栈是否为空  }  . | |
| |  | | --- | | 数据结构和算法实现 |   采用栈的方法实现：  1.创建一个栈，用于存储操作数。  2.遍历后缀表达式的每一个元素，如果是操作数，则将其入栈；如果是运算符，则从栈中取出两个操作数进行计算，然后将计算结果入栈。  3.遍历完整个后缀表达式后，栈中只剩下一个元素，即为表达式的计算结果。  // 自定义栈结构  struct Stack {  int data[100000];  int top;  };  核心算法如下:  int evaluatePostfixExpression(string& expression) {  Stack s;  initStack(s);  stringstream ss(expression);  string token;  int result;  while (ss >> token) {  if (token == "+" || token == "-" || token == "\*" || token == "/") {  int b = pop(s);  int a = pop(s);  if (token == "+") {  result = a + b;  }  else if (token == "-") {  result = a - b;  }  else if (token == "\*") {  result = a \* b;  }  else {  result = a / b;  }  push(s, result);  }  else {  push(s, stoi(token));  }  }  return pop(s);  } | |
| |  | | --- | | 算法分析 |   该算法的时间复杂度为O(n)，空间复杂度为O(n)，其中n为后缀表达式的长度 | |
| |  | | --- | | 项目小结 |   本项目通过实现后缀表达式求值的算法，加深了对栈的理解和应用，同时也提高了代码实现能力。这个代码实现了后缀表达式求值的功能，使用了自定义栈的数据结构。算法的时间复杂度为O(n)，其中n为后缀表达式的长度. | |

|  |  |
| --- | --- |
| 项目名称 | 中缀表达式转后缀表达式 |
| 内容和目的 | 本项目的目的是实现一个中缀表达式转后缀表达式的程序，输入一个中缀表达式，输出对应的后缀表达式。 |
| |  | | --- | | 项目分析与总体设计 |   在本项目中，我们将使用栈来实现中缀表达式转换。具体算法如下：  1.从左到右遍历中缀表达式中的每个字符。  2.如果当前字符是操作数，则直接输出到后缀表达式中。  3.如果当前字符是左括号，则将其入栈。  4.如果当前字符是右括号，则弹出栈中的元素，直到遇到左括号为止，并将弹出的元素输出到后缀表达式中。  5.如果当前字符是运算符，则弹出所有优先级大于等于该运算符的栈顶元素，并将弹出的元素输出到后缀表达式中，然后将该运算符入栈。  6.遍历完中缀表达式后，将栈中剩余的元素依次弹出，并输出到后缀表达式中。  7.后缀表达式即为最终结果。  抽象数据类型定义如下：  ADT set{  基本操作：  bool isOperator(char c)// 判断一个字符是否为操作符  bool higherOrEqualPriority(char a, char b) // 判断两个操作符的优先级，a的优先级是否大于等于b的优先级  string infixToPostfix(string infix) // 中缀表达式转后缀表达式  } | |
| |  | | --- | | 数据结构和算法实现 |   定义栈的结构体如下:  struct Stack {  char data[100]; // 存储栈中的元素  int top; // 栈顶指针  核心算法如下:  string infixToPostfix(string infix) {  OperatorStack s; // 定义一个运算符栈  s.init(); // 初始化栈  string postfix; // 定义一个字符串，用于存储后缀表达式  // 遍历中缀表达式中的每个字符  for (int i = 0; i < infix.length(); i++) {  char ch = infix[i];  // 如果是操作数，直接输出到后缀表达式中  if (isalnum(ch)) {  postfix += ch;  }  // 如果是左括号，入栈  else if (ch == '(') {  s.push(ch);  }  // 如果是右括号，弹出栈中的元素，直到遇到左括号  else if (ch == ')') {  while (!s.isEmpty() && s.getTop() != '(') {  postfix += s.pop();  }  s.pop(); // 弹出左括号  }  // 如果是运算符  else if (isOperator(ch)) {  // 弹出所有优先级大于等于该运算符的栈顶元素  while (!s.isEmpty() && isOperator(s.getTop()) && higherOrEqualPriority(s.getTop(), ch)) {  postfix += s.pop();  }  // 将该运算符入栈  s.push(ch);  }  }  // 将栈中剩余的元素依次弹出，输出到后缀表达式中  while (!s.isEmpty()) {  postfix += s.pop();  }  return postfix;  } | |
| |  | | --- | | 算法分析 |   时间复杂度：遍历中缀表达式需要O(n)的时间，每个字符最多入栈一次，弹出栈的次数也不会超过入栈的次数，因此时间复杂度为O(n)。  空间复杂度：需要一个栈来存储运算符，因此空间复杂度为O(n)。 | |
| |  | | --- | | 项目小结 |   通过本项目的实现，我们掌握了栈的基本操作和中缀表达式转换的算法。在实现中缀表达式转后缀表达式的过程中，我也学会了如何判断一个字符是否为操作符，如何判断两个操作符的优先级，以及如何将中缀表达式转换为后缀表达式。 | |

|  |  |
| --- | --- |
| 项目名称 | 二叉树的创建和文本显示 |
| 内容和目的 | 本项目旨在通过输入字符串来创建二叉树，并将其以镜像中序遍历的方式输出。 |
| |  | | --- | | 项目分析与总体设计 |   1.定义一个二叉树的结构体TNode，其中包含了一个字符数组data，以及左右子树的指针lchild和rchild。  2.定义一个Create\_T函数，用来根据输入的字符串来创建二叉树。具体实现方式为：先判断当前字符是否为空，若为空则该节点的左右子树均为空；否则创建一个新节点，并将该节点的data字段赋值为当前字符，然后递归创建左右子树。  3.定义一个MINTrave函数，用来对二叉树进行镜像中序遍历，并输出到控制台。具体实现方式为：先递归遍历右子树，然后输出当前节点的data字段，并递归遍历左子树。在输出当前节点的data字段时，为了使输出结果更加美观，每层先输出4倍层数的空格。  4.在主函数中，通过gets函数读取输入的字符串，然后调用Create\_T函数来创建二叉树，并将其以镜像中序遍历的方式输出.  抽象数据类型定义如下：  ADT set{  基本操作：  void ini\_T(Tree& T)//初始化二叉树  void Create\_T(Tree& t)//创建二叉树  void MINTrave(Tree t, int dep) //镜像中序遍历二叉树  } | |
| |  | | --- | | 数据结构和算法实现 |   定义的二叉树结构如下:  typedef struct T  {  char data[4];  struct T\* rchild, \* lchild;  } TNode, \* Tree;  核心算法如下:  void MINTrave(Tree t, int dep) //镜像中序遍历二叉树  {  if (t)  {  MINTrave(t->rchild, dep + 1); //先遍历右子树  int n = dep \* 4;  for (int i = 0; i < n; ++i) //每层先输出4倍层数的空格  cout << ' ';  cout << t->data << endl;  MINTrave(t->lchild, dep + 1);  }  } | |
| |  | | --- | | 算法分析 |   时间复杂度主要取决于二叉树的高度，如果二叉树是一棵平衡树，则时间复杂度为O(logn)，否则时间复杂度为O(n)。空间复杂度主要取决于二叉树的节点数，为O(n)。 | |
| |  | | --- | | 项目小结 |   通过本项目的实现，我掌握了二叉树的创建和镜像中序遍历的算法，并学会了如何使用字符串来表示二叉树。这对于我理解和实现二叉树相关的算法和数据结构具有重要意义。 | |

|  |  |
| --- | --- |
| 项目名称 | 表达式树的创建与输出 |
| 内容和目的 | 本项目的目的是实现一个表达式树的创建与输出程序，输入一个表达式，输出对应的表达式树。 |
| |  | | --- | | 项目分析与总体设计 |   具体算法如下：  1.从左到右遍历表达式中的每个字符。  2.如果当前字符是操作数，则创建一个新节点，并将其作为叶子节点插入到表达式树中。  3.如果当前字符是运算符，则创建一个新节点，并将其作为根节点插入到表达式树中，然后递归创建左子树和右子树。  4.遍历完表达式后，表达式树即为最终结果。  抽象数据类型定义如下：  ADT set{  基本操作：  void CreatBitree(bitree& T, int y, int num)//创建二叉树  void Print(bitree T, bitree parent)//打印  } | |
| |  | | --- | | 数据结构和算法实现 |   定义的表达式树结构如下:  typedef struct binode  {  char data[4];  int h;  int depth;  struct binode\* lchild, \* rchild;  } binode, \* bitree;  核心算法如下:  void CreatBitree(bitree& T, int y, int num)//创建二叉树  {  if (d[q][0] == '#') // 如果是叶子节点  {  T = NULL;  q++;  }  else  {  T = new binode; // 创建新节点  if (y == 1)  {  T->h = 1; // 根节点高度为1  }  else  {  T->h = 0; // 其他节点高度为0  }  T->depth = ++num; // 记录深度  strcpy(T->data, d[q++]); // 复制节点数  CreatBitree(T->lchild, 1, T->depth); // 创建左子树  CreatBitree(T->rchild, 0, T->depth); // 创建右子树  }  }  void Print(bitree T, bitree parent)//打印  {  if (T != NULL)  {  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') // 如果是运算符节点  {  cout << "(";  Print(T->lchild, T); // 遍历左子树  cout << T->data; // 输出当前节点  Print(T->rchild, T); // 遍历右子树  cout << ")";    }  else // 如果是数字节点  {  cout << T->data;  }  }  } | |
| |  | | --- | | 算法分析 |   时间复杂度：遍历表达式需要O(n)的时间，每个字符最多入栈一次，弹出栈的次数也不会超过入栈的次数，因此时间复杂度为O(n)。  空间复杂度：需要一个栈来存储运算符，因此空间复杂度为O(n)。 | |
| |  | | --- | | 项目小结 |   通过本项目的实现，我们掌握了递归创建表达式树和中序遍历输出表达式树的算法。在实现表达式树的过程中，我也学会了如何处理表达式中的数字和运算符，以及如何创建表达式树和输出表达式树。 | |

|  |  |
| --- | --- |
| 项目名称 | 表达式树的值 |
| 内容和目的 | 本项目旨在实现表达式树的创建、中序遍历和计算值的功能 |
| |  | | --- | | 项目分析与总体设计 |   具体算法如下:  1.定义二叉树结构体，包括节点数据、节点高度、节点深度、左右子树指针。  2.定义全局变量d、q、num1、sum，其中d用于存储输入的字符串，q表示当前字符串的行数，num1表示左右子树高度，sum表示计算结果。  3.实现创建二叉树的函数creatbitree，递归创建左右子树，并计算节点高度和深度。  4.实现中序遍历表达式树的函数travel，遍历左子树、输出当前节点的数据、遍历右子树，并在遍历运算符节点时加上括号。  5.实现计算表达式树的值的函数calculate，递归计算左右子树的值，并根据当前节点的运算符进行相应的运算。  6.在主函数中，逐行读取输入的字符串，将字符串按空格分割成多个字符串，调用creatbitree创建表达式树，调用travel中序遍历表达式树并输出，调用calculate计算表达式树的值并输出。  ADT set{  基本操作:  void creatbitree(bitree& T, int y, int num)// 创建二叉树  void travel(bitree T) // 中序遍历表达式树  int calculate(bitree T) // 计算表达式树的值  } | |
| |  | | --- | | 数据结构和算法实现 |   具体算法如下:  1.定义二叉树结构体，包括节点数据、节点高度、节点深度、左右子树指针。  2.定义全局变量d、q、num1、sum，其中d用于存储输入的字符串，q表示当前字符串的行数，num1表示左右子树高度，sum表示计算结果。  3.实现创建二叉树的函数creatbitree，递归创建左右子树，并计算节点高度和深度。  4.实现中序遍历表达式树的函数travel，遍历左子树、输出当前节点的数据、遍历右子树，并在遍历运算符节点时加上括号。  5.实现计算表达式树的值的函数calculate，递归计算左右子树的值，并根据当前节点的运算符进行相应的运算。  6.在主函数中，逐行读取输入的字符串，将字符串按空格分割成多个字符串，调用creatbitree创建表达式树，调用travel中序遍历表达式树并输出，调用calculate计算表达式树的值并输出。  定义的二叉树结构如下:  typedef struct binode {  char data[4]; // 存储节点数据  int h; // 存储节点高度  int depth; // 存储节点深度  struct binode\* lchild, \* rchild; // 左右子树指针  } binode, \* bitree;  核心算法如下:  void travel(bitree T) // 中序遍历表达式树  {  if (T != NULL) {  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') { // 如果当前节点是运算符  cout << "(";  travel(T->lchild); // 遍历左子树  cout << T->data; // 输出当前节点的数据  travel(T->rchild); // 遍历右子树  cout << ")";  }  else { // 如果当前节点是操作数  cout << T->data;  }  }  }  int calculate(bitree T) // 计算表达式树的值  {  if (T != NULL) {  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') { // 如果当前节点是运算符  switch (T->data[0]) {  case '+':  return (calculate(T->lchild) + calculate(T->rchild));  break;  case '-':  return (calculate(T->lchild) - calculate(T->rchild));  break;  case '\*':  return (calculate(T->lchild) \* calculate(T->rchild));  break;  case '/':  return (calculate(T->lchild) / calculate(T->rchild));  break;  }  }  else { // 如果当前节点是操作数  char a[1000];  strcpy(a, T->data);  int i, sum1 = 0;  for (i = 0; a[i] != '\0'; i++) {  sum1 = sum1 \* 10 + a[i] - '0'; // 将操作数字符串转换为整型  }  return sum1;  }  }  } | |
| |  | | --- | | 算法分析 |   时间复杂度：创建二叉树的时间复杂度为O(n)，其中n为节点个数；中序遍历表达式树的时间复杂度为O(n)，其中n为节点个数；计算表达式树的值的时间复杂度为O(n)，其中n为节点个数。因此，总时间复杂度为O(n)。  空间复杂度：创建二叉树的空间复杂度为O(n)，其中n为节点个数；中序遍历表达式树的空间复杂度为O(h)，其中h为树的高度；计算表达式树的值的空间复杂度为O(h)，其中h为树的高度。因此，总空间复杂度为O(n)。 | |
| |  | | --- | | 项目小结 |   本项目主要涉及二叉树的创建、遍历和计算表达式树的值。通过实现该项目，我学习了二叉树的相关知识，掌握了二叉树的创建、遍历和计算表达式树的值的方法。 | |

|  |  |
| --- | --- |
| 项目名称 | 24点游戏(Ⅰ) |
| 内容和目的 | 本项目的目的是实现24点游戏，即给定4个数字，通过加、减、乘、除等运算，计算出24。 |
| |  | | --- | | 项目分析与总体设计 |   具体的算法步骤如下:  1.定义二叉树节点结构体BiTNode和二叉树指针类型BiTree。  2.定义Print函数，用于打印表达式树的中缀表达式。  3.定义CreatTree函数，用于创建表达式树。  4.定义Cal函数，用于计算表达式树的值。  5.在main函数中，通过循环读入多个表达式，对每个表达式进行如下操作：  a. 创建一个新节点，将读入的第一个字符赋值给该节点的数据域。  b. 递归创建该节点的左右子树。  c. 计算表达式树的值，如果在23.999999~24.000001之间，输出表达式和结果；否则输出NO。  抽象数据类型定义如下：  ADT set{  基本操作：  void Print(BiTree T)//打印数字和符号  void CreatTree(BiTree& T) //创建表达式树  double Cal(BiTree T) //计算表达式树的值  } | |
| |  | | --- | | 数据结构和算法实现 |   具体的算法步骤如下:  1.定义二叉树节点结构体BiTNode和二叉树指针类型BiTree。  2.定义Print函数，用于打印表达式树的中缀表达式。  3.定义CreatTree函数，用于创建表达式树。  4.定义Cal函数，用于计算表达式树的值。  5.在main函数中，通过循环读入多个表达式，对每个表达式进行如下操作：  a. 创建一个新节点，将读入的第一个字符赋值给该节点的数据域。  b. 递归创建该节点的左右子树。  c. 计算表达式树的值，如果在23.999999~24.000001之间，输出表达式和结果；否则输出NO。  定义的表达式树结构如下:  typedef struct Node  {  string data;  Node\* left, \* right;  }BiTNode, \* BiTree;  核心算法如下:  double Cal(BiTree T) //计算表达式树的值  {  double ret = 0;  double num1, num2;  if (T != NULL) //如果当前节点不为空  {  double num1 = Cal(T->left); //递归计算左子树的值  double num2 = Cal(T->right); //递归计算右子树的值  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') //如果当前节点是运算符  {  switch (T->data[0]) //根据运算符计算结果  {  case '+':  return num1 + num2;  case'-':  return num1 - num2;  case'\*':  return num1 \* num2;  case'/':  if (num2 - 0 >= 0.000001) //除数不为0  return num1 / num2;  default:  break;  }  }  else //如果当前节点是数字  {  int i;  string a;  a = T->data;  double ret = 0;  for (i = 0; a[i] != '\0'; i++)  ret = ret \* 10.00 + a[i] - '0' + 0.0; //将数字字符串转化为数字  return ret;  }  }  } | |
| |  | | --- | | 算法分析 |   时间复杂度：Print函数、CreatTree函数和Cal函数都采用了递归的方式，时间复杂度为O(n)，其中n为表达式树中节点的个数。  空间复杂度：由于需要创建表达式树，空间复杂度为O(n)，其中n为表达式树中节点的个数。 | |
| |  | | --- | | 项目小结 |   本题是一道典型的表达式树应用题，需要掌握表达式树的创建、遍历和计算方法，同时需要注意浮点数比较的精度问题。让我更加注重浮点数比较的精度 | |

**附录:**

|  |  |
| --- | --- |
| 项目名称 | 复杂度分析 |
| 代码 | 复杂度分析(I) #include<iostream>  using namespace std;  int main()  {  long long n;  while (~scanf("%lld", &n))  {  if (n > 2)  cout << ((n - 3) \* (n - 2) \* (2 \* n - 5) / 6 + (n - 3) \* (n - 2) / 2) / 2 << " " << n - 1 + n - 2 + n << endl;  else  cout << "0 RANDOM" << endl;  } }复杂度分析(II) #include<iostream>  using namespace std;  int main()  {  long long n;  while (cin >> n)  {  if (n > 1)  cout << (n + 1) \* n \* (n - 1) / 6 << " " << 3 \* (n + 1) << endl;  else  cout << "0 RANDOM" << endl;  }  return 0; } |

|  |  |
| --- | --- |
| 项目名称 | Josephus问题 |
| 代码 | Josephus问题(I) #include<iostream>  using namespace std;  int main()  {  int i, j, n, cnt = 0, t = 0, m;  while (cin >> n)  {  int a[1001];  for (i = 1; i <= n; i++)  {  a[i] = 1;//初始化  }  for (i = 1; i <= n; i++)  {  for (j = 1; j <= n; j++)  {  if (a[j] != 0 && cnt == 0)//记录下标  {  cnt++;  m = j;//记录下标  }  else if (a[j] != 0)  {  cnt++;  }  if (cnt == 2)//到2时结束  {  cnt = 0;  a[j] = 0;  t++;//轮数  }  if (t == n - 1)  {  cout << m << endl;  break;  }  }  if (t == n - 1)  {  t = 0;  break;  }  }  }  return 0; }Josephus问题(II) #include<iostream>  using namespace std;  int main()  {  long long n, i, cnt = 1, flag;  while (cin >> n)  {  flag = 1;  if (n == 1)  {  cout << "1" << endl;  flag = 0;  }  for (i = 1; i <= n; i++)  {  if (n == 2 \* cnt)//找到n的范围  {  cout << "1" << endl;  flag = 0;  break;  }  else if (n < 2 \* cnt)//2的倍数输出1  {  cnt /= 2;  break;//n不为2的倍数  }  else  {  cnt \*= 2;  }  }  if (flag == 1)  {  cout << (n - 2 \* cnt) \* 2 + 1 << endl;  }  cnt = 1;  }  return 0; } |

|  |  |
| --- | --- |
| 项目名称 | 真值表 |
| 代码 | 真值表(I) #include<iostream>  #include<string>  #include<cstring>  using namespace std;  int Input(char str[])//输出符号和字母  {  int i,p=0;  int c[1000] = { 0 };  for (i = 0; i<strlen(str); i++)  {  if (str[i] == '!' || str[i] == '(' || str[i] == ')' || str[i] == '^')  {  cout << str[i] << " ";  }  else if (str[i] >= 'a' &&str[i] <= 'z')  {  int b = str[i];  if (c[b - 97] == 0)  {  c[b - 97] = 1;  p++; //2的指数,真值表取值有2^n次  }  cout << str[i] << " ";  }  else if (str[i] == '|' || str[i] == '-')  {  cout << str[i] << str[i + 1] << " ";  i++;  }  else if (str[i] == '<')  {  cout << str[i] << str[i + 1] << str[i + 2] << " ";  i += 2;  }  }  cout << endl;  int n = 1;  for (i = 0; i < 26; i++)  {  if (c[i] == 1 && n < p)  {  printf("%c ", i + 97);  n++;  }  else if (c[i] == 1 && n == p)  {  printf("%c", i + 97);  }  }  cout << endl;  return p;//返回指数  }  void Calculate(int p,int ret)//输出所有变量组合  {  int i, k;  int a[1000] = { 0 };  int flag = 0;  while (ret)  {  a[flag++] = ret % 2;  ret /= 2;  }  for (i = p - 1; i >= 0; i--)  {  cout << a[i];  if (i != 0)  {  cout << " ";  }  }  cout << endl;  }  int main()  {  char str[100];  int p,i;  while (cin >> str)  {  int ret = 1;  p=Input(str);  for (i = 0; i < p; i++)  {  ret \*= 2;  }  ret -= 1;  while (ret >= 0)  {  Calculate(p, ret);  ret--;  }  }  return 0;  } 真值表(II) #include <bits/stdc++.h>  #define MAXSIZE 201  using namespace std;  char str[MAXSIZE];  // 计算逻辑运算结果  int re(int aa, int bb, char ch)  {  if (ch == '|') {  if (aa == 0 && bb == 0) {  return 0;  }  else {  return 1;  }  }  else if (ch == '^') {  if (aa == 1 && bb == 1) {  return 1;  }  else {  return 0;  }  }  else if (ch == '-') {  if (aa == 1 && bb == 1) {  return 1;  }  else if (aa == 1 && bb == 0) {  return 0;  }  else {  return 1;  }  }  else if (ch == '<') {  if (aa == 1 && bb == 1) {  return 1;  }  else if (aa == 0 && bb == 0) {  return 1;  }  else {  return 0;  }  }  else if (ch == '&') {  if (aa == 1 && bb == 1) {  return 1;  }  else {  return 0;  }  }  return 0;  }  // 统计变量个数  int count\_var(char str[])  {  int a[26] = { 0 };  int count = 0;  for (int i = 0; str[i] != '\0'; i++) {  if (str[i] >= 'a' && str[i] <= 'z') {  int b = str[i];  if (a[b - 97] == 0) {  a[b - 97] = 1;  count++;  }  }  }  return count;  }  // 输出表头  void print\_header(int a[])  {  for (int i = 0; i <= 25; i++) {  if (a[i] == 1) {  cout << (char)(i + 97) << " ";  }  }  cout << str << " ";  cout << endl;  }  // 计算表达式的值  void cal(int sum, int j, char b[], char alg[])  {  int i;  int a[15] = { 0 };  int flag = 0;  // 将当前的sum转化为二进制，并存入数组a中  while (sum / 2) {  a[flag] = sum % 2;  flag++;  sum /= 2;  }  a[flag] = sum;  flag++;  // 输出当前变量取值的组合  for (i = j - 1; i >= 0; i--) {  cout << a[i] << " ";  }  stack<int> opnd; // 操作数栈  stack<char> optr; // 操作符栈  sort(alg, alg + j); // 将变量按字典序排序  for (i = 0; i < strlen(b);) {  // 如果当前字符为变量，将对应的取值入栈  if (b[i] >= 'a' && b[i] <= 'z') {  for (int jj = 0; jj < j; jj++) {  if (b[i] == alg[jj]) {  opnd.push(a[j - jj - 1]);  }  }  i++;  continue;  }  // 如果操作符栈为空且当前字符不为变量，将当前字符入栈  if ((optr.empty() == true) && (!(b[i] >= 'a' && b[i] <= 'z'))) {  optr.push(b[i]);  if (b[i] == '|' || b[i] == '-') {  i += 2;  }  else if (b[i] == '^' || b[i] == '!' || b[i] == '(' || b[i] == ')') {  i++;  }  else if (b[i] == '<') {  i += 3;  }  continue;  }  // 如果操作数栈为空且当前字符为'!'，将'!'入栈  if (opnd.empty() == true && b[i] == '!') {  optr.push(b[i]);  i++;  continue;  }  int pan, x, y;  // 判断当前字符与操作符栈顶元素的优先级关系  if ((optr.top() == '(') && (b[i] == ')')) {  pan = 0;  }  else if (b[i] == ')') {  pan = 1;  }  else if (optr.top() == '(') {  pan = -1;  }  else if (b[i] == '(') {  pan = -1;  }  else {  if (b[i] == '!') {  x = 5;  }  else if (b[i] == '^') {  x = 4;  }  else if (b[i] == '|') {  x = 3;  }  else if (b[i] == '-') {  x = 2;  }  else if (b[i] == '<') {  x = 2;  }  else {  x = 1;  }  if (optr.top() == '!') {  y = 5;  }  else if (optr.top() == '^') {  y = 4;  }  else if (optr.top() == '|') {  y = 3;  }  else if (optr.top() == '-') {  y = 2;  }  else if (optr.top() == '<') {  y = 2;  }  else {  y = 1;  }  if (x - y > 0) {  pan = -1;  }  else {  pan = 1;  }  }  switch (pan) {  case -1:  optr.push(b[i]);  break;  case 0:  optr.pop();  break;  case 1:  char ch;  ch = optr.top();  optr.pop();  if (ch == '!') {  int aa = opnd.top();  aa = !aa;  opnd.top() = aa;  break;  }  else {  int aa, bb;  bb = opnd.top();  opnd.pop();  aa = opnd.top();  opnd.pop();  opnd.push(re(aa, bb, ch));  break;  }  }  if (pan == 1) {  continue;  }  if ((b[i] == '|') || (b[i] == '-')) {  i += 2;  }  else if ((b[i] == '^') || (b[i] == '!') || (b[i] == '(') || (b[i] == ')')) {  i++;  }  else {  i += 3;  }  }  // 将剩余的操作符和操作数依次出栈，计算表达式的值  while (optr.empty() != true) {  char ch;  ch = optr.top();  optr.pop();  if (ch == '!') {  int aa;  aa = opnd.top();  aa = !aa;  opnd.top() = aa;  }  else {  int bb, aa;  bb = opnd.top();  opnd.pop();  aa = opnd.top();  opnd.pop();  opnd.push(re(aa, bb, ch));  }  }  cout << opnd.top() << endl;  }  int main()  {  while (cin.getline(str, MAXSIZE)) {  int count = count\_var(str); // 统计变量个数  int a[26] = { 0 };  char alg[15];  char b[200];  int j = 0, k = 0;  // 将变量按字典序存入alg数组，并记录每个变量是否出现  for (int i = 0; str[i] != '\0'; i++) {  if (str[i] != ' ') {  b[k] = str[i];  k++;  }  if (str[i] >= 'a' && str[i] <= 'z') {  int b = str[i];  if (a[b - 97] == 0) {  a[b - 97] = 1;  alg[j] = str[i];  j++;  }  }  }  b[k] = '\0';  print\_header(a); // 输出表头  // 枚举所有变量取值的组合，并计算表达式的值  int sum = pow(2, count) - 1;  while (sum >= 0) {  cal(sum, count, b, alg);  sum--;  }  }  return 0; } |

|  |  |
| --- | --- |
| 项目名称 | 单词检查 |
| 代码 | 单词检查(I) #include <iostream>  #include <cstring>  using namespace std;  const int MAX\_DICT\_WORDS = 10000;  const int MAX\_CHECK\_WORDS = 50;  const int MAX\_WORD\_LEN = 15;  // 判断两个单词是否相似  bool isSimilar(const char\* word1, const char\* word2) {  int len1 = strlen(word1);  int len2 = strlen(word2);  // 长度相差大于1，不相似  if (abs(len1 - len2) > 1) {  return false;  }  // 长度相等，最多有一个字符不同  if (len1 == len2) {  int diff = 0;  for (int i = 0; i < len1; ++i) {  if (word1[i] != word2[i]) {  ++diff;  if (diff > 1) {  return false;  }  }  }  return diff == 1;  }  // 长度相差1，可以通过插入或删除一个字符得到  if (len1 > len2) {  swap(word1, word2);  swap(len1, len2);  }  int i = 0, j = 0;  while (i < len1 && j < len2) {  if (word1[i] != word2[j]) {  if (i != j) {  return false;  }  ++j;  }  else {  ++i;  ++j;  }  }  return true;  }  int main() {  char dictionary[MAX\_DICT\_WORDS][MAX\_WORD\_LEN + 1];  char checkWords[MAX\_CHECK\_WORDS][MAX\_WORD\_LEN + 1];  int dictSize = 0, checkSize = 0;  // 读取字典  while (cin.getline(dictionary[dictSize], MAX\_WORD\_LEN + 1)) {  if (strcmp(dictionary[dictSize], "#") == 0) {  break;  }  ++dictSize;  }  // 读取待检查的单词  while (cin.getline(checkWords[checkSize], MAX\_WORD\_LEN + 1)) {  if (strcmp(checkWords[checkSize], "#") == 0) {  break;  }  ++checkSize;  }  // 检查单词  for (int i = 0; i < checkSize; ++i) {  bool isCorrect = false;  bool hasSuggestion = false;  cout << checkWords[i];  for (int j = 0; j < dictSize; ++j) {  if (strcmp(checkWords[i], dictionary[j]) == 0) {  isCorrect = true;  break;  }  }  if (isCorrect) {  cout << " is correct" << endl;  }  else {  cout << ':';  for (int j = 0; j < dictSize; ++j) {  if (isSimilar(checkWords[i], dictionary[j])) {  cout << ' ' << dictionary[j];  hasSuggestion = true;  }  }  if (!hasSuggestion) {  cout << endl;  }  else {  cout << endl;  }  }  }  return 0; }单词检查(II) #include <iostream>  #include <cstring>  using namespace std;  //定义结构数组  typedef struct  {  int len;  char data[20];  }Sqlist;  // 定义二叉树节点  typedef struct Node {  char data[20];  Node\* left;  Node\* right;  }BiTNode, \* BiTree;  // 后序遍历二叉排序树  void postorder(BiTree T)  {  if (T != NULL) {  postorder(T->left);  postorder(T->right);  cout << T->data << " ";  }  }  //初始化树  void Initial(BiTree& T)  {  T = NULL;  }  //创建二叉排序树  BiTree Creat(char\* a, BiTree T)  {  if (T == NULL)  {  T = new BiTNode;  strcpy(T->data, a);  T->left = NULL, T->right = NULL;  }  else if (strcmp(a, T->data) < 0)  {  T->left = Creat(a, T->left);//小于根，置于左子树  }  else if (strcmp(a, T->data) > 0)  {  T->right = Creat(a, T->right);  }  return T;  }  //判断单词是否正确并修改  void Issimilar(Sqlist S, int n, Sqlist L[])  {  int i, f = 1;  for (i = 0; i < n; i++)  {  if (strcmp(L[i].data, S.data) == 0)  {  cout << S.data << " is correct" << endl;  f = 0;  break;  }  }  if (f == 1)//与字典不相等  {  cout << S.data << ":";  int diff = S.len - L[i].len;  for (i = 0; i < n; i++)  {  if (S.len - L[i].len == 1)//比字典多一个字母  {  int r = 0;//记录不相等的字母个数  for (int j = 0, k = 0; L[i].data[j] != '\0'; j++, k++)  {  if (L[i].data[j] != S.data[k])  {  r++;  j--;//比较字符后面的字母  }  if (r >= 2)break;//不相同的字母超过两个无法修改  }  if (r <= 1) cout << " " << L[i].data;  }  if (L[i].len - S.len == 1)//比字典少一个字母  {  int r = 0;  for (int j = 0, k = 0; S.data[k] != '\0'; j++, k++)  {  if (L[i].data[j] != S.data[k])  {  r++;  k--;//比较字典后面的字母  }  if (r >= 2)break;//不相同的字母超过两个无法修改  }  if (r <= 1) cout << " " << L[i].data;  }  if (S.len - L[i].len == 0)//与字典字母数相同  {  int r = 0;//记录不相等的字母个数  for (int j = 0, k = 0; L[i].data[j] != '\0'; j++, k++)  {  if (L[i].data[j] != S.data[k])  {  r++;  }  if (r >= 2)break;//不相同的字母超过两个无法修改  }  if (r <= 1) cout << " " << L[i].data;  }  }  cout << endl;  }  }  int main()  {  int i = 0, n = 0;  Sqlist S;  Sqlist L[10001];  BiTree T;  Initial(T);  while (cin >> L[i].data)  {  if (L[i].data[0] == '#')break;  L[i].len = strlen(L[i].data);  T = Creat(L[i].data, T);  i++;  n++;  }  postorder(T);  cout << endl;  while (cin >> S.data)  {  if (S.data[0] == '#')break;  S.len = strlen(S.data);  Issimilar(S, n, L);  }  return 0;  } |

|  |  |
| --- | --- |
| 项目名称 | 后缀表达式求值 |
| 代码 | #include <iostream>  #include <string>  using namespace std;  // 自定义栈结构  struct Stack {  int data[100000];  int top;  };  // 初始化栈  void initStack(Stack& s) {  s.top = -1;  }  // 判断栈是否为空  bool isEmpty(Stack& s) {  return s.top == -1;  }  // 入栈  void push(Stack& s, int x) {  s.data[++s.top] = x;  }  // 出栈  int pop(Stack& s) {  if (!isEmpty(s)) {  return s.data[s.top--];  }  return -1;  }  // 判断一个字符是否为操作符  bool isOperator(char c) {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  // 计算后缀表达式  int evaluatePostfixExpression(string& expression) {  Stack s;  initStack(s);  int i = 0;  while (i < expression.length()) {  string token;  // 获取下一个token  while (i < expression.length() && expression[i] != ' ') {  token += expression[i];  i++;  }  i++; // 跳过空格  if (token == "+" || token == "-" || token == "\*" || token == "/") {  int b = pop(s);  int a = pop(s);  if (token == "+") {  push(s, a + b);  }  else if (token == "-") {  push(s, a - b);  }  else if (token == "\*") {  push(s, a \* b);  }  else {  push(s, a / b);  }  }  else {  push(s, stoi(token));  }  }  return pop(s);  }  int main() {  string expression;  getline(cin, expression);  expression.erase(expression.find('@')); // 去掉'@'符号  int result = evaluatePostfixExpression(expression);  cout << result << endl;  return 0; } |

|  |  |
| --- | --- |
| 项目名称 | 中缀表达式转后缀表达式 |
| 代码 | #include <iostream>  #include <string>  using namespace std;  // 定义一个运算符栈的结构体  struct OperatorStack {  char data[100]; // 存储栈中的元素  int top; // 栈顶指针  // 初始化栈  void init() {  top = -1;  }  // 判断栈是否为空  bool isEmpty() {  return top == -1;  }  // 判断栈是否已满  bool isFull() {  return top == 99;  }  // 入栈  void push(char ch) {  if (!isFull()) {  top++;  data[top] = ch;  }  }  // 出栈  char pop() {  char ch = '\0';  if (!isEmpty()) {  ch = data[top];  top--;  }  return ch;  }  // 获取栈顶元素  char getTop() {  char ch = '\0';  if (!isEmpty()) {  ch = data[top];  }  return ch;  }  };  // 判断一个字符是否为操作符  bool isOperator(char c) {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  // 判断两个操作符的优先级，a的优先级是否大于等于b的优先级  bool higherOrEqualPriority(char a, char b) {  int priorityA, priorityB;  if (a == '+' || a == '-') {  priorityA = 1;  }  else {  priorityA = 2;  }  if (b == '+' || b == '-') {  priorityB = 1;  }  else {  priorityB = 2;  }  return priorityA >= priorityB;  }  // 中缀表达式转后缀表达式  string infixToPostfix(string infix) {  OperatorStack s; // 定义一个运算符栈  s.init(); // 初始化栈  string postfix; // 定义一个字符串，用于存储后缀表达式  // 遍历中缀表达式中的每个字符  for (int i = 0; i < infix.length(); i++) {  char ch = infix[i];  // 如果是操作数，直接输出到后缀表达式中  if (isalnum(ch)) {  postfix += ch;  }  // 如果是左括号，入栈  else if (ch == '(') {  s.push(ch);  }  // 如果是右括号，弹出栈中的元素，直到遇到左括号  else if (ch == ')') {  while (!s.isEmpty() && s.getTop() != '(') {  postfix += s.pop();  }  s.pop(); // 弹出左括号  }  // 如果是运算符  else if (isOperator(ch)) {  // 弹出所有优先级大于等于该运算符的栈顶元素  while (!s.isEmpty() && isOperator(s.getTop()) && higherOrEqualPriority(s.getTop(), ch)) {  postfix += s.pop();  }  // 将该运算符入栈  s.push(ch);  }  }  // 将栈中剩余的元素依次弹出，输出到后缀表达式中  while (!s.isEmpty()) {  postfix += s.pop();  }  return postfix;  }  int main() {  string infix;  cin >> infix;  cout << infixToPostfix(infix) << endl;  return 0; } |

|  |  |
| --- | --- |
| 项目名称 | 二叉树的创建和文本显示 |
| 代码 | #include <iostream>  #include <cstring>  using namespace std;  typedef struct T  {  char data[4];  struct T\* rchild, \* lchild;  } TNode, \* Tree;  char a[100][4]; //模拟树的每层的字符  int depth = 0;  void ini\_T(Tree& T)//初始化二叉树  {  T = new TNode;  T->lchild = NULL;  T->rchild = NULL;  }  void Create\_T(Tree& t)//创建二叉树  {  if (a[depth][0] == '#')  {  t = NULL;  depth++; //当一个结点为空时意味着该结点的父节点创建成功，深度+1  }  else  {  t = new TNode;  strcpy(t->data, a[depth++]);  Create\_T(t->lchild);  Create\_T(t->rchild);  }  }  void MINTrave(Tree t, int dep) //镜像中序遍历二叉树  {  if (t)  {  MINTrave(t->rchild, dep + 1); //先遍历右子树  int n = dep \* 4;  for (int i = 0; i < n; ++i) //每层先输出4倍层数的空格  cout << ' ';  cout << t->data << endl;  MINTrave(t->lchild, dep + 1);  }  }  int main()  {  char b[1000];  while (gets(b) != NULL)  {  Tree T;  ini\_T(T);  int i, p = 0, q = 0;  for (i = 0; b[i]; ++i)  {  if (b[i] != ' ')  a[p][q++] = b[i];  else //遇到空格时，该结点创建完毕  {  a[p++][q] = '\0'; //p++ 移到下一个结点  q = 0;  }  }  a[p++][q++] = '\0';  Create\_T(T);  MINTrave(T, 0);  depth = 0; //将深度更新为0  cout << endl;  }  return 0; } |

|  |  |
| --- | --- |
| 项目名称 | 表达式树的创建与输出 |
| 代码 | #include <iostream>  #include <cstring>  using namespace std;  typedef struct binode  {  char data[4];  int h;  int depth;  struct binode\* lchild, \* rchild;  } binode, \* bitree;  char d[100][100];  int q = 0, num1;  void CreatBitree(bitree& T, int y, int num)//创建二叉树  {  if (d[q][0] == '#') // 如果是叶子节点  {  T = NULL;  q++;  }  else  {  T = new binode; // 创建新节点  if (y == 1)  {  T->h = 1; // 根节点高度为1  }  else  {  T->h = 0; // 其他节点高度为0  }  T->depth = ++num; // 记录深度  strcpy(T->data, d[q++]); // 复制节点数  CreatBitree(T->lchild, 1, T->depth); // 创建左子树  CreatBitree(T->rchild, 0, T->depth); // 创建右子树  }  }  void Print(bitree T, bitree parent)//打印  {  if (T != NULL)  {  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') // 如果是运算符节点  {  cout << "(";  Print(T->lchild, T); // 遍历左子树  cout << T->data; // 输出当前节点  Print(T->rchild, T); // 遍历右子树  cout << ")";    }  else // 如果是数字节点  {  cout << T->data;  }  }  }  int main()  {  char a[500];  bitree T;  while (fgets(a, 500, stdin) != NULL) // 读入数据  {  int i, j = 0, k = 0;  q = 0;  for (i = 0; a[i] != '\0'; i++)  {  if (a[i] != ' ')  {  d[j][k++] = a[i]; // 处理数据  }  else  {  d[j][k++] = '\0';  k = 0;  j++;  }  }  d[j++][k++] = '\0';  CreatBitree(T, 2, 0); // 创建二叉树  Print(T, NULL); // 遍历二叉树并输出  cout << endl;  }  } |

|  |  |
| --- | --- |
| 项目名称 | 表达式树的值 |
| 代码 | #include <iostream>  #include <cstring>  #include <cstdio>  using namespace std;  // 定义二叉树结构体  typedef struct binode {  char data[4]; // 存储节点数据  int h; // 存储节点高度  int depth; // 存储节点深度  struct binode\* lchild, \* rchild; // 左右子树指针  } binode, \* bitree;  char d[100][100]; // 存储输入的字符串  int q = 0, num1, sum; // q表示当前字符串的行数，num1表示左右子树高度，sum表示计算结果  void creatbitree(bitree& T, int y, int num)// 创建二叉树  {  if (d[q][0] == '#') { // 如果当前节点数据为#，则表示该节点为空  T = NULL;  q++;  }  else {  T = new binode; // 分配新的节点空间  if (y == 1) { // 如果是左子树，则高度为1  T->h = 1;  }  else { // 如果是右子树，则高度为0  T->h = 0;  }  T->depth = ++num; // 计算深度  strcpy(T->data, d[q++]); // 将节点数据复制到当前节点中  creatbitree(T->lchild, 1, T->depth); // 递归创建左子树  creatbitree(T->rchild, 0, T->depth); // 递归创建右子树  }  }  void travel(bitree T) // 中序遍历表达式树  {  if (T != NULL) {  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') { // 如果当前节点是运算符  cout << "(";  travel(T->lchild); // 遍历左子树  cout << T->data; // 输出当前节点的数据  travel(T->rchild); // 遍历右子树  cout << ")";  }  else { // 如果当前节点是操作数  cout << T->data;  }  }  }  int calculate(bitree T) // 计算表达式树的值  {  if (T != NULL) {  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') { // 如果当前节点是运算符  switch (T->data[0]) {  case '+':  return (calculate(T->lchild) + calculate(T->rchild));  break;  case '-':  return (calculate(T->lchild) - calculate(T->rchild));  break;  case '\*':  return (calculate(T->lchild) \* calculate(T->rchild));  break;  case '/':  return (calculate(T->lchild) / calculate(T->rchild));  break;  }  }  else { // 如果当前节点是操作数  char a[1000];  strcpy(a, T->data);  int i, sum1 = 0;  for (i = 0; a[i] != '\0'; i++) {  sum1 = sum1 \* 10 + a[i] - '0'; // 将操作数字符串转换为整型  }  return sum1;  }  }  }  int main() {  char a[500];  bitree T;  while (fgets(a, sizeof(a), stdin) != NULL) { // 逐行读取输入的字符串  int i, j = 0, k = 0;  q = 0;  sum = 0;  for (i = 0; a[i] != '\0'; i++) { // 将字符串按空格分割成多个字符串  if (a[i] != ' ') {  d[j][k++] = a[i];  }  else {  d[j][k++] = '\0';  k = 0;  j++;  }  }  d[j++][k++] = '\0';  creatbitree(T, 2, 0); // 创建表达式树  travel(T); // 中序遍历表达式树  cout << "=" << calculate(T) << endl; // 计算表达式树的值并输出  }  return 0; } |

|  |  |
| --- | --- |
| 项目名称 | 24点游戏(Ⅰ) |
| 代码 | #include<iostream>  using namespace std;  typedef struct Node  {  string data;  Node\* left, \* right;  }BiTNode, \* BiTree;  void Print(BiTree T)//打印数字和符号  {  if (T == NULL) //如果当前节点为空，直接返回  {  return;  }  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '/' || T->data[0] == '\*')//如果当前节点是运算符，打印括号并递归打印左右子树  {  cout << "(";  Print(T->left);  cout << T->data;  Print(T->right);  cout << ")";  }  else cout << T->data; //否则直接打印当前节点的值  }  void CreatTree(BiTree& T) //创建表达式树  {  string ch;  cin >> ch; //从标准输入流中读入一个字符串  if (ch[0] == '#') //如果读入的字符串是"#"，表示当前节点为空  {  T = NULL;  }  else //否则创建一个新节点，将读入的字符串赋值给该节点的数据域，并递归创建左右子树  {  T = new BiTNode;  T->data = ch;  CreatTree(T->left);  CreatTree(T->right);  }  }  double Cal(BiTree T) //计算表达式树的值  {  double ret = 0;  double num1, num2;  if (T != NULL) //如果当前节点不为空  {  double num1 = Cal(T->left); //递归计算左子树的值  double num2 = Cal(T->right); //递归计算右子树的值  if (T->data[0] == '+' || T->data[0] == '-' || T->data[0] == '\*' || T->data[0] == '/') //如果当前节点是运算符  {  switch (T->data[0]) //根据运算符计算结果  {  case '+':  return num1 + num2;  case'-':  return num1 - num2;  case'\*':  return num1 \* num2;  case'/':  if (num2 - 0 >= 0.000001) //除数不为0  return num1 / num2;  default:  break;  }  }  else //如果当前节点是数字  {  int i;  string a;  a = T->data;  double ret = 0;  for (i = 0; a[i] != '\0'; i++)  ret = ret \* 10.00 + a[i] - '0' + 0.0; //将数字字符串转化为数字  return ret;  }  }  }  int main()  {  BiTree T;  string ch;  while (cin >> ch)  {  double ret = 0;  T = new BiTNode; //创建一个新节点，将读入的字符串赋值给该节点的数据域，并递归创建左右子树  T->data = ch[0];  CreatTree(T->left);  CreatTree(T->right);  if (Cal(T) <= 24.000001 && Cal(T) >= 23.999999) //如果表达式树的值在23.999999~24.000001之间，输出表达式和结果  {  Print(T);  cout << "=24" << endl;  }  else //否则输出NO  cout << "NO" << endl;  }  return 0; } |