湖南科技大学计算机科学与工程学院 2023-2024第二学期数据结构课程设计报告

**专业班级：** 23物联网一班

**姓 名：** 曹烨

**学 号：** 2305040107

**指导教师：** 王焕宇

**时 间**： 2024.6.16-2024.6.28

**地 点**： 逸夫楼220

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

#### 目 录

|  |  |  |
| --- | --- | --- |
| **项目名称** | **完成日期** | **页码** |
| 复杂度分析(Ⅰ) | 2024.6.18 | P2~3 |
| 复杂度分析(Ⅱ) | 2024.6.18 | P2~3 |
| Josephus问题(Ⅰ) | 2024.6.18 | P4~6 |
| Josephus问题(Ⅱ) | 2024.6.18 | P4~6 |
| Josephus问题(Ⅲ) | 2024.6.18 | P4~6 |
| 真值表(Ⅰ) | 2024.6.18 | P7~10 |
| 真值表(Ⅱ) | 2024.6.18 | P7~10 |
| 单词检查(Ⅰ)- 顺序表实现 | 2024.6.18 | P11~15 |
| 单词检查(Ⅱ)- 二叉排序树实现 | 2024.6.18 | P11~15 |
| 后缀表达式求值 | 2024.6.19 | P16~18 |
| 中缀表达式转后缀表达式 | 2024.6.19 | P19~21 |
| 二叉树的创建和文本显示 | 2024.6.19 | P22~25 |
| 表达式树的创建与输出 | 2024.6.19 | P26~28 |
| 表达式树的值 | 2024.6.19 | P29~30 |
| 24点游戏(Ⅰ) | 2024.6.20 | P31~33 |
| 推箱子游戏-广度优先搜索版本 | 2024.6.21 | P34-36 |
| 带权路径长度 | 2024.6.24 | P37-40 |
| 自来水管道 | 2024.6.24 | P41-42 |
| 最小时间 | 2024.6.24 | P43-44 |

|  |  |
| --- | --- |
| **项目名称** | 复杂度分析 |
| **内容和目的** | 编写两个程序，分析给出代码的某个语句执行次数和执行完后某个程序的值。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** |   集合的抽象数据类型定义如下：  ADT set{  基本操作：  void cacu(long long count , long long total , long long n);  } | |
| |  | | --- | | **数据结构和算法实现** |   #include<iostream>  using namespace std;  #define ll long long  int main()  {  ll n;  while(cin>>n)  {  int i=0,j=0,k=0;  ll count=0;  ll total=0;  if(n>2)  {  count=1.0\*(n-3)\*(n-2)\*(n-1)/6;  total=3+(n-2)\*3;  cout<<count<<" "<<total<<'\n';  }  else  cout<<0<<" "<<"RANDOM"<<'\n';  }  }  #include<bits/stdc++.h>  using namespace std;  #define ll long long  int main()  {  ll n;  while(cin>>n)  {  int i=0,j=0,k=0;  ll count=0;  ll total=0;  if(n>1)  {  count=1.0\*(n+1)\*(n)\*(n-1)/6;  total=9+(n-2)\*3;  cout<<count<<" "<<total<<'\n';  }  else  cout<<0<<" "<<"RANDOM"<<'\n';  }  } | |
| |  | | --- | | **算法分析** |   第一个实验：  时间复杂度：  因为所有操作都是基于输入n的简单算术运算，计算量不随着输入的大小而变化，时间复杂度为O(1);  空间复杂度：  只使用了常量级别的额外空间来存储几个变量，空间复杂度为O(1);  第二个实验：  时间复杂度：  同样基于输入n的简单算术运算，计算量不随着输入的大小而变化，时间复杂度为O(1);  空间复杂度：  只使用了常量级别的额外空间来存储几个变量O(1);  综上所述，该两个算法的时间复杂度和空间复杂度都为O(1) | |
| |  | | --- | | **项目小结** |   该题目考查了数据结构中对复杂度的相关认识和了解，以及部分数学理论知识。通过推导出公式来优化程序的时间复杂度和空间复杂度，我们可以得出高效的算法。两个程序通过公式直接计算结果，在保持时间和空间复杂度为O(1)的情况下，展示了对输入数据进行快速计算的有效方法。 | |

|  |  |
| --- | --- |
| **项目名称** | Josephus问题 |
| **内容和目的** | n个人排成一圈，按顺时针方向依次编号1，2，3…n。从编号为1的人开始顺时针"一二"报数，报到2的人退出圈子。这样不断循环下去，圈子里的人将不断减少。最终一定会剩下一个人。试问最后剩下的人的编号。 |
| |  | | --- | | **项目分析与总体设计** |   算法一：根据题意作答  集合的抽象数据类型定义如下：  ADT set{  基本操作：  LinkList CreateList(int n)//创建新的循环链表  LinkList Execute(LinkList h, int k)//删除第k个结点，返回第k+1个结点指针  };   1. 使用不带头结点的循环链表来表示n个人 2. 首先创建一个从1到n的不带头结点的循环链表，返回指向结点1的头指针 3. 调用n次Execute函数删除第k个结点（含循环绕回的情况），Execute函数返回第k+1个结点的指针 4. 最后输出最后剩下的人的编号   算法二：找规律   1. 根据算法一，输入1~15的数，得出的数据为：  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | | 1 | 1 | 3 | 1 | 3 | 5 | 7 | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |  1. 由表格可推出规律：   每段数据呈现一个2的n-1次方个，每段数据中的数据都显示出2n-1的规律   1. 由此可通过找规律写出相应的算法内容 | |
| |  | | --- | | **数据结构和算法实现** |   算法一：  先定义集合类型和结点类型，如下所示：  typedef struct LNode  {  int data;  struct LNode \*next;  } LNode, \*LinkList;  建立新的不带头结点的循环链表，算法如下：  LinkList CreateList(int n)  {  LinkList L;  L=NULL;  LinkList p,s;  int i,j,k;  L=new LNode;  L->data=1;  L->next=NULL;  s=L;  p=L;  for(i=2;i<=n;i++)  {  p=new LNode;  p->data=i;  p->next=NULL;  s->next=p;  s=p;  }  p->next=L;  return L;  }  删除第k个结点，返回第k+1个结点的指针，算法如下：  LinkList Execute(LinkList h, int k)  {  if(h->next==h)return h;  LinkList p,s;  p=h;  int j=2;  while((p->next!=h)&&(j<k))  {  p=p->next;j++;  }  if((p->next==h)||(j>k))return h;  s=p->next;  p->next=s->next;  delete s;  return p->next;  }  算法二：  找规律可知  void solve(int n)  {  int i, m;  i=0;  while(1)  {  if(pow(2,i)>=n+1)break; //找到该数据在哪一段  i++;  }  m=2\*(n+1-pow(2,i-1))-1; // 数学公式运算  cout<<m<<'\n'; //输出  } | |
| |  | | --- | | **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(1)，空间复杂度为O(1)；  算法一通过循环链表进行题意模拟，最后得出唯一一个剩下的编号。  算法二通过数学推导进行数学计算，耗时较短。 | |
| |  | | --- | | **项目小结** |   该题目考查了线性表中的循环链表和部分数学知识。本题通过建立不带头结点的循环链表来进行相关操作，并通过找规律的方法大大简化了计算过程。这使得我们对循环链表的应用和数学推导在算法中的作用有了更深入的理解。 | |

|  |  |
| --- | --- |
| **项目名称** | 真值表 |
| **内容和目的** | 真值表是用于逻辑中的一类数学用表，用来计算逻辑表示式在每一个逻辑变量取值组合下的值。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。帮助我们更深刻地理解逻辑运算和真值表的概念和应用 |
| |  | | --- | | **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:  Void Judge(string s, string &s0,int test[],int &total)//记录逻辑变量是否出现  Void OutPrint(int counting[],int total)//输出真值表的值    }  算法二:  抽象数据类型定义如下：  ADT set{  基本操作:  int answer(int k1, int k2, string ch)//逻辑运算结果  int Prece(string top, string s)//优先级判断  void Calculate(string s, int total, string s0, int counting[])//计算表达式的值  } | |
| |  | | --- | | **数据结构和算法实现** |   算法一：  Judge函数记录逻辑变量，test数组用于记录26个小写字母，防止其出现表达式中同时存在多个相同的逻辑变量而导致错误，具体算法如下：  if(s[i]>='a'&&s[i]<='z')  {  if(test[s[i]-'a']!=1)  {  s0.push\_back(s[i]);  s0.push\_back(' ');  total++;  test[s[i]-'a']=1;  }  cout<<s[i]<<" ";  }  OutPrint函数用于输出真值表的值，由于真值表中只存在0和1这两个数字，并且按照格式要求，由大到小输出逻辑值。因此需要转化为二进制并存储在counting数组中。根据真值表的基本性质，total个逻辑变量可以构成2的total次方个组合情况。因此可用如下算法构建：  int r=pow(2,total)-1;  int count=1;  int co=0;  int counting[100001];  for(i=r;i>=0;i--)  {  int f=i;  co=0;  for(j=1;j<=total;j++)  {  counting[co]=f%2;  f=f/2;  co++;  }  for(k=co-1;k>=0;k--)cout<<counting[k]<<" ";  cout<<endl;  }  算法二：  Answer函数用于对两个逻辑值做出逻辑运算，具体算法如下：  int answer(int k1, int k2, string ch)  {  if (ch == "||")  {  if (k1 == 1 && k2 == 1) return 1;  if (k1 == 1 && k2 == 0) return 1;  if (k1 == 0 && k2 == 1) return 1;  if (k1 == 0 && k2 == 0) return 0;  }  else if (ch == "<->")  {  if (k1 == 1 && k2 == 1) return 1;  if (k1 == 1 && k2 == 0) return 0;  if (k1 == 0 && k2 == 1) return 0;  if (k1 == 0 && k2 == 0) return 1;  }  else if (ch == "->")  {  if (k1 == 1 && k2 == 1) return 1;  if (k1 == 1 && k2 == 0) return 1;  if (k1 == 0 && k2 == 1) return 0;  if (k1 == 0 && k2 == 0) return 1;  }  else if (ch == "^")  {  if (k1 == 1 && k2 == 1) return 1;  if (k1 == 1 && k2 == 0) return 0;  if (k1 == 0 && k2 == 1) return 0;  if (k1 == 0 && k2 == 0) return 0;  }  }  Prece函数用于判断该逻辑符号的优先级，引入括号，规定基本逻辑联接词优先顺序从高到低依次是：( )、!、∧、||、->、<->。 同一优先级，从左到右顺序进行。具体算法如下：  int Prece(string top, string s)  {  if (top == " ")  return 0;  else  {  if(top=="!"&&s=="!")return 0;  if (top == "(" && s == ")") return 0;  else if (s == ")") return 3;  else if (s == "(") return 2;  else  {  if (top == "(") return 2;  else if (top == "!") return 1;  else if (top == "^")  {  if (s == "!") return 2;  else return 1;  }  else if (top == "||")  {  if (s == "^" || s == "!") return 2;  else return 1;  }  else if (top == "->")  {  if (s == "^" || s == "||" || s == "!") return 2;  else return 1;  }  else if (top == "<->")  {  if (s == "^" || s == "||" || s == "!" || s == "->") return 2;  else return 1;  }  }  }  }  Calculate主要用于入栈出栈等运算操作，内部会调用Prece函数来进行优先级判断，用answer函数来计算两个逻辑值做某个运算的结果。具体可分为以下几步：   1. 遍历输入的字符串，判断其字符是否为逻辑变量，若是则将对应的逻辑值存入sD栈中（逻辑值栈）然后continue跳转至新一轮循环。 2. 当符号栈为空且字符不为变量时，除非为空格，否则对进行一定处理后存入sR栈中（运算符栈）。然后continue跳转至新一轮循环。 3. 特判若符号栈为空且字符为“！”时，做一定处理后存入栈中。然后continue跳转至新一轮循环。 4. 取出符号栈中栈顶的元素，用字符串a进行储存，从s中找寻逻辑运算符，进行一定处理后存入字符串b。 5. 进行运算符优先级判断，若等于则弹出符号栈的栈顶元素；若小于则将该运算符推入符号栈中；若大于则将符号栈弹出并用ch储存起来，接着用k1,k2存储逻辑值栈的两个值，并调用answer函数进行运算，运算完的结果存入逻辑值栈中。 6. 最后若符号栈中还有剩余元素，一个个将符号栈的栈顶元素弹出，并与逻辑值栈的元素进行运算。最后逻辑值栈的栈顶元素则为运算结果，立刻输出。 | |
| |  | | --- | | **算法分析** |   算法一：  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。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，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)。  算法二：  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。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和逻辑值栈和运算符栈的空间。变量取值可以使用一个长度为n的数组来存储，逻辑值栈和运算符栈的最大长度为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** |   本项目通过实现真值表的生成，加深了对离散数学中的逻辑运算和真值表的理解。项目采用顺序表和栈等数据结构，通过不同算法的实现，展示了数据结构在逻辑运算中的典型应用。不仅提高了对逻辑运算和真值表的理解，还锻炼了数据结构和算法的应用能力。 | |

|  |  |
| --- | --- |
| **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：  int InitList(SqList &L);//顺序表初始化  int ListInsert(SqList &L,int i,string e)//将字符串插入顺序表  int ListCheck(SqList L,string e)//查看字符串与顺序表中的元素是否完全匹配  int ListSecCheck(SqList L,string e)//字符串与顺序表元素不完全匹配  }  用二叉树的抽象数据类型定义如下：  ADT set{  基本操作：  void BST\_Init(BiTree &T);//初始化二叉排序树  void BSTInsert(BiTree &T,string e)//将字符串插入二叉排序树  BiTree BSTCheck(BiTree T,string e)//查看字符串与二叉排序树的元素是否匹配  void BST\_Post\_Travel(BiTree T)//二叉排序树后序遍历  } | |
| |  | | --- | | **数据结构和算法实现** |   算法一：  先定义集合类型和结点类型，如下所示：  typedef struct SqList{  string \*elem;  int length;  }SqList;  首先进行顺序表初始化，配置好顺序表中的elem和length；  其次开始输入字符串，若字符串不为“#”，则调用ListInsert函数，将字符串插入顺序表中，并长度加1。函数参数解释：L表示顺序表，i表示插入位置，e表示插入的字符串。  顺序表初始化：  int InitList(SqList &L)  {  L.elem=new string[100001];  L.length=0;  return 1;  }  字符串插入至顺序表  int ListInsert(SqList &L,int i,string e)  {  int j=0;  if((i<1)||(i>L.length+1))return 0;  if(L.length==100001)return 0;  for(j=L.length-1;j>=i-1;j--)  {  L.elem[j+1]=L.elem[j];  }  L.elem[i-1]=e;  L.length++;  return 1;  }  完全匹配判断：  当顺序表内的字符串与该字符串完全一致，则返回该元素在表内的编号  int ListCheck(SqList L,string e)  {  int i;  for(i=0;i<L.length;i++)  {  if(L.elem[i]==e)  return i+1;  }  return 0;  }  不完全匹配判断：  int ListSecCheck(SqList L,string e)  {  int i,j;  int t;  int count=0;  cout<<e<<": ";  for(i=0;i<L.length;i++)//所有线性表遍历  {  if(e.length()==L.elem[i].length())//string和线性表中的字符串长度相等  {  count=0;  j=0;t=0;  while(L.elem[i][j])  {  if(L.elem[i][j]!=e[t])//遍历线性表中第i个字符串  count++;  j++;  t++;  if(count>=2)//两次不一样就断  break;  }  if(count<=1)  cout<<L.elem[i]<<" ";  }  if(e.length()==L.elem[i].length()-1)//string长度=线性表中的字符串长度-1  {  count=0;  j=0;t=0;  while(e[t])//哪个短就while哪个  {  if(L.elem[i][j]!=e[t])  {  count++;  t--;  }  j++;  t++;  if(count>=2)  break;  }  if(count<=1)//符合要求  cout<<L.elem[i]<<" ";  }  if(e.length()==L.elem[i].length()+1)//string长度=线性表中的字符串长度+1  {  count=0;  j=0;t=0;  while(L.elem[i][j])//哪个短就while哪个  {  if(L.elem[i][j]!=e[t])  {  count++;  j--;  }  j++;  t++;  if(count>=2)  break;  }  if(count<=1)  cout<<L.elem[i]<<" ";  }  }  cout<<endl;  }  算法二：  首先进行顺序表初始化，配置好顺序表中的elem和length；  其次开始输入字符串，若字符串不为“#”，则调用BSTInsert函数，将字符串插入二叉排序树中，当字典输入完毕后，输出二叉排序树的后序遍历。  二叉排序树初始化：  void BST\_Init(BiTree &T)  {  T=NULL;  }  二叉排序树插入算法：  void BSTInsert(BiTree &T,string e)  {  BiTree s;  if(!T)  {  s=new BiTNode;  s->elem=e;  s->lchild=NULL;  s->rchild=NULL;  T=s;  }  else if(e<T->elem)  BSTInsert(T->lchild,e);  else if(e>T->elem)  BSTInsert(T->rchild,e);  }  二叉排序树查找算法：  BiTree BSTCheck(BiTree T,string e)  {  if((!T)||e==T->elem)return T;  else if(e<T->elem) return BSTCheck(T->lchild,e);  else return BSTCheck(T->rchild,e);  }  二叉排序树后序遍历：  void BST\_Post\_Travel(BiTree T)  {  if(T)  {  BST\_Post\_Travel(T->lchild);  BST\_Post\_Travel(T->rchild);  cout<<T->elem<<" ";  }  } | |
| |  | | --- | | **算法分析** |   算法一：  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。  其中n为字典中单词的个数，其中m为单词的长度。  顺序表初始化的时间复杂度为O(1)，插入算法时间复杂度为O(n)，由于完全匹配算法匹配的时候需要遍历，时间复杂度为O(n)，不完全匹配的算法，时间复杂度取决与n和m，时间复杂度为O(n\*m)。  空间复杂度主要取决于存储字典和待检查的单词的空间。  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)。  算法二：  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。  查找时，若输入的单词与字典中的单词完全匹配，则输出该字符串接上“ is correct”。否则进入不完全匹配。不完全匹配有以下几种情况：   1. 与字典单词字母数相同 2. 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：  插入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.  查找相似单词的时间复杂度为 O(n\*m)。  综上所述，该算法的时间复杂度为O(nlogn+n\*m) | |
| |  | | --- | | **项目小结** |   该题目考查了数据结构中对线性表和二叉排序树的相关知识，二叉排序树是主要通过递归来进行插入、查找和遍历。线性表主要使用顺序表，对数组元素和表长进行操作。本项目成功实现了一个基于顺序表和二叉排序树的拼写检查程序，能够高效地处理和检查文本中的拼写错误，并提供相应的拼写建议。通过分析，确认其时间和空间复杂度均为线性或对数级别，具有较好的性能表现 | |

|  |  |
| --- | --- |
| **项目名称** | 后缀表达式求值 |
| **内容和目的** | 为了便于处理表达式，常常将普通表达式（称为中缀表示）转换为后缀表达式。通过本题可以更深刻地了解栈在表达式方面处理的应用。 |
| |  | | --- | | **项目分析与总体设计** |   项目分析：   1. 由于该题需要读入超过两位长度的数字，并且每个元素直接存在输入空格，所以需要进行多位数字的处理储存。 2. 之后，凡遇操作数则将之压进栈，遇运算符则从栈中弹出两个操作数进行该运算，将运算结果压栈。 3. 然后继续扫描，直到后缀表达式被扫描完毕为止，此时栈底元素即为该后缀表达式的值。   ADT set{  基本操作：  void Change\_Num (char ch,int &signal,int &num)//将单位数字转化成多位  void Push\_Num (char ch,int &signal,int &num)//将数字塞入栈中  void TestSignal(char ch)//对两个数字进行计算  } | |
| |  | | --- | | **数据结构和算法实现** |   本题使用了栈的数据结构，用于存储数字。  算法思路：   1. 由于“@”为输入终止符，所以当输入的字符为“@”时，循环结束 2. 算法是通过ch字符来进行读入，所以得先将数字进行整合，防止多位数字出现错误，由Change\_Num函数处理。栈s1为全局变量。 3. 当输入空格时，通过调用Push\_Num函数，将整合好的数字存入栈中。 4. 当输入运算符时，调用TestSignal函数进行运算，将栈中的两个元素弹出来与运算符进行运算，运算结果存入栈中。 5. 最后当输入终止符时，输出栈顶元素。   具体算法如下：  void Change\_Num(char ch,int &signal,int &num)  {  signal=1;//标志，防止影响正常读入  num=num\*10+(ch-'0');  }  void Push\_Num(char ch,int &signal,int &num)  {  if(signal==1)  {  s1.push(num);//将数字推入栈中  num=0;  signal=0;  }  }  void TestSignal(char ch)  {  if(ch=='+')//运算符为加法时  {  int a,b,c;  a=s1.top();  s1.pop();  b=s1.top();  s1.pop();  c=a+b;  s1.push(c);  }  else if(ch=='-')//运算符为减法时  {  int a,b,c;  a=s1.top();  s1.pop();  b=s1.top();  s1.pop();  c=b-a;  s1.push(c);  }  else if(ch=='\*')//运算符为乘法时  {  int a,b,c;  a=s1.top();  s1.pop();  b=s1.top();  s1.pop();  c=b\*a;  s1.push(c);  }  else if(ch=='/')//运算符为除法时  {  int a,b,c;  a=s1.top();  s1.pop();  b=s1.top();  s1.pop();  c=b/a;  s1.push(c);  }  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  每个字符都需要读取和判断。这些操作的时间复杂度为O(1)假设输入长度为n，那么整体读取和判断的时间复杂度为O(n)。  Change\_Num 函数：在遇到数字字符时调用，每次操作是常数时间O(1)，所以总体时间复杂度还是 O(n)  Push\_Num 函数：当遇到空格字符时调用，每次操作是常数时间 O(1)  TestSignal 函数：在遇到运算符时调用，每次操作需要从栈中弹出两个元素并进行计算，再将结果压入栈中，仍然是O(1)时间复杂度  总时间复杂度为O(n)  空间复杂度：  主要由栈的空间来决定，用于存储操作数和中间计算结果。最坏情况下，栈中存储的元素个数不会超过输入长度n，因为每个操作数都会被压入栈中，而每次运算符会弹出两个元素，压入一个元素。所以空间复杂度为O(n)  综上所述，该算法的时间复杂度为O(n)，空间复杂度为O(n)。 | |
| |  | | --- | | **项目小结** |   本题考查了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。了解到了栈的应用在表达式计算方面的特殊之处。该算法能够高效地处理和计算输入表达式。通过分析，确认其时间和空间复杂度均为线性，具有较好的性能表现。未来可以通过改进错误处理和输入方式进一步提升其稳定性和灵活性。 | |

|  |  |
| --- | --- |
| **项目名称** | 中缀表达式转后缀表达式 |
| **内容和目的** | 考查了栈对表达式的应用，以及中缀表达式转后缀表达式的实现方法。输入一个中缀表达式，输出一个与中缀表达式相对于的后缀表达式。 |
| |  | | --- | | **项目分析与总体设计** |   本题给出了中缀表达式转化为后缀表达式的方法和步骤：  1.遇到操作数：直接输出（添加到后缀表达式中）  2.栈为空时，遇到运算符，直接入栈  3.遇到左括号：将其入栈  4.遇到右括号：执行出栈操作，并将出栈的元素输出，直到弹出栈的是左括号，括号不输出。  5.遇到其他运算符：加减乘除：弹出所有优先级大于或者等于该运算符的栈顶元素，然后将该运算符入栈  6.最终将栈中的元素依次出栈，输出。  抽象数据类型定义如下：  ADT set{  基本操作：  int Compare(char ch,char ch0)//比较运算符优先级  } | |
| |  | | --- | | **数据结构和算法实现** |   本题使用了栈的数据结构，用于存储运算符和操作数  算法思路：   1. 多次输入字符，当字符为操作数时，直接输出 2. 当栈为空，且字符为运算符（不包括括号），则直接存入栈中。 3. 当字符为左括号“(”，则存入栈中。 4. 当字符为右括号，一个个弹出栈中的运算符并输出，直到栈顶元素为左括号为止，并将左括号弹出。 5. 若字符为运算符（不包括括号），则进入循环，若栈为空则终止，将运算符与栈顶元素做优先级比较，若栈顶元素的优先级大于等于该运算符，则输出并弹出栈顶元素。最后将该运算符存入栈中 6. 最后将栈里所有的元素依次输出即可。   具体算法如下：  int Compare(char ch,char ch0)  {  if(ch0=='+'||ch0=='-')  {  if(ch=='\*'||ch=='/')  return 1;  else return 0;  }  else if(ch0=='\*'||ch0=='/')  {  return 0;  }  else if(ch0!='(')  return 0;  else return 1;  }  int main()  {  char ch;  stack<char> s;  while(cin>>ch)  {  if(ch=='\n')break;  else if(ch>='A'&&ch<='Z'||ch>='a'&&ch<='z')  {  cout<<ch;  }  elseif(s.empty()&&ch=='+'||s.empty()&&ch=='-'||s.empty()&&ch=='\*'||s.empty()&&ch=='/')  {  s.push(ch);  }  else if(ch=='(')  s.push(ch);  else if(ch==')')  {  while(s.top()!='(')  {  char ch1=s.top();  cout<<ch1;  s.pop();  }  if(s.top()=='(')  {  s.pop();  }  }  else if(ch=='+'||ch=='-'||ch=='\*'||ch=='/')  {  while(1)  {  if(s.empty())break;  if(Compare(ch,s.top())!=0)break;  cout<<s.top();  s.pop();  }  s.push(ch);  }  }  while(!s.empty())  {  cout<<s.top();  s.pop();  }  cout<<endl;  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  该算法的时间复杂度为O(n)，表示对栈操作所花费的时间。  空间复杂度：  空间复杂度为O(n)，需要栈来存储数据。  综上所述，该算法的时间复杂度为O(n)，空间复杂度为O(n)。 | |
| |  | | --- | | **项目小结** |   该项目考察了对数据结构栈的应用以及代码编写的逻辑操作，通过完成中缀表达式转后缀表达式的项目，我深入理解了栈在表达式处理中的重要应用。这个项目不仅考察了基本的栈操作，还涉及到运算符优先级的判断和处理，特别是括号的处理。在实现过程中，我学会了如何有效地利用栈来处理运算符，确保输出的后缀表达式与输入的中缀表达式等价。 | |

|  |  |
| --- | --- |
| **项目名称** | 二叉树的创建和文本显示 |
| **内容和目的** | 编一个程序，读入先序遍历字符串，根据此字符串建立一棵二叉树（以指针方式存储）各结点数据（长度不超过3），用空格分开，其中“#”代表空树。建立起此二叉树以后，再按要求输出二叉树。 |
| |  | | --- | | **项目分析与总体设计** |   该项目可用两种方法解决  算法一：   1. 建立二叉树结构体BiTNode，指针类型为\*BiTree。结构体中加入表示层次的layer参数。便于实现要求输出 2. 正常创建一个二叉树，将层次layer全部置零。后通过bfs层次遍历改变每个结点的层次值。 3. 找到该二叉树的深度。通过规律可知，该输出顺序为中序遍历的反向输出，所以对该二叉树进行逆中序遍历。   ADT set{  基本操作：  void CreateBiTree(BiTree &T,string s0[],int i)//创建二叉树  int FindDepth(BiTree T)//数的深度  int bfs(BiTree T)//广度优先搜索进行层次遍历  void DisMidTraverse(BiTree T,int t)//逆中序遍历  }  算法二：  与算法一不同的是，该算法通过给二叉树创建算法加入一个变量，作为记录该结点的层次，不需使用广度优先搜索进行层次遍历来记录层次。其他的几乎不变。  ADT set{  基本操作：  void CreateBiTNode(BiTree &T,string s0[],int i,int num0)//创建二叉树并记录结点层次  int FindDepth(BiTree T)//树的深度  void DisMidTraverse(BiTree T,int t)//逆中序遍历  } | |
| |  | | --- | | **数据结构和算法实现** |   算法思路：  算法一：   1. 首先构建二叉树自定义结构体BiTNode   typedef struct BiTNode{  string ch;  struct BiTNode \*lchild,\*rchild;  int layer;  }BiTNode,\*BiTree;   1. 由于该项目为多行输入，则每次输入用一个字符串存起来再进行处理，可通过分割字符串和构建二维数组来进行处理。 2. 构建新的二叉树，通过递归实现。 3. 通过广度优先搜索来进行层次遍历，补全各个结点中layer的值，并调用FindDep函数找到树的深度 4. 通过逆中序遍历DisMidTraverse进行输出。通过各个结点的层次，显示对应的二叉树   具体算法如下：  创建二叉树  void CreateBiTree(BiTree &T,string s0[],int i)  {  if(s0[i]=="#")  {  T=NULL;  Count1=i;  }  else  {  T=new BiTNode;  T->ch=s0[i];  T->layer=0;//将层次全部清零  Count1=i;//全局变量，用于调用string类型数组中的字符串  CreateBiTree(T->lchild,s0,Count1+1);  CreateBiTree(T->rchild,s0,Count1+1);  }  }  找到该树的深度  int FindDepth(BiTree T)  {  if(T==NULL)return 0;  else  {  int m,n;  m=FindDepth(T->lchild);  n=FindDepth(T->rchild);  if(m>n) return (m+1);  else return (n+1);  }  }  通过队列实现BFS  int bfs(BiTree T)  {  if(T==NULL)return 0;  q.push(T);  while(!q.empty())  {  BiTree p;  p=q.front();  q.pop();  if(p->lchild)  {  q.push(p->lchild);  p->lchild->layer=p->layer+1;  }  if(p->rchild)  {  q.push(p->rchild);  p->rchild->layer=p->layer+1;  }  }  }  逆向中序遍历  void DisMidTraverse(BiTree T,int t)  {  if(T)  {  int depth=T->layer;  //cout<<T->ch<<":"<<t-depth<<endl;  DisMidTraverse(T->rchild,t);    for(int i=1;i<=depth;i++)  {  cout<<" ";  }  cout<<T->ch<<endl;  DisMidTraverse(T->lchild,t);  }  }  算法二：   1. 首先构建二叉树自定义结构体（见算法一）   2. 由于该项目为多行输入，则每次输入用一个字符串存起来再进行处理，可通过分割字符串和构建二维数组来进行处理。  3. 构建新的二叉树，参数中的num0用于构建层次。其他方面与算法一类似。  4. 通过逆中序遍历DisMidTraverse进行输出。通过各个结点的层次，显示对应的二叉树  具体算法如下（在此只展示创建二叉树的函数，其他与算法一类似）  void CreateBiTNode(BiTree &T,string s0[],int i,int num0)  {  if(s0[i]=="#")  {  T=NULL;  Count1=i;  }  else  {  T=new BiTNode;  T->ch=s0[i];  T->layer=num0;//标记层次  Count1=i;  CreateBiTNode(T->lchild,s0,Count1+1,num0+1);  CreateBiTNode(T->rchild,s0,Count1+1,num0+1);  }  } | |
| |  | | --- | | **算法分析** |   算法一：  时间复杂度主要取决于二叉树的深度，若为完全二叉树，则时间复杂度为O(logn)，否则为O(n)。空间复杂度主要取决于二叉树的结点数，也为O(n )  该算法中通过队列使用了广度优先搜索来进行层次遍历。  算法二：  时间复杂度为O(logn)，空间复杂度为O(n)。  该算法相对于上个算法较为简便，不需要使用BFS。 | |
| |  | | --- | | **项目小结** |   通过本项目，我们成功实现了一个二叉树的数据结构及其基本操作，并且通过递归的方式实现了二叉树结构的文本显示。这不仅增强了对二叉树基本概念和操作的理解，也为进一步研究和应用树形数据结构打下了坚实的基础。项目中，通过广度优先搜索和层次遍历，对二叉树的构建和显示有了更加深刻的认识。同时，探索了通过递归直接记录结点层次的优化方法，提高了程序的效率。 | |

|  |  |
| --- | --- |
| **项目名称** | 表达式树的创建与输出 |
| **内容和目的** | 表达式树是一种用于表示数学表达式的二叉树。通过表达式树，可以方便地进行表达式的评估、简化和转换等操作。本项目的主要目的是实现表达式树的创建，并提供方法将表达式树以文本形式输出 |
| |  | | --- | | **项目分析与总体设计** |   表达式树的创建需要解析输入的数学表达式，并根据表达式的优先级构建对应的二叉树。每个结点可以是一个操作符或操作数。  具体设计如下：  1.解析输入的中缀表达式，转换为后缀表达式。  2.根据后缀表达式构建表达式树  ADT set{  基本操作：  void CreateTree(BiTree &T,string s0[],int i)  int DFSTree(BiTree T)  int CheckSignal(BiTree T)  } | |
| |  | | --- | | **数据结构和算法实现** |   本项目通过DFS深度优先搜索算法进行表达式树的输出  首先构建二叉树自定义结构体BiTNode  typedef struct BiTNode{  string data;  struct BiTNode \*lchild,\*rchild;  }BiTNode,\*BiTree;  创建二叉树：  void CreateTree(BiTree &T,string s0[],int i)  {  if(s0[i]=="#")  {  T=NULL;  counting=i;  }  else  {  T=new BiTNode;  T->data=s0[i];  counting=i;  CreateTree(T->lchild,s0,counting+1);  CreateTree(T->rchild,s0,counting+1);  }  }  检测符号：  int CheckSignal(BiTree T)  {  if(T->data=="+"||T->data=="-"||T->data=="\*"||T->data=="/")  {  return 1;  }  return 0;  }  深度优先搜索进行中序遍历：  int DFSTree(BiTree T)  {  if(T==NULL)return 0;  else  {  if(CheckSignal(T))  {  cout<<"(";  DFSTree(T->lchild);  cout<<T->data;  DFSTree(T->rchild);  cout<<")";  }  else  {  cout<<T->data;  }  }  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  CreateTree和DFSTree的总时间复杂度为O(n)。其中n是表达式中操作符和操作数的总数。  空间复杂度：  字符串数组s0的空间复杂度为O(n)。  创建二叉树和递归调用的空间复杂度为O(n)。  综上所述，算法的时间复杂度为O(n)，空间复杂度为O(n)。 | |
| |  | | --- | | **项目小结** |   通过本项目，我们实现了表达式树的创建和输出功能。表达式树在数学表达式的评估、简化和转换中具有重要作用。我们通过解析输入的中缀表达式并转换为后缀表达式，再通过后缀表达式构建表达式树，最后通过深度优先搜索进行中序遍历输出表达式树。这不仅增强了对二叉树基本概念和操作的理解，也加深了对表达式解析的认识，为进一步学习和应用表达式树提供了基础。项目中涉及的递归和树形数据结构的操作，对我们理解数据结构和算法的深度有了显著提升 | |

|  |  |
| --- | --- |
| **项目名称** | 表达式树的值 |
| **内容和目的** | 表达式树是一种用于表示数学表达式的二叉树。在表达式树中，叶结点表示操作数，非叶结点表示操作符。通过表达式树，可以方便地进行表达式的评估和简化。本项目的目标是根据输入的表达式树的先序遍历字符串，计算表达式的值。 |
| |  | | --- | | **项目分析与总体设计** |   本项目在表达式树的创建与输出的基础上，添加了表达式树的值计算的功能。通过二叉树来进行表达式的计算。  ADT set{  基本操作：  int Calculate(string data,int l,int r)  int TotalCalculate(BiTree T)  } | |
| |  | | --- | | **数据结构和算法实现** |   本项目由表达式树的创建与输出的基础上建立新的功能，通过二叉树来进行表达式求值的操作，其他方面区别不大。  TotalCalculate通过递归遍历二叉树，并返回最后的结果。  Calculate函数对两个数进行基本的四则运算，返回的运算结果成为新的叶子结点。叶子结点为数字，非叶子结点则为运算符。  具体算法如下：  对两个数进行四则运算  int Calculate(string data,int l,int r)  {  if(data=="+")  {  return (l+r);  }  else if(data=="-")  {  return (l-r);  }  else if(data=="\*")  {  return (l\*r);  }  else if(data=="/")  {  return (l/r);  }  }  表达式求值：  通过递归来进行树的计算和遍历。当为叶子结点时，将字符串转化为数字存入total变量，直接返回。当为非叶子结点时，lval为左子结点递归返回的值，rval为右子结点递归返回的值，调用calculate进行两个数之间的运算并返回。  int TotalCalculate(BiTree T)  {  int lval=0;  int rval=0;  if(T->lchild==NULL&&T->rchild==NULL)  {  int total=stoi(T->data);  return total;  }  else  {  lval=TotalCalculate(T->lchild);  rval=TotalCalculate(T->rchild);  return Calculate(T->data,lval,rval);  }  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  经过一系列操作，CreateTree 函数时间复杂度为O(n)，TotalCalculate 函数和calculate时间复杂度也为O(n)  空间复杂度：  通过递归等一系列操作，空间复杂度也为O(n)  综上所述，时间复杂度为O(n)，空间复杂度为O(n)。 | |
| |  | | --- | | **项目小结** |   表达式树（Expression Tree）是一种用于表示和计算数学表达式的数据结构。它将表达式的运算符和操作数以树形结构组织起来，使得可以方便地进行表达式的求值和转换。表达式树作为一种灵活且强大的数据结构，可以有效地表示和处理各种复杂的数学表达式。通过适当的构建和遍历，可以实现表达式的高效求值和优化，适用于多种计算和编程场景中。 | |

|  |  |
| --- | --- |
| **项目名称** | 24点游戏(Ⅰ) |
| **内容和目的** | 24点游戏是一种基于数学运算的益智游戏，旨在通过组合四个给定的数字，使用加减乘除运算符，使得运算结果为24。这个项目的目标是实现一个程序，能够自动解决和验证给定的四个数字是否可以通过合法的运算得到24。 |
| |  | | --- | | **项目分析与总体设计** |   该项目有以下几点需要注意的地方：  1.表达式树的构建： 使用表达式树来表示和计算可能的数学表达式。每个结点可以是操作数或操作符，通过树的深度优先遍历来构建和求解表达式。  2.回溯搜索算法： 使用回溯法来生成和验证所有可能的运算组合，以找到满足条件的表达式。从四个数字中选择两个数字和一个操作符，计算结果，将结果与剩余的数字组合生成新的表达式，直到找到结果为24或无法生成新表达式为止。  3.异常情况处理： 考虑除法中分母为0的情况，以及递归生成表达式时的边界条件。  本项目可以以计算表达式树的值来进行改进。其中需要对A、J、Q、K改为1,11,12,13来进行计算。由于有浮点数的存在，相关函数类型需要修改。  ADT set{  基本操作：  double Caculate(string data,double l,double r)  double TotalCaculate(BiTree T)  int DFSTree(BiTree T)  } | |
| |  | | --- | | **数据结构和算法实现** |   首先使用较为简单的算法对特殊数字（A,J,Q,K）进行处理。再进行表达式树的创建。将调用TotalCalculate函数进行运算并返回给total。将total与24进行相减并取绝对值，考虑到一定的误差，大于0.00001即输出NO。否则调用DFSTree输出表达式，然后按格式要求输出24即可。  具体算法如下：  int DFSTree(BiTree T)  {  if(T==NULL)return 0;  else  {  if(CheckSignal(T))  {  cout<<"(";  DFSTree(T->lchild);  cout<<T->data;  DFSTree(T->rchild);  cout<<")";  }  else  {  cout<<T->data;  }  }  }  int CheckSignal(BiTree T)  {  if(T->data=="+"||T->data=="-"||T->data=="\*"||T->data=="/")  {  return 1;  }  return 0;  }  double Caculate(string data,double l,double r)  {  if(data=="+")  {  return (l+r);  }  else if(data=="-")  {  return (l-r);  }  else if(data=="\*")  {  return (l\*r);  }  else if(data=="/")  {  if(r==0)return 300000;  return (1.0\*l/r);  }  }  double TotalCaculate(BiTree T)  {  double lval=0.0;  double rval=0.0;  if(T->lchild==NULL&&T->rchild==NULL)  {  double total=stold(T->data);  return total;  }  else  {  lval=TotalCaculate(T->lchild);  rval=TotalCaculate(T->rchild);  return Caculate(T->data,lval,rval);  }  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  CreateTree 和 DFSTree 的总时间复杂度为 O(n)。  TotalCaculate 的时间复杂度也是 O(n)  空间复杂度：  字符串数组 s0 的空间复杂度为 O(n)。  创建二叉树和递归调用的空间复杂度为 O(n)。  综上所述，该算法的时间复杂度为O(n)，空间复杂度为O(n)。 | |
| |  | | --- | | **项目小结** |   通过本项目，我们成功实现了24点游戏的自动求解程序。该项目通过构建表达式树和回溯搜索算法，能够有效地验证给定的四个数字是否可以通过合法的运算得到24。项目中处理了特殊字符（A, J, Q, K）和除法中的异常情况，通过递归方法实现了表达式树的创建和运算。这不仅提高了我们对数学表达式解析的理解，也加强了对数据结构和算法设计的能力，积累了在数学推理和算法实现方面的经验和技能，为解决更复杂的数学问题打下了坚实的基础。 | |

|  |  |
| --- | --- |
| **项目名称** | 推箱子游戏-广度优先搜索版本 |
| **内容和目的** | 推箱子是一款经典游戏。这里我们玩的是一个简单版本,就是在一个N\*M的地图上，有1个玩家、1个箱子、1个目的地以及若干障碍，其余是空地。玩家可以往上下左右4个方向移动，但是不能移动出地图或者移动到障碍里去。如果往这个方向移动推到了箱子，箱子也会按这个方向移动一格，当然，箱子也不能被推出地图或推到障碍里。当箱子被推到目的地以后，游戏目标达成。现在告诉你游戏开始是初始的地图布局，请问玩家至少要多少步才能达成目标？ |
| |  | | --- | | **项目分析与总体设计** |   ADT set{  基本操作：  int bfs();//广度优先搜索  int OutBarrier(int x,int y)//越界判断  }  本项目需要通过广度优先搜索来进行推箱子操作，用队列进行广度优先搜索算法，有箱子和人两个目标。该算法可以通过使用一个四维数组来标记人和箱子的坐标。 | |
| |  | | --- | | **数据结构和算法实现** |   算法思路：  本题通过定义以下四维数组来进行是否走过该路径的标记。  int dis[20][20][20][20]={0};  再定义一个结点来作为人和箱子坐标的结构体，  typedef struct{  int x1,y1;  int x2,y2;  }Dis;  并定义Dis结构体为参数的队列queue<Dis> q;  定义好可以走动的四个位置，分别为(-1,0) (0,-1) (1,0),(0,1)。  int dx[]={-1,0,1,0};  int dy[]={0,-1,0,1};  接着按照题目要求，记录好“人”，“箱子”，“终点”。之后进行广度优先搜索。  分别记录好人和箱子的坐标，人与箱子重合的时候即表示推动箱子  int bfs()  {  q.push({startx,starty,boxx,boxy});  dis[startx][starty][boxx][boxy]=1;    while(!q.empty())  {  int i;  Dis now;  now=q.front();  q.pop();  if(now.x2==endx&&now.y2==endy)  return cout<<dis[now.x1][now.y1][now.x2][now.y2]-1<<endl,0;  for(i=0;i<4;i++)  {  if(OutBarrier(now.x1+dx[i],now.y1+dy[i]))  {  if(now.x1+dx[i]==now.x2&&now.y1+dy[i]==now.y2)  {  if(OutBarrier(now.x2+dx[i],now.y2+dy[i]))  {  if(dis[now.x1+dx[i]][now.y1+dy[i]][now.x2+dx[i]][now.y2+dy[i]]==0)  {  dis[now.x1+dx[i]][now.y1+dy[i]][now.x2+dx[i]][now.y2+dy[i]]=dis[now.x1][now.y1][now.x2][now.y2]+1;  Dis next;  next={now.x1+dx[i],now.y1+dy[i],now.x2+dx[i],now.y2+dy[i]};  q.push(next);  }  }  }  else  {  if(dis[now.x1+dx[i]][now.y1+dy[i]][now.x2][now.y2]==0)  {  dis[now.x1+dx[i]][now.y1+dy[i]][now.x2][now.y2]=dis[now.x1][now.y1][now.x2][now.y2]+1;  Dis next;  next={now.x1+dx[i],now.y1+dy[i],now.x2,now.y2};  q.push(next);  }  }    }  }  }  return cout<<-1<<endl,0;  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  时间复杂度主要由BFS的执行次数决定。在最坏情况下，BFS需要访问所有可能的状态。状态由玩家和箱子的坐标决定，因此总状态数为O(n^2\*m^2)，其中 n和 m 分别是网格的行数和列数。  空间复杂度：  队列 q 用于存储 BFS 的状态。在最坏情况下，队列中可能存储所有的状态O(n^2\*m^2)  四维数组dis用于记录从起始状态到每个状态的距离，理论上也是O(n^2\*m^2)  综上所述，该算法的时间复杂度为O(n^2\*m^2)，空间复杂度也为O(n^2\*m^2) | |
| |  | | --- | | **项目小结** |   实现了一个基于BFS（广度优先搜索）的求解迷宫问题，其中包括一个玩家、一个箱子和一个目标位置。玩家需要推动箱子到达目标位置，代码在二维网格上进行操作，判断如何从起始状态移动箱子到目标位置。该项目使我们更深入了解了广度优先搜索的应用和相关操作。同时加强了对数据结构（如队列、四维数组）和算法设计的能力，为解决类似的迷宫问题和其他搜索问题提供了宝贵的经验和技能。 | |

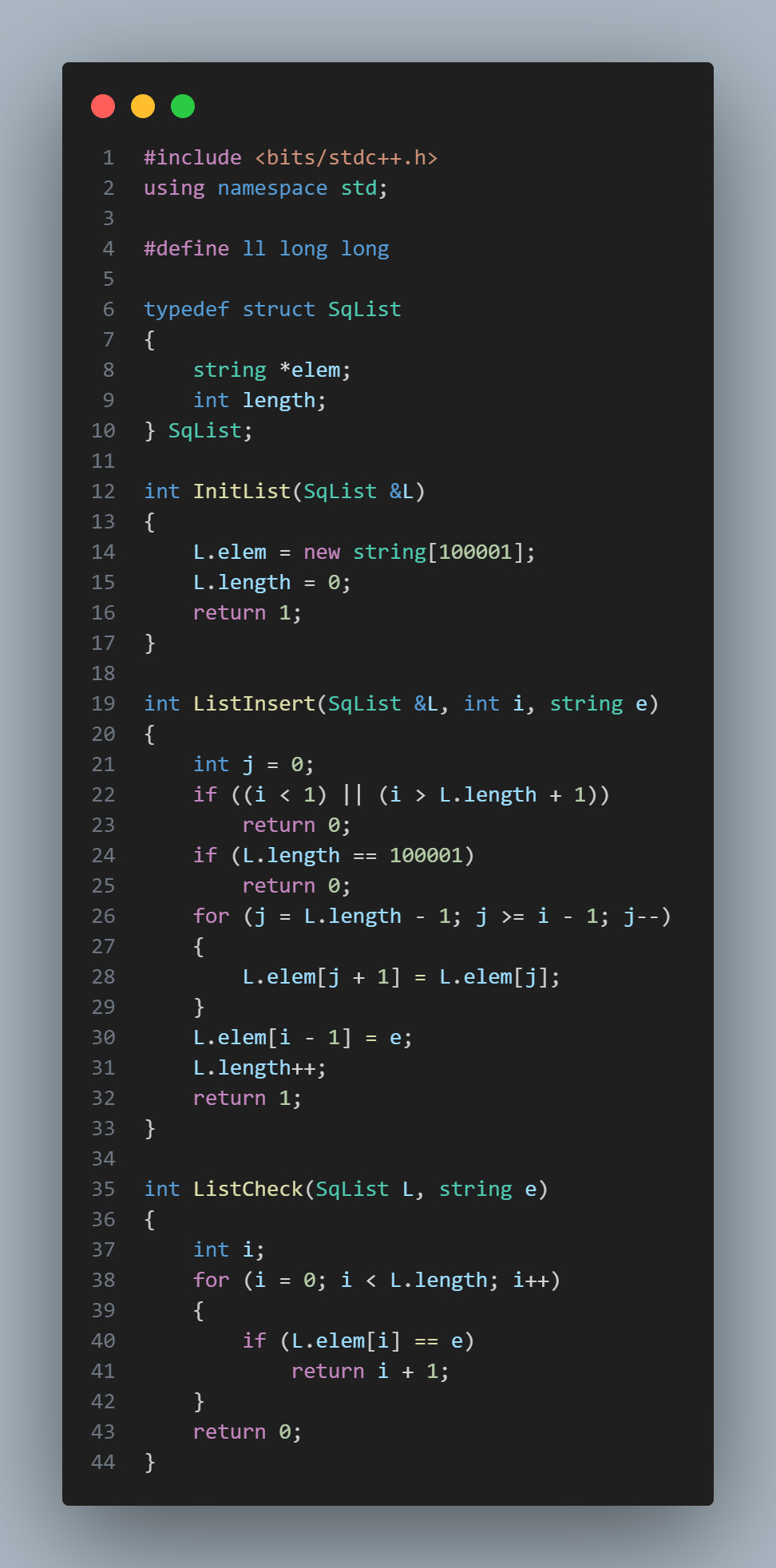
|  |  |
| --- | --- |
| **项目名称** | 带权路径长度 |
| **内容和目的** | 给定n个权值作为n个叶子结点，构造哈夫曼树, 求其带权路径长度。目的是学会哈夫曼树的基本构建和带权路径长度的求解 |
| |  | | --- | | **项目分析与总体设计** |   算法一：常规的构造哈夫曼树求解带权路径长度  ADT set{  基本操作：  void Select(HmTree &HT,ll n,ll &s1,ll &s2)//找哈夫曼树中两个权值最小的  void CreateHmTree(HmTree &HT,ll n)//创建哈夫曼树  void FindLayer(HmTree HT,ll n)//找到各个结点的层次，用于求带权路径长度  ll MinLen(HmTree HT,ll n)//求解带权路径长度  }  算法二：通过堆排序来求解带权路径长度  构造小根堆来进行堆排序，可以使用容器priority\_queue优先队列来进行求解 | |
| |  | | --- | | **数据结构和算法实现** |   算法一：  1.宏定义long long int为ll，并建立新的哈夫曼结点类型  #define ll long long  typedef struct{  ll weight;  ll parent,lchild,rchild;  ll layer;  }HTNode,\*HmTree;   1. 创建哈夫曼树   void Select(HmTree &HT,ll n,ll &s1,ll &s2)  {  ll i,j,k;  ll min1=0x3f3f3f3f;ll min2=0x3f3f3f3f;  s1=0;s2=0;  for(i=1;i<=n;i++)  {  if(HT[i].parent!=0)continue;  if(HT[i].weight<min1)  {  if(min1<min2)  {  min2=min1;  s2=s1;  }  min1=HT[i].weight;  s1=i;  }    else if(HT[i].weight<min2)  {  min2=HT[i].weight;  s2=i;  }  }  }  void CreateHmTree(HmTree &HT,ll n)  {  if(n<=1)return;  ll m=2\*n-1;  HT=new HTNode[m+1];  int i,j;  for(i=1;i<=m;i++)  {  HT[i].parent=0;HT[i].rchild=0;HT[i].lchild=0;  HT[i].layer=0;  }  for(i=1;i<=n;i++)  cin>>HT[i].weight;  for(i=n+1;i<=m;i++)  {  ll s1,s2;  Select(HT,i-1,s1,s2);  HT[s1].parent=i;HT[s2].parent=i;  HT[i].lchild=s1;HT[i].rchild=s2;  HT[i].weight=HT[s1].weight+HT[s2].weight;  }  }     1. 求结点层次   void FindLayer(HmTree HT,ll n)  {  ll i,j,k;  for(i=1;i<=n;i++)  {  ll count=0;  j=i;  while(HT[j].parent!=0)  {  count++;  j=HT[j].parent;  }  HT[i].layer=count;  }  }   1. 求解带权路径长度，带权路径长度对1000000007取模后输出   ll MinLen(HmTree HT,ll n)  {  ll i,j;  ll total=0;  for(i=1;i<=n;i++)  {  total+=(HT[i].weight\*HT[i].layer);  }  return total;  }  算法二：  1.构造优先队列，小根堆  priority\_queue<ll,vector<ll>,greater<ll>> q;  2.将输入的数字全部放入优先队列中。  3.易知带权路径长度也可以等于除根结点外每个结点权值之和。  while(q.size()>1)  {  ll a=q.top();  q.pop();  ll b=q.top();  q.pop();  ll c=a+b;  q.push(c);  total=(total+c)%1000000007;  } | |
| |  | | --- | | **算法分析** |   算法一：  时间复杂度：  由于该算法需要构建哈夫曼树，选择两个最小权值的结点配合哈夫曼树的创建，时间复杂度为O(n^2)。标记层次需要花费O(nlogn)的时间，求最小权值只需遍历，为O(n)。  空间复杂度：  构建哈夫曼树需要存储2n-1个结点，总的空间复杂度为O(n)。  综上所述，该算法的时间复杂度为O(n^2)，空间复杂度为O(n)  算法二：  时间复杂度：  首先将所有元素插入优先队列。插入n元素的时间复杂度是O(nlogn)。  合并操作时间复杂度同样为O(nlogn)。  空间复杂度：  主要由优先队列的存储空间决定，需要存储n个元素，空间复杂度为O(n)。  综上所述，该算法的时间复杂度为O(nlogn)，空间复杂度为O(n)。 | |
| |  | | --- | | **项目小结** |   优先队列方法在时间复杂度上优于传统哈夫曼树方法，因此在处理大规模数据时更为高效。该项目考查了对哈夫曼树的基本操作和应用，对带权路径长度的理解和求解。本项目可以通过堆排序来进行复杂度优化。通过优先队列的应用，实现了对哈夫曼树在大规模数据处理中的效率优化，提升了对数据结构和算法优化的理解和实践能力。 | |

|  |  |
| --- | --- |
| **项目名称** | 自来水管道 |
| **内容和目的** | 校园内有若干需要供水的点，每两个供水点可能存在多种铺设路径。对于每一种铺设路径，其成本是预知的。任务要求最终铺设的管道保证任意两点可以直接或间接的联通，同时总成本最低。 |
| |  | | --- | | **项目分析与总体设计** |   此项目需要通过最小生成树来进行求解。已知的两种算法可以用Prim算法和kruskal算法进行最小生成树的构建。Prim算法为归并点的操作，Kruskal算法可以通过归并边来进行最小生成树构建。本项目运用的Kruskal算法配合并查集压缩路径来求解最小生成树  ADT set{  基本操作：  Void memset();初始化  void sort(edge+1,edge+k,cmp);对edge数组进行自定义排序  bool cmp(Node a, Node b)//规定通过权值进行排序  int Found(int x, int pre[])//并查集找该结点的根结点  } | |
| |  | | --- | | **数据结构和算法实现** |   算法思路：   1. 读取顶点数 p 和边数 r。定义结点结构体。初始化邻接矩阵 Close 用于存储边的权值。初始化 vexset 数组，用于并查集的父结点记录。   typedef struct  {  int start, end;  int cost;  } Node;   1. 读取每条边的信息（两个顶点和边的权值），更新邻接矩阵 Close。确保记录最小的边权值。 2. 遍历邻接矩阵，将所有边存入 edge 数组。 3. 将 edge 数组按照边权值从小到大排序 4. 使用并查集来管理和检测是否形成环。 5. 遍历排序后的边数组，依次选择权值最小的边，检查其两端顶点是否属于不同集合（即不形成环）。如果不同，则合并这两个顶点所在的集合，并将边加入到最小生成树中。 6. 当选中的边数达到 p-1 时（即最小生成树包含 p-1 条边），停止算法。   具体算法如下：  bool cmp(Node a, Node b)  {  return a.cost < b.cost;  }  int Found(int x, int pre[])  {  if (x == pre[x])  return x;  else  return pre[x] = Found(pre[x], pre);  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  初始化邻接矩阵的需要消耗O(n^2)的时间，将所有边存入edge数组同样需要O(n^2)的时间。边排序使用sort函数，时间复杂度为O(mlogm)，n为顶点，m为边数。合并压缩路径消耗的时间可以忽略不计  空间复杂度：  主要取决于邻接矩阵、边数组和并查集数组，总空间复杂度为O(m+n^2)  综上所述，该算法时间复杂度为O(mlogm+n^2)，空间复杂度为O(m+n^2)。 | |
| |  | | --- | | **项目小结** |   Kruskal算法通过归并边并逐步选择权值最小且不形成环的边，最终构建出最小生成树。这个实现充分利用了并查集进行高效的环检测和集合合并，保证了算法的高效性。通过本项目，加深了对最小生成树算法的理解和应用，尤其是对Kruskal算法和并查集的掌握。在实现过程中，对算法优化和数据结构选择有了更深入的思考，这对未来解决类似问题提供了宝贵的经验和启示。  Kruskal算法适合处理稀疏图，因为它通过边的排序来选择最小权值的边，不需要像Prim算法那样频繁更新和检查节点集合。使用prim算法进行归并点也可以达到相同的效果。 | |

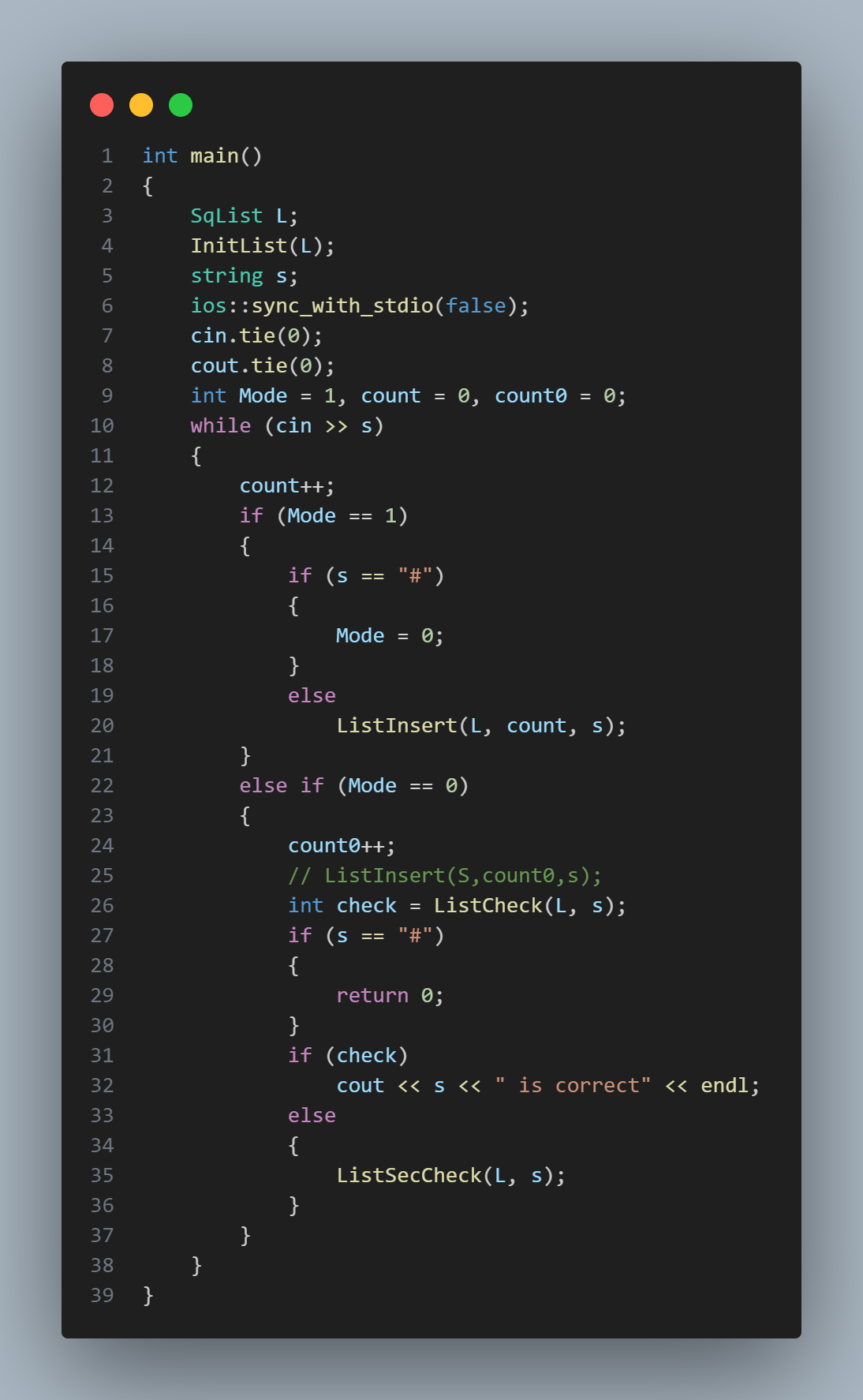
|  |  |
| --- | --- |
| **项目名称** | 最小时间 |
| **内容和目的** | 有多个城市组成一个铁路交通网络。任意两个城市之间有直连铁路，或者通过其他城市间接到达。给定某个城市，要求在M时间内能从该城市到达任意指定的另一城市，求最小的M。 |
| |  | | --- | | **项目分析与总体设计** |   该项目需要用到最短路径的算法，其中可以使用迪杰斯特拉(Dijkstra)算法和Floyd算法。本项目使用的是迪杰斯特拉(Dijkstra)算法，计算图中从起始点到其他所有点的最短路径，然后输出其中路径最长的，确保在M实践内可以从源点到达任意一个城市。  ADT set{  基本操作：  Void Memset()//初始化  void minpath(int a[][201],int dis[],int path[],int vis[],int n)//求源点到其他点的最短路径  } | |
| |  | | --- | | **数据结构和算法实现** |   算法思路：   1. 首先建立以下几个数组来进行辅助操作，在建立二维矩阵a存储边权值。   int path[10001];//原点到i的前驱结点  int dis[10001];//原点到i的最短路径长度  int vis[10001];//标记dis是否走过   1. 对a进行初始化，矩阵内所有元素设为无穷大，path内所有元素设为-1，vis数组初始化为0，dis存储1到其他结点的最短路径长度，初始化为最大值。 2. 若输入不为‘x’，通过string转int型，将字符串转化为整型数字。并标记在a[i][j]矩阵。否则该点置无穷大。 3. 调用minpath函数，每次从未访问的结点中选取一个当前距离最小的结点作为当前结点。 4. 更新当前结点的邻居结点的最短路径长度，如果通过当前结点到达某个邻居结点的路径比原来的路径短，则更新该邻居结点的最短路径长度   具体算法如下：  void minpath(int a[][201],int dis[],int path[],int vis[],int n)  {  int i,j,k;  for(i=1;i<=n;i++)  {  dis[i]=a[1][i];  if(a[1][i]<0x3f3f3f3f)  path[i]=1;  }  vis[1]=1;  dis[1]=0;  for(i=1;i<=n;i++)  {  int min=0x3f3f3f3f,min0;  for(j=1;j<=n;j++)  if(!vis[j]&&min>dis[j])  {  min=dis[j];  min0=j;  }  vis[min0]=1;  for(j=1;j<=n;j++)  {  if(a[min0][j]+dis[min0]<dis[j]&&!vis[j])  {  path[j]=min0;  dis[j]=dis[min0]+a[min0][j];  }  }  }  } | |
| |  | | --- | | **算法分析** |   时间复杂度：  初始化距离和路径数组的时间复杂度为O(n)。对a矩阵进行操作的时间复杂度为O(n^2)  空间复杂度：  a矩阵所消耗的空间为O(n^2)，其他辅助数组所花费的空间为O(n)  综上所述，该算法的时间复杂度为O(n^2)，空间复杂度为O(n^2) | |
| |  | | --- | | **项目小结** |   此项目考查了最短路径的算法，本算法适用于中等规模的图，但对于非常大的图，其效率可能不是最优的。对于大规模稀疏图，使用堆优化的迪杰斯特拉Dijkstra算法（即使用优先队列）可以将时间复杂度降低到O((n+m)logn)。进一步提高算法的效率。  通过本项目，深入理解了迪杰斯特拉算法在解决实际路径规划问题中的应用，对其原理和优化有了更清晰的认识。本项目不仅解决了校园铁路交通网络中的路径规划问题，还提升了在算法设计和实现中的技能和经验，为未来处理类似复杂网络问题提供了坚实的基础 | |

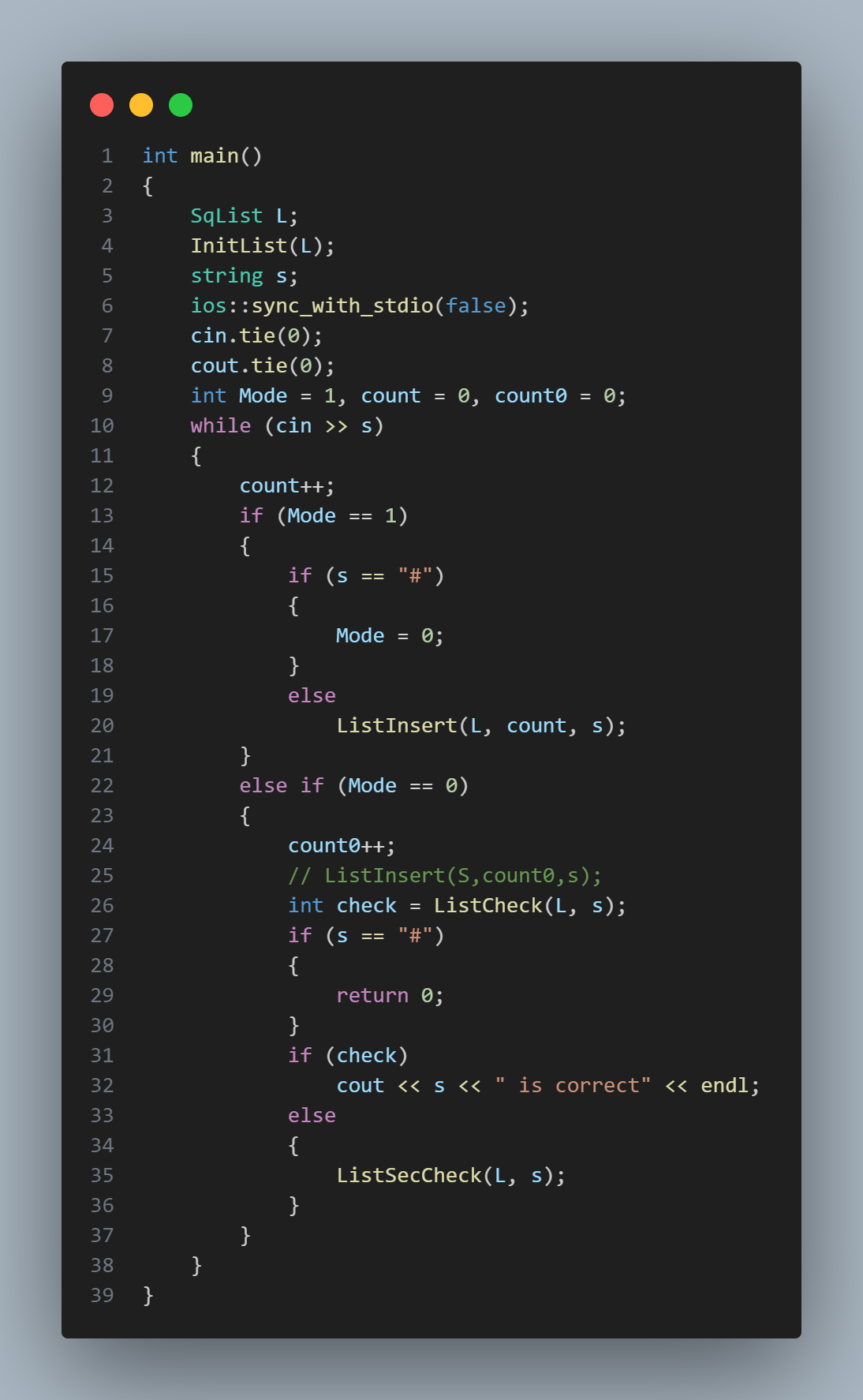
### 问题 I: 单词检查(Ⅰ)- 顺序表实现

**附录**

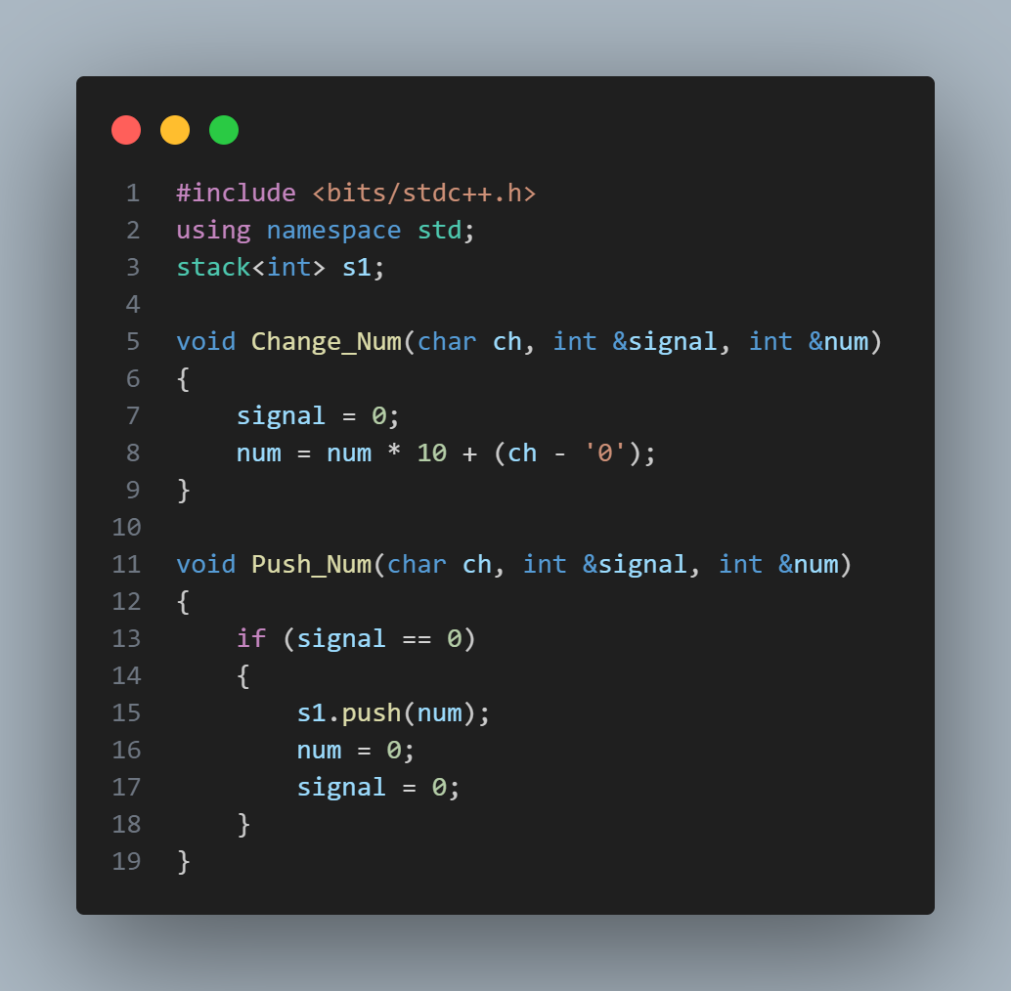


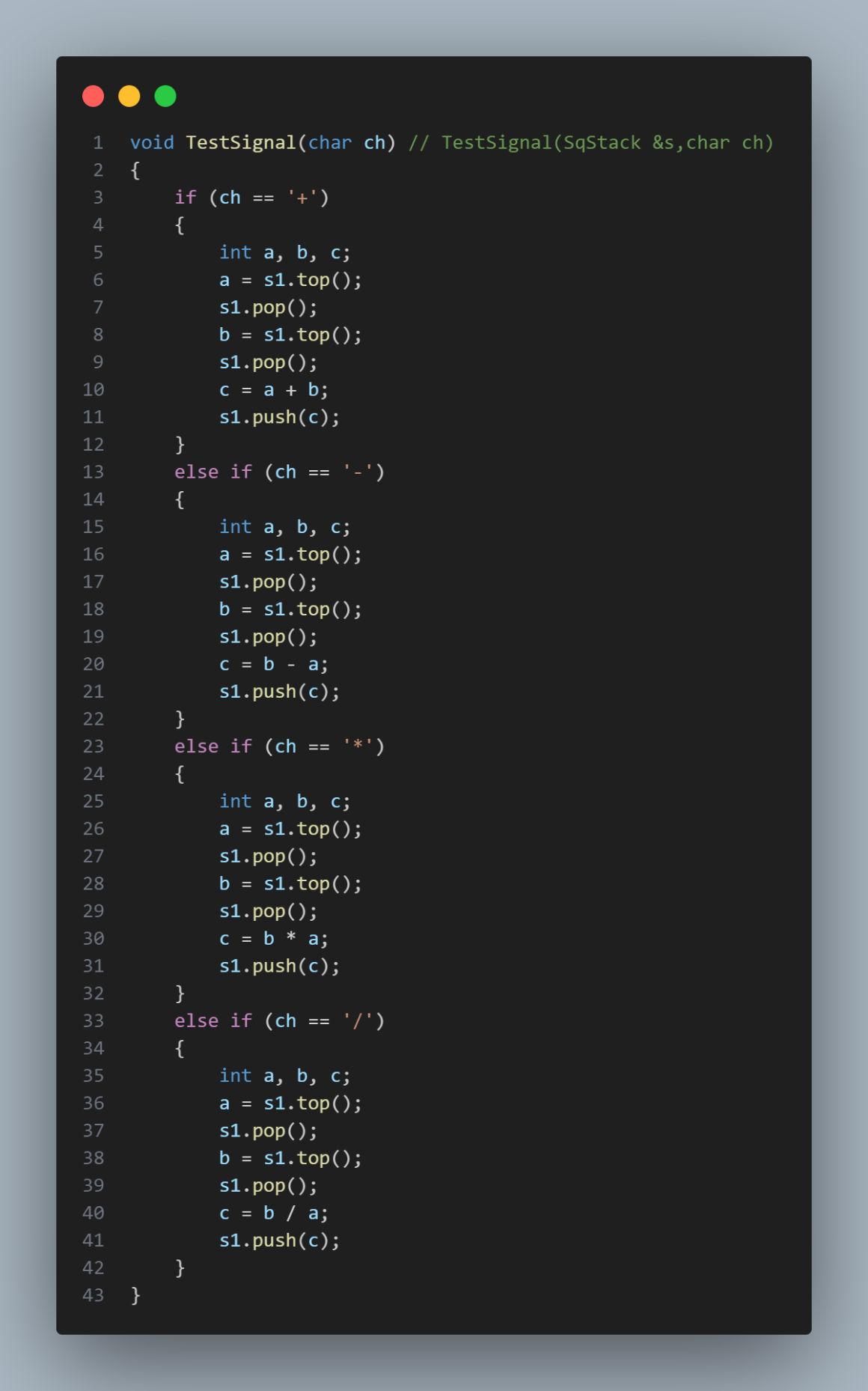


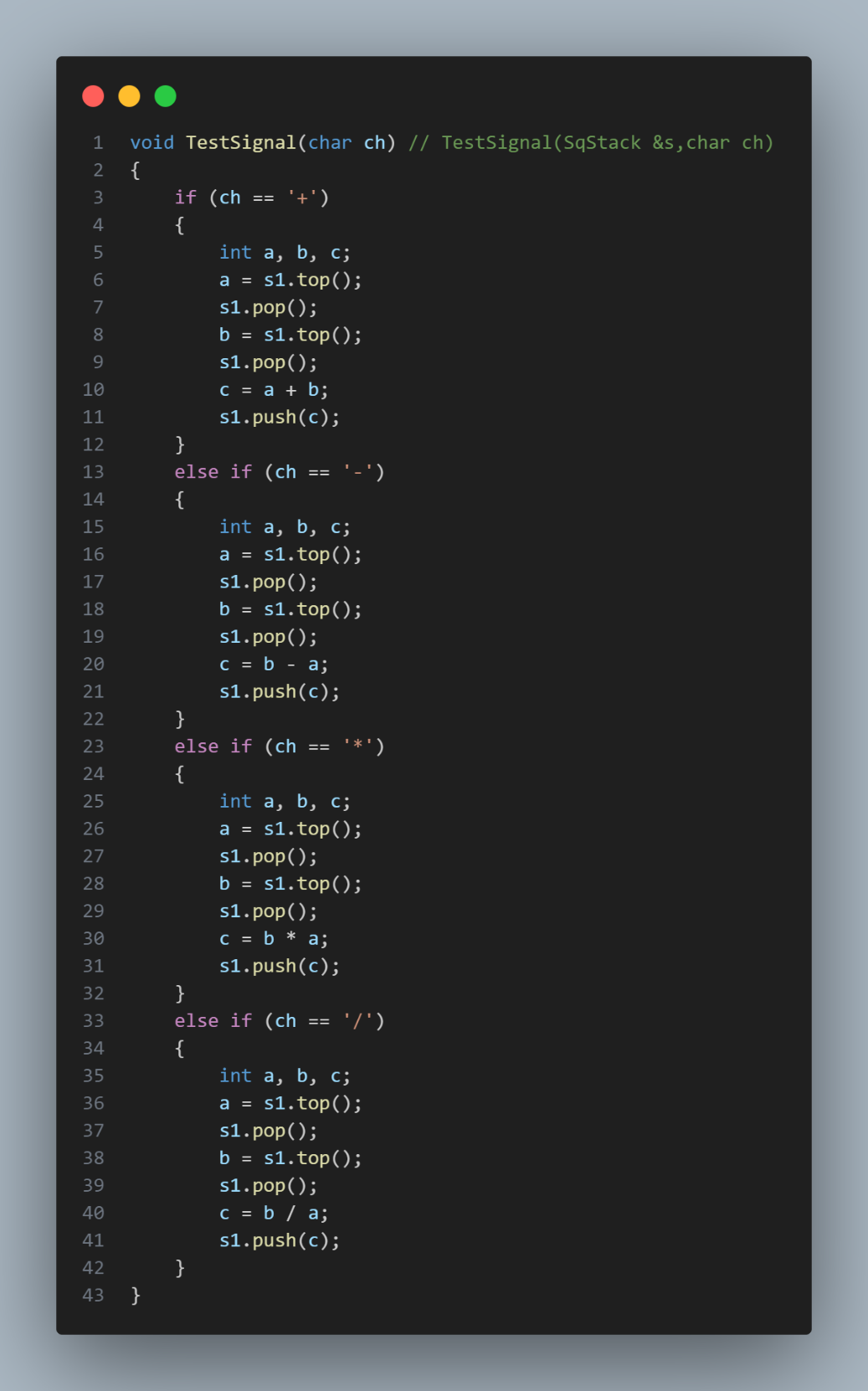


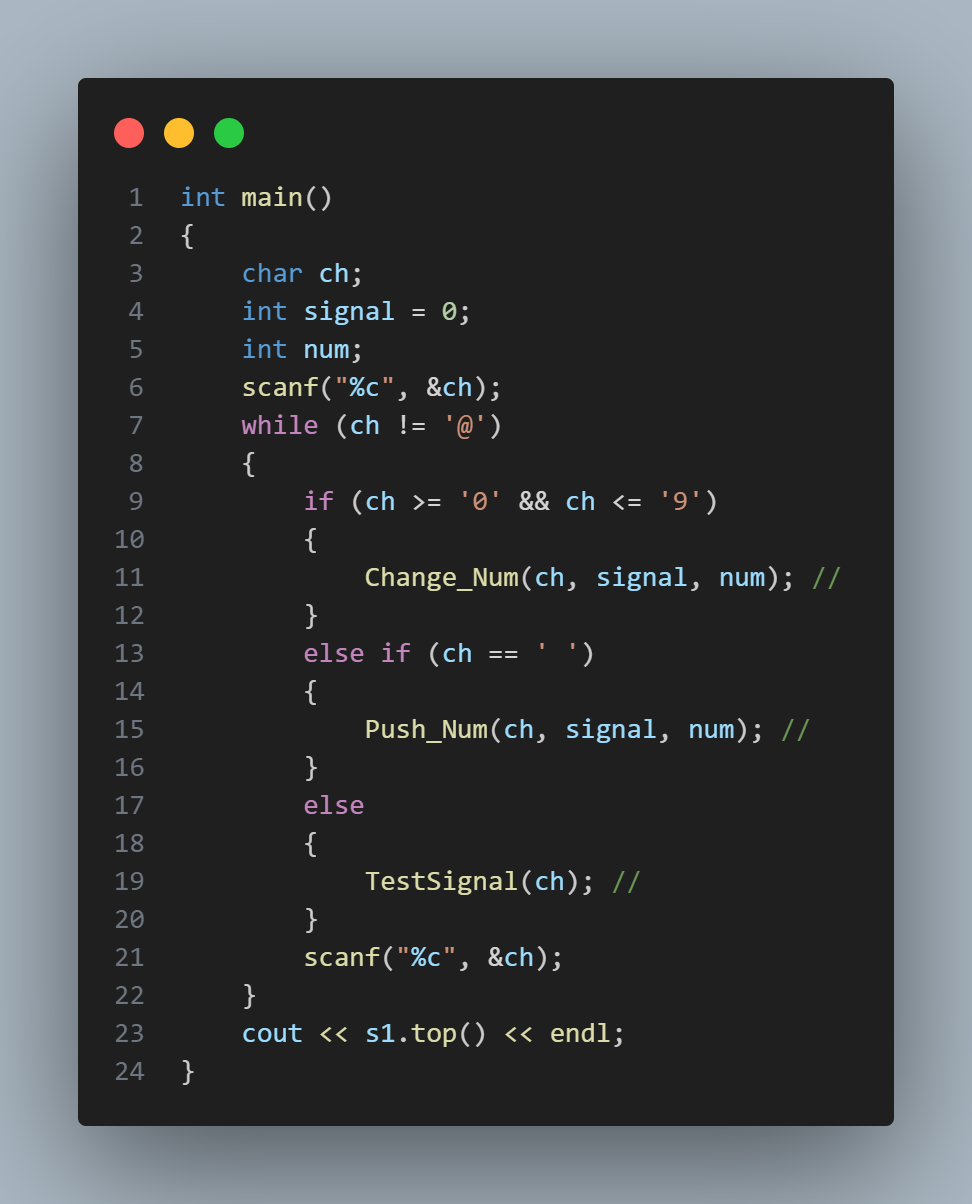


### 问题 L: 后缀表达式求值

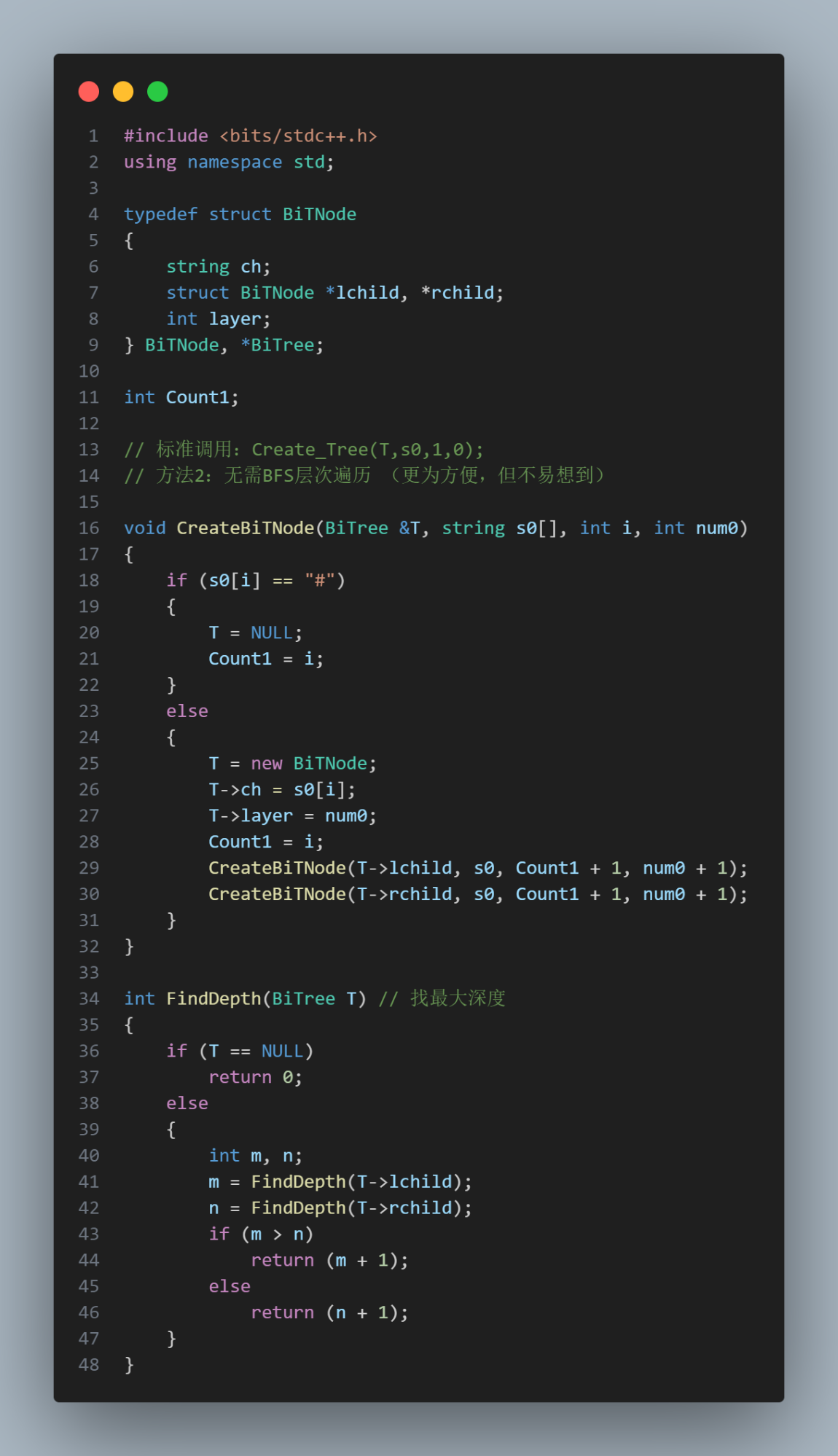


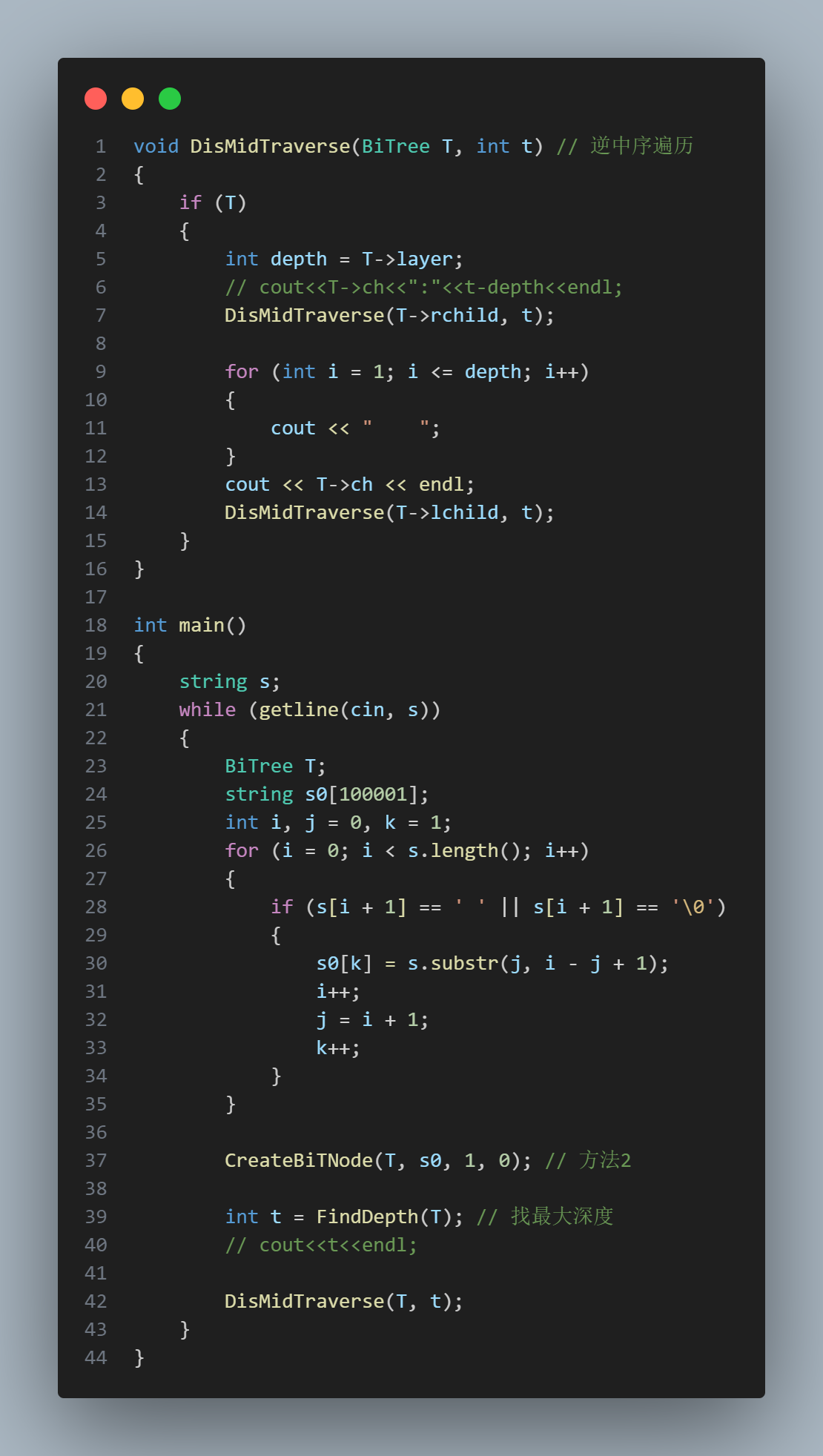






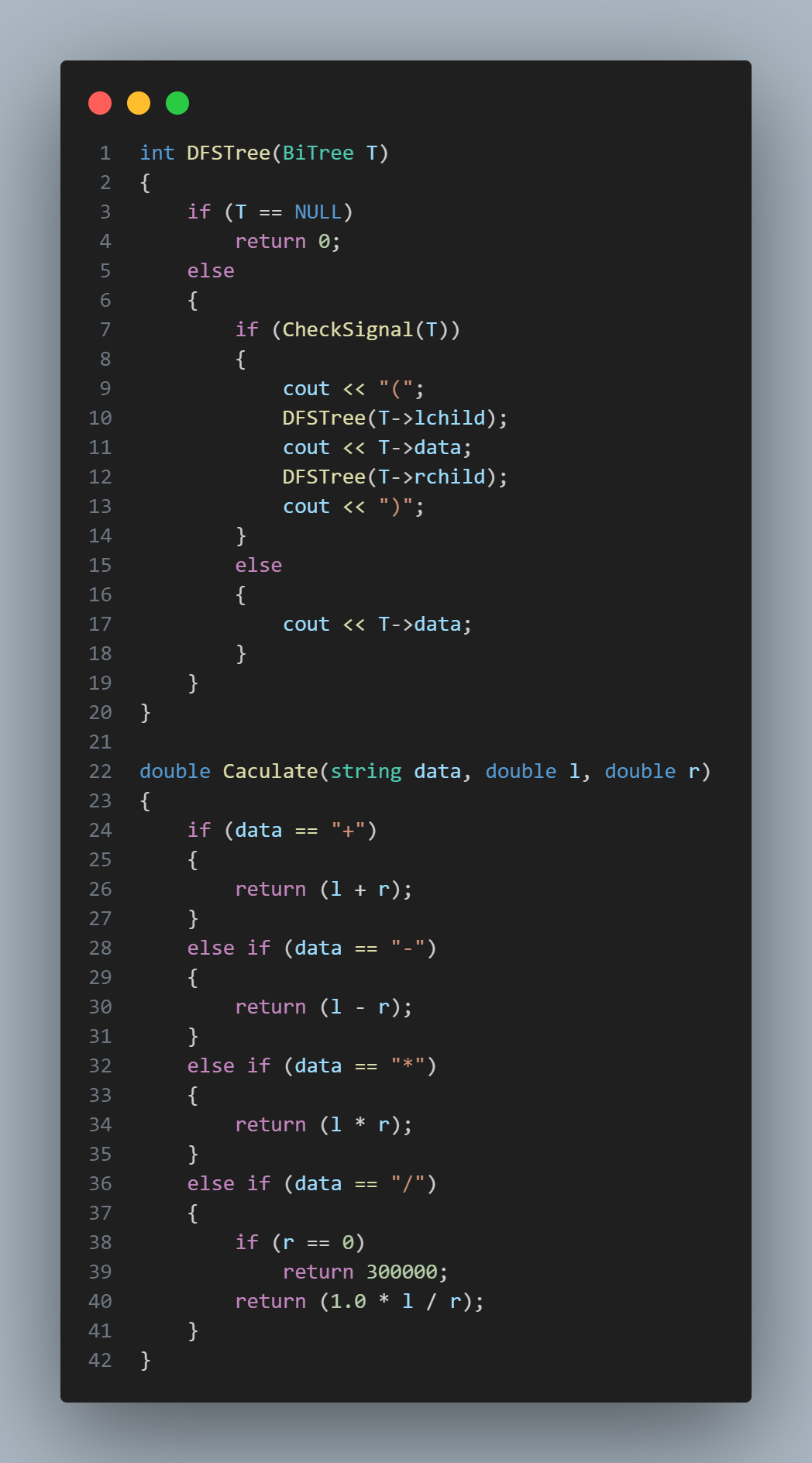
### 问题 N: 二叉树的创建和文本显示

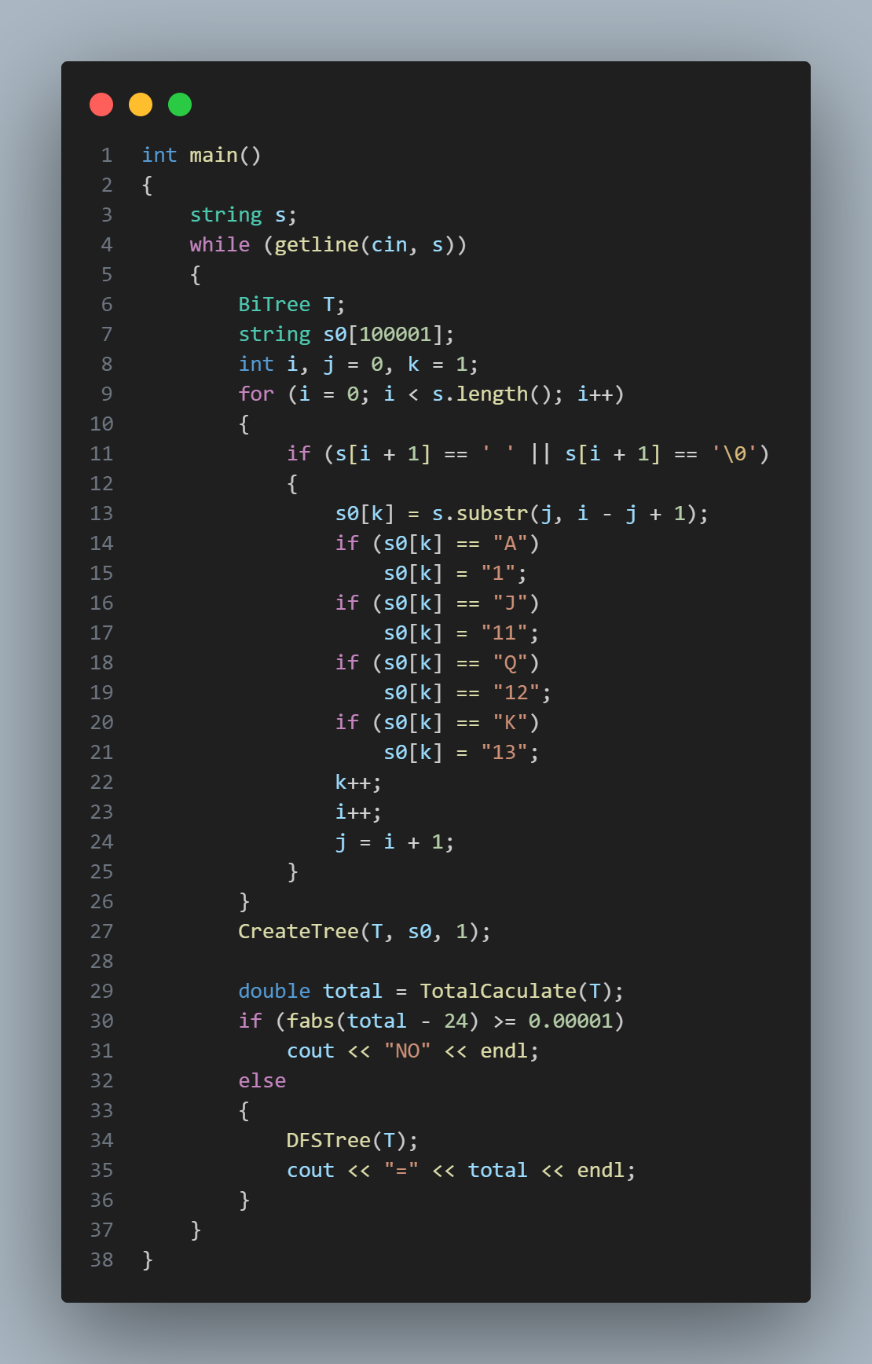
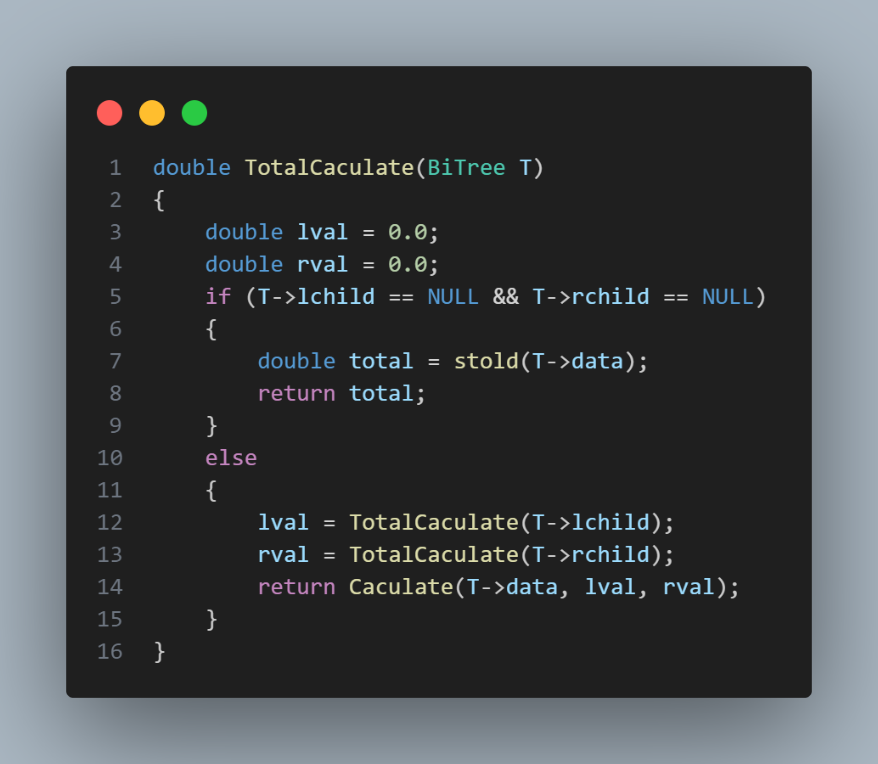




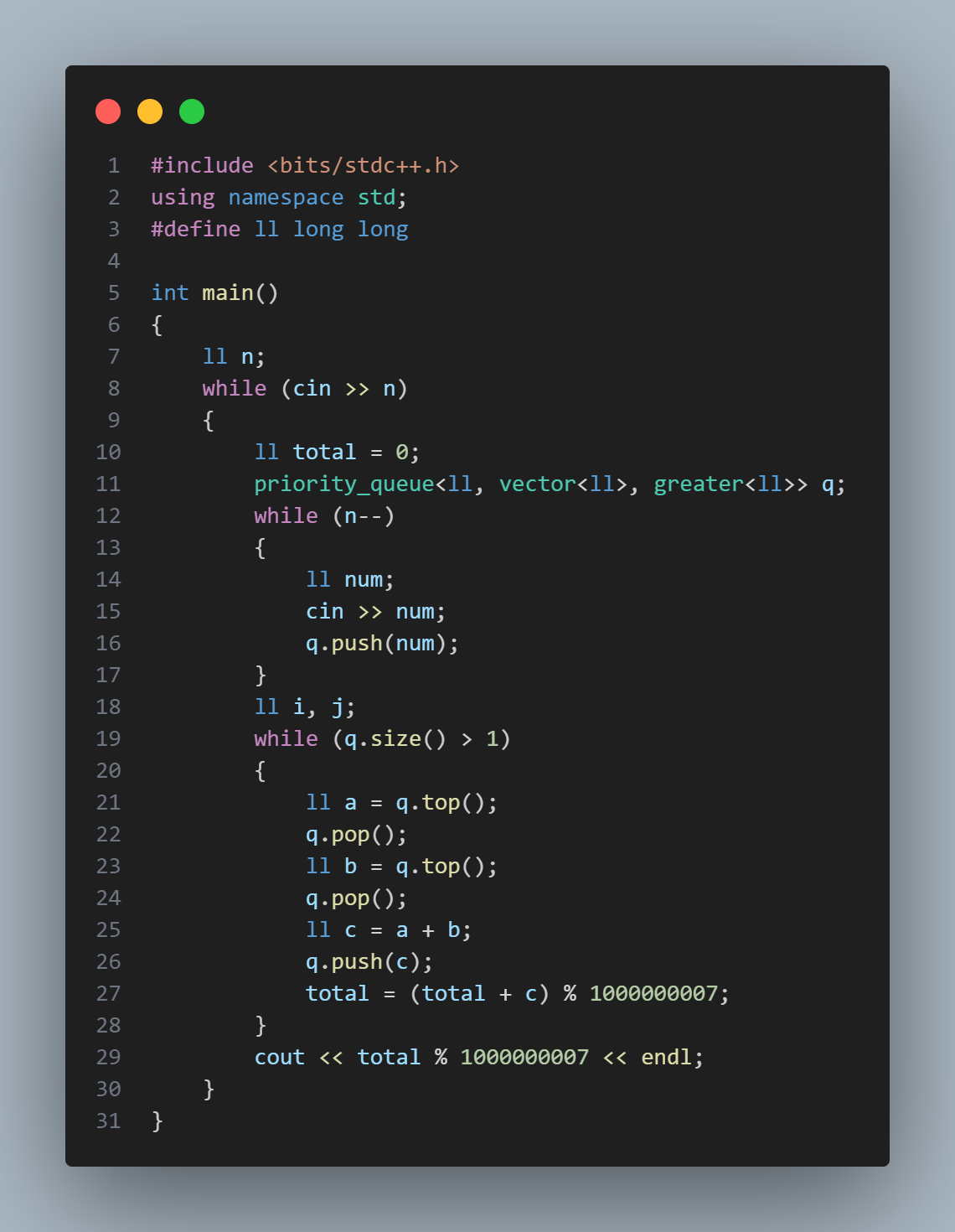
### 问题 Q: 24点游戏(Ⅰ)







### 问题 W: 带权路径长度



### 问题 Y: 最小时间

