

《 算法与数据结构 》

实验报告本

|  |  |
| --- | --- |
| 班 级： | **智机试验2301** |
| 学 号： | **23010341** |
| 姓 名： | **杳泽** |
| 指导教师： | **赵敏** |

信息科学与工程学院

2024年 10 月

**实 验 报 告 （ 1 ）**

|  |  |
| --- | --- |
| **实验名称**：线性结构实验 | **实验地点**：信息楼211、宿舍 |
| **所使用的工具软件及环境：**  windows 11版本 10.0.22631 版本 22631:  GNU gdb (GDB) 7.6.1,  gcc version 6.3.0 (MinGW.org GCC-6.3.0-1) win32,  Vscode 1.94.2  Ubuntu 24.04 LTS :  gcc version 13.2.0 (Ubuntu 13.2.0-23ubuntu4) posix,  GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git,  valgrind-3.22.0 | |
| **一、实验目的：**  1.熟悉数据结构和编程语言的集成开发环境，掌握程序设计与实现的能力，分析算法的复杂度。  2.要求掌握线性表的基本操作：插入、删除、查找等运算在顺序存储结构和链式存储结构上的运算。  3.熟练掌握堆栈和队列的基本操作，栈在表达式求解中的应用，双端队列的应用。 | |
| **二、实验内容及要求**（题目内容及输入输出要求，任选三题完成）**：**  1.请设计实现两个链式存储的一元多项式乘法运算的算法（可以参考P89的代码3.19），并分析该算法的时间复杂度。  2.请利用堆栈求中缀表达式值。先编写程序将中缀表达式转换为后缀表达式，再结合教材中求后缀表达式程序，求解中缀表达式的值。  3.双端队列（deque，即double-ended queue的缩写）是一种具有队列和栈性质的数据结构，即可以（也只能）在线性表的两端进行插入（入队）和删除（出队）操作，也就是说元素可以从队头出队和入队，也可以从队尾出队和入队。若以顺序存储方式实现双端队列，请编写例程实现下列操作。（1）Push（X,D）：将元素X插入到双端队列的头；（2）Pop（D)：删除双端队列的头元素，并返回；（3）Inject（X,D）：将元素X插入到双端队列的尾部；（4）Eject（D)：删除双端队列的尾部元素，并返回。  4.单链表分段逆转：给定一个带头结点的单链表和一个整数K，要求将链表中的每K个结点做一次逆转。  要求1：设计人性化的输入输出，可将实验结果清晰的表示出来。  要求2：请将.docx文件以及.c程序上传至超星平台“第3章实验报告”。 | |
| **三、程序设计运行结果（说明设计思路，解释使用的数据结构，计算时空复杂度，核心代码，运行界面截图）**  **1、设计思路**  第2题有些过于注重于程序健壮性的注释以及编程规范了，valgrind检查内存泄漏花了太多时间，导致 3和4实在没时间查内存是否完全安全了。//不考虑exit引起的内存泄漏;  1. 将多项式P2的每一项分别与P1多项式相乘，将形成的新项插入到中间结果多项式，该中间结果多项式一开始为空，并以指数递减的顺序维持当前的运算中间状态。当有新项需要插入时，相当于在一个递减链表中插入一个新节点，并维持递减顺序。如果插入的新节点的指数与链表中的某节点指数一样，则将他们的系数相加；如果相加后结果为0，则从中间结果链表中删除相应节点，否则更改链表中的系数值；如果不存在指数相同的节点，则将新节点插入到相应位置。时间复杂度应为O(n^2);采取模块化编程，将声明写在头文件内，定义写在源文件内，不同功能的函数放在不同的源文件内。使用gdb进行断点调试，valgrind进行内存泄露检查。将输入案例存储在txt文件内，并使用< 重定向符来用输入案例输入。编写了人性化的输入输出，并且完备了内存回收机制，正常运行时可以完全通过valgrind的各项检查。//不考虑exit（）退出的内存回收。  2.考虑了表达式输入形式错误，如括号未闭合，除零等情况。但暂时没有考虑小数运算，因为这会对数据结构产生较大影响。  中缀表达式转后缀表达式：生成临时堆栈s和后缀堆栈back\_s；  若遇到操作数，直接存入back\_s后缀堆栈；  若遇到左括号，将其压入临时堆栈s；  若遇到右括号，表明括号扫描完毕，s栈顶运算符退栈并存入后缀堆栈back\_s，直到遇到左括号，左括号退栈但不存入后缀堆栈。  若遇到运算符，若该运算符优先级大于s栈顶运算符优先级，则压入s栈；若优先级小于等于s栈顶运算符，将s栈顶运算符退栈并存入back\_s，再比较新的s栈顶运算符，按同样方法处理，直到该运算符大于s栈顶运算符优先级为止，将该运算符压入s栈；  若中缀表达式各对象处理完毕，则把s栈内留存的运算符依次退栈并入栈back\_s，最后将back\_s内的数据逆转（因为back\_s原本的出栈顺序和输出顺序是反的，应该用队列存）  规范化函数声明的写法，使得鼠标悬停在函数名上可以看见该函数的相关提示。  采取模块化编程，将声明写在头文件内，定义写在源文件内，使用#pragma once 预编译指令防止头文件被多次编译，不同功能的函数放在不同的源文件内。重定向stderr用来生成log文件辅助调试。使用gdb进行断点调试，valgrind进行内存泄露检查。  编写了人性化的输入输出，并且完备了内存回收机制，正常运行时可以完全通过valgrind的各项检查。//不考虑exit（）退出的内存回收。  3. 参考了实验一ppt的实现思路，自己编写了双端队列的初始化：分配Deque内存->分配Data的内存->初始化Data内存->初始化front 和rear以及maxsize 成员；  自己编写了人性化的输入输出，格式化了输出，并且打印了数据的内存地址和rear以及front的指向。采取模块化编程，将声明写在头文件内，定义写在源文件内，使用#pragma once 预编译指令防止头文件被多次编译，不同功能的函数放在不同的源文件内。重定向stderr用来生成log文件辅助调试。使用gdb进行断点调试，valgrind进行内存泄露检查。Valrind检查无内存泄漏，但有其他潜在安全风险（实在没时间排查了，有时候在对堆栈数据使用strcpy都会触发valgrind的报错，2的复制堆栈就是自己写的函数，没用strcpy）。  4.参考了实验一ppt的实现思路。自己编写了人性化的输入输出，格式化了输出，并且打印了数据的内存地址。采取模块化编程，将声明写在头文件内，定义写在源文件内，使用#pragma once 预编译指令防止头文件被多次编译，不同功能的函数放在不同的源文件内。重定向stderr用来生成log文件辅助调试。使用gdb进行断点调试。  **2、数据结构的描述**  1.//详情参见poly.h 以及相关.h 头文件  typedef struct PolyNode \*PtrToPolyNode;  /\*  @brief 多项式节点  \*/  struct PolyNode{      int coef;       //系数      int expon;      //指数      PtrToPolyNode Next;     //下一个节点指针  };  typedef PtrToPolyNode Polynomial;  2.//详情参见stack.h mid\_to\_back.h 这两个头文件只包含声明,不包含定义,编写了标准化的函数注解,总长超过100行  /\*  @brief stack 栈节点 非链表形式 动态分配  @param element\_type \*data 存放的数据  @param position top     栈顶元素下标  @param int max\_size  容量  \*/  struct snode{      element\_type \*data;      //个人感觉 这个top可用可不用 因为可以根据strlen和max\_size的关系直接判断存储元素的位置和栈是否满       //不过top确实方便      position top;      int max\_size;  };  typedef struct snode \*stack;  3. //详情见exp-1/1-3/main.c  typedef char ElementType ;  typedef int Position;  typedef struct QNode \*Deque;  struct QNode {      //队列结构      ElementType \*Data;      //数据      Position Front, Rear;       //首元素的下标和尾元素+1的下标      int MAXSIZE;        //双端队列容纳的最大数据容量  };  4.//详情见exp-1/1-4/main.c  //23010341 杳泽  typedef char element\_type;       //使用char类型数据  // 定义链表结点结构体  typedef struct ListNode {      element\_type val;           //节点存储的值      struct ListNode \*next;  //指向下一个节点的指针  }ListNode;  **3、程序代码**  //exp-1/1-1  //echo.h  #include "poly.h"  /\*  @brief 格式化输出多项式  @param Polynomial 要输出的多项式的表头  \*/  void printpoly(Polynomial p)  ;  /\*  @brief 原样输出多项式  \*/  void printpoly\_raw(Polynomial p);    //echo.c  #include "echo.h"  #include "poly.h"    void printpoly(Polynomial p)        //格式化输出多项式  {      printf("\n多项式:");      if(!p) printf("0 0\n");         //输出0多项式      else      {          printf("%+d(X1)^%d",p->coef,p->expon);      //格式化输出,显示系数符号          p=p->Next;          for(int i=2;p;i++)          {              printf("%+d(X%d)^%d",p->coef,i,p->expon);   //格式化输出,显示系数符号              p=p->Next;          }          printf("\n");      }      return;  }  void printpoly\_raw(Polynomial p)            //原样输出   多项式     23010341杳泽  {      printf("\n多项式:系数 指数\n");      if(!p) printf("0 0\n");      else      {          printf("%d %d ",p->coef,p->expon);          p=p->Next;          for(int i=2;p;i++)          {              printf("%d %d ",p->coef,p->expon);              p=p->Next;          }          printf("\n");      }      return;  }  //main.c  #include "poly.h"  #include "echo.h"    int main()  {      Polynomial p1,p2,p;      p1=ReadPoly();      p2=ReadPoly();      p=Mult(p1,p2);      printpoly(p);       //格式化输出      printpoly\_raw(p);       //原样输出      del\_poly(p1);      del\_poly(p2);      del\_poly(p);      return 0;  }  //poly.h  //多项式相关   23010341 杳泽  #include <stdio.h>  #include <stdlib.h>    /\*  必须加 #pragma once  \*/  #pragma once  typedef struct PolyNode \*PtrToPolyNode;  /\*  @brief 多项式节点  \*/  struct PolyNode{      int coef;       //系数      int expon;      //指数      PtrToPolyNode Next;     //下一个节点指针  };  typedef PtrToPolyNode Polynomial;    /\*  @brief 插入多项式节点  @param int coef 项的系数  @param int expon 指数  @param  @return None  \*/  void attach(int coef,int expon,Polynomial \*PtrRear);  //23010341  /\*  @brief 通过scanf建立多项式  @return Polynomial 多项式表头  \*/  Polynomial ReadPoly();  /\*  @brief 多项式乘法  @param Polynomial   第1个多项式  @param Polynomial       第2个 23010341 杳泽  @return Polynomial 多项式相乘的结果多项式的表头  \*/  Polynomial Mult(Polynomial p1,Polynomial p2);  /\*  @brief 删除整个多项式 并free 防止内存泄漏  @param Polynomial 要删除的多项式表头  \*/  void del\_poly(Polynomial p);  //poly.c  //多项式相关的源文件  #include "poly.h"    void attach(int coef,int expon,Polynomial \*PtrRear)  {      Polynomial p;       //申请新节点      p=(Polynomial)malloc(sizeof(struct PolyNode));      p->coef=coef;      p->expon=expon;      p->Next=NULL;           //存储节点数据      (\*PtrRear)->Next=p;      \*PtrRear=p;         //更新ptrrear      return;  }  Polynomial ReadPoly()  {   //建立多项式      Polynomial p,rear,t;      int coef,expon,n;      printf("请输入项数:");      scanf("%d",&n);      printf("%d\n",n);       //重定向txt输入的回显 echo      p=(Polynomial)malloc(sizeof(struct PolyNode));      //分配头结点      p->Next=NULL;rear=p;      while(n--)      {          printf("系数,指数:");          scanf("%d,%d",&coef,&expon);          printf(" %d,%d ",coef,expon);          attach(coef,expon,&rear);           //尾插节点      }      printf("\n")    ; //重定向txt输入的回显 echo      t=p;p=p->Next;free(t);           //删除临时头结点      return p;  }  Polynomial Mult(Polynomial p1,Polynomial p2)  {      Polynomial p,rear,t1,t2,t;      int coef,expon;      if(!p1||!p2)        return NULL;        //p1,p2有一个为NULL      t1=p1;t2=p2;      p=(Polynomial)malloc(sizeof(struct PolyNode));      p->Next=NULL;      rear=p;      while(t2)       //系数相加          O(n)      {          attach(t1->coef\*t2->coef,t1->expon+t2->expon,&rear);          t2=t2->Next;      }      t1=t1->Next;      while(t1)       //系数相乘      {          t2=p2;rear=p;          while(t2)          {              expon=t1->expon+t2->expon;              coef=t1->coef\*t2->coef;         //找到合适的插入位置              while(rear->Next&&rear->Next->expon>expon)              {                  rear=rear->Next;              }              if(rear->Next&&rear->Next->expon==expon)              {                  if(rear->Next->coef+coef)                  {                      rear->Next->coef+=coef;                  }                  else                  {       //系数相加后为0   删除该项                      t=rear->Next;                      rear->Next=t->Next;                      free(t);                      t=NULL;                  }              }              else              {                  t=(Polynomial)malloc(sizeof(struct PolyNode));  //  23010341杳泽                  t->coef=coef;                  t->expon=expon;                  t->Next=rear->Next;     //                  rear->Next=t;                  rear=rear->Next;              }              t2=t2->Next;          }          t1=t1->Next;      }      t=p;p=p->Next;free(t);t=NULL;      return p;  }    void del\_poly(Polynomial p)  {      Polynomial t;     //临时工作指针      while(p->Next)      //只要p->next不为空就继续删除      {          t=p->Next;      //存储下一个节点地址          free(p);          p=t;      }      free(p);        //释放最后一个节点      p=NULL;      return;  }    //test1.txt 其余见附件压缩包  4  3,4 -5,2 6,1 -2,0  3  5,20 -7,4 3,1  //exp-1/1-2  // calculate-expr.c  #include "calculate-expr.h"  void calculate\_back\_expr(char \*back\_expr)  {      int len = strlen(back\_expr);      stack t = init\_stack(len); //    临时栈      for (int i = 0; i < len; i++)      {          if (isdigit(back\_expr[i])) // 是数字          {              push\_back(t, back\_expr[i]);          }          else          {              switch (back\_expr[i])              {              case '+':              case '-':              case '\*':              case '/':              {                  int op[2] = {0}; // 运算数                  int result = 0;                  for (int j = 0; j < 2; j++) // 二元运算符 弹2个                  {                      if (!is\_empty(t)) // 栈不空                      {                          op[1 - j] = (int)(pop(t) - 48);                      }                      else // 表达式错误                      {                          print\_err("wrong expr! calculate\_back\_expr()");                          exit(expr\_wrong);                      }                  }                  if (back\_expr[i] == '+')                      result = op[0] + op[1];                  else if (back\_expr[i] == '-')                      result = op[0] - op[1];                  else if (back\_expr[i] == '\*')                      result = op[0] \* op[1];                  else // 除法                  {                      if (op[1] != 0)                      {                          result = op[0] / op[1];                      }                      else                      {                          print\_err("divide zero!. calculate\_bacK\_expr()");                          exit(divide\_zero);                      }                  }                  push\_back(t,(char)(result+48));     //运算结果入栈; 理想情况下是整数                  break;              }              default: // 表达式错误                  print\_err("wrong expr! calculate\_back\_expr()");                  exit(expr\_wrong);                  break;              }          }      }      printf("expr-result:%d\n",(int)(pop(t)-48));      free\_stack(t);  }  // calculate-expr.h  //23010341 杳泽  #pragma once  //表达式求值相关  堆栈实现  #include "stack.h"  #include "err.h"  extern const int expr\_wrong;  extern const int divide\_zero;  /\*  @brief 计算后缀表达式的值并输出  @param char\* 后缀表达式  @return None  \*/  void calculate\_back\_expr(char \*back\_expr);    //err.c  #include "err.h"    void print\_err(char \*str)        //perror(str)    并写入stderr  {      perror(str);      fprintf(stderr,"errno:%d\n\n",errno) ;      fprintf(stdout,"%s.errno:%d\n",str,errno);      return ;  }  void redirect\_stderr(char \*str)      //重定向stderr 至str.log  {      char location[20];      sprintf(location,"./%s.log",str);       //生成存储文件名      if(freopen(location, "a", stderr) == NULL)fprintf(stdout,"error redirecting stdout\n");      return ;  }    //err.h  //用于错误输出 23010341杳泽  #pragma once  #include "headers.h"  /\*  @brief perror(str) 并写入stderr  @param char\* str要写入的错误信息  @return None  \*/  void print\_err(char \*str);  /\*  @brief 重定向stderr至 str.log  @param  char\* 重定向后的文件主名  @return None  \*/  void redirect\_stderr(char \*str) ;    // headers.h  //类似于 #include <std/bitsc++.h>       集合了所需的所有头文件的头文件  #pragma once  #include <errno.h>  #include <stdio.h>  #include <stdbool.h>  #include <stdlib.h>  #include <string.h>  #include <math.h>  #include <time.h>  #include <ctype.h>    // main.c  #include "headers.h"  #include "stack.h"  #include "mid\_to\_back.h"  #include "calculate-expr.h"  #include "err.h"  const int max\_len=31;  const int stack\_init\_fail=-1;  const int push\_back\_fail=-2;  const int scanf\_mid\_to\_back\_fail=-3;  const int expr\_wrong=-4;  const int realloc\_failed=-5;  const int malloc\_failed=-6;  const int divide\_zero=-7;  int main()  {      redirect\_stderr("mid\_to\_back") ;        //重定向stderr 至mid\_to\_back.log      stack back\_expr=init\_stack(31);      back\_expr=scanf\_mid\_to\_back(back\_expr);     //      reverse\_data(back\_expr);            //应当使用队列;这里为了方便逆转了一下栈内数据 以达到顺序输出的效果      char \*data=print\_stack(back\_expr);      printf("\n");      calculate\_back\_expr(data);      free\_stack(back\_expr);      free(data);      data=NULL;      return 0;  }  //2+3\*(7-4)+8/4  //13  // 2\*(9+6/3-5)+4  //16  // mid\_to\_back.c  #include "mid\_to\_back.h"  #include "err.h"  stack scanf\_mid\_to\_back(stack back\_s)  {      stack s = init\_stack(31); // 分配空间 临时栈      if (!s)      {          print\_err("malloc failed. scanf\_mid\_to\_back()\n");          exit(scanf\_mid\_to\_back\_fail);      }      char mid[31];      printf("中缀表达式:");      fgets(mid,31,stdin); // 读取字符串      mid[strlen(mid)-1]='\0';        //去除结尾\n      for (int i = 0; i < strlen(mid); i++)      {          if (isdigit(mid[i])) // 是操作数          {              push\_back(back\_s, mid[i]); // 存入后缀表达式          }          else          {              switch (mid[i])              {              case '(':                 // 左括号                  push\_back(s, mid[i]); // 存入临时栈                  break;              case ')':             // 右括号                  if (!is\_empty(s)) // 栈非空                  {                      for (char top\_char = pop(s); top\_char != '('; top\_char = pop(s)) // 不是左括号                      {                          push\_back(back\_s, top\_char); // 出栈并存入后缀表达式                          if (is\_empty(s))             // 表达式没有左括号 有问题                          {                              print\_err("wrong expr! scanf\_mid\_to\_back()");                              exit(expr\_wrong);                          }                      }                      // 是左括号就不存                      break;                  }                  else // expr wrong表达式有问题                  {                      print\_err("wrong expr! scanf\_mid\_to\_back()");                      exit(expr\_wrong);                  }                  break;              case '+': // 运算符              case '\*':              case '-':              case '/':                  if (!is\_empty(s)) // 栈非空  s内存贮的只会是运算符/括号                  {                      if (pri\_cmp(mid[i], top(s)) > 0) // 大于栈顶符号运算优先级                      {                          push\_back(s, mid[i]);                          break;                      }                      // 小于等于栈顶运算符优先级                      push\_back(back\_s, pop(s)); // 退栈并存入后缀表达式                      i--;    //仍然判断当前字符                      continue;                  // 继续比较                  }                  else                  {                      push\_back(s, mid[i]); // 空栈 直接压栈                  }                  break;              default: // N/A符号                  print\_err("wrong expr! scanf\_mid\_to\_back()\n");                  exit(scanf\_mid\_to\_back\_fail);              }          }      }      for(int i=0;!is\_empty(s);i++)  //处理完毕  统一出栈并存入后缀表达式      {          push\_back(back\_s,pop(s));      }      free\_stack(s);        //释放s内存      s=NULL;      return back\_s;  }  int pri\_cmp(char op1, char op2)  {      if (priority(op1) < priority(op2))      {          return 1;      }      if (priority(op1) == priority(op2))      {          return 0;      }      if (priority(op1) > priority(op2))      {          return -1;      }  }  int priority(char op)  {      switch (op)      {      case '\*':          return 1;      case '/':          return 1;      case '+':          return 2;      case '-':          return 2;      case '(':   //左括号优先级要最低          return 20;      default:          print\_err("wrong expr! priority()\n");          exit(expr\_wrong);          break;      }  }  // mid\_to\_back.h  #include "stack.h"  extern const int scanf\_mid\_to\_back\_fail;  extern const int expr\_wrong;  /\*  @brief 读取中缀表达式并转为后缀  @param stack 存储表达式的栈  \*/  stack scanf\_mid\_to\_back(stack back\_s);  /\*  @brief 比较2个运算符优先级  @param char 要判断的字符op1,op2  @return int 1:op1优先级高于op2  0:op1==op2  -1:op1<op2  \*/  int pri\_cmp(char op1,char op2);  /\*  @brief 获取运算符优先级  @param char 运算符  @return int 优先级;越小优先级越高。  \*/  int priority(char op);  // stack.c  #include "stack.h"  #include "err.h"  stack init\_stack(int max\_size)  {      stack s = (stack)malloc(sizeof(struct snode));      s->data = (char \*)malloc(sizeof(char) \* max\_size); // 分配空间      if (!s || !s->data)      {          print\_err("stack init err. init\_stack()\n"); // 写入错误信息          exit(stack\_init\_fail);      }      s->data[0] = '\0';      s->max\_size = max\_size;      s->top = -1;      return s;  }  bool is\_empty(stack s)  {      return (s->top == -1) ? true : false;  }  bool is\_full(stack s)  {      return ((s->top) < (s->max\_size - 1)) ? false : true;  }  bool push\_back(stack s, element\_type data)  {      if (!is\_full(s))      {          s->data[++s->top] = data; // 存储元素          return true;      }      else // 栈满      {          return false;      }  }  element\_type top(stack s)  {      if (!is\_empty(s)) // 栈不空      {          return s->data[s->top];      }      else          return '\0';  }  element\_type pop(stack s)  {      char pop\_char;      if (!is\_empty(s)) // 栈非空      {          pop\_char = s->data[s->top];          s->data[s->top] = '\0';          s->top--;          return pop\_char;      }      else          return '\0';  }  char\* print\_stack(stack s)  {      stack cp = cp\_stack(s); // 复制      printf("栈地址:%p\n栈内元素:", s);      char \*data=(char\*)malloc(sizeof(char)\*(s->top+1)+1);        //结尾放\0 方便字符串处理      for(int i=0;!is\_empty(cp);i++)      {            data[i]=pop(cp);          printf("%c ", data[i]);      }      data[s->top+1]='\0';        //字符串结尾      free\_stack(cp);      return data;  }  stack cp\_stack(stack s)  {      stack cp = init\_stack(s->max\_size);      stack\_data\_cp(cp, s);      cp->top = s->top;      cp->max\_size = s->max\_size; // 复制信息      return cp;  }  void free\_stack(stack s)  {      free(s->data); // 清空并释放数据      s->data = NULL;      free(s); // 清空并释放栈      s = NULL;      return;  }  void reverse\_data(stack s)  {      char tmp='\0';      int len=s->top+1;       //数据长度      for(int i=0;i<(int)(len/2);i++)      {          tmp=s->data[i];     //记录头部元素          s->data[i]=s->data[len-1-i];        //交换头尾数据          s->data[len-1-i]=tmp;      }      return;  }  bool stack\_data\_cp(stack s1,stack s2)  {      if ((s1->max\_size)<(s2->top+1))  //无法容纳      {          return false;      }      for(int i=0;i<(s2->top+1);i++)      {          s1->data[i]=s2->data[i];        //复制      }      return true;  }    // stack.h  //栈实现相关  //23010341 杳泽  #pragma once  #include "headers.h"    //表达式最大长度限制  extern const int max\_len;  extern const int stack\_init\_fail;  extern const int push\_back\_fail;  extern const int realloc\_failed;  extern const int malloc\_failed;  typedef char element\_type;  typedef int position;    /\*  @brief stack 栈节点 非链表形式 动态分配  @param element\_type \*data 存放的数据  @param position top     栈顶元素下标  @param int max\_size  容量  \*/  struct snode{      element\_type \*data;      //个人感觉 这个top可用可不用 因为可以根据strlen和max\_size的关系直接判断存储元素的位置和栈是否满       //不过top确实方便      position top;      int max\_size;  };  typedef struct snode \*stack;    /\*  @brief 初始化容量为max\_size的空栈 构造函数  @param int max\_size 栈容量  @return stack  栈的表头  \*/  stack init\_stack(int max\_size);  /\*  @brief 判断栈空  @param stack s 栈的指针  @return bool true->栈空  \*/  bool is\_empty(stack s);  /\*  @brief 判断栈满  @param stack s栈的指针  @return true->栈满  \*/  bool is\_full(stack s);  /\*  @brief 入栈  @param element\_type data入栈数据  @param stack s栈的指针  @return bool true->成功  \*/  bool push\_back(stack s,element\_type data);  /\*  brief 获取栈顶元素  @param stack s 栈的指针  @return element\_type 栈顶元素 | element\_type '\0'(如果是空栈)  \*/  element\_type top(stack s);  /\*  @brief 出栈并返回  @param stack s 栈的指针  @return element\_type 出栈的元素 | '\0'  如果是空栈  \*/  element\_type pop(stack s);  /\*  @brief 打印栈内存储数据  @param char\*,stack 存储数据的字符串指针,要打印的栈的指针  @return 栈内数据  \*/  char\* print\_stack(stack s);  /\*  @brief 深复制 栈 (占据新的内存空间)  @param stack 要复制的栈  @return stack 复制栈的指针  \*/  stack cp\_stack(stack s);  /\*  @brief 清空 并free 掉栈  @param stack 栈的指针  @return None  \*/  void free\_stack(stack s);  /\*  @brief 颠倒顺序,栈中的数据  @param stack 栈的指针  \*/  void reverse\_data(stack s);    /\*  @brief 复制s2->data 至s1->data  替代strcpy  valgrind 检查使用strcpy有问题 所以才写的这个函数  ==40410== Conditional jump or move depends on uninitialised value(s)  ==40410==    at 0x484F397: strcpy (in /usr/libexec/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==40410==    by 0x109B24: cp\_stack (stack.c:81)  ==40410==    by 0x109A81: print\_stack (stack.c:68)  ==40410==    by 0x109452: main (main.c:20)  @param stack,stack 目标栈指针,要复制的栈的指针  @return bool 成功返回true  \*/  bool stack\_data\_cp(stack s1,stack s2);    //exp-1/1-3  // main.c  //23010341 杳泽  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <time.h>  #include <errno.h>  #include <stdbool.h>  #define ERROR -1;  typedef char ElementType ;  typedef int Position;  typedef struct QNode \*Deque;  struct QNode {      //队列结构      ElementType \*Data;      //数据      Position Front, Rear;       //首元素的下标和尾元素+1的下标      int MAXSIZE;        //双端队列容纳的最大数据容量  };  //初始化双端队列  Deque init\_deque(int max\_size)  {      Deque head=(Deque)malloc(sizeof(struct QNode));      head->Data=(char \*)malloc(sizeof(char)\*max\_size);      head->MAXSIZE=max\_size;      head->Front=0;      head->Rear=0;       //初始化      return head;    //返回  }  //判断是否满队列  bool IsFull(Deque Q)  {      if((Q->Rear+1)%Q->MAXSIZE == Q->Front)      {          return true;      }      return false;  }  //判断队列是否为空  bool IsEmpty(Deque Q)  {      if(Q->Rear == Q->Front)      {          return true;      }      return false;  }    //删除头部数据  ElementType Pop(Deque Q)  {      if(IsEmpty(Q))      {          printf("队列空！\n");          return ERROR;      }      ElementType data = Q->Data[Q->Front];      Q->Front=(Q->Front+1)%Q->MAXSIZE; // 向后移动front指针      return data;  }  //尾插数据  bool Inject(Deque Q,ElementType x)  {      if(IsFull(Q))      {          printf("队列满！\n");          return false;      }      Q->Data[Q->Rear] = x; // 数据插入      Q->Rear = (Q->Rear+1)%Q->MAXSIZE; // 向后移动rear指针      return true;  }  //头插数据  bool Push(Deque Q, ElementType x)  {      if(IsFull(Q))      {          printf("队列已满！\n");          return false;      }      // 计算要插入的位置（数组下标）      Q->Front = (Q->Front-1+Q->MAXSIZE)%Q->MAXSIZE;      // 数据插入      Q->Data[Q->Front] = x;      return true;  }  //尾部删除数据  ElementType Eject(Deque Q)  {      if(IsEmpty(Q))      {          printf("队列空！\n");          return ERROR;      }      // 向前移动rear指针      Q->Rear=(Q->Rear-1+Q->MAXSIZE)%Q->MAXSIZE;      ElementType data=Q->Data[Q->Rear];      return data;  }  //打印Deque内的元素  void print\_deque(Deque Q)  {      for(int i=0;i<Q->MAXSIZE;i++)      {          if(Q->Data[i]<33||Q->Data[i]>126)      //占位符          {              printf("占位符    |");          }          else printf("%c         |",Q->Data[i]);      //打印数据     用\t总是莫名其妙对不齐      }      printf("\n");      for(int i=0;i<Q->MAXSIZE;i++)      {          printf("[%p]|",&Q->Data[i]);    //打印地址      }      printf("\t Front:%p,Rear:%p",&Q->Data[Q->Front],&Q->Data[Q->Rear]);      printf("\n\n");  }  void del\_Deque(Deque Q)  {      free(Q->Data);      Q->Data=NULL;      free(Q);      Q=NULL;  }  int main()  {      Deque Q=init\_deque(8);      if(!Push(Q,'a'))    printf("push err. a\n");      if(!Push(Q,'b'))    printf("push err. b\n");      print\_deque(Q);      if(!Inject(Q,'3'))   printf("inject err. b\n");      print\_deque(Q);      Pop(Q);     //头删      Eject(Q);       //尾删      print\_deque(Q);        del\_Deque(Q);      return 0;  }  //exp-1/1-4  //err.c  #include "err.h"    void print\_err(char \*str)        //perror(str)    并写入stderr  {      perror(str);      fprintf(stderr,"errno:%d\n\n",errno) ;      fprintf(stdout,"%s.errno:%d\n",str,errno);      return ;  }  void redirect\_stderr(char \*str)      //重定向stderr 至str.log  {      char location[20];      sprintf(location,"./%s.log",str);       //生成存储文件名      if(freopen(location, "a", stderr) == NULL)fprintf(stdout,"error redirecting stdout\n");      return ;  }    //err.h  //用于错误输出 23010341杳泽  #pragma once  #include "headers.h"  /\*  @brief perror(str) 并写入stderr  @param char\* str要写入的错误信息  @return None  \*/  void print\_err(char \*str);  /\*  @brief 重定向stderr至 str.log  @param  char\* 重定向后的文件主名  @return None  \*/  void redirect\_stderr(char \*str) ;    // headers.h  //类似于 #include <std/bitsc++.h>       集合了所需的所有头文件的头文件  #pragma once  #include <errno.h>  #include <stdio.h>  #include <stdbool.h>  #include <stdlib.h>  #include <string.h>  #include <math.h>  #include <time.h>  #include <ctype.h>    // main.c  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include "err.h"  //23010341 杳泽  typedef char element\_type;       //使用char类型数据  // 定义链表结点结构体  typedef struct ListNode {      element\_type val;           //节点存储的值      struct ListNode \*next;  //指向下一个节点的指针  }ListNode;  // 创建新结点,返回新节点指针  ListNode\* createNode(element\_type val) {      ListNode\* newNode = (ListNode\*)malloc(sizeof(ListNode));        //分配空间      if(!newNode)        //malloc failed      {          print\_err("malloc failed.");          exit(-1);           //直接退出      }      newNode->val = val;      newNode->next = NULL;      return newNode;  }  // 逆转链表  void K\_Reverse(ListNode \*list, int K)  {      int len=0;      //链表元素个数(不含头节点)      ListNode \*first=NULL,\*second=NULL,\*third=NULL,\*s=NULL,\*head=NULL;       //初始化指针      ListNode \*L=list;       //复制链表 头节点指针      while(L->next)      //统计链表元素个数      {          len++;          L=L->next;      }        head=list;      int num=len/K;      while(K>1&&num--)       //K=1无需反转      {          first=head->next;          second=first->next;          s=first;        //记录第一个反转节点          int j=K-1;  //需要反转K-1次          while(j--)      //翻转一次          {              third=second->next;              second->next=first;              first=second;              second=third;          }          head->next=first;          s->next=second;          head=s;      }  }  // 打印链表  void printList(ListNode\* head) {      ListNode\* current = head->next; // 跳过头结点      while (current->next != NULL)      {          printf("%c [%p] -> ", current->val,current);          current = current->next;      }      printf("NULL\n");       //表尾指向NULL  }  // 释放链表内存  void freeList(ListNode\* head) {      ListNode\* current = head;      ListNode\* next;      while (current != NULL) {          next = current->next;          free(current);          current = next;      }  }  int main()  {      redirect\_stderr("exp-1-4.log");        //重定向stderr      // 创建带头结点的单链表      char \*str=(char\*)malloc(sizeof(char)\*40);      if(!str)      {          print\_err("malloc failed. main()->str");          exit(-1);      }      printf("请输入str:");      fgets(str,40,stdin);      str[strlen(str)-1]='\0';   //替换结尾\n 为\0      ListNode \*list=createNode('\0');    //表头指向头节点      ListNode \*t=list;       //工作指针      for(int i=0;i<strlen(str)+1;i++)      {          ListNode \*node=createNode(str[i]);          t->next=node;       //链接          t=t->next;  //指向下一个空位置      }      t=NULL;      printList(list);       //打印逆转前的链表      // 调用函数，K=3      K\_Reverse(list, 3);        printf("Reversed List:\n");      printList(list);      // 释放链表内存      freeList(list);      return 0;  }  **4、运行界面截图**  **1.**    **//文件重定向输入**      **2.**              **//log文件**  **3.**      **4.**    **//K=3**    **//K=2** | |

**成绩：\_\_\_\_\_\_\_\_\_\_\_\_** **任课教师签名：\_\_\_\_\_\_\_\_\_\_\_\_** 2024年 月 日