数据结构上机实验

连接逻辑结构与存储结构

编著：李金龙

组织：中国科学技术大学

时间：5:49, November 18, 2024

版本：0.1.9

用途：本科生《数据结构》实验指导

Can machines think? — Alan Mathison Turing

目录

第一部分

基础知识

第一章 数据结构学习内容概要

内容提要

- 数据结构课程学什么
- 问题建模与逻辑结构
- 存储结构与编程求解问题
- 复杂应用问题的求解算法

给定一个问题，用计算机编程求解，如何迈出第一步，是开始学习计算机程序设计的关键，也是很多学生学计算机面临的第一个困难。如果这第一步跨不出去，可能会让学生丧失学习程序设计的信心和兴趣。数据结构这门课将帮助学生逐步认识程序设计的主要步骤和方法，了解各种程序工作的基本过程和原理，为进一步学习计算机专业的后续课程奠定基础。

1.1 数据结构课程学什么

1.2 问题建模与逻辑结构

1.3 存储结构与编程求解问题

1.4 复杂应用问题的求解算法

第二章 数据结构上机实验常见错误解析(C语言代码)

内容提要

- 指针错误
- 逻辑错误
- 其它类型

说明：每种错误有一两句话来解释说明一下原因，给出例子，同时给出修改的方法和结果，参考第一个错误的写法。相同的错误，可以有多个例子。

2.1 指针错误

1. 调用函数scanf()从键盘输入数据时，实参是保存输入数据的变量的地址，例如

```
int a;
scanf(”%d”,a);
```

编译器显示的错误信息为：”warning: format ‘%d’ expects argument of type ‘int *’, but argument 2 has type ‘int’ ”，即函数scanf的格式串%d对应的参数是保存整数数据的变量地址（int *），但是对应的第二个参数的类型是整型int。所以，修改的方法就是在整型变量a之前应该添加’&’，改成‘scanf(”%d”,&a);’。

2. 下一个指针错误

2.2 逻辑错误

2.3 其它类型错误

第二部分

实验题

第三章 线性结构

内容提要

- 实验目的
- 实验内容
- 实验说明
- 检查/提交要求

3.1 实验目的

复习C程序设计语言知识

- 掌握指针的概念和使用：指针、函数指针、数组名、函数名等
- 预编译指令的使用
- 熟悉写代码的规范
- 掌握代码调试技术
- 学会绘制流程图

3.2 实验内容

- 实验内容一：线性表的链式实现
- 实验内容二：实现顺序栈

3.3 实验说明

1. 实验内容一：线性表的链式实现
   - 完成线性表抽象数据类型ADT的链式存储的设计与实现
   - 利用函数指针调用不同的遍历访问函数
   - 绘制3个流程图：链表中查找给定元素，链表中插入一个节点，链表中删除一个节点。不限制绘制流程图的工具和方法，推荐使用plantuml： https://plantuml.com/zh/
2. 实验内容二：实现顺序栈
   - 完成顺序栈抽象数据类型
   - ADT 的链式存储的设计与实现
   - 使用宏定义实现可以同时定义字符栈、整数栈和结构体栈，并可以使用同一个宏分别对字符栈、整数栈和结构体栈进行操作
3. 可选方案:针对动手编写代码能力尚有欠缺的同学
   - 先阅读代码 linklistc.png,hstack.png（上机实验目录下查找该文件），学习、理解和掌握所列代码
   - 再依据自己的理解，重新输入一遍代码，编译、调试和运行
   - 添加注释（遵循前面的要求）

3.4 检查/提交要求

- 代码能正确编译、运行和输出结果
- 代码注释要求：三行代码至少有一行注释
- 变量、函数等命名有意义
- 提交时间：9月25日24:00时

第四章 高精度数运算

内容提要

- 实验目的
- 实验说明
- 实验内容
- 检查/提交要求

4.1 实验目的

复习线性表、串等 C 语言知识

- 熟练线性表的概念和使用
- 预编译指令的使用
- 熟悉写代码的规范
- 掌握代码调试技术

4.2 实验内容

- 高精度数运算

4.3 实验说明

- 完成线性表抽象数据类型 ADT 的链式存储设计与实现
- 实现实数x(`$1024 < x < 1024$)的加减乘运算，要求运算精确到2^{-n},n 是一个输入参数或预定义参数
- 用十进制或二进制串（线性表）表示实数x
- 构造 ADT，具有读入十进制实数的功能，实现到 N 进制的转换，N 的值在输入时指定且 N 小于 20，实现加减乘三个基本操作，并输出对应的 N 进制结果和十进制结果
- 实现一个复杂操作：单变量多项式求值。例如求函数 $f(x) = \frac{3}{7}x^3\text{-}\frac{1}{3}x^2 + 4$ 的值（其中为$x = 1.4$，精度为：$n = 200$，$\frac{3}{7}$ 和$\frac{1}{3}$等x 前的系数可以不使用高精度）。
- 提示：在运行时解析输入的多项式字符串，分析出每项的系数和幂数，调用基本操作完成计算

4.4 检查/提交要求

- 代码能正确编译、运行和输出结果（能正确完成测试样例即可）
  - 实现二进制和十进制之间的转换: 输入一个高精度十进制小数，转换为高精度二进制小数；输入高精度二进制小数，转换为高精度十进制小数
  - 实现从十进制到 N 进制的转换（其中 N 在输入时指定且 N 小于 20），并实现加减乘操作
    - 输入高精度十进制小数，转换为高精度 N 进制
    - 可以将输入的数进行加减乘计算，并输出对应的十进制和 N 进制结果

- 单变量多项式求值
  - 完成函数$f(x)=\frac{3}{7}x^3\text{-}\frac{1}{3}x^2+4$的值的计算，其中为$x=1.4$，精度为：$n=200$
  - 可以实现一个和示例类似的计算
- 代码注释要求：三行代码至少有一行注释
- 变量、函数等命名有意义
- 提交时间：10月16日24：00

第五章 线性结构的高级操作

内容提要

- 实验目的
- 实验说明
- 实验内容
- 检查/提交要求

5.1 实验目的

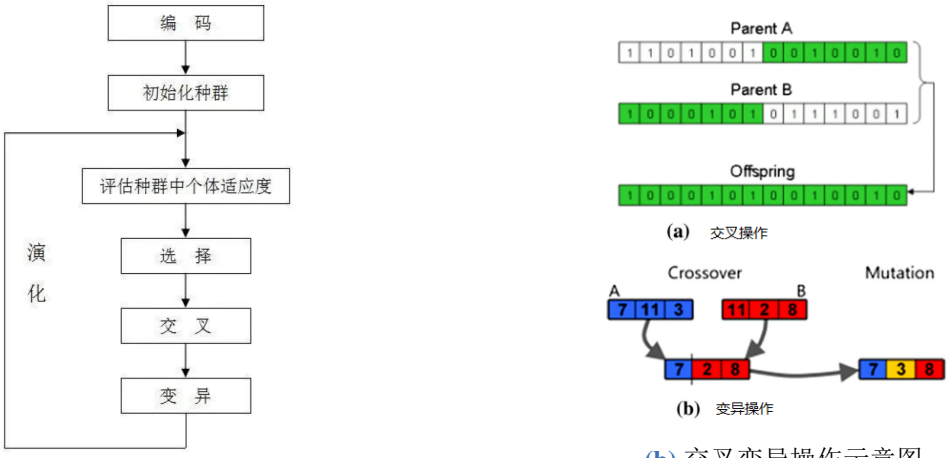
学会广义表的使用，学习编写基础算法

- 了解广义表的构成
- 学会将数据映射到广义表中
- 熟悉写代码的规范
- 掌握代码调试技术

5.2 实验内容

以下题目二选一。

- 进化（遗传）算法求解连续函数的最小值
- 用爬山法求解n-皇后的一个解



(a) 进化算法流程图

(b) 交叉变异操作示意图

图 5.1: 进化算法示意图

5.3 实验说明

1. 实验内容一：进化（遗传）算法求解连续函数的最小值 具体要求: 给定函数$f(x)$，求解其在区间$[0, 16)$ 上的最小值

   进化算法思想：

   - 进化算法/遗传算法是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。直观上：优秀的父母会诞生出优秀的孩子
   - 进化算法流程图如图5.1 (a)所示
     - 编码：二进制编码
     - 初始化：初始个体数为N
     - 评估：这里的评估函数应为函数的值
     - 选择：从个体中选择n个作为父代常用方法：最佳保留法，轮盘赌选择
     - 遗传：染色体交叉
     - 变异：基因突变
     - 遗传变异后得到的子代作为新一轮个体重复评估、选择、遗传、变异
   - 进化算法两个主要的操作：交叉和变异，如图5.1 (b)所示

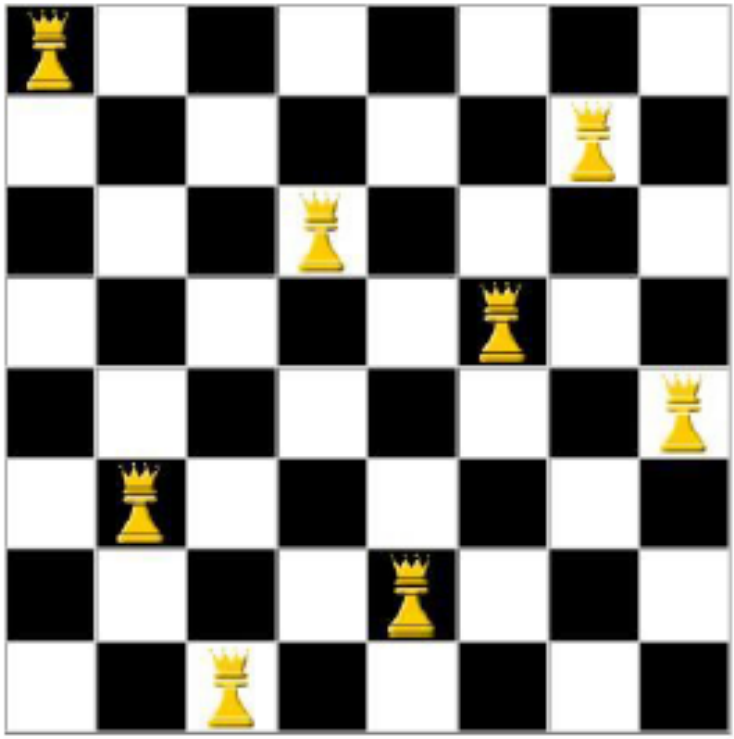


图 5.2: 交叉变异操作示意图

2. 实验内容二：用爬山法求解n-皇后的一个解
   - n-皇后问题的随机搜索算法，找到解即可
   - 问题描述：将n 个皇后放置在$n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击，如图所示

- 我们定义一个长度为n 的一维数组solution，solution[i]=row, 表示第i 列的皇后在row 行，并且solution 是0 到$n-1$ 的一个排列，这样巧妙的避免了在水平与竖直方向上的皇后冲突
- 操作：将其中两个皇后的行数进行一次互换
- 邻居：原棋盘经过一次“操作”后的棋盘
- 目标函数：可相互攻击到的皇后对数。我们希望目标函数越小，目标函数为0 时即为我们找到的解
- 本题中我们使用爬山法求解：每次我们找出初始棋盘的所有邻居中冲突最小的作为新的初始棋盘，若所有邻居均不优于初始棋盘，则随机生成另一个初始棋盘。

5.4 检查/提交要求

- 代码能正确编译、运行和输出结果
- 理清算法思路，需要说明算法是如何实现的
- 代码注释要求：三行代码至少有一行注释
- 变量、函数等命名有意义
- 提交时间：10月30日24:00时

第六章 串的实现

内容提要

- 实验目的
- 实验说明
- 实验内容
- 检查/提交要求

6.1 实验目的

熟练掌握并实现串的 ADT

- 掌握串的定义与基本操作，并利用不同存储方式实现串的 ADT
- 预编译指令的使用
- 熟悉写代码的规范
- 掌握代码调试技术

6.2 实验内容

- 串的ADT的实现

6.3 实验说明

- 采用两种不同的存储方式 (定长顺序结构存储、堆分配存储和块链存储三选二) 实现串的 ADT
- 实现串的基本操作，包括但不限于：
  - 初始化: 初始化串
  - 销毁: 销毁串，释放空间
  - 清空: 清为空串
  - 求长度: 返回串中的元素个数，称为串的长度
  - 模式匹配: 定位子串的位置，要求使用 KMP 算法实现
  - 求子串: 返回某个起始位置的某长度的子串
  - 替换:Replace(S, T, V), S 是主串, 用 V 替换主串 S 中出现的所有与 T 相等的不重叠的子串
  - 拼接: 拼接两个串
  - 遍历: 依次输出串中所有字符

6.4 检查/提交要求

- 代码能正确编译、运行和输出结果，即两种存储方式、基本操作准确无误
- 代码注释要求：三行代码至少有一行注释
- 变量、函数等命名有意义
- 提交时间：11月6日24:00时

第七章 二叉树的实现及应用

内容提要

- 实验目的
- 实验说明
- 实验内容
- 检查/提交要求

7.1 实验目的

掌握二叉树的实现与应用

- 掌握二叉树这一数据结构的代码实现
- 能够应用二叉树结构来实现复杂操作
- 掌握递归的算法思想
- 掌握代码调试技术

7.2 实验内容

- 二叉树的实现及应用

7.3 实验说明

- 读懂老师给出的代码框架中的代码（二叉链表实现的二叉树），几种构造二叉树的方法的代码细节都要读懂，代码下载：
  https://rec.ustc.edu.cn/share/dd8337e0-9241-11ed-b64a-eb675d79527e
- 实现操作：删除节点value=x 的节点及其子树
- 实现操作：给定id1 和id2，输出其最近共同祖先id，节点的id 具有唯一性
- 实现操作：给定id 值，输出从根节点到id 值节点的路径，用“左右右...”的左右孩子指针标记从根到节点的路径
- 编写递归算法，输出二叉树节点中最大的value 和最小的value 之差

7.4 检查/提交要求

- 代码能正确编译、运行和输出结果
- 代码注释要求：三行代码至少有一行注释
- 能回答出助教提问的关于代码框架中代码的问题
- 变量、函数等命名有意义
- 提交时间：11月13日24:00时

第八章 哈夫曼树的实现及应用

内容提要

- 实验目的
- 实验说明
- 实验内容
- 检查/提交要求

8.1 实验目的

掌握哈夫曼树的实现与应用

- 掌握哈夫曼树这一数据结构的代码实现
- 能够应用二叉树结构来实现压缩和解压缩操作
- 掌握哈夫曼编码的算法思想
- 掌握代码调试技术

8.2 实验内容

- 哈夫曼树的实现及应用

8.3 实验说明

- 读懂老师给出的代码框架中的代码（结构体数组实现的哈夫曼树），代码细节也要读懂，然后自己使用三叉链表(” 三叉” 指*parent, *lchild 和*rchild) 实现哈夫曼树，并实现以下操作
- 实现操作：压缩并解压一个文件(如.txt 文件)
- 代码下载：
  https://rec.ustc.edu.cn/share/f3b11a00-916e-11ed-8453-356902ba045b,哈夫曼树的显示web显示部分需要依据实验五的参考代码修改。

8.4 检查/提交要求

- 代码能正确编译、运行和输出结果
- 代码注释要求：三行代码至少有一行注释
- 能回答出助教提问的关于代码框架中代码的问题
- 变量、函数等命名有意义
- 提交时间：11月20日24:00时

第九章 图的 ADT 的实现

内容提要

- ❑ 实验目的
- ❑ 实验内容
- ❑ 实验说明
- ❑ 检查/提交要求

9.1 实验目的

掌握图相关的知识

- 掌握图的概念和使用
- 预编译指令的使用
- 熟悉写代码的规范
- 掌握代码调试技术

9.2 实验内容

- 图的 ADT 的实现

9.3 实验说明

- 完成图的 ADT 的设计与实现
- 读懂给出代码框架
- 实现图的基本操作包括，但是不限于：图的创建、删除顶点、增加顶点、增加边、删除边、查找顶点、修改边或顶点
- 图的存储结构自行设计，同时设计实现一个复杂操作，验证上述基本操作，并说明采该存储结构实现这个复杂操作是否合适（用代码注释说明，且向助教说明结构和操作是否适配
- 代码框架下载：
  `https://rec.ustc.edu.cn/share/3f8c3a50-9244-11ed-94f4-67e66ab69b5a`

9.4 检查/提交要求

- 代码能正确编译、运行和实现并验证图的基本操作和自己设计的复杂操作，可以通过使用的存储结构输出整个图以验证操作的实现是否正确
- 代码注释要求：三行代码至少有一行注释
- 能回答出助教提问的关于代码框架中代码的问题
- 变量、函数等命名有意义
- 提交时间：12月4日24:00时

第十章 图的高级操作实现

内容提要

- 实验目的
- 实验说明
- 实验内容
- 检查/提交要求

10.1 实验目的

掌握、理解图相关的知识，实现图的复杂算法

- 掌握图典型算法的实现
- 预编译指令的使用
- 熟悉写代码的规范
- 掌握代码调试技术

10.2 实验内容

- 图的高级操作实现

10.3 实验说明

- 基于实验 7 的 ADT，实现 4 个操作
- 实现基于 Fringe 集合的图搜索算法，包括广度优先搜索，深度优先搜索和 Dijistra 算法
- 实现求图的联通片的数量的算法
- 代码框架下载：
  `https://rec.ustc.edu.cn/share/3f8c3a50-9244-11ed-94f4-67e66ab69b5a`

10.4 检查/提交要求

- 代码能正确编译、运行和实现并验证要求的 4 个操作
  - 广度优先搜索，依次输出搜索时经过的结点
  - 深度优先搜索，依次输出搜索时经过的结点
  - Dijistra，依次输出遍历的结点及距离、最短路径
  - 求联通片数量，输出结果
- 代码注释要求：三行代码至少有一行注释
- 能回答出助教提问的关于代码框架中代码的问题
- 变量、函数等命名有意义
- 提交时间：12月19日24:00时

第三部分

参考解答

第十一章 实验参考代码

11.1 实验一参考代码

11.1.1 熟悉链表操作

代码下载链接：https://rec.ustc.edu.cn/share/3e386a60-9169-11ed-ba57-4dfe7b98e8ee

```
#include <stdio.h>
#include <stdlib.h>

//需要定义什么样的链表，就将数据域的数据类型定义为ElemType
#define ElemType int

//用于调试，把下面这句宏定义注释掉，那么宏定义#ifdef ...# endif 包括的语句在编译时全部忽略
#define _DEBUG_ 1


//静态部分的定义
typedef struct Lnode {          //链表结点
    ElemType data;              //结点数据域
    struct Lnode * next;        //结点链域

} LinkNode, *LinkList;

//
LinkNode *first ;          //为型的变量，为单链表的头指针 firstLinkList

//以下先给出基本操作的实现

/*** 初始化链表     initList ()
 *** 在内存中将头指针赋予正确的值，考虑下面三种初始化的方法
 ***/
LinkNode *initList (){        //初始化链表，在内存中生成头结点
    LinkNode *p = (LinkNode *)malloc(sizeof(LinkNode));
    if (!p) {
        printf ("初始化分配头结点失败！\n");
        exit (0);
    }
    p->next = NULL;       //p->指针赋值为空，通常用于判断链表的结束；域不用，故不用赋值nextdata
    return p;
}

#ifdef _DEBUG_

LinkNode head;          //定一个头结点，结构体变量，全局变量
```

```
/*** 考虑下面这两种初始化 ***/
void  initList1 (LinkNode *p){
    p = (LinkNode *)malloc(sizeof(LinkNode));
    if (!p) {
        printf ("初始化分配头结点失败！\n");
        exit (0);
    }
    p->next = NULL;       //p->指针赋值为空，通常用于判断链表的结束；域不用，故不用赋值nextdata
}

void  initList2 (){
    first  = &head;
    first ->next = NULL;
}
#endif

/*** 求链表的长度    ListLength()
 *** 需要将链表遍历一次时间复杂度为 ,O(n链表长度=)
 ***/
int  ListLength(LinkList  p){
    int  count = 0;
    while (p->next!=NULL){ //为什么直接用指针循环，不用辅助变量也可以？为什么不会丢失链表？p
        count++;
        p = p->next;
    }
    // for( int  count=0;p->next!=NULL;p=p->next,count++);
    return count;
}

/*** 判断链表是否为空    ListEmpty()
 *** 如果为空，返回，否则返回非零，时间复杂度为 0O(1)
 ***/
int  ListEmpty(LinkList  p){
    if (p->next!=NULL)
        return 1;
    return 0;
}

/*** 清空链表    ClearEmpty()
 *** 循环删除链表第一个节点，时间复杂度为 O(n)
 ***/
void  ClearList(LinkList  p){
    LinkNode *q;          //额外辅助变量
    while(p->next!=NULL){
        q = p->next;   //辅助变量指向第一个存放数据的节点数据元素/
        p-> next = q->next;  //头指针的指向第二个元素next
        free(q);          //释放第一个节点占据的存储空间
    }   //循环停止时，头指针的指向，表示没有真正的数据元素存在nextNULL
}
```

```
/*** 销毁链表   DestroyEmpty()
 *** 先清空链表，然后去释放头结点分配的空间，时间复杂度为 O(n)
 ***/
void DestroyList(LinkList p){
    ClearList(p);
    free(p);   //用 initList()初始化，用 initList2()初始化时，不用任何操作。  initList1()呢？
}

/*** 取链表第个节点的数据 i  GetElem()
 *** 遍历链表，注意边界条件的检查，时间复杂度为 O(n)
 *** 返回表示取值失败，非零表示成功 0
 ***/
int GetElem(LinkList p, int i, ElemType *e){
    int k = 0;
    while(p=p->next){ //语言中，赋值表达式的值就是等号左边的值；该句完成赋值的同时，循环判断是否为空Cp
        k++;
        if (k==i){
            *e = p->data;
            return k;
        }
    }
    return 0;
} //为什么不判断输入是否正确？i

/*** 查找链表数据元素为的节点 e  LocateElem()
 *** 遍历链表，时间复杂度为 O(n)
 *** 返回指向数据元素 data=的第一个节点e 适用于数据域是整数或者字符，否则不能用）('='
 ***/
LinkNode *LocateElem(LinkList p, ElemType e){
    while(p=p->next){
        if (p->data == e)
            return p;
    }
    return NULL;
}

/*** 查找链表某个节点的前驱节点   PriorElem()
 *** 遍历链表，注意边界条件的检查，时间复杂度为 O(n)
 *** 返回时，表示没找到  NULLcur_e
 ***/
//void PriorElem(LinkList p, LinkNode *cur_e, LinkNode *pre_e){ 错误为什么？  //, 无法得到想要的返回指针
LinkNode *PriorElem(LinkList p, LinkNode *cur_e){
    for(;p->next;p=p->next)
        if (p->next == cur_e)
            return p;
    return NULL;
}
```

```
/*** 查找链表某个节点的后继节点    NextElem()
 *** 直接返回的指针，时间复杂度为 cur_enextO(1)
 *** 返回时，表示没后继   NULLcur_e; 可以写一个循环来检查是否在链表中cur_e
 ***/
LinkNode *NextElem(LinkList p, LinkNode *cur_e){
    return cur_e->next;
}

/*** 向链表中插入一个节点     ListInsert ()
 *** 遍历链表，查找插入位置，时间复杂度为 O(n)
 *** 插入成功返回插入节点指针，返回时，表示插入失败   NULL
 ***/
LinkNode *ListInsert (LinkList p, int i, ElemType e){
    if (i<1) return NULL;    //位置，太小i
    for (;p;p=p->next)
        if (--i<1){ //找到插入位置
            LinkNode *q = (LinkNode *)malloc(sizeof (LinkNode));
            if (!q) {
                printf ("插入节点时，分配空间失败！\n");
                exit (0) ;
            }
            q->next = p->next; //新节点的指向循环到当前位置的nextpnext
            p->next = q;          // 当前位置的指针指向新节点，这两行代码不能交换次序next
            q->data = e;
            return q;
        }
    return NULL;    //位置，太大，超过链表长度i
}

/*** 从链表中删除一个节点     ListDelete ()
 *** 遍历链表，查找删除位置，时间复杂度为 O(n)
 *** 删除成功返回非零，返回时，表示删除失败 0
 ***/
int ListDelete (LinkList p, int i, ElemType *e){
    if (i<1) return 0;    //位置，太小i
    LinkNode *q = p;          //将视为循环变量，指向的前驱pqp
    for(p=p->next;p;p=p->next){
        if (--i<1){ //找到删除位置
            q->next = p->next; //的指向当的，前驱越过当前节点qnextpnext
            *e = p->data;
            free (p);          //释放空间
            return 1;
        }
        q = p;
    }
    return 0;    //位置，太大，超过链表长度i
}

/*** 定义节点访问函数  visit ()
```

```
 *** 采用函数指针来实现，供 ListTranverse ()函数调用
 ***/
void  PrintLinkNode(LinkNode *p){
      printf ("%d,",p->data);  //适用于是整数data
}

void  Add2(LinkNode *p){
     p->data += 2;     // visit ()函数的实现，每个元素+2
      printf ("+2,");
}

//用于调试代码
void  DebugLinkNode(LinkNode *p){
      printf ("结点-(*addr)=value :  ");
      printf ("(*%lx)=%d\n",p,p->data);  //仅适用于为整数data
}

/*** 遍历链表     ListTraverse ()
 *** 遍历链表，查找删除位置，时间复杂度为 O(n)
 *** 删除成功返回非零，返回时，表示删除失败 0
 ***/
void  ListTraverse (LinkList  p,void  (*ptrFunc)(LinkNode *ptr)){
      printf ("链表表长(=%d)：", ListLength(p));
     while(p=p->next)
          (*ptrFunc)(p);
      printf ("\n");
     //ListLength(p); 错误，为什么？  //
}

int  main(){  //用于测试基本操作

     //void  (*ptrFunc)(LinkNode *p) = PrintLinkNode; 声明函数指针   //
     void  (*ptrFunc)(LinkNode *p) = DebugLinkNode;  //声明函数指针

     //初始化
      first  =  initList ();
     //  initList1 ( first ); 这种初始化怎么样？    //
     //  initList2 (); 如何？          //


     //在不同位置插入数据
     ListTraverse ( first ,ptrFunc);
      ListInsert ( first ,1,2) ;
      ListInsert ( first ,1,3) ;
      ListInsert ( first ,1,4) ;
      ListInsert ( first ,1,5) ;
     ListTraverse ( first ,ptrFunc);
      ListInsert ( first ,1,6) ;
      ListInsert ( first ,1,7) ;
```

```
ListInsert ( first ,1,8) ;
ListInsert ( first ,1,9) ;
ListTraverse ( first ,ptrFunc);
ListInsert ( first ,3,666) ;
ListInsert ( first ,5,777) ;
ListInsert ( first ,7,888) ;
ListInsert ( first ,9,999) ;
ListTraverse ( first ,ptrFunc);

//查找第个数据或者值为的数据ix
ElemType ei;
printf (”取数据之前 %d ——”, ei);
GetElem( first ,10,&ei);
printf (”读取的数据为 %d \n”,ei);

LinkNode *q = LocateElem( first , 888);
if (!q)
    printf (”没找到值所对应的结点\n”);
else{
    q = PriorElem( first ,q);
    printf (”找到结点的前驱为 %d —— ”, q->data);
    printf (”找到结点为 %d —— ”, q->next->data);
    if (q->next->next)
        printf (”找到结点的后继为 %d ”, NextElem( first ,NextElem( first ,q))->data);
    printf (”\n”);
}

//删除数据
printf (”删除前的值 %d —— ”, ei);
if ( ListDelete ( first ,15,&ei)>0)
    printf (”删除的值为 %d\n”, ei);
else
    printf (”删除失败 %d \n”, ei);

ListTraverse ( first ,ptrFunc);
printf (”删除前的值 %d —— ”, ei);
if ( ListDelete ( first ,10,&ei)>0)
    printf (”删除的值为 %d\n”, ei);
else
    printf (”删除失败 %d \n”, ei);

printf (”删除前的值 %d —— ”, ei);
if ( ListDelete ( first ,6,&ei)>0)
    printf (”删除的值为 %d\n”, ei);
else
    printf (”删除失败 %d \n”, ei);
ListTraverse ( first ,ptrFunc);

ptrFunc = Add2;                    // visit ()函数定义为Add2()
```

```
    printf ("每个数据元素准备+2\n");
    ListTraverse ( first ,ptrFunc);          //将链表每个数据都加2
    printf ("完成后，新的链表：+2");
    ListTraverse ( first ,PrintLinkNode);   //直接用函数名当参数，进行调用
    ListTraverse ( first ,PrintLinkNode);
    ListTraverse ( first ,Add2);

    //销毁链表，销毁过程中调用了清空链表
    DestroyList ( first );

    return 0;
}
```

11.1.2 顺序栈的实现

代码下载链接：https://rec.ustc.edu.cn/share/5bdda3b0-9169-11ed-a4ad-85472e2d3ddf

```
#include <stdlib.h>
#include <stdio.h>

#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10

//顺序整数栈的静态结构
unsigned    int_stacksize ;   //预分配的栈空间大小
int * int_stacktop_ptr ;      //栈顶指针
int * int_stackbase_ptr ;     //栈底指针

/***
 *** 下面用宏定义来实现 8 个栈的基本操作
 ***  ## 表示把前后两个参数连接起来
 ***  gcc -E hstack.cpp 只做预编译，输出宏定义展开后的结果，用于粗略检查宏定义是否存在问题
 ***/
#define  initStack ( stack ) stack ## _stackbase_ptr = ( stack *)malloc( sizeof ( stack )*STACK_INIT_SIZE);\
                             if ( stack ## _stackbase_ptr ){\
                                 stack ## _stacktop_ptr = stack ## _stackbase_ptr ;\
                                 stack ## _stacksize = STACK_INIT_SIZE;\
                             } else exit (0)

#define stackEmpty(stack) stack ## _stackbase_ptr == stack ## _stacktop_ptr ? 1:0

#define getTop( stack ,e) stack ## _stackbase_ptr == stack ## _stacktop_ptr ? 0:(e = *( stack ## _stacktop_ptr -1),1)

#define  clearStack ( stack ) stack ## _stacktop_ptr = stack ## _stackbase_ptr

#define  destroyStack ( stack ) free ( stack ## _stackbase_ptr )
```

```
#define stackLength(stack) stack ## _stacktop_ptr - stack ## _stackbase_ptr

#define pop(stack,e) (stack ## _stackbase_ptr == stack ## _stacktop_ptr)? 0:(e =*(--stack ## _stacktop_ptr),1)

#define push(stack,e) if (stack ## _stacktop_ptr - stack ## _stackbase_ptr >= stack ## _stacksize){\
                        stack ## _stackbase_ptr = (stack *)realloc(stack ## _stackbase_ptr, \
                          (stack ## _stacksize + STACKINCREMENT)*sizeof(stack));\
                        if (! stack ## _stackbase_ptr) exit(0);\
                        stack ## _stacktop_ptr = stack ## _stackbase_ptr + stack ## _stacksize;\
                        stack ## _stacksize += STACKINCREMENT;}\
                      *(stack ## _stacktop_ptr++) = e


//栈的遍历操作 stackTraverse() 忽略

/*** 定义其它基本类型的栈，比如字符栈或者结构体栈，只需要定义静态结构，基本操作重复利用上面的宏定义
 *** 下面的例子分别定义了字符栈和结构体栈
 ***/
unsigned  char_stacksize;      //预分配的栈空间大小
char * char_stacktop_ptr;      //栈顶指针
char * char_stackbase_ptr;     //栈底指针

//结构体栈的静态部分，未测试
typedef struct node {          //用给定义的栈取名字typedef
    int data[10];
    float x,y;
} tnode;

unsigned tnode_stacksize;      //预分配的栈空间大小
tnode * tnode_stacktop_ptr;    //栈顶指针
tnode * tnode_stackbase_ptr;   //栈底指针

//定义栈的，目的是为了只用基本操作实现复杂功能，防止误操作ADT
int main(){

    initStack(int);
    initStack(char);
    initStack(tnode);

    //测试整数栈
    int x;
    if (pop(int,x))
        printf("出栈成功 %d\n",x);
    else
        printf("栈空，不能出栈\n");

    printf("栈中有 %d 个元素\n",stackLength(int));

    if (stackEmpty(int))
```

```
        printf ("栈空，无法取栈顶\n");
    else
        if (getTop(int,x))
            printf ("栈顶元素是 %d \n", x);

    push(int,3);
    printf ("栈中有 %d 个元素\n",stackLength(int));

    push(int,4);
    push(int,5);

    printf ("栈中有 %d 个元素\n",stackLength(int));

    if (pop(int,x))
        printf ("出栈成功 %d\n",x);
    else
        printf ("栈空，不能出栈\n");

    printf ("栈中有 %d 个元素\n",stackLength(int));

    if (stackEmpty(int))
        printf ("栈空，无法取栈顶\n");
    else
        if (getTop(int,x))
            printf ("栈顶元素是 %d \n", x);

    clearStack (int);
}
```

11.2 实验二参考代码

11.2.1 高精度运算

代码下载链接：https://rec.ustc.edu.cn/share/51242370-916b-11ed-9aea-07abf44cdf11，提供代码的同学未具名(需要认领的同学联系我)。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 可以使用宏定义来规定精度
#define INTE_MAX 1000
#define DECI_MAX 1000
// 输出精度
#define PRINT_MAX 50
```

```
typedef struct num{ // 高精度数的存储结构
    int negative; // 正负，正号为，负号为01
    int data[INTE_MAX + DECI_MAX]; // 存储数据
    int *number; // 令number = data + DECI_MAX 则num = ... + number[-1] * base^-1 + number[0] * base^0 + number[1]
        * base^1 +...
    int base;
}num;

num* initNUM();
num* ReadFromString(char * str, int base); // 从一个数字字符串读取生成高精度数
void printnum(num* N); // 输出高精度数
num* baseConversion(num* N, int base); // return 转化为A 进制下的数base
num * add(num * A, num* B); // return A + B
num * minus(num * A, num* B); // return A - B
num * multi(num* A,num* B); // return A * B
num * calculate (char * fx,char * x, int inbase, int outbase); // 仅含有数字字母加减乘的多项式计算

int main(){
    num * tmp,* t1 , * t2;
    printf ("测试读取字符串：\n");
    tmp = ReadFromString("-204.2", 10);
    printnum(tmp);

    printf ("测试进制转化：\n");
    t1 = baseConversion(tmp, 2);
    printnum(t1);
    t2 = baseConversion(tmp, 10);
    tmp = ReadFromString("-11001100.00110011001100110011001100110011001100110011001100110011001100", 2);
    t2 = baseConversion(tmp, 10);
    printnum(t2);

    printf ("测试加减乘：\n");
    t1 = ReadFromString("1.14514",10);
    t2 = ReadFromString("-1.919810",10);
    printnum(add(t1,t2));
    printnum(minus(t1,t2));
    printnum(multi(t1,t2));
    printf ("上述转化为进制：7\n");
    printnum(baseConversion(add(t1,t2),7));
    printnum(baseConversion(minus(t1,t2),7));
    printnum(baseConversion(multi(t1,t2),7));

    printf ("多项式测试：\n");
    printf ("测试进制转化：\n");
    tmp = calculate ("-204.2","0",10,2);printnum(tmp); // 将进制的转进制10204.22
    tmp = calculate ("x","-11001100.00110011001100110011001100110011001100110011001100110011001100",2,10);printnum(tmp); // 将进
        制的转进制可以看到，常数多项式可以考虑这两种输入方式2-11001100.0011001110.
    printf ("测试加减乘：\n");
    tmp = calculate ("1.14514+-1.919810","8008208820",10,10);printnum(tmp); // 符号优先级同一般机器语
```

```
        言 http ://home.ustc.edu.cn/~ziheng/pic/test.png
    tmp = calculate ("1.14514--1.919810","8008208820",10,10);printnum(tmp);
    tmp = calculate ("1.14514*-1.919810","8008208820",10,10);printnum(tmp);
    printf ("上述转化为进制：7\n");
    tmp = calculate ("1.14514+-1.919810","0",10,7);printnum(tmp);
    tmp = calculate ("1.14514--1.919810","0",10,7);printnum(tmp);
    tmp = calculate ("1.14514*-1.919810","0",10,7);printnum(tmp);
    printf ("含有变量的多项式测试：\n");
    tmp = calculate ("12.31379131*x*x+x+8.2137619836821388","1.612368921371923122414",10,10);printnum(tmp);
    tmp = calculate ("-12.31379131*x*x+x+-8.2137619836821388","1.612368921371923122414",10,10);printnum(tmp); // 测
        试负号
    tmp = calculate ("x*-12.31379131*x-x-8.2137619836821388","1.612368921371923122414",10,10);printnum(tmp); // 测
        试负号
    return 0;
}

num* initNUM(){
    num* tmp = (num*)malloc(sizeof(num));
    tmp->base = 0;
    tmp->negative = 0;
    tmp->number = tmp->data + DECI_MAX;
    for(int i = -DECI_MAX; i < INTE_MAX; ++i)
        tmp->number[i] = 0;
    return tmp;
}

num* baseConversion(num* N, int base){  // return 转化为A 进制下的数base
    num* tmp = initNUM();
    if(N->base == base){
        *tmp = *N;
        tmp->number = tmp->data + DECI_MAX;
        return tmp;
    }
    tmp->base = base;                   // 进制
    tmp->negative = N->negative;  // 正负与原来一致
    // 整数部分
    for(int i = 0; i < INTE_MAX; ++i){
        for(int j = 0; j < INTE_MAX; ++j)
            tmp->number[j] *= N->base;
        tmp->number[0] += N->number[INTE_MAX - 1 - i];
        for(int j = 0; j < INTE_MAX - 1; ++j)
            if(tmp->number[j] >= base){
                tmp->number[j + 1] += tmp->number[j] / base;
                tmp->number[j] = tmp->number[j] % base;
            }
    }
    // 小数部分
    int deci[DECI_MAX];
    for(int i = 1; i < DECI_MAX; ++i) deci[i] = N->number[-i];  // 保存的小数部分N
    for(int i = 1; i < DECI_MAX; ++i) {
```

```
        deci[0] = 0;
        for (int j = 1; j < DECI_MAX; ++j)
            deci[j] *= base;
        for (int j = DECI_MAX - 1;j > 0; --j) {
            deci[j - 1] += deci[j] / N->base;
            deci[j] = deci[j] % N->base;
        }
        tmp->number[--i] = deci[0];
    }
    return tmp;
}

int abscompare(num * A, num* B){    // 比较 abs(A) 与 abs(B) 的大小
    A = baseConversion(A, B->base);
    for(int i = INTE_MAX - 1; i >= -DECI_MAX; --i) // 从高位比较
        if (A->number[i] != B->number[i]){
            int tmp = A->number[i] - B->number[i];
            free (A);
            return tmp > 0 ? 1 : -1;
        }
    free (A);
    return 0;
}

num * minus(num * A, num* B){   // return A - B
    if (A->negative != B->negative){    // 符号不同AB
        B->negative = 1 - B->negative; // 将符号改为一致AB
        num* tmp = add(A,B);
        B->negative = 1 - B->negative;
        return tmp;
    }
    if (abscompare(A,B) < 0){    // |A| > |B，| 则 A - B 变号，return -(B - A)
        num* tmp = minus(B,A);
        tmp->negative = 1 - tmp->negative;
        return tmp;
    }
    // 同号，且AB|A| > |B| , A - B
    num* tmp = baseConversion(A,B->base);  // tmp = A ，且与进制一样B
    // 借位
    for(int i = -DECI_MAX; i < INTE_MAX; ++i){ // 从低位向高位遍历
        tmp->number[i] -= B->number[i];
        if (tmp->number[i] < 0){
            tmp->number[i + 1] --;
            tmp->number[i] += tmp->base;
        }
    }
    return tmp;
}
```

```
num * add(num * A, num* B){ // return A + B
    if (A->negative != B->negative){    // 符号不同AB
        B->negative = 1 - B->negative; // 将符号改为一致AB
        num* tmp = minus(A,B);
        B->negative = 1 - B->negative;
        return tmp;
    }
    // 同号AB
    num* tmp = baseConversion(A,B->base);  // tmp = A，且与进制一样B
    // 进位
    for(int i = -DECI_MAX; i < INTE_MAX; ++i){ // 从低位向高位遍历
        tmp->number[i] += B->number[i];
        if (tmp->number[i] >= tmp->base){
            tmp->number[i + 1] ++;
            tmp->number[i] -= tmp->base;
        }
    }
    return tmp;
}

num * multi(num* A,num* B){    // A * B
    num* tmp = initNUM();
    A = baseConversion(A,B->base); // 令转化到与一个进制AB
    tmp->negative = A->negative == B->negative ? 0 : 1; // 同号得正，异号为负
    tmp->base = B->base;
    // 我们知道 (a0 + a1*p^1 + a2*p^2 ...) * (b0 + b1*p^1 + .... ) = sum_i (sum_k+j=i ak + bj )*p^i
    for(int i = -DECI_MAX; i < INTE_MAX; ++i){
        for(int j = -DECI_MAX; j < INTE_MAX; ++j)
            if (i - j >= -DECI_MAX && i - j < INTE_MAX) // i - j 也需要满足在精度范围内
                tmp->number[i] += A->number[j] * B->number[i - j]; // sum_i (sum_k+j=i ak + bj ) * p^i 其中是 ki-j
        // 进位
        if (tmp->number[i] >= tmp->base){
            tmp->number[i + 1] = tmp->number[i] / tmp->base;
            tmp->number[i] = tmp->number[i] % tmp->base;
        }
    }
    return tmp;
}

void printnum(num* N){
    if (N->negative) printf("-");
    int i;
    for(i = INTE_MAX - 1;i >= 1 && !N->number[i]; --i); // 从高位到低位找到非位0
    for( ; i >= -PRINT_MAX; --i){
        if (N->number[i] < 10)    printf ("%d",N->number[i]);
        else  if (N->number[i] < 36) printf("%c",'a' + N->number[i]-10);
        if (i == 0)    printf (".");
    }
    printf ("\n");
```

```
}

num* ReadFromString(char * str, int base){  // 从一个数字字符串读取生成高精度数
    // 如 ReadFromString("123546.2321", 10) 返回一个进制下的高精度数10
    num* tmp = initNUM();
    tmp->base = base;
    if( str [0] == '-'){  // 处理负号
        tmp->negative = 1;
        str ++;
    }
    int dot;
    for(dot = 0; str [dot] && str[dot]!= '.'; ++dot);      // 找到小数点所在位置
    for(int i = 0; i < dot; ++i)                  // 整数
        tmp->number[i] = (str[dot - i - 1] >= '0' && str[dot - i - 1] <= '9') ?  str [dot - i - 1] - '0' : str [dot - i
            - 1] - 'a' + 10;  // 数字 or 字
            母
    for(int i = -1; str [dot - i ]; --i ) // 小数
        tmp->number[i] = (str[dot - i] >= '0' && str[dot - i] <= '9') ?  str [dot - i] - '0' : str [dot - i] - 'a' + 10;
              // 数字 or 字
            母
    return tmp;
}

// 提供了一种大一统的输入，可处理所有样例，基本上实现单变量多项式就可以实现所有样例（毕竟常数也是多项式
// 当然也可以自行设计其他类型的输入
/***
 * 是多项式字符串fx
 * 是多项式自变量的值x
 * 是输入多项式与自变量的进制inbase
 * 是最终结果的进制outbase
***/
num *calculate(char * fx,char * x, int inbase, int outbase){     // 仅存在fx 数字字母 +-*
    int i , j;
    char fx1[999] = {0};
    for(i = 0; fx[i] && fx[i] != '+'; ++i); // 检测第一个加法
    if(fx[i]){       // 形如 A + B
        strncpy(fx1 ,fx , i);
        num * t1 = calculate (fx1, x, inbase, outbase);
        num * t2 = calculate (fx + i + 1, x, inbase, outbase);
        num * tmp = add(t1 ,t2); free (t1 ); free (t2);
        t1 = baseConversion(tmp,outbase); free (tmp);
        return t1;
    }

    for(i = j = 0; fx[j ]; ++j)             // 检测最后一个减法
        if(j > 0 && fx[j] == '-' && fx[j - 1] != '*' && fx[j - 1] != '-')    // 需要排除负数的 -
            i = j;
    if(i > 0){  // 形如 A - B 且中全是乘法 B
        strncpy(fx1 ,fx , i);
        num * t1 = calculate (fx1, x, inbase, outbase);
```

```
        num * t2 = calculate(fx + i + 1, x, inbase, outbase);
        num * tmp = minus(t1,t2); free(t1); free(t2);
        t1 = baseConversion(tmp,outbase); free(tmp);
        return t1;
    }

    for(i = 0; fx[i] && fx[i] != '*'; ++i); // 检测第一个乘法
    if(fx[i]){
        strncpy(fx1,fx, i);
        num * t1 = calculate(fx1, x, inbase, outbase);
        num * t2 = calculate(fx + i + 1, x, inbase, outbase);
        num * tmp = multi(t1,t2); free(t1); free(t2);
        t1 = baseConversion(tmp,outbase); free(tmp);
        return t1;
    }
    // 单独一个数
    int negative = 0;
    if(*fx == '-'){
        fx++;
        negative = 1;
    }
    num * tmp = ReadFromString(*fx == 'x' ? x : fx,inbase);
    tmp->negative = negative;
    num * t1 = baseConversion(tmp,outbase); free(tmp);
    return t1;
}
```

11.3 实验三参考代码

11.3.1 进化算法

代码下载链接：https://rec.ustc.edu.cn/share/56843360-916c-11ed-84e9-895d5b7895b6，提供代码的同学未具名(需要认领的同学联系我)。

```
//遗传算法：求解优化问题的通用算法，人工智能中会非常实用
//将问题的解表示为x0串，然后搜索最优的二进制串，使得目标函数值-1f(x)达到最小

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

/** 算法参数设置 **/
//定义串的长度，可用于表示解的精度0-1
#define SLEN 200
```

```
//定义进化最大代数
#define MAXGEN 100

//变异概率
#define mProb 5.0/SLEN

//父群体与子群体大小，子代必须是偶数（通过交叉染色体生成后代时候，一次生成两个后代）
#define PSIZE 20
#define CSIZE 30
#define SIZE (PSIZE + CSIZE)

//一个解的定义
typedef struct {
    int x[SLEN];   //x解的自变量，串:0—1
    double y;      //y=f(x)要优化问题的目标函数值,
} Solution;

//定义一个解集解数组：称之为群体/population
Solution pop[SIZE];                    //解集，父代和子代都存储在这里
Solution *parent = pop;                //当前代，也就是父代
Solution *children = pop + PSIZE;      //子代解集

//函数声明
void printPop(Solution *p,const char *str);

//将串0—1解码为实数x*xo 假定整数,4，bitsSLEN—4为小数部分长度bits
void decode(int *x, double *xo){
    for(int i = *xo = 0; i < SLEN; ++i)
        *xo += x[i] * pow(2,3 - i);
}

//计算y=f(x) ，串 0—1的长度x SLEN
//例子：求y=x*x—3x的最小值+2 ，假设整数部分4，小数部分bitsSLEN—4bits
double func1(int *x){
    double xo;
    decode(x,&xo);   //将串0—1解码成真正的解xxo
    return xo*xo—3*xo+2;    //计算目标函数值
}

//计算一个群体的所有解的目标函数值y ，给出了函数指针，支持个函数的优化
void evaluate(Solution *P, double ptrf(int *)){
    for(int i = 0; i < SIZE; ++i)
        P[i].y = ptrf(P[i].x);

}

//算法初始化：分配两个解集所需的空间，随机生成中的解，并计算其值currentPopy
void initialize (){
```

```
    for(int i = 0; i < PSIZE;++i){  //初始化第一代父代
        for(int j = 0;j < SLEN;++j) //对每个解的串，随机生成0—1
            parent[i].x[j] = rand() % 2;
    }
    evaluate(pop,func1);
}

//从父代中选择两个解，通过杂交生成两个子代个体
//父代两个解通过选择出来（锦标选择）PK
void crossover(){ //交叉算子
    int k = 0;
    while(k < CSIZE){  //逐步生成子代，一次两个
        //随机选择两个父代个体
        //随机确定父代个体染色体交换位点
        int p1 = rand() % PSIZE, p2 = rand() % PSIZE;
        int point = rand() % SLEN;
        for(int i = 0; i < SLEN; ++i){
            children[k].x[i]     = i > point ? parent[p1].x[i] : parent[p2].x[i];
            children[k + 1].x[i] = i < point ? parent[p1].x[i] : parent[p2].x[i];
        }
        k = k + 2;
    }
}

//对子代中的个体进行变异：变异概率为mProb
//所谓变异就是x[j的取值] 互换：0—1 0 <——> 1
void mutate(){ //变异算子
    for(int i=0;i<CSIZE;++i)
        for(int j=0;j<SLEN;++j)
            if((rand()%10000)/10000.0<mProb)
                children[i].x[j] = 1 - children[i].x[j];

}

//从和中选择下一代个体，有多种选择算法，但是通常都是先把两个群体中最好的保留下来 currentPopoffspring ，然后
//方法：选择最好的1个为下一代（截断选择）PSIZE
//方法：给每个个体一个选择概率，2值小（好）的被选择的概率就高，然后依据此概率分布随机采样个yPSIZE
//方法：锦标选择，随机选择3个，相互，留下最好的放入下一代，依次选择个kpkPSIZE （不删除被选择了的）
void select(int k){ //选择算子 ：采用锦标选择
    int best, temp;
    Solution tmp[PSIZE];
    for(int i = 0;i < PSIZE; ++i){ //一个一个子代选择
        best = rand() % SIZE;
        for(int j = 1; j < k; ++j){
            temp = rand() % SIZE;
            if(pop[best].y > pop[temp].y)
                best = temp;
        }
        memcpy(&(tmp[i]),&(pop[best]),sizeof(Solution)); //选择出来的解，复制到临时解集中
```

```
    }
    memcpy(parent,tmp,sizeof(Solution)*PSIZE);
}

//输出群体的信息
void printPop(Solution *p,const char *str){
    printf("%s解集信息如下/: \n", str);
    for (int i=0;i<PSIZE;++i){
        double x;
        decode(p[i].x,&x);
        printf("个体 %3d : y=%10.6lf=f(%2.6ld,",i,p[i].y,x);
        for (int j=0;j<SLEN;++j)
            printf("%d",p[i].x[j]);
        printf(")\n");
    }
}

int main(){
    int seed = 999;
    srand(seed); //设置随机数种子，使得算法结果可以重现
    initialize ();
    printf("第 %d 代 ",0);
    printPop(parent,"当前群体");

    for(int gen = 1;gen < MAXGEN; gen++){
        crossover ();
        mutate();
        evaluate (pop,func1);
        select (4);
        // printf 第(" %d 代 ",gen);
        // printPop(parent 当前群体 ,"");
    }
    printf("第 %d 代 ", MAXGEN);
    printPop(parent,"最终群体");
    return 1;
}
```

11.3.2 爬山法解n皇后问题

版本一代码下载链接：https://rec.ustc.edu.cn/share/ac08b640-916c-11ed-97ba-4d56d2f7f3fe，提供代码的同学未具名(需要认领的同学联系我)。

```
//n皇后问题的随机搜索算法，找到解即可—
//用一维数组来存放每一列皇后的行号

#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>

//采用堆分配存储的方式实现
int *solution; //每列一个皇后， solution[col]=row表示第,列的皇后在行colrow
int len;              //皇后数量，len*的棋盘len
int size;        //每个棋盘邻居棋盘的数量 size=(len-1)*len记录数组最大容量/2

//函数声明
void swap(int , int);
void restart();
void initSolution();
int traverseNeighbors();
int evaluate();

int main(){
    long seed = 820;  //随机数种子，如果这个数不改变，那么每次算法运行的结果都是一样的
    // seed = rand(); 这一句注释掉，等于每次的随机数种子都一样//
    srand(seed);
    //  printf请输入皇后个数：("");
    //  scanf("%d",&len);          //  键盘读入棋盘的大小
    len = 6;     //  棋盘大小
    solution = (int *)malloc(sizeof(int)*len);
    if (!solution) return 0; //分配空间失败

    initSolution();          //  初始化一个棋盘布局

    int flag;
    clock_t t0 = clock();
    while((flag = traverseNeighbors())!=0)
        if (flag<0) restart();
    clock_t t1 = clock();
    printf("冲突数为：%d\得到的解为：n",evaluate());
    printf("所花费的时间为：%5f 秒\n", 1.0*(t1 - t0)/1000);
    for(int i=0;i<len;++i)
        printf("%d,", solution[i]);
    free(solution);
    return 0;
}

/*** 将棋盘的第列和第列交换 ij ***/
void swap(int i, int j){
    if(i == j) return;
    int tmp = solution[j];
    solution[j] = solution[i];
    solution[i] = tmp;
}


/*** 初始化一个棋盘布局将邻居数组准备好 , ***/
```

```
void initSolution (){
    // 随机给初始布局或给定某个初始布局
    // int a[6] = {0,5,2,1,3,4};
    for(int i=0;i<len;++i)
        solution [i]=i;
    for(int i = 0; i < 2*len; ++i)  // 对棋盘进行2*次随机交换len
        swap(rand()%len,rand()%len);
}

/*** 计算棋盘的评价函数（棋盘的价值 : value可相互攻击到的皇后对数）=
 *** 当皇后相互攻击不到时，目标函数的值 value==0
 ***   solution [i]==solution [j ], ( solution [i]—solution [j ]) /( i—j)或时（==1—1语言实现会有），两个皇后彼此能攻击到，
      Cbugvalue++
 *** 棋盘编码方式能保证不会出现  solution [i]==solution [j，故该条件可以不用判断]
 *** 思考：该函数的时间性能是否能提高？
 *** 答案：利用前一个，生成邻居的 valuevalue
 int evaluate(int prevalue, int col1, int col2) {
    int value = 0;
    int i, j;
    if (prevalue == 0) {
        for (i = 0;i < len;i++)
            for (j = i + 1;j < len;j++)
                if (j — i == abs(*(solution + i) — *(solution + j)))
                    value++;
    }
    else {
        value = prevalue;
        for (i = 0;i < len;i++) {
            if (i != col1 && i != col2) {
                if (abs(i — col1) == abs(*(solution + i) — *(solution + col2)))
                    value——;
                if (abs(i — col2) == abs(*(solution + i) — *(solution + col1)))
                    value——;
                if (abs(i — col1) == abs(*(solution + i) — *(solution + col1)))
                    value++;
                if (abs(i — col2) == abs(*(solution + i) — *(solution + col2)))
                    value++;
            }
        }

    }
    return value;
}
 ***/
int evaluate (){
    int value=0;
    //todo
    for(int i = 0; i < len; ++i)
        for(int j = i + 1; j < len; ++j)
```

```
            if( solution[i] - solution[j] == i - j || solution[i] - solution[j] == j - i)
                value++;

    return value;
}

/*** 随机置换 randShuffle() ，无放回的均匀随机采样（被采样集合在不断缩小，直至为，算法停止）0
 *** 对数组 neighbors 进行随机置换操作
 *** 用到的数据： size= 的长度neighbors
 *** best found 策略用不着该函数
 ***/
void randShuffle(){ //每次随机挑选第个元素出来k然后存放在数组“当前末尾”,
    //todo
}


/*** 检查当前解（棋盘）的邻居棋盘（交换 solution[i和]solution[j],和随机选择，共有ijn(n-1)种不同的可能）/2
 *** 保留冲突最小的棋盘，将它视为新的解（ best 策略）found, 返回构建新解时交换而来的邻居编号current
 *** 若返回表示没有找到更好的邻居，需要重启 -1, restart() 返回表示找到所求的解，,0
 *** 返回大于的值表示需要更新当前解 0后要继续遍历新解的邻居solution
 ***/
int traverseNeighbors(){
    int min_value = evaluate();      // 当前棋盘的所有邻居的最小评价函数值
    int min_col = -1,min_row = -1; // 上述对应的邻居

    //遍历邻居，评估每个邻居冲突皇后对的个数，找到最小冲突对数的邻居
    for(int i = 0; i < len; ++i)
        for(int j = i + 1; j < len; ++j){
            swap(i,j);  // 生成邻居棋盘
            if(evaluate() < min_value){
                min_value = evaluate();
                min_col = i; min_row = j;
            }
            swap(i,j);  // 还原棋盘
        }

    if (min_col == -1) {  //如果遍历了所有的邻居节点，也找不到更好的移动方式，那么就需要重新启动算法，初始棋局改变一下
        printf("找不到解，正在重新生成初始解和运行搜索算法中...\n");
        return -1;
    }
    swap(min_col,min_row);  // 找到更好的移动方式，那么就以这个邻居进行下一次爬山
    return min_value;
}


/*** 当遍历完所有邻居，也没有找到 y的棋盘布局，==0
 *** 那么随机重置，然后再用爬山法搜索 solution
 *** 随机重启：邻居是邻域内的“微小”的扰动，增加扰动范围，比如随机交换 times次=20
```

```
 ***/
void  restart (){
    int  times  = 20;   //  随机交换解的次数，扰动大小设置,times即可>1
    for(int  i  = 0;  i  < times;  ++i)
        swap(rand()%len,rand()%len);
    puts(" restart  () 被调用！ ");
}
```

版本二代码下载链接：https://rec.ustc.edu.cn/share/d0ae8030-916c-11ed-a8a3-d1e47e221c17，感谢罗浩铭同学提供代码。

```
//  n皇后问题的随机搜索算法，找到解即可—
//用一维数组来存放每一列皇后的行号
//#include  " link_list  .h"
#include  <stdio.h>
#include  <stdlib.h>
#include  <time.h>
#include  <string.h>

#pragma GCC optimize(3)

//#define  MODE_ONE_LINE 0          仅获得  //中的  target_linei
//#define  MODE_GET_LINE_CONFLICT 1 获得//line_conflict

typedef  struct   _target_line
{
    int  i;  //  为冲突数最大的行i    //弃用，原为第行冲突数最小的列对应的皇后所在行ji
    //  int  *  line_conflict  ;  第//行中每一列的冲突数，若列冲突数为，则行列的皇后移至行列后，除了可能与交换的皇后
        冲突外不会与其它皇后冲突in0jnin
    int  *ld_base;
    int  *rd_base;  //参见evaluate ()函数
}  target_line  ;
//采用堆分配存储的方式实现
int  *solution;  //每列一个皇后，  solution [col]=row表示第,列的皇后在行colrow
int  len;          //皇后数量，  len*的棋盘len

//函数声明
void  initSolution  ();
void  restart  ();
void  swap(int,  int);
int  traverseNeighbors ();
int  evaluate ();
 target_line     evaluate_get_target_line  ();  //获得最大冲突数的行

int  clip(int  a,  int  min,  int  max);  //将的取值限制在a[min,  max范围内]
void  randShuffle ();

int  main()
```

```
{

    clock_t  start_t , finish_t ; //程序运行起始与终止时间
     start_t  = clock();
    long seed = 19260817; //随机数种子，如果这个数不改变，那么每次算法运行的结果都是一样的，即使用rand()函数
    srand(seed);              //这一句注释掉，等于每次的随机数种子都不一样

    //  printf 请输入皇后个数：("");
    //  scanf("%d",&len);           //  键盘读入棋盘的大小
    len = 20000;                                    //  棋盘大小
    solution = (int *)malloc( sizeof (int) * len); //一行
    if (! solution )
        return 0; //分配空间失败

     initSolution (); //  初始化一个棋盘布局

    int flag ;
    clock_t temp_t; //用于间隔输出evaluate()的值
    while (( flag = traverseNeighbors () ) != 0)
        if (clock() > 100 + temp_t)
        {
            temp_t = clock();
             printf (" evaluation : %d\n", evaluate () );
        }
    if ( flag < 0)
         restart ();
     finish_t = clock();
    //  printf ("%d\n", count_sol );
    printf ("\nIt took %ldms to get solution .\n", finish_t - start_t );

    printf ("Press any key to get solution .\n");
    getchar ();
    printf ("冲突数为：%d\得到的解为：n", evaluate () );
    for (int i = 0; i < len; ++i)
         printf ("%d,", solution [i ]);
     finish_t = clock();
    //  printf ("%d\n", count_sol );
    printf ("\nIt took %ldms to complete.\n", finish_t - start_t );
    getchar ();
    free ( solution );
    return 0;
}

/*** 将棋盘的第列和第列交换 ij ***/
void swap(int i, int j)
{
    int temp;
    temp = solution [i ];
     solution [i ] = solution [j ];
```

```
    solution[j] = temp;
    // todo finished
}

/*** 初始化一个棋盘布局将邻居数组准备好 , ***/
void initSolution()
{
    // 随机给初始布局或给定某个初始布局
    for (int i = 0; i < len; ++i)
        solution[i] = i; //先赋值0 ~ len-1
    randShuffle();          // 大于时原来的解法有大问题len40000
                            /*for (int i = 0; i < 2*len; ++i) // 对棋盘进行2*次随机交换len
                                swap(rand() % len, rand() % len);*/
}

/*** 计算棋盘的评价函数（棋盘的价值 : value可相互攻击到的皇后对数）=
 *** 当皇后相互攻击不到时，目标函数的值 value==0
 ***   solution[i]==solution[j], (solution[i]-solution[j])/(i-j)或时（==1-1语言实现会有），两个皇后彼此能攻击到，
       Cbugvalue++
 *** 棋盘编码方式能保证不会出现  solution[i]==solution[j，故该条件可以不用判断]
 *** 思考：该函数的时间性能是否能提高？
 ***/
int evaluate()
{
    int i;
    int value = 0;
    int *ld_base = (int *)malloc(2 * len * sizeof(int)); //往左移的对角线领地，表示有多少个皇后左对角线与此格冲突
    int *rd_base = (int *)malloc(2 * len * sizeof(int)); //往右移的对角线领地，表示有多少个皇后右对角线与此格冲突
    for (i = 0; i < 2*len; i++)
    {
        ld_base[i] = 0;
        rd_base[i] = 0;
    }
    int *ld = ld_base;          //此指针将右移，模拟位运算左移，用来，同时函数结束时，由可以得到每一行
        的 basefreeld_baseld
    int *rd = rd_base + len; //此指针将左移，模拟位运算右移，用来，同时函数结束时，由可以得到每一行
        的 basefreerd_baserd
    //算法可以保证同一列的皇后个数为，所以不需要考虑这种情况位运算算法里的0(row)

    for (i = 0; i < len; i++)
    {
        value += ld[solution[i]];
        value += rd[solution[i]];
        ld[solution[i]]++; //增加与此格左对角线冲突的皇后数
        rd[solution[i]]++; //增加与此格右对角线冲突的皇后数
        ld++;              //指针右移，模拟位运算左移
        rd--;              //指针左移，模拟位运算右移
    }

    free(ld_base);
```

```
    free(rd_base);
    // todo finished
    return value;
}
 target_line     evaluate_get_target_line  () //获得最大冲突数的行
{
     target_line   result ;

    int i;

    int value = 0;
    int *ld_base = (int *)malloc(2 * len * sizeof(int)); //往左移的对角线领地，表示有多少个皇后左对角线与此格冲突
    int *rd_base = (int *)malloc(2 * len * sizeof(int)); //往右移的对角线领地，表示有多少个皇后右对角线与此格冲突

    int num;
    int max_num = 0;
    int max_i = 0;
    for (i = 0; i < 2 * len; i++)
    {
        ld_base[i] = 0;
        rd_base[i] = 0;
    }
    int *ld = ld_base;          //此指针将右移，模拟位运算左移，用来，同时函数结束时，由可以得到每一行
        的 basefreeld_baseld
    int *rd = rd_base + len; //此指针将左移，模拟位运算右移，用来，同时函数结束时，由可以得到每一行
        的 basefreerd_baserd
    //算法可以保证同一列的皇后个数为，所以不需要考虑这种情况位运算算法里的0(row)

    for (i = 0; i < len; i++)
    {
        num = ld[ solution [i ]] + rd[ solution [i ]];
        if (num > max_num)
        {
            max_i = i;
            max_num = num;
        }
        value += num;
        ld[ solution [i ]]++; //增加与此格左对角线冲突的皇后数
        rd[ solution [i ]]++; //增加与此格右对角线冲突的皇后数
        ld++;                 //指针右移，模拟位运算左移
        rd--;                 //指针左移，模拟位运算右移
    }
    /*int * line_conflict = (int *)malloc(len * sizeof(int)); 第//行中每一列的冲突数i
    memcpy(line_conflict, ld + max_i, len * sizeof(int));
    for (i = 0; i < len; i++)
    {
        line_conflict [i] += *(rd - max_i + i);
    }*/
    // free(ld_base);
    // free(rd_base);
```

```
    //  todo  finished

    result.i = max_i;
    result.ld_base = ld_base;
    result.rd_base = rd_base;

    return result;
}

/*** 随机置换 randShuffle() ，无放回的均匀随机采样（被采样集合在不断缩小，直至为，算法停止）0
 *** 对数组 neighbors 进行随机置换操作
 *** 用到的数据： size= 的长度neighbors
 ***  best found 策略用不着该函数
 ***/
void randShuffle()
{
    //洗牌算法
    int i;
    for (i = len - 1; i > 0; i--)
    {
        swap(i, rand() % i); //每次随机挑选第个元素出来i然后存放在数组“当前末尾”,
    }
    // todo finished
}

/*** 检查当前解（棋盘）的邻居棋盘（交换 solution[i和]solution[j],和随机选择，共有ijn(n-1)种不同的可能）/2
 *** 保留冲突最小的棋盘，将它视为新的解（ best 策略）found
 *** 若返回表示没有找到更好的邻居，需要重启 -1, restart() 返回表示找到所求的解，,0
 *** 返回大于的值表示需要更新当前解 0后要继续遍历新解的邻居solution
 ***/
int traverseNeighbors()
{
    target_line target;
    int min_value = evaluate(); // 当前棋盘的所有邻居的最小评价函数值
    if (min_value == 0)
        return min_value;
    int min_col = -1, min_row = -1; // 上述对应的邻居
    int smaller_times = 0;              //最小值刷新的次数
    const int SMALLER_TIMES_MAX = 1; //最小值刷新的次数达到此值时可停止搜索

    //遍历邻居，此处不再遍历每个邻居最小冲突对数刷新次数达到一定上限即可更换,
    int i, j;
    int evaluation;
    /*
    int adaptive_thershold = 0; 规定//的起始搜索位置，若很小则为，否则为随机数imin_value0
    if (min_value < 20 || min_value < 0.01 * len)
        adaptive_thershold = 0;
    else
        adaptive_thershold = rand() % clip(rand() % min_value + 0.5 * min_value, 1, len); 玄学调参//
```

```
for (i = adaptive_thershold ; i < len; i++)
{*/
target = evaluate_get_target_line (); // 为最大冲突数的行i
i = target.i;
int *ld = target.ld_base;          //往左移的对角线领地，表示有多少个皇后左对角线与此格冲突
int *rd = target.rd_base + len;    //往右移的对角线领地，表示有多少个皇后右对角线与此格冲突
int conflict ;                     //某行某列的冲突数
if (min_value < (int)(0.001 * len) + 10)
{
    for (j = 0; j < len; j++)
    {
        if (j == i || j - i == solution[j] - solution[i] || i - j == solution[j] - solution[i]) //冲突
            continue;
        conflict = *(ld + i + solution[j]) + *(rd - i + solution[j]) + *(ld + j + solution[i]) + *(rd - j +
            solution[i]);
        if (conflict > 0)
            continue;
        swap(i, j); //暂时交换
        evaluation = evaluate();
        if (evaluation < min_value) //刷新最小值记录
        {
            min_value = evaluation;
            min_col = i;
            min_row = j;
            smaller_times++;
        }
        swap(i, j); //移回去
        if (min_value == 0 || smaller_times >= SMALLER_TIMES_MAX)
            break;
    }
    if (min_col == -1) //若快速算法未得到解，则老老实实使用普通算法
    {
        for (j = 0; j < len; j++)
        {
            if (j == i)
                continue;
            swap(i, j); //暂时交换
            evaluation = evaluate();
            if (evaluation < min_value) //刷新最小值记录
            {
                min_value = evaluation;
                min_col = i;
                min_row = j;
                smaller_times++;
            }
            swap(i, j); //移回去
            if (min_value == 0 || smaller_times >= SMALLER_TIMES_MAX)
                break;
```

```
            }
        }
    }
    else
    {
        int min_conflict = len; //所有列中最小的冲突数

        int min_j = (i + 1) % len;
        for (j = 0; j < len; j++)
        {
            if (j == i || j - i == solution[j] - solution[i] || i - j == solution[j] - solution[i]) //冲突
                continue;
            conflict = *(ld + i + solution[j]) + *(rd - i + solution[j]) + *(ld + j + solution[i]) + *(rd - j +
                solution[i]);
            if (conflict <= min_conflict) // rand() % 2 == 0)
            {

                min_conflict = conflict;
                min_j = j;

            }
        }
        swap(i, min_j);
        free(target.ld_base);
        free(target.rd_base);
        return min_value;
        //这里为了效率，的不准确，但保证不会误变成，同时为防止已找到解而未发现，前
            面 returnmin_value0evaluate()后面加了是否为的判断0
    }

    // todo finished
    free(target.ld_base);
    free(target.rd_base);
    if (min_col == -1)
    { //如果遍历了所有的邻居节点，也找不到更好的移动方式，那么就需要重新启动算法，初始棋局改变一下
        printf("找不到解，正在重新生成初始解和运行搜索算法中...\n");
        return -1;
    }
    swap(min_col, min_row); // 找到更好的移动方式，那么就以这个邻居进行下一次爬山
    return min_value;
}

/*** 当遍历完所有邻居，也没有找到更好的邻居
 *** 那么随机重置，然后再用爬山法搜索 solution
 *** 随机重启：邻居是邻域内的“微小”的扰动，增加扰动范围，比如随机交换 times次=20
 ***/
void restart()
{
    int times = len / 10 + 2; // 随机交换解的次数，扰动大小设置,times即可>1
```

```
    for (int i = 0; i < times; ++i)
        swap(rand() % len, rand() % len);
    puts(" restart ()被调用！ ");
}
int clip(int a, int min, int max) //若将的取值限制在a[min, max范围内]
{
    if (a > max)
        return max;
    if (a < min)
        return min;
    else
        return a;
}
```

版本三，感谢罗浩铭同学提供代码，代码下载和说明参考版本二的链接。

```
#include <vector>
#include <cstdio>
#include <cstdlib>
#include <time.h>
#define BUFFER_LEN 1000000
#define MAX_NUM_LEN 20
using namespace std;
int num_lenth = 1;  //全局变量，数字字符串的长度必须在每次初始化或者数字变长时正确更新,!
int buffer_pos = 0; //全局变量，请注意正确更新!
FILE *fp;
void output(char *buffer, char *num, int times); //输出构造法答案,代表进行timestimes次运算输出—1+2,个数times
void plus2(char *num);                           //对字符串进行运算+2

void clear_num(char *num);          //清空数字字符串
void clear_buffer (char *buffer); //清空 buffer
int get_num_lenth(int n);           //获得数字长度

void test_plus2 (void);

int main(void)
{
    //本程序为构造法
    clock_t start_t , finish_t ;
    fp = fopen("output . txt ", "w");
    char buffer [BUFFER_LEN + 1] = {0};
    char num[MAX_NUM_LEN] = {0}; //数字字符串后面带一个空格专门为输出服务,,

    long long n; // 皇后的nn
    int mid;

    printf ("Please enter the number n:");
    scanf("%lld", &n);
```

```
// n = 10000002;
printf("\n");
fprintf(fp, "%lld\n", n);

start_t = clock();

int i, j;
mid = n / 2;
if (n % 6 != 2 && n % 6 != 3)
{
    num[0] = '2';
    num[1] = ' ';
    num_lenth = 1;

    if (n % 2 == 0)
    {

        output(buffer, num, mid);
        clear_num(num);
        num[0] = '1';
        num[1] = ' ';
        num_lenth = 1;
        output(buffer, num, mid);
    }
    if (n % 2 == 1)
    {

        output(buffer, num, mid);
        clear_num(num);
        num[0] = '1';
        num[1] = ' ';
        num_lenth = 1;
        output(buffer, num, mid);
        fprintf(fp, "%lld", n);
    }
}
else
{
    if (n % 2 == 0)
    {
        if (mid % 2 == 0)
        {
            clear_num(num);
            sprintf(num, "%d ", mid);
            num_lenth = get_num_lenth(mid);
            output(buffer, num, (n - mid) / 2 + 1);

            clear_num(num);
            sprintf(num, "2 ");
```

```
            num_lenth = 1;
            output(buffer, num, (mid - 2) / 2);

            clear_num(num);
            sprintf(num, "%d ", mid + 3);
            num_lenth = get_num_lenth(mid + 3);
            output(buffer, num, (n - mid - 4) / 2 + 1);

            clear_num(num);
            sprintf(num, "1 ");
            num_lenth = 1;
            output(buffer, num, mid / 2 + 1);
        }
        else
        {
            clear_num(num);
            sprintf(num, "%d ", mid);
            num_lenth = get_num_lenth(mid);
            output(buffer, num, (n - 1 - mid) / 2 + 1);

            clear_num(num);
            sprintf(num, "1 ");
            num_lenth = 1;
            output(buffer, num, (mid - 3) / 2 + 1);

            clear_num(num);
            sprintf(num, "%d ", mid + 3);
            num_lenth = get_num_lenth(mid + 3);
            output(buffer, num, (n - mid - 3) / 2 + 1);

            clear_num(num);
            sprintf(num, "2 ");
            num_lenth = 1;
            output(buffer, num, (mid - 1) / 2 + 1);
        }
    }
    else
    {
        if (mid % 2 == 0)
        {
            clear_num(num);
            sprintf(num, "%d ", mid);
            num_lenth = get_num_lenth(mid);
            output(buffer, num, (n - 1 - mid) / 2 + 1);

            clear_num(num);
            sprintf(num, "2 ");
            num_lenth = 1;
            output(buffer, num, (mid - 4) / 2 + 1);
```

```
                clear_num(num);
                sprintf(num, "%d ", mid + 3);
                num_lenth = get_num_lenth(mid + 3);
                output(buffer, num, (n - mid - 5) / 2 + 1);

                clear_num(num);
                sprintf(num, "1 ");
                num_lenth = 1;
                output(buffer, num, (mid) / 2 + 1);
                fprintf(fp, "%lld", n);
            }
            else
            {
                clear_num(num);
                sprintf(num, "%d ", mid);
                num_lenth = get_num_lenth(mid);
                output(buffer, num, (n - 2 - mid) / 2 + 1);

                clear_num(num);
                sprintf(num, "1 ");
                num_lenth = 1;
                output(buffer, num, (mid - 3) / 2 + 1);

                clear_num(num);
                sprintf(num, "%d ", mid + 3);
                num_lenth = get_num_lenth(mid + 3);
                output(buffer, num, (n - mid - 4) / 2 + 1);

                clear_num(num);
                sprintf(num, "2 ");
                num_lenth = 1;
                output(buffer, num, (mid - 1) / 2 + 1);
                fprintf(fp, "%lld", n);
            }
        }
    }

    finish_t = clock();

    printf("It took %ldms to complete.\n", finish_t - start_t);
    fclose(fp);
    system("pause");
    return 0;
}
void output(char *buffer, char *num, int times) //输出构造法答案,代表进行timestimes次运算输出—1+2,个数times
{
    char *dest;
    char *src;
```

```
    int i;
    for (i = 0; i < times; i++)
    {

        if (buffer_pos + num_lenth + 1 > BUFFER_LEN)
        {
            dest = buffer + buffer_pos;
            src = num;
            while ((*dest++ = *src++) != '\0' && buffer_pos < BUFFER_LEN) // strcpy
            {
                buffer_pos++;
            }
            fputs(buffer, fp);
            clear_buffer(buffer);
            buffer_pos = 0;
            dest = buffer;
            while ((*dest++ = *src++) != '\0') // strcpy
            {
                buffer_pos++;
            }
        }
        else
        {
            dest = buffer + buffer_pos;
            src = num;
            while ((*dest++ = *src++) != '\0')
                ;
            buffer_pos += (num_lenth + 1);
        }
        plus2(num);
    }
    fputs(buffer, fp);
    clear_buffer(buffer);
    buffer_pos = 0;
}
void plus2(char *num) //对字符串进行运算+2
{
    int i, carry = 0; // 为进位carry
    if (num[num_lenth - 1] < '8')
    {
        num[num_lenth - 1] += 2;
    }
    else
    {
        num[num_lenth - 1] -= 8;
        carry = 1;
        for (i = num_lenth - 2; i >= 0; i--)
        {
            if (num[i] < '9')
```

```
            {
                num[i]++;
                carry = 0;
                break;
            }
            else
            {
                num[i] -= 9;
                carry = 1;
            }
        }
    }
    if (carry == 1) // 要增数字长度了注意数字后面有个空格也要移动,
    {
        num_lenth++;
        for (i = num_lenth + 1; i > 0; i--)
        {
            num[i] = num[i - 1];
        }
        num[0] = '1';
    }
}
void clear_num(char *num) //清空数字字符串
{
    int i;
    for (i = 0; i < MAX_NUM_LEN; i++)
    {
        num[i] = 0;
    }
}
void clear_buffer(char *buffer) // 清空 buffer
{
    int i;
    for (i = 0; i < BUFFER_LEN + 1; i++)
    {
        buffer[i] = '\0';
    }
}
int get_num_lenth(int n) // 获得数字长度
{
    int count = 0;
    if (n == 0)
    {
        return 1;
    }
    while (n != 0)
    {
        n = n / 10;
        count++;
```

```
    }

    return count;
}
void test_plus2 (void)
{
    char num[MAX_NUM_LEN] = {0}; //数字字符串
    num[0] = '1';
    num[1] = '4';
    num_lenth = 2;
    int i;
    for (i = 0; i < 214748364; i++)
    {
        plus2(num);
    }
    puts(num);
}
```

11.4 实验四参考代码

11.4.1 串的实现

代码下载链接：https://rec.ustc.edu.cn/share/e292b340-916d-11ed-b070-c90c81ef9c0d，提供代码的同学未具名(需要认领的同学联系我)。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MODE0
// #define MODE1
// #define MODE2

#ifdef MODE0
//定长顺序结构存储

#define MAXSIZE 200

typedef struct stringType{
    int length;
    char stringValue[MAXSIZE];
}* StringType;

StringType initString () {
    StringType str = malloc(sizeof(struct stringType));
    str->length = 0;
```

```
    return str;
}
//初始化串

void insert (StringType str, int k, char ch) {
    if (str->length == MAXSIZE) exit(-1);
    for (int i = str->length; i > k; i--)
        str->stringValue[i] = str->stringValue[i - 1];
    str->stringValue[k] = ch;
    str->length++;
}
//在原串的第个字符前插入字符，kchk,,……=01

char StringAt (StringType str,int k) {
    return str->stringValue[k];
}
//返回串的第个字符，kk,,……=01

int length(StringType str) {
    return str->length;
}
//返回串的长度

void StringClear (StringType str) {
    str->length = 0;
}
//清空字符串

void StringDestroy (StringType str) {
    free(str);
}
//销毁字符串

#endif

#ifdef MODE1
//堆分配存储

#define INITSIZE 20
#define INCREASEMENT 5

typedef struct stringType{
    int length;
    int space;
    char *stringValue;
}* StringType;

StringType initString () {
    StringType temp = malloc(sizeof(struct stringType));
```

```
    temp->length = 0;
    temp->space = INITSIZE;
    temp->stringValue = malloc(INITSIZE*sizeof(char));
    return temp;
}
//初始化串

void insert (StringType str, int k, char ch) {
    if (str->length == str->space) {
        str->stringValue = realloc (str->stringValue, str->space + INCREASEMENT);
        str->space += INCREASEMENT;
    }
    for (int i = str->length; i > k; i--) {
        str->stringValue[i] = str->stringValue[i - 1];
    }
    str->stringValue[k] = ch;
    str->length++;
}
//在原串的第个字符前插入字符，kchk,,……=01

char StringAt (StringType str,int n) {
    return str->stringValue[n];
}
//返回串的第个字符，kk,,……=01

int length(StringType str) {
    return str->length;
}
//返回串的长度

void StringClear (StringType str) {
    free(str->stringValue);
    str->space = INITSIZE;
    str->length = 0;
    str->stringValue = malloc(INITSIZE*sizeof(char));
}
//清空字符串

void StringDestroy (StringType str) {
    free(str->stringValue);
    free(str);
}
//销毁字符串

#endif

#ifdef MODE2

#define NODESIZE 4
```

```
typedef struct node {
    char ch[NODESIZE];
    struct node *next;
}Node;

typedef struct {
    int length;
    Node *stringValue;
}* StringType;

StringType initString () {
    StringType temp = malloc( sizeof (StringType));
    temp->length = 0;
    temp->stringValue = malloc(sizeof(Node));
    temp->stringValue->next = NULL;
    return temp;
}
//初始化串

void insert (StringType str, int k, char ch) {
    int nodeN = k / NODESIZE, offset = k % NODESIZE;
    Node *p = str->stringValue;
    for (int i = 0; i < nodeN; i++)
        p = p->next;
    int is_overflow = 0;
    char overflow;
    if (k == str->length) {
        overflow = ch;
        is_overflow = 1;
    }
    while (p != NULL) {
        if ( str->length >= (nodeN + 1) * NODESIZE) {
            overflow = p->ch[NODESIZE - 1];
            is_overflow = 1;
        } else {
            is_overflow = 0;
        }
        for (int i = NODESIZE - 1; i > offset; i--) {
            p->ch[i] = p->ch[i - 1];
        }
        if (nodeN * NODESIZE + offset == k) p->ch[offset] = ch;
        else p->ch[offset] = overflow;
        p = p->next;
        nodeN += 1;
        offset = 0;
    }
    if ( is_overflow ) {
        for (p = str->stringValue; p->next != NULL; p = p->next) ;
```

```
            p->next = (Node *)malloc(sizeof(Node));
            p->next->ch[0] = overflow;
            p->next->next = NULL;
        }
        str->length++;
}
//在原串的第个字符前插入字符，kchk,,……=01

char StringAt (StringType str,int n) {
        Node *q = str->stringValue;
        while (n >= NODESIZE) {
            q = q->next;
            n -= NODESIZE;
        }
        return q->ch[n];
}
//返回串的第个字符，kk,,……=01

int length(StringType str) {
        return str->length;
}
//返回串的长度

void StringClear (StringType str) {
        Node *p,*q;
        p = q = str->stringValue;
        while (p->next != NULL) {
            q = p->next;
            free(p);
            p = q;
        }
        str->stringValue = malloc(sizeof(Node));
        str->stringValue->next = NULL;
        str->length = 0;
}
//清空字符串

void StringDestroy (StringType str) {
        StringClear(str);
        free(str->stringValue);
        free(str);
}
//销毁字符串

#endif

//以下操作仅需要使用上面封装好的函数，而不关注串的具体存储方式。

void StringPrint (StringType str) {
```

```
    for (int i = 0; i < length(str); i++)
        printf("%c", StringAt(str, i));
    printf("\n");
}
//打印串中所有元素

void StringConcat(StringType ans,StringType str) {
    for (int i = 0; i < str->length; i++) {
        insert(ans, length(ans), StringAt(str, i));
    }
}
//串的拼接，将串拼接在串的末尾。 strans

void Next(StringType pat, int *next) {
    memset(next, 0, length(pat) * sizeof(int));
    int M = length(pat);
    next[0] = -1;
    next[1] = 0;
    int k = 2,j = 2;
    while (j < M) {
        k = next[j - 1];
        while (k > 0 && StringAt(pat, j - 1) != StringAt(pat, k))
            k = next[k];
        if (k <= 0) {
            if (StringAt(pat, 0) == StringAt(pat, j - 1)) next[j] = 1;
            else next[j] = 0;
        } else next[j] = k + 1;
        j++;
    }
}
//算法中求解数组KMPnext

int KMP(StringType txt,StringType pat) {
    int * next = malloc(length(pat) * sizeof(int));
    Next(pat, next);
    int i = 0, j = 0;
    int M = length(pat);
    int N = length(txt);
    while (i<N && j<M) {
        if (j==-1 || StringAt(txt,i)==StringAt(pat,j)) {
            ++i;
            ++j;
        } else j = next[j];
    }
    free(next);
    if (j>=M) return i-M;
    else return -1;
}
//串的模式匹配，为主串，为模式串，返回值为模式串在主串中第一次出现的位置。 txtpat
```

```
/*
    StringPrint(txt);     //abaabcab
    StringPrint(pat);     //abc
    KMP(txt, pat);        //3
*/

StringType SubString(StringType str,int begin,int size){
    StringType res = initString();
    for(int i = 0; i < size; i++){
        insert(res, length(res), StringAt(str, begin + i));
    }
    return res;
}
//求子串，结果为主串中从第个字符开始后的个字符，beginsizebegin,,……=01
/*
    StringPrint(txt);         //abaabcab
    SubString(txt, 1, 2);     //ba
*/

void StringReplace(StringType *str,StringType T,StringType P){
    int pos = KMP(*str, T);
    StringType temp = initString();
    while(pos != -1){
        StringConcat(temp, SubString(*str, 0, pos));
        StringConcat(temp, P);
        StringType p = SubString(*str, pos + length(T), length(*str) - pos - length(T));
        StringDestroy(*str);
        *str = p;
        pos = KMP(*str, T);
    }
    StringConcat(temp, *str);
    StringDestroy(*str);
    *str = temp;
}
//字符串替换，将主串中每个与串相同的子串替换为串strTP

int main(){
    //测试样例
    StringType txt = initString();
    insert(txt, 0, 'b');
    insert(txt, 0, 'c');
    insert(txt, 0, 'b');
    insert(txt, 0, 'a');
    insert(txt, 0, 'a');
    insert(txt, 0, 'c');
    insert(txt, 0, 'b');
    insert(txt, 0, 'a');
    insert(txt, 0, 'a');
    insert(txt, 0, 'b');
```

```
    insert(txt, 0, 'a');
    StringPrint(txt);
    //abaabcaabcb

    StringType pat = initString();
    insert(pat, 0, 'c');
    insert(pat, 0, 'b');
    printf("%d\n", KMP(txt, pat));
    //4

    StringConcat(txt, pat);
    StringPrint(txt);

    StringClear(pat);
    insert(pat, 0, 'b');

    StringType str = initString();
    insert(str, 0, 'b');
    insert(str, 0, 'b');
    StringReplace(&txt, pat, str);
    StringPrint(txt);
    //abbaabbcaabbcbb
    StringDestroy(str);
    StringDestroy(txt);
    StringDestroy(pat);
    return 0;
}
```

11.5 实验五参考代码

11.5.1 树的基本操作

代码下载链接：https://rec.ustc.edu.cn/share/2ff51190-916e-11ed-9a1e-d3cd5d5ba539,感谢常文正同学提供代码。

```
#ifndef _BITREE_CPP_
#define _BITREE_CPP_
//二叉树相关的实现代码
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//定义存放数据的类型假设是可以存放多个数据的结构体类型,
typedef struct
{
    int id;
```

```
    int value;
} ElemType;

//定义二叉树的结点
typedef struct tnode
{
    ElemType data;
    struct tnode *lchild;
    struct tnode *rchild;
} bNode;

#include "showGT.h" //展示树和图的API

/*下面的代码是四种不同的构建二叉树的算法，已经输入信息的不同而不同**— ***/
//构造二叉树（二叉排序树），输入长度为的整数数组 lenvalue
//  数组第个元素为根的，后续依次和根比较，若比根小，就和根的左子树比较；否则和右子树比较； value0valuevalue
//依次类推，若比较时遇到左子树或右子树为空，则以该值构建新的树节点，并称为对应的左子树或右子树。 value
void cPrintNode(bNode *a);
bNode *buildBTreeByValue(int *value, int len)
{
    bNode *root = (bNode *)malloc(sizeof(bNode));
    if (!root)
        exit(0);
    int idx = 0;
    root->data.id = idx;
    root->data.value = value[idx];
    root->lchild = root->rchild = NULL;
    cPrintNode(root);

    for (idx = 1; idx < len; ++idx)
    {
        //生成新节点
        bNode *t = (bNode *)malloc(sizeof(bNode));
        if (!t)
            exit(0);
        t->data.id = idx;
        t->data.value = value[idx];
        t->lchild = t->rchild = NULL;
        cPrintNode(t);

        //查找新节点在二叉树的准确位置
        bNode *q; //  的双亲结点p
        for (bNode *p = root; p; (value[idx] < p->data.value) ? (p = p->lchild) : (p = p->rchild))
            q = p;
        (value[idx] < q->data.value) ? (q->lchild = t) : (q->rchild = t); //将新节点链入二叉树
        printf(", 其父节点地址 %lx", q);
    }
    return root;
}
```

```
//构造二叉树输入长度为,的数组和lenedgevalue下标即为结点编号,， id
//  edge[id表示编号为]的结点的双亲结点的编号id(0~len-1)，根结点的双亲编号为-1
//  数组记录了结点的数据域值valuevalue
//算法思想：为每个结点构造一个结点，将它们的地址存在一个指针数组中；然后依据将这些结点链接成二叉树edge
bNode *buildBTreeByEdge(int *value, int *edge, int len)
{
    //首先需要检查数组是否代表一棵二叉树：唯一的edge，-10~len每个树至多出现次-12,edge[i]!=i这里省略.
    bNode **allnodes = (bNode **)malloc(sizeof(bNode *) * len); // 个指针len
    if (! allnodes)
        exit (0);
    for (int i = 0; i < len; ++i)
    {
        allnodes[i] = (bNode *)malloc(sizeof(bNode));
        if (!( allnodes[i]))
            exit (0);
        allnodes[i]->data.id = i;
        allnodes[i]->data.value = value[i];
        allnodes[i]->lchild = allnodes[i]->rchild = NULL;
    }

    bNode *root;
    for (int i = 0; i < len; ++i)
    { //二叉树不唯一，左右子树可以互换，优先构造左子树
        if (edge[i] != -1)
            (allnodes[edge[i]]->lchild == NULL) ? (allnodes[edge[i]]->lchild = allnodes[i]) : (allnodes[edge[i]]->
                rchild = allnodes[i]);
        else
            root = allnodes[i];
    }
    return root;
}

//递归算法：构造二叉树输入长度为,len=2n的数组+1，二叉树的节点数量为prelistn
//先序序列 prelist [i若序列中遇到空结点，其值为],，中有NULLNODEprelistn个+1NULLNODE
#define NULLNODE -9999
bNode *buildBTreeByPrelist(int * prelist , int len)
{
    bNode *t;
    static int idx = 0; //递归中每次调用自身都需要访问的“特殊”变量用静态变量或全局变量
    if ( prelist [idx] == NULLNODE)
    {
        idx++;
        return NULL;
    }
    if (!( t = (bNode *)malloc(sizeof(bNode))))
        exit (0);
    t->data.value = prelist [idx];
    t->data.id = idx;
```

```
    idx++; //准备访问 prelist [idx的下一个数据构建结点 ],无法放入形参列表idx
    t->lchild = buildBTreeByPrelist ( prelist , len);
    t->rchild = buildBTreeByPrelist ( prelist , len);
    return t;
} //如果中的数据无法全部放入到一棵二叉树，如何检测？(构造结束后 prelist  idx<len-1)
//学习形参列表中无法表示子问题的“特征时”时，采用全局变量或静态变量来处理

//递归算法先序遍历():构造二叉树 ，输入先序序列和中序序列，以及两个序列各自的上界preinpu,和下界iupl, il
bNode *buildBTreeByPI(int *pre, int pl, int pu, int *in, int il, int iu)
{
    printf ("\n%d-%d,%d-%d", pl, pu, il, iu);
    int il1, iu1, il2, iu2, pl1, pl2, pu1, pu2; //根节点把中序序列分成两个子序列，同样得到先徐序列的两个子序列，
        这是子序列的下标
    if (pu - pl != iu - il)
        exit (0); //检查输入参数
    if (pu - pl < 0)
        return NULL; //若序列为空，返回空结点空树/
    bNode *t = (bNode *)malloc(sizeof(bNode));
    if (!t)
        exit (0);
    t->data.value = pre[pl];
    t->data.id = pl;
    for (int i = il; i <= iu; ++i)
    {
        if (in[i] == pre[pl])
        {
            iu1 = i - 1;
            il1 = il;
            il2 = i + 1;
            iu2 = iu; //完成中序序列的切割in
            pl1 = pl + 1;
            pu1 = pl + (iu1 - il1 + 1);
            pl2 = pu1 + 1;
            pu2 = pu; //完成先徐序列的切割pre
        }
    }
    t->lchild = buildBTreeByPI(pre, pl1, pu1, in, il1, iu1);
    t->rchild = buildBTreeByPI(pre, pl2, pu2, in, il2, iu2);
    return t;
}

/*** 二叉树的其它基本操作  ***/
//查找给定结点的父节点p ，非空时，表示已经找到，就不需要递归调用，直接返回t
bNode *parent(bNode *root, bNode *p)
{
    bNode *t = NULL;
    if (!root || root == p)
        return NULL; //根无父节点，空树也没有的父节点p
```

```
    if (p && (root->lchild == p || root->rchild == p))
        return root;
    if (root->lchild && !t)
        t = parent(root->lchild, p);
    if (root->rchild && !t)
        t = parent(root->rchild, p);
    return t;
}

//由结点的（可以改成其它数据域的值）查找结点，返回结点指针id 。先序遍历）(
bNode *locateByID(bNode *root, int id)
{
    if (!root)
        return NULL;
    if (root->data.id == id)
        return root;
    bNode *t = locateByID(root->lchild, id);
    if (t)
        return t;
    return locateByID(root->rchild, id);
}

//采用后序遍历销毁树。思考：可以用先序遍历吗？
void DestroyBTree(bNode *root)
{
    if (root->lchild)
    {
        DestroyBTree(root->lchild);
        root->lchild = NULL;
    }
    if (root->rchild)
    {
        DestroyBTree(root->rchild);
        root->rchild = NULL;
    }
    if (!root->lchild && !root->rchild)
        free(root);
}

/*下面的代码是结点访问函数**- visit() 的不同实现方式，实现对单个结点的处理，函数名通常作为二叉树遍历函数的参数 ***/
//打印某个结点的信息依据,的定义而修改Elemtype简单版本（不输出存储地址）,
void bPrintNode(bNode *a)
{
    printf("\n(id = %2d) = %5d", a->data.id, a->data.value);
}

//打印某个结点的信息依据,的定义而修改Elemtype详细版本（输出存储地址）,
void cPrintNode(bNode *a)
```

```
{
    printf ("\n*(addr = %lx):( id=%2d)= %5d", a, a->data.id, a->data.value);
}

//测试函数值，结点数据域的值value+1
void add1(bNode *a)
{
    a->data.value += 1;
}

/*下面的代码是实现先序遍历、中序遍历和后序遍历的模板，大多数二叉树的操作基于对它们的修改**- ***/
//先序遍历二叉树
void preTraverseTree (bNode *root, void (*ptrf)(bNode *a))
{
    if (root)
    { //递归出口
        (*ptrf)(root);
        preTraverseTree (root->lchild, (*ptrf));
        preTraverseTree (root->rchild, (*ptrf));
    }
}

//中序遍历二叉树
void inTraverseTree (bNode *root, void (*ptrf)(bNode *a))
{
    if (root)
    { //递归出口
        inTraverseTree (root->lchild, (*ptrf));
        (*ptrf)(root);
        inTraverseTree (root->rchild, (*ptrf));
    }
}

//后序遍历二叉树
void postTraverseTree (bNode *root, void (*ptrf)(bNode *a))
{
    if (root)
    { //递归出口
        postTraverseTree (root->lchild, (*ptrf));
        postTraverseTree (root->rchild, (*ptrf));
        (*ptrf)(root);
    }
}

/*下面的代码用于线索化二叉树：层序遍历结果以链表链式队列**-()形式输出 ***/
//以下构成一个链式队列，存放二叉树的结点的地址，用于层序遍历二叉树
struct qNode
{                           //队列结点
    bNode *ptr;             //队列的节点包括的数据域是二叉树结点的指针
```

```
    struct qNode *next; //队列的下一个结点，队列是单链表
};

typedef struct queue
{
    struct qNode *front;
    struct qNode *rear;
} linkQueue;

//从二叉树生成层序遍历的队列：分层线索化。输入二叉树，输出队列
//将该队列当成单链表遍历一次，可以实现层序遍历
//该算法也可以稍作修改即得到“层序遍历”算法（尝试一下）
linkQueue *ToHiraQueue(bNode *root)
{
    linkQueue *hQueue = (linkQueue *)malloc(sizeof(linkQueue));
    if (!hQueue)
        exit(0);
    if (!root)
    {
        hQueue->front = NULL;
        hQueue->rear = NULL;
        return hQueue;
    }

    struct qNode *head;                                    //线索化过程中的队头不能修改(hQueue->，防止丢失front)，队尾直接用hQueue->rear
    struct qNode *t = (struct qNode *)malloc(sizeof(struct qNode)); //创建队列哨兵结点（头结点，不存放有效数据的结点）
    if (!t)
        exit(0);
    hQueue->front = hQueue->rear = head = t; //此时是空队列

    // 入队root
    t = (struct qNode *)malloc(sizeof(struct qNode)); //创建队列结点
    if (!t)
        exit(0);
    t->ptr = root; //数据域赋值，指针可以不用赋值next(why?)
    hQueue->rear->next = t;
    hQueue->rear = t;

    while (head != hQueue->rear)
    {                          //队列不空 ,是哨兵结点，数据无效，有效数据从headhead->开始next
        head = head->next; //出队（模拟出队，因为要返回线索化的队列，所以不能修改队列的头指针)
        // printf ("de-%lx,",head->ptr); 打印语句，调试用，查看入队、出队过程//
        if (head->ptr->lchild)
        { // head->存放数据（结点）的左孩子非空，构建新队列结点，入队next
            t = (struct qNode *)malloc(sizeof(struct qNode));
            if (!t)
                exit(0);
            t->ptr = head->ptr->lchild; //新结点的数据域赋值
```

```
            hQueue->rear->next = t;     // 新节点接到队列尾部
            hQueue->rear = t;
            //  printf ("en-%lx,",t->ptr); 打印语句，调试用，查看入队、出队过程//
        }
        if (head->ptr->rchild)
        { // 存放数据（结点）的右孩子非空，构建新队列结点，入队head
            t = (struct qNode *)malloc(sizeof(struct qNode));
            if (!t)
                exit (0);
            t->ptr = head->ptr->rchild; //新结点的数据域赋值
            hQueue->rear->next = t;     // 新节点接到队列尾部
            hQueue->rear = t;
            //  printf ("en-%lx,",t->ptr)打印语句，调试用，查看入队、出队过程;//
        }
    }
    return hQueue;
} // 思考如何实现先序中序后序遍历的线索化？并输出线索化后的队列//

/*** 下面是一些典型应用 ***/
// 删除节点value=的节点及其子树x
void DeleteNode(bNode *root, int x)
{
    if (!root) // 空树
        return;
    if (root->lchild->data.value == x)
    {
        DestroyBTree(root->lchild);
        root->lchild = NULL;
        return;
    }
    if (root->rchild->data.value == x)
    {
        DestroyBTree(root->rchild);
        root->rchild = NULL;
        return;
    }
    DeleteNode(root->lchild, x);
    DeleteNode(root->rchild, x);
}

//给定id1,，找到它们的最近公共祖先的id2id
int FindLCA(bNode *root, int id1, int id2)
{
    if (!root)  // 空树
        return -1;
    if (root->data.id == id1 || root->data.id == id2) // 找到或id1id2
        return root->data.id;
    int left = FindLCA(root->lchild, id1, id2);
    int right = FindLCA(root->rchild, id1, id2);
```

```
    if (left != -1 && right != -1) //左右子树都找到了或id1id2
        return root->data.id;
    else if (left != -1) //  left !=-1 && right==-1
        return left;
    else if (right != -1) //  left ==-1 && right!=-1
        return right;
    else
        return -1;
}

//给定，输出根节点到这个节点的路径ID初始,设为level0 ,存左右孩子指针标记，path表示左孩子，表示右孩子-11
void FindPath(bNode *root, int id, int *path, int level)
{
    if (!root) //空树
        return;
    int path1[100];
    memcpy(path1, path, sizeof(path1));
    if (root->data.id == id) //设置出口条件
    {
        printf("\npath:");
        for (int i = 0; i < level; i++)
            {
                if (path[i] == -1)
                    printf("L->");
                else
                    printf("R->");
            }
        printf("%d", root->data.id);
        return;
    }
    path1[level] = -1; //左孩子
    FindPath(root->lchild, id, path1, level + 1);
    memcpy(path, path1, sizeof(path1));
    path1[level] = 1; //右孩子
    FindPath(root->rchild, id, path1, level + 1);
}

//递归实现输出二叉树中最大和最小之差valuevalue
int ValueMax(bNode *root)
{
    static int max = 0;
    static int min = 100;
    if (!root) //空树
        return 0;
    if (root->data.value > max)
        max = root->data.value;
    if (root->data.value < min)
        min = root->data.value;
    ValueMax(root->lchild);
```

```
    ValueMax(root->rchild);
    return max - min;
}

//求二叉树的高度深度/
int deepth(bNode *root)
{
    if (!root)
        return 0;
    int d = deepth(root->lchild);
    int n = deepth(root->rchild);
    return (d > n) ? d + 1 : n + 1;
}

//求二叉树度为的结点数目2
int node2(bNode *root)
{
    int c = 0;
    if (!root)
        return 0;
    if (root->lchild && root->rchild)
        c = 1;
    return c + node2(root->lchild) + node2(root->rchild);
}

//求二叉树度为的结点数目1
int node1(bNode *root)
{
    int c = 0;
    if (!root)
        return 0;
    if ((!root->lchild && root->rchild) || (root->lchild && !root->rchild))
        c = 1;
    return c + node1(root->lchild) + node1(root->rchild);
}

//求二叉树度为的结点数目0 叶子结点数目）(
int node0(bNode *root)
{
    if (!root)
        return 0;
    if (!root->lchild && !root->rchild)
        return 1;
    return node0(root->lchild) + node0(root->rchild);
}

//求二叉树度为的结点数目的0之和value
int value0(bNode *root)
{
```

```
    if (!root)
        return 0;
    if (!root->lchild && !root->rchild)
        return root->data.value;
    return value0(root->lchild) + value0(root->rchild);
}

int main()
{
    int val[] = {22, 32, 6, 12, 75, 9, 88, 13, 41, 7, 16, 17};
    int n = 12;

    printf("\测试二叉树生成n value:");
    bNode *t1 = buildBTreeByValue(val, n); //测试由值数组生成二叉树
    printf("\先序遍历n");
    preTraverseTree(t1, add1); //测试先序遍历
    // printf("\中序遍历n");
    // inTraverseTree(t1, cPrintNode); 测试中序遍历//
    // printf("\后序遍历n");
    // postTraverseTree(t1, cPrintNode); 测试后续遍历//
    saveTree(t1, "sg1.html");

    int path[100];
    memset(path, 0, sizeof(path));
    FindPath(t1, 8, path, 0); //测试输出根节点到某个节点的路径

    int id1 = 7, id2 = 9;
    printf("\测试二叉树n%和d%的dLCA:%d", val[id1], val[id2], FindLCA(t1, id1, id2)); //测试求两个节点的最近公共祖先

    printf("\测试二叉树最大数值差n:%d", ValueMax(t1)); //测试二叉树最大数值差

    DeleteNode(t1, 7); //测试删除结点
    printf("\中序遍历n");
    inTraverseTree(t1, cPrintNode); //测试中序遍历
    saveTree(t1, "sg1-1.html");

    DestroyBTree(t1);

    // printf("\测试二叉树生成n edge :");
    // int edge[7] = {5, 6, 6, 1, 2, 2, -1}; 增加边数组//
    // n = 7;
    // t1 = buildBTreeByEdge(val, edge, n); 测试由值数组和边生成二叉树//
    // preTraverseTree(t1, bPrintNode); 测试先序遍历 //
    // inTraverseTree(t1, add1);          测试中序遍历 //
    // postTraverseTree(t1, cPrintNode);  测试后续遍历 //
    // // saveTree(t1, "sg2.html");
    // DestroyBTree(t1);
```

```
//  printf (”\测试二叉树生成n  Prelist :”) ;
//  int  prelist [15] = {17, 32, 41, –9999, –9999, –9999, 13, 7, –9999, –9999, 16, 22, –9999, –9999, –9999};
//  t1 = buildBTreeByPrelist ( prelist , 15);
//  preTraverseTree (t1 , cPrintNode); 测试先序遍历 //
//  inTraverseTree (t1 , cPrintNode);  测试中序遍历  //
//  postTraverseTree (t1 , bPrintNode); 测试后续遍历//
//  // saveTree(t1 , ”sg3.html”);

// 测试层序线索化//
//  printf (”\测试层序线索化n:”) ;
// linkQueue *qlist = ToHiraQueue(t1);
//  for ( struct qNode *p = qlist –>front; p != qlist –>rear; p = p–>next)
//   printf (”%d, ”, p–>next–>ptr–>data.value); 打印线索化链表//
//  printf (”\n”);

// 测试结点查找和求父节点//
//  printf (”\测试结点查找和求父节点n:”) ;
// bNode *tn = locateByID(t1 , 2);
//  if (!tn)
//   printf (”\没找到结点n”);
//  else
//  {
//   cPrintNode(tn);
//   printf (”\父节点为：n”);
//   tn = parent(t1 , tn);
//   if (!tn)
//       printf 没找到(””) ;
//   else
//       cPrintNode(tn);
// }
// DestroyBTree(t1);

// 测试先序序列和中序序列构造二叉树//
//  printf (”\测试二叉树生成n PI:”) ;
//  int pre[] = {17, 32, 41, 13, 7, 16, 22};
//  int in[] = {41, 32, 17, 7, 13, 22, 16};
//  t1 = buildBTreeByPI(pre, 0, n – 1, in, 0, n – 1);
//  if (t1)
//  {
//   preTraverseTree (t1 , cPrintNode); 测试先序遍历 //
//   inTraverseTree (t1 , cPrintNode); 测试中序遍历  //
//   postTraverseTree (t1 , cPrintNode); 测试后续遍历//
//  }

//  // saveTree(t1 , ”sg4.html”);

// 典型应用测试//
//  printf (”\树高度n %d ”, deepth(t1));
//  printf (”\树度为的结点数目n2 %d ”, node2(t1));
```

```
    //  printf ("\树度为的结点数目n0 %d ", node0(t1));
    //  printf ("\树度为的结点数目n1 %d ", node1(t1));
    //  printf ("\树度为的结点之和n0value %d ", value0(t1));

    // DestroyBTree(t1);
    return 0;
}

#endif
```

致谢

1. 本书部分内容由授课班级的助教梁永濠、丁家和、陈新宇、唐晨铨和刘阳协助整理、收集和编写。
2. 本书基于ElegantLaTeX 项目组提供的书籍模板完成撰写。ElegantLaTeX 项目组致力于打造一系列美观、优雅、简便的模板方便用户使用。目前由 ElegantNote，ElegantBook，ElegantPaper 组成，分别用于排版笔记，书籍和工作论文。
3. 本书内容用于教学，部分引用图表来自互联网，不一一指出原始出处，如有侵犯了原作者知识产权的情形，请告知并联系jlli@ustc.edu.cn.