華中科技大學

课 程 实 验 报 告

课程名称： 数据结构实验

专业班级 CS2206

学　　号 U202215522

姓　　名 崔顺杰

指导教师 袁凌 徐海涛

报告日期 2023 年 5 月 29 日

计算机科学与技术学院

目　　录

1 基于链式存储结构的线性表实现

1.1 系统设计

1.1.1 主要功能及框架

本系统是基于链式存储结构的线性表实现，提供了一系列对线性表进行操作的功能。下面列出了实现的**主要功能**：

1. 初始化线性表（InitList）：创建一个空的线性表，为头节点分配内存并将其 next 指针设置为 NULL。
2. 销毁线性表（DestroyList）：释放线性表中所有节点的内存空间，包括头节点。
3. 清空线性表（ClearList）：删除线性表中的所有元素，使其成为空表。
4. 判断线性表是否为空（ListEmpty）：判断线性表是否为空表，通过检查头节点的 next 指针是否为 NULL 来确定。
5. 获取线性表的长度（ListLength）：计算线性表中元素的数量，遍历链表并计数节点直到链表末尾。
6. 获取指定位置的元素（GetElem）：获取线性表中指定位置的元素值，遍历链表并计数节点直到达到指定位置。
7. 查找元素的位置（LocateElem）：在线性表中查找指定元素的位置，遍历链表并逐个比较节点的数据。
8. 获取元素的前驱（PriorElem）：获取线性表中指定元素的前驱元素值，遍历链表并逐个比较节点的数据。
9. 获取元素的后继（NextElem）：获取线性表中指定元素的后继元素值，遍历链表并逐个比较节点的数据。
10. 插入元素（ListInsert）：在指定位置插入元素，遍历链表并计数节点直到达到插入位置。
11. 删除元素（ListDelete）：删除指定位置的元素，遍历链表并计数节点直到达到删除位置。
12. 遍历线性表（ListTraverse）：依次输出线性表中的所有元素值，遍历链表并输出每个节点的数据。
13. 逆置线性表（reverseList）：将线性表中的元素顺序逆置，通过修改节点的

next 指针实现逆置。

14. 删除倒数第 n 个元素（RemoveNthFromEnd）：删除线性表中倒数第 n 个元素，计算链表长度并遍历至对应位置进行删除。
15. 对线性表进行排序（sortList）：将线性表中的元素按照升序进行排序，通过比较节点的数据值进行交换。

本程序分为三**个文件**，分别是：

1. define.h 头文件：定义了常量、数据类型和函数声明，供其他文件引用和共享。
2. define.cpp 源文件：包含了各个功能函数的定义，实现了对线性表的操作。
3. main.cpp 源文件：包含了程序的入口函数 main，实现了菜单交互和用户操作的逻辑。

程序的**框架**:

main 函数是程序的入口, 也是程序的主线, 在 main 函数中通过一个 while 循环保证程序不断地接受输入, 处理输入。同时通过一个 switch 判断来处理各种情况，索引到相应的函数，执行相应的操作。main 函数也承担着交互的人物，包括提示输入，回馈输出等。

1.1.2 有关常量、类型、函数的定义和声明

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
typedef int status;
typedef int ElemType;  // 数据元素类型定义
```

```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef int ElemType;
typedef struct LNode {  // 单链表（链式结构）结点的定义
  ElemType data;
  struct LNode *next;
} LNode, *LinkList;
// 函数声明
status reverseList(LinkList &L);
status InitList(LinkList &L);
status DestroyList(LinkList &L);
status ClearList(LinkList &L);
status ListEmpty(LinkList L);
int ListLength(LinkList L);
status GetElem(LinkList L, int i, ElemType &e);
status LocateElem(LinkList L, ElemType e);
status PriorElem(LinkList L, ElemType e, ElemType &pre);
status NextElem(LinkList L, ElemType e, ElemType &next);
status ListInsert(LinkList &L, int i, ElemType e);
status ListDelete(LinkList &L, int i, ElemType &e);
status ListTraverse(LinkList L);
status SaveList(LinkList L, char FileName[]);
status LoadList(LinkList &L, char FileName[]);
status RemoveNthFromEnd(LinkList L,int n);
status sortList(LinkList L);
```

1.2 功能实现

本程序基于链式存储结构实现了以下功能：

1.2.1 初始化线性表（InitList）

```
status InitList(LinkList &L) {
  if (L == NULL) {  // 不存在
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    return OK;
  } else {  // 存在
    return INFEASIBLE;
  }
}
```

这段代码是一个初始化链表的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L。

2. 接下来，代码检查链表是否为空。如果链表为空，继续执行初始化操作；如果链表不为空，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表为空，代码分配内存空间，创建链表头节点 L，并将其 next 指针设置为 NULL。

4. 返回操作成功的标志码 OK。

5. 如果链表不为空，则返回错误代码 INFEASIBLE。

总结：该代码用于初始化链表，即创建一个空的链表头节点。如果链表已经存在（头节点非空），则无法执行初始化操作，返回错误代码 INFEASIBLE。如果链表为空，通过动态内存分配创建链表头节点，并将其 next 指针设置为 NULL，表示链表为空。最后返回操作成功的标志码 OK。

1.2.2 销毁线性表（DestroyList）

```
status DestroyList(LinkList &L) {
  if (L == NULL) {  //不存在
    return INFEASIBLE;
```

```
4     } else {
5       LinkList tmp;
6       while (L->next != NULL) {  // 递归地找下一个，并依次 free 掉
7         tmp = L->next;
8         free(L);
9         L = tmp;
10      }
11      free(L);  // free 掉最后一个
12      L = NULL;
13      return OK;
14    }
15  }
```

这段代码是一个销毁链表的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L。

2. 接下来，代码检查链表是否为空。如果链表为空，返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表不为空，进入 else 分支。

4. 声明一个临时指针 tmp 用于存储下一个节点。

5. 进入循环，循环条件是链表头节点的 next 指针不为空。在每次循环中，将链表头节点的 next 指针赋给 tmp，以便稍后释放节点。

6. 使用 free 函数释放链表头节点。

7. 将链表头节点指向 tmp，即将下一个节点作为新的链表头节点。

8. 循环结束后，链表中的所有节点都被释放掉，此时链表为空。

9. 最后使用 free 函数释放最后一个节点。

10. 将链表头节点指针 L 设置为 NULL，表示链表已被销毁。

11. 返回操作成功的标志码 OK。

12. 如果链表为空，返回错误代码 INFEASIBLE。

总结：该代码用于销毁链表，即释放链表中所有节点的内存空间。通过循环遍历链表，并使用 free 函数逐个释放节点。最后将链表头节点指针 L 设置为 NULL，表示链表已被销毁。如果链表为空，则返回错误代码 INFEASIBLE。

1.2.3 清空线性表（ClearList）

```
status ClearList(LinkList &L) {
  if (L) {  //存在
    LinkList tmp;
    LinkList temp;
    tmp = L->next;  // 保留头节点，从第二个开始
    L->next = NULL;
    if(tmp == NULL){
      return ERROR;
    }
    while (tmp->next) {  //依次销毁
      temp = tmp->next;
      free(tmp);
      tmp = temp;
    }
    free(tmp);
    tmp = NULL;
    return OK;
  } else {
    return INFEASIBLE;
  }
}
```

这段代码是一个清空链表的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L。

2. 接下来，代码检查链表是否存在。如果链表存在，继续执行清空操作；如果链表不存在，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表存在，声明两个临时指针 tmp 和 temp。

4. 将链表头节点的 next 指针赋给 tmp，即保留头节点，从第二个节点开始清空。

5. 将链表头节点的 next 指针设置为 NULL，表示链表中只剩下头节点。

6. 检查 tmp 是否为 NULL，如果为 NULL，表示链表中只有头节点，无需清空，返回错误代码 ERROR。

7. 进入循环，循环条件是 tmp 的 next 指针不为 NULL。在每次循环中，将 tmp 的 next 指针赋给 temp，以便稍后释放节点。

8. 使用 free 函数释放节点 tmp。

9. 将 temp 赋给 tmp，即将下一个节点作为新的 tmp。

10. 循环结束后，链表中除了头节点以外的所有节点都被释放掉。

11. 使用 free 函数释放最后一个节点 tmp。

12. 将 tmp 指针设置为 NULL，表示链表已经被清空。

13. 返回操作成功的标志码 OK。

14. 如果链表不存在，则返回错误代码 INFEASIBLE。

总结：该代码用于清空链表，即释放链表中除了头节点以外的所有节点的内存空间。通过循环遍历链表，并使用 free 函数逐个释放节点。最后将链表头节点指针 L 的 next 指针设置为 NULL，表示链表中只剩下头节点。如果链表中只有头节点或者链表为空，则返回错误代码 ERROR。如果链表不存在，则返回错误代码 INFEASIBLE。

1.2.4 判断线性表是否为空（ListEmpty）

```
status ListEmpty(LinkList L) {
  if (L) {
    if (L->next) {
      return FALSE;
    } else {
      return TRUE;
    }
  } else {
    return INFEASIBLE;
  }
}
```

该函数用于判断线性表是否为这段代码是一个判断链表是否为空的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L。

2. 接下来，代码检查链表是否存在。如果链表存在，继续执行判断操作；如果链表不存在，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表存在，进入 if 分支。

4. 再次检查链表头节点的 next 指针是否存在。如果存在，说明链表不为空，返回逻辑值 FALSE。

5. 如果链表头节点的 next 指针不存在，说明链表为空，返回逻辑值 TRUE。

6. 如果链表不存在，则返回错误代码 INFEASIBLE。

总结：该代码用于判断链表是否为空。首先检查链表是否存在，如果存在，则根据链表头节点的 next 指针判断链表是否为空。如果链表存在且不为空，则返回逻辑值 FALSE；如果链表存在且为空，则返回逻辑值 TRUE。如果链表不存在，则返回错误代码 INFEASIBLE。

1.2.5 求线性表的长度（ListLength）

```
int ListLength(LinkList L) {
  if (L) {
    int i = 0;
    LinkList tmp;
    tmp = L;
    while (tmp->next) {  //依次递归数有几个
      i++;
      tmp = tmp->next;
    }
    return i;
  } else {
    return INFEASIBLE;
  }
}
```

这段代码是一个计算链表长度的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L。

2. 接下来，代码检查链表是否存在。如果链表存在，继续执行计算长度的操作；如果链表不存在，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表存在，声明一个整型变量 i 并初始化为 0，用于计数链表长度。

4. 声明一个临时指针 tmp，并将其指向链表头节点 L，用于遍历链表。

5. 进入循环，循环条件是 tmp 的 next 指针不为 NULL。在每次循环中，计数器 i 加 1，表示遇到一个节点，然后将 tmp 指向下一个节点。

6. 循环结束后，遍历完整个链表，计数器 i 的值即为链表的长度。

7. 返回链表的长度 i。

8. 如果链表不存在，则返回错误代码 INFEASIBLE。

总结：该代码用于计算链表的长度。通过遍历链表并累加计数器 i 的值，即可得到链表的长度。如果链表存在，则返回链表的长度；如果链表不存在，则返回错误代码 INFEASIBLE。

1.2.6 获取元素（GetElem）

```
status GetElem(LinkList L, int i, ElemType &e) {
  if (L) {
    LinkList tmp = L;
    int num = 0;
    while (tmp->next) {  //递归遍历
      num++;
      tmp = tmp->next;
      if (i == num) {  //找到了
        e = tmp->data;
        return OK;
      }
    }
    return ERROR;
  } else {
    return INFEASIBLE;
  }
}
```

这段代码是一个获取链表指定位置元素的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L，要获取的位置 i 以及一个引用参数 e，用于存储获取到的元素值。

2. 接下来，代码检查链表是否存在。如果链表存在，继续执行获取操作；如果链表不存在，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表存在，声明一个临时指针 tmp，并将其指向链表头节点 L，用于遍历链表。

4. 声明一个整型变量 num 并初始化为 0，用于计数当前节点位置。

5. 进入循环，循环条件是 tmp 的 next 指针不为 NULL。在每次循环中，计数器 num 加 1，表示遇到一个节点，然后将 tmp 指向下一个节点。

6. 在循环中，检查当前节点位置 num 是否等于要获取的位置 i。如果相等，则说明找到了指定位置的节点，将节点的数据值赋给引用参数 e，并返回操作成功的标志码 OK。

7. 如果循环结束后仍未找到指定位置的节点，则返回错误代码 ERROR 表示获取操作失败。

8. 如果链表不存在，则返回错误代码 INFEASIBLE。

总结：该代码通过遍历链表并计数节点位置，找到指定位置 i 的节点，并将其数据值赋给引用参数 e。如果链表存在且成功找到指定位置的节点，则返回操作成功的标志码 OK；如果链表不存在，则返回错误代码 INFEASIBLE；如果无法找到指定位置的节点，则返回错误代码 ERROR。

1.2.7 查找元素位置（LocateElem）

```
status LocateElem(LinkList L, ElemType e) {
  if (L) {  // 存在
    LinkList tmp = L;
    int num = 0;
    while (tmp->next) {  // 递归遍历
      num++;
      tmp = tmp->next;
      if (tmp->data == e) {  //找到
        return num;
```

```
10        }
11      }
12      return ERROR;
13    } else {
14      return INFEASIBLE;
15    }
16  }
```

这段代码是一个定位元素在链表中位置的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L 和要定位的元素 e。

2. 接下来，代码检查链表是否存在。如果链表存在，继续执行定位操作；如果链表不存在，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表存在，声明一个临时指针 tmp，并将其指向链表头节点 L，用于遍历链表。

4. 声明一个整型变量 num 并初始化为 0，用于计数当前节点位置。

5. 进入循环，循环条件是 tmp 的 next 指针不为 NULL。在每次循环中，计数器 num 加 1，表示遇到一个节点，然后将 tmp 指向下一个节点。

6. 在循环中，检查当前节点的数据值是否等于要定位的元素 e。如果相等，则说明找到了元素 e，返回当前节点的位置 num。

7. 如果循环结束后仍未找到元素 e，则返回错误代码 ERROR 表示定位操作失败。

8. 如果链表不存在，则返回错误代码 INFEASIBLE。

总结：该代码通过遍历链表并计数节点位置，找到与要定位的元素 e 相等的节点，并返回其位置。如果链表存在且成功找到元素 e，则返回元素 e 的位置；如果链表不存在，则返回错误代码 INFEASIBLE；如果无法找到元素 e，则返回错误代码 ERROR。

1.2.8 插入元素（ListInsert）

```
status ListInsert(LinkList &L, int i, ElemType e) {
  if (L) {
    LinkList tmp = L;
    int num = 0;
```

```
    while (tmp->next) {
      num++;
      if (num == i) {  // 一般情况
        LinkList temp = (LinkList)malloc(sizeof(LNode));
        temp->next = tmp->next;  // 先连上后面
        temp->data = e;
        tmp->next = temp;  // 再连上前面
        return OK;
      }
      tmp = tmp->next;
    }
    if (num + 1 == i) {  // 特殊情况
      LinkList temp = (LinkList)malloc(sizeof(LNode));
      temp->next = tmp->next;
      temp->data = e;
      tmp->next = temp;
      return OK;
    }
    return ERROR;
  } else {
    return INFEASIBLE;
  }
}
```

详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L 和要插入的位置 i 以及要插入的元素 e。

2. 接下来，代码检查链表是否为空。如果链表不为空，继续执行插入操作；如果链表为空，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表不为空，代码创建一个临时指针 tmp 并将其指向链表头节点 L，同时初始化一个变量 num 为 0，用于计算当前节点位置。

4. 进入循环，遍历链表直到 tmp 指针指向最后一个节点（tmp->next 为

NULL)。在循环中，每次迭代 num 加 1，表示当前节点的位置。

5. 在循环中，代码检查当前节点的位置 num 是否等于要插入的位置 i。如果相等，则说明找到了要插入的位置，进入一般情况的处理。

6. 在一般情况下，代码创建一个临时指针 temp，并为其分配内存空间，以创建新的节点。将新节点的数据域设置为要插入的元素 e，将新节点的 next 指针指向当前节点 tmp 的下一个节点。然后将当前节点 tmp 的 next 指针指向新节点 temp，完成插入操作。

7. 如果在循环中没有找到插入位置（num+1 != i），说明要插入的位置超出了链表的长度，进入特殊情况的处理。

8. 在特殊情况下，代码执行与一般情况相同的插入操作，将新节点插入到链表末尾。

9. 如果在循环中没有找到插入位置并且也不是特殊情况，则返回错误代码 ERROR 表示插入操作失败。

总结：该代码通过遍历链表找到要插入的位置，然后在该位置前面插入一个新节点，将元素 e 存储在新节点中。对于一般情况，新节点将插入到当前节点和下一个节点之间；对于特殊情况，新节点将插入到链表末尾。如果链表为空，则返回错误代码 INFEASIBLE。

1.2.9 删除元素（ListDelete）

```
status ListDelete(LinkList &L, int i, ElemType &e) {
  if (L) {  // 存在
    LinkList tmp = L;
    int num = 0;
    while (tmp->next) {
      num++;
      if (num == i) {   //找到第 i 个
        LinkList temp = tmp->next->next;
        e = tmp->next->data;
        free(tmp->next);
        tmp->next = temp;
        return OK;
```

```
13        }
14        tmp = tmp->next;
15      }
16      return ERROR;
17    } else {
18      return INFEASIBLE;
19    }
20  }
```

该代码是一个在链表中删除指定位置元素的函数。下面对代码进行详细分析:

1. 首先，传入的参数是一个指向链表头节点的指针 L，要删除的位置 i，以及一个引用参数 e，用于存储被删除元素的值。

2. 接下来，代码检查链表是否为空。如果链表不为空，继续执行删除操作；如果链表为空，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表不为空，代码创建一个临时指针 tmp 并将其指向链表头节点 L，同时初始化一个变量 num 为 0，用于计算当前节点位置。

4. 进入循环，遍历链表直到 tmp 指针指向最后一个节点（tmp->next 为 NULL）。在循环中，每次迭代 num 加 1，表示当前节点的位置。

5. 在循环中，代码检查当前节点的位置 num 是否等于要删除的位置 i。如果相等，则说明找到了要删除的位置，进入删除操作。

6. 在删除操作中，代码通过临时指针 temp 指向要删除节点的下一个节点（tmp->next->next），将要删除节点的数据域值赋给变量 e，释放要删除节点的内存空间，然后将当前节点 tmp 的 next 指针指向 temp，完成删除操作。

7. 如果在循环中没有找到要删除的位置，则返回错误代码 ERROR 表示删除操作失败。

8. 如果链表为空，则返回错误代码 INFEASIBLE。

总结:该代码通过遍历链表找到要删除的位置,然后将该位置节点删除,并将被删除节点的值赋给引用参数 e。如果链表为空，则返回错误代码 INFEASIBLE。

1.2.10 打印元素（ListTraverse）

```
status ListTraverse(LinkList L) {
  if (L) {
    L = L->next;
    while (L && L->next) {  //依次遍历


      printf("%d ", L->data);
      L = L->next;
    }
    if (L != NULL) printf("%d", L->data);
    return OK;
  } else {
    return INFEASIBLE;
  }
}
```

该函数用于依次打印线性表中的元素。首先判断线性表是否存在，若不存在则返回错误状态码 INFEASIBLE。然后通过循环遍历链表中的每个结点，并打印结点的数据元素值。最后返回状态码 OK。

1.2.11 逆置链表（reverseList）

```
status reverseList(LinkList &L){
  if(L){
    LinkList tmp = L->next;
    L->next = NULL;
    while(tmp){
      LinkList temp = tmp->next;
      tmp->next = L->next;
      L->next = tmp;
      tmp = temp;
```

```
    }
    return OK;
  }else{
    return INFEASIBLE;
  }
}
```

该代码是一个反转链表的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L。

2. 接下来，代码检查链表是否为空。如果链表不为空，继续执行反转操作；如果链表为空，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表不为空，代码创建一个临时指针 tmp，并将其指向链表头节点的下一个节点（L->next）。这里假设链表头节点不参与反转操作，从头节点的下一个节点开始反转。

4. 将链表头节点的 next 指针设置为 NULL，表示反转后的最后一个节点的 next 指针应该为 NULL。

5. 进入循环，遍历链表。在每次循环中，将临时指针 tmp 的 next 指针指向反转链表的头部（L->next），然后将链表头节点的 next 指针指向 tmp，完成一次节点的反转。

6. 在节点反转完成后，更新临时指针 tmp 为原链表中的下一个节点（tmp->next）。

7. 重复步骤 5 和步骤 6，直到遍历完整个链表，实现链表的反转。

8. 返回操作成功的标志码 OK。

9. 如果链表为空，则返回错误代码 INFEASIBLE。

总结：该代码通过遍历链表并修改节点的指针关系，实现了对链表的反转操作。在遍历过程中，将每个节点的 next 指针指向上一个节点，最终完成整个链表的反转。如果链表为空，则返回错误代码 INFEASIBLE。

1.2.12 删除倒数第 n 个元素（RemoveNthFromEnd）

```
status RemoveNthFromEnd(LinkList L,int n){
  if(L){
    LinkList tmp = L->next;
```

```
    int num = 0;
    while(tmp){
      num++;
      tmp = tmp->next;
    }
    if(num < n){
      return ERROR;
    }else{
      tmp = L;
      for(int i = 0;i < num - n;i++){
        tmp = tmp->next;
      }
      LinkList temp = tmp->next->next;
      free(tmp->next);
      tmp->next = temp;
      return OK;
    }
  }else{
    return INFEASIBLE;
  }
}
```

该代码是一个从链表末尾删除倒数第 n 个节点的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L 以及要删除的倒数第 n 个节点的位置 n。

2. 接下来，代码检查链表是否为空。如果链表不为空，继续执行删除操作；如果链表为空，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表不为空，代码创建一个临时指针 tmp，并将其指向链表头节点的下一个节点（L->next）。这里假设链表头节点不参与删除操作，从头节点的下一个节点开始计数。

4. 进入循环，遍历链表并计算链表的长度。在每次循环中，num 加 1，表示

当前节点的位置。

5. 循环结束后，如果链表的长度 num 小于要删除的倒数第 n 个节点的位置 n，则返回错误代码 ERROR。说明链表的长度不足以删除倒数第 n 个节点。

6. 如果链表的长度 num 大于等于要删除的倒数第 n 个节点的位置 n，进入删除操作。

7. 将临时指针 tmp 重新指向链表头节点 L，从头开始遍历链表。

8. 进入循环，循环次数为链表长度 num 减去要删除的倒数第 n 个节点的位置 n。在每次循环中，临时指针 tmp 移动到待删除节点的前一个节点位置。

9. 循环结束后，临时指针 tmp 指向待删除节点的前一个节点。

10. 创建临时指针 temp，指向待删除节点的下一个节点（tmp->next->next）。

11. 释放待删除节点的内存空间。

12. 将临时指针 tmp 的 next 指针指向临时指针 temp，完成删除操作。

13. 返回操作成功的标志码 OK。

14. 如果链表为空，则返回错误代码 INFEASIBLE。

总结：该代码通过遍历链表计算链表的长度，然后根据链表长度和倒数第 n 个节点的位置，找到待删除节点的前一个节点，并完成删除操作。如果链表为空，则返回错误代码 INFEASIBLE。如果链表的长度不足以删除倒数第 n 个节点，则返回错误代码 ERROR。

1.2.13 元素排序（sortList）

```
status sortList(LinkList L){
  if(L){
    LinkList tmp = L->next;
    while(tmp){
      LinkList temp = tmp->next;
      while(temp){
        if(tmp->data > temp->data){
          ElemType e = tmp->data;
          tmp->data = temp->data;
          temp->data = e;
        }
```

```
12          temp = temp->next;
13        }
14        tmp = tmp->next;
15      }
16      return OK;
17    }else{
18      return INFEASIBLE;
19    }
20  }
```

该代码是一个对链表进行升序排序的函数。下面对代码进行详细分析：

1. 首先，传入的参数是一个指向链表头节点的指针 L。

2. 接下来，代码检查链表是否为空。如果链表不为空，继续执行排序操作；如果链表为空，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表不为空，代码创建一个临时指针 tmp，并将其指向链表头节点的下一个节点（L->next）。这里假设链表头节点不参与排序操作，从头节点的下一个节点开始排序。

4. 进入外层循环，遍历链表。在每次循环中，临时指针 tmp 指向当前需要比较的节点。

5. 进入内层循环，从当前节点 tmp 的下一个节点开始遍历链表。在每次循环中，临时指针 temp 指向待比较的节点。

6. 在内层循环中，比较当前节点 tmp 的数据值与待比较节点 temp 的数据值的大小。如果当前节点 tmp 的数据值大于待比较节点 temp 的数据值，则交换两个节点的数据值。

7. 内层循环结束后，继续下一个待比较节点，将临时指针 temp 指向待比较节点的下一个节点。

8. 外层循环结束后，继续下一个需要比较的节点，将临时指针 tmp 指向下一个节点。

9. 完成所有比较和交换操作后，链表中的节点按升序排列。

10. 返回操作成功的标志码 OK。

11. 如果链表为空，则返回错误代码 INFEASIBLE。

总结：该代码通过冒泡排序算法对链表中的节点进行升序排序。从头节点的

下一个节点开始，依次比较相邻节点的数据值，并进行交换，直到所有节点按升序排列。如果链表为空，则返回错误代码 INFEASIBLE。

1.2.14 保存到文件和从文件读取

```
// 如果线性表 L 存在，将线性表 L 的的元素写到 FileName 文件中，返回 OK，否则返回 INFE
status SaveList(LinkList L, char FileName[]) {
  if (L) {
    FILE *fp = fopen(FileName, "w");
    L = L->next;
    while (L && L->next) {  // 仍为依次遍历，只是写入文件
      fprintf(fp, "%d ", L->data);
      L = L->next;
    }
    if (L != NULL) fprintf(fp, "%d\n", L->data);
    fclose(fp);
    return OK;
  } else {
    return INFEASIBLE;
  }
}

// 如果线性表 L 不存在，将 FileName 文件中的数据读入到线性表 L 中，返回 OK，否则返回
status LoadList(LinkList &L, char FileName[]) {
  if (L == NULL) {
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    LinkList tmp = L;
    FILE *fp = fopen(FileName, "r");
    if(fp == NULL){
      return ERROR;
    }
```

```
28      int elem;
29      while (fscanf(fp, "%d", &elem) != EOF) {    // 依次读入即可
30        tmp->next = (LinkList)malloc(sizeof(LNode));
31        tmp = tmp->next;
32        tmp->next = NULL;
33        tmp->data = elem;
34      }
35      fclose(fp);
36      return OK;
37    } else {
38      return INFEASIBLE;
39    }
40  }
```

这两段代码分别是保存线性表到文件和从文件加载线性表的函数。下面对代码进行详细分析：

SaveList 函数：1. 首先，传入的参数是一个指向链表头节点的指针 L 以及一个文件名 FileName。

2. 接下来，代码检查链表是否存在。如果链表存在，继续执行保存操作；如果链表不存在，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表存在，代码打开一个文件指针 fp，并以写入模式（”w”）打开名为 FileName 的文件。

4. 将链表头节点的下一个节点赋值给链表头节点 L，跳过头节点。

5. 进入循环，遍历链表。在每次循环中，将当前节点 L 的数据值写入文件中，以空格分隔。

6. 循环结束后，检查链表最后一个节点 L 是否存在。如果存在，将其数据值写入文件中，并以换行符结束。

7. 关闭文件指针 fp。

8. 返回操作成功的标志码 OK。

9. 如果链表不存在，则返回错误代码 INFEASIBLE。

LoadList 函数：1. 首先，传入的参数是一个指向链表头节点的指针 L 以及一个文件名 FileName。

2. 接下来，代码检查链表是否为空。如果链表为空，继续执行加载操作；如果链表不为空，则返回错误代码 INFEASIBLE 表示无法执行操作。

3. 如果链表为空，代码分配内存空间，创建链表头节点 L，并将其 next 指针设置为 NULL。

4. 创建一个临时指针 tmp，并将其指向链表头节点 L，用于遍历链表。

5. 打开一个文件指针 fp，并以读取模式（”r”）打开名为 FileName 的文件。

6. 检查文件指针 fp 是否为空，如果为空，则返回错误代码 ERROR 表示无法打开文件。

7. 声明一个变量 elem，用于存储从文件中读取的数据值。

8. 进入循环，通过 fscanf 函数从文件中逐个读取数据值，直到文件结束（EOF）。在每次循环中，将读取到的数据值赋给变量 elem。

9. 在循环中，创建新的节点并为其分配内存空间，将新节点的数据值设置为变量 elem，将新节点插入到链表中的 tmp 节点后面。

10. 循环结束后，关闭文件指针 fp。

11. 返回操作成功的标志码 OK。

12. 如果链表不为空，则返回错误代码 INFEASIBLE。

总结：SaveList 函数将链表中的元素按照顺序保存到文件中，LoadList 函数从文件中读取数据，并将数据存储到链表中。如果链表为空，则表示操作无法执行，返回 INFEASIBLE。在 LoadList 函数中，如果无法打开文件，则返回错误代码 ERROR。

1.2.15 多表操作的实现

由于是先实现的单表操作，多表操作基于单表操作的基础上不便于大量修改，所以采用构建一个表头结点数组

```
LinkList arr[30] = {NULL};
```

在后续操作中，切换表只需要更改数组下标即可，保证了函数的统一性。

1.3 实验小结

通过本次实验，成功实现了一个基于链式存储结构的线性表系统。通过对系统中各个函数的分析和测试，我对线性表的创建、插入、删除、查找、排序等操作有了更深入的了解。同时，我也学习了链表的基本概念和实现方式，并掌握了链表的常用操作方法。通过实践，我进一步加深了对数据结构的理解和应用能力。

2 基于二叉链表的二叉树实现

2.1 系统设计

2.1.1 主要功能

本菜单实现了以下功能：

1. 创建二叉树：根据输入的节点定义数组，使用先根遍历的方式创建二叉树。

2. 销毁二叉树：释放二叉树的所有节点，并将根节点指针置为空。

3. 清空二叉树：将二叉树的所有节点数据清空，但保留根节点。

4. 判断二叉树是否为空：判断二叉树是否为空树。

5. 求二叉树深度：计算二叉树的深度（高度）。

6. 获取元素：根据给定的值，查找并返回对应的节点。

7. 更改结点值：根据给定的值，修改对应节点的数据。

8. 获取兄弟节点：根据给定的值，查找并返回对应节点的兄弟节点。

9. 插入节点：根据给定的值和插入位置（左子树或右子树），在相应位置插入新的节点。

10. 删除元素：根据给定的值，删除对应的节点。

11. 先序遍历：按照先序遍历的顺序遍历二叉树，并对每个节点进行访问操作。

12. 中序遍历：按照中序遍历的顺序遍历二叉树，并对每个节点进行访问操作。

13. 后序遍历：按照后序遍历的顺序遍历二叉树，并对每个节点进行访问操作。

14. 层序遍历：按照层序遍历的顺序遍历二叉树，并对每个节点进行访问操作。

15. 从根节点开始的最大路径：计算二叉树中从根节点开始的最大路径和。

16. 保存到文件：将二叉树的节点数据保存到文件中。

17. 从文件读取：从文件中读取节点数据，创建二叉树。

18. 切换二叉树：切换当前操作的二叉树。

19. 交换左右子树：交换二叉树的所有节点的左右子树。

20. 求最近公共祖先：根据给定的两个节点值，找到二叉树中它们的最近公共祖先节点。

通过菜单选择不同的操作来对二叉树进行创建、修改、查询、删除等操作。

2.1.2 设计模块

1. 数据结构模块：定义了表示二叉树节点的结构体，并定义了二叉树的基本操作函数。这些函数包括创建二叉树、销毁二叉树、判断二叉树是否为空、求二叉树深度等。这个模块的目的是提供对二叉树的基本操作和管理。

2. 遍历模块：定义了先序遍历、中序遍历、后序遍历和层序遍历的函数。这些函数实现了对二叉树进行不同方式的遍历，并对每个节点进行访问。遍历模块的目的是提供对二叉树节点的访问和处理。

3. 操作模块：定义了一系列对二叉树进行操作的函数，如获取元素、更改节点值、获取兄弟节点、插入节点、删除节点等。这些函数实现了对二叉树结构的修改和查询，以满足用户的需求。

4. 文件操作模块：定义了将二叉树节点数据保存到文件和从文件中读取数据创建二叉树的函数。这些函数实现了将二叉树持久化到文件的功能，以及从文件中还原二叉树的功能。

5. 主函数模块：主函数模块负责程序的入口和用户界面。它提供了一个菜单界面，通过用户输入选择不同的操作，然后调用相应的函数实现功能。主函数模块也负责切换当前操作的二叉树，以及错误处理和用户交互。

这个程序的设计框架通过模块化的方式将不同功能的代码分离开来，提高了代码的可读性和可维护性。它使用了数据结构模块管理二叉树的节点，遍历模块实现节点的访问，操作模块提供对二叉树的修改和查询，文件操作模块实现二叉树数据的持久化，而主函数模块则提供用户界面和整合各个模块的功能。

2.1.3 程序相关类型定义和函数声明

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define TRUE 1
```

```
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

typedef int status;
typedef int KeyType;
typedef struct {
  KeyType key;
  char others[20];
} TElemType;  // 二叉树结点类型定义

typedef struct BiTNode {  // 二叉链表结点的定义
  TElemType data;
  struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

// 函数声明
void visit(BiTree t);  // 用于遍历输出
void CreTree(BiTree &T, TElemType definition[], int tmp[]);  // 用于创建二叉树
status CreateBiTree(BiTree &T, TElemType definition[]);  // 用于创建二叉树
status ClearBiTree(BiTree &T);              // 用于销毁二叉树
int BiTreeDepth(BiTree T);                  // 用于求二叉树深度
BiTNode *LocateNode(BiTree T, KeyType e);  // 用于查找结点
status Assign(BiTree &T, KeyType e, TElemType value);  // 用于修改结点
BiTNode *GetSibling(BiTree T, KeyType e);  // 用于求结点的兄弟结点
status InsertNode(BiTree &T, KeyType e, int LR, TElemType c);  // 用于插入结点
BiTNode *Getfather(BiTree T, KeyType e);  // 用于求结点的父结点
status DeleteNode(BiTree &T, KeyType e);  // 用于删除结点
status PreOrderTraverse(BiTree T, void (*visit)(BiTree));  // 用于先序遍历
```

```
status InOrderTraverse(BiTree T, void (*visit)(BiTree));   // 用于中序遍历
status PostOrderTraverse(BiTree T, void (*visit)(BiTree)); // 用于后序遍历
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree)); // 用于层序遍历
status SaveBiTree(BiTree T, char FileName[]);   // 用于保存二叉树
status LoadBiTree(BiTree &T, char FileName[]); // 用于加载二叉树
int MaxPathSum(BiTree T);  // 用于求二叉树中的最大路径和
BiTree LowestCommonAncestor(BiTree T, int e1,
                            int e2);  // 用于求二叉树中两个结点的最近公共祖先
int InvertTree(BiTree T);  // 用于求二叉树的镜像
```

2.2 功能实现

2.2.1 创建二叉树

```
// 根据带空枝的二叉树先根遍历序列 definition 构造一棵二叉树，将根节点指针赋值给 T 并
void CreTree(BiTree &T, TElemType definition[], int tmp[]) {
  int i = 0;
  for (i = 0; i < 50; i++) {  // 找到第一个未被访问的结点
    if (tmp[i] == 0) {
      break;
    }
  }
  tmp[i] = 1;
  if (strcmp(definition[i].others, "null")) {  // 如果不是空结点
    T = (BiTree)malloc(sizeof(BiTNode));
    memcpy(&T->data, &definition[i], sizeof(TElemType));  // 复制结点数据
    CreTree(T->lchild, definition, tmp);  // 递归创建左子树
    CreTree(T->rchild, definition, tmp);  // 递归创建右子树
  } else {
    T = NULL;
  }
}
```

```
status CreateBiTree(BiTree &T, TElemType definition[]) {
  if (T) {
    return ERROR;
  }
  int tmp[50];
  int i = 0;
  while (definition[i].key != -1) {  // 检查是否有重复的 key
    if (i == 0) {
      tmp[0] = definition[i].key;
    } else {
      for (int j = 0; j < i; j++) {
        if (definition[i].key && tmp[j] == definition[i].key) {
          return ERROR;  // 如果有重复的 key, 返回 ERROR
        }
      }
      tmp[i] = definition[i].key;
    }
    i++;
  }
  int tmp2[50] = {0};
  CreTree(T, definition, tmp2);  // 创建二叉树
  return OK;
}
```

上述代码是一个用于构造二叉树的函数，它根据带空枝的二叉树先根遍历序列 ‘definition‘ 来创建一棵二叉树。下面是对代码的详细分析：

```
void CreTree(BiTree &T, TElemType definition[], int tmp[])
```

这是一个递归函数，用于创建二叉树。它的参数包括二叉树的根节点指针 ‘T‘、带空枝的二叉树先根遍历序列 ‘definition‘，以及一个用于标记节点是否已被访问的数组 ‘tmp‘。

```
int i = 0;
```

```
for (i = 0; i < 50; i++) {
  if (tmp[i] == 0) {
    break;
  }
}
tmp[i] = 1;
```

这部分代码用于找到第一个未被访问的节点。遍历 ‘tmp‘ 数组，找到值为 0 的位置 ‘i‘，并将其标记为已访问（赋值为 1）。

```
if (strcmp(definition[i].others, "null")) {
  T = (BiTree)malloc(sizeof(BiTNode));
  memcpy(&T->data, &definition[i], sizeof(TElemType));
  CreTree(T->lchild, definition, tmp);
  CreTree(T->rchild, definition, tmp);
} else {
  T = NULL;
}
```

在这段代码中，首先通过比较节点的 ‘others‘ 字段是否为”null”，来判断节点是否为空节点。如果不是空节点，就为当前节点 ‘T‘ 分配内存，并将节点数据从 ‘definition[i]‘ 复制到 ‘T->data‘ 中。然后，通过递归调用 ‘CreTree‘ 函数，分别创建左子树和右子树。

```
status CreateBiTree(BiTree &T, TElemType definition[])
```

这是一个用于创建二叉树的函数。它的参数包括二叉树的根节点指针 ‘T‘ 和带空枝的二叉树先根遍历序列 ‘definition‘。

```
if (T) {
  return ERROR;
}
```

这部分代码检查传入的根节点指针 ‘T‘ 是否已经指向了一个二叉树。如果 ‘T‘ 不为空（即已经指向了一个二叉树），则返回错误状态。

```
int tmp[50];
int i = 0;
while (definition[i].key != -1) {
  if (i == 0) {
    tmp[0] = definition[i].key;
  } else {
    for (int j = 0; j < i; j++) {
      if (definition[i].key && tmp[j] == definition[i].key) {
        return ERROR;
      }
    }
    tmp[i] = definition[i].key;
  }
  i++;
}
```

这部分代码用于检查 ‘definition‘ 中的节点是否有重复的 ‘key‘ 值。首先，定义一个临时数组 ‘tmp‘ 用于存储已经访问过的 ‘key‘ 值。然后，通过遍历 ‘definition‘ 数组，将 ‘key‘ 值存储到 ‘tmp‘ 数组中，并在每次存储前检查是否与之前的 ‘key‘ 值重复。如果发现重复的 ‘key‘ 值，就返回错误状态。

```
int tmp2[50] = {0};
CreTree(T, definition, tmp2);
```

这部分代码创建一个临时数组 ‘tmp2‘，并将其初始化为全 0。然后，调用 ‘CreTree‘ 函数来创建二叉树，传入根节点指针 ‘T‘、‘definition‘ 数组以及临时数组 ‘tmp2‘。

最后，函数返回操作成功的状态。

总体来说，这段代码通过先根遍历序列 ‘definition‘ 来构造一棵带空枝的二叉树，并且在创建过程中进行了重复 ‘key‘ 值的检查。它利用递归的方式遍历序列并创建二叉树的节点，并通过 ‘tmp‘ 数组进行节点的访问标记。这种设计可以有效地构建出正确的二叉树结构。

2.2.2 销毁二叉树

```
// 将二叉树设置成空，并删除所有结点，释放结点空间
status ClearBiTree(BiTree &T) {
  if (T) {
    ClearBiTree(T->lchild);  // 递归删除左子树
    ClearBiTree(T->rchild);  // 递归删除右子树
    free(T);
    T = NULL;
  } else {
  }
  return OK;
}
```

上述代码是一个用于清空二叉树并释放结点空间的函数。让我们逐行进行详细分析：

1. ‘status ClearBiTree(BiTree T)‘: 这是函数的定义，它接受一个指向二叉树根结点的指针作为参数，并返回一个状态值。

2. ‘if (T) ... else ... ‘: 这是一个条件语句，判断二叉树是否为空。如果树非空（即指针 T 指向有效结点），则执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

3. ‘ClearBiTree(T->lchild);‘: 这是一个递归调用，用于删除左子树。通过调用 ClearBiTree 函数，传递左子树根结点的指针，可以递归地删除整个左子树。

4. ‘ClearBiTree(T->rchild);‘: 这也是一个递归调用，用于删除右子树。通过调用 ClearBiTree 函数，传递右子树根结点的指针，可以递归地删除整个右子树。

5. ‘free(T);‘: 这行代码释放当前结点的内存空间。在删除左右子树后，通过调用 ‘free‘函数释放当前结点的内存。

6. ‘T = NULL;‘: 将指针 T 设置为 NULL，表示当前结点已被删除。

7. ‘return OK;‘: 返回一个状态值，表示函数执行成功。

通过递归调用，ClearBiTree 函数会先删除左子树，然后删除右子树，最后释放根结点的内存空间，从而清空整个二叉树。这种递归的删除方式可以确保所有结点都被正确释放，避免内存泄漏。

2.2.3 求树的深度

```
// 求二叉树 T 的深度
int BiTreeDepth(BiTree T) {
  if (T) {
    int left = BiTreeDepth(T->lchild);    // 递归求左子树
    int right = BiTreeDepth(T->rchild);  //递归求右子树
    if (left > right) {
      return left + 1;
    } else {
      return right + 1;
    }
  } else {
    return 0;
  }
}
```

上述代码是用于求解二叉树的深度的函数。让我们逐行进行详细分析：

1. ‘int BiTreeDepth(BiTree T)‘: 这是函数的定义，它接受一个指向二叉树根结点的指针作为参数，并返回一个整数值表示二叉树的深度。

2. ‘if (T) ... else ... ‘: 这是一个条件语句，判断二叉树是否为空。如果树非空（即指针 T 指向有效结点），则执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

3. ‘int left = BiTreeDepth(T->lchild);‘: 这是一个递归调用，用于计算左子树的深度。通过调用 BiTreeDepth 函数，传递左子树根结点的指针，可以递归地计算左子树的深度。

4. ‘int right = BiTreeDepth(T->rchild);‘: 这也是一个递归调用，用于计算右子树的深度。通过调用 BiTreeDepth 函数，传递右子树根结点的指针，可以递归地计算右子树的深度。

5. ‘if (left > right) ... else ... ‘: 这是一个条件语句，用于比较左子树的深度和右子树的深度。如果左子树的深度大于右子树的深度，返回 left + 1；否则，返回 right + 1。这样可以确保返回的深度值是左右子树中较大的深度加 1。

6. ‘return 0;‘: 当二叉树为空时，直接返回深度为 0。

通过递归调用，BiTreeDepth 函数会先递归计算左子树的深度，然后递归计算右子树的深度，最后返回左右子树中较大的深度加 1，即为整个二叉树的深度。这种递归的求深度方式可以准确地计算出二叉树的深度。

2.2.4 查找结点

```
// 查找结点
BiTNode *LocateNode(BiTree T, KeyType e) {
  if (T) {
    if (T->data.key == e) {
      return T;
    }
    BiTree tmp;
    if ((tmp = LocateNode(T->lchild, e))) {  //递归在左子树中查找
      return tmp;
    } else if ((tmp = LocateNode(T->rchild, e))) {  //递归在右子树中查找
      return tmp;
    }
  } else {
    return NULL;
  }
  return NULL;
}
```

上述代码是用于在二叉树中查找指定值的结点，并返回该结点的指针。让我们逐行进行详细分析：

1. ‘BiTNode *LocateNode(BiTree T, KeyType e)‘: 这是函数的定义，它接受一个指向二叉树根结点的指针和一个关键字值作为参数，并返回一个指向找到的结点的指针。

2. ‘if (T) ... else ... ‘: 这是一个条件语句，判断二叉树是否为空。如果树非空（即指针 T 指向有效结点），则执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

3. ‘if (T->data.key == e) ... ‘: 这是一个条件语句，用于判断当前结点的关键字值是否等于目标值 e。如果相等，表示找到了目标结点，直接返回当前结点的指针。

4. ‘BiTree tmp;‘: 声明一个临时变量 tmp，用于存储递归查找的结果。

5. ‘if ((tmp = LocateNode(T->lchild, e))) ... ‘: 这是一个递归调用，用于在左子树中查找目标结点。通过调用 LocateNode 函数，传递左子树根结点的指针和目标值 e，可以递归地在左子树中查找。

6. ‘else if ((tmp = LocateNode(T->rchild, e))) ... ‘: 这也是一个递归调用，用于在右子树中查找目标结点。通过调用 LocateNode 函数，传递右子树根结点的指针和目标值 e，可以递归地在右子树中查找。

7. ‘return NULL;‘: 当二叉树为空或者递归查找结束后没有找到目标结点时，返回空指针表示未找到。

通过递归调用，在二叉树中进行深度优先搜索，先检查当前结点是否为目标结点，如果是，则返回当前结点的指针；如果不是，则递归地在左子树和右子树中查找目标结点。如果最终没有找到目标结点，则返回空指针。这种递归的查找方式可以在二叉树中准确地找到指定值的结点。

2.2.5 结点赋值

```
// 实现结点赋值
status Assign(BiTree &T, KeyType e, TElemType value) {
  BiTree tmp;
  BiTree tmp2;
  tmp = LocateNode(T, e);
  tmp2 = LocateNode(T, value.key);
  if (tmp2 && tmp2 != tmp) {  //说明没找到
    return ERROR;
  }
  if (tmp) {  //找到，改写
    memcpy(&tmp->data, &value, sizeof(TElemType));
    return OK;
  } else {
```

```
14      return ERROR;
15    }
16  }
```

上述代码是用于实现给二叉树中的指定结点赋值的函数。让我们逐行进行详细分析：

1. ‘status Assign(BiTree T, KeyType e, TElemType value)‘: 这是函数的定义，它接受一个指向二叉树根结点的指针、一个关键字值 e 和一个待赋值的结点数据 value 作为参数，并返回一个状态值。

2. ‘BiTree tmp; BiTree tmp2;‘: 声明两个临时变量 tmp 和 tmp2，用于存储查找到的结点指针。

3. ‘tmp = LocateNode(T, e);‘: 调用 LocateNode 函数在二叉树中查找关键字值为 e 的结点，并将结果赋值给 tmp。

4. ‘tmp2 = LocateNode(T, value.key);‘: 调用 LocateNode 函数在二叉树中查找关键字值为 value.key 的结点，并将结果赋值给 tmp2。

5. ‘if (tmp2 tmp2 != tmp) ... ‘: 这是一个条件语句，用于判断是否找到了关键字值为 value.key 的结点。如果 tmp2 非空且与 tmp 指向的结点不同，表示没有找到与 value.key 对应的结点，返回错误状态。

6. ‘if (tmp) ... else ... ‘: 这是一个条件语句，用于判断是否找到了关键字值为 e 的结点。如果 tmp 非空，表示找到了目标结点，执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

7. ‘memcpy(tmp->data, value, sizeof(TElemType));‘: 将 value 的值复制到 tmp 指向的结点的 data 字段中。这里使用 memcpy 函数进行内存复制，确保数据的正确赋值。

8. ‘return OK;‘: 返回状态值表示赋值操作成功。

9. ‘return ERROR;‘: 返回状态值表示赋值操作失败。

通过调用 LocateNode 函数，在二叉树中查找关键字值为 e 的结点和 value.key 的结点。如果找到了 value.key 对应的结点，并且不同于 e 对应的结点，表示未找到正确的结点，返回错误状态。如果找到了 e 对应的结点，将 value 的值复制给该结点的 data 字段，并返回成功状态。如果未找到 e 对应的结点，则返回错误状态。这种实现方式可以根据关键字值找到对应的结点，并进行值的赋值操作。

2.2.6 获得兄弟结点

```
// 实现获得兄弟结点
BiTNode *GetSibling(BiTree T, KeyType e) {
  if (T) {
    if (T->data.key == e) {
      return NULL;
    }
    if (T->lchild && T->rchild) {  //两个孩子都存在，此处为单独判断根节点
      if (T->lchild->data.key == e) {
        return T->rchild;
      }
      if (T->rchild->data.key == e) {
        return T->lchild;
      }
    }
    BiTree tmp = GetSibling(T->lchild, e);  // 判断完成，递归寻找左子树和右子树
    if (tmp) {
      return tmp;
    } else {
      tmp = GetSibling(T->rchild, e);
      return tmp;
    }
  } else {
    return NULL;
  }
}
```

上述代码是用于在二叉树中获取指定结点的兄弟结点的函数。让我们逐行进行详细分析：

1. ‘BiTNode *GetSibling(BiTree T, KeyType e)‘: 这是函数的定义，它接受一个指向二叉树根结点的指针和一个关键字值 e 作为参数，并返回一个指向找到

的兄弟结点的指针。

2. ‘if (T) ... else ... ‘: 这是一个条件语句，判断二叉树是否为空。如果树非空（即指针 T 指向有效结点），则执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

3. ‘if (T->data.key == e) ... ‘: 这是一个条件语句，用于判断当前结点的关键字值是否等于目标值 e。如果相等，表示找到了目标结点，返回空指针表示没有兄弟结点。

4. ‘if (T->lchild T->rchild) ... ‘: 这是一个条件语句，判断当前结点是否同时具有左孩子和右孩子。如果是根节点，并且左右孩子都存在，才进入 if 语句块。

5. ‘if (T->lchild->data.key == e) ... ‘: 这是一个条件语句，判断当前结点的左孩子的关键字值是否等于目标值 e。如果相等，表示找到了目标结点的左孩子，返回右孩子的指针作为兄弟结点。

6. ‘if (T->rchild->data.key == e) ... ‘: 这是一个条件语句，判断当前结点的右孩子的关键字值是否等于目标值 e。如果相等，表示找到了目标结点的右孩子，返回左孩子的指针作为兄弟结点。

7. ‘BiTree tmp = GetSibling(T->lchild, e);‘: 这是一个递归调用，用于在左子树中查找目标结点的兄弟结点。通过调用 GetSibling 函数，传递左子树根结点的指针和目标值 e，可以递归地在左子树中查找。

8. ‘if (tmp) ... else ... ‘: 这是一个条件语句，判断是否找到了目标结点的兄弟结点。如果 tmp 非空，表示找到了兄弟结点，直接返回 tmp 指针；否则，执行 else 语句块中的代码。

9. ‘tmp = GetSibling(T->rchild, e);‘: 这是一个递归调用，用于在右子树中查找目标结点的兄弟结点。通过调用 GetSibling 函数，传递右子树根结点的指针和目标值 e，可以递归地在右子树中查找。

10. ‘return NULL;‘: 当二叉树为空或者递归查找结束后没有找到目标结点的兄弟结点时，返回空指针表示未找到。

通过递归调用，GetSibling 函数会在二叉树中深度优先搜索，先检查当前结点是否为目标结点，如果是，则返回空指针表示没有兄弟结点。如果当前结点为根节点且同时具有左孩子和右孩子，会判断左右孩子的关键字值是否等于目标值 e，返回相应的兄弟结点。如果未找到兄弟结点，则递归地在左子树和右子树中查找，直到找到目标结点的兄弟结点或者遍历完整个二叉树。这种实现方式可

以在二叉树中准确地找到指定结点的兄弟结点。

2.2.7 插入结点

```
// 插入结点
status InsertNode(BiTree &T, KeyType e, int LR, TElemType c) {
  BiTree tmp;
  BiTree tmp3 = LocateNode(T, c.key);
  tmp = LocateNode(T, e);
  if (tmp3) {
    return ERROR;
  }
  if (LR == -1) {
    tmp = (BiTree)malloc(sizeof(BiTNode));
    memcpy(&tmp->data, &c, sizeof(TElemType));
    tmp->rchild = T;
    T = tmp;
    T->lchild = NULL;
    return OK;
  }  // 以上为处理根节点
  if (tmp == NULL) {
    return ERROR;
  }
  BiTree tmp2;  //非根节点，按要求处理
  if (LR == 1) {
    tmp2 = (BiTree)malloc(sizeof(BiTNode));
    memcpy(&tmp2->data, &c, sizeof(TElemType));
    tmp2->rchild = tmp->rchild;
    tmp2->lchild = NULL;
    tmp->rchild = tmp2;
    return OK;
  } else if (LR == 0) {
```

```
29      tmp2 = (BiTree)malloc(sizeof(BiTNode));
30      memcpy(&tmp2->data, &c, sizeof(TElemType));
31      tmp2->rchild = tmp->lchild;
32      tmp2->lchild = NULL;
33      tmp->lchild = tmp2;
34      return OK;
35    }
36    return OK;
37  }
```

上述代码是用于在二叉树中插入结点的函数。让我们逐行进行详细分析：

1. ‘status InsertNode(BiTree T, KeyType e, int LR, TElemType c)‘: 这是函数的定义，它接受一个指向二叉树根结点的指针 T、一个关键字值 e、一个插入位置标志 LR（-1 表示作为根结点的左孩子，0 表示作为 e 结点的左孩子，1 表示作为 e 结点的右孩子）以及待插入的结点数据 c 作为参数，并返回一个状态值。

2. ‘BiTree tmp; BiTree tmp3 = LocateNode(T, c.key); tmp = LocateNode(T, e);‘: 声明两个临时变量 tmp 和 tmp3，其中 tmp 用于存储查找到的插入位置结点的指针，tmp3 用于存储关键字值为 c.key 的结点的指针。

3. ‘if (tmp3) return ERROR; ‘: 如果 tmp3 非空，表示关键字值为 c.key 的结点已经存在于二叉树中，返回错误状态。

4. ‘if (LR == -1) ... ‘: 这是一个条件语句，判断插入位置标志 LR 的值。如果 LR 为-1，表示要将新结点作为根结点的左孩子插入。

5. ‘tmp = (BiTree)malloc(sizeof(BiTNode)); memcpy(tmp->data, c, sizeof(TElemType));‘: 分配新结点的内存空间，并将新结点的数据 c 复制到 tmp 指向的结点的 data 字段中。

6. ‘tmp->rchild = T; T = tmp; T->lchild = NULL;‘: 修改指针关系，将新结点 tmp 作为根结点 T 的左孩子，并设置新结点的右孩子为原来的根结点 T。

7. ‘if (tmp == NULL) return ERROR; ‘: 如果 tmp 为空，表示未找到插入位置的结点，返回错误状态。

8. ‘BiTree tmp2;‘: 声明一个临时变量 tmp2，用于存储待插入的新结点。

9. ‘if (LR == 1) ... else if (LR == 0) ... ‘: 这是一个条件语句，根据插入位置标志 LR 的值进行不同的处理。如果 LR 为 1，表示要将新结点作为 e 结点的右

孩子插入；如果 LR 为 0，表示要将新结点作为 e 结点的左孩子插入。

10. ‘tmp2 = (BiTree)malloc(sizeof(BiTNode)); memcpy(tmp2->data, c, sizeof(TElemType));‘: 分配新结点的内存空间，并将新结点的数据 c 复制到 tmp2 指向的结点的 data 字段中。

11. ‘tmp2->rchild = tmp->rchild; tmp2->lchild = NULL; tmp->rchild = tmp2;‘: 修改指针关系，将新结点 tmp2 插入到 e 结点的右侧，将 e 结点原有的右孩子作为新结点的右孩子。

12. ‘tmp2 = (BiTree)malloc(sizeof(BiTNode)); memcpy(tmp2->data, c, sizeof(TElemType));‘: 分配新结点的内存空间，并将新结点的数据 c 复制到 tmp2 指向的结点的 data 字段中。

13. ‘tmp2->rchild = tmp->lchild; tmp2->lchild = NULL; tmp->lchild = tmp2;‘: 修改指针关系，将新结点 tmp2 插入到 e 结点的左侧，将 e 结点原有的左孩子作为新结点的右孩子。

14. ‘return OK;‘: 返回状态值表示插入操作成功。

通过使用 malloc 函数分配内存空间，并设置指针关系，可以在指定位置插入新结点。根据插入位置标志 LR 的值，可以插入为根结点的左孩子、e 结点的左孩子或 e 结点的右孩子。如果插入位置已经存在结点或未找到插入位置，则返回错误状态。

2.2.8 删除结点

```
// 删除节点
BiTNode *Getfather(BiTree T, KeyType e) {  //删除节点需要先找到对应节点的父节点，所以
  if (T) {
    if (T->data.key == e) {  //根节点
      return NULL;
    }
    if (T->lchild && T->rchild) {
      if (T->lchild->data.key == e) {
        return T;
      }
      if (T->rchild->data.key == e) {
```

```
12          return T;
13        }
14      } else if (T->lchild) {
15        if (T->lchild->data.key == e) {
16          return T;
17        }
18      } else if (T->rchild) {
19        if (T->rchild->data.key == e) {
20          return T;
21        }
22      }  // 以上为特殊情况
23      //一下不特殊，直接遍历去找
24      BiTree tmp = Getfather(T->lchild, e);
25      if (tmp) {
26        return tmp;
27      } else {
28        tmp = Getfather(T->rchild, e);
29        return tmp;
30      }
31    } else {
32      return NULL;
33    }
34  }
35  status DeleteNode(BiTree &T, KeyType e) {
36    BiTree tmp2;
37    if (T->data.key == e) {  //删除根节点
38      if (T->lchild && T->rchild) {
39        tmp2 = T->lchild;
40        while (tmp2 && tmp2->rchild) {  //找到左子树的最右节点
41          tmp2 = tmp2->rchild;
42        }
```

```
43        tmp2->rchild = T->rchild;
44        tmp2 = T->lchild;
45        free(T);
46        T = tmp2;
47        return OK;
48  //左右子树都存在，将左子树的最右节点的右孩子指向右子树，然后将左子树作为根节点
49      } else if (T->lchild) {
50        tmp2 = T->lchild;
51        free(T);
52        T = tmp2;
53      } else {
54        tmp2 = T->rchild;
55        free(T);
56        T = tmp2;
57      }
58    }
59    BiTree tmp = Getfather(T, e);  //找到父节点
60    if (tmp) {
61      if (tmp->lchild && tmp->lchild->data.key == e) {  //判断是左孩子还是右孩子
62        if (tmp->lchild->lchild) {  //左子树存在
63          if (tmp->lchild->rchild) {  //左右子树都存在
64            tmp2 = tmp->lchild->lchild;
65            while (tmp2->rchild) {
66              tmp2 = tmp2->rchild;
67            }
68            tmp2->rchild = tmp->lchild->rchild;
69            tmp2 = tmp->lchild->lchild;
70            free(tmp->lchild);
71            tmp->lchild = tmp2;
72            return OK;
73          } else {  //左子树存在，右子树不存在
```

```
74            tmp2 = tmp->lchild->lchild;
75            free(tmp->lchild);
76            tmp->lchild = tmp2;
77            return OK;
78          }
79        } else {
80          if (tmp->lchild->rchild) {  //左子树不存在，右子树存在
81            tmp2 = tmp->lchild->rchild;
82            free(tmp->lchild);
83            tmp->lchild = tmp2;
84            return OK;
85          } else {  //左右子树都不存在
86            free(tmp->lchild);
87            tmp->lchild = NULL;
88            return OK;
89          }
90        }
91
92      } else {
93        if (tmp->rchild->lchild) {  //判断是左孩子还是右孩子
94          if (tmp->rchild->rchild) {  //左右子树都存在
95            tmp2 = tmp->rchild->lchild;
96            while (tmp2->rchild) {
97              tmp2 = tmp2->rchild;
98            }
99            tmp2->rchild = tmp->rchild->rchild;
100           tmp2 = tmp->rchild->lchild;
101           free(tmp->rchild);
102           tmp->rchild = tmp2;
103           return OK;
104         } else {  //只有左子树存在
```

```
105             tmp2 = tmp->rchild->lchild;
106             free(tmp->rchild);
107             tmp->rchild = tmp2;
108             return OK;
109           }
110         } else {  //左子树不存在
111           if (tmp->rchild->rchild) {  //右子树存在
112             tmp2 = tmp->rchild->rchild;
113             free(tmp->rchild);
114             tmp->rchild = tmp2;
115             return OK;
116           } else {  //左右子树都不存在
117             free(tmp->rchild);
118             tmp->rchild = NULL;
119             return OK;
120           }
121         }
122       }
123
124     } else {  //没有找到父节点，说明没有这个节点
125       return ERROR;
126     }
127 }
```

上述代码是用于删除二叉树中指定结点的函数。让我们逐行进行详细分析：

1. ‘BiTNode *Getfather(BiTree T, KeyType e) ... ‘: 这是一个辅助函数，用于查找指定结点的父节点。它接受一个指向二叉树根结点的指针 T 和一个关键字值 e 作为参数，并返回一个指向找到的父节点的指针。

2. ‘status DeleteNode(BiTree T, KeyType e) ... ‘: 这是函数的定义，它接受一个指向二叉树根结点的指针 T 和一个关键字值 e 作为参数，并返回一个状态值。

3. ‘BiTree tmp2;‘: 声明一个临时变量 tmp2，用于存储待操作的结点。

4. ‘if (T->data.key == e) ... ‘: 这是一个条件语句，判断根结点的关键字值是

否等于目标值 e。如果相等，表示要删除根结点。

5. ‘if (T->lchild T->rchild) ... ‘: 这是一个条件语句，判断根结点的左孩子和右孩子是否都存在。如果是根节点且左右孩子都存在，才进入 if 语句块。

6. ‘tmp2 = T->lchild; while (tmp2 tmp2->rchild) tmp2 = tmp2->rchild; tmp2->rchild = T->rchild; tmp2 = T->lchild; free(T); T = tmp2;‘: 在删除根结点的情况下，将根结点的左子树的最右结点（即最右的叶子结点）的右孩子指向根结点的右子树，然后将左子树作为新的根结点。

7. ‘else if (T->lchild) ... ‘: 这是一个条件语句，判断根结点的左孩子是否存在。如果只有左孩子存在，执行 else if 语句块中的代码。

8. ‘tmp2 = T->lchild; free(T); T = tmp2;‘: 在删除根结点的情况下，将左孩子作为新的根结点。

9. ‘else ... ‘: 这是一个条件语句，判断根结点的右孩子是否存在。如果只有右孩子存在，执行 else 语句块中的代码。

10. ‘tmp2 = T->rchild; free(T); T = tmp2;‘: 在删除根结点的情况下，将右孩子作为新的根结点。

11. ‘BiTree tmp = Getfather(T, e);‘: 调用 Getfather 函数，查找目标结点的父节点。

12. ‘if (tmp) ... ‘: 这是一个条件语句，判断是否找到了目标结点的父节点。如果找到了父节点，执行 if 语

句块中的代码；否则，执行 else 语句块中的代码。

13. ‘if (tmp->lchild tmp->lchild->data.key == e) ... ‘: 这是一个条件语句，判断目标结点是父节点的左孩子还是右孩子。如果目标结点是父节点的左孩子，执行 if 语句块中的代码。

14. ‘if (tmp->lchild->lchild) ... else ... ‘: 这是一个条件语句，判断目标结点的左子树是否存在。如果左子树存在，执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

15. ‘if (tmp->lchild->rchild) ... else ... ‘: 这是一个条件语句，判断目标结点的右子树是否存在。如果右子树存在，执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

16. ‘else ... ‘: 这是一个条件语句，处理目标结点是父节点的右孩子的情况。

17. ‘if (tmp->rchild->lchild) ... else ... ‘: 这是一个条件语句，判断目标结点

的左子树是否存在。如果左子树存在，执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

18. ‘if (tmp->rchild->rchild) ... else ... ‘: 这是一个条件语句，判断目标结点的右子树是否存在。如果右子树存在，执行 if 语句块中的代码；否则，执行 else 语句块中的代码。

19. ‘else ... ‘: 这是一个条件语句，处理目标结点的左右子树都不存在的情况。

20. ‘return ERROR;‘: 如果没有找到目标结点的父节点，表示目标结点不存在于二叉树中，返回错误状态。

通过调用 Getfather 函数，找到目标结点的父节点，然后根据不同的情况进行删除操作。在删除根结点的情况下，需要将根结点的左子树中最右的叶子结点与右子树进行合并，或者只保留左子树或右子树作为新的根结点。在删除非根结点的情况下，根据目标结点是父节点的左孩子还是右孩子，以及目标结点的子树情况，进行不同的指针调整和内存释放操作。这种实现方式可以在二叉树中准确地删除指定的结点。

2.2.9 遍历二叉树

先序遍历

```
void visit(BiTree t) { printf("%d %s\n", t->data.key, t->data.others); }

// 先序遍历二叉树 T
status PreOrderTraverse(BiTree T, void (*visit)(BiTree)) {
  if (T) {
    visit(T);
    PreOrderTraverse(T->lchild, visit);
    PreOrderTraverse(T->rchild, visit);
  }
  return OK;
}
```

中序遍历

```c
// 中序遍历二叉树 T
status InOrderTraverse(BiTree T, void (*visit)(BiTree)) {
  if (T) {
    InOrderTraverse(T->lchild, visit);
    visit(T);
    InOrderTraverse(T->rchild, visit);
  }
  return OK;
}
```

后序遍历

```c
// 后序遍历二叉树 T
status PostOrderTraverse(BiTree T, void (*visit)(BiTree)) {
  if (T) {
    PostOrderTraverse(T->lchild, visit);
    PostOrderTraverse(T->rchild, visit);
    visit(T);
  }
  return OK;
}
```

层序遍历

```c
// 按层遍历二叉树 T
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree)) {
  int begin, end;
  begin = end = 0;
  BiTree arr[50];
  if (T == NULL) {  //如果树为空，返回错误
    return ERROR;
  }
  arr[end++] = T;
  while (begin != end) {  //如果 begin 等于 end，说明队列为空，结束循环
```

```
11      visit(arr[begin]);
12      if (arr[begin]->lchild) {
13        arr[end++] = arr[begin]->lchild;
14      }
15      if (arr[begin]->rchild) {
16        arr[end++] = arr[begin]->rchild;
17      }
18      begin++;
19    }
20    return OK;
21  }
```

按层遍历二叉树的函数逐行进行详细分析：

1. ‘status LevelOrderTraverse(BiTree T, void (*visit)(BiTree))‘：这是函数的定义，它接受一个指向二叉树根结点的指针 T 和一个函数指针 visit 作为参数，并返回一个状态值。

2. ‘int begin, end; begin = end = 0;‘：声明两个整型变量 begin 和 end，用于标记遍历过程中队列的起始位置和结束位置，并初始化为 0。

3. ‘BiTree arr[50];‘：声明一个大小为 50 的数组 arr，用于存储遍历过程中的结点。

4. ‘if (T == NULL) return ERROR; ‘：如果二叉树为空，即根结点指针 T 为空，返回错误状态。

5. ‘arr[end++] = T;‘：将根结点 T 放入数组 arr 中，并将 end 自增，表示结束位置后移。

6. ‘while (begin != end) ... ‘：这是一个循环语句，判断队列是否为空。如果 begin 等于 end，说明队列为空，结束循环。

7. ‘visit(arr[begin]);‘：调用函数指针 visit，访问队列中的当前结点。

8. ‘if (arr[begin]->lchild) arr[end++] = arr[begin]->lchild; ‘：如果当前结点存在左孩子，将左孩子放入队列中，并将 end 自增，表示结束位置后移。

9. ‘if (arr[begin]->rchild) arr[end++] = arr[begin]->rchild; ‘：如果当前结点存在右孩子，将右孩子放入队列中，并将 end 自增，表示结束位置后移。

10. ‘begin++;‘：将 begin 自增，表示起始位置后移，准备访问下一个结点。

11. ‘return OK;‘: 返回状态值表示遍历操作成功。

该函数通过使用一个数组作为队列，从根结点开始按层遍历二叉树。首先将根结点放入队列中，然后进入循环，不断从队列中取出结点并访问，同时将其子结点放入队列中。通过不断更新 begin 和 end 的值，实现了层序遍历。该遍历方式可以按层次顺序访问二叉树的所有结点。

2.2.10 文件读写

```
// 将二叉树的结点数据写入到文件 FileName 中
status SaveBiTree(BiTree T, char FileName[]) {
  FILE *fd = fopen(FileName, "w");
  if (T) {
    BiTree p = T;
    BiTree stack[50];  //定义一个栈
    int num = 0;
    while (NULL != p || num > 0) {  //如果 p 不为空或者栈不为空，循环
      while (NULL != p) {
        fprintf(fd, "%d %s\n", p->data.key, p->data.others);  //写入文件
        stack[num++] = p;
        p = p->lchild;
      }
      if (p == NULL) {   // 特殊情况，写入一个 null
        fprintf(fd, "%d %s\n", 0, "null");
      }
      num--;
      p = stack[num];
      p = p->rchild;
    }
    fprintf(fd, "%d %s\n", 0, "null");
    fclose(fd);
    return OK;
  } else {
```

```
25      return ERROR;
26    }
27  }
28  // 读入文件 FileName 的结点数据，创建二叉树
29  status LoadBiTree(BiTree &T, char FileName[]) {
30    FILE *fd = fopen(FileName, "r");
31    BiTree stack[50];  // 也是需要一个栈
32    int top = 0;
33    int key;
34    char others[20];
35    if (T == NULL) {
36      BiTree tmp;
37      BiTree node;
38      T = (BiTree)malloc(sizeof(BiTNode));
39      T->lchild = NULL;
40      T->rchild = NULL;
41      fscanf(fd, "%d %s", &key, others);
42      T->data.key = key;
43      strcpy(T->data.others, others);
44      stack[top] = T;
45      tmp = T;
46      int lr = 0;
47      while (top >= 0) {  // 栈不空，依次读取，并写入栈，
48        if (fscanf(fd, "%d %s", &key, others) == EOF) {
49          break;
50        }
51        if (strcmp(others, "null") == 0) {
52          if (lr == 0) {
53            tmp->lchild = NULL;
54            lr = 1;
55          } else {
```

```
56            if (tmp->rchild) {
57            } else {
58              tmp->rchild = NULL;
59            }
60            top--;
61            if (top < 0) {
62              break;
63            }
64            tmp = stack[top];
65            lr = 1;
66          }
67
68        } else {
69          node = (BiTree)malloc(sizeof(BiTNode));
70          node->data.key = key;
71          strcpy(node->data.others, others);
72          node->lchild = NULL;
73          node->rchild = NULL;
74          if (lr == 0) {
75            tmp->lchild = node;
76            stack[++top] = tmp->lchild;
77            tmp = tmp->lchild;
78          } else {
79            tmp->rchild = node;
80            stack[++top] = tmp->rchild;
81            tmp = tmp->rchild;
82            lr = 0;
83          }
84        }
85      }
86      return OK;
```

```
87    } else {
88      return ERROR;
89    }
90  }
```

上述代码实现了将二叉树的结点数据写入文件和从文件中读取结点数据创建二叉树的功能。

函数 ‘status SaveBiTree(BiTree T, char FileName[])‘用于将二叉树的结点数据写入文件中。它接受一个指向二叉树根结点的指针 T 和一个文件名 FileName 作为参数，并返回一个状态值。函数首先打开指定文件，并创建一个文件指针 fd，以写入模式打开文件。然后通过使用栈的方式，从根结点开始深度优先遍历二叉树，将每个结点的数据写入文件中，包括关键字值和其他数据。在遍历过程中，先将当前结点入栈，然后依次将左孩子入栈，并将当前结点指向左孩子，直到左子树为空。当左子树为空时，判断当前结点是否为空，若为空，表示特殊情况，写入一个表示空结点的标记。然后从栈中弹出一个结点，将当前结点指向该结点的右孩子，并继续遍历右子树。直到栈为空，遍历结束。最后，关闭文件指针 fd，返回状态值表示写入操作是否成功。

函数 ‘status LoadBiTree(BiTree T, char FileName[])‘用于从文件中读取结点数据，并创建二叉树。它接受一个指向二叉树根结点的指针 T 和一个文件名 FileName 作为参数，并返回一个状态值。函数首先打开指定文件，并创建一个文件指针 fd，以读取模式打开文件。然后声明一个栈 stack，用于辅助构建二叉树。接下来，读取文件中的第一个结点数据，并将其作为根结点构建二叉树。然后从文件中逐个读取结点数据，根据数据内容构建二叉树。遇到关键字值和其他数据表示空结点的标记时，根据栈的情况进行相应的处理，调整指针关系。直到文件读取结束或栈为空，读取和构建过程结束。最后，关闭文件指针 fd，返回状态值表示读取和构建操作是否成功。

这两个函数通过文件的读取和写入，实现了将二叉树的结点数据保存到文件中以及从文件中恢复二叉树的功能。

2.2.11 求最大路径和

```
1  // 二叉树从根节点开始的最大路径和
2  int MaxPathSum(BiTree T) {
```

```
  int max = 0;
  int tmp = 0;
  if (T) {
    tmp = T->data.key;
    max = tmp;
    if (T->lchild) {
      tmp += MaxPathSum(T->lchild);  //递归寻找
    }
    if (T->rchild) {
      max += MaxPathSum(T->rchild);
    }
    if (tmp > max) {
      max = tmp;
    }
  }
  return max;
}
```

上述代码实现了求二叉树从根节点开始的最大路径和的功能。

函数 ‘int MaxPathSum(BiTree T)‘接受一个指向二叉树根节点的指针 T，并返回一个整数值表示最大路径和。函数使用递归的方式遍历二叉树，从根节点开始计算路径和。

首先，声明两个变量 ‘max‘和 ‘tmp‘，分别用于保存当前的最大路径和和临时路径和。

然后，判断根节点是否为空。如果根节点为空，表示遍历结束，返回 0。

如果根节点不为空，将根节点的关键字值赋给 ‘tmp‘和 ‘max‘，作为初始值。

接下来，分别判断左子树和右子树是否存在。如果左子树存在，递归调用 ‘MaxPathSum‘函数计算左子树的最大路径和，并将结果加到 ‘tmp‘上。如果右子树存在，递归调用 ‘MaxPathSum‘函数计算右子树的最大路径和，并将结果加到 ‘max‘上。

最后，比较 ‘tmp‘和 ‘max‘的值，将较大的值赋给 ‘max‘，表示当前的最大路径和。

最终，返回 ‘max‘作为二叉树从根节点开始的最大路径和。

该函数通过递归遍历二叉树的方式，计算出从根节点开始的最大路径和，包括根节点及其子树中的结点。

2.2.12 求最近公共祖先

```
// 两节点的最近公共祖先
BiTree LowestCommonAncestor(BiTree T, int e1, int e2) {
  if (T) {
    if (T->data.key == e1 || T->data.key == e2) {
      return T;
    }    // 也是用到递归
    BiTree l = LowestCommonAncestor(T->lchild, e1, e2);
    BiTree r = LowestCommonAncestor(T->rchild, e1, e2);
    if (l && r) {
      return T;
    } else if (l) {
      return l;
    } else if (r) {
      return r;
    } else {
      return NULL;
    }
  } else {
    return NULL;
  }
}
```

上述代码实现了找到两个节点的最近公共祖先的功能。

函数 ‘BiTree LowestCommonAncestor(BiTree T, int e1, int e2)‘接受一个指向二叉树根节点的指针 T，以及两个节点的关键字值 e1 和 e2 作为参数，并返回一个指向最近公共祖先节点的指针。

函数使用递归的方式遍历二叉树，从根节点开始查找。

首先，判断根节点是否为空。如果为空，表示遍历结束，返回 NULL。

如果根节点不为空，判断根节点的关键字值是否等于 e1 或 e2。如果相等，表示根节点即为最近公共祖先，直接返回根节点。

如果根节点的关键字值不等于 e1 或 e2，则分别在左子树和右子树中递归调用 ‘LowestCommonAncestor‘函数，查找 e1 和 e2 的最近公共祖先。

如果左子树和右子树的结果都非空，说明 e1 和 e2 分别位于根节点的左子树和右子树中，此时根节点即为最近公共祖先，返回根节点。

如果左子树的结果非空，而右子树的结果为空，说明 e1 和 e2 都位于根节点的左子树中，返回左子树的结果作为最近公共祖先。

如果右子树的结果非空，而左子树的结果为空，说明 e1 和 e2 都位于根节点的右子树中，返回右子树的结果作为最近公共祖先。

如果左子树和右子树的结果都为空，说明 e1 和 e2 都不在根节点及其子树中，返回 NULL。

最终，返回找到的最近公共祖先节点的指针。

该函数通过递归遍历二叉树的方式，找到给定两个节点的最近公共祖先节点，并返回该节点的指针。

2.2.13 翻转二叉树

```
// 翻转二叉树，互换所有左右节点
int InvertTree(BiTree T) {
  if (T) {
    BiTree tmp = T->lchild;
    T->lchild = T->rchild;
    T->rchild = tmp;
    InvertTree(T->lchild);
    InvertTree(T->rchild);
  }
  return OK;
}
```

上述代码实现了翻转二叉树的功能。

函数 ‘int InvertTree(BiTree T)‘接受一个指向二叉树根节点的指针 T，并返回一个状态值。

函数使用递归的方式遍历二叉树，从根节点开始翻转。

首先，判断根节点是否为空。如果为空，表示遍历结束，返回 OK 状态。

如果根节点不为空，首先交换根节点的左孩子和右孩子，实现节点的翻转。

然后，递归调用 ‘InvertTree‘函数对根节点的左子树和右子树进行翻转。

通过递归的方式，会不断向下遍历二叉树，将每个节点的左孩子和右孩子互换位置，实现整个二叉树的翻转。

最终，返回 OK 状态表示翻转操作完成。

该函数通过递归遍历二叉树的方式，实现了将二叉树中所有节点的左孩子和右孩子进行互换的操作，从而实现二叉树的翻转。

2.2.14 实现多树操作

实现多树操作的方式和链表实现的线性表是相同的，都是构建了一个头节点数组，然后切换二叉树时只需要更改数组下标即可。

2.3 实验小结

通过实现这些功能，学到以下几点：

1. 理解二叉树的基本结构和性质：通过实现二叉树的各种操作，深入理解了二叉树的节点、左孩子、右孩子之间的关系，以及二叉树的遍历方式。

2. 掌握二叉树的常用操作：通过实现二叉树的插入、删除、查找、赋值、翻转等操作，掌握了二叉树的常用操作技巧，能够对二叉树进行各种操作和修改。

3. 熟悉递归算法：二叉树的很多操作都使用了递归算法进行实现，通过实现这些功能，熟悉了递归的思想和使用方式，了解递归在二叉树操作中的应用。

4. 理解算法的复杂度：通过实现这些功能，对算法的时间和空间复杂度有了更深入的了解。不同的操作对应着不同的算法复杂度，例如遍历、查找等操作的复杂度是多少。

5. 锻炼编程能力：通过实践，锻炼了自己的编程能力，包括代码实现、调试和测试的能力，提高了对数据结构和算法的理解和应用能力。

总而言之，通过实现这些功能，在理论和实践中更深入地学习和掌握了二叉树的基本概念、常用操作和算法思想，提升编程和问题解决能力。

附录 A　基于顺序存储结构的线性表实现

头文件 define.h :

```
/* Linear Table On Sequence Structure */
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*---------page 10 on textbook ---------*/
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

typedef int status;
typedef int ElemType;  // 数据元素类型定义

/*-------page 22 on textbook -------*/
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef struct {  // 顺序表（顺序结构）的定义
  ElemType *elem;
  int length;
  int listsize;
} SqList;
/*-----page 19 on textbook ---------*/
status InitList(SqList &L);
status DestroyList(SqList &L);
status ClearList(SqList &L);
```

```
status ListEmpty(SqList L);
status ListLength(SqList L);
status GetElem(SqList L, int i, ElemType &e);
status LocateElem(SqList L, ElemType e);  // 简化过
status PriorElem(SqList L, ElemType cur, ElemType &pre_e);
status NextElem(SqList L, ElemType cur, ElemType &next_e);
status ListInsert(SqList &L, int i, ElemType e);
status ListDelete(SqList &L, int i, ElemType &e);
status ListTrabverse(SqList L);  // 简化过
status MaxSubArray(SqList L); // 求最大连续子数组和 初始条件是线性表 L 已存在且非空
status SubArrayNum(SqList L,ElemType k);  // 初始条件是线性表 L 已存在且非空，操作结
status SortList(SqList &L);  //初始条件是线性表 L 已存在；操作结果是将 L 由小到大排序
void swap(ElemType &a, ElemType &b);
status  SaveList(SqList L, char FileName[]);    //将线性表 L 保存到文件 FileName 中
status  LoadList(SqList &L, char FileName[]);   //从文件 FileName 中读取线性表 L
/*-------------------------------------------*/
```

函数定义文件 define.cpp :

```
#include "define.h"

// function definition
status InitList(SqList &L) {
  L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
  if (!L.elem) exit(OVERFLOW);
  L.length = 0;
  L.listsize = LIST_INIT_SIZE;
  return OK;
}

status DestroyList(SqList &L) {  //销毁线性表
  if (L.elem != NULL) {    // 线性表存在
    free(L.elem);
```

```
15      L.elem = NULL;
16      L.length = 0;
17      L.listsize = 0;
18      return OK;
19    } else {  // 线性表不存在
20      return INFEASIBLE;
21    }
22  }
23
24  status ClearList(SqList &L) {
25    if (L.elem != NULL) {  // 线性表存在
26      L.length = 0;
27      return OK;
28    } else {
29      return INFEASIBLE;
30    }
31  }
32
33  status ListEmpty(SqList L) {  // 判断线性表是否为空
34    if (L.elem != NULL) {
35      if (L.length == 0) {  // 线性表为空
36        return TRUE;
37      } else {
38        return FALSE;
39      }
40    } else {   // 线性表不存在
41      return INFEASIBLE;
42    }
43  }
44
45  status ListLength(SqList L) {   // 返回线性表的长度
```

```
46    if (L.elem) {  // 线性表存在
47      return L.length;
48    } else {
49      return INFEASIBLE;
50    }
51  }
52  status GetElem(SqList L, int i, ElemType &e) {  // 用 e 返回 L 中第 i 个数据元素的值
53    if (L.elem) {   // 线性表存在
54      if (i > L.length || i < 1) {
55        return ERROR;
56      } else {
57        e = L.elem[i - 1];
58        return OK;
59      }
60    } else {
61      return INFEASIBLE;
62    }
63  }
64  status LocateElem(SqList L, ElemType e) {
65    if (L.elem) {
66      int i;
67      for (i = 0; i < L.length; i++) {
68        if (L.elem[i] == e) {
69          return i + 1;
70        }
71      }
72      return 0;
73    } else {
74      return INFEASIBLE;
75    }
76  }
```

```
77  status PriorElem(SqList L, ElemType cur, ElemType &pre_e) {  // 返回 cur 的前驱
78    if (L.elem) {
79      int i;
80      for (i = 0; i < L.length; i++) {
81        if (L.elem[i] == pre_e) {  // 找到 cur
82          if (i == 0) {
83            return ERROR;  // cur 为第一个元素
84          } else {
85            pre_e = L.elem[i - 1];
86            return OK;
87          }
88        }
89      }
90      return ERROR;     // 未找到 cur
91    } else {
92      return INFEASIBLE;
93    }
94  }
95  status NextElem(SqList L, ElemType cur, ElemType &next_e) {  // 返回 cur 的后继
96    if (L.elem) {
97      int i;
98      for (i = 0; i < L.length; i++) {
99        if (L.elem[i] == cur) {
100         if (i == L.length - 1) {  // cur 为最后一个元素
101           return ERROR;
102         } else {  // cur 不是最后一个元素
103           next_e = L.elem[i + 1];
104           return OK;
105         }
106       }
107     }
```

```
108     return ERROR;
109   } else {
110     return INFEASIBLE;
111   }
112 }
113 status ListInsert(SqList &L, int i, ElemType e) { // 在第 i 个位置插入元素 e
114   if (L.elem) {     // 线性表存在
115     if (i < 1 || i > L.length + 1) {
116       return ERROR;
117     } else {
118       if (L.length == L.listsize) {  // 线性表已满
119         ElemType *tem =
120             (ElemType *)realloc(L.elem, sizeof(ElemType) * L.listsize * 2);
121         if (tem) {
122           L.elem = tem;  // 重新分配内存
123           L.listsize *= 2;
124         }
125       }
126 
127       int t;
128       if (L.length != 0 && i != L.length + 1) {  // 插入位置不在表尾
129         for (t = L.length - 1; t >= i - 1; t--) {  // 将插入位置后的元素后移
130           L.elem[t + 1] = L.elem[t];
131         }
132         L.elem[i - 1] = e;
133       } else {
134         L.elem[i - 1] = e;
135       }
136       L.length++;
137       return OK;
138     }
```

```
139   } else {  // 线性表不存在
140     return INFEASIBLE;
141   }
142 }
143 status ListDelete(SqList &L, int i, ElemType &e) {  // 删除第 i 个元素
144   if (L.elem) {
145     if (L.length >= i && i > 0) {  // 线性表存在
146       e = L.elem[i - 1];
147       for (int j = i - 1; j < L.length - 1; j++) {  // 将删除位置后的元素前移
148         L.elem[j] = L.elem[j + 1];
149       }
150       L.length--;
151       return OK;
152     } else {
153       return ERROR;
154     }
155   } else {
156     return INFEASIBLE;  // 线性表不存在
157   }
158 }
159 status ListTrabverse(SqList L) {  // 遍历线性表
160   int i;
161   printf("\n-----------all elements ----------------------\n");
162   for (i = 0; i < L.length; i++) printf("%d ", L.elem[i]);
163   printf("\n------------------ end ----------------------\n");
164   return L.length;
165 }
166
167 status MaxSubArray(SqList L) {  // 求最大子序列和
168   if (L.elem == NULL) return INFEASIBLE;
169   int i, j, k, max, sum;
```

```
170   int start, end;
171   max = 0;
172   for (i = 0; i < L.length; i++) {  // 简单遍历
173     sum = 0;
174     for (j = i; j < L.length; j++) {  // 简单遍历
175       sum += L.elem[j];   // 求和
176       if (sum > max) {  // 比较
177         max = sum;
178         start = i;
179         end = j;
180       }
181     }
182   }
183   printf(" 最大子序列和为%d, 其下标为%d 到%d\n", max, start, end);
184   return OK;
185 }
186 
187 status SubArrayNum(SqList L, ElemType k) {  // 求和为 k 的子序列个数
188   if (L.elem == NULL) return INFEASIBLE;
189   int i, j, sum;
190   int count = 0;
191   for (i = 0; i < L.length; i++) {  // 简单遍历
192     sum = 0;
193     for (j = i; j < L.length; j++) {
194       sum += L.elem[j];
195       if (sum == k) {  // 比较
196         count++;
197       }
198     }
199   }
200   printf(" 和为%d 的子序列个数为%d\n", k, count);
```

```
201   return OK;
202 }
203
204 status SortList(SqList &L) {
205   if (L.elem == NULL) return INFEASIBLE;  // 线性表不存在
206   int i, j, min;
207   for (i = 0; i < L.length; i++) {  // 简单遍历
208     min = i;
209     for (j = i + 1; j < L.length; j++) {  // 简单遍历
210       if (L.elem[j] < L.elem[min]) {
211         min = j;
212       }
213     }
214     if (min != i) {
215       swap(L.elem[i], L.elem[min]);
216     }
217   }
218   return OK;
219 }
220
221 void swap(ElemType &a, ElemType &b) {  // 交换两个元素
222   ElemType temp = a;
223   a = b;
224   b = temp;
225 }
226
227 status  SaveList(SqList L, char FileName[]) {  // 保存线性表
228 ^^Iif (!L.elem)
229 ^^I^^Ireturn INFEASIBLE;
230 ^^Ielse {
231 ^^I^^IFILE *fp;
```

```
232 ^^I^^Ifp = fopen(FileName, "w");  // 打开文件
233 ^^I^^Ifor (int i = 0; i < L.length; i++)
234 ^^I^^I^^Ifprintf(fp, "%d ", L.elem[i]);
235 ^^I^^Ifclose(fp);
236 ^^I^^Ireturn OK;
237 ^^I}
238 }
239
240 status  LoadList(SqList &L, char FileName[]) {  // 读取线性表
241 ^^Iif (L.elem)  // 线性表已存在
242 ^^I^^Ireturn INFEASIBLE;
243 ^^Ielse {
244 ^^I^^IFILE *fp;
245 ^^I^^Ifp = fopen(FileName, "r");
246     if(fp == NULL) return ERROR;
247 ^^I^^IL.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
248 ^^I^^Iint len = 0;
249 ^^I^^Iwhile (!feof(fp))
250 ^^I^^I^^Ifscanf(fp, "%d", &L.elem[len++]);
251 ^^I^^IL.length = len - 1;
252 ^^I^^Ifclose(fp);
253 ^^I^^Ireturn OK;
254 ^^I}
255 }
256
```

主要流程文件 main.cpp :

```
1 #include "define.h"
2 int main(void) {
3   SqList LS[20];  // 顺序表数组
4   int i = 0;      // 顺序表数组下标
5   int length;
```

```
6   int tmp_status;
7   int op = 1;
8   int pos = 0;          // 插入位置 or 删除位置 or 查找位置
9   int elem = 0;         // 获取的元素
10  int pre_or_next = 0;  // 前驱 or 后继
11  char filename[20];
12  while (op) {
13    system("cls");
14    printf("\n\n");
15    printf("      Menu for Linear Table On Sequence Structure \n");
16    printf("-------------------------------------------------\n");
17    printf("    ^^I  1. InitList       7. LocateElem\n");
18    printf("    ^^I  2. DestroyList    8. PriorElem\n");
19    printf("    ^^I  3. ClearList      9. NextElem \n");
20    printf("    ^^I  4. ListEmpty      10. ListInsert\n");
21    printf("    ^^I  5. ListLength     11. ListDelete\n");
22    printf("    ^^I  6. GetElem        12. ListTraverse\n");
23    printf("        13. MaxSubArray   14. SubArrayNum\n");
24    printf("    ^^I  15. SortList      16. Save\n");
25    printf("    ^^I  17. Load          18. ChangeList\n");
26    printf("    ^^I  0. Exit\n");
27    printf("-------------------------------------------------\n");
28    printf("    请选择你的操作 [0~18]:");
29    scanf("%d", &op);
30    switch (op) {
31      case 1:
32        if (InitList(LS[i]) == OK)
33          printf(" 线性表创建成功! \n");
34        else
35          printf(" 线性表创建失败! \n");
36        getchar();
```

```
37          getchar();
38          break;
39        case 2:
40          if (DestroyList(LS[i]) == OK)
41            printf(" 线性表销毁成功! \n");
42          else
43            printf(" 不存在线性表\n");
44          getchar();
45          getchar();
46          break;
47        case 3:
48          if (ClearList(LS[i]) == OK)
49            printf(" 线性表元素已清除\n");
50          else
51            printf(" 不存在线性表\n");
52          getchar();
53          getchar();
54          break;
55        case 4:
56          tmp_status = ListEmpty(LS[i]);
57          if (tmp_status == TRUE)
58            printf(" 线性表无元素\n");
59          else if (tmp_status == FALSE)
60            printf(" 线性表元素不为空\n");
61          else
62            printf(" 不存在线性表\n");
63          getchar();
64          getchar();
65          break;
66        case 5:
67          length = ListLength(LS[i]);
```

```
68          if (length == INFEASIBLE)
69            printf(" 线性表为空\n");
70          else
71            printf(" 线性表长度为%d\n", length);
72          getchar();
73          getchar();
74          break;
75        case 6:
76          printf(" 请输入要查找的位置: ");
77          scanf("%d", &pos);
78          tmp_status = GetElem(LS[i], pos, elem);
79          if (tmp_status == OK)
80            printf(" 第%d 个元素为%d\n", pos, elem);
81          else if (tmp_status == ERROR)
82            printf(" 输入位置错误\n");
83          else
84            printf(" 不存在线性表\n");
85          getchar();
86          getchar();
87          break;
88        case 7:
89          printf(" 请输入要查找的元素: ");
90          scanf("%d", &elem);
91          tmp_status = LocateElem(LS[i], elem);
92          if (tmp_status == INFEASIBLE)
93            printf(" 不存在线性表\n");
94          else if (tmp_status == 0)
95            printf(" 不存在该元素\n");
96          else
97            printf(" 元素%d 的位置为%d\n", elem, tmp_status);
98          getchar();
```

```
99          getchar();
100         break;
101       case 8:
102         printf(" 输入你想查找前驱元素的元素: ");
103         scanf("%d", &elem);
104         tmp_status = PriorElem(LS[i], elem, pre_or_next);
105         if (tmp_status == INFEASIBLE)
106           printf(" 不存在线性表\n");
107         else if (tmp_status == ERROR)
108           printf(" 不存在前驱元素\n");
109         else
110           printf(" 元素%d 的前驱元素为%d\n", elem, pre_or_next);
111         getchar();
112         getchar();
113         break;
114       case 9:
115         printf(" 输入你想查找后继元素的元素: ");
116         scanf("%d", &elem);
117         tmp_status = NextElem(LS[i], elem, pre_or_next);
118         if (tmp_status == INFEASIBLE)
119           printf(" 不存在线性表\n");
120         else if (tmp_status == ERROR)
121           printf(" 不存在后继元素\n");
122         else
123           printf(" 元素%d 的后继元素为%d\n", elem, pre_or_next);
124         getchar();
125         getchar();
126         break;
127       case 10:
128         printf(" 输入你想插入的位置: ");
129         scanf("%d", &pos);
```

```
130         printf("\n");
131         printf(" 输入你想插入的元素: ");
132         scanf("%d", &elem);
133         tmp_status = ListInsert(LS[i], pos, elem);
134         if (tmp_status == OK)
135           printf(" 插入成功\n");
136         else if (tmp_status == ERROR)
137           printf(" 插入位置错误\n");
138         else
139           printf(" 不存在线性表\n");
140         getchar();
141         getchar();
142         break;
143       case 11:
144         printf(" 输入你想删除的位置: ");
145         scanf("%d", &pos);
146         tmp_status = ListDelete(LS[i], pos, elem);
147         if (tmp_status == OK)
148           printf(" 删除成功, 删除元素为%d\n", elem);
149         else if (tmp_status == ERROR)
150           printf(" 删除位置错误\n");
151         else
152           printf(" 不存在线性表\n");
153         getchar();
154         getchar();
155         break;
156       case 12:
157         // printf("\n----ListTrabverse 功能待实现! \n");
158         if (!ListTrabverse(LS[i])) printf(" 线性表是空表! \n");
159         getchar();
160         getchar();
```

```
161           break;
162         case 13:
163           tmp_status = MaxSubArray(LS[i]);
164           if (tmp_status == INFEASIBLE) printf(" 不存在线性表\n");
165           getchar();
166           getchar();
167           break;
168         case 14:
169           printf(" 输入你想找寻的字串和: ");
170           scanf("%d", &elem);
171           tmp_status = SubArrayNum(LS[i], elem);
172           if (tmp_status == INFEASIBLE) printf(" 不存在线性表\n");
173           getchar();
174           getchar();
175           break;
176         case 15:
177           tmp_status = SortList(LS[i]);
178           if (tmp_status == INFEASIBLE) printf(" 不存在线性表\n");
179           getchar();
180           getchar();
181           break;
182         case 16:
183           printf(" 输入放入的文件名: ");
184           scanf("%s", filename);
185           tmp_status = SaveList(LS[i], filename);
186           if (tmp_status == INFEASIBLE)
187             printf(" 线性表已经有内容, 不能覆盖\n");
188           else
189             printf(" 保存成功! \n");
190           getchar();
191           getchar();
```

```
192           break;
193         case 17:
194           printf(" 输入读取的文件名: ");
195           scanf("%s", filename);
196           tmp_status = LoadList(LS[i], filename);
197           if (tmp_status == INFEASIBLE)
198             printf(" 不存在线性表\n");
199           else if (tmp_status == ERROR)
200             printf(" 文件打开失败! \n");
201           else
202             printf(" 读取成功! \n");
203           getchar();
204           getchar();
205           break;
206         case 18:
207           printf(" 输入你想切换的线性表 (1~20):\n");
208           scanf("%d", &i);
209           i--;
210           printf(" 切换成功! \n");
211           getchar();
212           break;
213         case 0:
214           break;
215         default:
216           printf(" 输入错误, 请重新输入! \n");
217       }  // end of switch
218     }    // end of while
219     printf(" 欢迎下次再使用本系统! \n");
220     return 0;
221   }  // end of main()
222
```

```
223 /*--------page 23 on textbook --------------------*/
```

附录 B　基于链式存储结构的线性表实现

头文件 define.h :

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

typedef int status;
typedef int ElemType;  // 数据元素类型定义

#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef int ElemType;
typedef struct LNode {  // 单链表（链式结构）结点的定义
  ElemType data;
  struct LNode *next;
} LNode, *LinkList;

// 函数声明
status reverseList(LinkList &L);
status InitList(LinkList &L);
status DestroyList(LinkList &L);
status ClearList(LinkList &L);
status ListEmpty(LinkList L);
```

```
int ListLength(LinkList L);
status GetElem(LinkList L, int i, ElemType &e);
status LocateElem(LinkList L, ElemType e);
status PriorElem(LinkList L, ElemType e, ElemType &pre);
status NextElem(LinkList L, ElemType e, ElemType &next);
status ListInsert(LinkList &L, int i, ElemType e);
status ListDelete(LinkList &L, int i, ElemType &e);
status ListTraverse(LinkList L);
status SaveList(LinkList L, char FileName[]);
status LoadList(LinkList &L, char FileName[]);
status RemoveNthFromEnd(LinkList L,int n);
status sortList(LinkList L);
```

函数定义文件 define.cpp :

```
#include "define.h"
// 函数定义

// 线性表 L 不存在，构造一个空的线性表，返回 OK，否则返回 INFEASIBLE。
status InitList(LinkList &L) {
  if (L == NULL) {  // 不存在
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    return OK;
  } else {  // 存在
    return INFEASIBLE;
  }
}

// 如果线性表 L 存在，销毁线性表 L，释放数据元素的空间，返回 OK，否则返回 INFEASIBLE
status DestroyList(LinkList &L) {
  if (L == NULL) {  //不存在
    return INFEASIBLE;
```

```
  } else {
    LinkList tmp;
    while (L->next != NULL) {  // 递归地找下一个，并依次 free 掉
      tmp = L->next;
      free(L);
      L = tmp;
    }
    free(L);  // free 掉最后一个
    L = NULL;
    return OK;
  }
}

// 如果线性表 L 存在，删除线性表 L 中的所有元素，返回 OK，否则返回 INFEASIBLE。
status ClearList(LinkList &L) {
  if (L) {  //存在
    LinkList tmp;
    LinkList temp;
    tmp = L->next;  // 保留头节点，从第二个开始
    L->next = NULL;
    if(tmp == NULL){
      return ERROR;
    }
    while (tmp->next) {  //依次销毁
      temp = tmp->next;
      free(tmp);
      tmp = temp;
    }
    free(tmp);
    tmp = NULL;
    return OK;
```

```
50    } else {
51      return INFEASIBLE;
52    }
53  }
54
55  // 如果线性表 L 存在，判断线性表 L 是否为空，空就返回 TRUE，否则返回 FALSE；如果线性
56  status ListEmpty(LinkList L) {
57    if (L) {
58      if (L->next) {
59        return FALSE;
60      } else {
61        return TRUE;
62      }
63    } else {
64      return INFEASIBLE;
65    }
66  }
67
68  // 如果线性表 L 存在，返回线性表 L 的长度，否则返回 INFEASIBLE。
69  int ListLength(LinkList L) {
70    if (L) {
71      int i = 0;
72      LinkList tmp;
73      tmp = L;
74      while (tmp->next) {  //依次递归数有几个
75        i++;
76        tmp = tmp->next;
77      }
78      return i;
79    } else {
80      return INFEASIBLE;
```

```
81    }
82  }
83
84  // 如果线性表 L 存在，获取线性表 L 的第 i 个元素，保存在 e 中，返回 OK；如果 i 不合法
85  status GetElem(LinkList L, int i, ElemType &e) {
86    if (L) {
87      LinkList tmp = L;
88      int num = 0;
89      while (tmp->next) {  //递归遍历
90        num++;
91        tmp = tmp->next;
92        if (i == num) {  //找到了
93          e = tmp->data;
94          return OK;
95        }
96      }
97      return ERROR;
98    } else {
99      return INFEASIBLE;
100   }
101 }
102
103 // 如果线性表 L 存在，查找元素 e 在线性表 L 中的位置序号；如果 e 不存在，返回 ERROR；
104 status LocateElem(LinkList L, ElemType e) {
105   if (L) {  // 存在
106     LinkList tmp = L;
107     int num = 0;
108     while (tmp->next) {  // 递归遍历
109       num++;
110       tmp = tmp->next;
111       if (tmp->data == e) {  //找到
```

```
112           return num;
113         }
114       }
115       return ERROR;
116     } else {
117       return INFEASIBLE;
118     }
119   }
120
121   // 如果线性表 L 存在，获取线性表 L 中元素 e 的前驱，保存在 pre 中，返回 OK；如果没有
122   status PriorElem(LinkList L, ElemType e, ElemType &pre) {
123     if (L) {
124       LinkList tmp = L->next;
125       while (tmp && tmp->next) {  //递归遍历
126         L = tmp;
127         tmp = tmp->next;
128         if (tmp->data == e) {  // 找到
129           pre = L->data;
130           return OK;
131         }
132       }
133       return ERROR;
134     } else {
135       return INFEASIBLE;
136     }
137   }
138
139   // 如果线性表 L 存在，获取线性表 L 元素 e 的后继，保存在 next 中，返回 OK；如果没有后
140   status NextElem(LinkList L, ElemType e, ElemType &next) {
141     if (L) {  // 线性表存在
142       LinkList tmp = L;
```

```
143     while (tmp->next) {
144       tmp = tmp->next;
145       if (tmp->data == e) {
146         if (tmp->next) {
147           next = tmp->next->data;  // 看后继
148           return OK;
149         } else {
150           return ERROR;
151         }
152       }
153     }
154     return ERROR;
155   } else {
156     return INFEASIBLE;
157   }
158 }
159 
160 // 如果线性表 L 存在，将元素 e 插入到线性表 L 的第 i 个元素之前，返回 OK；当插入位置
161 status ListInsert(LinkList &L, int i, ElemType e) {
162   if (L) {
163     LinkList tmp = L;
164     int num = 0;
165     while (tmp->next) {
166       num++;
167       if (num == i) {  // 一般情况
168         LinkList temp = (LinkList)malloc(sizeof(LNode));
169         temp->next = tmp->next;  // 先连上后面
170         temp->data = e;
171         tmp->next = temp;  // 再连上前面
172         return OK;
173       }
```

```
174         tmp = tmp->next;
175       }
176       if (num + 1 == i) {  // 特殊情况
177         LinkList temp = (LinkList)malloc(sizeof(LNode));
178         temp->next = tmp->next;
179         temp->data = e;
180         tmp->next = temp;
181         return OK;
182       }
183       return ERROR;
184     } else {
185       return INFEASIBLE;
186     }
187   }
188
189   // 如果线性表 L 存在，删除线性表 L 的第 i 个元素，并保存在 e 中，返回 OK；当删除位置
190   status ListDelete(LinkList &L, int i, ElemType &e) {
191     if (L) {  // 存在
192       LinkList tmp = L;
193       int num = 0;
194       while (tmp->next) {
195         num++;
196         if (num == i) {   //找到第 i 个
197           LinkList temp = tmp->next->next;
198           e = tmp->next->data;
199           free(tmp->next);
200           tmp->next = temp;
201           return OK;
202         }
203         tmp = tmp->next;
204       }
```

```
205     return ERROR;
206   } else {
207     return INFEASIBLE;
208   }
209 }
210
211 // 如果线性表 L 存在，依次显示线性表中的元素，每个元素间空一格，返回 OK；如果线性表 L
212 status ListTraverse(LinkList L) {
213   if (L) {
214     L = L->next;
215     while (L && L->next) {  //依次遍历
216       printf("%d ", L->data);
217       L = L->next;
218     }
219     if (L != NULL) printf("%d", L->data);
220     return OK;
221   } else {
222     return INFEASIBLE;
223   }
224 }
225
226 // 如果线性表 L 存在，将线性表 L 的的元素写到 FileName 文件中，返回 OK，否则返回 INFE
227 status SaveList(LinkList L, char FileName[]) {
228   if (L) {
229     FILE *fp = fopen(FileName, "w");
230     L = L->next;
231     while (L && L->next) {  // 仍为依次遍历，只是写入文件
232       fprintf(fp, "%d ", L->data);
233       L = L->next;
234     }
235     if (L != NULL) fprintf(fp, "%d\n", L->data);
```

```
236     fclose(fp);
237     return OK;
238   } else {
239     return INFEASIBLE;
240   }
241 }
242 
243 // 如果线性表 L 不存在，将 FileName 文件中的数据读入到线性表 L 中，返回 OK，否则返回
244 status LoadList(LinkList &L, char FileName[]) {
245   if (L == NULL) {
246     L = (LinkList)malloc(sizeof(LNode));
247     L->next = NULL;
248     LinkList tmp = L;
249     FILE *fp = fopen(FileName, "r");
250     if(fp == NULL){
251       return ERROR;
252     }
253     int elem;
254     while (fscanf(fp, "%d", &elem) != EOF) {   // 依次读入即可
255       tmp->next = (LinkList)malloc(sizeof(LNode));
256       tmp = tmp->next;
257       tmp->next = NULL;
258       tmp->data = elem;
259     }
260     fclose(fp);
261     return OK;
262   } else {
263     return INFEASIBLE;
264   }
265 }
266 
```

```
267 // 逆置线性表
268 status reverseList(LinkList &L){
269   if(L){
270     LinkList tmp = L->next;
271     L->next = NULL;
272     while(tmp){
273       LinkList temp = tmp->next;
274       tmp->next = L->next;
275       L->next = tmp;
276       tmp = temp;
277     }
278     return OK;
279   }else{
280     return INFEASIBLE;
281   }
282 }
283
284 // 删除倒数第 n 个元素
285 status RemoveNthFromEnd(LinkList L,int n){
286   if(L){
287     LinkList tmp = L->next;
288     int num = 0;
289     while(tmp){
290       num++;
291       tmp = tmp->next;
292     }
293     if(num < n){
294       return ERROR;
295     }else{
296       tmp = L;
297       for(int i = 0;i < num - n;i++){
```

```
298           tmp = tmp->next;
299         }
300         LinkList temp = tmp->next->next;
301         free(tmp->next);
302         tmp->next = temp;
303         return OK;
304       }
305     }else{
306       return INFEASIBLE;
307     }
308   }
309   // 从小到大排序
310   status sortList(LinkList L){
311     if(L){
312       LinkList tmp = L->next;
313       while(tmp){
314         LinkList temp = tmp->next;
315         while(temp){
316           if(tmp->data > temp->data){
317             ElemType e = tmp->data;
318             tmp->data = temp->data;
319             temp->data = e;
320           }
321           temp = temp->next;
322         }
323         tmp = tmp->next;
324       }
325       return OK;
326     }else{
327       return INFEASIBLE;
328     }
```

```
329 }
330
```

主要流程文件 main.cpp :

```
1  #include "define.h"
2
3  // main
4  int main(void)
5  {
6    LinkList arr[30] = {NULL};
7    int num = 0;
8    int statuss;
9    char filename[20];
10   int len;
11   int i;
12   int e;
13   int n;
14   int op = 1;
15   while (op)
16   {
17     // system("cls");
18     printf("\n\n");
19     // 打印菜单
20     printf("      Menu for Linear Table On List Structure \n");
21     printf("-------------------------------------------------\n");
22     printf("    ^^I  1. 初始化                  7. 查找元素位置\n");
23     printf("    ^^I  2. 摧毁                    8. 求元素的前驱\n");
24     printf("    ^^I  3. 清空                    9. 求元素后继 \n");
25     printf("    ^^I  4. 判断是否为空            10. 插入元素\n");
26     printf("    ^^I  5. 求长度                  11. 删除元素\n");
27     printf("    ^^I  6. 获取元素                12. 打印元素\n");
28     printf("    ^^I  13. 逆置链表               14. 删除倒数第 n 个元素\n");
```

```
29      printf("    ^^I  15. 元素排序              16. 保存到文件\n");
30      printf("    ^^I  17. 从文件读取            18. 切换链表\n");
31      printf("    ^^I  0. 退出\n");
32      printf("-------------------------------------------------\n");
33      printf("    请选择你的操作 [0~18]:");
34      scanf("%d", &op);
35      switch (op)
36      {
37      case 1:
38        // 创建线性表
39        if (InitList(arr[num]) == OK)
40          printf(" 线性表创建成功! \n");
41        else
42          printf(" 线性表已经存在\n");
43        getchar();
44        getchar();
45        break;
46      case 2:
47        // 销毁线性表
48        if (DestroyList(arr[num]) == OK)
49        {
50          printf(" 线性表销毁成功! \n");
51          arr[num] = NULL;
52        }
53        else
54        {
55          printf(" 线性表不存在\n");
56        }
57        getchar();
58        getchar();
59        break;
```

```
60      case 3:
61        // 清空线性表
62        if (ClearList(arr[num]) == OK)
63        {
64          printf(" 线性表清空成功! \n");
65        }
66        else if (ClearList(arr[num]) == ERROR)
67        {
68          printf(" 线性表为空\n");
69        }
70        else
71        {
72          printf(" 线性表不存在\n");
73        }
74        getchar();
75        getchar();
76        break;
77      case 4:
78        // 判断线性表是否为空
79        if (ListEmpty(arr[num]) == TRUE)
80        {
81          printf(" 线性表为空! \n");
82        }
83        else if (ListEmpty(arr[num]) == FALSE)
84        {
85          printf(" 线性表不为空! \n");
86        }
87        else
88        {
89          printf(" 线性表不存在! \n");
90        }
```

```
91        getchar();
92        getchar();
93        break;
94      case 5:
95        // 求线性表长度
96        len;
97        if ((len = ListLength(arr[num])) == -1)
98        {
99          printf(" 线性表不存在! \n");
100       }
101       else
102       {
103         printf(" 线性表长度为%d\n", len);
104       }
105       getchar();
106       getchar();
107       break;
108     case 6:
109       // 获取元素
110       printf(" 输入要获取的元素位置: ");
111       scanf("%d", &i);
112       statuss = GetElem(arr[num], i, e);
113       if (statuss == OK)
114       {
115         printf(" 元素是: %d\n", e);
116       }
117       else if (statuss == ERROR)
118       {
119         printf(" 位置错误\n");
120       }
121       else
```

```
122         {
123           printf(" 线性表不存在! \n");
124         }
125         getchar();
126         getchar();
127         break;
128       case 7:
129         // 获取元素位置
130         int elem;
131         printf(" 请输入想获取位置的元素: ");
132         scanf("%d", &elem);
133         statuss = LocateElem(arr[num], elem);
134         if (statuss == ERROR)
135         {
136           printf(" 未找到该元素\n");
137         }
138         else if (statuss == INFEASIBLE)
139         {
140           printf(" 线性表不存在! \n");
141         }
142         else
143         {
144           printf(" 位置是: %d\n", statuss);
145         }
146         getchar();
147         getchar();
148         break;
149       case 8:
150         // 获取元素前驱
151         int pre, pre_v;
152         printf(" 请输入想获取前驱的元素: ");
```

```
153         scanf("%d", &pre);
154         statuss = PriorElem(arr[num], pre, pre_v);
155         if (statuss == ERROR)
156         {
157           printf(" 未找到该元素\n");
158         }
159         else if (statuss == INFEASIBLE)
160         {
161           printf(" 线性表不存在! \n");
162         }
163         else
164         {
165           printf(" 前驱是: %d\n", pre_v);
166         }
167         getchar();
168         getchar();
169         break;
170       case 9:
171         // 获取元素后继
172         int next, next_v;
173         printf(" 请输入想获取后继的元素: ");
174         scanf("%d", &next);
175         statuss = NextElem(arr[num], next, next_v);
176         if (statuss == ERROR)
177         {
178           printf(" 未找到该元素\n");
179         }
180         else if (statuss == INFEASIBLE)
181         {
182           printf(" 线性表不存在! \n");
183         }
```

```
184         else
185         {
186           printf(" 后继是: %d\n", next_v);
187         }
188         getchar();
189         getchar();
190         break;
191       case 10:
192         // 插入元素
193         printf(" 输入要插入的元素位置: ");
194         scanf("%d", &i);
195         printf(" 输入要插入的元素: ");
196         scanf("%d", &e);
197         if (ListInsert(arr[num], i, e) == OK)
198         {
199           printf(" 插入成功! \n");
200         }
201         else
202         {
203           printf(" 插入失败! \n");
204         }
205         getchar();
206         getchar();
207         break;
208       case 11:
209         // 删除元素
210         printf(" 输入要删除的元素位置: ");
211         scanf("%d", &i);
212         int de_v;
213         if (ListDelete(arr[num], i, de_v) == OK)
214         {
```

```
215         printf(" 删除成功! 删除的值为%d\n", de_v);
216       }
217       else
218       {
219         printf(" 删除失败! \n");
220       }
221       getchar();
222       getchar();
223       break;
224     case 12:
225       // 打印元素
226       if (!ListTraverse(arr[num]))
227         printf(" 线性表是空表! \n");
228       getchar();
229       getchar();
230       break;
231     case 13:
232       // 逆置线性表
233       statuss = reverseList(arr[num]);
234       if (statuss == OK)
235       {
236         printf(" 已逆置\n");
237       }
238       else
239       {
240         printf(" 线性表不存在\n");
241       }
242       getchar();
243       getchar();
244       break;
245     case 14:
```

```
246         // 删除倒数第 n 个元素
247         printf(" 输入要删除的元素序号: ");
248         scanf("%d", &n);
249         statuss = RemoveNthFromEnd(arr[num], n);
250         if (statuss == ERROR)
251         {
252           printf(" 位置序号错误\n");
253         }
254         else if (statuss == INFEASIBLE)
255         {
256           printf(" 线性表不存在\n");
257         }
258         else
259         {
260           printf(" 元素已删除\n");
261         }
262         getchar();
263         getchar();
264         break;
265       case 15:
266         // 线性表排序
267         statuss = sortList(arr[num]);
268         if (statuss == OK)
269         {
270           printf(" 排序完成\n");
271         }
272         else
273         {
274           printf(" 线性表不存在\n");
275         }
276         getchar();
```

```
277         getchar();
278         break;
279       case 16:
280         // 保存到文件
281         printf(" 输入要保存的文件名: ");
282         scanf("%s", filename);
283         statuss = SaveList(arr[num], filename);
284         if (statuss == OK)
285         {
286           printf(" 已存入\n");
287         }
288         else
289         {
290           printf(" 线性表不存在\n");
291         }
292         getchar();
293         getchar();
294         break;
295       case 17:
296         // 从文件中读取
297         printf(" 输入你要读取的文件名: ");
298         scanf("%s", filename);
299         statuss = LoadList(arr[num], filename);
300         if (statuss == OK)
301         {
302           printf(" 读取成功\n");
303         }
304         else
305         {
306           printf(" 读取失败\n");
307         }
```

```
308         getchar();
309         getchar();
310         break;
311       case 18:
312         // 切换线性表
313         int i;
314         printf(" 输入要切换到的线性表的序号: [0-29]");
315         scanf("%d", &num);
316         getchar();
317         getchar();
318         break;
319       case 0:
320         break;
321       } // end of switch
322     }   // end of while
323     printf(" 欢迎下次再使用本系统! \n");
324     return 0;
325   } // end of main()
326
```

附录 C　基于二叉链表的二叉树实现

头文件 define.h :

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

typedef int status;
typedef int KeyType;
typedef struct {
  KeyType key;
  char others[20];
} TElemType;  // 二叉树结点类型定义

typedef struct BiTNode {  // 二叉链表结点的定义
  TElemType data;
  struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

// 函数声明

void visit(BiTree t);  // 用于遍历输出
void CreTree(BiTree &T, TElemType definition[], int tmp[]);  // 用于创建二叉树
status CreateBiTree(BiTree &T, TElemType definition[]);  // 用于创建二叉树
```

```
status ClearBiTree(BiTree &T);              // 用于销毁二叉树
int BiTreeDepth(BiTree T);                  // 用于求二叉树深度
BiTNode *LocateNode(BiTree T, KeyType e);  // 用于查找结点
status Assign(BiTree &T, KeyType e, TElemType value);  // 用于修改结点
BiTNode *GetSibling(BiTree T, KeyType e);  // 用于求结点的兄弟结点
status InsertNode(BiTree &T, KeyType e, int LR, TElemType c);  // 用于插入结点
BiTNode *Getfather(BiTree T, KeyType e);  // 用于求结点的父结点
status DeleteNode(BiTree &T, KeyType e);  // 用于删除结点
status PreOrderTraverse(BiTree T, void (*visit)(BiTree));  // 用于先序遍历
status InOrderTraverse(BiTree T, void (*visit)(BiTree));   // 用于中序遍历
status PostOrderTraverse(BiTree T, void (*visit)(BiTree));  // 用于后序遍历
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree));  // 用于层序遍历
status SaveBiTree(BiTree T, char FileName[]);   // 用于保存二叉树
status LoadBiTree(BiTree &T, char FileName[]);  // 用于加载二叉树
int MaxPathSum(BiTree T);  // 用于求二叉树中的最大路径和
BiTree LowestCommonAncestor(BiTree T, int e1,
                            int e2);  // 用于求二叉树中两个结点的最近公共祖先
int InvertTree(BiTree T);  // 用于求二叉树的镜像

```

函数定义文件 define.cpp :

```
#include "define.h"

// 根据带空枝的二叉树先根遍历序列 definition 构造一棵二叉树，将根节点指针赋值给 T 并
void CreTree(BiTree &T, TElemType definition[], int tmp[]) {
  int i = 0;
  for (i = 0; i < 50; i++) {  // 找到第一个未被访问的结点
    if (tmp[i] == 0) {
      break;
    }
  }
  tmp[i] = 1;
```

```
12    if (strcmp(definition[i].others, "null")) {  // 如果不是空结点
13      T = (BiTree)malloc(sizeof(BiTNode));
14      memcpy(&T->data, &definition[i], sizeof(TElemType));  // 复制结点数据
15      CreTree(T->lchild, definition, tmp);  // 递归创建左子树
16      CreTree(T->rchild, definition, tmp);  // 递归创建右子树
17    } else {
18      T = NULL;
19    }
20  }
21  status CreateBiTree(BiTree &T, TElemType definition[]) {
22    if (T) {
23      return ERROR;
24    }
25    int tmp[50];
26    int i = 0;
27    while (definition[i].key != -1) {  // 检查是否有重复的 key
28      if (i == 0) {
29        tmp[0] = definition[i].key;
30      } else {
31        for (int j = 0; j < i; j++) {
32          if (definition[i].key && tmp[j] == definition[i].key) {
33            return ERROR;  // 如果有重复的 key, 返回 ERROR
34          }
35        }
36        tmp[i] = definition[i].key;
37      }
38      i++;
39    }
40    int tmp2[50] = {0};
41    CreTree(T, definition, tmp2);  // 创建二叉树
42    return OK;
```

```
43 }
44
45 // 将二叉树设置成空，并删除所有结点，释放结点空间
46 status ClearBiTree(BiTree &T) {
47   if (T) {
48     ClearBiTree(T->lchild);  // 递归删除左子树
49     ClearBiTree(T->rchild);  // 递归删除右子树
50     free(T);
51     T = NULL;
52   } else {
53   }
54   return OK;
55 }
56
57 // 求二叉树 T 的深度
58 int BiTreeDepth(BiTree T) {
59   if (T) {
60     int left = BiTreeDepth(T->lchild);    // 递归求左子树
61     int right = BiTreeDepth(T->rchild);  //递归求右子树
62     if (left > right) {
63       return left + 1;
64     } else {
65       return right + 1;
66     }
67   } else {
68     return 0;
69   }
70 }
71
72 // 查找结点
73 BiTNode *LocateNode(BiTree T, KeyType e) {
```

```
74    if (T) {
75      if (T->data.key == e) {
76        return T;
77      }
78      BiTree tmp;
79      if ((tmp = LocateNode(T->lchild, e))) {  //递归在左子树中查找
80        return tmp;
81      } else if ((tmp = LocateNode(T->rchild, e))) {  //递归在右子树中查找
82        return tmp;
83      }
84    } else {
85      return NULL;
86    }
87    return NULL;
88  }
89
90  // 实现结点赋值
91  status Assign(BiTree &T, KeyType e, TElemType value) {
92    BiTree tmp;
93    BiTree tmp2;
94    tmp = LocateNode(T, e);
95    tmp2 = LocateNode(T, value.key);
96    if (tmp2 && tmp2 != tmp) {  //说明没找到
97      return ERROR;
98    }
99    if (tmp) {  //找到，改写
100     memcpy(&tmp->data, &value, sizeof(TElemType));
101     return OK;
102   } else {
103     return ERROR;
104   }
```

```
105 }
106
107 // 实现获得兄弟结点
108 BiTNode *GetSibling(BiTree T, KeyType e) {
109   if (T) {
110     if (T->data.key == e) {
111       return NULL;
112     }
113     if (T->lchild && T->rchild) {  //两个孩子都存在，此处为单独判断根节点
114       if (T->lchild->data.key == e) {
115         return T->rchild;
116       }
117       if (T->rchild->data.key == e) {
118         return T->lchild;
119       }
120     }
121     BiTree tmp = GetSibling(T->lchild, e);  // 判断完成，递归寻找左子树和右子树
122     if (tmp) {
123       return tmp;
124     } else {
125       tmp = GetSibling(T->rchild, e);
126       return tmp;
127     }
128   } else {
129     return NULL;
130   }
131 }
132
133 // 插入结点
134 status InsertNode(BiTree &T, KeyType e, int LR, TElemType c) {
135   BiTree tmp;
```

```
136   BiTree tmp3 = LocateNode(T, c.key);
137   tmp = LocateNode(T, e);
138   if (tmp3) {
139     return ERROR;
140   }
141   if (LR == -1) {
142     tmp = (BiTree)malloc(sizeof(BiTNode));
143     memcpy(&tmp->data, &c, sizeof(TElemType));
144     tmp->rchild = T;
145     T = tmp;
146     T->lchild = NULL;
147     return OK;
148   }  // 以上为处理根节点
149   if (tmp == NULL) {
150     return ERROR;
151   }
152   BiTree tmp2;  //非根节点，按要求处理
153   if (LR == 1) {
154     tmp2 = (BiTree)malloc(sizeof(BiTNode));
155     memcpy(&tmp2->data, &c, sizeof(TElemType));
156     tmp2->rchild = tmp->rchild;
157     tmp2->lchild = NULL;
158     tmp->rchild = tmp2;
159     return OK;
160   } else if (LR == 0) {
161     tmp2 = (BiTree)malloc(sizeof(BiTNode));
162     memcpy(&tmp2->data, &c, sizeof(TElemType));
163     tmp2->rchild = tmp->lchild;
164     tmp2->lchild = NULL;
165     tmp->lchild = tmp2;
166     return OK;
```

```
167   }
168   return OK;
169 }
170
171 // 删除节点
172 BiTNode *Getfather(BiTree T, KeyType e) {  //删除节点需要先找到对应节点的父节点，所以
173   if (T) {
174     if (T->data.key == e) {  //根节点
175       return NULL;
176     }
177     if (T->lchild && T->rchild) {
178       if (T->lchild->data.key == e) {
179         return T;
180       }
181       if (T->rchild->data.key == e) {
182         return T;
183       }
184     } else if (T->lchild) {
185       if (T->lchild->data.key == e) {
186         return T;
187       }
188     } else if (T->rchild) {
189       if (T->rchild->data.key == e) {
190         return T;
191       }
192     }  // 以上为特殊情况
193     //一下不特殊，直接遍历去找
194     BiTree tmp = Getfather(T->lchild, e);
195     if (tmp) {
196       return tmp;
197     } else {
```

```
198         tmp = Getfather(T->rchild, e);
199         return tmp;
200       }
201     } else {
202       return NULL;
203     }
204 }
205 status DeleteNode(BiTree &T, KeyType e) {
206   BiTree tmp2;
207   if (T->data.key == e) {  //删除根节点
208     if (T->lchild && T->rchild) {
209       tmp2 = T->lchild;
210       while (tmp2 && tmp2->rchild) {  //找到左子树的最右节点
211         tmp2 = tmp2->rchild;
212       }
213       tmp2->rchild = T->rchild;
214       tmp2 = T->lchild;
215       free(T);
216       T = tmp2;
217       return OK;
218 //左右子树都存在，将左子树的最右节点的右孩子指向右子树，然后将左子树作为根节点
219     } else if (T->lchild) {
220       tmp2 = T->lchild;
221       free(T);
222       T = tmp2;
223     } else {
224       tmp2 = T->rchild;
225       free(T);
226       T = tmp2;
227     }
228   }
```

```
229   BiTree tmp = Getfather(T, e);  //找到父节点
230   if (tmp) {
231     if (tmp->lchild && tmp->lchild->data.key == e) {  //判断是左孩子还是右孩子
232       if (tmp->lchild->lchild) {  //左子树存在
233         if (tmp->lchild->rchild) {  //左右子树都存在
234           tmp2 = tmp->lchild->lchild;
235           while (tmp2->rchild) {
236             tmp2 = tmp2->rchild;
237           }
238           tmp2->rchild = tmp->lchild->rchild;
239           tmp2 = tmp->lchild->lchild;
240           free(tmp->lchild);
241           tmp->lchild = tmp2;
242           return OK;
243         } else {  //左子树存在，右子树不存在
244           tmp2 = tmp->lchild->lchild;
245           free(tmp->lchild);
246           tmp->lchild = tmp2;
247           return OK;
248         }
249       } else {
250         if (tmp->lchild->rchild) {  //左子树不存在，右子树存在
251           tmp2 = tmp->lchild->rchild;
252           free(tmp->lchild);
253           tmp->lchild = tmp2;
254           return OK;
255         } else {  //左右子树都不存在
256           free(tmp->lchild);
257           tmp->lchild = NULL;
258           return OK;
259         }
```

```
260       }
261
262     } else {
263       if (tmp->rchild->lchild) {  //判断是左孩子还是右孩子
264         if (tmp->rchild->rchild) {  //左右子树都存在
265           tmp2 = tmp->rchild->lchild;
266           while (tmp2->rchild) {
267             tmp2 = tmp2->rchild;
268           }
269           tmp2->rchild = tmp->rchild->rchild;
270           tmp2 = tmp->rchild->lchild;
271           free(tmp->rchild);
272           tmp->rchild = tmp2;
273           return OK;
274         } else {  //只有左子树存在
275           tmp2 = tmp->rchild->lchild;
276           free(tmp->rchild);
277           tmp->rchild = tmp2;
278           return OK;
279         }
280       } else {  //左子树不存在
281         if (tmp->rchild->rchild) {  //右子树存在
282           tmp2 = tmp->rchild->rchild;
283           free(tmp->rchild);
284           tmp->rchild = tmp2;
285           return OK;
286         } else {  //左右子树都不存在
287           free(tmp->rchild);
288           tmp->rchild = NULL;
289           return OK;
290         }
```

```
291         }
292       }
293
294     } else {  //没有找到父节点，说明没有这个节点
295       return ERROR;
296     }
297   }
298
299   void visit(BiTree t) { printf("%d %s\n", t->data.key, t->data.others); }
300
301   // 先序遍历二叉树 T
302   status PreOrderTraverse(BiTree T, void (*visit)(BiTree)) {
303     if (T) {
304       visit(T);
305       PreOrderTraverse(T->lchild, visit);
306       PreOrderTraverse(T->rchild, visit);
307     }
308     return OK;
309   }
310
311   // 中序遍历二叉树 T
312   status InOrderTraverse(BiTree T, void (*visit)(BiTree)) {
313     if (T) {
314       InOrderTraverse(T->lchild, visit);
315       visit(T);
316       InOrderTraverse(T->rchild, visit);
317     }
318     return OK;
319   }
320
321   // 后序遍历二叉树 T
```

```
322 status PostOrderTraverse(BiTree T, void (*visit)(BiTree)) {
323   if (T) {
324     PostOrderTraverse(T->lchild, visit);
325     PostOrderTraverse(T->rchild, visit);
326     visit(T);
327   }
328   return OK;
329 }
330
331 // 按层遍历二叉树 T
332 status LevelOrderTraverse(BiTree T, void (*visit)(BiTree)) {
333   int begin, end;
334   begin = end = 0;
335   BiTree arr[50];
336   if (T == NULL) {  //如果树为空，返回错误
337     return ERROR;
338   }
339   arr[end++] = T;
340   while (begin != end) {  //如果 begin 等于 end，说明队列为空，结束循环
341     visit(arr[begin]);
342     if (arr[begin]->lchild) {
343       arr[end++] = arr[begin]->lchild;
344     }
345     if (arr[begin]->rchild) {
346       arr[end++] = arr[begin]->rchild;
347     }
348     begin++;
349   }
350   return OK;
351 }
352
```

```
353 // 将二叉树的结点数据写入到文件 FileName 中
354 status SaveBiTree(BiTree T, char FileName[]) {
355   FILE *fd = fopen(FileName, "w");
356   if (T) {
357     BiTree p = T;
358     BiTree stack[50];  //定义一个栈
359     int num = 0;
360     while (NULL != p || num > 0) {  //如果 p 不为空或者栈不为空，循环
361       while (NULL != p) {
362         fprintf(fd, "%d %s\n", p->data.key, p->data.others);  //写入文件
363         stack[num++] = p;
364         p = p->lchild;
365       }
366       if (p == NULL) {   // 特殊情况，写入一个 null
367         fprintf(fd, "%d %s\n", 0, "null");
368       }
369       num--;
370       p = stack[num];
371       p = p->rchild;
372     }
373     fprintf(fd, "%d %s\n", 0, "null");
374     fclose(fd);
375     return OK;
376   } else {
377     return ERROR;
378   }
379 }
380 // 读入文件 FileName 的结点数据，创建二叉树
381 status LoadBiTree(BiTree &T, char FileName[]) {
382   FILE *fd = fopen(FileName, "r");
383   BiTree stack[50];  // 也是需要一个栈
```

```
384   int top = 0;
385   int key;
386   char others[20];
387   if (T == NULL) {
388     BiTree tmp;
389     BiTree node;
390     T = (BiTree)malloc(sizeof(BiTNode));
391     T->lchild = NULL;
392     T->rchild = NULL;
393     fscanf(fd, "%d %s", &key, others);
394     T->data.key = key;
395     strcpy(T->data.others, others);
396     stack[top] = T;
397     tmp = T;
398     int lr = 0;
399     while (top >= 0) {  // 栈不空，依次读取，并写入栈，
400       if (fscanf(fd, "%d %s", &key, others) == EOF) {
401         break;
402       }
403       if (strcmp(others, "null") == 0) {
404         if (lr == 0) {
405           tmp->lchild = NULL;
406           lr = 1;
407         } else {
408           if (tmp->rchild) {
409           } else {
410             tmp->rchild = NULL;
411           }
412           top--;
413           if (top < 0) {
414             break;
```

```
415             }
416             tmp = stack[top];
417             lr = 1;
418           }
419
420         } else {
421           node = (BiTree)malloc(sizeof(BiTNode));
422           node->data.key = key;
423           strcpy(node->data.others, others);
424           node->lchild = NULL;
425           node->rchild = NULL;
426           if (lr == 0) {
427             tmp->lchild = node;
428             stack[++top] = tmp->lchild;
429             tmp = tmp->lchild;
430           } else {
431             tmp->rchild = node;
432             stack[++top] = tmp->rchild;
433             tmp = tmp->rchild;
434             lr = 0;
435           }
436         }
437       }
438       return OK;
439     } else {
440       return ERROR;
441     }
442 }
443
444 // 二叉树从根节点开始的最大路径和
445 int MaxPathSum(BiTree T) {
```

```
446   int max = 0;
447   int tmp = 0;
448   if (T) {
449     tmp = T->data.key;
450     max = tmp;
451     if (T->lchild) {
452       tmp += MaxPathSum(T->lchild);  //递归寻找
453     }
454     if (T->rchild) {
455       max += MaxPathSum(T->rchild);
456     }
457     if (tmp > max) {
458       max = tmp;
459     }
460   }
461   return max;
462 }
463
464 // 两节点的最近公共祖先
465 BiTree LowestCommonAncestor(BiTree T, int e1, int e2) {
466   if (T) {
467     if (T->data.key == e1 || T->data.key == e2) {
468       return T;
469     }    // 也是用到递归
470     BiTree l = LowestCommonAncestor(T->lchild, e1, e2);
471     BiTree r = LowestCommonAncestor(T->rchild, e1, e2);
472     if (l && r) {
473       return T;
474     } else if (l) {
475       return l;
476     } else if (r) {
```

```
477       return r;
478     } else {
479       return NULL;
480     }
481   } else {
482     return NULL;
483   }
484 }
485
486 // 翻转二叉树，互换所有左右节点
487 int InvertTree(BiTree T) {
488   if (T) {
489     BiTree tmp = T->lchild;
490     T->lchild = T->rchild;
491     T->rchild = tmp;
492     InvertTree(T->lchild);
493     InvertTree(T->rchild);
494   }
495   return OK;
496 }
497
```

主要流程文件 main.cpp :

```
1 #include "define.h"
2
3 // main
4 int main(void) {
5   BiTree list[30] = {NULL};
6   TElemType arr[30];
7   int num = 0;
8   int op = 1;
9   int len;
```

```
10    int i;
11    int count;
12    BiTree tmp;
13    char filename[20];
14    TElemType ttt;
15    int statuss;
16    while (op) {
17      // system("cls");
18      printf("\n\n");
19      // 打印菜单
20      printf("      Menu for Linear Table On List Structure \n");
21      printf("-------------------------------------------------\n");
22      printf("    ^^I  1. 初始化                          7. 更改节点值\n");
23      printf("    ^^I  2. 摧毁                            8. 获取兄弟节点\n");
24      printf("    ^^I  3. 清空                            9. 插入元素\n");
25      printf("    ^^I  4. 判断是否为空                    10. 删除元素\n");
26      printf("    ^^I  5. 求深度                          11. 先序遍历\n");
27      printf("    ^^I  6. 获取元素                        12. 中序遍历\n");
28      printf("    ^^I  13. 后序遍历                       14. 层序遍历\n");
29      printf("    ^^I  15. 从根节点开始的最大路径          16. 保存到文件\n");
30      printf("    ^^I  17. 从文件读取                     18. 切换二叉树\n");
31      printf("    ^^I  19. 交换左右子树                   20. 求最近公共祖先\n");
32      printf("    ^^I  0. 退出\n");
33      printf("-------------------------------------------------\n");
34      printf("    请选择你的操作 [0~20]:");
35      scanf("%d", &op);
36      switch (op) {
37        case 1:
38          // 创建二叉树
39          count = 0;
40          do{
```

```
41            scanf("%d %s", &arr[count].key, arr[count].others);
42          }
43          while (arr[count++].key != -1);
44          if (CreateBiTree(list[num], arr) == OK)
45            printf(" 二叉树创建成功! \n");
46          else
47            printf(" 二叉树已经存在或关键字相同\n");
48          getchar();
49          getchar();
50          break;
51        case 2:
52          // 销毁二叉树
53          if (list[num] == NULL) {
54            printf(" 二叉树不存在\n");
55            getchar();
56            getchar();
57            break;
58          }
59          if (ClearBiTree(list[num]) == OK) {
60            printf(" 二叉树销毁成功! \n");
61            list[num] = NULL;
62          }
63          getchar();
64          getchar();
65          break;
66        case 3:
67          // 清空二叉树
68          if (list[num] == NULL) {
69            printf(" 二叉树不存在\n");
70          } else {
71            ClearBiTree(list[num]->lchild);
```

```
72            ClearBiTree(list[num]->rchild);
73            list[num]->data.key = -1;
74            printf(" 已清空\n");
75          }
76          getchar();
77          getchar();
78          break;
79        case 4:
80          // 判断二叉树是否为空
81          if (list[num] == NULL) {
82            printf(" 二叉树不存在\n");
83          } else if (list[num]->data.key == -1 && list[num]->lchild == NULL &&
84                     list[num]->rchild == NULL) {
85            printf(" 二叉树为空\n");
86          } else {
87            printf(" 二叉树不为空\n");
88          }
89          getchar();
90          getchar();
91          break;
92        case 5:
93          // 求二叉树深度
94          if (list[num] == NULL) {
95            printf(" 二叉树不存在\n");
96            getchar();
97            getchar();
98            break;
99          }
100         if ((len = BiTreeDepth(list[num]))) {
101           printf(" 二叉树深度为%d\n", len);
102         }
```

```
103         getchar();
104         getchar();
105         break;
106       case 6:
107         // 获取元素
108         printf(" 输入要获取的元素: ");
109         scanf("%d", &i);
110         tmp = LocateNode(list[num], i);
111         if (tmp == NULL) {
112           printf(" 未找到\n");
113         } else {
114           printf(" 找到了, 结点标签为%s", tmp->data.others);
115         }
116         getchar();
117         getchar();
118         break;
119       case 7:
120         // 更改结点值
121         int elem;
122         printf(" 请输入想更改值的节点值: ");
123         scanf("%d", &elem);
124         printf(" 输入想更改成的值: ");
125         scanf("%d", &ttt.key);
126         printf(" 输入更改的值的标签: ");
127         scanf("%s", ttt.others);
128         statuss = Assign(list[num], elem, ttt);
129         if (statuss == ERROR) {
130           printf(" 未找到该元素\n");
131         } else {
132           printf(" 更改成功\n");
133         }
```

```
134         getchar();
135         getchar();
136         break;
137       case 8:
138         // 获取兄弟节点
139         int pre;
140         printf(" 请输入想获取兄弟节点的元素的值: ");
141         scanf("%d", &pre);
142         tmp = GetSibling(list[num], pre);
143         if (tmp == NULL) {
144           printf(" 未找到该元素\n");
145         } else {
146           printf(" 兄弟节点的值是: %d, 标签是%s\n", tmp->data.key,
147                  tmp->data.others);
148         }
149         getchar();
150         getchar();
151         break;
152       case 9:
153         // 插入节点
154         int next,LR;
155         printf(" 请输入想插入到的元素: ");
156         scanf("%d", &next);
157         printf(" 请输入插入方式 LR: ");
158         scanf("%d",&LR);
159         printf(" 输入想插入的值: ");
160         scanf("%d", &ttt.key);
161         printf(" 输入插入的值的标签: ");
162         scanf("%s",ttt.others);
163         statuss = InsertNode(list[num],next,LR,ttt);
164         if (statuss == ERROR) {
```

```
165           printf(" 插入失败\n");
166         } else {
167           printf(" 插入成功\n");
168         }
169         getchar();
170         getchar();
171         break;
172       case 10:
173         // 删除元素
174         printf(" 输入要删除的元素: ");
175         scanf("%d", &i);
176         statuss = DeleteNode(list[num], i);
177         if (statuss == OK) {
178           printf(" 删除成功! \n");
179         } else {
180           printf(" 删除失败! \n");
181         }
182         getchar();
183         getchar();
184         break;
185       case 11:
186         // 前序遍历
187         PreOrderTraverse(list[num],visit);
188         getchar();
189         getchar();
190         break;
191       case 12:
192         // 中序遍历
193         InOrderTraverse(list[num],visit);
194         getchar();
195         getchar();
```

```
196           break;
197         case 13:
198           // 后序遍历
199           PostOrderTraverse(list[num],visit);
200           getchar();
201           getchar();
202           break;
203         case 14:
204           // 层序遍历
205           LevelOrderTraverse(list[num],visit);
206           getchar();
207           getchar();
208           break;
209         case 15:
210           // 求最大路径和
211           len = MaxPathSum(list[num]);
212           printf("%d",len);
213           getchar();
214           getchar();
215           break;
216         case 16:
217           // 保存到文件
218           printf(" 输入要保存的文件名: ");
219           scanf("%s", filename);
220           statuss = SaveBiTree(list[num], filename);
221           if (statuss == OK) {
222             printf(" 已存入\n");
223           } else {
224             printf(" 二叉树不存在\n");
225           }
226           getchar();
```

```
227           getchar();
228           break;
229         case 17:
230           // 从文件中读取
231           printf(" 输入你要读取的文件名: ");
232           scanf("%s", filename);
233           statuss = LoadBiTree(list[num], filename);
234           if (statuss == OK) {
235             printf(" 读取成功\n");
236           } else {
237             printf(" 读取失败\n");
238           }
239           getchar();
240           getchar();
241           break;
242         case 18:
243           // 切换 BiTree
244           printf(" 输入要切换到的二叉树的序号: [0-29]");
245           scanf("%d", &num);
246           printf(" 已切换到第%d 个二叉树\n", num);
247           list[num] = NULL;
248           getchar();
249           getchar();
250           break;
251         case 19:
252           // 交换左右子树
253           statuss = InvertTree(list[num]);
254           if (statuss == OK) {
255             printf(" 交换成功\n");
256           } else {
257             printf(" 交换失败\n");
```

```
258         }
259         getchar();
260         getchar();
261         break;
262       case 20:
263         // 求两节点的最近公共祖先
264         int a, b;
265         printf(" 输入两个元素: ");
266         scanf("%d %d",&a,&b);
267         tmp = LowestCommonAncestor(list[num],a,b);
268         if(tmp == NULL){
269           printf(" 无公共祖先\n");
270         }else{
271           printf(" 公共祖先是%d, 标签是%s",tmp->data.key,tmp->data.others);
272         }
273         getchar();
274         getchar();
275         break;
276       case 0:
277         break;
278     }  // end of switch
279   }    // end of while
280   printf(" 欢迎下次再使用本系统! \n");
281   return 0;
282 }  // end of main()
283
```

附录 D　基于邻接表的图实现

头文件 define.h :

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

#include "stdlib.h"

#define TRUE 1              // 表示真值
#define FALSE 0             // 表示为假
#define OK 1                // 返回状态为已完成
#define ERROR 0             // 返回状态出错
#define INFEASIBLE -1       // 返回状态不合理
#define OVERFLOW -2         // 返回状态溢出
#define MAX_VERTEX_NUM 20   // 定义最大容量
typedef int status;         // 定义状态数据类型
typedef int KeyType;        // 定义关键字数据类型

typedef struct {
  KeyType key;      // 关键字
  char others[20];  // 结点名称
} VertexType;       // 顶点类型定义

typedef struct ArcNode {    // 表结点类型定义
  int adjvex;               // 顶点位置编号
  struct ArcNode* nextarc;  // 下一个表结点指针
} ArcNode;

typedef struct VNode {  // 头结点及其数组类型定义
  VertexType data;      // 顶点信息
  ArcNode* firstarc;    // 指向第一条弧
```

```
29 } VNode, AdjList[MAX_VERTEX_NUM];
30
31 typedef struct {       // 邻接表的类型定义
32   AdjList vertices;    // 头结点数组
33   int vexnum, arcnum;  // 顶点数、弧数
34 } ALGraph;
35
36 typedef struct QNode {  // 结点的队列
37   VertexType data;       // 结点数据
38   struct QNode* next;    // 指向下个结点的指针
39 } QNode, *Queue;
40
41 typedef struct {
42   Queue front;  // 队列队头
43   Queue rear;   // 队列队尾
44 } Linkqueue;    // 结点队列
45
46 typedef struct {  // 多图的管理表定义
47   struct {
48     char name[30];  // 单图的名称
49     ALGraph G;      // 单图
50   } elem[10];       // 10 个多图
51   int length;       // 定义当前多图的数量
52 } LISTS;            // 多图的定义
53
54 void print1(void);                              // 输出函数 1
55 void print2(void);                              // 输出函数 2
56 void print3(void);                              // 输出函数 3
57 status JudgeV(VertexType V[], ALGraph& G);  // 查找 V 是否合法
58 status JudgeVR(VertexType V[], KeyType VR[][2], ALGraph& G);  // 查找 VR 是否合法
59 status CreateCraph(ALGraph& G, VertexType V[], KeyType VR[][2]);  // 创建无向图
```

```
status DestroyGraph(ALGraph& G);      // 销毁无向图
int LocateVex(ALGraph G, KeyType u);  // 查找顶点
int Compare(ALGraph G, KeyType u,
            VertexType value);  // 赋值操作辅助函数比较关键字唯一性
status PutVex(ALGraph& G, KeyType u, VertexType value);  // 顶点赋值
int FirstAdjVex(ALGraph G, KeyType u);                // 获得第一邻接点
int NextAdjVex(ALGraph G, KeyType v, KeyType w);      // 获得下一邻接点
status InsertVex(ALGraph& G, VertexType v);           // 插入顶点
status DeleteVex(ALGraph& G, KeyType v);              // 删除顶点
status InsertArc(ALGraph& G, KeyType v, KeyType w);   // 插入弧
status DeleteArc(ALGraph& G, KeyType v, KeyType w);   // 删除弧
void DFS(ALGraph G, int v, void (*visit)(VertexType),
         int visited[]);  // 深度遍历辅助函数
status DFSTraverse(ALGraph G, void (*visit)(VertexType));  // 深度遍历
status BFSTraverse(ALGraph G, void (*visit)(VertexType));  // 广度遍历
void visit(VertexType v);                                  // 输出函数
status SaveGraph(ALGraph G, char FileName[]);              // 保存文件
status LoadGraph(ALGraph& G, char FileName[]);             // 读取文件
status RemoveList(LISTS& Lists, char ListName[]);          // 多图移除
status LocateList(LISTS Lists, char ListName[]);           // 多图查找
status AddList(LISTS& Lists, char ListName[]);             // 多图表名创建
status MoreSaveGraph(ALGraph G, FILE* fp);                 // 多图文件保存
status MoreLoadGraph(ALGraph& G, FILE* fp);                // 多图文件读取
status InitQueue(Linkqueue& Q);                            // 创建图的队列
status QueueEmpty(Linkqueue Q);  // 判断图的队列是否为空
status enQueue(Linkqueue& Q, VertexType value);  // 将为 value 的结点入队
status deQueue(
    Linkqueue& Q,
    VertexType& value);  // 将结点出队，并将出队的结点数据保存在 value 中
status ConnectedComponentsNums(ALGraph G);  //
status ShortestPathLength(ALGraph G, KeyType v,
```

```
                        KeyType w);  // 两个顶点的最短路径
void VerticesSetLessThank(ALGraph G, KeyType v, KeyType k);

ALGraph G;    // 创建单图
LISTS Lists;  // 创建多图

int op;             // 操作序号
int number;         // 储存查找图的序号
char saveName[30];  // 暂存图名
int distance[21];   // 储存最短路径长度
int visited[100];   // 记录深搜时某个结点是否搜过
bool mark[21];      // 记录宽搜时某个结点是否搜过

/* 函数 */
void print1(void) {
  printf(
      "\n-----------------单图操作菜单--------------------\n\
^^I1. 创建图          2. 销毁图            3. 查找顶点\n\
^^I4. 顶点赋值        5. 获得第一邻接顶点  6. 获得下一邻接点\n\
^^I7. 插入顶点        8. 删除顶点          9. 插入弧\n\
^^I10. 删除弧         11. 深度优先搜索遍历 12. 广度优先搜索遍历\n\
^^I13. 文件保存       14. 文件读取         15. 距离小于 k 的顶点集合\n\
^^I16. 顶点间最短路径和长度  17. 连通分量     0. 停止操作\n\
^^I 请输入【0~17】\n\n请输入你需要的操作: ");
}
void print2(void) {
  printf(
      "\n-----------------多图操作中的单图操作菜单--------------------\n\
^^I1. 创建图          2. 销毁图            3. 查找顶点\n\
^^I4. 顶点赋值        5. 获得第一邻接顶点  6. 获得下一邻接点\n\
^^I7. 插入顶点        8. 删除顶点          9. 插入弧\n\
```

```
^^I10. 删除弧          11. 深度优先搜索遍历  12. 广度优先搜索遍历\n\
^^I13. 文件保存        14. 文件读取          15. 距离小于 k 的顶点集合\n\
^^I16. 顶点间最短路径和长度  17. 连通分量     18. 返回多图菜单\n\
^^I0. 停止操作\n\
^^I 请输入【0~18】\n\n请输入你需要的操作: ");
}
void print3(void) {
  printf(
      "\n-----------------多图操作菜单--------------------\n\
^^I1. 创建多图        2. 移除某图        3. 查找某图\n\
^^I4. 单独操作        5. 遍历多图        6. 清空多图\n\
^^I0. 停止操作        请输入【0~6】\n\n请输入你需要的操作: ");
}

status JudgeV(VertexType V[], ALGraph& G) {
  int i, j;
  for (i = 0; V[i].key != -1; i++) {
    for (j = i + 1; V[j].key != -1; j++) {
      if (V[i].key == V[j].key) return ERROR;
    }
  }
  if (i > 20)
    return ERROR;
  else {
    G.vexnum = i;
    return OK;
  }
}
// 比较 V 的合法性, 不合法返回 ERROR;

status JudgeVR(VertexType V[], KeyType VR[][2], ALGraph& G) {
```

```
153   int i, j, m, n = 0;
154   for (i = 0, j = 0; VR[i][j] != -1; i++) {
155     for (m = 0; V[m].key != -1; m++) {
156       if (VR[i][j] == V[m].key) n++;
157       if (VR[i][j + 1] == V[m].key) n++;
158     }
159     if (n != 2) return ERROR;
160     n = 0, j = 0;
161   }
162   G.arcnum = i;
163   return OK;
164 }
165 // 比较 VR 的合法性，不合法返回 ERROR;
166
167 status CreateCraph(ALGraph& G, VertexType V[], KeyType VR[][2]) {
168   G.vexnum = 0, G.arcnum = 0;
169   if (JudgeV(V, G) == ERROR)
170     return ERROR;
171   else {
172     int i;
173     for (i = 0; i < G.vexnum; i++) {
174       G.vertices[i].data.key = V[i].key;
175       strcpy(G.vertices[i].data.others, V[i].others);
176       G.vertices[i].firstarc = NULL;
177     }
178     if (JudgeVR(V, VR, G) == ERROR) {
179       G.vexnum = 0, G.arcnum = 0;
180       return ERROR;
181     }
182
183     int j;
```

```
184     for (j = 0; j < G.arcnum; j++) {
185       ArcNode* p = (ArcNode*)malloc(sizeof(ArcNode));
186       p->adjvex = LocateVex(G, VR[j][1]);
187       p->nextarc = G.vertices[LocateVex(G, VR[j][0])].firstarc;
188       G.vertices[LocateVex(G, VR[j][0])].firstarc = p;
189 
190       ArcNode* q = (ArcNode*)malloc(sizeof(ArcNode));
191       q->adjvex = LocateVex(G, VR[j][0]);
192       q->nextarc = G.vertices[LocateVex(G, VR[j][1])].firstarc;
193       G.vertices[LocateVex(G, VR[j][1])].firstarc = q;
194     }
195   }
196   return OK;
197 }
198 // 根据 V 和 VR 构造图 T 并返回 OK，如果 V 和 VR 不正确，返回 ERROR，如果有相同的关键
199 
200 void visit(VertexType v) { printf(" %d %s", v.key, v.others); }
201 // 输出函数
202 
203 status DestroyGraph(ALGraph& G) {
204   int i;
205   ArcNode *p, *q;
206   for (i = 0; i < G.vexnum; i++) {
207     p = G.vertices[i].firstarc;
208     while (p) {
209       q = p->nextarc;
210       free(p);
211       p = q;
212     }
213   }
214   G.vexnum = 0;
```

```
215   G.arcnum = 0;
216   return OK;
217 }
218 // 销毁图
219
220 int LocateVex(ALGraph G, KeyType u) {
221   int i;
222   for (i = 0; i < G.vexnum; i++) {
223     if (u == G.vertices[i].data.key) return i;
224   }
225   return -1;
226 }
227 // 根据 u 在图 G 中查找顶点，查找成功返回位序，否则返回-1
228
229 int Compare(ALGraph G, KeyType u, VertexType value) {
230   int i, NOu = -1, NOvalue = -1;
231   for (i = 0; i < G.vexnum; i++) {
232     if (G.vertices[i].data.key == u) NOu = i;
233     if (G.vertices[i].data.key == value.key) NOvalue = i;
234   }
235   if (NOu == -1) return -1;
236   if (NOvalue != -1 && NOu != NOvalue) return -1;
237   return NOu;
238 }
239 // 比较赋值操作是否违反关键字唯一性
240
241 status PutVex(ALGraph& G, KeyType u, VertexType value) {
242   int n;
243   if ((n = Compare(G, u, value)) == -1) return ERROR;
244   G.vertices[n].data.key = value.key;
245   strcpy(G.vertices[n].data.others, value.others);
```

```
246   return OK;
247 }
248 // 根据 u 在图 G 中查找顶点，查找成功将该顶点值修改成 value，返回 OK；如果查找失败或
249
250 int FirstAdjVex(ALGraph G, KeyType u) {
251   int n;
252   if ((n = LocateVex(G, u)) == -1) return -1;  // 查找失败返回-1
253   if (G.vertices[n].firstarc == NULL)
254     return -2;  // 如果查找成功但没有第一邻接点则返回-2
255   return G.vertices[n].firstarc->adjvex;
256 }
257 // 根据 u 在图 G 中查找顶点，查找成功返回顶点 u 的第一邻接顶点位序，否则返回-1；
258
259 int NextAdjVex(ALGraph G, KeyType v, KeyType w) {
260   int iv, iw;
261   if ((iv = LocateVex(G, v)) == -1) return -1;
262   if ((iw = LocateVex(G, w)) == -1) return -1;
263
264   ArcNode* p = G.vertices[iv].firstarc;
265   while (p) {
266     if (p->adjvex == iw && p->nextarc != NULL) return p->nextarc->adjvex;
267     p = p->nextarc;
268   }
269   return -1;
270 }
271 // 根据 u 在图 G 中查找顶点，查找成功返回顶点 v 的邻接顶点相对于 w 的下一邻接顶点的位
272
273 status InsertVex(ALGraph& G, VertexType v) {
274   if (LocateVex(G, v.key) != -1) return ERROR;
275   if (G.vexnum == MAX_VERTEX_NUM) return ERROR;
276   G.vertices[G.vexnum].data.key = v.key;
```

```
277   strcpy(G.vertices[G.vexnum].data.others, v.others);
278   G.vertices[G.vexnum].firstarc = NULL;
279   G.vexnum++;
280   return OK;
281 }
282 // 在图 G 中插入顶点 v，成功返回 OK，否则返回 ERROR
283
284 status DeleteVex(ALGraph& G, KeyType v) {
285   int n, i;
286   if ((n = LocateVex(G, v)) == -1 || G.vexnum == 1 || G.vexnum == 0)
287     return ERROR;
288   ArcNode *p = G.vertices[n].firstarc, *q, *save, *delt;
289   while (p) {
290     q = p->nextarc;
291     save = G.vertices[p->adjvex].firstarc;
292     if (save->adjvex == n)
293       G.vertices[p->adjvex].firstarc = save->nextarc;
294     else {
295       while (save->nextarc) {  // 删除其他结点的相关边
296         if (save->nextarc->adjvex == n) {
297           delt = save->nextarc;
298           save->nextarc = save->nextarc->nextarc;
299           free(delt);
300           break;
301         }
302         save = save->nextarc;
303       }
304     }
305     free(p);
306     p = q;
307   }
```

```
308   for (i = n + 1; i < G.vexnum; i++) {
309     G.vertices[i - 1] = G.vertices[i];
310   }
311   G.vexnum--;
312   for (i = 0; i < G.vexnum; i++) {
313     p = G.vertices[i].firstarc;
314     while (p) {
315       if (p->adjvex >= n) p->adjvex--;
316       p = p->nextarc;
317     }
318   }
319   return OK;
320 }
321 // 在图 G 中删除关键字 v 对应的顶点以及相关的弧，成功返回 OK，否则返回 ERROR
322
323 status InsertArc(ALGraph& G, KeyType v, KeyType w) {
324   int i, j;
325   ArcNode *p, *q;
326   if ((i = LocateVex(G, v)) == -1 || (j = LocateVex(G, w)) == -1) return ERROR;
327   p = G.vertices[i].firstarc;
328
329   while (p) {
330     if (p->adjvex == j) {
331       return ERROR;
332     }
333     p = p->nextarc;
334   }
335   p = (ArcNode*)malloc(sizeof(ArcNode));
336   q = (ArcNode*)malloc(sizeof(ArcNode));
337   p->adjvex = i;
338   q->adjvex = j;
```

```
339   p->nextarc = G.vertices[j].firstarc;
340   G.vertices[j].firstarc = p;
341   q->nextarc = G.vertices[i].firstarc;
342   G.vertices[i].firstarc = q;
343
344   return OK;
345 }
346 // 在图 G 中增加弧 <v,w>, 成功返回 OK, 否则返回 ERROR
347
348 status DeleteArc(ALGraph& G, KeyType v, KeyType w) {
349   int i, j, n = 0;
350   ArcNode *p, *q;
351   if ((i = LocateVex(G, v)) == -1 || (j = LocateVex(G, w)) == -1) return ERROR;
352   p = G.vertices[i].firstarc;
353
354   if (p->adjvex == j) {
355     G.vertices[i].firstarc = p->nextarc;
356     free(p);
357   } else {
358     while (p->nextarc) {
359       if (p->nextarc->adjvex == j) {
360         q = p->nextarc;
361         p->nextarc = p->nextarc->nextarc;
362         free(q);
363         n++;
364         break;
365       }
366       p = p->nextarc;
367     }
368     if (n == 0) return ERROR;
369   }
```

```
370
371   q = G.vertices[j].firstarc;
372   if (q->adjvex == i) {
373     G.vertices[j].firstarc = q->nextarc;
374     free(q);
375   } else {
376     while (q->nextarc) {
377       if (q->nextarc->adjvex == i) {
378         p = q->nextarc;
379         q->nextarc = q->nextarc->nextarc;
380         free(p);
381         break;
382       }
383       q = q->nextarc;
384     }
385   }
386   return OK;
387 }
388 // 在图 G 中删除弧 <v,w>, 成功返回 OK, 否则返回 ERROR
389
390 void DFS(ALGraph G, int v, void (*visit)(VertexType), int visited[]) {
391   int w;
392   visited[v] = 1;
393   visit(G.vertices[v].data);
394   for (w = FirstAdjVex(G, G.vertices[v].data.key); w >= 0;
395        w = NextAdjVex(G, G.vertices[v].data.key, G.vertices[w].data.key))
396     if (!visited[w])  // 处理所有未访问的邻接顶点
397       DFS(G, w, visit, visited);
398 }
399
400 status DFSTraverse(ALGraph G, void (*visit)(VertexType)) {
```

```
401   int v;
402   for (v = 0; v < G.vexnum; v++)  // 初始化各顶点未访问状态
403     visited[v] = 0;
404   for (v = 0; v < G.vexnum; v++)
405     if (!visited[v])  // 从一个未访问的顶点开始
406       DFS(G, v, visit, visited);
407   return OK;
408 }
409 // 对图 G 进行深度优先搜索遍历，依次对图中的每一个顶点使用函数 visit 访问一次，且仅访
410
411 status BFSTraverse(ALGraph G, void (*visit)(VertexType)) {
412   int v, w, u;
413   for (v = 0; v < G.vexnum; v++) visited[v] = 0;
414   int Que[100], in = 0, out = 0;
415
416   for (v = 0; v < G.vexnum; v++)  // 按顶点位置序号依次选择顶点
417     if (!visited[v]) {            // 遇到未访问过的顶点开始遍历
418       visited[v] = 1;
419       visit(G.vertices[v].data);
420       Que[in] = v;
421       in++;
422       while (in != out) {
423         u = Que[out];
424         out++;
425         for (w = FirstAdjVex(G, G.vertices[u].data.key); w >= 0;
426              w = NextAdjVex(G, G.vertices[u].data.key, G.vertices[w].data.key))
427           if (!visited[w]) {
428             visited[w] = 1;
429             visit(G.vertices[w].data);
430             Que[in] = w;
431             in++;
```

```
432             }
433         }
434       }
435   return OK;
436 }
437 // 对图 G 进行广度优先搜索遍历，依次对图中的每一个顶点使用函数 visit 访问一次，且仅访
438
439 status SaveGraph(ALGraph G, char FileName[]) {
440   FILE* fp = fopen(FileName, "w");
441   int i, j;
442   for (i = 0; i < G.vexnum; i++) {
443     fprintf(fp, "%d %s ", G.vertices[i].data.key, G.vertices[i].data.others);
444   }
445   fprintf(fp, "-1 nil ");
446   int VR[20][20];
447
448   for (i = 0; i < G.vexnum; i++) {
449     ArcNode* p = G.vertices[i].firstarc;
450     while (p) {
451       VR[i][p->adjvex] = 1;
452       p = p->nextarc;
453     }
454   }
455   for (i = 0; i < G.vexnum; i++) {
456     for (j = i; j < G.vexnum; j++) {
457       if (VR[i][j] == 1)
458         fprintf(fp, "%d %d ", G.vertices[i].data.key, G.vertices[j].data.key);
459     }
460   }
461   fprintf(fp, "-1 -1 ");
462   fclose(fp);
```

```
463   return OK;
464 }
465 // 将图的数据写入到文件 FileName 中
466
467 status LoadGraph(ALGraph& G, char FileName[]) {
468   FILE* fp = fopen(FileName, "r");
469   VertexType V[10];
470   KeyType VR[100][2];
471   int i = 0, n = 0;
472   do {
473     fscanf(fp, "%d%s", &V[i].key, V[i].others);
474   } while (V[i++].key != -1);
475
476   do {
477     fscanf(fp, "%d%d", &VR[n][0], &VR[n][1]);
478   } while (VR[n++][0] != -1);
479
480   if (CreateCraph(G, V, VR) == ERROR) {
481     fclose(fp);
482     return ERROR;
483   }
484   fclose(fp);
485   return OK;
486 }
487 // 读入文件 FileName 的图数据，创建图的邻接表
488
489 status RemoveList(LISTS& L, char ListName[]) {
490   int n;
491   for (n = 0; n < Lists.length; n++) {
492     if (!strcmp(Lists.elem[n].name, ListName)) {
493       for (; n < Lists.length - 1; n++) {
```

```
494           Lists.elem[n] = Lists.elem[n + 1];
495         }
496         Lists.length--;
497         return OK;
498       }
499     }
500     return ERROR;
501   }
502
503   status LocateList(
504       LISTS Lists,
505       char ListName
506           [])  // 在 Lists 中查找一个名称为 ListName 的线性表，成功返回逻辑序号，否则返
507   {
508     int n;
509     for (n = 0; n < Lists.length; n++) {
510       if (!strcmp(Lists.elem[n].name, ListName)) return (n + 1);
511     }
512     return ERROR;
513   }
514
515   status AddList(LISTS& Lists, char ListName[])  // 创建表名
516   {
517     strcpy(Lists.elem[Lists.length].name, ListName);
518     Lists.length++;
519     return OK;
520   }
521
522   status MoreSaveGraph(ALGraph G, FILE* fp) {
523     int i, j;
524     for (i = 0; i < G.vexnum; i++) {
```

```
525       fprintf(fp, "%d %s ", G.vertices[i].data.key, G.vertices[i].data.others);
526     }
527     fprintf(fp, "-1 nil ");
528     int VR[20][20];
529
530     for (i = 0; i < G.vexnum; i++) {
531       ArcNode* p = G.vertices[i].firstarc;
532       while (p) {
533         VR[i][p->adjvex] = 1;
534         p = p->nextarc;
535       }
536     }
537     for (i = 0; i < G.vexnum; i++) {
538       for (j = i; j < G.vexnum; j++) {
539         if (VR[i][j] == 1)
540           fprintf(fp, "%d %d ", G.vertices[i].data.key, G.vertices[j].data.key);
541       }
542     }
543     fprintf(fp, "-1 -1");
544     return OK;
545 }
546
547 status MoreLoadGraph(ALGraph& G, FILE* fp) {
548     VertexType V[10];
549     KeyType VR[100][2];
550     int i = 0, n = 0;
551     do {
552       fscanf(fp, "%d%s", &V[i].key, V[i].others);
553     } while (V[i++].key != -1);
554
555     do {
```

```
556       fscanf(fp, "%d%d", &VR[n][0], &VR[n][1]);
557     } while (VR[n++][0] != -1);
558
559     if (CreateCraph(G, V, VR) == ERROR) return ERROR;
560     return OK;
561   }
562
563   status InitQueue(Linkqueue& Q) {
564     Q.front = Q.rear = (QNode*)malloc(sizeof(QNode));
565     if (!Q.front) return ERROR;
566     Q.front->next = NULL;
567     return OK;
568   }
569
570   status QueueEmpty(Linkqueue Q) {
571     if (Q.front == Q.rear)
572       return TRUE;
573     else
574       return FALSE;
575   }
576
577   status enQueue(Linkqueue& Q, VertexType value) {
578     Queue p = (Queue)malloc(sizeof(QNode));
579     if (!p) return ERROR;
580     p->data = value;
581     p->next = NULL;
582     Q.rear->next = p;
583     Q.rear = p;
584     return OK;
585   }
586
```

```
587 status deQueue(Linkqueue& Q, VertexType& value) {
588   if (Q.front == Q.rear) return ERROR;
589   Queue p = Q.front->next;
590   value = p->data;
591   Q.front->next = p->next;
592   if (Q.rear == p) Q.rear = Q.front;
593   free(p);
594   return OK;
595 }
596
597 status ShortestPathLength(ALGraph G, KeyType v,
598                           KeyType w)  // 返回顶点 v 与顶点 w 的最短路径长度
599 {
600   int i, j, n;
601   VertexType top;
602   top.key = v;
603   Linkqueue Q;
604   InitQueue(Q);
605   for (i = 0; i < G.vexnum; i++) distance[G.vertices[i].data.key] = 20;
606   distance[v] = 0;
607   int k = LocateVex(G, v);
608   enQueue(Q, G.vertices[k].data);
609   while (!QueueEmpty(Q)) {
610     deQueue(Q, top);
611     if (top.key == w) break;
612     for (j = FirstAdjVex(G, top.key); j >= 0;
613          j = NextAdjVex(G, top.key, G.vertices[j].data.key))  // 返回位序
614     {
615       if (distance[G.vertices[j].data.key] == 20) {
616         distance[G.vertices[j].data.key] = distance[top.key] + 1;
617         enQueue(Q, G.vertices[j].data);
```

```
618         }
619       }
620     }
621     n = distance[w];
622     return n;
623   }
624
625   void VerticesSetLessThank(ALGraph G, KeyType v,
626                             KeyType k)  // 返回与顶点 v 距离小于 k 的顶点集合
627   {
628     int i, j, flag = 0;
629     for (i = 0; i < G.vexnum; i++) {
630       if (G.vertices[i].data.key == v) continue;
631       j = ShortestPathLength(G, v, G.vertices[i].data.key);
632       if (j < k) {
633         printf("%d %s ", G.vertices[i].data.key, G.vertices[i].data.others);
634         flag = 1;
635       }
636     }
637     if (flag == 0) printf(" 不存在! ");
638   }
639
640   void dfs(ALGraph& G, KeyType v) {
641     mark[v] = TRUE;
642     for (int w = FirstAdjVex(G, v); w >= 0;
643          w = NextAdjVex(G, v, G.vertices[w].data.key)) {
644       if (!mark[G.vertices[w].data.key]) dfs(G, G.vertices[w].data.key);
645     }
646   }
647
648   status ConnectedComponentsNums(ALGraph G)  // 求图中连通分量个数
```

```
649 {
650   int count = 0, i;
651   for (i = 0; i < G.vexnum; i++)
652     mark[G.vertices[i].data.key] = FALSE;  // 标记数组记录关键字
653   for (i = 0; i < G.vexnum; i++) {
654     if (!mark[G.vertices[i].data.key]) {
655       dfs(G, G.vertices[i].data.key);
656       count++;
657     }
658   }
659   return count;
660 }
661
```

主要流程文件 main.cpp :

```
1  #include "define.h"
2
3  int main() {
4    printf(
5        "-----------------基于邻接表的无向图系统--------------------\n\n");  // 一级菜
6    printf("1. 单图操作\t\t2. 多图操作\t\t0. 停止操作\n\n请输入你需要的操作: ");
7
8    scanf("%d", &op);  // 输入操作数
9    if (op == 1) {     // 单图操作 (二级菜单)
10     printf(
11         "\n-----------------单图操作菜单--------------------\n\
12 ^^I^^I1. 创建图          2. 销毁图          3. 查找顶点\n\
13 ^^I^^I4. 顶点赋值        5. 获得第一邻接顶点  6. 获得下一邻接点\n\
14 ^^I^^I7. 插入顶点        8. 删除顶点        9. 插入弧\n\
15 ^^I^^I10. 删除弧         11. 深度优先搜索遍历  12. 广度优先搜索遍历\n\
16 ^^I^^I13. 文件保存       14. 文件读取       15. 距离小于 k 的顶点集合\n\
17 ^^I^^I16. 顶点间最短路径和长度  17. 连通分量     0. 停止操作\n\
```

```
18  ^^I^^I 请输入【0~17】\n\n请输入你需要的操作: ");
19      G.vexnum = 0, G.arcnum = 0;  // 初始化顶点数和边数
20      scanf("%d", &op);            // 输入操作
21      while (op) {
22        switch (op) {
23          case 1: {  // 创建图
24            if (G.vexnum == 0) {
25              VertexType V[30];    // 暂存顶点数据
26              KeyType VR[100][2];  // 暂存弧的数据
27              int ans, i = 0;
28              printf(" 请输入顶点数据【以-1 为结尾】: ");
29              do {  // 输入顶点数据
30                scanf("%d%s", &V[i].key, V[i].others);
31              } while (V[i++].key != -1);
32              printf(" 请输入弧数据【以-1 为结尾】: ");
33              i = 0;
34              do {  // 输入弧的数据
35                scanf("%d%d", &VR[i][0], &VR[i][1]);
36              } while (VR[i++][0] != -1);
37              ans = CreateCraph(G, V, VR);  // 创建无向图
38              if (ans == OK) {
39                printf(" 无向图创建成功! 遍历为: \n");
40                for (i = 0; i < G.vexnum; i++) {
41                  ArcNode* p = G.vertices[i].firstarc;
42                  printf("%d %s", G.vertices[i].data.key,
43                         G.vertices[i].data.others);
44                  while (p) {
45                    printf(" %d", p->adjvex);
46                    p = p->nextarc;
47                  }
48                  printf("\n");
```

```
49              }
50              printf("\n");
51            } else
52              printf(" 无向图创建失败! \n");
53          } else
54            printf(" 无向图图存在, 无法创建! \n");
55          getchar();
56          getchar();
57          system("cls");
58          break;
59        }
60        case 2: {  // 销毁图
61          if (G.vexnum != 0) {
62            DestroyGraph(G);  // 调用销毁图的函数
63            printf(" 无向图销毁成功! \n");
64          } else
65            printf(" 无向图为空, 无需销毁! \n");
66          getchar();
67          getchar();
68          system("cls");
69          break;
70        }
71        case 3: {  // 查找顶点
72          if (G.vexnum != 0) {
73            int e, ans;
74            printf(" 请输入你想查找结点的关键字: ");
75            scanf("%d", &e);
76            ans = LocateVex(G, e);  // 调用查找结点函数
77            if (ans == -1)
78              printf(" 查找失败! \n");
79            else
```

```
80              printf(" 对应结点为 %d,%s\n", G.vertices[ans].data.key,
81                     G.vertices[ans].data.others);
82            } else
83              printf(" 无向图为空, 查找失败! \n");
84            getchar();
85            getchar();
86            system("cls");
87            break;
88          }
89          case 4: {  // 顶点赋值
90            if (G.vexnum != 0) {
91              VertexType e;
92              int ans, key;
93              printf(" 输入替换的顶点关键字: ");
94              scanf("%d", &key);
95              printf(" 输入赋值: ");
96              scanf("%d%s", &e.key, e.others);
97
98              ans = PutVex(G, key, e);  // 调用赋值结点函数
99              if (ans == OK) {
100               printf(" 赋值成功, 深度搜索遍历为: ");
101               BFSTraverse(G, visit);
102               printf("\n");
103             } else
104               printf(" 赋值操作失败! 有可能是结点不存在! \n");
105
106           } else
107             printf(" 无向图图为空, 赋值操作失败! \n");
108           getchar();
109           getchar();
110           system("cls");
```

```
111             break;
112           }
113           case 5: {  // 获得第一邻接点
114             if (G.vexnum != 0) {
115               int key, ans;
116               printf(" 请输入需要获得第一邻接顶点的关键字: ");
117               scanf("%d", &key);
118               ans = FirstAdjVex(G, key);  // 调用查找第一邻接点的函数
119               if (ans == -2) printf(" 顶点无第一邻接顶点\n");
120               if (ans == -1) printf(" 无该顶点!\n");
121               if (ans >= 0)
122                 printf("%d 结点的第一邻接点是%d,%s\n", key,
123                        G.vertices[ans].data.key, G.vertices[ans].data.others);
124             } else
125               printf(" 无向图为空! \n");
126             getchar();
127             getchar();
128             system("cls");
129             break;
130           }
131           case 6: {  // 获得下一邻接点
132             if (G.vexnum != 0) {
133               int key, relat, ans;
134               printf(" 输入待查找结点和相对结点: ");
135               scanf("%d%d", &key, &relat);
136               ans = NextAdjVex(G, key, relat);  // 调用查找相对节点的函数
137               if (ans == -1)
138                 printf(" 获取失败! \n");
139               else {
140                 printf("%d 相对于%d 的顶点为%d,%s\n", key, relat,
141                        G.vertices[ans].data.key, G.vertices[ans].data.others);
```

```
142             }
143           } else
144             printf(" 无向图为空，获取失败！\n");
145           getchar();
146           getchar();
147           system("cls");
148           break;
149         }
150         case 7: {  // 插入顶点
151           if (G.vexnum != 0) {
152             VertexType e;
153             int ans, i;
154             printf(" 输入待插入的顶点数据：");
155             scanf("%d%s", &e.key, e.others);
156             ans = InsertVex(G, e);  // 调用插入函数
157             if (ans == OK) {
158               printf(" 插入成功，遍历为：\n");
159               for (i = 0; i < G.vexnum; i++) {
160                 ArcNode* p = G.vertices[i].firstarc;
161                 printf("%d %s", G.vertices[i].data.key,
162                        G.vertices[i].data.others);
163                 while (p) {
164                   printf(" %d", p->adjvex);
165                   p = p->nextarc;
166                 }
167                 printf("\n");
168               }
169               printf("\n");
170             } else
171               printf(" 插入操作失败！\n");
172           } else
```

```
173             printf(" 无向图为空，请先创建图！\n");
174           getchar();
175           getchar();
176           system("cls");
177           break;
178         }
179         case 8: {  // 删除顶点
180           if (G.vexnum != 0) {
181             int e, ans, i;
182             printf(" 请输入删除结点：");
183             scanf("%d", &e);
184             ans = DeleteVex(G, e);  // 调用删除顶点函数
185             if (ans == OK) {
186               printf(" 删除成功，遍历为：\n");
187               for (i = 0; i < G.vexnum; i++) {
188                 ArcNode* p = G.vertices[i].firstarc;
189                 printf("%d %s", G.vertices[i].data.key,
190                        G.vertices[i].data.others);
191                 while (p) {
192                   printf(" %d", p->adjvex);
193                   p = p->nextarc;
194                 }
195                 printf("\n");
196               }
197               printf("\n");
198             } else
199               printf(" 删除操作失败！\n");
200           } else
201             printf(" 图为空，删除操作失败！\n");
202           getchar();
203           getchar();
```

```
204             system("cls");
205             break;
206           }
207           case 9: {  // 添加弧
208             if (G.vexnum != 0) {
209               int ans, i, v, w;
210               printf(" 输入待插入的弧数据: ");
211               scanf("%d%d", &v, &w);
212               ans = InsertArc(G, v, w);  // 调用添加弧的函数
213               if (ans == OK) {
214                 printf(" 添加成功, 遍历为: \n");  // 添加成功就直接遍历
215                 for (i = 0; i < G.vexnum; i++) {
216                   ArcNode* p = G.vertices[i].firstarc;
217                   printf("%d %s", G.vertices[i].data.key,
218                          G.vertices[i].data.others);
219                   while (p) {
220                     printf(" %d", p->adjvex);
221                     p = p->nextarc;
222                   }
223                   printf("\n");
224                 }
225                 printf("\n");
226               } else
227                 printf(" 插入操作失败! \n");
228             } else
229               printf(" 无向图为空, 请先创建图! \n");
230             getchar();
231             getchar();
232             system("cls");
233             break;
234           }
```

```
235         case 10: {  // 删除弧
236           if (G.vexnum != 0) {
237             int v, w, ans, i;
238             printf(" 请输入删除弧: ");
239             scanf("%d%d", &v, &w);
240             ans = DeleteArc(G, v, w);  // 调用删除弧的函数
241             if (ans == OK) {
242               printf(" 删除成功, 遍历为: \n");
243               for (i = 0; i < G.vexnum; i++) {
244                 ArcNode* p = G.vertices[i].firstarc;
245                 printf("%d %s", G.vertices[i].data.key,
246                        G.vertices[i].data.others);
247                 while (p) {
248                   printf(" %d", p->adjvex);
249                   p = p->nextarc;
250                 }
251                 printf("\n");
252               }
253               printf("\n");
254             } else
255               printf(" 删除操作失败! \n");
256           } else
257             printf(" 图为空, 删除操作失败! \n");
258           getchar();
259           getchar();
260           system("cls");
261           break;
262         }
263         case 11: {  // 深度遍历
264           if (G.vexnum != 0) {
265             printf(" 深度优先搜索遍历: ");
```

```
266             DFSTraverse(G, visit);  // 调用深度优先遍历函数
267             putchar('\n');
268           } else
269             printf(" 无向图不存在! 遍历为空! \n");
270           getchar();
271           getchar();
272           system("cls");
273           break;
274         }
275         case 12: {  // 广度遍历
276           if (G.vexnum != 0) {
277             printf(" 广度优先搜索遍历: ");
278             BFSTraverse(G, visit);  // 调用广度优先边路函数
279             putchar('\n');
280           } else
281             printf(" 无向图不存在! 遍历为空! \n");
282           getchar();
283           getchar();
284           system("cls");
285           break;
286         }
287         case 13: {  // 文件保存
288           if (G.vexnum != 0) {
289             char FileName[30];  // 目标文件名
290             printf(" 请输入要保存的文件名:\n");
291             scanf("%s", FileName);
292             if (SaveGraph(G, FileName) == OK)
293               printf(" 保存成功! \n");  // 调用文件保存函数
294           } else
295             printf(" 图为空, 保存失败! \n");
296           getchar();
```

```
297             getchar();
298             system("cls");
299             break;
300           }
301           case 14: {  // 文件读取
302             if (G.vexnum == 0) {
303               char FileName[30];  // 目标文件名
304               printf(" 请输入要读取的文件名:\n");
305               scanf("%s", FileName);
306               if (LoadGraph(G, FileName) == OK)
307                 printf(" 读取成功! \n");  // 调用文件读取函数
308             } else
309               printf(" 图不为空, 无法读取文件! \n");
310             getchar();
311             getchar();
312             system("cls");
313             break;
314           }
315           case 15: {  // 查找与顶点距离小于 K 的顶点
316             if (G.vexnum != 0) {
317               int v, k;
318               printf(" 请输入顶点的关键字及查找距离! \n");
319               scanf("%d%d", &v, &k);
320               printf(" 与顶点距离小于%d 的顶点有\n", k);
321               VerticesSetLessThank(G, v, k);  // 调用查找与顶点距离小于 k 的点的函数
322             } else
323               printf(" 无向图为空!\n");
324             getchar();
325             getchar();
326             system("cls");
327             break;
```

```
328         }
329         case 16: {  // 输出两个顶点的最短路径
330           if (G.vexnum != 0) {
331             int v, w, k;
332             printf(" 请输入两个顶点的关键字!\n");
333             scanf("%d%d", &v, &w);
334             k = ShortestPathLength(G, v, w);  // 调用求两个顶点的最短路径的长度
335             printf(" 这两个顶点间的最短路径是%d", k);
336           } else
337             printf(" 无向图为空!\n");
338           getchar();
339           getchar();
340           system("cls");
341           break;
342         }
343         case 17: {  // 求图的连通分量
344           if (G.vexnum != 0) {
345             printf(" 图的连通分量有%d 个! \n",
346                    ConnectedComponentsNums(G));  // 调用求图的连通分量的函数
347           } else
348             printf(" 无向图为空!\n");
349           getchar();
350           getchar();
351           system("cls");
352           break;
353         }
354         default: {
355           printf(" 输入错误, 请重新输入! \n");
356           getchar();
357           getchar();
358           system("cls");
```

```
359           }
360         }
361         print1();
362         scanf("%d", &op);
363       }
364     }
365
366     if (op == 2) {        // 多图操作（二级菜单）
367       Lists.length = 0;  // 初始化 Lists 长度
368     A:;
369       printf(
370           "\n-----------------多图操作菜单--------------------\n\
371 ^^I^^I1. 创建多图          2. 移除某图          3. 查找某图\n\
372 ^^I^^I4. 单独操作          5. 遍历多图          6. 清空多图\n\
373 ^^I^^I0. 停止操作          请输入【0~6】\n\n请输入你需要的操作: ");
374       scanf("%d", &op);
375       while (op) {
376         switch (op) {
377           case 1: {  // 创建多图中的单图
378             printf(" 输入创建图的名称: ");
379             scanf("%s", saveName);
380
381             VertexType V[30];    // 暂时储存顶点数据的数组
382             KeyType VR[100][2];  // 暂时储存边的数据的数组
383             int ans, i = 0;
384             printf(" 请输入顶点数据【以-1 为结尾】: ");
385             do {
386               scanf("%d%s", &V[i].key, V[i].others);
387             } while (V[i++].key != -1);
388             printf(" 请输入弧数据【以-1 为结尾】: ");
389             i = 0;
```

```
390             do {
391               scanf("%d%d", &VR[i][0], &VR[i][1]);
392             } while (VR[i++][0] != -1);
393             ans = CreateCraph(Lists.elem[Lists.length].G, V, VR);
394             if (ans == OK) {
395               AddList(Lists, saveName);
396               printf(" 创建成功! ");
397               putchar('\n');
398             } else
399               printf(" 创建失败! \n");
400             getchar();
401             getchar();
402             system("cls");
403             break;
404           }
405           case 2: {  // 移除某图
406             printf(" 输入你想移除图的名称: ");
407             scanf("%s", saveName);
408             if (RemoveList(Lists, saveName) == ERROR) {
409               printf(" 无该名的无向图! \n");
410             } else
411               printf(" 删除成功! \n");
412             getchar();
413             getchar();
414             system("cls");
415             break;
416           }
417           case 3: {  // 查找某图
418             printf(" 输入你想查找树的名称: ");
419             scanf("%s", saveName);
420             if ((number = LocateList(Lists, saveName)) == ERROR) {
```

```
421             printf(" 无该名的无向图! \n");
422             getchar();
423             getchar();
424             system("cls");
425             break;
426           }  // 调用查找树的函数
427           printf(" 查找成功: %s \n", saveName);
428
429           if (Lists.elem[number - 1].G.vexnum == 0) {
430             printf(" 无数据! \n");
431           } else {
432             printf(" 广度搜索遍历: ");
433             BFSTraverse(Lists.elem[number - 1].G, visit);
434             putchar('\n');
435           }
436           getchar();
437           getchar();
438           system("cls");
439           break;
440         }
441         case 4: {  // 单独操作
442           printf(" 输入你想单独操作图的名称: ");
443           scanf("%s", saveName);
444           if ((number = LocateList(Lists, saveName)) == ERROR) {
445             printf(" 无该名称的无向图! \n");
446             getchar();
447             getchar();
448             system("cls");
449             break;
450           }  // 三级菜单
451           printf(
```

```
452                 "\n-----------------单图操作菜单--------------------\n\
453 ^^I^^I^^I^^I1. 创建图            2. 销毁图              3. 查找顶点\n\
454 ^^I^^I^^I^^I4. 顶点赋值          5. 获得第一邻接顶点    6. 获得下一邻接点\n\
455 ^^I^^I^^I^^I7. 插入顶点          8. 删除顶点            9. 插入弧\n\
456 ^^I^^I^^I^^I10. 删除弧           11. 深度优先搜索遍历   12. 广度优先搜索遍历\n\
457 ^^I^^I^^I^^I13. 文件保存         14. 文件读取           15. 距离小于 k 的顶点集合\n\
458 ^^I^^I^^I^^I16. 顶点间最短路径和长度  17. 连通分量      18. 返回多图菜单\n\
459 ^^I^^I^^I^^I0. 停止操作\n\
460 ^^I^^I^^I^^I 请输入【0~17】\n\n请输入你需要的操作: ");
461           scanf("%d", &op);  // 输入操作
462           while (op) {
463             switch (op) {
464               case 1: {  // 创建图
465                 if (Lists.elem[number - 1].G.vexnum == 0) {
466                   VertexType V[30];    // 暂时储存顶点的数据
467                   KeyType VR[100][2];  // 暂时储存边的数据
468                   int ans, i = 0;
469                   printf(" 请输入结点数据【以-1 为结尾】: ");
470                   do {
471                     scanf("%d%s", &V[i].key, V[i].others);
472                   } while (V[i++].key != -1);
473                   printf(" 请输入弧数据【以-1 为结尾】: ");
474                   i = 0;
475                   do {
476                     scanf("%d%d", &VR[i][0], &VR[i][1]);
477                   } while (VR[i++][0] != -1);
478                   ans = CreateCraph(Lists.elem[number - 1].G, V, VR);
479                   if (ans == OK) {
480                     printf(" 无向图创建成功! 遍历为: \n");
481                     for (i = 0; i < Lists.elem[number - 1].G.vexnum; i++) {
482                       ArcNode* p =
```

```
                    Lists.elem[number - 1].G.vertices[i].firstarc;
                printf("%d %s",
                       Lists.elem[number - 1].G.vertices[i].data.key,
                       Lists.elem[number - 1].G.vertices[i].data.others);
                while (p) {
                  printf(" %d", p->adjvex);
                  p = p->nextarc;
                }
                printf("\n");
              }
              printf("\n");
            } else
              printf(" 无向图创建失败! \n");
          } else
            printf(" 无向图图存在, 无法创建! \n");
          getchar();
          getchar();
          system("cls");
          break;
        }
        case 2: {  // 销毁图
          if (Lists.elem[number - 1].G.vexnum != 0) {
            DestroyGraph(Lists.elem[number - 1].G);  // 调用销毁图的函数
            printf(" 无向图销毁成功! \n");
          } else
            printf(" 无向图为空, 无需销毁! \n");
          getchar();
          getchar();
          system("cls");
          break;
        }
```

```
514                 case 3: {  // 查找顶点
515                   if (Lists.elem[number - 1].G.vexnum != 0) {
516                     int e, ans;
517                     printf(" 请输入你想查找结点的关键字: ");
518                     scanf("%d", &e);
519                     ans = LocateVex(Lists.elem[number - 1].G,
520                                     e);  // 调用查找顶点的函数
521                     if (ans == -1)
522                       printf(" 查找失败! \n");
523                     else
524                       printf(" 对应结点为 %d,%s\n",
525                              Lists.elem[number - 1].G.vertices[ans].data.key,
526                              Lists.elem[number - 1].G.vertices[ans].data.others);
527                   } else
528                     printf(" 无向图为空, 查找失败! \n");
529                   getchar();
530                   getchar();
531                   system("cls");
532                   break;
533                 }
534                 case 4: {  // 顶点赋值
535                   if (Lists.elem[number - 1].G.vexnum != 0) {
536                     VertexType e;
537                     int ans, key;
538                     printf(" 输入替换的顶点关键字: ");
539                     scanf("%d", &key);
540                     printf(" 输入赋值: ");
541                     scanf("%d%s", &e.key, e.others);
542
543                     ans = PutVex(Lists.elem[number - 1].G, key,
544                                  e);  // 调用查找顶点的函数
```

```
545                 if (ans == OK) {
546                   printf(" 赋值成功，深度搜索遍历为: ");
547                   BFSTraverse(Lists.elem[number - 1].G, visit);
548                   printf("\n");
549                 } else
550                   printf(" 赋值操作失败! \n");
551
552               } else
553                 printf(" 无向图图为空，赋值操作失败! \n");
554               getchar();
555               getchar();
556               system("cls");
557               break;
558             }
559             case 5: {  // 获得第一邻接点
560               if (Lists.elem[number - 1].G.vexnum != 0) {
561                 int key, ans;
562                 printf(" 请输入需要获得第一邻接顶点的关键字: ");
563                 scanf("%d", &key);
564                 ans = FirstAdjVex(Lists.elem[number - 1].G,
565                                   key);  // 调用求第一邻接点的函数
566                 if (ans == -2) printf(" 顶点无第一邻接顶点\n");
567                 if (ans == -1) printf(" 无该顶点!\n");
568                 if (ans >= 0)
569                   printf("%d 结点的第一邻接点是%d,%s\n", key,
570                          Lists.elem[number - 1].G.vertices[ans].data.key,
571                          Lists.elem[number - 1].G.vertices[ans].data.others);
572               } else
573                 printf(" 无向图为空! \n");
574               getchar();
575               getchar();
```

```
576                 system("cls");
577                 break;
578               }
579               case 6: {  // 获得下一邻接点
580                 if (Lists.elem[number - 1].G.vexnum != 0) {
581                   int key, relat, ans;
582                   printf(" 输入待查找结点和相对结点: ");
583                   scanf("%d%d", &key, &relat);
584                   ans = NextAdjVex(Lists.elem[number - 1].G, key,
585                                    relat);  // 调用获取下一邻接点的函数
586                   if (ans == -1)
587                     printf(" 获取失败! \n");
588                   else {
589                     printf("%d 相对于%d 的顶点为%d,%s\n", key, relat,
590                            Lists.elem[number - 1].G.vertices[ans].data.key,
591                            Lists.elem[number - 1].G.vertices[ans].data.others);
592                   }
593                 } else
594                   printf(" 无向图为空, 获取失败! \n");
595                 getchar();
596                 getchar();
597                 system("cls");
598                 break;
599               }
600               case 7: {  // 插入顶点
601                 if (Lists.elem[number - 1].G.vexnum != 0) {
602                   VertexType e;
603                   int ans, i;
604                   printf(" 输入待插入的顶点数据: ");
605                   scanf("%d%s", &e.key, e.others);
606                   ans = InsertVex(Lists.elem[number - 1].G,
```

```
607                                 e);  // 调用插入节点的函数
608                 if (ans == OK) {
609                   printf(" 插入成功, 遍历为: \n");
610                   for (i = 0; i < Lists.elem[number - 1].G.vexnum; i++) {
611                     ArcNode* p =
612                         Lists.elem[number - 1].G.vertices[i].firstarc;
613                     printf("%d %s",
614                            Lists.elem[number - 1].G.vertices[i].data.key,
615                            Lists.elem[number - 1].G.vertices[i].data.others);
616                     while (p) {
617                       printf(" %d", p->adjvex);
618                       p = p->nextarc;
619                     }
620                     printf("\n");
621                   }
622                   printf("\n");
623                 } else
624                   printf(" 插入操作失败! \n");
625               } else
626                 printf(" 无向图为空, 请先创建图! \n");
627               getchar();
628               getchar();
629               system("cls");
630               break;
631             }
632             case 8: {  // 删除顶点
633               if (Lists.elem[number - 1].G.vexnum != 0) {
634                 int e, ans, i;
635                 printf(" 请输入删除结点: ");
636                 scanf("%d", &e);
637                 ans = DeleteVex(Lists.elem[number - 1].G,
```

```
638                                 e);  // 调用删除顶点的函数
639                     if (ans == OK) {
640                       printf(" 删除成功, 遍历为: \n");
641                       for (i = 0; i < Lists.elem[number - 1].G.vexnum; i++) {
642                         ArcNode* p =
643                             Lists.elem[number - 1].G.vertices[i].firstarc;
644                         printf("%d %s",
645                                Lists.elem[number - 1].G.vertices[i].data.key,
646                                Lists.elem[number - 1].G.vertices[i].data.others);
647                         while (p) {
648                           printf(" %d", p->adjvex);
649                           p = p->nextarc;
650                         }
651                         printf("\n");
652                       }
653                       printf("\n");
654                     } else
655                       printf(" 删除操作失败! \n");
656                   } else
657                     printf(" 图为空, 删除操作失败! \n");
658                   getchar();
659                   getchar();
660                   system("cls");
661                   break;
662                 }
663                 case 9: {  // 添加弧
664                   if (Lists.elem[number - 1].G.vexnum != 0) {
665                     int ans, i, v, w;
666                     printf(" 输入待插入的弧数据: ");
667                     scanf("%d%d", &v, &w);
668                     ans = InsertArc(Lists.elem[number - 1].G, v,
```

```
669                                 w);  // 调用添加弧的函数
670                 if (ans == OK) {
671                   printf(" 添加成功, 遍历为: \n");
672                   for (i = 0; i < Lists.elem[number - 1].G.vexnum; i++) {
673                     ArcNode* p =
674                         Lists.elem[number - 1].G.vertices[i].firstarc;
675                     printf("%d %s",
676                            Lists.elem[number - 1].G.vertices[i].data.key,
677                            Lists.elem[number - 1].G.vertices[i].data.others);
678                     while (p) {
679                       printf(" %d", p->adjvex);
680                       p = p->nextarc;
681                     }
682                     printf("\n");
683                   }
684                   printf("\n");
685                 } else
686                   printf(" 插入操作失败! \n");
687               } else
688                 printf(" 无向图为空, 请先创建图! \n");
689               getchar();
690               getchar();
691               system("cls");
692               break;
693             }
694             case 10: {  // 删除弧
695               if (Lists.elem[number - 1].G.vexnum != 0) {
696                 int v, w, ans, i;
697                 printf(" 请输入删除弧: ");
698                 scanf("%d%d", &v, &w);
699                 ans = DeleteArc(Lists.elem[number - 1].G, v,
```

```
700                                 w);  // 调用删除弧的函数
701                 if (ans == OK) {
702                   printf(" 删除成功, 遍历为: \n");
703                   for (i = 0; i < Lists.elem[number - 1].G.vexnum; i++) {
704                     ArcNode* p =
705                         Lists.elem[number - 1].G.vertices[i].firstarc;
706                     printf("%d %s",
707                            Lists.elem[number - 1].G.vertices[i].data.key,
708                            Lists.elem[number - 1].G.vertices[i].data.others);
709                     while (p) {
710                       printf(" %d", p->adjvex);
711                       p = p->nextarc;
712                     }
713                     printf("\n");
714                   }
715                   printf("\n");
716                 } else
717                   printf(" 删除操作失败! \n");
718               } else
719                 printf(" 图为空, 删除操作失败! \n");
720               getchar();
721               getchar();
722               system("cls");
723               break;
724             }
725             case 11: {  // 深度遍历
726               if (Lists.elem[number - 1].G.vexnum != 0) {
727                 printf(" 深度优先搜索遍历: ");
728                 DFSTraverse(Lists.elem[number - 1].G,
729                             visit);  // 调用深度优先遍历的函数
730                 putchar('\n');
```

```
731                 } else
732                   printf(" 遍历为空! \n");
733                 getchar();
734                 getchar();
735                 system("cls");
736                 break;
737               }
738               case 12: {  // 广度遍历
739                 if (Lists.elem[number - 1].G.vexnum != 0) {
740                   printf(" 广度优先搜索遍历: ");
741                   BFSTraverse(Lists.elem[number - 1].G,
742                               visit);  // 调用广度优先遍历的函数
743                   putchar('\n');
744                 } else
745                   printf(" 遍历为空! \n");
746                 getchar();
747                 getchar();
748                 system("cls");
749                 break;
750               }
751               case 13: {  // 文件保存
752                 if (Lists.elem[number - 1].G.vexnum != 0) {
753                   char FileName[30];  // 目标文件名
754                   printf(" 请输入要保存的文件名:\n");
755                   scanf("%s", FileName);
756                   if (SaveGraph(Lists.elem[number - 1].G, FileName) == OK)
757                     printf(" 保存成功! \n");  // 调用保存文件的函数
758                 } else
759                   printf(" 图为空, 保存失败! \n");
760                 getchar();
761                 getchar();
```

```
762                 system("cls");
763                 break;
764               }
765               case 14: {  // 文件读取
766                 if (Lists.elem[number - 1].G.vexnum == 0) {
767                   char FileName[30];  // 目标文件名
768                   printf(" 请输入要读取的文件名:\n");
769                   scanf("%s", FileName);
770                   if (LoadGraph(Lists.elem[number - 1].G, FileName) == OK)
771                     printf(" 读取成功! \n");  // 调用读取文件的函数
772                 } else
773                   printf(" 图不为空, 无法读取文件! \n");
774                 getchar();
775                 getchar();
776                 system("cls");
777                 break;
778               }
779               case 15: {  // 查找与顶点距离小于 k 的顶点
780                 if (Lists.elem[number - 1].G.vexnum !=
781                     0) {  // 检查无向图是否为空
782                   int v, k;
783                   printf(" 请输入顶点的关键字及查找距离! \n");
784                   scanf("%d%d", &v, &k);
785                   printf(" 与顶点距离小于%d 的顶点有", k);
786                   VerticesSetLessThank(Lists.elem[number - 1].G, v,
787                                        k);  // 调用函数查找与顶点距离小于 k 的顶点
788                 } else
789                   printf(" 无向图为空!\n");
790                 getchar();
791                 getchar();
792                 system("cls");
```

```
793                 break;
794               }
795               case 16: {  // 计算两个顶点的最短路径
796                 if (Lists.elem[number - 1].G.vexnum != 0) {
797                   int v, w, k;
798                   printf(" 请输入两个顶点的关键字!\n");
799                   scanf("%d%d", &v, &w);
800                   k = ShortestPathLength(Lists.elem[number - 1].G, v,
801                                          w);  // 调用计算两个顶点最短路径的函数
802                   printf(" 这两个顶点间的最短路径是%d", k);
803                 } else
804                   printf(" 无向图为空!\n");
805                 getchar();
806                 getchar();
807                 break;
808               }
809               case 17: {  // 求图的连通分量
810                 if (Lists.elem[number - 1].G.vexnum != 0) {
811                   printf(" 图的连通分量有%d 个! \n",
812                          ConnectedComponentsNums(
813                              Lists.elem[number - 1]
814                                  .G));  // 调用求图的联通分量的函数
815                 } else
816                   printf(" 无向图为空!\n");
817                 getchar();
818                 getchar();
819                 system("cls");
820                 break;
821               }
822               case 18: {
823                 system("cls");
```

```
824                 goto A;
825                 break;
826               }
827               default: {
828                 printf(" 输入错误, 请重新输入! \n");
829                 getchar();
830                 getchar();
831                 system("cls");
832               }
833             }
834             print2();
835             scanf("%d", &op);
836           }
837         }
838         case 5: {  // 遍历多图
839           if (Lists.length == 0) {
840             printf(" 多图无数据! \n");
841             getchar();
842             getchar();
843             system("cls");
844             break;
845           }
846           int i;
847           for (i = 0; i < Lists.length; i++) {
848             printf("%s ", Lists.elem[i].name);
849             if (Lists.elem[i].G.vexnum == 0) {
850               printf(" 无数据! \n");
851               continue;
852             } else {
853               printf(" 广度优先遍历: ");
854               BFSTraverse(Lists.elem[i].G, visit);
```

```
855                 putchar('\n');
856               }
857             }
858             getchar();
859             getchar();
860             system("cls");
861             break;
862           }
863           case 6: {  // 清空多图
864             if (Lists.length == 0) {
865               printf(" 多图已为空!! \n");
866               getchar();
867               getchar();
868               system("cls");
869               break;
870             }
871             Lists.length = 0;
872             printf(" 清空成功! \n");
873             getchar();
874             getchar();
875             system("cls");
876             break;
877           }
878           default: {
879             printf(" 输入错误, 请重新输入! \n");
880             getchar();
881             getchar();
882             system("cls");
883           }
884         }
885         print3();
```

```
886         scanf("%d", &op);
887       }
888     }
889
890     if (op == 0) {  // 停止操作
891       printf(" 系统停止! \n");
892       return 0;
893     }
894
895     printf(" 输入错误! \n");
896     return 0;
897 }
```